174 - DATA2410 Reliable Transport Protocol

Introduction

In this portfolio, we will go through implementation of reliable transport protocol which provides reliability on top of UDP and test how it interacts in a virtual network with loss and delay. This document's technical section will cover the implementation of application.py and DRTP.py, which will carry out different data transactions over the network in a dependable manner. In addition to guaranteeing dependable data delivery, the protocol will also guarantee accurate data ordering and the absence of duplicates. The other aspect of this portfolio will be testing the application to ensure that reliability is working correctly in order to handle exceptions such as packet loss, ordering issues and duplicates.

The application will take a file like a jpg or txt, then it will convert the file to an array of payloads. The payloads will be sent over DRTP and will be received by the receiver in a reliable way. This solution mainly takes inspiration from a preexisting reliable transport protocol known as TCP, which has been fine-tuned over several decades to handle reliable transmission of data via networks. We used Python socket programming, where each socket uses UDP to deliver data from a sender to a recipient, to replicate the dependable components of TCP. Because UDP is inherently unreliable, we will use TCP principles to address this shortcoming. Firstly, we establish a communication channel by conducting a three-way handshake. We then transmit the data using the reliability mode Go-Back-N (GBN). Ultimately, once the data transfer is finished, we will gracefully close the connection with a two-way handshake, this differs from how TCP handles its teardown where it uses a four-way handshake.

In order for the reader to fully comprehend the workings of our Data Reliable Transport Protocol (DRTP) and to fully grasp what is the application is doing. In the report we compare theoretical concepts to our code. In the second section we will test out code in Mininet with delay and loss. For the test we have to rely on the code we have implemented and if it is functional. We will start up two hosts and utilise one as a sender and the other as a receiver in the Mininet topology environment. There will be a router positioned between the two hosts, and it is connected to them via links. The test will demonstrate data transmission through the throughput. Lastly, we will draw our conclusions based on what we expectati and results and discuss the impact of our work.

We will not discus how to run the application and what the flags do as this is shown in the README.md file and how the file is converted to bytes and back to a file as this does not have much to do with how the protocol works.

Implementation

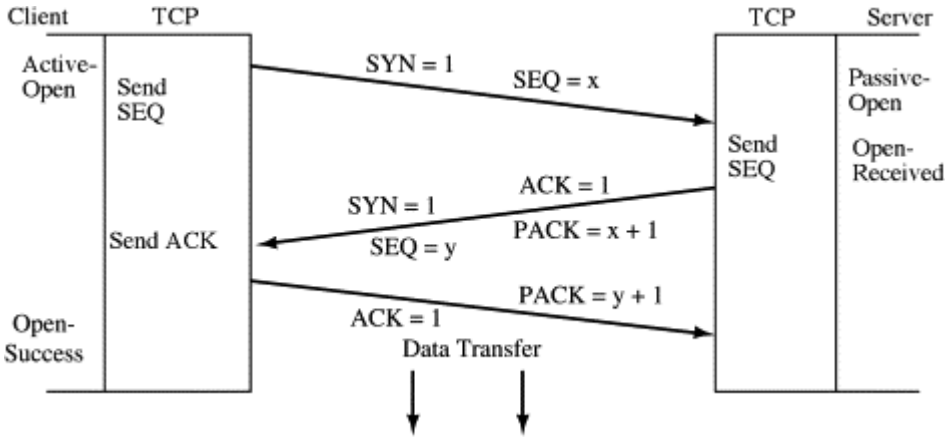
We will now delve into the theoretical foundation of our application. From the start of the three-way handshake, then a in depth look at how the Go-Back-N (GBN) and cases where the packets get lost to see how it is reliable, and then finally how the teardown is executed by using a two-way handshake.

The DRTP header facilitates transmission of data over UDP (User Datagram Protocol) by adding essential information before application data. Comprising three fields, Sequence Number, Acknowledgment Number, and Flags, the DRTP header employs a 16-bit format for each field. Application data is sent in 994-byte chunks, with each packet consisting of the DRTP header followed by the data. The total size of each packet, including the header and application data, amounts to 1000 bytes.

The flags field are SYN, ACK, FIN, and a reserved RST flag, offers additional functionalities such as connection establishment and teardown. Through this structure, the DRTP header ensures reliable communication between sender and receiver and provide necessary protocol functionalities.

Three-way handshake

The three-way handshake serves two main purposes. The purpose of the handshake is to ensure that both parties are aware that they are prepared to transfer data and to agree on the first sequence numbers, which are sent and acknowledged during the exchange to ensure there are no misunderstandings. The sequence numbers are usually random so that there are no hijackers to malicious send or receive data.



Source: https://www.sciencedirect.com/topics/computer-science/three-way-handshake

Step 1 (SYN): The client transmits with SYN flag bit on in the first step to establish a connection with the server where it sends its own (Synchronise Sequence Number). This tells the server when the client is likely to start communication and what sequence number it starts with.

Step 2 (ACK + SYN): The server sets the SYN-ACK flag bits in response to the client request. The server acknowledges the sequence number and a phantom packet from the client with the ACK flag bit on and sending to back to the client. It also sends its own (Synchronise Sequence Number) with the SYN flag bit on.

