
UM-SJTU JOINT INSTITUTE
COMPUTER NETWORKS
(VE489)

TERM PROJECT

SU 2020

Name	ID
Xie Jinglei	517370910022

Date: July, 2020

1 Part 1

In this part, we consider the following network structure:

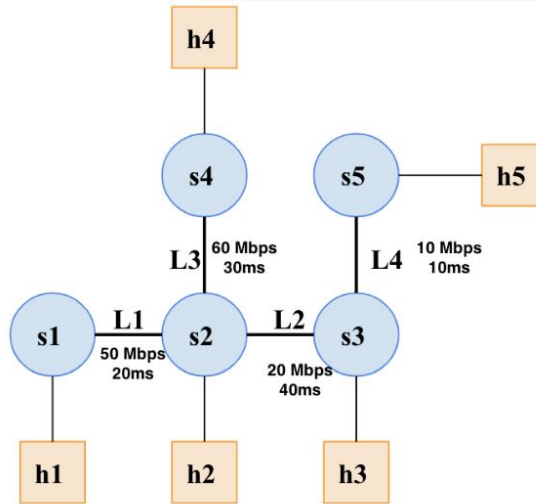


Figure 1: Mininet network topology

1.1 Link latency using ping

To measure the latency between h_1 and h_2 , we can do

```
h1 ping -c 10 h2
```

The results are shown in Figure 2. So the average round-trip time (RTT) is 43.021 ms. From the topology graph, link L1 is used, and the latency between h_1 and h_2 should be 20 ms. We can see that the RTT is close to 2 times link latency (40 ms).

```
mininet> h1 ping -c 10 h2
PING 10.0.0.2 (10.0.0.2) 56(84) bytes of data:
64 bytes from 10.0.0.2: icmp_seq=1 ttl=64 time=50.9 ms
64 bytes from 10.0.0.2: icmp_seq=2 ttl=64 time=43.9 ms
64 bytes from 10.0.0.2: icmp_seq=3 ttl=64 time=42.0 ms
64 bytes from 10.0.0.2: icmp_seq=4 ttl=64 time=41.4 ms
64 bytes from 10.0.0.2: icmp_seq=5 ttl=64 time=41.8 ms
64 bytes from 10.0.0.2: icmp_seq=6 ttl=64 time=41.8 ms
64 bytes from 10.0.0.2: icmp_seq=7 ttl=64 time=42.3 ms
64 bytes from 10.0.0.2: icmp_seq=8 ttl=64 time=41.9 ms
64 bytes from 10.0.0.2: icmp_seq=9 ttl=64 time=41.9 ms
64 bytes from 10.0.0.2: icmp_seq=10 ttl=64 time=41.8 ms

--- 10.0.0.2 ping statistics ---
10 packets transmitted, 10 received, 0% packet loss, time 9014ms
rtt min/avg/max/mdev = 41.461/43.021/50.906/2.719 ms
mininet>
```

Figure 2: h_1 ping h_2

To measure the latency between h_3 and h_5 , we can do

```
h3 ping -c 10 h5
```

The results are shown in Figure 3. So the average round-trip time (RTT) is 23.470 ms. From the topology graph, link L4 is used, and the latency between h_1 and h_2 should be 10 ms. We can see that the RTT is close to 2 times link latency (20 ms).

```
mininet> h3 ping -c 10 h5
PING 10.0.0.5 (10.0.0.5) 56(84) bytes of data.
64 bytes from 10.0.0.5: icmp_seq=1 ttl=64 time=29.7 ms
64 bytes from 10.0.0.5: icmp_seq=2 ttl=64 time=30.2 ms
64 bytes from 10.0.0.5: icmp_seq=3 ttl=64 time=22.5 ms
64 bytes from 10.0.0.5: icmp_seq=4 ttl=64 time=21.3 ms
64 bytes from 10.0.0.5: icmp_seq=5 ttl=64 time=22.2 ms
64 bytes from 10.0.0.5: icmp_seq=6 ttl=64 time=20.7 ms
64 bytes from 10.0.0.5: icmp_seq=7 ttl=64 time=21.8 ms
64 bytes from 10.0.0.5: icmp_seq=8 ttl=64 time=21.8 ms
64 bytes from 10.0.0.5: icmp_seq=9 ttl=64 time=21.9 ms
64 bytes from 10.0.0.5: icmp_seq=10 ttl=64 time=22.0 ms

--- 10.0.0.5 ping statistics ---
10 packets transmitted, 10 received, 0% packet loss, time 9014ms
rtt min/avg/max/mdev = 20.732/23.470/30.274/3.309 ms
mininet>
```

Figure 3: h_3 ping h_5

1.2 Path latency using ping

To measure the latency between h_1 and h_5 , we can do

```
h1 ping -c 10 h5
```

The results are shown in Figure 4. So the measured average round-trip delay is 148.212 ms. From the topology graph, link L1, L2 and L4 are used, and the latency should be $20 + 40 + 10 = 70(ms)$. So the theoretical delay is $70 \times 2 = 140(ms)$. We can see that the measured round-trip delay is close to the theoretical delay.

```
mininet> h1 ping -c 10 h5
PING 10.0.0.5 (10.0.0.5) 56(84) bytes of data.
64 bytes from 10.0.0.5: icmp_seq=1 ttl=64 time=171 ms
64 bytes from 10.0.0.5: icmp_seq=2 ttl=64 time=156 ms
64 bytes from 10.0.0.5: icmp_seq=3 ttl=64 time=145 ms
64 bytes from 10.0.0.5: icmp_seq=4 ttl=64 time=144 ms
64 bytes from 10.0.0.5: icmp_seq=5 ttl=64 time=145 ms
64 bytes from 10.0.0.5: icmp_seq=6 ttl=64 time=144 ms
64 bytes from 10.0.0.5: icmp_seq=7 ttl=64 time=143 ms
64 bytes from 10.0.0.5: icmp_seq=8 ttl=64 time=142 ms
64 bytes from 10.0.0.5: icmp_seq=9 ttl=64 time=143 ms
64 bytes from 10.0.0.5: icmp_seq=10 ttl=64 time=144 ms

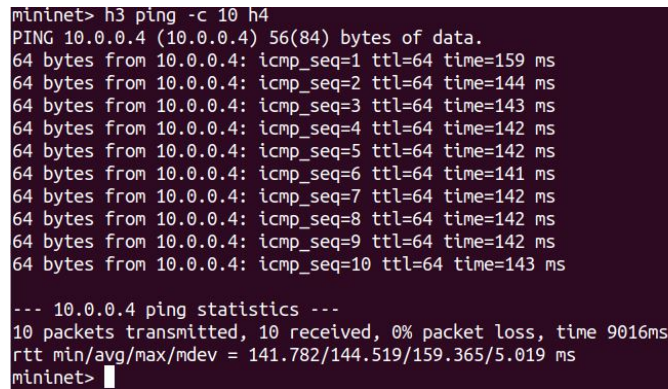
--- 10.0.0.5 ping statistics ---
10 packets transmitted, 10 received, 0% packet loss, time 9012ms
rtt min/avg/max/mdev = 142.799/148.212/171.850/8.687 ms
mininet>
```

Figure 4: h_1 ping h_5

To measure the latency between h_3 and h_4 , we can do

```
h3 ping -c 10 h4
```

The results are shown in Figure 5. So the measured average round-trip delay is 144.519 ms. From the topology graph, link L2 and L3 are used, and the latency should be $40 + 30 = 70(ms)$. So the theoretical delay is $70 \times 2 = 140(ms)$. We can see that the measured round-trip delay is close to the theoretical delay.

A terminal window showing the output of a ping command from host h3 to host h4. The output displays 10 successful ping attempts with round-trip times ranging from 139 ms to 159 ms. A summary line shows an average round-trip time of 144.519 ms.

```
mininet> h3 ping -c 10 h4
PING 10.0.0.4 (10.0.0.4) 56(84) bytes of data:
64 bytes from 10.0.0.4: icmp_seq=1 ttl=64 time=159 ms
64 bytes from 10.0.0.4: icmp_seq=2 ttl=64 time=144 ms
64 bytes from 10.0.0.4: icmp_seq=3 ttl=64 time=143 ms
64 bytes from 10.0.0.4: icmp_seq=4 ttl=64 time=142 ms
64 bytes from 10.0.0.4: icmp_seq=5 ttl=64 time=142 ms
64 bytes from 10.0.0.4: icmp_seq=6 ttl=64 time=141 ms
64 bytes from 10.0.0.4: icmp_seq=7 ttl=64 time=142 ms
64 bytes from 10.0.0.4: icmp_seq=8 ttl=64 time=142 ms
64 bytes from 10.0.0.4: icmp_seq=9 ttl=64 time=142 ms
64 bytes from 10.0.0.4: icmp_seq=10 ttl=64 time=143 ms

--- 10.0.0.4 ping statistics ---
10 packets transmitted, 10 received, 0% packet loss, time 9016ms
rtt min/avg/max/mdev = 141.782/144.519/159.365/5.019 ms
mininet>
```

Figure 5: h_3 ping h_4

1.3 Link bandwidth using *iperf*

To measure the bandwidth between h_1 and h_2 , first I set h_1 to be server:

```
xterm h1
# In terminal of h1:
iperf -s -p 80
```

Then I set h_2 as client and send data to h_1 (10 seconds):

```
xterm h2
# In terminal of h2:
iperf -c 10.0.0.1 -p 80 -t 10
```

The results are shown in Figure 6. From server's side, the measured bandwidth is 41.6 Mbps; from the client's side, the measured bandwidth is 44.0 Mbps. They are both close to the value given in the topology graph (L1: 50 Mbps). However, they are both lower than that value. That should be normal, because the bandwidth sometimes cannot be totally occupied by the data transferred. Also, we can see that the server received 53.1 MB of data, and the client sent 53.1 MB of data. They are equal to each other.

