TCP-like UDP

Entry No.: 2022CS51136 Anubhav Pandey

Entry No.: 2022CS51139 Abhiram Dharme

Contents

Introduction

This assignment aims to implement UDP with TCP-like functionality by adding reliability and congestion control. UDP lacks inherent mechanisms for reliable transmission and congestion management. In this report, we document our approach to developing a client-server application for file transfer, covering aspects of reliability, congestion control, and comparisons between TCP Reno and TCP Cubic as part of the bonus section. Our analysis includes performance observations across various loss and delay conditions.

1 Reliability

The goal of reliability in UDP-based file transfer is to ensure complete, ordered data delivery between server and client, similar to TCP's guarantees. To achieve this, we designed mechanisms for acknowledgment (ACKs), retransmission, fast recovery, and a timeout strategy.

1.1 Mechanisms Implemented

- **Acknowledgments (ACKs)**: Each packet is confirmed by the client using cumulative ACKs. We also explored delayed ACKs to improve performance, but cumulative ACK performed better.
- **Retransmissions and Fast Recovery**: The server resends packets on timeout or upon receiving 3 duplicate ACKs. Fast recovery allows retransmission without waiting for a timeout, crucial for reducing latency under moderate loss.
- **Packet Numbering**: Sequence numbers allow the client to reorder packets.
- **Timeout Calculation**: The server calculates timeout using smoothed RTT (srtt) and deviation (devrtt), improving resilience to network variability. *srtt* and *devrtt* are updated with each new RTT sample.

1.2 Loss Experiment

We conducted experiments varying packet loss from 0% to 5% with gaps of 0.5%, testing with and without fast recovery enabled, each sets ofers 5 times to average out. Results show that transfer time increases with loss rate, and the benefits of fast recovery become more clear as loss rates increase, reducing time significantly by retransmitting based on duplicate ACKs rather than waiting for timeouts.

Figure 1: Time vs Loss Experiment plot with and without fast recovery.

Explaination

With higher packet loss, more packets require retransmission. Fast recovery optimizes this by retransmitting upon receiving duplicate ACKs, preventing the server from waiting for a full timeout, which significantly reduces delays, especially under moderate to high packet loss.

1.3 Delay Experiment

Delays were varied from 0 ms to 200 ms in 20 ms steps, with a fixed 1% packet loss rate. Fast recovery improved the efficiency by reducing delays in retransmission, as the server does not wait for a full timeout. Transfer time increased with delay, as seen in the trend, but the gap between recovery and non-recovery configurations widened with higher delays and this maybe be due to the reason that more no. of ACKs are sent for out-of-order receipts increasing overall time of transmission.

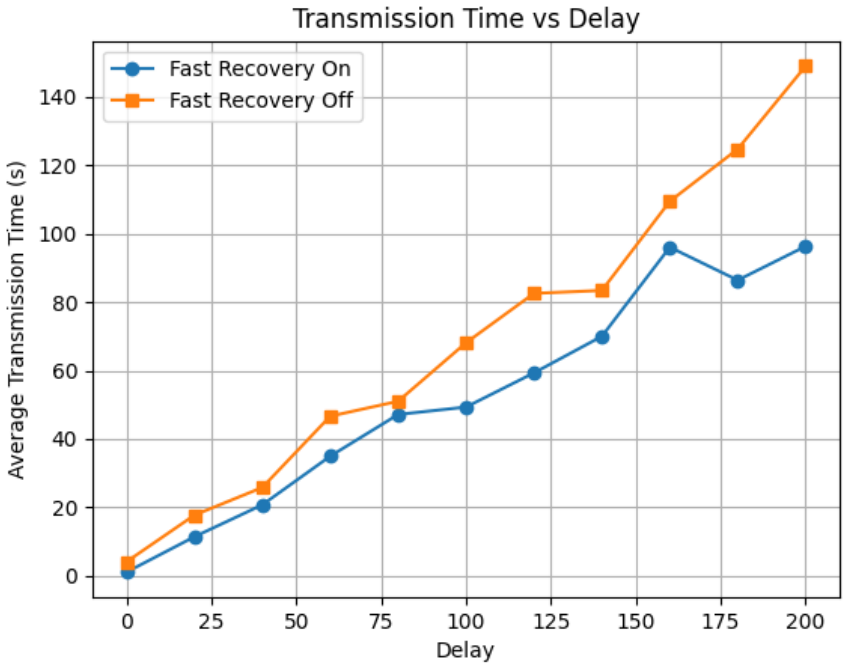

Figure 2: Time vs Delay Experiment plot with and without fast recovery.

