7.1 Creating Client and Server Sockets (UDP and TCP sockets)
7.2 Reading from and writing to a Socket
7.3 Writing the Server Side of a Socket

Socekt *: If the IP address is like an office building main phone number, a socket is like the extension numbers for offices*. So the combination of IP and Port called the socket, uniquely identify an "office" (server process). You will see unique identifiers like 192.168.2.100:80 where 80 is the port. Just like in an office, it is possible no process is listening at a port. That is, there is no server waiting for requests at that port. Ports run from 1..65535. 1..1024 require root privileges to use and ports 1..255 are reserved for common processes like: 80: HTTP, 110: POP, 25: SMTP, 22: SSH
We will see that we can write these programs without any knowledge of the technologies under the hood (which include operating system resources, routing between networks, address lookup, physical transmission media, etc.). Each of these applications use the *client-server* model, which is roughly

1. One program, called the *server* blocks waiting for a client to connect to it
2. A client connects
3. The server and the client exchange information until they're done
4. The client and the server both close their connection

The only pieces of background information you need are:

- Hosts have *ports*, numbered from 0-65535. Servers listen on a port. Some port numbers are reserved so you can't use them when you write your own server.
- Multiple clients can be communicating with a server on a given port. Each client connection is assigned a separate *socket* on that port.
- Client applications get a port and a address on the client machine when they connect successfully with a server.

Goal : Socket programming to build any networked application e.g. www(IE, Firefox), FTP (WinSCP)
The four applications are

- A trivial date server and client, illustrating simple one-way communication. The server sends data to the client only.
- A capitalize server and client, illustrating two-way communication. Since the dialog between the client and server can comprise an unbounded number of messages back and forth, the server is *threaded* to service multiple clients efficiently.
- A two-player tic tac toe game, illustrating a server that needs to keep track of the state of a game, and inform each client of it, so they can each update their own displays.
- A multi-user chat application, in which a server must broadcast messages to all of its clients.

e.g. Telephone Dial Analogue

- Socket() : end point for communication
- Bind() : Assign a unique telephone number
- Listen() : Wait for caller
- Connect() : Dial a number
- Accept() : Receive a call
- Send()/ Receive(), Read()/ Write(): Talk or Message exchange
- Close() : Hang up.

The set of functions comprising the API include primarily:

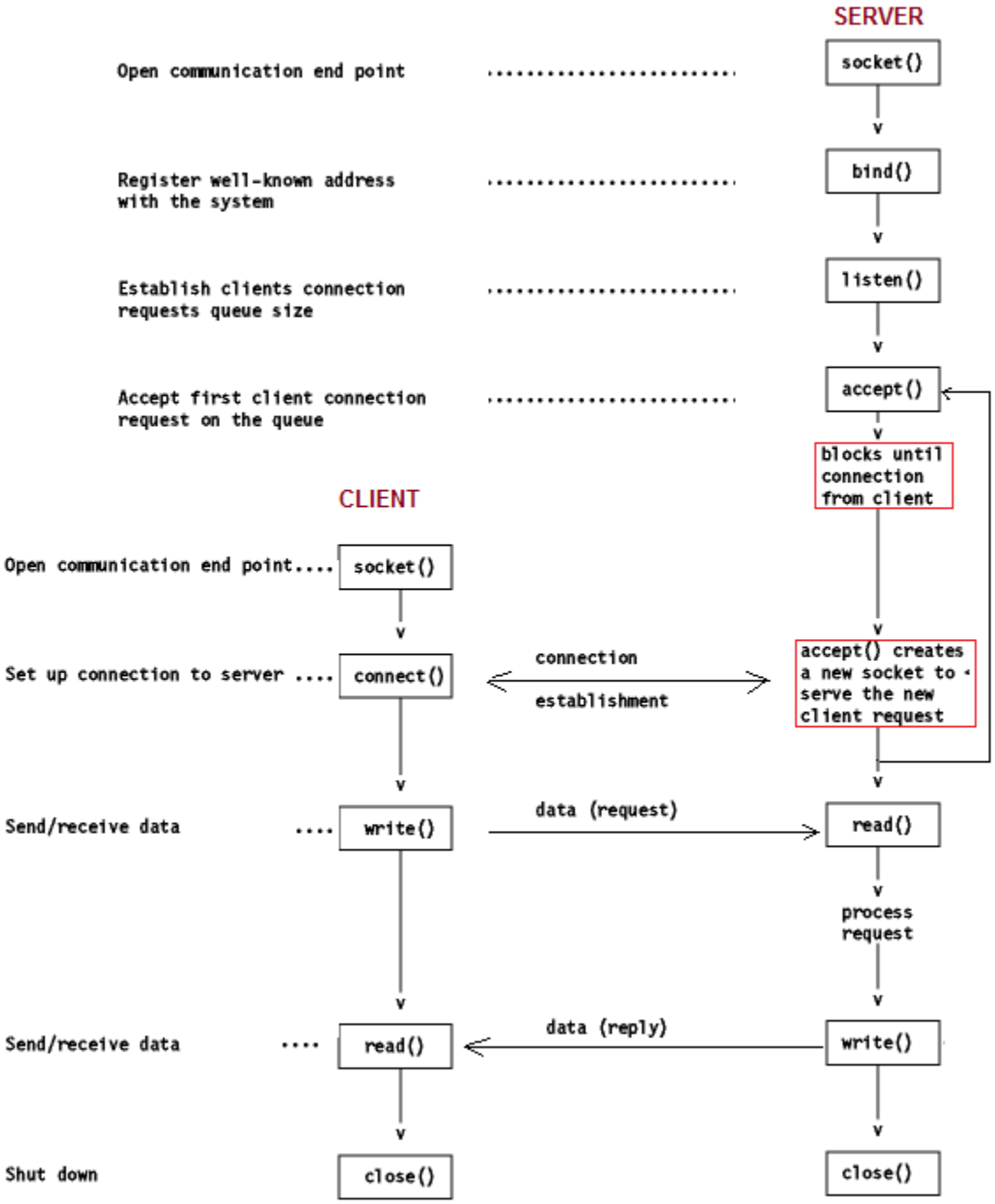
- socket( )
- bind( )
- connect( )
- listen( ) and accept( )
- read( ), recv( ), recvfrom( ), or recvmsg( )
- write( ), send( ), sendto( ), or sendmsg( )
- close( )

bind( ) is used particularly by server programs, and connect( ) by client programs. The others are used by both.

- bind( ) applies a "name" to an existing socket, so that it can be referred to. The essence of this "name" or identifier, is an IP-address-and-port-number pair.
- connect( ) searches out an already named (or "already bind'ed") program out on the network, by name. So calls to bind( ) supply, as parameters, information that tells which socket and what name. And calls to connect( ) supply a local socket, and the name of a remote socket to search for, locate, and connect to.

Once a server program has created a socket and named it with bind( ) giving it an IP address and port number, should any program anywhere on the network give that same name to the connect( ) function, that program will find our server program and they will link up. The server program uses the accept( ) function to react to the client's connect( ). So the accept( ) must be called before the connect(

) is issued. Once all this connecting and accepting is done, both sides can freely use read/write or recv/send to get stuff shipped to each other.



7.1 Creating Client and Server Sockets (UDP and TCP sockets)

Client Socket Basics

A socket establishes the connecting endpoints between two hosts. The *Socket* class provided by Java is used for both clients and servers. The basic operations area is as follows:

- Connect to remote host.
- Send and receive data.
- Close a connection.
- Bind to a port.
- Listen to incoming data.
- Accept remote connections on the bounded port.