Step 3 (ACK): The client acknowledges the server's sequence number and a phantom packet with ACK flag bit on, and they both establish a trustworthy connection to begin the actual data transfer.

All the first Synchronise Sequence Number are randomly generated to avoid imposters in TPC but for simplicity we don't have any random generation.

Here is how the application has handled the three-way handshake:

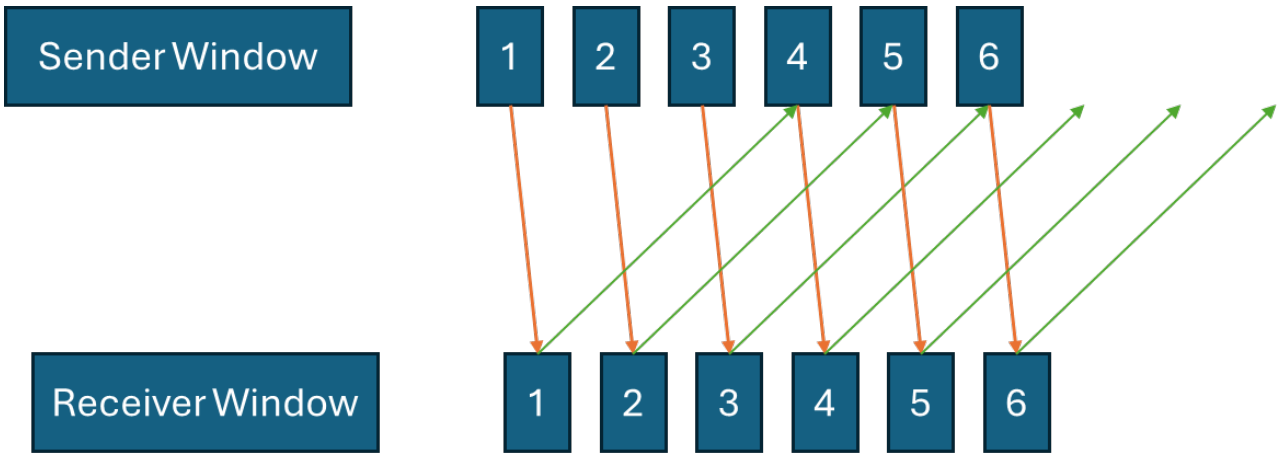
```python
# Send SYN packet
packet = send_packet(seq_num, 0, set_flags(1, 0, 0, 0))
client_socket.send(packet)
print("SYN packet is sent")
print_header(packet[:6], True)

# Receive SYN-ACK packet
packet = client_socket.recv(DRTP_struct.size)
ack_num, seq_num, flags = unpack_header(packet)
if flags[0] == 1 and flags[1] == 1:
    print("SYN-ACK packet is received")
    print_header(packet[:6], False)
else:
    print("SYN-ACK packet is not received")
    socket.error("SYN-ACK packet is not received")

# Send ACK packet
ack_num += 1
packet = send_packet(seq_num, ack_num, set_flags(0, 1, 0, 0))
client_socket.send(packet)
print("ACK packet is sent")
print_header(packet[:6], True)
```

```python
# Receive SYN packet
packet, client_address = server_socket.recvfrom(DRTP_struct.size)
ack_num , _, flags = unpack_header(packet)
print("SYN packet is received")
print_header(packet[:6], False)

# Send SYN-ACK packet
seq_num = 0
if flags[0] == 1:
    packet = send_packet(seq_num, ack_num + 1, set_flags(1, 1, 0, 0))
    server_socket.sendto(packet, client_address)
    print("SYN-ACK packet is sent")
    print_header(packet[:6], True)
else:
    socket.error("SYN packet is not received")

# Receive ACK packet
packet, _ = server_socket.recvfrom(DRTP_struct.size)
ack_num, seq_num, flags = unpack_header(packet)
if ack_num == seq_num + 1 and flags[1] == 1:
    print("ACK packet is received")
    print_header(packet[:6], False)
else:
    socket.error("ACK packet is not received")
```

Go-Back-N (GNB)

Go-Back-N is used in networks to ensure reliable data transmission. It allows the sender to send multiple payloads before needing an acknowledgment for the first one, this enhances efficiency and throughput. However, if an error is detected in any packet, Go-Back-N requires retransmission of that packet and all the other ones in the window, ensuring that data is received correctly and in order. This protocol is particularly useful in scenarios where reliable delivery is critical. The file and filename is converted to bytes before the start of the GBN and is written back to the filename in the output folder.

