Home exam DATA2410 report

Simple transport protocol UDP/DRTP

Candidate 204

Introduction

The objective of the DATA2410 Reliable Transport Protocol (DRTP) project is to come up with a reliable data transfer application over UDP— which despite being efficient and fast, does not possess reliability mechanisms like TCP. In light of this, I seek to forge a custom protocol that upholds reliability as well as in-order delivery between two nodal points on a network; all in an attempt to make up for what UDP lacks. The project itself entails crafting a file transfer application that can act in either: client or server. From the client end, a file is read from the local system and sent over using DRTP via UDP; while on the flip side at the server end, upon reception, the file is written to its file system ensuring integrity and order throughout. Key features of the DRTP protocol include:

- Connection Establishment and Termination: Using a three-way handshake mechanism similar to TCP to establish a reliable connection and gracefully terminate it.
- Packet Structure: Implementing a custom packet structure that includes a 6-byte header for sequence numbers, acknowledgment numbers, and flags, ensuring effective communication and control.
- Go-Back-N (GBN) Reliability: Utilizing the GBN protocol with a fixed window size to manage packet transmission and ensure reliable delivery. The sender retransmits packets if acknowledgments are not received within a specified timeout period. In our case its set to 500ms but can be easily changed.
- Throughput Measurement: The server calculates and displays the throughput based on the amount of data received and the elapsed time, providing insights into the efficiency of the transfer process.

The application supports various configurations to test and demonstrate its reliability and performance:

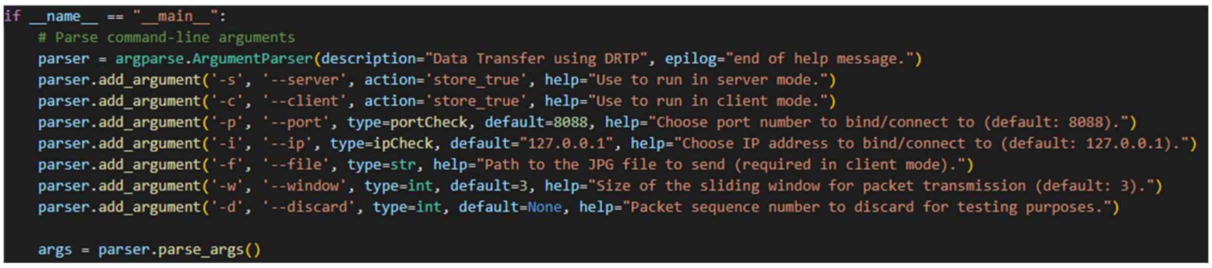
- Window Sizes: Different window sizes (3, 5, and 10 packets) to analyze the impact on throughput.
- Round-Trip Time (RTT) Variations: Simulating different network conditions by adjusting the RTT between 50ms, 100ms and 200ms.

- Packet Loss Simulation: Using the tc-netem tool to simulate packet loss and evaluate the protocol's robustness in adverse conditions.

The project not only underscores the fundamental concepts of reliable data transfer but also provides a practical implementation that can be tested and evaluated under various network scenarios. By conducting experiments with different window sizes, RTTs, and packet loss rates, the project demonstrates how these factors influence throughput and reliability, offering valuable insights into the performance of our DRTP project.

Application.py

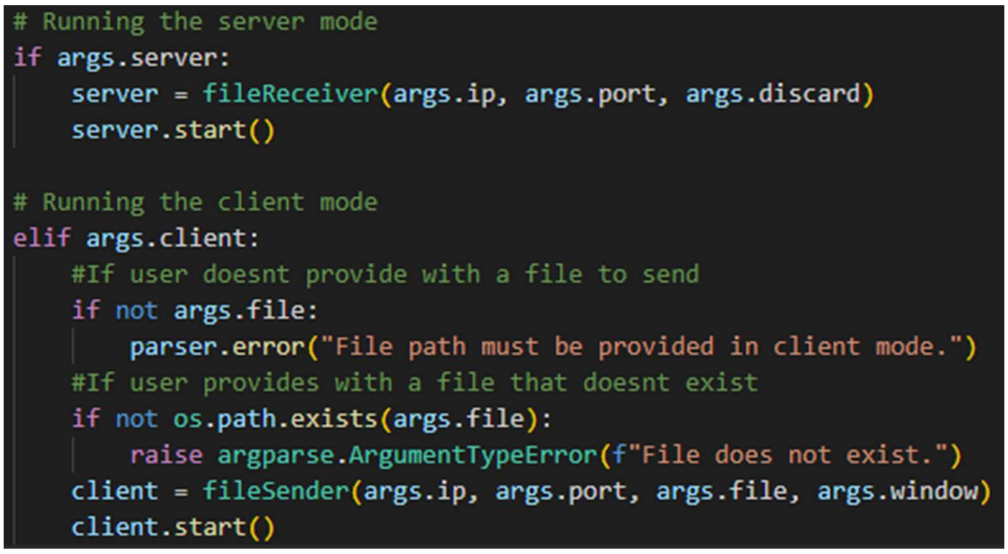
This script is the entry point for the application. It uses argparse to parse command-line arguments such as: Server, Client, IP, Port, File, Window and Discard (figure 1).



```python
if __name__ == "__main__":
    # Parse command-line arguments
    parser = argparse.ArgumentParser(description="Data Transfer using DRTP", epilog="end of help message.")
    parser.add_argument('-s', '--server', action='store_true', help="Use to run in server mode.")
    parser.add_argument('-c', '--client', action='store_true', help="Use to run in client mode.")
    parser.add_argument('-p', '--port', type=portCheck, default=8088, help="Choose port number to bind/connect to (default: 8088).")
    parser.add_argument('-i', '--ip', type=ipCheck, default="127.0.0.1", help="Choose IP address to bind/connect to (default: 127.0.0.1).")
    parser.add_argument('-f', '--file', type=str, help="Path to the JPG file to send (required in client mode).")
    parser.add_argument('-w', '--window', type=int, default=3, help="Size of the sliding window for packet transmission (default: 3).")
    parser.add_argument('-d', '--discard', type=int, default=None, help="Packet sequence number to discard for testing purposes.")

    args = parser.parse_args()
```

Figure 1

Application.py also determines whether to run the application in server- or client mode (figure2).

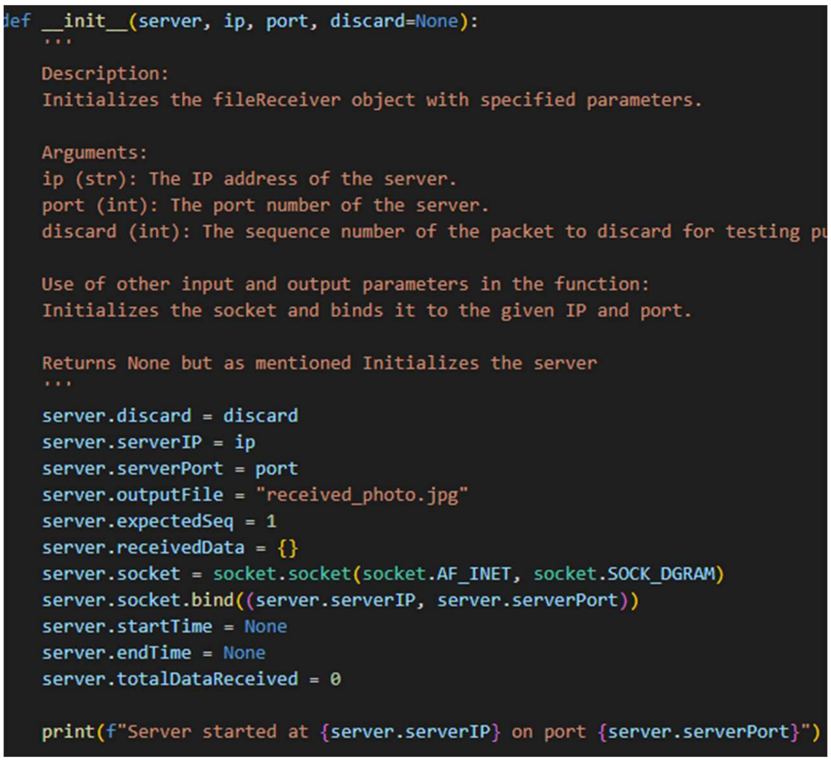


```python
# Running the server mode
if args.server:
    server = fileReceiver(args.ip, args.port, args.discard)
    server.start()

# Running the client mode
elif args.client:
    #If user doesnt provide with a file to send
    if not args.file:
        parser.error("File path must be provided in client mode.")
    #If user provides with a file that doesnt exist
    if not os.path.exists(args.file):
        raise argparse.ArgumentTypeError(f"File does not exist.")
    client = fileSender(args.ip, args.port, args.file, args.window)
    client.start()
```

This script also contains error handling such as: trying to run the application in both server and client mode at the same time, not choosing either server or client mode and more.

Server.py

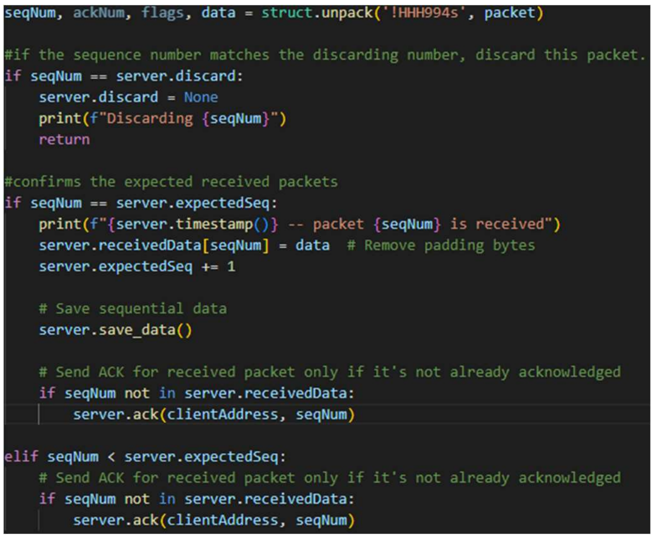
This script is the file receiver of this project, hence why I made it as class fileReceiver. The script starts with the initialization of the server mode, which includes inter alia necessary arguments for the server and UDP socket (figure 2).



```
ef __init__(server, ip, port, discard=None):
    '''
    Description:
    Initializes the fileReceiver object with specified parameters.

    Arguments:
    ip (str): The IP address of the server.
    port (int): The port number of the server.
    discard (int): The sequence number of the packet to discard for testing pu

    Use of other input and output parameters in the function:
    Initializes the socket and binds it to the given IP and port.

    Returns None but as mentioned Initializes the server
    '''
    server.discard = discard
    server.serverIP = ip
    server.serverPort = port
    server.outputFile = "received_photo.jpg"
    server.expectedSeq = 1
    server.receivedData = {}
    server.socket = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
    server.socket.bind((server.serverIP, server.serverPort))
    server.startTime = None
    server.endTime = None
    server.totalDataReceived = 0

    print(f"Server started at {server.serverIP} on port {server.serverPort}")
```

Figure 2

After providing the application with the necessary arguments, the server starts to run and listens to incoming clients on the specified IP and port. The connection between the server and a client establishes only after a successful three-way handshake similar to TCP. The server awaits a Syn packet from the client, responds with a Syn-Ack packet and awaits an Ack packet from the client. Only then, the connection establishes. We can now start our file receiving process. At this point of the process, the server has to handle the incoming data packets (figure 3).

