VE489 Project Report

Name: Chenyun Tao Student ID: 517370910072

1 Mininet Experiments

1.1 Link latency using `ping`

The result of measuring the latency between the pair (h_1, h_2) is shown in Figure 1. Here the link L_1 is used, which has a delay of 20ms. From Figure 1, we can see RTT is 43.509ms in average, which is close to 2 times link latency as 40ms.

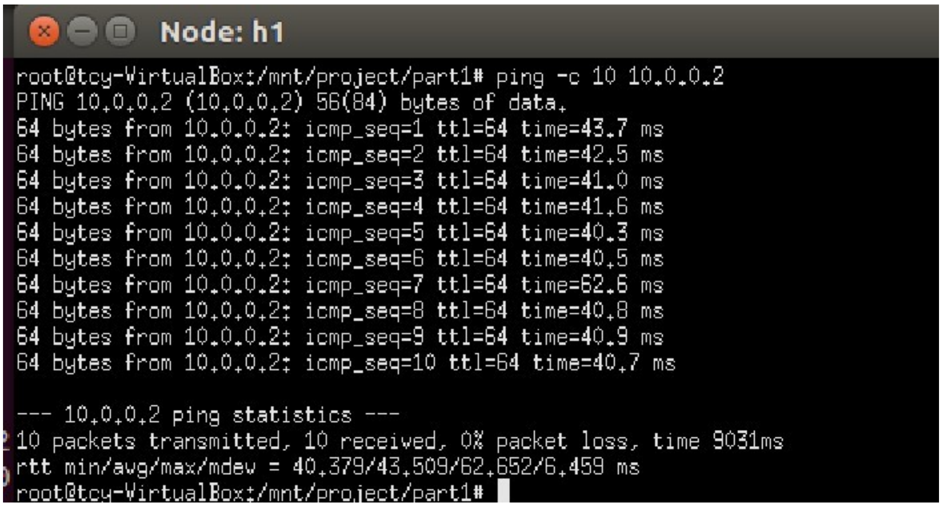


Figure 1: Latency between the pair (h_1, h_2)

The result of measuring the latency between the pair (h_3, h_5) is shown in Figure 2. Here the link L_4 is used, which has a delay of 10ms. From Figure 2, we can see RTT is 22.759ms in average, which is close to 2 times link latency as 20ms.

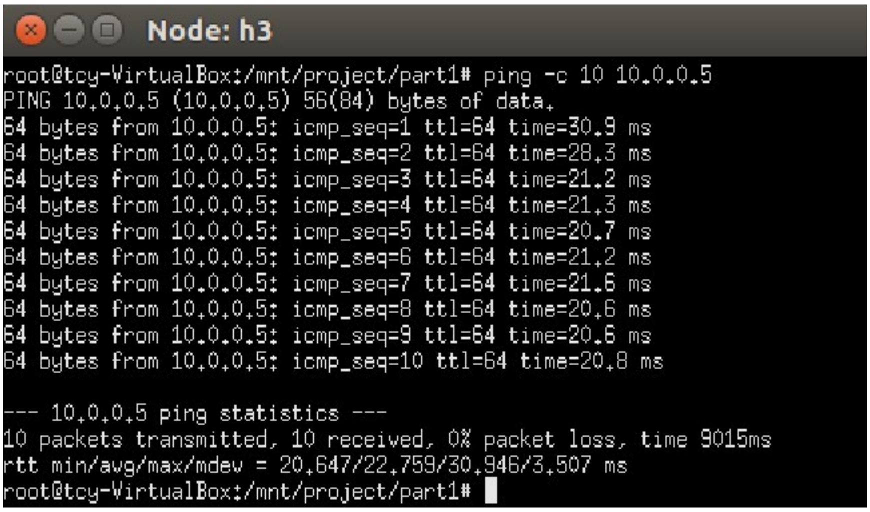


Figure 2: Latency between the pair (h_3, h_5)

1.2 Path latency using `ping`

The result of measuring the latency between the pair (h_1, h_5) is shown in Figure 3. Here the links L_1, L_2, L_4 are used, and the overall theoretical delay is $20 + 40 + 10 = 70$ms. From Figure 3, we can see RTT is 145.184ms in average, which is close to 2 times path latency as 140ms.

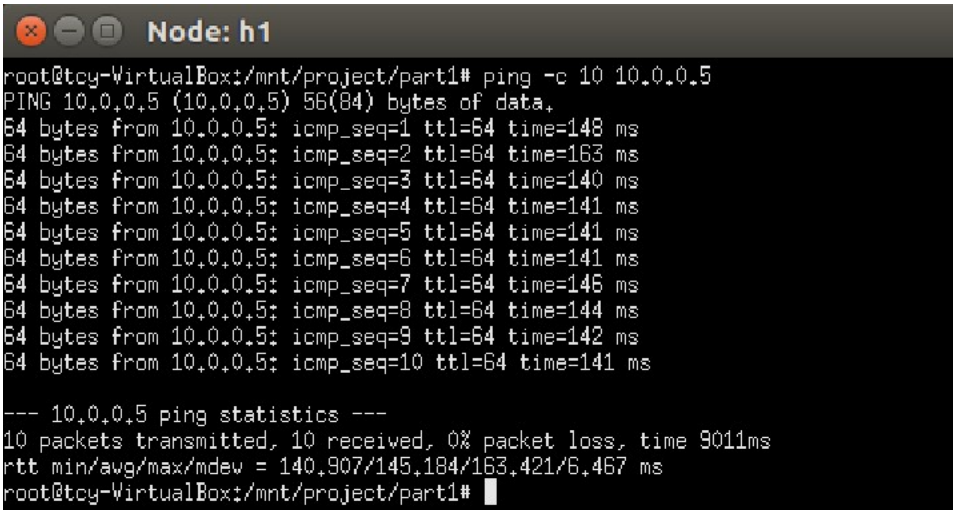


Figure 3: Latency between the pair (h_1, h_5)

The result of measuring the latency between the pair (h_3, h_4) is shown in Figure 4. Here the link L_2, L_3 are used, and the overall theoretical delay is $40 + 30 = 70$ms. From Figure 4, we can see RTT is 144.801ms in average, which is close to 2 times path latency as 140ms.

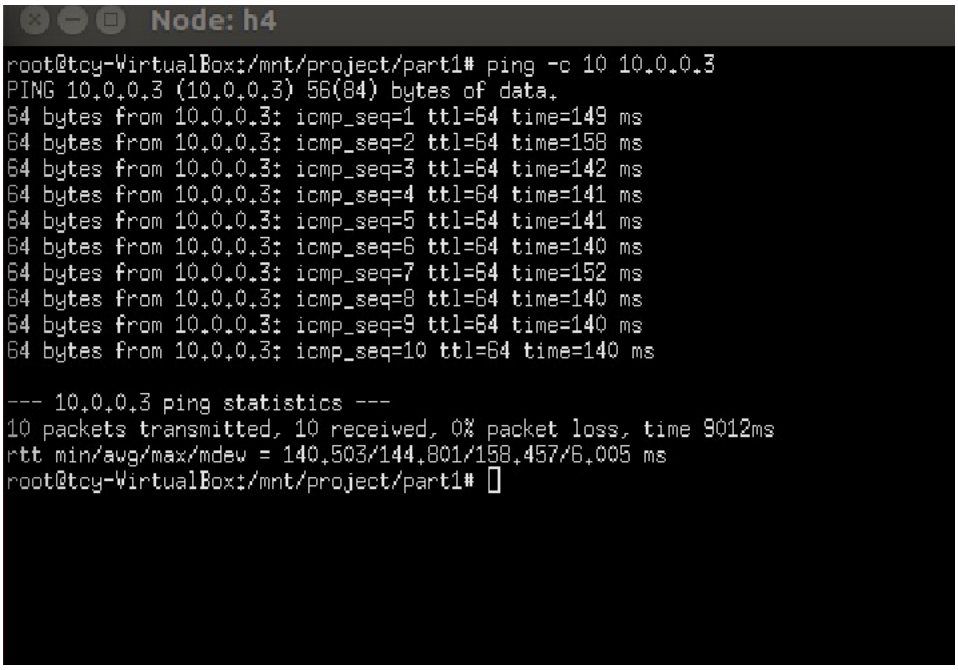


Figure 4: Latency between the pair (h_3, h_4)

1.3 Link bandwidth using `iperf`

The result of measuring the bandwidth between hosts (h_1, h_2) is shown in Figure 5. Here h_1 runs the server mode, and h_2 runs the client mode. From Figure 5, we can see the bandwidth calculated on server is 46.1Mbps, and the bandwidth calculated on client is 48.4Mbps. Both of them are close to the link L_1's bandwidth as 50Mbps. 58.6 Mbytes data are transferred and 58.6 Mbytes data are received, which are equal.

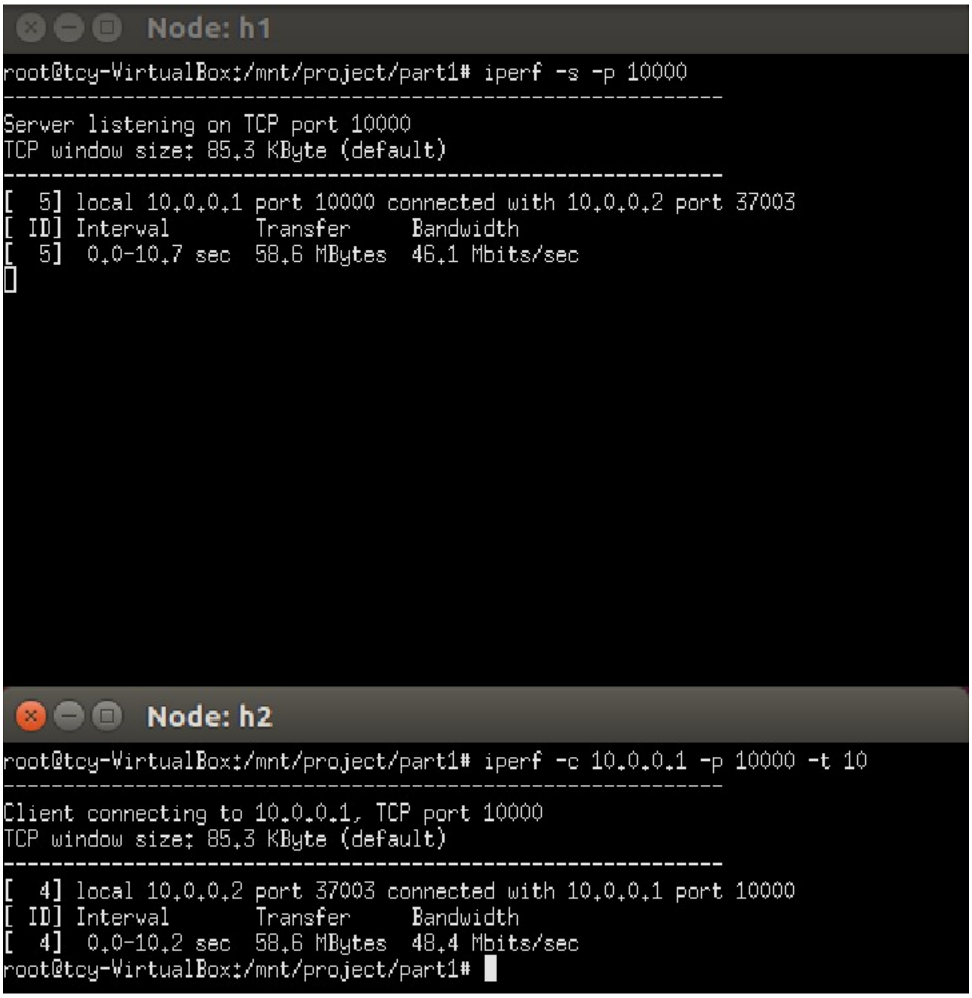


Figure 5: Bandwidth between the hosts (h_1, h_2)