The image shows two terminal windows side-by-side. The left window, titled "Node: h1", shows the server side of an iperf test. It starts with the command `iperf -s -p 80`. The output shows it is listening on TCP port 80. A client connects from 10.0.0.1 port 80 to the local 10.0.0.2 port 33626. The test results for the interval 0.0-10.7 sec show a transfer of 53.1 MBytes and a bandwidth of 41.6 Mbits/sec. The right window, titled "Node: h2", shows the client side. It starts with the command `iperf -c 10.0.0.1 -p 80 -t 10`. The output shows it is connecting to 10.0.0.1 TCP port 80. It connects to the local 10.0.0.2 port 33626. The test results for the interval 0.0-10.1 sec show a transfer of 53.1 MBytes and a bandwidth of 44.0 Mbits/sec.

```

"Node: h1"
root@ubuntu:/newDisk/ve489/pi-part1# iperf -s -p 80
-----
Server listening on TCP port 80
TCP window size: 85.3 KByte (default)
-----
[ 28] local 10.0.0.1 port 80 connected with 10.0.0.2 port 33626
[ ID] Interval      Transfer    Bandwidth
[ 28] 0.0-10.7 sec  53.1 MBytes 41.6 Mbits/sec
[  ]

"Node: h2"
root@ubuntu:/newDisk/ve489/pi-part1# iperf -c 10.0.0.1 -p 80 -t 10
-----
Client connecting to 10.0.0.1, TCP port 80
TCP window size: 85.3 KByte (default)
-----
[ 27] local 10.0.0.2 port 33626 connected with 10.0.0.1 port 80
[ ID] Interval      Transfer    Bandwidth
[ 27] 0.0-10.1 sec  53.1 MBytes 44.0 Mbits/sec
root@ubuntu:/newDisk/ve489/pi-part1#

```

Figure 6: Bandwidth measurement between h_1 and h_2

To measure the bandwidth between h_3 and h_5 , first I set h_3 to be server:

```

xterm h3
# In terminal of h3:
iperf -s -p 80

```

Then I set h_5 as client and send data to h_3 (10 seconds):

```

xterm h5
# In terminal of h5:
iperf -c 10.0.0.3 -p 80 -t 10

```

The results are shown in Figure 7. From server's side, the measured bandwidth is 9.53 Mbps; from the client's side, the measured bandwidth is 11.2 Mbps. They are both very close to the value given in the topology graph (L4: 10 Mbps). Also, we can see that the server received 13.4 MB of data, and the client sent 13.4 MB of data. They are equal to each other.

The image shows two terminal windows side-by-side. The left window, titled "Node: h3", shows the server side. It starts with the command `iperf -s -p 80`. The output shows it is listening on TCP port 80. A client connects from 10.0.0.5 port 59968 to the local 10.0.0.3 port 80. The test results for the interval 0.0-11.8 sec show a transfer of 13.4 MBytes and a bandwidth of 9.53 Mbits/sec. The right window, titled "Node: h5", shows the client side. It starts with the command `iperf -c 10.0.0.3 -p 80 -t 10`. The output shows it is connecting to 10.0.0.3 TCP port 80. It connects to the local 10.0.0.5 port 59968. The test results for the interval 0.0-10.0 sec show a transfer of 13.4 MBytes and a bandwidth of 11.2 Mbits/sec.

```

"Node: h3"
xroot@ubuntu:/newDisk/ve489/pi-part1# iperf -s -p 80
-----
Server listening on TCP port 80
TCP window size: 85.3 KByte (default)
-----
[ 28] local 10.0.0.3 port 80 connected with 10.0.0.5 port 59968
[ ID] Interval      Transfer    Bandwidth
[ 28] 0.0-11.8 sec  13.4 MBytes 9.53 Mbits/sec
[  ]

"Node: h5"
root@ubuntu:/newDisk/ve489/pi-part1# iperf -c 10.0.0.3 -p 80 -t 10
-----
Client connecting to 10.0.0.3, TCP port 80
TCP window size: 85.3 KByte (default)
-----
[ 27] local 10.0.0.5 port 59968 connected with 10.0.0.3 port 80
[ ID] Interval      Transfer    Bandwidth
[ 27] 0.0-10.0 sec  13.4 MBytes 11.2 Mbits/sec
root@ubuntu:/newDisk/ve489/pi-part1#

```

Figure 7: Bandwidth measurement between h_3 and h_5

1.4 Path throughput using *iperf*

To measure the path throughput between h_1 and h_5 , first I set h_1 to be server:

```
xterm h1
# In terminal of h1:
iperf -s -p 80
```

Then I set h_5 as client and send data to h_1 (10 seconds):

```
xterm h5
# In terminal of h5:
iperf -c 10.0.0.1 -p 80 -t 10
```

The results are shown in Figure 8. From server's side, the measured bandwidth is 9.20 Mbps; from the client's side, the measured bandwidth is 11.7 Mbps. From the given topology graph, the bottleneck link in this path is L4, which has a bandwidth of 10 Mbps. The measured path throughputs from both server and client are close to this value.

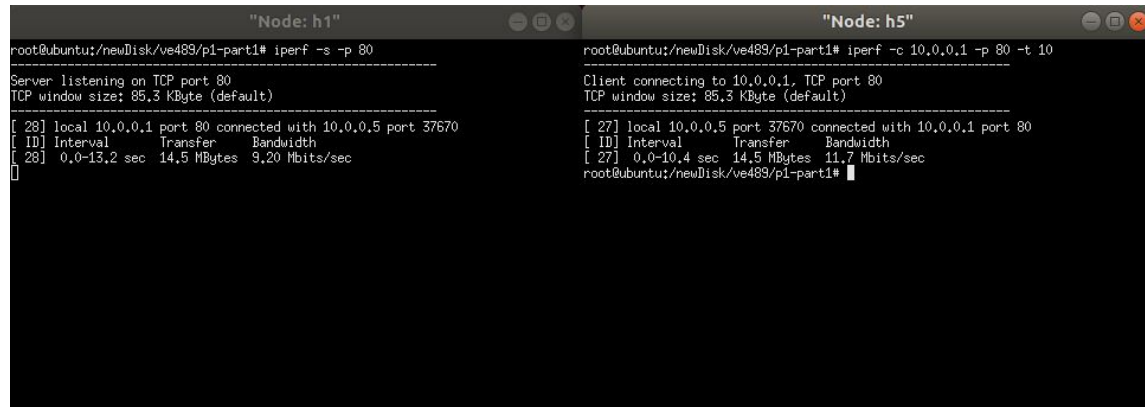


Figure 8: Path throughput measurement between h_1 and h_5

To measure the path throughput between h_3 and h_4 , first I set h_3 to be server:

```
xterm h3
# In terminal of h3:
iperf -s -p 80
```

Then I set h_4 as client and send data to h_3 (10 seconds):

```
xterm h4
# In terminal of h4:
iperf -c 10.0.0.3 -p 80 -t 10
```

The results are shown in Figure 9. From server's side, the measured bandwidth is 17.9 Mbps; from the client's side, the measured bandwidth is 23.9 Mbps. From the given topology graph, the bottleneck link in this path is L2, which has a bandwidth of 20 Mbps. The measured path throughputs from both server and client are close to this value.

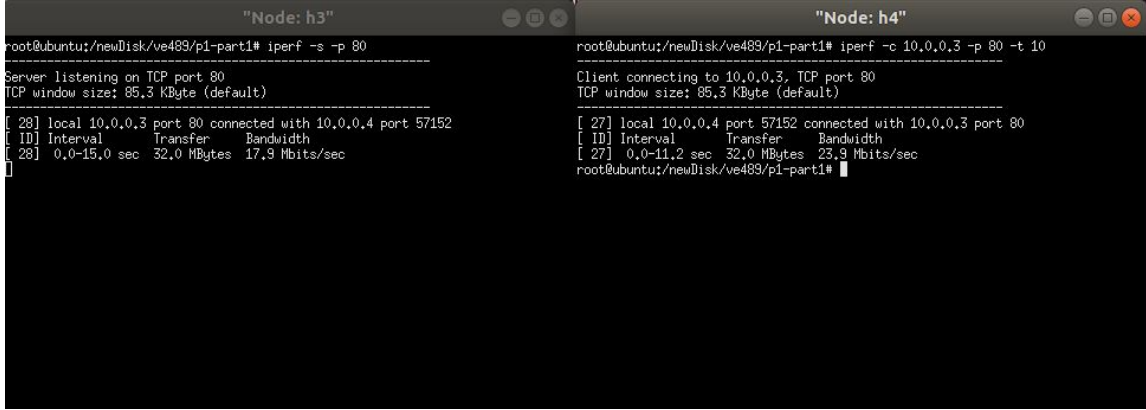


Figure 9: Path throughput measurement between h_3 and h_4

1.5 Multiplexing

We consider the situation when (h_1, h_5) and (h_3, h_4) are sharing the same link and doing information exchange at the same time.

1.5.1 Latency measurement

Commands for latency measurements (ping 10 times):

```

# In terminal of h5:
ping -c 10 10.0.0.1
# In terminal of h4:
ping -c 10 10.0.0.3

```

The data transmission began almost at the same time, and lasted for 10 seconds. The results are shown in Figure 10.

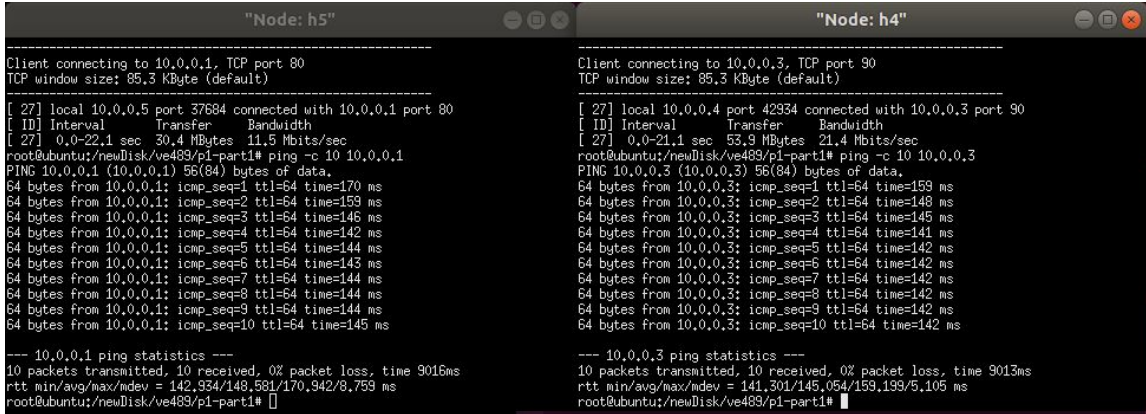


Figure 10: Simultaneous latency measurement for (h_1, h_5) and (h_3, h_4)

We can see that the average round-trip time between h_1 and h_5 is 148.581 ms, and the average round-trip time between h_3 and h_4 is 145.054 ms. Compared with results from section 1.2 (h_1 and h_5 : 148.212 ms; h_3 and h_4 : 144.519 ms), the new round-trip time is both a little bit longer. It is because of the effect of multiplexing, where the link L2 is shared. As a result, the data transformation is a bit slower, since L2 has to transfer different packets (both from h_5 and h_4) to different destinations (s2 or s3), under a heavier load. Still, the difference is not significant, as the data amount is small.

