Project title:

DATA2410 Reliable Transport Protocol (DRTP)

A file transfer application

Inspera candidate number: 111

Candidate: 111

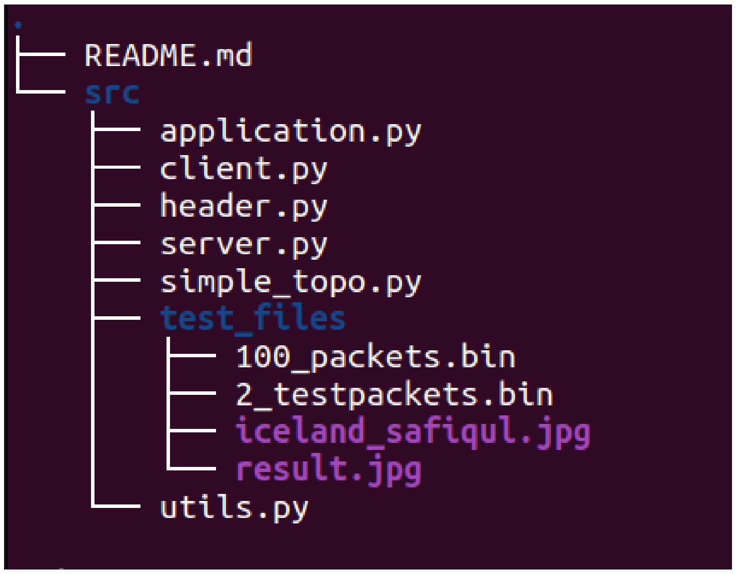
Introduction

For this home exam, we were asked to create a reliable transport protocol for transferring files from a client to a server. The protocol will use UDP as a base, and on top of that there should be a three-way handshake for establishing the connection and a connection tear-down when all the packets have been received and acknowledged. For sending the data, the Go-Back-N reliability function should be implemented, which utilizes a sliding window for higher throughput. On the server side it's required that the packets are received in-order, and if the server receives a packet out of order, it will discard it.

The application will be tested in a virtual network using Mininet, which can virtualize loss and round-trip time between two nodes in a network. It will be used to calculate throughput of different cases.

Implementation

To invoke the server or client with the wanted parameters, argparse [1] is used. This module was used in the first obligatory assignment, and some of the code has been copied from that assignment and later modified. [2] For creating a server socket and a client socket, obligatory assignment 2 has been used as a base, in combination with information from the professor's GitHub on how to implement UDP. [3][4] All client functions are gathered in one file and server functions in another. The main program, application.py, references the client and server files. Below, in Figure 1, you can see the file structure for the solution.



```
.
├── README.md
└── src
    ├── application.py
    ├── client.py
    ├── header.py
    ├── server.py
    ├── simple_topo.py
    ├── test_files
    │   ├── 100_packets.bin
    │   ├── 2_testpackets.bin
    │   ├── iceland_safiqul.jpg
    │   └── result.jpg
    └── utils.py
```

Figure 1: File structure for the DATA2410 Reliable Transport Protocol (DRTP).

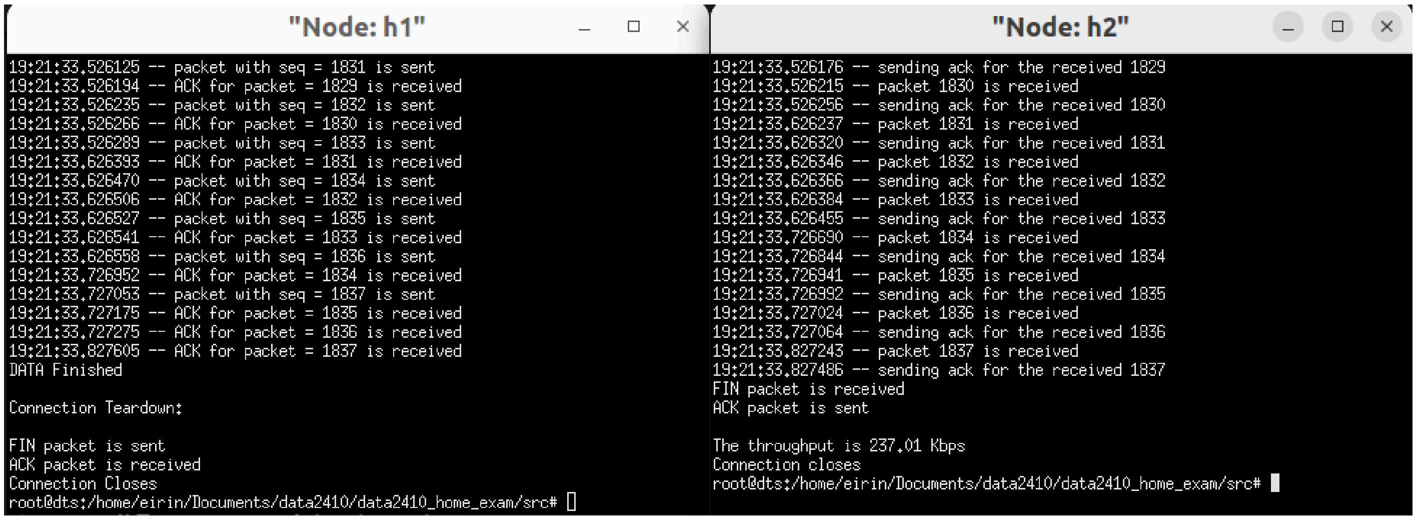
After having a functioning UDP socket and client, a three-way handshake and connection teardown was implemented using the flags of the header. A header class is created to contain all functions and information related to the header. Code from the professor's GitHub about headers [5] has been used and modified to fit this assignment. With the three-way handshake and connection teardown no data is sent before the connection is established, and no data is sent after the connection is torn down.

