2.7 Socket Programming with TCP

Now that we have looked at a number of important network applications, let's explore how network application programs are actually written. In this section we'll write application programs that use TCP; in the following section we'll write programs that use UDP.

Recall from Section 2.1 that many network applications consist of a pair of programs—a client program and a server program—residing in two different end systems. When these two programs are executed, a client and a server process are created, and these processes communicate with each other by reading from and writing to sockets. When creating a network application, the developer's main task is to write the code for both the client and server programs.

There are two sorts of network applications. One sort is an implementation of a protocol standard defined in, for example, an RFC. For such an implementation, the client and server programs must conform to the rules dictated by the RFC. For example, the client program could be an implementation of the client side of the FTP protocol, described in Section 2.3 and explicitly defined in RFC 959; similarly, the server program could be an implementation of the FTP server protocol, also explicitly defined in RFC 959. If one developer writes code for the client program and an independent developer writes code for the server program, and both developers carefully follow the rules of the RFC, then the two programs will be able to interoperate. Indeed, many of today's network applications involve communication between client and server programs that have been created by independent developers—for example, a Firefox browser communicating with an Apache Web server, or an FTP client on a PC uploading a file to a Linux FTP server. When a client or server program implements a protocol defined in an RFC, it should use the port number associated with the protocol. (Port numbers were briefly discussed in Section 2.1. They are covered in more detail in Chapter 3.)

The other sort of network application is a proprietary network application. In this case the application-layer protocol used by the client and server programs do not

necessarily conform to any existing RFC. A single developer (or development team) creates both the client and server programs, and the developer has complete control over what goes in the code. But because the code does not implement a public-domain protocol, other independent developers will not be able to develop code that interoperates with the application. When developing a proprietary application, the developer must be careful not to use one of the well-known port numbers defined in the RFCs.

In this and the next section, we examine the key issues in developing a proprietary client-server application. During the development phase, one of the first decisions the developer must make is whether the application is to run over TCP or over UDP. Recall that TCP is connection oriented and provides a reliable byte-stream channel through which data flows between two end systems. UDP is connectionless and sends independent packets of data from one end system to the other, without any guarantees about delivery.

In this section we develop a simple client application that runs over TCP; in the next section, we develop a simple client application that runs over UDP. We present these simple TCP and UDP applications in Java. We could have written the code in C or C++, but we opted for Java mostly because the applications are more neatly and cleanly written in Java. With Java there are fewer lines of code, and each line can be explained to the novice programmer without much difficulty. But there is no need to be frightened if you are not familiar with Java. You should be able to follow the code if you have experience programming in another language.

For readers who are interested in client/server programming in C, there are several good references available [Donahoo 2001; Stevens 1997; Frost 1994; Kurose 1996].

2.7.1 Socket Programming with TCP

Recall from Section 2.1 that processes running on different machines communicate with each other by sending messages into sockets. We said that each process was analogous to a house and the process's socket is analogous to a door. As shown in Figure 2.30, the socket is the door between the application process and TCP. The application developer has control of everything on the application-layer side of the socket; however, it has little control of the transport-layer side. (At the very most, the application developer has the ability to fix a few TCP parameters, such as maximum buffer size and maximum segment size.)

Now let's take a closer look at the interaction of the client and server programs. The client has the job of initiating contact with the server. In order for the server to be able to react to the client's initial contact, the server has to be ready. This implies two things. First, the server program cannot be dormant—that is, it must be running as a process before the client attempts to initiate contact. Second, the server program must have some sort of door—more precisely, a socket—that welcomes some initial contact from a client process running on an arbitrary host. Using our house/door analogy for a process/socket, we will sometimes refer to the client's initial contact as "knocking on the welcoming door."

