**2.7 Socket Programming: Creating Network Applications**

Now that we’ve looked at a number of important network applications, let’s explore how network application programs are actually created. Recall from Section 2.1 that a typical network application consists 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 process and a server process are created, and these processes communicate with each other by reading from, and writing to, sockets. When creat- ing a network application, the developer’s main task is therefore to write the code for both the client and server programs.

There are two types of network applications. One type is an implementation whose operation is specified in a protocol standard, such as an RFC or some other standards document; such an application is sometimes referred to as “open,” since the rules specifying its operation are known to all. For such an implementation, the client and server programs must conform to the rules dic- tated 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 another 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 a BitTorrent client communicat- ing with BitTorrent tracker.

The other type of network application is a proprietary network application. In this case the client and server programs employ an application-layer protocol that has *not* been openly published in an RFC or elsewhere. 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 an open protocol, other independent developers will not be able to develop code that interoperates with the application.

In this section, we’ll examine the key issues in developing a client-server appli- cation, and we’ll “get our hands dirty” by looking at code that implements a very simple 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. Recall also that when a client or server program implements a protocol defined by an RFC, it should use the well-known port number associated with the protocol; conversely, when developing a propri- etary application, the developer must be careful to avoid using such well-known port numbers. (Port numbers were briefly discussed in Section 2.1. They are cov- ered in more detail in Chapter 3.)

We introduce UDP and TCP socket programming by way of a simple UDP appli- cation and a simple TCP application. We present the simple UDP and TCP applica- tions in Python. We could have written the code in Java, C, or C++, but we chose Python mostly because Python clearly exposes the key socket concepts. With Python there are fewer lines of code, and each line can be explained to the novice program- mer without difficulty. But there’s no need to be frightened if you are not familiar with Python. You should be able to easily follow the code if you have experience program- ming in Java, C, or C++.

If you are interested in client-server programming with Java, you are encour- aged to see the companion Web site for this textbook; in fact, you can find there all the examples in this section (and associated labs) in Java. For readers who are inter- ested in client-server programming in C, there are several good references available [Donahoo 2001; Stevens 1997; Frost 1994; Kurose 1996]; our Python examples below have a similar look and feel to C.

**2.7.1 Socket Programming with UDP**

In this subsection, we’ll write simple client-server programs that use UDP; in the following section, we’ll write similar programs that use TCP.

Recall from Section 2.1 that processes running on different machines communi- cate with each other by sending messages into sockets. We said that each process is analogous to a house and the process’s socket is analogous to a door. The application resides on one side of the door in the house; the transport-layer protocol resides on the other side of the door in the outside world. 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.

Now let’s take a closer look at the interaction between two communicating processes that use UDP sockets. Before the sending process can push a packet of data out the socket door, when using UDP, it must first attach a destination address to the packet. After the packet passes through the sender’s socket, the Internet will use this destination address to route the packet through the Internet to the socket in the receiving process. When the packet arrives at the receiving socket, the receiving process will retrieve the packet through the socket, and then inspect the packet’s contents and take appropriate action.

So you may be now wondering, what goes into the destination address that is attached to the packet? As you might expect, the destination host’s IP address is part of the destination address. By including the destination IP address in the packet, the routers in the Internet will be able to route the packet through the Internet to the desti- nation host. But because a host may be running many network application processes, each with one or more sockets, it is also necessary to identify the particular socket in the destination host. When a socket is created, an identifier, called a **port number**, is assigned to it. So, as you might expect, the packet’s destination address also includes the socket’s port number. In summary, the sending process attaches to the packet a des- tination address which consists of the destination host’s IP address and the destination socket’s port number. Moreover, as we shall soon see, the sender’s source address— consisting of the IP address of the source host and the port number of the source socket—are also attached to the packet. However, attaching the source address to the packet is typically *not* done by the UDP application code; instead it is automatically done by the underlying operating system.

