DATA2410 Reliable Transport Protocol (DRTP)

2410 Datanettverk og skytjenester

Home Examination Spring 2025

Candidate number:	233
Course code:	DATA2410
Course name:	Datanettverk og skytjenester / Networking and cloud computing
Study programme:	Bachelorstudium i ingeniørfag – data / Bachelor's Degree Programme in Software Engineering
Number of words:	3092
Submission date:	19.05.2025

OSLO METROPOLITAN UNIVERSITY
STORBYUNIVERSITETET

Introduction

This project is about implementing a reliable transport protocol with the name of DRTP: DATA2140 Reliable transport protocol.

It is built with python and sockets on top of the unreliable, but simple UDP.

It takes advantage of sliding window to increase its throughput by having multiple packets in flight at the same time. It also implements the Go-Back-N strategy to make it reliable by discarding out-of-order packets and only saving data in order. Where missing packets are resent by the sender when no ACKs arrive.

The goal of the program is to be launched from one file called application.py two network nodes. Where an jpg image is transferred with the protocol from the client to the server. The image should arrive identical to the way it was sent. The arguments decides if the program is run as an server or client, and other settings like IP, port and window size. For information about the arguments, read the README.md or do 'python3 application.py -h' for help.

Implementation

The code is divided into four files for organizational purposes. 'application.py' that does argument parsing/validation and then start either the server in 'server.py', or client in 'client.py' based on arguments provided. Lastly, there is the 'utils.py' file that has functions and classes that are common for the client and server.

The code is designed to check validity of the arguments as much as possible when they are parsed. For example, the -c and -s flag is in a mutual exclusion group in argparse. So, if both is used, and argparse error is returned. With the other arguments I have tried to use custom argparse types that raises ArgparseTypeError if they aren't allowed. For example, with `range_check_int(min_int, max_int=None)` checks that the input is an int, and is between the two provided values, where max is optional. Used both for port with range [1024, 65535] and that the window is 1 or larger. IP address is validated that is a valid IPv4 address with `ip_address(ip)`. And if filename is provided, its checked if the file exist, readable and correct format with `file_name_check(file_name)`. After parsing, the code also informs the user if any options provided the server/client does not use and is ignored. This way issues is caught as early as possible and don't need be checked later in the code. For example, Client doesn't start a connection just to find out the file does not exist later. `file_name_check(file_name)` as example:

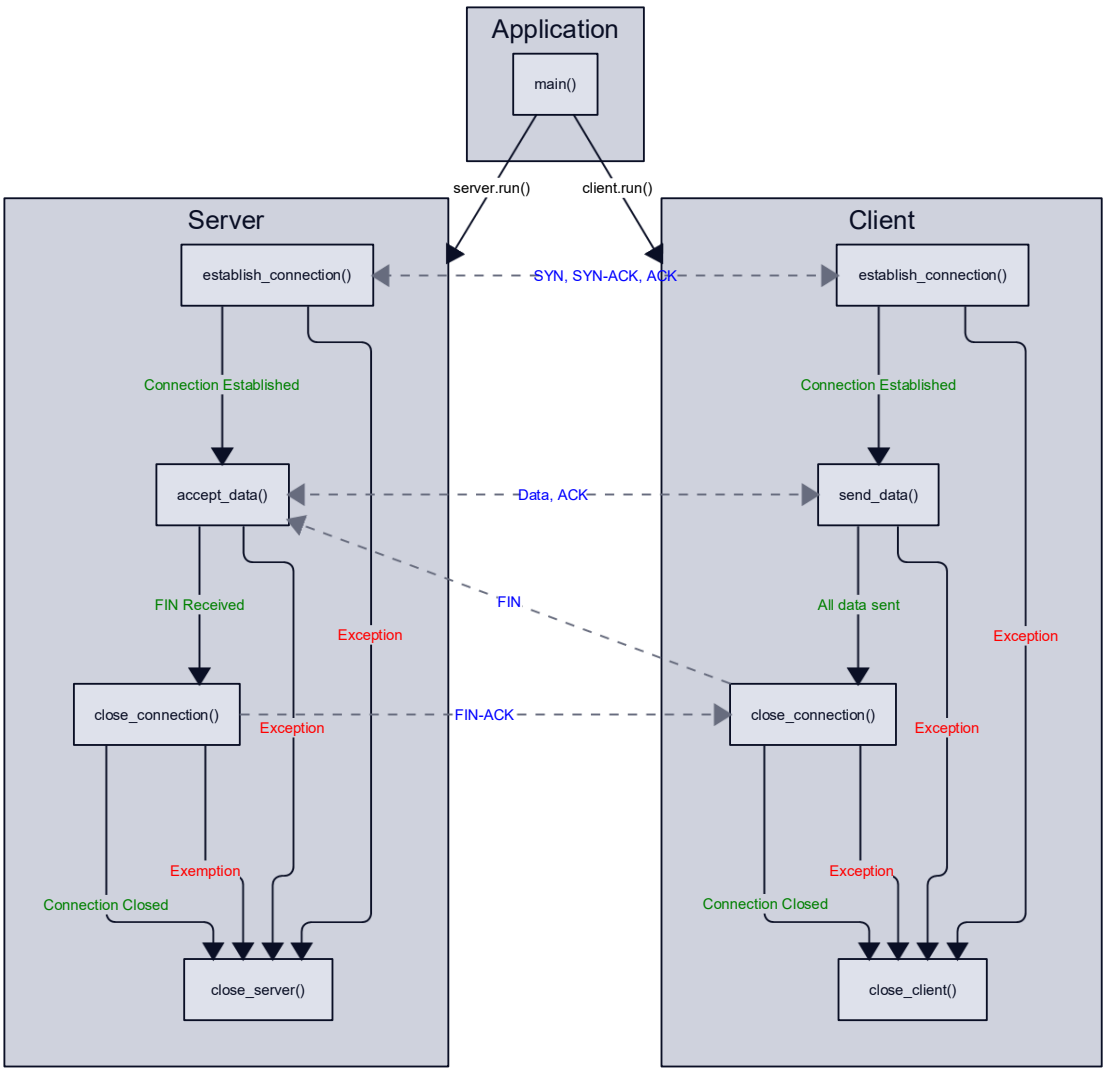
```
def file_name_check(file_name) -> str:
    """
    Custom type for argparse. Check if the file exists and is readable if it's provided.
    Also checks if the file is a correct file format. (.jpg, .JPG, .jpeg, .JPEG)
    :param file_name: Name of the file to be checked.
    :return: Filename as a string.
    :raises argparse ArgumentTypeError: If the file does not exist or is not readable.
    """
    if file_name == "": return file_name
    if not isfile(file_name) or not access(file_name, R_OK):
        raise argparse.ArgumentTypeError(f"{file_name} does not exist or is not readable")
    if not file_name.lower().endswith((".jpg", ".jpeg")):
        raise argparse.ArgumentTypeError(f"{file_name} is not a valid image file type, must be jpg, jpeg")
    return str(file_name)
```

Server and Client is implemented as classes with variables because that made it easier with not having to input all the variables in functions. Can more often call e.g. "self.variable" instead. A `run(self)`

method is the starting point for both, where the main methods of them are started from. It also exists gracefully with the `self.exit_server(0)` and the `self.exit_client(0)` methods if keyboard interrupt happens. Where open file and sockets are closed before the scripts exits. Those two methods are also used when exceptions happen.

In general, the code is written with a design principle that Server and Client is divided into main methods for connection establishment phase, data transfer phase, and connection teardown phase. Where the different methods only returns when they are successful or exits when an Exception is thrown. Ignoring packets with incorrect flags. As by the task description + FAQ seem to be fine if it's done that way.

