APRIL 30, 2023

ENHANCING UDP RELIABILITY:

A STUDY OF STOP AND WAIT, GO-BACK-N, AND SELECTIVE REPEAT TECHNIQUES

ELINE JØRGENSEN
s362056

GAUTE KJELSTADLI
s362066

ANJU LE
s360712

NIKLAS HAVNAAS
s356237

1 Introduction

The rapid increasing dependency on network communications has led to necessity in development of reliable and efficient transport protocols. Two of the most commonly used transport protocols are TCP (Transport Control Protocol) and UDP (User Datagram Protocol). While TCP provides reliability and error correction, it may not be the most suitable option due to its connection-oriented nature, which can result in higher overhead and latency. Conversely, UDP is connectionless and faster but lacks reliability, making it unsuitable when guaranteed data delivery is required.

To address these limitations, we implemented DRTP, a custom transport protocol that provides reliable data delivery on top of UDP. By leveraging the benefits of both TCP and UDP, DRTP ensures that data is reliably delivered in order, without missing data or duplicates. The protocol is designed to establish a reliable connection, maintain flow control, and gracefully close the connection while transferring files between two network nodes.

The DRTP implementation consists of two parts: The DRTP protocol itself and a file transfer application with a client and a server component. Users can transfer files using DRTP/UDP while choosing between three different reliability functions: Stop and Wait, Go-Back-N and Selective Repeat.

This report briefly describes the DRTP's architecture, the file transfer application and the various reliability functions. The goal is to enhance understanding of transport protocols and their potential applications in different networking scenarios.

2 Background

Before delving into the Reliable Transport Protocol (DRTP), it is essential to understand some key concepts related to transport protocols and their role in network communication. This section provides a brief overview of three fundamental reliability mechanisms in data transfer and its functions within the DRTP.

Transport protocols are responsible for end-to-end communication between two nodes in a network. They provide crucial services such as data transfer, error detection and flow control.

The **Stop and Wait** protocol is a fundamental technique in reliable data transfer. The sender transmits a single packet and then waits for an acknowledgement (ACK) from the receiver before sending the next packet. If the ACK is not received within a specific timeout period, the sender retransmits the packet. This protocol ensures data transfer but has limited performance due to waiting for ACK's after every transmission.

The **Go-Back-N protocol** is an improvement over Stop and Wait. It allows the sender to transmit multiple packets without waiting for an ACK for each packet, using a fixed window size. If an ACK is not received within a timeout period, the sender retransmits all unacknowledged packets. This protocol increases network resource utilization and throughput, but still requires retransmission of multiple packets even if only one packet is lost.

Selective Repeat (SR) protocol optimizes the data transfer process further. Instead of retransmitting all unacknowledged packets like GBN, SR retransmits only the lost or corrupted packets. The receiver stores out-of-order packets in a buffer, assembling them in the correct order before passing them to the application. This protocol provides higher throughput and more efficient error recovery in comparison to Stop and Wait and Go-Back-N.
In the context of this project, these reliability functions offer a range of trade-offs between simplicity, performance and robustness to cater to various networking scenarios.

3 Implementation

3.1 DRTP.py

This code implements a simplified version of the TCP protocol over UDP, referred to as DRTP. The protocol is implemented as a class that encapsulates various functionalities such as sending and receiving packets, creating and parsing packet headers, and performing the TCP three-way handshake for establishing and closing connections.

The DRTP class begins by initializing the server IP, port, socket, and flag variables in the constructor. The send_packet method sends a packet to a given address using UDP sockets 'sendto' method, while receive_packet receives a packet using UDP sockets 'recvfrom' method.

The create_packet method assembles a packet with a header and data. The header is structured as a byte string using the struct module, which includes a sequence number, acknowledgement number, flags indicating the type of packet (e.g., ACK, FIN, or SYN), and the packet's window size. Following the header, the data payload is appended. If there is data to be transmitted, it is added directly after the header. In cases where there isn't any data to transmit, an empty byte string is appended to the header, thereby ensuring the packet structure remains consistent.

The parse_packet method disassembles a packet, extracting its header and data. The header, which is the first 12 bytes of the packet, is parsed into sequence number, acknowledgement number, flags, and window size. The data, if any, is extracted as the rest of the packet after the 12-byte header.