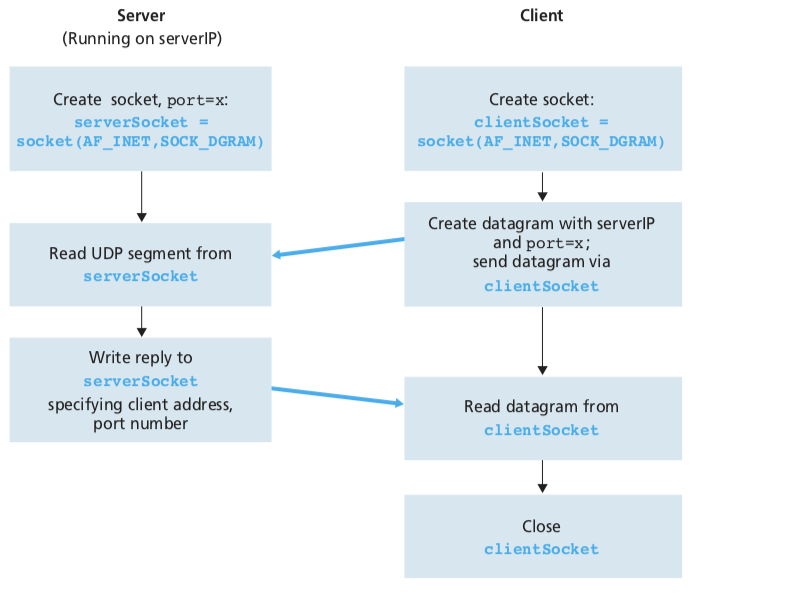
We’ll use the following simple client-server application to demonstrate socket programming for both UDP and TCP:

1. The client reads a line of characters (data) from its keyboard and sends the data to the server.
2. The server receives the data and converts the characters to uppercase.
3. The server sends the modified data to the client.
4. The client receives the modified data and displays the line on its screen.

Figure 2.28 highlights the main socket-related activity of the client and server that communicate over the UDP transport service.

Now let’s get our hands dirty and take a look at the client-server program pair for a UDP implementation of this simple application. We also provide a detailed, line-by-line analysis after each program. We’ll begin with the UDP client, which will send a simple application-level message to the server. In order for the server to be able to receive and reply to the client’s message, it must be ready and running— that is, it must be running as a process before the client sends its message.

The client program is called UDPClient.py, and the server program is called UDPServer.py. In order to emphasize the key issues, we intentionally provide code that is minimal. “Good code” would certainly have a few more auxiliary lines, in particular for handling error cases. For this application, we have arbitrarily chosen 12000 for the server port number.



**Figure 2.28** 􏰀The client-server application using UDP

**UDPClient.py**

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

from socket import \*  
serverName = ‘hostname’  
serverPort = 12000  
clientSocket = socket(socket.AF\_INET, socket.SOCK\_DGRAM) message = raw\_input(’Input lowercase sentence:’) clientSocket.sendto(message,(serverName, serverPort)) modifiedMessage, serverAddress = clientSocket.recvfrom(2048) print modifiedMessage

clientSocket.close()

Now let’s take a look at the various lines of code in UDPClient.py.

from socket import \*  
The socket module forms the basis of all network communications in Python. By

including this line, we will be able to create sockets within our program.

serverName = ‘hostname’ serverPort = 12000

The first line sets the string serverName to hostname. Here, we provide a string containing either the IP address of the server (e.g., “128.138.32.126”) or the host- name of the server (e.g., “cis.poly.edu”). If we use the hostname, then a DNS lookup will automatically be performed to get the IP address.) The second line sets the inte- ger variable serverPort to 12000.

clientSocket = socket(socket.AF\_INET, socket.SOCK\_DGRAM)

This line creates the client’s socket, called clientSocket. The first parameter indicates the address family; in particular, AF\_INET indicates that the underlying network is using IPv4. (Do not worry about this now—we will discuss IPv4 in Chapter 4.) The second parameter indicates that the socket is of type SOCK\_DGRAM, which means it is a UDP socket (rather than a TCP socket). Note that we are not specifying the port number of the client socket when we create it; we are instead let- ting the operating system do this for us. Now that the client process’s door has been created, we will want to create a message to send through the door.

message = raw\_input(*’*Input lowercase sentence:*’*)

raw\_input() is a built-in function in Python. When this command is executed, the user at the client is prompted with the words “Input data:” The user then uses her keyboard to input a line, which is put into the variable message. Now that we have a socket and a message, we will want to send the message through the socket to the destination host.

clientSocket.sendto(message,(serverName, serverPort))

In the above line, the method sendto() attaches the destination address (serverName, serverPort)tothemessageandsendstheresultingpacketinto the process’s socket, clientSocket. (As mentioned earlier, the source address is also attached to the packet, although this is done automatically rather than explicitly by the code.) Sending a client-to-server message via a UDP socket is that simple! After sending the packet, the client waits to receive data from the server.

modifiedMessage, serverAddress = clientSocket.recvfrom(2048)

With the above line, when a packet arrives from the Internet at the client’s socket, the packet’s data is put into the variable modifiedMessage and the packet’s source address is put into the variable serverAddress. The variable serverAddress contains both the server’s IP address and the server’s port number. The program UDPClient doesn’t actually need this server address infor- mation, since it already knows the server address from the outset; but this line of Python provides the server address nevertheless. The method recvfrom also takes the buffer size 2048 as input. (This buffer size works for most purposes.)

print modifiedMessage

This line prints out modifiedMessage on the user’s display. It should be the original line that the user typed, but now capitalized.

clientSocket.close()

This line closes the socket. The process then terminates.