This diagram to visualize how the code is built up and works:



For example the method for establishing connection on the Client side, `establish_connection(self)` method:

```
try:
    print("Connection Establishment Phase:\n")
    self.socket.sendto(create_packet(0, 0, Flag.SYN, 0), self.server_address)
    self.socket.settimeout(self.TIMEOUT)
    print("SYN packet is sent")

    # Waits for SYN | ACK ignores other packets
    while True:
```

```
        packet = self.socket.recv(1000)
        _seq_num, _ack_num, flags, receiver_window, _data = parse_packet(packet)

        if Flag.SYN | Flag.ACK == flags:
            print("SYN-ACK packet is received")
            self.socket.sendto(create_packet(0, 0, Flag.ACK, 0), self.server_address)
            print("ACK packet is sent\n"
                  "Connection established\n")
            break
        else:
            print("Received packet missing SYN or ACK flag\n")

    # Chooses the smallest window size between the client and receiver.
    self.window_size = min(self.window_size, receiver_window)
except timeout:
    print("\nError: Connection timed out while trying to establish connection")
    self.close_client(1)
except ConnectionError:
    print("\nError: Connection refused by server while trying to establish connection")
    self.close_client(1)
except Exception as e:
    print(f"\nUnexpected error: {e}")
    self.close_client(1)
```

Sends an packet with SYN flag, then waits for the server to respond with SYN-ACK flags and then send ACK flag to complete the three way handshake with the server. The client also receives servers window size and stores the smallest of it and its own window size, as the actual window size. The loop is only broken if the ACK flag is sent. Or if a exception is thrown that makes the application close. So that `send_data(self, start_seq_num=1)` only runs if the client thinks the handshake is complete.

Timeout on the client is set to 0.4 sec as specified in the task, while the timeout of the server is set to 2 sec to prevent it from hanging while the client has closed because of some error. But at the same time long enough that the client should make an error first.

When the data transfer phase happens, the only thing that the server do in the `accept_data(self, start_seq_num=1)` method is to wait for the correct data packet with right sequence number, writes it to file via the `FileHandler` class and sends back packet with ACK flag and ACK number for it. Ignoring and logging other packets with wrong flag or out of order (Other than logging them). Or the packet that is market to be discarded. And waits for a packet with FIN flag before starting closing the connection:

```
elif Flag.FIN == flags:
    self.close_connection(client_address)
    break
```

Where the client side is more complicated in the `send_data(self, start_seq_num=1)` as there is there most of the sliding window and Go-Back-N logic is:

```
    last_data_packet = float("inf")     # Temp value just so it compares true with an int
    last_retrans_num = 0
    retrans_attempts = 0

    try:
        self.send_window(range(next_ack , next_ack + self.window_size))

        #  Continue sending data packets per ACK until the last packet is ACKed.
        while True:
            try:
                packet = self.socket.recv(1000)
                _seq_num, ack_num, flags, _window, _data = parse_packet(packet)
```

```
            if Flag.ACK == flags and ack_num == next_ack:
                print(f"{time_now_log()} ACK for packet = {ack_num} is received")
                next_ack += 1
                next_seq_num = next_ack + self.window_size - 1
                data = ""

                # Check if the last data packet has been found.
                if last_data_packet == float('inf'):
                    data = self.file_handler.get_file_data(next_seq_num)
                    if data == b"":
                        last_data_packet = next_seq_num - 1

                # Sends data if it's not after the last data packet.
                if next_seq_num <= last_data_packet:
                    self.socket.sendto(create_packet(next_seq_num, 0, 0, 0, data), self.server_address)
                    print(f"{time_now_log()} packet with seq = {next_seq_num} is sent, sliding window =
{list(range(next_ack, min(next_ack + self.window_size, last_data_packet)))}")

                # Returns if the last data packet has been ACKed
                if next_ack > last_data_packet:
                    self.file_handler.close_file()
                    return
            else:
                print(f"Received packet with wrong flag or wrong ack number received")

    except → close here
```

Where the first thing that happens is that `send_window(self, window, retransmission=False)` sends all the packets with sequence number provided in the window list.

Window is remembered by having a pointer to the first and last sequence number in the current window. Instead of having a list of the previous packets as memory, I created a `FileHandler` class that you give a sequence length, and then can ask for data from by sequence number, and it calculates what data to read as binary data:

```
position = (segment_num - 1) * self.segment_size

self.file.seek(position)
return self.file.read(self.segment_size)
```

That way, it can easily recreate the packets on demand instead of specifically saving all of them them.

The Client then listens for ACK flag packet, for the data packets, and slides the window and sends the next data packet. This is the sliding window method. While it does that, it always checks for the last data packet and stores its sequence number, so that empty packets aren't sent.