Explaination

Higher delay adds to the time it takes for ACKs to reach the server, extending the round-trip time (RTT). Fast recovery again proves beneficial by retransmitting immediately on duplicate ACKs, effectively bypassing the need for a full timeout wait, which becomes more critical under high-delay conditions.

2 Congestion Control

To manage congestion, we implemented a TCP Reno-like algorithm using a sliding window approach to avoid overwhelming the network.

We can see in figure 3 the changes in **cwnd** (Congestion window) and **ssthresh**(Slow Start Threshold), The **Slow Start Phase** where the **cwnd** increases exponentially and then linearly after it. It reduces to **cwnd/2** or 1 MSS in case of congestion and timeouts.

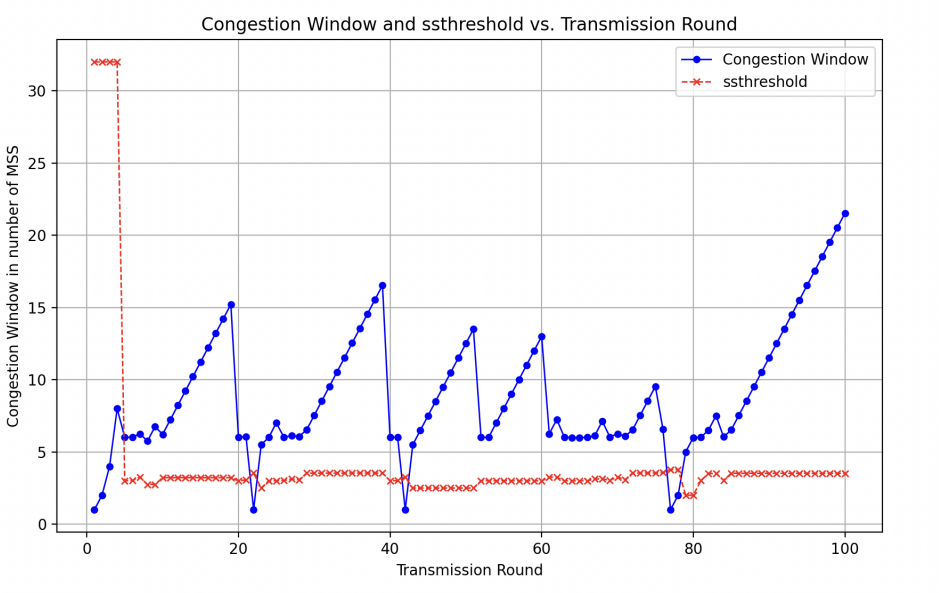

Figure 3: Comparison of cwnd and ssthreshold as a function of transmission round

2.1 Mechanisms Implemented

- **Slow Start and Congestion Avoidance**: The congestion window (cwnd) grows exponentially during slow start until reaching a threshold (ssthresh), after which it grows linearly to avoid congestion.
- **Fast Recovery**: If three duplicate ACKs are received, cwnd is halved, helping the server quickly resume transmission after minor packet losses.
- **Timeout Behavior**: In case of timeout, cwnd resets to 1 MSS, reinitiating slow start.

2.2 Throughput vs Delay Experiment

We repeated throughput analysis for delays ranging from 0 ms to 200 ms with gaps of 20ms and again 5 iterations of each specification. Results show that throughput inversely correlates with delay, consistent with the relationship $throughput \propto \frac{1}{RTT}$ (RTT $\alpha delay$).

