DRTP: A reliable transfer protocol

Candidate number: 160

Home exam 2025

DATA2410 Networking and cloud computing

OSLO METROPOLITAN UNIVERSITY
STORBYUNIVERSITETET

Content

Introduction

This project is about creating a simple reliable data transfer protocol called DATA2410 Reliable Transport Protocol (DRTP) on top of UDP. The goal was to develop a file transfer application that can operate as either a client (sender) or a server (receiver), ensuring reliable, in-order delivery of data without loss or duplication. Since UDP does not guarantee reliability, and sends data without any guarantees of delivery, correct order or error checking (Kurose & Ross, 2022, p. 226), the protocol adds necessary mechanisms such as connection establishment, acknowledgments, retransmissions, and connection teardown.

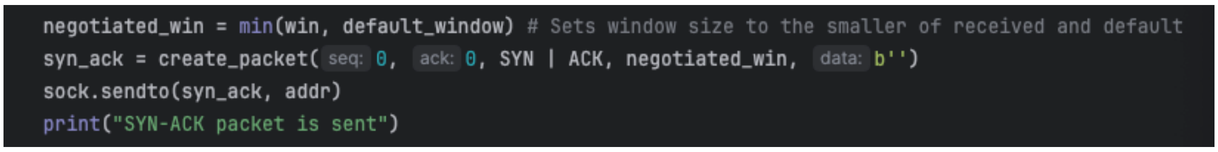
Implementation

Application.py, the main program, is run in either client or server mode, using command line arguments. All necessary fields are also provided through here, such as the IP address and port number for the receiver (server), the file to send, and the requested window size from the client. This is done by using Python's *argparse*, which creates a user-friendly command-line interface (Python Documentation, 2025a).

Before the client or server is invoked, the IP address and port are validated. Since the IP-argument provided is just a string and could be anything, it is important to check that this has the correct format. There are different ways to do this. I chose to use Python's *ipaddress* module, to avoid complex regex-validation and instead having a more readable single-line validation. It also checks for both IPv4 and IPv6 (Python Documentation, 2025b).

Connection establishment

To establish a reliable connection between the client and the server, the client initiates a three-way handshake, which is similar to how TCP does it (Kurose & Ross, 2022, p. 249-250). The client notifies the server that it wants to connect by sending a segment containing no data, only the SYN flag set (the SYN bit is set to one) and the window size set to the value requested through the command line argument. There is a total timeout set for 30 seconds on the server, to prevent it from waiting forever. When the server receives it (assuming it does), it sends back a SYN-ACK packet. Here, the SYN and ACK flag I set, and the window size is set to 15 (the maximum capacity for the receiver), unless the client has requested a smaller window. It is set to the smaller one of the two, preventing either side from exceeding the other's capacity.



```
negotiated_win = min(win, default_window) # Sets window size to the smaller of received and default
syn_ack = create_packet( seq: 0,  ack: 0, SYN | ACK, negotiated_win,  data: b'')
sock.sendto(syn_ack, addr)
print("SYN-ACK packet is sent")
```

Throughout this exchange, the header is parsed and the flags are extracted with help from functions defined in the header.py. Finally, when the client receives the SYN-ACK, it gives a final acknowledgement by sending an ACK-packet back, also using the window size agreed on. The window allows the sender to be more efficient, sending multiple packets at once without waiting for acknowledgments, but at the same time it is important to not exceed the receiver's ability to buffer these packets (Islam, 07.02.2025). Once the three steps are completed, they are ready to send data to each other.

Errors such as packet loss are handled with try/except blocks with timeouts and retransmissions. If the SYN-packet get lost, the SYN-ACK will also get lost, and the client will retransmit the SYN on timeout. It will try this a maximum of 5 times. It can be challenging to estimate a reasonable time to wait if you don’t know how far away the receiver is. A default has been 3 seconds, some also use 6 seconds. 3-6 seconds can be long if users get impatient (Islam, 07.02.2025). When testing with 2 seconds on the client and 3 seconds on the server, I found that this worked fine in most cases. It provides a bit more tolerance on the server side, and a little quicker retries on the client side.