If I timeout happens, `send_window(self, window, True)` is used to resend the whole window again. This is what creates the Go-Back-N strategy and makes the protocol reliable. Number of retries on the same sequence number is kept track of, if it gets above 8, its exits. This is to prevent and possible infinite loop where client keeps sending packets because it keeps timing out if no response from server happens for some reason. I believe this is also a mechanism that TCP can have.

When all data is sent, the `close_connection(self)` sends packet with FIN flag to the server to start closing the connection. That triggers method with the same name on server, that responds with FIN-ACK. That again triggers the server and client to close gracefully with the `exit_client(0)`/`exit_server(0)` methods. As mentioned earlier, if any Exception happen, the client and server closes with error code 1.

Discussion

Measured result for 1 and 2, Window sizes at RTT 50ms, 100ms an 200ms

I have combined the result of the tests in discussion question 1 and 2 as I think its logical and systematic to put it all in one table as it's the same tests just with different RTT.

Test results for window sizes at RTT 50ms, 100ms and 200ms

		RTT		
		50ms	100ms	200ms
Window Size	3	0,42 Mbps +- 0,01	0,23 Mbps +- 0	0.12 Mbps +- 0
	5	0,70 Mbps +- 0,01	0,38 Mbps +- 0	0,19 Mbps +- 0
	10	1,37 Mbps +- 0,02	0,75 Mbps +- 0,01	0,38 Mbps +- 0
	15	2,06 Mbps +- 0,02	1,13 Mbps +- 0,01	0,58 Mbps +- 0
	20	2,06 Mbps +- 0,02	1,13 Mbps +- 0,01	0,58 Mbps +- 0
	25	2,06 Mbps +- 0,02	1,12 Mbps +- 0,01	0,58 Mbps +- 0

Note: Ran the tests 3-5 times each because I saw some variance in the result, so I didn't include the ones that for some reason was little lower than others.

1 Discussion: window sizes at RTT 100ms

The results clearly show that with increased window size from 3 to 15, the throughput also increases. As expected, throughput when window size is for example 10, is about twice of if it is 5. That makes sense because there are more packets in flight. This shows that using the sliding window mechanic is a powerful method to increase the throughput over a network connection.

The results also show no increase after size 15 on the client's side. That's because the server is set to always have a window of 15 that it sends to the client with the header, as specified in the task. Client chooses the lowest of its own and the server at connection establishment time to not overwhelm the server.

Sometimes when I tested, but rarely, out of order happened even if no packet was lost when having window of 15+ I am not sure if Mininet is supposed to emulate that that can happen or if it was a bug. It can def happen in real life if for example the network changes little bit. If a router drops and they must take a different path for example. In those cases, the out of packets are discarded and rand throughput gets little bit lower.

In theory, the throughput should be Window size * Packet size / RTT. So, at the different window sizes, it should be with window size of 3:

3 * 1000 B / 0,1 s = 30 000 B/s = 0,24 Mbps

The rest of the values for other window sizes can be seen in the table on the next page.

That matches quite well with the result I got in the tests. Test result is a little lower, but I am guessing that is for example about small inefficiencies from the optimal in python code or the Mininet simulation.

Calculated throughput for window sizes at RTT 50ms, 100ms an 200ms

		RTT		
		50ms	100ms	200ms
Window size	3	0,48 Mbps	0,24 Mbps	0,12 Mbps
	5	0,8 Mbps	0,4 Mbps	0,2 Mbps
	10	1,6 Mbps	0,8 Mbps	0,4 Mbps
	15	2,4 Mbps	1,2 Mbps	0,6 Mbps
	20	2,4 Mbps (Limited to window 15 because server)	1,2 Mbps (Limited to window 15 because server)	0,6 Mbps (Limited to window 15 because server)
	25	2,4 Mbps (Limited to window 15 because server)	1,2 Mbps (Limited to window 15 because server)	0,6 Mbps (Limited to window 15 because server)

2 Discussion window sizes at RTT 50ms and 200ms

Results of the tests is in the table on the previous page.

