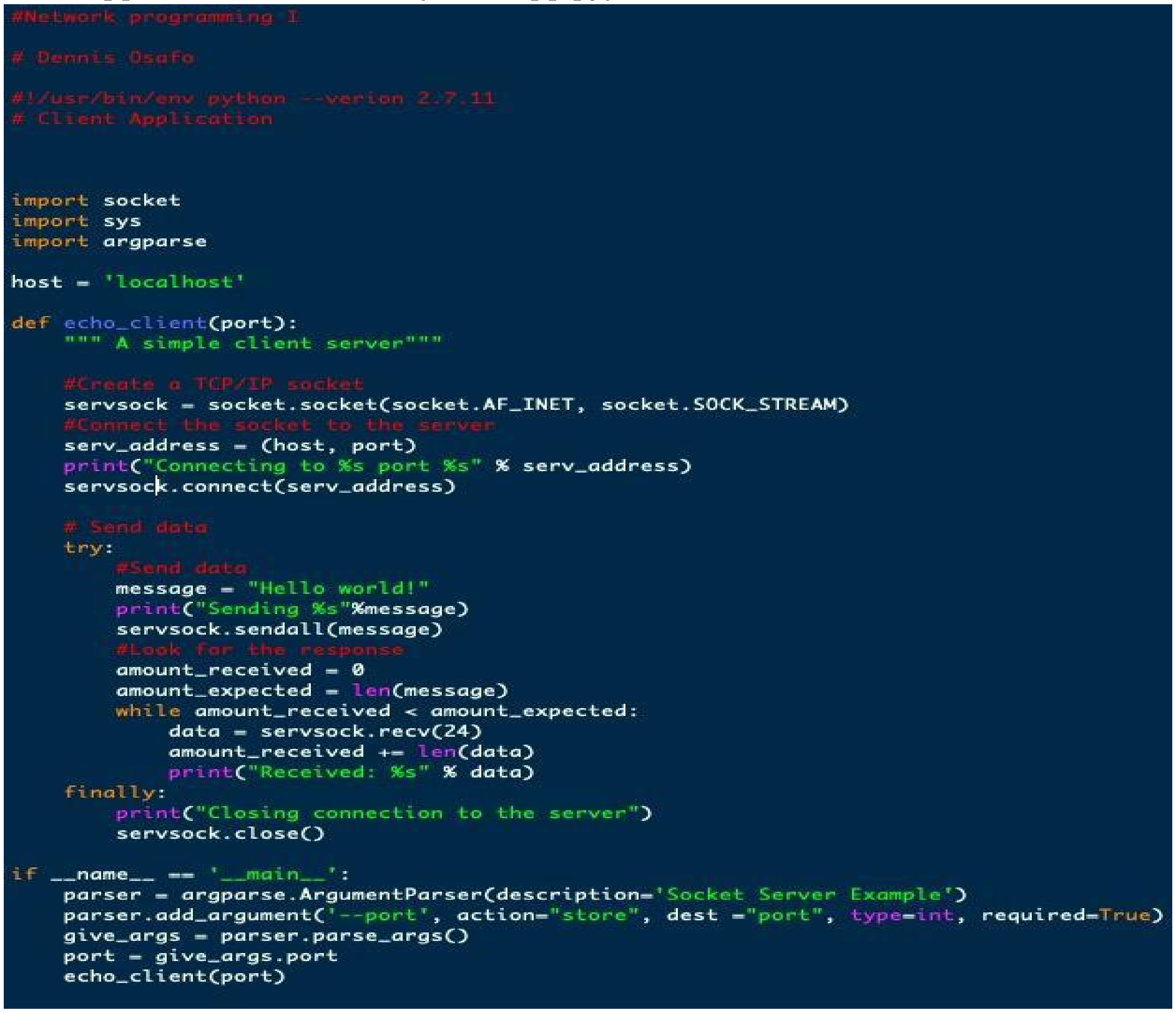
Echo Application to measure TCP Performance

Echo Client-Server Application

Overview: Implement a client and a server that communicate over a network using TCP. The server is essentially an echo server, which simply echoes the message it receives from the client. Here is what the server and client application must accomplish:

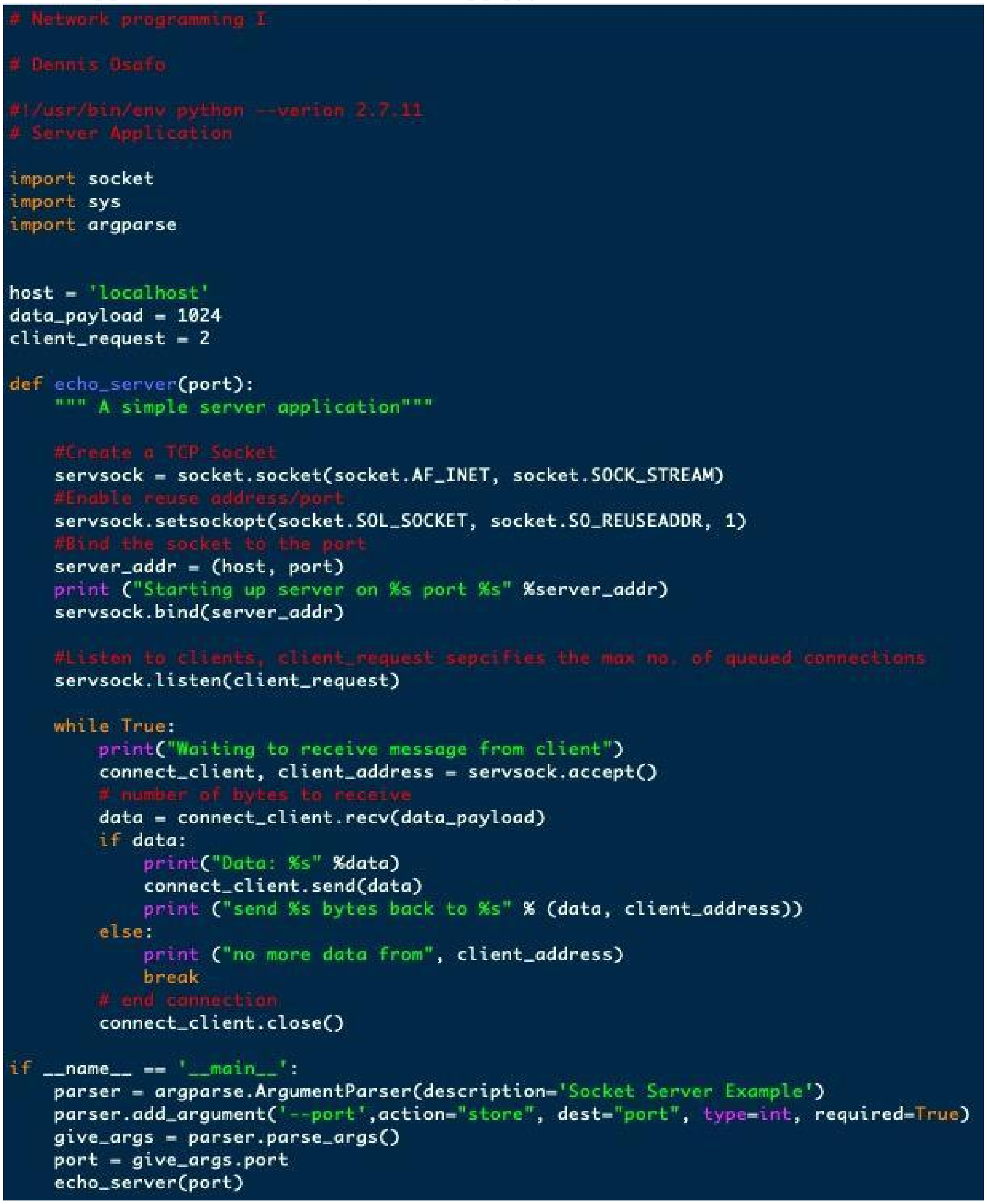
- The server should accept, as a command line argument, a port number that it will run at. After being started, the server should repeatedly accept an input message from a client and send back the same message
- The client should accept, as command line arguments, a host name (or IP address), as well as a port number for the server. Using this information, it creates a connection (using TCP) with the server, which should be running already. The client program then sends a message (text string) to the server using the connection. When it receives back the message, it prints it and exits.