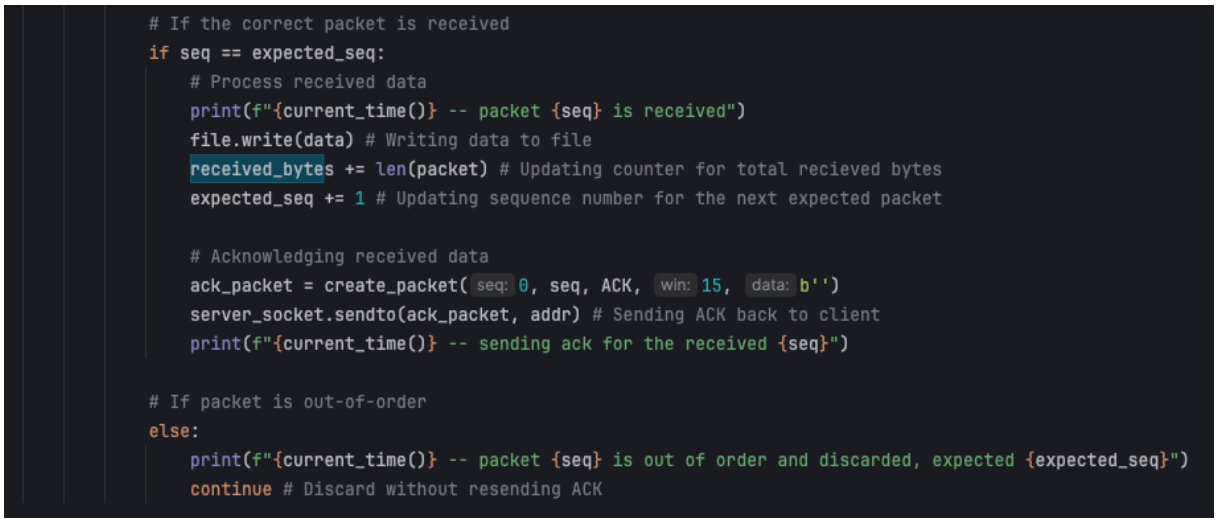
Data Transfer

Packets are sent reliably using Go-Back-N, which uses a sliding window mechanism. The sender (client) can send multiple packets, up to a fixed window size (3 as default in this assignment), without waiting for an acknowledgment after each one. As ACKs arrive, the window slides forward, and more packets can be sent. ACKs are cumulative, meaning an ACK for packet *n* confirms that all packets up to and including *n* were received correctly (Islam, 07.02.2025).

The client reads a file and splits it into chunks (packets) of 992 bytes, which are stored in a list. Error handling is included to ensure the file path is valid. The negotiated window size is returned from the three_way_handshake() method, if the connection establishment was successful. Three key variables control the sending process: base (earliest unacknowledged packet), next_seq (next packet to send), and total_packets (all packets to be sent). The client then enters a loop where it sends all packets within the current window using the create_packet() function from header.py. The header.py file is based on and adapted from Safiqul’s header code on Github (Islam, 2023), and includes useful variables and methods used for packing and unpacking.

The client uses a timer (400ms) for the oldest unacknowledged packet. When an ACK is received correctly, base is updated to ack + 1 (the window slides). The timer is either restarted (if there are still unacknowledged packets) or stopped (if all packets are acknowledged). If a timeout occurs, the client performs Go-Back-N retransmission by resetting next_seq to base, and all packets in the window are retransmitted (Kurose & Ross, 2022, p. 218).

On the server side, A UDP socket waits for a connection. The server only accepts packets that arrive in-order, tracked using sequence numbers (expected_seq). Inside the main loop, it receives packets, separates the header and the data, and parses them. If a packet arrives in-order (with the expected sequence number), the data is written to the file and an ACK is sent. If a packet is missing or arrives out of order, it is discarded. In Go-Back-N, the receiver usually sends an ACK for the last correctly received packet, but it does not retransmit on duplicate ACKs like TCP, and it does not store out-of-order packets, so any lost or out out-of-order packets are retransmitted (Kurose & Ross, 2022, p. 219). As specified in this assignment, the DRTP server does not send any ACK at all when an out-of-order packet arrives, it is just ignored (using continue).



```python
            # If the correct packet is received
            if seq == expected_seq:
                # Process received data
                print(f"{current_time()} -- packet {seq} is received")
                file.write(data) # Writing data to file
                received_bytes += len(packet) # Updating counter for total recieved bytes
                expected_seq += 1 # Updating sequence number for the next expected packet

                # Acknowledging received data
                ack_packet = create_packet( seq: 0, seq, ACK,  win: 15,  data: b'')
                server_socket.sendto(ack_packet, addr) # Sending ACK back to client
                print(f"{current_time()} -- sending ack for the received {seq}")

            # If packet is out-of-order
            else:
                print(f"{current_time()} -- packet {seq} is out of order and discarded, expected {expected_seq}")
                continue # Discard without resending ACK
```

If a FIN flag is set in the packet, the server calls connection_teardown() and calculate_throughput() and exits the loop.

Connection teardown

To close the connection, a two-way handshake is used. This logic is separated into their own methods (just like the connection establishment) to improve readability and making it easier to debug and test. When the data transfer is completed, the client calls the connection_teardown() method. This sends a packet with the FIN flag to the server, indicating that the client has finished sending data.