The syn_server and syn_client methods implement the server-side and client-side behaviors for the SYN/SYN-ACK handshake, respectively. This is a part of the TCP three-way handshake process for establishing a connection. Syn_server waits for a SYN packet from the client, sends a SYN-ACK packet in response, and syn_client initiates the handshake by sending a SYN packet and waits for a SYN-ACK response from the server. After SYN-ACK is received from the server, the client sends an ACK back to the server, indicating a successful handshake.

Lastly, the close method closes the connection using the UDP socket's close method. This provides a way to close the socket connection when it is no longer needed.

3.2 Application.py

In this implementation, in order to make use of the DRTP program, we constructed a separate application file that consists of two main components, client and server. The server listens for incoming file transmissions, while the client initiates the actual transmission. Both server and client implement the DRTP protocol to handle the communication.

The program takes several command-line arguments, such as -s to run as a server or -c as a client, -p to specify the port number to listen on or to connect to, and -f to specify the file to transfer. The reliability function can be set with the -r option, and the window size for the sliding window used in GBN and SR protocols can be set with -w. There is also an option to run test cases using the -t option. *Further explanation on how to run these flags is in the readme file.*

The application supports a few test cases, such as the “skip_ack” test case, which simulates a scenario where the server intentionally skips sending an acknowledgement for a received packet. This is

implemented in each reliability function's server code. Similarly, “skip_seq” is a test case implemented in the GBN and SR clients to simulate packet loss and out-of-order delivery by deliberately skipping to send a packet. In addition, we added a test case “duplicate”, which retransmits an old sequence number, to test the servers ability to handle duplicate packets. These test cases helps to evaluate the performance and robustness of the reliability functions under different scenarios

As mentioned, the application’s main functions are the client and server. The client initiates a three way handshake with the server, selects the appropriate reliability function, and starts the file transfer process. Based on the specified reliability protocol, the client waits for an acknowledgement from the server before sending the next packet. Once all the data has been sent, the client in like manner sends a FIN packet to the server to indicate that the transfer is complete. It also calculates and displays the elapsed time, transferred data, and throughput after the file transfer is complete.

The server function listens for incoming connections. Once a connection is established, the server waits for the client to initiate a three way handshake. After the handshake is completed, the server starts receiving data packets from the client, acknowledging them as they arrive according to the given reliability protocol. Once all the data has been received, the server then sends acknowledgement for the FIN packet back to the client, indicating that the transfer is complete.

The Stop-and-Wait protocol comprises two primary functions: *stop_and_wait_server* and *stop_and_wait_client*. The server function operates in a continuous loop, receiving data from the client and writing it to a specified file. For each received packet, the server sends an acknowledgement back to the client, confirming receipt of the packet. If packets are received out of order or if duplicate packets are detected, the server dismisses these packets, sending an acknowledgement only for the last correctly received packet.
The client function operates by sending one packet at a time and waiting for an acknowledgement from the server before sending the next packet. This ensures reliable data transfer. However, if a packet is lost during transmission, resulting in a timeout at the client side, the client function retransmits the lost packet..

Go-Back-N and SR also consist of their own server and client functions. But instead of sending one packet at a time, the Go-Back-N and SR protocols utilize a sliding window mechanism, enabling multiple packets to be in transit simultaneously. The difference between these two is that if a packet is lost and a timeout occurs, the Go-Back-N client retransmits all the packets in the window, starting from the lost packet, while the SR client only retransmits the unacknowledged packets. These approaches enhance throughput compared to Stop-and-Wait. However, if the window size is insufficient, inefficiencies may arise, which we will discuss further in the results section of this report.

In summary, this file transfer application provides a simple and flexible way to transfer files between a client and server using the DRTP protocol. The user can choose between different reliability functions to suit their needs, and the program is designed to handle various network conditions, ensuring a robust file transfer experience.

4 Preliminary information

In order to test our code, we used a simple topology called simple-topo.py that starts a virtual network in Mininet. This virtual network consists of two hosts connected by links to a router. Each link has a bandwidth of 100 and a delay of 25ms (which we changed along the way to achieve our desired RTT).

During the testing phase of our implementation of the DRTP protocol and file transfer application, we used the ping command to verify the correctness of our desired RTT (round trip time). We also used the same picture for all test cases: Photo.jpg, to be able to compare the results. The tests were conducted on the same computer to ensure consistency.

