2.7 Socket Programming with UDP

We learned in the previous section that when two processes communicate over TCP, from the perspective of the processes it is as if there is 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 reliably byte stream channel -- the sequence of bytes received by the receiving process is exactly the sequence 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 exclude 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. Thus UDP is similar to a taxi service -- each time a group of people get in a taxi, the group has to inform the driver of the destination address. As with TCP, the destination address is a tuple consisting of the IP address of the destination host and the port number of the destination process. We shall refer to the batch of information bytes along with the IP destination address and port number as the the "packet".

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 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 guarantee that it will eventually get the datagram to its ultimate destination; the taxi could break down. In other terms, *UDP provides an unreliable transport service to its communication processes* -- it makes no guarantees that a datagram will reach its ultimate destination.

In this section we will illustrate UDP client-server programming by redeveloping the same application of the previous section, but this time over UDP. We shall also see that the Java code for UDP is different from the TCP code in many important ways. In particular, we shall see that there is (i) no initial handshaking between the two processes, and therefore no need for a welcoming socket, (ii) no streams are attached to the sockets, (iii) the sending hosts creates "packets" by attaching the IP destination address and port number to

each batch of bytes it sends, and (iv) the receiving process must unravel to 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 through its socket and prints the line on its standard output (monitor).

UDPClient.java

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

```
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();

    }
}
```

The program UDPClient.java constructs one stream and one socket, as shown in Figure 2.7-1. 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, i.e., 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 stream attached to a Socket object, UDP will push individual packets through the DatagramSocket object.

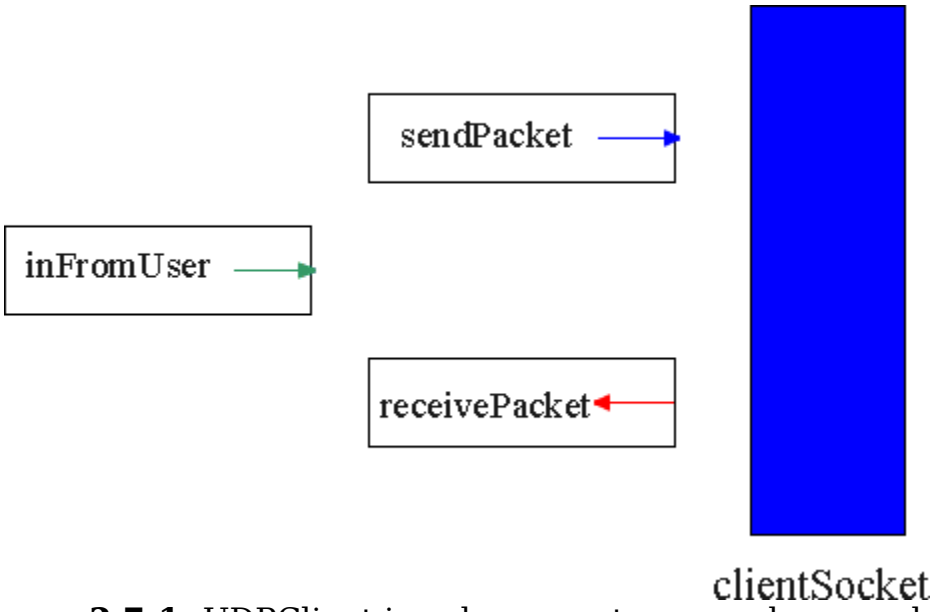

Figure 2.7-1: UDPClient.java has one stream and one socket.

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

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

The above 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 hostname 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 shall need to obtain the address of the process. Part of this address is the IP address of the destination host. The above line invokes a DNS look up that translates "hostname" (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);
```

The above line constructs the packet, **sendPacket**, that the 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 line into a stream, which had a logical direct connection to the server; UDP creates a packet which 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);
```

The above, which is also present in TCPClient, prints out the string **modifiedSentence** 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 tranport-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 2.7-2. 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.

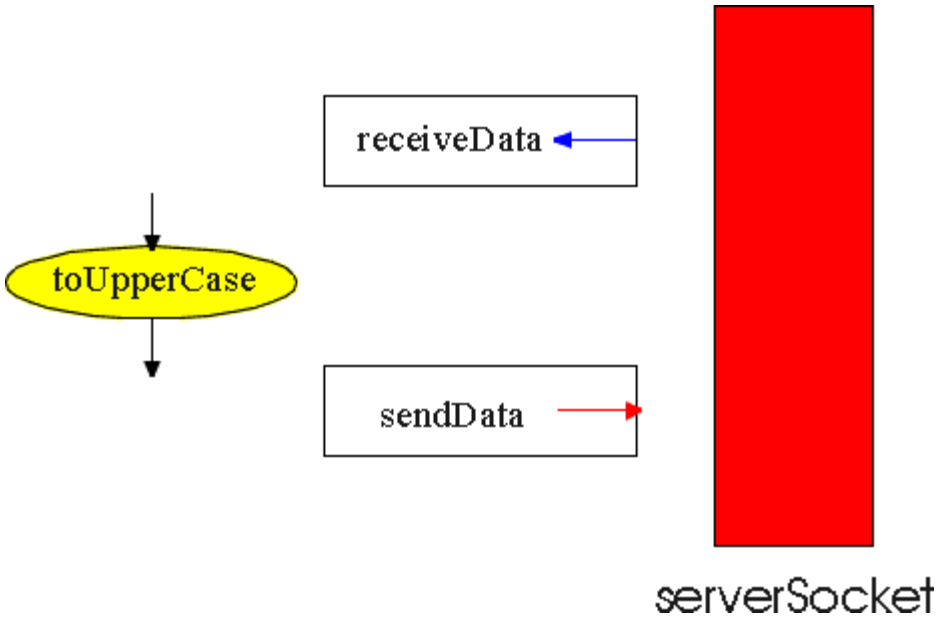

Figure 2.7-2: UDPServer.java has one socket.

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

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

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 spawn 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, when you execute the client side, the client process does not attempt to initiate a connection with the server. Once you have executed the client and server programs, you may use the application by typing a line at the client.

Return to Table Of Contents