No Loss in GNB



When there is no loss, the Go-Back-N protocol operates smoothly as follows:

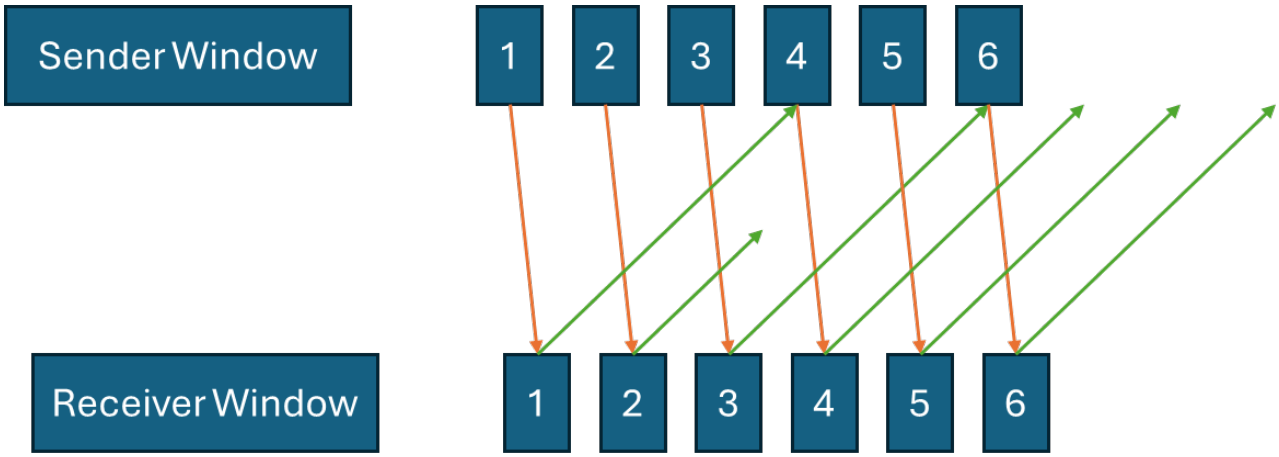
1. **Sender Window:** The sender maintains a window of size e.g. 3, this represents the number of packets it can send without needing an acknowledgment for the first packet in the window.
2. **Packets Transmission:** The sender sends the packets one after another within the window size. Each packet is sequentially numbered.
3. **Acknowledgments:** The receiver receives each packet and sends an acknowledgment (ACK) back to the sender for the last correctly received packet.
4. **Sliding Window:** As the sender receives acknowledgments for each packet, it slides its window forward, allowing it to send new packets. E.g. if the window size is 3 and packets from 1 to 3 have been sent, upon receiving an ACK for packet 1, the sender can now send packet 5 and so on.

In this scenario with no loss so every packet is successfully received and acknowledged in order. This results in efficient and continuous data transmission. Here in my code, you can see how that has been implemented. In the header no flags are set as we are only sending data and not SYN, ACK or FIN.

```
while True:
    # Send the packets
    while(len(slidding_window) < window_size and seq_num < len(payload)):
        seq_num += 1
        slidding_window.append(seq_num)
        if seq_num > len(payload):
            break
        packet = send_packet(seq_num, ack_num, set_flags(0, 0, 0, 0), payload[seq_num - 1])
        client_socket.send(packet)
        print(f"{datetime.datetime.now().strftime('%H:%M:%S.%f')} -- packet with seq = {seq_
        print_header(packet[:6], True)

    # Receive the ACKs
    try:
        packet = client_socket.recv(DRTP_struct.size)
        _, check_ack_num, flags = unpack_header(packet)
        if check_ack_num == exspected_ack:
            slidding_window.pop(0)
            exspected_ack += 1
            print(f"{datetime.datetime.now().strftime('%H:%M:%S.%f')} -- ACK for packet = {c
            print_header(packet[:6], False)
```

```
# Send ACK for the received packet
if ack_num <= excpected_ack_num and not flags[2] == 1: # Will send ack for packets in order and any previous
    print(f"{datetime.datetime.now().strftime('%H:%M:%S.%f')} -- packet {ack_num} is received")
    # add the payload to the list if the packet is not a duplicate
    if ack_num == excpected_ack_num:
        packets.append(packet)
    else:
        print(f"{datetime.datetime.now().strftime('%H:%M:%S.%f')} -- duplicate packet {ack_num} is received")
    packet = send_packet(seq_num, ack_num + 1, set_flags(0, 1, 0, 0))
    server_socket.sendto(packet, client_address)
    excpected_ack_num += 1
    print(f"{datetime.datetime.now().strftime('%H:%M:%S.%f')} -- sending ack for the received {ack_num}")
    print_header(packet[:6], True)
```

Receiver Side Loss (GBN)



