Reliable Transport Protocol(DTRP)

Kandidatnummer: s374220
Emnekode: DATA2410
Emnenavn: Datanettverk og skytjenester
Studieprogram: Bachelor i Dataingeniørfag
Antall ord: 2537
Innleveringsfrist: 21.05.2024

OSLO METROPOLITAN UNIVERSITY
STORBYUNIVERSITETET

Introduction:

In developing my Reliable Transfer Protocol (DRTP) file transfer application, I used our networking PowerPoints for guidance and tested the code with Mininet, a popular network emulator. This approach helped me make sure the design was both theoretically sound and practical. By combining what I learned from the textbook with real-world testing, I aimed to create a protocol that is simple, reliable, and efficient. Let's look at my main design choices and see how they contribute to the reliability of the application.

My implementation:

There are several ways to solve the task given by the professor. Firstly, we need to look at the requirements in Github. What we need is:

- ONE file that will be called using specific arguments (From both client- and server-side)
- The application must implement UDP sockets and must implement a sliding window
- The client will send a file from the computer, where the server will write the file to its filesystem
- The client and server must be on the same Port and IP address for file transfers to take place
- The client will read data in chunks of 994 bytes, and both will send headers that are 6 bytes in size
- The headers used will include either SYN, ACK, or FIN flags. The reset flag won't be used
- The protocol must implement a three-way handshake and connection teardown.
- The protocol will also implement the Go-Back-N function for reliability and a 500 MS timeout.

I decided to implement two classes. One for the server, and the other for the client. This would make the code more modular and more readable. It would also remove a lot of the boiler code. When first starting to write the code, I wrote constants.

```
HEADER_SIZE = 6 # bytes
DATA_SIZE = 994 # bytes
TIMEOUT = 0.5  # in seconds
SYN_FLAG = 0b1000
ACK_FLAG = 0b0100
FIN_FLAG = 0b0010
```

The constants will be used a lot inside both classes; therefore, it will make the code more modular to have them be global. Afterward, I started writing the logic for how the application should parse arguments. I chose to make the code that parses arguments into 3 separate functions. One function parses the arguments and gives an object. Then I chose to send the object to one of two functions, based on whether the object is for the client or the server. The second function is where the object gets validated. That is where we check for errors in the parsed object, and if there are the connection gets closed. Here is the function that checks the server arguments:

```
def check_server(args):
    """Check the server arguments."""
    if args.file:
        print("-s option cannot accept -f argument.")
        sys.exit(1)
```

After finishing parsing the arguments, I started work on my Server class. I started building the constructor. I chose to use a constructor because it provides a central place to initialize all the attributes of the server. It also helps when calling methods, so that I don't have to repeatedly pass the same parameters to different methods. I also initialized the UDP socket from the constructor because it ensures that the socket is ready

for use as soon as an instance of the server is created. The server is supposed to wait for a SYN Packet indefinitely, so I set the timeout to None.

```
    def __init__(self, host, port, discard):
        self.host = host
        self.port = port
        self.sock = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
        self.sock.bind((self.host, self.port))
        self.discard = discard
        self.sock.settimeout(None)
```

I also built two methods for unpacking and packing headers. It takes in 3 arguments the seq_num, ack_num, and flags. It uses struct.pack to convert these values into binary, so that it complies with the requirements from the task. The unpack header method uses struct.unpack to get data from the packet.

```
def pack_header(seq_num, ack_num, flags):
    return struct.pack('!HHH', seq_num, ack_num, flags)
def unpack_header(header):
    return struct.unpack('!HHH', header)
```

After initializing the socket and binding it to the specified IP address and port, the *start_server* method is called from the constructor. This method is responsible for sending you to a function, where you will be waiting for the client to establish a connection. It enters a loop where it continuously listens for incoming SYN packets.

```
        try:
            packet, sender_address = self.sock.recvfrom(HEADER_SIZE)
            _, _, flags = unpack_header(packet)
            if flags & SYN_FLAG:
                print("SYN packet received")
                self.send_syn_ack(sender_address)
```

Upon receiving a SYN packet, the server extracts the flags from the packet header. If the received packet contains the SYN flag, indicating a connection request from the client. The server then sends you to the method *send_syn_ack*, where the server sends a SYN-ACK packet back to the client to acknowledge the request and signal its readiness to establish a connection.

```
try:
            flags = SYN_FLAG | ACK_FLAG
            syn_ack_packet = pack_header(0, 1, flags)
            self.sock.settimeout(TIMEOUT)
            self.sock.sendto(syn_ack_packet, sender_address)
            print("SYN-ACK packet is sent")
            self.receive_ack()
```