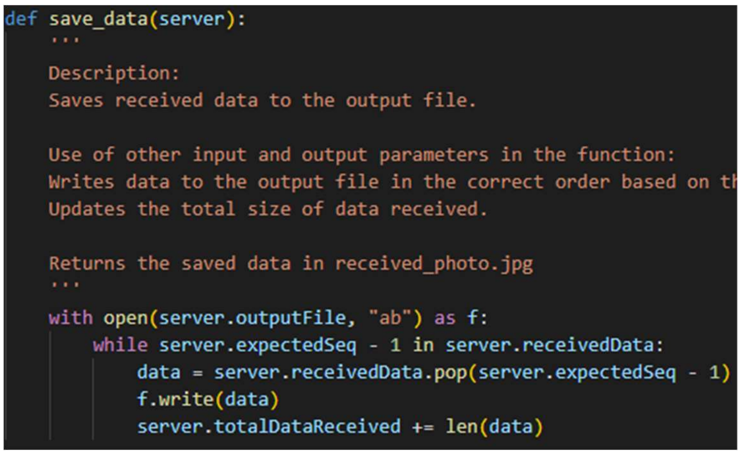


```
seqNum, ackNum, flags, data = struct.unpack('!HHH994s', packet)

#if the sequence number matches the discarding number, discard this packet.
if seqNum == server.discard:
    server.discard = None
    print(f"Discarding {seqNum}")
    return

#confirms the expected received packets
if seqNum == server.expectedSeq:
    print(f"{server.timestamp()} -- packet {seqNum} is received")
    server.receivedData[seqNum] = data  # Remove padding bytes
    server.expectedSeq += 1

    # Save sequential data
    server.save_data()

    # Send ACK for received packet only if it's not already acknowledged
    if seqNum not in server.receivedData:
        server.ack(clientAddress, seqNum)

elif seqNum < server.expectedSeq:
    # Send ACK for received packet only if it's not already acknowledged
    if seqNum not in server.receivedData:
        server.ack(clientAddress, seqNum)
```

Figure 3

The script handles packets by confirming the expected receiving packets in the correct order and saves the data. The server knows that the packet is in correct order when the seq number matches the expected seq. If we receive packets out of order, we simply don't send ack for them and don't save the data.

Since this project expects to receive a jpg file, we save our data in received_photo.jpg which is made when the connection has been established between server and client.



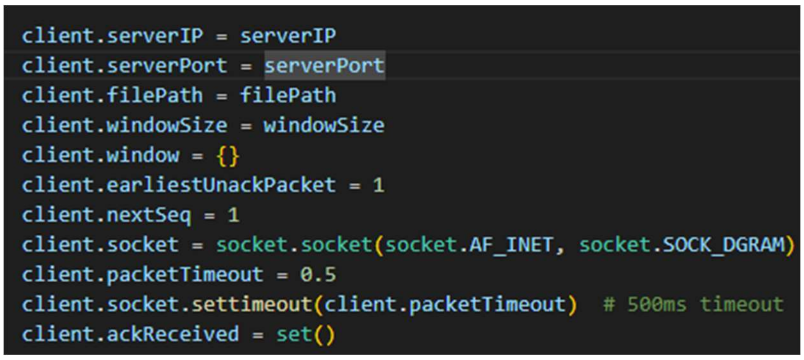
```
def save_data(server):
    '''
    Description:
    Saves received data to the output file.

    Use of other input and output parameters in the function:
    Writes data to the output file in the correct order based on th
    Updates the total size of data received.

    Returns the saved data in received_photo.jpg
    '''
    with open(server.outputFile, "ab") as f:
        while server.expectedSeq - 1 in server.receivedData:
            data = server.receivedData.pop(server.expectedSeq - 1)
            f.write(data)
            server.totalDataReceived += len(data)
```

Figure 4

Since the data is non-textual information, we append the data in binary mode. This also allows us to received multiple data packets without failure because we append the data to the end of the file without overwriting the existing content. We get our data from the receivedData dictionary by looping as long as the seq number minus 1, to ensure correct order. Inside the loop, the data is popped from the dictionary and stored in the data variable. We also keep track of the total data received, for later on when we're calculating our throughput.

Client.py

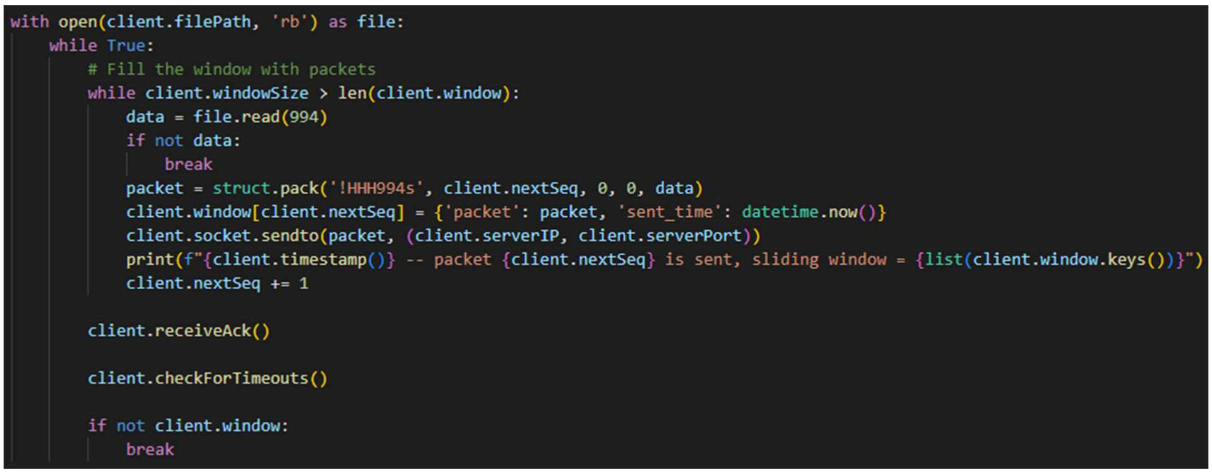
This script is the file sender of this project, hence why I made it as class fileSender. The script starts with the initialization of the client mode, which includes inter alia necessary arguments for the client. UDP socket, packet timeout and more (figure 5).



```python
client.serverIP = serverIP
client.serverPort = serverPort
client.filePath = filePath
client.windowSize = windowSize
client.window = {}
client.earliestUnackPacket = 1
client.nextSeq = 1
client.socket = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
client.packetTimeout = 0.5
client.socket.settimeout(client.packetTimeout)  # 500ms timeout
client.ackReceived = set()
```

Figure 5

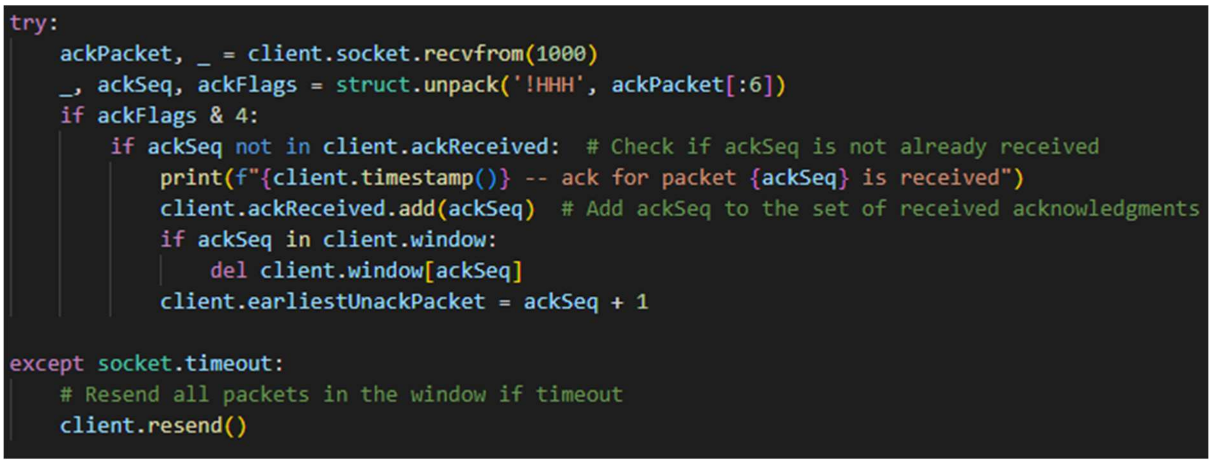
After providing necessary information such as: which IP and port the client should connect to, and which file we want to send, we seek to attempt a connection to the server with the prementioned three-way handshake. If we manage to establish a connection with the server, we send our desired jpg file (figure 6).



```python
with open(client.filePath, 'rb') as file:
    while True:
        # Fill the window with packets
        while client.windowSize > len(client.window):
            data = file.read(994)
            if not data:
                break
            packet = struct.pack('!HHH994s', client.nextSeq, 0, 0, data)
            client.window[client.nextSeq] = {'packet': packet, 'sent_time': datetime.now()}
            client.socket.sendto(packet, (client.serverIP, client.serverPort))
            print(f"{client.timestamp()} -- packet {client.nextSeq} is sent, sliding window = {list(client.window.keys())}")
            client.nextSeq += 1

        client.receiveAck()

        client.checkForTimeouts()

        if not client.window:
            break
```

Figure 6

We send our file by reading our jpg file up to 994 bytes of data from the file at a time. This is done because we are supposed to send packets of 1000 bytes including a 6 byte header. We do this until we don't have more data to read. This data fills inn the sliding window with packets including the sequence number. The packet is then added to the sliding window dictionary including a timestamp and is sent to the server. When sending these packets, the client waits to receive ack from the server and also checks for packet timeouts to handle retransmissions if necessary.