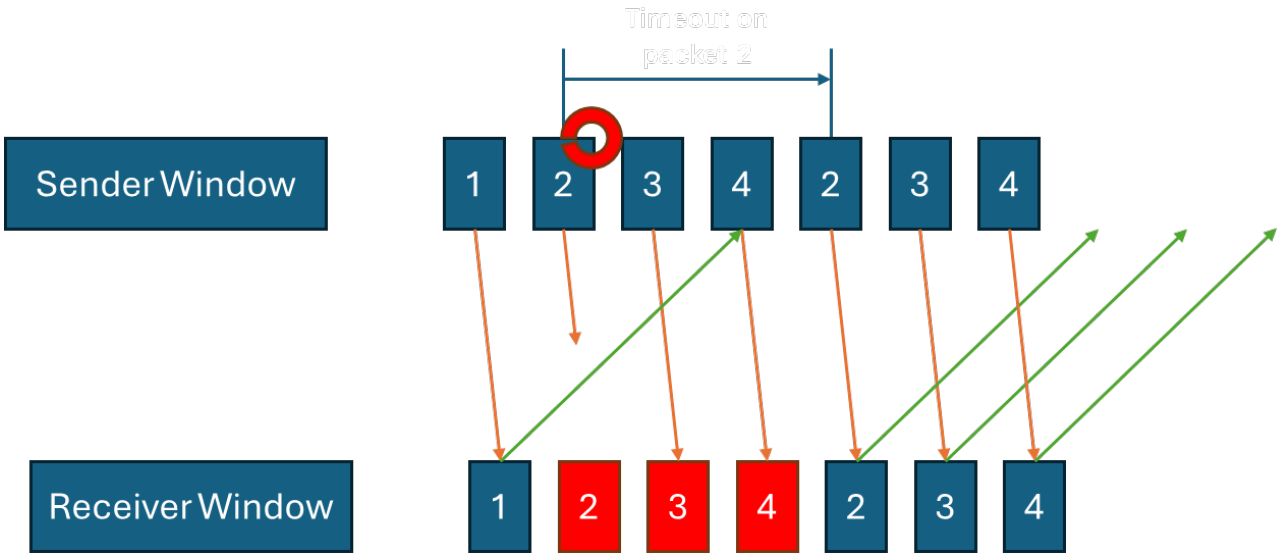
How the Go-Back-N protocol handles a situation where an acknowledgment (ACK) is lost at the receiver side:

1. **Loss Occurs:** Let's assume the ACK for packet 2 is lost during transmission back to the sender. Indicating that the receiver has received packets 1, 2 and 3.
2. **Sliding Window at Sender:** The sender receives ACK for packet 1. It slides its window forward accordingly. The sender does not receive the ACK for frame 2 due to the loss.
3. **Cumulative Acknowledgment Received:** The sender receives ACK for packet 3 (the cumulative ACK), which indicates that frames 1, 2 and 3 have been correctly received by the receiver.
4. **Sliding Window Forward:** Upon receiving the cumulative ACK for packet 3, the sender slides its window forward to 4, 5 and 6, and then sending them bringing us back to track.

This ensures that the Go-Back-N protocol efficiently handles the loss of ACKs by relying on cumulative acknowledgments to move the sender's window forward, avoiding unnecessary retransmissions. This is simple to implement in the code, as if we receive an ACK packet, we can just loop to that packet + 1.

```
# Receive the ACKs
try:
    packet = client_socket.recv(DRTP_struct.size)
    _, check_ack_num, flags = unpack_header(packet)
    if check_ack_num == exspected_ack:
        slidding_window.pop(0)
        exspected_ack += 1
        print(f"{datetime.datetime.now().strftime('%
        print_header(packet[:6], False)
    else:
        while check_ack_num != exspected_ack:
            slidding_window.pop(0)
            print(f"{datetime.datetime.now().strftim
            exspected_ack += 1
        print_header(packet[:6], False)
```

Sender Side Loss (GBN)



How the Go-Back-N protocol handles a situation where there is a loss from the sender:

1. **Loss Occurs:** Let's assume packet 2 is lost during transmission.
2. **Acknowledgments:** The receiver receives packets 1 correctly and sends acknowledgment (ACK) for the packet. The receiver does not receive packet 2, so it does not send an ACK for it. The receiver receives packets 3 and 4 but must discard them because packet 2 is missing. It should continue to acknowledge the highest correctly received packet in sequence, but for simplicity we don't.
3. **Sliding Window at Sender:** The sender receives ACK for packet 1. It slides its window forward, assuming the next packets will be acknowledged in order. The sender continues sending packet within its window limit which is 4.
4. **Timeout for Missing ACK:** The sender does not receive an ACK for packet 2 within the expected timeout period (500ms). The timeout triggers the sender to realize that packet 2 and all subsequent packets up to the current window need to be retransmitted as the server has discarded them.
5. **Retransmission:** The sender retransmits all the packet numbers in its window so packets 2, 3 and 4 are resent. This fixes the lost packets and the packets that have been discarded.

This process ensures that the Go-Back-N protocol can recover from lost packets and maintain reliability and ordered. Here in the code, you can see that we loop thought each window and retransmit the packet.

```
# Resend window on timeout
except socket.timeout:
    print(f"{datetime.datetime.now().strftime('%H:%M:%S.%f')} -- RTO occured")
    for seq_num in slidding_window:
        if seq_num > len(payload):
            break
        packet = send_packet(seq_num, ack_num, set_flags(0, 0, 0, 0), payload[seq_num - 1])
        client_socket.send(packet)
        print(f"{datetime.datetime.now().strftime('%H:%M:%S.%f')} -- retransmitting packet w
```

