COL334 Assignment-4

Bhyri Siddhartha Roy
2022CS11105

Mudavath Sai Nikhil
2022CS11633

Introduction:

TCP (Transmission Control Protocol) and UDP (User Datagram Protocol) are two fundamental transport layer protocols in computer networks. For this assignment, the objective is to design and implement reliable file transfer protocol over UDP (User Datagram Protocol) sockets. Unlike TCP, UDP is a connectionless protocol that does not inherently provide reliability, ordered delivery, or congestion control, making it an efficient but "unreliable" protocol for data transfer.

Part 1: Reliability - Implement a reliable UDP by incorporating features such as acknowledgments, packet retransmission, fast recovery, sequence numbering, and timeout handling.
Part 2: Congestion Control - Integrate congestion control like TCP Reno, including mechanisms for slow start, congestion avoidance, and fast recovery.

PART-1 RELIABILITY:

IMPLEMENTATION:

CLIENT:

Client sends “START” message until it receives some data from the server.

In the receive function on the client side, after receiving the packet,we extract the sequence number from the packet and check the below conditions:

1.1 expected_seq_num==received_seq_num:
in this case we write the data present in the packet to the file and then calculate the next expected seq num by adding the length of the received segment. Now if the next expected seq num is present in the buffer then we pop it from the buffer and write to the file and this process repeats until the next expected_seq_num is present in the buffer. Then after when the next expected_seq_num is not in the buffer then we send a ack for this sequence number.

1.2 expected_seq_num>received_seq_num:
in this case since the expected_seq_num is less than received_Seq_num which means that this is a duplicate packet we send a ack for this packet with the received seq num.

1.3 expected_seq_num<received_seq_num:

in this case since we received an advance packet, we store it in the buffer and then send a ack with the expected seq_num, so that we get the correct in order packet.

When the end message (“EOF”) is received in the packet, then client sends a ack for the received end message.Now the client starts a timer of 30 seconds and then stops after the 30 seconds if no “CLOSE” message is received. If the “CLOSE” message is received then it closes immediately.

SERVER:

1. After server receives a “START” message from the server it starts reading the file and sends segments.

2. Server uses a sliding window approach with WINDOW_SIZE to control the number of packets sent at any moment, simulating TCP’s sliding window.Each packet is generated with a unique sequence number and timestamp, which helps track when it was sent, allowing RTT calculations for adaptive timeout and retransmission.server now waits for ACKs to arrive after sending a packet.

3. When an ACK is received it checks the sequence number of the ACK and determines if it is new or a duplicate. If new, it advances the window, marking packets up to that sequence number as acknowledged.
For duplicate ACKs, the function counts them; on receiving three duplicate ACKs for the same sequence, it invokes fast_recovery() to retransmit the suspected lost packet (similar to TCP’s Fast Retransmit).

4. Timeout and Retransmission:
After sending a packet server uses retransmission timeout (RTO) .The timeout period is recalculated after each ACK to reflect the latest network conditions. If an ACK isn’t received and timeout occurs for a sent packet then retransmission takes place and now the timeout will be twice the previous rto

5. **TIMEOUT CALCULATION:**
Server uses rto calculation based on the method described in RFC 6298. It uses two varaibles named SRTT and RTTVAR and they are calculated as below.
   1. Initially rto is set to 1.
   2. After the first RTT measurment (R) is made :
      SRTT <- R
      RTTVAR <- R/2
      RTO <- SRTT + 4*RTTVAR
   3. For the next RTT measurements made rto is calculated as:
      RTTVAR <- 0.75 * RTTVAR + 0.25* abs(SRTT - R’)
      SRTT <- 0.875 * SRTT + 0.125 * R’
      RTO <- SRTT + 4*RTTVAR

   Now the calculated RTO is set as timeout for the next transmission of packet.

