Paras Gupta
19110128

CS 433
Computer Networks
Assignment 2

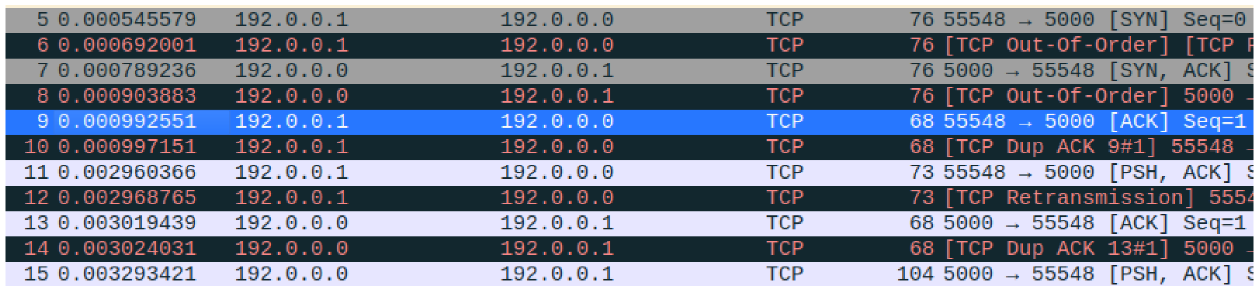
Part I: Port over the Network Application built in Assignment-01 to Mininet:

Q1.

a. A simple linear topology was created using Mininet containing two hosts A and B, connected using a switch. The server.py file written in Assignment 1 was changed to bind to the IP address of host A (192.0.0.0). The client.py was also changed to bind to the same IP address (192.0.0.0). Nothing else was changed in the code written in Assignment 1.

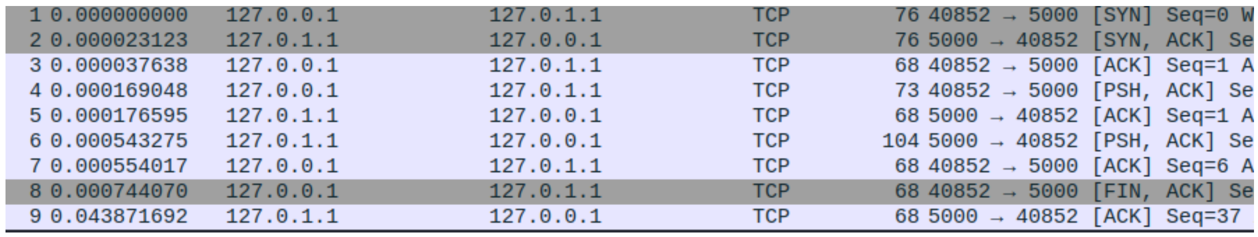
   The network topology was then simulated in Mininet and the files, server.py & client.py, were run on the two host nodes. The client was made to request the service 'CWD' from the server using encryption mode 2 - 'substitute'.

b. The following screenshot shows the packets captured by Wireshark when the client requested the above service from the server on Mininet.



```
 5 0.000545579   192.0.0.1          192.0.0.0          TCP         76 55548 → 5000 [SYN] Seq=0
 6 0.000692001   192.0.0.1          192.0.0.0          TCP         76 [TCP Out-Of-Order] [TCP R
 7 0.000789236   192.0.0.0          192.0.0.1          TCP         76 5000 → 55548 [SYN, ACK] S
 8 0.000903883   192.0.0.0          192.0.0.1          TCP         76 [TCP Out-Of-Order] 5000 -
 9 0.000992551   192.0.0.1          192.0.0.0          TCP         68 55548 → 5000 [ACK] Seq=1
10 0.000997151   192.0.0.1          192.0.0.0          TCP         68 [TCP Dup ACK 9#1] 55548 -
11 0.002960366   192.0.0.1          192.0.0.0          TCP         73 55548 → 5000 [PSH, ACK] S
12 0.002968765   192.0.0.1          192.0.0.0          TCP         73 [TCP Retransmission] 5554
13 0.003019439   192.0.0.0          192.0.0.1          TCP         68 5000 → 55548 [ACK] Seq=1
14 0.003024031   192.0.0.0          192.0.0.1          TCP         68 [TCP Dup ACK 13#1] 5000 -
15 0.003293421   192.0.0.0          192.0.0.1          TCP        104 5000 → 55548 [PSH, ACK] S
```

   For comparison, the same service was requested by the client without Mininet. The following screenshot shows the Wireshark capture for the same.



```
1 0.000000000   127.0.0.1          127.0.1.1          TCP         76 40852 → 5000 [SYN] Seq=0 W
2 0.000023123   127.0.1.1          127.0.0.1          TCP         76 5000 → 40852 [SYN, ACK] Se
3 0.000037638   127.0.0.1          127.0.1.1          TCP         68 40852 → 5000 [ACK] Seq=1 A
4 0.000169048   127.0.0.1          127.0.1.1          TCP         73 40852 → 5000 [PSH, ACK] Se
5 0.000176595   127.0.1.1          127.0.0.1          TCP         68 5000 → 40852 [ACK] Seq=1 A
6 0.000543275   127.0.1.1          127.0.0.1          TCP        104 5000 → 40852 [PSH, ACK] Se
7 0.000554017   127.0.0.1          127.0.1.1          TCP         68 40852 → 5000 [ACK] Seq=6 A
8 0.000744070   127.0.0.1          127.0.1.1          TCP         68 40852 → 5000 [FIN, ACK] Se
9 0.043871692   127.0.1.1          127.0.0.1          TCP         68 5000 → 40852 [ACK] Seq=37
```

It can be observed that the RTT value with Mininet is larger than the RTT value with Mininet. Thus performance, when measured using RTT as the metric, degrades while using Mininet. The reasons for the same are given below:

i. There is a switch between the hosts representing the server and client in Mininet, which can add to the delays.
ii. Mininet is an emulator that provides an interface to specify bandwidth and link constraints. These constraints can add to performance degradation. However, in the socket-based client server code, all this is taken care of by the system.

