

1. TCPEchoServer

//Server that echoes back client's messages.
//At end of dialogue, sends message indicating number of
//messages received. Uses TCP.

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

public class TCPEchoServer
{
    private static ServerSocket servSock;
    private static final int PORT = 1234;

    public static void main(String[] args)
    {
        System.out.println("Opening port...\n");
        try
        {
            servSock = new ServerSocket(PORT);    //Step 1.
        }
        catch(IOException ioEx)
        {
            System.out.println(
                "Unable to attach to port!");
            System.exit(1);
        }
        do
        {
            handleClient();
        }
```

```

    }while (true);
}

private static void handleClient()
{
    Socket link = null;                                //Step 2.

    try
    {
        link = servSock.accept();                      //Step 2.

        Scanner input =
            new Scanner(link.getInputStream()); //Step 3.
        PrintWriter output =
            new PrintWriter(
                link.getOutputStream(),true); //Step 3.

        int numMessages = 0;
        String message = input.nextLine(); //Step 4.
        while (!message.equals("***CLOSE***"))
        {
            System.out.println("Message received.");
            numMessages++;
            output.println("Message " + numMessages
                + ": " + message); //Step 4.
            message = input.nextLine();
        }
        output.println(numMessages
            + " messages received."); //Step 4.
    }
    catch(IOException ioEx)
    {
        ioEx.printStackTrace();
    }

    finally
    {
        try
        {
            System.out.println(
                "\n* Closing connection... *");
            link.close(); //Step 5.
        }
        catch(IOException ioEx)
        {
            System.out.println(
                "Unable to disconnect!");
            System.exit(1);
        }
    }
}

```

```

    }
}
}

```

Setting up the corresponding client involves four steps:

1. Establish a connection to the server.

We create a *Socket* object, supplying its constructor with the following two arguments:

- the server's IP address (of type *InetAddress*);
- the appropriate port number for the service.

(The port number for server and client programs must be the same, of course!)

For simplicity's sake, we shall place client and server on the same host, which will allow us to retrieve the IP address by calling static method *getLocalHost* of class *InetAddress*. For example:

```

Socket link =
    new Socket(InetAddress.getLocalHost(), 1234);

```

2. Set up input and output streams.

These are set up in exactly the same way as the server streams were set up (by calling methods *getInputStream* and *getOutputStream* of the *Socket* object that was created in step 2).

3. Send and receive data.

The *Scanner* object at the client end will receive messages sent by the *PrintWriter* object at the server end, while the *PrintWriter* object at the client end will send messages that are received by the *Scanner* object at the server end (using methods *nextLine* and *println* respectively).

4. Close the connection.

This is exactly the same as for the server process (using method *close* of class *Socket*).

The code below shows the client program for our example. In addition to an input stream to accept messages from the server, our client program will need to set up an input stream (as another *Scanner* object) to accept user messages from the keyboard. As for the server, the lines of code corresponding to each of the above steps have been clearly marked with emboldened comments.

2. TCPEchoClient

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

public class TCPEchoClient
{
    private static InetAddress host;
    private static final int PORT = 1234;

    public static void main(String[] args)
    {
        try
        {
            host = InetAddress.getLocalHost();
        }
        catch(UnknownHostException uhEx)
        {
            System.out.println("Host ID not found!");
            System.exit(1);
        }
        accessServer();
    }

    private static void accessServer()
    {
        Socket link = null;                                //Step 1.

        try
        {
            link = new Socket(host,PORT);                    //Step 1.

            Scanner input =
                new Scanner(link.getInputStream()); //Step 2.

            PrintWriter output =
                new PrintWriter(
                    link.getOutputStream(),true); //Step 2.

            //Set up stream for keyboard entry...
            Scanner userEntry = new Scanner(System.in);

            String message, response;
            do
            {
                System.out.print("Enter message: ");
                message = userEntry.nextLine();
                output.println(message);                //Step 3.
                response = input.nextLine();              //Step 3.
            }
        }
    }
}
```

```

        System.out.println("\nSERVER> "+response);
    }while (!message.equals("***CLOSE***"));
}
catch(IOException ioEx)
{
    ioEx.printStackTrace();
}

finally
{
    try
    {
        System.out.println(
            "\n* Closing connection... *");
        link.close();
    }
}
//Step 4.

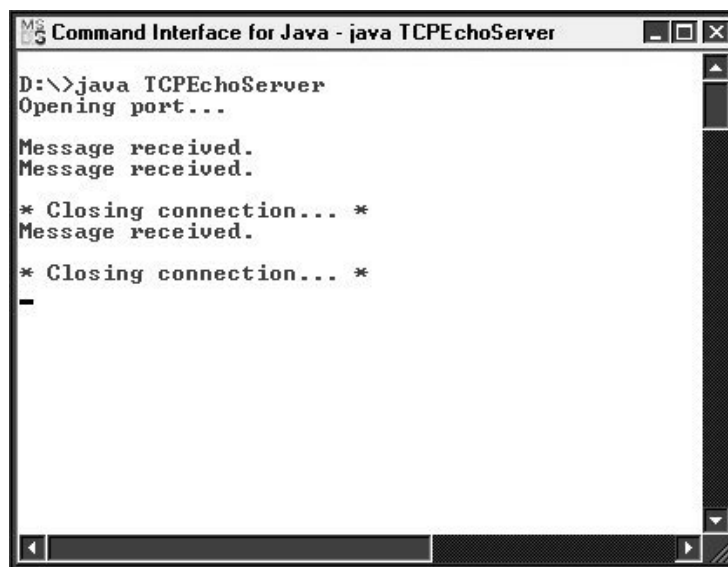
```

```

        catch(IOException ioEx)
        {
            System.out.println(
                "Unable to disconnect!");
            System.exit(1);
        }
    }
}

```

For the preceding client-server application to work, TCP/IP must be installed and working. How are you to know whether this is the case for your machine? Well, if there is a working Internet connection on your machine, then TCP/IP **is** running. In order to start the application, first open two command windows and then start the server running in one window and the client in the other. (Make sure that the server is running first, in order to avoid having the client program crash!) The example screenshots in Figures 2.3 and 2.4 show the dialogues between the server and two consecutive clients for this application. Note that, in order to stop the TCPEchoServer program, Ctrl-C has to be entered from the keyboard.

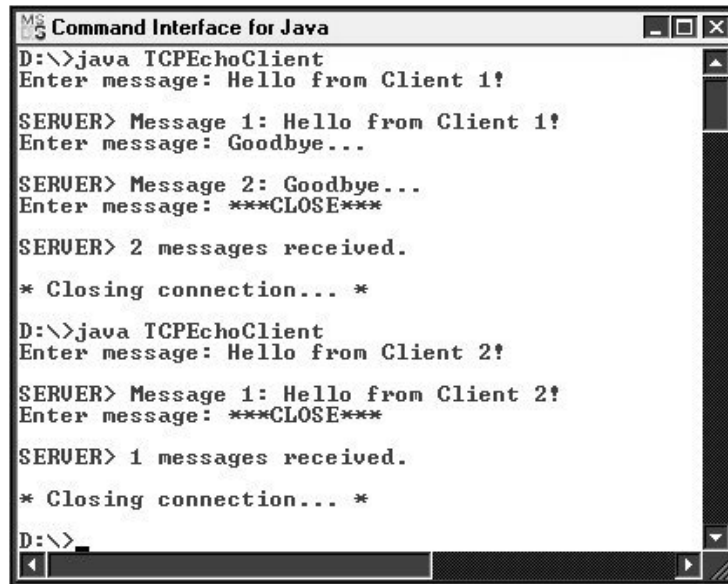


```

D:\>java TCPEchoServer
Opening port...
Message received.
Message received.
* Closing connection... *
Message received.
* Closing connection... *
-

```

Figure 2.3 Example output from the TCPEchoServer program.



```
MS-DOS Command Interface for Java
D:\>java TCPEchoClient
Enter message: Hello from Client 1!
SERVER> Message 1: Hello from Client 1!
Enter message: Goodbye...
SERVER> Message 2: Goodbye...
Enter message: ***CLOSE***
SERVER> 2 messages received.
* Closing connection... *
D:\>java TCPEchoClient
Enter message: Hello from Client 2!
SERVER> Message 1: Hello from Client 2!
Enter message: ***CLOSE***
SERVER> 1 messages received.
* Closing connection... *
D:\>
```

Figure 2.4 Example output from the TCPEchoClient program.

