CHAPTER 6 TCP/UDP COMMUNICATION IN JAVA

Chapter 5 discussed the BSD Internet socket, which is currently the most commonly used API for network programming. The BSD Internet domain sockets use the TCP/IP (or UDP/IP) protocol suite as the communication protocols among processes. In this chapter we want to address the TCP/UDP programming in Java, since the Java language is currently the most commonly used language to implement a distributed computing system. Java provides the reliable stream-based communication for TCP as well as the unreliable datagram communication for UDP.

6.1 Java Sockets

The socket API in Java is provided in the *java.net* package which has several classes supporting socket-based client/server communication.

6.1.1 Java Net Package

Client and server sockets for connection-oriented and connectionless communication are implemented by the *Socket, ServerSocket, DatagramSocket,* and *MulticastSocket* classes in the *java.net* package. In addition to these socket related classes, the *java.net* package also contains other classes that can help the communication. In particular, the *InetAddress* class encapsulates Internet IP addresses and supports conversion between dotted decimal addresses and host names. The *DatagramPacket* class is used to construct UDP datagram packets. The *SocketImpl* and *DatagramSocketImpl* classes and the *SocketImplFactory* interface provide hooks for implementing custom sockets.

High-level browser-server Web connections are implemented through the *URL, URLConnection, HttpURLConnection,* and *URLEncoder* classes. The *ContentHandler* and *URLStreamHandler* classes are abstract classes that can be used for the implementation of Web content and stream handlers. They are supported by the *ContentHandlerFactory* and *URLStreamHandlerFactory* interfaces. The *FileNameMap* interface is used to map filenames to MIME types.

The classes in the *java.net* package can be listed as follows:

The Classes

- `ContentHandler`
- `DatagramPacket`

- DatagramSocket
- DatagramSocketImpl
- HttpURLConnection
- InetAddress
- MulticastSocket
- ServerSocket
- Socket
- SocketImpl
- URL
- URLConnection
- URLEncoder
- URLStreamHandler

The Interfaces

- ContentHandlerFactory
- FileNameMap
- SocketImplFactory
- URLStreamHandlerFactory

Exceptions

- BindException
- ConnectException
- MalformedURLException
- NoRouteToHostException
- ProtocolException
- SocketException
- UnknownHostException
- UnknownServiceException

In this section we mainly discuss the *Socket* and *ServerSocket* classes. Other classes such as *DatagramSocket* and *DatagramPacket* etc. are discussed later.

6.1.2 The Socket Class

Connection-based sockets for clients are implemented through the *Socket* class. These sockets are used to develop client applications that utilize services provided by connection-oriented server applications.

The access methods of the *Socket* class are used to access the I/O streams (a stream is a high level abstraction representing a Java connection channel, a file, or a memory buffer, and is the basis for most Java communications) and connection parameters associated with a connected socket. Here are some access methods for the Socket class:

- Methods for accessing the socket information. The *getInetAddress()* and *getPort()* methods get the IP address of the destination host and the destination host port number to which the socket is connected. The *getLocalPort()* method returns the source host local port number associated with the socket. The *getLocalAddress()* method returns the local IP address associated with the socket.
- Methods for I/O. The *getInputStream()* and *getOutputStream()* methods are used to access the input and output streams associated with a socket.
- Other interesting methods. The *close()* method is used to close a socket. The *toString()* method returns a string representation of the socket.

6.1.3 The ServerSocket Class

A server works like a receptionist. She sits in the front desk of a company and waits for customers. She has no idea who will come or when they come. However, once a customer comes in, the receptionist will normally arrange a suitable staff of the company to actually work on the customer's request. After that, the receptionist will then wait for the next customer to arrive, or to serve the next waiting customer. Java provides the *ServerSocket* class to allow programmers to write servers that behave like a receptionist.

A TCP server socket is implemented by the *ServerSocket* class. Once a Java *ServerSocket* is established, it runs on a server and listens on a particular port of the server machine for incoming TCP connections. When a client *Socket* on a remote host attempts to connect to the port, the server tries to accept the connection request, negotiates the connection between the server and the client, and opens a regular *Socket* between the two hosts for the regular communication between the client and the server. The *ServerSocket* cannot be used for regular communications.

Only one client can connect to a server's *ServerSocket* any time. Multiple clients trying to connect to the same port on a server at the same time will be queued up. However, once the server has established a regular *Socket* for client and server communicaiton, the next queued client will be served. Incoming data is distinguished by the server port to which it is addressed, the client host and the client port from which it came.

Similarly, no more than one server socket can listen to a particular port on a host at one time. Therefore, since a server may need to handle many connections at once, server programs tend to be heavily multi-threaded. Generally speaking, a server socket listening on a port will only accept connections (just like the receptionist). It then passes off the actual processing of connections to a separate thread (just like an actual staff to serve the customer request).

The *ServerSocket* has three constructors that specify a paort to which the server socket is to listen for incoming connection requests, an optional maximum connection request queue length, and an optional Internet address. The Internet address argument allows *multihomed* hosts (that is, hosts with more than one

Internet address) to limit connections to a specific interface. These three constructors are listed as follows:

```
public ServerSocket(int port) throws IOException, BindException
public ServerSocket(int port, int queueLength) throws IOException,
  BindException
public ServerSocket(int port, int queueLength, InetAddress
  bindAddress) throws IOException
```

The operating system stores incoming connection requests addressed to a particular port in an FIFO (first-in-first-out) queue. The default length of the queue is normally 50 (this can vary from operating system to operating system). Incoming connections are refused if the queue is already full and the operating system is responsible for anaging the incoming. The default queue length can be changed by using the above constructors (up to the maximum length set by the operating system). Most medium size applications set the default queue length between 5 and 50.

Normally you only need to specify a port you want to listen on in the constructor of a *ServerSocket,* as shown below:

```
try {
   ServerSocket ss = new ServerSocket(80);
 }
 catch (IOException e) {
   System.err.println(e);
 }
```

The newly created *ServerSocket* object will attempt to bind to the port on the local host given by the port argument (80 in the above example). However, if the port is already occupied by server, then a *java.net.BindException,* a subclass of *java.io.IOException,* is thrown, as no more than one process or thread can listen to a particular port at a time. This includes non-Java processes or threads. For example, if there is already an HTTP server running on port 80 (the default port for an HTTP server), the above program segment will not be able to bind to port 80. On Unix systems (but not Windows or the Mac) users' programs must be running as root to bind to a port between 1 and 1023.