On the client side:

For the client, I will be building some similar methods as in the server class. I started on the constructor.

```
    def __init__(self, host, port, file_path, window_size):
        self.file_path = file_path
        self.window_start = 0  # Start of the sliding window
        self.window_size = window_size  # Size of the sliding window
        self.window = set()  # Initialize an empty set to store sent packets
        self.sock = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
        self.sock.settimeout(TIMEOUT)  # Set socket timeout
```

I have added attributes for the sliding window (will delve deeper into it later), and I have added an attribute for the file path. Like the server class, the client class initializes the socket and establishes a connection with the server upon instantiation. The send_syn_packet method sends a SYN packet to the server to initiate the connection request.

```
        try:
            self.sock.settimeout(TIMEOUT)
            received_header, sender_address = self.sock.recvfrom(HEADER_SIZE)
            _, received_ack_num, received_flags =
unpack_header(received_header)
            if received_flags & SYN_FLAG and received_flags & ACK_FLAG:
                print("SYN-ACK packet is received")
                self.ack_num = received_ack_num
                self.send_ack(sender_address)
```

After sending the SYN packet, the client enters a loop where it waits to receive a SYN-ACK packet from the server. Upon receiving the SYN-ACK packet, the client verifies that the received packet contains both the SYN and ACK flags, indicating a successful response from the server. If the conditions are met, the client knows that the connection has been established, and it proceeds to the next step of sending the ACK to the server.

```
            header = pack_header(self.seq_num, self.ack_num, ACK_FLAG)
            self.sock.sendto(header, sender_address)
            self.send_data_packets()
```

Upon receiving an ACK packet, the server verifies the acknowledgment number and validates the packet's integrity to ensure that it corresponds to the SYN-ACK packet previously sent. If the ACK packet is valid and contains the expected acknowledgment number, the server considers the connection successfully established. Now begins the data transfer:

Data Transfer (Client-Side):

```
        with open(self.file_path, 'rb') as file:
            while True:
                while len(self.window) < self.window_size:
                    data = file.read(DATA_SIZE)
                    header = pack_header(self.seq_num, self.ack_num, 0)
                    packet = header + data
                    self.sock.send(packet)
                    self.window.add(self.seq_num)
                    print(f"{time.strftime('%H:%M:%S')} -- Data packet with
                    self.seq_num += 1
                try:
                    self.receive_ack()
                if len(data) < DATA_SIZE:
                    break
        for _ in range(len(self.window)):
            self.receive_ack()
        self.send_fin_packet()
```

The *send_data_packets* method is at the heart of transferring data from the client to the server. It starts by opening the specified file in binary mode and enters a loop where it reads portions of the file to send to the server. Each time it sends a packet, it checks if there's space available in the sliding window. If so, it constructs a packet containing the data along with necessary header information like sequence numbers and acknowledgment flags. This packet is then sent to the server using the UDP socket. Throughout this process, the method ensures that the sliding window mechanism remains intact, allowing for efficient data transfer.

Once all data packets have been sent, the method shifts into a cleanup phase. Here, it patiently waits to receive acknowledgments (ACKs) from the server for each packet sent. It keeps checking for ACKs until all packets are acknowledged, signifying the successful transfer of data. After ensuring that all data has been reliably transmitted, the method initiates the connection teardown process by sending a FIN packet to the server. This FIN packet serves as a signal to gracefully close the connection, marking the end of the file transfer operation.

```
            if received_flags & ACK_FLAG:
                print(f"{time.strftime('%H:%M:%S')} -- ACK for packet =
                # Check if the acknowledgment is a duplicate
                if received_ack_num == self.last_received_ack:
                    return  # Ignore duplicate acknowledgment
                self.last_received_ack = received_ack_num
                if received_ack_num in self.window:
                    self.window.remove(received_ack_num)
                # Slide window if possible
                while self.window_start in self.window:
                    self.window.remove(self.window_start)
                    self.window_start += 1
```

The *receive_ack* method plays a critical role in our DRTP protocol by managing acknowledgments from the server. After sending data packets, we need confirmation from the server that they arrived safely. This method keeps an eye out for these acknowledgments.

