DRTP PROJECT REPORT

Individual Exam DATA2410 2025

Summary

Report detailing my implementation of the DATA2410 Reliable Transport Protocol

Candidate Number 330

Introduction

This report details my implementation of the DATA2410 Reliable Transport Protocol (DRTP) and answers to the discussions, both provided in the assignment description (Islam, 2025) DRTP is a reliable transport protocol built on top of UDP, to provide reliability while still using UDP, much like QUIC is doing. The assignment limits the protocol to only require handling of a single, simple sender and receiver application.

The DRTP achieves reliability by setting up a connection between the sender and receiver and using a sliding window for data control flow. To achieve reliability in the data transfer, the DRTP implements a Go Back N (GBN) sliding window, where packets sent must be acknowledged by receiver before sending any more packets. These data packets and acknowledgements are sent over the connection created, and should a packet be missing or arrive out-of-order, the sender will be missing an acknowledgement for said packet. The sender waits for a default time of 400ms, before assuming the packet was lost and resending every packet in the window, because any out-of-order packet will be discarded by the receiver.

UDP is connection-less, meaning it does not set up a connection between sender and receiver. To allow reliable transfer of data, DRTP sets up a connection much like how TCP does, by initiating a three-way-handshake to create, and another handshake for connection teardown. The DRTP connection differs from TCP by not having a three-way-handshake for the teardown with FIN, FIN-ACK and last ACK, but instead using FIN and FIN-ACK.

To challenge myself, I have implemented the DRTP using a Finite State Machine much like TCP. This is not specified in the assignment description and is not a requirement.

Implementation

My implementation is based on course material, previous obligatory assignments and online resources (see references). The implementation is presented in four different parts, Finite State Machine, how the reliable connection is established and tore down, how data is reliably transported, and at last supportive features. The Finite State Machine is described first because it is integral to the design and structure, and understanding the rest of the implementation becomes harder without understanding its implementation.

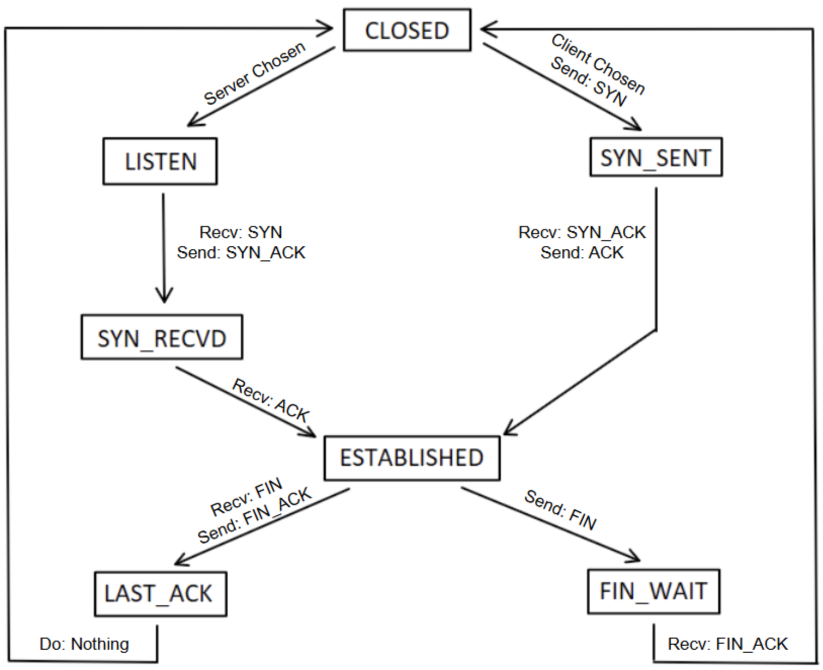
Finite State Machine

I chose to implement the protocol using a Finite State Machine (FSM) because it adds a stable and easily debugged structure where what functionality should be executed at certain moments and circumstances is easily defined. It is also how TCP is implemented, and the naming and general function of each DRTP-state is inspired by TCP's states (IBM, without date)

The Finite State Machine I have implemented is an object based Mealy Machine (Wikipedia, 2023). I originally built for a personal game development project based on a creator's guide (The Shaggy Dev, 2023). To fit the requirements of the DRTP I have made necessary modifications to the code and rewritten the code in Python (from Godot's GDScript).

For the FSM to work, all the applications functionality is divided into states, like the state LISTEN where the functionality of listening for incoming SYN-packets is written. The FSM works by holding an instance of the current state and passing function calls down to execute its code. The FSM changes what state the current state is when the current state returns a reference to one of the other state-instances, meaning that the state transfers are decided by the code written in the states, but handled by the FSM.

Overview of state flow:



Each state described:

- **CLOSED**: Initial state which proceeds to LISTEN if server and SYN_SENT if client. It is also the ending state which closes the connection
- **LISTEN**: Listens for connection requests. Proceeds to SYN_RECVD when a SYN-packet is received.
- **SYN_RECVD:** Sends a SYN_ACK-packet and awaits an ACK-packet before proceeding to ESTABLISHED
- **SYN_SENT**: Sends a SYN-packet and awaits a SYN_ACK-packet before proceeding to ESTABLISHED where the last ACK-packet is sent for the connection to be made.
- **ESTABLISHED**: Handles reliable data transfer. If sender, state sends ACK-packet to receiver as last part of initial handshake. Proceeds to FIN_WAIT when sender has gotten all data acknowledged, or LAST_ACK if receiver has received a FIN-packet.
- **FIN_WAIT**: Send FIN-packet to receiver. Await FIN_ACK-packet before proceeding to CLOSED.
- **LAST_ACK**: Send FIN_ACK-packet before proceeding to CLOSED.

For readability, the files which contain each state is named in all capital letters, which deviates from common practice.

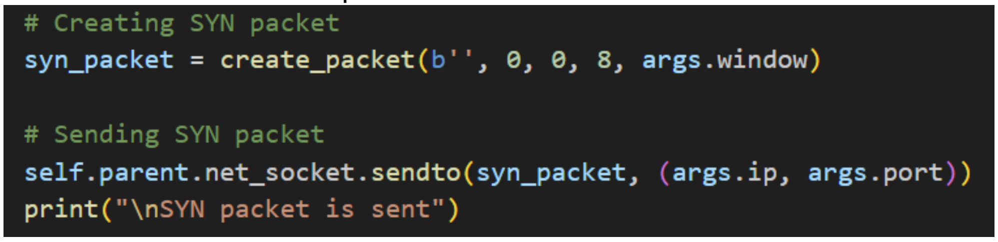
Reliable connection

To create a reliable connection the application needs to ensure that both sender and receiver are ready for data transfer before sending any data. And when data is no longer being sent, both sender and receiver knows and does not expect any data sent or received. This is implemented through handshakes.

Initial three-way-handshake.