Q2.

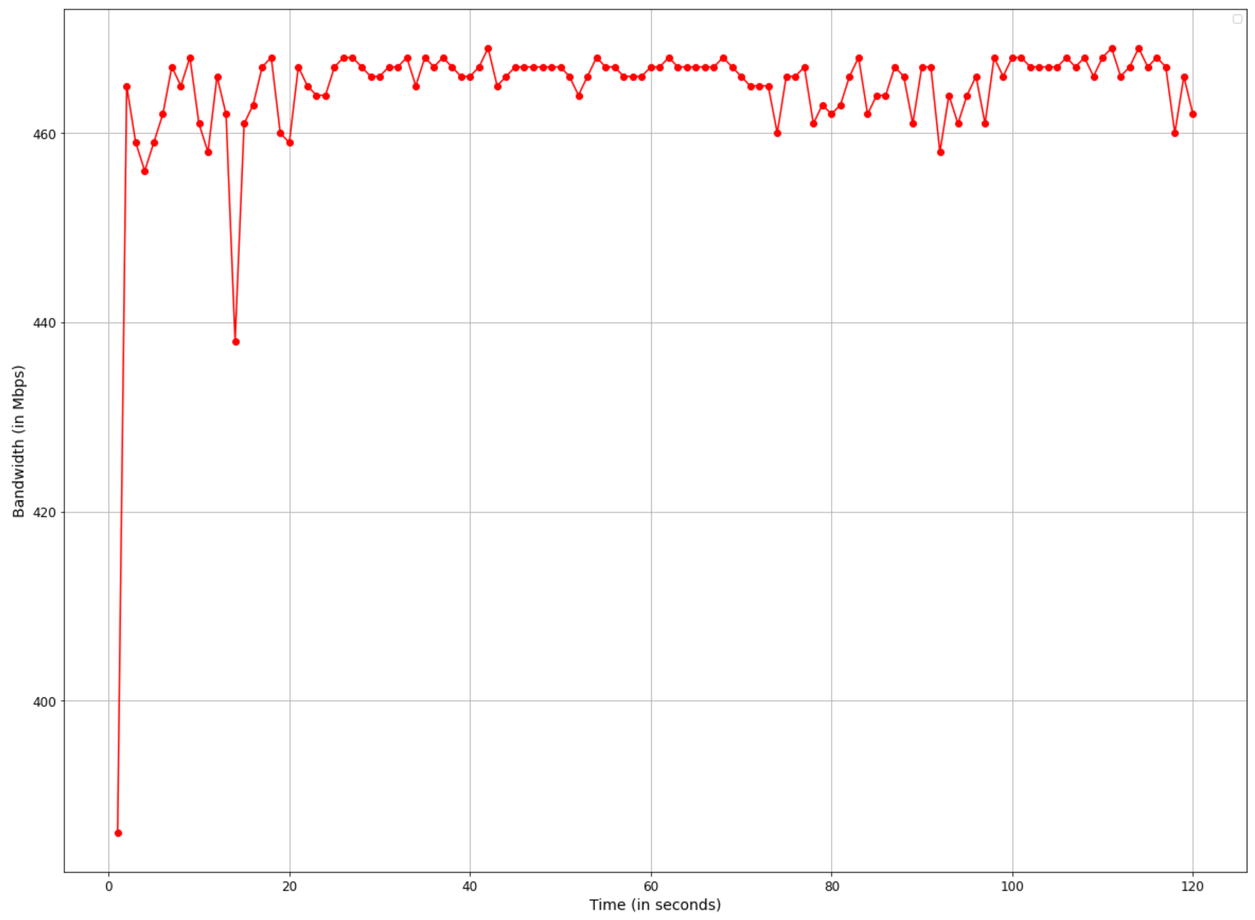
a. **Implementation:** The given network was implemented as a custom topology using Mininet. The implementation has been uploaded on GitHub in the file *q2_q3_topology.py*

b. **Latency:** The RTT was measured using the *ping* command, with the request count (-c) set to 120. The average RTT values over three test iterations for all node pairs are as follows:

   i. **AB** → *27.121 ms*
   ii. **AC** → *35.801 ms*
   iii. **AD** → *6.810 ms*
   iv. **BC** → *14.400 ms*
   v. **BD** → *27.290 ms*

   We can see that the observed average RTT for AB and AC is not the same. The only difference in the routes connecting A to B and A to C is the delay of Link 4, indicating that there is no effect of link delay on RTT. We can also see that the observed average RTT for AB and BD is almost the same, owing to their similar network route.