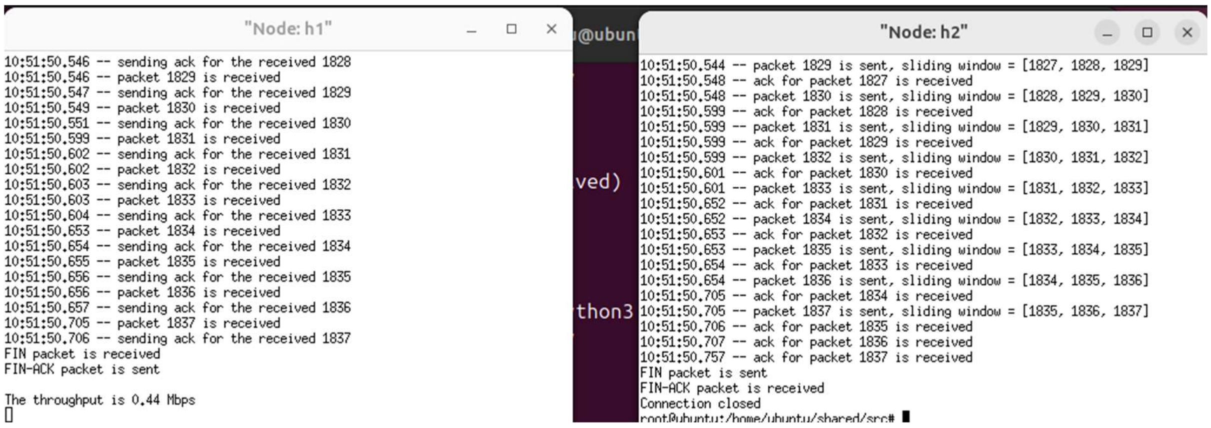
```
try:
    ackPacket, _ = client.socket.recvfrom(1000)
    _, ackSeq, ackFlags = struct.unpack('!HHH', ackPacket[:6])
    if ackFlags & 4:
        if ackSeq not in client.ackReceived:  # Check if ackSeq is not already received
            print(f"{client.timestamp()} -- ack for packet {ackSeq} is received")
            client.ackReceived.add(ackSeq)  # Add ackSeq to the set of received acknowledgments
            if ackSeq in client.window:
                del client.window[ackSeq]
            client.earliestUnackPacket = ackSeq + 1

except socket.timeout:
    # Resend all packets in the window if timeout
    client.resend()
```

Figure 7

The client attempts to receive the acknowledgement packet from the server. If an ack packet is received within the specified timeout (500ms), we proceed to unpack the ack packet. We check the flags to ensure that its an ack packet, not a fin packet. We also check if the ack seq number is not already received to ensure that we can remove the ack packet from the sliding window if present. If we don't receive an ack packet within our timeout, the code handles it by resending the packet.

Discussion 1

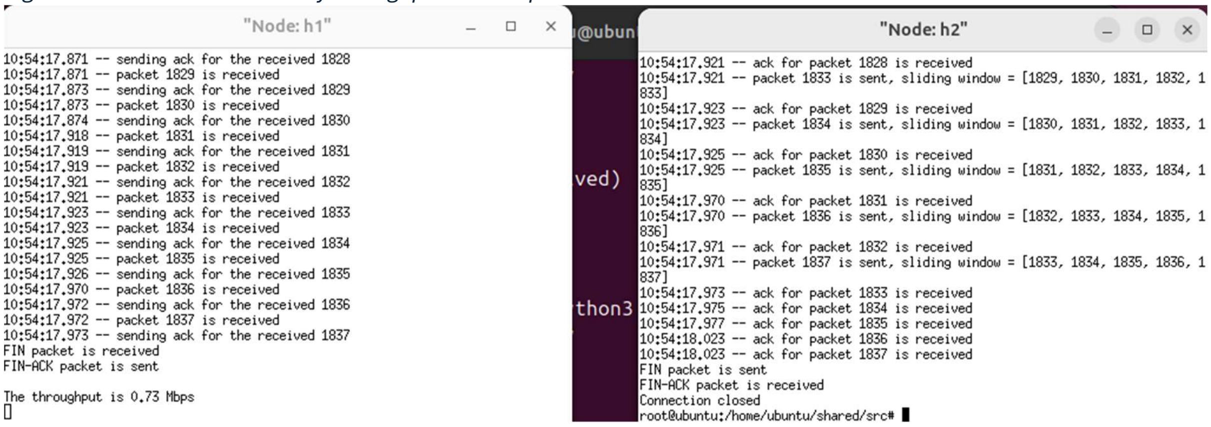
"Execute the file transfer application with window sizes of 3, 5, and 10. Calculate the throughput values for each of these configurations and provide an explanation for your results. For instance, discuss why you observe an increase in throughput as you increase the window size."



```
10:51:50.546 -- sending ack for the received 1828
10:51:50.546 -- packet 1829 is received
10:51:50.547 -- sending ack for the received 1829
10:51:50.549 -- packet 1830 is received
10:51:50.551 -- sending ack for the received 1830
10:51:50.599 -- packet 1831 is received
10:51:50.602 -- sending ack for the received 1831
10:51:50.602 -- packet 1832 is received
10:51:50.603 -- sending ack for the received 1832
10:51:50.603 -- packet 1833 is received
10:51:50.604 -- sending ack for the received 1833
10:51:50.653 -- packet 1834 is received
10:51:50.654 -- sending ack for the received 1834
10:51:50.655 -- packet 1835 is received
10:51:50.656 -- sending ack for the received 1835
10:51:50.656 -- packet 1836 is received
10:51:50.657 -- sending ack for the received 1836
10:51:50.705 -- packet 1837 is received
10:51:50.706 -- sending ack for the received 1837
FIN packet is received
FIN-ACK packet is sent

The throughput is 0.44 Mbps
```

```
10:51:50.544 -- packet 1829 is sent, sliding window = [1827, 1828, 1829]
10:51:50.548 -- ack for packet 1827 is received
10:51:50.548 -- packet 1830 is sent, sliding window = [1828, 1829, 1830]
10:51:50.599 -- ack for packet 1828 is received
10:51:50.599 -- packet 1831 is sent, sliding window = [1829, 1830, 1831]
10:51:50.599 -- ack for packet 1829 is received
10:51:50.599 -- packet 1832 is sent, sliding window = [1830, 1831, 1832]
10:51:50.601 -- ack for packet 1830 is received
10:51:50.601 -- packet 1833 is sent, sliding window = [1831, 1832, 1833]
10:51:50.652 -- ack for packet 1831 is received
10:51:50.652 -- packet 1834 is sent, sliding window = [1832, 1833, 1834]
10:51:50.653 -- ack for packet 1832 is received
10:51:50.653 -- packet 1835 is sent, sliding window = [1833, 1834, 1835]
10:51:50.654 -- ack for packet 1833 is received
10:51:50.654 -- packet 1836 is sent, sliding window = [1834, 1835, 1836]
10:51:50.705 -- ack for packet 1834 is received
10:51:50.705 -- packet 1837 is sent, sliding window = [1835, 1836, 1837]
10:51:50.706 -- ack for packet 1835 is received
10:51:50.707 -- ack for packet 1836 is received
10:51:50.757 -- ack for packet 1837 is received
FIN packet is sent
FIN-ACK packet is received
Connection closed
root@ubuntu:/home/ubuntu/shared/src#
```

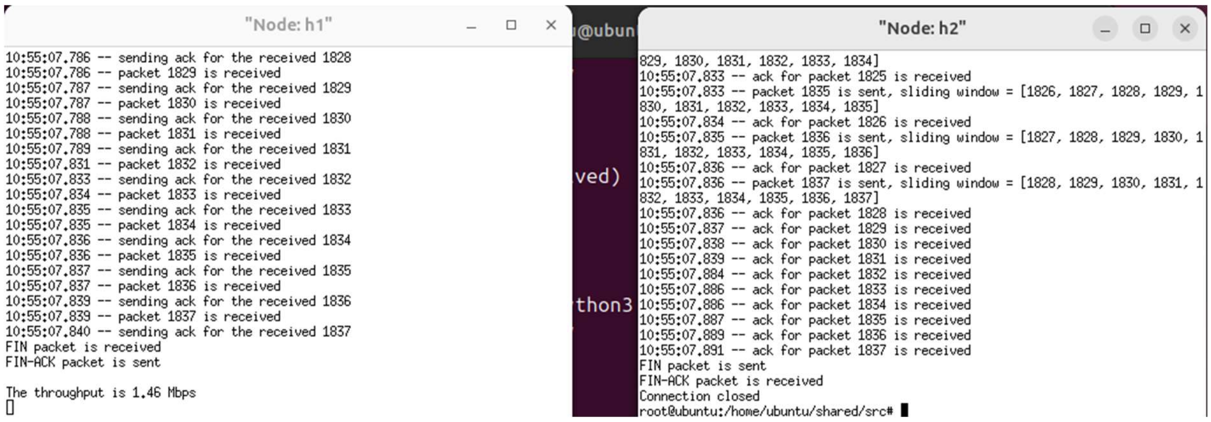
Figure 8 Window 3 100ms delay throughput 0.44Mbps



```
10:54:17.871 -- sending ack for the received 1828
10:54:17.871 -- packet 1829 is received
10:54:17.873 -- sending ack for the received 1829
10:54:17.873 -- packet 1830 is received
10:54:17.874 -- sending ack for the received 1830
10:54:17.918 -- packet 1831 is received
10:54:17.919 -- sending ack for the received 1831
10:54:17.919 -- packet 1832 is received
10:54:17.921 -- sending ack for the received 1832
10:54:17.921 -- packet 1833 is received
10:54:17.923 -- sending ack for the received 1833
10:54:17.923 -- packet 1834 is received
10:54:17.925 -- sending ack for the received 1834
10:54:17.925 -- packet 1835 is received
10:54:17.926 -- sending ack for the received 1835
10:54:17.970 -- packet 1836 is received
10:54:17.972 -- sending ack for the received 1836
10:54:17.972 -- packet 1837 is received
10:54:17.973 -- sending ack for the received 1837
FIN packet is received
FIN-ACK packet is sent

The throughput is 0.73 Mbps
```

```
10:54:17.921 -- ack for packet 1828 is received
10:54:17.921 -- packet 1833 is sent, sliding window = [1829, 1830, 1831, 1832, 1
833]
10:54:17.923 -- ack for packet 1829 is received
10:54:17.923 -- packet 1834 is sent, sliding window = [1830, 1831, 1832, 1833, 1
834]
10:54:17.925 -- ack for packet 1830 is received
10:54:17.925 -- packet 1835 is sent, sliding window = [1831, 1832, 1833, 1834, 1
835]
10:54:17.970 -- ack for packet 1831 is received
10:54:17.970 -- packet 1836 is sent, sliding window = [1832, 1833, 1834, 1835, 1
836]
10:54:17.971 -- ack for packet 1832 is received
10:54:17.971 -- packet 1837 is sent, sliding window = [1833, 1834, 1835, 1836, 1
837]
10:54:17.973 -- ack for packet 1833 is received
10:54:17.975 -- ack for packet 1834 is received
10:54:17.977 -- ack for packet 1835 is received
10:54:18.023 -- ack for packet 1836 is received
10:54:18.023 -- ack for packet 1837 is received
FIN packet is sent
FIN-ACK packet is received
Connection closed
root@ubuntu:/home/ubuntu/shared/src#
```