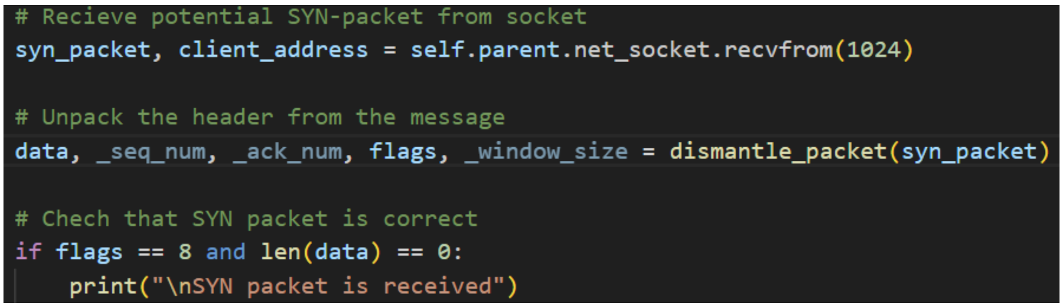
To ensure that both sender and receiver are ready for data transfer, I have implemented a three-way-handshake which begins with:

The sender sends a SYN-packet.



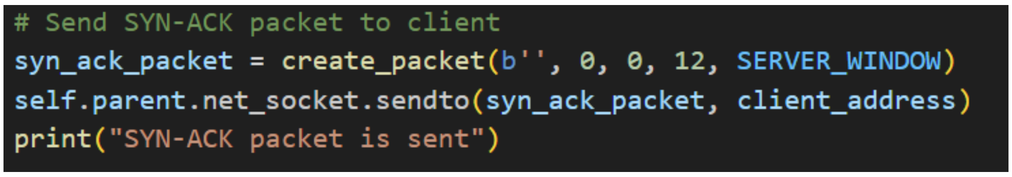
```
# Creating SYN packet
syn_packet = create_packet(b'', 0, 0, 8, args.window)

# Sending SYN packet
self.parent.net_socket.sendto(syn_packet, (args.ip, args.port))
print("\nSYN packet is sent")
```

The receiver receives and validates the packet.



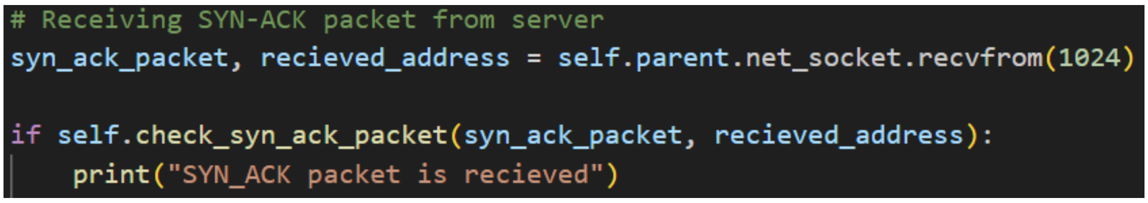
```
# Recieve potential SYN-packet from socket
syn_packet, client_address = self.parent.net_socket.recvfrom(1024)

# Unpack the header from the message
data, _seq_num, _ack_num, flags, _window_size = dismantle_packet(syn_packet)

# Chech that SYN packet is correct
if flags == 8 and len(data) == 0:
    print("\nSYN packet is received")
```

If packet is a valid SYN-packet, receiver returns a SYN_ACK-packet.



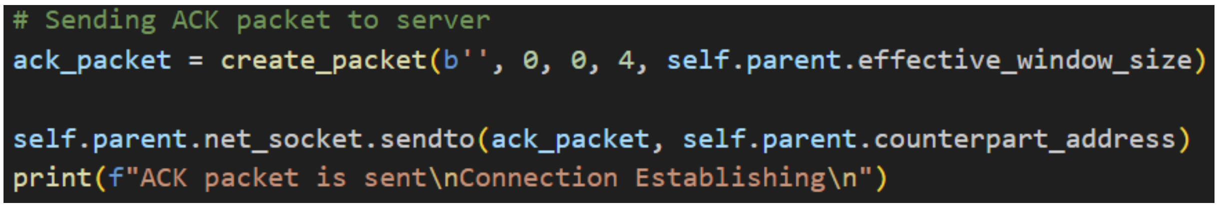
```
# Send SYN-ACK packet to client
syn_ack_packet = create_packet(b'', 0, 0, 12, SERVER_WINDOW)
self.parent.net_socket.sendto(syn_ack_packet, client_address)
print("SYN-ACK packet is sent")
```

Which the sender validates.



```
# Receiving SYN-ACK packet from server
syn_ack_packet, recieved_address = self.parent.net_socket.recvfrom(1024)

if self.check_syn_ack_packet(syn_ack_packet, recieved_address):
    print("SYN_ACK packet is recieved")
```

Before returning the last ACK of the handshake to the receiver.



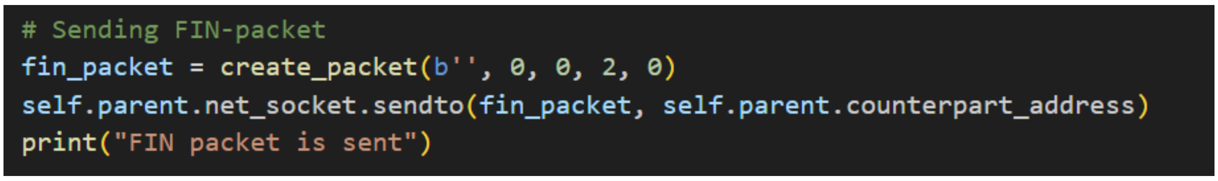
```
# Sending ACK packet to server
ack_packet = create_packet(b'', 0, 0, 4, self.parent.effective_window_size)

self.parent.net_socket.sendto(ack_packet, self.parent.counterpart_address)
print(f"ACK packet is sent\nConnection Establishing\n")
```

If at any point any of these packets are delayed or lost, the application will wait for a default of 5 timeouts before proceeding to the CLOSED-state and close down the connection. The exception to this is the final ACK of the handshake, which will be resent along with the first window if sender has not received any acknowledgements for their sent packets, indicating that the last ACK was not received, and the connection is not established.

Final two-way-handshake

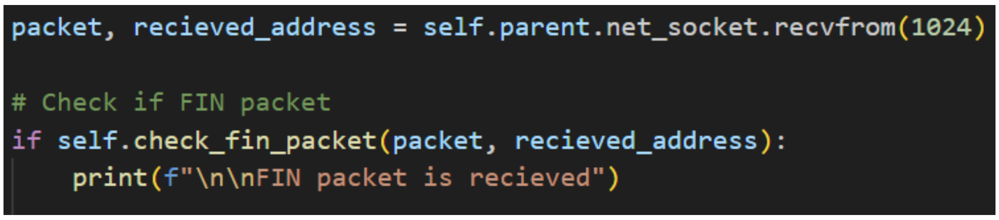
When all data has been transferred, a final two-way-handshake is initiated in order to make sure that the receiver knows that there is no more data to receive, and the sender knows the receiver got the knows. This handshake begins with:

The sender sends a FIN-packet to signal the start of the handshake.