1.5.2 Bandwidth measurement

The network is set as follows:

```
# In terminal of h1:
iperf -s -p 80
# In terminal of h3:
iperf -s -p 90
# In terminal of h5:
iperf -c 10.0.0.1 -p 80 -t 20
# In terminal of h4:
iperf -c 10.0.0.3 -p 90 -t 20
```

The data transmission began almost at the same time, and lasted for 20 seconds. The results are shown in Figure 11. For the connection (h_1, h_5), the bandwidth obtained at server side is 8.86 Mbps, and the bandwidth obtained at client side is 11.5 Mbps. The average bandwidth is 10.18 Mbps, which is smaller compared to the results in section 1.4: $(9.20 + 11.7)/2 = 10.45$ Mbps. For (h_3, h_4), the bandwidth obtained at server side is 18.3 Mbps, and the bandwidth obtained at client side is 21.4 Mbps. The average bandwidth is 19.85 Mbps, which is also smaller compared to the results in section 1.4: $(17.9 + 23.9)/2 = 20.9$ Mbps. Both bandwidth measurements become smaller because link L2 is shared. L2 has to transfer data for both connections (h_1, h_5) and (h_3, h_4), which lowers its efficiency.

Still, the bottleneck for (h_1, h_5) is L4, while the bottleneck for (h_3, h_4) is L2, which does not change. The decrement of bandwidth for (h_1, h_5) is about 1.59 Mbps, and the decrement of bandwidth for (h_3, h_4) is about 1.05 Mbps. These decrements are close to each other, thus the bandwidth of L2 should be shared fairly.

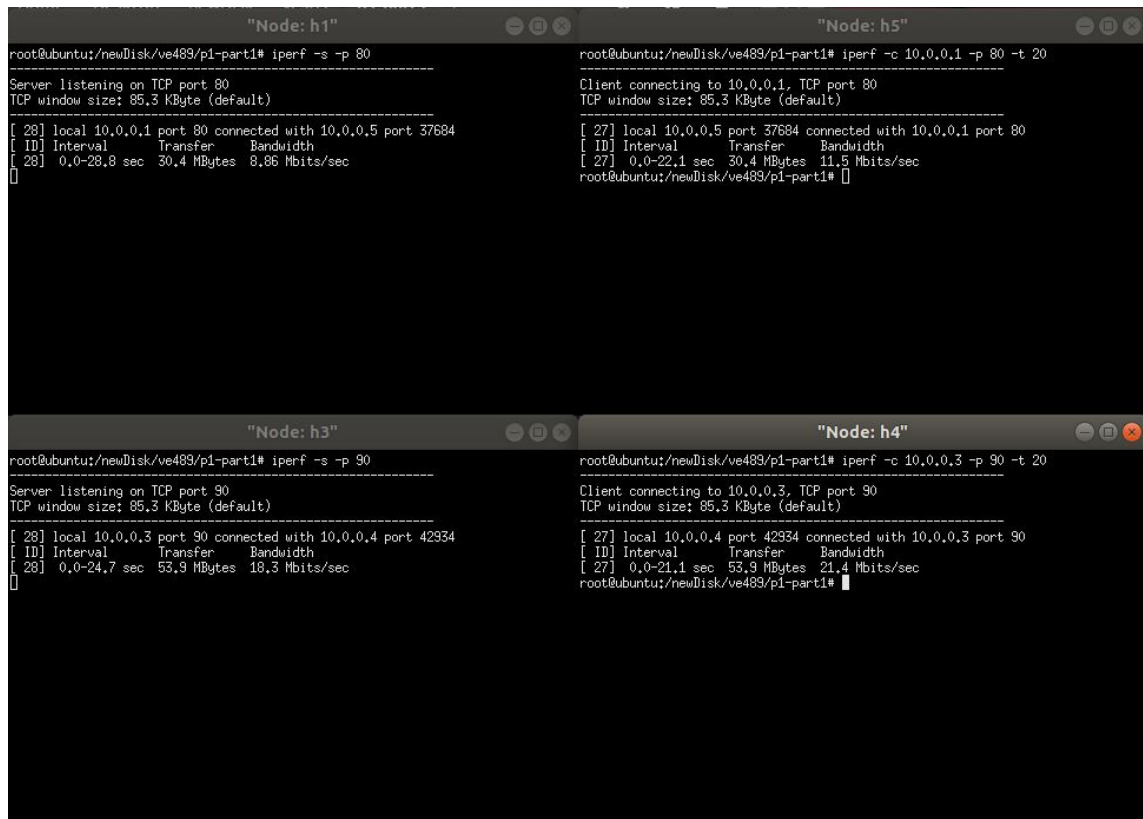


Figure 11: Simultaneous path throughput measurement for (h_1, h_5) and (h_3, h_4)

2 Part 2

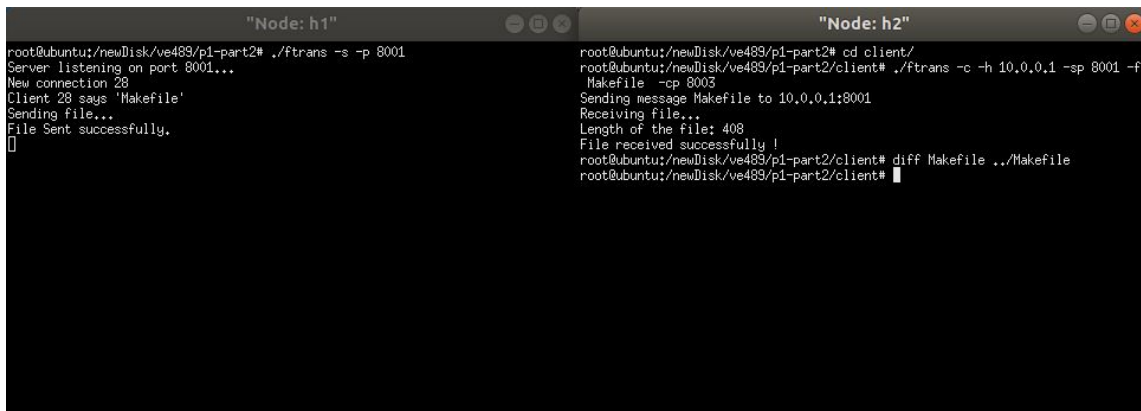
In this section, I use the network structure of `p1.topo.py` to test my code. I run the client in a separate directory called “client”.

2.1 Transferring the Makefile

The original Makefile is put in the parent directory of client, and the transferred Makefile should be in the client directory.

```
# In terminal of h1:
./ftrans -s -p 8001
# In terminal of h2:
./ftrans -c -h 10.0.0.1 -sp 8001 -f Makefile -cp 8002
```

The result is shown in Figure 12. We can see that the transferred Makefile is the same as the original one (diff outputs nothing).



The image shows two terminal windows side-by-side. The left window, titled "Node: h1", shows a server listening on port 8001. It receives a connection from client 28, who sends the filename 'Makefile'. The server sends the file successfully. The right window, titled "Node: h2", shows the client side. It changes to the 'client' directory and runs 'ftrans -c -h 10.0.0.1 -sp 8001 -f Makefile -cp 8003'. It sends the message 'Makefile' to the server, receives the file, and confirms the length is 408. Finally, it runs 'diff Makefile ../Makefile' and shows no output, indicating the files are identical.

```
"Node: h1"
root@ubuntu:/newDisk/ve489/p1-part2# ./ftrans -s -p 8001
Server listening on port 8001...
New connection 28
Client 28 says 'Makefile'
Sending File...
File Sent successfully.
[]

"Node: h2"
root@ubuntu:/newDisk/ve489/p1-part2# cd client/
root@ubuntu:/newDisk/ve489/p1-part2/client# ./ftrans -c -h 10.0.0.1 -sp 8001 -f
Makefile -cp 8003
Sending message Makefile to 10.0.0.1:8001
Receiving file...
Length of the file: 408
File received successfully !
root@ubuntu:/newDisk/ve489/p1-part2/client# diff Makefile ../Makefile
root@ubuntu:/newDisk/ve489/p1-part2/client#
```

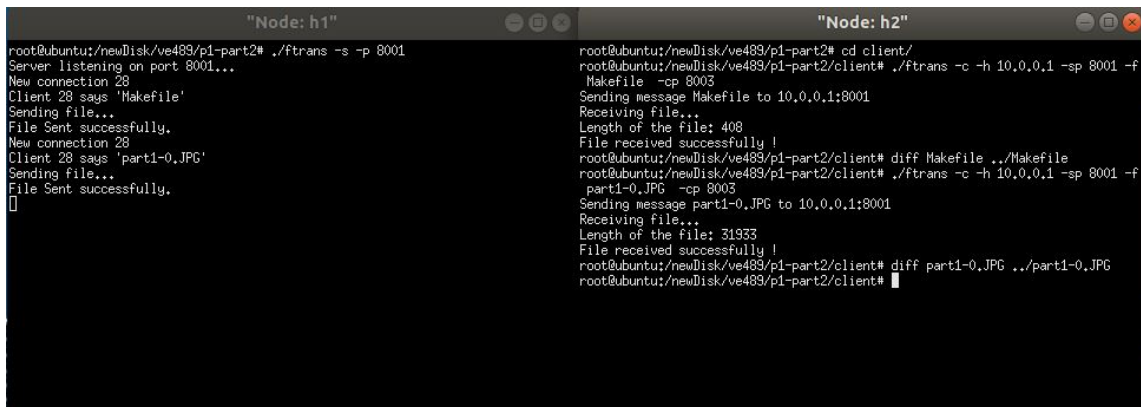
Figure 12: Transferring the Makefile

2.2 Transferring a non-text file

I also tried transferring a jpg file. The original part1-0.JPG is put in the parent directory of client, and the transferred part1-0.JPG should be in the client directory.

```
# In terminal of h1:
./ftrans -s -p 8001
# In terminal of h2:
./ftrans -c -h 10.0.0.1 -sp 8001 -f part1-0.JPG -cp 8002
```

The result is shown in Figure 13. We can see that the transferred part1-0.JPG is the same as the original one (diff outputs nothing).



