2023

PORTFOLIO 2

DATA2410 – OSLO METROPOLITAN UNIVERSITY

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

TABLE OF CONTENTS

1. Introduction

The main goal of this project is to implement a simple transport protocol (DRTP) - Data Reliable Transport Protocol - which can reliably transmit data over a User Datagram Protocol (UDP) connection. This program is built using python and socket programming 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 which will be used to run a simple topology. This will be useful to evaluate the virtual network resources and performance when using DRTP. We have also implemented two bonus assignments. The first assignment would be using ***tc-netem*** to emulate less than ideal circumstances and second would be to calculate per-packet RTT and set the timeout four times the measured RTT.

The test would be done by the custom written code to transfer a file, and to measure the throughput using different round-trip-times (RTTs). 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 handle the transmission and reception of data and acknowledgements. This results in a more maintainable and flexible code making it easier to alter the different reliability functions as needed.

However, the custom reliable transfer protocol has its limitations. One such example 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 presents the background, containing appropriate theoretical knowledge relevant for this project. The next chapter contains information about the implementation of DRTP, and what sort of functions are used. The different tests performed would be discussed in the following chapter, which also includes the results of the tests. Finally, the last chapters consist of a conclusion and a list of sources.

2. Background

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 packet loss and transmission mistakes among many 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.

Transport protocols help in achieving reliability as they sit on top of the network layer providing flow control, error detection and correction among several other services. A common issue that transport protocols try to solve is packet loss or corruption during transmission. In this project, we implemented three reliability functions to tackle such 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.

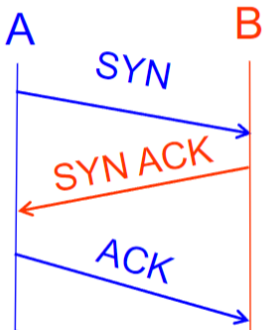
Go-Back-N applies a sliding window protocol where the sender transmits a certain number of packets. With this protocol, the receiver only accepts packets if they are in order and discards those which are not. The sliding window only moves with every in-order packet acknowledged. In the case of a timeout, 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 sliding window. This means that the receiver accepts all the packets even if they are out of order. The sender waits until all the packets within the window have been acknowledged before transmitting the next set of packets.