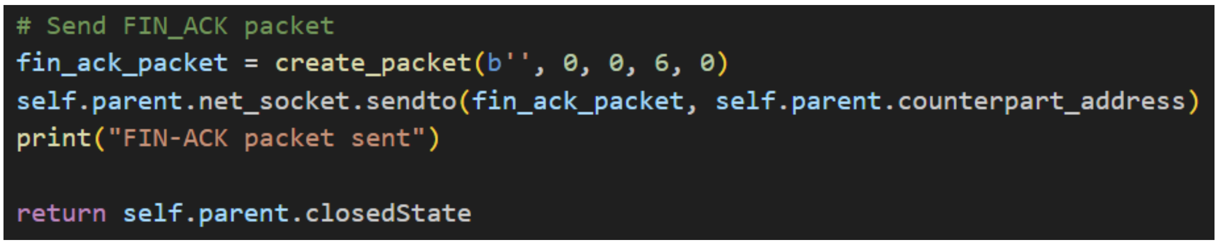
```
# Sending FIN-packet
fin_packet = create_packet(b'', 0, 0, 2, 0)
self.parent.net_socket.sendto(fin_packet, self.parent.counterpart_address)
print("FIN packet is sent")
```

The receiver receives and validates the FIN-packet.



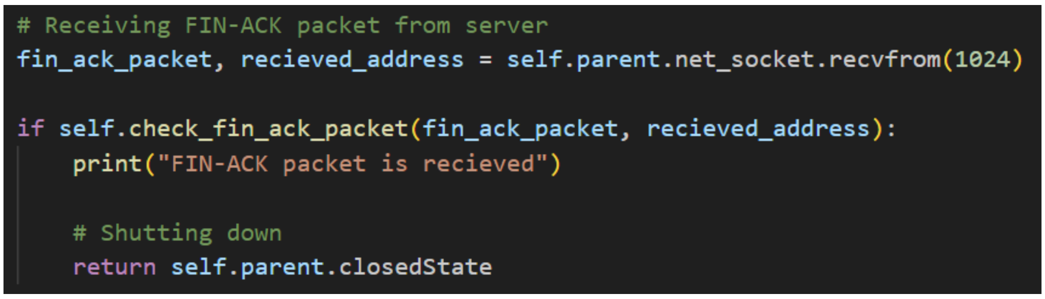
```
packet, recieved_address = self.parent.net_socket.recvfrom(1024)

# Check if FIN packet
if self.check_fin_packet(packet, recieved_address):
    print(f"\n\nFIN packet is recieved")
```

Before the receiver sends a FIN_ACK-packet in return and closing down.



```python
# Send FIN_ACK packet
fin_ack_packet = create_packet(b'', 0, 0, 6, 0)
self.parent.net_socket.sendto(fin_ack_packet, self.parent.counterpart_address)
print("FIN-ACK packet sent")

return self.parent.closedState
```

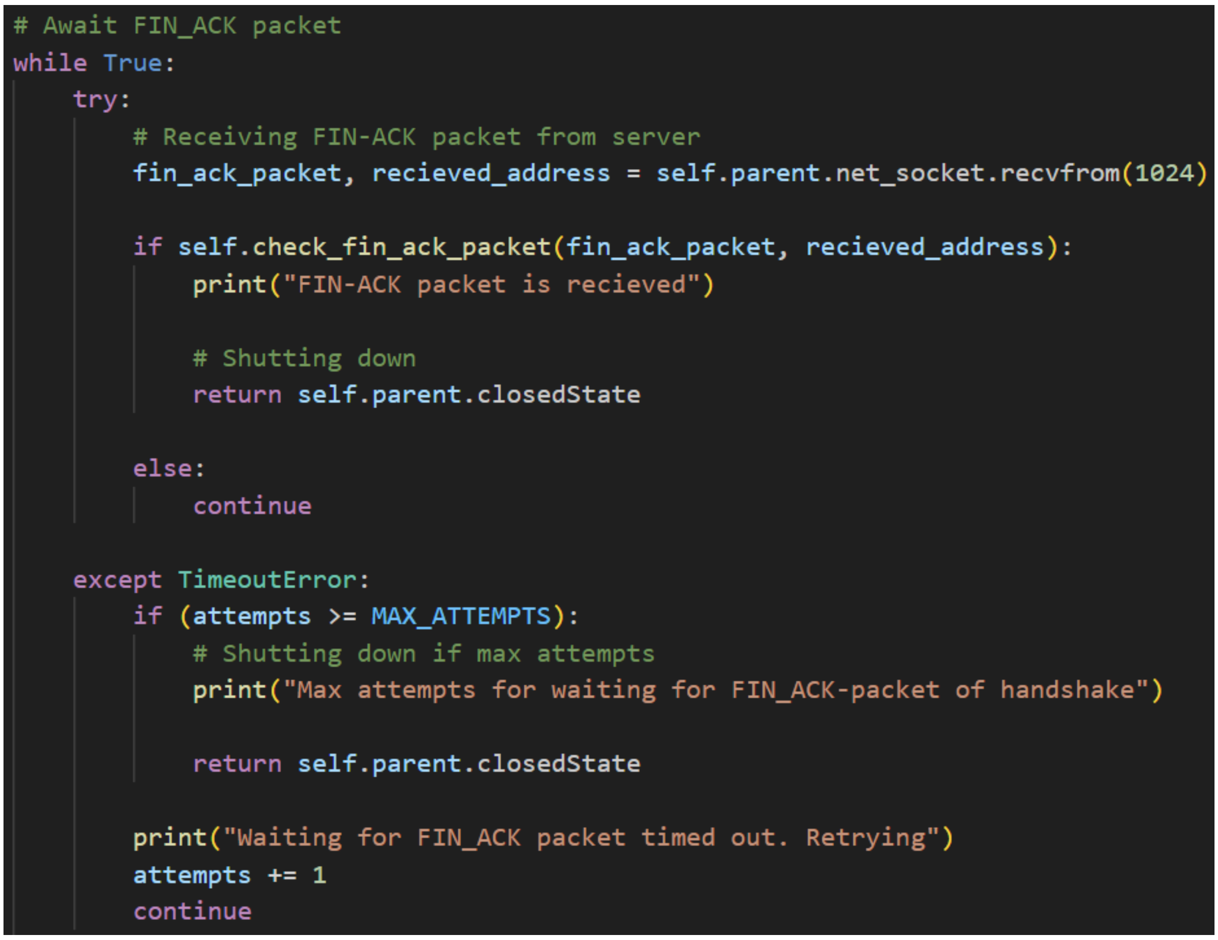
The sender waits for the FIN_ACK-packet before closing down.



```python
# Receiving FIN-ACK packet from server
fin_ack_packet, recieved_address = self.parent.net_socket.recvfrom(1024)

if self.check_fin_ack_packet(fin_ack_packet, recieved_address):
    print("FIN-ACK packet is recieved")

    # Shutting down
    return self.parent.closedState
```

Timeout handling

For every instance where the packet is waiting to receive a packet a timeout will result in a raised exception, which my code will allow for a default of five times before changing to the CLOSED state and closing the connection, as seen in the picture below where the FIN_ACK packet is awaited.