The image shows two terminal windows side-by-side. The left window, titled "Node: h1", shows the server listening on port 8001. It receives a connection from client 28, who sends 'part1-0.JPG'. The server sends the file successfully. The right window, titled "Node: h2", shows the client side. It runs 'ftrans -c -h 10.0.0.1 -sp 8001 -f part1-0.JPG -cp 8003'. It sends the message 'part1-0.JPG' to the server, receives the file, and confirms the length is 31933. Finally, it runs 'diff part1-0.JPG ../part1-0.JPG' and shows no output, indicating the files are identical.

```
"Node: h1"
root@ubuntu:/newDisk/ve489/p1-part2# ./ftrans -s -p 8001
Server listening on port 8001...
New connection 28
Client 28 says 'part1-0.JPG'
Sending File...
File Sent successfully.
[]

"Node: h2"
root@ubuntu:/newDisk/ve489/p1-part2# cd client/
root@ubuntu:/newDisk/ve489/p1-part2/client# ./ftrans -c -h 10.0.0.1 -sp 8001 -f
part1-0.JPG -cp 8003
Sending message part1-0.JPG to 10.0.0.1:8001
Receiving file...
Length of the file: 31933
File received successfully !
root@ubuntu:/newDisk/ve489/p1-part2/client# diff part1-0.JPG ../part1-0.JPG
root@ubuntu:/newDisk/ve489/p1-part2/client#
```

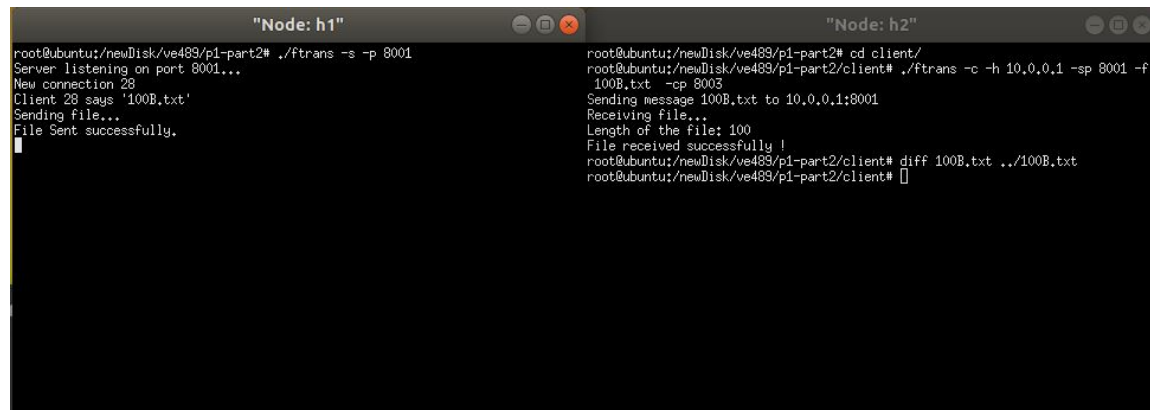
Figure 13: Transferring a jpg file

2.3 Transferring a 100B file

I created a file 100B.txt which is 100 bytes in size. The original file is put in the parent directory of client, and the transferred file should be in the client directory.

```
# In terminal of h1:
./ftrans -s -p 8001
# In terminal of h2:
./ftrans -c -h 10.0.0.1 -sp 8001 -f 100B.txt -cp 8002
```

The result is shown in Figure 14. We can see that the transferred 100B.txt is the same as the original one (diff outputs nothing).



```
"Node: h1"
root@ubuntu:/newDisk/ve489/p1-part2# ./ftrans -s -p 8001
Server listening on port 8001...
New connection 28
Client 28 says '100B.txt'
Sending file...
File Sent successfully.

"Node: h2"
root@ubuntu:/newDisk/ve489/p1-part2# cd client/
root@ubuntu:/newDisk/ve489/p1-part2/client# ./ftrans -c -h 10.0.0.1 -sp 8001 -f
100B.txt -cp 8002
Sending message 100B.txt to 10.0.0.1:8001
Receiving file...
Length of the file: 100
File received successfully !
root@ubuntu:/newDisk/ve489/p1-part2/client# diff 100B.txt ../100B.txt
root@ubuntu:/newDisk/ve489/p1-part2/client#
```

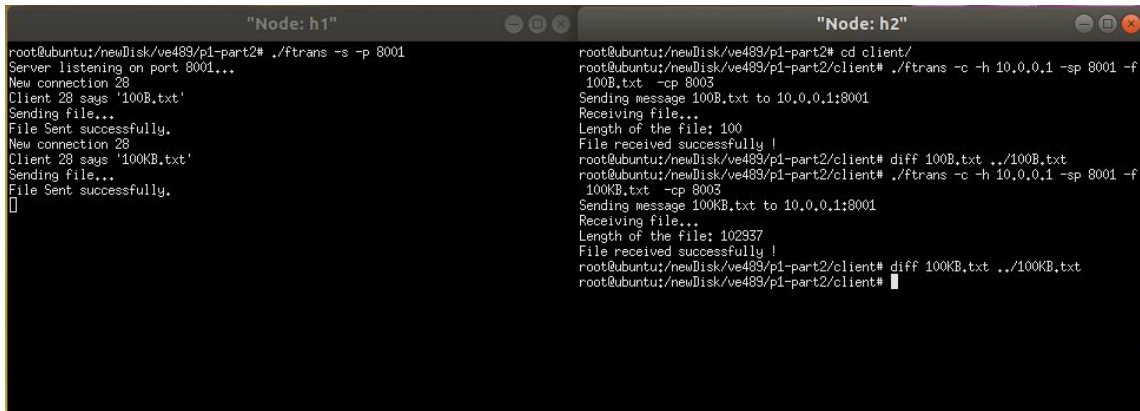
Figure 14: Transferring a 100 byte file

2.4 Transferring a 100KB file

I created a file 100KB.txt which is about 100KB in size. The original file is put in the parent directory of client, and the transferred file should be in the client directory.

```
# In terminal of h1:
./ftrans -s -p 8001
# In terminal of h2:
./ftrans -c -h 10.0.0.1 -sp 8001 -f 100KB.txt -cp 8002
```

The result is shown in Figure 15. We can see that the transferred 100KB.txt is the same as the original one (diff outputs nothing).



The image shows two terminal windows side-by-side. The left window, titled "Node: h1", shows a server listening on port 8001. It receives a connection from client 28, who sends a file named '100B.txt'. The server sends the file successfully. Then, client 28 sends a file named '100KB.txt', which the server also sends successfully. The right window, titled "Node: h2", shows the client side. It runs 'cd client/' and then 'diff 100B.txt ../100B.txt', which shows no differences. Then, it runs 'diff 100KB.txt ../100KB.txt', which also shows no differences, confirming the file was transferred correctly.

```
"Node: h1"
root@ubuntu:/newDisk/ve489/p1-part2# ./ftrans -s -p 8001
Server listening on port 8001...
New connection 28
Client 28 says '100B.txt'
Sending file...
File Sent successfully.
New connection 28
Client 28 says '100KB.txt'
Sending file...
File Sent successfully.
[]

"Node: h2"
root@ubuntu:/newDisk/ve489/p1-part2# cd client/
root@ubuntu:/newDisk/ve489/p1-part2/client# ./ftrans -c -h 10.0.0.1 -sp 8001 -f
100B.txt -cp 8003
Sending message 100B.txt to 10.0.0.1:8001
Receiving file...
Length of the file: 100
File received successfully !
root@ubuntu:/newDisk/ve489/p1-part2/client# diff 100B.txt ../100B.txt
root@ubuntu:/newDisk/ve489/p1-part2/client# ./ftrans -c -h 10.0.0.1 -sp 8001 -f
100KB.txt -cp 8003
Sending message 100KB.txt to 10.0.0.1:8001
Receiving file...
Length of the file: 102937
File received successfully !
root@ubuntu:/newDisk/ve489/p1-part2/client# diff 100KB.txt ../100KB.txt
root@ubuntu:/newDisk/ve489/p1-part2/client#
```

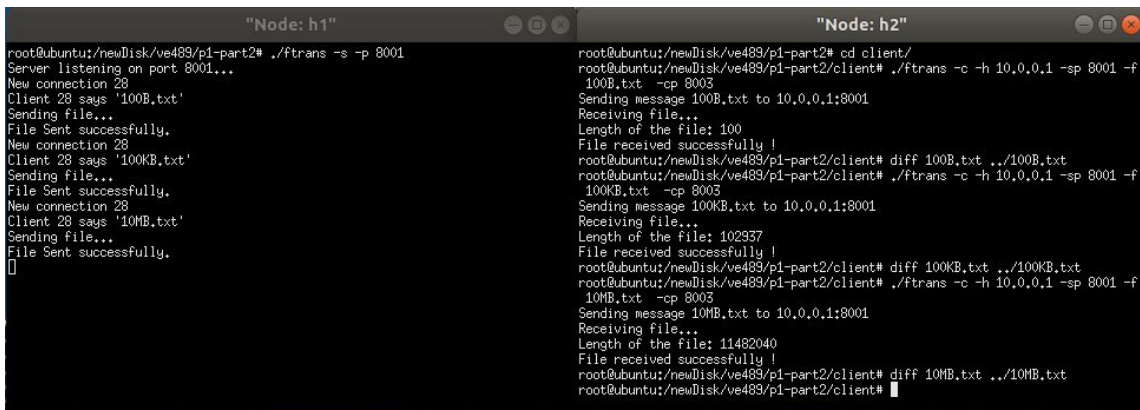
Figure 15: Transferring a 100KB file

2.5 Transferring a 10MB file

I created a file 10MB.txt which is about 10MB in size. The original file is put in the parent directory of client, and the transferred file should be in the client directory.

```
# In terminal of h1:
./ftrans -s -p 8001
# In terminal of h2:
./ftrans -c -h 10.0.0.1 -sp 8001 -f 10MB.txt -cp 8002
```

The result is shown in Figure 16. We can see that the transferred 10MB.txt is the same as the original one (diff outputs nothing).



