PORTFOLIO 2

DATA2410 – OSLO metropolitan university

FAHMI HAYBE MOHAMMED / s362106, BEHDAD NIKKHAH / s362085, TERESA PHAM / s345368, JAN NICOLE RUBIC YAO / s362049

2023

TABLE OF CONTENTS

[1. Introduction 2](#_Toc134645617)

[2. Background 3](#_Toc134645618)

[3. Implementation 4](#_Toc134645619)

[4. Discussion 7](#_Toc134645620)

[Test Case 1 7](#_Toc134645621)

[Test Case 2 9](#_Toc134645622)

[Environment 9](#_Toc134645623)

[Steps 9](#_Toc134645624)

[Test Data 9](#_Toc134645625)

[Expected Results 9](#_Toc134645626)

[Actual Results 9](#_Toc134645627)

[Test Case 3 10](#_Toc134645628)

[Environment 10](#_Toc134645629)

[Steps 10](#_Toc134645630)

[Test Data 10](#_Toc134645631)

[Expected Results 10](#_Toc134645632)

[Actual Results 10](#_Toc134645633)

[Test Case 4 11](#_Toc134645634)

[Bonus TC-NETEM 12](#_Toc134645635)

[5. Conclusions 13](#_Toc134645636)

[6. References 13](#_Toc134645637)

# 1. Introduction

An introduction should tell the reader why this work is interesting.

This report contains an implementation of a simple transport protocol (DRTP) - Data Reliable Transport Protocol. This program is built using python and socket module with UDP to transfer data. Given that UDP is an unreliable protocol, DRTP will ensure a reliable connection. The network tool Mininet is also an important tool to build a virtual network regarding the simple topology. This will be useful to evaluate the virtual network resources and performance when using DRTP. (Include the bonus as an implemented work)

The test would be done by the custom written code to transfer a file, and to measure the throughput using different RTTs (round-trip-time). Different various functions will be used to trigger retransmission of packets either due to loss or being skipped. The performed tests should reveal if the code is handling potential losses, reordering and duplicates.

The approach to solve the problem is to implement a three-way handshake between the client and server as a way of establishing a connection. Each reliable function has its own client and server methods which results in a more maintainable and flexible code. This makes it easier to alter the different reliability functions as needed.

However, the custom reliable transfer protocol has some limitations. One big limitation is the fact that the protocol does not check packets for corruption. A packet is “corrupted” when some of the bits within the packet have been altered during transmission and the packet has been modified or damaged in some way. Corrupted packets can cause errors in data transfer, which can result in the loss or corruption of data being transmitted. Another important limitation is the fact that the protocol does not support congestion control. It does not adjust its transfer rate based on network conditions. This can lead to network congestion and degraded performance if the network becomes congested.

This document is structured and divided into five different segments, including the list of references at the end of the document. The first chapter would start by presenting the background, containing appropriate theoretical instructions. The next chapter is the implementation of the custom/customized written code, and what sort of functions are used. The discussion of the different tests performed would be discussed in the next following chapter, also following the result of the tests. The last chapters consist of a conclusion and a list of sources.

# 2. Background

Your background section should include the appropriate theoretical background that your reader should know before delving into the details. Explain reliability, stop and wait, go back n and selective repeat

In computer networking, reliability refers to the capacity of a network to transmit data without errors or duplications. For important cases like real-time communication and financial transactions, it is one of the most crucial components of network performance. Several methods, for example error detection/correction, acknowledgements, sequencing, and retransmission, are used in networking to increase reliability. Reliability can be affected by several variables, for instance network congestion, packet loss, transmission mistakes, among others. Networking protocols, such as the User Datagram Protocol (UDP) and Transfer Control Protocol (TCP), have been created to address these problems and ensure dependable data transfer.

Reliable data delivery is crucial in computer networking to make sure that data is being transmitted correctly and error-free. Transport protocols help in achieving this as they sit on top of the network layer providing flow control, error detection and correction among other services. A common issue which transport protocols try to solve is packet loss or corruption during transmission. In this project, we implemented three reliability functions to tackle these problems: stop-and-wait, go-back-n, and selective repeat.

