5/16/2023

Portfolio 2

DATA2410 Reliable Transport Protocol (DRTP)

Moustapha Aljundi – s364546
Alan Ali - s356488
Mohammad Khir Almohammad- s343988
Ali Alghadban - s362111

INTRODUCTION

This report addresses the design and implementation of a reliable transport protocol, The Reliable Transport Protocol (DRTP). This protocol is built atop the User Datagram Protocol (UDP) to ensure reliable and in-order data delivery without losses or duplicates. The primary aim of this task is to develop a system that can reliably transfer files between two nodes in a network.

To achieve this, we have implemented two programs: DRTP, which offers a reliable connection over UDP, and a file transfer application, consisting of a simple file transfer client and server. Both programs have been developed in Python and make use of command-line arguments to take in user input.

We have placed special emphasis on ensuring that our DRTP protocol handles connection establishment and tear-down in a reliable manner, by implementing a three-way handshake similar to TCP. To achieve reliability, we have implemented three different reliability functions: stop-and-wait, Go-Back-N, and Selective-Repeat.

Later in the report, we will discuss the tests we conducted to ensure that our implementation was reliable and efficient. We have used a variety of scenarios to test our code, including simulating packet loss and retransmission of packets.

We will also present a thorough analysis of the throughput values for different window sizes and RTT values and explain the results we obtained. Lastly, we will demonstrate the efficacy of our solution through artificial test cases, and show how our system handles losses, reordering, and duplicates within the network.

BACKGROUND

STOP-AND-WAIT FUNCTION:

The *stop-and-wait* function, implemented here in the DRTP protocol, is a simple yet effective method for ensuring reliable data delivery over a network. The basic idea is that the sender sends a single packet, then waits for an acknowledgement (ACK) from the receiver before proceeding to send the next one.

The function *stop-and-wait* behaves differently depending on whether it's called by a server or a client:

1. **Server:** The server continually listens for incoming packets from the client. When a packet is received, the server parses the packet's header, extends its received file data with the packet's payload, and increments the acknowledgement number (ack). It then creates an ACK packet using this ack number and sends it back to the client. If a “FIN” flag (indicating the end of data transmission) is received, the server breaks the loop, writes the received data to a file, and returns.

   In the case of the “skip_ack” test scenario, the server intentionally skips sending an ACK for the 3rd packet (when *ack_counter* equals 2), to simulate a lost ACK and trigger a retransmission from the client.

2. **Client:** The client splits its data into chunks of 1460 bytes (the maximum payload size), then loops over these chunks. For each chunk, it creates a packet with a sequence number (sequens), an ACK number (ack), a “FIN” flag (if it's the last chunk), and the chunk data. The packet is then sent to the server.

After sending each packet, the client waits for an ACK from the server. If it receives a duplicate ACK (indicating the previous packet was lost), or if a timeout occurs (implying the ACK was lost), it resends the current packet. Otherwise, it increments the sequence number and proceeds to send the next packet.

In the “lose” test case, the client intentionally skips sending the 2nd packet (when *packet_counter* equals 2) to simulate a lost packet and observe the server's response.

This *stop-and-wait* protocol provides reliability but may not be efficient, especially for networks with high latency, as the sender remains idle for the duration of the round-trip time of each packet.

GBN FUNCTION:

The *gbn* function *(Go-Back-N)* in the DRTP protocol is a method for reliable data transmission over a network. Unlike the stop-and-wait method, Go-Back-N allows the sender to transmit multiple packets without waiting for an acknowledgement (ACK) for each one. This method helps to increase throughput and better utilize network resources, especially in high-latency environments.

The function *gbn* behaves differently depending on whether it's called by a server or a client:

1. **Server:** The server uses a separate thread (*packet_receiver*) to listen for incoming packets from the client. When a packet is received, the server checks whether its sequence number (seq) matches the expected sequence number (base). If it does, the payload is added to the received file data, an ACK is sent back to the client, and base is incremented. If a “FIN” flag is received (indicating the end of data transmission), the server ends the communication.

   If a packet arrives out of order (i.e., its seq is greater than base), the server stores it in a buffer (*window_packets*), maintaining the packets in ascending order of their sequence numbers.

   In the “skip_ack” test case, the server intentionally skips sending an ACK for the 3rd packet (when seq equals 2), to simulate a lost ACK and trigger a retransmission from the client.

2. **Client:** The client uses two separate threads - one for sending packets (*c_packet_sender*) and one for receiving ACKs (*c_packet_receiver*). The sender thread splits the data into chunks, creates packets from the chunks, and sends them to the server, up to a maximum of N unacknowledged packets at a time. The receiver thread listens for ACKs from the server and removes all acknowledged packets from the window.

   If an ACK is not received within a certain period, the client resends all unacknowledged packets in the window. If a “FIN” flag is received, the client ends the communication.

   In the “lose” test case, the client intentionally skips sending the 3rd packet (when *“c_next_seq_num”* equals 2), to simulate a lost packet and observe the server's response.

SR FUNCTION:

The *sr* function (Selective Repeat) is a method for reliable data transmission over a network. Like Go-Back-N, Selective Repeat allows the sender to transmit multiple packets without waiting for an acknowledgement (ACK) for each one. However, unlike *Go-Back-N*, Selective Repeat deals with packet loss more efficiently by only retransmitting the lost packets instead of all packets after the lost one.

