




MAY 11, 2023

# IMPLEMENTATION OF DATA2410 RELIABLE TRANSPORT (DRTP)

A SIMPLE TRANSPORT PROTOCOL

MISKI MOHAMAD YUSUF S364592, JOHAN TRYTI S362059, LATIFA AJRAM S349520, PETTER HALSNE S320947  
Oslo Metropolitan University



1	INTRODUCTION	1
2	BACKGROUND	1
3	D RTP	3
4	EXPERIMENTAL SETUP	6
4.1	Virtual network	6
4.2	Running the code and checking for errors	6
4.3	Emulating loss, duplication and reordering using NetEm (bonus 1)	7
4.4	Dynamic timeout (bonus 2)	8
5	RESULT AND DISCUSSION	8
5.1	Calculated throughput	8
5.2	Test case 1	9
5.2.1	Timeout for test case 1	9
5.2.2	Throughput for test case 1	9
5.3	Test case 2	10
5.4	Test case 3	11
5.5	Test case 4: Emulating delay, packet loss, reordering, and duplicates with tc-NetEm	12
5.5.1	Testing that NetEm changes network conditions correctly	12
5.5.2	Testing the efficiency of our code	13
5.5.3	Handling of reordering, loss, and duplication	16
6	CONCLUSIONS	18
7	REFERENCES	18

# 1 Introduction

In this report, we'll be introducing an implementation of a simple transport protocol -DATA2410 Reliable Transport (DRTP). DRTP's main goal is to ensure dependable data transmission on top of UDP by guaranteeing that data arrives in the right order, without any losses or duplicates. We also tested our program on a virtual network with random loss, reordering and duplication of packets. Our project consists of developing two key programs: DRTP itself, which offers a reliable connection over UDP, and a file transfer application that features a basic file transfer client and server.

There were a few limitations for this project:

- **Statistics:** This experiment is not done in a statistical fashion, as that was not the goal. This may lead to some problems when comparing data, as differences when running the experiment might cause the tests to have different outcomes which do not come from our code. For example, in one instance of running a Mininet, we noticed the delay was much higher than usual after the VM had been running for a long time. In addition, when testing for loss, duplication and reordering, the chance of these occurring is random, so running more tests would be useful. We decided to not use any statistical approach as that seemed outside the scope of the project, but we did send a relatively large .txt file of ca. 4.0 MB (which equals to sending ca. 2800 packets) when testing, as sending a large file will decrease the influence of randomness.
- **Hardware:** the whole experiment is run in a VM with limited resources. This means that instead of each node being their own machine, they all share the same resources. If there's hardware limitations when running the tests, it will therefore affect every node in the network, and the hardware will affect the efficiency of the tests.
- **Efficiency:** due to the scope of this project, the code will obviously not be as efficient as possible. This will cause some differences compared to if the code was "perfect". Our way to decide that our code was good enough was by looking at the time spent executing code vs the time the packets spent in transit. When we noticed that hardware only used a small amount of the total time, we were happy with the code implementation. Example: link latency = 100ms, time to send a packet and receive an ack = 102 ms. In this case, ca. 2% of the total time is spent due to hardware delays, which is more than good enough for our case.

In the remainder of this document, we will start by giving a description of the background of this project. Thereafter we explain the implementation of both DRTP and the file transfer application. In section 4, we explain the experimental setup of our project, as well as the various methods employed to test the reliability of our solution. In section 5, we show the results we got from testing and discuss these, related to the efficiency of our code and how the 3 different methods (SR, GBN and SAW) compared to each other. We also explain how our code handles packet loss, duplication, and reordering. Finally, we'll wrap the project up with a conclusion and the references used in this text.

## 2 Background

Endpoints in a network connect through links and switches. They need an ISP for internet access. They Communicate by exchanging packets. Protocols such as TCP and UDP help guide the packets.

The choice of transport layer protocol impacts transmission speed, protection against packet loss, assurance of the next packet's arrival, and security. TCP guarantees the server's existence and ensures packets arrive in the right order without loss. In contrast, UDP provides no guarantees and doesn't limit throughput. TCP is a connection-oriented protocol, a connection is established in

advance (handshake) before data is exchanged. When the transfer of data is finished, the connection is closed. UDP is a connectionless protocol, it means a connection is not established between sender and receiver before data is exchanged.

UDP adds a header with source and destination ports to the segment. That's (almost) all UDP does before it forwards to the network layer. There the packet receives an IP header and is then forwarded without a guarantee against data loss. The advantages of UDP are that you don't need a connection first, you can send as many packets as you want and UDP headers take up less space than TCP (8 bytes versus 20). UDP sends with a checksum on the packet. It makes it possible to check whether bits in the package have been changed. (Kurose, 2022)

In computer network settings ARQ (Automatic Repeat Request) Protocol is a technique that ensures the accuracy of transmitted information. To achieve this, three elements are required:

- Error detection: A way to determine if bit errors have occurred.
- Receiver feedback: The possibility to send feedback if the package has been received correctly. An ACK is sent when everything is fine, and NAK is when an error has occurred.
- Retransmission: If an error occurs while sending a package, it should be possible to send it again (Kurose, 2022)

To keep track of which packets have been successfully received, a sequence number can be assigned to each packet. In addition, the sender can set a timer, and if it does not receive an acknowledgment (ACK) within a certain amount of time, the packet is resent to ensure its delivery. The three reliable functions are stop and wait, Go-back-N and selective repeat.

- Stop-and-wait protocol: This method involves sending one packet at a time, and the next packet is sent only when the sender receives an acknowledgment (ACK) for the previous packet. If the sender doesn't receive an ACK within a set period, the same packet is resent. The packets in this system are numbered either 0 or 1 to keep track of them.
- Pipelining: Here the interval for the sequence area is increased and the sender and receiver use a larger buffer. Thus, it is possible to send out many packages at the same time. Afterward, the server can find out which packets need to be resent
- Go-back-N: Here there is a limitation on the number of packages we have not received an Ack for within the time frame. These packages come right after each other. With the limit of, for example, 5, packet 6 will not be sent until we have received an ACK on packets 0-4. The receiver discards out-of-order packets to simplify its buffer management. This approach allows the receiver to focus on delivering data in the correct order without storing misplaced packets. The sender manages the bounds of the window, while the receiver tracks the sequence number of the next in-order packet.
- Selective repeat: Here there is also a restriction on packages that we have not received an ACK on within the time frame: The difference from Go-Back-N is that the receiver sends ACK for all packets, and the sender only resends the packets for which it has not received an ACK back. The packets that arrive in the wrong order are buffered at the receiver until it has all the packets it needs to put them in the correct order before they are sent to the receiver's system. The timer helps prevent lost packets. Each packet needs a separate timer because only one packet will be sent if a timeout happens (Kurose, 2022).