When an ACK packet arrives, the method checks if it's a genuine acknowledgment by looking for the ACK flag in the header. If it is, it checks whether it's a duplicate or a new acknowledgment. Duplicates are ignored to avoid confusion. If it's new, the method updates its record and adjusts the sliding window if necessary to stay in sync with the server.

```
            for seq_num in self.window:
                with open(self.file_path, 'rb') as file:
                    print("RTO occured")
                    file.seek(seq_num * DATA_SIZE)
                    data = file.read(DATA_SIZE)
                    header = pack_header(seq_num, self.ack_num, 0)
                    packet = header + data
                    self.sock.send(packet)
                    print(f"{time.strftime('%H:%M:%S')} -- Retransmitted data
```

To handle delays or packet loss, we set a timeout. If no ACK arrives within this time, the method assumes something went wrong and triggers a resend of the unacknowledged packets from the set. The protocol seeks out the specific chunk of the file that needs to be resent by checking for it using the sliding window. This ensures that our file transfer stays intact and reliable. Also, the method handles any errors that might occur during the acknowledgment process, making sure our protocol remains robust. After the *receive_ack* is successful, it sends you back to the *send_data_packets* method. It is possible to combine the two methods, however, it would be at the behest of the readability and modularity of the code. That is why I separated it into two methods. Now let us look at the server side during the data transfer.

Data Transfer (Server-Side):

```
        try:
            expected_seq_num =0
            discard_seq_num = None  # Initialize discard sequence number
            if self.discard:
                discard_seq_num = self.discard
            with open("received_image.jpg", 'wb') as file:
                start_time = time.time()
                while True:
                    # Receive the packet
                    received_packet, sender_address =
self.sock.recvfrom(HEADER_SIZE + DATA_SIZE)
                    self.sock.settimeout(TIMEOUT)
                    header = received_packet[:HEADER_SIZE]
                    data = received_packet[HEADER_SIZE:]
```

The receive_data_packets function handles the reception of data packets from the client during the file transfer process. Once the connection is established, this method continuously listens for incoming packets. It starts by setting up variables like the expected sequence number and the sequence number to discard, if any, based on the user's choice. Then, it enters a loop where it receives packets from the client.

```
                    if received_flags & FIN_FLAG:
                        break  # Exit the loop if the FIN flag is set
                    if seq_num == discard_seq_num:
                        print(f"{time.strftime('%H:%M:%S')} -- Discarding pac
                        discard_seq_num=None
                        self.sock.settimeout(None)
                        continue
                    if seq_num == expected_seq_num:
                        print(f"{time.strftime('%H:%M:%S')} -- Packet
                        file.write(data)
                        # Send an acknowledgment for the received packet
                        ack_header = pack_header(0, seq_num, ACK_FLAG)
                        self.sock.sendto(ack_header, sender_address)
                        expected_seq_num +=1
                    else:
                        self.sock.settimeout(None)
                        print("Discarded")
                        continue # Skip processing this packet and wait for
                    if len(data) < DATA_SIZE:
                        break  # Exit the loop if the last packet is received
```

Upon receiving a packet, the method extracts the header and data from it. It checks if the packet contains the FIN flag, which indicates the end of the file transfer. If it does, the method breaks out of the loop. Otherwise, it proceeds to process the received packet. If the sequence number matches the expected sequence number, it writes the data to the file, sends an acknowledgment back to the client, and updates the expected sequence number. If the sequence number is out of order, indicating a potential packet loss or reordering, it skips processing the packet and waits for the client to resend it. Finally, when all packets are received and processed, it calculates the throughput of the file transfer and prints it out. If any errors occur during the process, they are caught and handled gracefully. This method ensures that data is reliably received from the client and written to the file while maintaining the integrity of the file transfer process. This brings us to how the connection gets torn down.

Connection-Teardown:

```
    def send_fin_packet(self):
        try:
            header = pack_header(self.seq_num, self.ack_num, FIN_FLAG)
            self.sock.send(header)
            print("FIN packet is sent")
            self.receive_fin_ack()

    def receive_fin_packet(self, sender_address):
          try:

            packet, sender_address = self.sock.recvfrom(HEADER_SIZE)
            _, _, flags = unpack_header(packet)
            if flags & FIN_FLAG:
                self.send_fin_ack(sender_address)
```

