Portfolio 2 - DRTP

(DATA2410 Reliable Transport Protocol)

Names and IDs:
Richard Johansen (s362063)
Joakim Hval (s362112)
Markus Einan (s315297)
Hamed Hussaini (s362086)

Table of contents

1. Introduction

In today's digital age, the ability to transfer large files quickly and efficiently has become essential for individuals and organisations alike. The task at hand is to explain the workings of a file transfer application and how it solves the problem of transferring large files.

The key topics that will be covered in this report include the various file transfer protocols, their strengths, and weaknesses, and how they are implemented in our file transfer application. Our focus is on providing a solution for sending data reliably, ensuring that data is delivered in the correct order, and handling missing data or duplicates effectively. We aim to create a robust and efficient file transfer mechanism capable of managing large files without susceptibility to data loss or corruption. We will explore how to use application.py to send data over UDP and discuss the benefits of using application.py to improve network transfer. File transfer applications employ a variety of protocols and techniques to guarantee that the transfer is fast, reliable, and secure.
Our approach to solving this problem involves conducting performance tests. We will use different testing scenarios to generate a variety of results, analyse these results, and present our findings. References to the relevant tools will be included in this report.

This report is organised as follows: The next section provides background information on performance testing and our file transfer application. Our approach and methodology are discussed in section three. In section four, we present and analyse our findings as test results, structured by task from 4.1 (task 1) to 4.6 (task 6). Finally, we conclude with a summary of our work and suggestions for potential future research and expansion.

Limitations and outcomes of our approach will be discussed in detail in this report. The file transfer application is a tool that allows users to transfer files quickly and easily between devices over a network.

2. Background

Reliability in file transfer means that files sent from one device arrive at another device correctly, without any loss or changes. This is achieved by using various methods that check and correct errors during the transfer. These methods are used in protocols like Stop-and-Wait, Go-Back-N (GBN), and Selective Repeat.

Stop-and-Wait
This protocol sends one packet at a time. After each packet is sent, it waits for a confirmation from the receiver before sending the next one. If the confirmation doesn't arrive, the sender sends the packet again. This method is simple, but not very fast.

Go-Back-N (GBN)
This protocol allows for sending several packets at once, which makes it faster than Stop-and-Wait. But if a packet gets lost or corrupted, the sender has to resend that packet and all the ones that followed it.

Selective Repeat
Like GBN, this protocol can send multiple packets at once. But if a packet gets lost or corrupted, only that packet is resent. The receiver can sort and reorganise packets, so it's more efficient but also more complex.

Choosing the right protocol depends on what the system needs, such as the acceptable error rate, the network conditions, and how many resources the system has. Our goal with this project is to implement these three reliability functions, and discuss the differences between them.

2.1 Teamwork

The Agile Scrum methodology, which we all learned as part of another course in our curriculum, is widely used in businesses for its iterative approach to project management that emphasises teamwork, accountability, and customer satisfaction. It's prevalent in software development and many other industries. It involves a set of practices and principles aimed at delivering high-quality products by empowering cross-functional teams to collaborate and adapt to change. Our four-member group adopted this methodology for our project, dividing tasks based on our strengths and interests to improve productivity and efficiency.

We began the project by creating a backlog of tasks and user stories, prioritising them, and assigning them to team members. We managed these tasks through a ticket system, which facilitated progress monitoring and issue identification. Each ticket represented a specific project feature or improvement. This system helped us to remain focused on completing tasks one at a time and avoid multiple people working on the same functionality simultaneously. We delegated tasks based on each member's strengths and expertise, ensuring that everyone worked on tasks they excelled at and enjoyed.

We held weekly stand-up meetings for progress updates, obstacle discussions, and future task planning. These meetings enabled quick issue identification and resolution, and kept us updated on each other's progress. We communicated regularly and effectively using tools like Slack, Zoom, and Discord, and ensured code quality through pair programming and code reviews.

An example of a ticket is:
“Adjust the stop and wait on the sender side to handle skipped acknowledgments.”

Tickets like this allowed for an equitable distribution of tasks, enabling us to focus on one thing at a time.

Overall, adopting the Agile Scrum framework helped us stay organised, focused, and motivated. We were able to break down the project into manageable tasks, collaborate effectively to complete them, and deliver a high-quality product on time.

2.2 The Technologies and Terminology we use

In this report, we will discuss the testing and troubleshooting of a network application using various technologies. In the following sections, we will explain how we used TC-Netem, ping, along with other technologies, to test and troubleshoot our application.

Ping

Ping is a commonly used tool that is used to test connectivity between two hosts, it can also be used to check if a network is reliable in the terms of detecting loss of packets and duplication of packets over a network.

UDP (User Datagram Protocol)

Is a connectionless transport protocol that operates on top of IP (Internet Protocol). It provides a simple, low-overhead, and unreliable delivery of datagrams between hosts.

DRTP - header

In a data transfer over a network with a reliable transfer protocol, the first part of a packet has a specific amount of bytes reserved to fit a header. This header is used to define what type of packet is being sent. A header could consist of different flags. For example, a SYN flag to establish a connection, an ACK flag to signal that a packet was received, a Seq number to inform the receiver of which packet has been sent or a FIN flag to close a connection. Without a header containing such information it would be hard to establish a reliable way to transfer data across a network.