After understanding the background concepts and protocols involved in network communication and data transfer, we are now ready to implement the DRTP.

### 3 DRTP

We started by developing a server and client that would communicate with each other via UDP protocol. We introduced a three-way handshake process for both parties to set up a connection. Additionally, we integrated three data transmission techniques: Stop-and-wait, Go-Back-N, and Selective repeat. These methods were handled by a function named 'transmitAndListen'.

Several libraries are used to accomplish various tasks within the program, 'header' is used for handling packet headers, while other standard Python libraries like 'argparse','socket','sys','re','os','random', and 'time' are also imported.'

A global variable named 'window' is set to 64000, which signifies the window size for the transmission methods.

Here is a description of each function:

**check\_IP function:** receives an IP address and verifies its validity. First it confirms the IP address has the correct format. It makes sure each number's IP address is within the range of 0 to 255. If the IP address doesn't meet these criteria, an exception is raised.

**check\_port function:** validate the provided port. it checks that the input is an integer and falls within the range of 1024 to 65535. If the port doesn't meet these requirements, an exception is raised.

**check\_file function:** verifies if the specified file path is valid by making sure that the file exists and can be accessed. If the file is invalid, an error message is displayed, and the program is terminated.

**check\_timeout:** checks that the inputted timeout is valid. If the timeout is float, the default timeout is set to that number. If the input is "dyn", the timeout is set to 0.5, and later changed based on the time it takes to receive an ack for each packet. If neither is true, an error is raised.

#### **handshakeServer function:**

This method implements a three-way handshake connection for server clients. The server receives a message from the client, and it checks if it is the first SYN message from the client. If it is, the server sends a SYN-ACK message to the client, which contains an acknowledgment number and sequence number. Then the server checks for the second SYN message from the client, when the functions print a message the handshake process is complete.

#### **handshakeClient function:**

This method implements a three-way handshake connection for server and clients. The client creates and sends SYN messages to the server. Then it waits for a SYN-ACK response from the server. When it gets the response, it will then send an ACK message to the server to complete the three-way handshake. We also implemented a timeout where the client will resend packages if the ack is not received.

#### **TransmittAndListen function:**

The method transmits data between client and server using one reliability function, either stop-and-wait (SAW), Go-Back-N (GBN) or Selective Repeat (SR). The reliability method used is taken from the user arguments. After data transmission is complete, the function enters the finish mode by sending a FIN message to the server. The method then waits for a response from the server and checks if the response is FIN-ACK message. If no ack is received, the client will resend the FIN message. If the FIN-ACK is received the client will close print out the relevant output (time spent, throughput, transfer,

packers retransmitted and average, minimum and maximum dynamically calculated timeout). Thereafter the client will close the socket.

**sendPacket function:**

Send a packet to the server. If the “loss” argument is invoked, then packet number ten will become lost one time, and then retransmitted later to the server.

If the dyn-flag is set, it also adds the ack number and time when the packet was sent to a global list, which getAck uses to dynamically set the timeout.

**getAck function:**

Receives a packet from the server and returns the ack number. If the dyn-flag is set, it dynamically calculates the timeout. It does this by getting the current time for any received acks. It then adds the difference time between the received acks and sent sums to the perPacketRoundTripTime array, overwriting the time from the ack function.

**specialSum function:**

Sums the last x numbers of the perPacketRoundTripTime, but only adds numbers smaller than 10000 to the sum. This prevents numbers from being added if acks are not received yet, as the sendingPacket function stores numbers there representing the number of seconds from 1970 until today. The check is needed in case packets get received in the wrong sequence.

**Stop\_and\_wait function:**

This function implements the Stop-and-wait protocol. The function has a loop where it sends data packets one at a time, and then waits for acknowledgment (ACK) from the server before sending the next packet. The loop continues until all packets have been sent and acknowledged. If no ACK is received within a specified timeout, the method retransmits the same packet.

**goBackN function:**

goBackN function sends packets to the server. The packets are stored in an array and the functions then send the packets in chunks of size windowSize (5, 10 or 15 packets per chunk). After sending the packet the function listens for acknowledgements (ACKs) from the server. After receiving acks from the server, checks if the received acks are correct (ignoring duplicates and not caring about the ordering), the functions then send the next set of packets.

**selectiveRepeat function:**

This method implements Selective repeat functionality. The function has a loop where it sends a certain number of packets, specified by window size, to the server. Then it waits for acks from the server, and when the acks is received, the function removes the sequence number from the list of packets that need to be retransmitted. If an ack is not received within a certain time period, the function enters a loop where it resends all packets that have not received an ack. It repeats this process until all packets have been sent and acknowledged and then returns the final seq number.

**PackFile function:**

This method takes in the filename that the user has passed into the argument parser. It then tries to open the file, view the content as binary and adds chunks of 1460 bytes to an array. This array then contains all the data that needs to be sent from the client to the server to recreate the file. The method returns the array to pass it to the reliability method chosen by the user.

**UnpackFile function:**

This method takes in the array containing the data received from the client and the filename specified by the user. It then unpacks the file and writes it to a new file with the specified path.

**sendAck function:**

This method is invoked by the server when it wants to send acks to the client. It takes the acknowledgment number, server socket and client Socket as arguments, and sends the corresponding ack-packet to the client. If the test case is “skipack” and the ack number is 33, it will refuse to send the packet once.

**createServer function:**

Creates a server with UDP socket and binds it to the given IP-address and port number. The server then listens for any client trying to invoke a handshake with it. After the handshake is done, the sequence number from the start of the conversation is passed, along with an empty array to the chosen reliability method. After the conversation is done, the reliability method returns the array and is passed along to unpacking to recreate the file.

**serverSaw function:**

ServerSaw receives messages from the client and checks if the flag in the message is zero or not. If the flag is zero, it's a packet containing data. The server will always return an ack for any packet returned and save it if it's not already saved. If the flag is not zero, we go into finish mode (see CheckForFinish function below).