Port number zero (0) is a special number. It lets the operating system to pick an available port. The details of the allocated port can be found out by using the *getLocalPort()* method. This is useful if the client and the server have already established a separate channel of communication over which the chosen port number can be communicated.

The methods provided by the *ServerSocket* include:

- The *accept()* method. It is used to cause the server socket to listen and wait until an incoming connection is established. It returns an object of class *Socket* once a connection is made. This *Socket* object is then used to carry out a service for a single client.

- The information methods. The *getInetAddress( )* method returns the address of the host to which the socket is connected. The *getLocalPort( )* method returns the port on which the server socket listens for an incoming connection.
- Other interesting methods. The *toString( )* method returns the socket's address and port number as a string in preparation for printing. The *close( )* method closes the server socket.

6.2 Building TCP Clients and Servers

6.2.1 Essential Components of Communication

The Java client-server communication shows some basic steps that are needed to establish a TCP communication connection. The essential components of any communication are:

- The underlying communication protocol. In this instance, the TCP.
- The application's communication protocol.
- The client program.
- The server program.

In a TCP communication, the following steps are needed:

- Create the server socket and listen to client connection request.
- Create the client socket and issue a connection request to the server.
- The server accepts the connection. The communication channel is then established and communications between the client and the server can be carried out using the application's communication protocol.

The application's protocol is like the following in the simplest case:

- The client sends a "Hello, Server" string to the server.
- The server replies a string "You have connected to the Very Simple Server.".
- Both client and server exit.

6.2.2 Implementing a TCP Client Program

The following steps are carried out when implementing our example TCP client program:

- Create a socket for communicating with the server on a specific port.
- Create an *InputStream,* in our case, a *BufferedReader,* to receive responses from the server.

- Create an *OutputStream,* in our case, a *PrintWriter,* to send messages to the server.
- Write to the *OutputStream.*
- Read from the *InputStream.*
- Close the *InputStream,* the *OutputStream,* and the socket before the client exits.

The client program, named `C.java`, is as follows.

```
import java.io.*;
import java.net.*;
public class C {
  public static final int DEFAULT_PORT = 6789;
  public static void usage() {
    System.out.println("Usage: java C [<port>]");
    System.exit(0);
  }
  public static void main(String[] args) {
    int port = DEFAULT_PORT;
    Socket s = null;
    // parse the port specification
    if ((args.length != 0) && (args.length != 1)) usage();
    if (args.length == 0) port = DEFAULT_PORT;
    else {
      try {
        port = Integer.parseInt(args[0]);
      }
      catch(NumberFormatException e) {
        usage();
      }
    }
    try {
      BufferedReader reader;
      PrintWriter writer;
      // create a socket to communicate to the specified host and port
      s = new Socket("localhost", port);
      // create streams for reading and writing
       reader = new BufferedReader(new
       InputStreamReader(s.getInputStream()));
    writer = new PrintWriter(new
OutputStreamWriter(s.getOutputStream()));
      // tell the user that we've connected
      System.out.println("Connected to " + s.getInetAddress() +
        ":" + s.getPort());
      String line;
    // write a line to the server
      writer.println("Hello, Server");
    writer.flush();
      // read the response (a line) from the server
      line = reader.readLine();
      // write the line to console
      System.out.println("Server says: " + line);
    reader.close();
    writer.close();
    }
    catch (IOException e) {
      System.err.println(e);
    }
```

```
    // always be sure to close the socket
    finally {
      try {
        if (s != null) s.close();
      }
      catch (IOException e2) { }
    }
  }
}
```

The program assumes that the server runs on the "local host" (i.e., with an IP address of 127.0.0.1) and uses a default port number of 6789. The statement

```
   System.out.println("Connected to
"+s.getInetAddress()+":"+s.getPort());
```

displays the server host IP address and the port number that the client has connected to.

6.2.3 Implementing a TCP Server Program

When implementing the example TCP server, the following steps are carried out:

- Create a server socket to listen and accept client connection requests.
- Create an *InputStream,* in our case, a *BufferedReader,* to read messages from the client.
- Create an *OutputStream,* in our case, a *PrintWriter,* to send replies to the client.
- Read from the *InputStream.*
- Write to the *OutputStream.*
- Close the *InputStream,* the *OutputStream,* and the socket before the server exits.

The server program, named `S.java`, is as follows:

```
import java.net.*;
import java.io.*;
public class S {
  public final static int DEFAULT_PORT = 6789;
  public static void main (String args[]) throws IOException {
    Socket client;
    if (args.length != 1)
      client = accept (DEFAULT_PORT);
    else
      client = accept (Integer.parseInt (args[0]));
    try {
      PrintWriter writer;
      BufferedReader reader;
        reader = new BufferedReader(new
        InputStreamReader(client.getInputStream()));
        writer = new PrintWriter(new
        OutputStreamWriter(client.getOutputStream()));
    // read a line
```

```
    String line = reader.readLine();
    System.out.println("Client says: " + line);
    // write a line
      writer.println ("You have connected to the Very Simple
Server.");
    writer.flush();
    reader.close();
    writer.close();
    } finally { // closing down the connection
      System.out.println ("Closing");
      client.close ();
    }
  }
  static Socket accept (int port) throws IOException {
    System.out.println ("Starting on port " + port);
    ServerSocket server = new ServerSocket (port);
    System.out.println ("Waiting");
    Socket client = server.accept ();
    System.out.println ("Accepted from " + client.getInetAddress ());
    server.close ();
    return client;
  }
}
```

The server uses a default port of 6789 for communication. When a connection request is accepted, the server uses the following statement to display the IP address of the client computer:

```
System.out.println ("Accepted from " + client.getInetAddress ());
```

6.3 Examples in Java

The above simple communication example is of no practical use at all. A number of issues need to be addressed in order to improve the simple example for practical use:

- The exchange of multiple messages between the client and the server.
- The ability to run the server and the client programs on any Internet host.
- The ability for the server to deal with multiple client connections simultaneously.

6.3.1 Exchange of Multiple Messages

The first issue is to define the application's communication protocol to allow multiple exchanges of messages. Here is an example:

The server:

- After establishing the connection, the server sends an initial message to the client.
- The server waits for the client's messages.