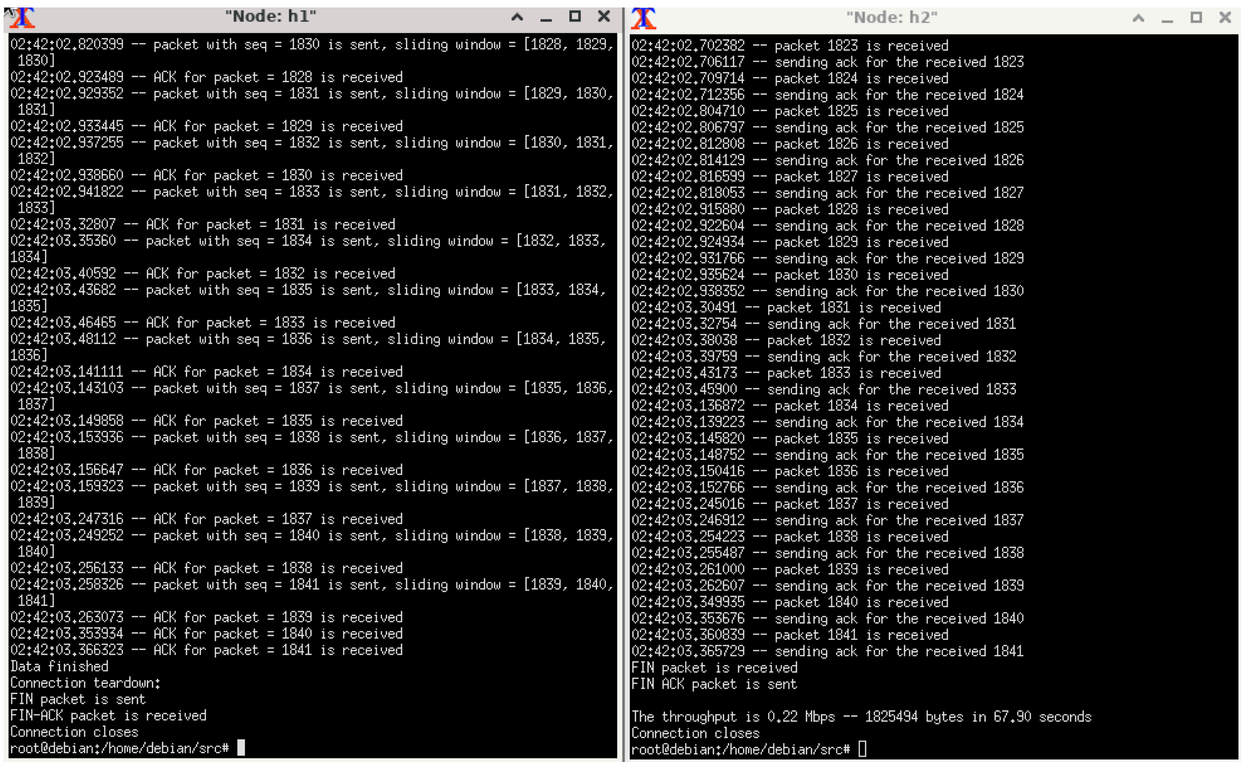
On the server side, this triggers its own connection_teardown() method, which responds with a FIN-ACK packet, confirming that it is also ready to close the connection. This exchange allows both sides to acknowledge that the communication is complete. If the client receives the FIN-ACK, it closes the socket. If no response is received within 3 seconds, a timeout occurs, and the client assumes the FIN packet was lost. It retransmits it, retrying up to 5 times (default, can be changed by passing a different value when calling the method). If no FIN-ACK is received after all retries, the client closes the socket regardless. This ensures that the connection is closed as gracefully as possible, even if something is wrong (e.g. packet loss or network issues).

Discussion

The code is tested in Mininet, using the provided simple-topo.py and the iceland-safiqul.jpg.