To simulate a network environment with moderate delay, we used an RTT (Round Trip Time) of 25ms for our tests in test-case 2 and 3, as well as for duplicate packets in test-case 4. This enabled us to assess and compare the efficiency of our implementation in equal network conditions.

We also considered scenarios where the error rate is higher, such as wireless network interference, which could result from interference by other devices or physical attenuation due to distance. Additionally, high network congestion caused by many users accessing the network simultaneously can lead to packet collisions, delays, and losses. Therefore, it is crucial to consider these factors when testing our implementation in real-world network environments.

5 Results and discussion

5.1 Test-case 1: Throughput

5.1.1 Results

Stop-and-wait:

RTT	**Transfer time**	**Throughput**
25	66.10s	0.39 Mbps
50	123.68s	0.21 Mbps
100	234.20s	0.11 Mbps

Go-Back-N:

RTT/Window size	**5**	**10**	**15**
25	12.15s 2.15 Mbps	6.27s 4.16 Mbps	4.11s 6.35 Mbps
50	24.19s 1.08 Mbps	12.59s 2.07 Mbps	8.33s 3.13 Mbps
100	47.36s 0.55 Mbps	23.66s 1.10 Mbps	15.98s 1.63 Mbps

Selective Repeat:

RTT/Window size	5	10	15
25	13.65s 1.91 Mbps	6.78s 3.85 Mbps	4.51s 5.78 Mbps
50	25.10s 1.04 Mbps	12.96s 2.01 Mbps	8.51s 3.07 Mbps
100	46.97s 0.56 Mbps	24.45s 1.07 Mbps	16.08s 1.62Mbps

5.1.2 Expectations

Our expectations, for all three transport protocols, are that the file transfer time should be shortest with a RTT (round trip time) of 25ms, approximately double with 50ms, and longest with a 100ms RTT. Similarly, the highest throughput should be achieved with a 25ms RTT, roughly half that with 50ms, and the lowest with 100ms. Among the three protocols, we expect the stop-and-wait protocol to have the longest transfer time and lowest throughput since it only sends one packet at a time and waits for an acknowledgment before sending a new one. In contrast, Go-Back-N and Selective Repeat should perform better, with the transfer time decreasing and the throughput increasing as the window size grows. Although Selective Repeat is more complex than Go-Back-N and therefore expected to be more efficient in networks with higher error rates, we don't expect a significant difference in our tests. This is due to the simple network topology used, and because we transferred a photo that has a relatively low error rate.

5.1.3 Discussion

As expected, the time taken to transfer the file and the achieved throughput are affected by the RTT. Higher RTT values (e.g., 100 ms) result in longer transfer times and lower throughput, while lower RTT values (e.g., 25 ms) lead to faster transfers and higher throughput. This behavior is consistent across all the protocols.

The stop-and-wait protocol has the longest transfer time and lowest throughput among the three protocols. This is expected because the stop-and-wait protocol sends a single packet at a time and waits for an ACK before sending the next packet. Consequently, the protocol's efficiency is directly limited by the RTT, as the sender remains idle during this time.

The Go-Back-N (GBN) protocol shows improvements in transfer time and throughput compared to stop-and-wait due to its ability to send multiple packets within the window size without waiting for individual acknowledgements. As the window size increases, the protocol can send more packets at once, which helps reduce the transfer time and increase the throughput. The efficiency improvement is particularly noticeable when the RTT is high, as the protocol can better utilize the available bandwidth by keeping the sender busy.

The Selective Repeat (SR) protocol also demonstrates better performance than stop-and-wait and shows similar improvements with increased window size as GBN. SR is expected to be more efficient than GBN, particularly in networks with higher error rates or packet loss, as it only retransmits the specific packets that were not acknowledged, rather than resending all packets within the window as GBN does. However, the difference in efficiency between GBN and SR is very small in this scenario due to the simple network topology and relatively stable network conditions. But we do see a slightly better performance with SR than GBN. In scenarios with more challenging network conditions, such

as frequent packet loss or higher error rates, the advantages of SR over GBN would become more apparent.

