

Tutorial on Network Programming with Python

Norman Matloff
University of California, Davis
©2003-2007, N. Matloff

April 10, 2007

Contents

1	Overview of Networks	3
1.1	Networks and MAC Addresses	3
1.2	The Internet and IP Addresses	3
1.3	Ports	3
1.4	Connectionless and Connection-Oriented Communication	4
1.5	Clients and Servers	5
2	Our Example Client/Server Pair	5
2.1	Analysis of the Server Program	6
2.2	Analysis of the Client Program	9
3	Role of the OS	9
3.1	Basic Operation	9
3.2	How the OS Distinguishes Between Multiple Connections	10
4	The <code>sendall()</code> Function	10
5	Sending Lines of Text	11
5.1	Remember, It's Just One Big Byte Stream, Not "Lines"	11
5.2	The Wonderful <code>makefile()</code> Function	11
5.3	Getting the Tail End of the Data	13
6	Dealing with Asynchronous Inputs	13

6.1	Nonblocking Sockets	14
6.2	Advanced Methods of Polling	17

1 Overview of Networks

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. See <http://heather.cs.ucdavis.edu/~matloff/Networks/Intro/NetIntro.pdf> for a more detailed introduction to networks and TCP/IP.

1.1 Networks and MAC Addresses

A **network** means a Local Area Network. Here machines are all on a single cable, or as is more common now, what amounts to a single cable (e.g. multiple wires running through a switch). The machines on the network communicate with each other by the **MAC addresses**, which are 48-bit serial numbers burned into their network interface cards (NICs). If machine A wishes to send something to machine B on the same network, A will put B's MAC address into the message packet. B will see the packet, recognize its own MAC address as destination, and accept the packet.

1.2 The Internet and IP Addresses

An internet—note the indefinite article and the lower-case *i*—is simply a connection of two or more networks. One starts, say, with two networks and places a computer on both of them (so it will have two NICs). Machines on one network can send to machines on the other by sending to the computer in common, which is acting as a **router**. These days, many routers are not full computers, but simply boxes that do only routing. One of these two networks can be then connected to a third in the same way, so we get a three-network internet, and so on. In some cases, two networks are connected by having a machine on one network connected to a machine on the other via a high-speed phone line, or even a satellite connection.

The Internet—note the definite article and the capital *I*—consists of millions of these networks connected in that manner.

On the Internet, every machine has an Internet Protocol (IP) address. The original ones were 32 bits wide, and the new ones 64. If machine A on one network wants to send to machine Z on an distant network, A sends to a router, which sends to another router and so on until the message finally reaches Z's network. The local router there then puts Z's MAC address in the packet, and places the packet on that network. Z will see it and accept the packet.

Note that when A sends the packet to a local router, the latter may not know how to get to Z. But it will send to another router “in that direction,” and after this process repeats enough times, Z will receive the message. Each router has only limited information about the entire network, but it knows enough to get the journey started.

1.3 Ports

The term *port* is overused in the computer world. In some cases it means something physical, such as a place into which you can plug your USB device. In our case here, though, ports are not physical. Instead, they are essentially tied to processes on a machine.

Think of our example above, where machine A sends to machine Z. That phrasing is not precise enough. Instead, we should say that a process on machine A sends to a process on machine Z. These are identified by ports, which are just numbers similar to file descriptors/handles.

So, when A sends to Z, it will send to a certain port at Z. Moreover, the packet will also state which process at A—stated in terms of a port number at A—sent the message, so that Z knows where to send a reply.

The ports below 1024 are reserved for the famous services, for example port 80 for HTTP. These are called **well-known ports**.¹ User programs use port numbers of 1024 or higher. By the way, keep in mind that a port stays in use for a few seconds after a connection is close; trying to start a connection at that port again within this time period will result in an exception.