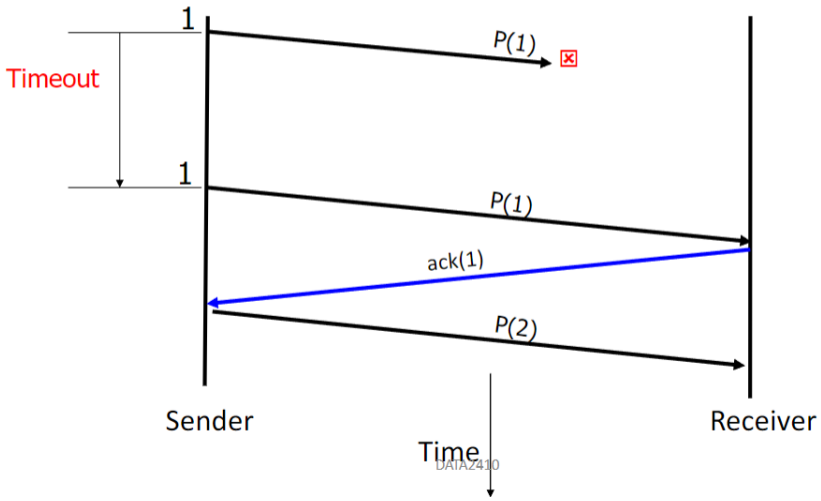
3. Implementation

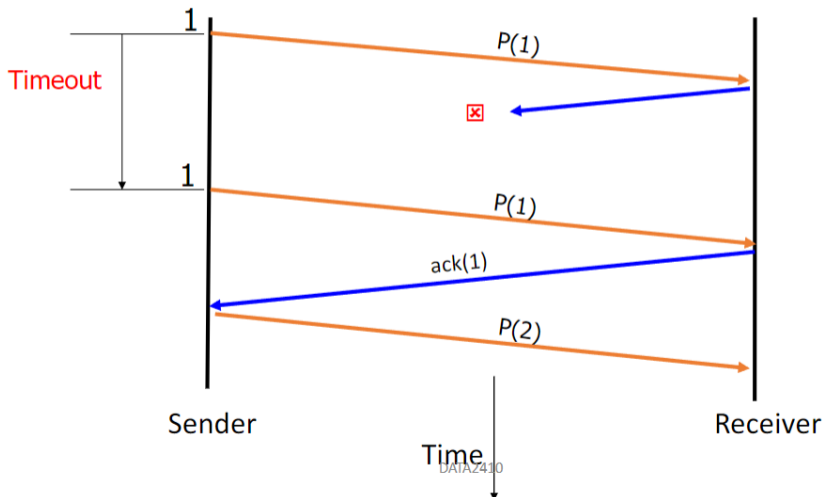
In this Portfolio, we are to implement a simple transport protocol – DATA2410 Reliable Transport Protocol (DRTP) that provides reliable data delivery on top of UDP. Our protocol will ensure that data is reliably delivered in-order without missing data or duplicates. To test our custom protocol, we also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



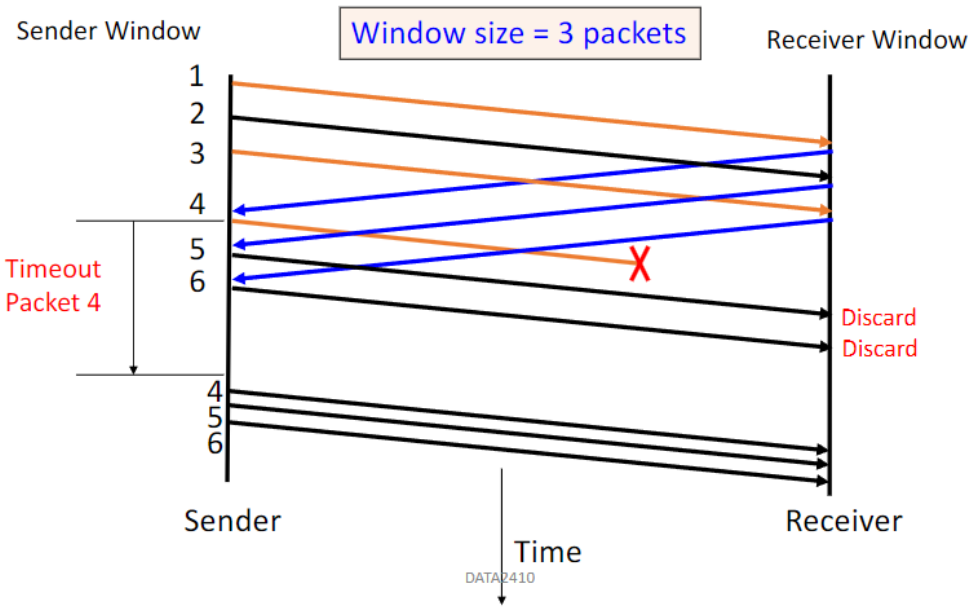
The Stop-And-Wait (SAW) reliable method is divided into the functions **"RECV_SAW()"** and **"SEND_SAW()"**. The former is responsible for receiving data packets sent by the sender and sending ACK packets to confirm the reception of each packet. The **"RECV_SAW()"** starts with a handshake with the sender 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 and 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 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.





The Go-Back-N (GBN) reliable method consists of "**RECV_GBN()"** and **"SEND_GBN()"**. The 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.