Client application source code(client_app.py)



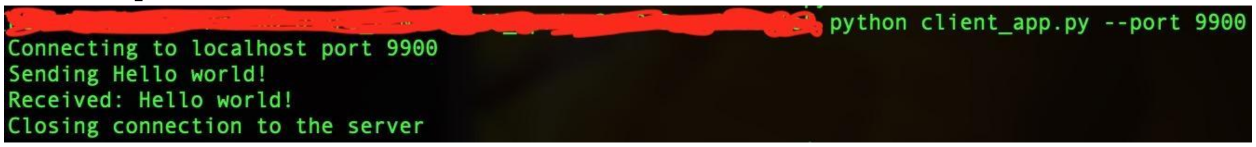
```
#Network programming I

# Dennis Osafo

#!/usr/bin/env python --verion 2.7.11
# Client Application



import socket
import sys
import argparse

host = 'localhost'

def echo_client(port):
    """ A simple client server"""

    #Create a TCP/IP socket
    servsock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    #Connect the socket to the server
    serv_address = (host, port)
    print("Connecting to %s port %s" % serv_address)
    servsock.connect(serv_address)

    # Send data
    try:
        #Send data
        message = "Hello world!"
        print("Sending %s"%message)
        servsock.sendall(message)
        #Look for the response
        amount_received = 0
        amount_expected = len(message)
        while amount_received < amount_expected:
            data = servsock.recv(24)
            amount_received += len(data)
            print("Received: %s" % data)
    finally:
        print("Closing connection to the server")
        servsock.close()

if __name__ == '__main__':
    parser = argparse.ArgumentParser(description='Socket Server Example')
    parser.add_argument('--port', action="store", dest ="port", type=int, required=True)
    give_args = parser.parse_args()
    port = give_args.port
    echo_client(port)
```

Server application source code (server_app.py)



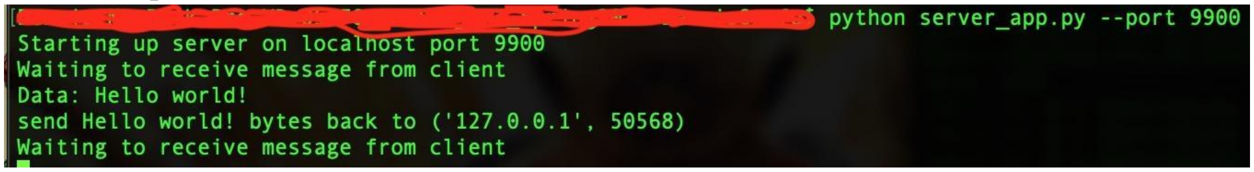
```
# Network programming I

# Dennis Osafo

#!/usr/bin/env python --verion 2.7.11
# Server Application

import socket
import sys
import argparse


host = 'localhost'
data_payload = 1024
client_request = 2

def echo_server(port):
    """ A simple server application"""

    #Create a TCP Socket
    servsock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    #Enable reuse address/port
    servsock.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
    #Bind the socket to the port
    server_addr = (host, port)
    print ("Starting up server on %s port %s" %server_addr)
    servsock.bind(server_addr)

    #Listen to clients, client_request sepcifies the max no. of queued connections
    servsock.listen(client_request)

    while True:
        print("Waiting to receive message from client")
        connect_client, client_address = servsock.accept()
        # number of bytes to receive
        data = connect_client.recv(data_payload)
        if data:
            print("Data: %s" %data)
            connect_client.send(data)
            print ("send %s bytes back to %s" % (data, client_address))
        else:
            print ("no more data from", client_address)
            break
        # end connection
        connect_client.close()

if __name__ == '__main__':
    parser = argparse.ArgumentParser(description='Socket Server Example')
    parser.add_argument('--port',action="store", dest="port", type=int, required=True)
    give_args = parser.parse_args()
    port = give_args.port
    echo_server(port)
```