In our experiments with SR and GBN, we observed an increase in timeouts when employing a high window size and low RTT. GBN, in the event of a single packet loss or delay, requires the retransmission of all unacknowledged packets within the current window, potentially causing unnecessary retransmissions. Conversely, SR selectively retransmits only the lost packet. Nevertheless, high window sizes may still contribute to buffer congestion for the receiver, resulting in packet loss and timeouts. Thus, we found that it is crucial to balance between window size and network delay to optimize performance in these protocols

5.2 Test-case 2: Skip acknowledgement

5.2.1 Results

Stop-and-wait:

Transfer time	**Throughput**
67.29 s	0.39 Mbps

GBN:

Window size	**Transfer time**	**Throughput**
5	14.12 s	1.85 Mbps
10	7.41s	3.52 Mbps
15	5.09 s	5.13 Mbps

SR:

Window size	**Transfer time**	**Throughput**
5	14.77 s	1.77 Mbps
10	7.34 s	3.56 Mbps
15	5.22 s	5.00 Mbps

5.2.2 Expectations

In the skip_ack test case, the receiver intentionally skips sending an acknowledgement (ACK) for a received packet, simulating a situation where an ACK is lost or delayed in the network. The expected results for each reliable protocol is as follows:

Stop-and-Wait: When the sender does not receive the expected ACK, it will wait for a timeout of 500ms before retransmitting the same packet. This process will continue until the sender receives the corresponding ACK. The sender's throughput is expected to be reduced in this test case due to the increased waiting time and retransmission.

Go-Back-N: In GBN, the sender sends multiple packets without waiting for individual ACKs. If the receiver skips an ACK, the sender will eventually detect the missing acknowledgements (either by

timeout or by receiving ACKs for subsequent packets). The sender will then retransmit all packets starting from the one with the missing ACK up to the current window. Therefore, we expect that this test case will show the efficiency of GBN in handling missing ACKs, but it will also highlight the drawback of retransmitting the entire window even if only one ACK is missing.

Selective-Repeat: In SR, the sender stores all acknowledged packets in a separate "received" dictionary. If the receiver skips an ACK, the sender will detect the missing acknowledgement. After a timeout, the sender will check if there are any packets missing from the dictionary and retransmit only the packet with the missing ACK (the packet missing from the received dictionary), instead of resending the entire window as in GBN. We expect this test to demonstrate higher efficiency of SR in handling skipped ACKs, as it selectively retransmits only the missing packets, reducing the number of unnecessary retransmissions.

5.2.3 Discussion

To better understand the implications of the skip_ack test case, we can compare the results with those obtained in the earlier test case without skip_ack.

When comparing the performance of the three protocols in terms of retransmission efficiency, stop-and-wait is the least efficient, as it sends one packet at a time and waits for an ACK before proceeding. GBN is more efficient as it can transmit multiple packets in a window, but it may need to retransmit the entire window even if only one ACK is missing.

Contrary to our expectations, SR acquired slightly lower efficiency than GBN. We suspect that this is because SR requires more complex processing at both the sender and receiver sides, like dealing with out-of-order packets. This additional overhead can slow down the protocol, especially when the packet loss rate is low, and the benefits of SR's selective retransmission don't come into play. Thus, GBN is in this case more resilient to skipped ACKs due to its ability to send multiple packets within the window size without waiting for individual acknowledgements. As a result, GBN can better utilize the available bandwidth and maintain higher throughput even when ACKs are skipped.

Overall, it was observed that the results for SR and GBN in the skip_ack test case more closely resemble those from the first test case compared to Stop-and-Wait. This indicates that these protocols more effectively handle skipped ACKs and maintain their performance levels under such conditions, due to their sliding window mechanisms which allow them to transmit multiple packets at a time.

In conclusion, the comparison of the skip_ack test case with the earlier test case provides valuable insights into the performance of the three protocols under different network conditions. While stop-and-wait remains the least efficient due to its single packet transmission and waiting mechanism, GBN and SR exhibit improved performance and resilience when dealing with skipped ACKs. It is important to consider these differences when selecting a suitable protocol for specific use cases and network conditions.

The server outputs demonstrate the distinct retransmission strategies employed by the different protocols when handling skipped ACKs. The Stop-and-Wait protocol, as expected, retransmits the packet with the skipped ACK and waits for a response before proceeding. GBN resends all packets from the one with the missing ACK up to the current window, while SR selectively retransmits only the packet with the missing ACK.