1.4 Connectionless and Connection-Oriented Communication

One can send one single packet at a time. We simply state that our message will consist of a single packet, and state the destination IP address or port. This is called **connectionless** communication, termed UDP under today's Internet protocol. It's simple, but not good for general use. For example, a message might get lost, and the sender would never know. Or, if the message is long, it is subject to corruption, which would not be caught by a connectionless setup, and also long messages monopolize network bandwidth.²

So, we usually use a **connection-oriented** method, TCP. What happens here is that a message from A to Z is first broken into pieces, which are sent separately from each other. At Z, the TCP layer of the network protocol stack in the OS will collect all the pieces, check them for errors, put them in the right order, etc. and deliver them to the proper process at Z.

We say that there is a **connection** between A and Z. Again, this is not physical. It merely is an agreement between the OSs at A and Z that they will exchange data between these processes/ports at A and Z in the orderly manner described above.

One point which is crucial to keep in mind is that under TCP, everything from machine A is considered one gigantic message. If the process at A, for instance, executes three network writes of 100 bytes each, it is considered one 300-byte message. And though that message may be split into pieces along the way to Z, the piece size will almost certainly not be 100 bytes, nor will the number of pieces likely be three.

This makes writing the program on the Z side more complicated. Say for instance A wishes to send four lines of text, and suppose the program at B knows it will be sent four lines of text. Under UDP, it would be natural to have the program at A send the data as four network writes, and the program at B would do four network reads.

But under TCP, no matter how many or few network writes the program at A does, the program at B will not know how many network reads to do. So, it must keep reading in a **while** loop until it receives a message saying that A has nothing more to send and has closed the connection. This makes the program at Z harder to write. For example, that program must break the data into lines on its own. (I'm talking about *user* programs here; in other words, TCP means more trouble for *you*.)

¹On UNIX machines, a list of these is available in `/etc/services`. You cannot start a server at these ports unless you are acting with root privileges.

²The **bandwidth** is the number of bits per second which can be sent. It should not be confused with **latency**, which is the end-to-end transit time for a message.

1.5 Clients and Servers

Connections are not completely symmetric.³ Instead, a connection consists of a **client** and a **server**. The latter sits around waiting for requests for connections, while the former makes such a request.

When you surf the Web, say to **http://www.google.com**, your Web browser is a client. The program you contact at Google is a server. When a server is run, it sets up business at a certain port, say 80 in the Web case. It then waits for clients to contact it. When a client does so, the server will usually assign a new port, say 56399, specifically for communication with that client, and then resume watching port 80 for new requests.

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**:

```
1 # simple illustration client/server pair; client program sends a string
2 # to server, which echoes it back to the client (in multiple copies),
3 # and the latter prints to the screen
4
5 # this is the server
6
7 import socket
8 import sys
9
10 # create a socket
11 s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
12
13 # associate the socket with a port
14 host = '' # can leave this blank on the server side
15 port = int(sys.argv[1])
16 s.bind((host, port))
17
18 # accept "call" from client
19 s.listen(1)
20 conn, addr = s.accept()
21 print 'client is at', addr
22
23 # read string from client (assumed here to be so short that one call to
24 # recv() is enough), and make multiple copies (to show the need for the
25 # "while" loop on the client side)
26
27 data = conn.recv(1000000)
28 data = 10000 * data # concatenate data with itself 999 times
29
30 # wait for the go-ahead signal from the keyboard (to demonstrate that
31 # recv() at the client will block until server sends)
32 z = raw_input()
33
34 # now send
35 conn.send(data)
36
37 # close the connection
38 conn.close()
```

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

³I'm speaking mainly of TCP here, but it mostly applies to UDP too.

```

1 # simple illustration client/server pair; client program sends a string
2 # to server, which echoes it back to the client (in multiple copies),
3 # and the latter prints to the screen
4
5 # this is the client
6
7 import socket
8 import sys
9
10 # create a socket
11 s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
12
13 # connect to server
14 host = sys.argv[1] # server address
15 port = int(sys.argv[2]) # server port
16 s.connect((host, port))
17
18 s.send(sys.argv[3]) # send test string
19
20 # read echo
21 i = 0
22 while(1):
23     data = s.recv(1000000) # read up to 1000000 bytes
24     i += 1
25     if (i < 5): # look only at the first part of the message
26         print data
27     if not data: # if end of data, leave loop
28         break
29     print 'received', len(data), 'bytes'
30
31 # close the connection
32 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 discussed earlier.

You should run this client/server pair before reading further.⁴ 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
```

(Make sure to start the server before you start the client!)