Client requests



```
[redacted] python client_app.py --port 9900
Connecting to localhost port 9900
Sending Hello world!
Received: Hello world!
Closing connection to the server
```

Server response



```
[redacted] python server_app.py --port 9900
 Starting up server on localhost port 9900
 Waiting to receive message from client
 Data: Hello world!
 send Hello world! bytes back to ('127.0.0.1', 50568)
 Waiting to receive message from client
```

Performing RTT and Throughput Measurements

Overview: We are extending the echo client-server application to measure the round trip time (RTT) and throughput of the path connecting the client to the server. To measure RTT, you will use TCP to send and receive messages of size 1, 100, 200,400, 800 and 1000 bytes. To measure throughput, you will use TCP to send and receive messages of size 1K, 2K, 4K, 8K, 16K and 32K bytes. For each measurement and for each message size, the client will send at least ten probe messages to the server, which will echo back the messages.

For the purpose of this work sample would only use 1,000 bytes to measure RTT and 2K to measure throughput.

Protocol phases: The echo application will be extended, however, by specifying the exact protocol interactions between the client and the server. This entails specifying the exact message formats, as well as the different communication phases, as outlined next.

1. **Connection Setup Phase (CSP):** This is the first phase in the protocol where the client informs the server that it wants to conduct active network measurements in order to compute the RTT and throughput of its path to the server.
    a. **CSP: Client -** After setting up a TCP connection to the server, the client must send a single message to the server having the following format: **<PROTOCOL PHASE><WS><M−TYPE><WS><MSG SIZE><WS><PROBES><WS><SERVER DELAY>\n**
        - **PROTOCOL PHASE:** The protocol phase during the initial setup will be denoted by the lower case character ‘s’. This allows the server to differentiate between the different protocol phases that the client can be operating at, as we will see.
        - **M-TYPE (Measurement Type):** Allows the client to specify whether it wants to compute the RTT, denoted by “RTT”, or the throughput, denoted by “TTPUT”.
        - **MSG SIZE (Message Size):** Specifies the number of bytes in the probe’s payload
        - **PROBES:** Allows the client to specify the number of measurement probes that the server should expect to receive. Once all the probe

messages have been echoed back and a sample measurement is taken for each one, the client should compute an estimate of the mean (average) RTT or mean throughput, depending on the type of measurement being performed.

- **SERVER DELAY:** Specifies the amount of time that the server should wait be-fore echoing the message back to the client. The default value should be 0. You will vary this value later to emulate paths with longer propagation delays. Even though increasing the server delay merely increases the processing time at the server, it nevertheless causes the feedback delay, observed by the sender, to in-crease, which has an effect somewhat similar to increasing the path's propagation delay.
- **WS:** A single white space to separate the different fields in the message. The white space could serve as a delimiter for the server when parsing or tokenizing the received message.
- The last is a new line character that indicates the end of the message.

b. **CSP: Server** - The server should parse the connection setup message to log the values of all the variables therein since they will be needed for error checking purposes. Upon the reception of a valid connection setup message, the server should respond with a text message containing the string "200 OK: Ready" informing the client that it can proceed to the next phase. On the other hand, if the connection setup message is incomplete or invalid, the server should respond with a text message containing the string "404 ERROR: Invalid Connection Setup Message" and then terminate the connection.

c. **CSP: Summary** - During correct operation, after setting up a TCP connection to the server, the client sends a single connection setup message to the server. The server parses and logs the information in the message and responds with a "200 OK: Ready "text message informing the client to proceed to the next phase.

2. **Measurement Phase (MP)** - In this phase, the client starts sending probe messages to the server in order to make the appropriate measurements required for computing the mean RTT or the mean throughput of the path connecting it to the server.

    a. **MP: Client -**The client should send the specified number of probe messages to the server with an increasing sequence number starting from 1. More specifically, the message format is as follows:<PROTOCOL PHASE><WS><PAYLOAD><WS><PROBE SEQUENCE NUMBER>\n