5.3 Test-case 3: Skip sequence

5.3.1 Results

GBN:

Window size	Transfer time	Throughput
5	14.20 s	1.84 Mbps
10	7.49 s	3.48 Mbps
15	5.15 s	5.07 Mbps

SR:

Window size	Transfer time	Throughput
5	14.01 s	1.86 Mbps
10	7.36 s	3.55 Mbps
15	5.16 s	5.06 Mbps

5.3.2 Expectations

In the skip_seq test case, the sender's main purpose is to deliberately skip sending packets with a specific sequence number. Resulting in out-of-order delivery, and eventually trigger retransmission, simulating a situation where a packet is lost. The expected results for each reliable functions is interpreted below:

Go-Back-N: When a packet is lost, the sender continues to send packets until it reaches the end of the window size. When the sender detects that a packet is lost, it will retransmit all packets starting from the lost packet up to the end of the window size. This test case will show the efficiency of GBN in handling lost packets, but it will also highlight the drawback of retransmitting multiple packets even if only one packet is lost.

Selective-Repeat: The sender maintains a separate acknowledgement status for each packet in the window. If a packet is lost, the sender will detect the missing packet and retransmit only that packet by sending a selective repeat request. This test case will demonstrate the higher efficiency of SR in handling lost packets, as it selectively retransmits only the missing packet, reducing the number of unnecessary retransmissions.

Overall, our expectations for this test case are to evaluate the efficiency and reliability of the GBN and SR protocols in handling lost or skipped packets. By analyzing the performance of the protocols with different window sizes and network conditions, we aim to identify the optimal configuration for each protocol to achieve reliable and efficient file transfer in various network environments.

5.3.3 Discussion

In the skip_seq test case, we intentionally skipped a sequence number to observe the out-of-order delivery effect and trigger retransmission. We then tested the performance of the GBN and SR protocols with different window sizes and finally analyzed the results.

Comparing the results of test case 1 and test case 3, it is evident that the SR protocol is more efficient in managing packet losses, as it only retransmits the specific packets that were not acknowledged. The advantages of SR over GBN become more apparent in more challenging network conditions, such as frequent packet loss or higher error rates. However, both GBN and SR protocols demonstrate

improvements in transfer times and throughput compared to stop-and-wait due to their ability to send multiple packets within the window size without waiting for individual acknowledgments.

Compared to the previous test case ("skip_ack"), we now see better results with SR than with GBN. We assume that this is because SR allows for individual acknowledgment of all packets and can handle out-of-order packets, buffering them until missing packets are received. This selective retransmission is where SR shows its strength, leading to better performance. On the other hand, GBN can't individually acknowledge packets and can't handle out-of-order packets. Thus, in a 'skip_seq' scenario, GBN will generally perform worse than SR.

Overall, our results demonstrate that the performance of the GBN and SR protocols can be improved by increasing the window size. Additionally, our results show that the SR protocol may have better performance than GBN, due to its selective retransmission approach. These findings can help guide future optimization of the DRTP protocol and file transfer application.

5.4 Test-case 4: Duplicate

5.4.1 Results

Stop-and-Wait:

Transfer time	Throughput
38.23 s	0.68 Mbps

GBN:

Window size	Transfer time	Throughput
5	13.68 s	1.91 Mbps
10	6.64 s	3.93 Mbps
15	4.59 s	5.69 Mbps

SR:

Window size	Transfer time	Throughput
5	14.19 s	1.84 Mbps
10	6.84 s	3.81 Mbps
15	4.61 s	5.65 Mbps

5.4.2 Expectations

In the duplicate test case the sender sends packets to the server, and intentionally resends the same packet twice. When this happens the server is supposed to discard the package and move on to send the rest of the packets until the file is sent. Below we have written our expected results for each protocol using this method.

Stop-And-Wait: When the Stop-And-Wait server receives a duplicate packet purposely sent from the client, it will discard the duplicate packet by only sending an ACK for the last correctly received packet.

Go-Back-N: If the window sent to the receiver contains duplicate packets, the server will discard the duplicate packet and send an ACK for the last correctly received packet. It will then proceed to receive the rest of the packets normally.