TC-Netem

One of the critical technologies we used in this project was TC-Netem, an open-source module built into the Linux kernel that emulates network conditions such as latency, packet loss, and bandwidth restrictions. TC-Netem is a powerful tool that allows users to emulate various network conditions to test the performance of their applications or systems. It is a part of the Linux Traffic Control (TC) subsystem, which is responsible for shaping and controlling network traffic.
The primary purpose of TC-Netem is to simulate different network conditions, such as latency, packet loss, jitter, and bandwidth limitations, in order to observe how network traffic behaves under these conditions. This is particularly useful for us who need to test our application under real-world conditions.

To test the network application, we first set up a virtual network environment using TC-Netem. We created a network environment that emulated real-world conditions, such as packet loss and latency. We used the TC-Netem command line interface to specify the network impairments we wanted to introduce and applied them to specific network interfaces and devices.

Using TC-Netem and other technologies, we were able to identify and validate our application. For example, we discovered that the network application was not handling packet loss correctly. We were able to use TC-Netem to simulate packet loss and test the application's ability to recover from these losses.

3. Implementation / code explanation

Our file transfer application is a software program that allows users to transfer files from one device to another over a network, such as the internet. The application typically consists of two components: a client and a server.

The client is the software that is installed on the user's device and is used to initiate file transfers. The server is the software that runs on the destination device and receives the files sent by the client.

To transfer a file, the user first opens the client application and selects the file they want to transfer. They then specify the destination device, usually by entering an IP address or hostname, and start the transfer.

Once the transfer is initiated, the client breaks the file down into smaller chunks, known as packets, and sends them over the network to the server. The server then receives the packets and reassembles them into a new file, identical to the original file.

3.1 The topology

The virtual network/topology used in this code consists of three nodes: h1, r2, and h3. These nodes are connected through two links, one between h1 and r2, and another between r2 and h3.

The link between h1 and r2 has a bandwidth of 100 Mbps and a delay of 5 ms. The link also has no packet loss or queuing delay. The parameters for this link are set using the addLink method with the params1 and params2 arguments.

The link between r2 and h3 has the same bandwidth and delay as the link between h1 and r2. However, it has a maximum queue size of 170 packets, which is set using the max_queue_size argument. This link also has no packet loss or queuing delay.

h1 is assigned the IP address 10.0.0.1/24, r2 is assigned 10.0.0.2/24 and 10.0.1.1/24, and h3 is assigned 10.0.1.2/24. These IP addresses are set using the params1 and params2 arguments in the addLink method.

The LinuxRouter class is used to enable IP forwarding on r2, allowing packets to be forwarded between the interfaces on the router.

The network topology is tested using the Mininet network emulator, which simulates a network of virtual hosts, switches, and routers. The CLI method is used to launch the Mininet CLI, which allows the user to interact with the virtual network and test various network configurations and applications. The net.pingAll() method is called to test the connectivity between all nodes in the network.

3.2 The DRTP (Data2410 Reliable Transfer Protocol) application

The DRTP protocol is implemented in Python, using the socket library for network communication. The code is organised into two main files: simple-topo.py and application.py.

The simple-topo.py file is a script that sets up a simple network topology using the Mininet network emulator. It creates two hosts (client and server) and connects them using a simple point-to-point link.

The application.py file is the main implementation of the DRTP protocol. It contains the following functions:

create_packet(), parse_header(), parse_flags()

The create_packet() function creates a packet with a header and data. The header includes sequence number, acknowledgement number, flags, and window size. When a file is received the parse_header() function is called. This function unpacks the header of a received packet. Furthermore, after the header is unpacked the program calls the parse_flags() function, where the flags field of the header is parsed.

stop_and_wait_sender(), stop_and_wait_receiver()

The stop_and_wait_sender() function implements the stop-and-wait protocol for the sender side. It reads data from a file and sends it over the network in packets, waiting for an ACK for each packet before sending the next one. The stop_and_wait_receiver() function receives the sent data, after the data has been received it writes the content to file, and sends back an ACK response to inform the sender that the packet has arrived.

go_back_n_sender(), go_back_n_receiver()

For the implementation of the Go-Back-N protocol we have used two functions, the go_back_n_sender() for the sender side and go_back_n_receiver() for the receiver side. The

go_back_n_sender function sends multiple packets (within a given window) and awaits an acknowledgment for each packet. If a timeout occurs on one of the packets, the lost packet and all packets with a higher sequence number gets retransmitted. In the go_back_n_reciever function packets that have been sent are received. Furthermore, the function sends an acknowledgment for each packet that has been received. If a packet is received out of order, this packet gets discarded, and no acknowledgement is being sent.

selective_repeat_sender(), selective_repeat_receiver()

In the implementation of the Selective Repeat protocol we also used two functions, the selective_repeat_sender() for the sender side, and the selective_repeat_receiver() for the receiver side. The selective_repeat_sender() function also sends multiple packets (within a given window) and awaits an acknowledgment for the packets sent. If a timeout occurs, it retransmits the lost packet. In the selective_repeat_receiver() function sent packets are received. if an out of order packet is received it gets placed in a buffer while waiting for the expected sequence number before writing the buffered packets to file.

main(), run_server(), run_client()

The main() function is the entry point of this application. It parses command line arguments and runs either the client or the server based on the provided flags. If the server is invoked the chosen arguments get sent into the run_server() function. This is where a server-side application is set up. The function creates a UDP socket and waits for a client to try and connect. If a client connection is established the server invokes one of the reliability functions where it will start to receive a file transferred from the client-side. If the client is invoked the chosen arguments get sent into the run_client() function. This is where a client-side application is set up. The function creates a UDP socket and tries to establish a connection to the ip-address and port given. When a connection is established, the client invokes one of the reliability functions where it starts sending a given file to the connected server.