Figure 9 Window 5 100ms delay throughput 0.73Mbps



```
10:55:07.786 -- sending ack for the received 1828
10:55:07.786 -- packet 1829 is received
10:55:07.787 -- sending ack for the received 1829
10:55:07.787 -- packet 1830 is received
10:55:07.788 -- sending ack for the received 1830
10:55:07.788 -- packet 1831 is received
10:55:07.789 -- sending ack for the received 1831
10:55:07.831 -- packet 1832 is received
10:55:07.833 -- sending ack for the received 1832
10:55:07.834 -- packet 1833 is received
10:55:07.835 -- sending ack for the received 1833
10:55:07.835 -- packet 1834 is received
10:55:07.836 -- sending ack for the received 1834
10:55:07.836 -- packet 1835 is received
10:55:07.837 -- sending ack for the received 1835
10:55:07.837 -- packet 1836 is received
10:55:07.839 -- sending ack for the received 1836
10:55:07.839 -- packet 1837 is received
10:55:07.840 -- sending ack for the received 1837
FIN packet is received
FIN-ACK packet is sent

The throughput is 1.46 Mbps
```

```
829, 1830, 1831, 1832, 1833, 1834]
10:55:07.833 -- ack for packet 1825 is received
10:55:07.833 -- packet 1835 is sent, sliding window = [1826, 1827, 1828, 1829, 1
830, 1831, 1832, 1833, 1834, 1835]
10:55:07.834 -- ack for packet 1826 is received
10:55:07.835 -- packet 1836 is sent, sliding window = [1827, 1828, 1829, 1830, 1
831, 1832, 1833, 1834, 1835, 1836]
10:55:07.836 -- ack for packet 1827 is received
10:55:07.836 -- packet 1837 is sent, sliding window = [1828, 1829, 1830, 1831, 1
832, 1833, 1834, 1835, 1836, 1837]
10:55:07.836 -- ack for packet 1828 is received
10:55:07.837 -- ack for packet 1829 is received
10:55:07.838 -- ack for packet 1830 is received
10:55:07.839 -- ack for packet 1831 is received
10:55:07.884 -- ack for packet 1832 is received
10:55:07.886 -- ack for packet 1833 is received
10:55:07.886 -- ack for packet 1834 is received
10:55:07.887 -- ack for packet 1835 is received
10:55:07.889 -- ack for packet 1836 is received
10:55:07.891 -- ack for packet 1837 is received
FIN packet is sent
FIN-ACK packet is received
Connection closed
root@ubuntu:/home/ubuntu/shared/src#
```

Figure 10 Window 10 100ms delay throughput 1.46Mbps

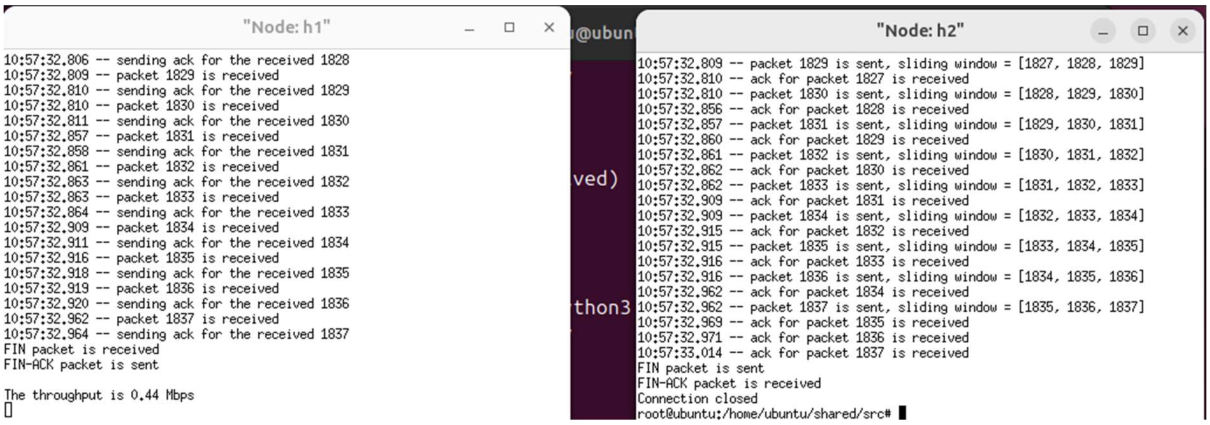
By looking at the results of executing the file transfer application with window sizes of 3, 5, and 10, we see a pattern of increase in throughput as the window size increases (figure 8-10). This result is consistent with the fundamental principles of data transmission and network protocols. With a lager window size, more packets can be in transit simultaneously between the server and client. Since there are more packets in flight at the same time, there is less idle time between transmissions, leading to higher throughput and reduced Round-

Trip Time. With a smaller window size, the client has to wait for acknowledgment packets more frequently, leading to increased protocol overhead. With a smaller window size, the client may not fully utilize available bandwith. Hence why larger window sizes provide more flexibility and improved congestion control.

To conclude, increasing the window size improves throughput by leveraging pipelining, reducing protocol overhead, optimizing congestion control, maximizing the bandwith-delay product, and mitigating the impact of round-trip time. Therefore, are my results showing higher throughput with larger window sizes.

Discussion 2

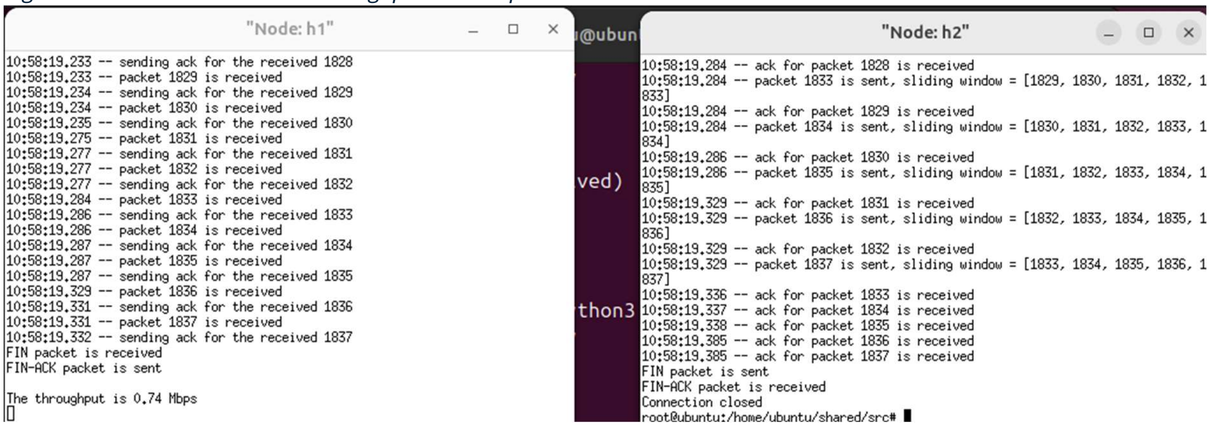
"Modify the RTT to 50ms and 200ms. Run the file transfer application with window sizes of 3, 5, and 10. Calculate throughput values for all these scenarios and provide an explanation for your results."



```
10:57:32.806 -- sending ack for the received 1828
10:57:32.809 -- packet 1829 is received
10:57:32.810 -- sending ack for the received 1829
10:57:32.810 -- packet 1830 is received
10:57:32.811 -- sending ack for the received 1830
10:57:32.857 -- packet 1831 is received
10:57:32.858 -- sending ack for the received 1831
10:57:32.861 -- packet 1832 is received
10:57:32.863 -- sending ack for the received 1832
10:57:32.863 -- packet 1833 is received
10:57:32.864 -- sending ack for the received 1833
10:57:32.909 -- packet 1834 is received
10:57:32.911 -- sending ack for the received 1834
10:57:32.916 -- packet 1835 is received
10:57:32.918 -- sending ack for the received 1835
10:57:32.919 -- packet 1836 is received
10:57:32.920 -- sending ack for the received 1836
10:57:32.962 -- packet 1837 is received
10:57:32.964 -- sending ack for the received 1837
FIN packet is received
FIN-ACK packet is sent

The throughput is 0.44 Mbps
```

```
10:57:32.809 -- packet 1829 is sent, sliding window = [1827, 1828, 1829]
10:57:32.810 -- ack for packet 1827 is received
10:57:32.810 -- packet 1830 is sent, sliding window = [1828, 1829, 1830]
10:57:32.856 -- ack for packet 1828 is received
10:57:32.857 -- packet 1831 is sent, sliding window = [1829, 1830, 1831]
10:57:32.860 -- ack for packet 1829 is received
10:57:32.861 -- packet 1832 is sent, sliding window = [1830, 1831, 1832]
10:57:32.862 -- ack for packet 1830 is received
10:57:32.862 -- packet 1833 is sent, sliding window = [1831, 1832, 1833]
10:57:32.909 -- ack for packet 1831 is received
10:57:32.909 -- packet 1834 is sent, sliding window = [1832, 1833, 1834]
10:57:32.915 -- ack for packet 1832 is received
10:57:32.915 -- packet 1835 is sent, sliding window = [1833, 1834, 1835]
10:57:32.916 -- ack for packet 1833 is received
10:57:32.916 -- packet 1836 is sent, sliding window = [1834, 1835, 1836]
10:57:32.962 -- ack for packet 1834 is received
10:57:32.962 -- packet 1837 is sent, sliding window = [1835, 1836, 1837]
10:57:32.969 -- ack for packet 1835 is received
10:57:32.971 -- ack for packet 1836 is received
10:57:33.014 -- ack for packet 1837 is received
FIN packet is sent
FIN-ACK packet is received
Connection closed
root@ubuntu:/home/ubuntu/shared/src#
```

