VE489

Computer Network

Report of Project

2020 Summer

Name:

Junjie Zhang

Student Number:

517021911128

Instructor:

Dr. Xudong Wang

July 28, 2020

Contents

1 Part I

1.1 Link latency using ping

```
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=44.2 ms
64 bytes from 10.0.0.2: icmp_seq=2 ttl=64 time=43.8 ms
64 bytes from 10.0.0.2: icmp_seq=3 ttl=64 time=41.7 ms
64 bytes from 10.0.0.2: icmp_seq=4 ttl=64 time=41.3 ms
64 bytes from 10.0.0.2: icmp_seq=5 ttl=64 time=41.4 ms
64 bytes from 10.0.0.2: icmp_seq=6 ttl=64 time=40.0 ms
64 bytes from 10.0.0.2: icmp_seq=7 ttl=64 time=40.0 ms
64 bytes from 10.0.0.2: icmp_seq=8 ttl=64 time=40.6 ms
64 bytes from 10.0.0.2: icmp_seq=9 ttl=64 time=40.0 ms
64 bytes from 10.0.0.2: icmp_seq=10 ttl=64 time=40.0 ms

--- 10.0.0.2 ping statistics ---
10 packets transmitted, 10 received, 0% packet loss, time 9017ms
rtt min/avg/max/mdev = 40.042/41.349/44.251/1.506 ms
```

Figure 1: Ping result between $h1$ and $h2$

From Figure 1 we can see that the average RTT is $41.348ms$ between $h1$ and $h2$, which indicates $L1$ link is used, and RTT is close to 2 times link latency.

```
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=23.4 ms
64 bytes from 10.0.0.5: icmp_seq=2 ttl=64 time=22.7 ms
64 bytes from 10.0.0.5: icmp_seq=3 ttl=64 time=21.5 ms
64 bytes from 10.0.0.5: icmp_seq=4 ttl=64 time=20.8 ms
64 bytes from 10.0.0.5: icmp_seq=5 ttl=64 time=20.6 ms
64 bytes from 10.0.0.5: icmp_seq=6 ttl=64 time=21.2 ms
64 bytes from 10.0.0.5: icmp_seq=7 ttl=64 time=20.7 ms
64 bytes from 10.0.0.5: icmp_seq=8 ttl=64 time=20.6 ms
64 bytes from 10.0.0.5: icmp_seq=9 ttl=64 time=21.5 ms
64 bytes from 10.0.0.5: icmp_seq=10 ttl=64 time=20.5 ms

--- 10.0.0.5 ping statistics ---
10 packets transmitted, 10 received, 0% packet loss, time 9012ms
rtt min/avg/max/mdev = 20.571/21.392/23.434/0.940 ms
```

Figure 2: Ping result between $h3$ and $h5$

From Figure 2 we can see that the average RTT is $41.348ms$ between $h3$ and $h5$, which indicates $L1$ link is used, and RTT is close to 2 times link latency.

1.2 Path latency using ping

For $h1$ and $h5$, the path should be $L1 \leftrightarrow L2 \leftrightarrow L4$, then the theoretical RTT is:

$$RTT_{(h1,h5)the} = 2 \times (40 + 20 + 10) = 140ms$$

From Figure 3, we can see that the average RTT is $142.530ms$, which is very close to the theoretical one.

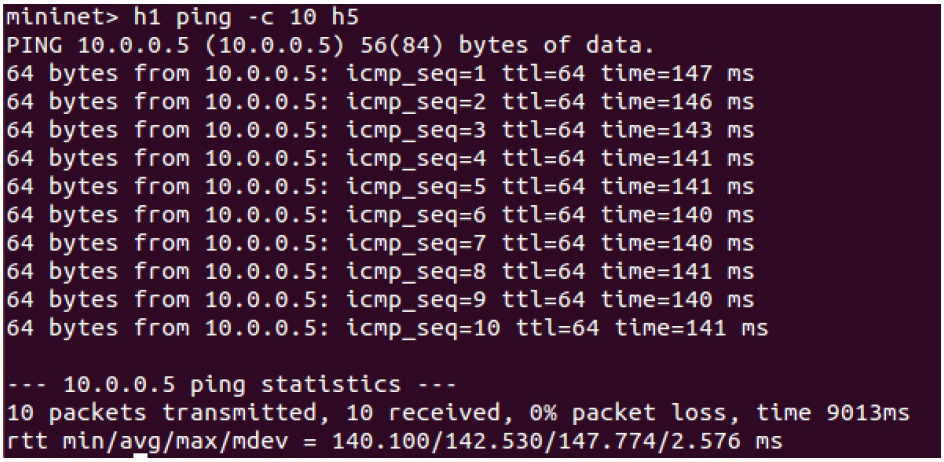


Figure 3: Ping result between $h1$ and $h5$

For $h3$ and $h4$, the path should be $L2 \leftrightarrow L3$, then the theoretical RTT is:

$$RTT_{(h3,h4)the} = 2 \times (40 + 30) = 140ms$$

From Figure 4, we can see that the average RTT is $142.700ms$, which is very close to the theoretical one.

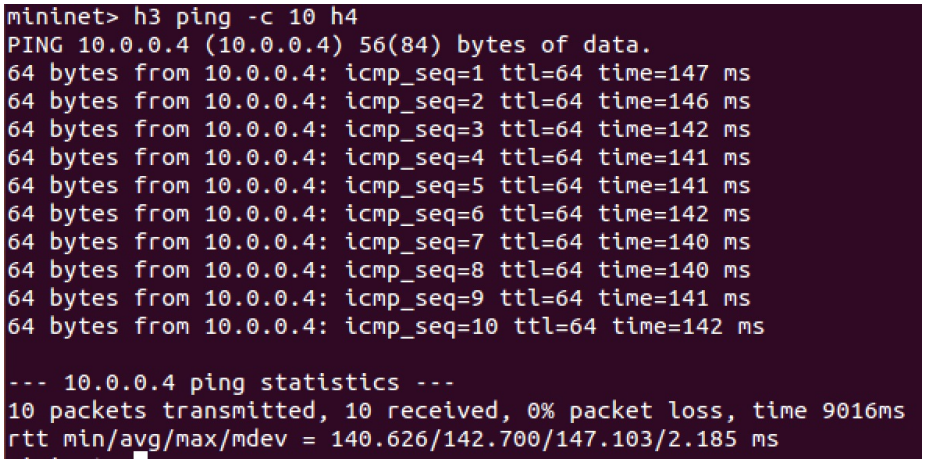


Figure 4: Ping result between $h3$ and $h4$

1.3 Link bandwidth using iperf

For $h1$ and $h2$, let $h1$ be the server and let $h2$ be the client. Then the bandwidth of $h1$ can be seen in Figure 5, which is $46.8Mbits/s$. And the bandwidth of $h2$ can be seen in Figure 6, which is $56.3Mbits/s$. The bandwidth of $L1$ is $50Mbits/s$, which is close to above two. And from Figure 5 and Figure 6 we can see that the sizes of data transferred and received are the same, which is $35.6MBytes$.

For $h3$ and $h5$, let $h3$ be the server and let $h5$ be the client. Then the bandwidth of $h3$ can be seen in Figure 7, which is $9.55Mbits/s$. And the bandwidth of $h2$ can be seen in Figure 8, which is $11.5Mbits/s$. The bandwidth of $L1$ is $10Mbits/s$, which is close to above two. And from Figure 7 and

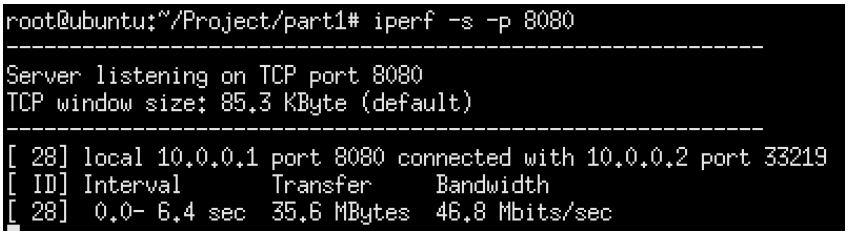


Figure 5: Server on $h1$

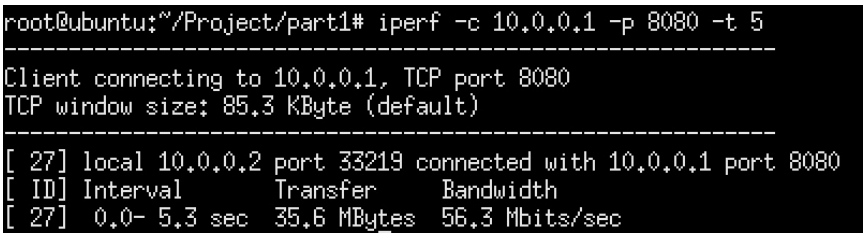


Figure 6: Client on $h2$

Figure 8 we can see that the sizes of data transferred and received are the same, which is $7.00MBytes$.

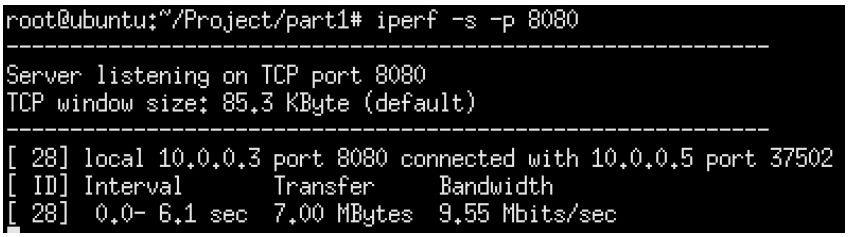


Figure 7: Server on $h3$

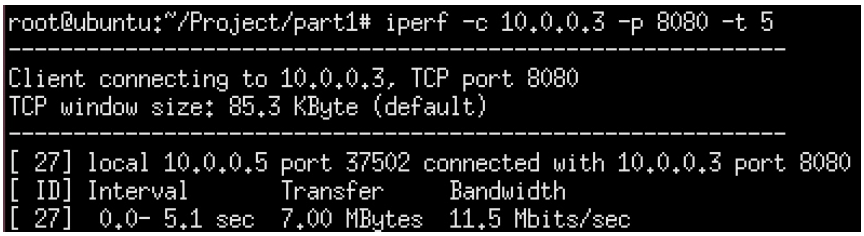


Figure 8: Client on $h5$

1.4 Path throughput using iperf

For $h1$ and $h5$, let $h1$ be the server and let $h5$ be the client. Then the bandwidth of $h1$ can be seen in Figure 9, which is $8.84Mbits/s$. And the bandwidth of $h5$ can be seen in Figure 10, which is $11.1Mbits/s$. The bottleneck link is $L4$, whose bandwidth is $10Mbits/s$, which is close to above two. And from Figure 9 and Figure 10 we can see that the sizes of data transferred and received are the same, which is $7.25MBytes$.

For $h3$ and $h4$, let $h3$ be the server and let $h4$ be the client. Then the bandwidth of $h3$ can be seen in Figure 11, which is $17.3Mbits/s$. And the bandwidth of $h5$ can be seen in Figure 12, which is

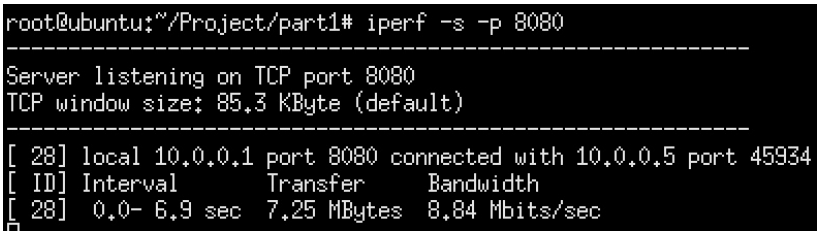


Figure 9: Server on $h1$

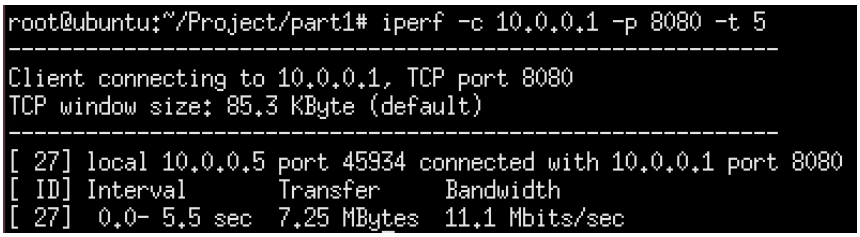


Figure 10: Client on $h5$

$19.1Mbits/s$. The bottleneck link is $L3$, whose bandwidth is $20Mbits/s$, which is close to above two. And from Figure 11 and Figure 12 we can see that the sizes of data transferred and received are the same, which is $11.5MBytes$.

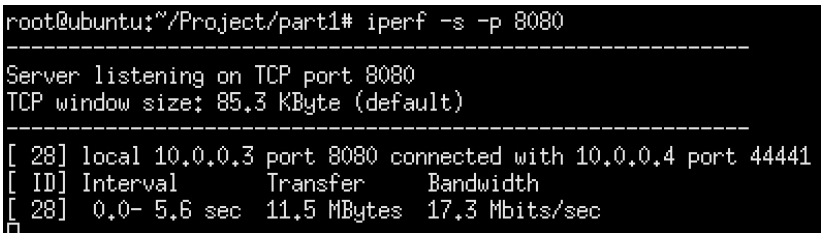


Figure 11: Server on $h3$

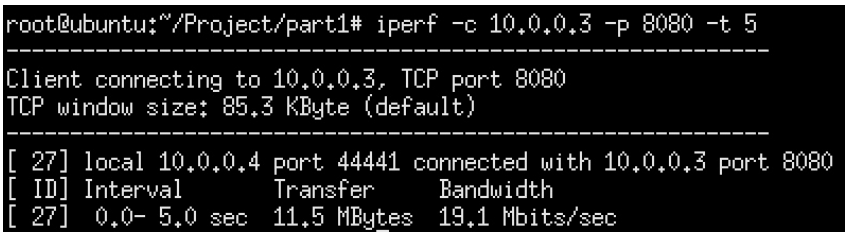


Figure 12: Client on $h4$

1.5 Multiplexing

For the latency. From Figure 13, we can see that the average RTT between $h1$ and $h5$ is $141.702ms$, which is close to the theoretical one. And from Figure 14, we can see that the average RTT between $h3$ and 45 is $142.995ms$, which is also close to the theoretical one. Compared with question 2, the result stay unchanged. And link $L2$ is shared, since the rate of sending packet by ping is much less than the bandwidth of $L2$, we can say that the link is shared fairly and the result won't change.