Communication between Server and Client

The DRTP protocol uses UDP as the underlying transport protocol for communication between the server and client. When a file transfer is initiated, the client sends a SYN packet to the server, which responds with a SYN-ACK packet to confirm the connection, this is called a 3-way handshake. The client then sends data packets containing the file contents, and the server sends acknowledgment packets in response.

DRTP supports three different reliability functions. To provide reliability, the DRTP protocol implements three different ARQ mechanisms: Stop-and-Wait, Go-Back-N, and Selective Repeat. The specific mechanism to use can be specified through a command-line argument when invoking the server and client applications.

Stop-and-Wait

The client, using the stop_and_wait_sender() function, reads data from a file, creates a packet using create_packet(), and sends it over the network to the server.

After each packet is sent, the client waits for an acknowledgement (ACK) from the server before proceeding to send the next packet. This ensures that every packet is received and acknowledged by the server, albeit at the cost of slower transmission rates due to the waiting period.

The server, using the stop_and_wait_receiver() function, receives the packet, parses the header using parse_header(), and sends an ACK back to the client for each packet. The server also writes the received data to a file.

Go-Back-N

In the Go-Back-N scenario, the client uses the go_back_n_sender() function to send multiple packets without waiting for an ACK for each one. This is achieved by maintaining a window of packets that can be unacknowledged at any given time.

If a timeout occurs (i.e., an ACK is not received within a specified period), the client will retransmit all unacknowledged packets. This means that if the server misses a single packet, the client will re-send all packets that were in the window at the time of the missed packet.

The server, with the go_back_n_receiver() function, receives packets and sends ACKs back to the client. If it receives a packet out of order, it discards it and waits for the retransmission from the client.

Selective Repeat

In the Selective Repeat mode, the client uses the selective_repeat_sender() function to send packets. Like in Go-Back-N, multiple packets can be sent without waiting for an ACK for each one, and a sliding window is used to control the flow of packets.

However, in contrast to Go-Back-N, when a timeout occurs, the client only retransmits the specific packet for which the ACK was not received. This can be more efficient than Go-Back-N, as it avoids unnecessary retransmissions.

The server uses the selective_repeat_receiver() function to receive packets, send ACKs back, and write the data to a file. If the ACK for a certain packet is lost, the client will retransmit the corresponding packet.

In each of these scenarios, the run_client() function sets up the client-side application, creates a UDP socket, establishes a connection with the server, sends a file based on the specified reliability mode, and tears down the connection. Similarly, the run_server() function sets up the server-side application, receives a file, and tears down the connection.

The main() function acts as the entry point of the script, parsing command line arguments and deciding whether to run the client or server based on the provided flags.

4. Discussion and performance evaluations

Throughout this project, we conducted various tests and evaluations to assess the performance and efficacy of our implementation. We analysed the throughput values for different reliable methods and window sizes, considering various round-trip times (RTTs) to understand their impact on data transfer. Additionally, we designed test cases to simulate scenarios such as skipped acknowledgments and out-of-order delivery to evaluate the robustness of our solution.

In this discussion, we will present our findings and analysis based on these tests and test cases. We will discuss the calculated throughput values for different configurations and provide insights into the results. Furthermore, we will examine the effects of skipped acknowledgments and out-of-order delivery on the GBN and SR reliable methods, reporting on the behaviour and performance of our implementation in handling such scenarios.

By fulfilling the requirements of this project and thoroughly evaluating our solution, we aim to demonstrate the effectiveness of our DRTP implementation in providing reliable data transfer over UDP. Through our discussion, we will provide a comprehensive understanding of the performance characteristics, strengths, and limitations of the implemented protocols and their applicability in real-world network environments.

4.1 Measuring throughput with stop_and_wait, GBN and GBN-SR

In Task 1 of the portfolio, we tested the file transfer application using different reliable protocols: stop-and-wait, Go-Back-N (GBN) with window sizes of 5, 10, and 15, and Selective-Repeat (SR) with window sizes of 5, 10, and 15. We also varied the round-trip times (RTTs) between 25ms, 50ms, and 100ms. The objective was to calculate the throughput values for each case and provide an explanation of the results. Throughput is a measure of the amount of data that can be transmitted over a network in each time period. It is influenced by several factors, including the protocol used, window size, RTT, and the presence of packet loss or reordering.

Test Results

Protocol	RTT 25ms	RTT 50ms	RTT 100ms
Stop and Wait	0.44 Mbps	0.23 Mbps	0.11 Mbps
Go-Back-N (window size 5)	2.11 Mbps	1.13 Mbps	0.57 Mbps
Go-Back-N (window size 10)	4.39 Mbps	2.26 Mbps	1.14 Mbps
Go-Back-N (window size 15)	6.03 Mbps	3.31 Mbps	1.67 Mbps
Selective Repeat (window size 5)	2.17 Mbps	1.1 Mbps	0.57 Mbps
Selective Repeat (window size 10)	4.3 Mbps	2.23 Mbps	1.14 Mbps
Selective Repeat (window size 15)	6.42 Mbps	3.27 Mbps	1.67 Mbps

Throughput is a measure of the amount of data that can be transmitted over a network in each time period. It is influenced by several factors, including the protocol used, window size, RTT, and the presence of packet loss or reordering. The impact of RTT on throughput is evident in the results. A shorter RTT generally leads to higher throughput because the sender can receive acknowledgments and adjust its transmission rate more quickly. Conversely, a longer RTT introduces additional delays in the communication, which can limit the throughput. Therefore, in our experiments, the case with an RTT of 25ms yielded higher throughputs compared to 50ms and 100ms.