Stop-and-wait is the simplest of the three reliability functions implemented in this project and is a commonly used protocol which ensures reliable data delivery. It does so by transmitting packets one by one and waiting for an acknowledgement (ACK) from the receiver before transmitting the next packet. In the case of a timeout where no ACK is received after a certain amount of time, the sender retransmits the packet.

Go-back-n applies a sliding window protocol where the sender transmits a certain number of packets. With this protocol, the receiver only accepts packets if they are in order and discards those which are not. Similar to the stop-and-wait protocol, when a timeout occurs the sender retransmits the packets after the last acknowledged in-order packet.

Additionally, selective repeat also applies a sliding window protocol. However, as opposed to retransmitting all the packets, the sender will only retransmit the lost packet(s) within the specified window. This means that the receiver accepts all the packets even if they are out of order, given that they are within the window. The sender waits until all the packets within the window have been acknowledged before transmitting the next set of packets.

# 3. Implementation

Add your implementation details here - with code snippets, diagrams and other details.

In this Portfolio, we are to implement a simple transport protocol – DATA2410 Reliable Transport Protocol (DRTP) that provides reliable data delivery on top of UDP. Our Protocol will ensure that data is reliably delivered in-order without missing data or duplicates. To test our custom protocol, we also had to implement a file transfer application program that uses DRTP/UDP protocol.

In our custom protocol, we implemented three reliability functions that users can choose from through the command line argument. The DRTP protocol code is divided into two functions for each reliable method. A receive function and a send function. Additionally, there are common functions that all three reliable methods use, such as the functions **initiate\_handshake** and **handle\_handshake** which are used for establishing a connection. It performs a three-way handshake between the client and server.

A picture containing text, handwriting, font, line

Description automatically generated



The Stop-and-wait reliable method is divided into the functions **“RECV\_SAW()”** and **“SEND\_SAW()”**. The former is responsible for receiving data packets sent by the sender and sending ACK packets to confirm the reception of each packet. The **“RECV\_SAW()”** starts with a handshake with the sender and initializes the expected sequence number and waits for data packets to arrive. Upon receiving a packet, the function first checks the sequence number. If it is the expected sequence number, the function sends an ACK message to confirm reception, increments the expected sequence number, and appends the received payload data. If the received packet has an incorrect sequence number, the function sends a duplicate ACK message or discards the out-of-order packet. If the FIN flag is set, the function sends an ACK message, closes the socket, and returns the received data. The **“SEND\_SAW()”** function begins with a handshake and sets the sequence number variable to just one. Once in a loop, it sends a data packet (up to 1460 bytes) with the current sequence number and waits for the receiver to respond with an ACK message. The sequence number is updated, and the next packet is sent if the ACK message is valid. It resends the previous packet with the prior sequence number if the ACK message is a duplicate. While awaiting an ACK message, if there is a timeout, the packet is resent with the current sequence number. When there is no more data to send, the function closes the connection.

A picture containing text, line, screenshot, diagram

Description automatically generated

A picture containing text, line, screenshot, diagram

Description automatically generated

The Go\_Back\_N reliable method consists of **RECV\_GBN()** and **SEND\_GBN()**. The former first performs the three-way handshake by calling the **handle\_handshake()** function. Then, it initializes variables such as **expected\_sequence\_number** and **received\_data** before it waits for data packets to arrive. When a packet arrives in-order, it sends back an ACK message. If SYN message is received, the function calls the **handle\_handshake()** function to redo the three-way handshake. If the received packet has the expected sequence number and is not a FIN message, the function appends the data to the **received\_data** variable and sends back an ACK message. If FIN message is received, the function sends back an ACK message, closes the socket, and returns the received data. Otherwise, if a packet arrives with the wrong sequence number and FIN flag is not set, it discards the packet.

The **SEND\_GBN()** initiates the three-way handshake by calling the **initiate\_handshake()** function and initializes variables such as next sequence number, base sequence number, unacknowledged packets, data offset, and whether FIN message has been sent. This function sends data packets as long as the number of packets in flight is less than the window size. Payload for each packet is calculated by slicing it from the data variable. **SEND\_GBN()** then sends the packet, adds it to unacknowledged packets, and increments the next sequence number and data offset. The function waits for an ACK message for the sent packet if a FIN message has not yet been sent. If an ACK message is received with the correct acknowledge number, the function moves the base sequence number for the window and updates the unacknowledged packets list. If no ACK message arrives, **SEND\_GBN()** resends all unacknowledged packets in the window. The function waits for an ACK message for the FIN message if one has been sent before closing the socket and exiting the loop.