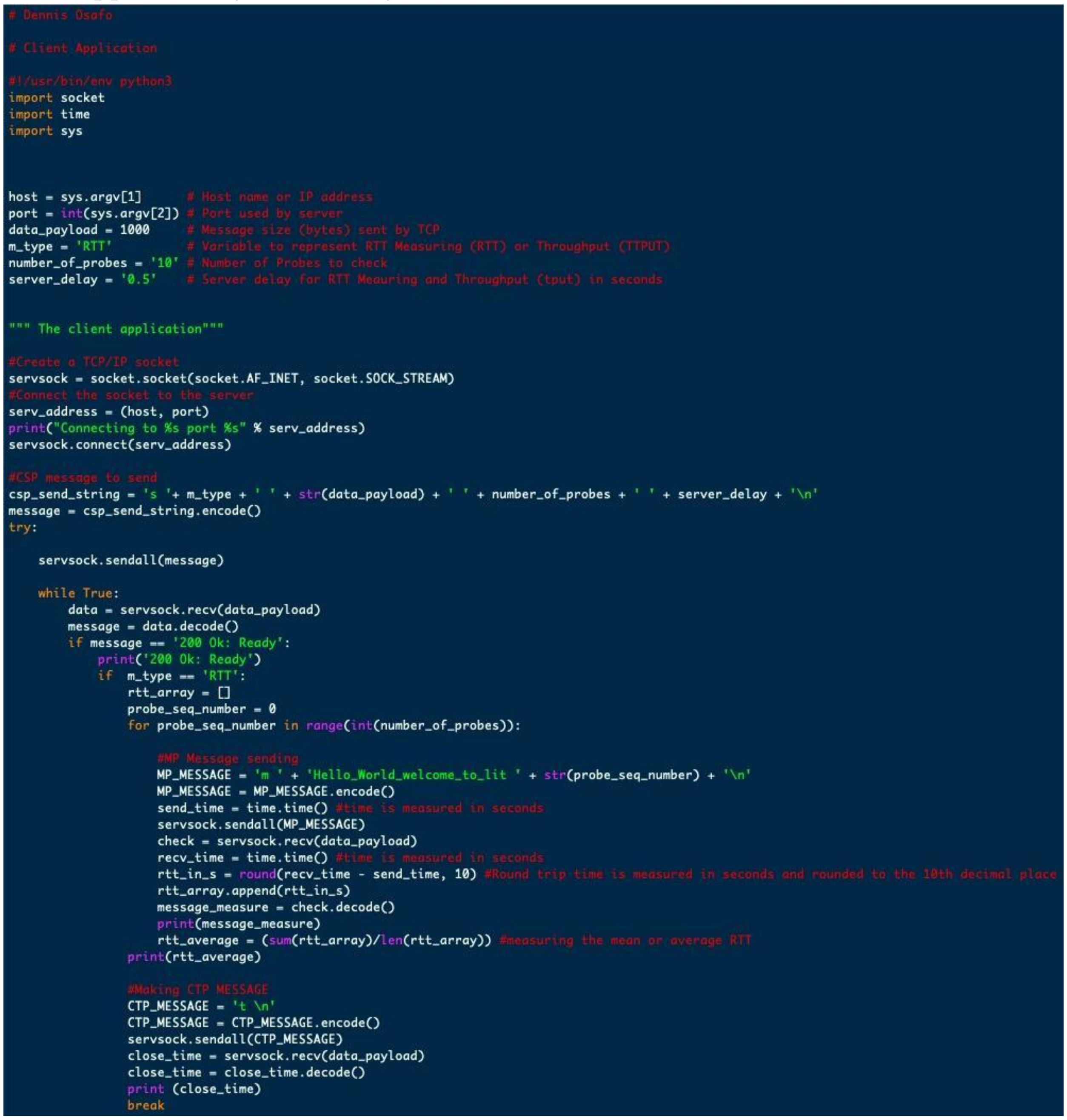
    - **PROTOCOL PHASE**: The protocol phase when conducting the measurements will be denoted by the lower case character 'm'.
    - **PAYLOAD:** This is the probe's payload and can be any arbitrary text whose size was specified in the connection setup message using the MESSAGE SIZE (MESS_SIZE) variable.
    - **PROBE SEQUENCE NUMBER:** The probe messages should have increasing sequence numbers starting from 1 up to the number of

        probes specified in the connection setup message using the NUMBER OF PROBES variable.