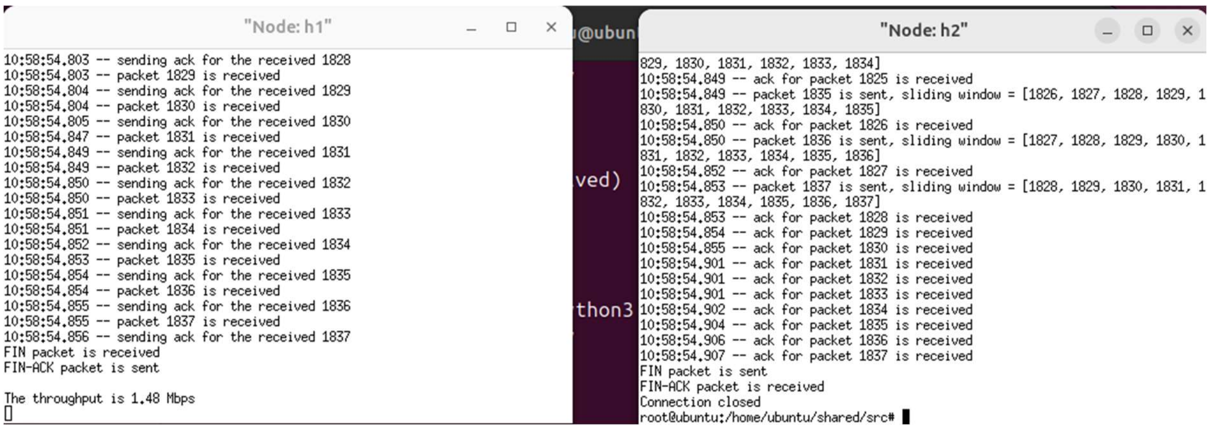
Figure 11 RTT 50ms window 3 Throughput 0.44Mbps



```
10:58:19.233 -- sending ack for the received 1828
10:58:19.233 -- packet 1829 is received
10:58:19.234 -- sending ack for the received 1829
10:58:19.234 -- packet 1830 is received
10:58:19.235 -- sending ack for the received 1830
10:58:19.275 -- packet 1831 is received
10:58:19.277 -- sending ack for the received 1831
10:58:19.277 -- packet 1832 is received
10:58:19.277 -- sending ack for the received 1832
10:58:19.284 -- packet 1833 is received
10:58:19.286 -- sending ack for the received 1833
10:58:19.286 -- packet 1834 is received
10:58:19.287 -- sending ack for the received 1834
10:58:19.287 -- packet 1835 is received
10:58:19.287 -- sending ack for the received 1835
10:58:19.329 -- packet 1836 is received
10:58:19.331 -- sending ack for the received 1836
10:58:19.331 -- packet 1837 is received
10:58:19.332 -- sending ack for the received 1837
FIN packet is received
FIN-ACK packet is sent

The throughput is 0.74 Mbps
```

```
10:58:19.284 -- ack for packet 1828 is received
10:58:19.284 -- packet 1833 is sent, sliding window = [1829, 1830, 1831, 1832, 1
833]
10:58:19.284 -- ack for packet 1829 is received
10:58:19.284 -- packet 1834 is sent, sliding window = [1830, 1831, 1832, 1833, 1
834]
10:58:19.286 -- ack for packet 1830 is received
10:58:19.286 -- packet 1835 is sent, sliding window = [1831, 1832, 1833, 1834, 1
835]
10:58:19.329 -- ack for packet 1831 is received
10:58:19.329 -- packet 1836 is sent, sliding window = [1832, 1833, 1834, 1835, 1
836]
10:58:19.329 -- ack for packet 1832 is received
10:58:19.329 -- packet 1837 is sent, sliding window = [1833, 1834, 1835, 1836, 1
837]
10:58:19.336 -- ack for packet 1833 is received
10:58:19.337 -- ack for packet 1834 is received
10:58:19.338 -- ack for packet 1835 is received
10:58:19.385 -- ack for packet 1836 is received
10:58:19.385 -- ack for packet 1837 is received
FIN packet is sent
FIN-ACK packet is received
Connection closed
root@ubuntu:/home/ubuntu/shared/src#
```

Figure 12 RTT 50ms window 5 Throughput 0.74Mbps



"Node: h1"

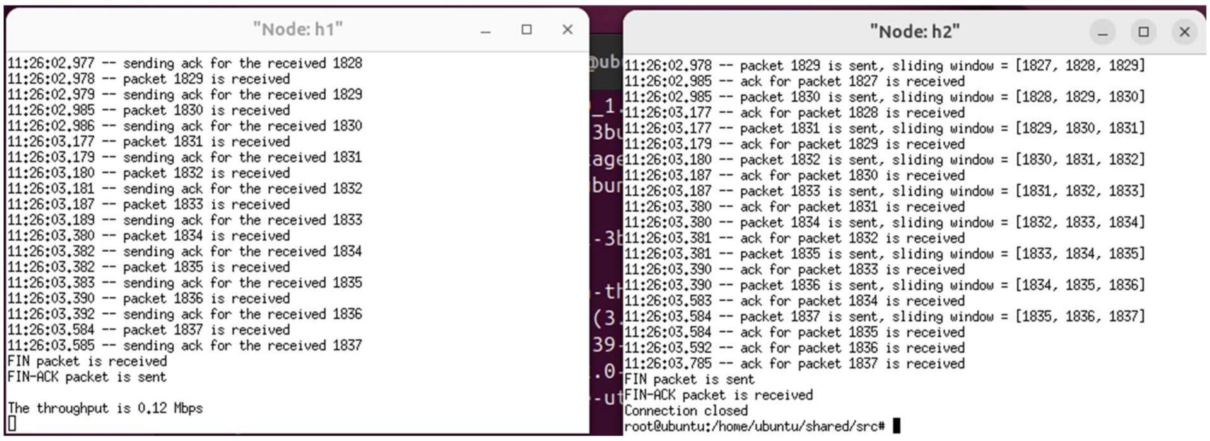
```
10:58:54.803 -- sending ack for the received 1828
10:58:54.803 -- packet 1829 is received
10:58:54.804 -- sending ack for the received 1829
10:58:54.804 -- packet 1830 is received
10:58:54.805 -- sending ack for the received 1830
10:58:54.847 -- packet 1831 is received
10:58:54.849 -- sending ack for the received 1831
10:58:54.849 -- packet 1832 is received
10:58:54.850 -- sending ack for the received 1832
10:58:54.850 -- packet 1833 is received
10:58:54.851 -- sending ack for the received 1833
10:58:54.851 -- packet 1834 is received
10:58:54.852 -- sending ack for the received 1834
10:58:54.853 -- packet 1835 is received
10:58:54.854 -- sending ack for the received 1835
10:58:54.854 -- packet 1836 is received
10:58:54.855 -- sending ack for the received 1836
10:58:54.855 -- packet 1837 is received
10:58:54.856 -- sending ack for the received 1837
FIN packet is received
FIN-ACK packet is sent

The throughput is 1.48 Mbps
```

"Node: h2"

```
829, 1830, 1831, 1832, 1833, 1834]
10:58:54.849 -- ack for packet 1825 is received
10:58:54.849 -- packet 1835 is sent, sliding window = [1826, 1827, 1828, 1829, 1
830, 1831, 1832, 1833, 1834, 1835]
10:58:54.850 -- ack for packet 1826 is received
10:58:54.850 -- packet 1836 is sent, sliding window = [1827, 1828, 1829, 1830, 1
831, 1832, 1833, 1834, 1835, 1836]
10:58:54.852 -- ack for packet 1827 is received
10:58:54.853 -- packet 1837 is sent, sliding window = [1828, 1829, 1830, 1831, 1
832, 1833, 1834, 1835, 1836, 1837]
10:58:54.853 -- ack for packet 1828 is received
10:58:54.854 -- ack for packet 1829 is received
10:58:54.855 -- ack for packet 1830 is received
10:58:54.901 -- ack for packet 1831 is received
10:58:54.901 -- ack for packet 1832 is received
10:58:54.901 -- ack for packet 1833 is received
10:58:54.902 -- ack for packet 1834 is received
10:58:54.904 -- ack for packet 1835 is received
10:58:54.906 -- ack for packet 1836 is received
10:58:54.907 -- ack for packet 1837 is received
FIN packet is sent
FIN-ACK packet is received
Connection closed
root@ubuntu:/home/ubuntu/shared/src#
```

Figure 13 RTT 50ms window 10 Throughput 1.48Mbps

With a lower RTT of 50 ms, the application achieves higher throughput compared to 100ms RTT. This improvement occurs because a lower RTT reduces the time taken for ack packets to travel back to the client, allowing for faster feedback and better utilization of available bandwith. As previously mentioned, increasing the window size leads to higher throughput.