Key:

● NATed peer

● Super peer and relay

— Neighboring relationships in overlay network
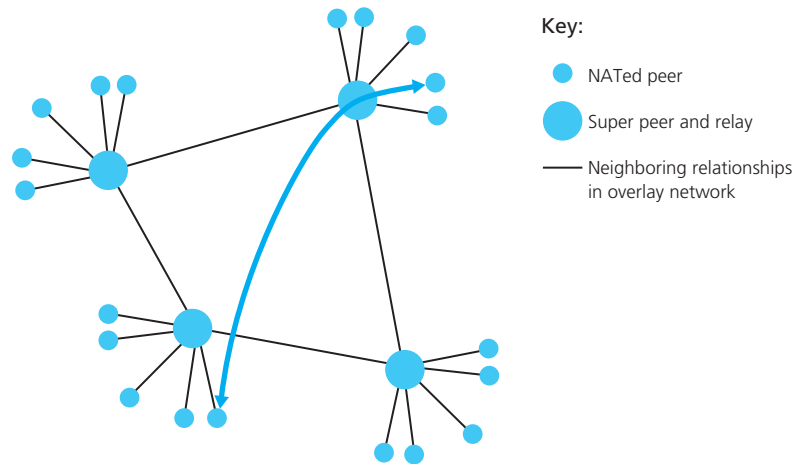
Figure 2.30 ♦ Processes communicating through TCP sockets

With the server process running, the client process can initiate a TCP connection to the server. This is done in the client program by creating a socket. When the client creates its socket, it specifies the address of the server process, namely, the IP address of the server host and the port number of the server process. Once the socket has been created in the client program, TCP in the client initiates a three-way handshake and establishes a TCP connection with the server. The three-way handshake, which takes place at the transport layer, is completely transparent to the client and server programs.

During the three-way handshake, the client process knocks on the welcoming door of the server process. When the server "hears" the knocking, it creates a new door—more precisely, a new socket—that is dedicated to that particular client. In our example below, the welcoming door is a `ServerSocket` object that we call the `welcomeSocket`. When a client knocks on this door, the program invokes `welcomeSocket`'s `accept()` method, which creates a new door for the client. At the end of the handshaking phase, a TCP connection exists between the client's socket and the server's new socket. Henceforth, we refer to the server's new, dedicated socket as the server's **connection socket**.

From the application's perspective, the TCP connection is a direct virtual pipe between the client's socket and the server's connection socket. The client process can send arbitrary bytes into its socket, and TCP guarantees that the server process will receive (through the connection socket) each byte in the order sent. TCP thus provides a **reliable byte-stream service** between the client and server processes. Furthermore, just as people can go in and out the same door, the client process not only sends bytes
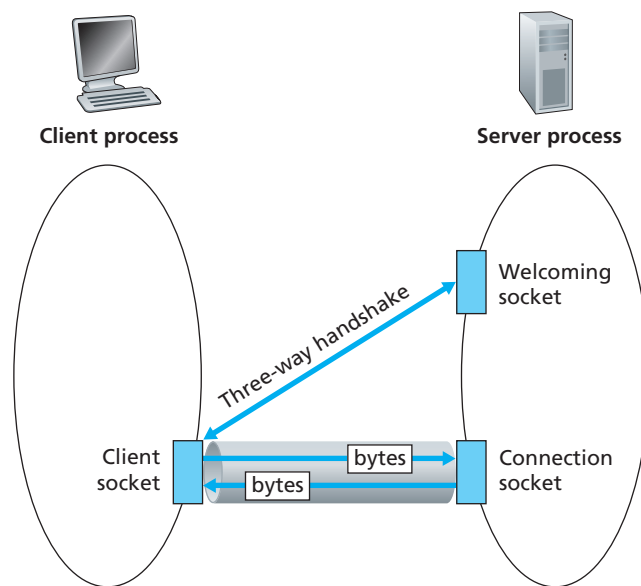
Figure 2.31 ♦ Client-socket, welcoming socket, and connection socket

into but also receives bytes from its socket; similarly, the server process not only receives bytes from but also sends bytes into its connection socket. This is illustrated in Figure 2.31. Because sockets play a central role in client/server applications, client/server application development is also referred to as socket programming.

