10.05.24 PORTFOLIO - 1

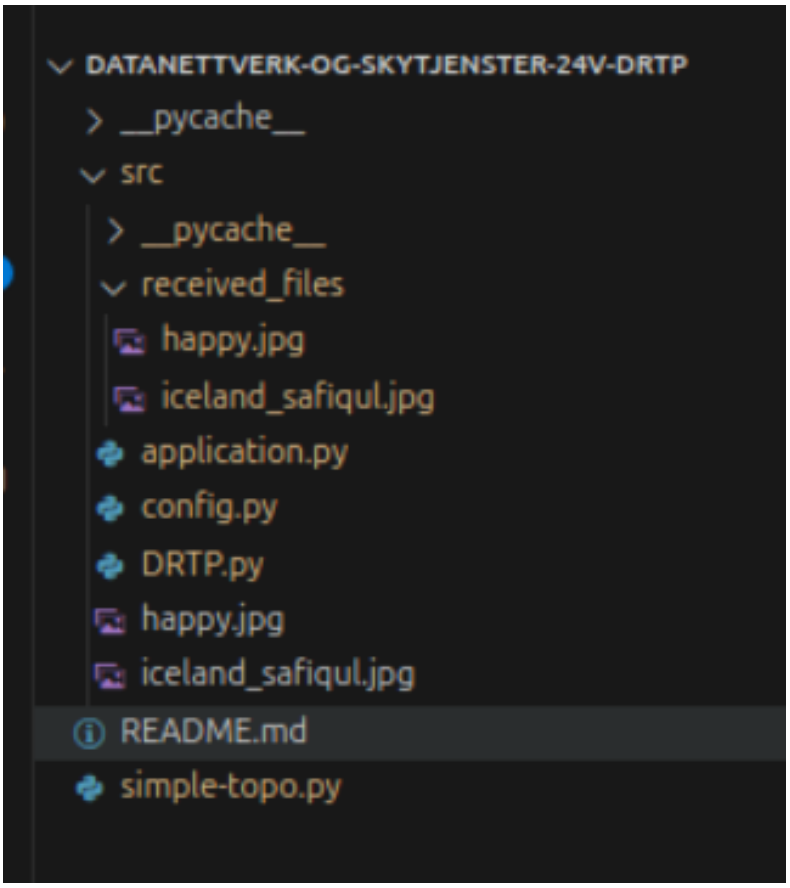
Canditat number: 167

DATA RELIABLE TRANSPORT PROTOCOL

1 Introduction

This document outlines the Reliable Transport Protocol (DRTP) project, designed to ensure reliable data transmission over the User Datagram Protocol (UDP), which traditionally lacks robustness in data integrity and order. DRTP represents a significant advancement in network protocol engineering by combining the efficiency of UDP with the reliability mechanisms typically associated with the Transmission Control Protocol (TCP). Intended for an academic and professional audience, this documentation delves into the theoretical motivations, architectural framework, and practical implementation of DRTP. Key sections include an analysis of the need for DRTP within the context of existing network protocols, an exploration of its structural design incorporating features like the three-way handshake and the Go Back N protocol, and a detailed overview of its technical implementation. Additionally, the document presents rigorous performance evaluations conducted in Mininet, simulating various network conditions to assess DRTP's performance. The aim is to provide readers with a comprehensive understanding of DRTP's operational mechanisms, developmental journey, and potential impact on enhancing data transmission reliability in network systems.

2 Architecture



The architecture of the Data Reliable Transport Protocol (DRTP) project is organized to ensure reliable data transfer over UDP by implementing a hybrid approach of TCP's reliability mechanisms. The project is structured into several key components within a "src" folder. The "config.py" file contains common definitions and constants such as header formats and protocol flags. The "application.py" file manages command-line argument parsing to run the application in either client or server mode, with validation functions to ensure correct input. The main logic

for the DRTP protocol is implemented in “DRTP.py”, where two primary classes, “DRTPClient” and “DRTPServer”, handle client-side and server-side operations respectively, including connection establishment, data transmission using a sliding window protocol, and connection termination. Supporting files include README.md for documentation, “simple-topo.py” for running the Mininet network simulator, and the “received_files” directory for storing transmitted files. Additionally, some images are included to aid in documentation or user guidance. The overall architecture is designed to be modular, facilitating clear separation of concerns and ease of understanding.