The function *sr* behaves differently depending on whether it's called by a server or a client:

1. **Server:** The server uses a separate thread (*packet_receiver*) to listen for incoming packets from the client. When a packet is received, the server sends an ACK back to the client and checks whether its sequence number (seq) matches the expected sequence number (*expected_seq_num*). If it does, the payload is added to the received file data, and *expected_seq_num* is incremented.

   If a packet arrives out of order (i.e., its seq is greater than *expected_seq_num*), the server stores it in a buffer (*received_packets*), maintaining the packets in ascending order of their sequence numbers. The server also checks the buffer after processing each packet to see if it can accept any of the buffered packets.

   If a "FIN" flag is received (indicating the end of data transmission), the server ends the communication.

   In the "skip_ack" test case, the server intentionally skips sending an ACK for the 2nd packet (when *ack_counter* equals 2), to simulate a lost ACK and trigger a retransmission from the client.

2. **Client:** The client uses two separate threads - one for sending packets (*c_packet_sender*) and one for receiving ACKs (*c_packet_receiver*). The sender thread splits the data into chunks, creates packets from the chunks, and sends them to the server, up to a maximum of N unacknowledged packets at a time.

   If an ACK is not received within a certain period (Timeout), the client only resends the unacknowledged packet, not all unacknowledged packets in the window. The receiver thread listens for ACKs from the server and removes the acknowledged packet from the window.

   If a "FIN" flag is received, the client ends the communication.

   In the "Lose" test case, the client intentionally skips sending the 2nd packet (when *c_next_seq_num* equals 2), to simulate a lost packet and observe the server's response.

IMPLEMENTATION

STOP-AND-WAIT THROUGHPUT VALUES:

- RTT (25ms):

```
------------------------------------------------------------
DURATION: 15.952 s      DATA SIZE: 0.93 MB      BANDWIDTH: 0.47 Mbps
------------------------------------------------------------
```

- RTT (50ms):

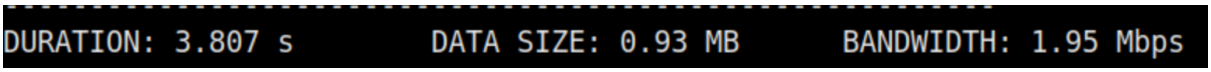
```
------------------------------------------------------------
DURATION: 32.492 s      DATA SIZE: 0.93 MB      BANDWIDTH: 0.23 Mbps
------------------------------------------------------------
-------[Client]: Initiated FIN handshake.--------

[Client]: Sending FIN to the server. Awaiting response...
[Client]: ACK received from server!
Client: Connection with server at 10.0.1.2:8088 has been closed

root@neo-virtual-machine:/home/neo/Downloads/Application_final copy# python3 ap
p.py -c -i 10.0.1.2 -p 8088 -r stop_and_wait -f b.jpg
```

- RTT (100ms):

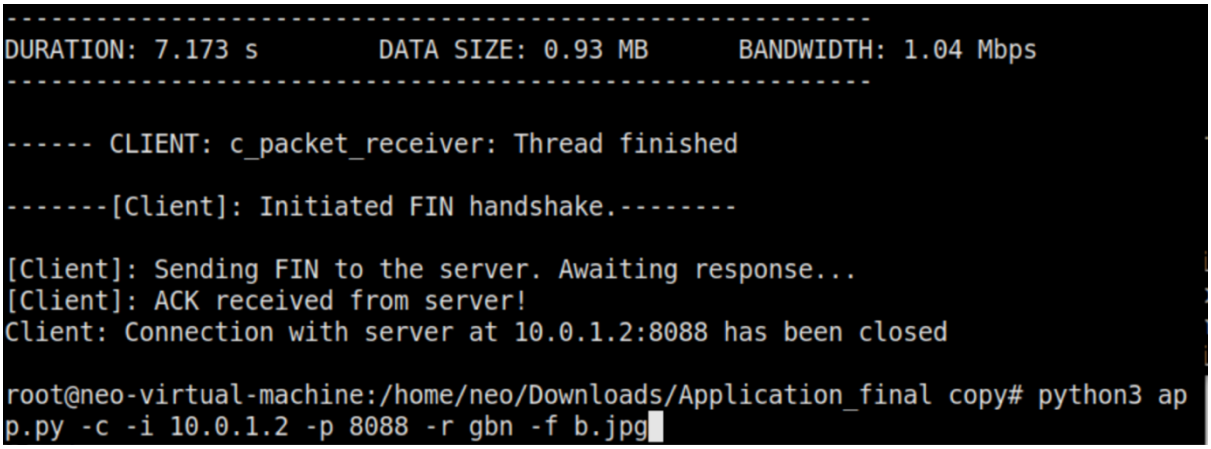
GBN THROUGHPUT VALUES:

When N=5

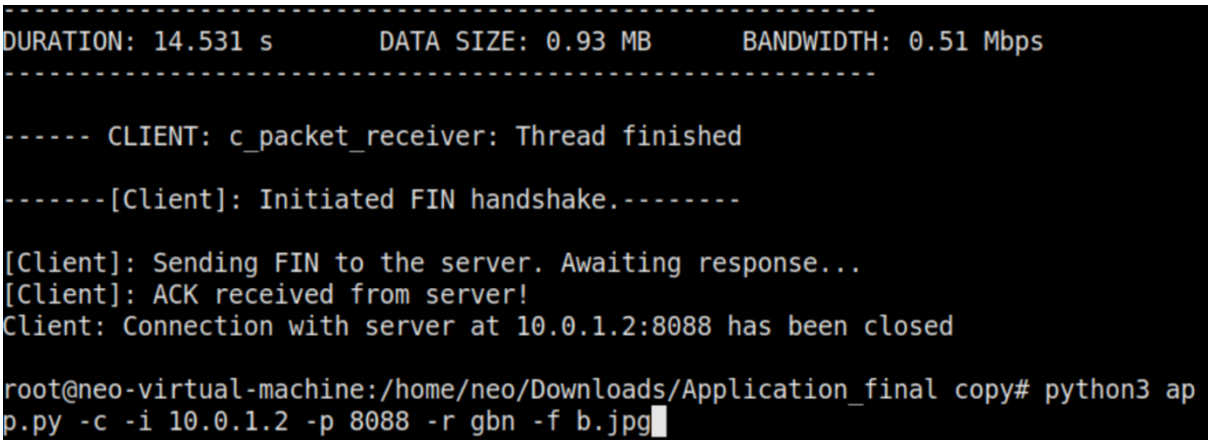
- RTT (25ms):



- RTT (50ms):



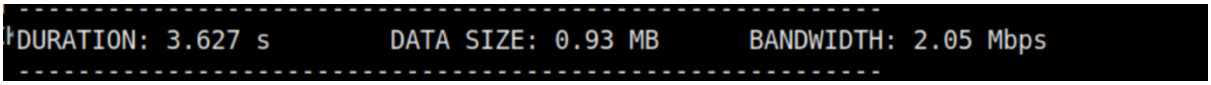
- RTT (100ms):



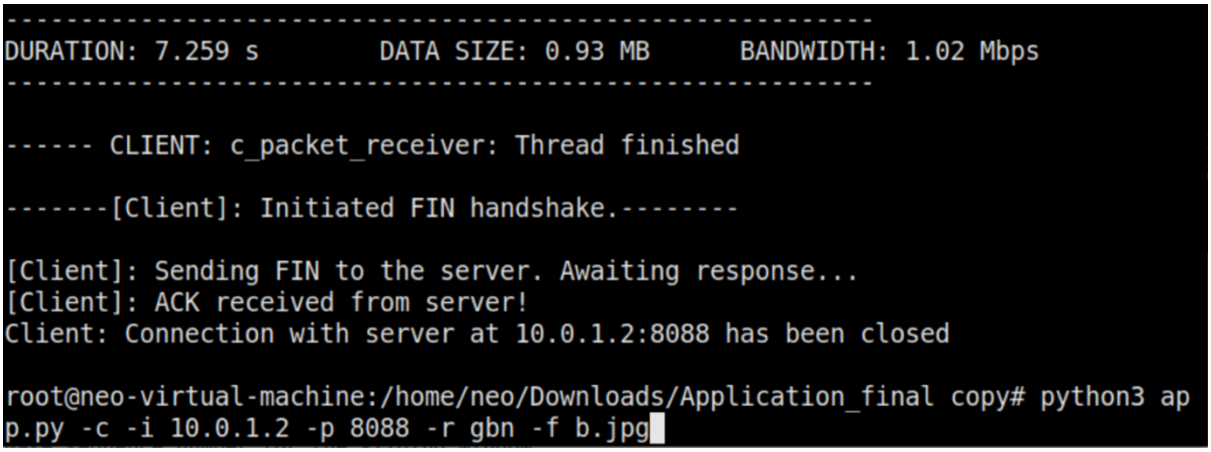
When N=10:

- RTT (25ms):

- RTT (50ms):

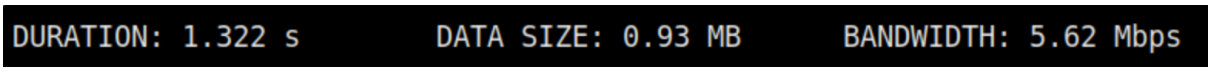


- RTT (100ms):

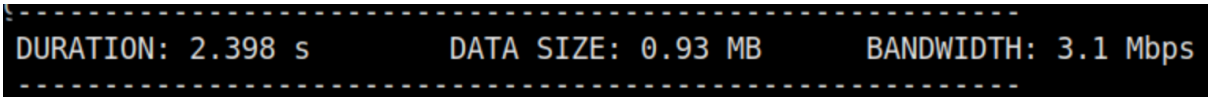


When N=15:

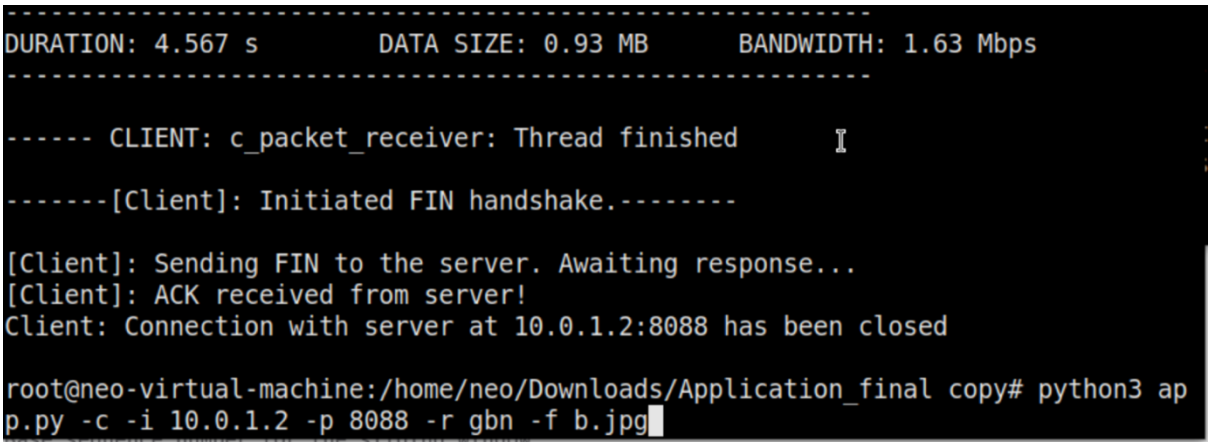
- RTT (25ms):



- RTT (50ms):



- RTT (100ms):



SR THROUGHPUT VALUES:

When N=5:

- RTT (25ms):

```
------------------------------------------------------------
DURATION: 3.807 s        DATA SIZE: 0.93 MB      BANDWIDTH: 1.95 Mbps
------------------------------------------------------------
```

- RTT (50ms):

```
------------------------------------------------------------
DURATION: 7.746 s        DATA SIZE: 0.93 MB      BANDWIDTH: 0.96 Mbps
------------------------------------------------------------
```

- RTT (100ms):

```
------------------------------------------------------------
DURATION: 14.41 s        DATA SIZE: 0.93 MB      BANDWIDTH: 0.52 Mbps
------------------------------------------------------------

------ CLIENT: c_packet_receiver: Thread finished

Client: NO MORE PACKETS TO SEND

------ CLIENT: c_packet_sender: Thread finished

-------[Client]: Initiated FIN handshake.--------

[Client]: Sending FIN to the server. Awaiting response...
[Client]: ACK received from server!
Client: Connection with server at 10.0.1.2:8088 has been closed

root@neo-virtual-machine:/home/neo/Downloads/Application_final copy# python3 ap
p.py -c -i 10.0.1.2 -p 8088 -r sr -f b.jpg
```

When N=10:

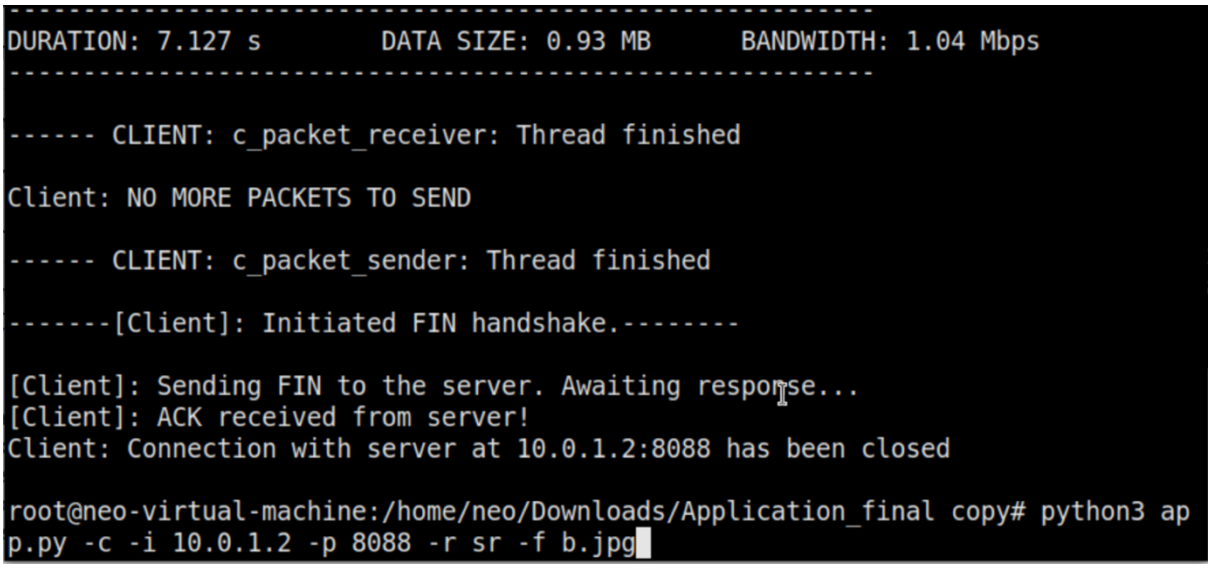
- RTT (25ms):

```
------------------------------------------------------------
DURATION: 1.923 s        DATA SIZE: 0.93 MB      BANDWIDTH: 3.86 Mbps
```

- RTT (50ms):

