Chapter 8

Networking with Python

From the beginning of the Internet, applications use network functionality through the **socket interface**. This interface exists in most languages and provides an easy way to send and receive data from a different host in the network. This chapter covers the basics of how a programmer uses the socket interface to implement distributed applications.

The TCP/IP protocols suite, the core of internet protocols, implements two data delivery services:

- TCP: A reliable and controlled protocol for transmission of data. TCP guarantees that all bits arrive at the destination and that the rate of transmission adapts to the congestion status of the network.

- UDP: An unreliable and uncontrolled protocol for transmission. UDP does not guarantee the delivery of every bit. Moreover, it is up to the application to decide the rate of transmission

8.1 TCP sockets

The structure of communications can be seen in Figure 8.1.

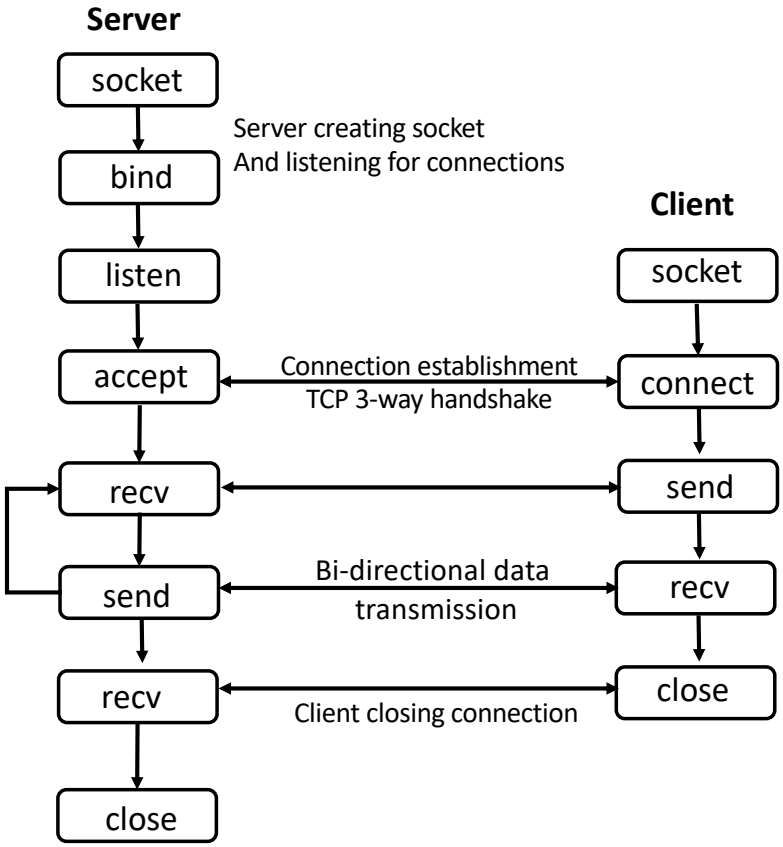

Figure 8.1: Socket structure

In Figure 8.2 we can see the code for a server implemented with sockets. At the core of the code is the **socket** structure. The server creates a socket structure, binds it to a host and port and then starts to listen for connections.

Once a connection arrives it uses the **conn** variable to receive and send data. For this it uses the recv and sendall methods.

```
#!/usr/bin/env python3

import socket

HOST = '127.0.0.1'  # Standard loopback interface address (localhost)
PORT = 65432        # Port to listen on (non-privileged ports are > 1023)

with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as s:
    s.bind((HOST, PORT))
    s.listen()
    conn, addr = s.accept()
    with conn:
        print('Connected by', addr)
        while True:
            data = conn.recv(1024)
            if not data:
                break
            conn.sendall(data)
```

Figure 8.2: Server code with sockets

In Figure **??** we can see the corresponding client. The **connect** method establishes a connection with the server. After the connection is established the client also uses the recv and sendall method to exchange data.

```
#!/usr/bin/env python3

import socket

HOST = '127.0.0.1'  # The server's hostname or IP address
PORT = 65432        # The port used by the server

with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as s:
    s.connect((HOST, PORT))
    s.sendall(b'Hello, world')
    data = s.recv(1024)

print('Received', repr(data))
```

Figure 8.3: Client code with sockets

8.2 UDP sockets

A different communication approach uses UDP. No connection Establishment takes place. Each UDP packet is its own entity. In Figure 8.4 we can see the UDP server. Differences to note:

- The type of socket is **SOCK_DGRAM**
- The server does not listen for connections. It blocks in d = s.recvfrom(1024) and receives a UDP datagram at a time.
- VERY IMPORTANT: clients and servers control the sending rate from the application. The operating system does not pace the data like in TCP. You can easily flood a network with packets.

```
 '''
Simple udp socket server
'''

import socket
import sys

HOST = ''
PORT = 8888

# Datagram (udp) socket

try:
        s=socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
        print 'Socket created'
except socket.error, msc :
        print 'Failed to create socket. Error Code :' +str(msg[0])+' Message '+ msg[1]
        sys.exit()

# Bind socket to local host and port
try:
        s.bind((HOST,PORT))
except socket.error, msg:
        print 'Bind failed. Error Code :' +str(msg[0]) + 'Message' +msg[1]
        sys.exit()

print 'Socket bind complete'

# now keep talking with the client
while 1:
        # receive data from client (data, addr)
        d = s.recvfrom(1024)
        data=d[0]
        addr = d[1]

        if not data:
                break
        reply = 'OK...'+data

        s.sendto(reply, addr)
        print 'Message[' +addr[0] + ':' + str(addr[1]) + '] - ' +data.strip()

s.close()
```