Selective-Repeat: The Selective Repeat algorithm sends ACKs separately for every received packet. When it encounters a duplicate packet it will discard the duplicate packet and send an ACK for the packet that was previously acknowledged.

5.4.3 Discussion

When testing duplicate packets with Stop-and-Wait we encountered something unexpected. Surprisingly, the transfer time was reduced significantly, and hence, the throughput increased as well. We find this quite strange as we thought the efficiency would worsen by sending a duplicate packet. We came to a conclusion that the most likely reason is because of the duplicate packet being sent immediately after the original packet with the same sequence number (in this case, sequence number 0). The duplicate might actually be received and acknowledged before the original packet, reducing the overall transfer time. Still, we did not expect it to increase this much by only sending one duplicate packet. Therefore, we suspect that our code might have limitations when handling duplicate packets with the Stop-and-Wait protocol.

The GBN protocol handles duplicate packets within a window more effectively. If a duplicate packet is detected within a window, it is discarded, and the rest of the window is processed. As the window size increases we can observe the drastic change in the delay compared to the Stop-and-Wait function. Which is reflected in both the reduced transfer time and increased throughput as the window size increases. However if a packet error occurs, all packets sent after the error must be resent. This means that GBN can still be inefficient if there are many errors or duplicates.

Contrary to our initial expectations, the GBN protocol displayed superior performance over the SR protocol. Despite SR's inherent advantage of individually acknowledging packers, leading to the discarding of duplicate packets within a window without the need to process the entire window, GBN managed to outperform SR in this instance. This resulted in a slight reduction in transfer time and an increase in throughput for GBN relative to SR.

While SR typically boasts the highest efficiency among the three protocols due to its selective retransmission of failed packets, in this particular scenario, GBN performed better than SR, highlighting the potential variability of protocol performance under differing network conditions. However, this performance discrepancy alters when the network environment is manipulated to send more duplicates, as observed in the bonus section utilizing tc-netem. This behavior is further elaborated in section 5.6.3.

5.5 *Bonus task* - Timeout Adjustment Based on Per-Packet Round-Trip Time: 4RTT

5.5.1 Implementation

The Round-Trip Time (RTT) is the duration calculated from the moment a data packet is dispatched from the client to the server until the moment the client receives an acknowledgement from the server confirming the packet's receipt. It is a crucial measure in network correspondence as it directly impacts the efficiency of data transference. If the client does not receive an acknowledgement from the server within a given timeout period, it assumes the packet was lost and retransmits it.

The multiplication by 4 is used to create a buffer, allowing for fluctuations in network circumstances. In a stable network a timeout of 500ms might be sufficient, but networks often experience congestion, routing changes, or other factors that can cause RTTs to differ. By setting the timeout to 4 RTT, this code is allowing the protocol to be more robust to these adjustments.

When an ack is received, the time taken (RTT) is calculated and used to update the average RTT. By taking the collected sum and dividing it by the packet count. This average RTT is then multiplied by 4 in order to calculate the new timeout value, which is then set as the new timeout for the socket.
If there is no packet sent nor received, the average RTT defaults to 0.5 seconds.

5.5.2 Results

Stop-and-wait:

RTT	Transfer time	Throughput
25	65.92 s	0.40 Mbps
50	122.99 s	0.21 Mbps
100	233.68 s	0.11 Mbps

Go-Back-N:

RTT/Window size	5	10	15
25	12.11 s 2.16 Mbps	7.77 s 3.36 Mbps	4.05s 6.45 Mbps
50	24.13 s 1.08 Mbps	11.87 s 2.20Mbps	7.97s 3.27 Mbps
100	46.30 s 0.56 Mbps	23.31 s 1.12 Mbps	15.56s 1.68 Mbps

Selective Repeat:

RTT/Window size	5	10	15
25	13.33s 1.96 Mbps	6.91s 3.78 Mbps	4.60s 5.67 Mbps
50	24.93s 1.05 Mbps	12.41s 2.10 Mbps	8.34s 3.13 Mbps
100	47.31s 0.55 Mbps	23.91s 1.09 Mbps	15.91s 1.62 Mbps

To begin with, our expectations for all three reliability functions in this scenario would be approximately the same as for test-case 1, but with an even quicker file transfer time and an even higher throughput when the roundtrip time is set to 4RTT. We particularly expect the results to be more effective than the original timeout of 0.5s.