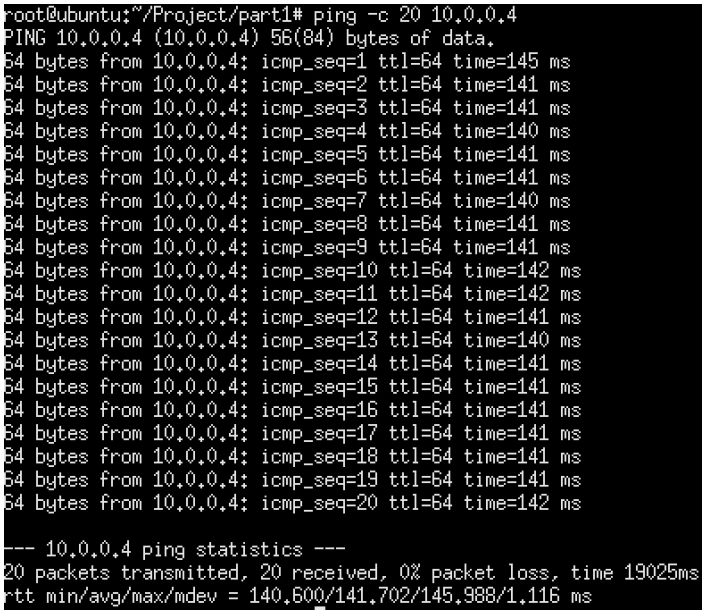


Figure 13: Ping result between $h1$ and $h5$

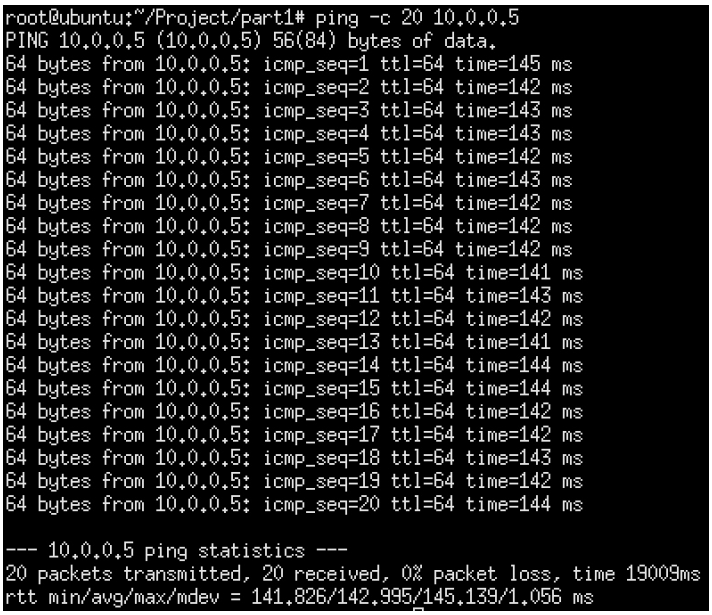


Figure 14: Ping result between $h1$ and $h5$

For the bandwidth. Let $h1$ and $h3$ be servers. Let $h3$ and $h4$ be clients. From Figure 15, we can see that the bandwidth for $h1$ is $2.33Mbits/s$ and the bandwidth for $h5$ is $2.33Mbits/s$. Both of them are not close to the bandwidth of the bottleneck link, $L4$, which is $10Mbits/s$. From Figure 16, we can see that the bandwidth for $h1$ is $17.6Mbits/s$ and the bandwidth for $h5$ is $22.8Mbits/s$. Both of them are relatively close to the bandwidth of the bottleneck link, $L2$, which is $20Mbits/s$. This indicate that the result is not the same as the one of question 4. The shared link is $L2$ whose bandwidth is $20Mbits/s$. Since the total bandwidth of two bottleneck link is $30Mbits$, which is larger than the bandwidth of the shared link, it means the result will differ from the individual one. And clearly, $L2$ is not shared fairly.

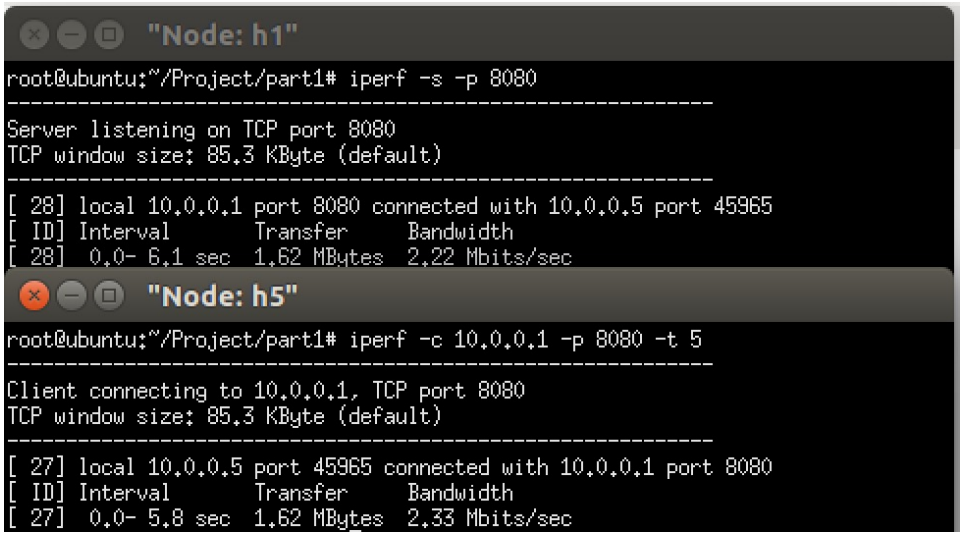


Figure 15: Server on $h1$ and Client on $h5$

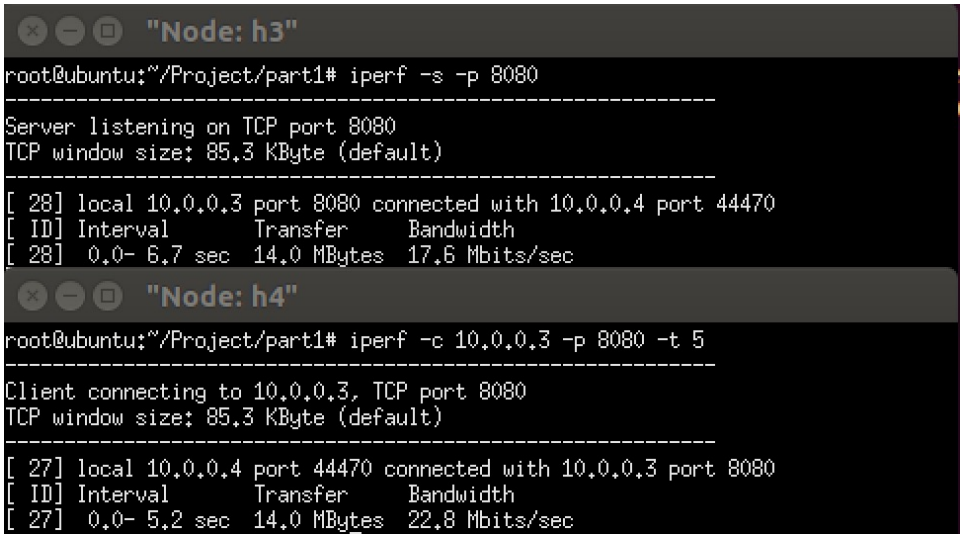


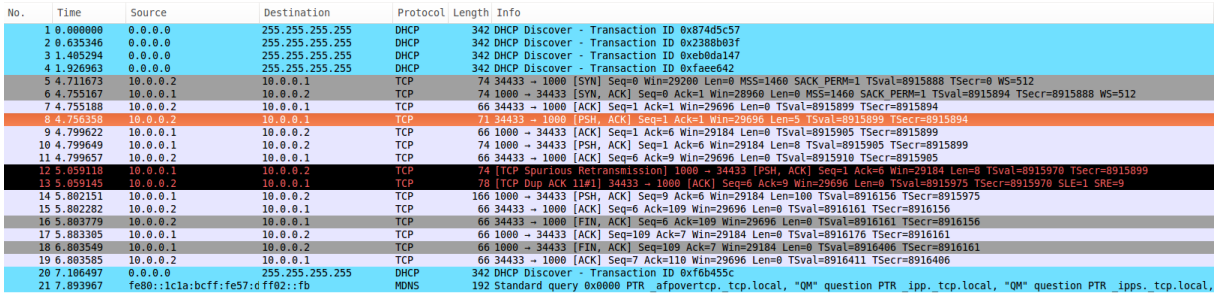
Figure 16: Server on $h3$ and Client on $h4$

2 Part II

2.1 Implementation

The whole process is as follows:

1. Client sends $filename$ to server.
2. Server reads $filename$ and open file.
3. Server sends *length* to client.
4. Client reads *length*.
5. Server sends $file\_data$ to client.
6. Client reads $file\_data$ (length bytes in total) and writes to $filename$.



No.	Time	Source	Destination	Protocol	Length	Info
1	0.000000	0.0.0.0	255.255.255.255	DHCP	342	DHCP Discover - Transaction ID 0x874d5c57
2	0.635346	0.0.0.0	255.255.255.255	DHCP	342	DHCP Discover - Transaction ID 0x2388b03f
3	1.405294	0.0.0.0	255.255.255.255	DHCP	342	DHCP Discover - Transaction ID 0xeb0da147
4	1.926963	0.0.0.0	255.255.255.255	DHCP	342	DHCP Discover - Transaction ID 0xfaee642
5	4.711673	10.0.0.2	10.0.0.1	TCP	74	34433 → 1000 [SYN] Seq=0 Win=29200 Len=0 MSS=1460 SACK_PERM=1 TSval=8915888 TSecr=0 WS=512
6	4.755167	10.0.0.1	10.0.0.2	TCP	74	1000 → 34433 [SYN, ACK] Seq=0 Ack=1 Win=28960 Len=0 MSS=1460 SACK_PERM=1 TSval=8915894 TSecr=8915888 WS=512
7	4.755188	10.0.0.2	10.0.0.1	TCP	66	34433 → 1000 [ACK] Seq=1 Ack=1 Win=29696 Len=0 TSval=8915899 TSecr=8915894
8	4.756358	10.0.0.2	10.0.0.1	TCP	71	34433 → 1000 [PSH, ACK] Seq=1 Ack=1 Win=29696 Len=5 TSval=8915899 TSecr=8915894
9	4.799622	10.0.0.1	10.0.0.2	TCP	66	1000 → 34433 [ACK] Seq=1 Ack=6 Win=29184 Len=0 TSval=8915905 TSecr=8915899
10	4.799649	10.0.0.1	10.0.0.2	TCP	74	1000 → 34433 [PSH, ACK] Seq=1 Ack=6 Win=29184 Len=8 TSval=8915905 TSecr=8915899
11	4.799657	10.0.0.2	10.0.0.1	TCP	66	34433 → 1000 [ACK] Seq=6 Ack=9 Win=29696 Len=0 TSval=8915910 TSecr=8915905
12	5.059118	10.0.0.1	10.0.0.2	TCP	74	[TCP Spurious Retransmission] 1000 → 34433 [PSH, ACK] Seq=1 Ack=6 Win=29184 Len=8 TSval=8915970 TSecr=8915899
13	5.059145	10.0.0.2	10.0.0.1	TCP	78	[TCP Dup ACK 11#1] 34433 → 1000 [ACK] Seq=6 Ack=9 Win=29696 Len=0 TSval=8915975 TSecr=8915970 SLE=1 SRE=9
14	5.802151	10.0.0.1	10.0.0.2	TCP	166	1000 → 34433 [PSH, ACK] Seq=9 Ack=6 Win=29184 Len=100 TSval=8916156 TSecr=8915975
15	5.802282	10.0.0.2	10.0.0.1	TCP	66	34433 → 1000 [ACK] Seq=6 Ack=109 Win=29696 Len=0 TSval=8916161 TSecr=8916156
16	5.803779	10.0.0.2	10.0.0.1	TCP	66	34433 → 1000 [FIN, ACK] Seq=6 Ack=109 Win=29696 Len=0 TSval=8916161 TSecr=8916156
17	5.803305	10.0.0.1	10.0.0.2	TCP	66	1000 → 34433 [ACK] Seq=109 Ack=7 Win=29184 Len=0 TSval=8916176 TSecr=8916161
18	6.803549	10.0.0.1	10.0.0.2	TCP	66	1000 → 34433 [FIN, ACK] Seq=109 Ack=7 Win=29184 Len=0 TSval=8916406 TSecr=8916161
19	6.803585	10.0.0.2	10.0.0.1	TCP	66	34433 → 1000 [ACK] Seq=7 Ack=110 Win=29696 Len=0 TSval=8916411 TSecr=8916406
20	7.106497	0.0.0.0	255.255.255.255	DHCP	342	DHCP Discover - Transaction ID 0xf6b455c
21	7.893967	fe80::1c1a:bcff:fe57:cff02::fb		MDNS	192	Standard query 0x0000 PTR _afpovertcp._tcp.local, "QM" question PTR _ipp._tcp.local, "QM" question PTR _ipps._tcp.local,

Figure 17: The packets captured when transmit file with size of 100B

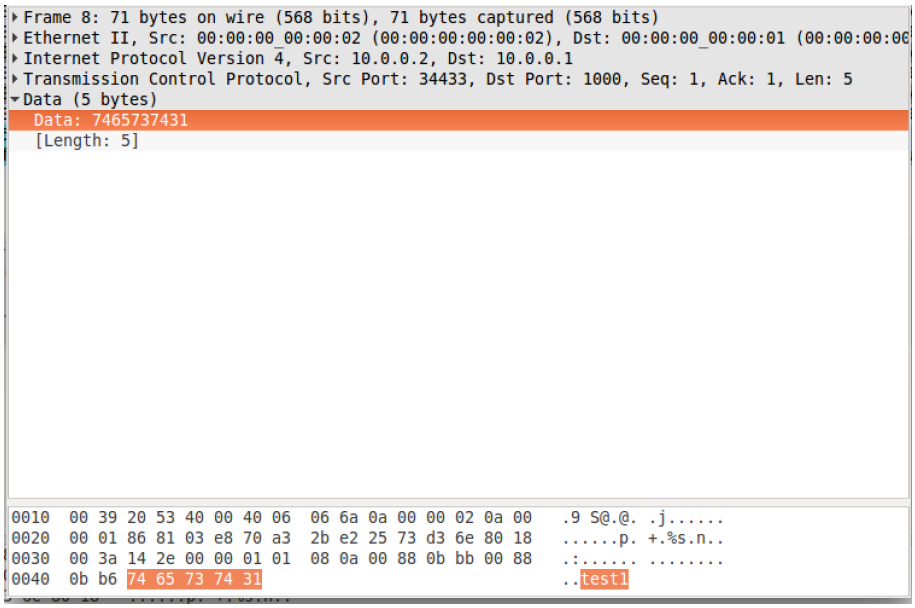


Figure 18: Detailed information about packet No.8

7. Client exits, server keeps running.

And I choose 1024 bytes as the buffer size, the code of `ftrans.cc` and `Makefile` can be seen in **Appendix**.

2.2 Results

The server is set on $h1$ and the client is set on $h2$.

2.2.1 Transmits file with size of 100B

The file is generated by command `dd if=/dev/zero of=test1 bs=1 count=100`

We captured the packets sent and received on client with `tcpdump`, and the result is shown through `WireShark`, which can be seen in Figure 17.

Then, we can see that `packet No.8` contains the file name, which is from $h2(10.0.0.2)$ to $h1(10.0.0.1)$ and can be seen in Figure 19. The file name is $test1$, which is correct.

And `packet No.10` contains the file length, which is from $h1$ to $h2$. The file length is $0x64 = 100$ bytes,which is correct.

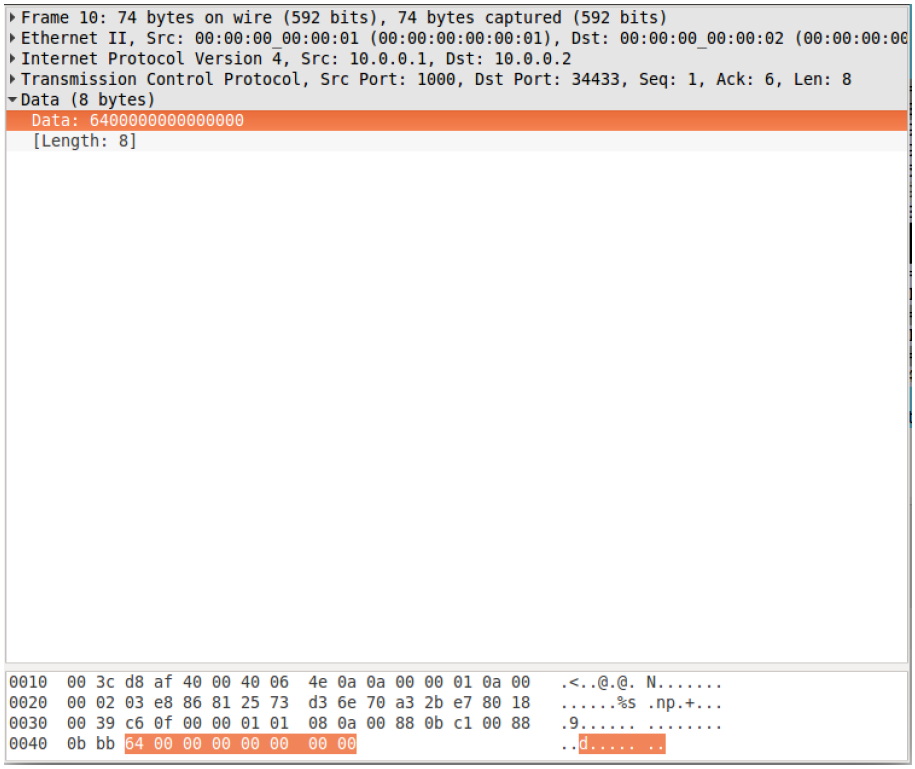


Figure 19: Detailed information about packet No.10

And `packet No.14` contains the data, which is from $h1$ to $h2$.By using `diff`, we can say that the transmission is correct.

2.2.2 Transmits file with size of 100KB

The file is generated by command `dd if=/dev/zero of=test1 bs=1KB count=100`

We captured the packets sent and received on client with `tcpdump`, and the part of the result is shown through `WireShark`, which can be seen in Figure 20.

