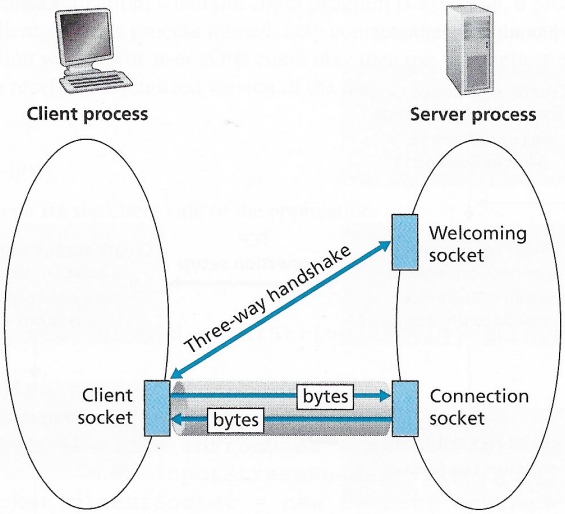


**Figure 2.29** ♦ Client-socket, welcoming socket, and connection socket

attached to some output source for the process, such as standard output (the monitor) or a socket out of which data flows into the Internet.

### 2.7.2 An Example Client-Server Application in Java

We use the following simple client-server application to demonstrate socket programming for both TCP and UDP:

1. A client reads a line from its **standard input** (keyboard) and sends the line out its socket to the server.
2. The server reads a line from its connection socket.
3. The server converts the line to uppercase.
4. The server sends the modified line out its connection socket to the client.
5. The client reads the modified line from its socket and prints the line on its **standard output** (monitor).

Figure 2.30 illustrates the main socket-related activity of the client and server.

Next we provide the client-server program pair for a TCP implementation of the application. We provide a detailed, line-by-line analysis after each program. The

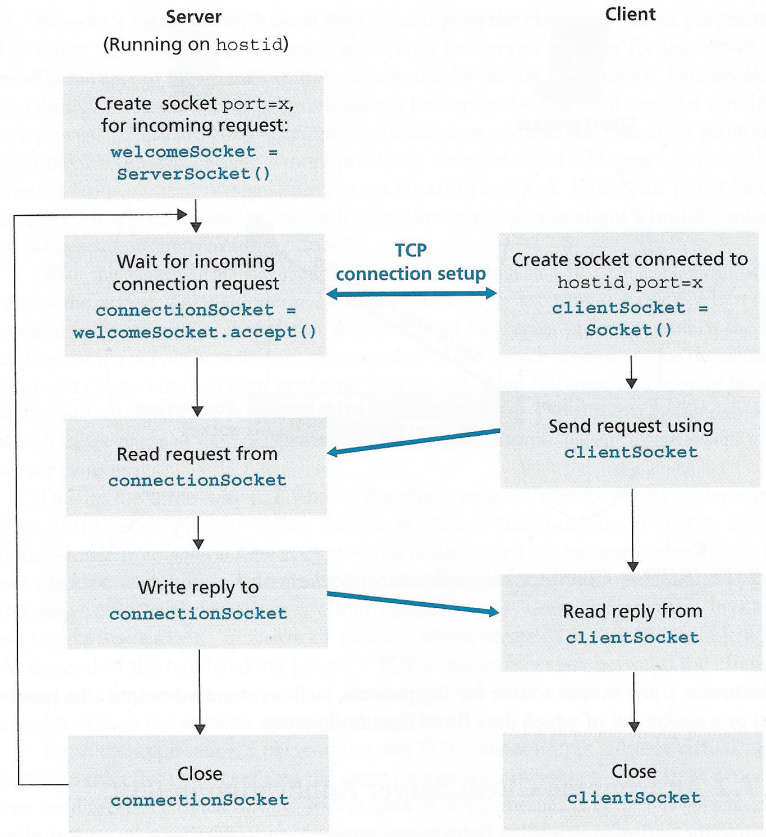


**Figure 2.30** ♦ The client-server application, using connection-oriented transport services

client program is called `TCPClient.java`, and the server program is called `TCPServer.java`. In order to emphasize the key issues, we intentionally provide code that is to the point but not bulletproof. "Good code" would certainly have a few more auxiliary lines.

Once the two programs are compiled on their respective hosts, the server program is first executed at the server host, which creates a server process at the server host. As discussed above, the server process waits to be contacted by a client process.

In this example application, when the client program is executed, a process is created at the client, and this process immediately contacts the server and establishes a TCP connection with it. The user at the client may then use the application to send a line and then receive a capitalized version of the line.