A diagram of a window size

Description automatically generated with low confidence

Furthermore, the Selective Repeat protocol is divided into two parts. The **RECV\_SR** function runs and handles the receiver or the server side while the **SEND\_SR** function runs and handles the sender or client side. Both of these functions are based off the code for the Go-back-n reliability function. This is because they work in similar ways, the main difference being in how the server handles the received packets. Instead of discarding out-of-order packets, the selective repeat function reorders the packet in the server. The function also selectively retransmits only the missing packets, hence the name.

The **RECV\_SR** function is mainly responsible for the reception of packets and the sending of acknowledgements. First, it runs the **handle\_handshake** function which initiates the three-way handshake to initiate the connection between the server and client. Thereafter, we initiate a dictionary variable **unacked\_packets** which keeps track of the unacknowledged packets. In the while loop, the difference to **RECV\_GBN** is that this function also checks if the number of unacknowledged packets is less than the window size along with if the received packet has the correct sequence number. Moreover, it has an additional loop which sends ACKs to received but unacknowledged packets appending the data to the **received\_data** variable, incrementing **expected\_seq\_num**, removing the packet from the **unacked\_packet** list and finally sending an ACK. Lastly, another difference is when the packet is out of order the function send an ACK for the last received in-order packet and adds it to the **unacked\_packet** list. It then iterates the same loop as before to send ACKs for received but unacknowledged packets.

The **SEND\_SR** function sends packets and handles retransmissions. This function is also similar to **SEND\_GBN** but when a packet is sent, a timer is also started along with the packet being added to the **unacked\_packet** list. The function also removes the ACKed packets from the list as opposed to moving the base sequence number as it is in **SEND\_GBN**. Finally, if a timeout occurs the code will enter a loop that iterates through **unacked\_packet**. For each unacknowledged packet, it checks if the current time minus the time the packet was sent is greater than or equal to 0.5s. If so, the packet is resent to the receiver. Then, the time is updated to ensure that the packet’s timeout is correct on the next iteration. The rest of the code is identical to the **SEND\_GBN** function.

A picture containing text, screenshot, line, parallel

Description automatically generated

In order to calculate the per-packet roundtrip time and dynamically set the timeout to 4RTTs, we first set the initial value of a variable (est\_rtt) to a conservative estimate of 0.5 seconds to keep track of the estimated roundtrip time. When a packet is sent, we record the time it was sent in send\_time. After receiving an acknowledgement, we calculate the RTT by subtracting the time the acknowledgment was received with send\_time, and save the result in variable rtt. The value of est\_rtt is then updated to 4 times the most recent determined rtt. Finally, we alter the socket's timeout value to reflect the updated value of est\_rtt using the settimeout() method.

# 4. Discussion

Test cases here and show how you handle losses, reordering and duplicate packets.

## Test Case 1

The objective for this test is to run the application file with the custom written reliable protocol -including StopAndWait, GoBackN and Selective Repeat. The application file is going to run with a specific latency in this matter the total RTT (round-trip-time). The RTT is the time it takes for a packet to be transferred from one end of the network to the destination, and back again. This is common for all of the protocols, but with a slightly difference in GoBackN and Selective Repeat. Here we have to also specify the window-size, which is the total number of packets which is going to transmitted. The latency and the window-size are an important tool to measure the throughput, as an important calculation of this test case. The result of the measured throughput would be discussed and stated.