All functions useful across the other files have been collected into a utils.py file. These functions are more general functions used to simplify the code in the other files and increase readability.

For the Go-Back-N reliability function, the code was modified to include a sliding window for sending of packets. In addition, the acknowledgements are sent from the server whenever it has received an in-order packet. A timeout is used when the server or client is waiting for packets from the other. If the accumulated size of the packets in the window is bigger than the transferring file, the window will be set to the maximum number of packets instead. The Go-Back-N in this assignment is a bit different than some descriptions I've seen. For all of them, when a data packet fails, the next packets in the window will be received out-of-order. In this implementation, no ACKs are sent when packets are received out-of-order, but the client will resend the whole window if there's a timeout. For other implementations, however, the server would send an ACK for the last in-order packet as a response to receiving an out-of-order packet.

Discussion

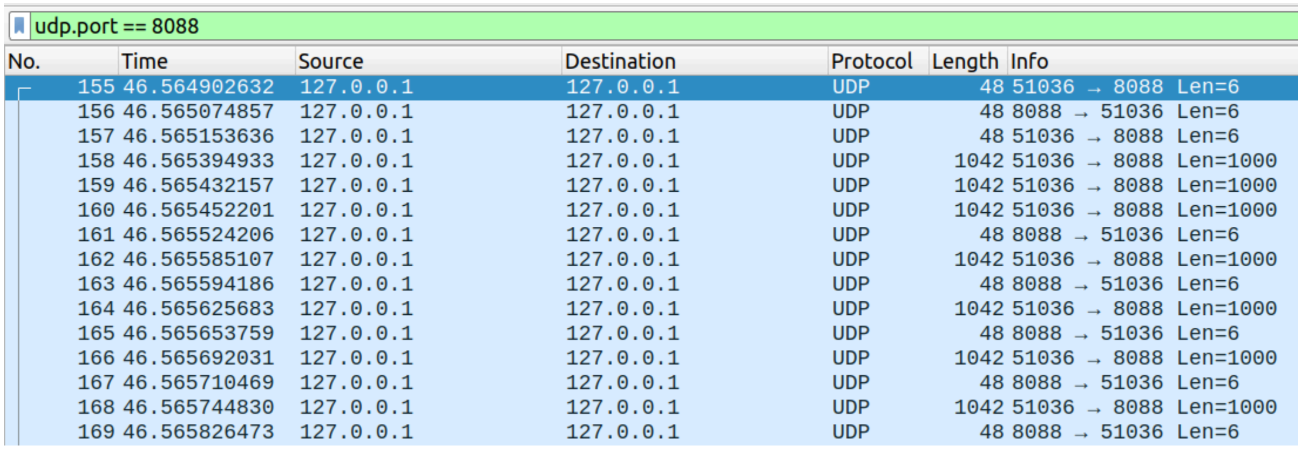
Testing the application in Mininet
In the simple_topo.py file, a virtual network is defined in Mininet. It has two nodes, h1 and h2, and one router, r, connecting them. The client runs on h1 (IP = 10.0.0.1) and the server runs on h2 (IP = 10.0.1.2). When using the program application.py, IP = 10.0.1.2 is used (this is not the default). In Figure 1 below, the end of a standard case is shown for a file transfer. Wireshark was also used to check the length of the packets, where the length of packets in the three-way handshake should be equal to the length of the header: 6 bytes. See Figure 2.



```
19:21:33.526125 -- packet with seq = 1831 is sent
19:21:33.526194 -- ACK for packet = 1829 is received
19:21:33.526235 -- packet with seq = 1832 is sent
19:21:33.526266 -- ACK for packet = 1830 is received
19:21:33.526289 -- packet with seq = 1833 is sent
19:21:33.626393 -- ACK for packet = 1831 is received
19:21:33.626470 -- packet with seq = 1834 is sent
19:21:33.626506 -- ACK for packet = 1832 is received
19:21:33.626527 -- packet with seq = 1835 is sent
19:21:33.626541 -- ACK for packet = 1833 is received
19:21:33.626558 -- packet with seq = 1836 is sent
19:21:33.726952 -- ACK for packet = 1834 is received
19:21:33.727053 -- packet with seq = 1837 is sent
19:21:33.727175 -- ACK for packet = 1835 is received
19:21:33.727275 -- ACK for packet = 1836 is received
19:21:33.827605 -- ACK for packet = 1837 is received
DATA Finished

Connection Teardown:

FIN packet is sent
ACK packet is received
Connection Closes
root@dts:/home/eirin/Documents/data2410/data2410_home_exam/src#
```

```
19:21:33.526176 -- sending ack for the received 1829
19:21:33.526215 -- packet 1830 is received
19:21:33.526256 -- sending ack for the received 1830
19:21:33.626237 -- packet 1831 is received
19:21:33.626320 -- sending ack for the received 1831
19:21:33.626346 -- packet 1832 is received
19:21:33.626366 -- sending ack for the received 1832
19:21:33.626384 -- packet 1833 is received
19:21:33.626455 -- sending ack for the received 1833
19:21:33.726690 -- packet 1834 is received
19:21:33.726844 -- sending ack for the received 1834
19:21:33.726941 -- packet 1835 is received
19:21:33.726992 -- sending ack for the received 1835
19:21:33.727024 -- packet 1836 is received
19:21:33.727064 -- sending ack for the received 1836
19:21:33.827243 -- packet 1837 is received
19:21:33.827486 -- sending ack for the received 1837
FIN packet is received
ACK packet is sent

The throughput is 237.01 Kbps
Connection closes
root@dts:/home/eirin/Documents/data2410/data2410_home_exam/src#
```