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.

2.1 Analysis of the Server Program

Let's look at the details of the server.

⁴The 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.

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. This is very much analogous to a file handle or file descriptor in applications involving files. Internally it is a pointer to information about our connection (not yet made) to an application on another machine. Again, at this point, it is merely a placeholder. We have not undertaken any network actions yet.

The two arguments state that we wish to the socket to be an Internet socket (**socket.AF_INET**), and that it will use TCP (**socket.SOCK_STREAM**), rather than UDP (**socket.SOCK_DGRAM**). Note that the constants used in the arguments are attributes of the module **socket**, so they are preceded by ‘socket.’; in C/C++, the analog is the **#include** file.

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. Note that **bind()** takes a single argument consisting of a two-element tuple, rather than two scalar arguments.

Line 19: The **listen()** method tells the OS that if any messages come in from the Internet specifying port 2000, then they should be considered to be requesting a connection to this socket.

The method’s argument tells the OS how many connection requests from remote clients to allow to be pending at any give time for port 2000. The argument 1 here tells the OS to allow only 1 pending connection request at a time.

We only care about one connection in this application, so we set the argument to 1. If we had set it to, say 5 (which is common), the OS would allow one active connection for this port, and four other pending connections for it. If a fifth pending request were to come it, it would be rejected, with a “connection refused” error.

That is about all **listen()** really does.

We term this socket to be consider it a **listening socket**. That means its sole purpose is to accept connections with clients; it is usually not used for the actual transfer of data back and forth between clients and the server.⁶

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 assign that client an **ephemeral port**, which is a port number 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. The OS on the server machine sets up a new socket,

⁵This could be another invocation of our server program, or a different program entirely. You could check this “by hand,” by running the UNIX **netstat** command (Windows has something similar), but it would be better to have your program do it, using a Python **try/except** construct.

⁶It could be used for that purpose, if our server only handles one client at a time.

⁷Which, technically, could be the same machine.

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

You might wonder why there are separate listening and connected sockets. Typically a server will simultaneously be connected to many clients. So it needs a separate socket for communication with each client. (It then must either set up a separate thread for each one, or use nonblocking I/O. More on the latter below.)

All this releases **accept()** from its blocking status, and it returns a two-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.⁸

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. It is traditionally set at 1024.