|  |  |  |
| --- | --- | --- |
| StopAndWait | RTT | Throughput |
| Test 1 | 25ms | 0.38Mbps |
| Test 2 | 50ms | 0.19Mbps |
| Test 3 | 100ms | 0.09Mbps |

|  |  |  |  |
| --- | --- | --- | --- |
| GoBackN | RTT (Latency) | Window-Size | Throughput |
| Test 1 | 25ms | 5 | 1.74 Mbps |
| Test 2 | 25ms | 10 | 3.23 Mbps |
| Test 3 | 25ms | 15 | 4.08 Mbps |
| Test 4 | 50ms | 5 | 0.87 Mbps |
| Test 5 | 50ms | 10 | 1.61 Mbps |
| Test 6 | 50ms | 15 | 2.06 Mbps |
| Test 7 | 100ms | 5 | 0.49 Mbps |
| Test 8 | 100ms | 10 | 0.97 Mbps |
| Test 9 | 100ms | 15 | 1.42 Mbps |

|  |  |  |  |
| --- | --- | --- | --- |
| Selective Repeat | RTT (Latency) | Window-Size | Throughput |
| Test 1 | 25ms | 5 | 1.93 Mbps |
| Test 2 | 25ms | 10 | 3.81 Mbps |
| Test 3 | 25ms | 15 | 5.83 Mbps |
| Test 4 | 50ms | 5 | 0.99 Mbps |
| Test 5 | 50ms | 10 | 1.95 Mbps |
| Test 6 | 50ms | 15 | 2.90 Mbps |
| Test 7 | 100ms | 5 | 0.49 Mbps |
| Test 8 | 100ms | 10 | 0.95 Mbps |
| Test 9 | 100ms | 15 | 1.42 Mbps |

In the first table, we are measuring the throughput of stop and wait- protocol. It would be performed three experiments, with different latencies as the description. The first experiment we are sending one packet form the client and waiting for acknowledgement message back from the server. The latency between one of the hosts is around 6ms, and the total RTT is 25ms. The throughput is measured to be 0.38 Mbps. Performing the other two experiments, it appears in decreased throughput. The cause for this is certainly because of the increase latency between the host and a total higher RTT. As we can see from the table, when the RTT double the measured throughput halves. We can also in some cases experience even a lower throughput due to retransmission of a packet. If a packet is sent from the client, and if it is not received an acknowledgment message in half a second, it would retransmit the same packet again. This has fortunately not happened in this experiment performed, making the measured throughput accurate.

The second table, we are measuring the throughput using the GoBackN-protocol. In this test it would be performed multiple tests with different latencies and specific window-size. In the first experiment- including test1-test3, we are going to send a number of packets from the client and wait for an acknowledgment message from the server. The purpose is to specify a window-size, which is the total amount of packets sent but not acknowledged. When the client receives an acknowledgment for a packet transmitted, it would send the next packets in order. The order is based on the sequence number. The first experiment we have chosen the default setting for the window-size (5 packets), which is transferring 5 packets. The total RTT for the first experiment is 25ms, which has 6ms of latency between each link. Realizing that the more we increase the window size, the greater is the measured throughput. When the window-size is 5, the measured throughput is 1.74 Mbps, but when the window size is 10 the calculated throughput is 3.23 Mbps. Resulting in an increased throughput when the specified window size is greater. An important observation is that when the latency is increased to 50ms in experiment two, it still follows the same pattern as when the latency was 25ms. The only difference is that the measured throughput is increasing when choosing different window sizes, but not as high as the previous latency. Concluding that both the latency and the window size has an important effect on what the measured throughput is. The pattern remains the same for all the experiment leading to increased throughput.

In last table, we are going to measure the throughput using the SelectiveRepeate-protocol. It would be performed the same series of experiments as the GoBackN table. The SelectiveRepeate-protocol is based on the GoBackN-protocol and works in a similar way, but the main difference is how the server handles the received packets. Instead of discarding the out-of-order packet and then retransmit the same packet in GoBackN-Protocol, the server in SelectiveRepeate saves/buffers the out-of-order packets. Out-of-order packets is based on packets sent with an unexpected sequence number. Making the SelectiveRepeate simple, due to retransmitting only the lost or missing packets. In the first experiment we are going to transfer packets, in which chosen window-size equals five. The total latency between the link is equivalent to the previous table with 6ms and making the total RTT 25ms. The calculated throughput for the packets sent with the window-size equal to 5 is 1.93 Mbps. With window-size equal to 10, the measured throughput is 3.81 Mbps. Then packets sent with the window-size equal to 15 have a measured throughput of 5.83 Mbps. Realizing that this protocol follows the pattern as the previous, the increased window-size is leading to a greater throughput. Comparing it to a higher total RTT in the other experiment, the measured throughput decreases because the induvial links have a higher latency. Still the pattern remains the same, when the window-size increases the throughput is slightly higher than a lower window-size. The conclusion is that both the GoBackN and SelectiveRepeate – protocol is similar, but with a signific difference. The SelectiveRepeate seems to have a higher measured throughput in the experiments compared to the GoBackN protocol. The last experiment (including test 7-9), it seems the calculated throughput drops, and is similar to the last experiment in GoBackN. The cause for this could be due to a higher latency between the links. Realizing that SelectiveRepeate is a faster protocol than GoBackN, since it only handles and retransmits the lost or missing packets sent from the client. Compared to GoBackN which is discarding the out-of-order packets and again retransmitting all packets in the window. This activity is leading the transferred packet slightly slower.

