COL334

Computer Networks

- Sanyam Garg (2022CS11078) & Aneeket Yadav (2022CS11116)

Part 1 Reliability

1. Protocol Design and Mechanisms

In this section, the design decisions made by us in our implementation have been mentioned. Most of our implementation follows RFC 5681 (TCP Congestion Control) and RFC 6298 (Computing TCP's Retransmission Timer).

1.1 Packet Format

Each data packet consists of:

- sequence_number: Represents the starting byte of the data in the packet.
- data_length: The length of the data in bytes.
- fin: A flag indicating the last packet.
- data: The actual data to be transmitted.

The **fin flag** helps us implement a 3-way handshake in which the server sends the fin flag after sending all the data, which is then acknowledged by the client followed by the final acknowledgement by the server.

1.2 Retransmission timer

The retransmission timer has been implemented according to RFC 2988, according to which $RTO = \max(1, SRTT + 4 \cdot RTTVAR)$

Due to our link delays being small, in most cases the RTO is 1ms. The RTO is doubled on a timeout and deflated back on receiving a new ack. All other mechanisms related to RTO have been implemented as mentioned in RFC 2988. In addition to this a maximum value of 4s has been imposed on the RTO.

As the window size in our implementation far exceeds the limits of TCP (for the case of 200 ms delay, window size for maintaining 50Mbps is 1.25MB, 20 times the limit defined by TCP), handling timeouts and packet losses becomes important and hence the above measures were taken to avoid spurious timeouts and retransmissions while optimizing throughput.

1.3 Window sizes

Window sizes are being set according to the formula: - $wnd = \frac{speed \cdot RTT}{(1\ MSS)}$

1.4 Fast Recovery

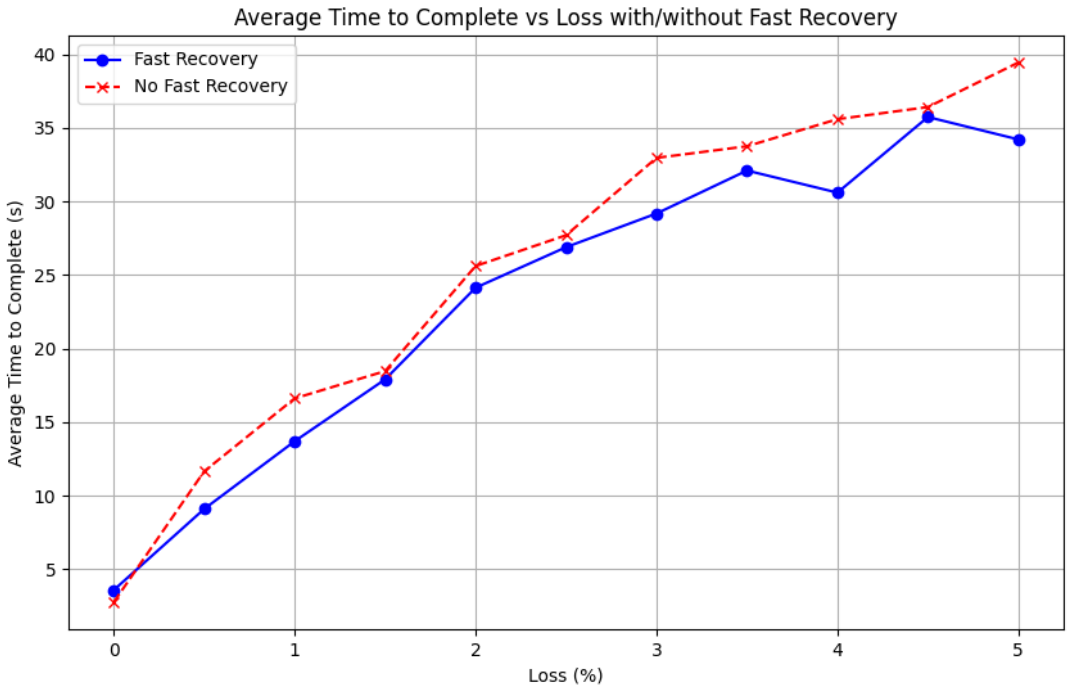
The server enters fast recovery when three duplicate ACKs are received for the same packet. This triggers an immediate retransmission, reducing wait time for a timeout. Fast recovery is particularly helpful when isolated packet losses occur, preventing delays associated with repeated retransmissions of packets that have already been received.