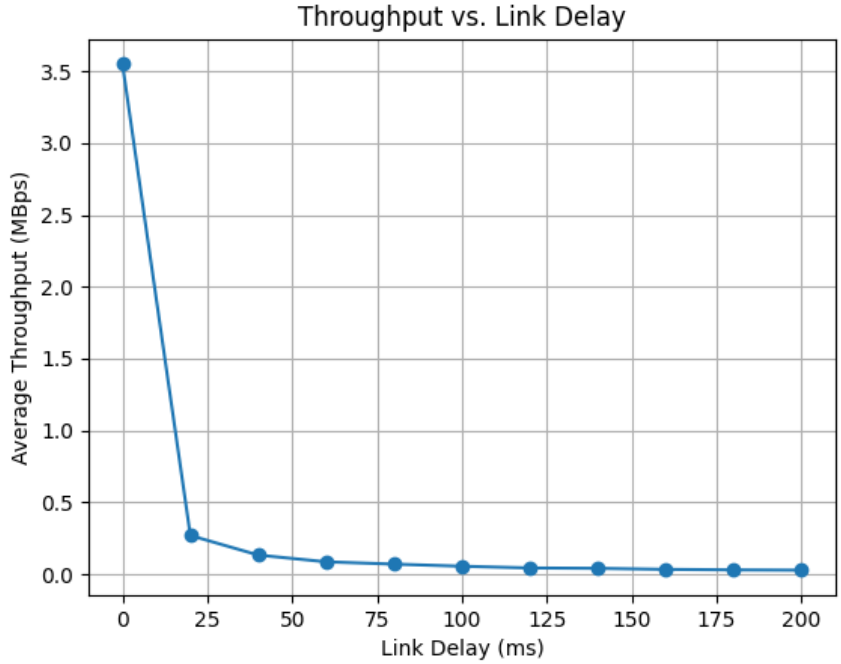

Figure 4: Throughput vs Delay, confirming inverse relation with delay.

Explaination

Higher delays increase the RTT, resulting in slower feedback for congestion control mechanisms, thus lowering throughput. This result aligns with the relationship where throughput is inversely proportional to RTT, making delay a direct factor in reducing throughput.

2.3 Throughput vs Loss Experiment

With packet loss rates from 0% to 5% in 0.5% increments and each loss experiment performed 5 times, our results show that throughput decreases as loss increases. Theoretically, throughput is inversely proportional to the square root of packet loss, which our experiment verified by the given plot analysis. Fast recovery maintained higher throughput under increasing losses by retransmitting immediately upon duplicate ACKs.

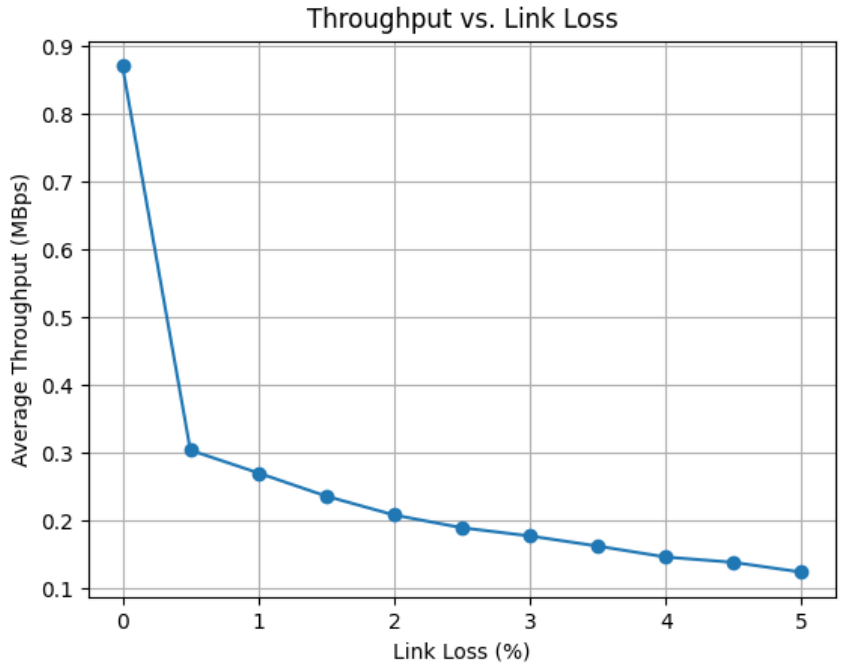

Figure 5: Throughput vs Loss, verifying the theoretical inverse relation.

Explaination

As packet loss rises, more retransmissions are required, which lowers the effective data rate. The TCP throughput formula suggests that throughput is inversely proportional to the square root of loss, consistent with our results. Fast recovery minimizes the time taken for retransmissions, maintaining a higher average throughput under higher loss conditions.

2.4 Fairness Experiment (Dumbbell Topology)

The Figure 6 displays the plot **Fairness** vs **Delay**. Here, the delay of the link between **switch2** and **server2** in Dumbbell Topology is changed from 0ms to 100ms with the gaps of 20ms keeping the other links' delays as 5ms and 5 iterations of each specification. Then the **Jain's Fairness Index** is calculated for all the delays and plotted against the delay of the link. The link between server2 and switch2 requires more time to transfer same amount of data so more time is allocated and hence the fairness index decreases with the increase in delay of the link. The delay for ideal distribution of resources so that Fairness is maximum is 5ms that is the same as all other links.