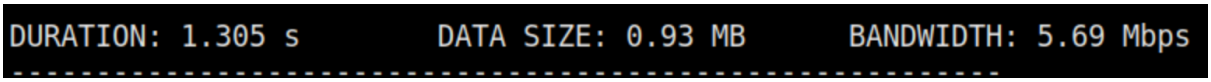
```
------------------------------------------------------------
DURATION: 4.019 s        DATA SIZE: 0.93 MB      BANDWIDTH: 1.85 Mbps
------------------------------------------------------------

------ CLIENT: c_packet_receiver: Thread finished

Client: NO MORE PACKETS TO SEND

------ CLIENT: c_packet_sender: Thread finished

-------[Client]: Initiated FIN handshake.--------

[Client]: Sending FIN to the server. Awaiting response...
[Client]: ACK received from server!
Client: Connection with server at 10.0.1.2:8088 has been closed

root@neo-virtual-machine:/home/neo/Downloads/Application_final copy# python3 ap
p.py -c -i 10.0.1.2 -p 8088 -r sr -f b.jpg
```

- RTT (100ms):

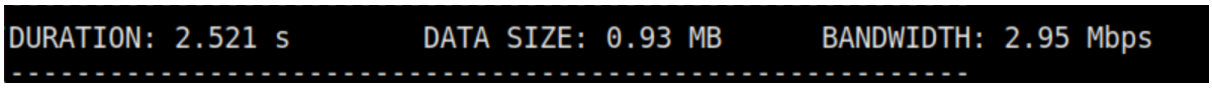


When N=15:

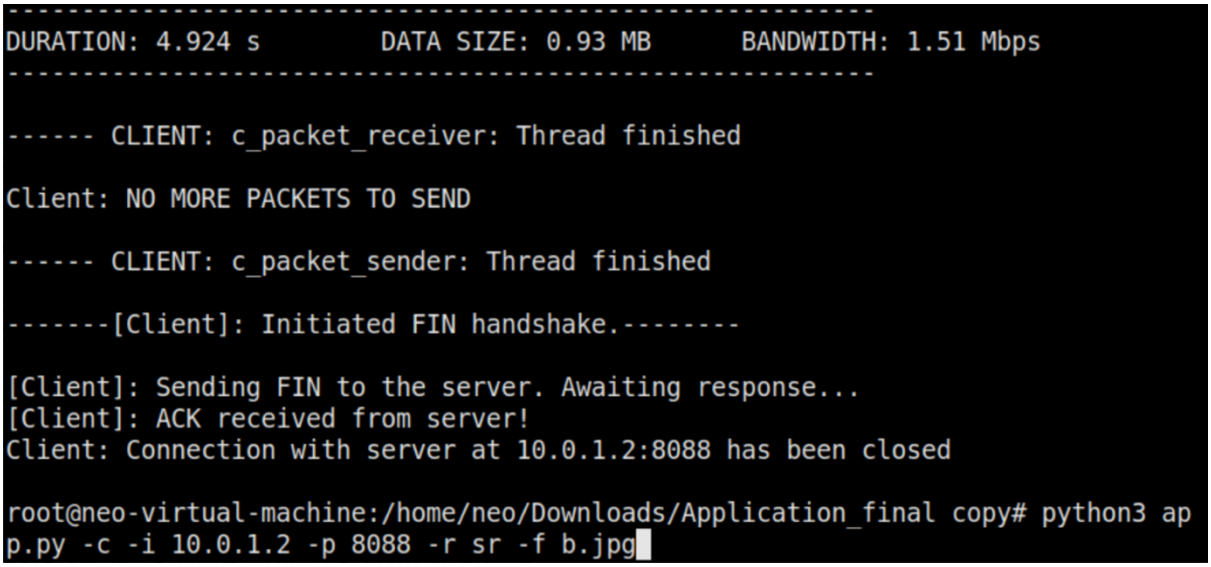
- RTT (25ms):



- RTT (50ms):



- RTT (100ms):



DISCUSSION

Stop-and-Wait Protocol:

- The throughput of the *Stop-and-Wait protocol* is significantly lower than that of the other two protocols. This is because it can only send one packet at a time and must wait for an acknowledgment before sending the next packet.
- As the RTT increases, the time spent waiting for acknowledgments becomes a larger proportion of the total time, which leads to a decrease in throughput. For instance, when RTT increased from 25ms to 50ms, the throughput halved from 0.47 Mbps to 0.23 Mbps. Similarly, when RTT increased from 50ms to 100ms, the throughput again halved from 0.23 Mbps to 0.11 Mbps.
- This pattern illustrates the inherent inefficiency of the *Stop-and-Wait* protocol in situations with high latency, as the sender spends a significant amount of time waiting for acknowledgments.

Go-Back-N Protocol:

- The *Go-Back-N* protocol shows a substantial improvement in throughput over the Stop-and-Wait protocol. This is because *Go-Back-N* allows for multiple packets to be "in-flight" at the same time, thereby better utilizing the available bandwidth.
- The throughput of the *Go-Back-N* protocol also decreases as RTT increases, but the decrease is less dramatic than for the Stop-and-Wait protocol. This is because *Go-Back-N* is less affected by latency thanks to its ability to have multiple in-flight packets.
- The effect of window size (N) is also evident in the *Go-Back-N* results. As N increases from 5 to 15, the throughput improves considerably. This is because a larger window size allows for more in-flight packets, leading to better utilization of the bandwidth. For instance, at 25ms RTT, when N increases from 5 to 15, the throughput increases from 1.95 Mbps to 5.62 Mbps. Similar trends are observed for other RTT values as well.