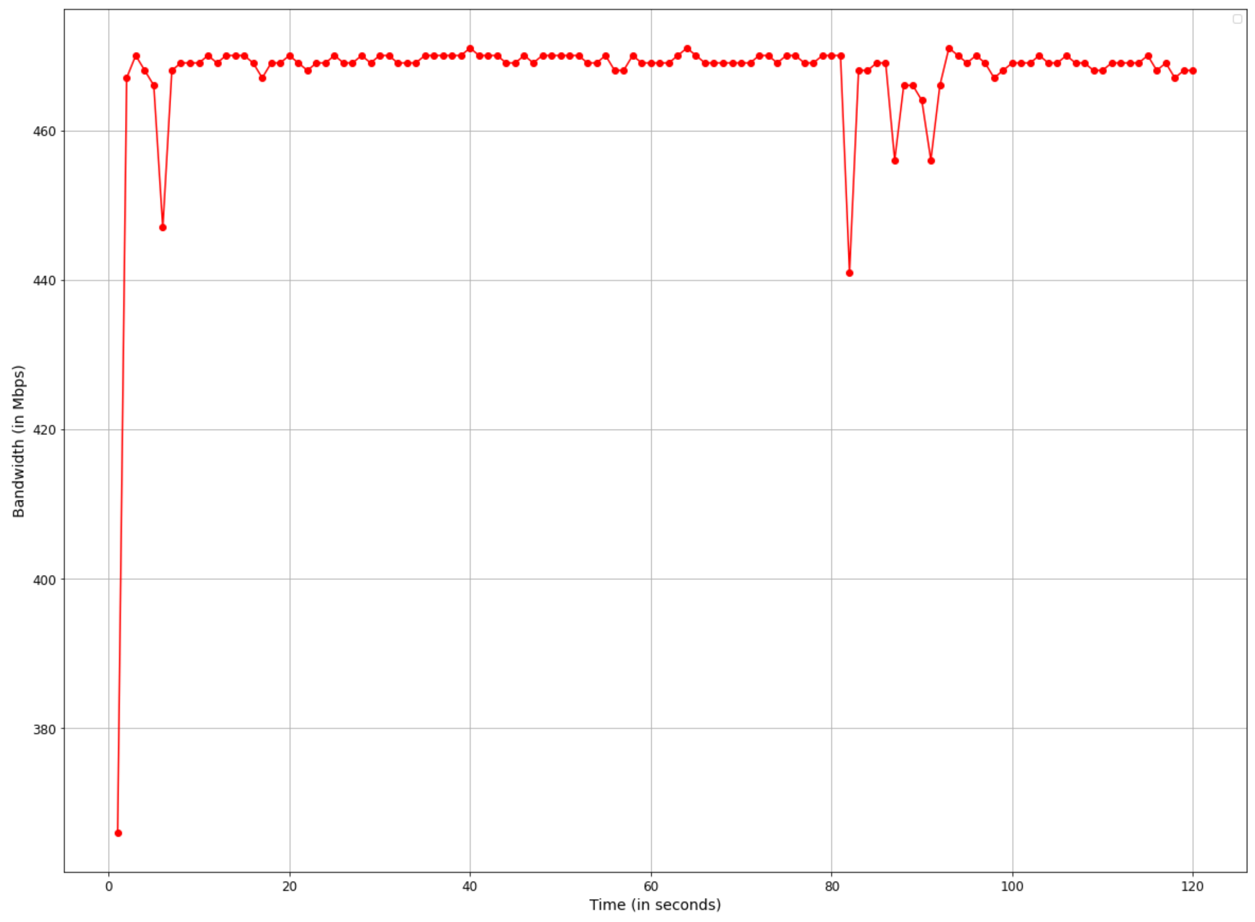
c. **Throughput:** The plot of throughput over time when a TCP connection is established between **iperf client B** and **iperf server A** is given below:



(c) Throughput: Throughput plot between Server A and Client B

We can observe that the bandwidth value over time is close to 470 Mbps, which is almost equal to the bottleneck bandwidth in the given network between host A and host B.