**UDPServer.py**

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

from socket import \*  
serverPort = 12000  
serverSocket = socket(AF\_INET, SOCK\_DGRAM) serverSocket.bind((*’’*, serverPort))  
print *”The server is ready to receive”* while 1:

message, clientAddress = serverSocket.recvfrom(2048) modifiedMessage = message.upper() serverSocket.sendto(modifiedMessage, clientAddress)

Note that the beginning of UDPServer is similar to UDPClient. It also imports the socket module, also sets the integer variable serverPort to 12000, and also cre- ates a socket of type SOCK\_DGRAM (a UDP socket). The first line of code that is significantly different from UDPClient is:

serverSocket.bind((*’’*, serverPort))

The above line binds (that is, assigns) the port number 12000 to the server’s socket. Thus in UDPServer, the code (written by the application developer) is explicitly assigning a port number to the socket. In this manner, when anyone sends a packet to port 12000 at the IP address of the server, that packet will be directed to this socket. UDPServer then enters a while loop; the while loop will allow UDPServer to receive and process packets from clients indefinitely. In the while loop, UDPServer waits for a packet to arrive.

message, clientAddress = serverSocket.recvfrom(2048)

This line of code is similar to what we saw in UDPClient. When a packet arrives at the server’s socket, the packet’s data is put into the variable message and the packet’s source address is put into the variable clientAddress. The variable clientAddress contains both the client’s IP address and the client’s port number. Here, UDPServer *will* make use of this address information, as it provides a return address, similar to the return address with ordinary postal mail. With this source address information, the server now knows to where it should direct its reply.

modifiedMessage = message.upper()

This line is the heart of our simple application. It takes the line sent by the client and uses the method upper() to capitalize it.

serverSocket.sendto(modifiedMessage, clientAddress)

This last line attaches the client’s address (IP address and port number) to the capi- talized message, and sends the resulting packet into the server’s socket. (As men- tioned earlier, the server address is also attached to the packet, although this is done automatically rather than explicitly by the code.) The Internet will then deliver the packet to this client address. After the server sends the packet, it remains in the while loop, waiting for another UDP packet to arrive (from any client running on any host).

To test the pair of programs, you install and compile UDPClient.py in one host and UDPServer.py in another host. Be sure to include the proper hostname or IP address of the server in UDPClient.py. Next, you execute UDPServer.py, the com- piled server program, in the server host. This creates a process in the server that idles until it is contacted by some client. Then you execute UDPClient.py, the com- piled client program, in the client. This creates a process in the client. Finally, to use the application at the client, you type a sentence followed by a carriage return.

To develop your own UDP client-server application, you can begin by slightly modifying the client or server programs. For example, instead of con- verting all the letters to uppercase, the server could count the number of times the letter *s* appears and return this number. Or you can modify the client so that after receiving a capitalized sentence, the user can continue to send more sentences to the server.