3. UDPEchoServer

//Server that echoes back client's messages.
//At end of dialogue, sends message indicating number of
//messages received. Uses datagrams.

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

public class UDPEchoServer
{
    private static final int PORT = 1234;
    private static DatagramSocket datagramSocket;
    private static DatagramPacket inPacket, outPacket;
    private static byte[] buffer;

    public static void main(String[] args)
    {
        System.out.println("Opening port...\n");
        try
        {
            datagramSocket =
                new DatagramSocket(PORT);           //Step 1.
        }
    }
}
```

```

        catch(SocketException sockEx)
        {
            System.out.println(
                "Unable to attach to port!");
            System.exit(1);
        }
        handleClient();
    }

    private static void handleClient()
    {
        try
        {
            String messageIn,messageOut;
            int numMessages = 0;

            do
            {
                buffer = new byte[256];           //Step 2.
                inPacket =
                    new DatagramPacket(
                        buffer, buffer.length);    //Step 3.
                datagramSocket.receive(inPacket); //Step 4.

                InetAddress clientAddress =
                    inPacket.getAddress(); //Step 5.
                int clientPort =
                    inPacket.getPort();      //Step 5.

                messageIn =
                    new String(inPacket.getData(),
                        0,inPacket.getLength()); //Step 6.

                System.out.println("Message received.");
                numMessages++;
                messageOut = "Message " + numMessages
                    + ": " + messageIn;

                outPacket =
                    new DatagramPacket(messageOut.getBytes(),
                        messageOut.length(),clientAddress,
                        clientPort);           //Step 7.
                datagramSocket.send(outPacket); //Step 8.
            }while (true);
        }
        catch(IOException ioEx)
        {
            ioEx.printStackTrace();
        }
    }

```



```

        finally //If exception thrown, close connection.
        {
            System.out.println(
                "\n* Closing connection... *");
            datagramSocket.close(); //Step 9.
        }
    }
}

```

Setting up the corresponding client requires the eight steps listed below:

1. Create a *DatagramSocket* object.

This is similar to the creation of a *DatagramSocket* object in the server program, but with the important difference that the constructor here requires no argument, since a default port (at the client end) will be used. For example:

```
DatagramSocket datagramSocket = new DatagramSocket();
```

2. Create the outgoing datagram.

This step is exactly as for step 7 of the server program. For example:

```

DatagramPacket outPacket =
    new DatagramPacket(message.getBytes(),
        message.length(), host, PORT);

```

8. Send the datagram message.

Just as for the server, this is achieved by calling method *send* of the *DatagramSocket* object, supplying our outgoing *DatagramPacket* object as an argument. For example:

```
datagramSocket.send(outPacket);
```

Steps 4-6 below are exactly the same as steps 2-4 of the server procedure.

4. Create a buffer for incoming datagrams.

For example:

```
byte[] buffer = new byte[256];
```

5. Create a *DatagramPacket* object for the incoming datagrams.

For example:

```

DatagramPacket inPacket =
    new DatagramPacket(buffer, buffer.length);

```

6. *Accept an incoming datagram.*

For example:

```
datagramSocket.receive(inPacket);
```

7. *Retrieve the data from the buffer.*

This is the same as step 6 in the server program. For example:

```
String response =  
    new String(inPacket.getData(), 0,  
               inPacket.getLength());
```

Steps 2-7 may then be repeated as many times as required.

8. *Close the DatagramSocket.*

This is the same as step 9 in the server program. For example:

```
datagramSocket.close();
```

As was the case in the server code, there is no checked exception generated by the above *close* method in the *finally* clause of the client program, so there will be no *try* block. In addition, since there is no inter-message connection maintained between client and server, there is no protocol required for closing down the dialogue. This means that we do not wish to send the final '***CLOSE***' string (though we shall continue to accept this from the user, since we need to know when to stop sending messages at the client end). The line of code (singular, this time) corresponding to each of the above steps will be indicated via an emboldened comment.

4. UDPEchoClient

```
import java.io.*;  
import java.net.*;  
import java.util.*;  
  
public class UDPEchoClient  
{  
    private static InetAddress host;  
    private static final int PORT = 1234;  
    private static DatagramSocket datagramSocket;  
    private static DatagramPacket inPacket, outPacket;  
    private static byte[] buffer;
```

```

public static void main(String[] args)
{
    try
    {
        host = InetAddress.getLocalHost();
    }
    catch(UnknownHostException uhEx)
    {
        System.out.println("Host ID not found!");
        System.exit(1);
    }
    accessServer();
}

private static void accessServer()
{
    try
    {
        //Step 1...
        datagramSocket = new DatagramSocket();

        //Set up stream for keyboard entry...
        Scanner userEntry = new Scanner(System.in);

        String message="", response="";
        do
        {
            System.out.print("Enter message: ");
            message = userEntry.nextLine();
            if (!message.equals("***CLOSE***"))
            {
                outPacket = new
                    DatagramPacket( message
                        .getBytes(),
                        message.length(),
                        host,PORT); //Step 2.

                //Step 3...
                datagramSocket.send(outPacket);
                buffer = new byte[256]; //Step 4.
                inPacket =
                    new DatagramPacket(
                        buffer, buffer.length); //Step 5.
                //Step 6...
                datagramSocket.receive(inPacket);
                response =
                    new String(inPacket.getData(),
                        0, inPacket.getLength()); //Step 7.
                System.out.println(
                    "\nSERVER> "+response);
            }
        }
    }
}

```

```

    }
    }while (!message.equals("***CLOSE***"));
}
catch(IOException ioEx)
{
    ioEx.printStackTrace();
}