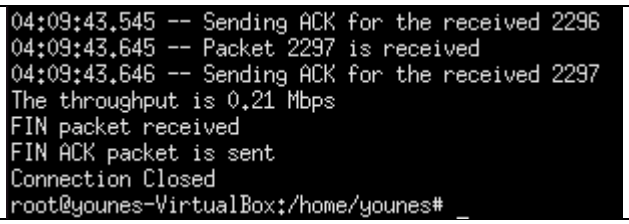
In the process of closing a connection in the DRTP protocol, both the server and client need to cooperate smoothly to wrap up their communication. On the server side, there's a function called receive_fin_packet that waits for a special signal, known as the FIN packet, from the client. This FIN packet tells the server that the client wants to end the connection. Once the server receives this signal, it acknowledges it by sending back a FIN-ACK packet using the send_fin_ack method. This action confirms to the client that the server has received the termination request and is ready to close the connection. To ensure everything goes according to plan, the server is equipped with robust error-handling capabilities, which help manage potential issues like timeouts or unexpected errors, ensuring a reliable connection closure process.

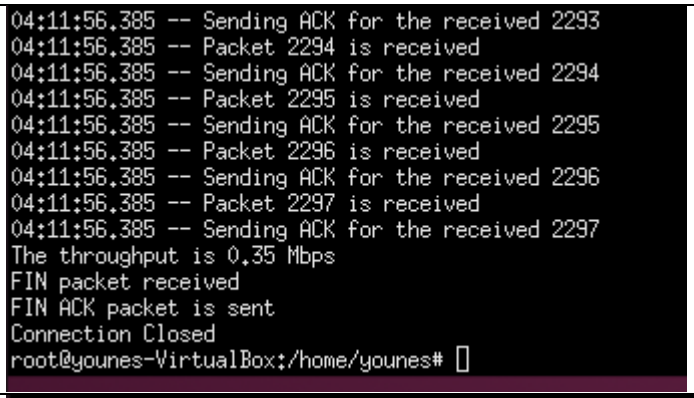
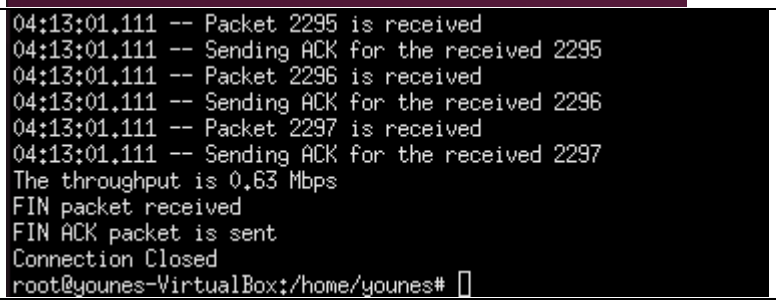
```
    def send_fin_ack(self, sender_address):
        try:
            header = pack_header(0, 0, ACK_FLAG|FIN_FLAG)
            self.sock.sendto(header, sender_address)
            print("FIN ACK packet is sent")
            self.close_connection()
    def receive_fin_ack(self):
        try:
            received_header = self.sock.recv(HEADER_SIZE)
            _, received_ack_num, received_flags =
unpack_header(received_header)
            if received_flags & ACK_FLAG and received_flags & FIN_FLAG:
                print("FIN-ACK packet is received")
                self.close_connection()
```

Meanwhile, on the client side, the teardown process begins with the send_fin_packet method, where the client sends a FIN packet to the server to indicate its desire to end the connection. After dispatching this packet, the client patiently waits for a response from the server. When the client receives a FIN-ACK packet from the server, it's a sign that the server has acknowledged the request to terminate the connection. With this confirmation in hand, the client proceeds to shut down the connection using the receive_fin_ack method. Just like its server counterpart, the client is well-prepared to handle any hiccups during this process, such as timeouts or errors, ensuring a smooth and orderly conclusion to the communication session. This is the end of the implementation of the protocol.

Discussion:

1)

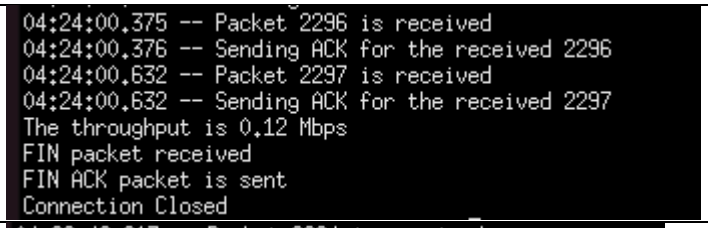
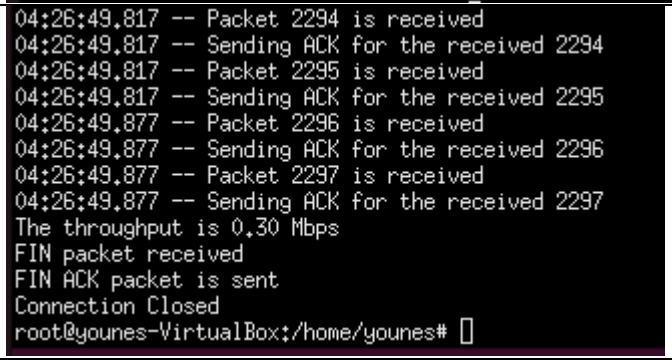
3 - windowsize		