1.5 Buffer on client side

The client has a large buffer to store out of order packets to accommodate for the large window sizes.

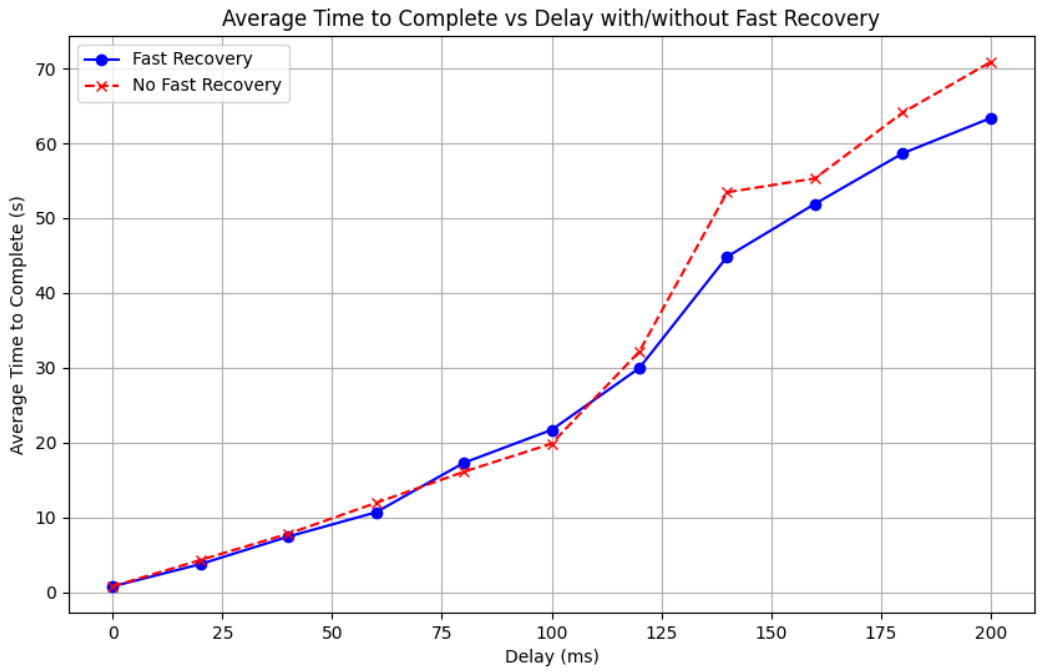
2. Results and Analysis

2.1 Impact of Fast Recovery with Varying Loss



From the graph we can see that fast recovery is beneficial and improves ttc by a small margin, with the margin increasing with loss %. Note that in absolute terms it is not necessary that fast recovery performs better, and the above plot shown has been obtained by removing the noise and averaging over 10 iterations for a 50 MB file. For small files and few iterations, it is possible that fast recovery gives worse results than no fast recovery.

2.2 Impact of Fast Recovery with Varying Delay



From the graph we can see that fast recovery is beneficial and improves ttc by a small margin, with the margin increasing with loss %. Note that in absolute terms it is not necessary that fast recovery performs better (as can be seen in the cases of delay 80ms and 100ms). The above plot shown has been obtained by removing the noise and averaging over 10 iterations for a 50 MB file.

3. Conclusion

The md5 hashes for all the received files match the sent file and hence reliability has been implemented correctly. Fast recovery has yielded slightly better results but does not guarantee better results. A major cause of this is the exceptionally high window sizes used in our assignment.

1. Protocol Design and Mechanisms

In this section, the design decisions made by us in our implementation have been mentioned. Most of our implementation follows RFC 5681 (TCP Congestion Control) and RFC 6298 (Computing TCP's Retransmission Timer).

All of the reliability code in the previous part is intact. The only differences are-

1) Instead of updating window size to maintain the speed, he use cwnd and update it as discussed in class and acc to RFC 5681.
2) Previuosly we sent all unacked packets again on a timeout, now we store which pakcets to resend and send them when the congestio nwindow allows. No new packet is sent until all the unacked packets at previous timeout are resent.
3) Slow start, fast recovery and congestion avoidance have been implemnted in complete compliance with RFC 5681.