3 Background

Three-way handshake

A three way handshake is used to establish a reliable connection between host and server before transmitting data. The project adapts a TCP three-way handshake mechanism, tailored for use with UDP by adding reliability functions.

This is needed because UDP does not guarantee the delivery, order, or integrity of the packets sent between hosts. The handshake process is critical in setting up a session where both parties agree on various session parameters like sequence number and acknowledgement number, which are crucial for reliable communication.

Client:

```
def initiate_connection(self):
    """ Initiates a connection to the server using a three-way handshake.
    use: Sends a SYN packet and waits for a SYN-ACK packet to complete the handshake then sends
    """
    self.send_packet(0, 0, self.SYN)
    print("SYN packet sent")
    while True:
        header, _, addr = self.receive_packet()
        if header:
            seq_num, ack_num, flags = header
            if flags == (self.SYN | self.ACK):
                print("SYN-ACK packet is received")
                self.send_packet(0, ack_num, self.ACK, addr=addr)
                print("ACK packet sent")
                print("Connection established")
                break  # Exit loop once handshake is complete
```

A three-way handshake starts wish a sender sends a SYN packet. This packet is characterized by the SYN flag being set (value of 1). The packet contains number 1 and the acknowledgment (ack) flag in this initial packet remains unset (value of 0) since it is not yet applicable. Upon receipt of this SYN packet, the recipient issues a response in the form of a SYN/ACK packet. This packet adjusts the sequence number received in the SYN packet upward by one (e.g 1+1 in this example) and includes a new sequence number from the receiver itself.

Server:

```
        self.close()

    def handle_syn(self, seq_num, addr):
        """
        Handles the SYN packet to establish a connection.
        Args:
            seq_num (int): The sequence number of the SYN packet.
            addr (tuple): The address from which the SYN packet is received.
        use: Sends a SYN-ACK packet and update the connection state to SYN_RCVD.
        """
        print("SYN packet is received")
        self.send_packet(0, seq_num + 1, self.SYN | self.ACK, addr=addr)
        print("SYN-ACK packet is sent")
        self.connection_state = "SYN_RCVD"

    def handle_established(self):
        """
        Handles the ACK packet to establish a connection.
        Args: None
        use: Updates the connection state to ESTABLISHED.
        """
        print("ACK packet is received")
        print("Connection established")
        print() # just to have some space
        self.connection_state = "ESTABLISHED"
```

This acknowledgment process involves the receiver incrementing the initial sequence number from the sender and sending this incremented value back. When the sender receives the SYN/ACK packet, it must acknowledge the sequence number proposed by Host B. To do this, Host A sends back an ACK packet confirming the sequence number received from the reciever. This completion of the three-way handshake establishes a secure connection where sequence numbers are crucial for maintaining the order of packets. Thus, the connection is both stable and prepared to detect and manage any disorder in packet delivery.

Go Back N Protocol (GBN)

The Go-Back-N (GBN) protocol is an automatic repeat request (ARQ) protocol that ensures reliable packet delivery using a sliding window mechanism. This allows the sender to transmit multiple packets sequentially without waiting for individual acknowledgments, up to a predetermined window size, 'N'. The sender maintains a buffer of the last 'N' sent packets awaiting acknowledgment. The receiver acknowledges the last correctly received packet and ignores any out-of-order packets, prompting the sender to retransmit from the unacknowledged packet upon receiving a duplicate acknowledgment or timeout.

GBN is efficient in low-error environments due to continuous packet transmission but can lead to bandwidth wastage in high-error conditions due to retransmission of multiple packets for a single lost one. It does not handle out-of-order packets well, increasing latency as the receiver waits for the missing packet. High latency networks further reduce efficiency due to delays in lost packet detection and acknowledgment reception.