Discussion #1

The screenshot below are the results from sending the Iceland-safiqul.jpg file with a window size of 3. Results for all window sizes are listed in *Table 1.*



```
02:42:02.820399 -- packet with seq = 1830 is sent, sliding window = [1828, 1829,
 1830]
02:42:02.923489 -- ACK for packet = 1828 is received
02:42:02.929352 -- packet with seq = 1831 is sent, sliding window = [1829, 1830,
 1831]
02:42:02.933445 -- ACK for packet = 1829 is received
02:42:02.937255 -- packet with seq = 1832 is sent, sliding window = [1830, 1831,
 1832]
02:42:02.938660 -- ACK for packet = 1830 is received
02:42:02.941822 -- packet with seq = 1833 is sent, sliding window = [1831, 1832,
 1833]
02:42:03.32807 -- ACK for packet = 1831 is received
02:42:03.35360 -- packet with seq = 1834 is sent, sliding window = [1832, 1833,
1834]
02:42:03.40592 -- ACK for packet = 1832 is received
02:42:03.43682 -- packet with seq = 1835 is sent, sliding window = [1833, 1834,
1835]
02:42:03.46465 -- ACK for packet = 1833 is received
02:42:03.48112 -- packet with seq = 1836 is sent, sliding window = [1834, 1835,
1836]
02:42:03.141111 -- ACK for packet = 1834 is received
02:42:03.143103 -- packet with seq = 1837 is sent, sliding window = [1835, 1836,
 1837]
02:42:03.149858 -- ACK for packet = 1835 is received
02:42:03.153936 -- packet with seq = 1838 is sent, sliding window = [1836, 1837,
 1838]
02:42:03.156647 -- ACK for packet = 1836 is received
02:42:03.159323 -- packet with seq = 1839 is sent, sliding window = [1837, 1838,
 1839]
02:42:03.247316 -- ACK for packet = 1837 is received
02:42:03.249252 -- packet with seq = 1840 is sent, sliding window = [1838, 1839,
 1840]
02:42:03.256133 -- ACK for packet = 1838 is received
02:42:03.258326 -- packet with seq = 1841 is sent, sliding window = [1839, 1840,
 1841]
02:42:03.263073 -- ACK for packet = 1839 is received
02:42:03.353934 -- ACK for packet = 1840 is received
02:42:03.366323 -- ACK for packet = 1841 is received
Data finished
Connection teardown:
FIN packet is sent
FIN-ACK packet is received
Connection closes
root@debian:/home/debian/src#
```

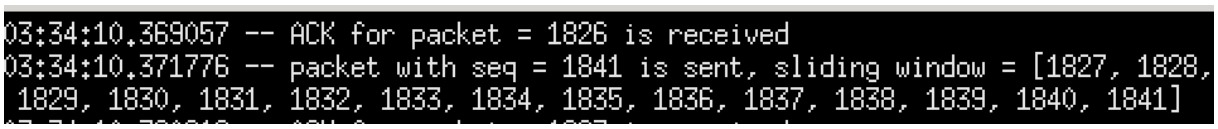
```
02:42:02.702382 -- packet 1823 is received
02:42:02.706117 -- sending ack for the received 1823
02:42:02.709714 -- packet 1824 is received
02:42:02.712356 -- sending ack for the received 1824
02:42:02.804710 -- packet 1825 is received
02:42:02.806797 -- sending ack for the received 1825
02:42:02.812808 -- packet 1826 is received
02:42:02.814129 -- sending ack for the received 1826
02:42:02.816599 -- packet 1827 is received
02:42:02.818053 -- sending ack for the received 1827
02:42:02.915880 -- packet 1828 is received
02:42:02.922604 -- sending ack for the received 1828
02:42:02.924934 -- packet 1829 is received
02:42:02.931766 -- sending ack for the received 1829
02:42:02.935624 -- packet 1830 is received
02:42:02.938352 -- sending ack for the received 1830
02:42:03.30491 -- packet 1831 is received
02:42:03.32754 -- sending ack for the received 1831
02:42:03.38038 -- packet 1832 is received
02:42:03.39759 -- sending ack for the received 1832
02:42:03.43173 -- packet 1833 is received
02:42:03.45900 -- sending ack for the received 1833
02:42:03.136872 -- packet 1834 is received
02:42:03.139223 -- sending ack for the received 1834
02:42:03.145820 -- packet 1835 is received
02:42:03.148752 -- sending ack for the received 1835
02:42:03.150416 -- packet 1836 is received
02:42:03.152766 -- sending ack for the received 1836
02:42:03.245016 -- packet 1837 is received
02:42:03.246912 -- sending ack for the received 1837
02:42:03.254223 -- packet 1838 is received
02:42:03.255487 -- sending ack for the received 1838
02:42:03.261000 -- packet 1839 is received
02:42:03.262607 -- sending ack for the received 1839
02:42:03.349935 -- packet 1840 is received
02:42:03.353676 -- sending ack for the received 1840
02:42:03.360839 -- packet 1841 is received
02:42:03.365729 -- sending ack for the received 1841
FIN packet is received
FIN ACK packet is sent

The throughput is 0.22 Mbps -- 1825494 bytes in 67.90 seconds
Connection closes
root@debian:/home/debian/src#
```

Table 1: *Results from discussion task 1, testing with different window sizes.*

Window size	Throughput	Transfer time
3	0.22 Mbps	67.90 sec
5	0.37 Mbps	39.93 sec
10	0.69 Mbps	21.12 sec
15	1.01 Mbps	14.40 sec
20	1.06 Mbps	13.84 sec
25	1.06 Mbps	13.79 sec