The last three operations are specific to servers only; they are implemented by the *ServerSocket* class. The client program work flow occurs in the following manner:

1. Create a new socket object.
2. Attempt to connect to the remote host
3. Once connection has succeeded, the local and remote hosts get hold of the input and output streams and can work in full duplex mode. The data received and sent can mean different things, depending on the protocol used (data sent/received from an FTP server can be different from an HTTP server). Generally, there is some kind of agreement established followed by data transmission.
4. Sockets must be closed at both ends after transmission is completed. Some protocols, such as HTTP, make sure that the connection is closed upon each request service. FTP, on the other hands, allows multiple requests to process before closing the connection.

Server Socket Basics

The *ServerSocket* class provides the niche to write everything about servers in Java. The main job of a server socket is to wait for incoming calls and respond accordingly. *ServerSocket* runs on the server bounded by a port and listens to incoming TCP connections. When a client *Socket* attempts to connect to that port, the server wakes up to negotiate the connection by opening a *Socket* between two hosts. This *Socket* object is used to send data to the clients. The work flow of the server program can be defined as follows:

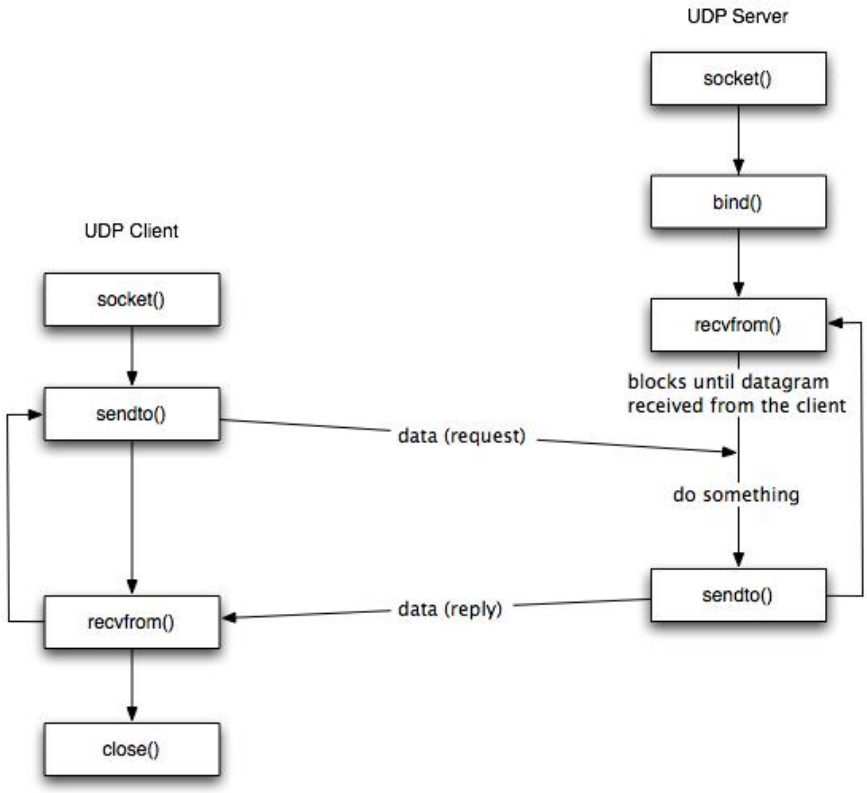
1. Create server socket on a particular port.
2. Listen to any attempt of connection to that port.
3. If a connection attempt succeeds, get the host of stream objects from the *Socket* and communicate with the client.
4. The communication, however, is established according to the agreed protocol.
5. Close the connection.
6. Wait further for any more communication attempts (return to Step 2).

Summarize saying that since TCP requires a connection, it needs the following APIs on server and client
server : socket(), bind(), **listen(), accept(),** recv(), send()
client : socket(), **connect(),** send(), recv()

And since UDP is a connectionless protocol, it does not require connect() on the client and listen() and accept() on the server
server : socket(), bind(), recvfrom(), sendto()
client : socket(), sendto(), recvfrom()

UDP/ Datagram communication:

The datagram communication protocol, known as UDP (user datagram protocol), is a connectionless protocol, meaning that each time you send datagrams, you also need to send the local socket descriptor and the receiving socket's address. As you can tell, additional data must be sent each time a communication is made. Applications built around UDP are DNS, NFS, SNMP and for example, some Skype services and streaming media.