Turning to the results of our implementation, we observe noticeable improvements in both transfer time and throughput with the 4RTT adjustment. Only with a slight difference in throughput in all three reliability functions.

This enhancement in performance can be attributed to the fact that a larger timeout value allows for more flexibility in the face of infrequent network conditions. It reduces the likelihood of unnecessary retransmissions, which can often be the result of prematurely assuming packet loss due to a small timeout value. As a result, the network assets are more efficiently utilized, leading to faster file transfers and higher throughput.

5.6 *Bonus task* - Simulating packet loss, reordering and duplicate packets: tc-netem

To get an even better understanding of how the different protocols behave under different conditions, we used the Linux tool "tc-netem". This tool can model real-life network behavior more accurately than our artificial test cases, and allows us to test with even higher rates.

It is also worth noting that for the purpose of making the output files more readable, print statements have been removed from the code. This step ensures a cleaner presentation of results, focusing solely on the performance and outcomes of the experiment. Additionally, all the tc-netem tests were conducted with a timeout value of 4RTT.

5.6.1 Packet loss

To test packet loss using tc-netem, we first deleted the queuing discipline for h1 before adding a new discipline with a delay of 6.25ms and loss rates of 1% and 10% to test under different conditions. To do this, we wrote the following lines in the Mininet terminal:

```
mininet> h1 tc qdisc del dev h1-eth0 root
mininet> h1 tc qdisc add dev h1-eth0 root netem delay 6.25 loss 1%
```

5.6.1.1 Results

GBN:

Window size	**1%**	**10%**
5	20.99 s 1.24 Mbps	154.62 s 0.17 Mbps
10	14.59 s 1.79 Mbps	137.01 s 0.19 Mbps
15	13.13 s 1.99 Mbps	134.48 s 0.19 Mbps

SR:

Window size	1%	10%
5	21.68 s 1.20 Mbps	102.44 s 0.25 Mbps
10	18.66 s 1.40 Mbps	79.15 s 0.33 Mbps
15	12.69 s 2.06 Mbps	64.36 s 0.41 Mbps

When testing packet loss with tc-netem, we expected a decrease in throughput and increase in transmission time in comparison to our artificial test cases. This is because tc-netem has the ability to simulate more accurate packet loss and because it simulates losing more than just one packet. In addition, we anticipate the SR protocol to outperform GBN due to its ability to selectively acknowledge and retransmit packets.

Upon evaluating the results, we observe a marginal difference between the Go-Back-N (GBN) and Selective Repeat (SR) protocols under a 1% loss rate, with SR demonstrating slightly better results. This contrasts with our findings in test case 3 ("skip_seq"). In that scenario, GBN unexpectedly outperformed SR. We believed that this was due to the fact that "skip_seq" only skips a single packet, which resulted in a negligible difference between the two protocols.

By using tc-netem rather than our artificial test cases, we are able to more comprehensively assess the performance of GBN and SR under varied conditions. This provides a more detailed understanding of the functional differences between the two protocols. Notably, under a higher loss rate of 10%, SR significantly outperforms GBN. This demonstrates that the advantage of SR's selective acknowledgement and retransmission becomes more pronounced under conditions of higher packet loss.

5.6.2 Reordering

When testing for reordering in tc-netem, we first deleted the queuing discipline again and added a new discipline of 6.25ms delay and 25% reordering to the router:

```
mininet> r2 tc qdisc del dev r2-eth1 root
mininet> r2 tc qdisc del dev r2-eth0 root
```

```
mininet> r2 tc qdisc add dev r2-eth1 root netem delay 6.25ms reorder 25% 50%
mininet> r2 tc qdisc add dev r2-eth0 root netem delay 6.25ms reorder 25% 50%
```

5.6.2.1 Results

GBN:

Window size	Transfer time	Throughput
5	62.12 s	0.42 Mbps
10	58.99 s	0.44 Mbps
15	58.00 s	0.45 Mbps

SR:

Window size	Transfer time	Throughput
5	14.33 s	1.82 Mbps
10	6.94 s	3.76 Mbps
15	4.55 s	5.73 Mbps

Test case 3 simulates packet loss and out-of-order delivery by deliberately skipping to send a packet. The 'out-of-order delivery' part can be interpreted as a form of reordering, but it's more of a packet skipping rather than changing the sequence of the packets. Therefore, tc-netem's reordering is a better tool for reordering.