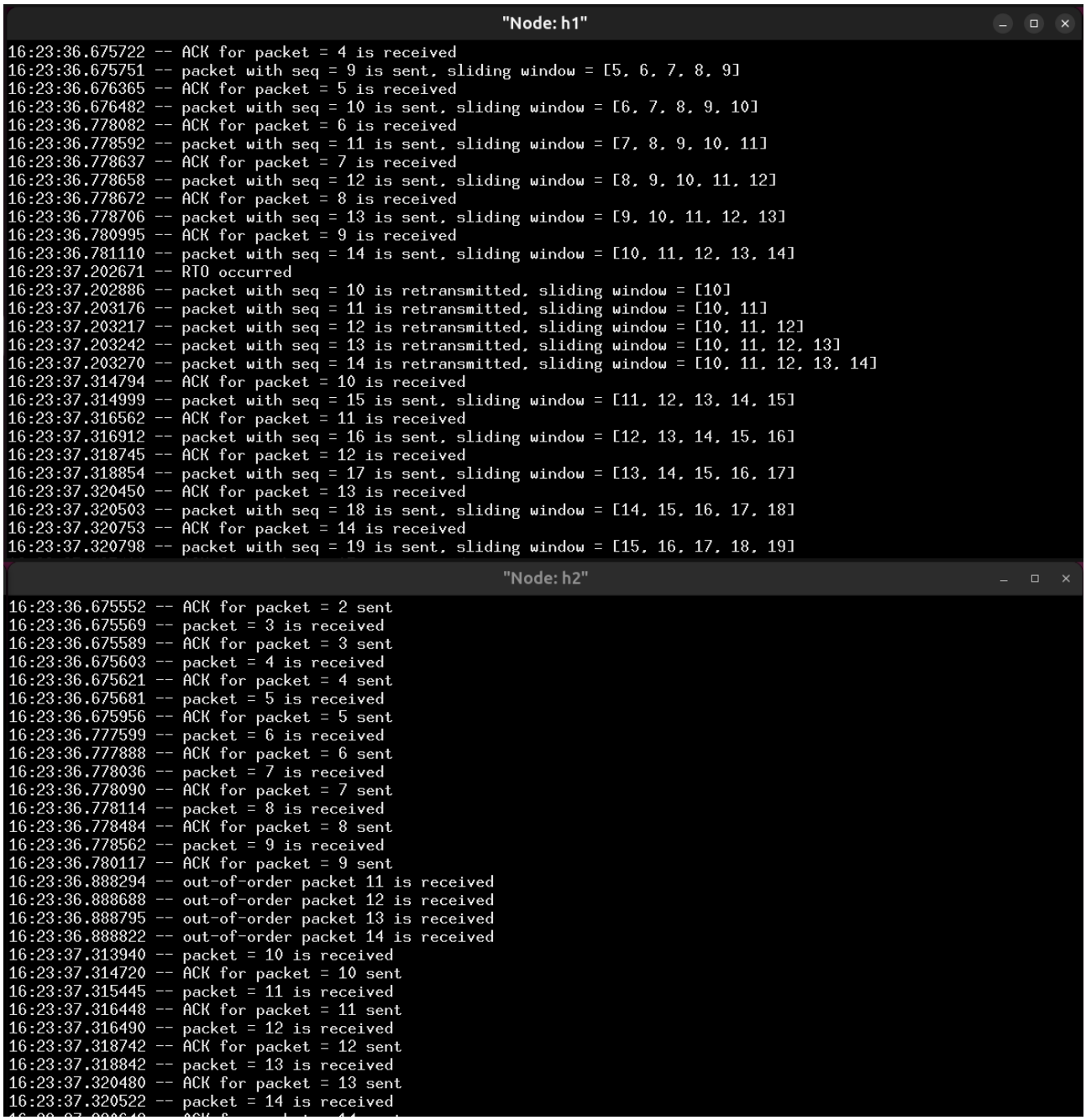
The test results show that while RTT doubles, the throughput gets about halved with the same window size. And that the throughput about doubles when the RTT get halved. This matches up with what makes logical sense and also matches quite with the calculated theoretical calculated throughput in the table above. Just a small difference that can be from other factors like delays in the python code.

As mentioned in the previous discussion question, reason for no throughput increase from 15 to 25 window size is because the window of the server is set to always be 15 as specified in the task.

That means that if your RTT doubles, you must double the window size to get the same throughput. Or half the window size if the RTT halves.