Closing the Connection - two way handshake

Client:

```python
def teardown_connection(self):
    """
    Tears down the connection by sending a FIN packet and handling the final ACK.
    use: Sends a FIN packet to close the connection and waits for the final ACK packet to close the connection.
    """
    print("Sending FIN packet")
    #send fin packet
    self.send_packet(self.next_seq_num, 0, self.FIN)
    self.next_seq_num += 1
    #receive ack packet
    header, _, addr = self.receive_packet()
    if header:
        seq_num, ack_num, flags = header
        if flags & self.ACK:
            print("Connection closed")
            self.socket.close()
```

Server:

```python
def handle_fin(self, seq_num, addr):
    """
    Handles the FIN packet to close the connection.
    Args:
        seq_num (int): The sequence number of the FIN packet.
        addr (tuple): The address from which the FIN packet is received.
    use: Sends a FIN-ACK packet and update the connection state to CLOSED
    """
    print("FIN packet is received")
    self.send_packet(seq_num + 1, 0, self.ACK, addr=addr)
    print("FIN-ACK packet is sent")
    self.connection_state = "CLOSED"
    print("Connection closed")
```

figure 3 illustrates the sequence involved in terminating a connection between two hosts. A(the sender) and B (the receiver). As shown in the figure, once all the data has been successfully transmitted from the sender to the receiver, the client initiates the closure process by sending a packet with the FIN flag set. Upon receiving this packet, the server acknowledges the request to terminate by sending an ACK back to the client. Following this acknowledgment, the server proceeds to close its end of the connection. The client, after receiving the ACK from the server, completes the disconnection process by shutting down its side of the connection. This orderly shutdown ensures that both client and server are prepared for subsequent connections without any lingering issues.

4 Implementation of DRTP / coding details

The Reliable Transport Protocol (DRTP) enhances data transfer reliability between clients and servers by ensuring all data is properly received and acknowledged. Using socket programming, DRTP establishes communication channels based on the unreliable UDP framework. The implementation in “application.py” includes argument parsing for user-specific customization via terminal parameters. The following sections detail key components and functionalities of our code.

application.py

```
41
42  def main():
43      parser = argparse.ArgumentParser(description="DRTP: Data Reliable Transfer Protocol based on UDP")
44      parser.add_argument("-c", "--client", action="store_true", help="Run in client mode")
45      parser.add_argument("-s", "--server", action="store_true", help="Run in server mode")
46      parser.add_argument("-i", "--ip", type=check_ip, help="IP address of the server", default="127.0.0.1")
47      parser.add_argument("-p", "--port", type=check_port, help="Port number", default=8080)
48      parser.add_argument("-f", "--file", type=check_file_exists, help="File to send (required in client mode)")
49      parser.add_argument("-w", "--window", type=check_window_size, help="Sliding window size", default=3)
50      parser.add_argument("-d", "--discard", type=check_discard, help="Sequence number of the packet to discard (server mode)", default=None)
51
52      args = parser.parse_args()
53
```

The parsing code defines command-line options to customize the application's operation in either client or server mode, and to specify network parameters such as IP addresses and port numbers.

The script outlines options including:

- -c or --client: This puts the application into client mode.
- -s or --server: This sets the application to server mode.
- -i or --ip: This parameter allows the user to specify the server's IP address, it is set by default to "127.0.0.1" which represents localhost.
- -p or --port: This sets the network port for the connection, with a default of 8088.
- -f or --file: This is required in client mode, it specifies the file to be transmitted.
- -w or --window: Defines the sliding window size with a default of 3, impacting how data is managed and acknowledged between the client and server. It can be set to 5 and 10 as well.
- -d or --discard: Available in server mode, it allows for testing packet loss by specifying a packet sequence number to intentionally discard.

This class also includes validation functions to ensure the integrity of user inputs, such as validity of IP addresses, port numbers, window sizes, discard parameters for simulating packet loss, and the existence of a file that is going to be transferred.

server logic

The class DRTPServer manages server operations for the Data Realiable Transport Protocol (DRTP), focusing on establishing and maintaining a connection with clients and handls data transfers ensuring proper termination. It sets up the server environment by binding to a specified IP and Port and simulate network unreablity.

The server starts off by listening to an incoming connection. Binds to the default port and IP address and awaits for a SYN packet from client. UDP does not ensure that a connection line is initiated before transmitting data, therefor the first step is to ensure that the connection is made between server and client via a three-way handshake, a TCP-like reliable connection over UDP.

```python
def process_buffered_packets(self, addr):
    """
    process_buffered_packets processes the buffered packets that are received out-of-order.
    Args:
        addr (tuple): The address to send the ACK packet.
    use: Sends an ACK packet for the buffered packets and updates the last acknowledged sequence number.
    """
    # Attempt to process any buffered packets that can now be accepted
    while self.last_acked_seq + 1 in self.buffered_packets:
        seq_num = self.last_acked_seq + 1
        data = self.buffered_packets.pop(seq_num)
        print(f"{datetime.datetime.now().strftime('%H:%M:%S.%f')} -- packet {seq_num} from buffer is now received")
        self.send_packet(0, seq_num, self.ACK, addr=addr)
        print(f"{datetime.datetime.now().strftime('%H:%M:%S.%f')} -- sending ack for the buffered received {seq_num}")
        self.last_acked_seq = seq_num
```