Figure 1: Shows a screenshot of the standard file transfer using Mininet.



```
udp.port == 8088
No.  Time          Source     Destination Protocol Length Info
155 46.564902632 127.0.0.1  127.0.0.1   UDP        48 51036 → 8088 Len=6
156 46.565074857 127.0.0.1  127.0.0.1   UDP        48 8088 → 51036 Len=6
157 46.565153636 127.0.0.1  127.0.0.1   UDP        48 51036 → 8088 Len=6
158 46.565394933 127.0.0.1  127.0.0.1   UDP      1042 51036 → 8088 Len=1000
159 46.565432157 127.0.0.1  127.0.0.1   UDP      1042 51036 → 8088 Len=1000
160 46.565452201 127.0.0.1  127.0.0.1   UDP      1042 51036 → 8088 Len=1000
161 46.565524206 127.0.0.1  127.0.0.1   UDP        48 8088 → 51036 Len=6
162 46.565585107 127.0.0.1  127.0.0.1   UDP      1042 51036 → 8088 Len=1000
163 46.565594186 127.0.0.1  127.0.0.1   UDP        48 8088 → 51036 Len=6
164 46.565625683 127.0.0.1  127.0.0.1   UDP      1042 51036 → 8088 Len=1000
165 46.565653759 127.0.0.1  127.0.0.1   UDP        48 8088 → 51036 Len=6
166 46.565692031 127.0.0.1  127.0.0.1   UDP      1042 51036 → 8088 Len=1000
167 46.565710469 127.0.0.1  127.0.0.1   UDP        48 8088 → 51036 Len=6
168 46.565744830 127.0.0.1  127.0.0.1   UDP      1042 51036 → 8088 Len=1000
169 46.565826473 127.0.0.1  127.0.0.1   UDP        48 8088 → 51036 Len=6
```

Figure 2: Packet transfer captured in Wireshark, for UDP and port 8088. Length of packets without data is 6 bytes (header size) and packets containing data is 1000 bytes.

Discussion part 1

The file was transferred from the client to the server in Mininet, using a round-trip time (RTT) of 100 ms, and window sizes 3, 5 and 10. All of the throughput results are listed in Table 1. THe throughput was calculated using file size divided by the total time it took to transfer the file. Here, the three-way handshake and connection teardown is included. In my mind it makes sense to include these because they are necessary code for the application to run properly. When transferring a smaller file, these code parts will be a bigger fraction of the time than for bigger files.
Beware that the throughput varies a bit with each time it is run in Mininet. Even if the round-trip time is the same, other things like CPU can influence the throughput. When completing these tests, I've tried to keep the number of active processes to a minimum.