Write a test case to skip an ack - this will trigger retransmission. Test with all three reliable functions.

## Test Case 2

The purpose of this test case is to verify that the reliable transport protocol can recover from skipped ACK message. Skipped ACK messages can occur in a network where packets are dropped or delayed, and the receiver fails to send an ACK message. This test case is important because it ensures that the reliable transport protocol can recover from skipped ACK messages. Skipped ACK message can cause data loss.

### Environment

* Python version: 3.9.2
* Operating system: Debian 11
* Testing framework: Virtual Mininet topology

### Steps

1. Start the server/receiver with the command **‘python3 application.py -s -i <IP address> -p <port number> -r SAW/GBN/SR -t skip\_ack’**. Replace <IP address> with the address of the host and choose one of the three reliable methods.
2. Start the client in client/sender mode with the command **‘python3 application.py -i <IP address> -p <port number> -r SAW/GBN/SR** ‘. Replace <IP address> with the server IP address, and make sure to use the same port number as the server as well as the same reliable method.
3. Monitor the terminal to check for retransmission message.
4. Verify that the client/sender retransmits the unacknowledged packet or packets in the window.

### Test Data

* IP address = 10.0.1.2
* Port number = 12000
* File name = testFile.jpg

### Expected Results

* Stop\_And\_Wait: The client/sender should retransmit the previously sent packet because of a timeout in Stop\_And\_Wait.
* Go\_Back\_N: the client should retransmit all packets in the window.
* Selective Repeat: The client should retransmit only packets not ACK’ed by the receiver.

### Actual Results

All retransmissions happened as expected for each reliable method.

Write a test case to skip a sequence number to show the out-of-order delivery effect. This will also trigger retransmission. Test with GBN, and SR. Report your results.

## Test Case 3

This test case checks the ability of the DRTP protocol to recover from the loss of a packet. It simulates a scenario in which retransmission takes place after skipping packet with sequence number 5. This test case is significant because it validates DRTP's ability to handle packet loss, a frequent occurrence in network communication.

### Environment

* Python version: 3.9.2
* Operating system: Debian 11
* Testing framework: Virtual Mininet topology

### Steps

1. Start the server/receiver with Go\_Back\_N or Selective Repeat reliable method by running **‘python3 application.py -s -i <IP address> -p <port number> -r GBN/SR -w <window size [only in Selective Repeat]>’**. Replace **<IP address>** and port number with the address of the host and the chosen port number, respectively. Additionally choose the window size you want to test the protocol with. Note that you can choose window size when running the program in server mode only with Selective Repeat.
2. Start the client/sender with Go\_Back\_N or Selective Repeat by running **‘python3 application.py -c -i <IP address> -p <port number> -r GBN/SR -t loss -w <window size>’**. Replace **<IP address>** with the server IP address, and make sure to use the same port number as the server as well as the same reliable method. Also choose the window size you want to test the program with.
3. Monitor the terminal to check for retransmission message.
4. Verify that the expected number of packets was retransmitted or that the skipped packet was retransmitted.

### Test Data

* IP address = 10.0.1.2
* Port number = 12000
* File name = testFile.jpg

### Expected Results

* When the server/receiver receives the packets where a sequence number is skipped in Go\_Back\_N reliable function, it will discard out-of-order packets. Timeout will occur on the client/sender side, triggering retransmission of all packets in the window, including the skipped packet.
* In Selective Repeat, timeout will occur only for the unacknowledged packet, resulting in a retransmission.