The image shows two terminal windows side-by-side. The left window, titled "Node: h1", shows the server listening on port 8001. It receives connections from client 28 for '100B.txt' and '100KB.txt', sending them successfully. Then, client 28 sends a file named '10MB.txt', which the server also sends successfully. The right window, titled "Node: h2", shows the client side. It runs 'cd client/' and then 'diff 100B.txt ../100B.txt' and 'diff 100KB.txt ../100KB.txt', both showing no differences. Then, it runs 'diff 10MB.txt ../10MB.txt', which also shows no differences, confirming the 10MB file was transferred correctly.

```
"Node: h1"
root@ubuntu:/newDisk/ve489/p1-part2# ./ftrans -s -p 8001
Server listening on port 8001...
New connection 28
Client 28 says '100B.txt'
Sending file...
File Sent successfully.
New connection 28
Client 28 says '100KB.txt'
Sending file...
File Sent successfully.
New connection 28
Client 28 says '10MB.txt'
Sending file...
File Sent successfully.
[]

"Node: h2"
root@ubuntu:/newDisk/ve489/p1-part2# cd client/
root@ubuntu:/newDisk/ve489/p1-part2/client# ./ftrans -c -h 10.0.0.1 -sp 8001 -f
100B.txt -cp 8003
Sending message 100B.txt to 10.0.0.1:8001
Receiving file...
Length of the file: 100
File received successfully !
root@ubuntu:/newDisk/ve489/p1-part2/client# diff 100B.txt ../100B.txt
root@ubuntu:/newDisk/ve489/p1-part2/client# ./ftrans -c -h 10.0.0.1 -sp 8001 -f
100KB.txt -cp 8003
Sending message 100KB.txt to 10.0.0.1:8001
Receiving file...
Length of the file: 102937
File received successfully !
root@ubuntu:/newDisk/ve489/p1-part2/client# diff 100KB.txt ../100KB.txt
root@ubuntu:/newDisk/ve489/p1-part2/client# ./ftrans -c -h 10.0.0.1 -sp 8001 -f
10MB.txt -cp 8003
Sending message 10MB.txt to 10.0.0.1:8001
Receiving file...
Length of the file: 11482040
File received successfully !
root@ubuntu:/newDisk/ve489/p1-part2/client# diff 10MB.txt ../10MB.txt
root@ubuntu:/newDisk/ve489/p1-part2/client#
```

Figure 16: Transferring a 10MB file

2.6 Packets captured when transferring Makefile

To capture the packets, I open 2 terminals for h2. In one of the terminals, I execute tcpdump:

```
tcpdump -i any -s 65535 -w dumpfile
```

I use the same command in section 2.1 to transfer the Makefile:

```
# In terminal of h1:
./ftrans -s -p 8001
# In terminal of h2:
./ftrans -c -h 10.0.0.1 -sp 8001 -f Makefile -cp 8002
```

The dumpfile is successfully generated. Then I open the file in Wireahark, as shown in Figure 17:

No.	Time	Source	Destination	Protocol	Length	Info
1	0.000000	10.0.0.2	10.0.0.1	TCP	76	teradataordbms(8002) → vcom-tunnel(8001) [SYN] Seq=0 Win=42340 Len=0 MSS=1460 SACK_PERM...
2	0.042459	10.0.0.1	10.0.0.2	TCP	76	vcom-tunnel(8001) → teradataordbms(8002) [SYN, ACK] Seq=0 Ack=1 Win=43440 Len=0 MSS=146...
3	0.042569	10.0.0.2	10.0.0.1	TCP	68	teradataordbms(8002) → vcom-tunnel(8001) [ACK] Seq=1 Ack=1 Win=42496 Len=0 TSval=294040...
4	0.042701	10.0.0.2	10.0.0.1	TCP	76	teradataordbms(8002) → vcom-tunnel(8001) [PSH, ACK] Seq=1 Ack=1 Win=42496 Len=8 TSval=2...
5	0.084434	10.0.0.1	10.0.0.2	TCP	68	vcom-tunnel(8001) → teradataordbms(8002) [ACK] Seq=1 Ack=9 Win=43520 Len=0 TSval=345338...
6	0.084473	10.0.0.1	10.0.0.2	TCP	76	vcom-tunnel(8001) → teradataordbms(8002) [PSH, ACK] Seq=1 Ack=9 Win=43520 Len=8 TSval=3...
7	0.084492	10.0.0.2	10.0.0.1	TCP	68	teradataordbms(8002) → vcom-tunnel(8001) [ACK] Seq=9 Ack=9 Win=42496 Len=0 TSval=294040...
8	0.084515	10.0.0.1	10.0.0.2	TCP	580	vcom-tunnel(8001) → teradataordbms(8002) [FIN, PSH, ACK] Seq=9 Ack=9 Win=43520 Len=512...
9	0.085136	10.0.0.2	10.0.0.1	TCP	68	teradataordbms(8002) → vcom-tunnel(8001) [RST, ACK] Seq=9 Ack=522 Win=42496 Len=0 TSval=...

Figure 17: Packets captured

2.6.1 file name

The packet that contains the file name “Makefile” is shown in Figure 18:

```
> Frame 4: 76 bytes on wire (608 bits), 76 bytes captured (608 bits)
> Linux cooked capture
> Internet Protocol Version 4, Src: 10.0.0.2 (10.0.0.2), Dst: 10.0.0.1 (10.0.0.1)
> Transmission Control Protocol, Src Port: teradataordbms (8002), Dst Port: vcom-tunnel (8001), Seq: 1, Ack: 1, Len: 8
▼ Data (8 bytes)
  Data: 4d616b6566696c65
  [Length: 8]
```

```
0000  00 04 00 01 00 06 00 00 00 00 00 02 00 00 08 00  .....
0010  45 00 00 3c 27 70 40 00 40 06 ff 49 0a 00 00 02  E<'p@. @.I...
0020  0a 00 00 01 1f 42 1f 41 85 93 2c 2d df 73 83 9c  ...B.A...s...
0030  80 18 00 53 14 31 00 00 01 01 08 0a af 42 fb 1d  ...S1...B...
0040  14 95 70 b8 4d 61 6b 65 66 69 6c 65             ...pMake file
```

Figure 18: Packet with file name

We can see that the packet contains the string “Makefile”.

2.6.2 file length

The packet that contains the length of “Makefile” is shown in Figure 19:

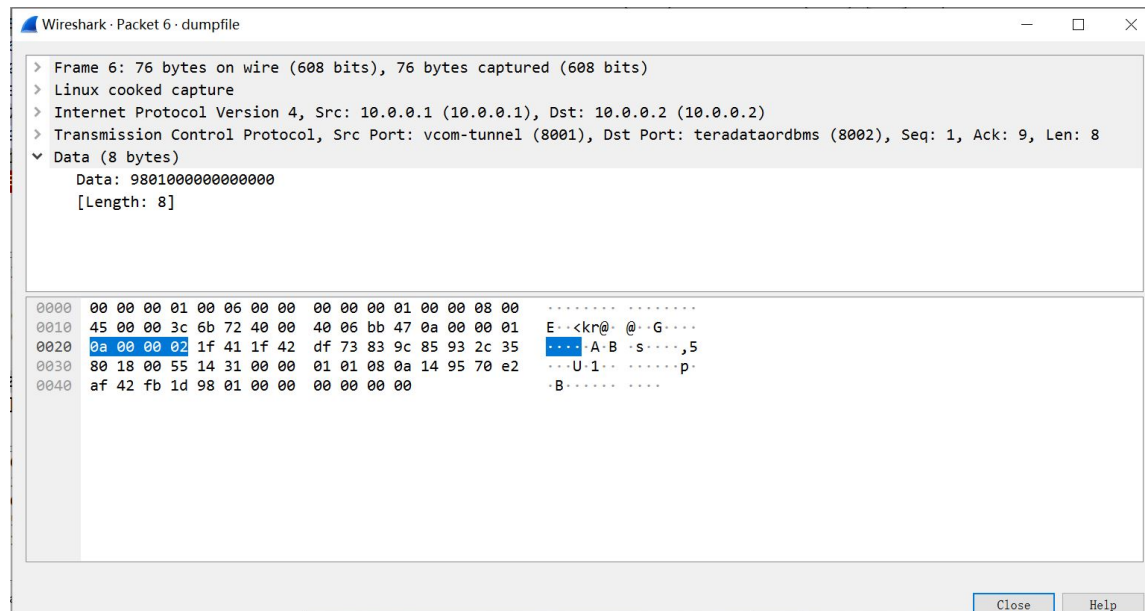


Figure 19: Packet with file length

The information in this packet is a bit trickier. This packet is the only one between the packet with file name and the packet with file content that contains data. However, the data shown here is 9801000000000000 (hex) instead of ascii of 408 (the size of Makefile), because I send the length of Makefile in a “long” type variable instead of a string. Following is what I write for sending the file length:

```
long length = ftell(fp);

if (send(sockfd, &length, sizeof(long), 0) == -1) {
    perror("Error sending on stream socket");
    return;
}
```

It is right because the size of “long” type is exactly 8 bytes, which is the same as the “data” shown in the packet; also, the received value of length is exactly 408 (as shown in Figure 20. I print the length of the file out in the client).

```

"Node: h2"
root@ubuntu:/newDisk/ve489/p1-part2/client# ./ftrans -c -h 10.0.0.1 -sp 8001 -f
Makefile -cp 8002
Sending message Makefile to 10.0.0.1:8001
Receiving file...
Length of the file: 408
File received successfully !
root@ubuntu:/newDisk/ve489/p1-part2/client#

```

Figure 20: File length printed at the client's terminal

It turns out that the “long” type 408 is encoded as 9801000000000000 (hex) when being transferred by TCP.