The result of measuring the bandwidth between hosts (h_3, h_5) is shown in Figure 6. Here h_3 runs the server mode, and h_5 runs the client mode. From Figure 6, we can see the bandwidth calculated on server is 9.02Mbps, and the bandwidth calculated on client is 10.6Mbps. Both of them are close to the link L_4's bandwidth as 10Mbps. 13.1 Mbytes data are transferred and 13.1 Mbytes data are received, which are equal.

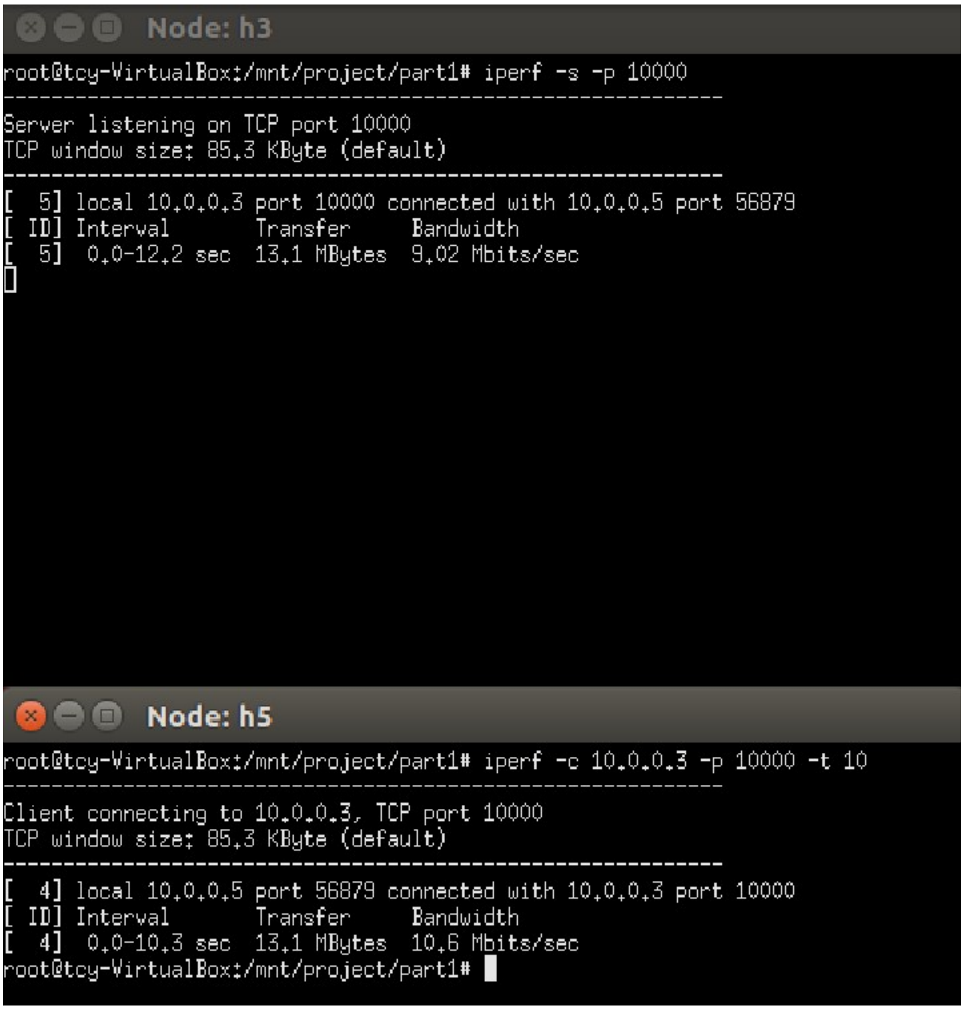


Figure 6: Bandwidth between the hosts (h_3, h_5)

1.4 Path throughput using `iperf`

The result of measuring the bandwidth between hosts (h_1, h_5) is shown in Figure 7. Here h_1 runs the server mode, and h_5 runs the client mode. From Figure 7, we can see the bandwidth calculated on server is 9.13Mbps, and the bandwidth calculated on client is 11.5Mbps. The path goes throught the link L_1, L_2, L_4, which has the bandwidth of 50Mbps, 20Mbps, and 10Mbps, respectively. Here L_4 is the bottleneck link in this path, and the result is close to its bandwidth. 14.4 Mbytes data are transferred and 14.4 Mbytes data are received, which are equal.

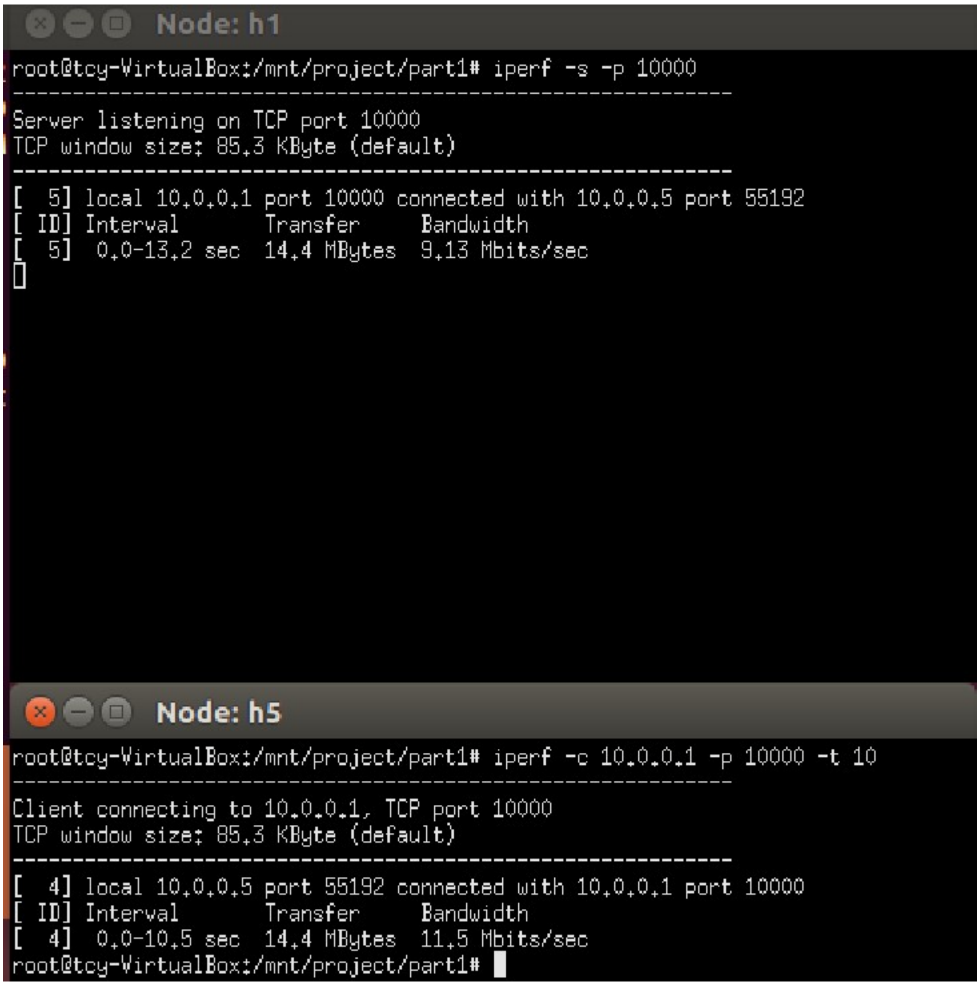


Figure 7: Bandwidth between the hosts (h_1, h_5)

The result of measuring the bandwidth between hosts (h_3, h_4) is shown in Figure 8. Here h_4 runs the server mode, and h_3 runs the client mode. From Figure 8, we can see the bandwidth calculated on server is 18.2Mbps, and the bandwidth calculated on client is 21.4Mbps. The path goes throught the link L_2, L_3, which has the bandwidth of 20Mbps and 60Mbps, respectively. Here L_2 is the bottleneck link in this path, and the result is close to its bandwidth. 26.0 Mbytes data are transferred and 26.0 Mbytes data are received, which are equal.