- When the message arrives, the serve responds with an "OK" to the clients and then displays the message. If the incoming message is "Server Exit", then the server exits. Otherwise, it returns to the waiting step.

The client:

- After a successful connection, the client displays the initial response from the server.
- The client reads a line from the keyboard, and sends it to the server. Then the client reads the response from the server and displays it.
- The input string from the keyboard is checked. The client exits if the keyboard input string is "Server Exit" or the server is disconnected. Otherwise, it returns to the previous step.

The server program, named `S1.java`, is as follows:

```
import java.net.*;
import java.io.*;
public class S1 {
  public final static int DEFAULT_PORT = 6789;
  public static void main (String args[]) throws IOException {
    Socket client;
    if (args.length != 1)
      client = accept (DEFAULT_PORT);
    else
      client = accept (Integer.parseInt (args[0]));
    try {
      PrintWriter writer;
      BufferedReader reader;
        reader = new BufferedReader(new
        InputStreamReader(client.getInputStream()));
        writer = new PrintWriter(new
        OutputStreamWriter(client.getOutputStream()));
      writer.println ("You are now connected to the Simple Echo
Server.");
    writer.flush();
      for (;;) {
      // read a line
      String line = reader.readLine();
        // and send back ACK
      writer.println("OK");
      writer.flush();
      System.out.println("Client says: " + line);
      if (line.equals("Server Exit")) {
        break;
        }
      }
    reader.close();
    writer.close();
    } finally {
      System.out.println ("Closing");
      client.close ();
    }
  }
  static Socket accept (int port) throws IOException {
    System.out.println ("Starting on port " + port);
```

```
    ServerSocket server = new ServerSocket (port);
    System.out.println ("Waiting");
    Socket client = server.accept ();
    System.out.println ("Accepted from " + client.getInetAddress ());
    server.close ();
    return client;
  }
}
```

The client program, named `C1.java`, is as follows:

```
import java.io.*;
import java.net.*;
public class C1 {
public static final int DEFAULT_PORT = 6789;
public static void usage() {
    System.out.println("Usage: java C1 [<port>]");
    System.exit(0);
  }
  public static void main(String[] args) {
    int port = DEFAULT_PORT;
    Socket s = null;
    int end = 0;
    // parse the port specification
    if ((args.length != 0) && (args.length != 1)) usage();
    if (args.length == 0) port = DEFAULT_PORT;
    else {
      try {
        port = Integer.parseInt(args[0]);
      }
      catch(NumberFormatException e) {
      usage();
      }
    }
    try {
      PrintWriter writer;
      BufferedReader reader;
      BufferedReader kbd;
      // create a socket to communicate to the specified host and port
      //InetAddress myhost = getLocalHost();
      s = new Socket("localhost", port);
      // create streams for reading and writing
      reader = new BufferedReader(new
InputStreamReader(s.getInputStream()));
    OutputStream sout = s.getOutputStream();
    writer = new PrintWriter(new
OutputStreamWriter(s.getOutputStream()));
      // create a stream for reading from keyboard
      kbd = new BufferedReader(new InputStreamReader(System.in));
      // tell the user that we've connected
      System.out.println("Connected to " + s.getInetAddress() +
        ":" + s.getPort());
      String line;
      // read the first response (a line) from the server
      line = reader.readLine();
      // write the line to console
      System.out.println(line);
      while (true) {
        // print a prompt
        System.out.print("> ");
        System.out.flush();
        // read a line from console, check for EOF
        line = kbd.readLine();
        if (line.equals("Server Exit")) end = 1;
        // send it to the server
        writer.println(line);
```

```
        writer.flush();
        // read a line from the server
        line = reader.readLine();
        // check if connection is closed, i.e., EOF
        if (line == null) {
          System.out.println("Connection closed by server.");
          break;
        }
        if (end == 1) {
            break;   }
        // write the line to console
        System.out.println("Server says: " + line);
      }
      reader.close();
      writer.close();
    }
    catch (IOException e) {
      System.err.println(e);
    }
    // always be sure to close the socket
    finally {
      try {
        if (s != null) s.close();
      }
      catch (IOException e2) { }
    }
  }
}
```

6.3.2 Executing the Programs on Internet Hosts

The first requirement to execute the client-server programs on Internet hosts is to know the IP addresses or/and the host names of the computers. The following program, named `InetExample.java`, from the text book displays the details of a host:

```
import java.net.*;
import java.io.*;
public class InetExample {
  public static void main (String args[]) {
    printLocalAddress ();
    Reader kbd = new FileReader (FileDescriptor.in);
    BufferedReader bufferedKbd = new BufferedReader (kbd);
    try {
      String name;
      do {
        System.out.print ("Enter a hostname or IP address: ");
        System.out.flush ();
        name = bufferedKbd.readLine ();
        if (name != null)
          printRemoteAddress (name);
      } while (name != null);
      System.out.println ("exit");
    } catch (IOException ex) {
      System.out.println ("Input error:");
      ex.printStackTrace ();
    }
  }
  static void printLocalAddress () {
    try {
      InetAddress myself = InetAddress.getLocalHost ();
```

```
      System.out.println ("My name : " + myself.getHostName ());
      System.out.println ("My IP : " + myself.getHostAddress ());
      System.out.println ("My class : " + ipClass (myself.getAddress
()));
    } catch (UnknownHostException ex) {
      System.out.println ("Failed to find myself:");
      ex.printStackTrace ();
    }
  }
  static char ipClass (byte[] ip) {
    int highByte = 0xff & ip[0];
    return (highByte < 128) ? 'A' : (highByte < 192) ? 'B' :
      (highByte < 224) ? 'C' : (highByte < 240) ? 'D' : 'E';
  }
  static void printRemoteAddress (String name) {
    try {
      System.out.println ("Looking up " + name + "...");
      InetAddress machine = InetAddress.getByName (name);
      System.out.println ("Host name : " + machine.getHostName ());
      System.out.println ("Host IP : " + machine.getHostAddress ());
      System.out.println ("Host class : " +
                          ipClass (machine.getAddress ()));
    } catch (UnknownHostException ex) {
      System.out.println ("Failed to lookup " + name);
    }
  }
}
```

The *main( )* method first calls the *PrintLocalAddress( )* method to display the local host name and IP address. Then it sits in a loop that reads host names from the keyboard and uses the *PrintRemoteAddress( )* method to display the *InetAddress* information about the host.