Once a connection has been successfully established, the server transitions into the data transmission phase, where it manages the flow of incoming data packets in a continuous loop. As each packet arrives, the server examines its sequence number to determine its position relative to previously received packets. If a packet arrives in sequence—that is, it is the next expected packet in the order—it is immediately acknowledged, and the data contained within it is processed accordingly. However, if a packet arrives out of sequence but is anticipated later (based on its sequence number), the server does not discard it; instead, it buffers the packet, holding it until the missing packets arrive and can be processed in the correct order. Additionally, if a packet is recognized as a duplicate—meaning it has already been received and acknowledged—the server sends an acknowledgment again. This repeated acknowledgment ensures that the client is aware that the packet was received previously, which is crucial for maintaining synchronization and data integrity between the server and client throughout the communication session.

In the server's operational sequence, a packet may be intentionally discarded, as specified by `discard_seq`, to evaluate the protocol's error handling abilities. Following data transmission, the server handles the connection's closure by expecting a FIN packet from the client, responding with a FIN-ACK, and transitioning to the "CLOSED" state, signaling the end of the session.

client logic

The “DRTPClient” class extends the functionality of the “DRTPBase” to specifically handle client-side operations for the Data Reliable Transport Protocol (DRTP), focusing on initiating connections, transmitting data, and properly closing the connection. This class is structured to manage data transfer using a sliding window protocol, which enables efficient packet management over UDP. Thos class starts off by setting up the client socket (UDP socket) by binding it to the IP address and port number. This is similar to the server.

This class also ensures that the client-side operations of the DRTP are robust, incorporating error handling and flow control mechanisms to enhance reliability over the inherently unreliable UDP. The structured approach, from connection setup through data transfer to connection termination, mirrors the logical flow outlined in the server logic, ensuring that both ends of the communication channel operate synchronously and efficiently.

```python
def send_data(self):
    """ Sends data using a sliding window protocol.
    use: Reads data from the file and sends it in packets. Manages the sliding window and retransmissions.
    """
    with open(self.filename, 'rb') as file:
    # First send the filename
        filename_packet = f"FILENAME:{os.path.basename(self.filename)}".encode('utf-8')
        self.send_packet(self.next_seq_num, 0, 0, filename_packet)
        self.window.append(self.next_seq_num)
        print(f"{datetime.datetime.now().strftime('%H:%M:%S.%f')} -- Packet with seq = {self.next_seq_num} is sent, sliding window = {self.window}")

        self.send_buffer[self.next_seq_num] = filename_packet
        self.next_seq_num += 1

        # Then send the file data
        data = file.read(self.PACKET_SIZE)
        while data or self.window:
            # Send new packets if the window is not full,
            while len(self.window) < self.window_size and data:
                self.send_packet(self.next_seq_num, 0, 0, data)
                self.window.append(self.next_seq_num)
                print(f"{datetime.datetime.now().strftime('%H:%M:%S.%f')} -- Packet with seq = {self.next_seq_num} is sent, sliding window = {self.window}")
                self.send_buffer[self.next_seq_num] = data
                self.next_seq_num += 1
                data = file.read(self.DATA_SIZE)

            self.handle_acknowledgments()
```

After establishing the connection, the client begins the data transmission phase. Using a sliding window protocol, it sends packets sequentially up to the window limit, which regulates how many packets can be sent before requiring an acknowledgment. This mechanism ensures continuous data flow and efficient bandwidth utilization. The client reads data from the specified file and sends it in segments. If the window is full, it waits for acknowledgments before sending more data, thereby adapting the transmission rate to network conditions.

```python
            self.teardown_connection()


def retransmit_unacknowledged_packets(self):
    """ Retransmits the unacknowledged packets in the window.
    use: Retransmits the packets in the window that have not been acknowledged by the server.
    """
    for seq in self.window:
        data = self.send_buffer[seq]
        self.send_packet(seq, 0, 0, data)
        print(f"{datetime.datetime.now().strftime('%H:%M:%S.%f')} -- Retransmitting packet with seq = {seq}")
        continue
```

Throughout the data transfer, the client continuously listens for acknowledgments from the server. Each acknowledgment received allows the client to update its window, removing acknowledged packets and making room for new ones to be sent. This process helps in managing the flow control and ensures that all data is received and acknowledged correctly. If an acknowledgment is not received within a certain time frame, indicating a possible packet loss, the client retransmits the unacknowledged packets, ensuring no data is lost.