Throughput plateaus at a window size of 15, as the receiver's maximum window capacity is fixed at 15. Even if the client requests a larger window than this, it will still be set to 15.



```
03:34:10.369057 -- ACK for packet = 1826 is received
03:34:10.371776 -- packet with seq = 1841 is sent, sliding window = [1827, 1828,
 1829, 1830, 1831, 1832, 1833, 1834, 1835, 1836, 1837, 1838, 1839, 1840, 1841]
```

We can see a clear relationship between window size and throughput. As the window size increase, the throughput increases as well. This shows an important aspect of the Go-Back-N. A larger window will allow more packets to be in flight at the same time, without waiting for ACKs, which improves efficiency (Islam, 07.02.2025). Unlike a stop-and-wait protocol, which is more limited by the round-trip time (RTT), it depends on the window size, allowing more data to be "in the pipe" at once, maximizing the link capacity (up to the bandwidth limit) (Kurose & Ross, 2022, p. 215).

Discussion #2

The results from this task is presented in table 2 and 3 below.

Table 2: *Results using RTT of 50 ms.*

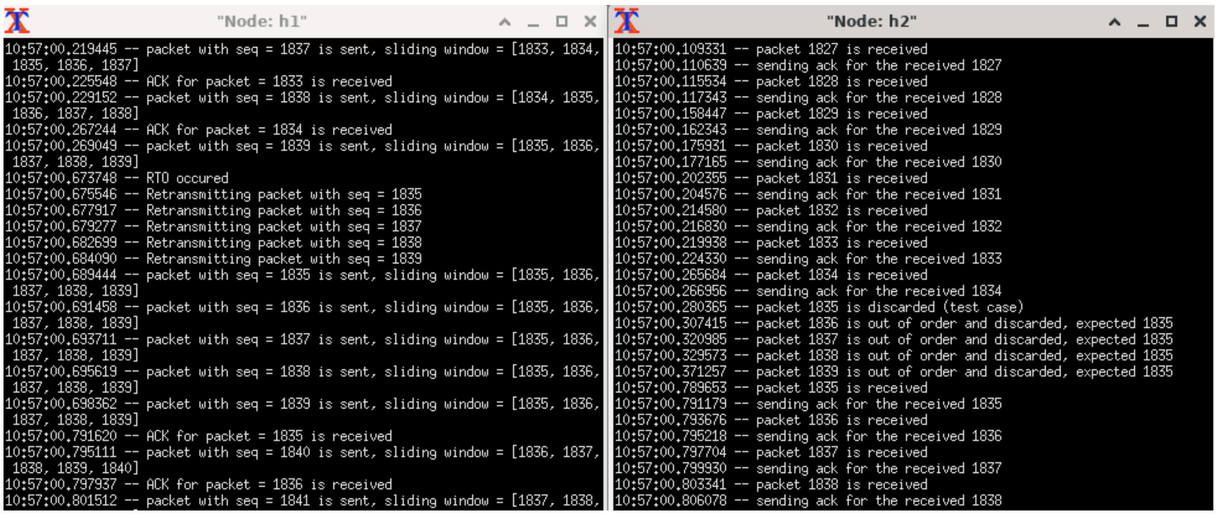
Window size	Throughput	Transfer time
3	0.41 Mbps	35.67 sec
5	0.67 Mbps	21.69 sec
10	1.20 Mbps	12.17 sec
15	1.32 Mbps	11.09 sec
20	1.30 Mbps	11.24 sec
25	1.20 Mbps	12.19 sec

Table 3: *Results using RTT of 200 ms.*

Window size	Throughput	Transfer time
3	0.11 Mbps	131.23 sec
5	0.18 Mbps	79.32 sec
10	0.37 Mbps	39.07 sec
15	0.57 Mbps	25.84 sec
20	0.57 Mbps	25.82 sec
25	0.57 Mbps	25.81 sec

The Round-Trip Time (RTT) is the time it takes for a packet to travel from client to server and back to the client (Kurose & Ross, 2022, p. 99). The results show that the longest RTT gives the lowest throughput, while the shortest RTT gives the highest throughput. It improves as the window grows. Longer RTT means slower ACK returns. This will reduce the efficiency, even with larger windows, because it defines how long the sender must wait before sending new data. Throughput drops by approximately 56% when the RTT increases 4x (even with the same window size). With a RTT of 50ms, ACKs return quickly, keeping the pipe full. At 200ms, ACKs take about 4 times longer, leaving the sender on waiting-mode more often.