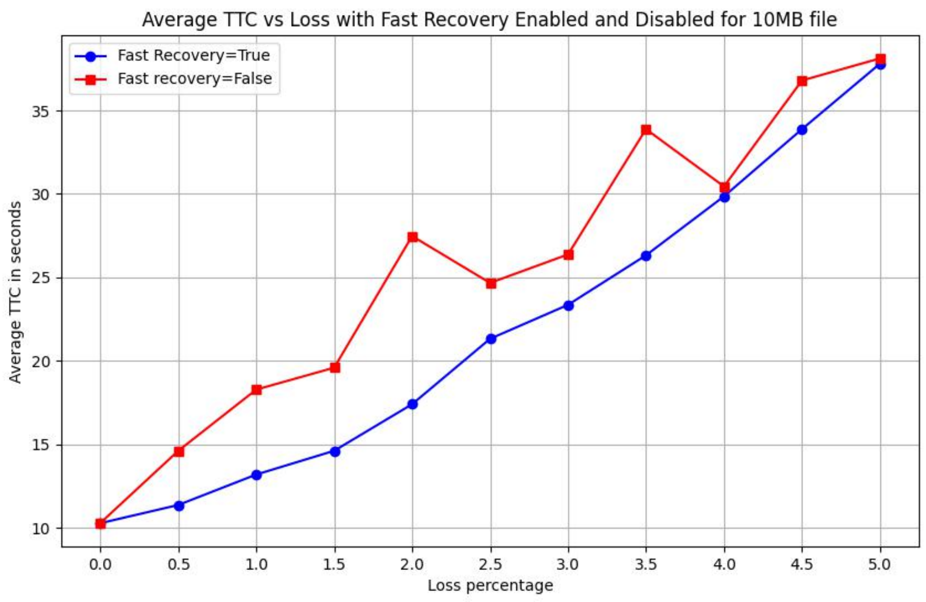
6. When the data in the file is completed then the server sends EOF for the client and waits for a ack from the client. When the server receives ack for EOF then server sends a CLOSE message to the client and closes the connection. If the client receives CLOSE then it closes else the timer of 30 seconds gets over and closes the connection.

PACKET FORMAT:

We have implemented the packet as json object which has fields as seq_num, data and a time stamp when the packet was sent for RTT calculation. The ack packet has fields like seq_num which indicates the next required seq_num and time_stamp which indicates when the packet was sent to which this ack corresponds to.

RESULTS AND ANALYSIS:

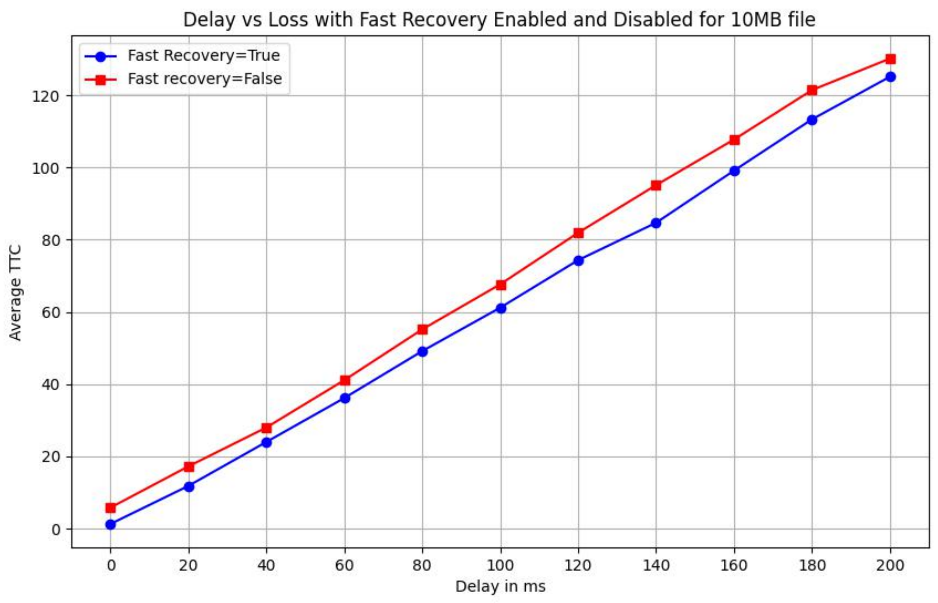
PACKET LOSS vs TRANSMISSION TIME FOR A 10MB FILE:



The graph above illustrates that Fast Recovery generally improves efficiency, as shown in figure, which tracks how Time to Complete (TTC) changes with Fast Recovery enabled. Fast Recovery typically reduces transmission time by quickly retransmitting lost packets as soon as three duplicate acknowledgments are received, rather than waiting for a timeout, allowing for earlier detection of packet loss.