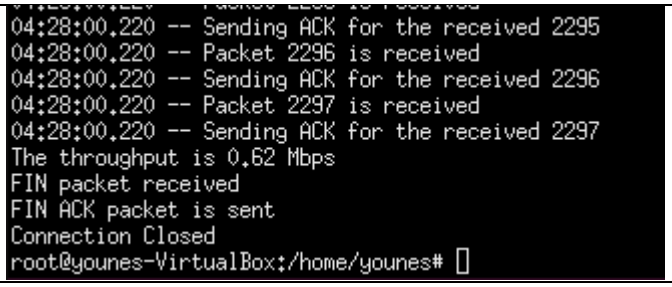
5 - windowsize	
10-windowsize	

In Task 1, I observed how adjusting the window size impacted throughput in UDP. For instance, with a window size of 3, the throughput was measured at 0.21 Mbps, while it increased to 0.35 Mbps for a window size of 5 and further to 0.63 Mbps for a window size of 10. This trend illustrates the sliding window protocol's mechanics, where a larger window size allows me to send more packets before awaiting acknowledgment from the receiver. Essentially, it's like having a bigger pipeline for data flow, enabling more efficient transmission and utilization of available bandwidth.

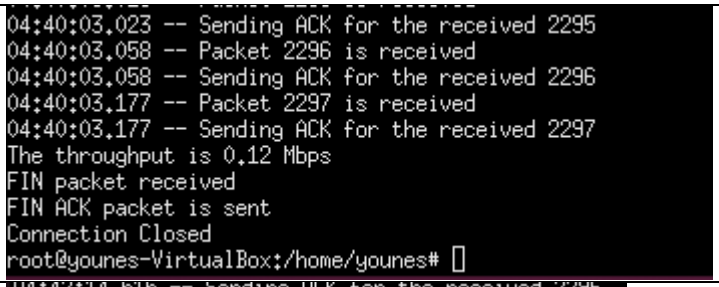
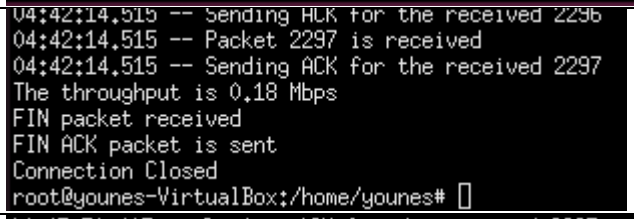
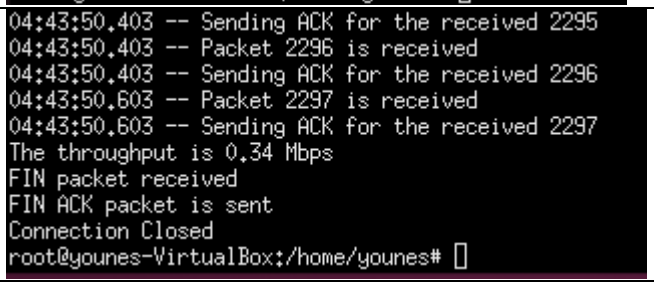
2)

50ms:

3-windowsize	
5-windowsize	

10-windowsize	

200ms:

3-windowsize	
5-windowsize	
10-windowsize	

Comparing Task 2 to Task 1, where the default RTT was 100 ms, adjustments to the RTT produced interesting results. When the RTT was decreased to 50 ms, the throughput for larger window sizes (5 and 10) showed improvements, with throughput values of 0.30 Mbps and 0.62 Mbps. However, for the smallest window size (3), the throughput decreased to 0.12 Mbps. This divergence from what was expected suggests other factors may have had an effect. Despite the faster acknowledgment times with a lower RTT, congestion could arise if the sender floods the network with too many packets, leading to retransmissions or packet loss.