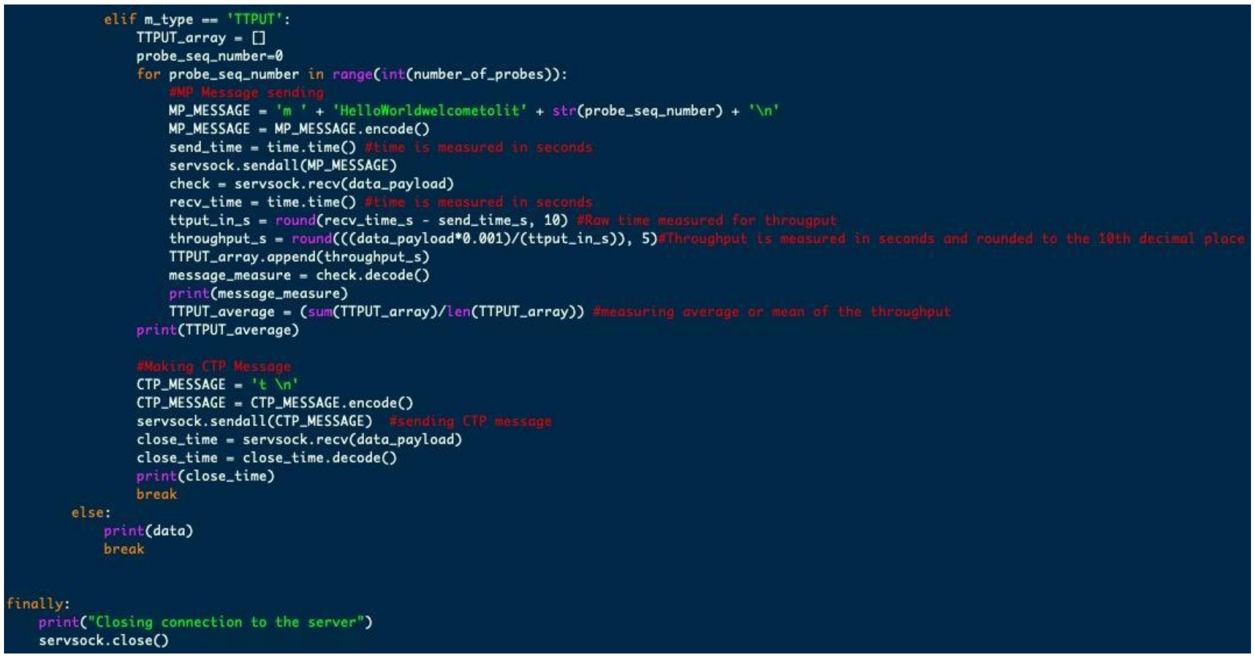
   b. **MP: Server-** the server should echo back every probe message received. It should also keep track of the probe sequence numbers to make sure they are indeed being incremented by 1 each time and do not exceed the number of probes specified in the connection setup phase. If the probe message is incomplete or invalid (contains an incorrect sequence number for example) the server should not echo the message back. Instead, the server should respond with a text message containing the string "404ERROR: Invalid Measurement Message" and then terminate the connection
   c. **MP: Summary-** The client repeatedly sends measurement messages to the server in an attempt to compute the mean RTT and/or mean throughput. A sample measurement is taken for each probe sent out. The server repeatedly echoes messages back to the client unless it detects erroneous behavior in which case it sends an error message and terminates the connection.

3. **Connection Termination Phase (CTP).** In this phase, the client and the server attempt to gracefully terminate the connection. We will outline the expected behavior from both the client and the server next.
   a. **CTP: Client** - The client should send a termination request to the server and then wait for a response (unless of course the server already terminated the connection due to an error in the Measurement Phase). Once a response is received, the client should terminate the connection. The message format is as follows: <PROTOCOL PHASE>\n
      - PROTOCOL PHASE: The protocol phase when terminating the connection will be denoted by the lower case character 't'
   b. **CTP: Server -** If the message format is correct, the server should respond with a text message containing the string "200 OK: Closing Connection". Otherwise, the server should respond with a text message containing the string "404 ERROR: In-valid Connection Termination Message". Either way the server should terminate the connection.
   c. **CTP: Summary** - During correct operation, the client sends a termination message, the server responds with "200 OK: Closing Connection" text message and then both terminate the connection.