Before providing our example client-server application, it is useful to discuss the notion of a stream. A **stream** is a sequence of characters that flow into or out of a process. Each stream is either an **input stream** for the process or an **output stream** for the process. If the stream is an input stream, then it is attached to some input source for the process, such as standard input (the keyboard) or a socket into which data flows from the Internet. If the stream is an output stream, then it is 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 2.32 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
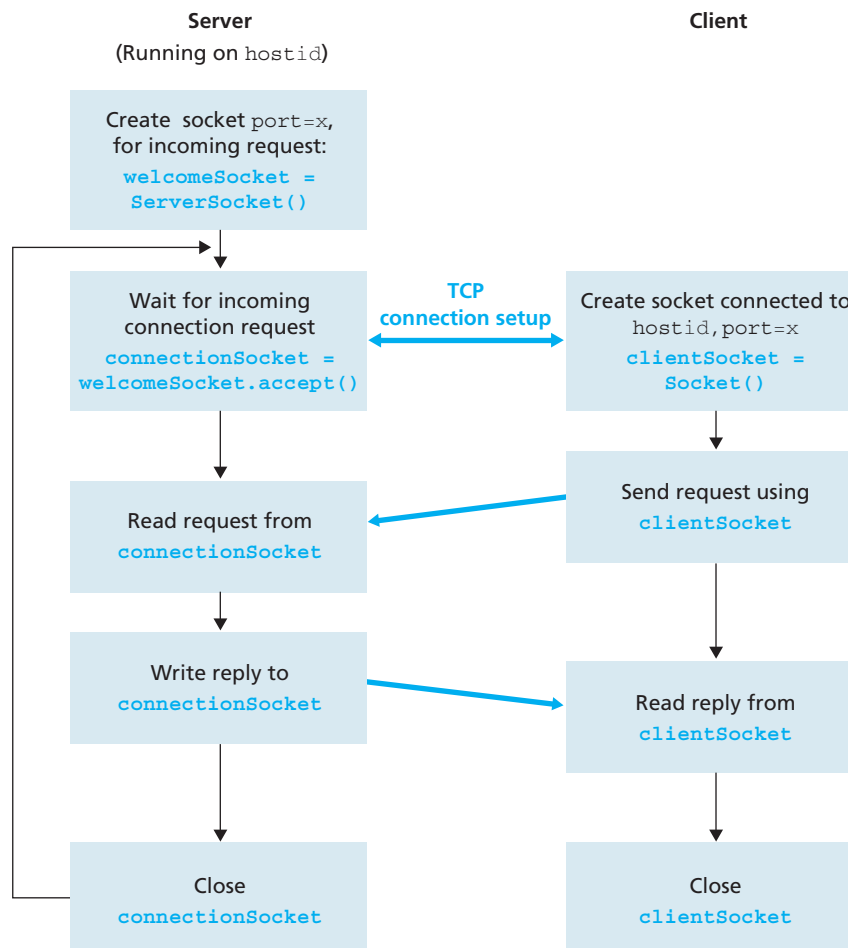
Figure 2.32 ♦ The client-server application, using connection-oriented transport services

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 pro-
gram 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
2.33. 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).
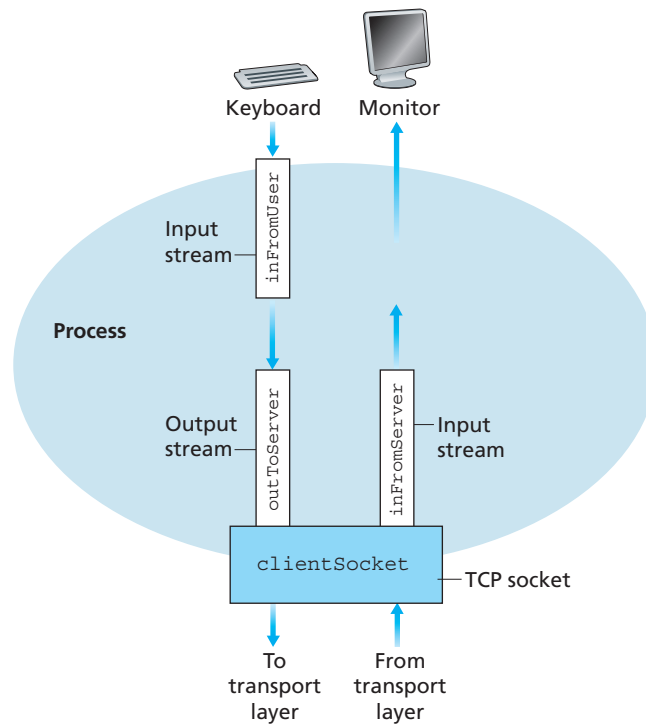
Figure 2.33 ♦ `TCPclient` has three streams through which characters flow

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`.