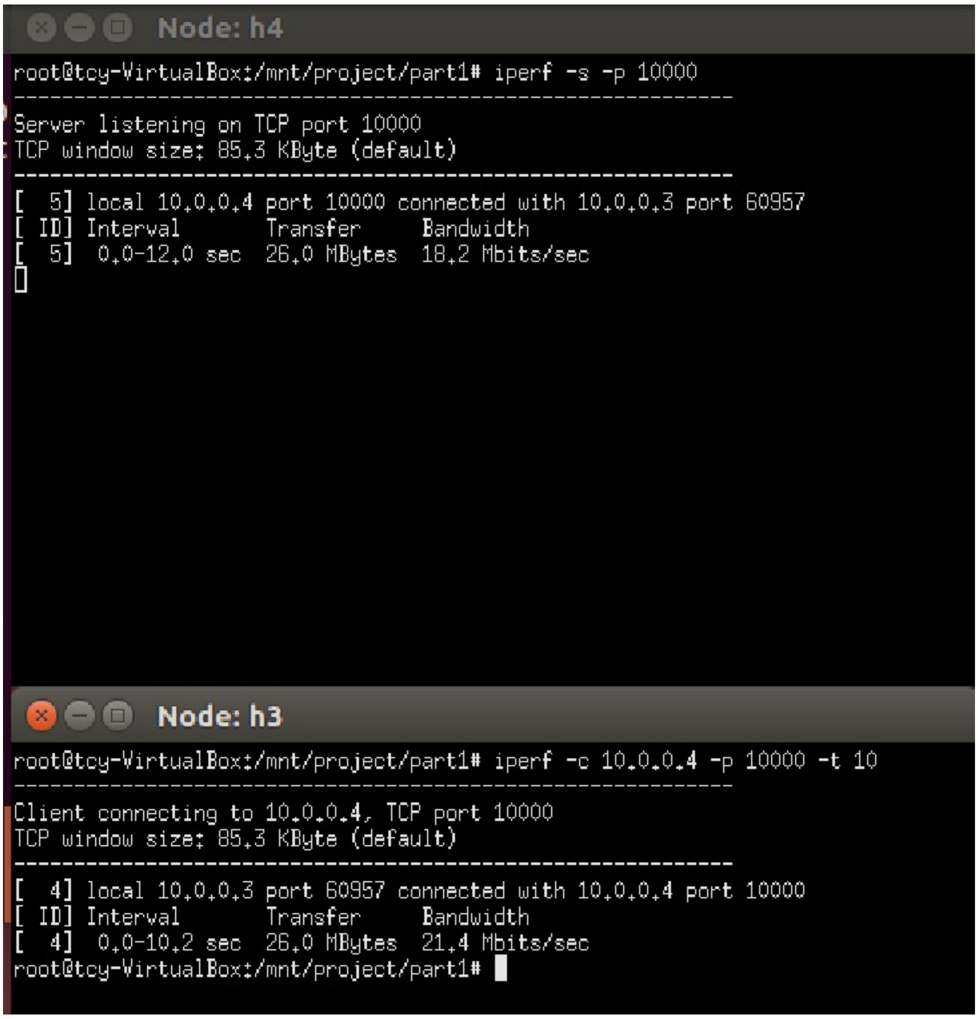


Figure 8: Bandwidth between the hosts (h_3, h_4)

1.5 Multiplexing

Suppose two pairs of hosts (h_1, h_5) and (h_3, h_4) are sharing the same link. There are 2 cases. One is that the two pairs share the link and transfer data in different directions, and the other one is that they share the link in exactly the same direction.

1.5.1 Sharing the same link in different directions

The result of measuring the latency between hosts (h_1, h_5) and (h_3, h_4) in this case is shown in Figure 9. From Figure 9, we can see RTT in average is 142.238ms and 141.280ms, respectively, corresponding to the latency of 71.119ms and 70.640ms. The results are close to the previous results in question 2, as the data is transferred in different directions.

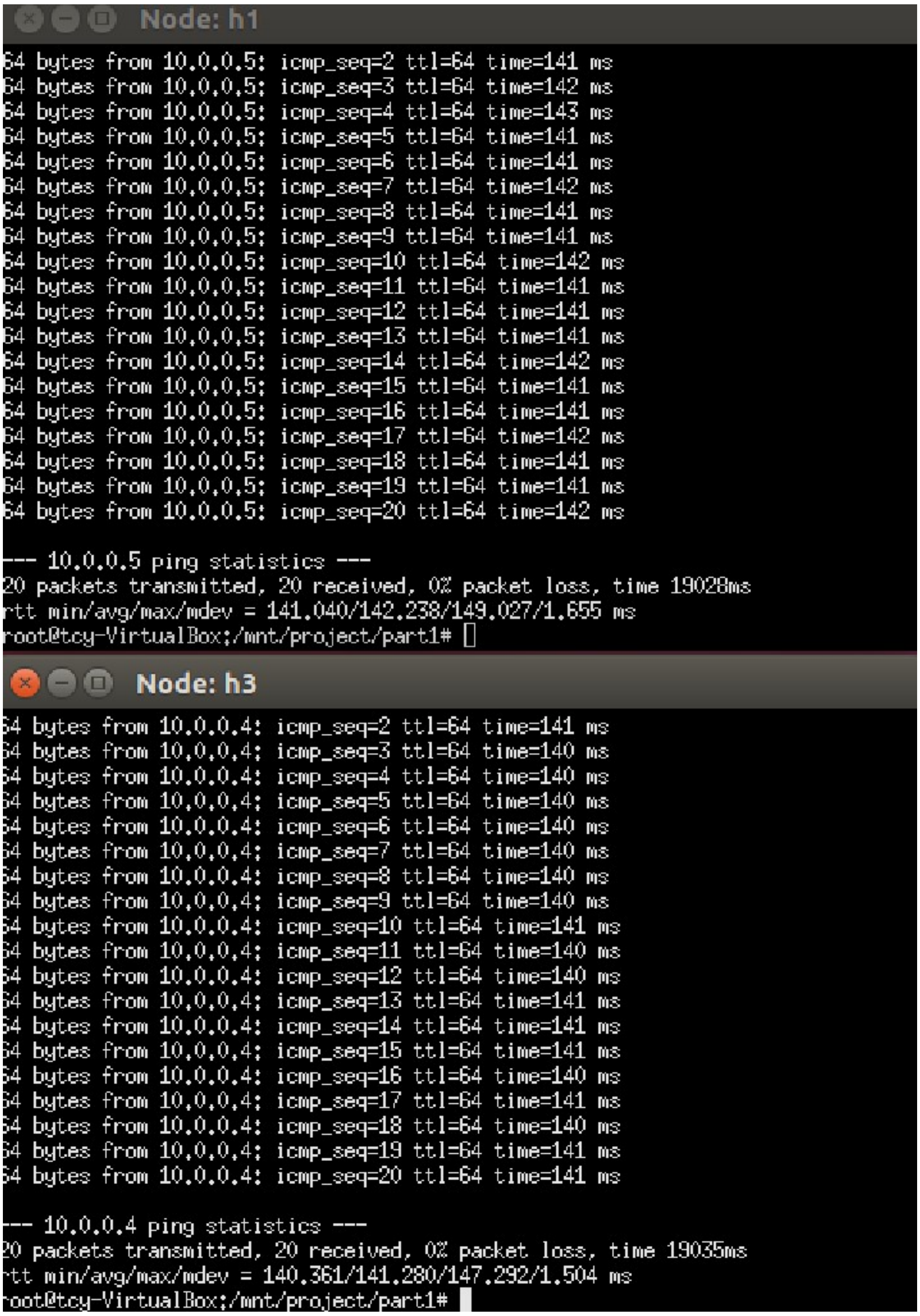


Figure 9: Latency between (h_1, h_5) and (h_3, h_4) sharing the same link, data transferred in different directions

The result of measuring the bandwidth between hosts (h_1, h_5) and (h_3, h_4) in this case is shown in Figure 10. Here h_1, h_3 runs the server mode, and h_5, h_4 runs the client mode. From Figure 10, we can see for (h_1, h_5), the bandwidth calculated on server is 7.42Mbps, and the bandwidth calculated on client is 8.78Mbps. For (h_3, h_4), the bandwidth calculated on server is 18.6Mbps, and the bandwidth calculated on client is 19.8Mbps. The bandwidths are all close to 10Mbps. The results are close to the previous results in question 4, as the data is transferred in different directions.

Link L_2 is shared, and (h_1, h_5) and (h_3, h_4) actually does not have to share the bandwidth in this case.

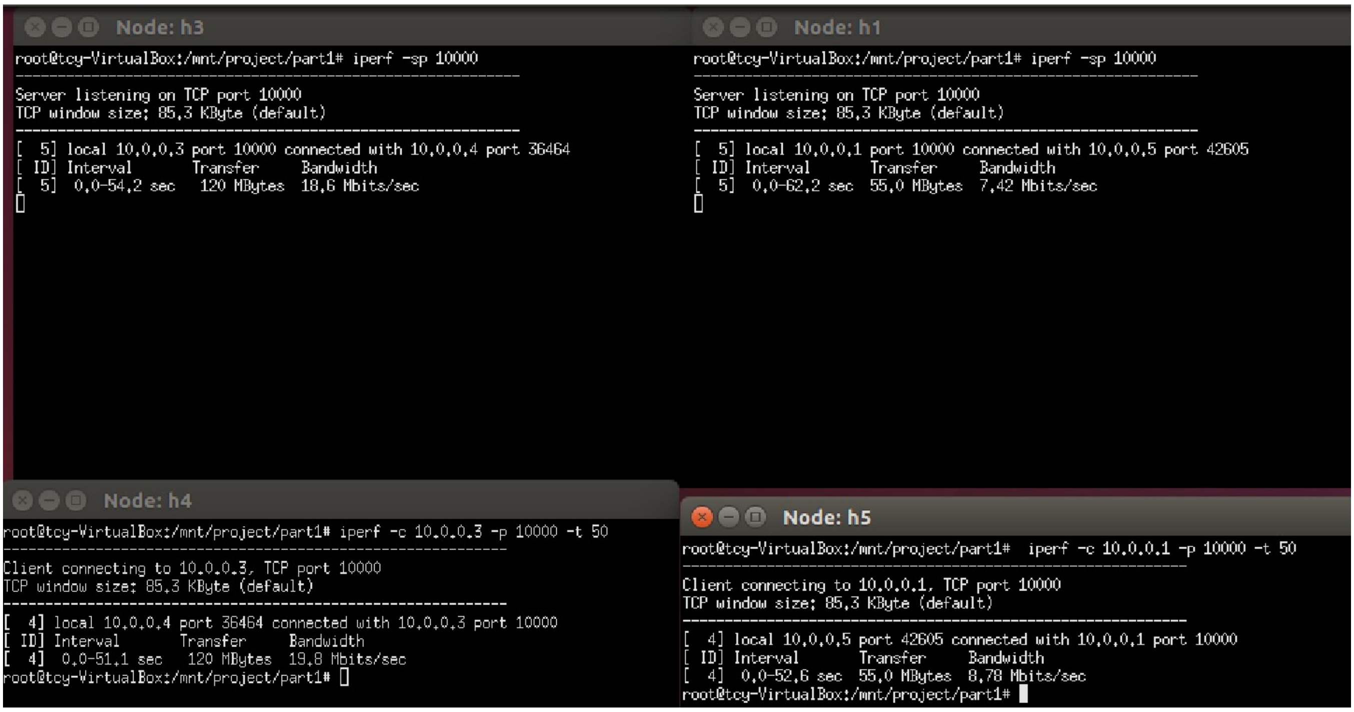


Figure 10: Bandwidth between (h_1, h_5) and (h_3, h_4) sharing the same link, data transferred in different directions

1.5.2 Sharing the same link in the same direction

The result of measuring the latency between hosts (h_1, h_5) and (h_3, h_4) in this case is shown in Figure 11. From Figure 11, we can see RTT in average is 141.092ms and 140.657ms, respectively, corresponding to the latency of 70.546ms and 70.3285ms. The results are close to the previous results in question 2. The reason that the latency does not change is that the data used by `ping` is small, and does not reach the limit of the bandwidth.