Selective Repeat Protocol:

- The *Selective Repeat protocol* also shows improved throughput over the *Stop-and-Wait* protocol and similar throughput to the *Go-Back-N* protocol for the same window size.
- As with *Go-Back-N*, the throughput of the *Selective Repeat protocol* decreases with increasing RTT, but the decrease is less pronounced due to the ability to have multiple in-flight packets.
- The effect of window size is similar to that observed for the *Go-Back-N* protocol. As N increases, the throughput increases as well, showing that the Selective Repeat protocol also benefits from a larger window size.

TEST CASE TO SKIP AN ACK AND LOSE:

#Skip an ack with stop and wait protocol:

When we run the test case “skip_ack”, the server will intentionally skip sending the acknowledgment (ACK) for the third packet (when *ack_counter* is 2). This simulates a situation where an ACK is lost in transit.

Server-Side: On the server side, when *ack_counter* equals 2, the server deliberately does not send an ACK for the received packet back to the client. This is to simulate a scenario where the ACK gets lost.

Client-Side: On the client side, after the packet is sent, it will wait for an ACK from the server. However, due to the “skip_ack” test case, the ACK for the third packet will not be received.

Timeout and Resend: Because the ACK is not received, the client will reach a timeout, and it will assume that either the packet or the ACK was lost in transit. As per the stop-and-wait protocol, the client will then resend the same packet.

Continuation: This process continues until the client receives a valid ACK from the server, upon which it will send the next packet.

```
Server is listening on 127.0.0.1:8080 Reliable method: stop_and_wait
Test case: skip_ack
-----Server: Starting handshake process------
```

```
Client: Preparing packet #4
Client: Packet #4 created with ACK #3 and flags 0
Client: Packet #4 sent to server
Client: Timeout, resending the packet
Client: Packet #4 created with ACK #3 and flags 0
Client: Packet #4 sent to server

Client: Received ACK #4 with seq #4 and flags 4
Client: Valid ACK received, preparing the next packet
```

#Skip an ack with SR protocol:

The test case “skip_ack” is designed to simulate the situation where an acknowledgement (ACK) from the server to the client gets lost or is ignored.

In the *sr()* method, the test case “skip_ack” has an effect on the server-side of the protocol, specifically within the *packet_receiver()* function.

This is what happens when we run the “skip_ack” test case:

When a packet is received by the server, it increments an *ack_counter* by one.

If the “skip_ack” test case is active and the *ack_counter* is 2, meaning it is the second packet received, the server will skip sending an acknowledgement back to the client for that packet. It prints a message: Server: 'Skipping' acknowledgement for *packet #{seq}*.

Skipping the acknowledgement effectively simulates a situation where the acknowledgement from the server to the client gets lost in the network.

This will force the client to resend the packet after the timeout period because it has not received an acknowledgement for the packet.

In the Selective Repeat (SR) protocol, only the specific unacknowledged packet will be resent, unlike in the Go-Back-N protocol where all packets sent after the unacknowledged one would also be resent.

This test case thus helps to validate whether the SR protocol implementation is correctly handling lost acknowledgements and is able to recover from such situations.

```
------
Server: Received packet seq #2, ACK #0, and flags 0
Server: 'Skipping' acknowledgement for packet #2 (Test c
ase: 'skip_ack')
```

#Skip an ack with GBN protocol:

The test case “skip_ack” is designed to simulate a situation where an acknowledgment from the server to the client gets lost in the network. Here's what happens when we this test case:

Server Side: The server continuously listens for incoming packets from the client. If the server receives a packet with a sequence number that equals the base sequence number, it appends the packet data to the received data. Usually, the server would then send an acknowledgment back to the client. However, if the test case is “skip_ack” and the sequence number of the packet equals *test_case_num* (which is 2 in the code), the server will “skip” sending this acknowledgment. This action simulates the loss of an acknowledgment during transmission.

Client Side: The client sends packets containing chunks of file data to the server and waits for acknowledgments. In the case of the “skip_ack” test, the client does not receive an acknowledgment for packet number 2.

Handling the Missing Acknowledgment: Since the client does not receive an acknowledgment for the second packet within a certain timeout period, it assumes that the acknowledgment was lost during transmission. As per the *Go-Back-N protocol*, the client then resends all packets in the window starting from the unacknowledged one (packet 2 in this case). It continues to do this until it receives the missing acknowledgment.

Continuation: This process continues with the client sending packets and the server acknowledging them until all packets are sent, received, and acknowledged correctly. The server then writes the received data to a file.

```
------
Server: Received packet seq #2, ACK #0, and flags 0

Server: Checking if packet seq #2 equals base 2
Server: 'Skipping' acknowledgement for packet #2 (Test c
ase: 'skip_ack')

------
```

TEST CASE TO SKIP A SEQUENCE NUMBER:

#Skip a sequence number with GBN protocol:

The “loss” test case is designed to simulate a situation where a packet from the client to the server gets lost in the network. Here's what happens when we run this test case:

Client Side: When the client is ready to send a packet with a sequence number that equals *test_case_num* (which is 2 in the code), and if the test case is “loss”, instead of sending the packet as usual, the client will skip sending this particular packet. This simulates a lost packet during transmission.

Server Side: The server continuously listens for incoming packets from the client. Since the second packet was not sent by the client, the server will not receive it. The server is expecting packets with sequence numbers in order. When it receives a packet with a sequence number greater than the base (indicating a missing packet), it will not process it, but will instead store it in *window_packets* to be processed later.