Client application (source code)



```python
# Dennis Osafo

# Client Application

#!/usr/bin/env python3
import socket
import time
import sys



host = sys.argv[1]      # Host name or IP address
port = int(sys.argv[2]) # Port used by server
data_payload = 1000     # Message size (bytes) sent by TCP
m_type = 'RTT'          # Variable to represent RTT Measuring (RTT) or Throughput (TTPUT)
number_of_probes = '10' # Number of Probes to check
server_delay = '0.5'    # Server delay for RTT Meauring and Throughput (tput) in seconds


""" The client application"""

#Create a TCP/IP socket
servsock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
#Connect the socket to the server
serv_address = (host, port)
print("Connecting to %s port %s" % serv_address)
servsock.connect(serv_address)

#CSP message to send
csp_send_string = 's '+ m_type + ' ' + str(data_payload) + ' ' + number_of_probes + ' ' + server_delay + '\n'
message = csp_send_string.encode()
try:

    servsock.sendall(message)

    while True:
        data = servsock.recv(data_payload)
        message = data.decode()
        if message == '200 Ok: Ready':
            print('200 Ok: Ready')
            if  m_type == 'RTT':
                rtt_array = []
                probe_seq_number = 0
                for probe_seq_number in range(int(number_of_probes)):

                    #MP Message sending
                    MP_MESSAGE = 'm ' + 'Hello_World_welcome_to_lit ' + str(probe_seq_number) + '\n'
                    MP_MESSAGE = MP_MESSAGE.encode()
                    send_time = time.time() #time is measured in seconds
                    servsock.sendall(MP_MESSAGE)
                    check = servsock.recv(data_payload)
                    recv_time = time.time() #time is measured in seconds
                    rtt_in_s = round(recv_time - send_time, 10) #Round trip time is measured in seconds and rounded to the 10th decimal place
                    rtt_array.append(rtt_in_s)
                    message_measure = check.decode()
                    print(message_measure)
                    rtt_average = (sum(rtt_array)/len(rtt_array)) #measuring the mean or average RTT
                print(rtt_average)

                #Making CTP MESSAGE
                CTP_MESSAGE = 't \n'
                CTP_MESSAGE = CTP_MESSAGE.encode()
                servsock.sendall(CTP_MESSAGE)
                close_time = servsock.recv(data_payload)
                close_time = close_time.decode()
                print (close_time)
                break
```



```python
            elif m_type == 'TTPUT':
                TTPUT_array = []
                probe_seq_number=0
                for probe_seq_number in range(int(number_of_probes)):
                    #MP Message sending
                    MP_MESSAGE = 'm ' + 'HelloWorldwelcometolit' + str(probe_seq_number) + '\n'
                    MP_MESSAGE = MP_MESSAGE.encode()
                    send_time = time.time() #time is measured in seconds
                    servsock.sendall(MP_MESSAGE)
                    check = servsock.recv(data_payload)
                    recv_time = time.time() #time is measured in seconds
                    ttput_in_s = round(recv_time_s - send_time_s, 10) #Raw time measured for througput
                    throughput_s = round(((data_payload*0.001)/(ttput_in_s)), 5)#Throughput is measured in seconds and rounded to the 10th decimal place
                    TTPUT_array.append(throughput_s)
                    message_measure = check.decode()
                    print(message_measure)
                    TTPUT_average = (sum(TTPUT_array)/len(TTPUT_array)) #measuring average or mean of the throughput
                print(TTPUT_average)

                #Making CTP Message
                CTP_MESSAGE = 't \n'
                CTP_MESSAGE = CTP_MESSAGE.encode()
                servsock.sendall(CTP_MESSAGE)  #sending CTP message
                close_time = servsock.recv(data_payload)
                close_time = close_time.decode()
                print(close_time)
                break
        else:
            print(data)
            break


finally:
    print("Closing connection to the server")
    servsock.close()
```