**serverGBN function:**

serverGBN works like serverSAW by checking the flag of each packet. If the flag is 0, it's a data packet. If the data received is in order, it will save the data to the list, and reply to the client with acks for all the received data. If the data is not received in the right order, then it discards the data and does not reply to the client. If the data received is already added to the list, then it still checks if the data is received in the right order and responds with acks to the client. But it does not save the data to avoid duplication.

**serverSR function:**

The “selectiverepeate” method for the server will run if the return code is not invoked. Inside the loop it will listen to new messages from the client. If the received message contains data, the function checks for two cases: if the seq number is received is the right one, then it will append to the receiveData list. If the sequence number is not the expected one, the function stores the message in the nestedBufferList. The other case is if the received message contains a finish flag, then it checks if all the messages have been received. If all the messages have been received, the function returns the receivedData list. The functions also send ACK message for every received message except for when “skipack” testcase is enabled. If it is enabled, the ack message is not sent for the message with sequence number 10 one time, and then handles the situation and continues afterwards (see CheckForFinish function below).

**CheckForFinish function:**

This method is invoked by the server in each reliability method. When the flag of a receiving message is not equal to zero, then we know the client most likely is done transmitting and wants to finish the connection with the server. If the FIN flag is equal to two, then we know the client wants to finish. The server then replies with a packet consisting of the right acknowledgment number and both the FIN and ACK flag set. After the packet is sent, the server closes the connection.

**createClient function:**

The method creates a UDP socket for the client. If the command line timeout is set to dyn, the socket timeout is set to 0.5 seconds. Otherwise, the timeout is set to the value of the timeout argument.

After setting the timeout value then a message is printed that indicates that the client is created. After that the function calls the handshake method to start transmission with the server.

#### Last part of the code:

We handle input from the command line for setting up the server and client, choosing a reliability method, and running optional test cases.

We create a parser for command-line inputs.

Options are set for the server (-s flag, IP binding -b) and client(-c flag, server IP -l, file to send -f).

Shared options include port -p and reliability method -r. Additional test options use the -t flag.

Users must either pick a server or a client, not both. For clients, we check port and IP, set socket timeout, and run create client (). For the server, we check port and IP, then run createServer(). If neither server nor client is picked, an error is shown, and the program stops.

## 4 Experimental setup

### 4.1 Virtual network

To run this experiment, we used VMware Workstation 17 as a virtual machine to run our program in. We then used the program Mininet to create a virtual network. The network created was a simple network with a client connected to a server via a router (see below, fig. 1. for the topology). Note that both end nodes could be used as either server or clients, but for simplicity we stuck with always using h1 as the client and h3 as server, and these names are used for them throughout this text.

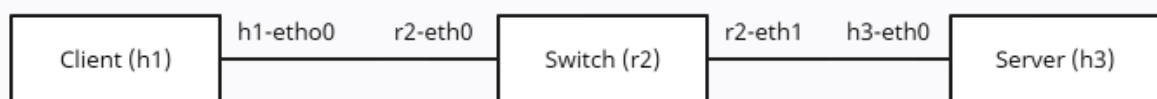


Figure 1: A diagram showing the topology for our virtual network.

The link from the client to switch is called h1-etho0, the link from r2 to the server is called r2-eth1, server to r2 is called h3-eth0 and r2-client is called r2-etho0. Each link has a delay equal to 1/4th of the total RTT. To change the RTT, we simply changed the delay in the file, and then saved a new file as for example RTT50, which we used to run Mininet for the topology with the new delay. Ping was used to ensure the Mininet topologies were correct.

### 4.2 Running the code and checking for errors

To run the code, we first started Mininet using the following command: `python3 path/RTTX`, where x was the RTT we wanted to test with. We then used the `xterm h1 h3` command, which opened a separate terminal for h1 and h3. h3 was then used to run the server and h1 to run the client. The files we sent for the different tests were either a file of size 0.1229 MB (84 packets, referred to as the small file, and called `fil.txt` in the code) or 4.0889 MB (2778 packets, referred to as the big file, called `numbers.txt` in the code), depending on the test case. For test case 1, we tested using the big file as we were interested in the throughput. When we are interested in the throughput, it is helpful to send a bigger file as that will cause more packages to be transferred and any big deviations will have a smaller impact on the total result. For test cases 2 and 3, we only wanted to test if “skip\_ack” and “loss” worked, therefore we used the smaller file. For the bonus tasks, we used the big file.



The smaller file is useful if we use a method which takes a long time to run otherwise, or if we are not interested in the throughput. An example for this could be to check if the “skip\_ack” and “loss” methods are working as supposed to. It’s also useful for testing in general, while the big file is useful when we know we are going to use the results for our rapport.

To check errors that might not cause the program to fail, we had the following checks:

- We used a simple program compare.py after running each test. Compare.py takes 2 files as input and sees if their content is identical. We used it to compare the original file and the file stored by the server and confirmed that they were identical for any test shown in this rapport.
- Used a variable to store the number of retransmitted packages and saw if it made sense when looking at the final output.
- Used a dynamic RTT and checked that the value calculated was logical. Also stored all RTTs and checked if the highest, lowest, and average were within expectations.
- Did the NetEm test for packet loss, duplication and reordering. In addition, we did all three at once.
- Compared the throughput of our program with a theoretical maximum throughput, and confirm that our result was reasonable.
- We tested sending both .png and .txt files of different sizes.

#### 4.3 Emulating loss, duplication and reordering using NetEm (bonus 1)

To use NetEm to change network conditions, we did the following:

- 1) First, we deleted the links which we wanted to change:  
h1 sudo tc qdisc del dev h1-eth0 root  
r2 sudo tc qdisc del dev r2-eth0 root2)
- 2) Then we replaced them with the test case that we wanted:  
h1 sudo tc qdisc add dev h1-eth0 root netem delay 25ms reorder 10% loss 10% duplicate 10%  
r2 sudo tc qdisc add dev r2-eth0 root netem delay 25ms reorder 10% loss 10% duplicate 10%

In the example provided, we have changed the links between h1 to r2 and r2 to h1 to have a 10% chance of packet loss, 10% chance of packet duplication and 10% chance of packet reordering. If we wanted to test for only one of these, we could have done it by not using the other arguments. As another example, the following makes link h1-eth0 have 10% loss and a link latency of 6.25 ms (making the RTT 25, assuming the other links also have a delay of 6.25 ms):

```
h1 sudo tc qdisc add dev h1-eth0 root NetEm delay 6.25ms loss 10%
```

Note that we always change two links, one from h1 to r2 and one from r2 to h1. This is because we want to both check if the program can handle duplication, reordering and duplications of both acks and packets, and they are sent through different links.