### Actual Results

The program behaved as expected.

Use your artificial testcases to show the efficacy of your solution. Your solution should be able to handle losses, reordering and duplicates.

## Test Case 4

As the test cases above show, our program is resilient against packet losses, reordering and duplicates. The Stop\_And\_Wait reliable method handles losses, reordering, and duplicates of packets as follows:

* Losses: The sender won't get an ACK message from the receiver within the specified timeout period if a packet is lost in transit. When the sender does not receive an ACK message in our artificial testcases, it resembles a packet loss. The sender will then resend the same packet with the same sequence number, and the receiver will discard any duplicate packets with the same sequence number.
* Reordering: A duplicate ACK message will be sent for the earlier sequence number if a packet is received out of order, and the receiver will then wait for the next packet with the expected sequence number. After the expected packet is received, the sender resumes delivering packets with the next sequence number after receiving an ACK message with the correct sequence number.
* Duplicates: If the receiver receives a duplicate packet with the same sequence number as the previous packet, it will ignore the duplicate packet and send a new ACK message for the previous sequence number. This will trigger retransmission at the sender side when it receives the new ACK message.

Meanwhile the Go\_Back\_N reliable method handles losses, reordering, and duplicates as follows:

* Losses: Our solution handles losses by implementing a timeout mechanism. If the sender does not receive an ACK within the specified timeout, as happens when the receiver skips sending an ACK message, it retransmits the unacknowledged packets. On the receiver side, it discards out-of-order packets. The sender waits for an ACK with the same sequence number after sending a packet with a specific sequence number.
* Reordering: Our method handles out-of-order packets by discarding them. If a packet has not been acknowledged yet, the sender resends that packet when it times out. This scenario is simulated with test case 3, where we skip a sequence number. The receiver discards all the out-of-order packets, triggering timeout at the sender side.
* Duplicates: The reliable method discards duplicate packets it receives in the receiver side. The sender keeps track of sent but not acknowledged packets, and when it receives an ACK, it updates the list of unacknowledged packets. Therefore, if the receiver sends an ACK for the same packet twice, the sender ignores the duplicate ACK.

Selective Repeat handles losses, reordering, and duplicates as follows:

* Losses: In our Selective Repeat, losses are handled by retransmitting unacknowledged packets. If a sender does not receive an ACK within the specified time out, it retransmits the unacknowledged packets, unlike Go\_Back\_N, which retransmits all packets in the window after no ACK message is received. In our test case, when the receiver skips sending the first ACK message, the sender retransmits all packets in the window again, because it assumes the packet to be lost.
* Reordering: Our Selective Repeate handles out-of-order packets by buffering them. Any out-of-order packet that arrives, will get buffered by the receiver. It then sends an ACK message for the last packet in order that was received. If the buffered packet’s sequence number matches the expected sequence number, the receiver sends an ACK for the buffered packet, and continues as usual. If not, the receiver waits for the missing packet. On the sender side, if a packet is not acknowledged, it retransmits the packet after a timeout.
* Duplicates: The Selective Repeat reliable function handles duplicate packets by using the sequence number of each packet to keep track of packets sent and received. If the receiver receives a duplicate packet, it discards this packet. The receiver does this because it maintains a list of received packets with sequence numbers and the expected sequence number of the next in-order packet. The sender also ignores the duplicate ACK.

## Bonus TC-NETEM

In addition to using the artificial testcases, we used tc-netem to emulate packet loss, reordering and duplicate packets to show the efficacy of our code. Before we can even use tc-netem, we must edit the provided simple-topology.py code. In lines 28 and 29, we must remove bandwidth and delay values, otherwise tc-netem will give us an exclusivity flag error. Next, we use xterm to open terminal for different nodes in the simple topology. In the terminal window for router r2 we write **‘sudo tc qdisc add dev r2-eth1 root netem delay 25ms loss 10%’**. This will emulate a round-trip-time of 25ms and a 10% loss in the link between router r2 and host h3, and by extension, the RTT between host h1 and h3 will be around 25ms.

# 5. Conclusions

A summary of what you have done, including key results.

# 6. References

Islam, Safiqul. (2023). The Transport Layer [Lecture slides], DATA2410: Networking and Cloud Computing. Oslo Metropolitan University.