finally
{
    System.out.println(
        "\n* Closing connection... *");
    datagramSocket.close();           //Step 8.
}
}
}

```

For the preceding application to work, UDP must be installed and working on the host machine. As for TCP/IP, if there is a working Internet connection on the machine, then UDP is running. Once again, in order to start the application, first open two command windows and then start the server running in one window and the client in the other. (Start the server *before* the client!) As before, the example screenshots in Figures 2.5 and 2.6 show the dialogues between the server and two clients. Observe the differences in output between this example and the corresponding TCP/IP example. (Note that the change at the client end is simply the rather subtle one of cumulative message-numbering.)

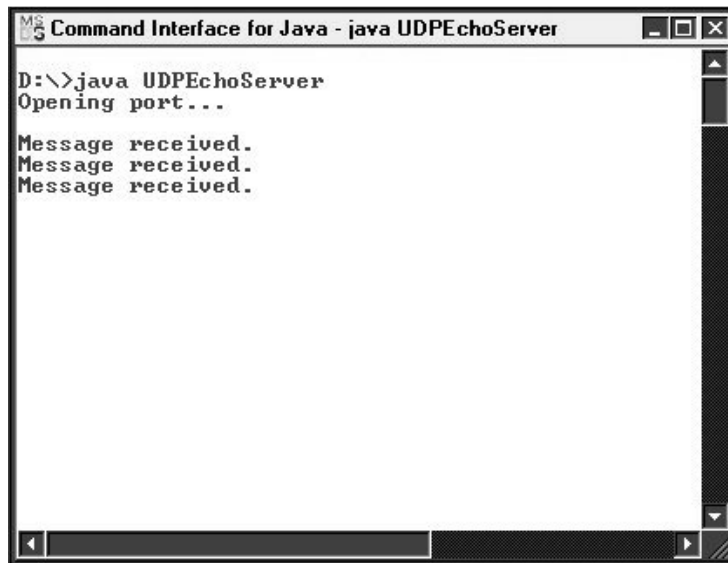


Figure 2.5 Example output from the UDPEchoServer program

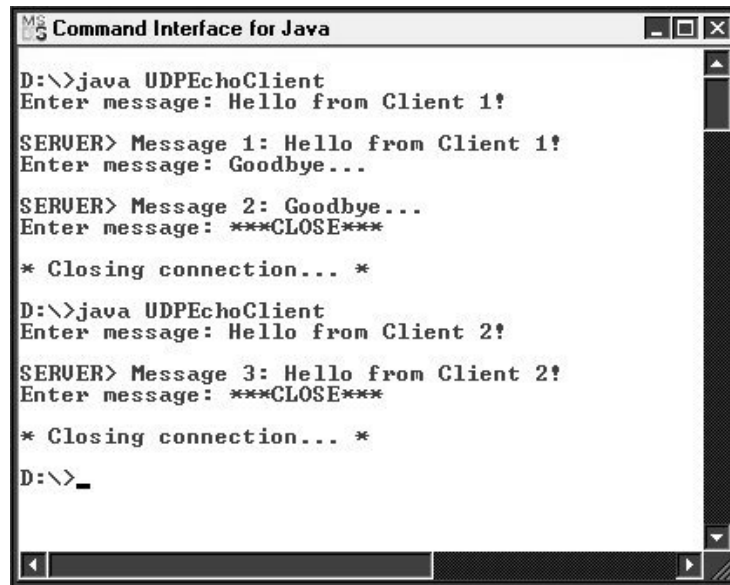


Figure 2.6 Example output from the UDPEchoClient program (with two clients connecting).

5. Echo Service

The following example involves building an echo service, which transmits any packet it receives straight back to the sender. The code uses no new networking classes or methods, but employs a special technique. It loops continuously to serve one client after another. Though only one UDP packet will be processed at a time, the delay between receiving a packet and dispatching it again is negligible, resulting in the illusion of concurrent processing.

```
import java.net.*;
import java.io.*;
public class EchoServer
{
    public static final int SERVICE_PORT = 7;
    public static final int BUFSIZE = 4096;
    private DatagramSocket socket;
    public EchoServer()
    {
        try
        {
            packets    socket = new
DatagramSocket( SERVICE_PORT );
            System.out.println ("Server active on port "
+ socket.getLocalPort() );
        }
    }
}
```

```

        catch (Exception e)
        {
            System.err.println ("Unable to bind port");
        }
    }
    public void serviceClients()
    {
        byte[] buffer = new byte[BUFSIZE];
        for (;;)
        {
            try
            {
                DatagramPacket packet = new DatagramPacket
                ( buffer, BUFSIZE );
                socket.receive(packet);
                System.out.println ("Packet received from " +
                packet.getAddress() + ":"
                + packet.getPort() + " of length " +
                packet.getLength() );
                socket.send(packet);
            }
            catch (IOException ioe)
            {
                System.err.println ("Error : " + ioe);
            }
        }
    }
    public static void main(String args[])
    {
        EchoServer server = new EchoServer();
        server.serviceClients();
    }
}

```

6. Echo Client

The following client can be used with the echo service and can easily be adapted to support other services. Repeated packets are sent to the echo service, and a timeout is caught to prevent the service from stalling if a packet becomes lost, and the client then waits to receive it. Remember that packet loss in an intranet environment is unlikely, but with slow network connections on the Internet it is quite possible.

```

import java.net.*;
import java.io.*;
public class EchoClient
{
    public static final int SERVICE_PORT = 7;
    public static final int BUFSIZE = 256;
    public static void main(String args[])
    {
        if (args.length != 1)
        {

```

```

        System.err.println ("Syntax - java
EchoClient hostname");
        return;
    }
    String hostname = args[0];
    InetAddress addr = null;
    try
    {
        addr = InetAddress.getByName(hostname);
    }
    catch (UnknownHostException uhe)
    {
        System.err.println ("Unable to resolve host");
        return;
    }
    try
    {
        DatagramSocket socket = new DatagramSocket();
        socket.setSoTimeout (2 * 1000);
        for (int i = 1 ; i <= 10; i++)
        {
            String message = "Packet number " + i ;
            char[] cArray = message.toCharArray();
            byte[] sendbuf = new byte[cArray.length];
            for (int offset = 0; offset <
cArray.length ; offset++)
            {
                sendbuf[offset] = (byte) cArray[offset];
            }
            DatagramPacket sendPacket = new
DatagramPacket(sendbuf,
cArray.length, addr, SERVICE_PORT);
            System.out.println ("Sending packet to " +
hostname);
            socket.send (sendPacket);
            System.out.print ("Waiting for packet.... ");
            byte[] recbuf = new byte[BUFSIZE];
            DatagramPacket receivePacket = new
DatagramPacket(recbuf,
BUFSIZE);
            boolean timeout = false;
            try
            {
                socket.receive (receivePacket);
            }
            catch (InterruptedException ioe)
            {
                timeout = true;
            }
            if (!timeout)
            {
                System.out.println ("packet received!");
            }
        }
    }
}

```

```

        System.out.println ("Details : " +
receivePacket.getAddress());
        ByteArrayInputStream bin = new
ByteArrayInputStream (receivePacket.getData(), 0,
receivePacket.getLength() );
        BufferedReader reader = new
BufferedReader ( new InputStreamReader ( bin ) );
        for (;;)
        {
            String line = reader.readLine();
            if (line == null) break;
            else System.out.println (line);
        }
    }
else
{
    System.out.println ("packet lost!");
}
try
{
    Thread.sleep(1000);
}
catch (InterruptedException ie)
{
}
}
catch (IOException ioe)
{
    System.err.println ("Socket error " + ioe);
}
}
}

```

7. DaytimeClient

Having discussed the functionality of the Socket class, we will now examine a complete TCP client. The client we'll look at here is a daytime client, which, as its name suggests, connects to a daytime server to read the current day and time. Establishing a socket connection and reading from it is a fairly simple process, requiring very little code. By default, the daytime service runs on port 13. Not every machine has a daytime server running, but a Unix server would be a good system to run the client against. If you do not have access to a Unix server, code for a TCP daytime server is given in [Section 8](#) –the client can be run against it.

```

import java.net.* ;
import java.io.*;
public class DaytimeClient
{
    public static final int SERVICE_PORT = 13;
    public static void main(String args[])

```



```

        {
            if (args.length != 1)
            {
                System.out.println ("Syntax - DaytimeClient
host");
                return;
            }
            String hostname = args[0];
            try
            {
                Socket daytime = new Socket (hostname,
SERVICE_PORT);
                System.out.println ("Connection established");
                daytime.setSoTimeout ( 2000 );
                BufferedReader reader = new BufferedReader
( new InputStreamReader
(daytime.getInputStream() ));
                System.out.println ("Results : " +
reader.readLine());
                daytime.close();
            }
            catch (IOException ioe)
            {
                System.err.println ("Error " + ioe);
            }
        }
    }
}

```

8. DaytimeServer

One of the most enjoyable parts of networking is writing a network server. Clients send requests and respond to data sent back, but the server performs most of the real work. This next example is of a daytime server (which you can test using the client described in [Section 7](#)).