To allow our client-server program to run on any host, we only need to change the client program; the server program can remain the same. Here is the new client program, named as `C2.java`:

```
import java.io.*;
import java.net.*;
public class C2 {
  public static final int DEFAULT_PORT = 6789;
  public static void usage() {
    System.out.println("Usage: java C2 <serverhost>");
    System.exit(0);
  }
  public static void main(String[] args) {
    int port = DEFAULT_PORT;
    String address = "";
    Socket s = null;
    int end = 0;
    // parse the port specification
    if ((args.length != 0) && (args.length != 1)) usage();
    if (args.length == 0) {
      port = DEFAULT_PORT;
    address = "localhost";
    } else {
        address = args[0];
    }
    try {
```

```
      PrintWriter writer;
      BufferedReader reader;
      BufferedReader kbd;
      // create a socket to communicate to the specified host and port
      s = new Socket(address, port);
      // create streams for reading and writing
        reader = new BufferedReader(new
        InputStreamReader(s.getInputStream()));
    OutputStream sout = s.getOutputStream();
    writer = new PrintWriter(new
OutputStreamWriter(s.getOutputStream()));
      // create a stream for reading from keyboard
      kbd = new BufferedReader(new InputStreamReader(System.in));
      // tell the user that we've connected
      System.out.println("Connected to " + s.getInetAddress() +
        ":" + s.getPort());
      String line;
      // read the first response (a line) from the server
      line = reader.readLine();
      // write the line to console
      System.out.println(line);
      while (true) {
        // print a prompt
        System.out.print("> ");
        System.out.flush();
        // read a line from console, check for EOF
        line = kbd.readLine();
        if (line.equals("Server Exit")) end = 1;
        // send it to the server
        writer.println(line);
        writer.flush();
        // read a line from the server
        line = reader.readLine();
        // check if connection is closed, i.e., EOF
        if (line == null) {
          System.out.println("Connection closed by server.");
          break;
        }
        if (end == 1) {
        break;
      }
        // write the line to console
        System.out.println("Server says: " + line);
      }
    reader.close();
    writer.close();
    }
    catch (IOException e) {
      System.err.println(e);
    }
    // always be sure to close the socket
    finally {
      try {
        if (s != null) s.close();
      }
      catch (IOException e2) { }
    }
  }
}
```

In this version, we use a variable

```
String address = "";
```

to store the IP address entered from the keyboard. The socket is then created using the following statement:

```
s = new Socket(address, port);
```

6.3.3 Supporting Multiple Clients

To support multiple clients, only the server program needs to be changed; the client program remains the same. When the server is initialized, we obtain the server socket and wait for client connection requests. When a client connection request is accepted, we use a thread to deal with the accepted incoming client connection. The server then goes back to wait for new connection requests. Clients can issue two commands during this time, one is a "Client Exit" command, telling the server that the current client is willing to disconnect. The other is the "Server Exit" command, in which the whole program exits.

```
import java.net.*;
import java.io.*;

public class S3 extends Thread {
  public final static int DEFAULT_PORT = 6789;
  private Socket client = null;

  public S3(Socket inSock) {
    super("echoServer");
    client = inSock;
  }

  public void run() {
    Socket cSock = client;
    PrintWriter writer;
    BufferedReader reader;
    try {
    String line;
        System.out.println ("Accepted from " +
cSock.getInetAddress());
        reader = new BufferedReader(new
        InputStreamReader(cSock.getInputStream()));
        writer = new PrintWriter(new
        OutputStreamWriter(cSock.getOutputStream()));
      writer.println ("You are now connected to the Simple Echo
Server.");
    writer.flush();
      for (;;) {
      // read a line
      line = reader.readLine();
        // and send back ACK
      writer.println("OK");
      writer.flush();
      System.out.println("Client says: " + line);
      if (line.equals("Server Exit") || line.equals("Client Exit"))
break;
      }
```

```
      System.out.println ("Closing the client " +
cSock.getInetAddress());
      reader.close();
    writer.close();
      cSock.close ();
    if (line.equals("Server Exit")) {
        System.out.println ("Closing the server");
        // server.close ();
      System.exit(0);
    }
    } catch (IOException e1) {
      System.err.println("Exception: " + e1.getMessage());
    System.exit(1);
    }
  }

  public static void main (String args[]) {
    ServerSocket server = null;
    try {
      server = new ServerSocket(DEFAULT_PORT);
      System.out.println ("Starting on port " + DEFAULT_PORT);
    } catch (IOException e) {
      System.err.println("Exception: could't make server socket.");
    System.exit(1);
    }
    while (true) {
      Socket incomingSocket = null;
      // wait fot a connection request
      System.out.println("Waiting...");
      try {
        incomingSocket = server.accept();
        // call a thread to deal with a connection
        S3 es = new S3(incomingSocket);
        es.start();
      } catch (IOException e) {
        System.err.println("Exception: could't make server socket.");
         System.exit(1); }
    }
  }
}
```

6.4 A More Complex Example - A Java Messaging Program using TCP

In this section we use connection-oriented mechanism (TCP) to build a simple "messaging" system. The system consists of a server program and a client program. The client has two basic functions: a message sending function and a message receiving function. Using the message sending function, one user (the sender) can send a message addressed to another user (the receiver) to the system The server will then store the message for the receiver until the receiver uses the message receiving function of the client to retrieve his/her messages from the server. Then the received message is deleted from the server. Also, the sender should be able to accept simultaneous connections from multiple clients and should be able to send the same message to a number of receivers simultaneously.

6.4.1 The Design

The system is divided into a server component and a client component. The action sequence of the server is as follows:

- Wait for a client to connect and build a connection when accept request from client
- Create a new server thread object to handle the connection
- Receive the username from the sender and store it in the username list
- Broadcast the username list to all users
- Receive the message addressed to specified users (receiver) from the client and store the message for those users respectively.
- Send back to client all messages stored for a particular user when required.

The action sequence of the client is as follows:

- Set up an interface for the user to interact with the system
- Connect to the server
- Send own username to the server
- Receive username list from the server and show it in the system
- Get the message from the user's input
- Get the selection of receivers from the username list. The specified message will be addressed to all selected receivers
- Send the message to the server
- Retrieve all messages for the user from the server
- Close the connection to the server

For better scalability and modularity, the client program is divided into two classes. One class, MessageClient, provides functions for connecting to the server; storing a message and retrieving messages. Another class, MessageApplet, provides the graphical user interface to the user and invokes functions in MessageClient to carry out the communication to the server. To achieve this, a MessageApplet is associated with a MessageClient.