Discussion #3


"Node: h1"

```
10:57:00.219445 -- packet with seq = 1837 is sent, sliding window = [1833, 1834,
 1835, 1836, 1837]
10:57:00.225548 -- ACK for packet = 1833 is received
10:57:00.229152 -- packet with seq = 1838 is sent, sliding window = [1834, 1835,
 1836, 1837, 1838]
10:57:00.267244 -- ACK for packet = 1834 is received
10:57:00.269049 -- packet with seq = 1839 is sent, sliding window = [1835, 1836,
 1837, 1838, 1839]
10:57:00.673748 -- RTO occured
10:57:00.675546 -- Retransmitting packet with seq = 1835
10:57:00.677917 -- Retransmitting packet with seq = 1836
10:57:00.679277 -- Retransmitting packet with seq = 1837
10:57:00.682699 -- Retransmitting packet with seq = 1838
10:57:00.684090 -- Retransmitting packet with seq = 1839
10:57:00.689444 -- packet with seq = 1835 is sent, sliding window = [1835, 1836,
 1837, 1838, 1839]
10:57:00.691458 -- packet with seq = 1836 is sent, sliding window = [1835, 1836,
 1837, 1838, 1839]
10:57:00.693711 -- packet with seq = 1837 is sent, sliding window = [1835, 1836,
 1837, 1838, 1839]
10:57:00.695619 -- packet with seq = 1838 is sent, sliding window = [1835, 1836,
 1837, 1838, 1839]
10:57:00.698362 -- packet with seq = 1839 is sent, sliding window = [1835, 1836,
 1837, 1838, 1839]
10:57:00.791620 -- ACK for packet = 1835 is received
10:57:00.795111 -- packet with seq = 1840 is sent, sliding window = [1836, 1837,
 1838, 1839, 1840]
10:57:00.797937 -- ACK for packet = 1836 is received
10:57:00.801512 -- packet with seq = 1841 is sent, sliding window = [1837, 1838,
```

"Node: h2"

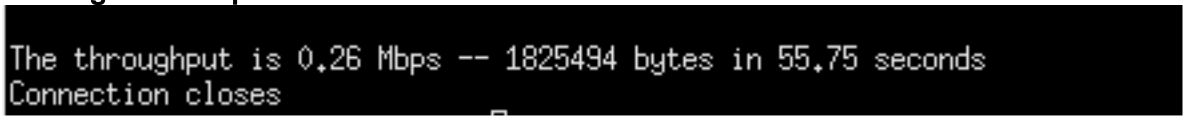
```
10:57:00.109331 -- packet 1827 is received
10:57:00.110639 -- sending ack for the received 1827
10:57:00.115534 -- packet 1828 is received
10:57:00.117343 -- sending ack for the received 1828
10:57:00.158447 -- packet 1829 is received
10:57:00.162343 -- sending ack for the received 1829
10:57:00.175931 -- packet 1830 is received
10:57:00.177165 -- sending ack for the received 1830
10:57:00.202355 -- packet 1831 is received
10:57:00.204576 -- sending ack for the received 1831
10:57:00.214580 -- packet 1832 is received
10:57:00.216830 -- sending ack for the received 1832
10:57:00.219938 -- packet 1833 is received
10:57:00.224330 -- sending ack for the received 1833
10:57:00.265684 -- packet 1834 is received
10:57:00.266956 -- sending ack for the received 1834
10:57:00.280365 -- packet 1835 is discarded (test case)
10:57:00.307415 -- packet 1836 is out of order and discarded, expected 1835
10:57:00.320985 -- packet 1837 is out of order and discarded, expected 1835
10:57:00.329573 -- packet 1838 is out of order and discarded, expected 1835
10:57:00.371257 -- packet 1839 is out of order and discarded, expected 1835
10:57:00.789653 -- packet 1835 is received
10:57:00.791179 -- sending ack for the received 1835
10:57:00.793676 -- packet 1836 is received
10:57:00.795218 -- sending ack for the received 1836
10:57:00.797704 -- packet 1837 is received
10:57:00.799930 -- sending ack for the received 1837
10:57:00.803341 -- packet 1838 is received
10:57:00.806078 -- sending ack for the received 1838
```

In this test, packet with sequence number 1835 was set to be discarded with the -d argument on the server side. The client waits for an ACK for 400ms, and at timeout (RTO occurred) it retransmits all the unacknowledged packets in the window.

Discussion #4

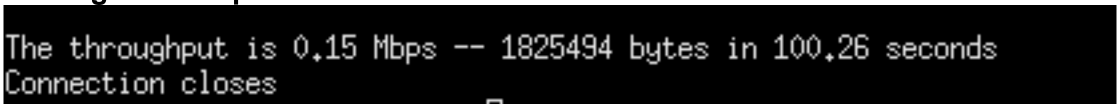
Using window size 5 for all these tests.