Table 1: Throughput of a file transfer using DRTP in Mininet with round-trip time of 100 ms and window sizes 3, 4 and 5.

Window size	Round-trip time [ms]	Throughput [Mbps]
3	100	0.24
5	100	0.39
10	100	0.78

The throughput is calculated based on the total time used by the program and the size of the file. Since the size of the file is constant, the throughput is a function of the total time it takes for the transfer. From Table 1 you can see that the throughput increases with an increase in the window size. Doubling the window size results in double the throughput. This result fits well with what we have learned about sliding windows. Without a sliding window ("stop and wait"), the throughput would be equivalent to data size divided by round-trip time. When using a sliding window, the throughput is equivalent to data size divided by round-trip time, and multiplied by window size.

Stop and wait: $throughput \sim \frac{data}{RTT}$

Sliding window: $throughput \sim window\ size \cdot \frac{data}{RTT}$

Out of curiosity, I tested with a file of 100 packets and window size 100: the throughput was 3.60 Mbps. The same file, but with a window size of 10 gave a throughput of 0.69 Mbps. This doesn't fit with the equation above. The reason might be other constraints, maybe the processing time is higher for sending all the packets and then waiting for all the ACKs, then sending it with window 10. In the throughput results, handshake and connection teardown is included. Maybe the results would be closer to what we expect, based on the equation, if the handshake and connection teardown wasn't included.

Discussion part 2

The program was run again, but with different round-trip times (RTT): 50 ms and 200 ms. The results are listed in Table 2. The function for calculating throughput seems valid in respect to round-trip times as well. The throughput is approximately twice as large for RTT 50, compared to RTT 100 (with similar window size).

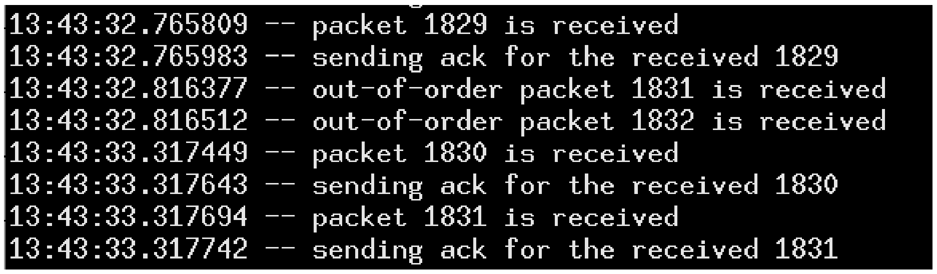
Table 2: Shows throughput in Mbps for different window sizes and different round-trip times, for the transfer of the same file.

Window size	Round-trip time [ms]	Throughput [Mbps]
3	50	0.47
5	50	0.78
10	50	1.54
3	200	0.12
5	200	0.20
10	200	0.39

Discussion part 3

For the discard test, I've chosen to use a delay of 50 ms and window size 3, and compare these results with the results in Table 1. The throughput was 0.46 Mbps. Packet no 1830 is dropped, and there's no other loss. So, with only one packet lost at the server side, the throughput almost didn't change.