Then, we can see that `packet No.8` contains the file name, which is from $h2(10.0.0.2)$ to $h1(10.0.0.1)$ and can be seen in Figure 19. The file name is $test2$, which is correct.

And `packet No.10` contains the file length, which is from $h1$ to $h2$. The file length is $0x0186a0 =$ 100000 bytes = 100KB, which is correct.

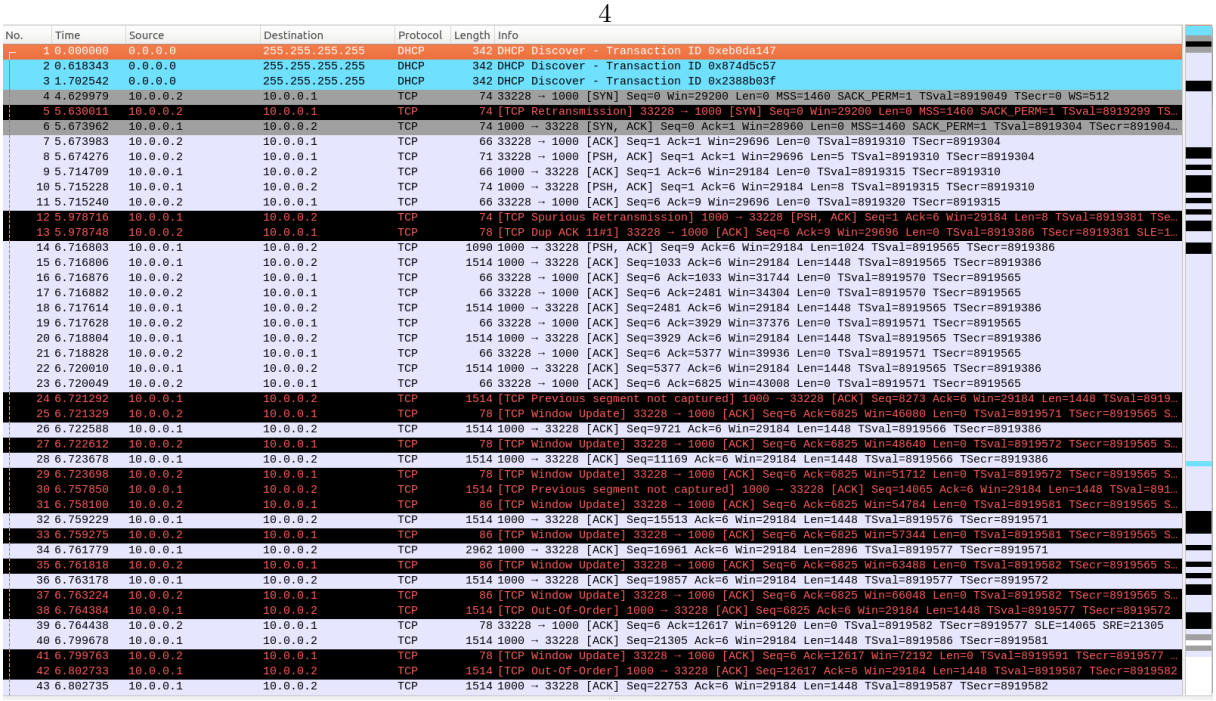
And packets with size of more than 1000 bytes contain the data, which is from $h1$ to $h2$. By using `diff`, we can say that the transmission is correct.

2.2.3 Transmits file with size of 10MB

The file is generated by command `dd if=/dev/zero of=test1 bs=1MB count=10`

We captured the packets sent and received on client with `tcpdump`, and the part of the result is shown through `WireShark`, which can be seen in Figure 23.

Then, we can see that `packet No.22` contains the file name, which is from $h2(10.0.0.2)$ to $h1(10.0.0.1)$ and can be seen in Figure 24. The file name is $test3$, which is correct.



No.	Time	Source	Destination	Protocol	Length	Info
1	0.000000	0.0.0.0	255.255.255.255	DHCP	342	DHCP Discover - Transaction ID 0xeb0da147
2	0.618343	0.0.0.0	255.255.255.255	DHCP	342	DHCP Discover - Transaction ID 0x874d5c57
3	1.702542	0.0.0.0	255.255.255.255	DHCP	342	DHCP Discover - Transaction ID 0x2388b03f
4	4.629979	10.0.0.2	10.0.0.1	TCP	74	33228 → 1000 [SYN] Seq=0 Win=29200 Len=0 MSS=1460 SACK_PERM=1 TSval=8919049 TSecr=0 WS=512
5	5.630011	10.0.0.2	10.0.0.1	TCP	74	[TCP Retransmission] 33228 → 1000 [SYN] Seq=0 Win=29200 Len=0 MSS=1460 SACK_PERM=1 TSval=8919299 TS…
6	5.673962	10.0.0.1	10.0.0.2	TCP	74	1000 → 33228 [SYN, ACK] Seq=0 Ack=1 Win=28960 Len=0 MSS=1460 SACK_PERM=1 TSval=8919304 TSecr=891904…
7	5.673983	10.0.0.2	10.0.0.1	TCP	66	33228 → 1000 [ACK] Seq=1 Ack=1 Win=29696 Len=0 TSval=8919310 TSecr=8919304
8	5.674276	10.0.0.2	10.0.0.1	TCP	71	33228 → 1000 [PSH, ACK] Seq=1 Ack=1 Win=29696 Len=5 TSval=8919310 TSecr=8919304
9	5.714709	10.0.0.1	10.0.0.2	TCP	66	1000 → 33228 [ACK] Seq=1 Ack=6 Win=29184 Len=0 TSval=8919315 TSecr=8919310
10	5.715228	10.0.0.1	10.0.0.2	TCP	74	1000 → 33228 [PSH, ACK] Seq=1 Ack=6 Win=29184 Len=8 TSval=8919315 TSecr=8919310
11	5.715240	10.0.0.2	10.0.0.1	TCP	66	33228 → 1000 [ACK] Seq=6 Ack=9 Win=29696 Len=0 TSval=8919320 TSecr=8919315
12	5.978716	10.0.0.1	10.0.0.2	TCP	74	[TCP Spurious Retransmission] 1000 → 33228 [PSH, ACK] Seq=1 Ack=6 Win=29184 Len=8 TSval=8919381 TSe…
13	5.978748	10.0.0.2	10.0.0.1	TCP	78	[TCP Dup ACK 11#1] 33228 → 1000 [ACK] Seq=6 Ack=9 Win=29696 Len=0 TSval=8919386 TSecr=8919381 SLE=1…
14	6.716803	10.0.0.1	10.0.0.2	TCP	1090	1000 → 33228 [PSH, ACK] Seq=9 Ack=6 Win=29184 Len=1024 TSval=8919565 TSecr=8919386
15	6.716806	10.0.0.1	10.0.0.2	TCP	1514	1000 → 33228 [ACK] Seq=1033 Ack=6 Win=29184 Len=1448 TSval=8919565 TSecr=8919386
16	6.716876	10.0.0.2	10.0.0.1	TCP	66	33228 → 1000 [ACK] Seq=6 Ack=1033 Win=31744 Len=0 TSval=8919570 TSecr=8919565
17	6.716882	10.0.0.2	10.0.0.1	TCP	66	33228 → 1000 [ACK] Seq=6 Ack=2481 Win=34304 Len=0 TSval=8919570 TSecr=8919565
18	6.717614	10.0.0.1	10.0.0.2	TCP	1514	1000 → 33228 [ACK] Seq=2481 Ack=6 Win=29184 Len=1448 TSval=8919565 TSecr=8919386
19	6.717628	10.0.0.2	10.0.0.1	TCP	66	33228 → 1000 [ACK] Seq=6 Ack=3929 Win=37376 Len=0 TSval=8919571 TSecr=8919565
20	6.718804	10.0.0.1	10.0.0.2	TCP	1514	1000 → 33228 [ACK] Seq=3929 Ack=6 Win=29184 Len=1448 TSval=8919565 TSecr=8919386
21	6.718828	10.0.0.2	10.0.0.1	TCP	66	33228 → 1000 [ACK] Seq=6 Ack=5377 Win=39936 Len=0 TSval=8919571 TSecr=8919565
22	6.720010	10.0.0.1	10.0.0.2	TCP	1514	1000 → 33228 [ACK] Seq=5377 Ack=6 Win=29184 Len=1448 TSval=8919565 TSecr=8919386
23	6.720049	10.0.0.2	10.0.0.1	TCP	66	33228 → 1000 [ACK] Seq=6 Ack=6825 Win=43008 Len=0 TSval=8919571 TSecr=8919565
24	6.721292	10.0.0.1	10.0.0.2	TCP	1514	[TCP Previous segment not captured] 1000 → 33228 [ACK] Seq=8273 Ack=6 Win=29184 Len=1448 TSval=8919…
25	6.721329	10.0.0.2	10.0.0.1	TCP	78	[TCP Window Update] 33228 → 1000 [ACK] Seq=6 Ack=6825 Win=46080 Len=0 TSval=8919571 TSecr=8919565 S…
26	6.722588	10.0.0.1	10.0.0.2	TCP	1514	1000 → 33228 [ACK] Seq=9721 Ack=6 Win=29184 Len=1448 TSval=8919566 TSecr=8919386
27	6.722612	10.0.0.2	10.0.0.1	TCP	78	[TCP Window Update] 33228 → 1000 [ACK] Seq=6 Ack=6825 Win=48640 Len=0 TSval=8919572 TSecr=8919565 S…
28	6.723678	10.0.0.1	10.0.0.2	TCP	1514	1000 → 33228 [ACK] Seq=11169 Ack=6 Win=29184 Len=1448 TSval=8919566 TSecr=8919386
29	6.723698	10.0.0.2	10.0.0.1	TCP	78	[TCP Window Update] 33228 → 1000 [ACK] Seq=6 Ack=6825 Win=51712 Len=0 TSval=8919572 TSecr=8919565 S…
30	6.757850	10.0.0.1	10.0.0.2	TCP	1514	[TCP Previous segment not captured] 1000 → 33228 [ACK] Seq=14065 Ack=6 Win=29184 Len=1448 TSval=891…
31	6.758100	10.0.0.2	10.0.0.1	TCP	86	[TCP Window Update] 33228 → 1000 [ACK] Seq=6 Ack=6825 Win=54784 Len=0 TSval=8919581 TSecr=8919565 S…
32	6.759229	10.0.0.1	10.0.0.2	TCP	1514	1000 → 33228 [ACK] Seq=15513 Ack=6 Win=29184 Len=1448 TSval=8919576 TSecr=8919571
33	6.759275	10.0.0.2	10.0.0.1	TCP	86	[TCP Window Update] 33228 → 1000 [ACK] Seq=6 Ack=6825 Win=57344 Len=0 TSval=8919581 TSecr=8919565 S…
34	6.761779	10.0.0.1	10.0.0.2	TCP	2962	1000 → 33228 [ACK] Seq=16961 Ack=6 Win=29184 Len=2896 TSval=8919577 TSecr=8919571
35	6.761818	10.0.0.2	10.0.0.1	TCP	86	[TCP Window Update] 33228 → 1000 [ACK] Seq=6 Ack=6825 Win=63488 Len=0 TSval=8919582 TSecr=8919565 S…
36	6.763178	10.0.0.1	10.0.0.2	TCP	1514	1000 → 33228 [ACK] Seq=19857 Ack=6 Win=29184 Len=1448 TSval=8919577 TSecr=8919572
37	6.763224	10.0.0.2	10.0.0.1	TCP	86	[TCP Window Update] 33228 → 1000 [ACK] Seq=6 Ack=6825 Win=66048 Len=0 TSval=8919582 TSecr=8919565 S…
38	6.764384	10.0.0.1	10.0.0.2	TCP	1514	[TCP Out-Of-Order] 1000 → 33228 [ACK] Seq=6825 Ack=6 Win=29184 Len=1448 TSval=8919577 TSecr=8919572
39	6.764438	10.0.0.2	10.0.0.1	TCP	78	33228 → 1000 [ACK] Seq=6 Ack=12617 Win=69120 Len=0 TSval=8919582 TSecr=8919577 SLE=14065 SRE=21305
40	6.799678	10.0.0.1	10.0.0.2	TCP	1514	1000 → 33228 [ACK] Seq=21305 Ack=6 Win=29184 Len=1448 TSval=8919586 TSecr=8919581
41	6.799763	10.0.0.2	10.0.0.1	TCP	78	[TCP Window Update] 33228 → 1000 [ACK] Seq=6 Ack=12617 Win=72192 Len=0 TSval=8919591 TSecr=8919577 …
42	6.802733	10.0.0.1	10.0.0.2	TCP	1514	[TCP Out-Of-Order] 1000 → 33228 [ACK] Seq=12617 Ack=6 Win=29184 Len=1448 TSval=8919587 TSecr=8919582
43	6.802735	10.0.0.1	10.0.0.2	TCP	1514	1000 → 33228 [ACK] Seq=22753 Ack=6 Win=29184 Len=1448 TSval=8919587 TSecr=8919582

Figure 20: Part of the packets captured when transmit file with size of 100KB

Figure 21: Detailed information about packet No.8

Figure 22: Detailed information about packet No.10

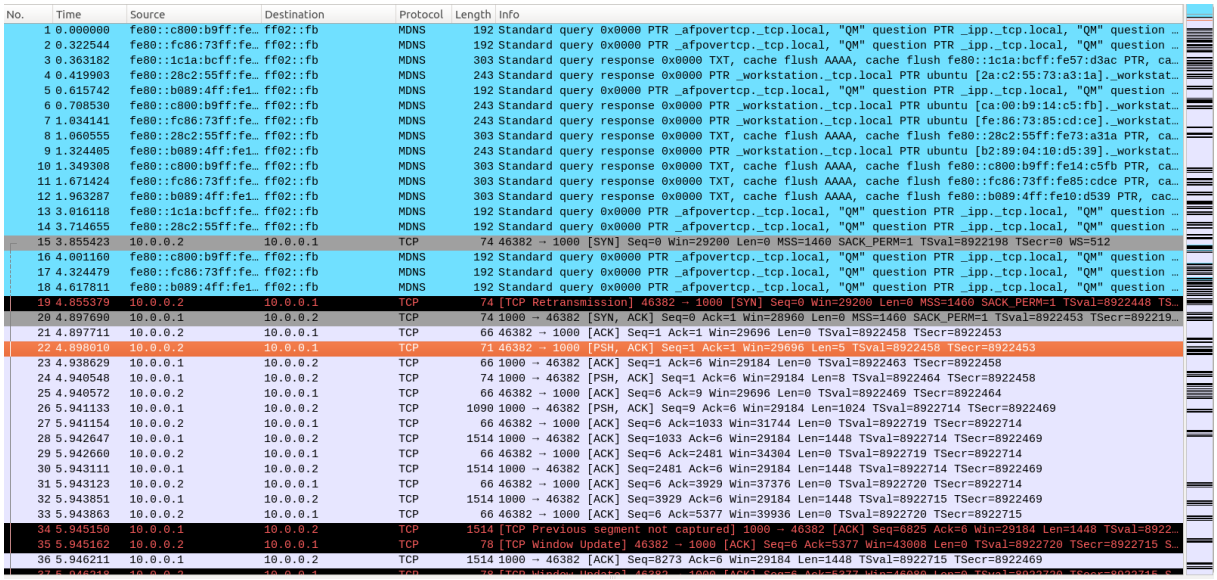


Figure 23: Part of the packets captured when transmit file with size of 10MB

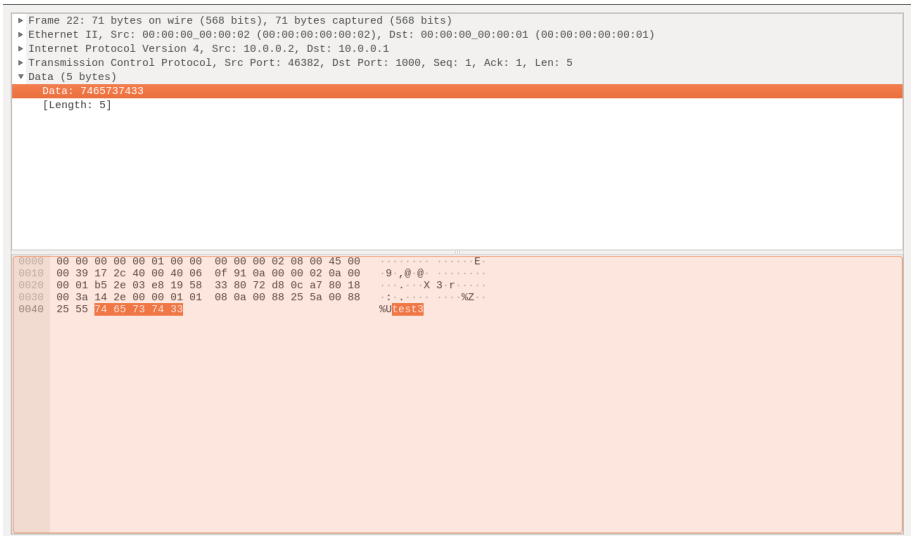


Figure 24: Detailed information about packet No.22

Figure 25: Detailed information about packet No.24

And `packet No.24` contains the file length, which is from $h1$ to $h2$. The file length is $0x989680 = 10000000$ bytes $= 10$MB, which is correct.