The stop-and-wait protocol, being a simple one, has the lowest throughput among the tested protocols. This is because it operates on a send-wait-receive paradigm, where the sender must wait for an acknowledgment before sending the next packet. As a result, the effective transmission rate is limited, and the throughput is constrained. It can be observed that the data transfer rate decreases as the RTT value increases. Creating double the round-trip-time creates longer transfer time, which leads to a slower data transfer rate.

On the other hand, the Go-back-n and Selective repeat protocols outperform the Stop and Wait protocol with much higher data transfer rates. As the window size increases, the data transfer rate increases accordingly. This is because both protocols allow multiple packets to be sent before receiving an acknowledgement, which results in faster data transfer.

Furthermore, it can be observed that the Selective repeat protocol generally performs slightly better than the Go-back-n protocol with the same window size and RTT value. This is because the Selective repeat protocol only retransmits lost packets, while the Go-back-n protocol retransmits every packet sent after the lost packet, which results in higher overhead and slower data transfer rate.

In conclusion, the results show that the choice of protocol and window size can have a significant impact on the data transfer rate in a network. The Go-back-n and Selective repeat protocols with larger window sizes provide better data transfer rates, and the Selective repeat protocol is generally more efficient than the Go-back-n protocol. These findings are consistent with previous studies on network protocols and provide insights for optimising data transfer in a network.

Discussion

The results show that the performance of different error control schemes varies significantly with respect to the round-trip time (RTT) and window size. Specifically, the stop-and-wait scheme has the lowest throughput for all RTT values, while the go-back-n and selective repeat schemes have higher throughput. GBN and SR protocols with larger window sizes exhibit higher throughputs compared to stop-and-wait. Increasing the window size allows the sender to transmit multiple packets before receiving acknowledgments, effectively increasing the transmission rate. As a result, the throughput improves with larger window sizes.

These results are in line with what we theorised for the data transmission. This is because the stop-and-wait scheme is known to have low efficiency because it transmits only one packet at a time and waits for an acknowledgment before sending the next packet. In contrast, go-back-n and selective repeat schemes can transmit multiple packets before receiving an acknowledgment, which makes them more efficient. However, as the window size increases, the probability of packet loss also increases, which can lead to retransmissions and longer delays. Also, as the window size increased we expected around twice the throughput rate for 5 to 10, and this was shown to be true for all results. From 10 to 15 we expected a half (50%) increase in throughput because 5 extra would be 50% of 10. This was also shown to be the case, as we theorised, because all answers from 10 to 15 showed an increase of around 50%.

The results suggest that the choice of error control scheme and window size should be made based on the specific requirements of the system, such as the desired throughput and delay, as well as the characteristics of the network, such as the RTT and error rate. In general, larger window sizes can improve throughput, but they also increase the probability of packet loss and retransmissions.

It is important to note that the results are based on a simulation, and the actual performance of the error control schemes may vary depending on the specific network conditions and hardware used. Future studies can extend this work by evaluating the performance of different error control schemes on real-world networks and hardware.

In conclusion, the analysis of throughput values for different reliable protocols and varying RTTs provides insights into their performance characteristics. GBN and SR with larger window sizes offer higher throughputs compared to stop-and-wait, and shorter RTTs generally result in better throughput. The findings can also guide the selection of error control schemes for specific applications and network conditions.

4.2 Skipping ack to trigger retransmission

The second task involves writing a test case to skip an ACK, which triggers retransmission, and testing it with all three reliable functions: stop_and_wait, GBN, and SR. This test case allows us to evaluate the behaviour of the protocols when an acknowledgment is skipped, leading to the retransmission of packets. When the ACK for a packet is skipped, it creates a scenario where the sender does not receive the acknowledgment and assumes that the packet was lost in the network. Consequently, the sender retransmits the packet to ensure reliable and in order delivery. This test case helps us assess the effectiveness of the reliability functions in handling such scenarios and ensuring successful data transfer.

Test Results

Protocol	**Stop and wait**	**GBN**	**SR**
Packets sent	98	98	98
Packets received and acknowledged	98	98	98
Discarded/Out-of-Order Packets	0	36	0
Retransmitted Packets	9	45	9
Total Time Transferred (seconds)	7.26	5.56	5.79
Total Bits Transferred	1143096	1143096	1143096
Throughput (Mbps)	0.16	0.21	0.20

All three reliability methods have their own way of handling losses, here is how they each do it:

Stop and Wait Protocol (stop_and_wait()):
The sender successfully sent packets 1 to 4 without any issues. When running our test case with the stop_and_wait protocol. However, the ACK for packet 5 was lost, triggering a timeout at the sender's side. The sender retransmitted the packet 5. Eventually, all packets were successfully transmitted, and the file transfer was completed.

Go-Back-N (GBN()):
With a window size of 5, the sender sent packets 1 to 5 without any issues. The ACK for packet 5 was lost, resulting in a timeout and retransmission of packet 5 and the out of order packets. The sender successfully retransmitted the lost packet and the out of order packets and continued the file transfer. The total time taken for the transfer was recorded.