2.6.3 data

The packet that contains the data (content) of “Makefile” is shown in Figure 21:

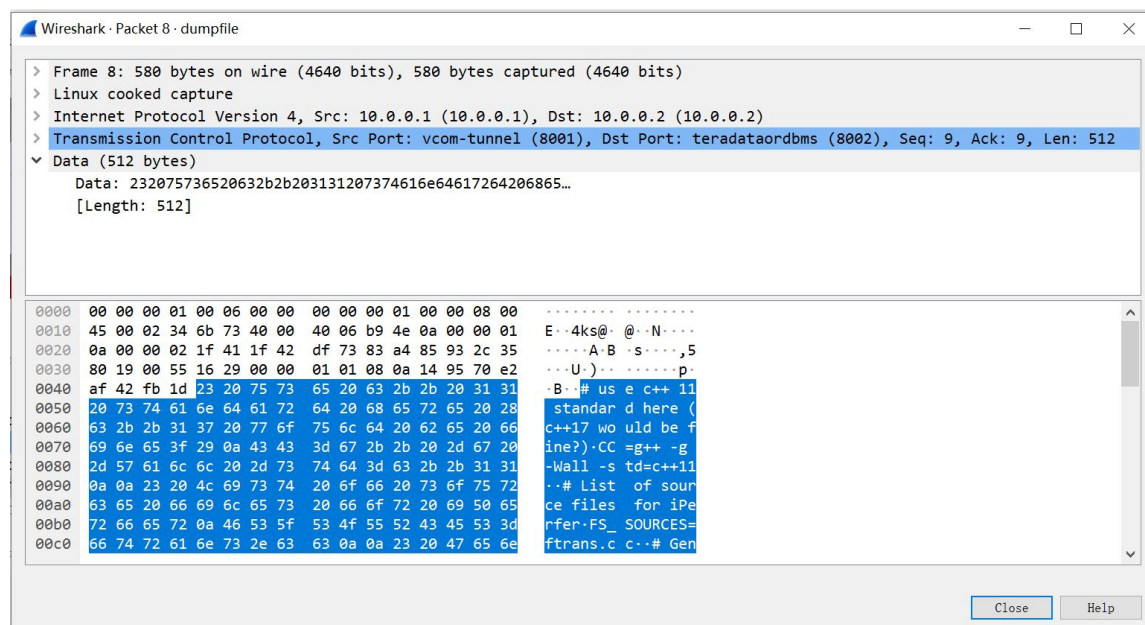


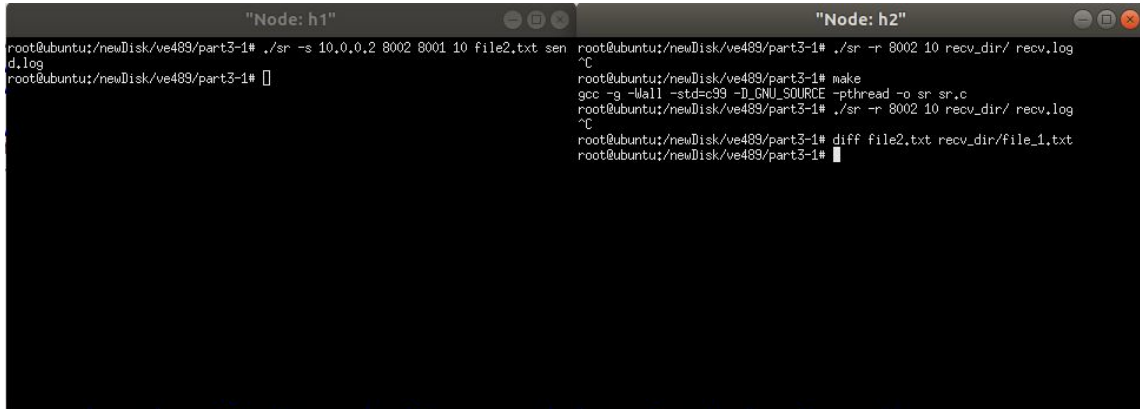
Figure 21: Packet with file content

We can see that the packet contains the content of Makefile.

3 Part 3

3.1 Part 3-1: a simple SR

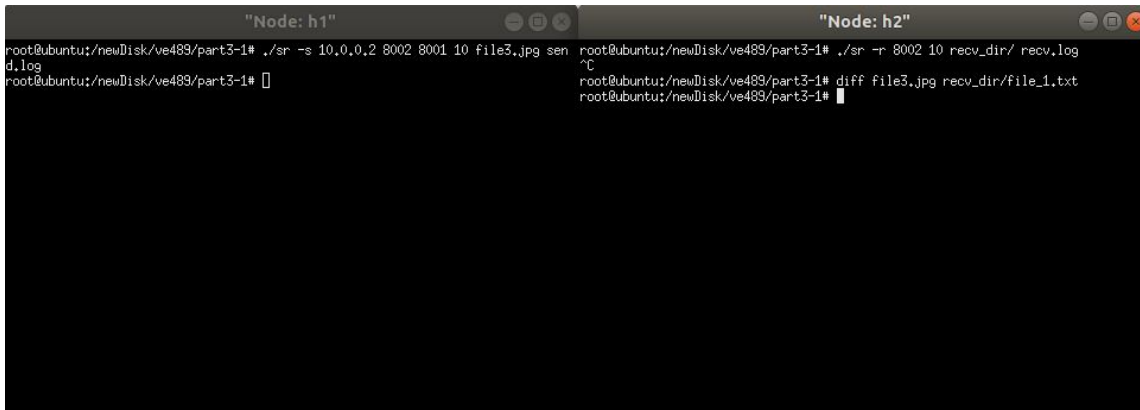
The sender and receiver are implemented in *sr.c*. After simple tests, we can see that the program can deliver file2.txt (Figure 22) and file3.jpg (Figure 23) successfully. The “diff” command outputs nothing.



```
"Node: h1"
root@ubuntu:/newDisk/ve489/part3-1# ./sr -s 10.0.0.2 8002 8001 10 file2.txt send.log
root@ubuntu:/newDisk/ve489/part3-1#

"Node: h2"
root@ubuntu:/newDisk/ve489/part3-1# ./sr -r 8002 10 recv_dir/ recv.log
^C
root@ubuntu:/newDisk/ve489/part3-1# make
gcc -g -Wall -std=c99 -D_GNU_SOURCE -pthread -o sr sr.c
root@ubuntu:/newDisk/ve489/part3-1# ./sr -r 8002 10 recv_dir/ recv.log
^C
root@ubuntu:/newDisk/ve489/part3-1# diff file2.txt recv_dir/file_1.txt
root@ubuntu:/newDisk/ve489/part3-1#
```

Figure 22: Transmit the given file2.txt



```
"Node: h1"
root@ubuntu:/newDisk/ve489/part3-1# ./sr -s 10.0.0.2 8002 8001 10 file3.jpg send.log
root@ubuntu:/newDisk/ve489/part3-1#

"Node: h2"
root@ubuntu:/newDisk/ve489/part3-1# ./sr -r 8002 10 recv_dir/ recv.log
^C
root@ubuntu:/newDisk/ve489/part3-1# diff file3.jpg recv_dir/file_1.txt
root@ubuntu:/newDisk/ve489/part3-1#
```

Figure 23: Transmit the given file3.jpg

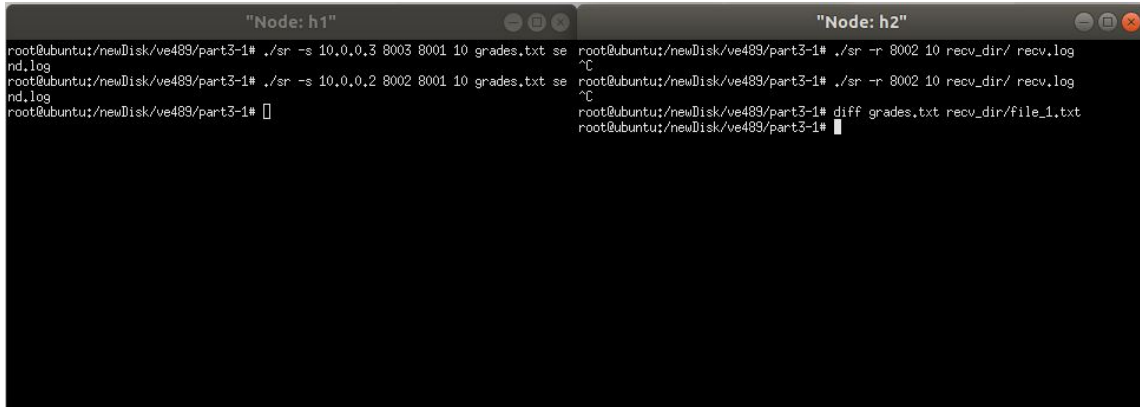
More general testing are shown in the following section.

3.1.1 Testing

In this section, I use a file called “grades.txt” for testing. The file is 2.9MB in size.