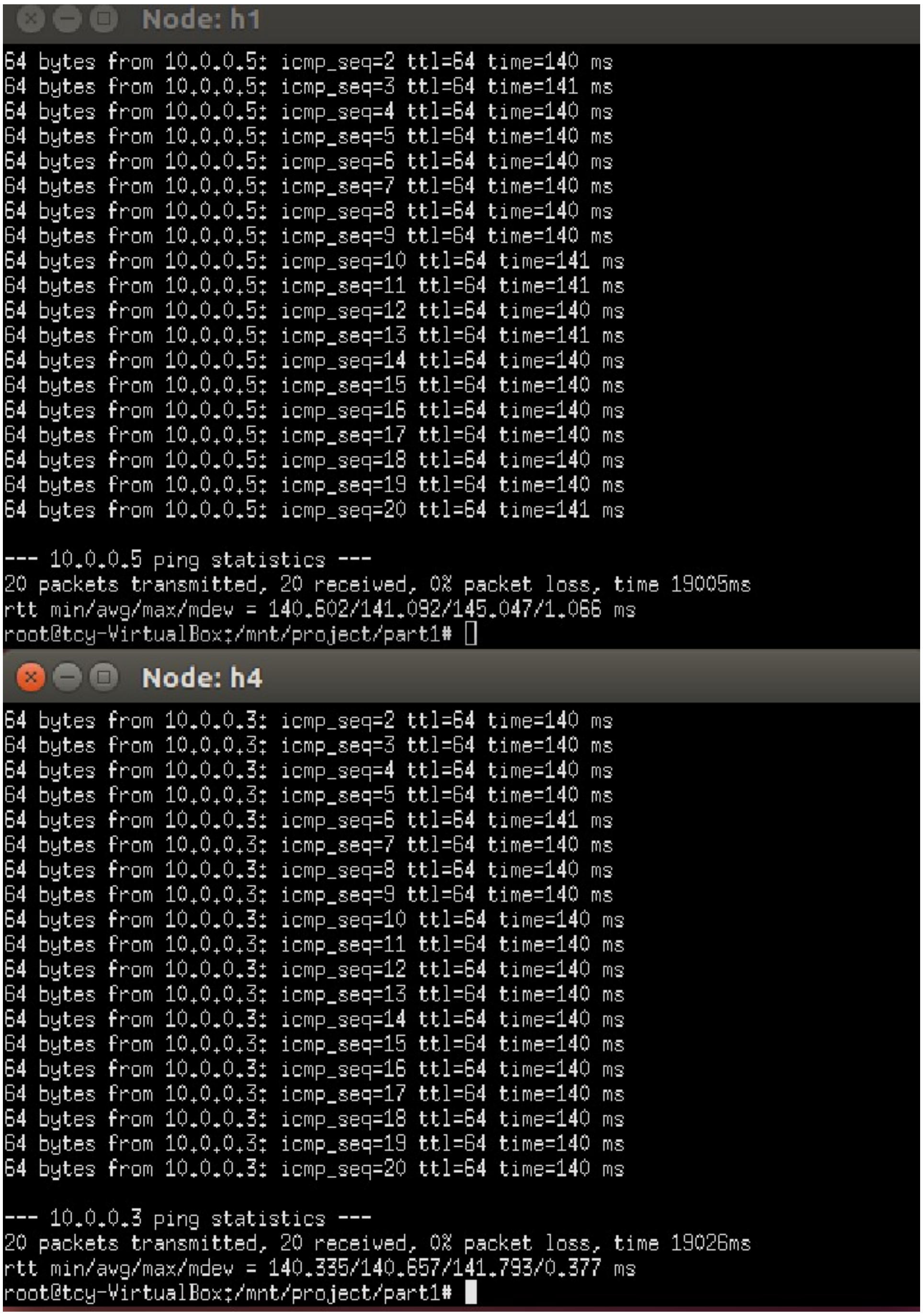


Figure 11: Latency between (h_1, h_5) and (h_3, h_4) sharing the same link, data transferred in the same direction

The result of measuring the bandwidth between hosts (h_1, h_5) and (h_3, h_4) in this case is shown in Figure 12. Here h_1, h_4 runs the server mode, and h_5, h_3 runs the client mode. From Figure 12, we can see for (h_1, h_5), the bandwidth calculated on server is 8.95Mbps, and the bandwidth calculated on client is 9.84Mbps. For (h_3, h_4), the bandwidth calculated on server is 9.95Mbps, and the bandwidth calculated on client is 11.0Mbps. The bandwidths are all close to 10Mbps. The results are smaller than the previous results in question 4. Here L_2's bandwidth, 20Mbps, is smaller than the sum of the bandwidth of (h_1, h_5) and (h_3, h_4) when the pairs are connecting separately, which is 30Mbps, and thus has to be split for the two pairs sharing this link. Since the path throughput is determined by the bottleneck link, and now L_2 is the bottleneck for both of the pairs, the bandwidth changes.

Link L_2 in the direction (s_2, s_3) is shared, and (h_1, h_5) and (h_3, h_4) share the link bandwidth fairly.

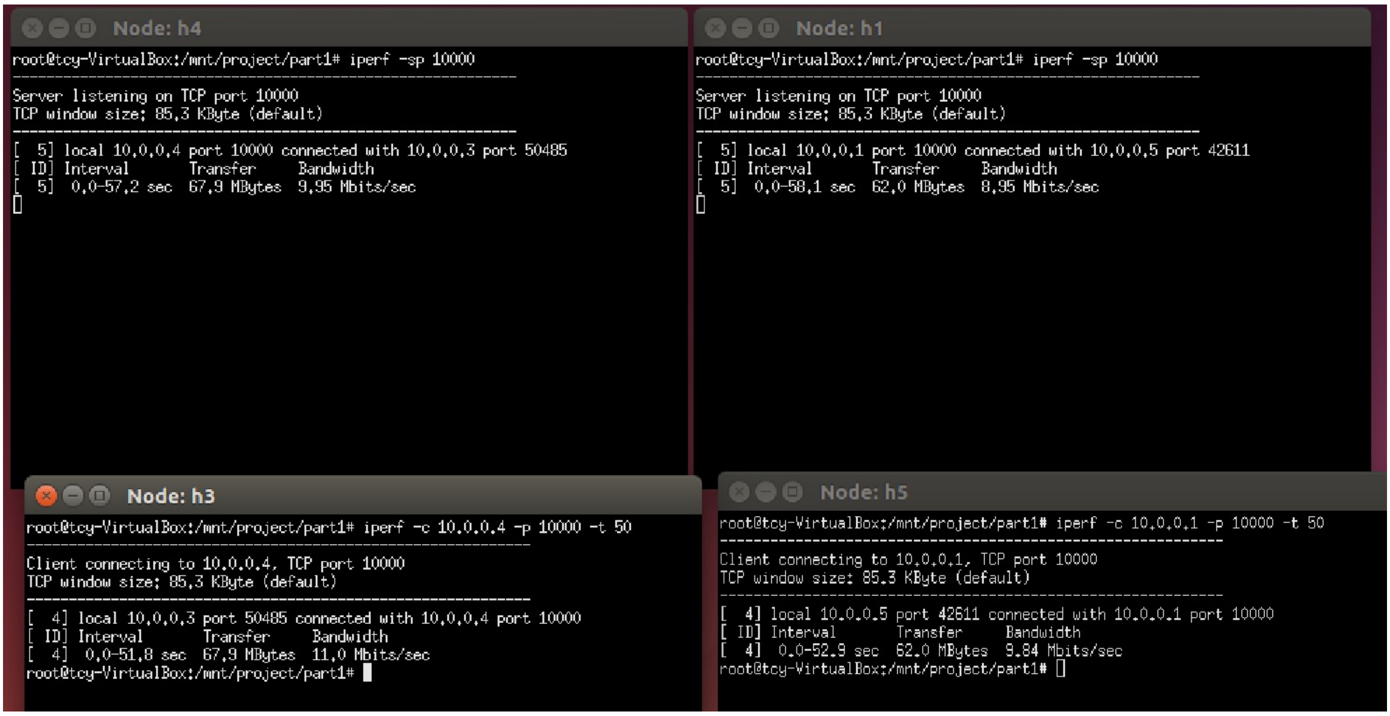


Figure 12: Bandwidth between (h_1, h_5) and (h_3, h_4) sharing the same link, data transferred in the same direction

2 TCP File Transfer Server

In this part, a TCP file transfer server `ftrans` is implemented.