The server program also has two classes: MessageServer and MessageServerThread. The MessageServer class uses a username list to store the names for all users.. The MessageServerThread class, which is used to handle interaction with individual MessageClient, needs to store the messages belong to the client. The MessageServer also has a list of MessageServerThread. According to the creation and destroy of threads, MessageServer constantly updates its username list and MessageServerThread list.

6.4.2 The Implementation

To launch a server, a user can choose a port number to run the server on a machine. When the port number is not specified, a default port number (6789) is used. The user can also specify the Internet address and the port number of the server to run the client java application. When the address of the server is not provided, a default value is set to localhost.

A simple application protocol is designed to enable the communication between the client and the server: Any messages passing through the system is started with a number at the range of 1 to 4.

- Number 1 indicates storing message. It is used only from the client to the server. Following the number 1, is a number indicating the number of receivers. Then follows an index list of receivers. After that is the message.
- Number 2 indicates retrieval of messages for a particular user. When the client requests its messages from the server, it merely sends a number 2. The response of the server also starts with a number 2. After that, a number indicating the number of messages is sent, followed by the messages one by one.
- Number 3 indicates the query and answer of a message. The server sends a number 3 to the client at the start of a connection session. The client replies with its username following a number 3.
- Number 4 indicates the broadcast of the username list. It is only used by the server: a server sends a number 4 to the client indicating the following messages is the username list. Firstly, the number of usernames is sent. Then, the usernames are sent one by one.

Empty strings are not permitted to be stored on the server. When the client runs in java applet inside a web page, the Java policy needs to be set to allow the client machine to trust the server program. The general method to set this is through the running of the policytool on the client machine and to add socket permission into the permission list. A message without a receiver cannot be sent.

The username is contained in the Applet parameter from the HTML file. If it is not specified, the username will be set to the IP address and port number of the client machine in the format of xxx.xxx.xxx.xxx:yyy.

To launch the server, type in `java MessageServer [portno]` in the command line and the server will up on the specified port. If port number is not provided, the server will use the default port number of 6789. The server prints a message and awaits until a client requests a connection. For every message stored the server prints out the received message. For a client retrieving messages, the server prints out a message to indicate the event. When a client quits the program, the server prints out the information indicating the corresponding username is removed from the username list.

The MessageApplet class is an applet runs in a web page or appletviewer. Files message1.html, message2.html and message3.html are created with different value of username parameters. Type `appletviewer message[1-3].html` will start the applet with a particular username. Three parameters, the host name, the port number and the username, can be set as parameters in the HTML file. If they are not specified, the system will use the default values, which are localhost and 6789. A textfield is placed on the upper part of the applet with a store button on its right. Type message in the textfield and click the store button then the message will be sent to the server. The textfield is then cleared. Underneath the textfield, there is a textarea that supports multiple lines. Right to the textarea is a button called retrieve. By clicking this button, messages are retrieved from the server and displayed in the textarea. The textarea is not editable. The messages will stay in the textarea until next retrieval. At the bottom, there is a List showing all the usernames of clients currently connected to the server. To send a message addressing to particular users, the username should be selected and shown in highlighted color. The list will be updated automatically during the execution of the program.

6.4.3 The Programs

The MessageServer.java program is shown below:

```
// MessageServer.java
import java.io.*;
import java.net.*;
import java.util.Vector;

class MessageServer {
  Vector threadlist=new Vector();
  Vector userlist=new Vector();
  ServerSocket welcomesocket;
  Socket connectionsocket;
  int port=6789;
  public void run(String[] argv) {
    if((argv.length<1)) {
      System.err.println("No port number is given! using default");
    } else {
      if((argv.length>2)) {
        System.err.println("Usage: java MessageServer port");
        ystem.exit(0);
      } else {
        try {
          port=Integer.parseInt(argv[0]);
        } catch(Exception e) {
          System.err.println("Usage: java MessageServer port");
          System.exit(0);
        }
      }
    }
    try {
      welcomesocket=new ServerSocket(port);
      while(welcomesocket!=null) {
        connectionsocket=welcomesocket.accept();
        if(connectionsocket!=null) {
          System.out.println("Connection built to"
            +connectionsocket.getInetAddress().toString()
            +": "+connectionsocket.getPort());
        }
        MessageServerThread mst=new
MessageServerThread(connectionsocket,this);
```

```
        threadlist.add(mst);
        mst.getUsername();
        mst.start();

      }
    }catch(Exception e){System.out.println("Failed to provide
service");}
  }

  public void addMessages(int[] index,String themessage) {
    for(int i=0;i<index.length;i++) {
      MessageServerThread
mst=(MessageServerThread)threadlist.elementAt(index[i]);
      mst.addMessage(themessage);
    }
  }

  public void addUser(String username) {
    System.out.println("add a user into the list");
    userlist.add(username);
    broadcastUserlist();
  }

  public void removeUser(MessageServerThread mst) {
    System.out.println("Remove a user who exited");
    int i=threadlist.lastIndexOf(mst);
    threadlist.removeElement(mst);
    userlist.removeElementAt(i);
    broadcastUserlist();
  }

  public void broadcastUserlist() {
    for(int i=0;i<threadlist.size();i++) {
      MessageServerThread
mst=(MessageServerThread)threadlist.elementAt(i);
        mst.sendUserlist(userlist);
    }
  }

  public static void main(String[] args) {
    MessageServer ms=new MessageServer();
    ms.run(args);
  }
}
```