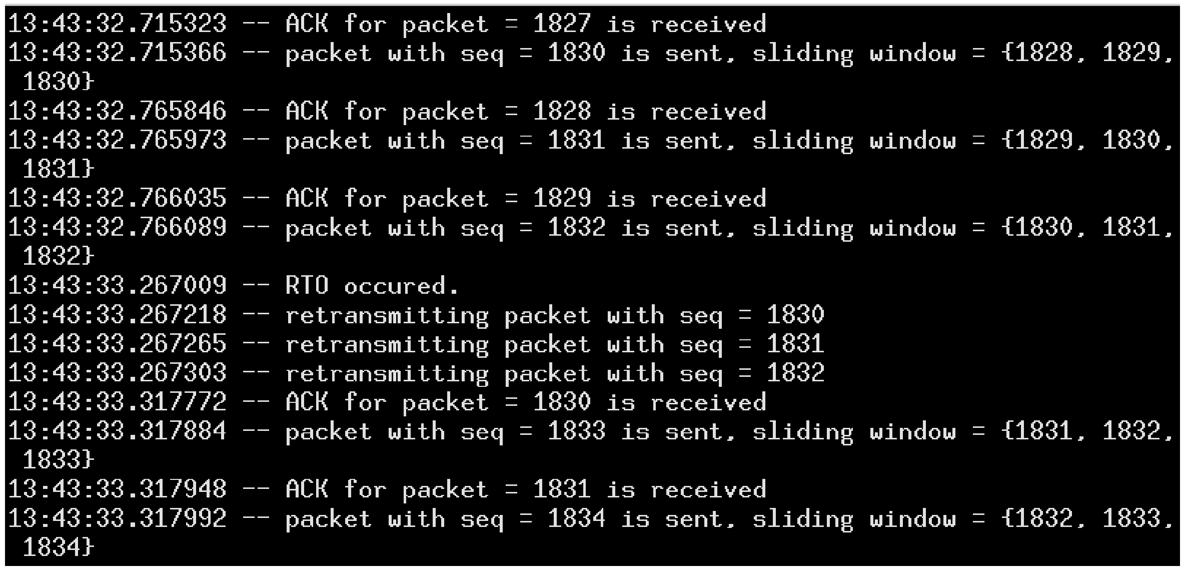
The client sends all packets, but when packet number 1830 arrives at the server side, it gets discarded. The server also receives 1831 and 1832, out-of-order, since the window size is 3. See Figure 3.



```
13:43:32.765809 -- packet 1829 is received
13:43:32.765983 -- sending ack for the received 1829
13:43:32.816377 -- out-of-order packet 1831 is received
13:43:32.816512 -- out-of-order packet 1832 is received
13:43:33.317449 -- packet 1830 is received
13:43:33.317643 -- sending ack for the received 1830
13:43:33.317694 -- packet 1831 is received
13:43:33.317742 -- sending ack for the received 1831
```

Figure 3: Shows how the server reacts when the packet 1830 is discarded.

On the other side of the connection, the client waits for an acknowledgement that the server has received packet 1830, but it doesn't get an acknowledgement. When the client has waited longer than the timeout, it will write "RTO occurred" and retransmit all the packets from the current window. As soon as it has an acknowledgement of the first packet in that window, the sliding window will shift, and the last packet in the window will be sent to the server. See Figure 4 below for the client view.



```
13:43:32.715323 -- ACK for packet = 1827 is received
13:43:32.715366 -- packet with seq = 1830 is sent, sliding window = {1828, 1829,
 1830}
13:43:32.765846 -- ACK for packet = 1828 is received
13:43:32.765973 -- packet with seq = 1831 is sent, sliding window = {1829, 1830,
 1831}
13:43:32.766035 -- ACK for packet = 1829 is received
13:43:32.766089 -- packet with seq = 1832 is sent, sliding window = {1830, 1831,
 1832}
13:43:33.267009 -- RTO occured.
13:43:33.267218 -- retransmitting packet with seq = 1830
13:43:33.267265 -- retransmitting packet with seq = 1831
13:43:33.267303 -- retransmitting packet with seq = 1832
13:43:33.317772 -- ACK for packet = 1830 is received
13:43:33.317884 -- packet with seq = 1833 is sent, sliding window = {1831, 1832,
 1833}
13:43:33.317948 -- ACK for packet = 1831 is received
13:43:33.317992 -- packet with seq = 1834 is sent, sliding window = {1832, 1833,
 1834}
```

Figure 4: Client view of how it looks when the server discards packet 1830. The client waits until timeout and retransmits all packets in the sliding window.

Discussion part 4:

Running the program yet again with different rates of loss. RTT = 100 ms and window = 3 or 10. Not surprisingly, the throughput goes down with increasing loss, see Table 3. For loss of 10% and 20%, it looks like doubling the loss will give approximately half the throughput. This is however not true when comparing 5% loss with 10% loss.

I tried to run it with 50% loss, but it was terminated after a couple of hours. Since two packets are needed for every successful data packet transfer, a 50% loss could for example mean that all ACKs are lost and the data transfer is never completed, so it makes sense that it takes a long time (maybe it never gets completed at all).

Table 3: Shows throughput of transfer of a file with DRTP using different types of loss and two different types of window size (3 and 10).

Window	**Loss [%]**	**Throughput [Mbps]**
3	2	0.18
3	5	0.13
3	10	0.09
3	20	0.05

Loss of packets is a more complicated case than discarding one data packet. When packets are lost it can be all types of packets: those involved in the handshake, connection teardown, data packets and acknowledgements for the data packets. Basically, any one of the packet types in Figure 5 below. When developing this code and testing it with loss, those types of cases occurred from time to time. Since those errors only occur from time to time it’s easier to forget about them, but the code should take them into account.