To test the code, I create 3 files with various size ranging from roughly {100B, 100KB, 10MB} (see Figure 13) and transfer them. Client is at 10.0.0.2:8002 and server is at 10.0.0.1:8001. The server runs directly in the directory `part2\`, and the client runs in the directory `part2\client`. The screenshots of the transfering process are shown in Figure 14, 15 and 16. By using `diff` to check the difference, we can see that the delivered and original files are just the same (see Figure 17), which means the files are transferred successfully.

```
tcy1999@DESKTOP-CFUBOFV:/mnt/c/tcy1999/VE489/Project/finalProject/part2$ wc -c 100B.txt
104 100B.txt
tcy1999@DESKTOP-CFUBOFV:/mnt/c/tcy1999/VE489/Project/finalProject/part2$ wc -c 100KB.txt
102493 100KB.txt
tcy1999@DESKTOP-CFUBOFV:/mnt/c/tcy1999/VE489/Project/finalProject/part2$ wc -c 10MB.txt
10422129 10MB.txt
```

Figure 13: 3 files with various size ranging from roughly {100B, 100KB, 10MB}

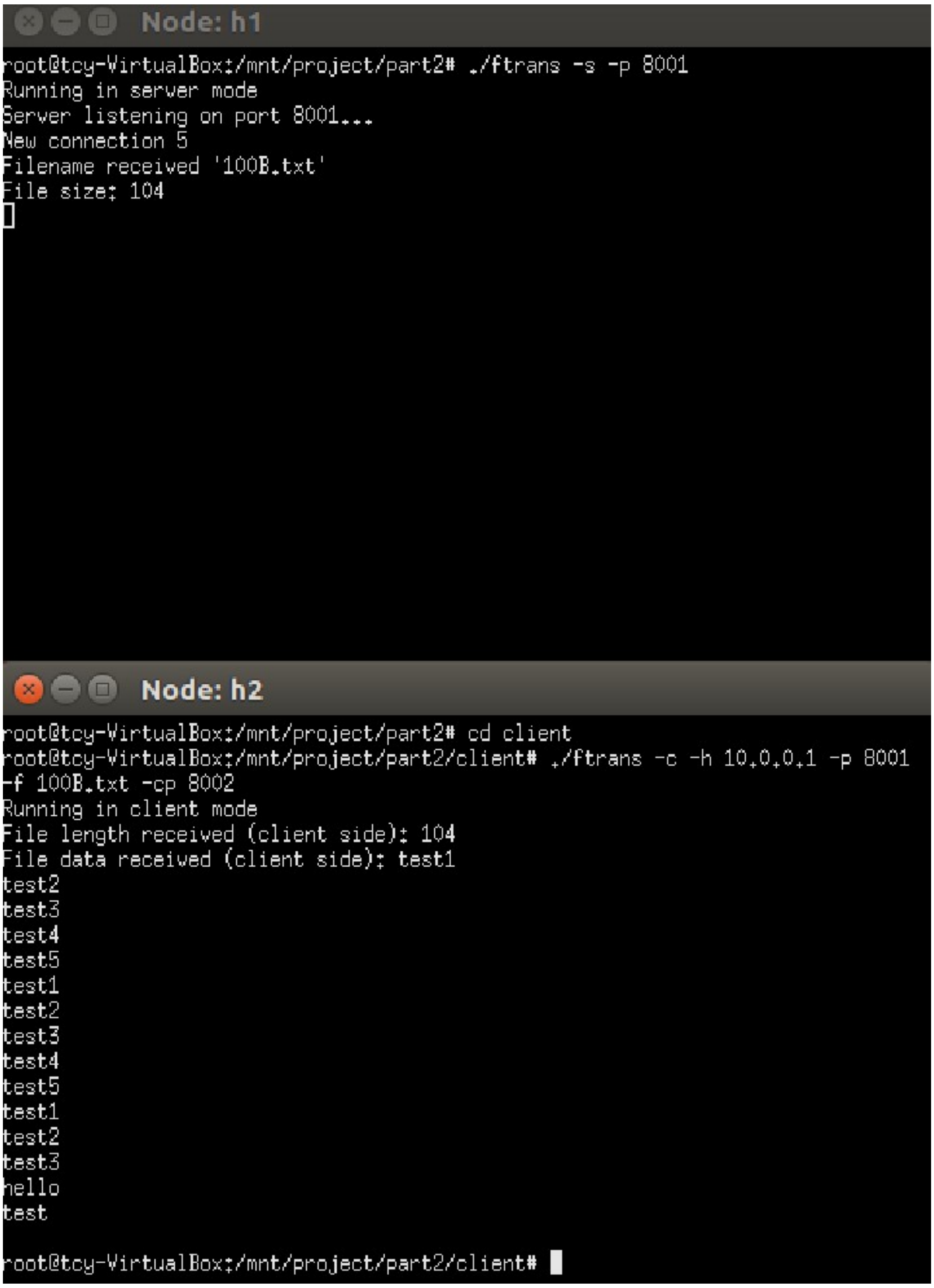


Figure 14: Transferring the file with the size 100B

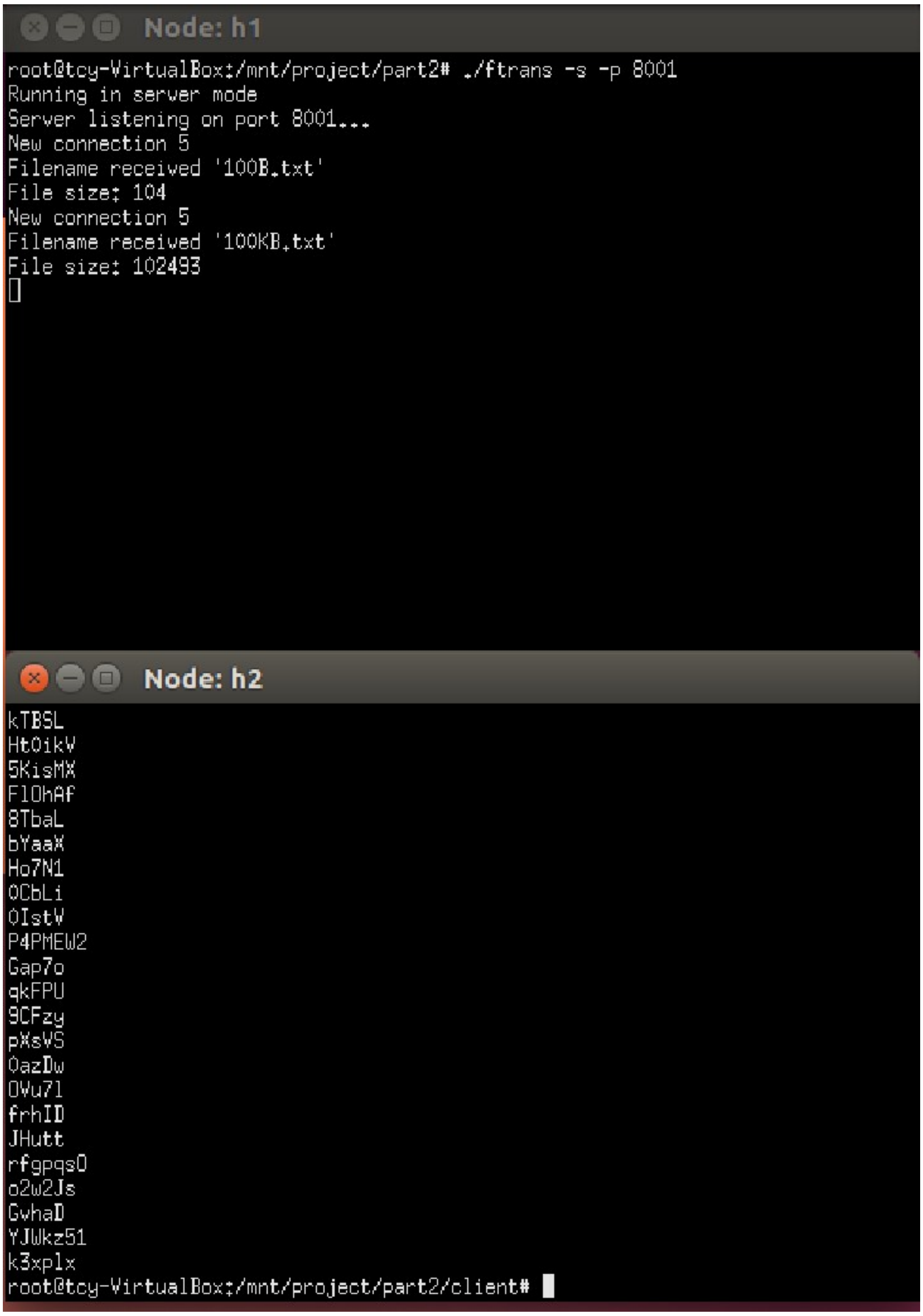


Figure 15: Transferring the file with the size 100KB

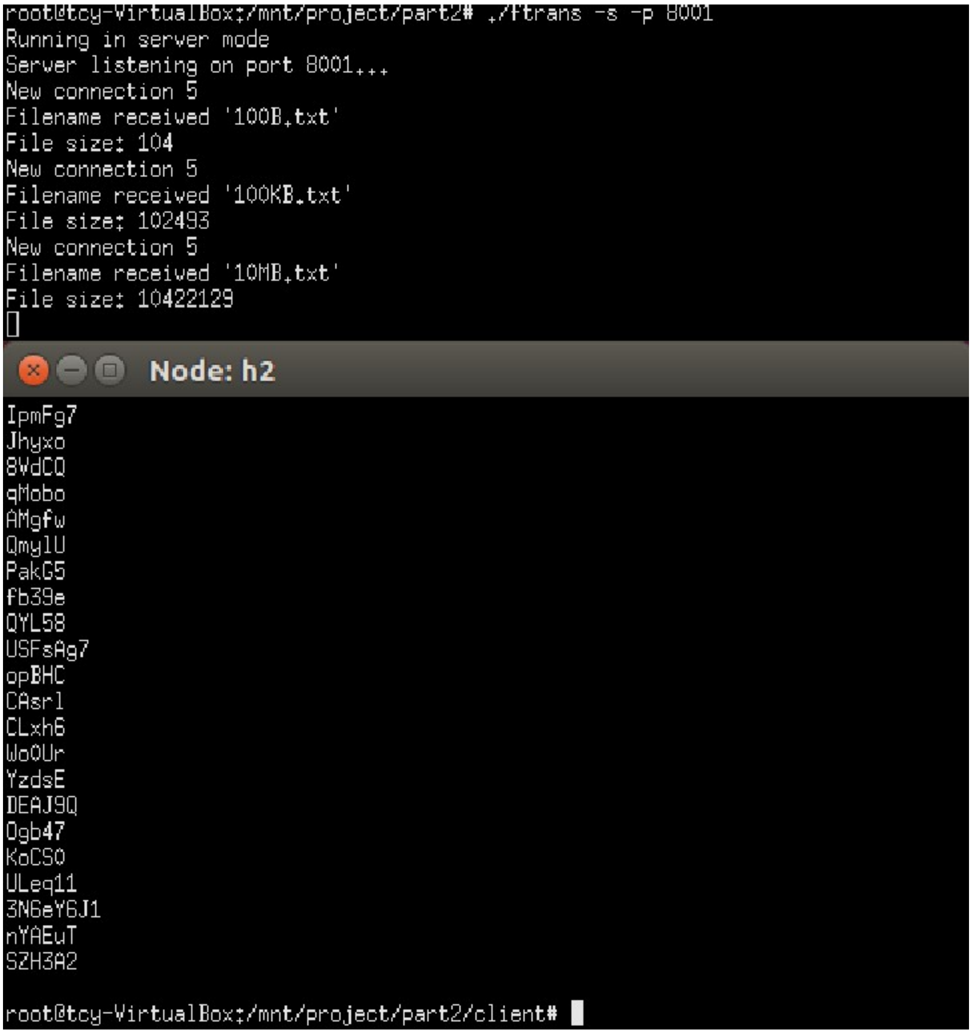


Figure 16: Transferring the file with the size 10MB

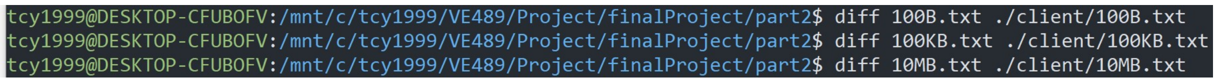
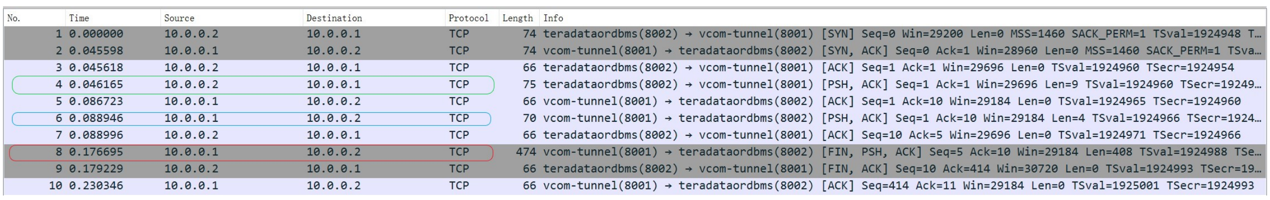