Selective Repeat (SR()):
With a window size of 5, the sender sends packets 1 to 5. The ACK for packet 5 was lost, triggering a retransmission packet 5. The sender successfully retransmitted the lost packet and continued sending packets. After the successful retransmission, out of order packets that had been buffered got written to file in the correct order. The total time taken for the transfer was recorded.

The stop and wait protocol always waits for an ack of the last sent packet before sending the next packet sequence. Whenever a packet is sent to the receiver a timeout timer is started while the sender awaits an ack. If an ack is skipped or lost the timeout will be reached and triggered an exception, making the sender retransmit the same packet with a new timeout timer.

In the Go-Back-N protocol, a window size of 5 was used. The client successfully transferred all 98 packets, but some packets throughout the file transfer did not receive an ACK from the receiver. In

the Go-Back_N protocol all packets received out of order are to be discarded and retransmitted in the correct order. In this case even if a packet is received, the ACK for this packet might have been lost on the way back to the sender. This will eventually trigger the timeout timer which will then trigger the retransmission of the packet that did not receive an ACK and retransmit all the packets sent after the lost packet. Furthermore, when the receiver receives the packet where the ACK got lost, it will discard all the out of order packets that have a higher sequence number.

In the Selective Repeat protocol, a window size of 5 was used. The client successfully transferred all 98 packets, but some packets throughout the file transfer did not receive an ACK from the receiver. In the Selective Repeat protocol all packets received out of order will be buffered in the receiver. In this case even if a packet is received, the ACK for this packet might have been lost on the way back to the sender. This will eventually trigger the timeout timer which will then trigger the retransmission of the packet that did not receive an ACK. Furthermore, when the receiver receives the packet where the ACK got lost, it will write the received packet to the file and the current window gets updated. After this the receiver will check the buffer for the next expected sequence packet, and if the next sequence packet exists it will write this packet to file.

Discussion

Throughout this task we have written a test case to check if our program is reliable even though an ack from the server is lost under a file transfer. When implementing this artificial test case, we decided that the program was to skip the sending of an ACK message every fifth sequence number received on the server until a specified number of skipped ACK's had been reached. We chose to have the same requirements for the test case for all three protocols so that we could easily compare the results we received after running this test.

We first evaluated the performance of the stop_and_wait protocol in this test case. As the name suggests, the stop_and_wait protocol operates by sending one packet at a time and waiting for the acknowledgment before sending the next packet. Therefore, when an ACK is skipped, it introduces a delay in the transmission process. When we looked at the results after performing this test, we noticed that the throughput was lower, and the total time transferred was higher than the others. This correlates with the theory that stop_and_wait takes longer to run. Even though this protocol wasn't the most efficient we noticed that the protocol was good when it came to securing in order delivery, making it a good option if you would want a secure method that ensures that a file transfer is sent and received in the correct order.

Based on the calculated time and throughput measurements from the different protocols, the time and throughput values from the stop_and_wait protocol differentiates itself from the two others. This is because the stop_and_wait protocol's functionality works differently than the Go-Back-N and Selective Repeat protocols which work quite similarly. We can see that all the protocols were able to transfer the file correctly by looking at the number of bits transferred. It is when it comes to the efficiency of the protocols the differences begin to show. This is because the Go-Back-N and the Selective Repeat protocols simultaneously send multiple packets which results in the protocols using less time to transfer all the packets in the file transfer.

When comparing the results for Go-Back-N towards the results for Selective Repeat, there is a big difference when it comes to the amount of discarded and retransmitted packets. The Go-Back-N protocol discarded 36 out of order packets and retransmitted 45 packets compared to the Selective Repeat protocol where 0 packets were discarded and only 9 packets were retransmitted. This happened because these two protocols handle losses differently. The Go-Back-N protocol discards every packet received out of order and has to retransmit these packets, where the Selective Repeat protocol simply buffers the out of order packets and waits to receive the retransmitted packet before writing all packets to file in order.

Our results when running each protocol with our test case we can see that all protocols successfully managed to transfer all packets of a file and deliver them in order between a sender and receiver. As we expected there was a big difference in the performance between the

stop_and_wait protocol compared to the Go-Back-N and the Selective Repeat protocols, when it comes to the time taken and the throughput of the file transfer.

4.3 Skipping a sequence number to show out-of-order effect

In this task, we designed a test case to simulate the skipping of a sequence number, which triggers out-of-order delivery of packets. We evaluated the performance of the Go-Back-N (GBN) and Selective Repeat (SR) reliability functions for the handling of this scenario.

To test the effect of skipping a sequence number, we intentionally dropped a packet with a specific sequence number at the network level. This caused the receiver to receive packets out of order, as the skipped packet was not delivered. As a result, the receiver buffer contained packets with sequence numbers that were higher than expected, disrupting the expected ordering.

Test Results

Protocol	GBN	SR
Packets sent	98	98
Packets received and acknowledged	98	98
Discarded/Out-of-Order Packets	36	0
Retransmitted Packets	45	9
Total Time Transferred (seconds)	5.54	5.76
Total Bits Transferred	1143096	1143096
Throughput (Mbps)	0.21	0.20

In the Go-Back-N protocol, a window size of 5 was used. The client successfully sent all 98 packets, but some packets throughout the file transfer were lost which resulted in some packets getting received out of order. In the Go-Back_N protocol all packets received out of order are to be discarded and retransmitted in the correct order. In this case even if only one single packet is lost, all the packets sent after the lost packet also have to be discarded from the receiver and get retransmitted. This will result in the retransmission of multiple packets so that every packet is received in order.