By default, the reordering in NetEm works by randomly making packets skip the link delay. For example, for an RTT of 100, adding random reordering to a link means to make a packet have a chance to skip the 25 second delay. As a result of this, the dynamic RTT calculations for packet reordering is lower than otherwise.

#### 4.4 Dynamic timeout (bonus 2)

To calculate a dynamic timeout, we used the “sendPacket” and “recieveAck” functions, which are always called when we send and receive packets. sendPackets always adds the current time and seq number of the packets sent into an array (even if they are already present). recieveAcks then adds the time of any received packet to the time of the last sent packet with the corresponding seq number. This ensures that only the value for the last sent packet is updated.

We then use a function “specialsum” to calculate the RTT, before multiplying it by four to get the timeout. Specialsum is explained in section 3 in more detail. Specialsum is used to calculate the average RTT for the last X packages. X is a rather arbitrarily chosen number, which is 3\*packet size for GBN and SR and 10 for SAW. The number is higher for GBN and SR as each packet group is sent and received at the same time (assuming no losses etc), and can therefore in a way be looked on as one data point. Therefore we thought it was prudent to have SR and GBN being based on the window size.

Of course, it would be even better to dynamically adjust the amount of data used in the RTT calculation over time as more points become available, especially for GBN and SR, as 3 window sizes might be very little if we count that as 3 data points. Then we could have weighted the newer data points higher in the final calculation. We decided against this approach as it seemed too large for this task. Therefore we used our much more simple calculation with a fixed amount of data used for the calculations and no weighting for the most recent packages.

In addition, for GBN, treating each window as one data point instead of several could also be beneficial. The same is somewhat true for SR, but a key difference between SR and GBN is that the server side on SR sends an ack once it receives a packet, instead of waiting for all packets in the current window.

We also used an output for the calculated timeout to see if our results made sense. This output shows the minimum, maximum and average round trip time for packets sent.

## 5 Result and discussion

### 5.1 Calculated throughput

We calculated an expected throughput by using the following formula:

$$\text{calculated throughput} = \frac{\text{Packet Size}}{\text{perPacketRoundTripTime}}$$

When window size was used for SR and GBN, we simply multiplied the calculated throughput by the window size.

The calculated throughput is the maximum possible throughput and assumes no delay from anything but link latency. This actual throughput will of course be lower due to hardware.

We also expect that the ratio between throughput and the calculated throughput will decrease when RTT increases, and further decrease when window size increases. This is because lower RTT will cause a relatively higher amount of the total time to be used by the hardware. Window size has roughly the same logic. If the window size is bigger, processing the data will take more time, but when calculating the theoretical throughput, we assume all packets are sent instantaneously.

We also added an efficiency row to the tables, which is the throughput we got from each test divided by the calculated throughput.

## 5.2 Test case 1

### 5.2.1 Timeout for test case 1

For all results, the dynamically calculated timeout was between 0.03 and 0.27 higher than  $RTT \cdot 4$ . We are happy with this result, as this means the RTT is much lower than what the RTT would have been if set dynamically, and close to the link delay  $\cdot 4$ . RTT will be discussed in more detail in the RTT section.

### 5.2.2 Throughput for test case 1

Below are tables for the results of sending a packet of ca 4 MB over a virtual network with a link latency of 100, 50, or 25 ms using the SAW method (table 1), GBN method (table 2) or SR method (table 3). Note that for test case 1, GBN and SR will work very similarly, so they are largely discussed together.

*Table 1: Table 1: RTT and throughput for SAW. Each column represents one test done with the RTT equal to 100, 50, or 25. Efficiency is equal to the throughput divided by the max throughput. The tests were done by sending a packet with a size equal to ca. 4 MB, which amounts to sending ca. 2800 packets.*

Link latency (ms)	100	50	25
Throughput (Mbps)	0.1149	0.2305	0.4363
Max throughput (Mbps)	0.11776	0.23552	0.47104
Efficiency	0.98	0.98	0.93

*Table 2: Table 1: RTT and throughput for SAW. Each column represents one test done with the RTT equal to 100, 50, or 25. Efficiency is equal to the throughput divided by the max throughput. The tests were done by sending a packet with a size equal to ca. 4 MB, which amounts to sending ca. 2800 packets.*

Link latency (ms)	100	100	100	50	50	50	25	25	25
Window size	5	10	15	5	10	15	5	10	15
Throughput (Mbps)	0.568	1.118	1.651	1.127	2.196	3.186	2.097	3.997	5.671
Max throughput (Mbps)	0.589	1.178	1.766	1.178	2.355	3.533	2.355	4.710	7.066
Efficiency	0.96	0.95	0.93	0.96	0.93	0.90	0.89	0.85	0.80

*Table 3: Table 3: RTT, window size and throughput for GBN. Each column represents one test done with the RTT and windowSize written in the column. Efficiency is equal to the throughput divided by the max throughput. The tests were done by sending a packet with a packet with a size equal to ca. 4 MB, which amounts to sending ca. 2800 packets.*

Link latency (ms)	100	100	100	50	50	50	25	25	25
Window Size	5	10	15	5	10	15	5	10	15
Throughput (Mbps)	0.570	1.118	1.668	1.135	2.223	3.247	2.123	4.069	5.860
Max throughput (Mbps)	0.589	1.178	1.766	1.178	2.355	3.533	2.355	4.710	7.066
Efficiency	0.97	0.95	0.94	0.96	0.94	0.92	0.90	0.86	0.83

A general observation from all three test cases is that the decrease in RTT is roughly proportional to the increase in throughput.

A shorter RTT means it takes less time for a sender to receive an acknowledgment for received data, so the sender can transmit the next packet faster, increasing throughput.

In GBN and SR, as the window size increases, so does the throughput. This is because a larger window size lets the sender send more packets at once before waiting for acknowledgments, which makes better utilization of the network capacity.