Figure 17: Using **diff** to check the difference between the delivered and original files

I also transfer the `Makefile` to check the correctness, and here I capture the packets sent and received on client with `tcpdump`. Dumping the output into a `.pcap` file and opening it with wireshark, the result is shown in Figure 18. The packet in green contains the requested filename (detail in Figure 19), the packet in blue contains the file size (detail in Figure 20), and the packet in red contains the file data (detail in Figure 21).



No.	Time	Source	Destination	Protocol	Length	Info
1	0.000000	10.0.0.2	10.0.0.1	TCP	74	teradataordbms(8002) → vcom-tunnel(8001) [SYN] Seq=0 Win=29200 Len=0 MSS=1460 SACK_PERM=1 TSval=1924948 T…
2	0.045598	10.0.0.1	10.0.0.2	TCP	74	vcom-tunnel(8001) → teradataordbms(8002) [SYN, ACK] Seq=0 Ack=1 Win=28960 Len=0 MSS=1460 SACK_PERM=1 TSva…
3	0.045618	10.0.0.2	10.0.0.1	TCP	66	teradataordbms(8002) → vcom-tunnel(8001) [ACK] Seq=1 Ack=1 Win=29696 Len=0 TSval=1924960 TSecr=1924954
4	0.046165	10.0.0.2	10.0.0.1	TCP	75	teradataordbms(8002) → vcom-tunnel(8001) [PSH, ACK] Seq=1 Ack=1 Win=29696 Len=9 TSval=1924960 TSecr=19249…
5	0.086723	10.0.0.1	10.0.0.2	TCP	66	vcom-tunnel(8001) → teradataordbms(8002) [ACK] Seq=1 Ack=10 Win=29184 Len=0 TSval=1924965 TSecr=1924960
6	0.088946	10.0.0.1	10.0.0.2	TCP	70	vcom-tunnel(8001) → teradataordbms(8002) [PSH, ACK] Seq=1 Ack=10 Win=29184 Len=4 TSval=1924966 TSecr=1924…
7	0.088996	10.0.0.2	10.0.0.1	TCP	66	teradataordbms(8002) → vcom-tunnel(8001) [ACK] Seq=10 Ack=5 Win=29696 Len=0 TSval=1924971 TSecr=1924966
8	0.176695	10.0.0.1	10.0.0.2	TCP	474	vcom-tunnel(8001) → teradataordbms(8002) [FIN, PSH, ACK] Seq=5 Ack=10 Win=29184 Len=408 TSval=1924988 TSe…
9	0.179229	10.0.0.2	10.0.0.1	TCP	66	teradataordbms(8002) → vcom-tunnel(8001) [FIN, ACK] Seq=10 Ack=414 Win=30720 Len=0 TSval=1924993 TSecr=19…
10	0.230346	10.0.0.1	10.0.0.2	TCP	66	vcom-tunnel(8001) → teradataordbms(8002) [ACK] Seq=414 Ack=11 Win=29184 Len=0 TSval=1925001 TSecr=1924993

Figure 18: The packets sent and received on client

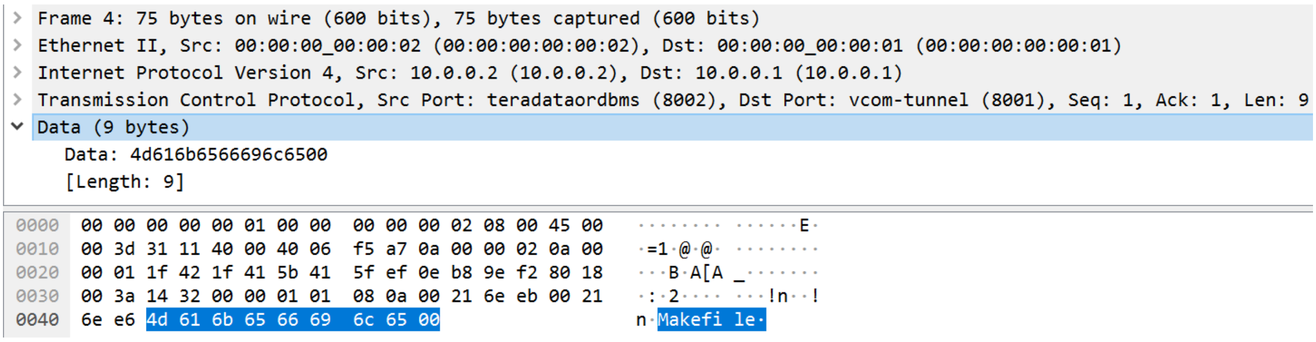


Figure 19: The packet that contains the requested filename

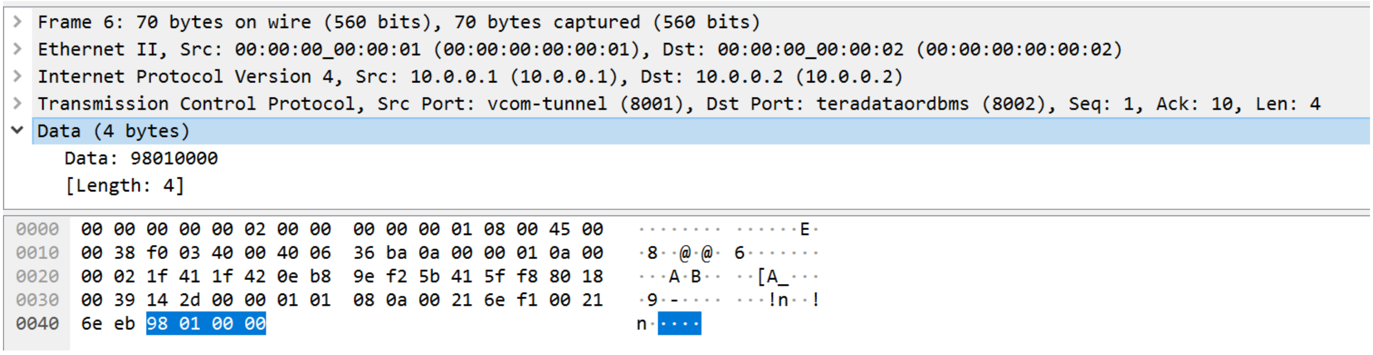


Figure 20: The packet that contains the file size, which is a 4-byte int

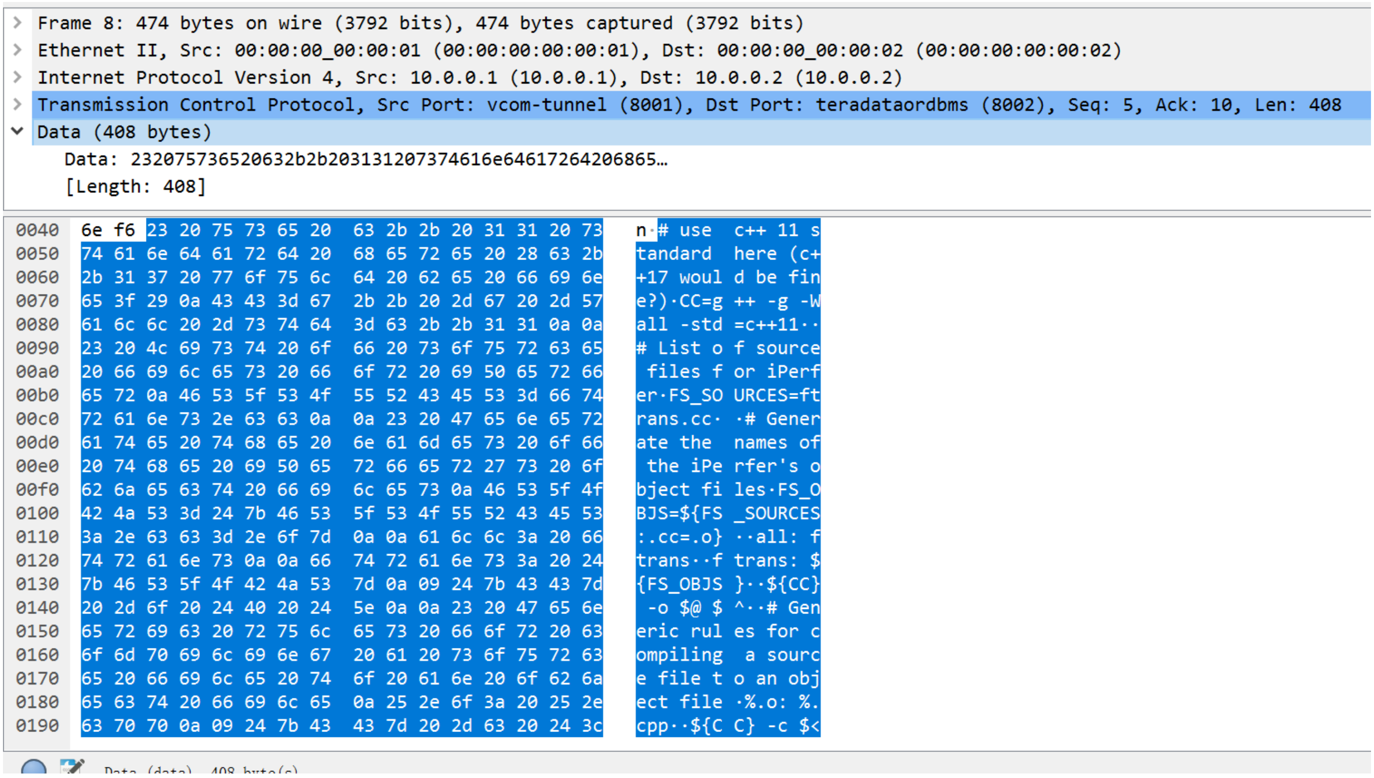


Figure 21: The packet that contains the the file data, 408 bytes in total

3 Reliable Transmission

3.1 Simple SR

In this section, window size is set to be 10 on both sender and receiver. A 10MB file is transferred. A part of log on sender and receiver will be used to analyze the result, and everything after # is added afterwards as comments.