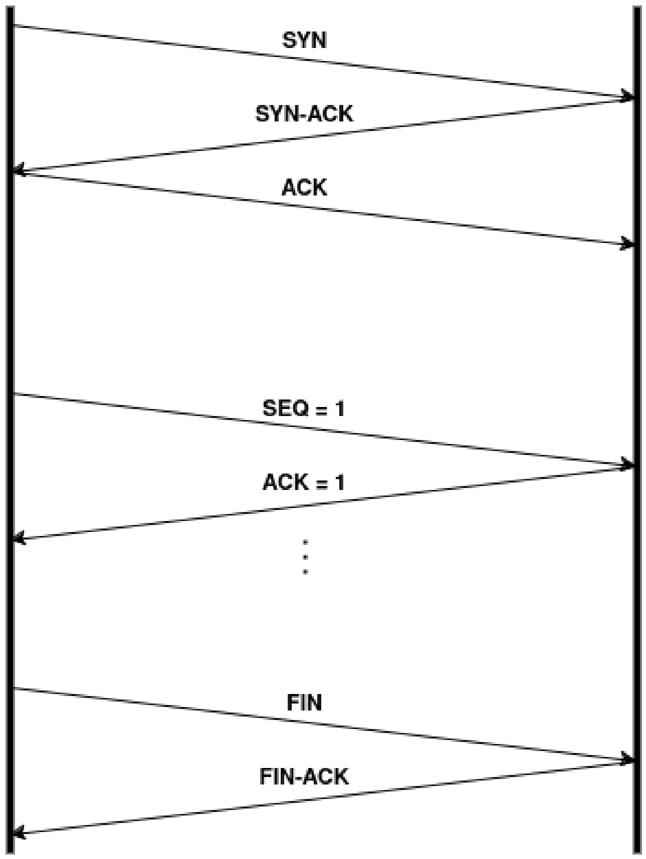
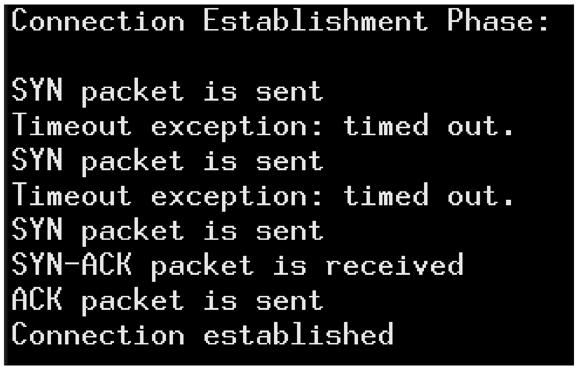


Figure 5: Illustration of all packets that possibly can be lost. The upper part is the handshake, the middle part illustrates transfer of data packets with sequence numbers and acknowledge numbers, and the lower part illustrates the connection tear down. Any of these can be lost in a network with loss.

Many of these issues only surface when stressing the system, so I've run the code several times with a loss of 20%.

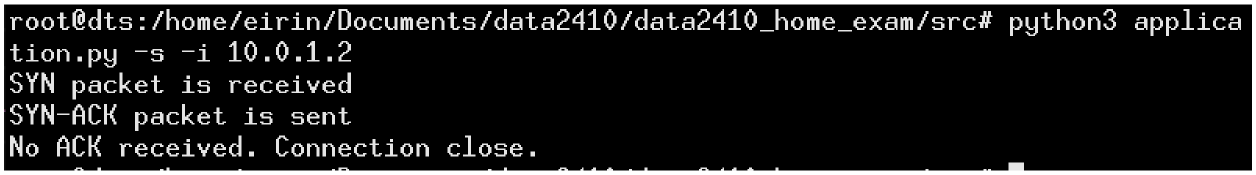
Handshake SYN loss: The client sends the SYN packet to start the connection, but it never arrives at the server. The server will continue waiting for a SYN packet. In my code the client will resend the SYN packet if it doesn't receive a SYN-ACK packet after 500 ms. Example in Figure 6.



```
Connection Establishment Phase:

SYN packet is sent
Timeout exception: timed out.
SYN packet is sent
Timeout exception: timed out.
SYN packet is sent
SYN-ACK packet is received
ACK packet is sent
Connection established
```

Figure 6: The client resends a SYN packet if no SYN-ACK packet is received.