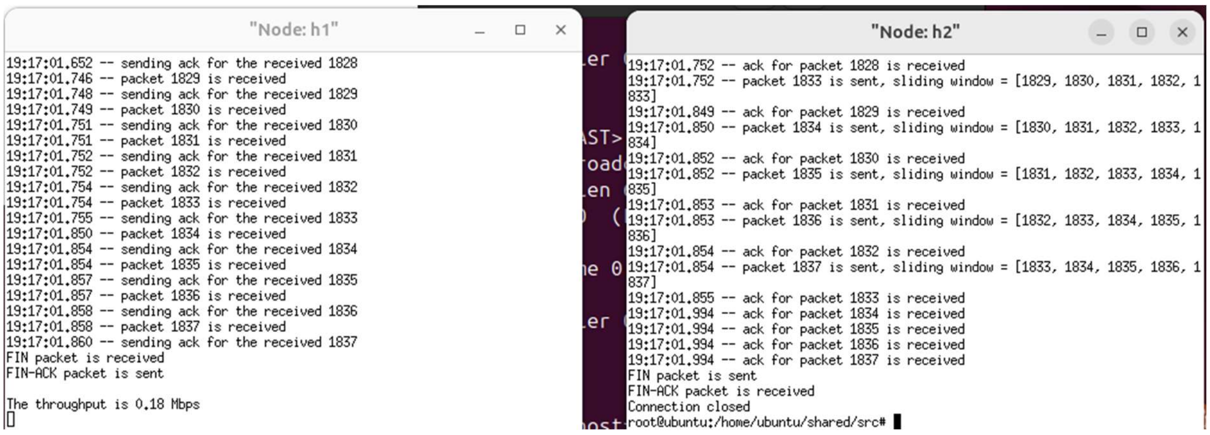
"Node: h1"

```
11:26:02.977 -- sending ack for the received 1828
11:26:02.978 -- packet 1829 is received
11:26:02.979 -- sending ack for the received 1829
11:26:02.985 -- packet 1830 is received
11:26:02.986 -- sending ack for the received 1830
11:26:03.177 -- packet 1831 is received
11:26:03.179 -- sending ack for the received 1831
11:26:03.180 -- packet 1832 is received
11:26:03.181 -- sending ack for the received 1832
11:26:03.187 -- packet 1833 is received
11:26:03.189 -- sending ack for the received 1833
11:26:03.380 -- packet 1834 is received
11:26:03.382 -- sending ack for the received 1834
11:26:03.382 -- packet 1835 is received
11:26:03.383 -- sending ack for the received 1835
11:26:03.390 -- packet 1836 is received
11:26:03.392 -- sending ack for the received 1836
11:26:03.584 -- packet 1837 is received
11:26:03.585 -- sending ack for the received 1837
FIN packet is received
FIN-ACK packet is sent

The throughput is 0.12 Mbps
```

"Node: h2"

```
11:26:02.978 -- packet 1829 is sent, sliding window = [1827, 1828, 1829]
11:26:02.985 -- ack for packet 1827 is received
11:26:02.985 -- packet 1830 is sent, sliding window = [1828, 1829, 1830]
11:26:03.177 -- ack for packet 1828 is received
11:26:03.177 -- packet 1831 is sent, sliding window = [1829, 1830, 1831]
11:26:03.179 -- ack for packet 1829 is received
11:26:03.180 -- packet 1832 is sent, sliding window = [1830, 1831, 1832]
11:26:03.187 -- ack for packet 1830 is received
11:26:03.187 -- packet 1833 is sent, sliding window = [1831, 1832, 1833]
11:26:03.380 -- ack for packet 1831 is received
11:26:03.380 -- packet 1834 is sent, sliding window = [1832, 1833, 1834]
11:26:03.381 -- ack for packet 1832 is received
11:26:03.381 -- packet 1835 is sent, sliding window = [1833, 1834, 1835]
11:26:03.390 -- ack for packet 1833 is received
11:26:03.390 -- packet 1836 is sent, sliding window = [1834, 1835, 1836]
11:26:03.583 -- ack for packet 1834 is received
11:26:03.584 -- packet 1837 is sent, sliding window = [1835, 1836, 1837]
11:26:03.584 -- ack for packet 1835 is received
11:26:03.592 -- ack for packet 1836 is received
11:26:03.785 -- ack for packet 1837 is received
FIN packet is sent
FIN-ACK packet is received
Connection closed
root@ubuntu:/home/ubuntu/shared/src#
```

Figure 14 RTT 200ms window 3 Throughput 0.12Mbps



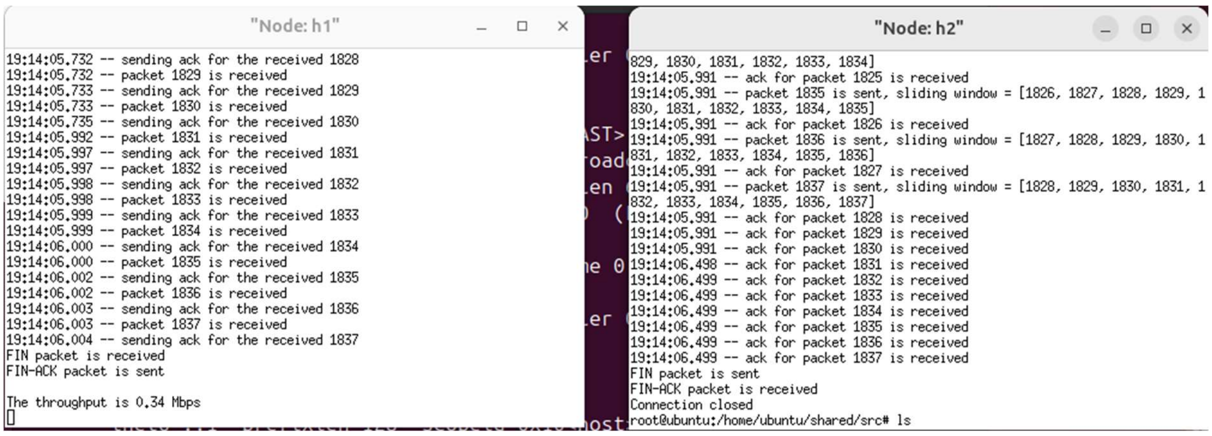
"Node: h1"

```
19:17:01.652 -- sending ack for the received 1828
19:17:01.746 -- packet 1829 is received
19:17:01.748 -- sending ack for the received 1829
19:17:01.749 -- packet 1830 is received
19:17:01.751 -- sending ack for the received 1830
19:17:01.751 -- packet 1831 is received
19:17:01.752 -- sending ack for the received 1831
19:17:01.752 -- packet 1832 is received
19:17:01.754 -- sending ack for the received 1832
19:17:01.754 -- packet 1833 is received
19:17:01.755 -- sending ack for the received 1833
19:17:01.850 -- packet 1834 is received
19:17:01.854 -- sending ack for the received 1834
19:17:01.854 -- packet 1835 is received
19:17:01.857 -- sending ack for the received 1835
19:17:01.857 -- packet 1836 is received
19:17:01.858 -- sending ack for the received 1836
19:17:01.858 -- packet 1837 is received
19:17:01.860 -- sending ack for the received 1837
FIN packet is received
FIN-ACK packet is sent

The throughput is 0.18 Mbps
```

"Node: h2"

```
19:17:01.752 -- ack for packet 1828 is received
19:17:01.752 -- packet 1833 is sent, sliding window = [1829, 1830, 1831, 1832, 1
833]
19:17:01.849 -- ack for packet 1829 is received
19:17:01.850 -- packet 1834 is sent, sliding window = [1830, 1831, 1832, 1833, 1
834]
19:17:01.852 -- ack for packet 1830 is received
19:17:01.852 -- packet 1835 is sent, sliding window = [1831, 1832, 1833, 1834, 1
835]
19:17:01.853 -- ack for packet 1831 is received
19:17:01.853 -- packet 1836 is sent, sliding window = [1832, 1833, 1834, 1835, 1
836]
19:17:01.854 -- ack for packet 1832 is received
19:17:01.854 -- packet 1837 is sent, sliding window = [1833, 1834, 1835, 1836, 1
837]
19:17:01.855 -- ack for packet 1833 is received
19:17:01.994 -- ack for packet 1834 is received
19:17:01.994 -- ack for packet 1835 is received
19:17:01.994 -- ack for packet 1836 is received
19:17:01.994 -- ack for packet 1837 is received
FIN packet is sent
FIN-ACK packet is received
Connection closed
root@ubuntu:/home/ubuntu/shared/src#
```

Figure 15 RTT 200ms window 5 Throughput 0.18Mbps



"Node: h1"

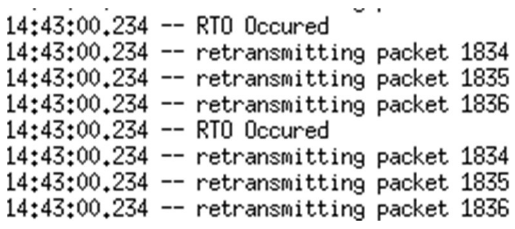
```
19:14:05.732 -- sending ack for the received 1828
19:14:05.732 -- packet 1829 is received
19:14:05.733 -- sending ack for the received 1829
19:14:05.733 -- packet 1830 is received
19:14:05.735 -- sending ack for the received 1830
19:14:05.992 -- packet 1831 is received
19:14:05.997 -- sending ack for the received 1831
19:14:05.997 -- packet 1832 is received
19:14:05.998 -- sending ack for the received 1832
19:14:05.998 -- packet 1833 is received
19:14:05.999 -- sending ack for the received 1833
19:14:05.999 -- packet 1834 is received
19:14:06.000 -- sending ack for the received 1834
19:14:06.000 -- packet 1835 is received
19:14:06.002 -- sending ack for the received 1835
19:14:06.002 -- packet 1836 is received
19:14:06.003 -- sending ack for the received 1836
19:14:06.003 -- packet 1837 is received
19:14:06.004 -- sending ack for the received 1837
FIN packet is received
FIN-ACK packet is sent

The throughput is 0.34 Mbps
```

"Node: h2"

```
829, 1830, 1831, 1832, 1833, 1834]
19:14:05.991 -- ack for packet 1825 is received
19:14:05.991 -- packet 1835 is sent, sliding window = [1826, 1827, 1828, 1829, 1
830, 1831, 1832, 1833, 1834, 1835]
19:14:05.991 -- ack for packet 1826 is received
19:14:05.991 -- packet 1836 is sent, sliding window = [1827, 1828, 1829, 1830, 1
831, 1832, 1833, 1834, 1835, 1836]
19:14:05.991 -- ack for packet 1827 is received
19:14:05.991 -- packet 1837 is sent, sliding window = [1828, 1829, 1830, 1831, 1
832, 1833, 1834, 1835, 1836, 1837]
19:14:05.991 -- ack for packet 1828 is received
19:14:05.991 -- ack for packet 1829 is received
19:14:05.991 -- ack for packet 1830 is received
19:14:06.498 -- ack for packet 1831 is received
19:14:06.499 -- ack for packet 1832 is received
19:14:06.499 -- ack for packet 1833 is received
19:14:06.499 -- ack for packet 1834 is received
19:14:06.499 -- ack for packet 1835 is received
19:14:06.499 -- ack for packet 1836 is received
19:14:06.499 -- ack for packet 1837 is received
FIN packet is sent
FIN-ACK packet is received
Connection closed
root@ubuntu:/home/ubuntu/shared/src# ls
```

Figure 16 RTT 200ms window 10 Throughput 0.34Mbps

Contrariwise, with an increased RTT of 200ms, the application experiences significantly lower throughput. The higher RTT introduces longer delays between packet transmission and ack receipt, leading to reduced efficiency in utilizing available bandwith.