Fast Recovery relies on duplicate ACKs to detect and handle packet loss. For large files, where packet bursts are common, congestion in the network can cause increased delays in duplicate ACKs, prolonging the recovery process.So we can observe some increase in average TTC for loss of 4.5% and 5 %.

DLEAY vs TRANSMISSION TIME FOR A 10MB FILE:



Fast recovery when enabled takes less time than when it is disabled. As the delay percentage increases average ttc also increases which is because of more retransmissions when the delay of link is increased.

PART-2: CONGESTION CONTROL

IMPLEMENTATION:

CLIENT:

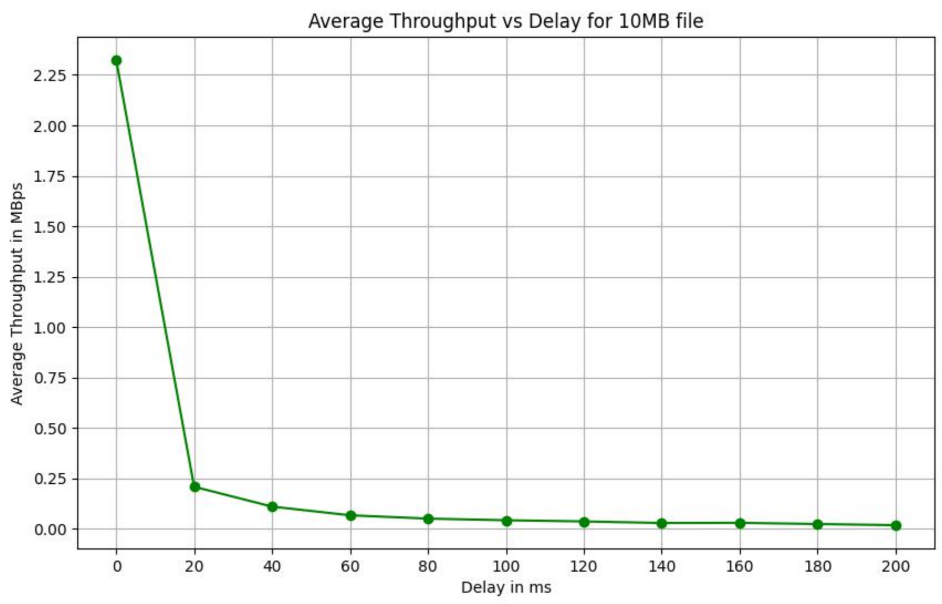
Client implementation remains the same as the previous part which implements reliable data transfer.

SERVER:

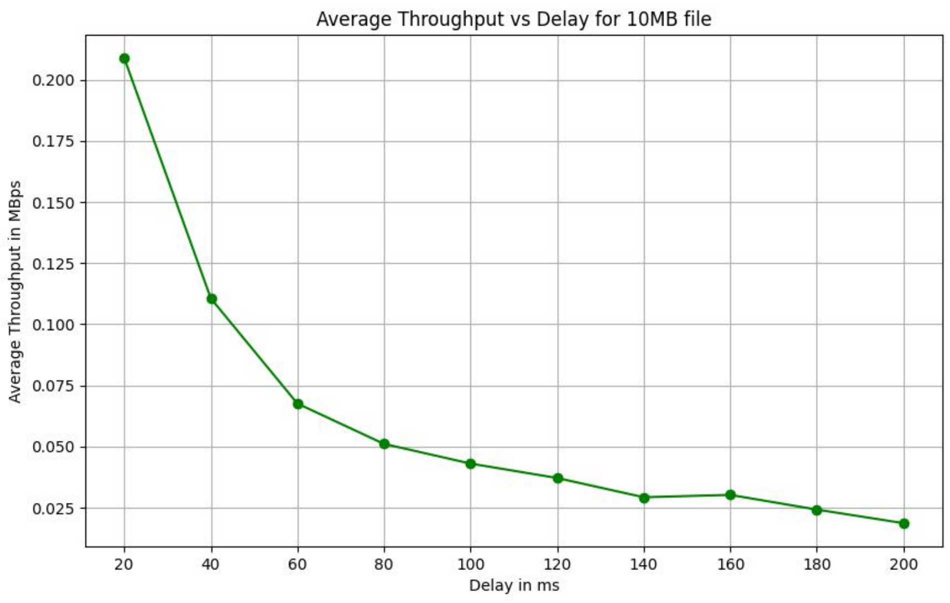
On the server side, congestion control mechanisms are implemented upon the reliable data transfer from part-1. Congestion window (cwnd) is dynamically updated based on the acks' received. Initially server is in slow start phase where cwnd is 1 and has exponential increase upon receiving acks. when cwnd crosses the ssthresh(slow start threshold) then server enters the congestion avoidance phase. Now when three duplicate acks are received either in slow start or congestion avoidance server goes to the fast recovery phase. Whenever a timeout occurs cwnd is updated to 1 and ssthresh (slow start threshold) is updated accordingly and server enters again the slow start phase.

RESULTS AND ANALYSIS:

THROUGHPUT VS LINK DELAY:



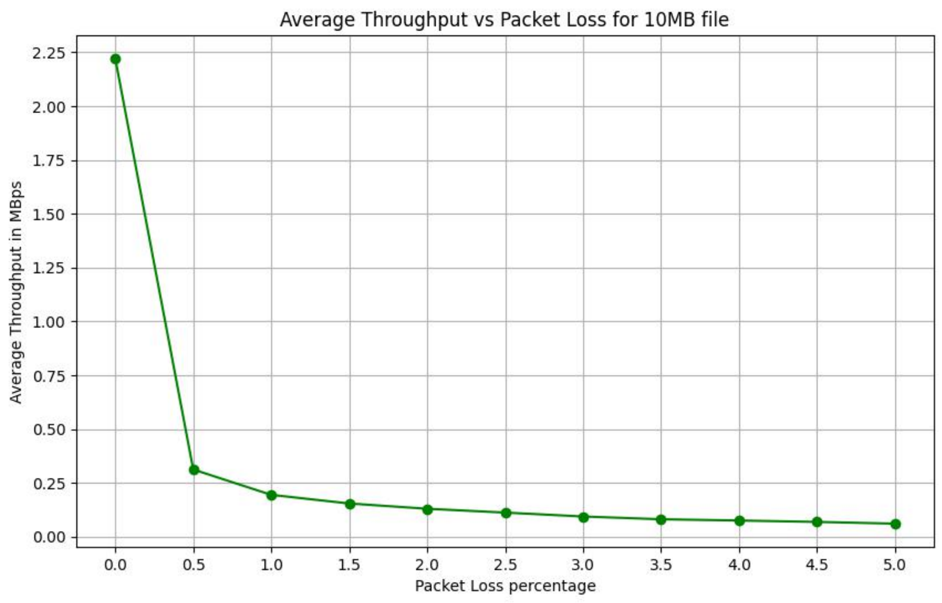
The above graph includes 0ms delay while the below doesn't include 0ms delay.