As shown in the Figure, the steps of establishing a UDP socket communication on the client side are as follows:

- socket() : Create a socket
- recvfrom() and sendto() : Send and receive data by means of the.

The steps of establishing a UDP socket communication on the server side are as follows:

- socket() : Create a socket ;
- bind() : Bind the socket to an address ;
- recvfrom() and sendto().: Send and receive data

TCP/ Stream communication:

The stream communication protocol is known as TCP (transfer control protocol). Unlike UDP, TCP is a connection-oriented protocol. In order to do communication over the TCP protocol, a connection must first be established between the pair of sockets. While one of the sockets listens for a connection request (server), the other asks for a connection (client). Once two sockets have been connected, they can be used to transmit data in both (or either one of the) directions.

Now, you might ask what protocol you should use -- UDP or TCP? This depends on the client/server application you are writing. The following discussion shows the differences between the UDP and TCP protocols; this might help you decide which protocol you should use.

In UDP, as you have read above, every time you send a datagram, you have to send the local descriptor and the socket address of the receiving socket along with it. Since TCP is a connection-oriented protocol, on the other hand, a connection must be established before communications between the pair of sockets start. So there is a connection setup time in TCP.

In UDP, there is a size limit of 64 kilobytes on datagrams you can send to a specified location, while in TCP there is no limit. Once a connection is established, the pair of sockets behaves like streams: All available data are read immediately in the same order in which they are received.

UDP is an unreliable protocol -- there is no guarantee that the datagrams you have sent will be received in the same order by the receiving socket. On the other hand, TCP is a reliable protocol; it is guaranteed that the packets you send will be received in the order in which they were sent.

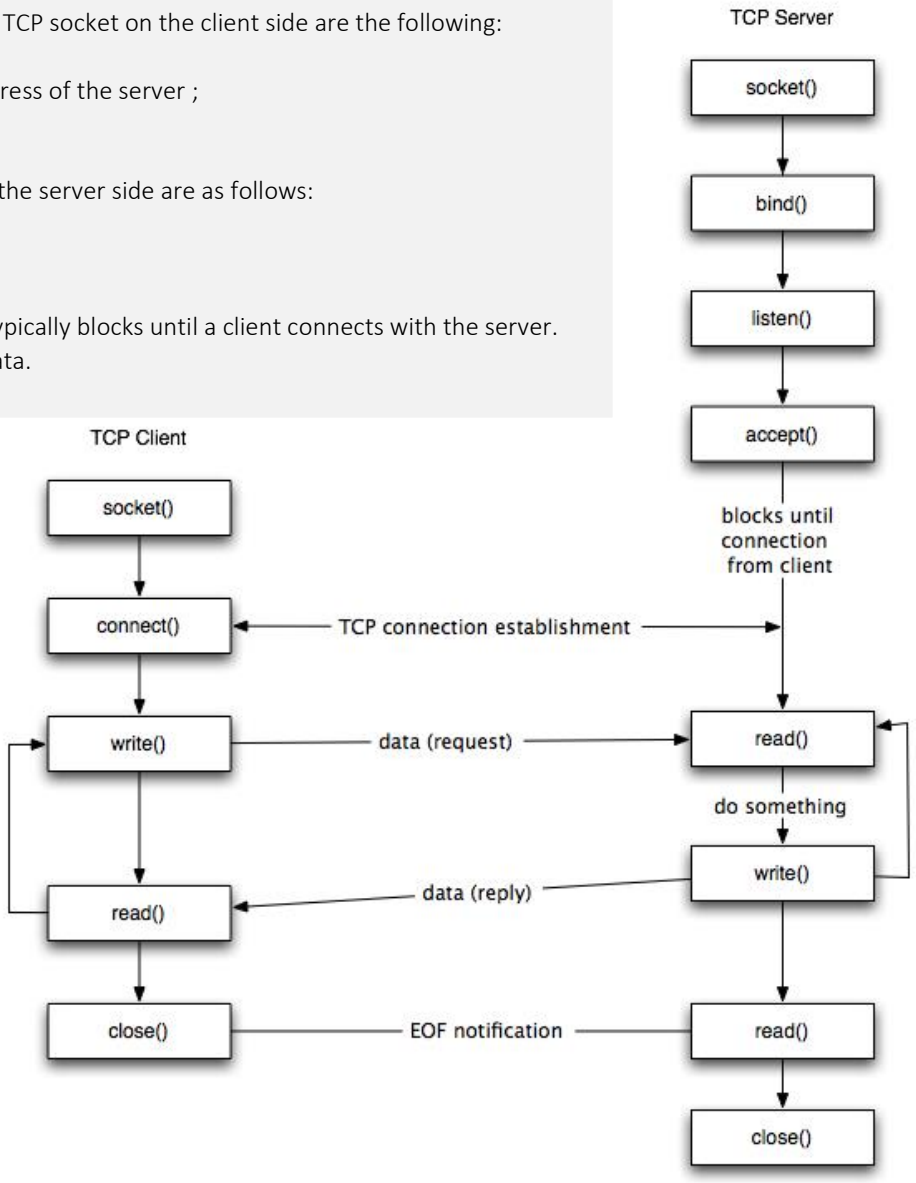
In short, TCP is useful for implementing network services -- such as remote login (rlogin, telnet) and file transfer (FTP) -- which require data of indefinite length to be transferred. UDP is less complex and incurs fewer overheads. It is often used in implementing client/server applications in distributed systems built over local area networks