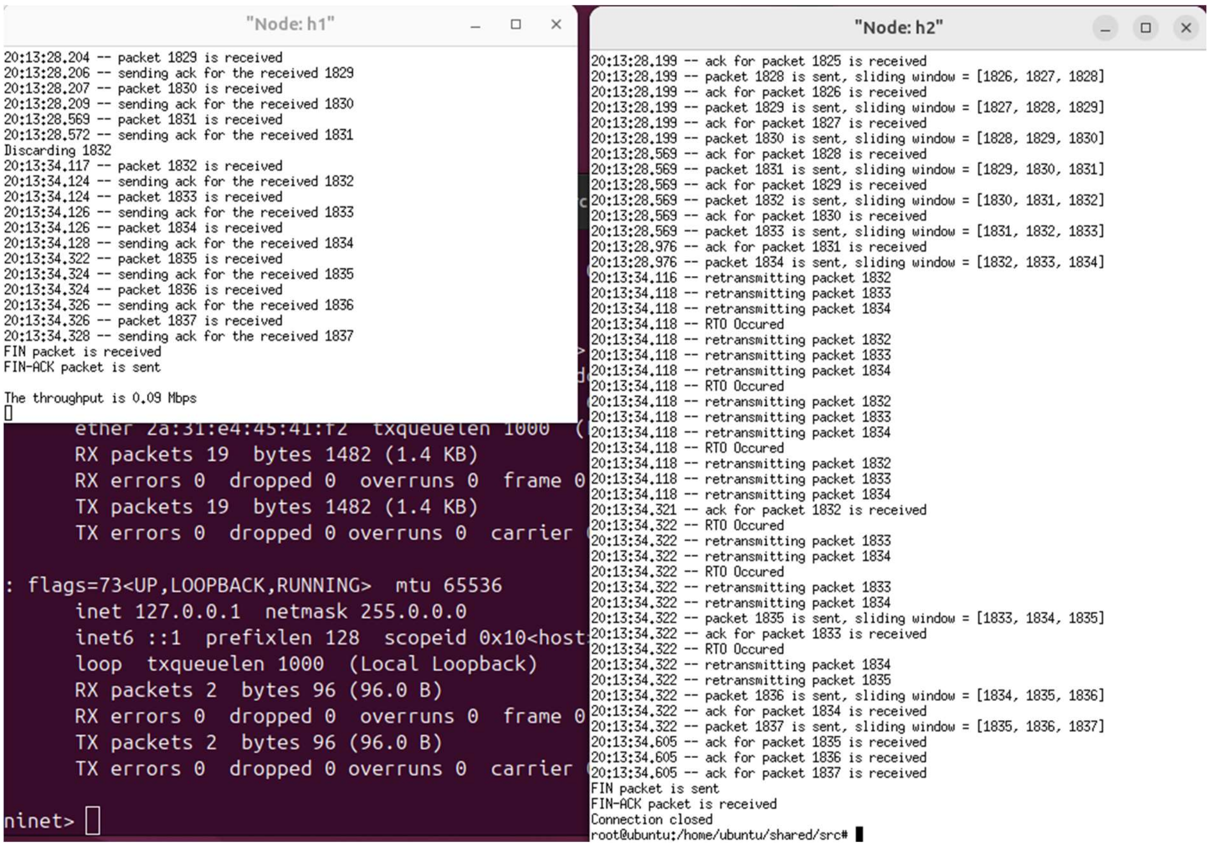
```
14:43:00.234 -- RTO Occured
14:43:00.234 -- retransmitting packet 1834
14:43:00.234 -- retransmitting packet 1835
14:43:00.234 -- retransmitting packet 1836
14:43:00.234 -- RTO Occured
14:43:00.234 -- retransmitting packet 1834
14:43:00.234 -- retransmitting packet 1835
14:43:00.234 -- retransmitting packet 1836
```

Figure 17 RTO Occuring when RTT 200ms

I also noticed when the RTT was set to 200ms, the application experienced RTO occurring, leading it to retransmitting some packets. After testing the application with increased packet timeouts, I have concluded that the RTO occurs because the ack packets don't have enough time to reach the client before the client times out. With increased packet timeout, the application has no need for retransmitting packets.

Discussion 3

"Use the --discard or -d flag on the server side to drop a packet, which will make the client resend it. Show how the reliable transport protocol works and how it deals with resent packets."



```
"Node: h1"
20:13:28.204 -- packet 1829 is received
20:13:28.206 -- sending ack for the received 1829
20:13:28.207 -- packet 1830 is received
20:13:28.209 -- sending ack for the received 1830
20:13:28.569 -- packet 1831 is received
20:13:28.572 -- sending ack for the received 1831
Discarding 1832
20:13:34.117 -- packet 1832 is received
20:13:34.124 -- sending ack for the received 1832
20:13:34.124 -- packet 1833 is received
20:13:34.126 -- sending ack for the received 1833
20:13:34.126 -- packet 1834 is received
20:13:34.128 -- sending ack for the received 1834
20:13:34.322 -- packet 1835 is received
20:13:34.324 -- sending ack for the received 1835
20:13:34.324 -- packet 1836 is received
20:13:34.326 -- sending ack for the received 1836
20:13:34.326 -- packet 1837 is received
20:13:34.328 -- sending ack for the received 1837
FIN packet is received
FIN-ACK packet is sent

The throughput is 0.09 Mbps
```

```
        ether 2a:31:e4:45:41:f2  txqueuelen 1000  (
        RX packets 19  bytes 1482 (1.4 KB)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 19  bytes 1482 (1.4 KB)
        TX errors 0  dropped 0 overruns 0  carrier

: flags=73<UP,LOOPBACK,RUNNING>  mtu 65536
        inet 127.0.0.1  netmask 255.0.0.0
        inet6 ::1  prefixlen 128  scopeid 0x10<host
        loop  txqueuelen 1000  (Local Loopback)
        RX packets 2  bytes 96 (96.0 B)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 2  bytes 96 (96.0 B)
        TX errors 0  dropped 0 overruns 0  carrier

ninet>
```

```
"Node: h2"
20:13:28.199 -- ack for packet 1825 is received
20:13:28.199 -- packet 1828 is sent, sliding window = [1826, 1827, 1828]
20:13:28.199 -- ack for packet 1826 is received
20:13:28.199 -- packet 1829 is sent, sliding window = [1827, 1828, 1829]
20:13:28.199 -- ack for packet 1827 is received
20:13:28.199 -- packet 1830 is sent, sliding window = [1828, 1829, 1830]
20:13:28.569 -- ack for packet 1828 is received
20:13:28.569 -- packet 1831 is sent, sliding window = [1829, 1830, 1831]
20:13:28.569 -- ack for packet 1829 is received
20:13:28.569 -- packet 1832 is sent, sliding window = [1830, 1831, 1832]
20:13:28.569 -- ack for packet 1830 is received
20:13:28.569 -- packet 1833 is sent, sliding window = [1831, 1832, 1833]
20:13:28.976 -- ack for packet 1831 is received
20:13:28.976 -- packet 1834 is sent, sliding window = [1832, 1833, 1834]
20:13:34.116 -- retransmitting packet 1832
20:13:34.118 -- retransmitting packet 1833
20:13:34.118 -- retransmitting packet 1834
20:13:34.118 -- RTO Occured
20:13:34.118 -- retransmitting packet 1832
20:13:34.118 -- retransmitting packet 1833
20:13:34.118 -- retransmitting packet 1834
20:13:34.118 -- RTO Occured
20:13:34.118 -- retransmitting packet 1832
20:13:34.118 -- retransmitting packet 1833
20:13:34.118 -- retransmitting packet 1834
20:13:34.118 -- RTO Occured
20:13:34.118 -- retransmitting packet 1832
20:13:34.118 -- retransmitting packet 1833
20:13:34.118 -- retransmitting packet 1834
20:13:34.321 -- ack for packet 1832 is received
20:13:34.322 -- RTO Occured
20:13:34.322 -- retransmitting packet 1833
20:13:34.322 -- retransmitting packet 1834
20:13:34.322 -- RTO Occured
20:13:34.322 -- retransmitting packet 1833
20:13:34.322 -- retransmitting packet 1834
20:13:34.322 -- packet 1835 is sent, sliding window = [1833, 1834, 1835]
20:13:34.322 -- ack for packet 1833 is received
20:13:34.322 -- RTO Occured
20:13:34.322 -- retransmitting packet 1834
20:13:34.322 -- retransmitting packet 1835
20:13:34.322 -- packet 1836 is sent, sliding window = [1834, 1835, 1836]
20:13:34.322 -- ack for packet 1834 is received
20:13:34.322 -- packet 1837 is sent, sliding window = [1835, 1836, 1837]
20:13:34.605 -- ack for packet 1835 is received
20:13:34.605 -- ack for packet 1836 is received
20:13:34.605 -- ack for packet 1837 is received
FIN packet is sent
FIN-ACK packet is received
Connection closed
root@ubuntu:/home/ubuntu/shared/src#
```