Handling the Missing Packet: Since the server does not receive the packet with sequence number 2, it will not send an acknowledgment for this packet. The client, after not receiving an acknowledgment for the second packet within a certain timeout period, assumes that the packet was lost during transmission. As per the *Go-Back-N* protocol, the client then resends all packets in the window starting from the unacknowledged one (packet 2 in this case). It continues to do this until it receives the missing acknowledgment.

Continuation: This process continues with the client sending packets and the server acknowledging them until all packets are sent, received, and acknowledged correctly. The server then writes the received data to a file.

```
Client: Received ACK #1 with flags 0
Client: Popped packet #1 from window_packets

Client: RESEND Window

Client: RESEND Window

Client: RESEND Window

Client: RESEND Window
```

#Skip a seqence number with SR protocol:

The “loss” test case simulates the condition where a packet is lost during transmission from the client to the server. This situation is handled within the *c_packet_sender()* function on the client-side of the protocol in the *sr()* method.

Here is what happens when we run the “loss” test case:

As the client prepares to send packets, it first divides the file data into chunks to fit into the packets.

For each chunk, it creates a packet and prepares to send it. The sequence number of the packet being sent is checked against the *test_case_num* which is defined as 2 at the start of the function.

If the “loss” test case is active and the sequence number of the packet being sent is the same as *test_case_num,* the client will skip sending that packet. It will print a message: Client: skipped sending packet #{*c_next_seq_num*} (Test case: 'lose').

This simulates a situation where a packet is lost during transmission, and the server never receives it.

After a timeout period, the client checks the time each packet in the window was sent. If the time elapsed since a packet was sent is more than the timeout, the client resends the packet.

In this case, because the client never received an acknowledgement for the “lost” packet (because it was never actually sent), the client will resend the packet after the timeout period.

This test case helps to verify that the Selective Repeat (SR) protocol implementation is correctly handling packet loss and is able to recover from such situations by resending the lost packet after a timeout.

```
Client: Creating chunk #2
Client: Created packet #2 with flags 0
Client: skipped sending packet #2 (Test case: 'lose')
```

```
------
Client: Received ACK #6 with flags 0
Client RESENT packet: 2
```

#Skip a sequence number with stop and wait protocol:

When we run the “lose” test case, the client will intentionally skip sending the second packet (when *packet_counter* is 2). This simulates a situation where a packet is lost in transit.

Client-Side: On the client side, when *packet_counter* equals 2, the client deliberately does not send the second packet. This is to simulate a scenario where a packet gets lost during transmission.

Server-Side: The server, waiting for the second packet, will not receive it. After a certain timeout period, it will realize that the packet has been lost. However, in a simple stop-and-wait protocol as implemented in the code, the server doesn't have a mechanism to request a specific packet. It only waits for the packets to come in order.

Resend: Back on the client side, because the client didn't receive an ACK for the second packet (as it was never sent), it will reach a timeout and resend the second packet.

This process continues until the client receives a valid ACK from the server for the resent packet, upon which it will send the next packet.

```
Client: Preparing packet #2
Client: Packet #2 created with ACK #1 and flags 0
Test case 'lose': Client is skipped sending packet #2
Client: Timeout, resending the packet
Client: Packet #2 created with ACK #1 and flags 0
Client: Packet #2 sent to server
```

#GBN test case “double”:

When the “double” test case is set, it's simulating a scenario where the client sends a packet twice. Let's assume that this happens at the second packet for simplicity. Here's how the client and server would behave under this condition:

1. The client starts the transmission by breaking the data into chunks and creating the first packet. This is sent to the server.
2. The server receives the packet, sends an ACK (acknowledgement) back to the client, and the base is increased by one.
3. The client receives the ACK, removes the packet from its window, and prepares the next packet.
4. Now, the client creates the second packet. Because the test case is set to “double”, the client deliberately sends this packet twice to the server. This is not a typical scenario but simulates the case where a packet might be duplicated due to some network issue.

```
------
Client: Creating chunk #2
Client: Created packet #2 with flags 0
Test case 'DOUBLE': Client is sending packet #2 twice

------
Client: Received ACK #1 with flags 0
Client: Popped packet #1 from window_packets
Client: Sent packet #2 to server
------
Client: Packet #2 sent to server twice
```

5. The server receives the second packet, sends an ACK back to the client, and increases the base by one. Shortly after, it receives what appears to be a retransmission of the second packet.
6. At this point, the server checks the sequence number of the received packet. It finds that it's the same as the base sequence number. In a typical scenario, this would indicate that this is the packet the server expects next. However, in this case, the server has already received this packet and sent an ACK.
7. The server sends another ACK for the second packet back to the client. This is because the Go-Back-N protocol doesn't have a mechanism to detect duplicate packets at the receiver side. It always sends an ACK for any packet it receives that has a sequence number equal to the base.

```
------
Server: Received packet seq #2, ACK #0, and flags 0

Server: Checking if packet seq #2 equals base 2

Server: Created ACK packet #2, with flags 0
Server: Sent ACK packet #2 to client
------

------
Server: Received packet seq #2, ACK #0, and flags 0

------
Server: Received packet seq #3, ACK #0, and flags 0

Server: Checking if packet seq #3 equals base 3

Server: Created ACK packet #3, with flags 0
Server: Sent ACK packet #3 to client
------
```

8. The client receives the second ACK for the second packet. Since it has already received the first ACK and moved the base, this ACK doesn't change anything in the client's state.
9. The client then continues to prepare and send the rest of the packets.
10. The server, after sending the second ACK for the second packet, expects the third packet (because it has increased the base by one after receiving the second packet). It proceeds as usual once it receives the third packet, oblivious of the fact that it had previously processed a duplicated packet.