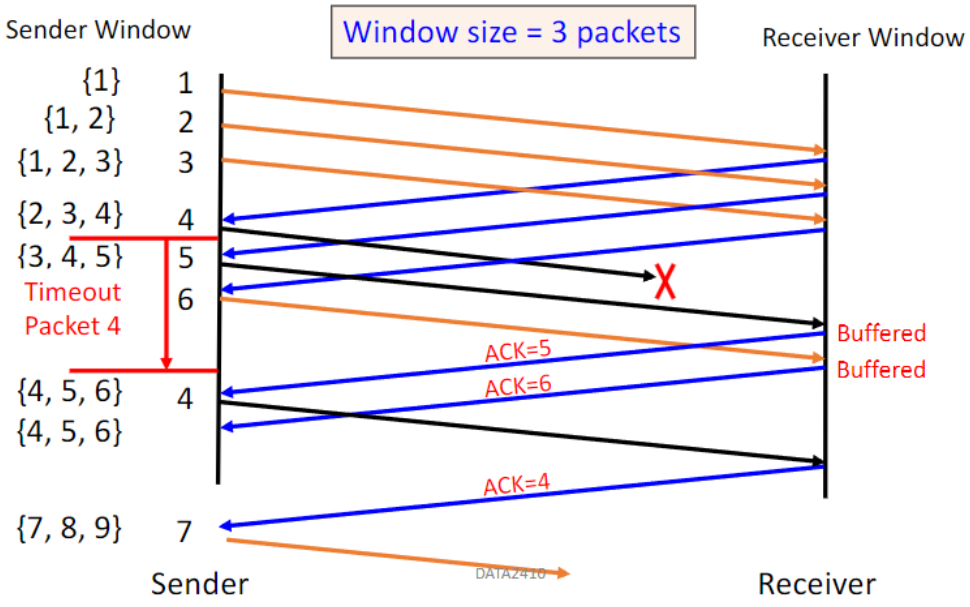
The "**SEND_GBN()"** initiates the three-way handshake by calling the "**initiate_handshake()"** function and initializes variables such as next sequence number, base sequence number, unacknowledged packets, data offset, and whether FIN message has been sent. This function sends data packets as long 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 acknowledgement 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.



Furthermore, the Selective-Repeat (SR) 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. The SR protocol is a variant of the GBN protocol, and it allows the receiver to individually acknowledge each packet received, instead of acknowledging them in sequence order like in GBN. The protocol buffers out-of-order packets instead of discarding them. 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. Thereafter, we initiate a dictionary variable "**unacked_packets"** which keeps track of the unacknowledged packets. In the while loop, unlike "**RECV_GBN()",** 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. Overall, it is this function that ensures that received packets are placed in the correct order in the receive buffer and that packets arriving in the wrong order are not discarded.

The "**SEND_SR()"** function sends packets and handles retransmissions. This function shares similarities with "**SEND_GBN()"** but with some modifications. If the sequence number of the ACK is in the "**unacked_packets"** list, the packet is removed from the dictionary, as opposed to "**SEND_GBN()"** which updates the list to packets with sequence numbers greater than or equal to the new base sequence number. The rest of the code is identical to the "**SEND_GBN()"** function.



To achieve better performance and obtain bonus points, we improved our timeout handling by dynamically setting the timeout to 4 times the estimated round-trip time (RTT) of each packet. First, we set the initial value of a variable "**est_rtt**" to a conservative estimate of 500ms to keep track of the estimated round-trip 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 the "**rtt"** variable. The value of "**est_rtt"** is then updated to 4 times the most recent **rtt** value. Finally, we alter the socket's timeout value to reflect the updated value of "**est_rtt"** using the "**settimeout(**)" method. This allows us to adaptively adjust the timeout value to the current

network conditions, which can significantly improve the reliability and efficiency of the data transfer process.

4. Discussion

Test Case 1

The objective for this test is to run the application file with the custom written reliable protocol – including Stop-and-Wait, Go-Back-N, and Selective-Repeat. The application file is going to run with a specific latency, which is the total round-trip-time (RTT). 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 the protocols, but with a slight difference in Go-Back-N and Selective-Repeat. In these two cases, we must also specify the window-size, which is the total number of packets which are going to be transmitted. The RTT and the window-size are important tools to measure the throughput and represent a significant calculation of this test case. The result of the measured throughput will be discussed and stated.