### TCPClient.java

Here is the code for the client side of the application:

```
import java.io.*;
import java.net.*;
class TCPClient {
   public static void main(String argv[]) throws Exception
   {
      String sentence;
      String modifiedSentence;
      BufferedReader inFromUser = new BufferedReader(
               new InputStreamReader(System.in));
      Socket clientSocket = new Socket("hostname", 6789);
      DataOutputStream outToServer = new DataOutputStream(
               clientSocket.getOutputStream());
      BufferedReader inFromServer =
         new BufferedReader(new InputStreamReader(
               clientSocket.getInputStream()));
      sentence = inFromUser.readLine();
      outToServer.writeBytes(sentence + '\n');
      modifiedSentence = inFromServer.readLine();
      System.out.println("FROM SERVER: " +
                         modifiedSentence);
      clientSocket.close();
   }
}
```

The program `TCPClient` creates three streams and one socket, as shown in Figure 2.31. The socket is called `clientSocket`. The stream `inFromUser` is an input stream to the program; it is attached to the standard input (that is, the keyboard). When the user types characters on the keyboard, the characters flow into the stream `inFromUser`. The stream `inFromServer` is another input stream to the program; it is attached to the socket. Characters that arrive from the network flow into the stream `inFromServer`. Finally, the stream `outToServer` is an output stream from the program; it is also attached to the socket. Characters that the client sends to the network flow into the stream `outToServer`.

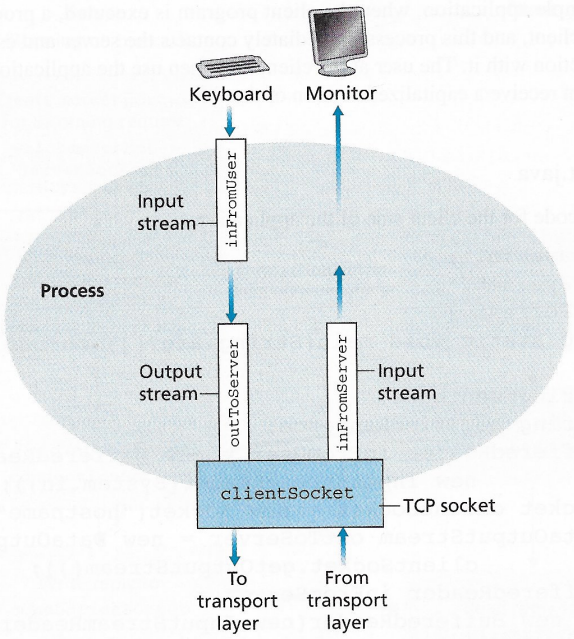


**Figure 2.31** ♦ `TCPclient` has three streams through which characters flow

Let's now take a look at the various lines in the code.

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

`java.io` and `java.net` are Java packages. The `java.io` package contains classes for input and output streams. In particular, the `java.io` package contains the `BufferedReader` and `DataOutputStream` classes, classes that the program uses to create the three streams previously illustrated. The `java.net` package provides classes for network support. In particular, it contains the `Socket` and `ServerSocket` classes. The `clientSocket` object of this program is derived from the `Socket` class.

```
class TCPClient {
    public static void main(String argv[]) throws Exception
          {......}
}
```

So far, what we've seen is standard stuff that you see at the beginning of most Java code. The third line is the beginning of a class definition block. The keyword `class` begins the class definition for the class named `TCPClient`. A class contains variables and methods. The variables and methods of the class are embraced by the curly brackets that begin and end the class definition block. The class `TCPClient` has no class variables and exactly one method, the `main()` method. Methods are similar to the functions or procedures in languages such as C; the `main()` method in the Java language is similar to the `main()` function in C and C++. When the Java interpreter executes an application (by being invoked upon the application's controlling class), it starts by calling the class's `main()` method. The `main()` method then calls all the other methods required to run the application. For this introduction to socket programming in Java, you may ignore the keywords `public`, `static`, `void`, `main`, and `throws Exceptions` (although you must include them in the code).

```
String sentence;
String modifiedSentence;
```

These above two lines declare objects of type `String`. The object `sentence` is the string typed by the user and sent to the server. The object `modifiedSentence` is the string obtained from the server and sent to the user's standard output.

```
BufferedReader inFromUser = new BufferedReader(
      new InputStreamReader(System.in));
```

The above line creates the stream object `inFromUser` of type `BufferedReader`. The input stream is initialized with `System.in`, which attaches the stream to the standard input. The command allows the client to read text from its keyboard.

```
Socket clientSocket = new Socket("hostname", 6789);
```

The above line creates the object `clientSocket` of type `Socket`. It also initiates the TCP connection between client and server. The string `"host-name"` must be replaced with the host name of the server (for example, `"apple.poly.edu"`). Before the TCP connection is actually initiated, the client performs a DNS lookup on the host name to obtain the host's IP address. The number 6789 is the port number. You can use a different port number, but you must make sure that you use the same port number at the server side of the application. As discussed earlier, the host's IP address along with the application's port number identifies the server process.

```
DataOutputStream outToServer =
   new DataOutputStream(clientSocket.getOutputStream());
BufferedReader inFromServer =
   new BufferedReader(new inputStreamReader(
                     clientSocket.getInputStream()));
```