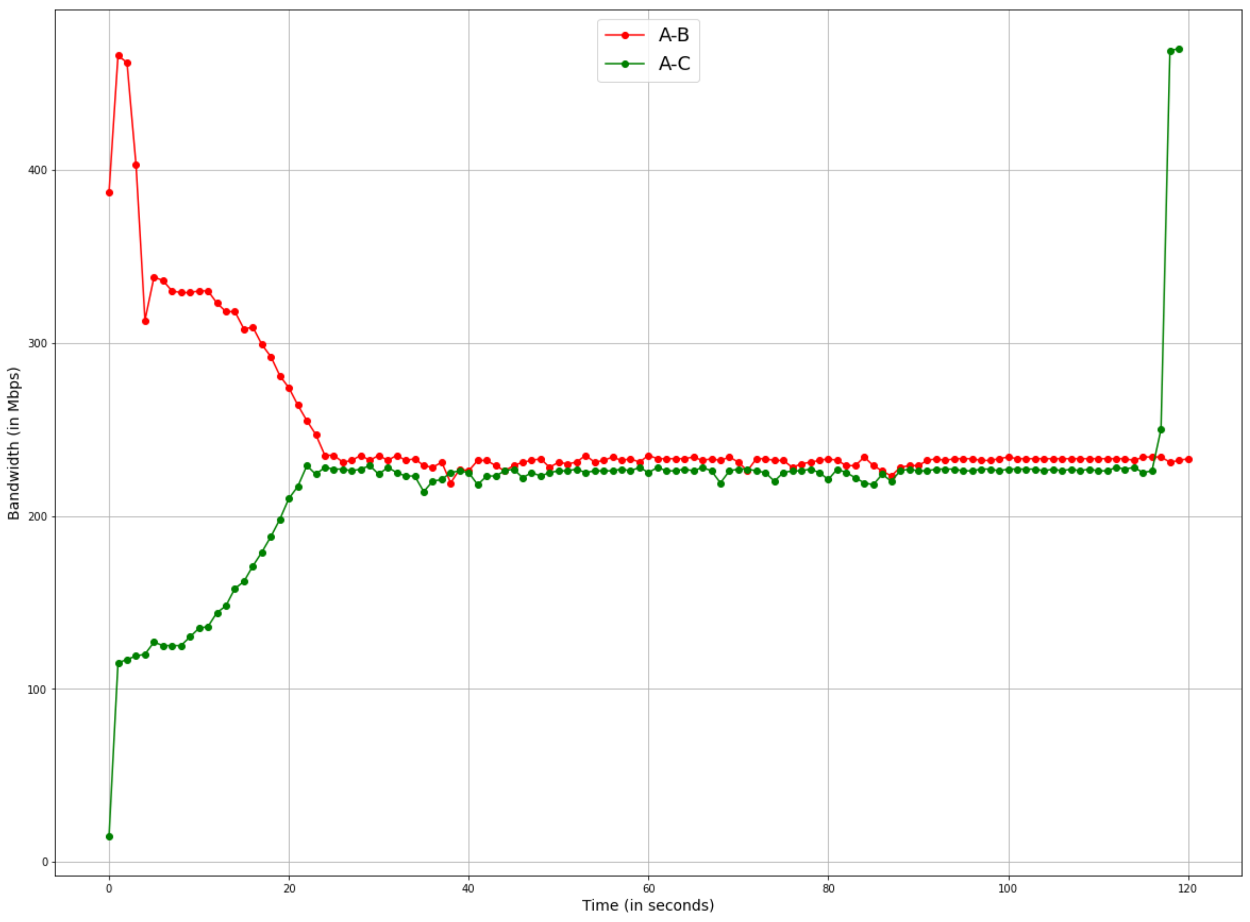
d. **Delay Impact:** The plot of throughput over time when a TCP connection is established between **iperf client C** and **iperf server A** is given below:



(d) Delay Impact: Throughput plot between Server A and Client C

The plots given in parts (c) and (d) can be compared to see that the average value of bandwidth over time remains the same in both cases. The only difference in the routes connecting A to B and A to C is the delay of Link 4, showing that there is no effect of link delay on the bandwidth of the connection.

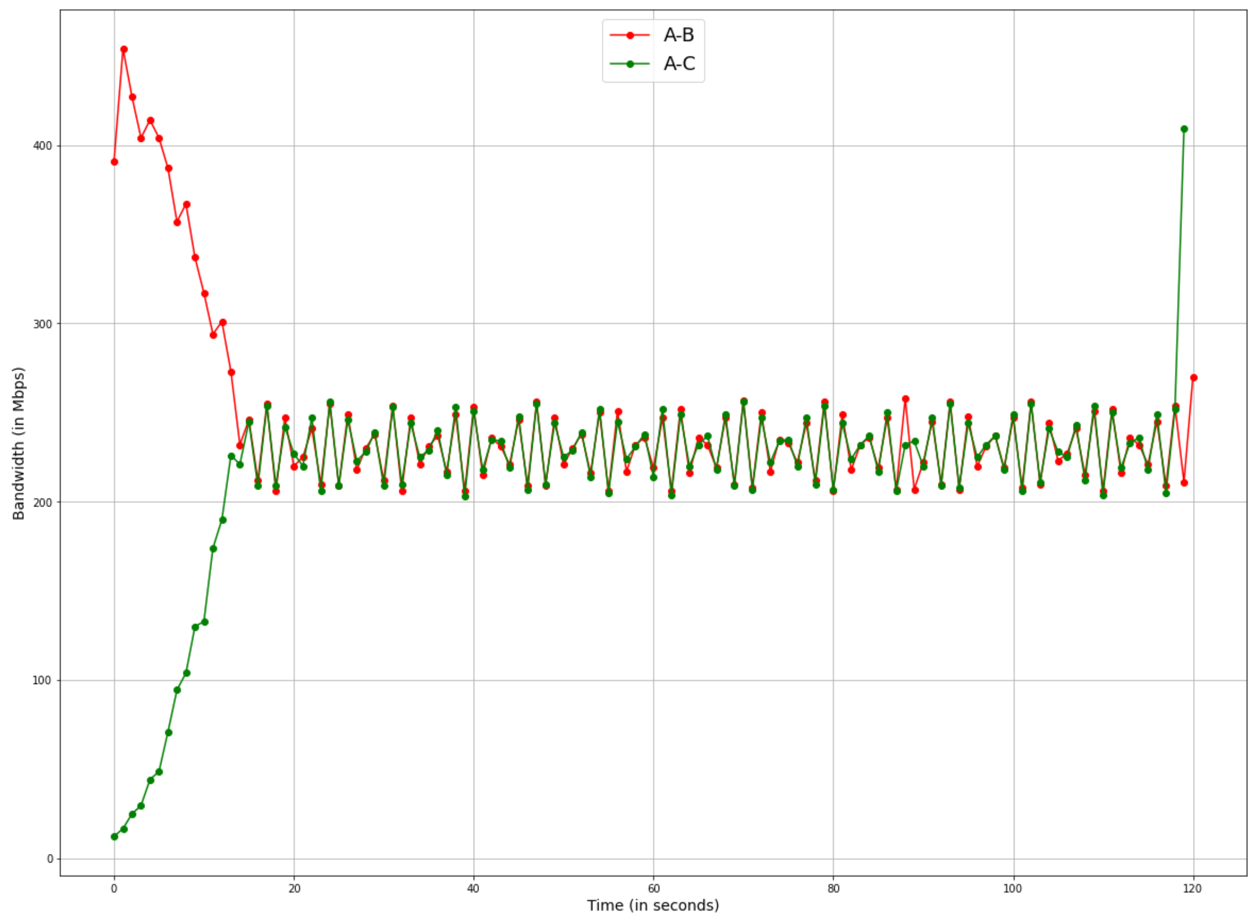
e. **Concurrency(i):** The plot of throughput over time when a TCP connection is simultaneously established between **iperf clients B, C** and **iperf server A** is given below:



(e) Concurrency(i): Throughput plot between Server A and concurrent clients B, C

We can observe the initial disparity between the bandwidths of connections A-B and A-C, caused by the connections not starting at precisely the same time. The connection A-B with was started earlier as it has a higher initial bandwidth. However, after some time, (~22 s) the bandwidth for both connections becomes almost the same and roughly equal to half (~250 Mbps) of the bottleneck bandwidth of Link 2 shared by them. At the end of the measured time interval, the bandwidth for A-C becomes higher showing that connection A-B had already ended.

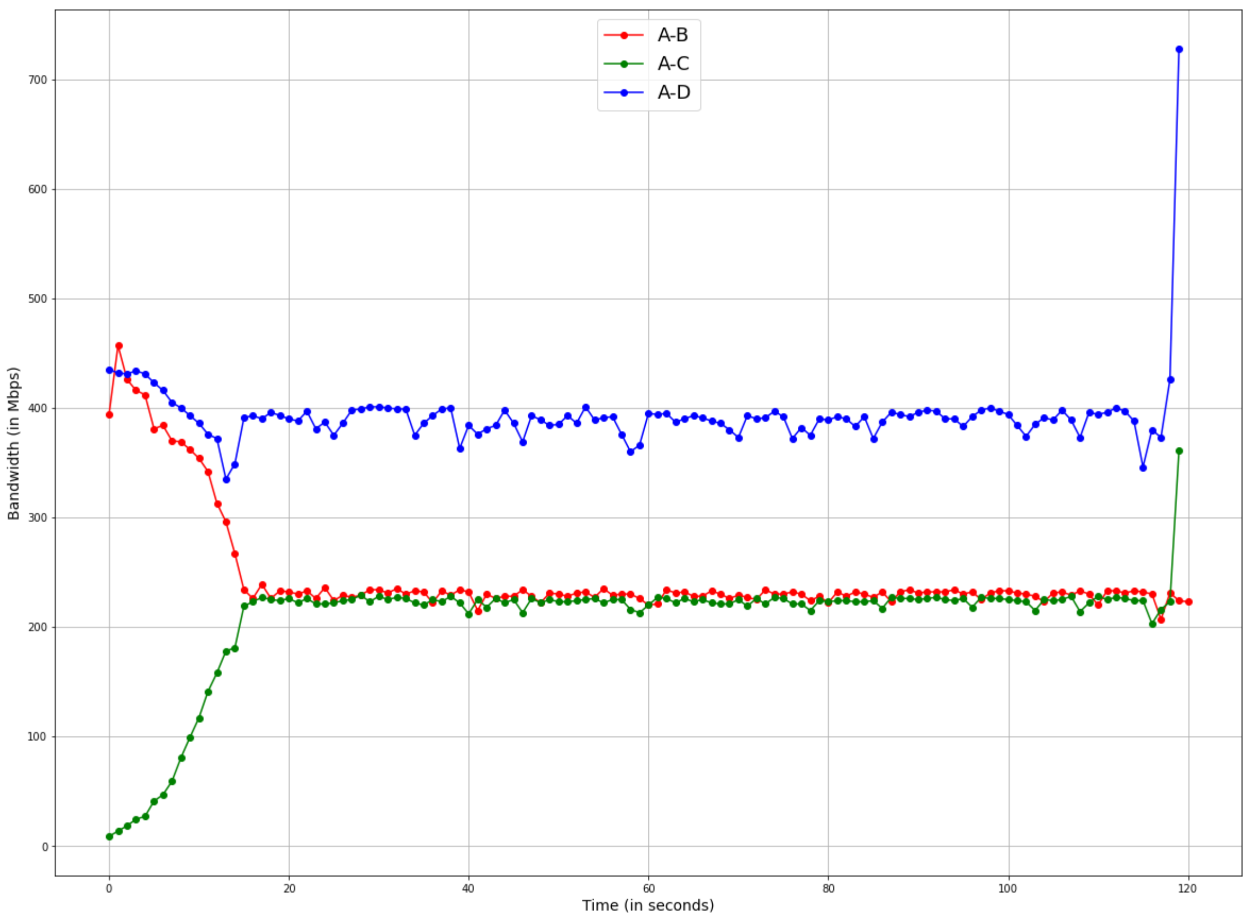
f. **Fairness(ii):** The delay for link 4 was changed to 1ms from the previous delay of 5ms. The plot of throughput over time when a TCP connection is simultaneously established between **iperf clients B, C** and **iperf server A** is given below:



(f) Fairness(ii): Throughput plot between Server A and concurrent clients B, C

The plots given in (e) and (f) follow the same trend as described in (d). The bottleneck bandwidth of 500 Mbps gets almost equally distributed among the two connections irrespective of the value of link delay.