Teardown/Two-way Handshake

The teardown is kept very simple. we will gracefully close the connection so that we make the sockets can be reused later. In the code you can see that after all the data has been sent the client sends a header with the FIN flag bit on. The server receives it and notifies the server that it has received the flag by sending it FIN and ACK flag bits on. This will close both the server and client sockets. The server will repack the payload to the output folder.

```python
# Send FIN packet after sending all the packets
if len(slidding_window) == 0 and seq_num == len(payload):
    packet = send_packet(check_ack_num, ack_num, set_flags(0, 0, 1, 0))
    client_socket.send(packet)
    print("\nDATA Finished\n\nConnection Teardown:\n\nFIN packet is sent")
    print_header(packet[:6], True)

    # Receive FIN-ACK packet
    packet = client_socket.recv(DRTP_struct.size)
    _, check_ack_num, flags = unpack_header(packet)
    if flags[2] == 1 and flags[1] == 1:
        print("FIN-ACK packet is received\nConnection Closes")
        print_header(packet[:6], False)
        client_socket.close()
        break
    else:
        print("FIN-ACK packet is not received")
        print_header(packet[:6], False)
        socket.error("FIN-ACK packet is not received")
```

```python
# FIN packet is received
elif flags[2] == 1:
    print("\nFIN packet is received")
    print_header(packet[:6], False)

    # Send FIN-ACK packet
    packet = send_packet(seq_num, ack_num + 1, set_flags(0, 1, 1, 0))
    server_socket.sendto(packet, client_address)
    print("FIN-ACK packet is sent\n")
    print_header(packet[:6], True)
    server_socket.close()
    break
else:
    print(f"{datetime.datetime.now().strftime('%H:%M:%S.%f')} -- out-of
```

Discussion

Window Size	50ms RTT	100ms RTT	200ms RTT
3	416.70 Kbps	211.62 Kbps	109.69 Kbps
5	687.04 Kbps	350.77 Kbps	183.66 Kbps
10	1.33 Mbps	697.34 Kbps	358.05 Kbps

Discussion#1 How Window Size Influences the Throughput

Reduced Idle Time:

Larger window sizes in the Go-Back-N protocol reduce idle time through several mechanisms.

Firstly, they allow more packets to be in flight simultaneously. E.g. with a larger window size such as 10, the sender can transmit packets 1 through 10 before needing an acknowledgment for packet 1. Compared to only packets 1, 2, and 3 with a window size of 3. This increase means more packets are "in flight" at any given time, keeping the sender occupied and reducing idle periods.

Additionally, larger window sizes enable continuous transmission. In smaller window sizes, the sender frequently pauses to wait for acknowledgments, creating idle time between the transmission of packet groups. With larger windows, the sender can continue sending new packets while waiting for acknowledgments of earlier packets. This overlap allows for a more consistent flow of data, minimizing idle periods and increasing overall throughput.

Moreover, larger window sizes facilitate the efficient use of network bandwidth. By maintaining continuous transmission, the sender ensures that the available bandwidth is fully utilized. In contrast, smaller window sizes result in frequent stops for acknowledgments, leaving parts of the bandwidth unused during idle times. With larger windows, more of the available bandwidth is effectively employed for data transmission, reducing waste and improving throughput.

Discussion#2 How RTT Influences the Throughput

50ms RTT:

With a lower RTT, acknowledgments are received faster, reducing the time the sender spends waiting.

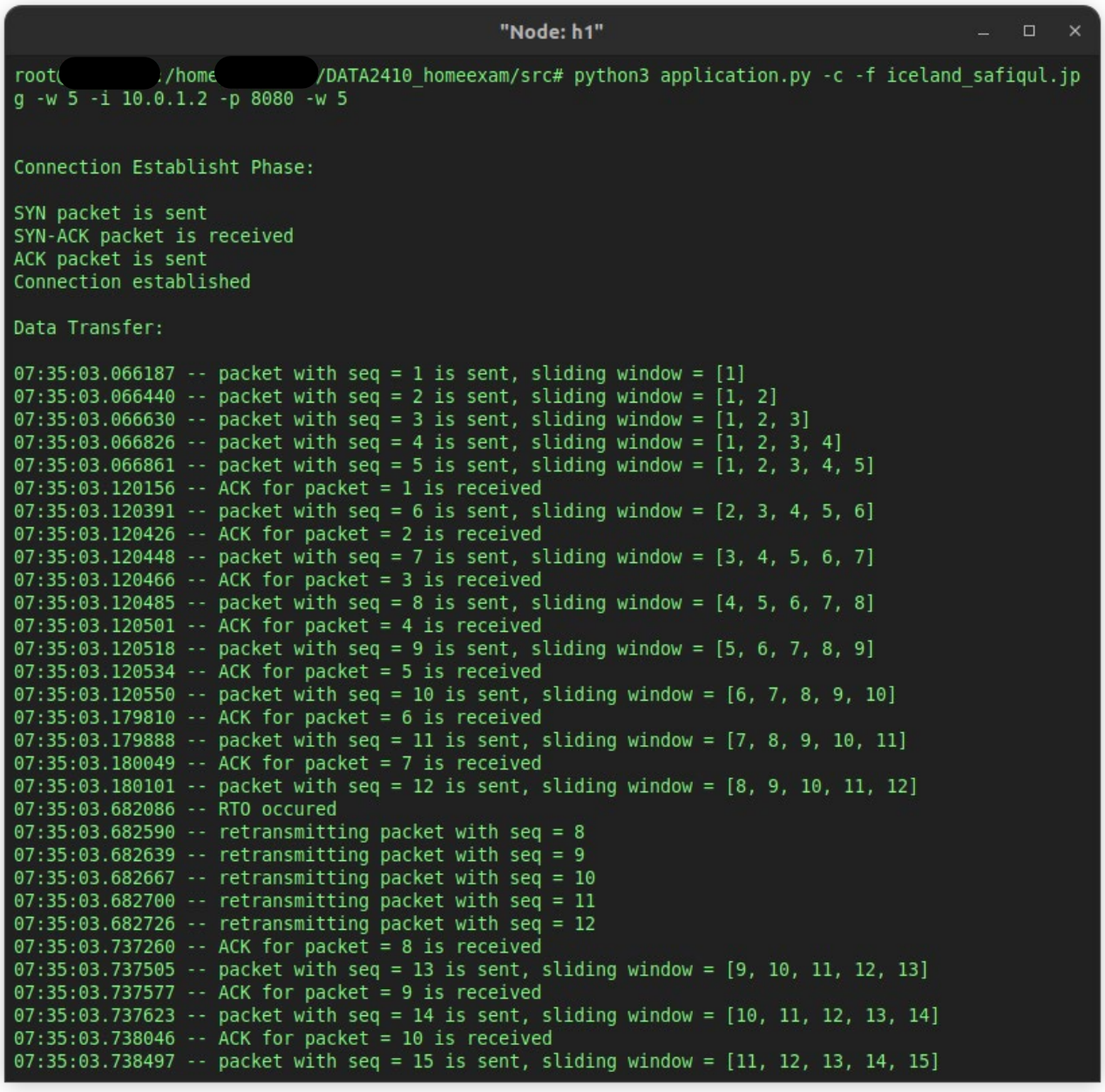
Smaller RTT values generally will result in higher throughput because the sender can transmit more packets in a given time frame.