Testing with a 2 percent loss rate



```
The throughput is 0.26 Mbps -- 1825494 bytes in 55.75 seconds
Connection closes
```

Testing with a 5 percent loss rate

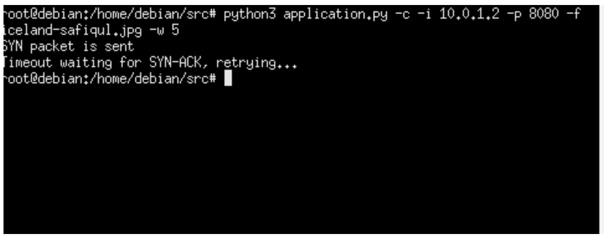


```
The throughput is 0.15 Mbps -- 1825494 bytes in 100.26 seconds
Connection closes
```

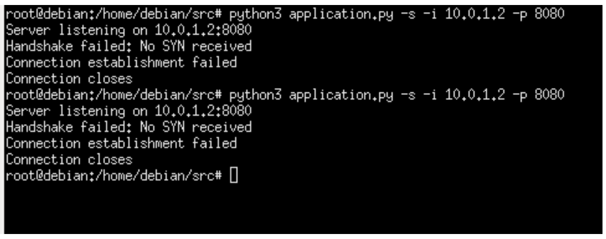
Throughput decreases as the loss rate rises. Every lost packet forces the sender to retransmit the whole window. If the window size is 5 and 1 packet is lost, all 5 are resent. With a 5% loss, 1 in 20 packets is lost, and 1 in 20 windows is retransmitted. 25% of the traffic is retransmissions (5% loss times 5-packet window). Because of the retransmission and timeout delays, the throughput goes down.

Testing with a 50 percent loss rate
When running the code with a 50% packet loss rate, various issues occurred because any packet type had a high probability of being dropped, unlike the single packet loss scenario in the data transfer (with the discard test case).

In my first attempt, the SYN packet was lost immediately. I had to make the handshake error handling more robust, with retries and retransmissions, to be able to see how it behaves in the data transfer phase and teardown phase with this loss rate.



```
oot@debian:/home/debian/src# python3 application.py -c -i 10.0.1.2 -p 8080 -f
iceland-safiqul.jpg -w 5
YN packet is sent
imeout waiting for SYN-ACK, retrying...
oot@debian:/home/debian/src# 
```

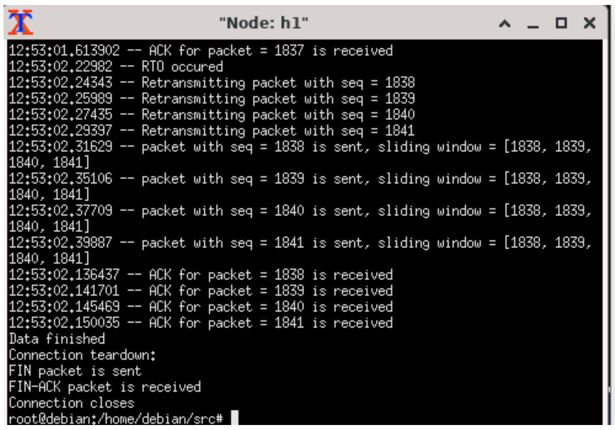


```
root@debian:/home/debian/src# python3 application.py -s -i 10.0.1.2 -p 8080
Server listening on 10.0.1.2:8080
Handshake failed: No SYN received
Connection establishment failed
Connection closes
root@debian:/home/debian/src# python3 application.py -s -i 10.0.1.2 -p 8080
Server listening on 10.0.1.2:8080
Handshake failed: No SYN received
Connection establishment failed
Connection closes
root@debian:/home/debian/src# 
```

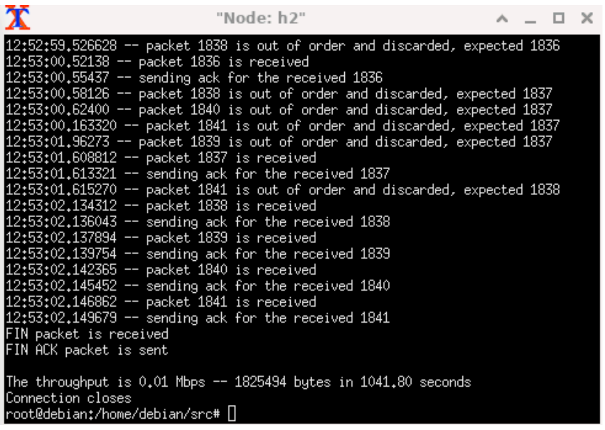
Even with improved error handling, the client did not always have enough time to retransmit the SYN before the server timed out. I therefore had to increase the server's total handshake timeout to 30 seconds, which provided enough time for the client to retransmit the SYN and complete the handshake (in all my attempts).