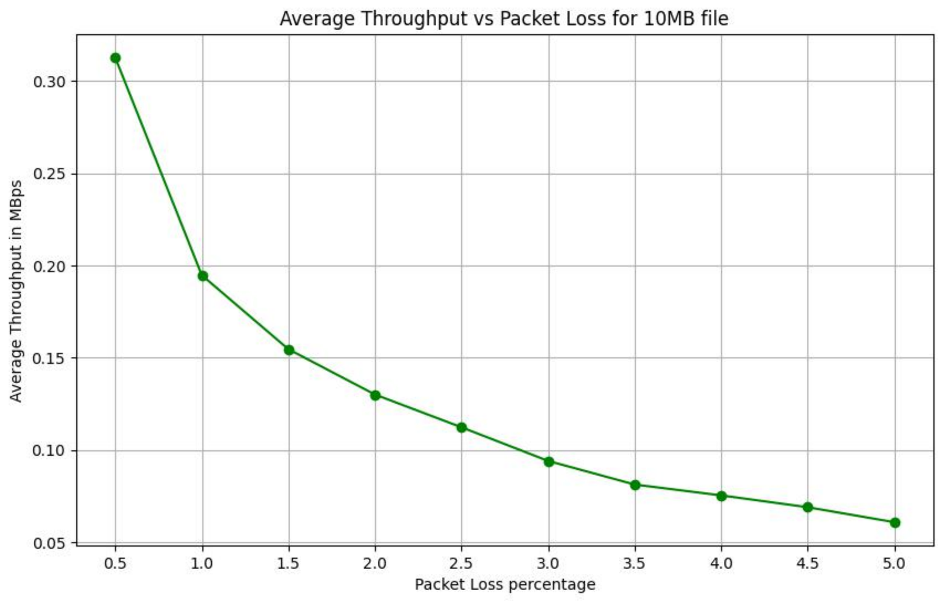
We can observe from the above graph that as delay increases throughput decreases. This is in line with the relation which is present in many TCP related literatures that throughput is inversely related to square root of packet loss.

$$\text{Throughput} \propto \frac{1}{RTT \times \sqrt{p}}$$

THROUGHPUT VS PACKET LOSS:

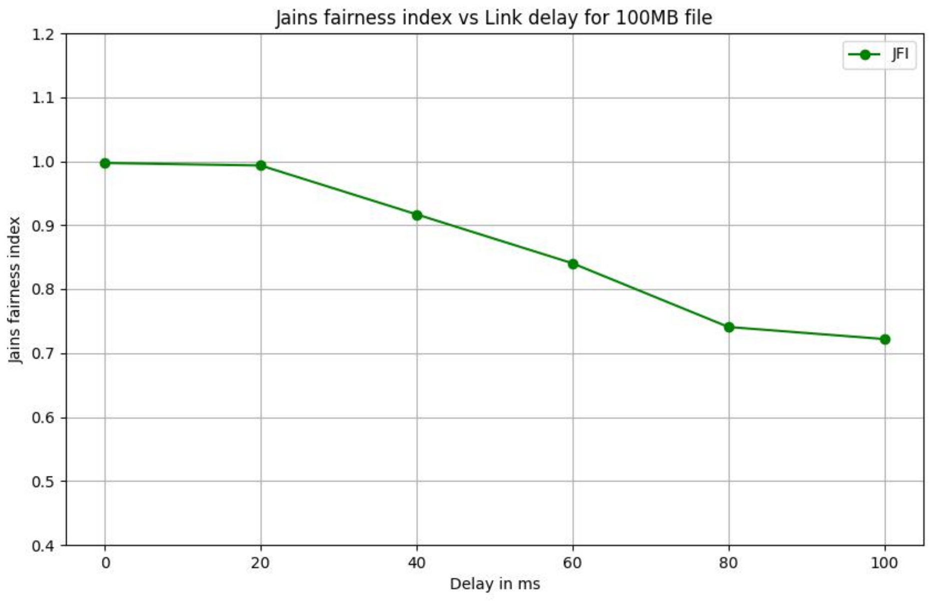


The above graph includes 0% packet loss while the below doesn't include 0% packet loss.



As we can observe from the above graph that as the packet loss percentage increases throughput decreases because we are implementing TCP's reno congestion control algorithm.When packets are lost, the time required for acknowledgment and retransmission adds latency. This delay can reduce the flow of data and impact throughput. So when packet loss occurs througput decreases and further which when packet loss is detected the server decreases the sending rate which further decreases the throughput.

FAIRNESS ANALYSIS:



The graph shows that the JFI (Jain’s Fairness Index) decreases as link delay increases. This decrease is due to the growing delay in the link between s2 and sw2, which exceeds 10 ms, while other link delays remain at 5 ms. As a result, s2 takes longer to receive packets, leading to a reduced fairness index.