Figure 8.4: UDP server

In Figure 8.6 we can see the UDP client

```
 '''
Simple udp socket client
'''

import socket
import sys

HOST = ''
PORT = 8888

# Datagram (udp) socket

try:
        s=socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
        print 'Socket created'
except socket.error, msc :
        print 'Failed to create socket.'
        sys.exit()


host= 'localhost'
port=8888

while(1):
        msg=raw_input('Enter message to send')

        try:
                # Set the whole string
                s.sendto(msg, (host, port))

                # receive data from client (data, addr)
                d= s.recvfrom(1024)
                reply = d[0]
                addr = d[1]

                print 'Server reply : ' + reply

        except socket.error, msg:
                print 'Error Code: ' + str(msg[0]) + ' Message ' + msg[1]
                sys.exit()
```

Figure 8.5: UDP client

```
# Bind socket to local host and port
try:
        s.bind((HOST,PORT))
except socket.error, msg:
        print 'Bind failed. Error Code :' +str(msg[0]) + 'Message' +msg[1]
        sys.exit()

print 'Socket bind complete'

# now keep talking with the client
while 1:
        # receive data from client (data, addr)
        d = s.recvfrom(1024)
        data=d[0]
        addr = d[1]

        if not data:
                break
        reply = 'OK...'+data

        s.sendto(reply, addr)
        print 'Message[' +addr[0] + ':' + str(addr[1]) + '] - ' +data.strip()

s.close()
```

Figure 8.6: UDP client (continued)

8.3 Multi-threads

The previous examples have a significant problem: they can only deal with one client simultaneously. To allow for the same server to deal with more than one client at the same time a program can use **threads**.

In Figure 8.7 we can see how we use threads in a general program.

```
      #!/usr/bin/env python

import threading
import time


class MyThread(threading.Thread):
    def run(self):
        print("{} started!".format(self.getName()))
        # "Thread-x started!"
        time.sleep(1)
        # Pretend to work for a second
        print("{} finished!".format(self.getName()))
        # "Thread-x finished!"

if __name__ == '__main__':
    for x in range(4):
    # Four times...
        mythread = MyThread(name = "Thread-{}".format(x + 1))
        # ...Instantiate a thread and pass a unique ID to it
        mythread.start()
        # ...Start the thread
        time.sleep(.9)
        # ...Wait 0.9 seconds before starting another
```

Figure 8.7: Threads

Figure 8.9 shows how we can incorporate the threads concept in a server. Things to note:

- ...

```
 # import socket programming library
import socket

# import thread module
from _thread import *
import threading

print_lock = threading.Lock()

# thread fuction
def threaded(c):
    while True:

        # data received from client
        data = c.recv(1024)
        if not data:
            print('Bye')

            # lock released on exit
            print_lock.release()
            break

        # reverse the given string from client
        data = data[::-1]

        # send back reversed string to client
        c.send(data)

    # connection closed
    c.close()
```

Figure 8.8: Multi-threaded server

```
def Main():
    host = ""

    # reverse a port on your computer
    # in our case it is 12345 but it
    # can be anything
    port = 12345
    s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    s.bind((host, port))
    print("socket binded to post", port)

    # put the socket into listening mode
    s.listen(5)
    print("socket is listening")

    # a forever loop until client wants to exit
    while True:

        # establish connection with client
        c, addr = s.accept()

        # lock acquired by client
        print_lock.acquire()
        print('Connected to :', addr[0], ':', addr[1])

        # Start a new thread and return its identifier
        start_new_thread(threaded, (c,))
    s.close()


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

Figure 8.9: Multi-threaded server (continued

Although we don't necessarily needs to change our client code, the code in Figure 8.10 shows small changes.

```
  # Import socket module
import socket


def Main():
    # local host IP '127.0.0.1'
    host = '127.0.0.1'

    # Define the port on which you want to connect
    port = 12345

    s = socket.socket(socket.AF_INET,socket.SOCK_STREAM)

    # connect to server on local computer
    s.connect((host,port))

    # message you send to server
    message = "SNS is the best course ever !"
    while True:

        # message sent to server
        s.send(message.encode('ascii'))

        # messaga received from server
        data = s.recv(1024)

        # print the received message
        # here it would be a reverse of sent message
        print('Received from the server :',str(data.decode('ascii')))

        # ask the client whether he wants to continue
        ans = input('\nDo you want to continue(y/n) :')
        if ans == 'y':
            continue
        else:
            break
    # close the connection
    s.close()

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

Figure 8.10: Client for Multi-threaded server

8.4 Exercises

1. Based on the multi-thread TCP server code presented here build an ISP automated costumer support *bot* that dialogues with a user. The bot asks introduces itself to the user and then asks questions to try to diagnose the problem.

2. Do the same exercise using UDP

3. Build a TCP client that connects to a web server, issues an HTTP GET command, retrieves data and calculates what was the throughput. You have to write in the socket

```
    GET / HTTP/1.1
Host: www.example.com
```

   Note that you you have to send a newline after the second line.