A. No reordering, no loss, no error:

As shown in Figure 24, the result is correct (diff outputs nothing).



```
"Node: h1"
root@ubuntu:/newDisk/ve489/part3-1# ./sr -s 10.0.0.3 8003 8001 10 grades.txt se
nd.log
root@ubuntu:/newDisk/ve489/part3-1# ./sr -s 10.0.0.2 8002 8001 10 grades.txt se
nd.log
root@ubuntu:/newDisk/ve489/part3-1#

"Node: h2"
root@ubuntu:/newDisk/ve489/part3-1# ./sr -r 8002 10 recv_dir/ recv.log
^C
root@ubuntu:/newDisk/ve489/part3-1# ./sr -r 8002 10 recv_dir/ recv.log
^C
root@ubuntu:/newDisk/ve489/part3-1# diff grades.txt recv_dir/file_1.txt
root@ubuntu:/newDisk/ve489/part3-1#
```

Figure 24: No reordering, no loss, no error

First few lines of sender's log: The window is slid when ACK is received continuously for packets from the start of the window. For example, initially the window is for DATA from 0 to 9. when ACK 1 is received, it means receiver has received DATA 0, and the window can be slid forward by 1 slot. So sender can send DATA 10. The log results are correct.

```
SYN 2142959629 0 0
ACK 2142959629 0 0
DATA 0 1456 -1738556393
DATA 1 1456 -1471765590
DATA 2 1456 1875770973
DATA 3 1456 2084513169
DATA 4 1456 -796555901
DATA 5 1456 894621758
DATA 6 1456 120980175
DATA 7 1456 1356487621
DATA 8 1456 1615251295
DATA 9 1456 202101618
ACK 1 0 0
DATA 10 1456 -2035574068
ACK 2 0 0
DATA 11 1456 -1267728366
ACK 3 0 0
DATA 12 1456 1842843765
ACK 4 0 0
DATA 13 1456 512032331
ACK 5 0 0
DATA 14 1456 -1936643549
```

First few lines of receiver's log: The window is slided when DATA is received continuously from the start of the window. For example, initially the window is for DATA from 0 to 9. When DATA 0 is received, the window can be slided forward by 1 slot, and ACK 1 is sent. There is then space for DATA 10 in the window. Similarly, when DATA 1 is received, the window can be slided forward again by 1 slot, ACK 2 is sent, and so on. The log results are correct.

```
SYN 2142959629 0 0
ACK 2142959629 0 0
DATA 0 1456 -1738556393
ACK 1 0 0
DATA 1 1456 -1471765590
ACK 2 0 0
DATA 2 1456 1875770973
ACK 3 0 0
DATA 3 1456 2084513169
ACK 4 0 0
DATA 4 1456 -796555901
ACK 5 0 0
DATA 5 1456 894621758
ACK 6 0 0
DATA 6 1456 120980175
ACK 7 0 0
DATA 7 1456 1356487621
ACK 8 0 0
DATA 8 1456 1615251295
ACK 9 0 0
DATA 9 1456 202101618
ACK 10 0 0
```

- B. 10% loss, no reordering, no error
As shown in Figure 25, the result is correct (diff outputs nothing).

```

"Node: h1"
root@ubuntu:/newDisk/ve489/part3-1# ./sr -s 10,0,0,3 8003 8001 10 grades.txt se
nd.log
root@ubuntu:/newDisk/ve489/part3-1# ./sr -s 10,0,0,2 8002 8001 10 grades.txt se
nd.log
root@ubuntu:/newDisk/ve489/part3-1# ./sr -s 10,0,0,3 8003 8001 10 grades.txt se
nd.log
root@ubuntu:/newDisk/ve489/part3-1#

"Node: h2"
root@ubuntu:/newDisk/ve489/part3-1# ./sr -r 8002 10 rcv_dir/ rcv.log
^C
root@ubuntu:/newDisk/ve489/part3-1# ./sr -r 8002 10 rcv_dir/ rcv.log
^C
root@ubuntu:/newDisk/ve489/part3-1# diff grades.txt rcv_dir/file_1.txt
root@ubuntu:/newDisk/ve489/part3-1# ./sr -r 8002 10 rcv_dir/ rcv.log
^C
root@ubuntu:/newDisk/ve489/part3-1# diff grades.txt rcv_dir/file_1.txt
root@ubuntu:/newDisk/ve489/part3-1#

```

Figure 25: 10% loss, no reordering, no error

Example of sender's log: We can see from the following log that data packet 1 is not received by receiver due to packet loss (receiver keeps acking 1). Then the whole window (from DATA 0 to 10) are resent. In the last line of the log segment, ACK 14 is received, which means that all data packets before 14 are received.

```

SYN 28626877 0 0
ACK 28626877 0 0
DATA 0 1456 -1738556393
DATA 1 1456 -1471765590
DATA 2 1456 1875770973
DATA 3 1456 2084513169
DATA 4 1456 -796555901
DATA 5 1456 894621758
DATA 6 1456 120980175
DATA 7 1456 1356487621
DATA 8 1456 1615251295
DATA 9 1456 202101618
ACK 1 0 0
DATA 10 1456 -2035574068
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
ACK 1 0 0
DATA 1 1456 -1471765590
DATA 2 1456 1875770973
DATA 3 1456 2084513169
DATA 4 1456 -796555901
DATA 5 1456 894621758

```

```

DATA 6 1456 120980175
DATA 7 1456 1356487621
DATA 8 1456 1615251295
DATA 9 1456 202101618
DATA 10 1456 -2035574068
ACK 4 0 0
DATA 11 1456 -1267728366
DATA 12 1456 1842843765
DATA 13 1456 512032331
ACK 4 0 0
ACK 4 0 0
ACK 4 0 0
ACK 4 0 0
ACK 4 0 0
ACK 4 0 0
ACK 4 0 0
ACK 4 0 0
ACK 4 0 0
DATA 4 1456 -796555901
DATA 5 1456 894621758
DATA 6 1456 120980175
DATA 7 1456 1356487621
DATA 8 1456 1615251295
DATA 9 1456 202101618
DATA 10 1456 -2035574068
DATA 11 1456 -1267728366
DATA 12 1456 1842843765
DATA 13 1456 512032331
ACK 14 0 0

```

Example of receiver's log: We can see from the following log that data packet 1 is not received by receiver due to packet loss (gap in window). So the receiver keeps sending ACK 1 to request for packet 1. After the data packet 1 is resent, the receiver ACK 4 because packets from 0-3 are received.

```

SYN 28626877 0 0
ACK 28626877 0 0
DATA 0 1456 -1738556393
ACK 1 0 0
DATA 2 1456 1875770973
ACK 1 0 0
DATA 3 1456 2084513169
ACK 1 0 0
DATA 5 1456 894621758
ACK 1 0 0
DATA 7 1456 1356487621
ACK 1 0 0

```

```

DATA 8 1456 1615251295
ACK 1 0 0
DATA 9 1456 202101618
ACK 1 0 0
DATA 10 1456 -2035574068
ACK 1 0 0
DATA 1 1456 -1471765590
ACK 4 0 0

```

- C. 10% error, no reordering, no loss
 As shown in Figure 26, the result is correct (diff outputs nothing).

```

"Node: h1"
root@ubuntu:/newDisk/ve489/part3-1# ./sr -s 10,0,0,3 8003 8001 10 grades.txt se
nd.log
root@ubuntu:/newDisk/ve489/part3-1# ./sr -s 10,0,0,2 8002 8001 10 grades.txt se
nd.log
root@ubuntu:/newDisk/ve489/part3-1# ./sr -s 10,0,0,3 8003 8001 10 grades.txt se
nd.log
root@ubuntu:/newDisk/ve489/part3-1# ./sr -s 10,0,0,3 8003 8001 10 grades.txt se
nd.log
root@ubuntu:/newDisk/ve489/part3-1#

"Node: h2"
root@ubuntu:/newDisk/ve489/part3-1# ./sr -r 8002 10 rcv_dir/ rcv.log
^C
root@ubuntu:/newDisk/ve489/part3-1# ./sr -r 8002 10 rcv_dir/ rcv.log
^C
root@ubuntu:/newDisk/ve489/part3-1# diff grades.txt rcv_dir/file_1.txt
root@ubuntu:/newDisk/ve489/part3-1# ./sr -r 8002 10 rcv_dir/ rcv.log
^C
root@ubuntu:/newDisk/ve489/part3-1# diff grades.txt rcv_dir/file_1.txt
root@ubuntu:/newDisk/ve489/part3-1# ./sr -r 8002 10 rcv_dir/ rcv.log
^C
root@ubuntu:/newDisk/ve489/part3-1# diff grades.txt rcv_dir/file_1.txt
root@ubuntu:/newDisk/ve489/part3-1#

```

Figure 26: 10% error, no reordering, no loss

Example of sender's log: We can see from the following log that data packet 29 is dropped by receiver due to packet corruption (receiver keeps acking 29). Then the whole window (from DATA 29 to 38) are resent.

```

DATA 29 1456 1828795412
ACK 21 0 0
DATA 30 1456 -648842574
ACK 22 0 0
DATA 31 1456 -1189617075
ACK 23 0 0
DATA 32 1456 1589191628
ACK 24 0 0
DATA 33 1456 591609665
ACK 25 0 0
DATA 34 1456 1216886506
ACK 26 0 0
DATA 35 1456 1648757586
ACK 27 0 0

```

```
DATA 36 1456 -1609929852
ACK 28 0 0
DATA 37 1456 -1625879429
ACK 29 0 0
DATA 38 1456 985801412
ACK 29 0 0
ACK 29 0 0
ACK 29 0 0
ACK 29 0 0
ACK 29 0 0
ACK 29 0 0
ACK 29 0 0
ACK 29 0 0
ACK 29 0 0
DATA 29 1456 1828795412
DATA 30 1456 -648842574
DATA 31 1456 -1189617075
DATA 32 1456 1589191628
DATA 33 1456 591609665
DATA 34 1456 1216886506
DATA 35 1456 1648757586
DATA 36 1456 -1609929852
DATA 37 1456 -1625879429
DATA 38 1456 985801412
```

Example of receiver's log: We can see from the following log that data packet 29 is corrupted (received with not ACKed). So the receiver keeps sending ACK 29 to request for packet 29 again. After the data packet 29 is resent, the receiver ACK 39 because packets from 0-38 are received.

```
ACK 28 0 0
DATA 28 1456 -1457788245
ACK 29 0 0
DATA 29 1456 1828795412
DATA 30 1456 -648842574
ACK 29 0 0
DATA 31 1456 -1189617075
ACK 29 0 0
DATA 32 1456 1589191628
ACK 29 0 0
DATA 33 1456 591609665
ACK 29 0 0
DATA 34 1456 1216886506
ACK 29 0 0
DATA 35 1456 1648757586
ACK 29 0 0
DATA 36 1456 -1609929852
```

```

ACK 29 0 0
DATA 37 1456 -1625879429
ACK 29 0 0
DATA 38 1456 985801412
ACK 29 0 0
DATA 29 1456 1828795412
ACK 39 0 0

```

- D. Reordering, no error, no loss
 I am the student who has mysterious segmentation fault of mitm (when involving reorder) in my computer. I have contacted TA and TA has agreed that I do not report this part.
- E. Reordering, 5% loss, 5% error
 I am the student who has mysterious segmentation fault of mitm (when involving reorder) in my computer. I have contacted TA and TA has agreed that I do not report this part.