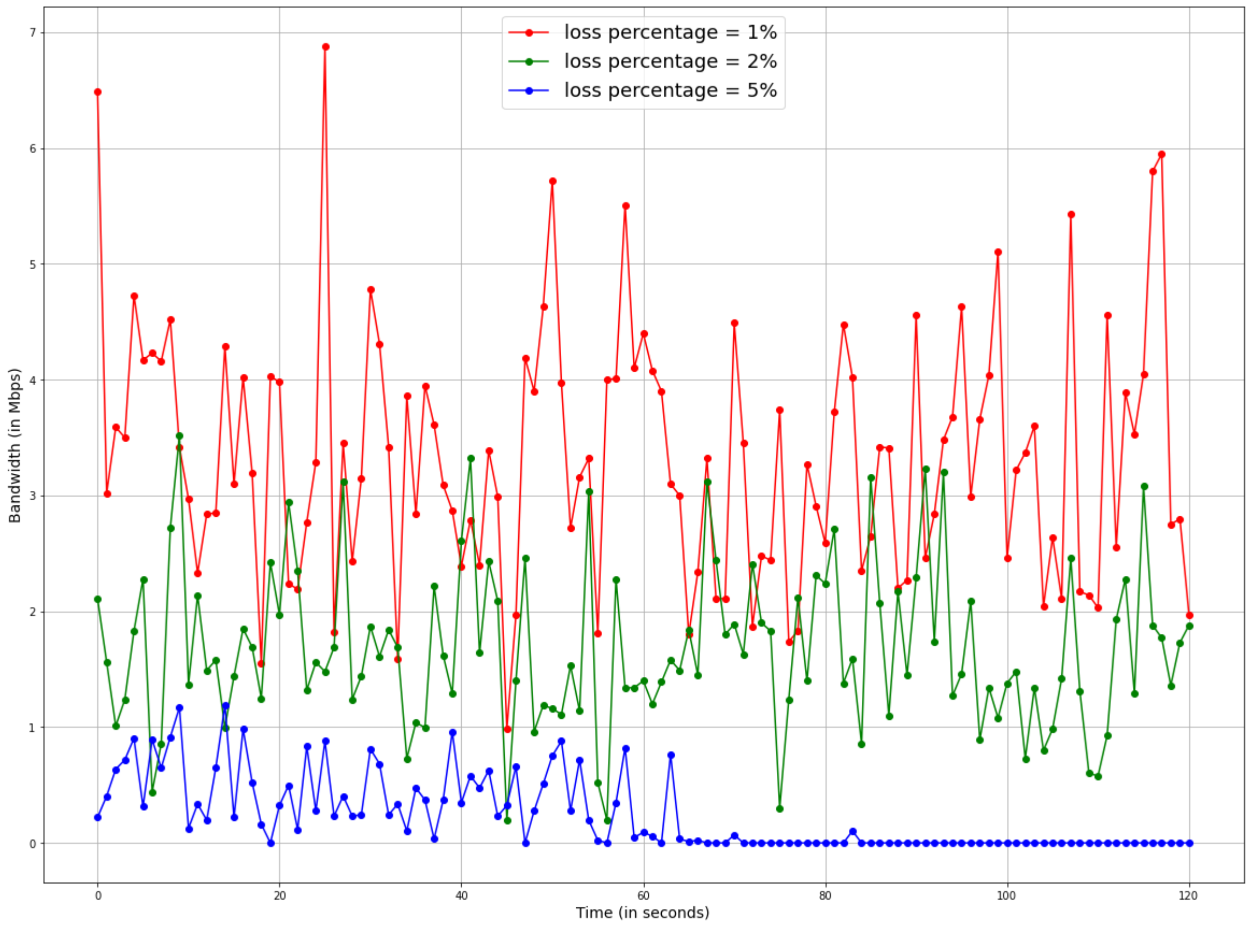
g. **Concurrency(iii):** The plot of throughput over time when a TCP connection is simultaneously established between **iperf clients B, C, D** and **iperf server A** is given below:



(g) Concurrency(iii): Throughput plot between Server A and concurrent clients B, C, D

The plot shows that the connections A-B and A-C still operate at a bandwidth of 250 Mbps even though a third connection A-D has been run concurrently. This is because the bottleneck link (Link 3) common between connections A-B and A-C is not part of A-D. The common link between all three connections is Link 1. Since A-B and A-C operate at 250 Mbps bandwidth, the remaining bandwidth of 500 Mbps (1000 Mbps - 250 Mbps - 250 Mbps) in Link 1 gets allocated to A-D. The same can also be observed in the above plot.

Q3.