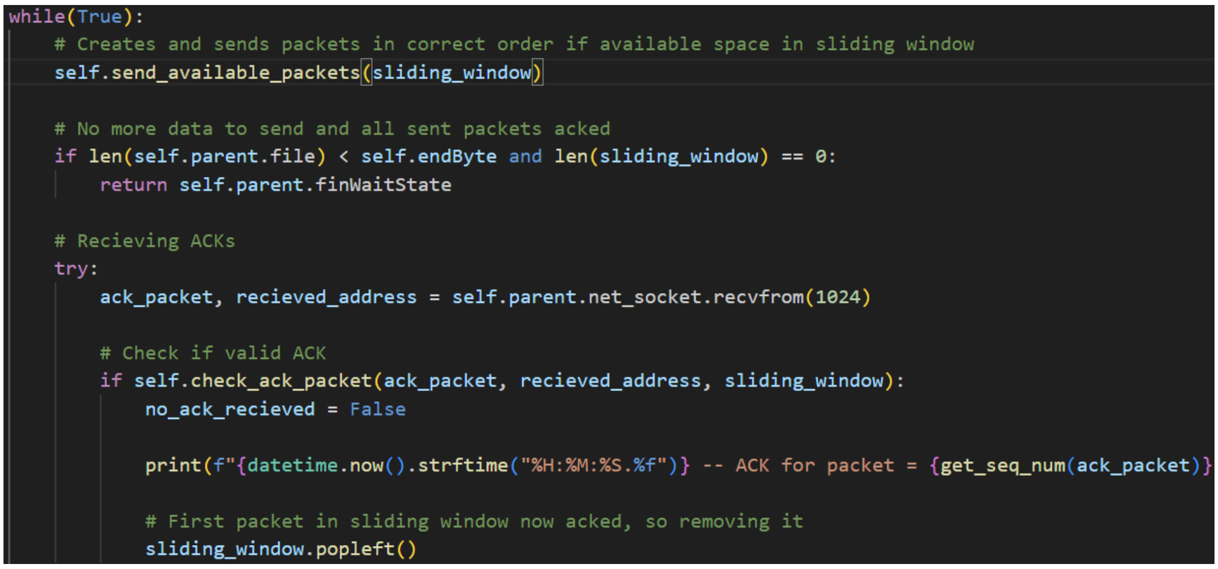
```python
# Await FIN_ACK packet
while True:
    try:
        # Receiving FIN-ACK packet from server
        fin_ack_packet, recieved_address = self.parent.net_socket.recvfrom(1024)

        if self.check_fin_ack_packet(fin_ack_packet, recieved_address):
            print("FIN-ACK packet is recieved")

            # Shutting down
            return self.parent.closedState

        else:
            continue

    except TimeoutError:
        if (attempts >= MAX_ATTEMPTS):
            # Shutting down if max attempts
            print("Max attempts for waiting for FIN_ACK-packet of handshake")

            return self.parent.closedState

        print("Waiting for FIN_ACK packet timed out. Retrying")
        attempts += 1
        continue
```

Reliable data transfer

To transport data reliably, my implementation uses a Go Back N (GBN) sliding window on top of the reliable connection. For the sender to be sure that the receiver has received a packet, the sender keeps note of what packets have been sent and waits for an ACK-packet containing the acknowledge number of the next packet, the sender then sends the packet which the ACK-packet “requested” through its acknowledge number. To keep more than one packet in flight, the GBN sliding window is used. The window always times the arrival of the acknowledgement for the first packet in the window. If the time exceeds the timeout period, the GBN sliding window resends all packets in the window. This is because on the receiving side, any out-of-order packet is discarded, meaning that if a packet is lost, all other packets after must be assumed discarded.

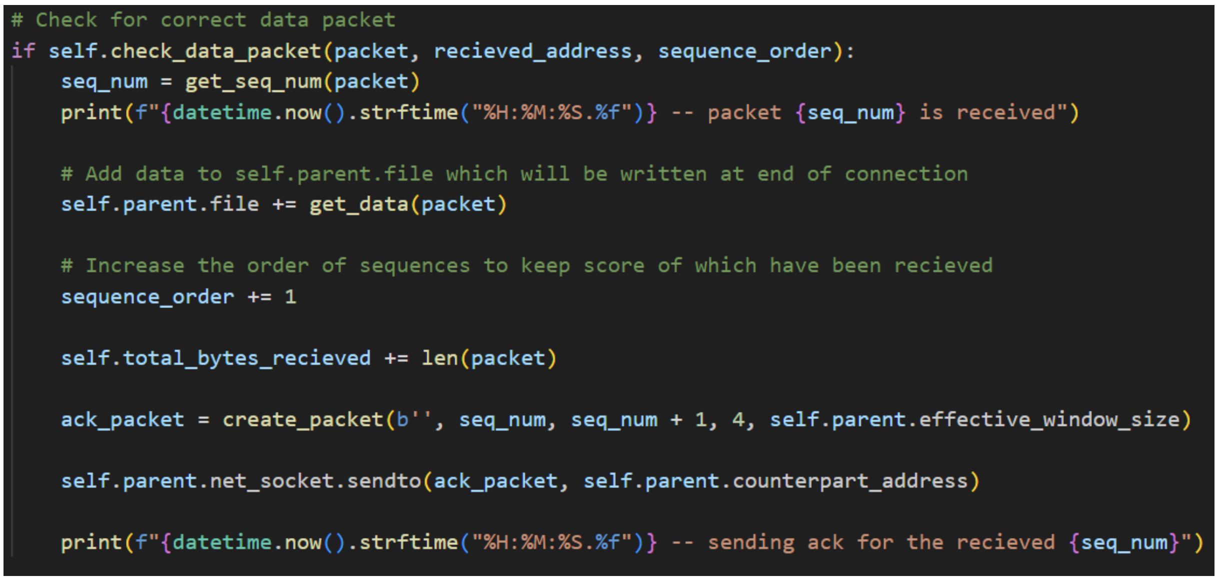
In my implementation I use a double ended queue (deque) for the sliding window. When a packet is sent, the entire packet is appended in the deque. This allows the application to easily append and pop packets and always keep the order of their appending. In addition, retransmitting all packets is as easy as looping over the deque and resending them.