3.1.1 No reordering, no loss, no error

The first few lines of log on sender and receiver in this case are shown below. When receiving an ACK packet, the sender's window is slided one step forward so that an outstanding packet is appended at the end of the window, and sent to the receiver. When receiving a data packet, the receiver's window is also slided one step forward.

The windows slide on both sides correctly. The reason is that the packets encounter no reordering, no loss, and no error, so they are just transferred in order. The seq of data packets and ACK packets increase 1 each time. In principle, we expect that when receiving an ACK packet, the sender's window is slided so that the beginning of the window is advanced to buffer the packet with the ACK's seq. The packets before this packet are moved from the window, and some outstanding packets are sent to the receiver and buffered at the end of the window. In this case, since ACKs increase 1 each time, proceeding one step conforms to the principle. When receiving a data packet in this case, the highest in-order sequence number in the receiver's window is just the data packet's seq, since everything comes before is in order and has been appended to file. Thus, the window is slided one step forward.

```
# sender log
SYN 543567240 0 0
ACK 543567240 0 0
DATA 0 1456 -1241856968
DATA 1 1456 650524154
DATA 2 1456 1532157781
DATA 3 1456 975835324
DATA 4 1456 524171520
DATA 5 1456 -2048262358
DATA 6 1456 -1316030808
DATA 7 1456 -2054977684
DATA 8 1456 1425245197
DATA 9 1456 35173067
ACK 1 0 0 # receive ACK for packet 1
DATA 10 1456 -1954964501 # window is slided, packet 10 is thus sent.
ACK 2 0 0 # receive ACK for packet 2
DATA 11 1456 1655566761 # window is slided, packet 11 is thus sent.
...
FIN 543567240 0 0
ACK 543567240 0 0

# receiver log
SYN 543567240 0 0
ACK 543567240 0 0
DATA 0 1456 -1241856968 # receive packet 0
ACK 1 0 0 # ACK 1 (and slide window)
DATA 1 1456 650524154 # receive packet 1
ACK 2 0 0 # ACK 2 (and slide window)
...
```

```
FIN 543567240 0 0
ACK 543567240 0 0
```

3.1.2 10% loss, no reordering, no error

We can see in sender's log, some packets that are not ACKed due to packet loss, for instance, data packet 1. The whole window is thus retransmitted and ACKed later, as here 10 data packets in the window (seq 1 from to 10) are resent, and after ACK 6 for packet 6 is received, which means every packet before 6 has been received (including the previous dropped packet 1), the window is advanced.

On receiver's log, there are gaps in its window due to packet loss. For example, after receiving data packet 0, we expect to receive packet 1, but receive packet 2, 3, ..., and thus there is a gap in the receiver's window for packet 1. As a result of this gap, the ACK number it sends back to the sender is 1, corresponding to packet 1. Until the receiver finally receives packet 1, it keeps sending ACK 1, and does not advance the window.

```
# sender log
...
DATA 0 1456 -1241856968
DATA 1 1456 650524154
DATA 2 1456 1532157781
DATA 3 1456 975835324
DATA 4 1456 524171520
DATA 5 1456 -2048262358
DATA 6 1456 -1316030808
DATA 7 1456 -2054977684
DATA 8 1456 1425245197
DATA 9 1456 35173067
ACK 1 0 0 # receive ACK for packet 1
DATA 10 1456 -1954964501
ACK 1 0 0 # still receive ACK 1
ACK 1 0 0
ACK 1 0 0
ACK 1 0 0
ACK 1 0 0
ACK 1 0 0
ACK 1 0 0
ACK 1 0 0
DATA 1 1456 650524154 # timeout is triggered, resend whole window
DATA 2 1456 1532157781
DATA 3 1456 975835324
DATA 4 1456 524171520
DATA 5 1456 -2048262358
DATA 6 1456 -1316030808
DATA 7 1456 -2054977684
DATA 8 1456 1425245197
DATA 9 1456 35173067
DATA 10 1456 -1954964501
ACK 6 0 0 # receive ACK, proceed.
DATA 11 1456 1655566761
DATA 12 1456 559528772
DATA 13 1456 -1062487663
DATA 14 1456 512949115
DATA 15 1456 611039600
```

```
...

# receiver log
...
DATA 0 1456 -1241856968
ACK 1 0 0
DATA 2 1456 1532157781 # expect packet 1 but receive packet 2
ACK 1 0 0
DATA 3 1456 975835324 # expect packet 1 but receive packet 3
ACK 1 0 0
DATA 4 1456 524171520
ACK 1 0 0
DATA 5 1456 -2048262358
ACK 1 0 0
DATA 7 1456 -2054977684
ACK 1 0 0
DATA 8 1456 1425245197
ACK 1 0 0
DATA 9 1456 35173067
ACK 1 0 0
DATA 10 1456 -1954964501
ACK 1 0 0
DATA 1 1456 650524154 # finally receive packet 1
ACK 6 0 0 # ACK 6 now
DATA 2 1456 1532157781
ACK 6 0 0
DATA 3 1456 975835324
ACK 6 0 0
...
```

3.1.3 10% error, no reordering, no loss

We can see in sender's log, some packets that are not ACKed due to packet corruption, for instance, data packet 2. The whole window is thus retransmitted and ACKed later, as here 10 data packets in the window (seq 2 from to 11) are resent, and after ACK 5 for packet 5 is received, which means every packet before 5 has been received (including the previous corrupted packet 2), the window is advanced.

On receiver's log, there are packets that are received but not ACKed. For example, the first time the receiver receives data packet 2, it is not ACKed. This is because it does not pass the crc test, which implies its payload is corrupted. As a result of not receiving a correct packet 2, the ACK number the receiver sends back to the sender is 2, corresponding to packet 2. Before it finally receives a correct packet 2, it keeps sending ACK 2, and does not advance the window. Finally, when the receiver receives packet 2 for the second time, it passes the crc test, which means this time it is not corrupted, and thus the window is advanced.

```
# sender log
...
DATA 0 1456 1114035668
DATA 1 1456 -1482431621
DATA 2 1456 1431705831
DATA 3 1456 -1729667388
DATA 4 1456 1258862328
DATA 5 1456 593347315
```

```
DATA 6 1456 2093615261
DATA 7 1456 -354287006
DATA 8 1456 1009651567
DATA 9 1456 -538543278
ACK 1 0 0 # packet 0 received and ACKed, expecting packet 1
DATA 10 1456 780314538
ACK 2 0 0 # packet 1 received and ACKed, expecting packet 2
DATA 11 1456 -552270487
ACK 2 0 0 # packet 2 not ACKed
ACK 2 0 0 # still receive ACK 2
ACK 2 0 0
ACK 2 0 0
ACK 2 0 0
ACK 2 0 0
ACK 2 0 0
ACK 2 0 0
DATA 2 1456 1431705831 # timeout is triggered, resend whole window
DATA 3 1456 -1729667388
DATA 4 1456 1258862328
DATA 5 1456 593347315
DATA 6 1456 2093615261
DATA 7 1456 -354287006
DATA 8 1456 1009651567
DATA 9 1456 -538543278
DATA 10 1456 780314538
DATA 11 1456 -552270487
ACK 5 0 0 # receive ACK, proceed
DATA 12 1456 1632393478
DATA 13 1456 1727104074
DATA 14 1456 1708177396
...

# receiver log
...
DATA 0 1456 1114035668
ACK 1 0 0
DATA 1 1456 -1482431621
ACK 2 0 0
DATA 2 1456 1431705831 # packet 2 (corrupted) received but not ACKed
DATA 3 1456 -1729667388
ACK 2 0 0 # still ACK 2, correct packet 2 has not been received
DATA 4 1456 1258862328
ACK 2 0 0
DATA 5 1456 593347315
DATA 6 1456 2093615261
ACK 2 0 0
DATA 7 1456 -354287006
ACK 2 0 0
DATA 8 1456 1009651567
ACK 2 0 0
DATA 9 1456 -538543278
ACK 2 0 0
DATA 10 1456 780314538
```

```
ACK 2 0 0
DATA 11 1456 -552270487
ACK 2 0 0
DATA 2 1456 1431705831 # packet 2 (correct) received
ACK 5 0 0 # ACK 5 and slide the window
...
```

3.1.4 Reordering, no error, no loss

The first few lines of sender and receiver's logs until the point where the first 10 packets are surely delivered are shown below. Here ACK 10 means every packet before 10 has been received, and the first 10 packets are packt 0 to 9. My sender and receiver cope with packet re-ordering in the following way: my receiver buffers the unordered packets in the receiver's window, and waits for the packets sent later to fill in the gaps. At the same time, the sender just waits for ACKs which indicates every packet before it has been received, and advances its window correspondingly. Finally when the receiver's window is full, everything is in order again. This seems to be correct as the receiver's window's seq is assigned in order, and buffering the packets into the receiver's window helps align the unordered packets.

```
# sender log
SYN 1195598076 0 0
ACK 1195598076 0 0
DATA 0 1456 -1241856968
DATA 1 1456 650524154
DATA 2 1456 1532157781
DATA 3 1456 975835324
DATA 4 1456 524171520
DATA 5 1456 -2048262358
DATA 6 1456 -1316030808
DATA 7 1456 -2054977684
DATA 8 1456 1425245197
DATA 9 1456 35173067
ACK 0 0 0
ACK 2 0 0
DATA 10 1456 -1954964501
DATA 11 1456 1655566761
ACK 2 0 0
ACK 4 0 0
DATA 12 1456 559528772
DATA 13 1456 -1062487663
ACK 5 0 0
DATA 14 1456 512949115
ACK 5 0 0
ACK 5 0 0
ACK 8 0 0
DATA 15 1456 611039600
DATA 16 1456 -1679194347
DATA 17 1456 815059174
ACK 9 0 0
DATA 18 1456 1181899337
ACK 10 0 0 # every packet before 10 has been received
...
# receiver log
```

```
SYN 1195598076 0 0
ACK 1195598076 0 0
DATA 1 1456 650524154
ACK 0 0 0
DATA 0 1456 -1241856968
ACK 2 0 0
DATA 3 1456 975835324
ACK 2 0 0
DATA 2 1456 1532157781
ACK 4 0 0
DATA 4 1456 524171520
ACK 5 0 0
DATA 7 1456 -2054977684
ACK 5 0 0
DATA 6 1456 -1316030808
ACK 5 0 0
DATA 5 1456 -2048262358
ACK 8 0 0
DATA 8 1456 1425245197
ACK 9 0 0
DATA 9 1456 35173067
ACK 10 0 0 # every packet before 10 has been received
...
```

3.2 Duplicate ACK

In this section, window size is still set to be 10 on both sender and receiver.