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 `TCP-Client` 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 `Buffered Reader`. 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 `out-ToServer` stream provides the process output to the socket. The `inFromServer` stream provides the process input from the socket (see Figure 2.33).

```
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 send-ing 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 reli-able 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 commu-nicate. 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 attach the destination process's address to the batch of bytes. And this must be done for each batch of bytes the sending process sends. As an analogy, consider a group of 20 persons who take five taxis to a common destination; as the people enter the taxis, each taxi driver must separately be informed of the destination. Thus, UDP is similar to a taxi service. The destination address is a tuple consisting of the IP address of the destination host and the port number of the destination process. We refer to the batch of information bytes along with the IP destination address and port number as the "packet." UDP provides an unreliable message-oriented service model, in that it makes a best effort to deliver the batch of bytes to the destination. It is message-oriented in that batches are bytes that are sent in a single zero operation at the send-ing side, will be delivered as a batch at the receiving side; this contrasts with TCP's byte-stream semantics. UDP service is best-effort in that UDP makes no guarantee that the batch of bytes will indeed be delivered. The UDP service thus contrasts sharply (in several respects) with TCP's reliable byte-stream service model.

After having created a packet, the sending process pushes the packet into the network through a socket. Continuing with our taxi analogy, at the other side of the sending socket, there is a taxi waiting for the packet. The taxi then drives the packet in the direction of the packet's destination address. However, the taxi does not guar-antee that it will eventually get the packet to its ultimate destination—the taxi could break down or suffer some other unforeseen problem. In other terms, *UDP provides an unreliable transport service to its communication processes*—it makes no guarantees that a packet will reach its ultimate destination.

In this section we illustrate socket programming by redeveloping the same application of the previous section, but this time over UDP. We'll see that the code for UDP is different from the TCP code in many important ways. In particular, there is (1) no initial handshaking between the two processes and therefore no need for a welcoming socket, (2) no streams are attached to the sockets, (3) the sending hosts create packets by attaching the IP destination address and port number to each batch of bytes it sends, and (4) the receiving process must unravel each received packet to obtain the packet's information bytes. Recall once again our simple application:

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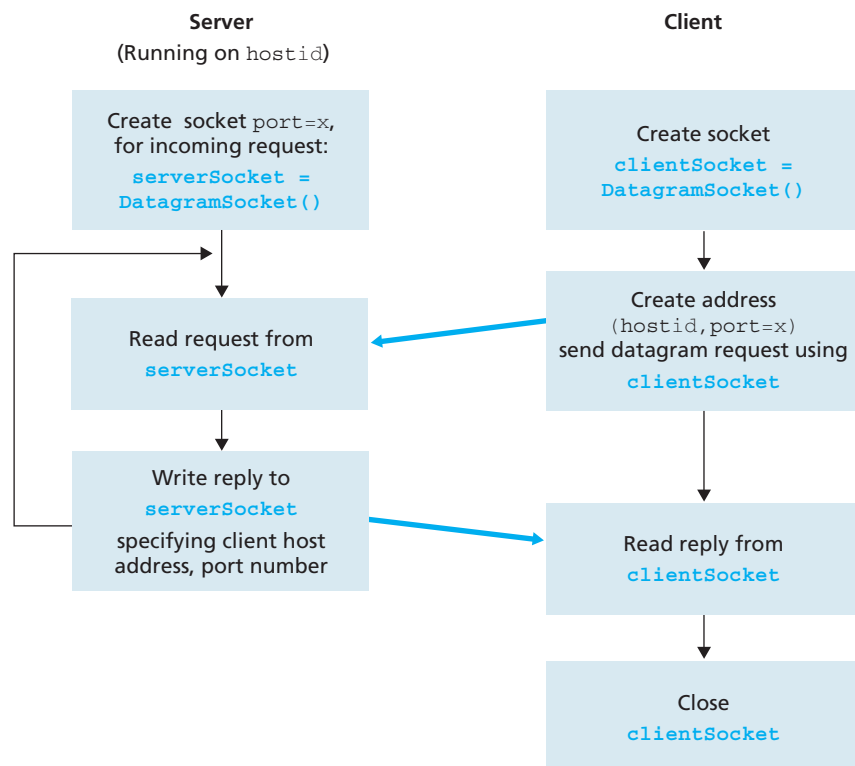
2. The server reads a line from its socket.
3. The server converts the line to uppercase.
4. The server sends the modified line out its socket to the client.
5. The client reads the modified line from its socket and prints the line on its standard output (monitor).

Figure 2.34 highlights the main socket-related activity of the client and server that communicate over a connectionless (UDP) transport service.
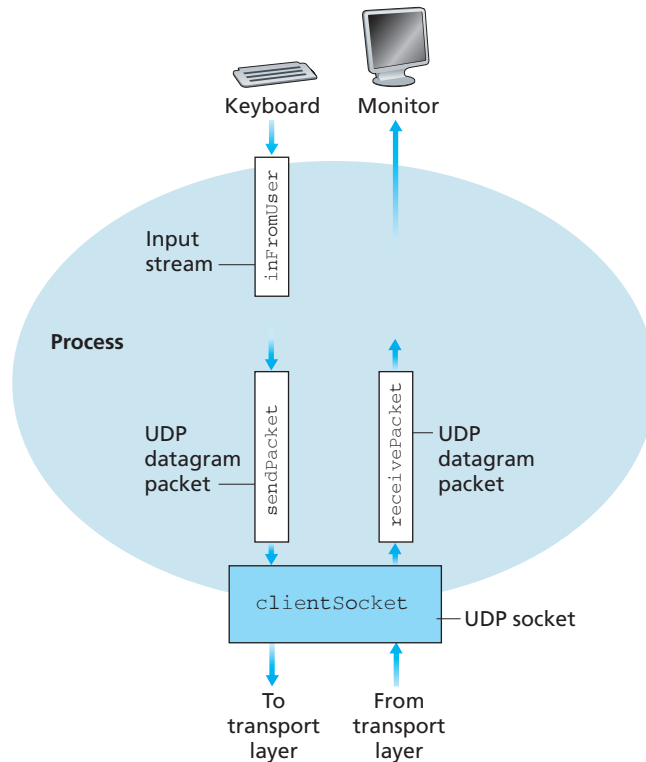
UDPClient.java

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