3 Discard flag to drop a packet

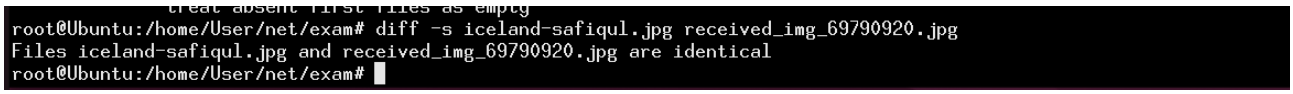
I decided to test it with dropping packet 10 with -d on a windows size of 5:



With one packet dropped, it looks like both acts as they should do with Go-Back-N strategy. The server listen for the next in order data packet. Packet with any other sequence number is logged as out-of-order in the console and discards them, and do not ACK them.

While the Client tries to have the window size of packets in flight at the time. When it doesn't get an reply in time, an RTO timeout happens after 0.4 sec and all packets are considered lost. Then it retransmits the whole window of packet 10-14 in this case, before continuing as normal.

The file arrives as it should, identical to file that was sent, other than the filename. This proves that the protocol is reliable even if one packet is lost:



4 Packet loss with window size 5 and RTT 100ms

Did not specify in the discussion what windows size should be used here, but I decided to use 5 as used in the example.

Packet loss	**Throughput**
2%	0,26 Mbps +- 0,00
5%	0,18 Mbps +- 0,01
50%	DNF

As you see by the numbers from the results in the table, increased packet loss means lower throughput as packets is lost. It lowered from 0,38 Mbps with zero packet loss to 0,26 Mbps for 2% loss and 0,18 Mbps for 5%. That means that there is around 31,5% lower throughput at 2% loss and around 52,6% for 5% when using window size of 5.

If a packet was lost while trying to establish a connection, both client and server would close with an error message.

Reason why the throughput gets hit so much more than the pack loss is, is that the Go-Back-N strategy requires packets to arrive in order, and if they don't, they are discarded and resent. So if the first packet of a window, or its ACK is lost, the whole window is lost and must get retransmitted. Magnifying the throughput loss per packet lost. Bigger window will make magnification even worse. That makes it even more important to lower packet loss when Go-Back-N is used.