**2.7.2 Socket Programming with TCP**

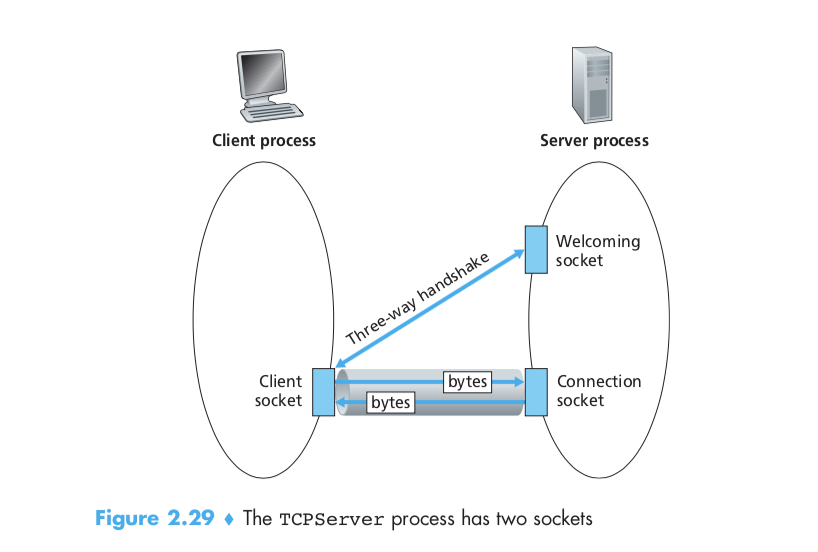
Unlike UDP, TCP is a connection-oriented protocol. This means that before the client and server can start to send data to each other, they first need to handshake and estab- lish a TCP connection. One end of the TCP connection is attached to the client socket and the other end is attached to a server socket. When creating the TCP connection, we associate with it the client socket address (IP address and port number) and the server socket address (IP address and port number). With the TCP connection estab- lished, when one side wants to send data to the other side, it just drops the data into the TCP connection via its socket. This is different from UDP, for which the server must attach a destination address to the packet before dropping it into the socket.

Now let’s take a closer look at the interaction of client and server programs in TCP. 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, as in the case of UDP, the TCP server must be run- ning as a process before the client attempts to initiate contact. Second, the server program must have a special door—more precisely, a special socket—that wel- comes 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.”

With the server process running, the client process can initiate a TCP connec- tion to the server. This is done in the client program by creating a TCP socket. When the client creates its TCP socket, it specifies the address of the welcoming socket in the server, namely, the IP address of the server host and the port number of the socket. After creating its socket, the client initiates a three-way handshake and establishes a TCP connection with the server. The three-way handshake, which takes place within the transport layer, is completely invisible to the client and server pro- grams.

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 TCP socket object that we call serverSocket; the newly created socket dedicated to the client making the connection is called connec- tionSocket. Students who are encountering TCP sockets for the first time some- times confuse the welcoming socket (which is the initial point of contact for all clients wanting to communicate with the server), and each newly created server-side connec- tion socket that is subsequently created for communicating with each client.

From the application’s perspective, the client’s socket and the server’s connec- tion socket are directly connected by a pipe. As shown in Figure 2.29, 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 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 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.



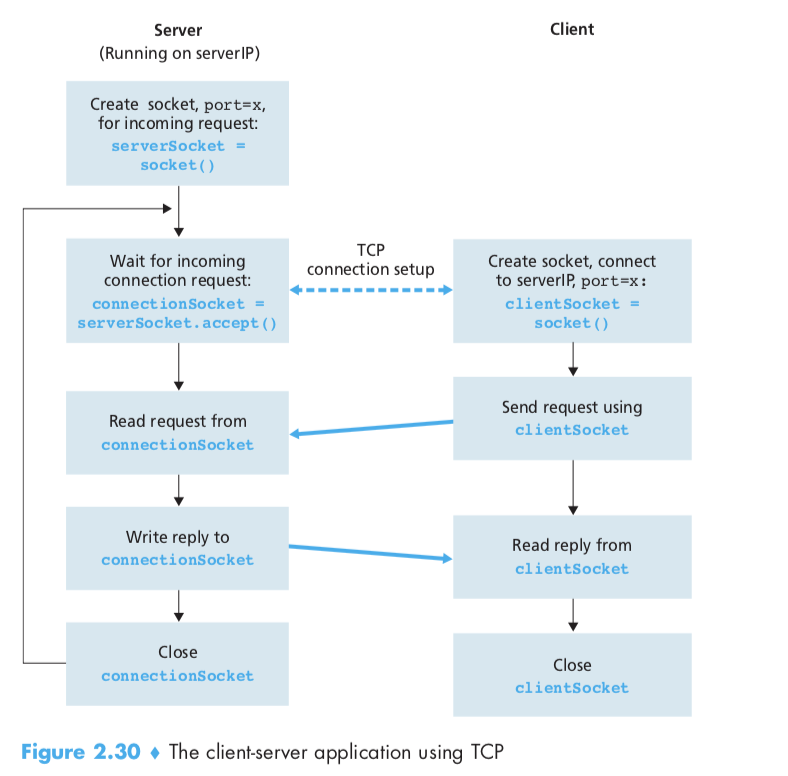
We use the same simple client-server application to demonstrate socket program- ming with TCP: The client sends one line of data to the server, the server capitalizes the line and sends it back to the client. Figure 2.30 highlights the main socket-related activity of the client and server that communicate over the TCP transport service.