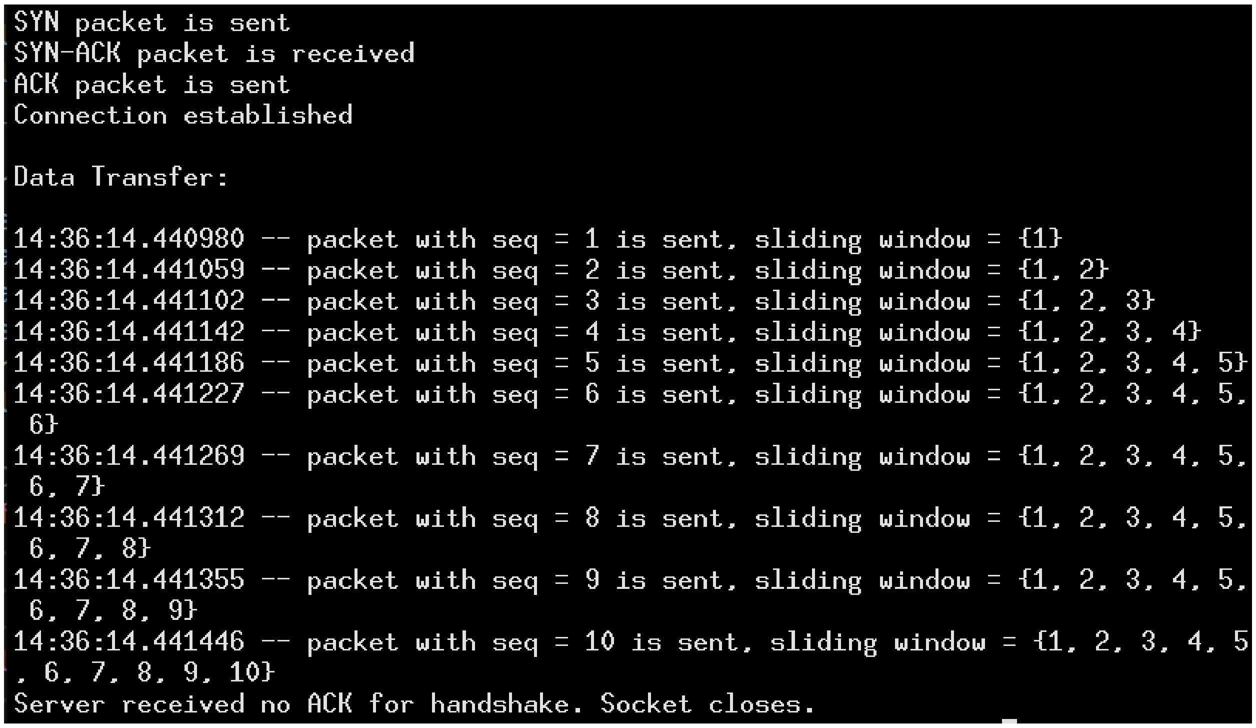
Handshake SYN-ACK loss or ACK loss: From the server perspective, if it has sent a SYN-ACK packet and it doesn't get ACK within a certain amount of time, either the SYN-ACK packet is lost or the ACK packet is lost, and there's no way of knowing which. Therefore, if it doesn't receive an ACK packet it closes down the connection. See Figure 7 for a printout from the terminal of how it looks from the server side.



```
root@dts:/home/eirin/Documents/data2410/data2410_home_exam/src# python3 applica
tion.py -s -i 10.0.1.2
SYN packet is received
SYN-ACK packet is sent
No ACK received. Connection close.
```

Figure 7: If the server doesn't receive an ACK in the handshake, it will close the connection.

On the client, when it has sent an ACK, it believes the connection is established and will start sending the first window of packets. However, it will receive a FIN flagged packet from the server and that will be the sign to close the connection. See Figure 8 for example.