Stop-And-Wait	RTT	Throughput
Test 1	25ms	0.38Mbps
Test 2	50ms	0.19Mbps
Test 3	100ms	0.09Mbps

In the first table, we are measuring the throughput of stop-and-wait protocol. Three experiments are conducted, with different latencies following the description. For the first experiment we are sending one packet from the client and waiting for an 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. When performing the other two experiments, there is decrease in the throughput. This is due to the increased latency between the host and a total higher RTT. As we can see from the table, when the RTT doubles the measured throughput halves. We can also, in some cases, experience an even lower throughput due to retransmission of a packet. If a packet is sent from the client, and if an acknowledgment message is not received in half a second, the client will retransmit the same packet again. Fortunately, this has not happened in this experiment, making the measured throughput accurate.

Go-Back-N	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

In the second table, we are measuring the throughput using the Go-Back-N protocol. Multiple tests are performed with different latencies and a specific window-size. In the first experiment – ranging from test1 to 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 sends the next packets in order. The order is based on the sequence number. In the first experiment we chose the default setting for the window-size (5 packets), which is thus transferring 5 packets. The total RTT for the

first experiment is 25ms, which has 6ms of latency between each link. We realized that the more we increase the window size, the greater the measured throughput will be. When the window-size is 5, the measured throughput is 1.74 Mbps, meanwhile when the window size is 10, the calculated throughput is 3.23 Mbps. This results in an increased throughput when the specified window size is greater. An important observation from experiment two – ranging from test4 to test6 – is that when the latency is increased to 50ms, it still follows the same pattern compared to 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. This concludes that both the latency and the window size have an important effect on what the measured throughput is. The pattern remains the same for all the experiments leading to increased throughput.

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 last table, we measure the throughput using the Selective-Repeat protocol. For this test, the same number of experiments as the Go-Back-N table are conducted. The Selective-Repeat protocol is based on the Go-Back-N 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 retransmitting the same packet in Go-Back-N protocol, the server in Selective-Repeat buffers the out-of-order packets. Out-of-order packets are sent with an unexpected sequence number. This makes Selective-Repeat simple, due to it retransmitting only the lost or missing packets. In the first three experiment we are going to transfer packets, in which the chosen window-size is five. The total latency between the links is equivalent to the previous table with 6ms, 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. Lastly, packets sent with the window-size equal to 15 have a measured throughput of 5.83 Mbps. This protocol follows the pattern as the previous tests, and the increased window-size leads to a greater throughput. When comparing the highest and the lowest RTT, the measured throughput decreases because the individual links have a higher latency. However, when the window-size increases the throughput also increases. We can conclude that Go-Back-N and Selective-Repeat protocol have their similarities, but with a significant difference in performance. The Selective Repeat seems to have a higher measured throughput in the experiments compared to the Go-Back-N protocol. The last three experiments, it seems the calculated throughput drops, and is similar to the last three experiments in Go-Back-N. The cause for this could be due to a higher latency between the links. Realizing that Selective-Repeat is faster in some scenarios than Go-Back-N, since it only handles and retransmits the lost or missing packets sent from the client. Compared to Go-Back-N which discards the out-of-order packets and retransmits all packets in the window. This action leads to decreased throughput, when packets get lost or are out-of-order.

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
- Window size = 5

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 only retransmit the packets that have not been ACK'ed by the receiver.

Actual Results

All retransmissions happened as expected for each reliable method.

	Stop-And-Wait	Go-Back-N	Selective-Repeat
Skip ack message throughput	0.48 Mbps	2.10 Mbps	2.08 Mbps

Test Case 3

This test case checks the ability of DRTP 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
- Window size = 5

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.