```

import java.net.*;
import java.io.*;
public class DaytimeServer
{
    public static final int SERVICE_PORT = 13;
    public static void main(String args[])
    {
        try
        {
            ServerSocket server = new ServerSocket
(SERVICE_PORT);
            System.out.println ("Daytime service
started");
            for (;;)
            {
                Socket nextClient =
server.accept();

```

```

        System.out.println ("Received
request from " +
nextClient.getInetAddress() + ":" +
nextClient.getPort() );
        OutputStream out =
nextClient.getOutputStream();
        PrintStream pout = new PrintStream
(out);
        pout.print( new java.util.Date() );
        out.flush();
        out.close();
        nextClient.close();
    }
}
catch (BindException be)
{
    System.err.println ("Service already
running on port " + SERVICE_PORT );
}
catch (IOException ioe)
{
    System.err.println ("I/O error - " +
ioe);
}
}
}

```

9. Daytime

The following program uses the *Daytime* protocol to obtain the date and time from port 13 of user-specified host(s). It provides a text field for input of the host name by the user and a text area for output of the host's response. There are also two buttons, one that the user presses after entry of the host name and the other that closes down the program. The text area is 'wrapped' in a *JScrollPane*, to cater for long lines of output, while the buttons are laid out on a separate panel. The application frame itself will handle the processing of button presses, and so implements the *ActionListener* interface. The window-closing code (encapsulated in an anonymous *WindowAdapter* object) ensures that any socket that has been opened is closed before exit from the program.

```

import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
import java.net.*;
import java.io.*;
import java.util.*;

public class GetRemoteTime extends JFrame
    implements ActionListener

```

```

{
    private JTextField hostInput;
    private JTextArea display;
    private JButton timeButton;
    private JButton exitButton;
    private JPanel buttonPanel;
    private static Socket socket = null;

    public static void main(String[] args)
    {
        GetRemoteTime frame = new GetRemoteTime();
        frame.setSize(400,300);
        frame.setVisible(true);

        frame.addWindowListener(
            new WindowAdapter()
            {
                public void windowClosing(
                    WindowEvent event)
                {
                    //Check whether a socket is open...
                    if (socket != null)
                    {
                        try
                        {
                            socket.close();
                        }
                    }
                }
            }
        );
    }
}

```

```

        catch (IOException ioEx)
        {
            System.out.println(
                "\nUnable to close link!\n");
            System.exit(1);
        }
        System.exit(0);
    }
}

);
}

public GetRemoteTime()
{
    hostInput = new JTextField(20);
    add(hostInput, BorderLayout.NORTH);

    display = new JTextArea(10,15);

    //Following two lines ensure that word-wrapping
    //occurs within the JTextArea...
    display.setWrapStyleWord(true);
    display.setLineWrap(true);

    add(new JScrollPane(display),
        BorderLayout.CENTER);

    buttonPanel = new JPanel();

    timeButton = new JButton("Get date and time ");
    timeButton.addActionListener(this);
    buttonPanel.add(timeButton);

    exitButton = new JButton("Exit");
    exitButton.addActionListener(this);
    buttonPanel.add(exitButton);

    add(buttonPanel, BorderLayout.SOUTH);
}

public void actionPerformed(ActionEvent event)
{
    if (event.getSource() == exitButton)
        System.exit(0);

    String theTime;

    //Accept host name from the user...
    String host = hostInput.getText();
    final int DAYTIME_PORT = 13;

```



```

try
{
    //Create a Socket object to connect to the
    //specified host on the relevant port...
    socket = new Socket(host, DAYTIME_PORT);

    //Create an input stream for the above Socket
    //and add string-reading functionality...

        Scanner input =
            new Scanner(socket.getInputStream());

        //Accept the host's response via the
        //above stream...
        theTime = input.nextLine();

        //Add the host's response to the text in
        //the JTextArea...
        display.append("The date/time at " + host
            + " is " + theTime + "\n");
        hostInput.setText("");
    }
    catch (UnknownHostException uhEx)
    {
        display.append("No such host!\n");
        hostInput.setText("");
    }
    catch (IOException ioEx)
    {
        display.append(ioEx.toString() + "\n");
    }

    finally
    {
        try
        {
            if (socket!=null)
                socket.close(); //Close link to host.
        }
    }
}

```

```

        catch(IOException ioEx)
        {
            System.out.println(
                "Unable to disconnect!");
            System.exit(1);
        }
    }
}

```

If we run this program and enter *ivy.shu.ac.uk* as our host name in the client's GUI, the result will look something like that shown in Figure 2.7. Unfortunately, it is rather difficult nowadays to find a host that is running the *Daytime* protocol. Even if one does find such a host, it may be that the user's own firewall blocks the output from the remote server. If this is the case, then the user will be unaware of this until the connection times out which may take some time! The user is advised to terminate the program (with Ctrl-C) if the waiting time appears to be excessive. One possible way round this problem is to write one's own 'daytime server'...

To illustrate just how easy it is to provide a server that implements the *Daytime* protocol, example code for such a server is shown below. The program makes use of class *Date* from package *java.util* to create a *Date* object that will automatically hold the current day, date and time on the server's host machine. To output the date held in the *Date* object, we can simply use *println* on the object and its *toString* method will be executed implicitly (though we could specify *toString* explicitly, if we wished).

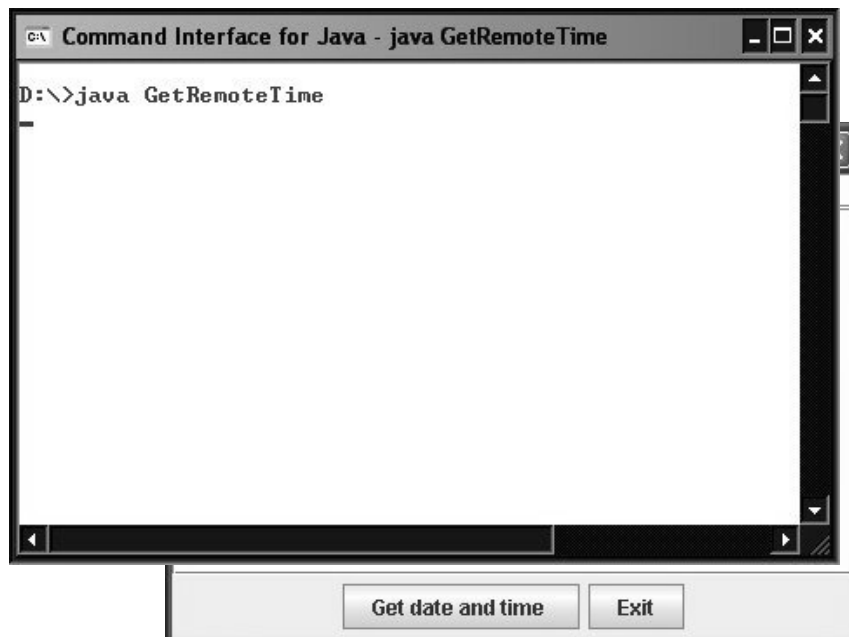


Figure 2.7 Example output from the *GetRemoteTime* program.

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

public class DaytimeServer
{
    public static void main(String[] args)
    {
        ServerSocket server;
        final int DAYTIME_PORT = 13;
        Socket socket;

        try
        {
            server = new ServerSocket(DAYTIME_PORT);
```



```

do
{
    socket = server.accept();
    PrintWriter output =
        new PrintWriter(
            socket.getOutputStream(),true);
    Date date = new Date();
    output.println(date);
    //Method toString executed in line above.

    socket.close();
}while (true);
}
catch (IOException ioEx)
{
    System.out.println(ioEx);
}
}
}

```

The server simply sends the date and time as a string and then closes the connection. If we run the client and server in separate command windows and enter *localhost* as our host name in the client's GUI, the result should look similar to that shown in Figure 2.7. Unfortunately, there is still a potential problem on some systems: since a low-numbered port (i.e., below 1024) is being used, the user may not have sufficient system rights to make use of the port. The solution in such circumstances is simple: change the port number (in both server and client) to a value above 1024. (E.g., change the value of *DAYTIME_PORT* from 13 to 1300.)

