Introduction to Network Programming with Python

Norman Matloff
University of California, Davis

April 20, 2003

Contents

1 Overview

The TCP/IP network protocol suite is the standard method for intermachine communication. Though originally integral only to the UNIX operating system, its usage spread to all OS types, and it is the basis of the entire Internet. This document will briefly introduce the subject of TCP/IP programming using the Python language.

A TCP/IP application consists of a pair of programs, called a **server** and a **client**. If for example you use a Web browser to view **www.yahoo.com**, the browser is the client, and the Web server at Yahoo headquarters is the server.

2 Our Example Client/Server Pair

As our main illustration of client/server programming in Python, we have modified a simple example in the Library Reference section of the Python documentation page, `http://www.python.org/doc/current/lib`. Here is the server, **tms.py**:

```
# simple illustration client/server pair; client program sends a string
# to server, which echoes it back to the client (in multiple copies),
# and the latter prints to the screen

# this is the server

import socket
import sys

# create a socket
s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)

# associate the socket with a port
host = ''  # can leave this blank on the server side
port = int(sys.argv[1])
s.bind((host, port))

# accept "call" from client
s.listen(1)
conn, addr = s.accept()
print 'client is at', addr

# read string from client (assumed here to be so short that one call to
# recv() is enough), and make multiple copies (to show the need for the
# "while" loop on the client side)

data = conn.recv(1024)
data = 10000 * data

# wait for the go-ahead signal from the keyboard (shows that recv() at
# the client will block until server sends)
z = raw_input()

# now send
conn.send(data)

```

```
# close the connection
conn.close()
```

And here is the client, **tmc.py**:

```
# simple illustration client/server pair; client program sends a string
# to server, which echoes it back to the client (in multiple copies),
# and the latter prints to the screen

# this is the client

import socket
import sys

# create a socket
s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)

# connect to server
host = sys.argv[1]  # server address
port = int(sys.argv[2])  # server port
s.connect((host, port))

s.send(sys.argv[3])  # send test string

# read echo
i = 0
while(1):
   data = s.recv(1024)  # read up to 1024 bytes
   i += 1
   if (i < 5):
      print data
      if not data:  # if end of data, leave loop
         break
   print 'received', len(data), 'bytes'

# close the connection
s.close()
```

This client/server pair doesn't do much. The client sends a test string to the server, and the server sends back multiple copies of the string. The client then prints the earlier part of that echoed material to the user's screen, to demonstrate that the echoing is working, and also prints the amount of data received on each read, to demonstrate the "chunky" nature of TCP.

You should run this client/server pair before reading further.[1] Start up the server on one machine, by typing

```
python tms.py 2000
```

and then start the client at another machine, by typing

```
python tmc.py server_machine_name 2000 abc
```

[1]The source file from which this document is created, **PyNet.tex**, should be available wherever you downloaded the PDF file. You can get the client and server programs from the source file, rather than having to type them up yourself.

The two main points to note when you run the programs are that (a) the client will block until you provide some keyboard input at the server machine, and (b) the client will receive data from the server in rather random-sized chunks.

3 Analysis of the Server Program

‘ Now, let’s look at the server.

Line 7: We import the **socket** class from Python’s library; this contains all the communication methods we need.

Line 11: We create a socket. At this point, it is merely a placeholder; we have not taken any network actions yet.[2] The two arguments state that we wish to the socket to be an Internet socket,[3] and that it will use the TCP method of organizing data.[4]

Line 16: We invoke the **socket** class’ **bind()** method. Say for example we specify port 2000 on the command line when we run the server (obtained on Line 15). When we call **bind()**, the operating system will first check to see whether port 2000 is already in use by some other process.[5] If so, an exception will be raised, but otherwise the OS will reserve port 2000 for the server. What that means is that from now on, whenever TCP data reaches this machine and specifies port 2000, that data will be copied to our server program.[6] Note that **bind()** takes a single argument consisting of a 2-element tuple, rather than 2 scalar arguments.

Line 19: The **listen()** method tells the OS to start watching for connection requests from remote clients on port 2000. The argument 1 here tells the OS to allow only 1 pending connection request at a time. In other words, any request which comes in while the first one is being processed is rejected. This argument is often set to 5 in other applications, but we will only have one request in our situation here anyway.

By calling **listen()** on the socket **s**, we have told the OS to consider it a **listening socket**. That means its sole purpose is to accept connections with clients; it is not used for the actual transfer of data back and forth between clients and the server.

Line 20: The **accept()** method tells the OS to wait for a connection request. It will block until a request comes in from a client at a remote machine.[7] That will occur when the client executes a **connect()** call (Line 16 of **tmc.py**). When that call occurs, the OS at the client machine will first set up an **ephemeral port** on that machine, which is a port for the server to use when sending information to the client. The OS on the client machine sends a connection request to the server machine, informing the latter as to (a) the Internet address of the client machine and (b) the ephemeral port of the client.

At that point, the connection has been established.[8] The OS on the server machine sets up a new socket,

[2]For those who have programmed in C/C++, a socket is very much like a file handle. It is meaningless until we call **fopen()** and assign the return value to the file handle variable.

[3]As opposed to something called a **UNIX socket**, which is internal to a machine, i.e. not on the Internet.

[4]This is the main method used today. There is also the UDP approach, which is also a member of the TCP/IP protocol suite, but used only in specialized applications.

[5]This could be another invocation of our server program, or a different program entirely.