As shown in the figure, the steps for establishing a TCP socket on the client side are the following:

- socket() : Create a socket ;
- connect() : Connect the socket to the address of the server ;
- read() and write() : Send and receive data
- close() : Close the connection

The steps involved in establishing a TCP socket on the server side are as follows:

- socket() : Create a socket;
- bind() : Bind the socket to an address;
- listen() : Listen for connections;
- accept() : Accept a connection. This call typically blocks until a client connects with the server.
- send() and receive().: Send and receive data.
- close() : Close the connection.



7.2 Reading from and writing to a Socket

Let's look at a simple example that illustrates how a program can establish a connection to a server program using the Socket class and then, how the client can send data to and receive data from the server through the socket.

This client program is straightforward and simple because the echo server implements a simple protocol. The client sends text to the server, and the server echoes it back. When your client programs are talking to a more complicated server such as an HTTP server, your client program will also be more complicated. However, the basics are much the same as they are in this program:

1. Open a socket.
2. Open an input stream and output stream to the socket.
3. Read from and write to the stream according to the server's protocol.
4. Close the streams.
5. Close the socket.

Only step 3 differs from client to client, depending on the server. The other steps remain largely the same.

7.3 Writing the Server Side of a Socket

This section shows you how to write a server and the client that goes with it. The server in the client/server pair serves up Knock Knock jokes. Knock Knock jokes are favored by children and are usually vehicles for bad puns. They go like this:

Server: "Knock knock!"
Client: "Who's there?"
Server: "Dexter."
Client: "Dexter who?"
Server: "Dexter halls with boughs of holly."
Client: "Groan."

The example consists of two independently running Java programs: the client program and the server program. The client program is implemented by a single class, KnockKnockClient, and is very similar to the EchoClient example from the previous section. The server program is implemented by two classes: KnockKnockServer and KnockKnockProtocol. KnockKnockServer, which is similar to EchoServer, contains the main method for the server program and performs the work of listening to the port, establishing connections, and reading from and writing to the socket. The class KnockKnockProtocol serves up the jokes. It keeps track of the current joke, the current state (sent knock knock, sent clue, and so on), and returns the various text pieces of the joke depending on the current state. This object implements the protocol—the language that the client and server have agreed to use to communicate.

The example program implements a client, EchoClient, that connects to an echo server. The echo server receives data from its client and echoes it back. The example EchoServer implements an echo server. (Alternatively, the client can connect to any host that supports the Echo Protocol.)