Now for an example that checks a range of ports on a specified host and reports on those ports that are providing a service. This works by the program trying to create a socket on each port number in turn. If a socket is created successfully, then there is an open port; otherwise, an *IOException* is thrown (and ignored by the program, which simply provides an empty catch clause). The program creates a text field for acceptance of the required URL(s) and sets this to an initial default value. It also provides a text area for the program's output and buttons for checking the ports and for exiting the program.

```

import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
import java.net.*;
import java.io.*;

public class PortScanner extends JFrame
    implements ActionListener

```

```

{
    private JLabel prompt;
    private JTextField hostInput;
    private JTextArea report;
    private JButton seekButton, exitButton;
    private JPanel hostPanel, buttonPanel;
    private static Socket socket = null;

    public static void main(String[] args)
    {
        PortScanner frame = new PortScanner();
        frame.setSize(400,300);
        frame.setVisible(true);

        frame.addWindowListener(
            new WindowAdapter()
            {
                public void windowClosing(
                    WindowEvent event)
                {
                    //Check whether a socket is open...
                    if (socket != null)
                    {
                        try
                        {
                            socket.close();
                        }
                    }
                }
            }
        );
    }
}

```

```

        catch (IOException ioEx)
        {
            System.out.println(
                "\nUnable to close link!\n");
            System.exit(1);
        }
        System.exit(0);
    }
}

);
}

public PortScanner()
{
    hostPanel = new JPanel();

    prompt = new JLabel("Host name: ");

    hostInput = new JTextField("ivy.shu.ac.uk", 25);
    hostPanel.add(prompt);
    hostPanel.add(hostInput);
    add(hostPanel, BorderLayout.NORTH);

    report = new JTextArea(10, 25);
    add(report, BorderLayout.CENTER);

    buttonPanel = new JPanel();

    seekButton = new JButton("Seek server ports ");
    seekButton.addActionListener(this);
    buttonPanel.add(seekButton);

    exitButton = new JButton("Exit");
    exitButton.addActionListener(this);
    buttonPanel.add(exitButton);

    add(buttonPanel, BorderLayout.SOUTH);
}

public void actionPerformed(ActionEvent event)
{
    if (event.getSource() == exitButton)
        System.exit(0);
    //Must have been the 'seek' button that was
    //pressed, so clear the output area of any
    //previous output...
    report.setText("");

    //Retrieve the URL from the input text field...
    String host = hostInput.getText();

    try
    {

```

```

                                ost);
report.append("IP address: "
//Convert the URL string into for (int i = 0; i < 25; i++)
an {
INetAddress
//object...
InetAddress
r
e
s
s

t
h
e
A
d
d
r
e
s
s

=

I
n
e
t
A
d
d
r
e
s
s
.
g
e
t
B
Y
N
a
m
e
(
h

```

```

try
{

//Attempt to establish a socket on
//port i...

        socket = new Socket(host, i);

        //If no IOException thrown, there must
        //be a service running on the port...
        report.append(
            "There is a server on port "
                + i + ".\n");

        socket.close();
    }
    catch (IOException ioEx)
    {}// No server on this port
}
}
catch (UnknownHostException uhEx)
{
    report.setText("Unknown host!");
}
}
}

```

When the above program was run for the default server (which is on the author's local network), the output from the GUI was as shown in Figure 2.8. Unfortunately, remote users' firewalls may block output from most of the ports for this default server (or any other remote server), causing the program to wait for each of these port accesses to time out. This is likely to take a **very** long time indeed! The reader is strongly advised to use a local server for the testing of this program (and to get clearance from your system administrator for port scanning, to be on the safe side). Even when running the program with a suitable local server, **be patient** when waiting for output, since this may take a minute or so, depending upon your system.

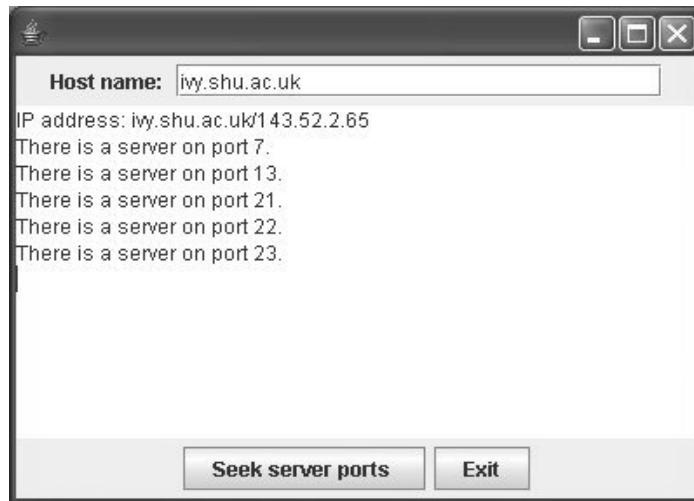


Figure 2.8 Example output from the PortScanner program.

10. SMTP Client

```
import java.io.*;
import java.net.*;
import java.util.*;
public class SMTPClient
{
    protected int port = 25;
    protected String hostname = "localhost";
    protected String from = "";
    protected String to = "";
    protected String subject = "";
    protected String body = "";
    protected Socket socket;
    protected BufferedReader br;
    protected PrintWriter pw;
    public SMTPClient() throws Exception
    {
        try
        {
            getInput();
            sendEmail();
        }
        catch (Exception e)
        {
            System.out.println ("Error sending message -
" + e);
        }
    }
    public static void main(String[] args) throws Exception
```

```

    {
        SMTPClient client = new SMTPClient();
    }
protected int readResponseCode() throws Exception
{
    String line = br.readLine();
    System.out.println("< "+line);
    line = line.substring(0,line.indexOf(" "));
    return Integer.parseInt(line);
}
protected void writeMsg(String msg) throws Exception
{
    pw.println(msg);
    pw.flush();
    System.out.println("> "+msg);
}
protected void closeConnection() throws Exception
{
    pw.flush();
    pw.close();
    br.close();
    socket.close();
}
protected void sendQuit() throws Exception
{
    System.out.println("Sending QUIT");
    writeMsg("QUIT");
    readResponseCode();
    System.out.println("Closing Connection");
    closeConnection();
}
protected void sendEmail() throws Exception
{
    System.out.println("Sending message now: Debug
below");
    System.out.println("-----
-----");
    System.out.println("Opening Socket");
    socket = new Socket(this.hostname,this.port);
    System.out.println("Creating Reader & Writer");
    br = new BufferedReader(new
InputStreamReader(socket.getInputStream()));
    pw = new PrintWriter(new
OutputStreamWriter(socket.getOutputStream()));
    System.out.println("Reading first line");    int
code = readResponseCode();
    if(code != 220)
    {
        socket.close();
        throw new Exception("Invalid SMTP Server");
    }
    System.out.println("Sending helo command");