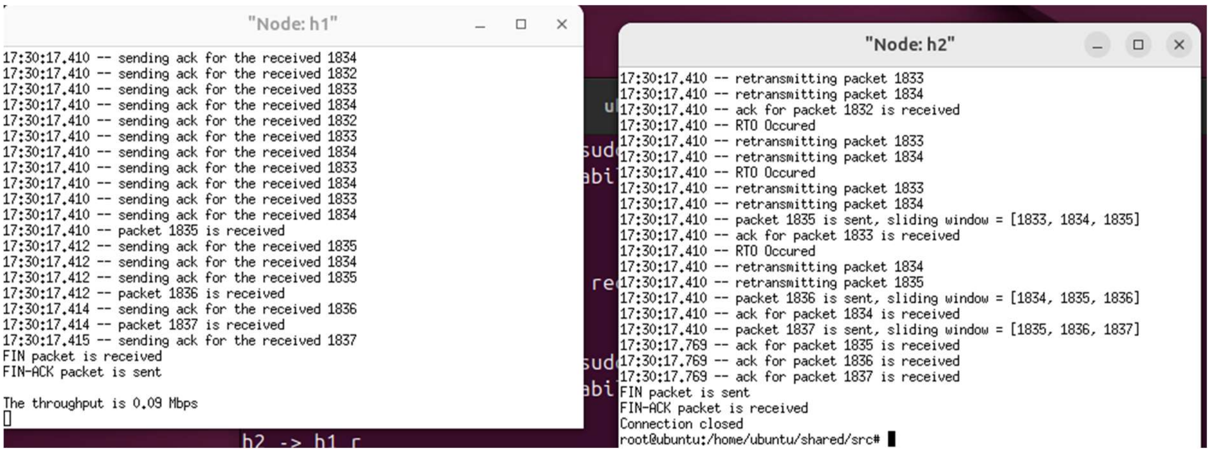
Figure 18 discard packet result

The client detects the missing acknowledgement and triggers RTO occurred and a retransmission of the affected packets within the current sliding window. The client logs indicate multiple instances of retransmissions for the dropped packets. Each retransmission attempt tries to ensure that the dropped packets are eventually received by the server. The acknowledgment for resent packets validates that the server has indeed received and acted upon those specific packet retransmissions. The client ensures that each packet is acknowledged before proceeding with the transmission of subsequent packets, maintaining the reliability of the data transfer process.

Discussion 4

“To demonstrate how effective your code is, use tc-netem (refer to the lab manual) to simulate packet loss. To do this, modify your simple-topo.py file by commenting out line#43 net["r"].cmd("tc qdisc add dev r-eth1 root netem delay 100ms") and uncommenting line#44: net["r"].cmd("tc qdisc add dev r-eth1 root netem delay 100ms

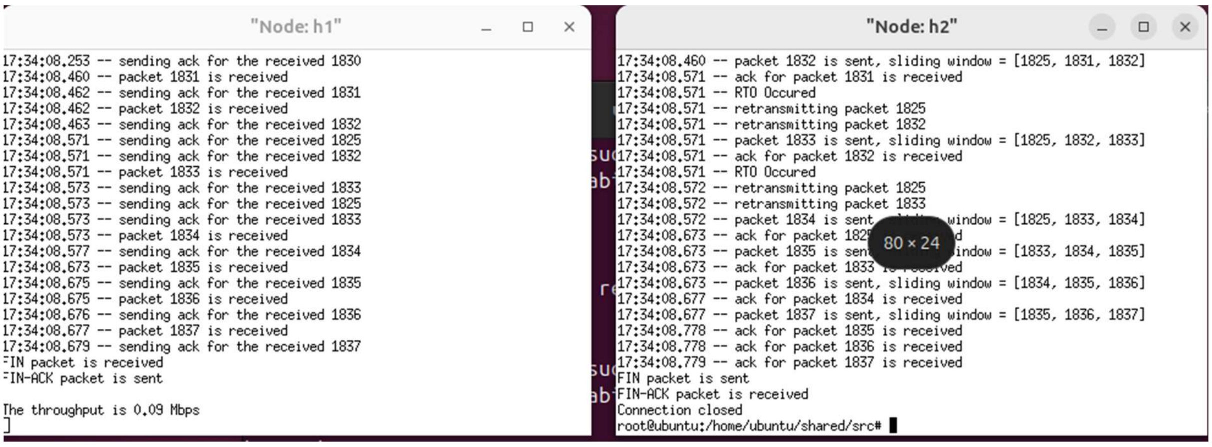
loss 2%"). Test with a loss rate of 5% as well. Include the results in your discussion section and explain what you observe."



```
"Node: h1"
17:30:17.410 -- sending ack for the received 1834
17:30:17.410 -- sending ack for the received 1832
17:30:17.410 -- sending ack for the received 1833
17:30:17.410 -- sending ack for the received 1834
17:30:17.410 -- sending ack for the received 1832
17:30:17.410 -- sending ack for the received 1833
17:30:17.410 -- sending ack for the received 1834
17:30:17.410 -- sending ack for the received 1833
17:30:17.410 -- sending ack for the received 1834
17:30:17.410 -- sending ack for the received 1833
17:30:17.410 -- sending ack for the received 1834
17:30:17.410 -- packet 1835 is received
17:30:17.412 -- sending ack for the received 1835
17:30:17.412 -- sending ack for the received 1834
17:30:17.412 -- sending ack for the received 1835
17:30:17.412 -- packet 1836 is received
17:30:17.414 -- sending ack for the received 1836
17:30:17.414 -- packet 1837 is received
17:30:17.415 -- sending ack for the received 1837
FIN packet is received
FIN-ACK packet is sent

The throughput is 0.09 Mbps
```

```
"Node: h2"
17:30:17.410 -- retransmitting packet 1833
17:30:17.410 -- retransmitting packet 1834
17:30:17.410 -- ack for packet 1832 is received
17:30:17.410 -- RTO Occured
17:30:17.410 -- retransmitting packet 1833
17:30:17.410 -- retransmitting packet 1834
17:30:17.410 -- RTO Occured
17:30:17.410 -- retransmitting packet 1833
17:30:17.410 -- retransmitting packet 1834
17:30:17.410 -- packet 1835 is sent, sliding window = [1833, 1834, 1835]
17:30:17.410 -- ack for packet 1833 is received
17:30:17.410 -- RTO Occured
17:30:17.410 -- retransmitting packet 1834
17:30:17.410 -- retransmitting packet 1835
17:30:17.410 -- packet 1836 is sent, sliding window = [1834, 1835, 1836]
17:30:17.410 -- ack for packet 1834 is received
17:30:17.410 -- packet 1837 is sent, sliding window = [1835, 1836, 1837]
17:30:17.769 -- ack for packet 1835 is received
17:30:17.769 -- ack for packet 1836 is received
17:30:17.769 -- ack for packet 1837 is received
FIN packet is sent
FIN-ACK packet is received
Connection closed
root@ubuntu:/home/ubuntu/shared/src#
```

Figure 19 2% loss rate



```
"Node: h1"
17:34:08.253 -- sending ack for the received 1830
17:34:08.460 -- packet 1831 is received
17:34:08.462 -- sending ack for the received 1831
17:34:08.462 -- packet 1832 is received
17:34:08.463 -- sending ack for the received 1832
17:34:08.571 -- sending ack for the received 1825
17:34:08.571 -- sending ack for the received 1832
17:34:08.571 -- packet 1833 is received
17:34:08.573 -- sending ack for the received 1833
17:34:08.573 -- sending ack for the received 1825
17:34:08.573 -- sending ack for the received 1833
17:34:08.573 -- packet 1834 is received
17:34:08.577 -- sending ack for the received 1834
17:34:08.673 -- packet 1835 is received
17:34:08.675 -- sending ack for the received 1835
17:34:08.675 -- packet 1836 is received
17:34:08.676 -- sending ack for the received 1836
17:34:08.677 -- packet 1837 is received
17:34:08.679 -- sending ack for the received 1837
FIN packet is received
FIN-ACK packet is sent

The throughput is 0.09 Mbps
```

```
"Node: h2"
17:34:08.460 -- packet 1832 is sent, sliding window = [1825, 1831, 1832]
17:34:08.571 -- ack for packet 1831 is received
17:34:08.571 -- RTO Occured
17:34:08.571 -- retransmitting packet 1825
17:34:08.571 -- retransmitting packet 1832
17:34:08.571 -- packet 1833 is sent, sliding window = [1825, 1832, 1833]
17:34:08.571 -- ack for packet 1832 is received
17:34:08.571 -- RTO Occured
17:34:08.572 -- retransmitting packet 1825
17:34:08.572 -- retransmitting packet 1833
17:34:08.572 -- packet 1834 is sent, [illegible]window = [1825, 1833, 1834]
17:34:08.673 -- ack for packet 182[illegible]d
17:34:08.673 -- packet 1835 is sen[illegible]indow = [1833, 1834, 1835]
17:34:08.673 -- ack for packet 1833 [illegible]ved
17:34:08.673 -- packet 1836 is sent, sliding window = [1834, 1835, 1836]
17:34:08.677 -- ack for packet 1834 is received
17:34:08.677 -- packet 1837 is sent, sliding window = [1835, 1836, 1837]
17:34:08.778 -- ack for packet 1835 is received
17:34:08.778 -- ack for packet 1836 is received
17:34:08.779 -- ack for packet 1837 is received
FIN packet is sent
FIN-ACK packet is received
Connection closed
root@ubuntu:/home/ubuntu/shared/src#
```

Figure 20 5% loss rate

Despite the simulated packet loss, my file transfer application successfully detected the missing acknowledgements and triggered retransmissions where needed. The client made sure that all lost packets were resent to the server to achieve reliable data transmission. Both the 2% and 5% loss rate scenarios ended up with a throughput of 0.09 Mbps. Despite the decrease in throughput, my application maintained reliable data transfer by prioritizing data integrity over speed in the presence of packet loss. My application's ability to adapt to varying network conditions highlights its robustness and flexibility.