In essence, the “double” test case does not disrupt the flow of the Go-Back-N protocol. The protocol is designed to handle packet retransmissions (which could occur due to packet losses), and a duplicated packet at the receiver side just appears as a normal retransmission.

#SR Test case “double”:

The “double” test case in the SR protocol is set up to deliberately send a packet twice from the client to the server to test how the system behaves under these conditions. Here is the detailed step-by-step behavior:

1. **Client-side behavior:**

    1. The client begins by preparing to send packets in chunks of data to the server.
    2. As the client continues to send packets, it keeps a count of the packets it has sent.
    3. When the client is about to send the second packet (where the *packet_counter* == *2*), it checks the “double” test case condition.
    4. If the test case is “double”, it sends the second packet twice in succession to the server, essentially simulating a duplicate packet scenario. This situation might occur in a real-world network due to issues like network hiccups, where the client might think that the packet was not sent successfully and hence attempts to send it again.

    5. The client continues to send the rest of the packets as normal, keeping track of the sent packets and awaiting ACKs from the server.

```
Client: Creating chunk #2
Client: Created packet #2 with flags 0
Client: Sent packet #2 to server

------
Client: Received ACK #1 with flags 0
Client: Sent packet #2 to server
Test case 'DOUBLE': Client is sending packet #2 twice

------
Client: Creating chunk #3
Client: Created packet #3 with flags 0

------
Client: Received ACK #2 with flags 0

------
Client: Received ACK #2 with flags 0
Client: Sent packet #3 to server
------
```

2. **Server-side behavior:**

    1. The server listens for incoming packets from the client.
    2. When it receives the second packet for the first time, it sends an ACK back to the client and increments the expected sequence number.
    3. When it receives the second packet again (the duplicate), it recognizes that this is a packet it has already received and acknowledged.
    4. Even in this case, the server sends an ACK back to the client for the received packet. This is because in the Selective Repeat protocol, the receiver sends an ACK for every correctly received packet, irrespective of its order.

```
------
Server: Received packet seq #2, ACK #0, and flags 0

Server: Created ACK packet #2, with flags 0
Server: Sent ACK packet #2 to client
------

------
Server: Received packet seq #2, ACK #0, and flags 0

Server: Created ACK packet #2, with flags 0
Server: Sent ACK packet #2 to client
------

------
Server: Received packet seq #3, ACK #0, and flags 0

Server: Created ACK packet #3, with flags 0
Server: Sent ACK packet #3 to client
------
```

    5. Since the server is already expecting the next sequence number, the duplicate packet does not get written into the received data.
    6. The server continues to listen for the rest of the packets, always expecting the packet with the next sequence number.

In this way, the “double” test case allows the SR protocol to demonstrate its robustness and ability to handle duplicate packets. Despite the client sending a duplicate packet, the protocol ensures that the data received by the server remains correct and in order.

CONCLUSIONS

Throughout this project, we have successfully implemented the DATA2410 Reliable Transport Protocol (DRTP) that ensures reliable, in-order data delivery without missing data or duplicates over UDP. The DRTP consisted of three main components: The *Stop-and-Wait* protocol, the *Go-Back-N (GBN)* protocol, and the *Selective Repeat (SR)* protocol. Each was tested under varying network conditions, including lost packets, lost acknowledgments, and duplicate packets.

The *Stop-and-Wait* protocol was effective in its simplicity, ensuring the reliability of data transmission through its method of waiting for an acknowledgment after each packet sent. However, its inefficiency in network utilization was evident, particularly in cases where an acknowledgment or packet was lost, as it required the client to resend the packet after a timeout period.

The *Go-Back-N* protocol presented a more efficient strategy, implementing a fixed window size for packet transmission. It effectively dealt with lost packets by resending all unacknowledged packets within a given timeout. However, its handling of lost acknowledgments was less efficient than the Selective Repeat protocol, as it required resending all packets from the lost acknowledgment onwards. Moreover, in the event of out-of-order delivery, the receiver discarded the packets, indicating a potential room for improvement.

The *Selective Repeat protocol*, on the other hand, showcased the highest efficiency among the three. It managed lost packets and acknowledgments by only resending the specific unacknowledged packet. Furthermore, it handled out-of-order deliveries by putting the packets in the correct place in the receive buffer, optimizing the performance of the protocol.

The file transfer application performed reliably with all three protocols, with varying degrees of throughput values based on the chosen protocol and given network conditions. It demonstrated the effectiveness of DRTP in ensuring reliable data transfer over UDP.

Testing and experimentation also revealed the efficacy of the DRTP in handling edge cases such as loss, reordering, and duplicates in a network. The inclusion of test cases for skipping acknowledgments and sequence numbers provided valuable insights into the behavior of each protocol under these conditions.

In conclusion, the DATA2410 Reliable Transport Protocol (DRTP) has proven to be an effective solution for reliable data transfer over UDP. By understanding the strengths and weaknesses of each protocol, developers can make informed decisions when choosing a protocol for their specific application, thereby enhancing reliability and efficiency in data transmission over networks.

REFERENCES