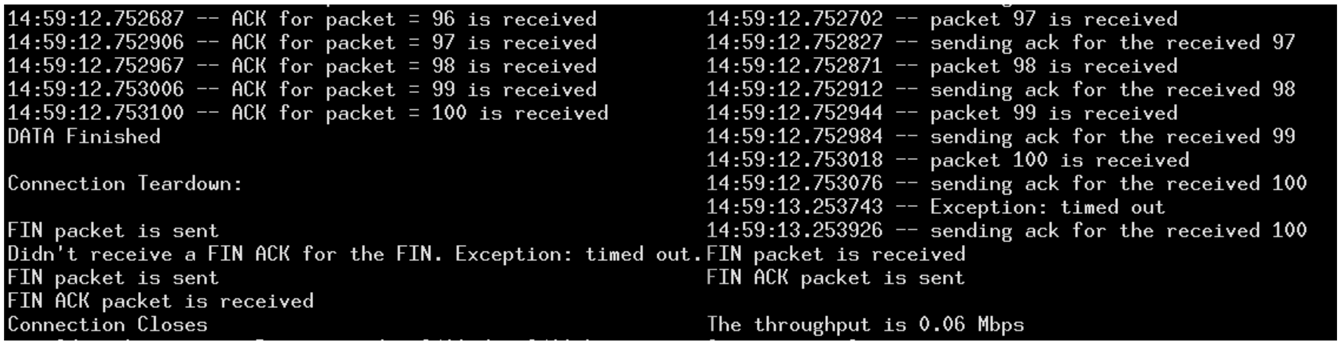
```
SYN packet is sent
SYN-ACK packet is received
ACK packet is sent
Connection established

Data Transfer:

14:36:14.440980 -- packet with seq = 1 is sent, sliding window = {1}
14:36:14.441059 -- packet with seq = 2 is sent, sliding window = {1, 2}
14:36:14.441102 -- packet with seq = 3 is sent, sliding window = {1, 2, 3}
14:36:14.441142 -- packet with seq = 4 is sent, sliding window = {1, 2, 3, 4}
14:36:14.441186 -- packet with seq = 5 is sent, sliding window = {1, 2, 3, 4, 5}
14:36:14.441227 -- packet with seq = 6 is sent, sliding window = {1, 2, 3, 4, 5,
 6}
14:36:14.441269 -- packet with seq = 7 is sent, sliding window = {1, 2, 3, 4, 5,
 6, 7}
14:36:14.441312 -- packet with seq = 8 is sent, sliding window = {1, 2, 3, 4, 5,
 6, 7, 8}
14:36:14.441355 -- packet with seq = 9 is sent, sliding window = {1, 2, 3, 4, 5,
 6, 7, 8, 9}
14:36:14.441446 -- packet with seq = 10 is sent, sliding window = {1, 2, 3, 4, 5
, 6, 7, 8, 9, 10}
Server received no ACK for handshake. Socket closes.
```

Figure 8: If the server doesn't receive the ACK in the handshake, it will tell the client to close the connection. This is the closing of the connection from the client side.

Data SEQ loss and ACK loss: this has been taken care of using a sliding window for sending packets, and acknowledging the last in-order packet received. If a SEQ packet is lost, the server will register the other packets as out-of-order, and after a timeout it will resend the last ACK if the expected packet doesn't arrive. The client will retransmit packets from the sliding window.

Connection teardown FIN loss: If the FIN is lost on its way to the server, the client will resend the FIN after a certain amount of time (500 ms). Example of this is given in Figure 9, when the client doesn't receive a FIN-ACK, it sends the FIN again. The server didn't receive the first FIN.

Candidate: 111



```
14:59:12.752687 -- ACK for packet = 96 is received        14:59:12.752702 -- packet 97 is received
14:59:12.752906 -- ACK for packet = 97 is received        14:59:12.752827 -- sending ack for the received 97
14:59:12.752967 -- ACK for packet = 98 is received        14:59:12.752871 -- packet 98 is received
14:59:12.753006 -- ACK for packet = 99 is received        14:59:12.752912 -- sending ack for the received 98
14:59:12.753100 -- ACK for packet = 100 is received       14:59:12.752944 -- packet 99 is received
DATA Finished                                             14:59:12.752984 -- sending ack for the received 99
                                                          14:59:12.753018 -- packet 100 is received
Connection Teardown:                                      14:59:12.753076 -- sending ack for the received 100
                                                          14:59:13.253743 -- Exception: timed out
FIN packet is sent                                        14:59:13.253926 -- sending ack for the received 100
Didn't receive a FIN ACK for the FIN. Exception: timed out.FIN packet is received
FIN packet is sent                                        FIN ACK packet is sent
FIN ACK packet is received
Connection Closes                                         The throughput is 0.06 Mbps
```

Figure 9: Shows the client on the left and the server on the right in the case where the FIN packet is lost on its way to the server.

Connection teardown FIN-ACK loss: If the FIN-ACK is lost, the server will believe the handshake is successful, but the client will continue to send FIN packets. This case has never occurred when testing in Mininet (even with 20% loss).

References

[1] Python 3.12.3 documentation. *argparse — Parser for command-line options, arguments and sub-commands*. Used 29th of April, 2024. https://docs.python.org/3/library/argparse.html
[2] Korvald, E. (16th of February, 2024) *eirinko/data2410oblig1*.
https://github.com/eirinko/data2410oblig1
[3] Korvald, E. (21st of March, 2024) *eirinko/data2410oblig2*.
https://github.com/eirinko/data2410oblig2
[4] Islam, S. (5th of February, 2023) *safiqul/2410/udp*.
https://github.com/safiqul/2410/tree/main/udp
[5] Islam, S. (24th of April, 2023) *safiqul/2410/header*.
https://github.com/safiqul/2410/blob/main/header/header.py