```

```

        writeMsg("HELO
"+InetAddress.getLocalHost().getHostName());
        code = readResponseCode();
        if (code != 250)
        {
            sendQuit();
            throw new Exception("Invalid SMTP Server");
        }
        System.out.println("Sending mail from command");
        writeMsg("MAIL FROM:<"+this.from+">");
        code = readResponseCode();
        if (code != 250)
        {
            sendQuit();
            throw new Exception("Invalid from address");
        }
        System.out.println("Sending rcpt to command");
        writeMsg("RCPT TO:<"+this.to+">");
        code = readResponseCode();
        if (code != 250)
        {
            sendQuit();
            throw new Exception("Invalid to address");
        }
        System.out.println("Sending data command");
        writeMsg("DATA");
        code = readResponseCode();
        if (code != 354)
        {
            sendQuit();
            throw new Exception("Data entry not
accepted");
        }
        System.out.println("Sending message");
        writeMsg("Subject: "+this.subject);
        writeMsg("To: "+this.to);
        writeMsg("From: "+this.from);
        writeMsg(""); writeMsg(body);
        code = readResponseCode();
        sendQuit();
        if (code != 250)
            throw new Exception("Message may not have been
sent correctly");
        else
            System.out.println("Message sent");
    }
    protected void getInput() throws Exception
    {
        String data=null;
        BufferedReader br = new BufferedReader ( new
InputStreamReader(System.in));
        System.out.print("Please enter SMTP server
hostname: ");

```



```

        data = br.readLine();
        if (data == null || data.equals(""))
            hostname="localhost";
        else
            hostname=data;
        System.out.print("Please enter FROM email address:
");
        data = br.readLine();
        from = data;
        System.out.print("Please enter TO email address :");
        data = br.readLine();
        if(!(data == null || data.equals("")))
            to=data;
        System.out.print("Please enter subject: ");
        data = br.readLine();
        subject=data;
        System.out.println("Please enter plain-text message
('.' character on a blank line signals end of
message):");
        StringBuffer buffer = new StringBuffer();
        String line = br.readLine();
        while(line != null)
        {
            if(line.equalsIgnoreCase("."))
            {
                break;
            }
            buffer.append(line);
            buffer.append("\n");
            line = br.readLine();
        }
        buffer.append(".\n");
        body = buffer.toString();
    }
}

```

How SMTPClientDemo Works

As can be seen here, the Simple Mail Transfer Protocol is quite straightforward, consisting of a single connection to a mail server using a TCP socket, followed by a series of short protocol commands that specify the details of the e-mail to be sent, as described in RFC 2821. While many network applications will require multiple threads of execution, as a general rule simple clients such as this one do not. This example merely asks for input from the user and then sends the message. To simplify understanding of this code, the networking code has been separated from the nonnetworking code (which consists mainly of code for obtaining input from the user). The basic skeleton of the application is as follows:

```

public class SMTPClientDemo
{

```

```

public static void main(String[] args) throws Exception
{
    SMTPClientDemo client = new SMTPClientDemo ();
}
// Constructs a new instance of the SMTP Client
public SMTPClientDemo () throws Exception;
// Send an email message via SMTP, adhering to the
protocol
// known as RFC 2821
protected void sendEmail() throws Exception;
// Check the SMTP response code for an error message
protected int readResponseCode() throws Exception;
// Write a protocol message both to the network socket and
to
// the screen
protected void writeMsg(String msg) throws Exception;
// Close all readers, streams and sockets
protected void closeConnection() throws Exception;
// Send the QUIT protocol message, and terminate
connection
protected void sendQuit() throws Exception;
// Obtain input from the user
protected void getInput() throws Exception;
}

```

We'll cover the e-mail-specific code, as the remainder of the application is fairly straightforward Java coding. Our class stores the message and network details inside protected variables (rather than private ones, so readers can create subclasses that access these variables if required). These variables are:

- String hostname
- int port (set to 25, the default for SMTP)
- String from
- String to
- String subject
- String body
- java.net.Socket socket
- BufferedReader br
- PrintWriter pw

The hostname, port, from, to, subject, and body variables are fairly self-explanatory—they are required for locating the mail server to send an SMTP message, and for the e-mail addressing and content details. The socket is used to communicate with the remote TCP server that will relay our mail for us, and the reader/writer objects are used for reading and sending SMTP messages. When the application is run, the `main()` method is executed by the Java Virtual Machine. This in turn creates an instance of our `SMTPClientDemo` application, which requests input data from the user and attempts to send an e-mail message. The process has been fairly simple, so far. Now let's turn to the application protocol code. Contained inside the `sendEmail()` method is the heart of the program. This is

where we connect a local TCP socket to a remote SMTP server, and send across protocol requests. While it's certainly possible to group this process into one single method, for convenience and readability, some of the workload has been distributed across several helper methods that decode SMTP responses, send protocol messages, and terminate connections. The first thing the `sendEmail()` method does is open a socket connection to the mail server located at port n , where n is represented by the member variable `port`. The SMTP normally uses port 25, but if your network/ISP uses a nonstandard port number for this service, you could modify the default value of this to the required port number. If an error occurs, an exception is thrown and caught by our error handler; otherwise a successful connection to the server is established. Once connected, we obtain input streams and output streams for the socket, and connect these to readers and writers for convenience. According to the SMTP specification, upon connecting, the server will send a response code in greeting, so the application checks for a valid message with the aid of helper method `readResponseCode()`. This helps identify that we really are talking to an SMTP service, and not some other service using port 25. The code for determining the response code is wrapped inside a helper method, called `readResponseCode()`. It takes no parameters, and returns an int representing the code number. It reads a line of text from the buffered reader that communicates with the SMTP server, outputs the result to the screen for illustrative purposes, and then strips off everything after the first space. Finally, it converts the result to an integer and passes it back, giving us the SMTP response code.

```
// Check the SMTP response code for an error message
protected int readResponseCode() throws Exception
{
    String line = br.readLine();
    System.out.println("< "+line);
    line = line.substring(0,line.indexOf(" "));
    return Integer.parseInt(line);
}
```

When we need to send a message back to the SMTP server, we use the helper method `writeMsg(String)` which displays the message to the screen for illustrative purposes and then sends it (via our `PrintWriter`) to the server.

```
// Write a protocol message both to the network socket and
// to
// the screen
protected void writeMsg(String msg) throws Exception
{
    pw.println(msg);
    pw.flush();
    System.out.println("> "+msg);
}
```

Using these helper methods, the `sendEmail()` method can send and receive SMTP messages. It checks, upon connecting, that the server has sent an OK message, which is represented by 220.

```
int code = readResponseCode();
if(code != 220) {
    socket.close();
    throw new Exception("Invalid SMTP Server");
}
```

The next step in SMTP is to send an identification message, telling the SMTP server who we really are. We send the hostname identification command, "HELO." The format is "HELO" followed by a space and the local hostname. The correct response is 250, which signals OK.

```
writeMsg("HELO
"+InetAddress.getLocalHost().getHostName());
code = readResponseCode();
if(code != 250)
{
    sendQuit();
    throw new Exception("Invalid SMTP Server");
}
```

If the server accepts the identification, the client is free to send a message. Each message has certain key aspects—a "From" address, one or more "To" addresses, a subject line, and a message body (the actual text of the message). The SMTP only requires fields that deal directly with delivery of messages, though other fields can be placed in the message body. For example, it is commonly accepted to put the subject line as the first line, with "Subject:" in front of it. You can include as many other optional fields as you like, but remember that not every mail client will support these fields. Furthermore, you should also repeat the "To" and "From" fields in your message body. Setting the sender and recipient addressing information under SMTP is fairly straightforward. Two commands are used, the "MAIL FROM:" and "RCPT TO:"

```
writeMsg("MAIL FROM:<"+this.from+">");
// Check response from server
// .....
writeMsg("RCPT TO:<"+this.to+">");
// Check response from server
// .....
```