The EchoClient example creates a socket, thereby getting a connection to the echo server. It reads input from the user on the standard input stream, and then forwards that text to the echo server by writing the text to the socket. The server echoes the input back through the socket to the client. The client program reads and displays the data passed back to it from the server.

Note that the EchoClient example both writes to and reads from its socket, thereby sending data to and receiving data from the echo server.

Let's walk through the program and investigate the interesting parts. The following statements in the try-with-resources statement in the EchoClient example are critical. These lines establish the socket connection between the client and the server and open a PrintWriter and a BufferedReader on the socket:

```
String hostName = args[0];
int portNumber = Integer.parseInt(args[1]);

try (
    Socket echoSocket = new Socket(hostName, portNumber);
    PrintWriter out =
        new PrintWriter(echoSocket.getOutputStream(), true);
    BufferedReader in =
        new BufferedReader(
            new InputStreamReader(echoSocket.getInputStream()));
    BufferedReader stdIn =
        new BufferedReader(
            new InputStreamReader(System.in))
)
```

The first statement in the try-with resources statement creates a new Socket object and names it echoSocket. The Socket constructor used here requires the name of the computer and the port number to which you want to connect. The example program uses the first command-line argument as the name of the computer (the host name) and the second command line argument as the port number. When you run this program on your computer, make sure that the host name you use is the fully qualified IP name of the computer to which you want to connect. For example, if your echo server is running on the computer echoserver.example.com and it is listening on port number 7, first run the following command from the computer echoserver.example.com if you want to use the EchoServer example as your echo server:

java EchoServer 7

Afterward, run the EchoClient example with the following command:

java EchoClient echoserver.example.com 7

The second statement in the try-with resources statement gets the socket's output stream and opens a PrintWriter on it. Similarly, the third statement gets the socket's input stream and opens a BufferedReader on it. The example uses readers and writers so that it can write Unicode characters over the socket.

To send data through the socket to the server, the EchoClient example needs to write to the PrintWriter. To get the server's response, EchoClient reads from the BufferedReader object stdIn, which is created in the fourth statement in the try-with resources statement. If you are not yet familiar with the Java platform's I/O classes, you may wish to read Basic I/O.

The next interesting part of the program is the while loop. The loop reads a line at a time from the standard input stream and immediately sends it to the server by writing it to the PrintWriter connected to the socket:

```
String userInput;
while ((userInput = stdIn.readLine()) != null) {
    out.println(userInput);
    System.out.println("echo: " + in.readLine());
}
```

The last statement in the while loop reads a line of information from the BufferedReader connected to the socket. The readLine method waits until the server echoes the information back to EchoClient. When readline returns, EchoClient prints the information to the standard output.

The while loop continues until the user types an end-of-input character. That is, the EchoClient example reads input from the user, sends it to the Echo server, gets a response from the server, and displays it, until it reaches the end-of-input. (You can type an end-of-input character by pressing Ctrl-C.) The while loop then terminates, and the Java runtime automatically closes the readers and writers connected to the socket and to the standard input stream, and it closes the socket connection to the server. The Java runtime closes these resources automatically because they were created in the try-with-resources statement. The Java runtime closes these resources in reverse order that they were created. (This is good because streams connected to a socket should be closed before the socket itself is closed.)

KnockKnockServer.java

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

public class KnockKnockServer {
    public static void main(String[] args) throws IOException {

        if (args.length != 1) {
            System.err.println("Usage: java KnockKnockServer <port number>");
            System.exit(1);
        }

        int portNumber = Integer.parseInt(args[0]);

        try (
            ServerSocket serverSocket = new ServerSocket(portNumber);
            Socket clientSocket = serverSocket.accept();
            PrintWriter out =
                new PrintWriter(clientSocket.getOutputStream(), true);
            BufferedReader in = new BufferedReader(
                new InputStreamReader(clientSocket.getInputStream()));
        ) {

