Introduction to Network Programming with Python

Norman Matloff
University of California, Davis

April 29, 2005

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

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(1000000)
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(1000000)  # read up to 1000000 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.

2.1 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. 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 made any network actions yet. But our calls **bind()** etc. below will result in more and more information being added to the place pointed to by our socket.

The two arguments state that we wish to the socket to be an Internet socket (**socket.AF_INET**), and that it will use the TCP method of organizing data (**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).

A port is merely an ID number, not anything physical. Since there may be many network connections on a machine at one time, we need a way to distinguish between them. The ID number serves this purpose. Port numbers 0-1023, the so-called **well-known ports**, are for standard services such as FTP (port 21), SSH (port 22) and HTTP (port 80).[2] You cannot start a server at these ports unless you are acting with root privileges.

When we call **bind()**, the operating system will first check to see whether port 2000 is already in use by some other process.[3] 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 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.

[2]On UNIX machines, a list of these is available in **/etc/services**.

[3]This 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.

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.[4]

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.[5] 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, 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.

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.[6]

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 discuss 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. 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'
```

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

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

[6]When 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.

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.[7] The closing of the connection will be sensed by the client, as discussed 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.

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.[8]

Note:

- **recv()** will block if no data has been received [9] 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.

[7]However, 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.

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

[9]This will not be the case if the socket is **nonblocking**. More on this in Section 6.

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, via the return value from the call to **send()**.

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. 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 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 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 More on the "Stream" Nature of TCP

5.1 Rememember, It's Just One Big Byte Stream, Not "Lines"

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.

5.2 The Wonderful `makefile()` Function

So, 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. [10] But Python allows you to avoid this work, by using the method **socket.makefile()**.

Recall that 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:

```
>>> 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

[10] One 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.

the host machine—even if they don't have an account on that machine.[11] See the comments at the beginning of the programs for usage.

Here is the client:

```
# wps.py

# client for the server for remote versions of the w and ps commands

# user can check load on machine without logging in (or even without
# having an account on the remote machine)

# usage:

#   python wps.py remotehostname port_num {w,ps}

# e.g. python wps.py nimbus.org 8888 w would cause the server at
# nimbus.org on port 8888 to run the UNIX w command there, and send the
# output of the command back to the client here

import socket,sys

def main():

   # set up connection
   s = socket.socket(socket.AF_INET,socket.SOCK_STREAM)
   host = sys.argv[1]  # server address
   port = int(sys.argv[2])  # server port
   s.connect((host,port))

   # send w or ps command to server, by writing that command to the "file"
   s.send(sys.argv[3])

   # read result as lines from the "file-like object"
   # wrap it to a "file-like object"
   flo = s.makefile('r',0)  # read-only, unbuffered
   # recall that stdout is a file-like object too
   sys.stdout.writelines(flo.readlines())

if __name__ == '__main__':
   main()
```

And here is the server:

```
# svr.py

# server for remote versions of the w and ps commands

# user can check load on machine without logging in (or even without
# having an account on the remote machine)

# usage:

```

[11] Some 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**.

```
#   python svr.py port_num

import socket,sys,os

def main():

   # create listening socket
   ls = socket.socket(socket.AF_INET,socket.SOCK_STREAM);
   port = int(sys.argv[1])
   ls.bind(('', port))

   # enter listening loop
   while (1):
      # accept "call" from client
      ls.listen(1)
      (conn, addr) = ls.accept()
      print 'client is at', addr
      # get and run command from client, then get its output
      rc = conn.recv(2)
      # run the command in a Unix-style pipe
      ppn = os.popen(rc)
      # ppn is a "file-like object," so can apply readlines()
      rl = ppn.readlines()
      # create a file-like object from the connection socket
      flo = conn.makefile('w',0)  # write-only, unbuffered
      flo.writelines(rl[:-1])
      # clean up
      # must close both the socket and the wrapper
      flo.close()
      conn.close()

if __name__ == '__main__':
   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. Yet it 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.[12]

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

[12]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.

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.

We could also use **xreadlines()** for the same purpose, but it is now deprecated.

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.[13]

6 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.[14]

One calls **recv()** in nonblocking mode 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.

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:

[13]TCP 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.

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

```
% python strclnt.py localhost 2000
enter a letter:a
a
enter a letter:b
ab
enter a letter:c
abc
enter a letter:d
abcuuvvd
enter a letter:e
abcuuvvdwwe
enter a letter:
```

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

```
% python strclnt.py localhost 2000
enter a letter:u
abcu
enter a letter:u
abcuu
enter a letter:v
abcuuv
enter a letter:v
abcuuvv
enter a letter:w
abcuuvvdw
enter a letter:w
abcuuvvdww
enter a letter:
```

Here is what happened at the terminal for the server:

```
% python strsvr.py 2000
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:

```
# strclnt.py:  simple illustration of nonblocking sockets

# two clients connect to server; each client repeatedly sends a letter k,
# which the server appends to a global string v and reports it to the
# client; k = ’’ means the client is dropping out; when all clients are
# gone, server prints the final string v

# this is the client; usage is
#    python clnt.py server_address server_port_number

import socket, sys

s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
host = sys.argv[1]  # server address
```

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

while(1):
   # get letter
   k = raw_input('enter a letter:')
   s.send(k)  # send k to server
   # if stop signal, then leave loop
   if k == '': break
   v = s.recv(1024)  # receive v from server (up to 1024 bytes)
   print v

s.close() # close socket
```

And here is the server:

```
# strsvr.py:  simple illustration of nonblocking sockets

# multiple clients connect to server; each client repeatedly sends a
# letter k, which the server adds to a global string v and echos back
# to the client; k = '' means the client is dropping out; when all
# clients are gone, server prints final value of v

# this is the server; usage is
#    python strsvr.py server_port_number

import socket, sys

# set up listening socket
lstn = socket.socket(socket.AF_INET, socket.SOCK_STREAM)

port = int(sys.argv[1])
# bind lstn socket to this port
lstn.bind(('', port))
lstn.listen(5)

# initialize total, v
v = ''

# initialize client socket list
cs = []

# in this example, a fixed number of clients, nc
nc = 2

# accept connections from the clients
for i in range(nc):
   (clnt,ap) = lstn.accept()
   # set this new socket to nonblocking mode
   clnt.setblocking(0)
   cs.append(clnt)

# now loop, always accepting input from whoever is ready, if any, until
# no clients are left
while (len(cs) > 0):
   # get next client, with effect of a circular queue
   clnt = cs.pop(0)
   cs.append(clnt)
   # can read from clnt? clnt closed connection?
   try:
      k = clnt.recv(1)
      if k == '':
         clnt.close()
         cs.remove(clnt)
      v += k
      clnt.send(v)
   except:  pass

lstn.close()
print 'the final value of v is', v
```

Here we only set the client sockets in nonblocking mode, not the listening socket. However, we could have the latter nonblocking too. And 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.

7 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.