```java
import java.io.*;
import java.net.*;
class UDPClient {
   public static void main(String args[]) throws Exception
   {
      BufferedReader inFromUser =
         new BufferedReader(new InputStreamReader
                     (System.in));
      DatagramSocket clientSocket = new DatagramSocket();
      InetAddress IPAddress =
                     InetAddress.getByName("hostname");
      byte[] sendData = new byte[1024];
      byte[] receiveData = new byte[1024];
      String sentence = inFromUser.readLine();
      sendData = sentence.getBytes();
      DatagramPacket sendPacket =
         new DatagramPacket(sendData, sendData.length,
                     IPAddress, 9876);
      clientSocket.send(sendPacket);
      DatagramPacket receivePacket =
         new DatagramPacket(receiveData,
                     receiveData.length);
      clientSocket.receive(receivePacket);
      String modifiedSentence =
         new String(receivePacket.getData());
      System.out.println("FROM SERVER:" +
                     modifiedSentence);
      clientSocket.close();
   }
}
```

<div align="center">

Server
(Running on `hostid`)

Client

</div>

Server	Client
Create socket `port=x`, for incoming request: **`serverSocket = DatagramSocket()`**	Create socket **`clientSocket = DatagramSocket()`**
Read request from **`serverSocket`**	Create address `(hostid,port=x)` send datagram request using **`clientSocket`**
Write reply to **`serverSocket`** specifying client host address, port number	Read reply from **`clientSocket`**
	Close **`clientSocket`**

Figure 2.34 ♦ The client-server application, using connectionless transport
services

The program `UDPClient.java` constructs one stream and one socket, as
shown in Figure 2.35. The socket is called `clientSocket`, and it is of type
`DatagramSocket`. Note that UDP uses a different kind of socket than TCP at
the client. In particular, with UDP our client uses a `DatagramSocket,` whereas
with TCP our client used a `Socket`. The stream `inFromUser` is an input stream
to the program; it is attached to the standard input, that is, to the keyboard. We had
an equivalent stream in our TCP version of the program. When the user types
characters on the keyboard, the characters flow into the stream `inFromUser`.
But in contrast with TCP, there are no streams (input or output) attached to the
socket. Instead of feeding bytes to the stream attached to a `Socket` object, UDP
will push individual packets through the `DatagramSocket` object.

Figure 2.35 ♦ `UDPClient` has one stream; the socket accepts packets from the process and delivers packets to the process.

Let's now take a look at the lines in the code that differ significantly from `TCPClient.java`.