The senders code where packets are sent if there is space in the window, file data is extracted and added to a packet if a valid ACK-packet (which also checks correct order) is received and is sent. And then the packet which was acknowledged is popped from the window:



```
while(True):
    # Creates and sends packets in correct order if available space in sliding window
    self.send_available_packets(sliding_window)

    # No more data to send and all sent packets acked
    if len(self.parent.file) < self.endByte and len(sliding_window) == 0:
        return self.parent.finWaitState

    # Recieving ACKs
    try:
        ack_packet, recieved_address = self.parent.net_socket.recvfrom(1024)

        # Check if valid ACK
        if self.check_ack_packet(ack_packet, recieved_address, sliding_window):
            no_ack_recieved = False

            print(f"{datetime.now().strftime("%H:%M:%S.%f")} -- ACK for packet = {get_seq_num(ack_packet)}

            # First packet in sliding window now acked, so removing it
            sliding_window.popleft()
```

The receivers code where a valid packet has its data extracted and an ACK-packet is sent in return:



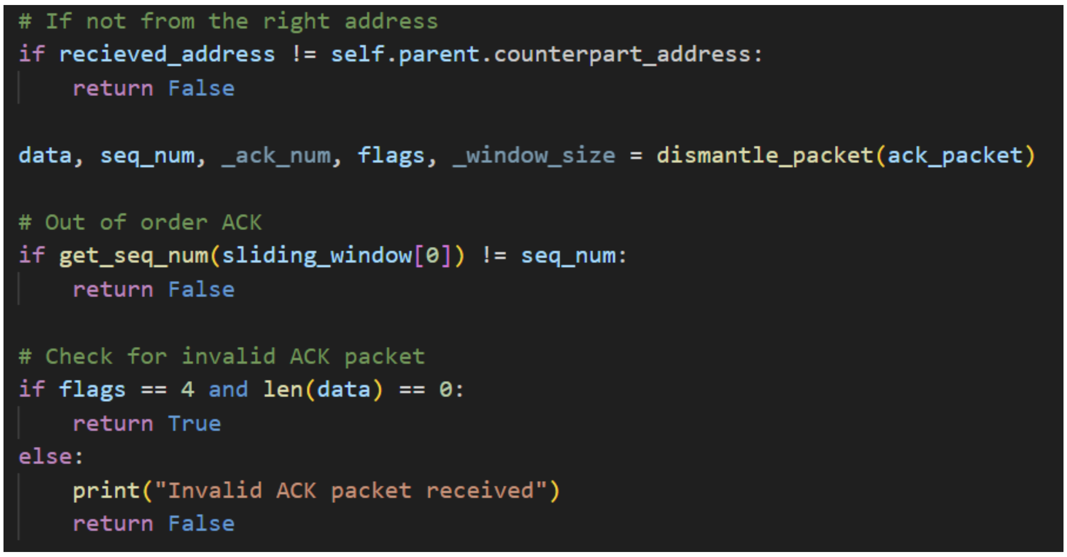
```python
# Check for correct data packet
if self.check_data_packet(packet, recieved_address, sequence_order):
    seq_num = get_seq_num(packet)
    print(f"{datetime.now().strftime("%H:%M:%S.%f")} -- packet {seq_num} is received")

    # Add data to self.parent.file which will be written at end of connection
    self.parent.file += get_data(packet)

    # Increase the order of sequences to keep score of which have been recieved
    sequence_order += 1

    self.total_bytes_recieved += len(packet)

    ack_packet = create_packet(b'', seq_num, seq_num + 1, 4, self.parent.effective_window_size)

    self.parent.net_socket.sendto(ack_packet, self.parent.counterpart_address)

    print(f"{datetime.now().strftime("%H:%M:%S.%f")} -- sending ack for the recieved {seq_num}")
```

Supporting features

Packet validating

All packets received by either receiver or sender is validated so that packets from the wrong address, packets with the wrong flag or packets which do not otherwise fit are ignored.

For example, here is the validation of ACK-packets acknowledging data-packets:



```python
# If not from the right address
if recieved_address != self.parent.counterpart_address:
    return False

data, seq_num, _ack_num, flags, _window_size = dismantle_packet(ack_packet)

# Out of order ACK
if get_seq_num(sliding_window[0]) != seq_num:
    return False

# Check for invalid ACK packet
if flags == 4 and len(data) == 0:
    return True
else:
    print("Invalid ACK packet received")
    return False
```

File reading and writing.

To read a file, the file to send is broken down into bytes so it can be added to a packet. To write, the accumulated bytes on the receiver side is written to the current directory by default.

DRTP packet creation and header parsing.

Functions to create a packet with file data in bytes and add the DRTP header of size 8 and dismantle a packet to extract the file data and header data. The header parser, flag parser and create packet function is sourced from a provided resource (Islam, 2023).

Argument parser.

To determine the application's specific function, arguments are added in the command line interface when running the command. These were specified in the assignment description (Islam, 2025).

Flag	Long flag	Input	Default	Type	Description
-s	--server	None	None	Boolean	Enable server mode (receiver)
-c	--client	None	None	Boolean	Enable client mode (sender)
-i	--ip	Ip address	127.0.0.1	String	Choose the ip address for receiver to listen on, or the ip address for the sender to send to.
-p	--port	Port number	8080	Integer	Choose the port for receiver to listen on, or the port for the sender to send to.
-f	--file	File path	None	String	Write the name of the file to send (ignored by server)
-w	--window	Size	3	Integer	Window size for senders sliding window. (Server is hardcoded to tell the client that max is 15)
-d	--discard	Packet number	1000000	Integer	Discard the packet with sequence number equal to the provided integer. (ignored by client)

Discussions

Answers to the discussion question portion of the assignment description.

1. Throughput for RTT = 100ms

Application is tested with different window sizes and resulting throughputs are recorded. Because of high variance in the resulting throughput, the recorded result is the best of a couple of tries. I think the variance is caused by hardware and inconsistent VM overhead. For the maximum possible throughput, I used the formula *(Mb * window_size) / RTT* where *Mb* is Megabit per packet, and *RTT*, the smallest ideal time (converted to seconds), is the dividend.

Window size	Throughput from testing in Mbps	Maximum possible throughput in Mbps
3	0,18	0,24
5	0,23	0,4
10	0,58	0,8
15	0,95	1,2
20	0,93	1,2
25	0,95	1,2

Here we can see that increasing the window size also increases the throughput. The sliding window controls how many packets can be “in flight” at one time, meaning that the higher the window size, the more packets are being sent at the same time. This results in a higher amount of data being received by the server, and the throughput increases, like in the formula.

The size of the sending window is set by taking the lowest of the window argument provided to the client, and the highest possible window the server can serve, which is set to 15 for DRTP. This means that the maximum size possible is 15, and the throughput for 20 and 25 will be the same as for 15.

2. Throughput for RTT = 50ms and 200ms

For both RTTs, throughput is recorded and calculated the same way as with RTT of 100ms (see *1. Throughput for RTT = 100ms* above).

RTT = 50ms

Window size	**Throughput from testing in Mbps**	**Maximum possible throughput in Mbps**
3	0,39	0,48
5	0,64	0,8
10	1,27	1,6
15	1,91	2,4
20	1,85	2,4
25	1,82	2,4

Here we can see the same phenomenon as previously, where the increased window size results in increasing throughput, which correlates to the *window_size* factor in the throughput formula. But we also see that all the ideal throughputs are double the size, and the recorded throughputs are about double the size. This is because the RTT, which is the dividend in the equation, is decreased to half (100ms / 2 = 50ms). And logically it makes sense because a single packet takes less time to arrive.