And packets with size of more than 1000 bytes contain the data, which is from $h1$ to $h2$. By using `diff`, we can say that the transmission is correct.

3 Part III

3.1 Implement a simple SR

3.1.1 Implementation

The code of `sr.c` and `makeFile` can be seen in **Appendix**.

3.1.2 Test and Results

There're five conditions, in each condition, I use the given file `file3.jpg` as the transferred file, whose size is 4.4MB. I run the sender on $h1$ (10.0.0.1) and run the receiver on $h2$ (10.0.0.2). The `mitm` program is running on $h3$ (10.0.0.3).

A. No reordering, no loss, no error.

By compared the file in receiver side and sender side with `diff`, the result is correct. The first few lines of log on sender is as following:

```
SYN 234756642 0 0
ACK 234756642 0 0
DATA 0 1456 3220623233
DATA 1 1456 2244073028
DATA 2 1456 1460592462
DATA 3 1456 3415882852
DATA 4 1456 3456527178
DATA 5 1456 1573440664
DATA 6 1456 3472368927
DATA 7 1456 3178870208
DATA 8 1456 279932998
DATA 9 1456 3426093530
ACK 1 0 0
```

Then, the window won't slide when `DATA` packets is sent, and the window will slide one packet when `ACK 1 0 0` comes.

The first few lines of log on receiver is as following:

```
SYN 234756642 0 0
ACK 234756642 0 0
DATA 0 1456 3220623233
ACK 1 0 0
DATA 1 1456 2244073028
ACK 2 0 0
DATA 2 1456 1460592462
ACK 3 0 0
DATA 3 1456 3415882852
```

Then, the window won't slide when `ACK` packets is sent. And in this case, since each `Data` packets comes with desired seq number, the window will slide for one packet for each `DATA` packets.

B. 10 % loss, no reordering, no error

By compared the file in receiver side and sender side with `diff`, the result is correct. As for the sender side, the evidence of retransmission can be seen in the following lines:

```
DATA 47 1456 2926046941
ACK 38 0 0
ACK 38 0 0
ACK 38 0 0
ACK 38 0 0
ACK 38 0 0
ACK 38 0 0
ACK 38 0 0
DATA 38 1456 2564180116
DATA 39 1456 214759100
DATA 40 1456 1477671586
DATA 41 1456 4058279732
DATA 42 1456 3378489669
DATA 43 1456 10853198
DATA 44 1456 1483690205
DATA 45 1456 2820064048
DATA 46 1456 720727197
DATA 47 1456 2926046941
ACK 42 0 0
```

It can be seen that at least `DATA 38` is lost. And after retransmission, the packets is ACKed with next seq number 42.

As for the receiver side, the evidence of packet loss can be seen in the following lines:

```
DATA 37 1456 1168804078
ACK 38 0 0
DATA 39 1456 214759100
ACK 38 0 0
DATA 40 1456 1477671586
ACK 38 0 0
DATA 41 1456 4058279732
ACK 38 0 0
DATA 43 1456 10853198
ACK 38 0 0
DATA 44 1456 1483690205
ACK 38 0 0
DATA 45 1456 2820064048
ACK 38 0 0
DATA 46 1456 720727197
```

```
ACK 38 0 0
DATA 47 1456 2926046941
ACK 38 0 0
DATA 38 1456 2564180116
ACK 42 0 0
```

Then, it can be seen that `DATA 38` is the gap.

Thus, `DATA 38` is lost.

C. 10 % error, no reordering, no loss

By compared the file in receiver side and sender side with `diff`, the result is correct. As for the sender side, the evidence of retransmission can be seen in the following lines:

```
DATA 10 1456 2805442653
ACK 2 0 0
DATA 11 1456 443311922
ACK 3 0 0
DATA 12 1456 4027845571
ACK 4 0 0
DATA 13 1456 1330974691
ACK 5 0 0
DATA 14 1456 2109819272
ACK 6 0 0
DATA 15 1456 965284191
ACK 7 0 0
DATA 16 1456 3311776188
ACK 8 0 0
DATA 17 1456 1539359076
ACK 9 0 0
DATA 18 1456 2360034080
ACK 10 0 0
DATA 19 1456 725513889
ACK 10 0 0
ACK 10 0 0
ACK 10 0 0
ACK 10 0 0
ACK 10 0 0
ACK 10 0 0
ACK 10 0 0
ACK 10 0 0
```

```
ACK 10 0 0
DATA 10 1456 2805442653
DATA 11 1456 443311922
DATA 12 1456 4027845571
DATA 13 1456 1330974691
DATA 14 1456 2109819272
DATA 15 1456 965284191
DATA 16 1456 3311776188
DATA 17 1456 1539359076
DATA 18 1456 2360034080
DATA 19 1456 725513889
ACK 20 0 0
```

It can be seen that at least `DATA 10` is corrupted since it has been transmitted but not ACKed. And after retransmission, the packets is ACKed with next seq number 20.

As for the receiver side, the evidence of packet loss can be seen in the following lines:

```
ACK 10 0 0
DATA 10 1456 2805442653
DATA 11 1456 443311922
ACK 10 0 0
```

Then, it can be seen that `DATA 10` is received but not ACKed due to corruption.

Thus, `DATA 10` is corrupted

D. Reordering, no error, no loss

By compared the file in receiver side and sender side with `diff`, the result is correct. The first few lines for log on sender is:

```
SYN 1194581707 0 0
ACK 1194581707 0 0
DATA 0 1456 3220623233
DATA 1 1456 2244073028
DATA 2 1456 1460592462
DATA 3 1456 3415882852
DATA 4 1456 3456527178
DATA 5 1456 1573440664
DATA 6 1456 3472368927
DATA 7 1456 3178870208
DATA 8 1456 279932998
DATA 9 1456 3426093530
ACK 0 0 0
```

```
ACK 0 0 0
ACK 0 0 0
ACK 0 0 0
ACK 0 0 0
ACK 0 0 0
ACK 0 0 0
ACK 8 0 0
DATA 10 1456 2805442653
ACK 10 0 0
```

Since `ACK 10 0 0` is received, we can say that the first 10 packets are delivered. And clearly, shuffle happens and make the first data packet to receiver is not `DATA 0`.

The first few lines for log on receiver is:

```
SYN 1194581707 0 0
ACK 1194581707 0 0
DATA 4 1456 3456527178
ACK 0 0 0
DATA 3 1456 3415882852
ACK 0 0 0
DATA 5 1456 1573440664
ACK 0 0 0
DATA 2 1456 1460592462
ACK 0 0 0
DATA 6 1456 3472368927
ACK 0 0 0
DATA 1 1456 2244073028
ACK 0 0 0
DATA 7 1456 3178870208
ACK 0 0 0
DATA 0 1456 3220623233
ACK 8 0 0
DATA 8 1456 279932998
ACK 9 0 0
DATA 9 1456 3426093530
ACK 10 0 0
```

As we can see, the order of data is completely re-ordered. So, how does receiver and sender make it correct. For receiver, it has a buffer. And the order within a buffer is correct, which ensures the correctness. As for a sender, A later ACK seq number may be shuffled to a recent timing, but since the

ACK means the packets before has been received, so it won't brought trouble.

E. Reordering, 5% loss, 5% error

By compared the file in receiver side and sender side with `diff`, the result is correct.

3.2 Make sender more efficient – leverage duplicate ACK on sender side

3.2.1 Implementation

The code of `sr.c` and `makeFile` can be seen in **Appendix**.

3.2.2 Tests and Results

I use the given file `file3.jpg` as the transferred file, whose size is 4.4MB. I run the sender on $h1$ (10.0.0.1) and run the receiver on $h2$ (10.0.0.2). The `mitm` program is running on $h3$ (10.0.0.3).

A. 10% loss, no reordering, no error.

The following lines inidcates that receive three duplicate ACKs and resend `DATA 1`.

```
ACK 1 0 0
DATA 10 1456 2805442653
ACK 1 0 0
ACK 1 0 0
DATA 1 1456 2244073028
```

B. Efficiency.

The transferring time of sender in part 3.1 is 1min51.650s and the transferring time of sender in part 3.2 is 1min35.900s Duplicate ACK make the sender more efficient.

3.3 Make sender more efficient – send NACK on receiver side

3.3.1 Implementation

The code of `sr.c` and `makeFile` can be seen in **Appendix**.

3.3.2 Tests and Results

I use the given file `file3.jpg` as the transferred file, whose size is 4.4MB. I run the sender on $h1$ (10.0.0.1) and run the receiver on $h2$ (10.0.0.2). The `mitm` program is running on $h3$ (10.0.0.3).

A. 10% loss, no reordering, no error.

The following lines inidcates that receive three duplicate ACKs and resend `DATA 1`.

```
ACK 1 0 0
DATA 10 1456 2805442653
ACK 1 0 0
ACK 1 0 0
```

```
DATA 1 1456 2244073028
```

B. Efficiency.

The transferring time of sender in part 3.1 is 1min51.650s, and the transferring time of sender in part 3.3 is 1min30.650s. NACK make the sender more efficient.

3.4 Throughput, delay and window size

Throughout under different condition can be seen in Table 1 It can be seen that the throught has

	delay = 0.01ms	delay = 10ms	delay = 50ms	delay = 100ms
window size = 10	2.722s	7.157s	31.871s	1min1.892s
window size = 50	0.758s	1.695s	6.934s	

Table 1: Throughout at different conditions

positive linear relationship with the inverse of the window size. And when the delay is not very small. the throughout seems has positive linear relationship with delay. When the delay is too small, other operation like file IO will influence the throughout greatly.

4 Appendix

4.1 Part II

4.1.1 makeFile

```
# use c++ 11 standard here (c++17 would be fine?)
CC=g++ -g -Wall -std=c++11

# List of source files for iPerfer
FS_SOURCES=ftrans.cc

# Generate the names of the iPerfer's object files
FS_OBJS=${FS_SOURCES:.cc=.o}

all: ftrans

ftrans: ${FS_OBJS}
  ${CC} -o $@ $^

# Generic rules for compiling a source file to an object file
%.o: %.cpp
  ${CC} -c $<
%.o: %.cc
  ${CC} -c $<

clean:
  rm -f ${FS_OBJS} ftrans
```

4.1.2 ftrans.c

```
# use c++ 11 standard here (c++17 would be fine?)
CC=g++ -g -Wall -std=c++11

# List of source files for iPerfer
FS_SOURCES=ftrans.cc
```

```
6
7   # Generate the names of the iPerfer's object files
8   FS_OBJS=${FS_SOURCES:.cc=.o}
9
10  all: ftrans
11
12  ftrans: ${FS_OBJS}
13    ${CC} -o $@ $^
14
15  # Generic rules for compiling a source file to an object file
16  %.o: %.cpp
17    ${CC} -c $<
18  %.o: %.cc
19    ${CC} -c $<
20
21  clean:
22    rm -f ${FS_OBJS} ftrans
```

4.2 Part III

4.2.1 makeFile

```
1   CC = gcc
2
3   CFLAGS  = -g -Wall -std=c99 -D_GNU_SOURCE -pthread
4
5   # the build target executable:
6   SR = sr
7   MITM = mitm
8
9   all: $(SR) $(MITM)
10
11  # assume the C code for TARGET is TARGET.c
12  $(SR): $(SR).c
13    $(CC) $(CFLAGS) -o $(SR) $(SR).c
14
15  $(MITM): $(MITM).c
16    $(CC) $(CFLAGS) -o $(MITM) $(MITM).c
17
18  clean:
19    $(RM) $(SR) $(MITM)
```