For both results, this is pretty much what we expected. Since we expect hardware delay to account for a much smaller portion of the delay than the link latency, a doubling in RTT or window size should roughly mean a doubling in throughput. Of course, the latency is small, or window size is big, the hardware will account for much more of the delay, and this relation between RTT or window size and throughput will no longer be the case. Also, a cap on the efficiency of window size will also be reached eventually due to the bandwidth.

Another result is that the efficiency decreases both with decreasing RTTs and increasing packet sizes. This supports what we've written above, as this causes the time spent by the hardware to account for a relatively larger amount of the total time.

In summary, the results show that both GBN and SR are effective, so throughput was higher for GBN and SR compared to SAW. While the Stop-and-Wait protocol waits for acknowledgement for each sent packet, GBN, and SR allow sending multiple packets simultaneously, which increases the utilization of network capacity.

### 5.3 Test case 2

At the server side we have generated an example of loss of ack message to a packet with sequence number 33 (see fig. 2).

In both "stop and wait", "Go-back-N" and "Selective repeat" we call on this method to send acks from server to the client.

```
def sendPacket(seq, data, clientSocket, serverConnection):  
  
    flags = 0 #sets flags variable to 0  
    packet = header.create_packet(seq, 0, flags, window, data) #calls the create_packet methon to create a packet  
    if (args.testcase == "loss" and seq == 10): #: Checks if the args.testcase flag is set to "loss"  
        print(f"Packet with sequenceNumber: {seq}, was lost")  
        args.testcase = "loss_completed"  
    else:  
        clientSocket.sendto(packet, serverConnection) #if the flag is not set, the packet is sent to the server  
  
    if args.timeout == "dyn": #If the timeout is dynamic  
        global perPacketRoundTripTime #tells the function to use these global arrays  
        perPacketRoundTripTime.append([seq,time.time()]) #Appending the seq number and time to a list.
```

Figure 2: sendAck() function called in serverSR(), serverGBN(), and serverSAW()

If the user of the application invokes "skipack" and we reach the package with sequence number 33, then the if-case will be triggered. This will lead to a print of a package becoming lost, and an ack will be sent.

To make sure the ack is only sent once, we change the args.Testcase argument when the ack has been sent.

The SAW method handles skipped entering a timeout if it does not receive an ack for the sent packet. It then resends the packet to the server, and the server will send the ack, which then gets processed as normal.

GBN handles it in a very similar way. If an ack is not received, it simply resends all the packets from the current window. The server then once more waits for all packets to be received in the correct order. Once all packets have been received, it once more sends the acks, and if the client receives them, they move to the new window. For retransmitted data, we've added a test case to ensure it's not stored on the server side, as it's already stored there.

SR being a hybrid between SAW and GBN handles it another way. If an ack is not received, the client will go to a timeout exception. It will then retransmit all packets which have not gotten a corresponding ack. This loop will repeat until all acks have been received, and then it will move to the next window.

## 5.4 Test case 3

On the client side we have created an example of loss of data for a packet with sequence number ten. In all the three reliability methods the `sendingPacket()` function is called when sending a package:

```
def sendPacket(seq, data, clientSocket, serverConnection):  
  
    flags = 0 #sets flags variable to 0  
    packet = header.create_packet(seq, 0, flags, window, data) #calls the create_packet method to create a packet  
    if (args.testcase == "loss" and seq == 10): #: Checks if the args.testcase flag is set to "loss"  
        print(f"Packet with sequenceNumber: {seq}, was lost")  
        args.testcase = "loss_completed"  
    else:  
        clientSocket.sendto(packet, serverConnection) #if the flag is not set, the packet is sent to the server  
  
    if args.timeout == "dyn": #If the timeout is dynamic  
        global perPacketRoundTripTime #tells the function to use these global arrays  
        perPacketRoundTripTime.append([seq,time.time()]) #Appending the seq number and time to a list.
```

Figure 3: `sendingPacket()` method called in `stop_and_wait()`, `goBackN()`, and `selectiveRepeat()`

When the client wants to transmit a package with data, this method is invoked. If the argument of "loss" is set by the user and the sequence number of the packet equals ten, then the packet will not be sent. Afterward, the argument's value will be set to "loss\_completed" instead of "loss". This ensures that we will go into the else case instead of the if-case, and therefore transmit rather than lose the packet.

Servers for "stop and wait" and "selective repeat" will send acknowledgement messages right away to the client when a package is received. If they don't receive it, they will not reply with an ack. Therefore, the client exactly knows which packet was lost in transmission and will retransmit this packet. The difference between SAW and SR is that SAW just retransmits the last package sent. SR needs to keep track of all the packages sent in each window size, and control which acknowledgments are received to which packets. SR does this using an array. When packet ten is lost, then the server will not receive an ack for it, but will only retransmit this packet, and not the rest of the last transmitted window size of packets.

This is not the case for GBN. The GBN server will simply discard all the packages if they are not correctly received in the right sequence. Thereafter, the client will enter a timeout, before retransmitting all packages. The client checks if all acks received are equal to the seq numbers it sent for the current window, but without caring about the order they appear in. It does this by using the built-in set function, which causes the order of the items to be irrelevant. If they are equal, the client moves on to send the next window.

```
def sendPacket(seq, data, clientSocket, serverConnection):

    flags = 0 #sets flags variable to 0
    packet = header.create_packet(seq, 0, flags, window, data) #calls the create_packet method to create a packet
    if (args.testcase == "loss" and seq == 10): #: Checks if the args.testcase flag is set to "loss"
        print(f"Packet with sequenceNumber: {seq}, was lost")
        args.testcase = "loss_completed"
    else:
        clientSocket.sendto(packet, serverConnection) #if the flag is not set, the packet is sent to the server

    if args.timeout == "dyn": #If the timeout is dynamic
        global perPacketRoundTripTime #tells the function to use these global arrays
        perPacketRoundTripTime.append([seq,time.time()]) #Appending the seq number and time to a list.
```

Figure 4: SendingPacket() method called in stop\_and\_wait(), goBackN(), and selectiveRepeat()

When the client wants to transmit a package with data, this method is invoked. If the argument of “loss” is set by the user and the sequence number of the packet equals ten, then the packet will not be sent. Afterward, the argument's value will be set to “loss\_completed” instead of “loss”. This ensures that we will go into the else case instead of the if-case, and therefore transmit rather than lose the packet.