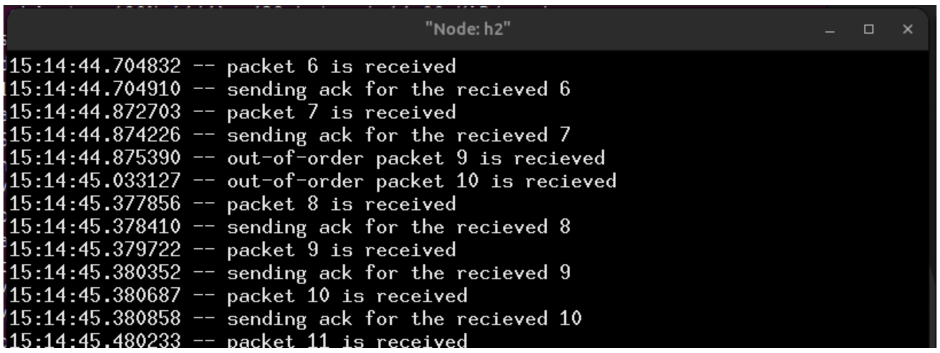
RTT = 200ms

Window size	**Throughput from testing in Mbps**	**Maximum possible throughput in Mbps**
3	0,10	0,12
5	0,16	0,2
10	0,34	0,4
15	0,53	0,6
20	0,52	0,6
25	0,48	0,6

The same applies for an RTT of 200ms, but the opposite way. Instead of increasing the throughput by a factor of 2, the throughput has decreased by a factor of 2. Again, we can see this from the equation, where RTT, the dividend is increased by a factor of 2. And once again, it makes sense logically because each packet takes double the time, and the number of packets in the window will take longer to reach the receiver.

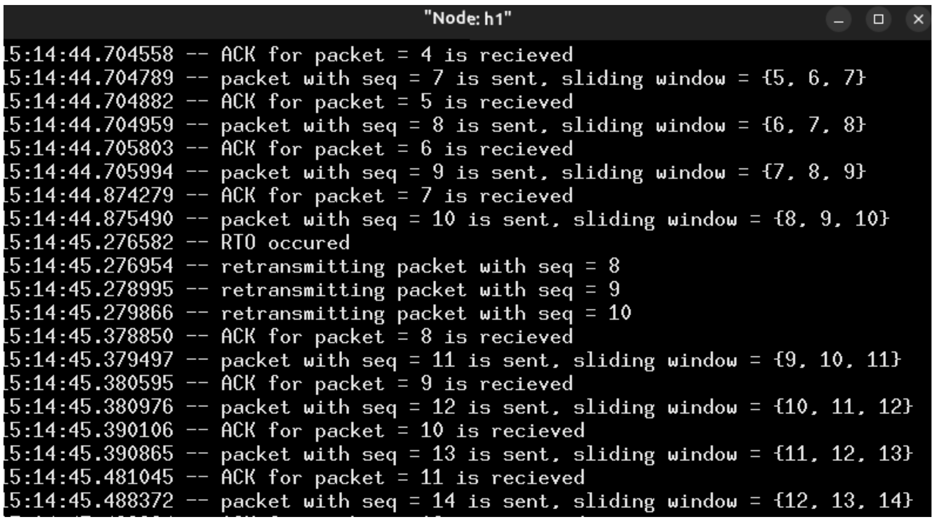
3. Discard

With the -w 8 option my application dropped packet nr. 8 in the run of the following screenshots:



"Node: h2"

```
15:14:44.704832 -- packet 6 is received
15:14:44.704910 -- sending ack for the recieved 6
15:14:44.872703 -- packet 7 is received
15:14:44.874226 -- sending ack for the recieved 7
15:14:44.875390 -- out-of-order packet 9 is recieved
15:14:45.033127 -- out-of-order packet 10 is recieved
15:14:45.377856 -- packet 8 is received
15:14:45.378410 -- sending ack for the recieved 8
15:14:45.379722 -- packet 9 is received
15:14:45.380352 -- sending ack for the recieved 9
15:14:45.380687 -- packet 10 is received
15:14:45.380858 -- sending ack for the recieved 10
15:14:45.480233 -- packet 11 is received
```



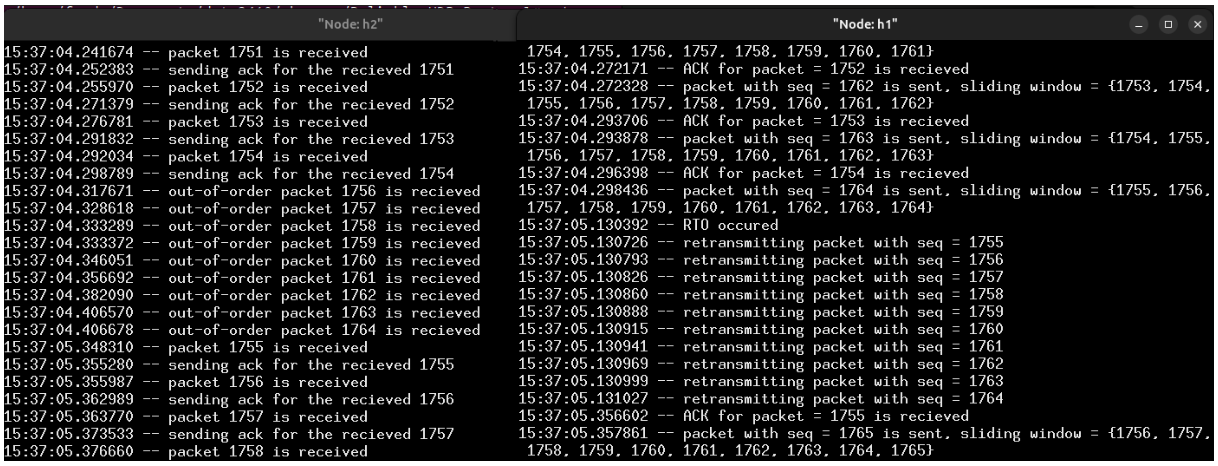
"Node: h1"

```
5:14:44.704558 -- ACK for packet = 4 is recieved
5:14:44.704789 -- packet with seq = 7 is sent, sliding window = {5, 6, 7}
5:14:44.704882 -- ACK for packet = 5 is recieved
5:14:44.704959 -- packet with seq = 8 is sent, sliding window = {6, 7, 8}
5:14:44.705803 -- ACK for packet = 6 is recieved
5:14:44.705994 -- packet with seq = 9 is sent, sliding window = {7, 8, 9}
5:14:44.874279 -- ACK for packet = 7 is recieved
5:14:44.875490 -- packet with seq = 10 is sent, sliding window = {8, 9, 10}
5:14:45.276582 -- RTO occured
5:14:45.276954 -- retransmitting packet with seq = 8
5:14:45.278995 -- retransmitting packet with seq = 9
5:14:45.279866 -- retransmitting packet with seq = 10
5:14:45.378850 -- ACK for packet = 8 is recieved
5:14:45.379497 -- packet with seq = 11 is sent, sliding window = {9, 10, 11}
5:14:45.380595 -- ACK for packet = 9 is recieved
5:14:45.380976 -- packet with seq = 12 is sent, sliding window = {10, 11, 12}
5:14:45.390106 -- ACK for packet = 10 is recieved
5:14:45.390865 -- packet with seq = 13 is sent, sliding window = {11, 12, 13}
5:14:45.481045 -- ACK for packet = 11 is recieved
5:14:45.488372 -- packet with seq = 14 is sent, sliding window = {12, 13, 14}
```

Here we can see the server h2 not fully receiving packet 8 and printing that received packet nr.9 and 10 are out-of-order. On the client h1 we can see that an RTO occurs where the ACK for packet 8 is expected. The client then retransmits every packet in the window. The ACK for packet 8 then arrives and the next packet is sent. Over at the server h2 we can se that packet 8 is received after the out-of-order packets have been received. Under *Implementation – Reliable data transfer* I have more details of how my code deals with a discarded packet.