Here is the MessageServerThread.java program:

```
// MessageServerThread.java
import java.util.*;
import java.net.*;
import java.io.*;

public class MessageServerThread extends Thread
{
  MessageServer server;
  Socket socket;
  DataInputStream dis; // input communication channel from client
  DataOutputStream dos; // output communication channel to client
  Vector clientmessages=new Vector();
  String clientmessage="";

  public MessageServerThread(Socket localSocket, MessageServer
localServer)
  {
    server = localServer;
    socket = localSocket;
    try {
      dis = new DataInputStream(socket.getInputStream());
```

```
      dos = new DataOutputStream(socket.getOutputStream());
    }`catch(Exception e)`{
      System.err.println("exception at ServerThread");
      dis=null;
      dos=null;
    }
  }

  public void getUsername() {
    try {
      dos.writeInt(3);
      dos.flush();
      String readusername=dis.readUTF();
      System.out.println(readusername+" has join the system");
      server.addUser(readusername);
      }catch(Exception e) {}
    }

  // waits for input from the client - either insert or delete
  public void run()
  {
    while(true) {
      try {
        int a=dis.readInt();
        if(a==1) {
          int noofusers=dis.readInt();
          int[] index=new int[noofusers];
          for(int i=0;i<noofusers;i++) {
            index[i]=dis.readInt();
          }
          clientmessage=dis.readUTF();
          server.addMessages(index,clientmessage);
          System.out.println("Recieved message: "+clientmessage);
        } else if(a==2) {
          dos.writeInt(2);
          dos.writeInt(clientmessages.size());
          dos.flush();
          for(int i=0;i<clientmessages.size();i++) {
            dos.writeUTF((String)clientmessages.elementAt(i));
            dos.flush();
          }
          clientmessages.removeAllElements();
          System.out.println("Message retrieved and reset");
        }
      } catch(Exception e) {
        System.out.println("Client closed the connection");
        server.removeUser(this);
        break;
      }
    }
  }

  public void addMessage(String themessage) {
    clientmessages.add(themessage);
  }

  public void sendUserlist(Vector ulist)
  {
    String s;
    try {
      dos.writeInt(4);
      dos.writeInt(ulist.size()); // tell user number of usernames
      for(int i = 0; i < ulist.size(); i++) {
        dos.writeUTF((String)ulist.elementAt(i));
      }
      dos.flush(); // flush to ensure sending data
    } catch(Exception e){System.out.println("exception at sending user
list"+e);}
```

```
  }
}
```

The MessageApplet.java program contains the source code of both the MessageApplet class and the MessageClient class. It is shown below:

```
// MessageApplet.java
import java.io.*;
import java.net.*;
import java.applet.*;
import java.awt.*;
import java.awt.event.*;

public class MessageApplet extends Applet implements ActionListener{
  TextField sendmessage=new TextField(40);
  TextArea returnmessage=new TextArea(15,40);
  List userlist=new List(10,true);
  Button store=new Button("  Store  ");
  Button retrival=new Button("Retrival");
  String message;
  String[] messages;
  MessageClient mc;
  String hostname;
  InetAddress thehost;
  int portno;
  String username="";

  public void init()
  {
    try {
      username=getParameter("username");
      hostname=getParameter("server");
      portno=Integer.parseInt(getParameter("port"));
    }catch(Exception e){
      System.out.println("Missing parameters! using default");
      hostname="localhost";
      portno=6789;
    }
    try {
      InetAddress thehost=InetAddress.getByName(hostname);
      mc=new MessageClient(thehost,portno,this);
      mc.start();
    }catch(Exception ee){
      showStatus("Cannot connect");
      System.out.println("Cannot connect");
    }
    setSize(400,450);
    store.addActionListener(this);
    retrival.addActionListener(this);
    returnmessage.setEditable(false);
    add(sendmessage);
    add(store);
    add(returnmessage);
    add(retrival);
    add(userlist);
  }

  public void actionPerformed(ActionEvent e)
  {
    if(e.getSource()==store) {
      if((sendmessage.getText().length()>=1)
        &&(userlist.getSelectedIndexes().length>0)) {
        mc.addMessage(userlist.getSelectedIndexes(),
          sendmessage.getText());
        sendmessage.setText("");
      }
    } else if(e.getSource()==retrival) {
```

```
      mc.getMessage();
    }
  }

  public void updateMessageBoard(String[] messages) {
    returnmessage.setText("");
    for(int i=0;i<messages.length;i++)
    returnmessage.append(messages[i]+"\n");
  }

  public void updateUserlist(String[] users) {
    userlist.removeAll();
    for(int i=0;i<users.length;i++)
    userlist.add(users[i]);
  }
}

class MessageClient extends Thread{
  DataInputStream infromserver;
  DataOutputStream outtoserver;
  MessageApplet ma;
  Socket clientsocket;
  public MessageClient(InetAddress host,int port,MessageApplet ma)
  {
    this.ma=ma;
    try {
      clientsocket=new Socket(host,port);
      System.out.println("Connection has been built to the server
        at "+host+":"+port);
    } catch(Exception e)
      {System.out.println("Cannot connect to the specified
        server!"+host+":"+port);}
    try {
      outtoserver=new
        DataOutputStream(clientsocket.getOutputStream());
      infromserver=new DataInputStream(clientsocket.getInputStream())
    } catch(Exception e) {System.out.println("Input or Output
      denied!");}
  }

  public void run()
  {
    try {
        while(infromserver!=null) {
        int a=infromserver.readInt();
        switch(a) {
          case(2):
            int num=infromserver.readInt();
            String[] returnvalue=new String[num];
            for(int i=0;i<num;i++)
              returnvalue[i]=infromserver.readUTF();
              ma.updateMessageBoard(returnvalue);
              break;
          case(3):
            if(ma.username.compareTo("")!=0)
              outtoserver.writeUTF(ma.username);
            else
              outtoserver.writeUTF(
                clientsocket.getLocalAddress().getHostAddress()
                +":"+clientsocket.getLocalPort());
            outtoserver.flush();
            break;
         case(4):
           int num1=infromserver.readInt();
           String[] returnvalue1=new String[num1];
           for(int i=0;i<num1;i++)
             returnvalue1[i]=infromserver.readUTF();
           ma.updateUserlist(returnvalue1);
```

```
            break;
          }
        }
      }catch(Exception e) {System.out.println("Failed to receive
        data"+e);}
    }

    public void addMessage(int[] indexes,String themessage)
    {
      try {
        outtoserver.writeInt(1);
        outtoserver.writeInt(indexes.length);
        for(int i=0;i<indexes.length;i++) {
          outtoserver.writeInt(indexes[i]);
        }
        outtoserver.writeUTF(themessage);
        outtoserver.flush();
      }catch(Exception e) {System.out.println("Failed to save message
        to server!"+e);}
    }

    public void getMessage()
    {
      try {
        outtoserver.writeInt(2);
        outtoserver.flush();
      }catch(Exception e) {System.out.println("Failed to recieve message
        from server!"+e);}
    }

    public void destory()
    {
      try {
        clientsocket.close();
      }catch(Exception e) {System.out.println("Failed to close the
        connection!"+e);}
    }
}
```

File message1.html is listed below:

```
<!DOCTYPE HTML><HTML><HEAD></HEAD><BODY>
<APPLET CODE="MessageApplet.class" CODEBASE="." WIDTH=400 HEIGHT=300>
<PARAM name="server" value="localhost">
<PARAM name="port" value="6789">
<PARAM name="username" value="User 1">
</APPLET>
</BODY></HTML>
```

The other two files, message2.html and message3.html, are almost the same as the message1.html file, except that the "username" is changed to "User 2" and "User 3", respectively.