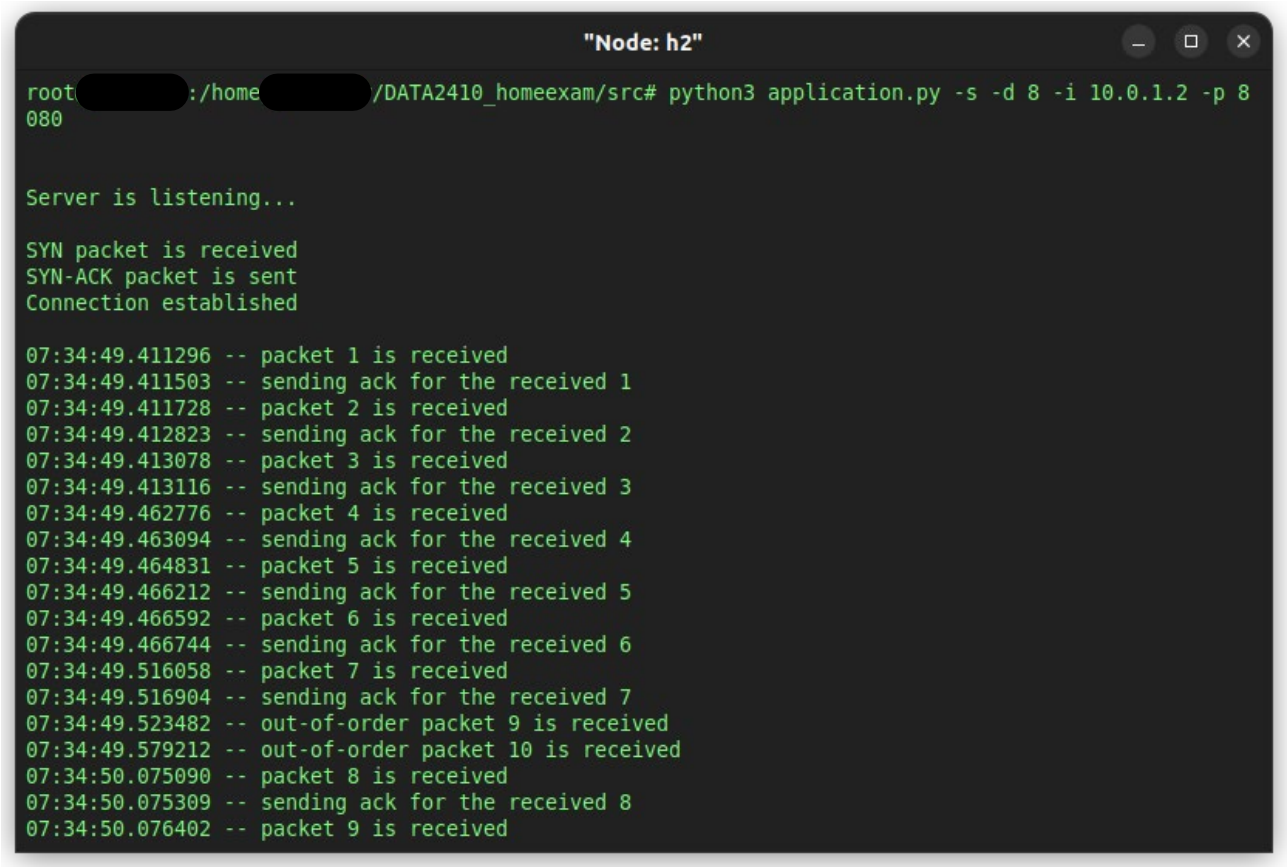
200ms RTT:

With a larger RTT, acknowledgments take longer to return to the sender, increasing the time the sender spends waiting.