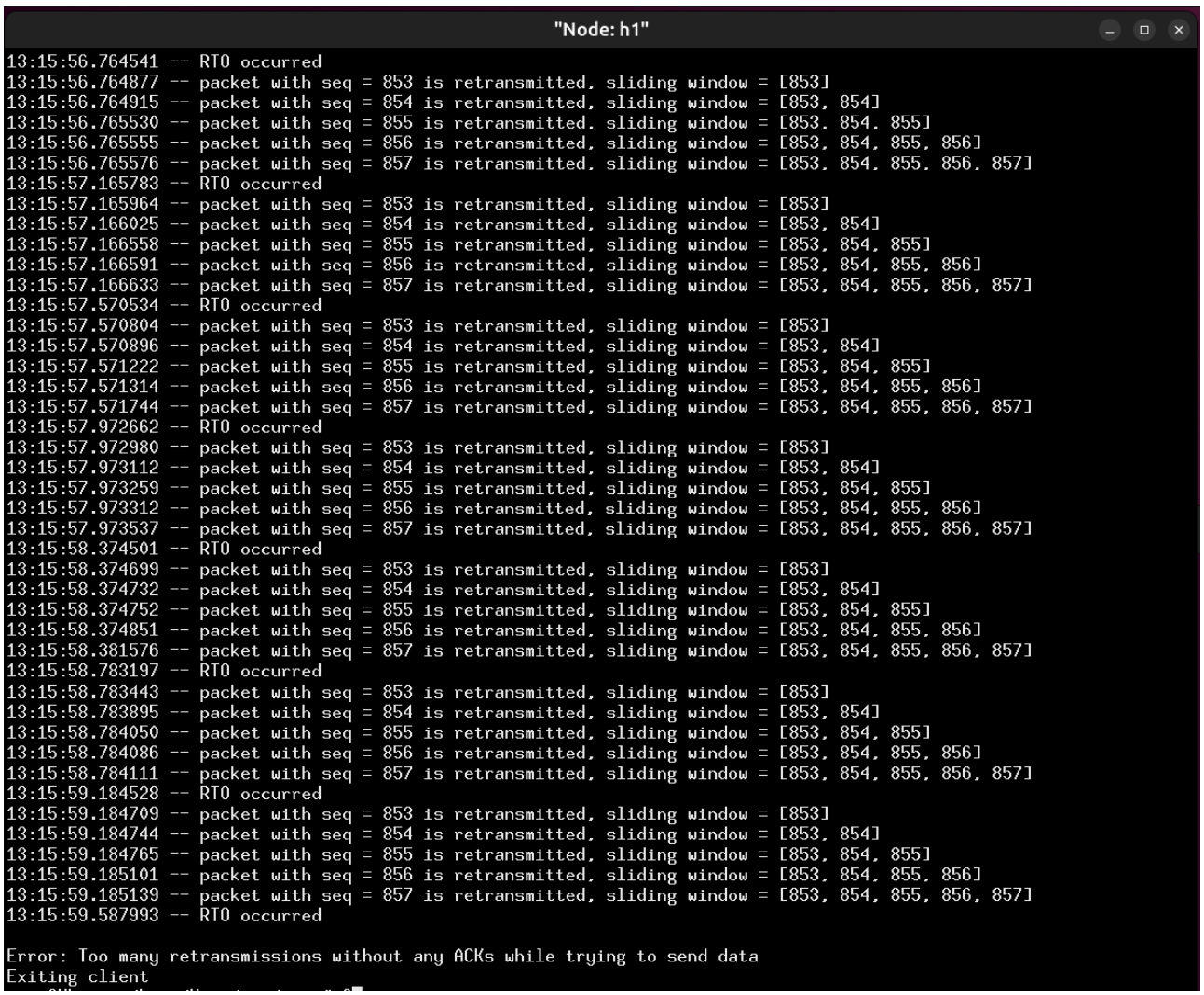
If an selective repeat that buffer packets side and only retransmits missing packets had been used, it would theoretically lead to less throughput lost when packets is lost. That is because then the whole window would not have been lost. Especially with higher percentage packet loss and larger windows sizes as window sizes would no longer be magnifying it.

The protocol is still reliable even with 5% packet loss, proven by that the image arrived identical as the one that was sent (other than the name):

```
root@Ubuntu:/home/User/net/exam# diff -s iceland-safiqul.jpg received_img_10709174.jpg
Files iceland-safiqul.jpg and received_img_10709174.jpg are identical
root@Ubuntu:/home/User/net/exam#
```

With 50% packet loss did not manage to complete. A lot of times trying to run it, the SYN/Establish connection failed to succeed because one of the 3 packets didn't arrive. The times it did manage to establish connection, sending data transfer went really slowly. Slower than what it would have been with window size of 1, but with no packet loss. And eventually, it failed to complete because my code exits if the same package needs to be transmitted too many times. That code is there to prevent infinite loop that for example can happen if the ACK of the establishment phase. Or other faults like the server have closed because an exception. It would have completed eventually, but very slowly if not for that.

Logically, that makes sense because if you need 2 packets to confirm a packet, both the data packet and the ACK packet. Only one lost means it fails, the chance for failure is quite high when both has a 50% chance to get lost.



This shows that with Go-Back-N algorithm, it's extremely important in real world scenario to lower packet loss. To for example avoid packets loss because of congestion.

5 Lost FIN-ACK packet

If FIN-ACK is lost with this protocol, the server will close the connection and not listen anymore right after the FIN-ACK is sent and don't know if its lost or not. While the client doesn't know if it's the FIN or FIN-ACK packet is lost. In this code, what happens then is that the client closes because of timeout. Dirty connection termination. It would have been better if it had tried to resend the FIN package up to a set number of times before closing or getting connection refused.

The consequence of this is that resources aren't freed up as fast as they could have been.

References

Github of Safiqul, the lecturer of the subject: https://github.com/safiqul

Material from the DATA2410 subject on Canvas.

Python documentation: https://docs.python.org

Object-Oriented Programming (OOP) in Python: https://realpython.com/python3-object-oriented-programming/

Multiple stackoverflow threads for small bits of code.