[6]There is nothing physical about the port, in contrast to items called *ports* in I/O cards. A port number is merely a mechanism which the OS uses to determine to which program incoming data should be copied.

[7]Which, technically, could be the same machine.

[8]The connection is not physical either. It merely is an agreement between the two machines to send data back and forth, with certain limits as to how much can be sent in one piece, etc.

termed a **connected socket**, which will be used in the server's communication with the remote client.[9]

All this releases **accept()** from its blocking status, and it returns a 2-element tuple. The first element of that tuple, assigned here to **conn**, is the connected socket. Again, this is what will be used to communicate with the client (e.g. on Line 35).

The second item returned by **accept()** tells us who the client is, i.e. the Internet address of the client, in case we need to know that.[10]

Line 27: The **recv()** method reads data from the given socket. The argument states the maximum number of bytes we are willing to receive. This depends on how much memory we are willing to have our server use; other than this concern, there is no reason not to make this argument extremely large.

It is absolutely crucial, though, to understand how TCP works in this regard. Consider a connection set up between a client X and server Y. The entirety of data that X sends to Y is considered one giant message. If for example X sends text data in the form of 27 lines totalling 619 characters, TCP makes no distinction between one line and another; TCP simply considers it to be one 619-byte message.

Yet, that 619-byte message might not arrive all at once. It might, for instance, come into two pieces, one of 402 bytes and the other of 217 bytes.[11] For that reason, we seldom see a one-time call to **recv()** in real production code, as we see here on Line 27. Instead, the call is typically part of a loop, as can be seen starting on Line 22 of the client, **tmc.py**. In other words, here on Line 27 of the server, we have been rather sloppy, going on the assumption that the data from the client will be so short that it will arrive in just one piece. In a production program, we would use a loop.

Line 28: In order to show the need for such a loop in general, I have modified the original example by making the data really long. Recall that in Python, "multiplying" a string means duplicating it. For example:

```
>>> x = 3*'abc'
>>> x
'abcabcabc'
```

Again, I put this in deliberately, so as to necessitate using a loop in the client, as we will see below.

Line 32: This too is inserted for the purpose of illustrating a principle later in the client. It takes some keyboard input at the server machine. The input is not actually used.

Line 35: The server finally sends its data to the client.

Line 38: The server closes the connection. At this point, the giant message considered to have been sent by the server to the client is complete. This will be sensed by the client, as discussed below.

[9]You might wonder why there are separate listening and connected sockets. Typically a server will simultaneously be connected to many clients. The server might "take turns" among the connection sockets, reading one then another then another, or actually establish separate processes or threads for each client. So it needs a separate socket for communication with each client.

[10]When I say "we," I mean "we, the authors of this server program." That information may be optional for us, though obviously to the OS on the machine where the server is running this information is vital. The OS also needs to know the client's ephemeral port, while "we" would almost never have a need for that.

[11]And again, that 402-byte piece may not consist of an integer number of lines. It may, and probably would, end somewhere in the middle of a line.

4 Analysis of the Client Program

Now, what about the client code?

Line 16: The client makes the connection with the server. Note that both the server's Internet machine address and the server's port number are needed. As soon as this line is executed, Line 20 on the server side, which had been waiting, will finally execute.

Line 18: The client sends its data to the server.

Lines 22ff: The client reads the message from the server. As explained earlier, this is done in a loop, because the message is likely to come in random-sized chunks.[12] Again, even though Line 35 of the server gave the data to its OS in one piece, the OS may not send it out to the network in one piece, and thus the client must loop, repeatedly calling **recv()**.

That raises the question of how the client will know that it has received the entire message sent by the server. The answer is that **recv()** will return an empty string when that occurs. And in turn, that will occur when the server executes **close()** on Line 38.[13]

Note:

- **recv()** will block if no data has been received but the connection has not been closed
- **recv()** will return a null value when the connection is closed

5 More on the "Stream" Nature of TCP

As mentioned earlier, TCP regards all the data sent by a client or server as one giant message. If the data consists of lines of text, TCP will not pay attention to the demarcations between lines. This means that if your application is text/line-oriented, you must handle such demarcation yourself. If for example the client sends lines of text to the server, your server code must look for newline characters and separate lines on its own. Note too that in one call to **recv()** we might receive, say, all of one line and part of the next line, in which case we must keep the latter for piecing together with the bytes we get from our next call to **recv()**.

In our example here, the client and server each execute **send()** only once, but in many applications they will alternate. The client will send something to the server, then the server will send something to the client, then the client will send something to the server, and so on. In such a situation, it will still be the case that the totality of all bytes sent by the client will be considered one single message by the TCP/IP system, and the same will be true for the server.

6 A Closer Look at the Role of Ports

When the server accepts a connection from a client, the connected socket will be given the same port as the listening socket. If the server has connections open with several clients, and the associated connected

[12]The client says it is willing to accept chunks of length as large as 1024 bytes, but as explained before, this is merely done to keep down the amount of memory used by the client.

[13]This would also occur if **conn** were to be garbage-collected when its scope ended, including the situation in which the server exits altogether.

sockets all use the same port, how does the OS at the server machine decide which connected socket to give incoming data to for that port?

The answer lies in the fact that different clients will have different Internet addresses. In fact, two clients could be on the same machine, and thus have the same Internet address, yet still be distinguished from each other by the OS at the server machine, because the two clients would have different ephemeral addresses. So it all wokrs out.