6.5 Datagram Communications in Java

6.5.1 Why Datagram Communication ?

For many applications the convenience of the TCP sockets outweighs the overhead required. However, for certain applications it is much more efficient to utilize datagrams: small, fixed-length messages sent between computers of a network.

A TCP connection carries a number of overhead factors. First, one needs to go through several steps to open a connection. This takes a certain time. Once a connection is open, sending and receiving data involves several steps. The final step is to tear down the connection after the communication. If one is to send a large amount of data that must be reliably delivered, then the TCP protocol is suitable. However, if one only needs to send a short, simple message quickly, then all these steps may not be worthwhile.

The difference between datagram and TCP connections is like the difference between post offices and telephones. With a telephone, you make a connection to a specific telephone number, if the person on the destination answers the phone, you two are able to talk for a certain period of time, exchanging an arbitrary amount of information, and then you close the connection. With a post offices, you typically send a letter with the destination address on the envelope. Because of errors in sorting, transportation and delivery and delays you cannot be certain if or when the person addressed receives the letter. The only way to know is to request and receive some kind of acknowledgement. You may retry several times if you get no response, then give up.

On an IP network such as the Internet, the UDP (User Datagram Protocol) is used to transmit fixed-length datagrams. This is the protocol that Java taps with the *DatagramSocket* class.

6.5.2 Java Datagram-based Classes

Datagrams have the following advantages:

- *Speed.* UDP involves low overhead since there is no need to set up connections, to maintain the order and correctness of the message delivery, or to tear down the connections after the communication.
- *Message-oriented instead of stream-oriented.* If the message to be sent is small and simple, it may be easier to simply send the chunk of bytes instead of going through the steps of converting it to and from streams.

Two *java.net* classes define the heart of datagram-based messaging in Java: they are the *DatagramSocket* and the *DatagramPacket* classes. A *DatagramSocket* is an interface through which *DatagramPackets* are transmitted. A *DatagramPacket* is simply an IP-specific wrapper for a block of data.

The *DatagramSocket* class provides a good interface to the UDP protocol. This class is responsible for sending and receiving *DatagramPacket* via the UDP protocol. The most commonly used *DatagramSocket* methods are listed below:

- *DatagramSocket*(). Constructor comes in two formats: one is used to specify the local port used and the other picks an ephemeral local port for you.

- *receive*(). Receive a *DatagramPacket* from any remote server.
- *send*(). Send a *DatagramPacket* to the remote server specified in the *DatagramPacket.*
- *close*(). Tear down local communication resources. After this method is called, the object involved is released.
- *getLocalPost*(). Return the local port this *DatagramSocket* is using.

Note that there are two flavors of *DatagramSocket:* those created to send *DatagramPackets,* and those created to receive *DatagramPackets.* A "send" *DatagramSocket* uses an ephemeral local port assigned by the native UDP implementation. A "receive" *DatagramSockect* requires a specific local port number.

A *DatagramPacket* represents the datagram transmitted via a *DatagramSocket.* The most frequently used methods of *DatagramPacket* are:

- *DatagramPacket*(). Constructor comes in two formats: a "send" packet and a "receive" packet. For the *send* packet, you need to specify a remote *InetAddress* and a port to which the packet should be sent, as well as a data buffer and length to be sent. For the *receive* packet, you need to provide an empty buffer into which data should be stored, and the maximum number of bytes to be stored.
- *getAddress*(). This method allows one to either obtain the *InetAddress* of the host that sends the *DatagramPacket,* or to obtain the *InetAddress* of the host to which this packet is addressed.
- *getData*(). This method allows one to access the raw binary data wrapped in the *DatagramPacket.*
- *getLength*(). This method allows one to determine the length of data wrapped in the *DatagramPacket* without getting a reference to the data block itself.
- *getPort*(). This method returns either the port of the server to which this packet will be sent, or the port of the server that sends this packet, depending on whether the packet was built to be sent or built to receive data.

It is also possible to exchange data via datagrams using the *Socket* class. To do so, you must use one of the *Socket* constructors that includes the Boolean *useStream* parameter, as in,

```
Socket(InetAddress address, int port, Boolean useStream)
```

and set *useStream* to *false.* This tells *Socket* to use the faster UDP. The advantage to using this interface is that it provides a stream interface to a datagram. Also, there is no need to instantiate and maintain a separate *DatagramPacket* to hold the data.

But there are significant disadvantages as well. First, there is no way to detect if a particular datagram sent does not arrive at the destination. Your stream interface can lie to you. Second, you still have to go through the hassle of setting up the connection.

UDP ports are separate from TCP ports. Each computer has 65,536 UDP ports as well as its 65,536 TCP ports. You can have a ServerSocket bound to TCP port 20 at the same time as a DatagramSocket is bound to UDP port 20. Most of the time it should be obvious from context whether or not I'm talking about TCP ports or UDP ports.