Throughput plot between Server A and Client B with added link losses

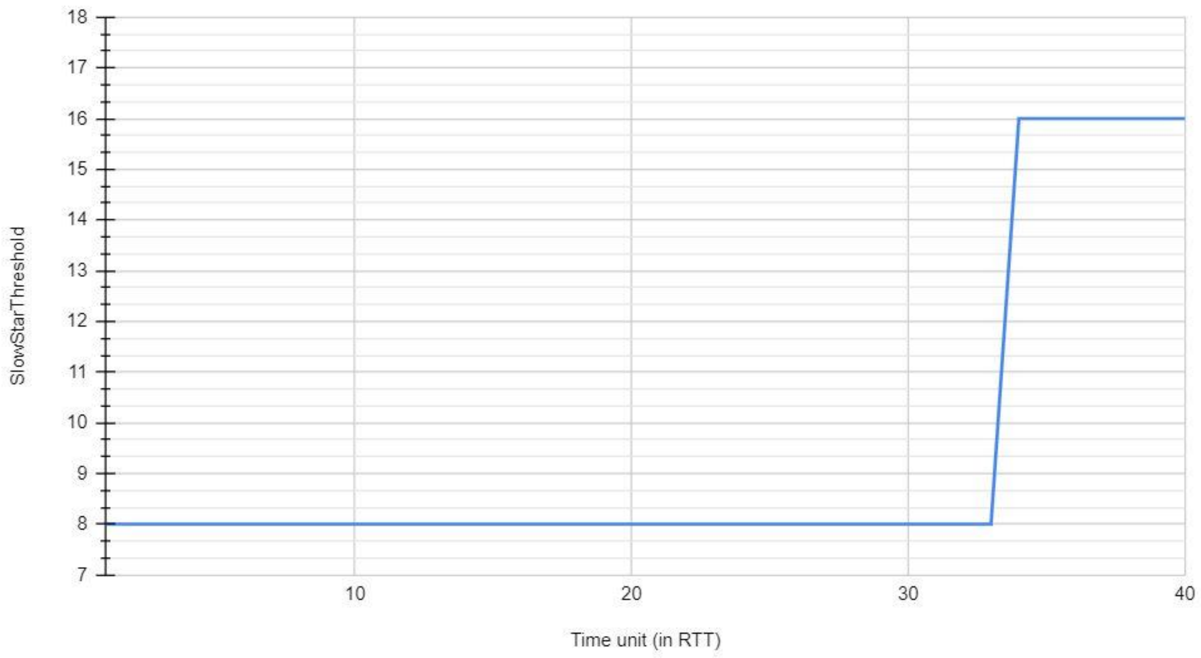
The connection between host A and host B in the network topology given in Q2 has been considered here. The loss percentage value for each of the links in connection A-B was set to 1%, 2%, and 5% and tested in three iterations respectively.

From the above plot, it can be observed that the throughput bandwidth decreases as the loss percentage increases. This is because more packets get lost while being transmitted across the connection, leading to a decreased value of observed throughput.

Although not shown in the graph, the total time for which the connection remains is observed to be more than 120 seconds, as the lost packets get retransmitted.

Q4.

a. The plot between SlowStart Threshold (measured in segments) and RTT interval is given below



b. We cannot determine whether the congestion control scheme is TCP Tahoe or TCP Reno based solely on the graph. In TCP Reno, if a packet loss is observed due to triple-ACK, the cwnd is decreased to half of its value. However, if the packet loss is observed due to time-out, the cwnd returns to its initial value. In the case of TCP Tahoe, the cwnd returns to its initial value, i.e. 1, in packet loss occurring due to either triple-ACK or time-out methods. Therefore, the cwnd value will drop to half of its previous value if the packet loss happens due to triple ACKs. Likewise, in the event of a timeout, TCP Reno's cwnd value will reset to its initial value.

c. This condition will hold true when the congestion window is the largest in size. For the given graph, the maximum number of packets = 24 in a time interval, RTT = 33units.

d. For a given instance, the minimum number of packet losses is 0, and the maximum number of packet losses would be 24 (when all the packets get lost for the maximum increase in congestion window). However, if we consider the whole session (from RTT = 0-40 units), the minimum number of packet losses will be 4

(1 packet loss corresponding to every minimum). And the maximum number of packet loss would be 62 (9, 13, 24, and 16 numbers of packets corresponding to every peak in the graph).

e. In interval 33, if the packet loss were due to 3Dup ACKs in a TCP Tahoe system, then the graph would have had no change. This is because the cwnd value drops to its initial value, as already depicted in the graph. However, if it were a TCP Reno system, the packet loss due to 3Dup ACKs would have dropped to half of its value. Therefore, the graph would have dropped from 16 to 8 congestion-size windows.

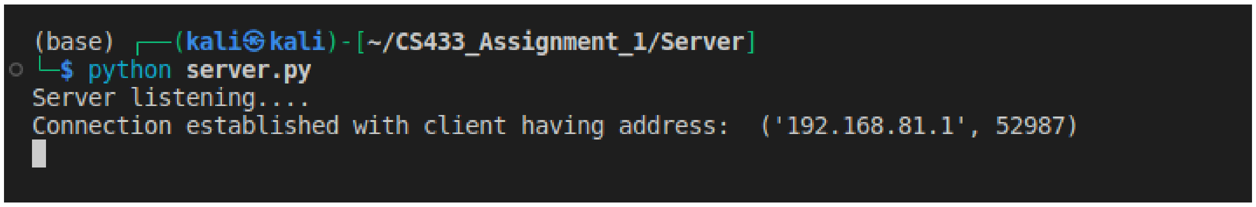
Q5.

It will not be possible to run the code written in Assignment 1 to establish a connection between a client and a server running on the host machine (Windows) and the Kali Linux VM respectively. In Assignment 1, while writing the code, we had chosen a random IP address for the server and let the client connect to the server with the same IP. This worked because both the client and server operated on the same system and shared the same network.