4. Packet Loss

Loss rate set to 2%

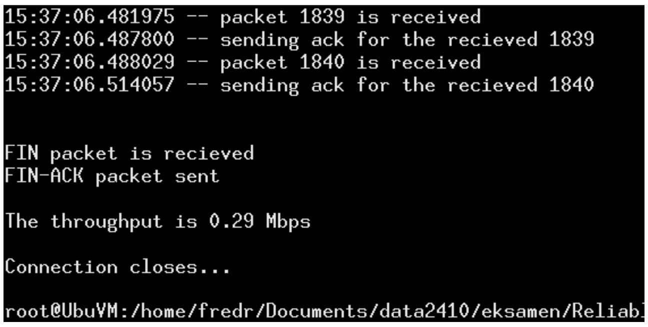


"Node: h2"

```
15:37:04.241674 -- packet 1751 is received
15:37:04.252383 -- sending ack for the recieved 1751
15:37:04.255970 -- packet 1752 is received
15:37:04.271379 -- sending ack for the recieved 1752
15:37:04.276781 -- packet 1753 is received
15:37:04.291832 -- sending ack for the recieved 1753
15:37:04.292034 -- packet 1754 is received
15:37:04.298789 -- sending ack for the recieved 1754
15:37:04.317671 -- out-of-order packet 1756 is recieved
15:37:04.328618 -- out-of-order packet 1757 is recieved
15:37:04.333289 -- out-of-order packet 1758 is recieved
15:37:04.333372 -- out-of-order packet 1759 is recieved
15:37:04.346051 -- out-of-order packet 1760 is recieved
15:37:04.356692 -- out-of-order packet 1761 is recieved
15:37:04.382090 -- out-of-order packet 1762 is recieved
15:37:04.406570 -- out-of-order packet 1763 is recieved
15:37:04.406678 -- out-of-order packet 1764 is recieved
15:37:05.348310 -- packet 1755 is received
15:37:05.355280 -- sending ack for the recieved 1755
15:37:05.355987 -- packet 1756 is received
15:37:05.362989 -- sending ack for the recieved 1756
15:37:05.363770 -- packet 1757 is received
15:37:05.373533 -- sending ack for the recieved 1757
15:37:05.376660 -- packet 1758 is received
```

"Node: h1"

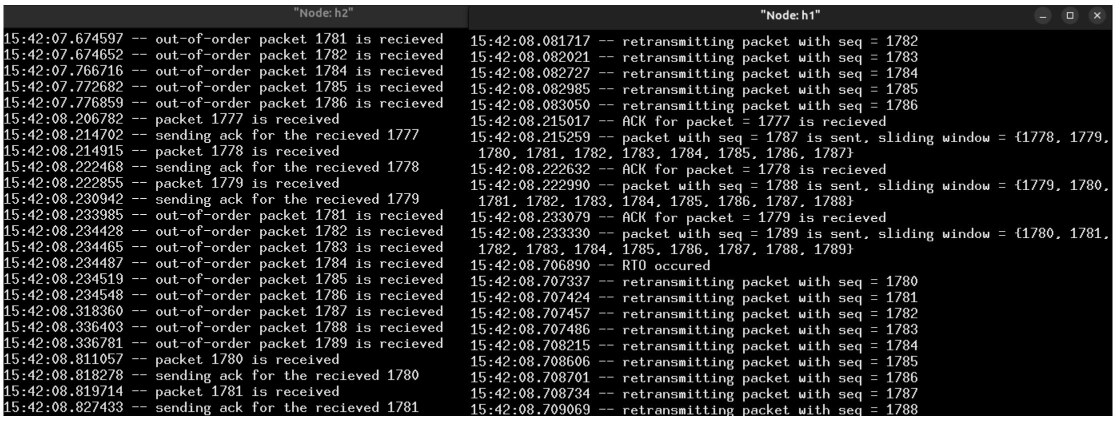
```
 1754, 1755, 1756, 1757, 1758, 1759, 1760, 1761}
15:37:04.272171 -- ACK for packet = 1752 is recieved
15:37:04.272328 -- packet with seq = 1762 is sent, sliding window = {1753, 1754,
 1755, 1756, 1757, 1758, 1759, 1760, 1761, 1762}
15:37:04.293706 -- ACK for packet = 1753 is recieved
15:37:04.293878 -- packet with seq = 1763 is sent, sliding window = {1754, 1755,
 1756, 1757, 1758, 1759, 1760, 1761, 1762, 1763}
15:37:04.296398 -- ACK for packet = 1754 is recieved
15:37:04.298436 -- packet with seq = 1764 is sent, sliding window = {1755, 1756,
 1757, 1758, 1759, 1760, 1761, 1762, 1763, 1764}
15:37:05.130392 -- RTO occured
15:37:05.130726 -- retransmitting packet with seq = 1755
15:37:05.130793 -- retransmitting packet with seq = 1756
15:37:05.130826 -- retransmitting packet with seq = 1757
15:37:05.130860 -- retransmitting packet with seq = 1758
15:37:05.130888 -- retransmitting packet with seq = 1759
15:37:05.130915 -- retransmitting packet with seq = 1760
15:37:05.130941 -- retransmitting packet with seq = 1761
15:37:05.130969 -- retransmitting packet with seq = 1762
15:37:05.130999 -- retransmitting packet with seq = 1763
15:37:05.131027 -- retransmitting packet with seq = 1764
15:37:05.356602 -- ACK for packet = 1755 is recieved
15:37:05.357861 -- packet with seq = 1765 is sent, sliding window = {1756, 1757,
 1758, 1759, 1760, 1761, 1762, 1763, 1764, 1765}
```



```
15:37:06.481975 -- packet 1839 is received
15:37:06.487800 -- sending ack for the recieved 1839
15:37:06.488029 -- packet 1840 is received
15:37:06.514057 -- sending ack for the recieved 1840


FIN packet is recieved
FIN-ACK packet sent

The throughput is 0.29 Mbps

Connection closes...

root@UbuVM:/home/fredr/Documents/data2410/eksamen/Reliab
```

With a window of 10, throughput fell to 0,29 Mbps from the previous 0,58. For every loss, there is at least 400ms added for waiting on an ACK for the lost packet, and then time for resending packets. This results in fewer processed packets per second and therefore lower throughput.

