A TCP Client-Server Program

In the last chapter, we studied TCP theory in detail. Now, we'll look at how we can code up TCP programs in Python.

WE'LL COVER THE FOLLOWING

Introduction

There are a few key points to be noted about TCP programs:

- TCP `connect()` calls induce full TCP three-way handshakes! As we saw in the last chapter, **three-way handshakes can fail and so can `connect()` calls**.
- A **new socket gets created for every new connection** in a TCP architecture.
- **One socket** on TCP servers is dedicated to **continuously listen** for **new** incoming connections.
- When a connection is successful, that listening socket creates a **new socket exclusively for that connection**.
- When the **connection terminates**, the associated **socket** gets **deleted**.
- Every socket, and hence **each connection, is identified by the unique 4-tuple: `(local_ip, local_port, remote_ip, remote_port)`**. All incoming TCP packets are examined to see whether their source and destination

addresses belong to any such currently connected sockets.

- Unlike UDP, **TCP segments will be delivered as long as the sender and receiver are connected by a path and they are both live**.
- A sending TCP entity might **split TCP segments into packets** and so, receiving TCP entities would have to reassemble them. This is unlikely in our small program but happens all the time in the real world. So we need to take care of when there is data leftover in the buffer to send or to receive after one call.

A TCP Server & Client Program

```
import argparse, socket

def recvall(sock, length):
    data = b''
    while len(data) < length:
        more = sock.recv(length - len(data))
        if not more:
            raise EOFError('was expecting %d bytes but only received'
                           ' %d bytes before the socket closed'
                           % (length, len(data)))
        data += more
    return data

def server(port):
    sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    sock.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
    sock.bind(('127.0.0.1', port))
    sock.listen(1)
    print('Listening at', sock.getsockname())
    while True:
        print('Waiting for a new connection')
        sc, sockname = sock.accept()
        print('Connection from', sockname)
        print('  Socket name:', sc.getsockname())
        print('  Socket peer:', sc.getpeername())
        message = recvall(sc, 16)
        print('  message from client:', repr(message))
        sc.sendall(b'Goodbye, client!')
        sc.close()
        print('  Closing socket')

def client(port):
    host = '127.0.0.1'
    sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    sock.connect((host, port))
    print('Client has been assigned the socket: ', sock.getsockname())
    sock.sendall(b'Greetings, server')
    reply = recvall(sock, 16)
    print('Server: ', repr(reply))
    sock.close()
```

```
if __name__ == '__main__':
    choices = {'client': client, 'server': server}
    parser = argparse.ArgumentParser(description='Send and receive over TCP')
    parser.add_argument('role', choices=choices, help='which role to play')
    parser.add_argument('-p', metavar='PORT', type=int, default=3000, help='TCP port (default
    args = parser.parse_args()
    function = choices[args.role]
    function(args.p)
```

As you can see, the client program is pretty much the same as a UDP client program. There are a few key differences which we will explore now:

Handling Fragmentation

`sendall()`

One of three things may happen at every `send()` call:

1. All the data you passed to it gets sent immediately.
2. None of the data gets transmitted.
3. *Part* of the data gets transmitted.

The `send()` function returns the length of the number of bytes it successfully transmitted, which can be used to check if the entire segment was sent.

Here's what code to handle partial or no transmission would look like:

```
bytes_sent = 0 # No bytes initially sent
while bytes_sent < len(message): # If number of bytes sent is less than the amount of data
  message_left = message[bytes_sent:] # Indexing and storing the part of the message remainir
  bytes_sent += sock.send(message_left) # Sending remaining message
```

Luckily, Python has its own implementation of this in a function called `sendall()`. It ensures that all of the data gets sent. Check out line **37** in the TCP server-client program above to see how it is used.

`recvall()`

Unfortunately, no equivalent to automatically handle fragmentation exists for the receiving end. Hence, we'd have to cater for the cases when:

- Part of the sent data arrives
- None of the sent data arrives

We do this by defining a function called `rcvall()` on **line 3**.

That’s the end of this chapter. In the next one, we’ll start on the network layer.