The Selective Repeat protocol also used a window size of 5. The client also managed to successfully send all 98 packets. In the Selective Repeat protocol only the lost packet has to be retransmitted, this is because the receiver is able to buffer out of order packets. When a packet is lost and a timeout occurs the protocol simply retransmits the lost packet. When this packet is received, acknowledged and written to file, the receiver goes through the buffer with out of order packets and writes them to file in the correct order.

Discussion

These results demonstrate that both protocols were able to successfully transfer the file, but GBN had a slightly higher throughput compared to SR. However, we thought SR would have a lower retransmission rate, this did not show in the results. It did however complete the test case with no discarded or out-of-order packets. The choice between these protocols would depend on the specific requirements of the application, considering factors such as reliability and efficiency.

Furthermore, how different test scenarios are performed could also affect how these protocols perform when compared against one another. Here are some observations we made based on the results:

We first evaluated the performance of the GBN reliability function in this scenario. When the receiver encountered an out-of-order packet, it did not acknowledge any packets until the missing packet was received. This behaviour allowed the sender to retransmit the missing packet, ensuring the correct ordering of packets at the receiver. However, the downside of GBN is that it requires the retransmission of multiple packets even if only one packet is missing, leading to potential inefficiencies in the network. The performance of SR relies on the assumption that the network experiences limited packet loss and out-of-order delivery. If the network conditions are highly unfavourable, with frequent packet losses and significant reordering, SR may not be able to fully leverage its benefits. In such cases, GBN's simpler approach of retransmitting a fixed window of packets may be more effective.

Next, we tested the SR reliability function, which combines the benefits of GBN and selective acknowledgments. When the receiver encountered an out-of-order packet, it stored the packet in the correct position in the receive buffer. The receiver then sent a selective acknowledgment to inform the sender about the missing packet, allowing the sender to retransmit only the necessary packets. This approach minimised the number of retransmissions and therefore we would say it is better in this use case. Even though we thought SR would improve network efficiency compared to GBN, it did not show in the test results.

There are 2 reasons we think caused this, and another explained in task 5. The specific test scenario used to evaluate the performance of SR and GBN. If the test scenario does not fully stress the advantages of SR, such as having minimal out-of-order delivery or limited retransmissions, the performance differences between the two functions would not be significant. This was shown in the results to be the case. Also It's important to consider the possibility of sampling bias in the results. If the test cases used in the evaluation were not diverse enough or did not capture a wide range of network conditions, the results may not be representative of the overall performance characteristics of SR and GBN.

In conclusion, our implementation of the DRTP protocol with GBN and SR reliability functions successfully handled the scenario of skipping a sequence number and out-of-order delivery. The SR function demonstrated superior performance by efficiently reordering packets and minimising retransmissions. These findings highlight the importance of selecting an appropriate reliability function based on the specific requirements and constraints of the network environment.

4.4 How well is losses, reordering and duplicates handled

The interpretation of our gathered data leads us to the conclusion that the results from our tests demonstrate our solution's efficacy in handling various network challenges, including losses, reordering, and duplicates. The reliability functions implemented, namely Stop and Wait, Go-Back-N, and Selective-Repeat, exhibited successful performance based on the different scenarios we tested. This was evidenced by our previous tasks and results.

Test Results

Test Case	Reliable Function	Result
Loss	Stop and Wait	Success
Loss	Go-Back-N (5)	Success

Test Case	Reliable Function	Result
Loss	Go-Back-N (10)	Success
Loss	Go-Back-N (15)	Success
Loss	GBN-SR (5)	Success
Loss	GBN-SR (10)	Success
Loss	GBN-SR (15)	Success
Reordering	Go-Back-N (5)	Success
Reordering	Selective-Repeat	Success
Duplicates	Go-Back-N (5)	Success
Duplicates	Selective-Repeat	Success

Note: "Success" indicates that the solution was able to handle the respective scenario effectively.

The test results indicate that all three reliability functions were able to handle packet losses effectively, as provided by all the previous test scenarios. The Stop and Wait protocol showed reliable retransmission upon timeout, ensuring the successful delivery of data. The Go-Back-N strategy with varying window sizes (5, 10, 15) exhibited efficient retransmission and recovery from packet losses, maintaining the integrity of the file transfer. The Selective-Repeat approach, which combines the benefits of Go-Back-N and Selective-Repeat, also proved successful in handling losses.

Furthermore, the solution demonstrated the ability to handle out-of-order delivery. The Go-Back-N and Selective-Repeat functions effectively reordered the received packets and correctly placed them in the receive buffer, allowing for the accurate reconstruction of the file at the receiver side.

The solution also addressed the issue of duplicate packets. Both the Go-Back-N and Selective-Repeat functions detected duplicate acknowledgments and prevented unnecessary retransmissions, ensuring efficient and reliable data transfer.

Discussion

The performance of the implemented reliability functions aligns with the concepts and principles discussed in the lecture slides. The Stop and Wait protocol, although simple, provides reliability by employing timeout-based retransmission. The Go-Back-N strategy, with its sliding window approach, enhances throughput by allowing multiple packets to be in transit simultaneously. The Selective-Repeat approach further optimises performance by selectively accepting out-of-order packets, reducing unnecessary retransmissions and improving overall efficiency.