4.2.2 sr.c in 3.1

```
1   #include "./SRHeader.h"
2   #include "./crc32.h"
3   #include <arpa/inet.h>
4   #include <netdb.h>
5   #include <stdint.h>
6   #include <stdio.h>
7   #include <stdlib.h>
8   #include <string.h>
9   #include <sys/socket.h>
10  #include <sys/stat.h> // struct stat
11  #include <sys/time.h>
12  #include <time.h>
13  #include <unistd.h>
14  // The size of packet would is len(header) + len(payload) = 4*4 + 1456 = 1472
15  #define MAX_MESSAGE_SIZE 1472
16
17  int logInfo(struct SRHeader header, FILE *fp) {
18    if (header.flag == SYN) {
```

```
      fprintf(fp, "%s %u %u %u\n", "SYN", header.seq, header.len, header.crc);
      printf("%s %u %u %u\n", "SYN", header.seq, header.len, header.crc);
    } else if (header.flag == FIN) {
      fprintf(fp, "%s %u %u %u\n", "FIN", header.seq, header.len, header.crc);
      printf("%s %u %u %u\n", "FIN", header.seq, header.len, header.crc);
    } else if (header.flag == DATA) {
      fprintf(fp, "%s %u %u %u\n", "DATA", header.seq, header.len, header.crc);
      printf("%s %u %u %u\n", "DATA", header.seq, header.len, header.crc);
    } else if (header.flag == ACK) {
      fprintf(fp, "%s %u %u %u\n", "ACK", header.seq, header.len, header.crc);
      printf("%s %u %u %u\n", "ACK", header.seq, header.len, header.crc);
    } else if (header.flag == NACK) {
      fprintf(fp, "%s %u %u %u\n", "NACK", header.seq, header.len, header.crc);
      printf("%s %u %u %u\n", "NACK", header.seq, header.len, header.crc);
    }
    return 0;
}

int runSender(char *recvIP, int recvPort, int senderPort,
              unsigned int windowSize, char *fileToSend, char *logFile) {
    FILE *logfp = fopen(logFile, "w+");
    // create UDP socket.
    int sockfd = socket(AF_INET, SOCK_DGRAM, IPPROTO_UDP); // use IPPROTO_UDP
    if (sockfd < 0) {
      perror("socket creation failed");
      exit(1);
    }

    // below is the same as TCP socket.
    struct sockaddr_in myAddr;
    memset(&myAddr, 0, sizeof(myAddr));
    myAddr.sin_family = AF_INET;
    myAddr.sin_addr.s_addr = INADDR_ANY;
    // bind to a specific port
    myAddr.sin_port = htons(senderPort);
    bind(sockfd, (struct sockaddr *)&myAddr, sizeof(myAddr));

    struct sockaddr_in recvAddr;
    struct hostent *host = gethostbyname(recvIP);
    memcpy(&(recvAddr.sin_addr), host->h_addr, host->h_length);
    recvAddr.sin_family = AF_INET;
    recvAddr.sin_port = htons(recvPort);

    // 1. The num of packets to be sent
    // 1.1 Read file length
    struct stat statBuffer;
    size_t fileLen = 0;
    size_t numSend = 0;
    size_t divFlag = 0;
    // Read the file length
    if (stat(fileToSend, &statBuffer) < 0) {
      // Handling reding file error
      perror("Error: Can not read file\n");
      exit(-1);
    }
    fileLen = statBuffer.st_size;
    // 1.2 Calculate num of packets to be sent
    divFlag = fileLen % MAX_PAYLOAD_SIZE;
    numSend = divFlag > 0 ? (1 + (fileLen / MAX_PAYLOAD_SIZE))
                          : fileLen / MAX_PAYLOAD_SIZE;
    printf("File size is %lu, need %lu packets to send\n", fileLen, numSend);

    // 2. send
```

```
82    unsigned int randSeq = 0;
83    char packet[MAX_MESSAGE_SIZE];
84    char msg[MAX_MESSAGE_SIZE];
85    struct SRHeader header;
86    memset(packet, 0, sizeof(packet));
87    // 2.1 send SYN, has random seq number
88    srand((unsigned)time(NULL));
89    randSeq = rand();
90    header.flag = SYN;
91    header.seq = randSeq;
92    header.len = 0;
93    header.crc = crc32(NULL, 0); // without payload;
94    memcpy(packet, (char *)&header, sizeof(header));
95    sendto(sockfd, (void *)packet, sizeof(header), 0,
96           (const struct sockaddr *)&recvAddr, sizeof(struct sockaddr_in));
97    logInfo(header, logfp);
98    printf("Send SYN to set up the connection\n");
99    // 2.2 wait for ACK and SYN packet
100   socklen_t sLen = sizeof(struct sockaddr_in);
101   struct sockaddr_in siOther;
102   struct SRHeader recvHeader;
103   struct timeval start, stop;
104   gettimeofday(&start, NULL);
105   gettimeofday(&stop, NULL);
106
107   // receive ack
108   while (1) {
109     if (recvfrom(sockfd, msg, MAX_MESSAGE_SIZE, MSG_DONTWAIT,
110                  (struct sockaddr *)&siOther, &sLen) > 0) {
111       memcpy(&recvHeader, msg, sizeof(recvHeader));
112       logInfo(recvHeader, logfp);
113       if (recvHeader.flag == ACK) {
114         printf("Connection set up \n");
115         break;
116       }
117     } else {
118       // resend SYN
119       sendto(sockfd, (void *)packet, sizeof(header), 0,
120              (const struct sockaddr *)&recvAddr, sizeof(struct sockaddr_in));
121       logInfo(header, logfp);
122     }
123     // time out for ACK
124     gettimeofday(&stop, NULL);
125     if ((double)(stop.tv_sec - start.tv_sec) * 1000 +
126             (double)(stop.tv_usec - start.tv_usec) / 1000 >
127         5000) {
128       perror("Error: do not receive ACK for SYN within 5 s. Give up\n");
129       fclose(logfp);
130       exit(4);
131     }
132   }
133   // 2.3 Send data
134   char *buf = malloc(windowSize * MAX_MESSAGE_SIZE);
135   memset(buf, 0, windowSize * MAX_MESSAGE_SIZE);
136   FILE *fp = fopen(fileToSend, "r");
137   // open file
138   if (!fp) {
139     // Handling file open error
140     perror("Error: Can not read file\n");
141     free(buf);
142     fclose(logfp);
143     fclose(fp);
144     exit(-1);
```

```
145     }
146
147     unsigned int currentSeq = 0; // send 0 for initial
148     unsigned int bufNum = 0;
149     int endFlag = 1;
150     size_t payloadLen = MAX_PAYLOAD_SIZE;
151     gettimeofday(&start, NULL);
152     while (endFlag) {
153       // check if timeout
154       gettimeofday(&stop, NULL);
155       // if timeout, resend all buffer
156       if ((double)(stop.tv_sec - start.tv_sec) * 1000 +
157             (double)(stop.tv_usec - start.tv_usec) / 1000 >
158           400) {
159         printf(
160             "No ack within 400ms, resend all packets in buffer. Start from %u\n",
161             currentSeq);
162         // resend
163         for (unsigned int ii = 0; ii < bufNum; ii++) {
164           payloadLen = ((currentSeq + ii == numSend - 1) && (divFlag > 0))
165                            ? divFlag
166                            : MAX_PAYLOAD_SIZE;
167           sendto(sockfd, (void *)(buf + ii * sizeof(packet)),
168                  payloadLen + sizeof(header), 0,
169                  (const struct sockaddr *)&recvAddr, sizeof(struct sockaddr_in));
170           struct SRHeader tempHeader;
171           memcpy(&tempHeader, buf + ii * sizeof(packet), sizeof(header));
172           logInfo(tempHeader, logfp);
173         }
174         // reset time
175         gettimeofday(&start, NULL);
176       }
177       // send and buffer
178
179       if ((bufNum < windowSize) && ((bufNum + currentSeq) < numSend)) {
180         payloadLen = ((currentSeq + bufNum == numSend - 1) && (divFlag > 0))
181                          ? divFlag
182                          : MAX_PAYLOAD_SIZE;
183         // form a packet
184         fread(packet + sizeof(header), payloadLen, 1, fp);
185         header.flag = DATA;
186         header.seq = currentSeq + bufNum;
187         header.len = payloadLen;
188         header.crc = crc32(packet + sizeof(header), payloadLen);
189         memcpy(packet, (char *)(&header), sizeof(header));
190         // send
191         sendto(sockfd, (void *)packet, payloadLen + sizeof(header), 0,
192                (const struct sockaddr *)&recvAddr, sizeof(struct sockaddr_in));
193         logInfo(header, logfp);
194         // buffer the packet
195         memcpy(buf + bufNum * MAX_MESSAGE_SIZE, packet, sizeof(packet));
196         bufNum++;
197       }
198       // receive ACK
199       if (recvfrom(sockfd, msg, MAX_MESSAGE_SIZE, MSG_DONTWAIT,
200                    (struct sockaddr *)&siOther, &sLen) != -1) {
201         memcpy(&recvHeader, msg, sizeof(recvHeader));
202         logInfo(recvHeader, logfp);
203         // Advance window
204         if (recvHeader.flag == ACK) {
205           if (recvHeader.seq > currentSeq) {
206             // move buffer
207             if (recvHeader.seq > currentSeq + bufNum) {
```

```
208             // clear buf
209             memset(buf, 0, windowSize * MAX_MESSAGE_SIZE);
210             bufNum = 0;
211           } else {
212             char *temp = malloc(windowSize * MAX_PAYLOAD_SIZE);
213             memcpy(temp, buf + (recvHeader.seq - currentSeq) * MAX_MESSAGE_SIZE,
214                    (bufNum + currentSeq - recvHeader.seq) * MAX_MESSAGE_SIZE);
215             memset(buf, 0, windowSize * MAX_MESSAGE_SIZE);
216             memcpy(buf, temp, windowSize * MAX_MESSAGE_SIZE);
217             free(temp);
218             bufNum = bufNum + currentSeq - recvHeader.seq;
219           }
220           // advance
221           currentSeq = recvHeader.seq;
222           if (currentSeq > (numSend - 1))
223             endFlag = 0;
224           // reset time
225           gettimeofday(&start, NULL);
226         }
227       }
228     }
229   }
230   // Send FIN
231   header.flag = FIN;
232   header.seq = randSeq;
233   header.len = 0;
234   header.crc = crc32(NULL, 0); // without payload;
235   memcpy(packet, (char *)&header, sizeof(header));
236   sendto(sockfd, (void *)packet, sizeof(header), 0,
237          (const struct sockaddr *)&recvAddr, sizeof(struct sockaddr_in));
238   logInfo(header, logfp);
239   printf("Tranmission finished: Send FIN");
240   // receive ack for FIN
241   while (1) {
242     if (recvfrom(sockfd, msg, MAX_MESSAGE_SIZE, MSG_DONTWAIT,
243                  (struct sockaddr *)&siOther, &sLen) != -1) {
244       memcpy(&recvHeader, msg, sizeof(recvHeader));
245       logInfo(recvHeader, logfp);
246       if (recvHeader.flag == ACK) {
247         printf("Receive ACK for FIN, done\n");
248         break;
249       }
250     } else {
251       // resend FIN
252       sendto(sockfd, (void *)packet, sizeof(header), 0,
253              (const struct sockaddr *)&recvAddr, sizeof(struct sockaddr_in));
254       logInfo(header, logfp);
255     }
256     // time out for ACK
257     gettimeofday(&stop, NULL);
258     if ((double)(stop.tv_sec - start.tv_sec) * 1000 +
259           (double)(stop.tv_usec - start.tv_usec) / 1000 >
260         5000) {
261       perror("Error: do not receive ACK for FIN within 5s. Give up\n");
262       fclose(fp);
263       fclose(logfp);
264       free(buf);
265       exit(4);
266     }
267   }
268   fclose(fp);
269   fclose(logfp);
270   free(buf);
```

```
271     return 0;
272   }
273
274   int runReceiver(int port, unsigned int windowSize, char *recvDir,
275                   char *logFile) {
276     FILE *logfp = fopen(logFile, "w+");
277     // Set up connection
278     struct sockaddr_in siMe;
279     int sockfd = socket(AF_INET, SOCK_DGRAM, IPPROTO_UDP);
280     if (sockfd == -1) {
281       fputs("socket creation failed!", stderr);
282       exit(2); // error, exit with 2
283     }
284     memset((char *)&siMe, 0, sizeof(struct sockaddr_in));
285     siMe.sin_family = AF_INET;
286     siMe.sin_port = htons(port);
287     siMe.sin_addr.s_addr = htonl(INADDR_ANY);
288     bind(sockfd, (struct sockaddr *)&siMe, sizeof(struct sockaddr_in));
289
290     socklen_t sLen = sizeof(struct sockaddr_in);
291     struct sockaddr_in siOther;
292     struct timeval start, stop;
293     int connectCnt = 0;
294     while (1) {
295       // 1. Ready for receiving SYN
296       char msg[MAX_MESSAGE_SIZE];
297       memset(msg, 0, MAX_MESSAGE_SIZE);
298       char ACKMsg[MAX_MESSAGE_SIZE];
299       memset(ACKMsg, 0, sizeof(ACKMsg));
300       unsigned SYNFINSeq = 0;
301       struct SRHeader recvHeader;
302       struct SRHeader ACKHeader;
303
304       gettimeofday(&start, NULL);
305       gettimeofday(&stop, NULL);
306       while (1) {
307         if (recvfrom(sockfd, msg, MAX_MESSAGE_SIZE, MSG_DONTWAIT,
308                      (struct sockaddr *)&siOther, &sLen) > 0) {
309           memcpy(&recvHeader, msg, sizeof(recvHeader));
310           logInfo(recvHeader, logfp);
311           if (recvHeader.flag == SYN) {
312             SYNFINSeq = recvHeader.seq;
313             // ACK for SYN
314             ACKHeader.flag = ACK;
315             ACKHeader.seq = SYNFINSeq;
316             ACKHeader.len = 0;
317             ACKHeader.crc = crc32(NULL, 0);
318             memcpy(ACKMsg, (char *)&ACKHeader, sizeof(ACKHeader));
319             sendto(sockfd, (void *)ACKMsg, sizeof(ACKHeader), 0,
320                    (const struct sockaddr *)&siOther, sizeof(struct sockaddr_in));
321             logInfo(ACKHeader, logfp);
322             break;
323           }
324         }
325         gettimeofday(&stop, NULL);
326         if ((double)(stop.tv_sec - start.tv_sec) * 1000 +
327                 (double)(stop.tv_usec - start.tv_usec) / 1000 >
328             5000) {
329           printf("I do not receive SYN within 5 s. Give up\n");
330           return 0;
331         }
332       }
333       // 2. Recive data
```

```
334
335       // 2.1 open file
336       // create file name
337       char fileName[30];
338       char name[13];
339       sprintf(name, "file_%d.txt", connectCnt);
340       strcpy(fileName, recvDir);
341       strcat(fileName, name);
342       FILE *fp = fopen(fileName, "w");
343       unsigned int currentACKSeq = 0;
344       char *buf = malloc(windowSize * (MAX_PAYLOAD_SIZE + 8)); // valid. len
345       memset(buf, 0, windowSize * (MAX_PAYLOAD_SIZE + 8));
346       int FINflag = 1; // 0 for receive FIN flag
347
348       // 2.2 receive data
349       printf("Start to receive file\n");
350       while (FINflag) {
351         // receive data
352         if (recvfrom(sockfd, msg, MAX_MESSAGE_SIZE, MSG_DONTWAIT,
353                      (struct sockaddr *)&siOther, &sLen) != -1) {
354           memcpy(&recvHeader, msg, sizeof(recvHeader));
355           logInfo(recvHeader, logfp);
356           // If it's data
357           if (recvHeader.flag == DATA) {
358             if (recvHeader.crc ==
359                 (crc32(msg + sizeof(recvHeader), recvHeader.len))) {
360               // check the condition for window operation
361               if (recvHeader.seq >= currentACKSeq + windowSize) {
362                 continue;
363               } else if (recvHeader.seq < currentACKSeq) {
364                 // send ACK currentACKSeq
365                 ACKHeader.flag = ACK;
366                 ACKHeader.seq = currentACKSeq;
367                 ACKHeader.len = 0;
368                 ACKHeader.crc = crc32(NULL, 0);
369                 memcpy(ACKMsg, (char *)&ACKHeader, sizeof(ACKHeader));
370                 sendto(sockfd, (void *)ACKMsg, sizeof(ACKHeader), 0,
371                        (const struct sockaddr *)&siOther,
372                        sizeof(struct sockaddr_in));
373                 logInfo(ACKHeader, logfp);
374               } else if ((recvHeader.seq > currentACKSeq) &&
375                          (recvHeader.seq < currentACKSeq + windowSize)) {
376                 // buffer it
377                 unsigned int temp = 1;
378                 memcpy(buf + (recvHeader.seq - currentACKSeq) *
379                                  (MAX_PAYLOAD_SIZE + 8),
380                        (char *)(&temp), sizeof(temp));
381                 memcpy(buf +
382                            (recvHeader.seq - currentACKSeq) *
383                                (MAX_PAYLOAD_SIZE + 8) +
384                            4,
385                        (char *)(&recvHeader.len), sizeof(temp));
386                 memcpy(buf +
387                            +(recvHeader.seq - currentACKSeq) *
388                                (MAX_PAYLOAD_SIZE + 8) +
389                            8,
390                        msg + sizeof(recvHeader), recvHeader.len);
391                 // send current ACK
392                 ACKHeader.flag = ACK;
393                 ACKHeader.seq = currentACKSeq;
394                 ACKHeader.len = 0;
395                 ACKHeader.crc = crc32(NULL, 0);
396                 memcpy(ACKMsg, (char *)&ACKHeader, sizeof(ACKHeader));
```

```
397                 sendto(sockfd, (void *)ACKMsg, sizeof(ACKHeader), 0,
398                        (const struct sockaddr *)&siOther,
399                        sizeof(struct sockaddr_in));
400                 logInfo(ACKHeader, logfp);
401               } else if (recvHeader.seq == currentACKSeq) {
402                 // calculate the highest avalible packet
403                 unsigned int numWrite = 1;
404                 for (numWrite = 1; numWrite < windowSize; numWrite++) {
405                   unsigned int temp;
406                   memcpy(&temp, buf + numWrite * (8 + MAX_PAYLOAD_SIZE),
407                          sizeof(temp));
408                   if (temp == 0)
409                     break;
410                 }
411                 fwrite(msg + sizeof(recvHeader), 1, recvHeader.len, fp);
412                 for (unsigned int ii = 1; ii < numWrite; ii++) {
413                   unsigned tempLen;
414                   memcpy(&tempLen, buf + ii * (8 + MAX_PAYLOAD_SIZE) + 4,
415                          sizeof(tempLen));
416                   fwrite(buf + ii * (8 + MAX_PAYLOAD_SIZE) + 8, 1, tempLen, fp);
417                 }
418                 // clear buff
419                 memset(buf, 0, windowSize * (MAX_PAYLOAD_SIZE + 4));
420                 // ACK
421                 currentACKSeq = currentACKSeq + numWrite;
422                 // send current ACK
423                 ACKHeader.flag = ACK;
424                 ACKHeader.seq = currentACKSeq;
425                 ACKHeader.len = 0;
426                 ACKHeader.crc = crc32(NULL, 0);
427                 memcpy(ACKMsg, (char *)&ACKHeader, sizeof(ACKHeader));
428                 sendto(sockfd, (void *)ACKMsg, sizeof(ACKHeader), 0,
429                        (const struct sockaddr *)&siOther,
430                        sizeof(struct sockaddr_in));
431                 logInfo(ACKHeader, logfp);
432               }
433             }
434           } else if (recvHeader.flag == FIN) {
435             FINflag = 0;
436             // send ACK
437             ACKHeader.flag = ACK;
438             ACKHeader.seq = SYNFINSeq;
439             ACKHeader.len = 0;
440             ACKHeader.crc = crc32(NULL, 0);
441             memcpy(ACKMsg, (char *)&ACKHeader, sizeof(ACKHeader));
442             sendto(sockfd, (void *)ACKMsg, sizeof(ACKHeader), 0,
443                    (const struct sockaddr *)&siOther, sizeof(struct sockaddr_in));
444             logInfo(ACKHeader, logfp);
445             // end
446             fclose(fp);
447             free(buf);
448             connectCnt++;
449             fclose(logfp);
450           }
451         }
452       }
453     }
454     fclose(logfp);
455     return 0;
456   }
457
458   int main(int argc, char **argv) {
459     // Sender mode
```

```
460   if (strcmp(argv[1], "-s") == 0) {
461     // Error for number of arguments
462     if (argc != 8) {
463       perror(
464           "./sr -s <receiver's IP> <receiver's port> <sender's port> <window "
465           "size> <file to send> <log file>");
466       return 1; // error: exit with 1
467     }
468     char *recvIP = argv[2];
469     int recvPort = atoi(argv[3]);
470     int senderPort = atoi(argv[4]);
471     unsigned int windowSize = atoi(argv[5]);
472     char *fileToSend = argv[6];
473     char *logFile = argv[7];
474     runSender(recvIP, recvPort, senderPort, windowSize, fileToSend, logFile);
475     return 0;
476   }
477
478   // Receiver mode
479   else if (strcmp(argv[1], "-r") == 0) {
480     // Error for number of arguments
481     if (argc != 6) {
482       perror("./sr -r <port> <window size> <recv dir> <log file>");
483       return 1; // error: exit with 1
484     }
485     // int runReceiver(int port, unsigned int windowSize, char *recvDir,
486     //                 char *logFile)
487     int port = atoi(argv[2]);
488     unsigned int windowSize = atoi(argv[3]);
489     char *recvDir = argv[4];
490     char *logFile = argv[5];
491     runReceiver(port, windowSize, recvDir, logFile);
492     return 0;
493   } else {
494     perror("./sr -s <receiver's IP> <receiver's port> <sender's port> <window "
495            "size> <file to send> <log file>\n ./sr -r <port> <window size> "
496            "<recv dir> <log file>\n");
497     return 1;
498   }
499   return 0;
500 }
```