Loss rate set to 5%

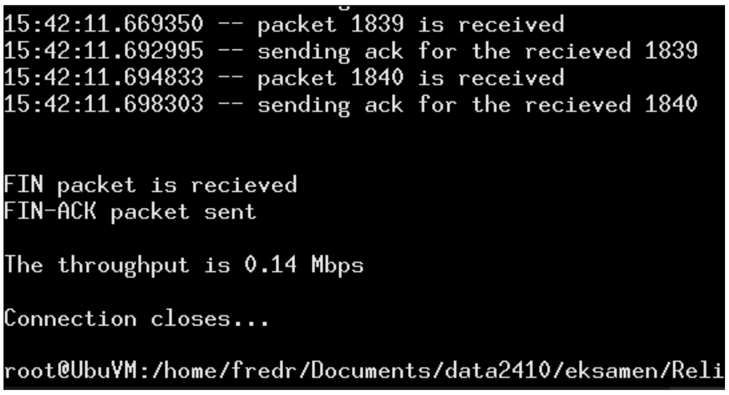


"Node: h2"

```
15:42:07.674597 -- out-of-order packet 1781 is recieved
15:42:07.674652 -- out-of-order packet 1782 is recieved
15:42:07.766716 -- out-of-order packet 1784 is recieved
15:42:07.772682 -- out-of-order packet 1785 is recieved
15:42:07.776859 -- out-of-order packet 1786 is recieved
15:42:08.206782 -- packet 1777 is received
15:42:08.214702 -- sending ack for the recieved 1777
15:42:08.214915 -- packet 1778 is received
15:42:08.222468 -- sending ack for the recieved 1778
15:42:08.222855 -- packet 1779 is received
15:42:08.230942 -- sending ack for the recieved 1779
15:42:08.233985 -- out-of-order packet 1781 is recieved
15:42:08.234428 -- out-of-order packet 1782 is recieved
15:42:08.234465 -- out-of-order packet 1783 is recieved
15:42:08.234487 -- out-of-order packet 1784 is recieved
15:42:08.234519 -- out-of-order packet 1785 is recieved
15:42:08.234548 -- out-of-order packet 1786 is recieved
15:42:08.318360 -- out-of-order packet 1787 is recieved
15:42:08.336403 -- out-of-order packet 1788 is recieved
15:42:08.336781 -- out-of-order packet 1789 is recieved
15:42:08.811057 -- packet 1780 is received
15:42:08.818278 -- sending ack for the recieved 1780
15:42:08.819714 -- packet 1781 is received
15:42:08.827433 -- sending ack for the recieved 1781
```

"Node: h1"

```
15:42:08.081717 -- retransmitting packet with seq = 1782
15:42:08.082021 -- retransmitting packet with seq = 1783
15:42:08.082727 -- retransmitting packet with seq = 1784
15:42:08.082985 -- retransmitting packet with seq = 1785
15:42:08.083050 -- retransmitting packet with seq = 1786
15:42:08.215017 -- ACK for packet = 1777 is recieved
15:42:08.215259 -- packet with seq = 1787 is sent, sliding window = {1778, 1779,
 1780, 1781, 1782, 1783, 1784, 1785, 1786, 1787}
15:42:08.222632 -- ACK for packet = 1778 is recieved
15:42:08.222990 -- packet with seq = 1788 is sent, sliding window = {1779, 1780,
 1781, 1782, 1783, 1784, 1785, 1786, 1787, 1788}
15:42:08.233079 -- ACK for packet = 1779 is recieved
15:42:08.233330 -- packet with seq = 1789 is sent, sliding window = {1780, 1781,
 1782, 1783, 1784, 1785, 1786, 1787, 1788, 1789}
15:42:08.706890 -- RTO occured
15:42:08.707337 -- retransmitting packet with seq = 1780
15:42:08.707424 -- retransmitting packet with seq = 1781
15:42:08.707457 -- retransmitting packet with seq = 1782
15:42:08.707486 -- retransmitting packet with seq = 1783
15:42:08.708215 -- retransmitting packet with seq = 1784
15:42:08.708606 -- retransmitting packet with seq = 1785
15:42:08.708701 -- retransmitting packet with seq = 1786
15:42:08.708734 -- retransmitting packet with seq = 1787
15:42:08.709069 -- retransmitting packet with seq = 1788
```



```
15:42:11.669350 -- packet 1839 is received
15:42:11.692995 -- sending ack for the recieved 1839
15:42:11.694833 -- packet 1840 is received
15:42:11.698303 -- sending ack for the recieved 1840


FIN packet is recieved
FIN-ACK packet sent

The throughput is 0.14 Mbps

Connection closes...

root@UbuVM:/home/fredr/Documents/data2410/eksamen/Reli
```

The time spent waiting for ACK-packets and resending packets has increased exponentially compared to time spent processing packets. Even though the percentage of lost packets only increased by 3 percentage-points, the throughput decreased to 0,14 Mbps. Meaning a lot more time was spent waiting.

Loss rate set to 50%

On the first try, neither the SYN nor SYN-ACK packets were lost (ACK might have been lost, but my implementation resends it), and the data transfer began. The transfer took ages and spent probably 99% of the time waiting for ACK-packets for lost packets. After probably 20 minutes there were so many consecutive losses that the server timed out because it had not received any packet for 10 seconds, which I set as the limit. A loss rate of 50% makes the data transfer abysmal, and for a more complex protocol there should be some way of trying different methods or resending on a different link (if possible).

5. Lost FIN_ACK packet

When a FIN_ACK-packet is lost, the sender, which initiated the final handshake with a FIN-packet, is left waiting. In my implementation, sender waits for the FIN_ACK-packet for a total of five timeouts before signalling for the State Machine to change state to CLOSED where the connection is closed down. But without this handling, the sender would be left waiting forever.

References

Islam, S. (2025, 25 April). Individual Assignment: DATA2410 Reliable Transport Protocol (DRTP). *GitHub*. https://github.com/safiqul/DRTP-v25

Islam, S. (2023, 24 April). header.py. *GitHub*. https://github.com/safiqul/2410/blob/main/header/header.py

Mealy machine. (2025, 14. April). *Wikipedia*. https://en.wikipedia.org/wiki/Mealy_machine

IBM. (no date). *Flowchart of TCP connections and their definition*. Ibm.com. https://www.ibm.com/support/pages/flowchart-tcp-connections-and-their-definition

The Shaggy Dev. (2023, 10. October). *Starter state machines in Godot 4* [video]. https://www.youtube.com/watch?v=oqFbZoA2lnU

The Shaggy Dev. (2023, 28. November). *Advanced state machine techniques in Godot 4* [video]. https://www.youtube.com/watch?v=bNdFXooM1MQ&t=6s