**TCPClient.py**

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

from socket import \*  
serverName = ’servername’  
serverPort = 12000  
clientSocket = socket(AF\_INET, SOCK\_STREAM) clientSocket.connect((serverName,serverPort)) sentence = raw\_input(‘Input lowercase sentence:’) clientSocket.send(sentence)

modifiedSentence = clientSocket.recv(1024) print ‘From Server:’, modifiedSentence clientSocket.close()



Let’s now take a look at the various lines in the code that differ significantly from the UDP implementation. The first such line is the creation of the client socket.

clientSocket = socket(AF\_INET, SOCK\_STREAM)

This line creates the client’s socket, called clientSocket. The first parameter again indicates that the underlying network is using IPv4. The second parameter indicates that the socket is of type SOCK\_STREAM, which means it is a TCP socket (rather than a UDP socket). Note that we are again not specifying the port number of the client socket when we create it; we are instead letting the operating system do this for us. Now the next line of code is very different from what we saw in UDPClient:

clientSocket.connect((serverName,serverPort))

Recall that before the client can send data to the server (or vice versa) using a TCP socket, a TCP connection must first be established between the client and server. The above line initiates the TCP connection between the client and server. The parameter of the connect() method is the address of the server side of the con- nection. After this line of code is executed, the three-way handshake is performed and a TCP connection is established between the client and server.

sentence = raw\_input(‘Input lowercase sentence:’)

As with UDPClient, the above obtains a sentence from the user. The string sentence continues to gather characters until the user ends the line by typing a carriage return. The next line of code is also very different from UDPClient:

clientSocket.send(sentence)

The above line sends the string sentence through the client’s socket and into the TCP connection. Note that the program does *not* explicitly create a packet and attach the destination address to the packet, as was the case with UDP sockets. Instead the client program simply drops the bytes in the string sentence into the TCP con- nection. The client then waits to receive bytes from the server.

modifiedSentence = clientSocket.recv(2048)

When characters arrive from the server, they get placed into the string modified- Sentence. Characters continue to accumulate in modifiedSentence until the line ends with a carriage return character. After printing the capitalized sentence, we close the client’s socket:

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.py**

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

from socket import \*  
serverPort = 12000  
serverSocket = socket(AF\_INET,SOCK\_STREAM) serverSocket.bind((‘’,serverPort)) serverSocket.listen(1)  
print ‘The server is ready to receive’ while 1:

connectionSocket, addr = serverSocket.accept() sentence = connectionSocket.recv(1024) capitalizedSentence = sentence.upper() connectionSocket.send(capitalizedSentence) connectionSocket.close()

Let’s now take a look at the lines that differ significantly from UDPServer and TCP- Client. As with TCPClient, the server creates a TCP socket with:

serverSocket=socket(AF\_INET,SOCK\_STREAM)  
Similar to UDPServer, we associate the server port number, serverPort, with

this socket:

serverSocket.bind((‘’,serverPort))

But with TCP, serverSocket will be our welcoming socket. After establish- ing this welcoming door, we will wait and listen for some client to knock on the door:

serverSocket.listen(1)

This line has the server listen for TCP connection requests from the client. The parameter specifies the maximum number of queued connections (at least 1).

connectionSocket, addr = serverSocket.accept()

When a client knocks on this door, the program invokes the accept() method for serverSocket, which creates a new socket in the server, called connec- tionSocket, dedicated to this particular client. The client and server then complete the handshaking, creating a TCP connection between the client’s clientSocket and the server’s connectionSocket. With the TCP connection established, the client and server can now send bytes to each other over the connection. With TCP, all bytes sent from one side not are not only guaranteed to arrive at the other side but also guaranteed arrive in order.

connectionSocket.close()

In this program, after sending the modified sentence to the client, we close the con- nection socket. But since serverSocket remains open, another client can now knock on the door and send the server a sentence to modify.

This completes our discussion of socket programming in TCP. You are encour- aged to run the two programs in two separate hosts, and also to modify them to achieve slightly different goals. You should compare the UDP program pair with the TCP program pair and see how they differ. You should also do many of the socket programming assignments described at the ends of Chapters 2, 4, and 7. Finally, we hope someday, after mastering these and more advanced socket programs, you will write your own popular network application, become very rich and famous, and remember the authors of this textbook!