On the flip side, when the RTT was increased to 200 milliseconds, speeds went down no matter how many packets were sent at once. Even when sending more data in one go, the speeds dropped. This is expected behavior. For example, speeds for sending 3, 5, and 10 at a time were 0.12 Mbps, 0.18 Mbps, and 0.34 Mbps. This shows that longer delays slow down how fast data moves in UDP.

3)

Server-side	04:47:30.202 -- Sending ACK for the received 2291 04:47:30.202 -- Discarding packet 2292 04:47:30.202 -- OUT OF ORDER Packet 2293 is received Discarded 04:47:30.402 -- OUT OF ORDER Packet 2294 is received Discarded 04:47:30.903 -- Packet 2292 is received 04:47:30.903 -- Sending ACK for the received 2292 04:47:30.904 -- Packet 2293 is received 04:47:30.904 -- Sending ACK for the received 2293
Client-Side	04:47:30.202 -- ACK for packet = 2291 is received 04:47:30.202 -- Data packet with seq = 2294 is sent, sliding window = {2292, 2293, 2294} 04:47:30.702 -- Socket timeout occurred while waiting for ACK packet. Retransmitting unacknowledged packets. RTO occured 04:47:30.703 -- Retransmitted data packet with seq = 2292 RTO occured 04:47:30.703 -- Retransmitted data packet with seq = 2293 RTO occured 04:47:30.703 -- Retransmitted data packet with seq = 2294 04:47:30.904 -- ACK for packet = 2292 is received 04:47:30.904 -- Data packet with seq = 2295 is sent, sliding window = {2293, 2294, 2295} 04:47:30.904 -- ACK for packet = 2293 is received 04:47:30.904 -- Data packet with seq = 2296 is sent, sliding window = {2294, 2295, 2296} 04:47:30.904 -- ACK for packet = 2294 is received 04:47:30.904 -- Data packet with seq = 2297 is sent, sliding window = {2295, 2296, 2297}

For this task, I chose to discard packet 2292. When I receive an acknowledgment (ACK) packet, I check if it acknowledges a packet within my window. If it does, the code considers the packet successfully transmitted, and I update my window accordingly. This approach ensures that I only send packets within the range given by the client and avoid overwhelming the server.

However, if I don't receive an acknowledgment within a certain time frame, I take measures to retransmit any unacknowledged packets. I iterate through my window, identifying which packets haven't been acknowledged, and resend them to the. This makes sure that any missing or lost packets are resent, keeping the reliability of the protocol.

Additionally, I handle duplicate ACKs gracefully by ignoring them to prevent unnecessary retransmissions. By keeping track of the last acknowledgment number I received, I can identify duplicate acknowledgments and avoid redundancy. This behavior helps me adjust the use of network resources and makes sure that I only retransmit when necessary.

4)

2% loss(5-window)	04:53:51.668 -- Packet 2296 is received 04:53:51.668 -- Sending ACK for the received 2296 04:53:51.668 -- Packet 2297 is received 04:53:51.668 -- Sending ACK for the received 2297 The throughput is 0.22 Mbps FIN packet received FIN ACK packet is sent Connection Closed root@younes-VirtualBox:/home/younes#
5% loss(5-window)	04:59:54.357 -- Packet 2296 is received 04:59:54.357 -- Sending ACK for the received 2296 04:59:54.958 -- Packet 2297 is received 04:59:54.958 -- Sending ACK for the received 2297 The throughput is 0.16 Mbps FIN packet received FIN ACK packet is sent Connection Closed

Task 4 highlighted how packet loss can severely impact UDP throughput. For example, with a 2% packet loss, throughput decreased to 0.22 Mbps, and with a 5% loss, it dropped further to 0.16 Mbps. Unlike TCP,

which employs mechanisms like selective acknowledgment and congestion control to recover from packet loss, UDP lacks this capability. Therefore, packet loss leads to retransmissions, delays, and reduced throughput. It's like sending letters through the mail; if some letters are lost along the way, I must resend them, leading to delays and slower overall delivery.

Conclusion:

In making my Reliable Transfer Protocol (DRTP) file transfer protocol, I mixed what I learned in my networking classes with tests using Mininet. This mix helped me make sure my design worked both on paper and in real life. I split the code into parts for the client and server to keep the code modular and reusable. I used constants to make sure the code was easy to understand. The important bits included using UDP sockets and a sliding window for reliability. I tested different things like window size and round-trip time to see how they affected how fast files could transfer and how reliable the transfer was.

Overall, the DRTP protocol I made is a simple, reliable, and efficient way to transfer files over UDP.