3.2.1 10% loss, no reordering, no error

In the first few lines of the sender's log, there is a case where my sender receives three duplicate ACK 0, and resends the corresponding packet 0 immediately.

```
# sender log
...
DATA 0 1456 1114035668
DATA 1 1456 -1482431621
DATA 2 1456 1431705831
DATA 3 1456 -1729667388
DATA 4 1456 1258862328
DATA 5 1456 593347315
DATA 6 1456 2093615261
DATA 7 1456 -354287006
DATA 8 1456 1009651567
DATA 9 1456 -538543278
ACK 0 0 0
ACK 0 0 0
ACK 0 0 0 # receive three duplicate ACK 0
DATA 0 1456 1114035668 # resends the corresponding packet 0 immediately
...
```

3.2.2 Effciency

In this case, a large file of size about 10MB is transferred under 10% loss link. The transferring time is 3m48.542s (shown in Figure 23), which is smaller than the transferring time in part 3.1 (shown in Figure 22), 4m28.692s. Duplicate ACK makes the sender more effcient.

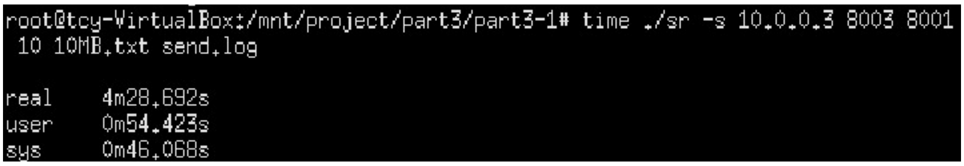


Figure 22: Transferring time for a 10MB file under 10% loss link, using program in part 3.1

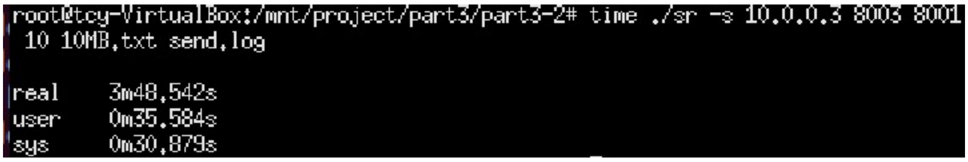


Figure 23: Transferring time for a 10MB file under 10% loss link, using program in part 3.2

3.3 NACK

3.3.1 10% loss, no reordering, no error

In the first few lines of the sender's log, there is a case where my sender retransmits packet 2 immediately after it receives a NACK 2. In receiver's log, the headers show that my receiver has received packet 1 and packet 3, but not packet 2, so there is a gap in its window for packet 2. The header NACK 2 indicates that my receiver sends back a NACK for this gap.

```
# sender log
...
DATA 0 1456 1114035668
DATA 1 1456 -1482431621
DATA 2 1456 1431705831
DATA 3 1456 -1729667388
DATA 4 1456 1258862328
DATA 5 1456 593347315
DATA 6 1456 2093615261
DATA 7 1456 -354287006
DATA 8 1456 1009651567
DATA 9 1456 -538543278
ACK 1 0 0
DATA 10 1456 780314538
ACK 2 0 0
DATA 11 1456 -552270487
NACK 2 0 0 # receives NACK 2
DATA 2 1456 1431705831 # retransmits packet 2 immediately
...

# receiver log
...
DATA 0 1456 1114035668
ACK 1 0 0
```

```
DATA 1 1456 -1482431621 # receives packet 1
ACK 2 0 0
DATA 3 1456 -1729667388 # receives packet 3, gap occurs
NACK 2 0 0 # sends back NACK 2
...
```

3.3.2 Effciency

In this case, a large file of size about 10MB is transferred under 10% loss link. The transferring time is 2m15.787s (shown in Figure 24), which is smaller than the transferring time in part 3.1 (shown in Figure 22), 4m28.692s. NACK makes the sender more effcient.

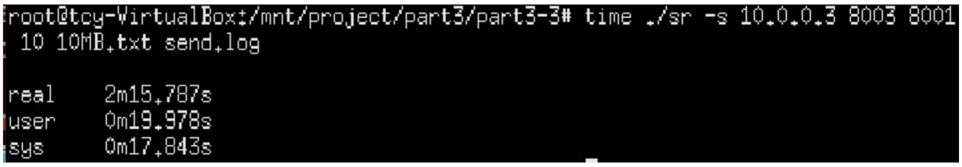


Figure 24: Transferring time for a 10MB file under 10% loss link, using program in part 3.3

3.4 Throughput, delay and window size

In this part, program from part 3.3 is used, and the network created by launching by `normal_topo.py` has no loss, no error, no packet reordering for transferring `file3.jpg` from host h_1 to h_2. The size of `file3.jpg` is 4354951 bytes (see Figure 25), so the throughput to be computed in the following part could be obtained as

$$\text{throughput} = 4354951 \times 8 \div \text{time} = 34839608 \div \text{time [bps]}$$

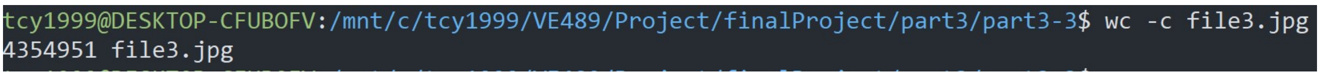


Figure 25: Size of file3.jpg

When the window size is 10, changing the delay of the link between h_1 and h_2 to $\{0.01, 10, 50, 100\}$ milliseconds. The results of time measurement with different delay on the sender side are shown in Figure 26, 27, 28, 29, $\{16.571, 20.401, 48.574, 78.047\}$ seconds. Using the formula above, the throughput for each delay is $\{2102444.511, 1707740.209, 717248.0751, 446392.6608\}$ bps.

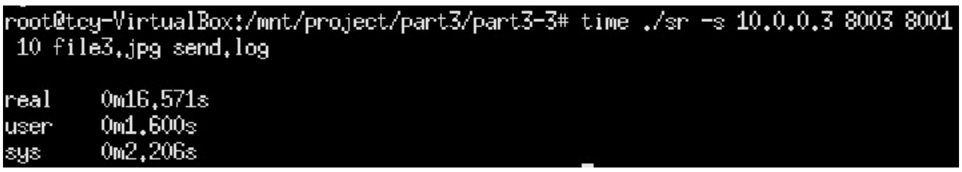


Figure 26: Time measurement with 0.01ms delay on the sender side, window size 10

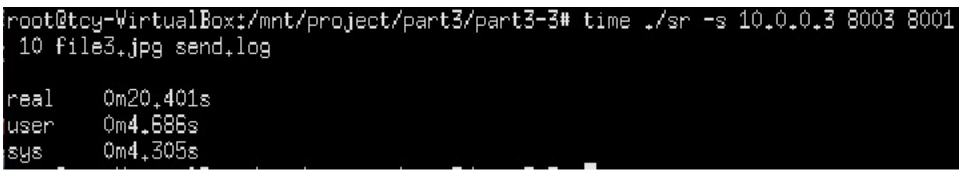


Figure 27: Time measurement with 10ms delay on the sender side, window size 10

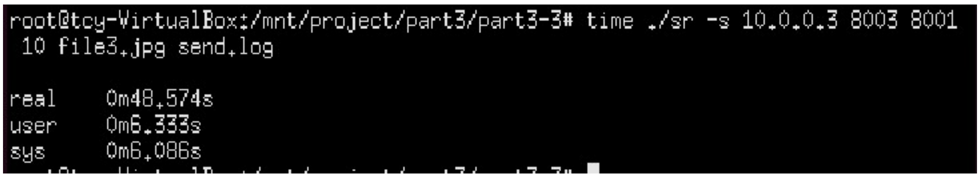


Figure 28: Time measurement with 50ms delay on the sender side, window size 10

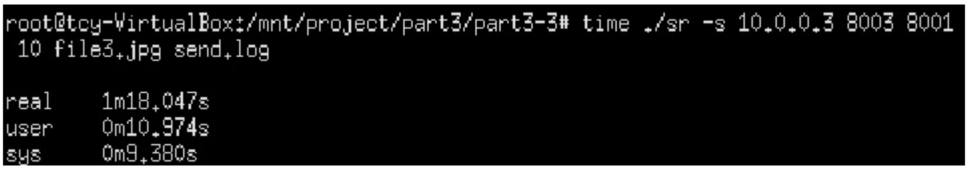


Figure 29: Time measurement with 100ms delay on the sender side, window size 10

When the window size is 50, the results of time measurement on the sender side are shown in Figure 30, 31, 32, 33, {9.053, 13.431, 15.553, 16.648} seconds. Using the formula above, the throughput for each delay is {3848404.728, 2593969.771, 2240057.095, 2092720.327} bps.

Figure 30: Time measurement with 0.01ms delay on the sender side, window size 50

Figure 31: Time measurement with 10ms delay on the sender side, window size 50

Figure 32: Time measurement with 50ms delay on the sender side, window size 50

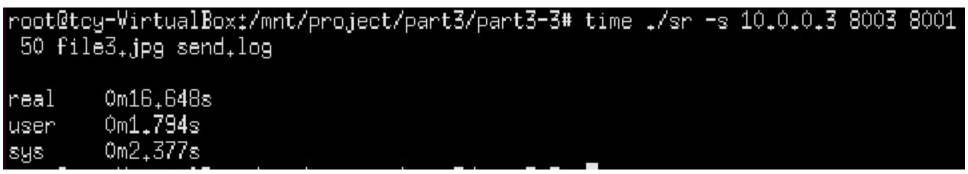


Figure 33: Time measurement with 100ms delay on the sender side, window size 50

We can see as the window size becomes larger, the throughput increases. However, as the delay becomes larger, the throuput decreases.