In this example, only one recipient is supported, but SMTP can handle multiple recipients. You should note that well-configured servers normally place a limit on the number of possible recipients, to prevent them from being used in spamming campaigns, and that some SMTP servers will reject messages if the sender is not part of their local or dial-up network. Once the e-mail addresses are set, we can send the data to the server. Our client sends a simple text message, unencumbered by attachments. The first step to send the data is to signal that we are ready

to send a message body, by issuing a "DATA" command. The valid response code for this is 354.

```
System.out.println("Sending data command");
writeMsg("DATA");
code = readResponseCode();
if(code != 354)
{
    sendQuit();
    throw new Exception("Data entry not accepted");
}
```

Now the client sends the message body. This must include relevant header fields, such as "To," "From," "Subject," and any other fields you may choose to add. Once the headers are complete, a blank line is sent, indicating that the message text will follow. After outputting the text, the client sends the message, which is then terminated by a period, then a carriage return/line-feed (which the user enters during data input). This tells the server that the message is complete.

```
writeMsg("Subject:"+this.subject);
writeMsg("");
writeMsg(body);
code = readResponseCode();
sendQuit();
if(code != 250)
    throw new Exception("Message may not have been sent
correctly");
else
    System.out.println("Message sent");
```

Finally, the "QUIT" command is sent, and the connection is terminated by invoking the `sendQuit()` method. Our transaction with the SMTP server is complete, and the message is on its way.

```
// Send the QUIT protocol message, and terminate
connection
protected void sendQuit() throws Exception
{
    System.out.println("Sending QUIT");
    writeMsg("QUIT");
    readResponseCode();
    System.out.println("Closing Connection");
    closeConnection();
}
```

Running SMTPClientDemo

After compiling, the application can be run by typing:

```
java SMTPClientDemo
```

The application will request the following information:

- The name of a valid SMTP server (such as that used by your e-mail program)
- The "From" address of the sender (e.g., your e-mail address)

- The "To" address of the recipient (e.g., your e-mail address, so you know that it was delivered)
- The "Subject" of the message
- The message contents

11. Pop3Client

```
import java.io.*;
import java.net.*;
import java.util.*;
class Pop3Client
{
    protected int port = 110;
    protected String hostname = "localhost";
    protected String username = "";
    protected String password = "";
    protected Socket socket;
    protected BufferedReader br;
    protected PrintWriter pw;
    public Pop3Client() throws Exception
    {
        try
        {
            getInput();
            displayEmails();
        }
        catch(Exception e)
        {
            System.err.println("Error occurred - details
follow");
            e.printStackTrace();
            System.out.println(e.getMessage());
        }
    }
    protected boolean responseIsOk() throws Exception
    {
        String line = br.readLine();
        System.out.println("< "+line);
        return line.toUpperCase().startsWith("+OK");
    }
    protected String readLine(boolean debug) throws
Exception
    {
        String line = br.readLine();
        if (debug)
            System.out.println("< "+line);
        else
            System.out.println(line);
        return line;
    }
    protected void writeMsg(String msg) throws Exception
    {
        pw.println(msg);
        pw.flush();
    }
}
```

```

        System.out.println("> "+msg);
    }
    protected void closeConnection() throws Exception
    {
        pw.flush();
        pw.close();
        br.close();
        socket.close();
    }
    protected void sendQuit() throws Exception
    {
        System.out.println("Sending QUIT");
        writeMsg("QUIT");
        readLine(true);
        System.out.println("Closing Connection");
        closeConnection();
    }
    protected void displayEmails() throws Exception
    {
        BufferedReader userInput = new
BufferedReader( new InputStreamReader (System.in) );
        System.out.println("Displaying mailbox with
protocol commands " and responses below");
        System.out.println("-----
-----+ "-----");
        System.out.println("Opening Socket");
        socket = new Socket(this.hostname, this.port);
        br = new BufferedReader(new
InputStreamReader(socket.getInputStream()));
        pw = new PrintWriter(new
OutputStreamWriter(socket.getOutputStream()));
        if(!responseIsOk())
        {
            socket.close();
            throw new Exception("Invalid POP3
Server");
        }
        System.out.println("Sending username");
        writeMsg("USER "+this.username);
        if(!responseIsOk())
        {
            sendQuit();
            throw new Exception("Invalid username");
        }
        System.out.println("Sending password");
        writeMsg("PASS "+this.password);
        if(!responseIsOk())
        {
            sendQuit();
            throw new Exception("Invalid password");
        }

        System.out.println("Checking mail");
    }

```

```

        writeMsg("STAT" );
        String line = readLine(true);
        StringTokenizer tokens = new
StringTokenizer(line, " ");
        tokens.nextToken();
        int messages =
Integer.parseInt(tokens.nextToken());
        int maxsize =
Integer.parseInt(tokens.nextToken());
        if (messages == 0)
        {
            System.out.println ("There are no
messages.");
            sendQuit();
            return;
        }
        System.out.println ("There are " + messages + "
messages.");
        System.out.println("Press enter to continue.");
        userInput.readLine();
        for(int i = 1; i <= messages ; i++)
        {
            System.out.println("Retrieving message
number "+i);
            writeMsg("RETR "+i);
            System.out.println("-----")
            line = readLine(false);
            while(line != null && !line.equals("."))
            {
                line = readLine(false);
            }
            System.out.println("-----");
            System.out.println("Press enter to
continue." + "To stop, type Q then enter");
            String response = userInput.readLine();
            if (response.toUpperCase().startsWith("Q"))
break;
        }
        sendQuit();
    }
    public static void main(String[] args) throws Exception
    {
        Pop3Client client = new Pop3Client();
    }
    protected void getInput() throws Exception
    {
        String data=null;
        BufferedReader br = new BufferedReader (new
InputStreamReader(System.in));
        System.out.print("Please enter POP3 server
hostname:");
        data = br.readLine();

```



```

        if(data == null || data.equals(""))
            hostname= "localhost";
        else
            hostname=data;
        System.out.print("Please enter mailbox
username:");
        data = br.readLine();
        if(!(data == null || data.equals(""))))
            username=data;
        System.out.print("Please enter mailbox
password:");
        data = br.readLine();
        if(!(data == null || data.equals(""))))
            password=data;
    }
}

```

How Pop3ClientDemo Works

The application is structured similarly to the SMTP client, in that the `main()` method constructs a new instance of the class, which then performs all of the work. In fact, the chief difference is a call to the `displayEmails` method instead of to the `sendEmail` method. In addition, there are several helper methods that assist in conducting communication via POP3. Like the SMTP application, several important variables are requested from the user. To retrieve email messages, you need to know the hostname of the mail server, the username of the account, and the password for accessing it. These details are obtained by the `getInput()` method of the application, which uses simple text I/O to request details from the user. These are then stored in member variables for later access. We also store the port number of the server (which is fixed to the default port of 110). If, for example, you needed to support nonstandard POP servers, you could modify this value.

```

public class Pop3ClientDemo
{
    protected int port = 110;
    protected String hostname = "localhost";
    protected String username = "";
    protected String password = "";
    // .....
}

```

The most important part of the application is the Post Office Protocol implementation. This is where we get to write real networking code. To make things clearer and more efficient, some of the protocol code is split into helper methods, which perform tasks such as processing a POP response to see that no error code was issued, and reading/writing protocol commands. The main work is one, however, in the `displayEmails()` method. We start by creating a network socket to the POP server, using the