4.2.3 sr.c in 3.2

```
1  #include "./SRHeader.h"
2  #include "./crc32.h"
3  #include <arpa/inet.h>
4  #include <netdb.h>
5  #include <stdint.h>
6  #include <stdio.h>
7  #include <stdlib.h>
8  #include <string.h>
9  #include <sys/socket.h>
10 #include <sys/stat.h> // struct stat
11 #include <sys/time.h>
12 #include <time.h>
13 #include <unistd.h>
14 // The size of packet would is len(header) + len(payload) = 4*4 + 1456 = 1472
15 #define MAX_MESSAGE_SIZE 1472
16
17 int logInfo(struct SRHeader header, FILE *fp) {
18   if (header.flag == SYN) {
19     fprintf(fp, "%s %u %u %u\n", "SYN", header.seq, header.len, header.crc);
```

```
20      printf("%s %u %u %u\n", "SYN", header.seq, header.len, header.crc);
21    } else if (header.flag == FIN) {
22      fprintf(fp, "%s %u %u %u\n", "FIN", header.seq, header.len, header.crc);
23      printf("%s %u %u %u\n", "FIN", header.seq, header.len, header.crc);
24    } else if (header.flag == DATA) {
25      fprintf(fp, "%s %u %u %u\n", "DATA", header.seq, header.len, header.crc);
26      printf("%s %u %u %u\n", "DATA", header.seq, header.len, header.crc);
27    } else if (header.flag == ACK) {
28      fprintf(fp, "%s %u %u %u\n", "ACK", header.seq, header.len, header.crc);
29      printf("%s %u %u %u\n", "ACK", header.seq, header.len, header.crc);
30    } else if (header.flag == NACK) {
31      fprintf(fp, "%s %u %u %u\n", "NACK", header.seq, header.len, header.crc);
32      printf("%s %u %u %u\n", "NACK", header.seq, header.len, header.crc);
33    }
34    return 0;
35  }
36
37  int runSender(char *recvIP, int recvPort, int senderPort,
38                unsigned int windowSize, char *fileToSend, char *logFile) {
39    FILE *logfp = fopen(logFile, "w+");
40    // create UDP socket.
41    int sockfd = socket(AF_INET, SOCK_DGRAM, IPPROTO_UDP); // use IPPROTO_UDP
42    if (sockfd < 0) {
43      perror("socket creation failed");
44      exit(1);
45    }
46
47    // below is the same as TCP socket.
48    struct sockaddr_in myAddr;
49    memset(&myAddr, 0, sizeof(myAddr));
50    myAddr.sin_family = AF_INET;
51    myAddr.sin_addr.s_addr = INADDR_ANY;
52    // bind to a specific port
53    myAddr.sin_port = htons(senderPort);
54    bind(sockfd, (struct sockaddr *)&myAddr, sizeof(myAddr));
55
56    struct sockaddr_in recvAddr;
57    struct hostent *host = gethostbyname(recvIP);
58    memcpy(&(recvAddr.sin_addr), host->h_addr, host->h_length);
59    recvAddr.sin_family = AF_INET;
60    recvAddr.sin_port = htons(recvPort);
61
62    // 1. The num of packets to be sent
63    // 1.1 Read file length
64    struct stat statBuffer;
65    size_t fileLen = 0;
66    size_t numSend = 0;
67    size_t divFlag = 0;
68    // Read the file length
69    if (stat(fileToSend, &statBuffer) < 0) {
70      // Handling reding file error
71      perror("Error: Can not read file\n");
72      exit(-1);
73    }
74    fileLen = statBuffer.st_size;
75    // 1.2 Calculate num of packets to be sent
76    divFlag = fileLen % MAX_PAYLOAD_SIZE;
77    numSend = divFlag > 0 ? (1 + (fileLen / MAX_PAYLOAD_SIZE))
78                          : fileLen / MAX_PAYLOAD_SIZE;
79    printf("File size is %lu, need %lu packets to send\n", fileLen, numSend);
80
81    // 2. send
82    unsigned int randSeq = 0;
```

```
83    char packet[MAX_MESSAGE_SIZE];
84    char msg[MAX_MESSAGE_SIZE];
85    struct SRHeader header;
86    memset(packet, 0, sizeof(packet));
87    // 2.1 send SYN, has random seq number
88    srand((unsigned)time(NULL));
89    randSeq = rand();
90    header.flag = SYN;
91    header.seq = randSeq;
92    header.len = 0;
93    header.crc = crc32(NULL, 0); // without payload;
94    memcpy(packet, (char *)&header, sizeof(header));
95    sendto(sockfd, (void *)packet, sizeof(header), 0,
96           (const struct sockaddr *)&recvAddr, sizeof(struct sockaddr_in));
97    logInfo(header, logfp);
98    printf("Send SYN to set up the connection\n");
99    // 2.2 wait for ACK and SYN packet
100   socklen_t sLen = sizeof(struct sockaddr_in);
101   struct sockaddr_in siOther;
102   struct SRHeader recvHeader;
103   struct timeval start, stop;
104   gettimeofday(&start, NULL);
105   gettimeofday(&stop, NULL);
106
107   // receive ack
108   while (1) {
109     if (recvfrom(sockfd, msg, MAX_MESSAGE_SIZE, MSG_DONTWAIT,
110                  (struct sockaddr *)&siOther, &sLen) > 0) {
111       memcpy(&recvHeader, msg, sizeof(recvHeader));
112       logInfo(recvHeader, logfp);
113       if (recvHeader.flag == ACK) {
114         printf("Connection set up \n");
115         break;
116       }
117     } else {
118       // resend SYN
119       sendto(sockfd, (void *)packet, sizeof(header), 0,
120              (const struct sockaddr *)&recvAddr, sizeof(struct sockaddr_in));
121       logInfo(header, logfp);
122     }
123     // time out for ACK
124     gettimeofday(&stop, NULL);
125     if ((double)(stop.tv_sec - start.tv_sec) * 1000 +
126             (double)(stop.tv_usec - start.tv_usec) / 1000 >
127         5000) {
128       perror("Error: do not receive ACK for SYN within 5 s. Give up\n");
129       fclose(logfp);
130       exit(4);
131     }
132   }
133   // 2.3 Send data
134   char *buf = malloc(windowSize * MAX_MESSAGE_SIZE);
135   memset(buf, 0, windowSize * MAX_MESSAGE_SIZE);
136   FILE *fp = fopen(fileToSend, "r");
137   // open file
138   if (!fp) {
139     // Handling file open error
140     perror("Error: Can not read file\n");
141     free(buf);
142     fclose(logfp);
143     fclose(fp);
144     exit(-1);
145   }
```

```
146
147     unsigned int currentSeq = 0; // send 0 for initial
148     unsigned int bufNum = 0;
149     int endFlag = 1;
150     unsigned dupCnt = 0;
151     unsigned dupSeq = 0;
152     size_t payloadLen = MAX_PAYLOAD_SIZE;
153     gettimeofday(&start, NULL);
154     while (endFlag) {
155       // check if timeout
156       gettimeofday(&stop, NULL);
157       // if timeout, resend all buffer
158       if ((double)(stop.tv_sec - start.tv_sec) * 1000 +
159               (double)(stop.tv_usec - start.tv_usec) / 1000 >
160           400) {
161         printf(
162             "No ack within 400ms, resend all packets in buffer. Start from %u\n",
163             currentSeq);
164         // resend
165         for (unsigned int ii = 0; ii < bufNum; ii++) {
166           payloadLen = ((currentSeq + ii == numSend - 1) && (divFlag > 0))
167                            ? divFlag
168                            : MAX_PAYLOAD_SIZE;
169           sendto(sockfd, (void *)(buf + ii * sizeof(packet)),
170                  payloadLen + sizeof(header), 0,
171                  (const struct sockaddr *)&recvAddr, sizeof(struct sockaddr_in));
172           struct SRHeader tempHeader;
173           memcpy(&tempHeader, buf + ii * sizeof(packet), sizeof(header));
174           logInfo(tempHeader, logfp);
175         }
176         // reset time
177         gettimeofday(&start, NULL);
178       }
179       // send and buffer
180
181       if ((bufNum < windowSize) && ((bufNum + currentSeq) < numSend)) {
182         payloadLen = ((currentSeq + bufNum == numSend - 1) && (divFlag > 0))
183                          ? divFlag
184                          : MAX_PAYLOAD_SIZE;
185         // form a packet
186         fread(packet + sizeof(header), payloadLen, 1, fp);
187         header.flag = DATA;
188         header.seq = currentSeq + bufNum;
189         header.len = payloadLen;
190         header.crc = crc32(packet + sizeof(header), payloadLen);
191         memcpy(packet, (char *)(&header), sizeof(header));
192         // send
193         sendto(sockfd, (void *)packet, payloadLen + sizeof(header), 0,
194                (const struct sockaddr *)&recvAddr, sizeof(struct sockaddr_in));
195         logInfo(header, logfp);
196         // buffer the packet
197         memcpy(buf + bufNum * MAX_MESSAGE_SIZE, packet, sizeof(packet));
198         bufNum++;
199       }
200       // receive ACK
201       if (recvfrom(sockfd, msg, MAX_MESSAGE_SIZE, MSG_DONTWAIT,
202                    (struct sockaddr *)&siOther, &sLen) != -1) {
203         memcpy(&recvHeader, msg, sizeof(recvHeader));
204         logInfo(recvHeader, logfp);
205         // duplicate ACK
206         if ((dupCnt == 0) || (recvHeader.seq != dupSeq)) {
207           dupSeq = recvHeader.seq;
208           dupCnt = 1;
```

```
209         } else if (recvHeader.seq == dupSeq) {
210           dupCnt++;
211           if (dupCnt >= 3) {
212             // resend it immediately
213             payloadLen = ((dupSeq == numSend - 1) && (divFlag > 0))
214                          ? divFlag
215                          : MAX_PAYLOAD_SIZE;
216             sendto(sockfd, (void *)(buf + (dupSeq - currentSeq) * sizeof(packet)),
217                    payloadLen + sizeof(header), 0,
218                    (const struct sockaddr *)&recvAddr,
219                    sizeof(struct sockaddr_in));
220             struct SRHeader tempHeader;
221             memcpy(&tempHeader, buf + (dupSeq - currentSeq) * sizeof(packet),
222                    sizeof(header));
223             logInfo(tempHeader, logfp);
224             // reset Cnt
225             dupCnt = 0;
226           }
227         }
228         // Advance window
229         if (recvHeader.flag == ACK) {
230           if (recvHeader.seq > currentSeq) {
231             // move buffer
232             if (recvHeader.seq > currentSeq + bufNum) {
233               // clear buf
234               memset(buf, 0, windowSize * MAX_MESSAGE_SIZE);
235               bufNum = 0;
236             } else {
237               char *temp = malloc(windowSize * MAX_PAYLOAD_SIZE);
238               memcpy(temp, buf + (recvHeader.seq - currentSeq) * MAX_MESSAGE_SIZE,
239                      (bufNum + currentSeq - recvHeader.seq) * MAX_MESSAGE_SIZE);
240               memset(buf, 0, windowSize * MAX_MESSAGE_SIZE);
241               memcpy(buf, temp, windowSize * MAX_MESSAGE_SIZE);
242               free(temp);
243               bufNum = bufNum + currentSeq - recvHeader.seq;
244             }
245             // advance
246             currentSeq = recvHeader.seq;
247             if (currentSeq > (numSend - 1))
248               endFlag = 0;
249           }
250         }
251         // reset time
252         gettimeofday(&start, NULL);
253       }
254     }
255     // Send FIN
256     header.flag = FIN;
257     header.seq = randSeq;
258     header.len = 0;
259     header.crc = crc32(NULL, 0); // without payload;
260     memcpy(packet, (char *)&header, sizeof(header));
261     sendto(sockfd, (void *)packet, sizeof(header), 0,
262            (const struct sockaddr *)&recvAddr, sizeof(struct sockaddr_in));
263     logInfo(header, logfp);
264     printf("Tranmission finished: Send FIN");
265     // receive ack for FIN
266     while (1) {
267       if (recvfrom(sockfd, msg, MAX_MESSAGE_SIZE, MSG_DONTWAIT,
268                    (struct sockaddr *)&siOther, &sLen) != -1) {
269         memcpy(&recvHeader, msg, sizeof(recvHeader));
270         logInfo(recvHeader, logfp);
271         if (recvHeader.flag == ACK) {
```

```
272           printf("Receive ACK for FIN, done\n");
273           break;
274         }
275       } else {
276         // resend FIN
277         sendto(sockfd, (void *)packet, sizeof(header), 0,
278                (const struct sockaddr *)&recvAddr, sizeof(struct sockaddr_in));
279         logInfo(header, logfp);
280       }
281       // time out for ACK
282       gettimeofday(&stop, NULL);
283       if ((double)(stop.tv_sec - start.tv_sec) * 1000 +
284             (double)(stop.tv_usec - start.tv_usec) / 1000 >
285           400) {
286         perror("Error: do not receive ACK for FIN within 400 ms. Give up\n");
287         fclose(fp);
288         fclose(logfp);
289         free(buf);
290         exit(4);
291       }
292     }
293     fclose(fp);
294     fclose(logfp);
295     free(buf);
296     return 0;
297   }
298
299   int runReceiver(int port, unsigned int windowSize, char *recvDir,
300                   char *logFile) {
301     FILE *logfp = fopen(logFile, "w+");
302     // Set up connection
303     struct sockaddr_in siMe;
304     int sockfd = socket(AF_INET, SOCK_DGRAM, IPPROTO_UDP);
305     if (sockfd == -1) {
306       fputs("socket creation failed!", stderr);
307       exit(2); // error, exit with 2
308     }
309     memset((char *)&siMe, 0, sizeof(struct sockaddr_in));
310     siMe.sin_family = AF_INET;
311     siMe.sin_port = htons(port);
312     siMe.sin_addr.s_addr = htonl(INADDR_ANY);
313     bind(sockfd, (struct sockaddr *)&siMe, sizeof(struct sockaddr_in));
314
315     socklen_t sLen = sizeof(struct sockaddr_in);
316     struct sockaddr_in siOther;
317     struct timeval start, stop;
318     int connectCnt = 0;
319     while (1) {
320       // 1. Ready for receiving SYN
321       char msg[MAX_MESSAGE_SIZE];
322       memset(msg, 0, MAX_MESSAGE_SIZE);
323       char ACKMsg[MAX_MESSAGE_SIZE];
324       memset(ACKMsg, 0, sizeof(ACKMsg));
325       unsigned SYNFINSeq = 0;
326       struct SRHeader recvHeader;
327       struct SRHeader ACKHeader;
328
329       gettimeofday(&start, NULL);
330       gettimeofday(&stop, NULL);
331       while (1) {
332         if (recvfrom(sockfd, msg, MAX_MESSAGE_SIZE, MSG_DONTWAIT,
333                      (struct sockaddr *)&siOther, &sLen) > 0) {
334           memcpy(&recvHeader, msg, sizeof(recvHeader));
```

```
335           logInfo(recvHeader, logfp);
336           if (recvHeader.flag == SYN) {
337             SYNFINSeq = recvHeader.seq;
338             // ACK for SYN
339             ACKHeader.flag = ACK;
340             ACKHeader.seq = SYNFINSeq;
341             ACKHeader.len = 0;
342             ACKHeader.crc = crc32(NULL, 0);
343             memcpy(ACKMsg, (char *)&ACKHeader, sizeof(ACKHeader));
344             sendto(sockfd, (void *)ACKMsg, sizeof(ACKHeader), 0,
345                    (const struct sockaddr *)&siOther, sizeof(struct sockaddr_in));
346             logInfo(ACKHeader, logfp);
347             break;
348           }
349         }
350         gettimeofday(&stop, NULL);
351         if ((double)(stop.tv_sec - start.tv_sec) * 1000 +
352                 (double)(stop.tv_usec - start.tv_usec) / 1000 >
353             5000) {
354           printf("I do not receive SYN within 5 s. Give up\n");
355           return 0;
356         }
357       }
358       // 2. Recive data
359
360       // 2.1 open file
361       // create file name
362       char fileName[30];
363       char name[13];
364       sprintf(name, "file_%d.txt", connectCnt);
365       strcpy(fileName, recvDir);
366       strcat(fileName, name);
367       FILE *fp = fopen(fileName, "w");
368       unsigned int currentACKSeq = 0;
369       char *buf = malloc(windowSize * (MAX_PAYLOAD_SIZE + 8)); // valid. len
370       memset(buf, 0, windowSize * (MAX_PAYLOAD_SIZE + 8));
371       int FINflag = 1; // 0 for receive FIN flag
372
373       // 2.2 receive data
374       printf("Start to receive file\n");
375       while (FINflag) {
376         // receive data
377         if (recvfrom(sockfd, msg, MAX_MESSAGE_SIZE, MSG_DONTWAIT,
378                      (struct sockaddr *)&siOther, &sLen) != -1) {
379           memcpy(&recvHeader, msg, sizeof(recvHeader));
380           logInfo(recvHeader, logfp);
381           // If it's data
382           if (recvHeader.flag == DATA) {
383             if (recvHeader.crc ==
384                 (crc32(msg + sizeof(recvHeader), recvHeader.len))) {
385               // check the condition for window operation
386               if (recvHeader.seq >= currentACKSeq + windowSize) {
387                 continue;
388               } else if (recvHeader.seq < currentACKSeq) {
389                 // send ACK currentACKSeq
390                 ACKHeader.flag = ACK;
391                 ACKHeader.seq = currentACKSeq;
392                 ACKHeader.len = 0;
393                 ACKHeader.crc = crc32(NULL, 0);
394                 memcpy(ACKMsg, (char *)&ACKHeader, sizeof(ACKHeader));
395                 sendto(sockfd, (void *)ACKMsg, sizeof(ACKHeader), 0,
396                        (const struct sockaddr *)&siOther,
397                        sizeof(struct sockaddr_in));
```

```
398               logInfo(ACKHeader, logfp);
399             } else if ((recvHeader.seq > currentACKSeq) &&
400                        (recvHeader.seq < currentACKSeq + windowSize)) {
401               // buffer it
402               unsigned int temp = 1;
403               memcpy(buf + (recvHeader.seq - currentACKSeq) *
404                                (MAX_PAYLOAD_SIZE + 8),
405                      (char *)(&temp), sizeof(temp));
406               memcpy(buf +
407                          (recvHeader.seq - currentACKSeq) *
408                              (MAX_PAYLOAD_SIZE + 8) +
409                          4,
410                      (char *)(&recvHeader.len), sizeof(temp));
411               memcpy(buf +
412                          +(recvHeader.seq - currentACKSeq) *
413                              (MAX_PAYLOAD_SIZE + 8) +
414                          8,
415                      msg + sizeof(recvHeader), recvHeader.len);
416               // send current ACK
417               ACKHeader.flag = ACK;
418               ACKHeader.seq = currentACKSeq;
419               ACKHeader.len = 0;
420               ACKHeader.crc = crc32(NULL, 0);
421               memcpy(ACKMsg, (char *)&ACKHeader, sizeof(ACKHeader));
422               sendto(sockfd, (void *)ACKMsg, sizeof(ACKHeader), 0,
423                      (const struct sockaddr *)&siOther,
424                      sizeof(struct sockaddr_in));
425               logInfo(ACKHeader, logfp);
426             } else if (recvHeader.seq == currentACKSeq) {
427               // calculate the highest avalible packet
428               unsigned int numWrite = 1;
429               for (numWrite = 1; numWrite < windowSize; numWrite++) {
430                 unsigned int temp;
431                 memcpy(&temp, buf + numWrite * (8 + MAX_PAYLOAD_SIZE),
432                        sizeof(temp));
433                 if (temp == 0)
434                   break;
435               }
436               fwrite(msg + sizeof(recvHeader), 1, recvHeader.len, fp);
437               for (unsigned int ii = 1; ii < numWrite; ii++) {
438                 unsigned tempLen;
439                 memcpy(&tempLen, buf + ii * (8 + MAX_PAYLOAD_SIZE) + 4,
440                        sizeof(tempLen));
441                 fwrite(buf + ii * (8 + MAX_PAYLOAD_SIZE) + 8, 1, tempLen, fp);
442               }
443               // clear buff
444               memset(buf, 0, windowSize * (MAX_PAYLOAD_SIZE + 4));
445               // ACK
446               currentACKSeq = currentACKSeq + numWrite;
447               // send current ACK
448               ACKHeader.flag = ACK;
449               ACKHeader.seq = currentACKSeq;
450               ACKHeader.len = 0;
451               ACKHeader.crc = crc32(NULL, 0);
452               memcpy(ACKMsg, (char *)&ACKHeader, sizeof(ACKHeader));
453               sendto(sockfd, (void *)ACKMsg, sizeof(ACKHeader), 0,
454                      (const struct sockaddr *)&siOther,
455                      sizeof(struct sockaddr_in));
456               logInfo(ACKHeader, logfp);
457             }
458           }
459         } else if (recvHeader.flag == FIN) {
460           FINflag = 0;
```

```
461             // send ACK
462             ACKHeader.flag = ACK;
463             ACKHeader.seq = SYNFINSeq;
464             ACKHeader.len = 0;
465             ACKHeader.crc = crc32(NULL, 0);
466             memcpy(ACKMsg, (char *)&ACKHeader, sizeof(ACKHeader));
467             sendto(sockfd, (void *)ACKMsg, sizeof(ACKHeader), 0,
468                    (const struct sockaddr *)&siOther, sizeof(struct sockaddr_in));
469             logInfo(ACKHeader, logfp);
470             // end
471             fclose(fp);
472             free(buf);
473             connectCnt++;
474             fclose(logfp);
475           }
476         }
477       }
478     }
479     fclose(logfp);
480     return 0;
481   }
482
483   int main(int argc, char **argv) {
484     // Sender mode
485     if (strcmp(argv[1], "-s") == 0) {
486       // Error for number of arguments
487       if (argc != 8) {
488         perror(
489             "./sr -s <receiver's IP> <receiver's port> <sender's port> <window "
490             "size> <file to send> <log file>");
491         return 1; // error: exit with 1
492       }
493       char *recvIP = argv[2];
494       int recvPort = atoi(argv[3]);
495       int senderPort = atoi(argv[4]);
496       unsigned int windowSize = atoi(argv[5]);
497       char *fileToSend = argv[6];
498       char *logFile = argv[7];
499       runSender(recvIP, recvPort, senderPort, windowSize, fileToSend, logFile);
500       return 0;
501     }
502
503     // Receiver mode
504     else if (strcmp(argv[1], "-r") == 0) {
505       // Error for number of arguments
506       if (argc != 6) {
507         perror("./sr -r <port> <window size> <recv dir> <log file>");
508         return 1; // error: exit with 1
509       }
510       // int runReceiver(int port, unsigned int windowSize, char *recvDir,
511       //                 char *logFile)
512       int port = atoi(argv[2]);
513       unsigned int windowSize = atoi(argv[3]);
514       char *recvDir = argv[4];
515       char *logFile = argv[5];
516       runReceiver(port, windowSize, recvDir, logFile);
517       return 0;
518     } else {
519       perror("./sr -s <receiver's IP> <receiver's port> <sender's port> <window "
520              "size> <file to send> <log file>\n ./sr -r <port> <window size> "
521              "<recv dir> <log file>\n");
522       return 1;
523     }
```

```
524     return 0;
525   }
```