            String inputLine, outputLine;
```

```
            // Initiate conversation with client
            KnockKnockProtocol kkp = new KnockKnockProtocol();
            outputLine = kkp.processInput(null);
            out.println(outputLine);

            while ((inputLine = in.readLine()) != null) {
                outputLine = kkp.processInput(inputLine);
                out.println(outputLine);
                if (outputLine.equals("Bye."))
                    break;
            }
        } catch (IOException e) {
            System.out.println("Exception caught when trying to listen on port "
                + portNumber + " or listening for a connection");
            System.out.println(e.getMessage());
        }
    }
}
```

KnockKnockProtocol.java

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

public class KnockKnockProtocol {
    private static final int WAITING = 0;
    private static final int SENTKNOCKKNOCK = 1;
    private static final int SENTCLUE = 2;
    private static final int ANOTHER = 3;

    private static final int NUMJOKES = 5;

    private int state = WAITING;
    private int currentJoke = 0;

    private String[] clues = { "Turnip", "Little Old Lady", "Atch", "Who", "Who" };
    private String[] answers = { "Turnip the heat, it's cold in here!",
                                 "I didn't know you could yodel!",
                                 "Bless you!",
                                 "Is there an owl in here?",
                                 "Is there an echo in here?" };

    public String processInput(String theInput) {
        String theOutput = null;

        if (state == WAITING) {
            theOutput = "Knock! Knock!";
            state = SENTKNOCKKNOCK;
        } else if (state == SENTKNOCKKNOCK) {
            if (theInput.equalsIgnoreCase("Who's there?")) {
                theOutput = clues[currentJoke];
                state = SENTCLUE;
            } else {
                theOutput = "You're supposed to say \"Who's there?\"! " +
                            "Try again. Knock! Knock!";
            }
        } else if (state == SENTCLUE) {
            if (theInput.equalsIgnoreCase(clues[currentJoke] + " who?")) {
                theOutput = answers[currentJoke] + " Want another? (y/n)";
                state = ANOTHER;
            } else {
```

```
                theOutput = "You're supposed to say \"" +
                            clues[currentJoke] +
                            " who?\"" +
                            "! Try again. Knock! Knock!";
                state = SENTKNOCKKNOCK;
            }
        } else if (state == ANOTHER) {
            if (theInput.equalsIgnoreCase("y")) {
                theOutput = "Knock! Knock!";
                if (currentJoke == (NUMJOKES - 1))
                    currentJoke = 0;
                else
                    currentJoke++;
                state = SENTKNOCKKNOCK;
            } else {
                theOutput = "Bye.";
                state = WAITING;
            }
        }
        return theOutput;
    }
}
```

KnockKnockClient.java

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

public class KnockKnockClient {
    public static void main(String[] args) throws IOException {

        if (args.length != 2) {
            System.err.println(
                "Usage: java EchoClient <host name> <port number>");
            System.exit(1);
        }

        String hostName = args[0];
        int portNumber = Integer.parseInt(args[1]);

        try (
            Socket kkSocket = new Socket(hostName, portNumber);
            PrintWriter out = new PrintWriter(kkSocket.getOutputStream(), true);
            BufferedReader in = new BufferedReader(
                new InputStreamReader(kkSocket.getInputStream()));
        ) {
            BufferedReader stdIn =
                new BufferedReader(new InputStreamReader(System.in));
            String fromServer;
            String fromUser;

            while ((fromServer = in.readLine()) != null) {
                System.out.println("Server: " + fromServer);
                if (fromServer.equals("Bye."))
                    break;

                fromUser = stdIn.readLine();
                if (fromUser != null) {
                    System.out.println("Client: " + fromUser);
                    out.println(fromUser);
                }
```

```
            }
        } catch (UnknownHostException e) {
            System.err.println("Don't know about host " + hostName);
            System.exit(1);
        } catch (IOException e) {
            System.err.println("Couldn't get I/O for the connection to " +
                hostName);
            System.exit(1);
        }
    }
}
```