3.2 Part 3-2: Make sender more efficient – leverage duplicate ACK on sender side

The program is implemented in *sr.c*. In this part, the sender might resend the window when no ACK is received for a long time, to avoid running into a dead loop. Still, catching duplicate ACK and resend immediately has higher priority.

3.2.1 Test duplicate ACK

In this section, I use a file called “grades.txt” for testing. The file is 2.9MB in size.

- A. 10% loss, no reordering, no error
 As shown in Figure 27, the result is correct (diff outputs nothing).

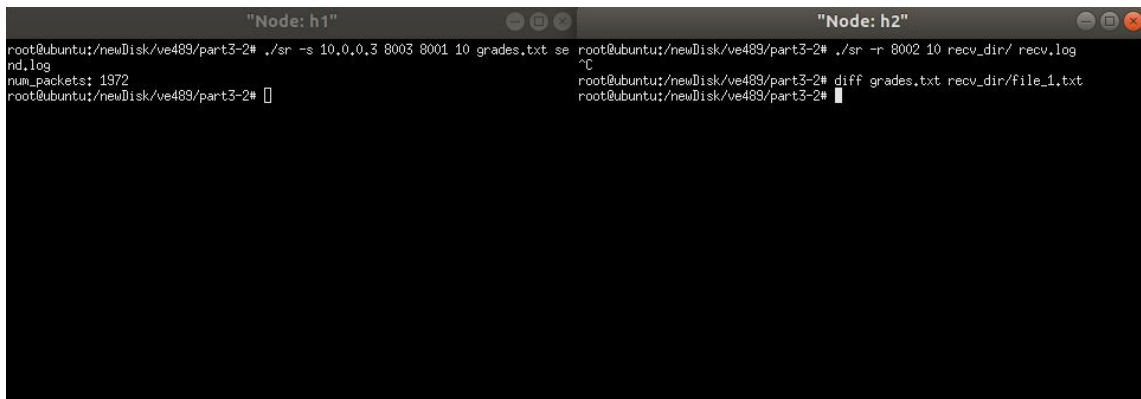


Figure 27: 10% loss, no reordering, no error

Example in sender's log: We can see from the following log segment that data packet 38 is not received by receiver due to packet loss (receiver keeps acking 38). After the sender sees the third duplicated ACK 38, it resends DATA 38 immediately.

```
DATA 44 1456 -1251490922
ACK 36 0 0
DATA 45 1456 955857756
ACK 37 0 0
DATA 46 1456 237601269
ACK 38 0 0
DATA 47 1456 -1272059016
ACK 38 0 0
ACK 38 0 0
DATA 38 1456 985801412
```

B. Efficiency

The run time of my sender for part 3.2 and part 3.1 are shown in Figure 28

```
root@ubuntu:/newDisk/ve489/part3-2# time ./sr -s 10,0,0,3 8003 8001 10 grades.t
xt send.log
num_packets: 1972

real    0m35.400s
user    0m30.285s
sys     0m2.108s
root@ubuntu:/newDisk/ve489/part3-2# cd ../part3-1
root@ubuntu:/newDisk/ve489/part3-1# time ./sr -s 10,0,0,3 8003 8001 10 grades.t
xt send.log

real    1m11.282s
user    1m5.649s
sys     0m3.936s
root@ubuntu:/newDisk/ve489/part3-1#
```

Figure 28: Comparison between part 3.2 and part 3.1

Comparing the user time, the *sr* in part 3.1 takes 1m5.649s, while the *sr* in part 3.2 takes only 30.285s. Therefore, the method of monitoring duplicate ACKs makes the sender more efficient.

3.3 Part 3-3: Make sender more efficient – send NACK on receiver side

The program is implemented in *sr.c*. Whenever a packet is received, the receiver looks for gaps in its window, and actively NACK the first gap that has not been NACKed before. On the sender side, whenever a NACK is received, it retransmits the corresponding packet immediately.

3.3.1 Test NACK

In this section, I use a file called “grades.txt” for testing. The file is 2.9MB in size.

A. 10% loss, no reordering, no error

As shown in Figure 29, the result is correct (diff outputs nothing).


```
"Node: h1" "Node: h2"
root@ubuntu:/newDisk/ve489/part3-3# ./sr -s 10.0.0.3 8003 8001 10 grades.txt se root@ubuntu:/newDisk/ve489/part3-3# ./sr -r 8002 10 recv_dir/ recv.log
nd.log ^C
root@ubuntu:/newDisk/ve489/part3-3# diff grades.txt recv_dir/file_1.txt
root@ubuntu:/newDisk/ve489/part3-3#
```

Figure 29: 10% loss, no reordering, no error

Example in sender's log: We can see from the following log segment that data packet 15 is not received by receiver due to packet loss (receiver sends NACK 15). After the sender sees NACK 15, it resends DATA 15 immediately.

```
DATA 10 1456 -2076735358
DATA 11 1456 1782074020
DATA 12 1456 -412228015
DATA 13 1456 1212748718
DATA 14 1456 307878823
DATA 15 1456 696661505
DATA 16 1456 -593465772
DATA 17 1456 -1676197402
DATA 18 1456 -1866699824
DATA 19 1456 -1756602615
ACK 10 0 0
ACK 11 0 0
DATA 20 1456 2140348665
ACK 12 0 0
DATA 21 1456 312225880
ACK 13 0 0
DATA 22 1456 -534181908
ACK 14 0 0
DATA 23 1456 104963459
ACK 15 0 0
DATA 24 1456 1938158092
NACK 15 0 0
DATA 15 1456 696661505
```

Example in receiver's log: Corresponding to the sender's log segment above, we can see from the following log segment that the receiver receives DATA 11, 12, 13, 14, and 16 in sequence.

However, DATA 15 is not received. So there is a gap in the window of receiver that corresponds to DATA 15. So it sends a NACK 15 to the sender.

```
DATA 10 1456 -2076735358
ACK 11 0 0
DATA 11 1456 1782074020
ACK 12 0 0
DATA 12 1456 -412228015
ACK 13 0 0
DATA 13 1456 1212748718
ACK 14 0 0
DATA 14 1456 307878823
ACK 15 0 0
DATA 16 1456 -593465772
NACK 15 0 0
```

B. Efficiency

The run time of my sender for part 3.3 and part 3.1 are shown in Figure 30. (The time have some difference from the previous tests in 3.2.1 because of the different working status of my computer).

```
root@ubuntu:/newDisk/ve489/part3-3# time ./sr -s 10.0.0.3 8003 8001 10 grades.t
xt send.log
real    1m22.180s
user    1m12.268s
sys     0m5.161s
root@ubuntu:/newDisk/ve489/part3-3# cd ../part3-1
root@ubuntu:/newDisk/ve489/part3-1# time ./sr -s 10.0.0.3 8003 8001 10 grades.t
xt send.log
num_packets: 1972
real    1m55.333s
user    1m42.156s
sys     0m6.166s
root@ubuntu:/newDisk/ve489/part3-1#
```

Figure 30: Comparison between part 3.3 and part 3.1

Comparing the user time, the *sr* in part 3.3 takes 1m12.268s, while the *sr* in part 3.1 takes 1m42.156s. Therefore, the method of NACK makes the sender more efficient.

3.4 Throughput, delay and window size

In this section I transfer the file *file3.jpg* (about 4.2MB) from host *h1* to *h2* using my code in part 3.3, with no loss, no error and no packet reordering. I use the following command for window size 10:

```
# For h1:
./sr -s 10.0.0.2 8002 8001 10 file3.jpg sender.log
# For h2:
./sr -r 8002 10 recv_dir/ recv.log
```

And use the following command for window size 50:

```
# For h1:
./sr -s 10.0.0.2 8002 8001 50 file3.jpg sender.log
# For h2:
./sr -r 8002 50 recv_dir/ recv.log
```

All measured time results for different delays and window sizes are shown in Table 1.

window size \ run time \ delay	0.01ms	10ms	50ms	100ms
10	0.912s	6.67s	30.803s	60.957s
50	0.907s	1.501s	6.415s	12.619s

Table 1: Run time measurements of sender (in seconds)

From the time results, we can calculate the throughput in bps by

$$throughput = \frac{file\ size(b)}{time(s)} = \frac{4.2 \times 1024^2 \times 8(b)}{time(s)} \quad (1)$$

The calculated results of throughput are shown in Table 3.4:

window size \ throughput \ delay	0.01ms	10ms	50ms	100ms
10	38631747.37 bps	5282181.949 bps	1143789.683 bps	577983.7197 bps
50	38844711.8 bps	23472454.1 bps	5492151.769 bps	2791992.519 bps

Table 2: Results of throughput (in bps)

From the table, we can see that when the window size is fixed, the larger the link delay is, the smaller the throughput is. That is because larger delay reduces the data transmission rate. Also, when the link delay is fixed, the larger the window size is, the larger the throughput is. That is because when there is no error or loss in transmission, sending more data at a time is more efficient (need to wait for less ACKs).

4 Conclusion

In this project, I used mininet to construct a network and did socket programming to learn about the TCP protocol and UDP protocol. It helps me a lot in understanding the working principle of the protocols.