4.2.4 sr.c in 3.3

```
1   #include "./SRHeader.h"
2   #include "./crc32.h"
3   #include <arpa/inet.h>
4   #include <netdb.h>
5   #include <stdint.h>
6   #include <stdio.h>
7   #include <stdlib.h>
8   #include <string.h>
9   #include <sys/socket.h>
10  #include <sys/stat.h> // struct stat
11  #include <sys/time.h>
12  #include <time.h>
13  #include <unistd.h>
14  // The size of packet would is len(header) + len(payload) = 4*4 + 1456 = 1472
15  #define MAX_MESSAGE_SIZE 1472
16
17  int logInfo(struct SRHeader header, FILE *fp) {
18    if (header.flag == SYN) {
19      fprintf(fp, "%s %u %u %u\n", "SYN", header.seq, header.len, header.crc);
20      printf("%s %u %u %u\n", "SYN", header.seq, header.len, header.crc);
21    } else if (header.flag == FIN) {
22      fprintf(fp, "%s %u %u %u\n", "FIN", header.seq, header.len, header.crc);
23      printf("%s %u %u %u\n", "FIN", header.seq, header.len, header.crc);
24    } else if (header.flag == DATA) {
25      fprintf(fp, "%s %u %u %u\n", "DATA", header.seq, header.len, header.crc);
26      printf("%s %u %u %u\n", "DATA", header.seq, header.len, header.crc);
27    } else if (header.flag == ACK) {
28      fprintf(fp, "%s %u %u %u\n", "ACK", header.seq, header.len, header.crc);
29      printf("%s %u %u %u\n", "ACK", header.seq, header.len, header.crc);
30    } else if (header.flag == NACK) {
31      fprintf(fp, "%s %u %u %u\n", "NACK", header.seq, header.len, header.crc);
32      printf("%s %u %u %u\n", "NACK", header.seq, header.len, header.crc);
33    }
34    return 0;
35  }
36
37  int runSender(char *recvIP, int recvPort, int senderPort,
38                unsigned int windowSize, char *fileToSend, char *logFile) {
39    FILE *logfp = fopen(logFile, "w+");
40    // create UDP socket.
41    int sockfd = socket(AF_INET, SOCK_DGRAM, IPPROTO_UDP); // use IPPROTO_UDP
42    if (sockfd < 0) {
43      perror("socket creation failed");
44      exit(1);
45    }
46
47    // below is the same as TCP socket.
48    struct sockaddr_in myAddr;
49    memset(&myAddr, 0, sizeof(myAddr));
50    myAddr.sin_family = AF_INET;
51    myAddr.sin_addr.s_addr = INADDR_ANY;
52    // bind to a specific port
53    myAddr.sin_port = htons(senderPort);
54    bind(sockfd, (struct sockaddr *)&myAddr, sizeof(myAddr));
55
56    struct sockaddr_in recvAddr;
57    struct hostent *host = gethostbyname(recvIP);
58    memcpy(&(recvAddr.sin_addr), host->h_addr, host->h_length);
```

```
59    recvAddr.sin_family = AF_INET;
60    recvAddr.sin_port = htons(recvPort);
61
62    // 1. The num of packets to be sent
63    // 1.1 Read file length
64    struct stat statBuffer;
65    size_t fileLen = 0;
66    size_t numSend = 0;
67    size_t divFlag = 0;
68    // Read the file length
69    if (stat(fileToSend, &statBuffer) < 0) {
70      // Handling reding file error
71      perror("Error: Can not read file\n");
72      exit(-1);
73    }
74    fileLen = statBuffer.st_size;
75    // 1.2 Calculate num of packets to be sent
76    divFlag = fileLen % MAX_PAYLOAD_SIZE;
77    numSend = divFlag > 0 ? (1 + (fileLen / MAX_PAYLOAD_SIZE))
78                          : fileLen / MAX_PAYLOAD_SIZE;
79    printf("File size is %lu, need %lu packets to send\n", fileLen, numSend);
80
81    // 2. send
82    unsigned int randSeq = 0;
83    char packet[MAX_MESSAGE_SIZE];
84    char msg[MAX_MESSAGE_SIZE];
85    struct SRHeader header;
86    memset(packet, 0, sizeof(packet));
87    // 2.1 send SYN, has random seq number
88    srand((unsigned)time(NULL));
89    randSeq = rand();
90    header.flag = SYN;
91    header.seq = randSeq;
92    header.len = 0;
93    header.crc = crc32(NULL, 0); // without payload;
94    memcpy(packet, (char *)&header, sizeof(header));
95    sendto(sockfd, (void *)packet, sizeof(header), 0,
96           (const struct sockaddr *)&recvAddr, sizeof(struct sockaddr_in));
97    logInfo(header, logfp);
98    printf("Send SYN to set up the connection\n");
99    // 2.2 wait for ACK and SYN packet
100   socklen_t sLen = sizeof(struct sockaddr_in);
101   struct sockaddr_in siOther;
102   struct SRHeader recvHeader;
103   struct timeval start, stop;
104   gettimeofday(&start, NULL);
105   gettimeofday(&stop, NULL);
106
107   // receive ack
108   while (1) {
109     if (recvfrom(sockfd, msg, MAX_MESSAGE_SIZE, MSG_DONTWAIT,
110                  (struct sockaddr *)&siOther, &sLen) > 0) {
111       memcpy(&recvHeader, msg, sizeof(recvHeader));
112       logInfo(recvHeader, logfp);
113       if (recvHeader.flag == ACK) {
114         printf("Connection set up \n");
115         break;
116       }
117     } else {
118       // resend SYN
119       sendto(sockfd, (void *)packet, sizeof(header), 0,
120              (const struct sockaddr *)&recvAddr, sizeof(struct sockaddr_in));
121       logInfo(header, logfp);
```

```
122       }
123       // time out for ACK
124       gettimeofday(&stop, NULL);
125       if ((double)(stop.tv_sec - start.tv_sec) * 1000 +
126               (double)(stop.tv_usec - start.tv_usec) / 1000 >
127           5000) {
128         perror("Error: do not receive ACK for SYN within 5 s. Give up\n");
129         fclose(logfp);
130         exit(4);
131       }
132     }
133     // 2.3 Send data
134     char *buf = malloc(windowSize * MAX_MESSAGE_SIZE);
135     memset(buf, 0, windowSize * MAX_MESSAGE_SIZE);
136     FILE *fp = fopen(fileToSend, "r");
137     // open file
138     if (!fp) {
139       // Handling file open error
140       perror("Error: Can not read file\n");
141       free(buf);
142       fclose(logfp);
143       fclose(fp);
144       exit(-1);
145     }
146
147     unsigned int currentSeq = 0; // send 0 for initial
148     unsigned int bufNum = 0;
149     int endFlag = 1;
150     size_t payloadLen = MAX_PAYLOAD_SIZE;
151     gettimeofday(&start, NULL);
152     while (endFlag) {
153       // check if timeout
154       gettimeofday(&stop, NULL);
155       // if timeout, resend all buffer
156       if ((double)(stop.tv_sec - start.tv_sec) * 1000 +
157               (double)(stop.tv_usec - start.tv_usec) / 1000 >
158           400) {
159         printf(
160             "No ack within 400ms, resend all packets in buffer. Start from %u\n",
161             currentSeq);
162         // resend
163         for (unsigned int ii = 0; ii < bufNum; ii++) {
164           payloadLen = ((currentSeq + ii == numSend - 1) && (divFlag > 0))
165                            ? divFlag
166                            : MAX_PAYLOAD_SIZE;
167           sendto(sockfd, (void *)(buf + ii * sizeof(packet)),
168                  payloadLen + sizeof(header), 0,
169                  (const struct sockaddr *)&recvAddr, sizeof(struct sockaddr_in));
170           struct SRHeader tempHeader;
171           memcpy(&tempHeader, buf + ii * sizeof(packet), sizeof(header));
172           logInfo(tempHeader, logfp);
173         }
174         // reset time
175         gettimeofday(&start, NULL);
176       }
177       // send and buffer
178       if ((bufNum < windowSize) && ((bufNum + currentSeq) < numSend)) {
179         payloadLen = ((currentSeq + bufNum == numSend - 1) && (divFlag > 0))
180                          ? divFlag
181                          : MAX_PAYLOAD_SIZE;
182         // form a packet
183         fread(packet + sizeof(header), payloadLen, 1, fp);
184         header.flag = DATA;
```

```
185         header.seq = currentSeq + bufNum;
186         header.len = payloadLen;
187         header.crc = crc32(packet + sizeof(header), payloadLen);
188         memcpy(packet, (char *)(&header), sizeof(header));
189         // send
190         sendto(sockfd, (void *)packet, payloadLen + sizeof(header), 0,
191                (const struct sockaddr *)&recvAddr, sizeof(struct sockaddr_in));
192         logInfo(header, logfp);
193         // buffer the packet
194         memcpy(buf + bufNum * MAX_MESSAGE_SIZE, packet, sizeof(packet));
195         bufNum++;
196       }
197       // receive ACK
198       if (recvfrom(sockfd, msg, MAX_MESSAGE_SIZE, MSG_DONTWAIT,
199                    (struct sockaddr *)&siOther, &sLen) != -1) {
200         memcpy(&recvHeader, msg, sizeof(recvHeader));
201         logInfo(recvHeader, logfp);
202         // Advance window
203         if (recvHeader.flag == ACK) {
204           if (recvHeader.seq > currentSeq) {
205             // move buffer
206             if (recvHeader.seq > currentSeq + bufNum) {
207               // clear buf
208               memset(buf, 0, windowSize * MAX_MESSAGE_SIZE);
209               bufNum = 0;
210             } else {
211               char *temp = malloc(windowSize * MAX_PAYLOAD_SIZE);
212               memcpy(temp, buf + (recvHeader.seq - currentSeq) * MAX_MESSAGE_SIZE,
213                      (bufNum + currentSeq - recvHeader.seq) * MAX_MESSAGE_SIZE);
214               memset(buf, 0, windowSize * MAX_MESSAGE_SIZE);
215               memcpy(buf, temp, windowSize * MAX_MESSAGE_SIZE);
216               free(temp);
217               bufNum = bufNum + currentSeq - recvHeader.seq;
218             }
219             // advance
220             currentSeq = recvHeader.seq;
221             if (currentSeq > (numSend - 1))
222               endFlag = 0;
223           }
224           // resend if NACK
225         } else if (recvHeader.flag == NACK) {
226           payloadLen = ((recvHeader.seq == numSend - 1) && (divFlag > 0))
227                            ? divFlag
228                            : MAX_PAYLOAD_SIZE;
229           sendto(sockfd,
230                  (void *)(buf + (recvHeader.seq - currentSeq) * sizeof(packet)),
231                  payloadLen + sizeof(header), 0,
232                  (const struct sockaddr *)&recvAddr, sizeof(struct sockaddr_in));
233           struct SRHeader tempHeader;
234           memcpy(&tempHeader,
235                  buf + (recvHeader.seq - currentSeq) * sizeof(packet),
236                  sizeof(header));
237           logInfo(tempHeader, logfp);
238         }
239         // reset time
240         gettimeofday(&start, NULL);
241       }
242     }
243     // Send FIN
244     header.flag = FIN;
245     header.seq = randSeq;
246     header.len = 0;
247     header.crc = crc32(NULL, 0); // without payload;
```

```
248     memcpy(packet, (char *)&header, sizeof(header));
249     sendto(sockfd, (void *)packet, sizeof(header), 0,
250            (const struct sockaddr *)&recvAddr, sizeof(struct sockaddr_in));
251     logInfo(header, logfp);
252     printf("Tranmission finished: Send FIN");
253     // receive ack for FIN
254     while (1) {
255       if (recvfrom(sockfd, msg, MAX_MESSAGE_SIZE, MSG_DONTWAIT,
256                    (struct sockaddr *)&siOther, &sLen) != -1) {
257         memcpy(&recvHeader, msg, sizeof(recvHeader));
258         logInfo(recvHeader, logfp);
259         if (recvHeader.flag == ACK) {
260           printf("Receive ACK for FIN, done\n");
261           break;
262         }
263       } else {
264         // resend FIN
265         sendto(sockfd, (void *)packet, sizeof(header), 0,
266                (const struct sockaddr *)&recvAddr, sizeof(struct sockaddr_in));
267         logInfo(header, logfp);
268       }
269       // time out for ACK
270       gettimeofday(&stop, NULL);
271       if ((double)(stop.tv_sec - start.tv_sec) * 1000 +
272               (double)(stop.tv_usec - start.tv_usec) / 1000 >
273           400) {