5 Discussion

50ms RTT

Delay (RTT)	Window Size	Throughput (Mbps)
	3	0.47

50 ms	5	0.78
	10	1.55

with smaller a RTT of 50ms, acknowledgement return quickley, allowing the sender to keep the pipeline full. Consequently, throughput is higher. For example, with a window size of 10, the throughput is 1.55 Mbps, which is significantly higher compared to smaller window size. This indicates that lower RTT values enable more efficient data transmission.

100ms RTT

Delay (RTT)	Window Size	Throughput (Mbps)
100 ms	3	0.24
	5	0.39
	10	0.79

When the RTT doubles to 100ms, the time it takes for acknowledgments to return also doubles, which reduces throughout. The throughput at a window size of 10 is 0.79 Mbps, which is about halp of the throughput at the same window size with 50ms RTT. This demonstrates the inverse relationship between RTT and throughput: as RTT increases, throughput decreases due to longer acknowledgment times.

200ms RTT

Delay (RTT)	Window Size	Throughput (Mbps)
200 ms	3	0.12
	5	0.20
	10	0.40

At an RTT of 200ms, the effect of delay is even more pronounced. The throughput drops further, with a window size of 10 having a throughput of 0.40 Mbps. This shows the significant impact high RTT values have on data transfer efficiency. Longer RTTs lead to slower acknowledgment returns, thus reducing the throughput.

How Window Size Impacted Throughput:

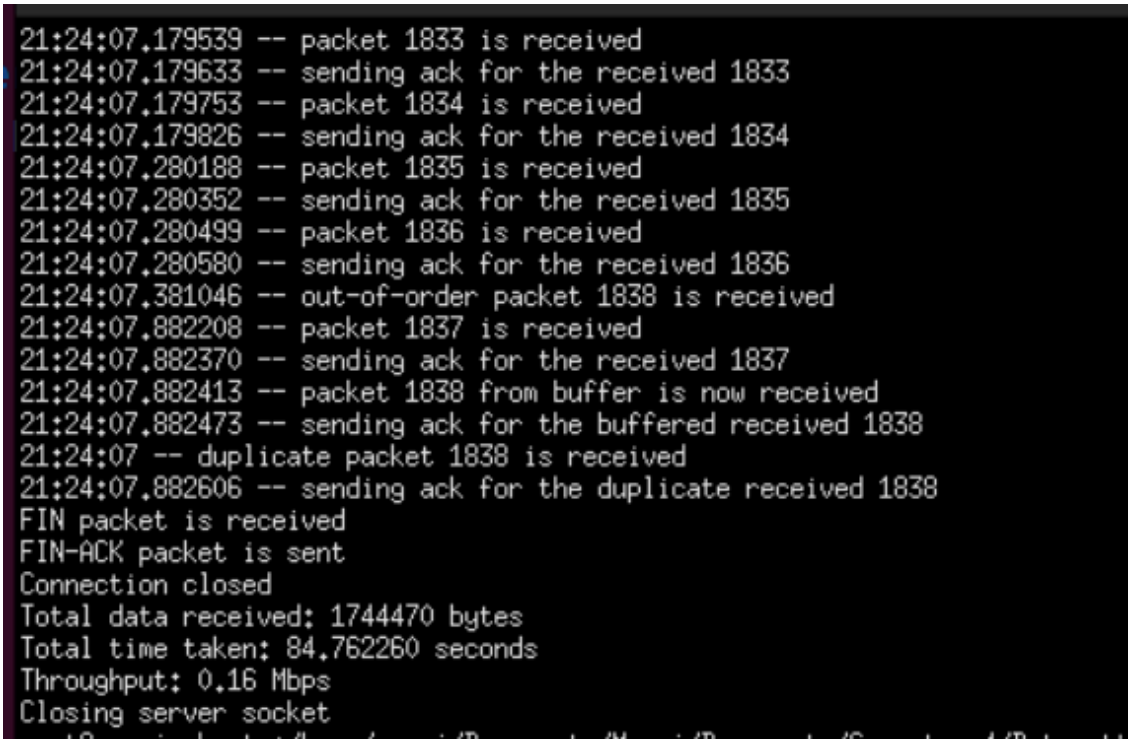
One thing to observe here is that the window size increases for a given delay. This can be expected because a larger window size allows more packets to be in transit before requiring an acknowledgement, thereby increasing the amount of data transferred over time. For example, at 50ms RTT, increasing the window size from 3 to 10 more than triples the throughput.