Part 2 Congestion Control

Protocol Design and Mechanisms

In this part, the focus shifts from simply ensuring reliable data transfer to actively managing network congestion. While maintaining the reliability mechanisms from Part 1, we now implement additional controls based on RFC 5681 to dynamically adjust the congestion window (cwnd) in response to network conditions. The adjustments made are crucial for optimizing throughput while preventing network overload, which would otherwise lead to excessive packet drops and retransmissions.

Key differences from part-1:

1. **Congestion Window (cwnd) Management**:
    a. **Part 1 Approach**: In Part 1, window sizes were set to achieve a specific speed, regardless of network congestion status.
    b. **Part 2 Approach**: Now, we follow a congestion-aware approach by using a congestion window (cwnd) that adapts based on feedback from the network, aligning with RFC 5681. This adaptive mechanism allows the protocol to maintain efficiency even under changing network conditions, adjusting its sending rate based on congestion signals like packet loss or delay.

This difference is meaningful because cwnd adapts to network capacity, thus preventing oversaturation of the network and reducing packet loss due to congestion.

2. **Timeout-Based Packet Resending**:
    a. **Part 1 Approach**: On a timeout, all unacknowledged packets were retransmitted in bulk. This ensured delivery but could cause a sudden surge in traffic, which might exacerbate congestion.
    b. **Part 2 Approach**: We now implement selective retransmission, storing a list of packets that need to be resent, and sending them as the congestion window (cwnd) permits. This throttled approach prevents excessive retransmissions during congestion, only allowing a controlled number of packets to be resent in each window.

This selective approach is more efficient under congestion, as it allows the protocol to resume transmission without overwhelming the network with redundant data.

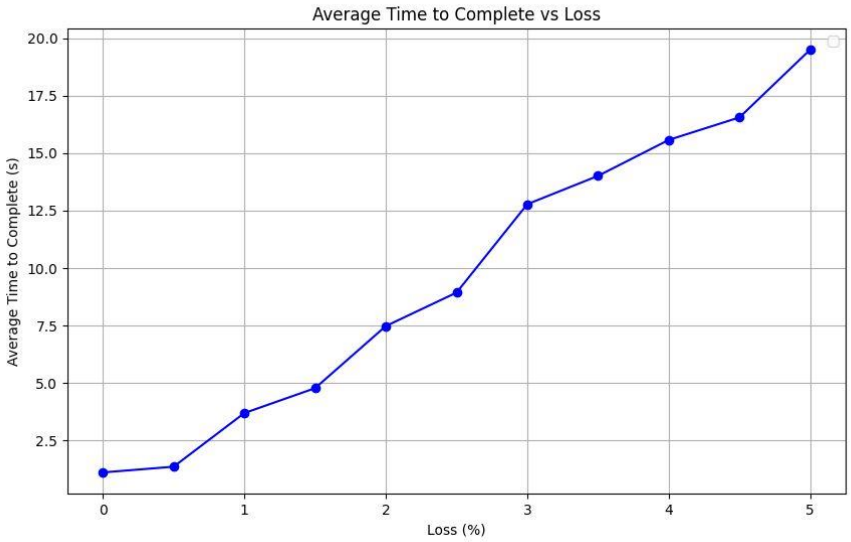
3. **Implementation of Slow Start, Fast Recovery, and Congestion Avoidance**:
    a. **Part 1 Approach**: Fast Recovery was applied when duplicate ACKs indicated packet loss, but other congestion control mechanisms (slow start and congestion avoidance) were not enforced.
    b. **Part 2 Approach**: In compliance with RFC 5681, we now fully implement **slow start, congestion avoidance, and fast recovery** mechanisms:

i. **Slow Start**: Initiates connection at a conservative rate, exponentially increasing cwnd until packet loss or a threshold is reached, minimizing initial congestion risks.
ii. **Congestion Avoidance**: Engages after slow start, gradually increasing cwnd linearly, adapting the rate to the network's capacity to prevent congestion.
iii. **Fast Recovery**: Helps the protocol quickly recover from isolated packet losses by retransmitting lost packets without resorting to the slow start phase, optimizing the time to resume steady data flow.