	Stop-And-Wait	Go-Back-N	Selective-Repeat
Skip seq_num throughput	0.5 Mbps	2.07 Mbps	2.09 Mbps

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 will not 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, which 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-Repeat 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

Results

	Stop-And-Wait	Go-Back-N	Selective-Repeat
Loss	0.13 Mbps	0.18 Mbps	0.21 Mbps
Reorder	0.79 Mbps	0.10 Mbps	2.59 Mbps
Duplicate	0.41 Mbps	2.02 Mbps	2.03 Mbps

Discussion

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. However, before we could use tc-netem, we had to make some modifications to the provided simple-topology.py code. Specifically, in lines 28 and 29, we had to remove bandwidth and delay values, to avoid an exclusivity flag error.

We used 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 effectively simulated the test cases in the network topology we wanted to test and allowed us to see how our code would perform under less-than-ideal circumstances. Next, we started the program in server mode with our desired reliable function in host h3. In host h1, we start the program in client mode, with the same reliable method as the server, and we must make sure to use the same IP address and port as the server is listening on.

To test how the program handles duplicates, we wrote the following command in router r2's Xterm terminal **'sudo tc qdisc add dev r2-eth1 root netem delay 25ms duplicates 10%'.** This will duplicate around 10 % of packets arriving at the interface r2-eth1. Our program behaved as described in Test Case 4 for all reliable methods.

Again, we typed in the command **'sudo tc qdisc add dev r2-eth1 root netem delay 25ms reorder 50% 50%'**. This tests the programs' ability to handle reordering of packets. As explained in Test Case 4, the program behaves differently when using the three reliable functions. We observed that our program handled the reordering of packets as expected for the different reliable methods.

The Selective-Repeat protocol generally outperforms both the Stop-And-Wait and Go-Back-N protocols, according to measured throughputs under various network limitations. Under loss and reordering scenarios, it shows greater throughput, and when handling duplicate packets, it performs on par with Go-Back-N. The Stop-And-Wait protocol consistently has the lowest throughputs in all cases, demonstrating how ineffective it is at coping with different network problems. This bonus assignment also highlights the efficacy of our reliable transfer protocol built on top of UDP.

5. Conclusions

In this report, we have used different tools and resources to evaluate the performance of the custom written applications. This is done by measuring the retransmission of lost, duplicated, or re-ordered packets, for each of the different reliable protocols implemented. To be able to assess the application's performance, various test cases are executed. The first test case was to evaluate the applications functionality, and this is determined by a series of tests using a simple virtual network. The measured performance of the network is calculated with a specified RTT between the hosts and would return a throughput with the unit Mbps. The other test cases evaluate how the application is handling the criteria.

When creating the application, we unfortunately experienced difficulties regarding the sequence of packets and ACK message for each arriving packet. Following this, we chose to simplify the process by sending an ACK message with acknowledge number same as the arriving packets´ sequence number. Additionally, it was challenging to structure the discussion section for each test case. A limitation worth mentioning is that the application does not check for packet-corruption.

The most important findings in this report include the two following elements. Firstly, when comparing the window-size to the RTT, the RTT would affect the value of the throughput more than the window-size. The higher total packets transferred would evidently cause the throughput to have a greater value. If we compare a small window-size to a higher window-size, the higher window-size will have an increased throughput. Secondly, after performing Test Case 2 and 3 as well as bonus assignment involving tc-netem, we found that Selective-Repeat generally has a higher throughput than both Stop-And-Wait and Go-Back-N reliable methods. Selective-Repeat is useful when the application needs to ensure the reliable delivery of all packets and when out-of-order packets are expected. Go-Back-N, on the other hand, is suitable for applications that require fast transmission and can tolerate some packet loss. Additionally, by dynamically setting the timeout to 4RTT for the bonus, the timeout value can better reflect the current network conditions. Using a fixed timeout may lead to unnecessary delays if the network is faster than the fixed timeout or premature retransmissions if the network is experiencing high latency.

To conclude, the importance of elements such as protocol selection, RTT and dynamic timeout adjustments for performance optimalisation and to guarantee a reliable packet delivery in different network environments is highlighted throughout this project, especially throughout the evaluation of DRTP.

6. References

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