Servers for “stop and wait” and “selective repeat” will send acknowledgement messages right away to the client when a package is received. If they don't receive it, they will not reply with an ack. Therefore, the client knows exactly which packet was lost in transmission and will retransmit this packet. The difference between SAW and SR is that SAW just retransmits the last package sent. SR needs to keep track of all the packages sent in each window size, and control which acknowledgments are received to which packets. SR does this using an array. When packet ten is lost, then the server will not receive an ack for it, but will only retransmit this packet, and not the rest of the last transmitted window size of packets.

This is not the case for GBN. The GBN server will simply discard all the packages if they are not correctly received in the right sequence. Thereafter, the client will enter a timeout, before retransmitting all packages. The client checks if all acks received are equal to the seq numbers it sent for the current window, but without caring about the order they appear in. It does this by using the built-in set function, which causes the order of the items to be irrelevant. If they are equal, the client moves on to send the next window.

## 5.5 Test case 4: Emulating delay, packet loss, reordering, and duplicates with tc-NetEm

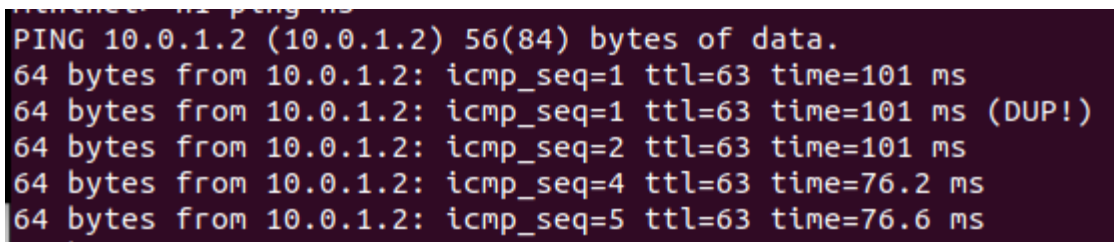
### 5.5.1 Testing that NetEm changes network conditions correctly

To test if NetEm is correctly implemented, we used the ping command. The figure 4 below shows that NetEm is correctly implemented with all three flags (loss, duplication, and reorderings). We see that loss is implemented because the packet with seq 3 is skipped. Duplication is seen in the packet with seq = 1. The reordering is a bit trickier to spot, but as written at the start of the paragraph, we know reordering is implemented as skipping the delay for the link. We, therefore, see that the reordering flag happens correctly as the packets with seq flags 4 and 5 have a time of  $\frac{3}{4}$  of the minimum RTT. Of course, no actual reordering happens when we use ping, as only one packet is sent at a time.

Another way we confirmed that reordering happened as we wanted was by testing with the GBN flag. GBN retransmits packets if they are out of order. When testing on the virtual network with only

reordering (no loss or duplicates), GBN retransmitted a lot of packages, as seen in the figure. 5. This figure shows output from a network with an RTT of 100, a 10% chance of reordering acks, and a 10% chance of reordering data sent to the server. Also note how the lowest calculated timeout is 0.314, which is caused by the way packet reordering happens as explained in section 4.4.

In theory, the RTT could even reach as low as 0.2, although that is highly unlikely. This would only happen if all the sent packets in a group are reordered (so that they end up in the original sequence, which they need to be for the server to send acks). The chance for this to happen with a window size of 10 is equal to  $(0.1)^{10} = 0.00000001\%$ , or in non-mathematical terms, not extremely likely. If that happens, one or more acks have to be reordered as well ( $1 - (0.9^{10}) = 65\%$ ), which would cause the ack to be received 50 ms (+ hardware delay) after the corresponding package was sent. 65% is also the chance that a group of packets have to be retransmitted, as one reordering causes all the packets the server sent to the client to have to be retransmitted.

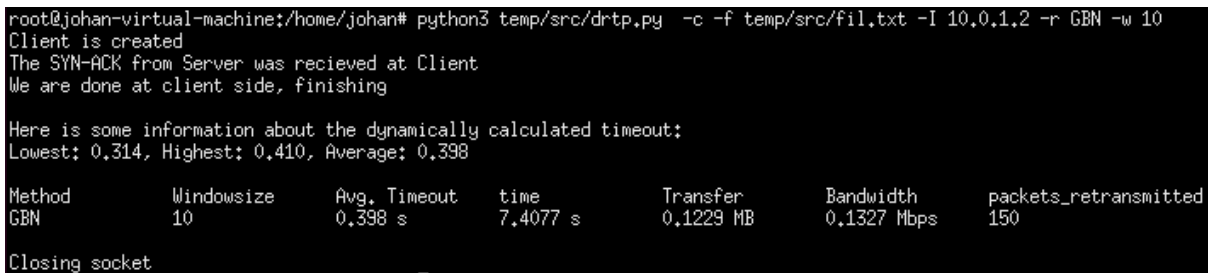


```

PING 10.0.1.2 (10.0.1.2) 56(84) bytes of data.
64 bytes from 10.0.1.2: icmp_seq=1 ttl=63 time=101 ms
64 bytes from 10.0.1.2: icmp_seq=1 ttl=63 time=101 ms (DUP!)
64 bytes from 10.0.1.2: icmp_seq=2 ttl=63 time=101 ms
64 bytes from 10.0.1.2: icmp_seq=4 ttl=63 time=76.2 ms
64 bytes from 10.0.1.2: icmp_seq=5 ttl=63 time=76.6 ms

```

Figure 5: The ping command after using the NetEm command to alter the network to include packet loss, replication, and duplication, with an RTT of 100 ms



```

root@johan-virtual-machine:/home/johan# python3 temp/src/dntp.py -c -f temp/src/fil.txt -I 10.0.1.2 -r GBN -w 10
Client is created
The SYN-ACK from Server was recieved at Client
We are done at client side, finishing

Here is some information about the dynamically calculated timeout:
Lowest: 0.314, Highest: 0.410, Average: 0.398

Method      Window size  Avg. Timeout  time      Transfer    Bandwidth    packets_retransmitted
GBN         10           0.398 s       7.4077 s  0.1229 MB   0.1327 Mbps  150

Closing socket

```

Figure 6: Output from a vm where each sent ack and seq have a 10% chance to be reordered, tested using GBN with a window size of 10 and an RTT of 100. Note that the RTT is 150, meaning that reordering happened for 15 groups of packets (15 groups =  $15 \cdot 10$  packets = 150 packets retransmitted).