The implemented reliability function also considers the possibility of buffer overflow, which is a result of sending more data than the receiver is able to handle at once . All reliability functions implement flow control. stop_and_wait sends one packet at a time, while GBN and SR use a set window size that only advances when a new ACK packet is received. This prevents the sender from overloading the network with too much traffic that could result in loss of data.

Comparing our results with existing tools such as tc-netem, it can be observed that our solution demonstrates similar behaviour to well-established reliable transport protocols. The ability to handle losses, reordering, and duplicates effectively is essential in ensuring reliable data delivery over unreliable networks such as UDP.

Overall, the test results validate the efficacy of our solution in providing reliable file transfer over UDP using the implemented reliability functions. The combination of these functions offers flexibility and adaptability in handling various network conditions, contributing to the overall success and efficiency of the DATA2410 Reliable Transport Protocol (DRTP).

4.5 Using tc-netem to show efficacy of our code (bonus)

We used tc-netem to emulate packet loss, reordering, and duplicate packets to demonstrate the efficacy of our code. This allowed us to test the performance of our DRTP implementation under various network conditions which are commonly encountered in real-world networks, and to validate its reliability. This was to add an additional layer of testing to our application because we ourselves coded the test cases “skip ack” and “loss” that we used to test the data transfer. This way we make sure that our tests are implemented correctly, by using an external testing tool we validate that the results are matching to the command-line utility.

Results

Test case	Emulated Network Condition	Throughput
loss_netem_stop_and_wait_client.txt	15% Packet Loss	0.09 Mbps
loss_netem_gbn_client.txt	15% Packet Loss	0.11 Mbps
loss_netem_sr_client.txt	15% Packet Loss	0.19 Mbps
loss_netem_stop_and_wait_client.txt	50% Packet Duplicate	0.21 Mbps
dup_netem_sr_client.txt	50% Packet Duplicate	3.15 Mbps
dup_netem_gbn_client.txt	50% Packet Duplicate	3.14 Mbps
reorder_netem_stop_and_wait_client.txt	50ms Delay, 50% Packet Reorder	0.28 Mbps
reorder_netem_gbn_client.txt	50ms Delay, 50% Packet Reorder	0.13 Mbps
reorder_netem_sr_client.txt	50ms Delay, 50% Packet Reorder	0.07 Mbps

Commands for tc-netem simulation we used:

Loss	“sudo tc qdisc replace dev h1-eth0 root netem loss 15%”
Duplicate	“sudo tc qdisc replace dev h1-eth0 root netem duplicate 50%”
Reorder	“sudo tc qdisc replace dev h1-eth0 root netem delay 50ms reorder 50% 50%”

Snippets of test cases from Test_case_tc-netem directory:

- loss_netem_stop_and_wait_client.txt
  - creating packet 79
  - sending packet nr 79
  - timeout occurs
  - preparing to retransmit packet nr 79
  - sending packet nr 79

- ack received for packet 79

Discussion

Before we ran any tc-netem tests we figured it would be crucial to really understand how these test cases operated. Therefore, we ran some tests with the commonly used ping command to gain some knowledge about this technology. These tests gave us a good foundation to expect how the tc-netem test cases would affect the results we would get when running these tests on our application.

When we started testing with tc-netem on our application, we encountered some small problems:

- Loss of packets during connection establishment:
  If we ran tests with a high loss percentage, we sometimes lost a packet during the 3-way handshake between server and client side. This is because, in our implementation, the packet loss tests were started after the connection was established. But when we tested with tc-netem, this was not the case. The same thing sometimes occurred while trying to close the connection using a 2-way handshake, if one of those packets were lost.
- Securing that duplicate packets are handled correctly:
  The application was implemented to not respond to any duplicate packets at all and just discarded if there were any attempts. However, by running tc-netem with higher packet loss, it caused some problems at the receiver end which did not get any ack. So it ended up in an infinite loop.

In the stop and wait function, we've had a successful outcome compared to the artificial test we implemented ourselves. The results showed a similar packet loss handling process, where we checked for duplicate and out of order packets. Our application securely handled the packet loss and retransmitted in the event of any losses. This prevented us from ending up in an infinite loop at the receiver's end.

For both GBN and SR, we've implemented a similar validation handling process but within a larger window. In GBN, if there were any out-of-order packets, the receiver could handle it by comparing the sequence number with the expected number and discarding it in the case of being out of order. But in the select_repeat function, the receiver would acknowledge any unacknowledged packets in the buffer. If there was any duplicate attempt, the receiver would just send an ACK and not write it to the file, because the sender will keep trying to resend until it gets an ACK. To prevent receiving duplicate packets at the sender's end, we would ignore it.

When comparing the results from the generated tc-netem tests to our own artificial test cases, we can see a significant change in the correlation between GBN and SR regarding loss. In the artificial test cases that we created, GBN slightly outperformed SR with 0.01 Mbps when it came to the calculated throughput and time used to complete a file transfer. When we look at the results from the tc-netem test on the other hand, we can see that SR outperformed GBN with 0.08 Mbps. This result backs our initial thoughts that SR would outperform GBN if loss occurs. The reason why the tc-netem tests and our artificial test for loss gives different results is because they perform the same test differently. In our artificial test case we simulate loss by skipping a specific packet number. In this test environment GBN outperforms SR since the retransmission of the lost and out of order packets happens simultaneously, while SR still has to search through the buffer of out of order packets before continuing. In the tc-netem test it simulates packet loss by removing a specific percentage of the packets sent through the interface. This means that every out of order packet that gets retransmitted with GBN also has a chance to get lost, resulting in more loss than with SR where out of order packets get buffered instead of retransmitted.