Larger RTT values may lead to lower throughput because the sender must wait longer before it can send new packets.

Discussion#3 Show the Reliability





Here we can see how the code handles RTO where it loops thought the window sending the packet again. The server discards the packets that are not in order and after 500ms the client resends all of the window from 8 to 12. More in-depth explanation is given in the implementation of the client side packet loss.

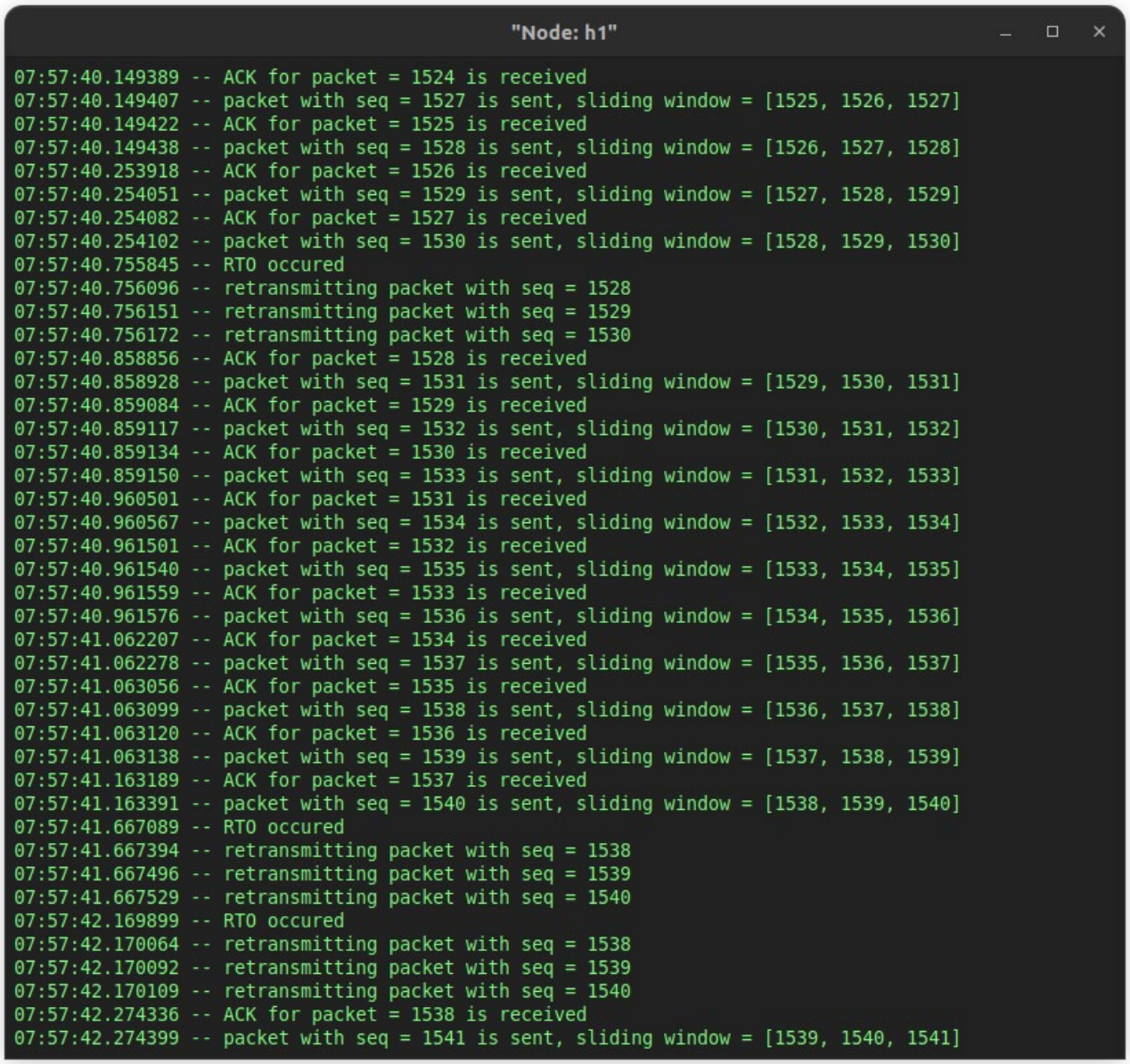
Inspera candidate nr. 174

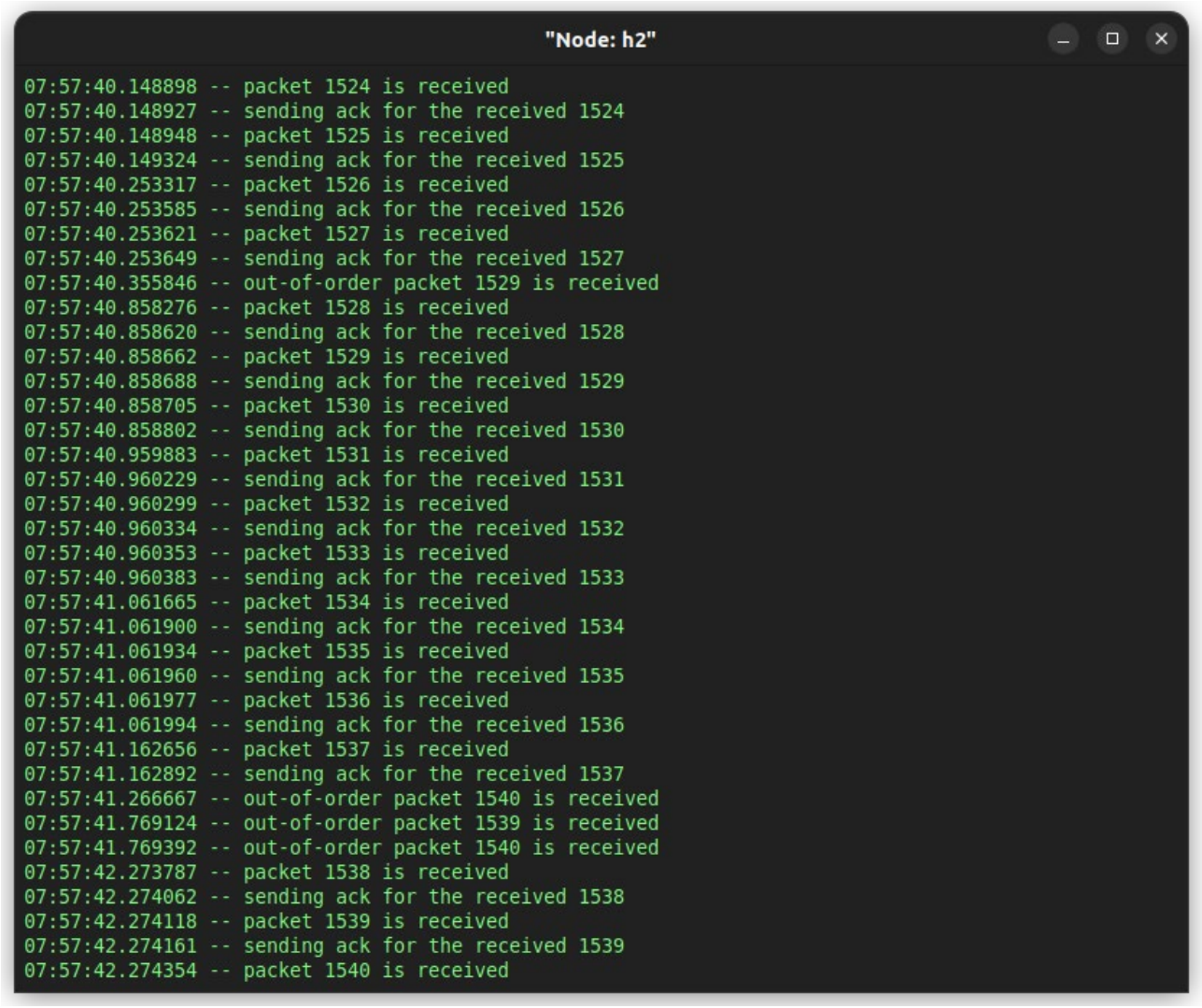
Discussion#4 Demonstrate Efficiency

2% Packet Loss Rate

Throughput: 168.09 Kbps

Explanation: With a 2% packet loss rate, the throughput is lower compared to scenarios without packet loss. This is because some packets are lost during transmission, requiring retransmission by the sender. The Go-Back-N protocol's mechanism for retransmission introduces additional delays and reduces the overall throughput.





5% Packet Loss Rate

Throughput: 128.54 Kbps

Explanation: With a higher packet loss rate of 5%, the throughput further decreases compared to the 2% packet loss scenario. The increased packet loss exacerbates the effects observed in the 2% packet loss scenario, leading to more frequent retransmissions and even lower throughput.

Observations