How RTT Impacted Throughput:

Another think to observe here is that throughput decreases as the delay increases for a given window size. This can be because a longer RTT means it takes more time for acknowledgment to

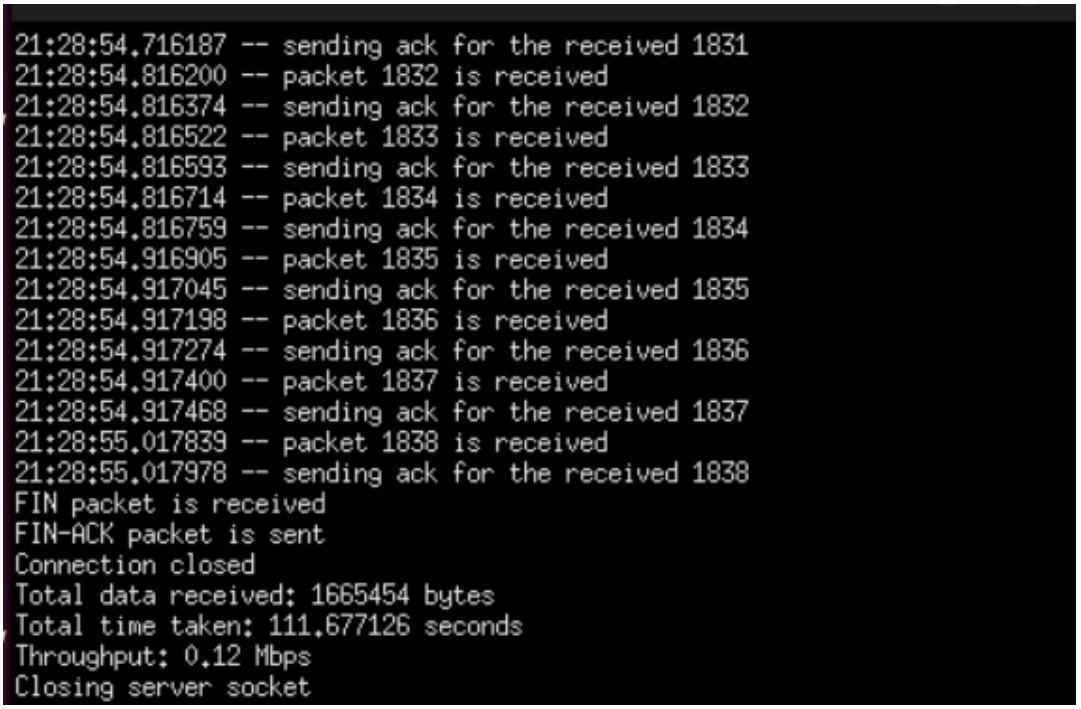
be received, thus reducing the efficiency of data transfer within the same time period. The effect is clearly observed across different RTTs, where higher delays consistently results in lower throughputs.

Simulation of packet loss with 2%:



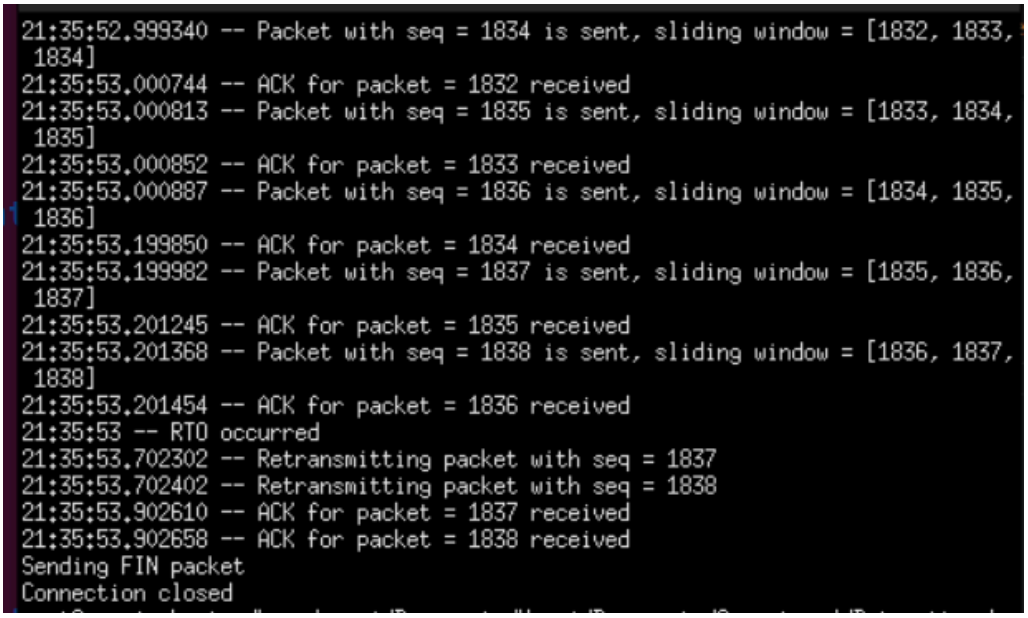
The Mininet test with a 100 ms delay and 2% packet loss demonstrates the reliability of the DRTP protocol. The logs show that the server correctly handles in-sequence, out-of-order, and duplicate packets, ensuring data integrity with proper acknowledgments. The network conditions significantly impact performance, resulting in a low throughput of 0.16 Mbps due to increased retransmissions and latency. Despite these challenges, DRTP effectively manages data transmission, confirming its robustness in adverse network environments.

Simulation of packet loss with 5%:



Testing DRTP with a 100 ms delay and 5% packet loss in Mininet shows a significant performance impact. The server successfully handles packets and maintains data integrity with proper acknowledgments. However, the increased packet loss results in more retransmissions, leading to a longer total transmission time of 111.677126 seconds and a lower throughput of 0.12 Mbps. Despite the challenging conditions, DRTP remains robust, though higher packet loss significantly affects network performance.

Test result for discarded package 1837:



The test with packet 1837 intentionally discarded shows that the DRTP protocol effectively handles packet loss and out-of-order packets. The server buffers the out-of-order packet 1838 and processes it once packet 1837 is retransmitted and received. Despite the discard, the protocol maintains data integrity through retransmissions, though this increases the total transmission time to 123.578091 seconds and maintains a throughput of 0.12 Mbps. This demonstrates DRTP's robustness in managing packet loss and ensuring reliable data transmission.

6 Conclusions

In conclusion the DRTP project represents a step, in network protocol development by boosting data transmission reliability over the typically unreliable UDP. By incorporating elements from TCP like the three way handshake and Go Back N protocol DRTP effectively ensures data integrity and sequence while preserving UDPs efficiency. This detailed documentation thoroughly explores DRTPs foundations, architectural design, practical implementation. Includes thorough performance evaluations using Mininet.

The modular structure of DRTP featuring defined components and functions enables separation of responsibilities and facilitates comprehension. Tests conducted under network conditions reveal that DRTP maintains resilience and dependability. Lower RTTs and larger window sizes enhance throughput while higher RTTs and increased loss significantly impact performance due to retransmissions and inherent latency.

Despite these challenges, DRTP consistently ensures data integrity and synchronization by effectively managing issues like packet loss, out-of-order packets, and retransmissions. The protocol's robustness is further evidenced by its ability to handle intentional packet discards, guaranteeing dependable data transmission even in challenging situations.

The protocol's ability to handle intentional packet discard further underscores its reliability and robustness. Overall, DRTP represents a practical and effective solution for enhancing UDP's reliability, making it a valuable addition to network protocol engineering. This documentation provides a comprehensive understanding of DRTP’s operational mechanisms, developmental journey, and potential impact on improving data transmission reliability in network systems.

7 References

assignment-guidelines.pdf

howtonetwork. (2023, October 28). *TCP and the Three-Way Handshake Explained [Follow-Along Lab]* 🚀 [Video]. YouTube. https://www.youtube.com/watch?v=wMc0H22nyA4

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

Neso Academy. (2020, April 9). *Go-Back-N ARQ* [Video]. YouTube. https://www.youtube.com/watch?v=QD3oCelHJ20