The above two lines create stream objects that are attached to the socket. The `outToServer` stream provides the process output to the socket. The `inFromServer` stream provides the process input from the socket (see Figure 2.31).

```
sentence = inFromUser.readLine();
```

This line places a line typed by the user into the string `sentence`. The string `sentence` continues to gather characters until the user ends the line by typing a carriage return. The line passes from standard input through the stream `inFromUser` into the string `sentence`.

```
outToServer.writeBytes(sentence + '\n');
```

The above line sends the string `sentence` augmented with a carriage return into the `outToServer` stream. The augmented sentence flows through the client's socket and into the TCP pipe. The client then waits to receive characters from the server.

```
modifiedSentence = inFromServer.readLine();
```

When characters arrive from the server, they flow through the stream `inFromServer` and get placed into the string `modifiedSentence`. Characters continue to accumulate in `modifiedSentence` until the line ends with a carriage return character.

```
System.out.println("FROM SERVER " + modifiedSentence);
```

The above line prints to the monitor the string `modifiedSentence` returned by the server.

```
clientSocket.close();
```

This last line closes the socket and, hence, closes the TCP connection between the client and the server. It causes TCP in the client to send a TCP message to TCP in the server (see Section 3.5).

### TCPServer.java

Now let's take a look at the server program.

```
import java.io.*;
import java.net.*;
class TCPServer {
```

```
    public static void main(String argv[]) throws Exception
        {
            String clientSentence;
            String capitalizedSentence;
            ServerSocket welcomeSocket = new ServerSocket
               (6789);
            while(true) {
               Socket connectionSocket = welcomeSocket.
                  accept();
               BufferedReader inFromClient =
                  new BufferedReader(new InputStreamReader(
                     connectionSocket.getInputStream()));
               DataOutputStream outToClient =
                  new DataOutputStream(
                     connectionSocket.getOutputStream());
               clientSentence = inFromClient.readLine();
               capitalizedSentence =
                     clientSentence.toUpperCase() + '\n';
               outToClient.writeBytes(capitalizedSentence);
            }
        }
}
```

`TCPServer` has many similarities with `TCPClient`. Let's now take a look at the lines in `TCPServer.java`. We will not comment on the lines that are identical or similar to commands in `TCPClient.java`.

The first line in `TCPServer` is substantially different from what we saw in `TCPClient`:

```
ServerSocket welcomeSocket = new ServerSocket(6789);
```

This line creates the object `welcomeSocket`, which is of type `ServerSocket`. The `welcomeSocket` is a sort of door that listens for a knock from some client. The `welcomeSocket` listens on port number 6789. The next line is

```
Socket connectionSocket = welcomeSocket.accept();
```

This line creates a *new* socket, called `connectionSocket`, when some client knocks on `welcomeSocket`. This socket also has port number 6789. (We'll explain why both sockets have the same port number in Chapter 3.) TCP then establishes a direct virtual pipe between `clientSocket` at the client and `connectionSocket` at the server. The client and server can then send bytes to each other

over the pipe, and all bytes sent arrive at the other side in order. With `connectionSocket` established, the server can continue to listen for requests from other clients for the application using `welcomeSocket`. (This version of the program doesn't actually listen for more connection requests, but it can be modified with threads to do so.) The program then creates several stream objects, analogous to the stream objects created in `clientSocket`. Now consider

```
capitalizedSentence = clientSentence.toUpperCase() + '\n';
```

This command is the heart of the application. It takes the line sent by the client, capitalizes it, and adds a carriage return. It uses the method `toUpperCase()`. All the other commands in the program are peripheral; they are used for communication with the client.

To test the program pair, you install and compile `TCPClient.java` in one host and `TCPServer.java` in another host. Be sure to include the proper hostname of the server in `TCPClient.java`. You next execute `TCPServer.class`, the compiled server program, in the server. This creates a process in the server that idles until it is contacted by some client. Then you execute `TCPClient.class`, the compiled client program, in the client. This creates a process in the client and establishes a TCP connection between the client and server processes. Finally, to use the application, you type a sentence followed by a carriage return.

To develop your own client-server application, you can begin by slightly modifying the programs. For example, instead of converting all the letters to uppercase, the server can count the number of times the letter *s* appears and return this number.

## 2.8 Socket Programming with UDP

We learned in the previous section that when two processes communicate over TCP, it is as if there were a pipe between the two processes. This pipe remains in place until one of the two processes closes it. When one of the processes wants to send some bytes to the other process, it simply inserts the bytes into the pipe. The sending process does not have to attach a destination address to the bytes because the pipe is logically connected to the destination. Furthermore, the pipe provides a reliable byte-stream channel—the sequence of bytes received by the receiving process is exactly the sequence of bytes that the sender inserted into the pipe.

UDP also allows two (or more) processes running on different hosts to communicate. However, UDP differs from TCP in many fundamental ways. First, UDP is a connectionless service—there isn't an initial handshaking phase during which a pipe is established between the two processes. Because UDP doesn't have a pipe, when a process wants to send a batch of bytes to another process, the sending process must