A limitation I discovered in the handshake is that if the client's final ACK packet is lost, the server never completes the handshake. Meanwhile, the client assumes the handshake succeeded and begins data transfer. The server then remains stuck in the handshake, printing "unexpected packet" errors while receiving data packets. This could be resolved by adding an additional "dummy" confirmation, such as a small "ack received" packet. However, since this assignment is asking for a basic three-way handshake, I have not implemented this fix. The handshake will therefore not be bulletproof, but it worked in most of my attempts after adding the error handling.

When it makes it past the handshake, I found that connection teardown also failed, because the FIN and/or FIN-ACK packets were lost as well. By adding more error handling and retry logic to the teardown as well, the transfer eventually completes successfully, but with many lost packets and retransmissions, and using a total transfer time of about 17 minutes.



```
12:53:01.613902 -- ACK for packet = 1837 is received
12:53:02.22982 -- RTO occured
12:53:02.24343 -- Retransmitting packet with seq = 1838
12:53:02.25989 -- Retransmitting packet with seq = 1839
12:53:02.27435 -- Retransmitting packet with seq = 1840
12:53:02.29397 -- Retransmitting packet with seq = 1841
12:53:02.31629 -- packet with seq = 1838 is sent, sliding window = [1838, 1839,
1840, 1841]
12:53:02.35106 -- packet with seq = 1839 is sent, sliding window = [1838, 1839,
1840, 1841]
12:53:02.37709 -- packet with seq = 1840 is sent, sliding window = [1838, 1839,
1840, 1841]
12:53:02.39887 -- packet with seq = 1841 is sent, sliding window = [1838, 1839,
1840, 1841]
12:53:02.136437 -- ACK for packet = 1838 is received
12:53:02.141701 -- ACK for packet = 1839 is received
12:53:02.145469 -- ACK for packet = 1840 is received
12:53:02.150035 -- ACK for packet = 1841 is received
Data finished
Connection teardown:
FIN packet is sent
FIN-ACK packet is received
Connection closes
root@debian:/home/debian/src# 
```



```
12:52:59.526628 -- packet 1838 is out of order and discarded, expected 1836
12:53:00.52138 -- packet 1836 is received
12:53:00.55437 -- sending ack for the received 1836
12:53:00.58126 -- packet 1838 is out of order and discarded, expected 1837
12:53:00.62400 -- packet 1840 is out of order and discarded, expected 1837
12:53:00.163320 -- packet 1841 is out of order and discarded, expected 1837
12:53:01.96273 -- packet 1839 is out of order and discarded, expected 1837
12:53:01.608812 -- packet 1837 is received
12:53:01.613321 -- sending ack for the received 1837
12:53:01.615270 -- packet 1841 is out of order and discarded, expected 1838
12:53:02.134312 -- packet 1838 is received
12:53:02.136043 -- sending ack for the received 1838
12:53:02.137894 -- packet 1839 is received
12:53:02.139754 -- sending ack for the received 1839
12:53:02.142365 -- packet 1840 is received
12:53:02.145452 -- sending ack for the received 1840
12:53:02.146862 -- packet 1841 is received
12:53:02.149679 -- sending ack for the received 1841
FIN packet is received
FIN ACK packet is sent

The throughput is 0.01 Mbps -- 1825494 bytes in 1041.80 seconds
Connection closes
root@debian:/home/debian/src# 
```

The throughput was 0.01 Mbps, and the transfer took approximately 17.5 minutes.

Discussion #5

If the FIN-ACK gets lost, the client will be stuck waiting at *sock.recvfrom(8).* The server will think the connection is closed (after sending FIN-ACK), but the client will be stuck hanging, waiting forever (or until killed) for the FIN-ACK that will never arrive. This leaves a half-open connection, it will not be closed gracefully.

With the added error handling and timeout, the client will wait for 3 seconds before resending the FIN. This repeats up to *max_retries* (default set to 5). After this, it exits the loop and close the socket anyway. It fails gracefully – even if the connection did not close correctly, it avoids hanging forever, logs the failure and close the socket.

References

Islam, Safiqul. (07.02.2025). *The transport layer* (lecture).

Islam, Safiqul. (2023). *header.py* [Source code]. GitHub. https://github.com/safiqul/2410/blob/main/header/header.py

Kurose, James F. & Ross, Keith W. (2022). *Computer networking: A top-down approach* (8th edition, global edition). Pearson Education.

Python Documentation. (2025a). *argparse — Parser for command-line options, arguments and subcommands*. https://docs.python.org/3/library/argparse.html

Python Documentation (2025b). *ipaddress — IPv4/IPv6 manipulation library.* https://docs.python.org/3/library/ipaddress.html