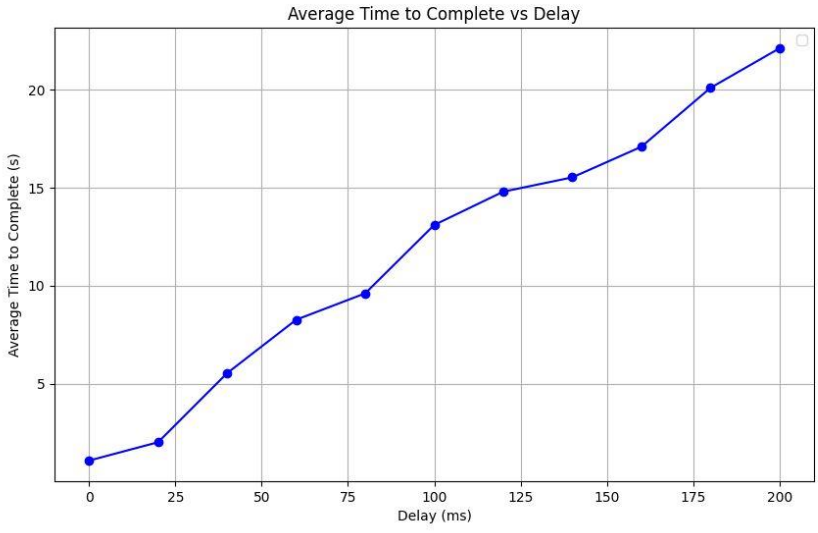
Analysis

Our analysis comprises variation of loss, efficiency and fairness.

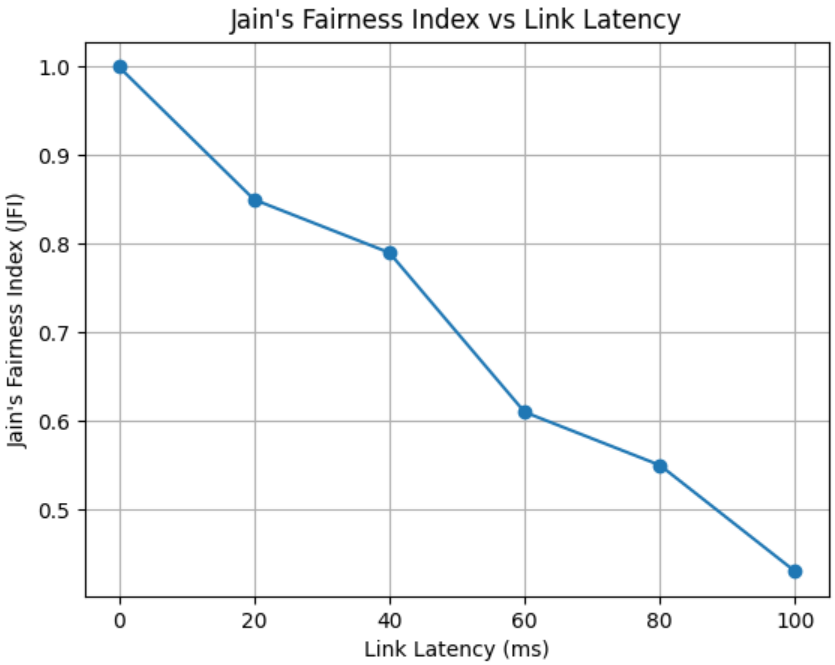
- Average time increases with loss as expected.



- Average time increases with delay as expected.



- Fairness value decreases approximately linearly with increase in delay as expected. However, in networks where RTTs are different for different agents, normalized fairness values are more relevant and meaningful as throughputs are bound to be different due to varying RTTs. Analyzing normalized fairness values, we can again see that TCP Reno is not RTT fair as most values range between 0.8 and 0.9.



Part 3 TCP CUBIC

We have implemented TCP cubic and observed the following -

1) Avg throughput is better than TCP Reno in almost all cases.
2) It is more RTT fair than TCP Reno.
3) As it adapts more quickly, it gives a better performance when compared to TCP Reno in both small and large delay cases.

Below graphs reflect the same -