Socket class. The next step is to obtain readers and writers connected to the socket stream, so that we can communicate with the server.

```
// Open a connection to POP3 server
System.out.println("Opening Socket");
socket = new Socket(this.hostname, this.port);
br = new BufferedReader(new
InputStreamReader(socket.getInputStream()));
pw = new PrintWriter(new
OutputStreamWriter(socket.getOutputStream()));
```

Upon establishing a connection, the server sends a POP response indicating that the server is ready for commands. If the client does not receive a valid POP response, either the server is malfunctioning or a non-POP server is operating on that port. For this reason, the client must always check the response code after opening a TCP connection. We use the `responseOk()` helper method to determine this, which returns a boolean value. Note the negation of the "if" statement—we close the connection only if the response is not okay (indicated by the "!" operator).

```
// If response from server is not okay
if(!responseIsOk())
{
socket.close();
throw new Exception("Invalid POP3 Server");
}
```

Now our application is ready to send POP commands. The first command that is sent is "USER," which identifies the user account that the client is trying to access. We send the command, along with the username, by using the helper method `writeMsg(String)`. This also outputs the command to the text console, so you can see how the protocol is working. We do the same then with the "PASS" command, which is used to authenticate the identity of the user.

```
// Login by sending USER and PASS commands
System.out.println("Sending username");
writeMsg("USER "+this.username);
if(!responseIsOk())
{
sendQuit();
throw new Exception("Invalid username");
}
writeMsg("PASS "+this.password);
if(!responseIsOk())
{
sendQuit();
throw new Exception("Invalid password");
}
```

At this point, the application will be ready to send commands, unless user authentication failed, in which case the application will have terminated. Now we can request the number of mail messages available from the server.

This is achieved by sending the "STAT" command, which returns the number of messages. We parse the response, looking for the message count, and then convert it to an int value.

```
writeMsg("STAT");
// ... and parse for number of messages
String line = readLine(true);
StringTokenizer tokens = new StringTokenizer(line, " ");
tokens.nextToken();
int messages = Integer.parseInt(tokens.nextToken());
int maxsize = Integer.parseInt(tokens.nextToken());

    Armed with this information, we can determine if the
    mailbox is empty or if there are messages to be retrieved.
    If so, the application simply loops and displays the
    message contents. During each loop, the user is presented
    with the option of quitting in case the message count is
    too high.

System.out.println ("There are " + messages + "
messages.");
System.out.println("Press enter to continue.");
userinput.readLine();
for(int i = 1; i <= messages ; i++)
{
    System.out.println("Retrieving message number "+i);
    writeMsg("RETR "+i);
    System.out.println("-----");
    line = readLine(false);
    while(line != null && !line.equals("."))
    {
        line = readLine(false);
    }
    System.out.println("-----");
    System.out.println("Press enter to continue. To stop, type
    Q then
    enter");
    String response = userinput.readLine();
    if (response.toUpperCase().startsWith("Q"))
        break;
    }
sendQuit();
```

Finally, we call the `sendQuit()` method, which shuts down the server connection. The application then quits and the task of reading mail using the Post Office Protocol is complete. As you can see, reading e-mail is a fairly simple task, perhaps even easier than sending it.

Running Pop3ClientDemo

Running the application, too, is quite manageable. After compiling, simply type:

```
java Pop3ClientDemo
```

You will be prompted for the hostname of your mail server (which, if you're not sure of, can be obtained by looking at your e-mail client's settings), the username,

and then the password. For example, a user might type the following:

```
java PopClientDemo
Please enter POP3 server hostname: myserver.myisp.com
Please enter mailbox username: johndoe
Please enter mailbox password: javaduke
```

12. Multithreading

This is another echo server implementation, but one that uses multithreading to return messages to multiple clients. It makes use of a support class called *ClientHandler* that extends class *Thread*. Whenever a new client makes connection, a *ClientHandler* thread is created to handle all subsequent communication with that particular client. When the *ClientHandler* thread is created, its constructor is supplied with a reference to the relevant socket.

Here's the code for the server...

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

public class MultiEchoServer
{
    private static ServerSocket serverSocket;
    private static final int PORT = 1234;

    public static void main(String[] args)
        throws IOException
    {
        try
        {
            serverS serverSocket = new ServerSocket(PORT);
        }
    }
}
```

```

        catch (IOException ioEx)
        {
            System.out.println("\nUnable to set up port!");
            System.exit(1);
        }

        do
        {
            //Wait for client...
            Socket client = serverSocket.accept();

            System.out.println("\nNew client accepted.\n");

            //Create a thread to handle communication with
            //this client and pass the constructor for this
            //thread a reference to the relevant socket...
            ClientHandler handler =
                new ClientHandler(client);
            handler.start(); //As usual, method calls run.
        }while (true);
    }
}

class ClientHandler extends Thread
{
    private Socket client;
    private Scanner input;
    private PrintWriter output;

    public ClientHandler(Socket socket)
    {
        //Set up reference to associated socket...
        client = socket;

        try
        {
            input = new Scanner(client.getInputStream());
            output = new PrintWriter(
                client.getOutputStream(), true);
        }
        catch (IOException ioEx)
        {
            ioEx.printStackTrace();
        }
    }

    public void run()
    {
        String received;

        do
        {

```

```

        //Accept message from client on
        //the socket's input stream...
        received = input.nextLine();

        //Echo message back to client on
        //the socket's output stream...
        output.println("ECHO: " + received);

        //Repeat above until 'QUIT' sent by client...
    }while (!received.equals("QUIT"));

    try
    {
        if (client!=null)
        {
            System.out.println(
                "Closing down connection...");
            client.close();
        }
    }

    catch(IOException ioEx)
    {
        System.out.println("Unable to disconnect!");
    }
}

```

The code required for the client program is exactly that which was employed in the *TCPEchoClient* program from the last chapter. However, since (i) there was only a modest amount of code in the *run* method for that program, (ii) we should avoid confusion with the *run* method of the *Thread* class and (iii) it'll make a change (!) without being harmful, all the executable code has been placed inside *main* in the *MultiEchoClient* program below.

```

import java.io.*;
import java.net.*;
import java.util.*;

public class MultiEchoClient
{
    private static InetAddress host;
    private static final int PORT = 1234;

    public static void main(String[] args)
    {
        try
        {

```

```

        host = InetAddress.getLocalHost();
    }
    catch(UnknownHostException uhEx)
    {
        System.out.println("\nHost ID not found!\n");
        System.exit(1);
    }
    sendMessages();
}

private static void sendMessages()
{
    Socket socket = null;

    try
    {
        socket = new Socket(host,PORT);

        Scanner networkInput =
            new Scanner(socket.getInputStream());
        PrintWriter networkOutput =
            new PrintWriter(
                socket.getOutputStream(),true);

        //Set up stream for keyboard entry...
        Scanner userEntry = new Scanner(System.in);

        String message, response;
        do
        {
            System.out.print(
                "Enter message ('QUIT' to exit): ");
            message = userEntry.nextLine();

            //Send message to server on the
            //socket's output stream...

            //Accept response from server on the
            //socket's input stream...
            networkOutput.println(message);
            response = networkInput.nextLine();

            //Display server's response to user...
            System.out.println(
                "\nSERVER> " + response);
        }while (!message.equals("QUIT"));
    }
    catch(IOException ioEx)

```

```
{
    ioEx.printStackTrace();
}

finally
{
    try
    {
        System.out.println(
            "\nClosing connection...");
        socket.close();
    }
}
```



```
        catch(IOException ioEx)
        {
            System.out.println(
                "Unable to disconnect!");
            System.exit(1);
        }
    }
}
```

If you wish to test the above application, you should start the server running in one command window and then start up two clients in separate command windows.