In case the server and the client live in different systems, their local IP addresses are no longer visible to each other, though they share the same network. In order to make the desired connection work, we need to change the network layer settings of the client and the server. We now need to work with the visible IPv4 address of the server that can be found by running the *ifconfig* command on the Kali Linux VM server. The obtained address was *192.168.81.129*

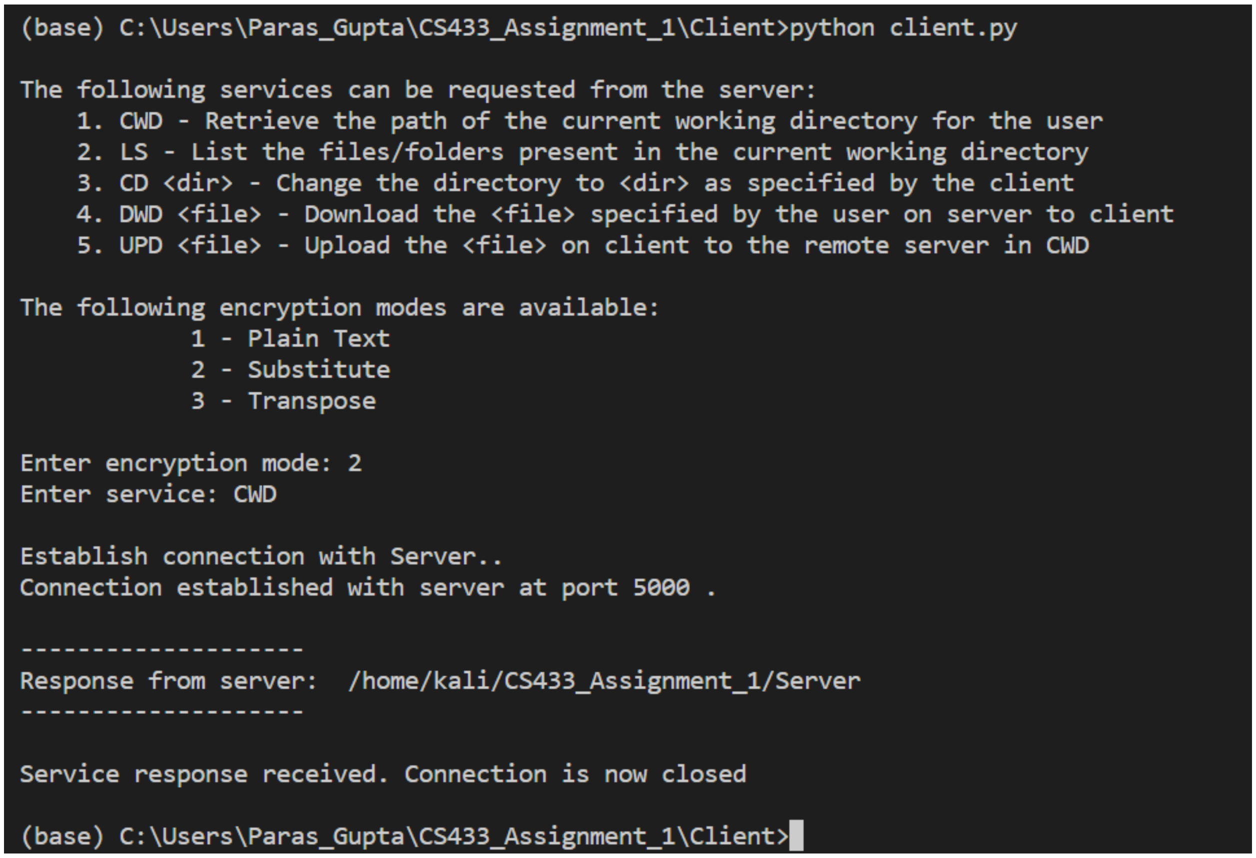
The IP address of the server was updated in the *server.py* file and a port number was chosen (*5000*). The *client.py* file in the host machine was then updated to bind it with the new IP address and port number. Thereby, a logical connection was established between the client and the server over the shared network.

The following images show an example of the established connection between the different systems. A simple 'CWD' service was requested from the server using the 'substitute' encryption mode'. The server responded with the current working directory "*/home/kali/CS433_Assignment_1/Server*" as can be seen in the client terminal below.



```
(base) ┌──(kali㉿kali)-[~/CS433_Assignment_1/Server]
└─$ python server.py
Server listening....
Connection established with client having address:  ('192.168.81.1', 52987)
```

Server at Kali Linux VM



```
(base) C:\Users\Paras_Gupta\CS433_Assignment_1\Client>python client.py

The following services can be requested from the server:
    1. CWD - Retrieve the path of the current working directory for the user
    2. LS - List the files/folders present in the current working directory
    3. CD <dir> - Change the directory to <dir> as specified by the client
    4. DWD <file> - Download the <file> specified by the user on server to client
    5. UPD <file> - Upload the <file> on client to the remote server in CWD

The following encryption modes are available:
            1 - Plain Text
            2 - Substitute
            3 - Transpose

Enter encryption mode: 2
Enter service: CWD

Establish connection with Server..
Connection established with server at port 5000 .

--------------------
Response from server:  /home/kali/CS433_Assignment_1/Server
--------------------

Service response received. Connection is now closed

(base) C:\Users\Paras_Gupta\CS433_Assignment_1\Client>
```

Client at Host Machine (Windows)