## 5.5.2 Testing the efficiency of our code

The results from section 5.2 show the efficiency of our code when there is no loss, reordering or duplication of data. There, SAW is by far the worst program to use, while GBN and SR are roughly equal, and both get much faster with increased packet sizes. However, when implementing loss, reordering and duplication, the results get vastly different. Then, we expect SR to be the best method, followed by GBN and SAW. SAW might outperform GBN, due to GBNs “throw-away-everything”-way of handling exceptions. If the loss/reordering/duplication is very big, or window size is very big, GBN is expected to have lower throughput than SAW.

To test these hypotheses, we performed the following tests:

1. Window size 5, 10% loss
2. Window size 10, 10% loss
3. Window size 15, 10%loss
4. Window size 5, 10% loss, 10% reordering and 10% packet duplication.
5. As a final test, we tested SR with window size 15 and 10% loss, reordering and duplication.



All these tests were performed with an RTT of 100. Note that 10% refers to that happening both when client sends a packet to server, and when server sends a packet to client. Tests 1-3 were done with the big file and a window size of 15. For test case 4, we still used the big file, but decided to use a window size of 5, as that drastically reduces the run time of GBN. Of course, another way to run the test without having to wait for such a long time could be to change the probabilities and/or RTTs, but we decided to stick with 10% chance and 100ms RTT for simplicity's sake. For test case 4, a window of 15 only has a 2% chance of moving on to the next package group.

Chance explained:

Chance for the packets to correctly received reach the server:

$(0.9 * 0.9 * 0.9)^{15}$  (Duplication, replication and reordering all make the server not send acks.)

Chance for the ack packets to correctly be received at the client:

$(0.9)^{15}$  (reordering or duplicates does not matter on the server side).

multiplying those chances together gives us the final chances, which is roughly 2%.

Tables 4-7 show the results for all these tests. Figure 6 shows the outputs for the “ultimate” test of our code: window size 15, SR, 10% reordering, loss and duplication.

Results from the tests:

*Table 4: results for SAW, tested with 10% loss for both the link between client and server, and from server to client. The link latency was 100 MS. The packet sent had a size of 4 MB.*

Link latency (ms)	100
Throughput (Mbps)	0.0609
Max throughput (Mbps)	0.11776
Efficiency	0.52
Average timeout (s)	0.406

*Table 5: results for GBN, tested with 10% loss for both the link between client and server, and from server to client. The link latency was 100 MS. The packet sent had a size of 4 MB.*

Link latency (ms)	100	100	100
Window size	5	10	15
Throughput (Mbps)	0.056	0.036	0.017
Max throughput (Mbps)	0.5888	1.178	1.766
Efficiency	0.10	0.03	0.01
Average timeout (s)	0.408	0.414	0.428



Table 6: results for SR, tested with 10% loss for both the link between client and server, and from server to client. The link latency was 100 MS. The packet sent had a size of 4 MB.

Link latency (ms)	100	100	100
Window size	5	10	15
Throughput (Mbps)	0.110	0.156	0.205
Max throughput (Mbps)	0.589	1.178	1.766
Efficiency	0.19	0.13	0.12
Average timeout (s)	0.408	0.409	0.409

Table 7:: results for SAW, GBN and SR, tested with 10% loss, 10% duplication and 10% reordering for both the link between server side and client side with a total link latency of 100 ms. The packet sent had a size of 4 MB, except for the test of GBN, which had a size of 0.1 MB.

Method	SAW	GBN	SR	SR
Link latency (ms)	100	100	100	100
Window size	1	5	5	15
Throughput (Mbps)	0.0768	0.0458	0.1255	0.2384
Max throughput (Mbps)	0.118	0.589	0.589	1.766
Efficiency	0.65	0.08	0.21	0.13
Average timeout (s)	0.378	0.386	0.386	0.388

```

root@johan-virtual-machine:/home/johan# python3 temp/src/drt.py -c -f temp/src/numbers.txt -I 10,0,1,2 -r SR -w 15
Client is created
The SYN-ACK from Server was recieved at Client
We are done at client side, finishing

Here is some information about the dynamically calculated timeout:
Lowest: 0.366, Highest: 0.426, Average: 0.388

Method      Window size  Avg. Timeout  time      Transfer    Throughput   packets_retransmitted
SR           15           0.388 s      137.2203 s 4.0889 MB   0.2384 Mbps   549

Closing socket

```

Figure 7: the output for SR for 10% loss, packet reordering and duplication with an RTT of 100 and window size of 15.

#### Discussion of the efficiency:

When there is no loss, reordering, or duplication, GBN and SR perform similarly and are much faster than SAW (see section 5.2). However, the situation changes when loss, reordering and duplication is introduced.

SAW originally demonstrates lower efficiency compared to GBN and SR. However, with packet loss, SAW has better throughput than GBN even when the window size is only 5 (explanation for GBNs poor throughput mbelow). SAW is of course significantly slower due to the packet loss, as seen if comparing tables 4 and 1.

GBN's throughput decreases with increasing window sizes under high packet loss conditions (see table 5, where throughput decreases from 0.056 to 0.036 to 0.017 with corresponding window sizes 5, 10 and 15). Since the chance for an error increase with increased window sizes, this decrease in throughput is within expectations. GBN's performance was even worse than SAW's under our test conditions.

SR maintains the best efficiency under all conditions. Its performance remains superior to both SAW and GBN under high packet loss, duplication, and reordering. As reordering and duplication does not affect SR to any mention worthy degree (as it does not trigger retransmission), we expected SR to really shine under those conditions. It is noted that SR has a higher calculated average throughput when reordering and duplication are introduced in addition to loss, compared to only loss. This is (at least partly) caused by reordering causing the calculated timeout to be lower (see section 5.5.1). SR's ability to retransmit only lost packets makes it the best choice in environments with loss, reordering and/or duplication.

In summary GBN is getting worse and worse with increasing window sizes when the packet loss is high enough. SAW performs badly overall and is not a good choice.

SR performs well, especially compared to the other two methods which are ineffective when there is a significant loss, duplication, or reordering. SR could of course become even better with improvements such as dynamically changing the window size or always sending a new packet when a correct ack is received, but that's outside the scope of this task.