```
DatagramSocket clientSocket = new DatagramSocket();
```

This line creates the object `clientSocket` of type `DatagramSocket`. In contrast with `TCPClient.java`, this line does not initiate a TCP connection. In particular, the client host does not contact the server host upon execution of this line. For this reason, the constructor `DatagramSocket()` does not take the server host name or port number as arguments. Using our door-pipe analogy, the execution of the above line creates a door for the client process but does not create a pipe between the two processes.

```
InetAddress IPAddress = InetAddress.getByName("hostname");
```

In order to send bytes to a destination process, we need the address of the process. Part of this address is the IP address of the destination host. The above line invokes a DNS lookup that translates the hostname (in this example, supplied in the code by the developer) to an IP address. DNS was also invoked by the TCP version of the client, although it was done there implicitly rather than explicitly. The method `getByName()` takes as an argument the hostname of the server and returns the IP address of this same server. It places this address in the object `IPAddress` of type `InetAddress`.

```
byte[] sendData = new byte[1024];
byte[] receiveData = new byte[1024];
```

The byte arrays `sendData` and `receiveData` will hold the data the client sends and receives, respectively.

```
sendData = sentence.getBytes();
```

The above line essentially performs a type conversion. It takes the string `sentence` and renames it as `sendData`, which is an array of bytes.

```
DatagramPacket sendPacket = new DatagramPacket(
    sendData, sendData.length, IPAddress, 9876);
```

This line constructs the packet, `sendPacket`, which the client will pop into the network through its socket. This packet includes that data that is contained in the packet, `sendData`, the length of this data, the IP address of the server, and the port number of the application (which we have set to 9876). Note that `sendPacket` is of type `DatagramPacket`.

```
clientSocket.send(sendPacket);
```

In the above line, the method `send()` of the object `clientSocket` takes the packet just constructed and pops it into the network through `clientSocket`. Once again, note that UDP sends the line of characters in a manner very different from TCP. TCP simply inserted the string of characters into a stream, which had a logical direct connection to the server; UDP creates a packet that includes the address of the server. After sending the packet, the client then waits to receive a packet from the server.

```
DatagramPacket receivePacket =
    new DatagramPacket(receiveData, receiveData.length);
```

In the above line, while waiting for the packet from the server, the client creates a place-holder for the packet, `receivePacket`, an object of type `DatagramPacket`.

```
clientSocket.receive(receivePacket);
```

The client idles until it receives a packet; when it does receive a packet, it puts the packet in `receivePacket`.