By looking at the results, we see a significant difference between the two protocols. GBN is showing a slight increase in performance with higher window sizes, but overall very similar results. SR, on the other hand, outperformed GBN by a lot, with much better results. This is due to SR's ability to handle out-of-order packets.

5.6.3 Duplicate packets

Lastly, to test duplicate packets, we again deleted the previous queuing discipline on r2 and replaced it with the following:

```
mininet> r2 tc qdisc del dev r2-eth1 root
mininet> r2 tc qdisc del dev r2-eth0 root
mininet> r2 tc qdisc add dev r2-eth1 root netem delay 6.25ms duplicate 10%
mininet> r2 tc qdisc add dev r2-eth0 root netem delay 6.25ms duplicate 10%
```

5.6.3.1 Results

Stop-and-wait:

Transfer time	Throughput
1.53 s	17.07 Mbps

GBN:

Window size	Transfer time	Throughput
5	13.69 s	1.91 Mbps
10	7.07 s	3.69 Mbps
15	4.59 s	5.68 Mbps

SR:

Window size	Transfer time	Throughput
5	13.57 s	1.92 Mbps
10	6.78 s	3.85 Mbps
15	4.56 s	5.73 Mbps

In this final test case, we utilized the tc-netem tool to simulate duplicate packets, applying a duplication rate of 10%. This implies that 10% of the packages will be resent as duplicates. In this case, we again notice the strange behavior of Stop-and-Wait with duplicates. Earlier in test-case-4, we saw that sending one duplicate packet resulted in twice as high throughput. Given the above results, we see that the throughput is extremely high and better than any of the other protocols. As mentioned earlier, we suspect that this is due to a limitation in the Stop-and-Wait protocols handling of duplicate packets.

Comparing SR and GBN, we observe that the results are very similar, with SR performing somewhat better. Despite the higher percentage of duplicated packets in this test case compared to test-case-4, the outcomes are remarkably alike. This indicates that both protocols effectively discard duplicate packets as intended, resulting in minimal impact on the overall throughput.

6 Conclusions

In conclusion, our experimental arrangements have provided a robust comparative analysis of different transport protocols, specifically Stop-and-Wait, Go-Back-N (GBN), and Selective Repeat (SR). As anticipated, the Stop-and-Wait protocol demonstrated the least optimal performance, confirming its limitations in terms of efficiency. This can be attributed to its fundamental design that requires the sender to wait for an acknowledgment of each packet before transmitting the next one.Which in this case resulted in a much higher transfer time than both GBN and SR protocol.

In contrast, both GBN and SR displayed significantly superior performance, largely due to their implementation of the sliding window mechanism. This technique allows for multiple packets to be transmitted at once without requiring immediate acknowledgment, thus leading to better utilization of the network bandwidth and increased efficiency.

However, the choice between GBN and SR isn't straightforward and depends on a variety of factors such as network conditions, error rates, and specific application requirements. While both these protocols outperformed Stop-and-Wait, their relative performance against each other varied based on the conditions under which they were tested.

Overall, the testing phase provided us with valuable insights into the challenges and considerations involved in implementing a reliable transport protocol over UDP and designing a file transfer application that utilizes this protocol. In particular, it has shown the importance of considering network conditions and error rates while choosing a transport protocol, as these factors can significantly influence the performance of the system. The results of our testing highlighted areas for improvement and further experimentation, which will enable us to develop a more efficient and precise system in the future.

7 References

Ross, K. & Kurose, J. (2022). *Computer networking: a top-down approach.* Pearson Education.

Wright, G. & Froehlich, A. (2023, April) Transport Layer: https://www.techtarget.com/searchnetworking/definition/Transport-layer

Javatpoint. Difference between Stop and Wait, Go-Back-N, and Selective Repeat Website name (5.5.23): https://www.javatpoint.com/difference-between-stop-and-wait-gobackn-and-selective-repeat

man7. tc-netem(8) — Linux manual page: https://man7.org/linux/man-pages/man8/tc-netem.8.html (15.5.2023)

cs.unm. Traffic Control Manual For Lab1https://www.cs.unm.edu/~crandall/netsfall13/TCtutorial.pdf (15.5.2023)