### 5.5.3 Handling of reordering, loss, and duplication

*Loss:*

As several cases of our code running as dealing with reordering, loss and duplication (or all three), this section is dedicated to explaining how we handle the different scenarios instead of showing that we can handle it, as we've already shown results where reordering, loss and duplication were present. We won't talk about dealing with loss here, as that's explained in sections 5.3 and 5.4. The examples given there for the loss of one packet will also count in the general case.

*Re-ordering:*

For SR, reordering is no problem, as SR is a function that is made to handle reordering from scratch. If a packet is received in the wrong sequence, it is simply placed in a buffer, and then it's saved on the server side once all the packets with the seq-numbers lower than it have been stored.

SAW has no problems with reordering, as it does not occur when sending one packet at a time.

GBN is a function that deals poorly with reordering. If packets are received at the server in the wrong order, it discards all packets and will not send acks. This causes a timeout at the client and packets to be retransmitted. This is still better than the alternative where packets would be stored in the wrong order.

For ACK packets, none of our programs care about the order of the acks as long as they are all received. Therefore, reordering of acks did not cause any problems for our program.

### *Duplication:*

SR on the server side deals with duplication by simply not saving any duplicated packets, but still sending acks for all packets received.

GBN deals with duplication just like it does with reordering. Press the metaphorical panic button, discard all packets, and send no acks. If duplicate acks are received by the client, it simply ignores the duplicates. This causes duplication of acks to not cause any significant delays, unlike the packets which do cause significant delays if duplicated.

Both GBN and SR deal with duplicate acks by ignoring them and continue waiting for all acks in the current window (or in SRs case, it might also be all acks for the packets it's currently retransmitting) to be recieved.

For SAW, we unexpectedly faced the most problems with duplications. This is because SAW only handles one ack at a time. If the ack is correct, then it sends the next packet. If not, it retransmits the last sent packet and once more waits for the correct packet to be transmitted. This causes a problem as it only handles one ack before sending a new packet, a large amount of acks packets can be built up in the buffer for the received messages.

Example:

Seq 2 is sent, duplicated, and 3 acks (one got duplicated) are returned.

Saw receives one ack and sends the next packet. However, 2 duplicate acks from the last package are still left in the buffer, and are instantaneously received. The client then retransmits the package 2 more times, as it sees that the 2 received acks are not correct for the corresponding packet. These packages which are wrongly sent can again cause more acks to build up if the duplication chance is high enough, causing an increasingly large amount of acks to be built up in the buffer. In fact, when we ran this test without having any programming in case to deal with this error and a duplication chance of 10%, we got over 18 000 retransmissions (see figure. 8). Of course, this caused our RTT calculations to fail as well. This was done when sending a file that had a size of only 85 packets, which shows how the exponential growth got out of hand very quickly. For this test case, the transmitted file was still identical (tested using compare.py, like always), which is why it's important to have checks in place other than just seeing if the received file is the correct one.

Another note for this error is that part of the problem is that the duplicate ack "stayed in the system", meaning that once a packet was duplicated in our original ack, it would have no way to remove the extra package from the "send-receive-cycle".

To deal with this error, we implemented a code in our recieve\_ack function which deleted any buffer in the clientSocket (line 234 in the code, see figure. 7 below).

```
if args.reliability == "SAW" and args.serverip != socket.gethostname(socket.gethostname()):
    #Checks for and discards duplicate packages (as they are recieved instantaneously) for the
    #The second check is needed as packets are recieved almost instantaneously when running the
    defaultTimeout = clientSocket.gettimeout()
    clientSocket.settimeout(0.01)
    while True:
        try:
            message, serverConnection = clientSocket.recvfrom(1472) #Listening for message from
        except:
            clientSocket.settimeout(defaultTimeout)
            break
```

Figure 8: Our implementation to clear the client Socket for any buffered acks for the SAW method.

```

root@johan-virtual-machine:/home/johan# python3 temp/src/dntp.py -c -f temp/src/fil.txt -I 10.0.1.2 -r SAW -w 5
Client is created
The SYN-ACK from Server was recieved at Client
We are done at client side, finishing

Here is some information about the dynamically calculated timeout:
Lowest: 0.345, Highest: 0.750, Average: 0.544

Method      WindowSize      Avg. Timeout      time      Transfer      Throughput      packets_retransmitted
SAW          1                0.544 s           20.4965 s  0.1229 MB      0.048 Mbps      19051

Closing socket

```

Figure 9: running SAW over the virtual network with 10% duplication without having any failsafe for duplication. 19051 packets retransmitted is an indication that something is wrong.

Other:

We made one line that deals with all three cases of loss, duplication and re-ordering at the GBN server side (figure. 9):

```

else:    #if the sequence number is not equal to the correct value, we
        bufferData.clear()
        checkSeqNum = ((seq-2) // args.windowSize) * args.windowSize + 2

```

Figure 10: a part of the code for the GBN at the server side.

The else is triggered if any packet is not the packet we expected (which we control with the checkSeqNum variable). If the packet is not the expected packet, the third line in the code snippet (figure. 7) resets the checkSeqNum variable to the first variable of the window corresponding to the wrongly received seq. Therefore, the server will only start saving data again once the next received data packet is at the start of the receiving window.

## 6 Conclusions

Our implementation has demonstrated that DRTP can provide reliable data transmission over UDP. Through the development of file transfer applications, we have effectively tested and compared the performance of Stop-and-wait, Go-Back-N, and Selective-repeat transmission methods. Our results show that both the Go-back-N and Selective-repeat protocols significantly outperform the Stop-and-Wait protocol. GBN can become worse than SAW under circumstances where losses, reordering or duplicate acknowledgments do occur.

The Selective-repeat protocol has the edge over Go-Back-N in environments with loss, reordering and packet duplication. If none of these conditions are present, then Go-back-N and Selective-repeat methods are performing almost equally.

Overall, our project has successfully implemented and tested the transport protocol, DRTP, providing reliable data transmission over UDP. Through the comparison of the different transmission methods, we have gained insights into their performance in ensuring data transmission without losses, duplicates, or out-of-order arrival.

## 7 References

Kurose, J. F. (2022). Computer networking: a top-down approach (8th edition, global edition.). Pearson Education.