```
String modifiedSentence =
    new String(receivePacket.getData());
```

The above line extracts the data from `receivePacket` and performs a type conversion, converting an array of bytes into the string `modifiedSentence`.

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

This line, which is also present in `TCPClient`, prints out the string `modified-Sentence` at the client's monitor.

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

This last line closes the socket. Because UDP is connectionless, this line does not cause the client to send a transport-layer message to the server (in contrast with `TCPClient`).

UDPServer.java

Let's now take a look at the server side of the application:

```
import java.io.*;
import java.net.*;
class UDPServer {
   public static void main(String args[]) throws Exception
       {
       DatagramSocket serverSocket = new
                            DatagramSocket(9876);
       byte[] receiveData = new byte[1024];
       byte[] sendData = new byte[1024];
       while(true)
           {
           DatagramPacket receivePacket =
                 new DatagramPacket(receiveData,
                         receiveData.length);
           serverSocket.receive(receivePacket);
           String sentence = new String(
                     receivePacket.getData());
```

```
        InetAddress IPAddress =
                  receivePacket.getAddress();
        int port = receivePacket.getPort();
        String capitalizedSentence =
                  sentence.toUpperCase();
        sendData = capitalizedSentence.getBytes();
        DatagramPacket sendPacket =
        new DatagramPacket(sendData,
              sendData.length, IPAddress, port);
        serverSocket.send(sendPacket);
     }
   }
}
```

The program `UDPServer.java` constructs one socket, as shown in Figure 2.36. The socket is called `serverSocket`. It is an object of type `DatagramSocket`, as was the socket in the client side of the application. Once again, no streams are attached to the socket.

Let's now take a look at the lines in the code that differ from `TCPServer.java`.

```
DatagramSocket serverSocket = new DatagramSocket(9876);
```
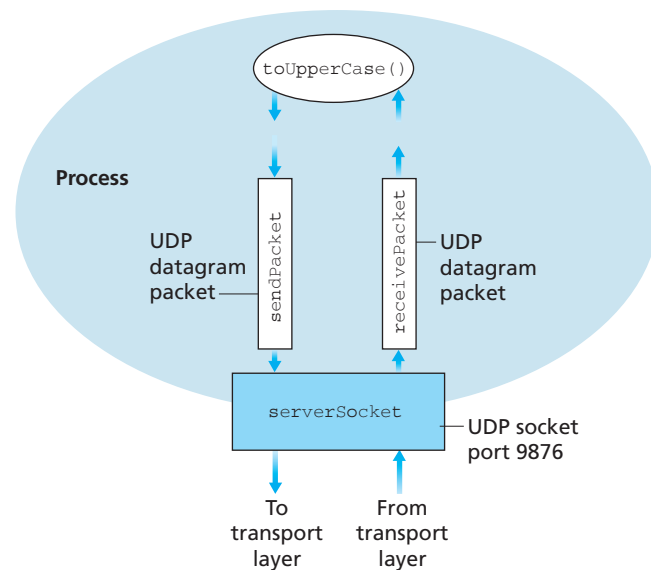

Figure 2.36 ♦ `UDPServer` has no streams; the socket accepts packets
from the process and delivers packets to the process.

The above line constructs the `DatagramSocket serverSocket` at port 9876. All data sent and received will pass through this socket. Because UDP is connectionless, we do not have to create a new socket and continue to listen for new connection requests, as done in `TCPServer.java`. If multiple clients access this application, they will all send their packets into this single door, `serverSocket`.

```
String sentence = new String(receivePacket.getData());
InetAddress IPAddress = receivePacket.getAddress();
int port = receivePacket.getPort();
```

The above three lines unravel the packet that arrives from the client. The first of the three lines extracts the data from the packet and places the data in the `String sentence`; it has an analogous line in `UDPClient`. The second line extracts the IP address; the third line extracts the client port number, which is chosen by the client and is different from the server port number 9876. (We will discuss client port numbers in some detail in the next chapter.) It is necessary for the server to obtain the address (IP address and port number) of the client, so that it can send the capitalized sentence back to the client.

That completes our analysis of the UDP program pair. To test the application, you install and compile `UDPClient.java` in one host and `UDPServer.java` in another host. (Be sure to include the proper hostname of the server in `UDPClient.java`.) Then execute the two programs on their respective hosts. Unlike with TCP, you can first execute the client side and then the server side. This is because the client process does not attempt to initiate a connection with the server when you execute the client program. Once you have executed the client and server programs, you may use the application by typing a line at the client.