Impact of Packet Loss: The presence of packet loss significantly reduces throughput in both scenarios. This highlights the importance of robust error handling mechanisms, such as those employed in the Go-Back-N protocol, to mitigate the effects of packet loss on data transmission performance.

Effectiveness of Retransmission: Despite the presence of packet loss, the Go-Back-N protocol's retransmission mechanism allows the sender to recover from lost packets and ensure reliable data delivery. However, this comes at the cost of reduced throughput, particularly in scenarios with higher packet loss rates.

Discussion# Other Cases

Server is not running:

References

Geeksforgeeks. (2021, October 26). *TCP 3-Way handshake process*. GeeksforGeeks. https://www.geeksforgeeks.org/tcp-3-way-handshake-process

Geeksforgeeks. (2023, August 16). *Sliding window protocol | Set 2 (Receiver side)*. GeeksforGeeks. https://www.geeksforgeeks.org/sliding-window-protocol-set-2-receiver-side/

Rasmusjs. (n.d.). *Rasmusjs/DATA2410-1-23V-Datanettverk-og-skytjenester-Portfolio-2--Data-Reliable-Transport-Protocol: Datanettverk Og skytjenester portfolio 2*. GitHub. https://github.com/rasmusjs/DATA2410-1-23V-Datanettverk-og-skytjenester-Portfolio-2--Data-Reliable-Transport-Protocol and pdf report

Sciencedirect. (n.d.). *The TCP handshake*. https://www.sciencedirect.com/topics/computer-science/three-way-handshake