6.6 Building UDP Servers and Clients

6.6.1 Sending and Receiving UDP Datagrams

To send data to a particular server, you must first convert the data into byte array. Next you pass this byte array, the length of the data in the array (most of the time this will be the length of the array), the local *InetAddress* and the port to which you wish to send it, into the *DatagramPacket* constructor. For example,

```
try {
  InetAddress turin = new InetAddess("turin.cm.deakin.edu.au");
  int chargen = 19;
  String s = "My second UDP Packet";
  byte[] b = s.getBytes();
  DatagramPacket dp = new DatagramPacket(b, b.length, turin, chargen);
}
catch (UnknownHostException e) {
  System.err.println(e);
}
```

Next you create a *DatagramSocket* object and pass the packet to its *send()* method. For example,

```
try {
  DatagramSocket sender = new DatagramSocket();
  sender.send(dp);
}
catch (IOException e) {
  System.err.println(e);
}
```

To receive data sent to you, you construct a *DatagramSocket* object with a port on which you want to listen. Then you pass an empty *DatagramPacket* object to the *DatagramSocket*'s *receive()* method.

```
  public synchronized void receive(DatagramPacket dp) throws
IOException
```

The calling thread blocks until a datagram is received. Then the datagram dp is filled with the data from that datagram. You can then use *getPort()* and *getAddress()* to tell where the packet came from, *getData()* to retrieve the data, and *getLength()* to see how many bytes were in the data. If the received packet was too long for the buffer, then it's truncated to the length of the buffer. For example,

```
try {
  byte buffer = new byte[65536]; // maximum size of an IP packet
  DatagramPacket dp = new DatagramPacket(buffer, buffer.length);
  DatagramSocket ds = new DatagramSocket(2134);
  ds.receive(dp);
  byte[] data = dp.getData();
  String s = new String(data, 0, data.getLength());
```

```
  System.out.println("Port " + dp.getPort() + " on " + dp.getAddress()
   + " sent this message:");
  System.out.println(s);
}
catch (IOException e) {
  System.err.println(e);
}
```

6.6.2 Datagram Server

The steps for setting up a datagram server are as follows:

- Create a *DatagramPocket* for receiving the data, indicating the buffer to hold the data and the maximum length of the buffer.
- Create a *DatagramSocket* on which to listen.
- Receive a packet from a client.

Here is a simple server example (`DatagramReceive.Java`).

```
import java.net.*;
public class DatagramReceive {
  static final int PORT = 7890;
  public static void main( String args[] ) throws Exception {
    String theReceiveString;
    byte[] theReceiveBuffer = new byte[ 2048 ];

    // Make a packet to receive into...
    DatagramPacket theReceivePacket =
      new DatagramPacket( theReceiveBuffer, theReceiveBuffer.length );

    // Make a socket to listen on...
    DatagramSocket theReceiveSocket = new DatagramSocket( PORT );

    // Receive a packet...
    theReceiveSocket.receive( theReceivePacket );

    // Convert the packet to a string...
    theReceiveString =
      new String( theReceiveBuffer, 0, theReceivePacket.getLength() );

    // Print out the string...
    System.out.println( theReceiveString );

    //  Close the socket...
    theReceiveSocket.close();
    }
}
```

The *main()* method first builds an empty *DatagramPacket* object using a designated buffer. Then it creates a *DatagramSocket* using the default port. The *DatagramSocket* will receive a *DatagramPacket* which will fill the previous *DatagramPacket.* Then the program extracts the string from this datagram packet and prints it out.

6.6.3 Datagram Client

The steps of setting up a datagram client are as follows:

- Find the destination's IP address.
- Create a *DatagramPacket* based on the destination address and the data to be sent.
- Create a *DatagramSocket* for sending the packet.
- Send the *DatagramPacket* over the *DatagramSocket.*

Here is the program named `DatagramSend.java`:

```
import java.net.*;
import java.io.IOException;

public class DatagramSend {
  static final int PORT = 7890;
  public static void main( String args[] ) throws Exception {
    String theStringToSend = "I'm a datagram and I'm O.K.";
    byte[] theByteArray = new byte[ theStringToSend.length() ];
    theByteArray = theStringToSend.getBytes();

    // Get the IP address of our destination...
    InetAddress theIPAddress = null;
    try {
      theIPAddress = InetAddress.getByName( "localhost" );
    } catch (UnknownHostException e) {
      System.out.println("Host not found: " + e);
      System.exit(1);
    }

    // Build the packet...
    DatagramPacket thePacket = new DatagramPacket( theByteArray,
      theStringToSend.length(),
      theIPAddress,
      PORT );

    // Now send the packet
    DatagramSocket theSocket = null;
    try {
      theSocket = new DatagramSocket();
    } catch (SocketException e) {
        System.out.println("Underlying network software has failed:
                           + e);
        System.exit(1);
    }
    try {
      theSocket.send( thePacket );
    } catch (IOException e) {
      System.out.println("IO Exception: " + e);
    }
    theSocket.close();
    }
}
```

The *main()* method first designs a string "`I'm a datagram and I'm O.K.`" and translates it into a byte array. It also gets the local IP address using the *InetAddress* class and then builds a *DatagramPacket* object using the above byte array, the length of the string, the IP address and the default port. Then it creates a sending *DatagramSocket* and sends out the *DatagramPacket* built before by invoking the *send()* method. After that, the program closes the *DatagramSocket.*

6.7 Summary

As mentioned in Chapter 5, the BSD Internet domain sockets use the TCP/IP (or UDP/IP) protocol suite as the communication protocols among processes. TCP is a connection-oriented protocol and it implements a connection as a stream of bytes from source to destination, while UDP is a connectionless transport protocol and uses datagrams to implement its communication. In this chapter we discussed the datagrams for the TCP/UDP communications. Java provides the reliable stream-based communication for TCP as well as the unreliable datagram communication for UDP. The stream-based communication is like a telephone system which has the connection built first, whereas datagram communication is like a mail system which has no fixed connection. Java various input/output streams allow application programs to input and output various data, such as bytes, string, file etc. The Java socket API provides the basis of TCP/UDP communication. Various examples presented in the chapter have been a great help for readers in writing Java TCP/UDP programs of their own.

Exercises

6.1 How does one build a connection between a Java socket and a Java server socket? Write a program. 6.2

6.2 How does one close a socket? Give an example. 6.2

6.3 How does one write a TCP server program? Give an example. 6.3.3

6.4 Write a client/server program using *stream* communication to implement the following functions: 6.4

1. The server can accept multiple clients and they can exchange multiple messages.
2. The server can send a message to all clients simultaneously.

Hint: use a separate thread to manage each connection and use *suspend()* and *resume()* methods of *Thread* class.

6.5 Compile the three programs in the chat example (6.4) and test run the server on one machine, the client instances on at least two machines. If possible, form a group to test run the chat program (the server runs on one site and the client is run on multiple sites).

6.6 Discuss the advantages and disadvantages of TCP and UDP. 6.5.1

6.7 Is it possible to exchange data via datagrams using the *Socket* class? How? 6.5.2

6.8 How does one write a datagram client program? 6.6.3

6.9 Write a client/server program using *datagram* communication to implement the following functions: the server can communicate with multiple clients and print out each client's message. Hint: use *getAddress()* to get a client address. 6.7