Lab: Client server communication using TCP

A simple server and client

We start with a simple TCP server and client. In this case, code for both are in the same file `tcp_simple.py`. First, look over the code for the server. It takes two parameters: *interface* and *port*. They are the network interface and port that the server will be listening at, meaning that the server will only accept incoming connections over that specific interface and port.

When the interface is provided, it could be the specific IP address of the server, or it could be the double quotes: `""`. If it's a specific IP address, for example, the external IP address of the server, then if a client tries to connect to it on the local machine, the server will not accept the connection.

Alternatively, the double quotes tell the server to accept connections on any network interface, so remote clients and local clients will be able to connect to the server.

It's important to include the line:

```
serverSock.setsockopt(SOL_SOCKET, SO_REUSEADDR, 1)
```

because without it, if the server program is executed multiple times too quickly, it could cause the error of `" socket.error: [Errno 98] Address already in use "`. An earlier execution has caused the socket to be in a `TIME_WAIT` state and the socket cannot be immediately reused. But if the `SO_REUSEADDR` socket flag is set, then even if the socket is in the `TIME_WAIT` state, it can be reused without waiting for its timeout to expire.

The program `tcp_simple.py` takes three arguments: role (client or server), host (IP address server is accepting connections at), port (that server is listening at). Suppose the server program is running on a machine with external IP address 10.0.2.15 and you want to set it to listen at port 1234, then you can execute

```
python tcp_simple.py server 10.0.2.15 1234
```

or

```
python tcp_simple.py server 0.0.0.0 1234
```

Both would work for the client:

```
$ python tcp_simple.py client 10.0.2.15 1234
```

Differences from UDP

The client is almost the same as the UDP client, it creates a socket, then calls `connect()` passing to it the address of the server. However there are important differences.

The TCP `connect()` is very different from the UDP one. It doesn't simply assigns the provided address to the destination in subsequent packets sent out. It does a lot more than that. It's a more complicated network operation that begins with the three-way handshake that establishes the persistent connection between client and server. Try to run the client without running the server first, the `connect()` will fail and give a connection refused error.

With this TCP connection, you don't have to worry about dropped packets, because the TCP takes care of reliable packet transmission by detecting dropped packets and retransmitting them. They are all guaranteed to arrive correctly and in order.

In UDP, data is explicitly put in a datagram packet and sent out. If it's too much data for one datagram, you are responsible for dividing it into several datagrams and send those out. UDP treats data as discrete units. In contrast, TCP sees incoming and outgoing data as *streams* with no pre-defined demarcations.

In UDP, `send()` and `recv()` mean 'send or receive this datagram'. Each datagram is either all received or none of it is, it's never the case that part of a datagram is received but not the rest. However, TCP could divide its data stream into multiple packets of different sizes, transmit them and then reassemble them at the receiver. This is of course practically impossible with such a tiny message of only 16 octets in our example program.

When the program calls (the TCP) `send()`, it goes to the operating system network stack, and three possibilities may occur:

1. The entirety of data succeeds in being put into the network card's outgoing buffer, which will be transmitted, and the call returns immediately.
2. The buffer is full, none of the data can be accepted, the `send()` is blocked and pauses the program.
3. The buffer has some room for part of the data, the call returns with the size of the data put in the buffer, with the remaining data untransmitted.

In case of the third scenario, it is necessary to check the value returned by the call to `send()`. If only part of the data was sent, then you must call `send()` again with the remaining data. This should be repeated in a loop until all the data has been sent.

Python conveniently provides another method `sendall()` that does this on your behalf, see the call to this method in the server code.

However, there is no equivalent `receiveall()` provided. Since the receiving side also face the same three possible scenarios outline above, we need to put `recv()` in a loop to make sure all the data has been received, see this loop in the client code.

(Python probably chose to not provide the receive all equivalent because the receiving side logic is usually complicated and tailored very specifically to the needs of the application to begin with, so it might as well leave the tailoring to the application developer entirely.)

Per connection socket

The TCP server is very different from UDP. It reserves a *listening* socket just for constantly waiting for requests for connection from clients. The port for this listening socket is publicized so clients know which port to send the initial request to. Once a connection request is received on the listening port, another socket is created just for this particular connection, named `connSock` in the code. The new socket is at a port number assigned by the OS, and this is the gate to the persistent TCP connection between server and this client. Every client kicks off another socket/connection.

As can be seen in the server code, `getsockname()` is called by the listening and connection sockets to find the TCP port number used.

TCP server and client that could deadlock

The server and client in the file `tcp_uppercase.py` do a little more than the simple ones seen earlier. The client sends a specific number of bytes (provided by user as a command-line argument) in 16-byte messages to the server, the server capitalizes all the characters and sends them back to the client. The server reads and processes small chunks of 1024 bytes at a time, see the call `recv(1024)` in the server code. Run server first: `python tcp_uppercase.py server "" 1234 64`, then run the client: `python tcp_uppercase.py`

`client 10.0.2.15 1234 64`.

The first argument tells the program which role, client or server, to run as. The second and third arguments are the IP address and port number of the server (listening socket). The fourth argument is the number of bytes sent by the client to server.

If you try larger amounts of data, such as 1 megabytes: `python tcp_uppercase.py client 10.0.2.15 1234 1000000`, it will still work correctly.

But if you try an even larger number like 1 gigabytes: `python tcp_uppercase.py client 10.0.2.15 1234 1000000000`, the server and client will eventually freeze. In my execution, the server froze at the message printed on screen: `8001648 bytes already capitalized`, and the client froze when the screen reads: `13871680 bytes sent`. In your execution, you will see different numbers. Note that the client raced ahead of the server in bytes sent.

Shouldn't the program work regardless of how many bytes are sent by the client? The bytes are sent in small messages to the server, the server processes them in small chunks and sends them back. It shouldn't matter how many bytes in total there are. So why did this happen?

Because it turned into a **deadlock** situation. First, the server and client stopped because the server's output buffer and the client's input buffer have been filled to capacity, and that triggered TCP to stop the socket from sending more data that is guaranteed to be discarded.

But why did the buffers fill up?

Because in the client code, the client sends all the bytes **first**, and only then does it start receiving the processed data from the server. So the processed data sent by the server will keep accumulating in the server's output buffer and the client's input buffer, without anyone taking data from them. They will both fill up and TCP will stop the socket from sending any more.

To prevent this deadlock from happening, there are two options.

1. Turn off blocking using socket options so that calls like send() and recv() will return immediately if they cannot send data anymore.

2. Use multiple threads or processes for sending and receiving, so one does not sequen-

tially follow the other.

We will look into both options in more detail later in the course.

TCP streams are like files

You can treat TCP data streams just like files, so a TCP socket can be treated like a regular Python file object. Modules such as pickle and json that can read and write data from a file, can also do so with a TCP socket. Python has a `makefile()` method on the socket object that returns a file object, but behind the curtain, it's actually using `send()` and `recv()` calls. See the following example:

```
from socket import *
s = socket(AF_INET, SOCK_STREAM)
f = s.makefile()
```

Now `f` is a file object that you read from and write to, but it's actually receiving from and sending to a socket.

Task

Using the server and client from the above section, modify the server to make a file object from its socket that is receiving data from the client. Have the client send some data to the server and the server will read from a file (which is actually a socket) line by line and print each line to screen.

To submit

Submit your server program from the task above. Your lab mark will be based on effort.