Using tc-netem helped us to implement a more efficient and correctly responding mechanism for the reliable functions. This is because we had to make some additional adjustments to the code, for the tests to run properly.

4.6 Calculating per-packet roundtrip (bonus)

In our reliability application, if chosen, the “calculate timeout (-ct)” option is used for dynamically adjusting the timeout period for resending packets in case of lost acknowledgments. This is an optional feature that can be invoked by setting the -ct argument to True. When enabled, the application calculates the RTT for every packet that it sends. Following this calculation, the application adjusts the timeout period to four times the calculated RTT. The reason for this is because of the dynamic nature of network conditions. by setting the timeout to four times RTT we will reduce the number of premature timeouts and ensure a relatively good detection of lost packets.

How the RTT-based timeout adjustment process works in our application
The process of sending packets and measuring their Round-Trip Time (RTT) consists of several steps. First, the time when the sender sends a packet is registered. This registered time will be used later as a reference point for calculating the RTT. Next, the sender waits for an acknowledgement (ACK) from the receiver indicating the successful delivery of the packet. Upon receiving the ACK, the sender registers the time when the ACK is received. This registered time represents the moment when the sender becomes aware of the packet's successful delivery. To calculate the RTT, the sender subtracts the registered time of sending from the registered time when the ACK was received. This calculation contains the time it took for the packet to travel from the sender to the receiver, as well as the time for the ACK to travel back to the sender. Based on the calculated RTT, the sender adjusts the timeout for the next packet. The timeout is set to four times the recently calculated RTT. This adjustment ensures that the sender will wait for an ACK for the next packet for a period of four times the RTT before assuming that the packet is lost. This entire process of sending packets, receiving ACKs, calculating RTT, and adjusting the timeout is repeated for each packet sent. This approach enhances the efficiency of the data transfer process and ensures that the timeout set is not too high or low even if the network conditions change.

5. Conclusions

To conclude this report, the implementation of the DATA2410 Reliable Transport Protocol (DRTP) has been successfully achieved, providing reliable data delivery over UDP. The developed protocol ensures that data is reliably transferred in-order without missing data or duplicates. The file transfer application, consisting of a client and server, effectively utilises the DRTP protocol to transfer files between two nodes in a network. By constructing packets and acknowledgements, establishing connections, and gracefully closing transfers, our application ensures the reliable and orderly transfer of files without data loss or duplication.

Throughout our testing and evaluation, we have observed the effectiveness of our application in handling various scenarios, including packet loss, reordering, and duplicates. Three different reliability functions were implemented to do this: Stop and Wait, Go-Back-N (GBN), and Selective-Repeat (SR). These functions allowed for the evaluation of different strategies for reliable data transfer. The performance of each function was analysed using various parameters such as window sizes and round-trip times (RTTs).

Task 1 involved running the file transfer application with different reliability protocols and measuring the throughput values. The results demonstrated the impact of different strategies on the overall performance. Throughput values were influenced by factors such as window sizes and RTTs. The analysis of these results provided valuable insights into the efficiency and effectiveness of the implemented reliability functions.

Task 2 involved creating a test case to skip an acknowledgment, triggering retransmission. This test case was executed with all three reliability functions. By simulating a scenario where an acknowledgment is skipped, the ability of the protocols to handle retransmission and recover from potential data loss was evaluated. The performance of each function was observed and analysed.

Task 3 focused on testing the out-of-order delivery effect by skipping a sequence number. This test case was specifically executed with GBN and SR reliability functions. The results showcased the capability of the protocols to handle out-of-order packets and retransmit as necessary. The impact of reordering on the data transfer process was assessed, providing valuable insights into the robustness of the implemented solutions.

Additionally, artificial test cases were designed to assess the overall efficacy of the implemented solution. These test cases aimed to evaluate the performance of the protocols. However, it is important to acknowledge the limitations of our solution. While our application provides reliable and efficient file transfers, there may still be potential areas for improvement. Further research and development could focus on optimising performance and exploring additional reliability functions. By combining theoretical knowledge, practical implementation, and thorough testing, we have successfully achieved the goals set forth in this project. Our file transfer application serves as a testament to the effectiveness of the DATA2410 Reliable Transport Protocol and its ability to provide reliable and efficient file transfers.

Overall, the development of the file transfer application has been a valuable learning experience, allowing us to apply the concepts and techniques learned throughout the course. It has deepened our understanding of network protocols, reliability functions, and the challenges involved in file transfers. The completion of this project has provided valuable insights into the design, implementation, and evaluation of a reliable transport protocol. The project not only enhanced our understanding of network protocols but also provided hands-on experience in developing practical solutions for reliable data transfer. The successful implementation of DRTP and as a file transfer application showcases the importance, and impact of reliable transport protocols in modern network communications.

6. References

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

ping(8). (n.d.). https://man.freebsd.org/cgi/man.cgi?query=ping&sektion=8

Wikipedia contributors. (2023b). User Datagram Protocol. *Wikipedia*. https://en.wikipedia.org/wiki/User_Datagram_Protocol