It is absolutely crucial, though, to keep in mind how TCP works in this regard. To review, 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. And 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. 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:

```
>>> 3*'abc'  
'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; it is merely a stalling mechanism.

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

Line 38: The server closes the connection. At this point, the sending of the giant message to the client is complete.⁹ The closing of the connection will be sensed by the client, as discussed below.

⁸When I say "we," I mean "we, the authors of this server program." That information may be optional for us, though obviously vital to the OS on the machine where the server is running. The OS also needs to know the client's ephemeral port, while "we" would almost never have a need for that.

⁹However, that doesn't necessarily mean that the message has arrived at the client yet, nor even that the message has even left the server's machine yet. See below.

2.2 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.

Again, the connection itself is not physical. It merely is an agreement made between the server and client to exchange data, in agreed-upon chunk sizes, etc.

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 chunks. 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 38.¹⁰

Note:

- `recv()` will block if no data has been received¹¹ but the connection has not been closed
- `recv()` will return an empty string when the connection is closed

3 Role of the OS

3.1 Basic Operation

As is the case with file functions, e.g. `os.open()`, the functions `socket.socket()`, `socket.bind()`, etc. are all wrappers to OS system calls.

The Python `socket.send()` calls the OS `send()`. The latter copies the data (which is an argument to the function) to the OS' buffer. Again, assuming we are using TCP, the OS will break the message into pieces before putting the data in the buffer. Characters from the latter are at various times picked up by the Ethernet card's device driver and sent out onto the network.

When a call to `send()` returns, that simply means that at least part of the given data has been copied from the application program to the OS' buffer. It does not mean that ALL of the data has been copied to the buffer, let alone saying that the characters have actually gotten onto the network yet, let alone saying they have reached the receiving end's OS, let alone saying they have reached the receiving end's application program. The OS will tell us how many bytes it accepted from us to send out onto the network, via the return value from the call to `send()`. (So, technically even `send()` should be in a loop, which iterates until all of our bytes have been accepted. See below.)

¹⁰This would also occur if `conn` were to be garbage-collected when its scope ended, including the situation in which the server exits altogether.

¹¹This will not be the case if the socket is `nonblocking`. More on this in Section 6.1.

The OS at the receiving end will receive the data, check for errors and ask the sending side to retransmit an erroneous chunk, piece the data back to together and place it in the OS' buffer. Each call to `recv()` by the application program on the receiving end will pick up whatever characters are currently in the buffer (up to the number specified in the argument to `recv()`).

3.2 How the OS Distinguishes Between Multiple Connections

When the server accepts a connection from a client, the connected socket will be given the same port as the listening socket. So, we'll have two different sockets, both using the same port. If the server has connections open with several clients, and the associated connected 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 a connection is defined by five numbers: The server port; the client (ephemeral) port; the server IP address; the client IP address; and the protocol (TCP or UDP). Different clients will usually have different Internet addresses, so that is a distinguishing aspect. But even more importantly, 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 works out.

4 The `sendall()` Function

We emphasized earlier why a call to `recv()` should be put in a loop. One might also ask whether `send()` should be put in a loop too.

Unless the socket is nonblocking, `send()` will block until the OS on our machine has enough buffer space to accept at least some of the data given to it by the application program via `send()`. When it does so, the OS will tell us how many bytes it accepted, via the return value from the call to `send()`. The question is, is it possible that this will not be all of the bytes we wanted to send? If so, we need to put `send()` in a loop, e.g. something like this, where we send a string `w` via a socket `s`:

```
while(len(w) > 0):
    ns = s.send(w) # ns will be the number of bytes sent
    w = w[ns:] # still need to send the rest
```

The best reference on TCP/IP programming (*UNIX Network Programming*, by Richard Stevens, pub. Prentice-Hall, vol. 1, 2nd ed., p.77) says that this problem is “normally” seen only if the socket is nonblocking. However, that was for UNIX, and in any case, the best he seemed to be able to say was “normally.” To be fully safe, one should put one's call to `send()` inside a loop, as shown above.

But as is often the case, Python recognizes that this is such a common operation that it should be automated. Thus Python provides the `sendall()` function. This function will not return until the entire string has been sent (in the sense stated above, i.e. completely copied to the OS' buffer).

The function `sendall()` should be used only with blocking sockets.

5 Sending Lines of Text

5.1 Remember, It's Just One Big Byte Stream, Not “Lines”

As discussed 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()`. This becomes a nuisance for the programmer.

In our example above, 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.

5.2 The Wonderful `makefile()` Function

As mentioned, if you are transferring text data between the client and server (in either direction), you've got to piece together each line on your own, a real pain. Not only might you get only part of a line during a receive, you might get part of the *next* line, which you would have to save for later use with reading the next line.¹² But Python allows you to avoid this work, by using the method `socket.makefile()`.

Python has the notion of a *file-like object*. This is a byte stream that you can treat as a “file,” thinking of it as consisting of “lines.” For example, we can invoke `readlines()`, a file function, on the standard input:

```
>>> import sys
>>> w = sys.stdin.readlines() # ctrl-d to end input
moon, sun
and
stars
>>> w
['moon, sun\n', 'and\n', 'stars\n']
```

Well, `socket.makefile()` allows you to do this with sockets, as seen in the following example.

Here we have a server which will allow anyone on the Internet to find out which processes are running on the host machine—even if they don't have an account on that machine.¹³ See the comments at the beginning of the programs for usage.

Here is the client:

```
1 # wps.py
2
```

¹²One way around this problem would be to read one byte at a time, i.e. call `recv()` with argument 1. But this would be very inefficient, as system calls have heavy overhead.

¹³Some people have trouble believing this. How could you access such information without even having an account on that machine? Well, the server is willing to give it to you. Imagine what terrible security problems we'd have if the server allowed one to run any command from the client, rather than just `w` and `ps`.

```

3 # client for the server for remote versions of the w and ps commands
4
5 # user can check load on machine without logging in (or even without
6 # having an account on the remote machine)
7
8 # usage:
9
10 # python wps.py remotehostname port_num {w,ps}
11
12 # e.g. python wps.py nimbus.org 8888 w would cause the server at
13 # nimbus.org on port 8888 to run the UNIX w command there, and send the
14 # output of the command back to the client here
15
16 import socket,sys
17
18 def main():
19
20     s = socket.socket(socket.AF_INET,socket.SOCK_STREAM)
21     host = sys.argv[1]
22     port = int(sys.argv[2])
23     s.connect((host,port))
24
25     # send w or ps command to server
26     s.send(sys.argv[3])
27
28     # create "file-like object" flo
29     flo = s.makefile('r',0) # read-only, unbuffered
30     # now can call readlines() on flo, and also use the fact that
31     # that stdout is a file-like object too
32     sys.stdout.writelines(flo.readlines())
33
34 if __name__ == '__main__':
35     main()

```

And here is the server:

```

1 # svr.py
2
3 # server for remote versions of the w and ps commands
4
5 # user can check load on machine without logging in (or even without
6 # having an account on the remote machine)
7
8 # usage:
9
10 # python svr.py port_num
11
12 import socket,sys,os
13
14 def main():
15
16     ls = socket.socket(socket.AF_INET,socket.SOCK_STREAM);
17     port = int(sys.argv[1])
18     ls.bind(('', port))
19
20     while (1):
21         ls.listen(1)
22         (conn, addr) = ls.accept()
23         print 'client is at', addr
24         # get w or ps command from client
25         rc = conn.recv(2)
26         # run the command in a Unix-style pipe
27         ppn = os.popen(rc)
28         # ppn is a "file-like object," so can apply readlines()
29         rl = ppn.readlines()
30         # create a file-like object from the connection socket

```

```

31     flo = conn.makefile('w',0) # write-only, unbuffered
32     # write the lines to the network
33     flo.writelines(rl[:-1])
34     # clean up
35     # must close both the socket and the wrapper
36     flo.close()
37     conn.close()
38
39 if __name__ == '__main__':
40     main()

```

5.3 Getting the Tail End of the Data

A common bug in network programming is that most of a data transfer works fine, but the receiving program hangs at the end, waiting to receive the very last portion of the data. The cause of this is that, for efficiency reasons, the TCP layer at the sending end generally waits until it accumulates a “large enough” chunk of data before it sending out. As long as the sending side has not closed the connection, the TCP layer there assumes that the program may be sending more data, and TCP will wait for it.

The simplest way to deal with this is for the sending side to close the connection. Its TCP layer then knows that it will be given no more data by the sending program, and thus TCP needs to send out whatever it has. This means that both client and server programs must do a lot of opening and closing of socket connections, which is inefficient.¹⁴

Also, if you are using `makefile()`, it would be best to not use `readlines()` directly, as this may cause the receiving end to wait until the sender closes the connection, so that the “file” is complete. It may be safer to do something like this:

```

# flo is an already-created file-like object from calling makefile()
for line in flo:
    ...

```

This way we are using `flo` as an iterator, which will not require all lines to be received before the loop is started.

An alternative is to use UDP instead of TCP. With UDP, if you send an n-byte message, it will all be send immediately, in one piece, and received as such. However, you then lose the reliability of TCP.¹⁵

6 Dealing with Asynchronous Inputs

In many applications a machine, typically a server, will be in a position in which input could come from several network sources, without knowing which one will come next. One way of dealing with that is to

¹⁴Yet such inefficiency is common, and is used for example in the HTTP (i.e. Web access) protocol. Each time you click the mouse on a given Web page, for instance, there is a new connection made. It is this lack of **state** in the protocol that necessitates the use of **cookies**. Since each Web action involves a separate connection, there is no “memory” between the actions. This would mean, for example, that if the Web site were password-protected, the server would have to ask you for your password at every single action, quite a nuisance. The workaround is to have your Web browser write a file to your local disk, recording that you have already passed the password test.

¹⁵TCP does error checking, including checking for lost packets. If the client and server are multiple hops apart in the Internet, it’s possible that some packets will be lost when an intermediate router has buffer overflow. If TCP at the receiving end doesn’t receive a packet by a certain **timeout** period, it will ask TCP at the sending end to retransmit the packet. Of course, we could do all this ourselves in UDP by adding complexity to our code, but that would defeat the purpose.

take a threaded approach, with the server having a separate thread for each possible client. Another way is to use nonblocking sockets.

6.1 Nonblocking Sockets

Note our statement that `recv()` blocks until either there is data available to be read or the sender has closed the connection holds only if the socket is in blocking mode. That mode is the default, but we can change a socket to nonblocking mode by calling `setblocking()` with argument 0.¹⁶

One calls `recv()` in nonblocking mode as part of a `try/except` pair. If data is available to be read from that socket, `recv()` works as usual, but if no data is available, an exception is raised. While one normally thinks of exceptions as cases in which a drastic execution error has occurred, in this setting it simply means that no data is yet available on this socket, and we can go to try another socket or do something else.

Why would we want to do this? Consider a server program which is connected to multiple clients simultaneously. Data will come in from the various clients at unpredictable times. The problem is that if we were to simply read from each one in order, it could easily happen that `read()` would block while reading from one client while data from another client is ready for reading. Imagine that you are the latter client, while the former client is out taking a long lunch! You'd be unable to use the server all that time!

One way to handle this is to use threads, setting up one thread for each client. Indeed, I have an example of this in my Python threads tutorial, at <http://heather.cs.ucdavis.edu/~matloff/Python/PyThreads.pdf>. But threads programming can be tricky, so one may turn to the alternative, which is to make the client sockets nonblocking.

The example below does nothing useful, but is a simple illustration of the principles. Each client keeps sending letters to the server; the server concatenates all the letters it receives, and sends the concatenated string back to a client whenever the client sends a character.

In a sample run of these programs, I started the server, then one client in one window, then another client is another window. (For convenience, I was doing all of this on the same machine, but a better illustration would be to use three different machines.) In the first window, I typed 'a', then 'b', then 'c'. Then I moved to the other window, and typed 'u', 'u', 'v' and 'v'. I then went back to the first window and typed 'd', etc. I ended each client session by simply hitting Enter instead of typing a letter.

Here is what happened at the terminal for the first client:

```
1 % python strclnt.py localhost 2000
2 enter a letter:a
3 a
4 enter a letter:b
5 ab
6 enter a letter:c
7 abc
8 enter a letter:d
9 abcuuvvd
10 enter a letter:e
11 abcuuvvdwwe
12 enter a letter:
```

Here is what happened at the terminal for the second client:

¹⁶As usual, this is done in a much cleaner, easier manner than in C/C++. In Python, one simple function call does it.