274         perror("Error: do not receive ACK for FIN within 400 ms. Give up\n");
275         fclose(fp);
276         fclose(logfp);
277         free(buf);
278         exit(4);
279       }
280     }
281     fclose(fp);
282     fclose(logfp);
283     free(buf);
284     return 0;
285   }
286
287   int runReceiver(int port, unsigned int windowSize, char *recvDir,
288                   char *logFile) {
289     FILE *logfp = fopen(logFile, "w+");
290     // Set up connection
291     struct sockaddr_in siMe;
292     int sockfd = socket(AF_INET, SOCK_DGRAM, IPPROTO_UDP);
293     if (sockfd == -1) {
294       fputs("socket creation failed!", stderr);
295       exit(2); // error, exit with 2
296     }
297     memset((char *)&siMe, 0, sizeof(struct sockaddr_in));
298     siMe.sin_family = AF_INET;
299     siMe.sin_port = htons(port);
300     siMe.sin_addr.s_addr = htonl(INADDR_ANY);
301     bind(sockfd, (struct sockaddr *)&siMe, sizeof(struct sockaddr_in));
302
303     socklen_t sLen = sizeof(struct sockaddr_in);
304     struct sockaddr_in siOther;
305     struct timeval start, stop;
306     int connectCnt = 0;
307     while (1) {
308       // 1. Ready for receiving SYN
309       char msg[MAX_MESSAGE_SIZE];
310       memset(msg, 0, MAX_MESSAGE_SIZE);
```

```
311       char ACKMsg[MAX_MESSAGE_SIZE];
312       memset(ACKMsg, 0, sizeof(ACKMsg));
313       unsigned SYNFINSeq = 0;
314       struct SRHeader recvHeader;
315       struct SRHeader ACKHeader;
316
317       gettimeofday(&start, NULL);
318       gettimeofday(&stop, NULL);
319       while (1) {
320         if (recvfrom(sockfd, msg, MAX_MESSAGE_SIZE, MSG_DONTWAIT,
321                      (struct sockaddr *)&siOther, &sLen) > 0) {
322           memcpy(&recvHeader, msg, sizeof(recvHeader));
323           logInfo(recvHeader, logfp);
324           if (recvHeader.flag == SYN) {
325             SYNFINSeq = recvHeader.seq;
326             // ACK for SYN
327             ACKHeader.flag = ACK;
328             ACKHeader.seq = SYNFINSeq;
329             ACKHeader.len = 0;
330             ACKHeader.crc = crc32(NULL, 0);
331             memcpy(ACKMsg, (char *)&ACKHeader, sizeof(ACKHeader));
332             sendto(sockfd, (void *)ACKMsg, sizeof(ACKHeader), 0,
333                    (const struct sockaddr *)&siOther, sizeof(struct sockaddr_in));
334             logInfo(ACKHeader, logfp);
335             break;
336           }
337         }
338         gettimeofday(&stop, NULL);
339         if ((double)(stop.tv_sec - start.tv_sec) * 1000 +
340                 (double)(stop.tv_usec - start.tv_usec) / 1000 >
341             5000) {
342           printf("I do not receive SYN within 5 s. Give up\n");
343           return 0;
344         }
345       }
346       // 2. Recive data
347
348       // 2.1 open file
349       // create file name
350       char fileName[30];
351       char name[13];
352       sprintf(name, "file_%d.txt", connectCnt);
353       strcpy(fileName, recvDir);
354       strcat(fileName, name);
355       FILE *fp = fopen(fileName, "w");
356       unsigned int currentACKSeq = 0;
357       char *buf = malloc(windowSize * (MAX_PAYLOAD_SIZE + 8)); // valid. len
358       memset(buf, 0, windowSize * (MAX_PAYLOAD_SIZE + 8));
359       int FINflag = 1; // 0 for receive FIN flag
360       int advanceFlag = 0;
361       // 2.2 receive data
362       printf("Start to receive file\n");
363       while (FINflag) {
364         advanceFlag = 0;
365         // receive data
366         if (recvfrom(sockfd, msg, MAX_MESSAGE_SIZE, MSG_DONTWAIT,
367                      (struct sockaddr *)&siOther, &sLen) != -1) {
368           memcpy(&recvHeader, msg, sizeof(recvHeader));
369           logInfo(recvHeader, logfp);
370           // If it's data
371           if (recvHeader.flag == DATA) {
372             if (recvHeader.crc ==
373                 (crc32(msg + sizeof(recvHeader), recvHeader.len))) {
```

```
374             // check the condition for window operation
375             if (recvHeader.seq >= currentACKSeq + windowSize) {
376               continue;
377             } else if (recvHeader.seq < currentACKSeq) {
378               // send ACK currentACKSeq
379               ACKHeader.flag = ACK;
380               ACKHeader.seq = currentACKSeq;
381               ACKHeader.len = 0;
382               ACKHeader.crc = crc32(NULL, 0);
383               memcpy(ACKMsg, (char *)&ACKHeader, sizeof(ACKHeader));
384               sendto(sockfd, (void *)ACKMsg, sizeof(ACKHeader), 0,
385                      (const struct sockaddr *)&siOther,
386                      sizeof(struct sockaddr_in));
387               logInfo(ACKHeader, logfp);
388             } else if ((recvHeader.seq > currentACKSeq) &&
389                        (recvHeader.seq < currentACKSeq + windowSize)) {
390               // buffer it
391               unsigned int temp = 1;
392               memcpy(buf + (recvHeader.seq - currentACKSeq) *
393                                (MAX_PAYLOAD_SIZE + 8),
394                      (char *)(&temp), sizeof(temp));
395               memcpy(buf +
396                          (recvHeader.seq - currentACKSeq) *
397                              (MAX_PAYLOAD_SIZE + 8) +
398                          4,
399                      (char *)(&recvHeader.len), sizeof(temp));
400               memcpy(buf +
401                          +(recvHeader.seq - currentACKSeq) *
402                              (MAX_PAYLOAD_SIZE + 8) +
403                          8,
404                      msg + sizeof(recvHeader), recvHeader.len);
405               // send current ACK
406               ACKHeader.flag = ACK;
407               ACKHeader.seq = currentACKSeq;
408               ACKHeader.len = 0;
409               ACKHeader.crc = crc32(NULL, 0);
410               memcpy(ACKMsg, (char *)&ACKHeader, sizeof(ACKHeader));
411               sendto(sockfd, (void *)ACKMsg, sizeof(ACKHeader), 0,
412                      (const struct sockaddr *)&siOther,
413                      sizeof(struct sockaddr_in));
414               logInfo(ACKHeader, logfp);
415             } else if (recvHeader.seq == currentACKSeq) {
416               advanceFlag = 1;
417               // calculate the highest avalible packet
418               unsigned int numWrite = 1;
419               for (numWrite = 1; numWrite < windowSize; numWrite++) {
420                 unsigned int temp;
421                 memcpy(&temp, buf + numWrite * (8 + MAX_PAYLOAD_SIZE),
422                        sizeof(temp));
423                 if (temp == 0)
424                   break;
425               }
426               fwrite(msg + sizeof(recvHeader), 1, recvHeader.len, fp);
427               for (unsigned int ii = 1; ii < numWrite; ii++) {
428                 unsigned tempLen;
429                 memcpy(&tempLen, buf + ii * (8 + MAX_PAYLOAD_SIZE) + 4,
430                        sizeof(tempLen));
431                 fwrite(buf + ii * (8 + MAX_PAYLOAD_SIZE) + 8, 1, tempLen, fp);
432               }
433               // clear buff
434               memset(buf, 0, windowSize * (MAX_PAYLOAD_SIZE + 4));
435               // ACK
436               currentACKSeq = currentACKSeq + numWrite;
```

```
437                 // send current ACK
438                 ACKHeader.flag = ACK;
439                 ACKHeader.seq = currentACKSeq;
440                 ACKHeader.len = 0;
441                 ACKHeader.crc = crc32(NULL, 0);
442                 memcpy(ACKMsg, (char *)&ACKHeader, sizeof(ACKHeader));
443                 sendto(sockfd, (void *)ACKMsg, sizeof(ACKHeader), 0,
444                        (const struct sockaddr *)&siOther,
445                        sizeof(struct sockaddr_in));
446                 logInfo(ACKHeader, logfp);
447               }
448             }
449             // check for gap and send NACK
450             unsigned int continueCnt = 1;
451             unsigned int NACKFlag = 0;
452             for (continueCnt = 0; continueCnt < windowSize; continueCnt++) {
453               unsigned int temp;
454               memcpy(&temp, buf + continueCnt * (8 + MAX_PAYLOAD_SIZE),
455                      sizeof(temp));
456               if (temp == 0)
457                 break;
458             }
459             if (continueCnt < windowSize) {
460               if (continueCnt == 0) {
461                 if (!advanceFlag)
462                   NACKFlag = 1;
463               } else
464                 NACKFlag = 1;
465               if (NACKFlag) {
466                 ACKHeader.flag = NACK;
467                 ACKHeader.seq = currentACKSeq + continueCnt;
468                 ACKHeader.len = 0;
469                 ACKHeader.crc = crc32(NULL, 0);
470                 memcpy(ACKMsg, (char *)&ACKHeader, sizeof(ACKHeader));
471                 sendto(sockfd, (void *)ACKMsg, sizeof(ACKHeader), 0,
472                        (const struct sockaddr *)&siOther,
473                        sizeof(struct sockaddr_in));
474                 logInfo(ACKHeader, logfp);
475               }
476             }
477           } else if (recvHeader.flag == FIN) {
478             FINflag = 0;
479             // send ACK
480             ACKHeader.flag = ACK;
481             ACKHeader.seq = SYNFINSeq;
482             ACKHeader.len = 0;
483             ACKHeader.crc = crc32(NULL, 0);
484             memcpy(ACKMsg, (char *)&ACKHeader, sizeof(ACKHeader));
485             sendto(sockfd, (void *)ACKMsg, sizeof(ACKHeader), 0,
486                    (const struct sockaddr *)&siOther, sizeof(struct sockaddr_in));
487             logInfo(ACKHeader, logfp);
488             // end
489             fclose(fp);
490             free(buf);
491             connectCnt++;
492             fclose(logfp);
493           }
494         }
495       }
496     }
497     fclose(logfp);
498     return 0;
499   }
```

```
500
501 int main(int argc, char **argv) {
502   // Sender mode
503   if (strcmp(argv[1], "-s") == 0) {
504     // Error for number of arguments
505     if (argc != 8) {
506       perror(
507           "./sr -s <receiver's IP> <receiver's port> <sender's port> <window "
508           "size> <file to send> <log file>");
509       return 1; // error: exit with 1
510     }
511     char *recvIP = argv[2];
512     int recvPort = atoi(argv[3]);
513     int senderPort = atoi(argv[4]);
514     unsigned int windowSize = atoi(argv[5]);
515     char *fileToSend = argv[6];
516     char *logFile = argv[7];
517     runSender(recvIP, recvPort, senderPort, windowSize, fileToSend, logFile);
518     return 0;
519   }
520
521   // Receiver mode
522   else if (strcmp(argv[1], "-r") == 0) {
523     // Error for number of arguments
524     if (argc != 6) {
525       perror("./sr -r <port> <window size> <recv dir> <log file>");
526       return 1; // error: exit with 1
527     }
528     // int runReceiver(int port, unsigned int windowSize, char *recvDir,
529     //             char *logFile)
530     int port = atoi(argv[2]);
531     unsigned int windowSize = atoi(argv[3]);
532     char *recvDir = argv[4];
533     char *logFile = argv[5];
534     runReceiver(port, windowSize, recvDir, logFile);
535     return 0;
536   } else {
537     perror("./sr -s <receiver's IP> <receiver's port> <sender's port> <window "
538            "size> <file to send> <log file>\n ./sr -r <port> <window size> "
539            "<recv dir> <log file>\n");
540     return 1;
541   }
542   return 0;
543 }
```