```

1 % python strclnt.py localhost 2000
2 enter a letter:u
3 abcu
4 enter a letter:u
5 abcuu
6 enter a letter:v
7 abcuuv
8 enter a letter:v
9 abcuuvv
10 enter a letter:w
11 abcuuvvdw
12 enter a letter:w
13 abcuuvvdww
14 enter a letter:

```

Here is what happened at the terminal for the server:

```

1 % python strsvr.py 2000
2 the final value of v is abcuuvvdwwe

```

Note that I first typed at the first client, but after typing 'c', I switched to the second client. After hitting the second 'v', I switched back to the first, etc.

Now, let's see the code. First, the client:

```

1 # strclnt.py: simple illustration of nonblocking sockets
2
3 # two clients connect to server; each client repeatedly sends a letter k,
4 # which the server appends to a global string v and reports it to the
5 # client; k = '' means the client is dropping out; when all clients are
6 # gone, server prints the final string v
7
8 # this is the client; usage is
9 #   python clnt.py server_address server_port_number
10
11 import socket, sys
12
13 s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
14 host = sys.argv[1] # server address
15 port = int(sys.argv[2]) # server port
16 s.connect((host, port))
17
18 while(1):
19     # get letter
20     k = raw_input('enter a letter:')
21     s.send(k) # send k to server
22     # if stop signal, then leave loop
23     if k == '': break
24     v = s.recv(1024) # receive v from server (up to 1024 bytes, assumed
25                     # to come in one chunk)
26     print v
27
28 s.close() # close socket

```

And here is the server:

```

1 # strsvr.py: simple illustration of nonblocking sockets
2
3 # multiple clients connect to server; each client repeatedly sends a
4 # letter k, which the server adds to a global string v and echos back

```

```

5 # to the client; k = '' means the client is dropping out; when all
6 # clients are gone, server prints final value of v
7
8 # this is the server; usage is
9 #   python strsvr.py server_port_number
10
11 import socket, sys
12
13 # set up listening socket
14 lstn = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
15
16 port = int(sys.argv[1])
17 # bind lstn socket to this port
18 lstn.bind(('', port))
19 lstn.listen(5)
20
21 # initialize concatenate string, v
22 v = ''
23
24 # initialize client socket list
25 cs = []
26
27 # in this example, a fixed number of clients, nc
28 nc = 2
29
30 # accept connections from the clients
31 for i in range(nc):
32     (clnt,ap) = lstn.accept()
33     # set this new socket to nonblocking mode
34     clnt.setblocking(0)
35     cs.append(clnt)
36
37 # now loop, always accepting input from whoever is ready, if any, until
38 # no clients are left
39 while (len(cs) > 0):
40     # get next client, with effect of a circular queue
41     clnt = cs.pop(0)
42     cs.append(clnt)
43     # something ready to read from clnt? clnt closed connection?
44     try:
45         k = clnt.recv(1) # try to receive one byte; if none is ready
46                         # yet, that is the "exception"
47         if k == '':
48             clnt.close()
49             cs.remove(clnt)
50         v += k # update the concatenated list
51         clnt.send(v)
52     except: pass
53
54 lstn.close()
55 print 'the final value of v is', v

```

Here we only set the client sockets in nonblocking mode, not the listening socket. However, if we wished to allow the number of clients to be unknown at the time execution starts, rather than a fixed number known ahead of time as in our example above, we would be forced to either make the listening socket nonblocking, or use threads.

Note once again that the actual setting of blocking/nonblocking mode is done by the OS. Python's **setblocking()** function merely makes system calls to make this happen. It should be said, though, that this Python function is far simpler to use than what must be done in C/C++ to set the mode.

6.2 Advanced Methods of Polling

In our example of nonblocking sockets above, we had to “manually” check whether a socket was ready to read. Today most OSs can automate that process for you. The “traditional” way to do this was via a UNIX system call named **select()**, which later was adopted by Windows as well. The more modern way for UNIX is another call, **poll()** (not yet available on Windows). Python has interfaces to both of these system calls, in the **select** module. Python also has modules **asyncore** and **asynchat** for similar purposes. I will not give the details here.

7 Other Libraries

Python has libraries for FTP, HTML processing and so on. In addition, a higher-level library which is quite popular is Twisted.