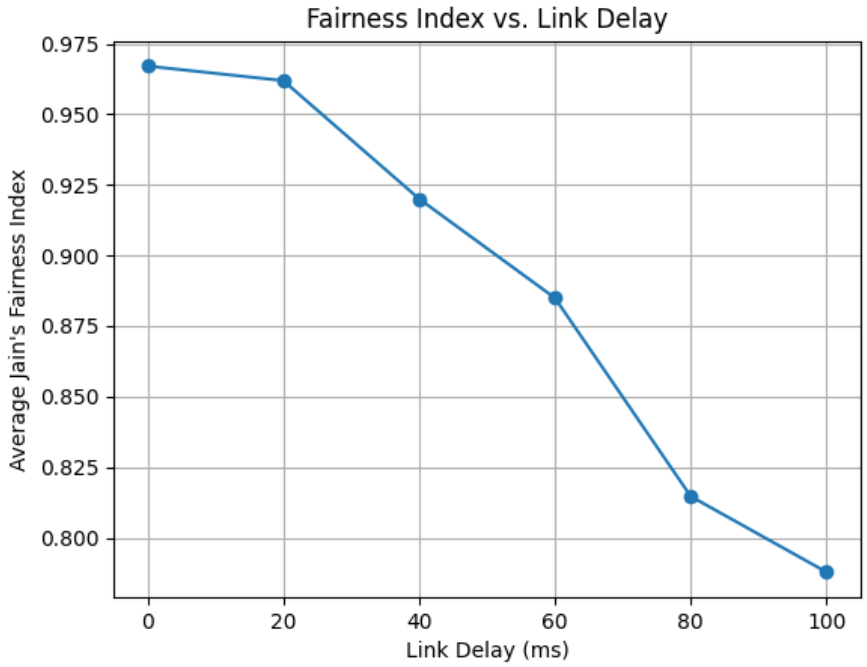

Figure 6: Fairness vs Delay using Jain's Fairness Index.

Explaination

The link with increased delay takes longer to transmit the same amount of data, meaning it receives more time in the transmission schedule, reducing the fairness of data distribution across all links. At an ideal delay (similar across all links), fairness maximizes as resources are more evenly distributed.

3 Bonus: TCP Cubic

In the bonus part, we implemented TCP Cubic, a congestion control algorithm with non-linear window growth based on the function $W(t) = C(t - K)^3 + W_{max}$, where $K = \sqrt[3]{W_{max}\beta/C}$. Parameters $\beta = 0.5$ and $C = 0.4$ balance rapid window expansion with congestion control.

3.1 Efficiency Comparison: TCP Reno vs TCP Cubic

Efficiency experiments show that TCP Cubic adapts better under high-delay, high-loss scenarios due to its cubic growth, which allows faster recovery from congestion events. We repeated the delay and loss experiments for TCP Cubic, observing better throughput at higher delays compared to TCP Reno.

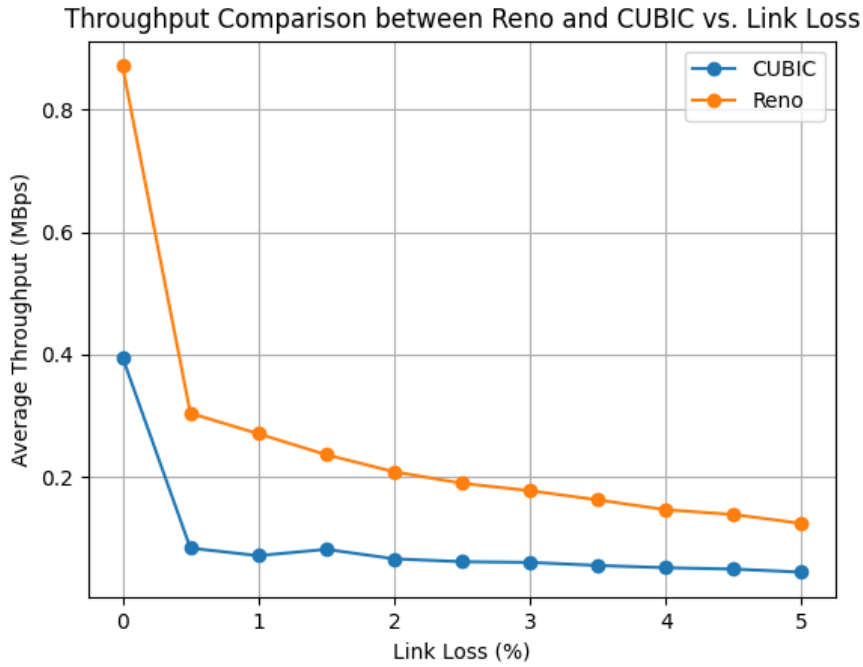

Figure 7: Throughput Comparison: TCP Reno and TCP Cubic vs Loss.

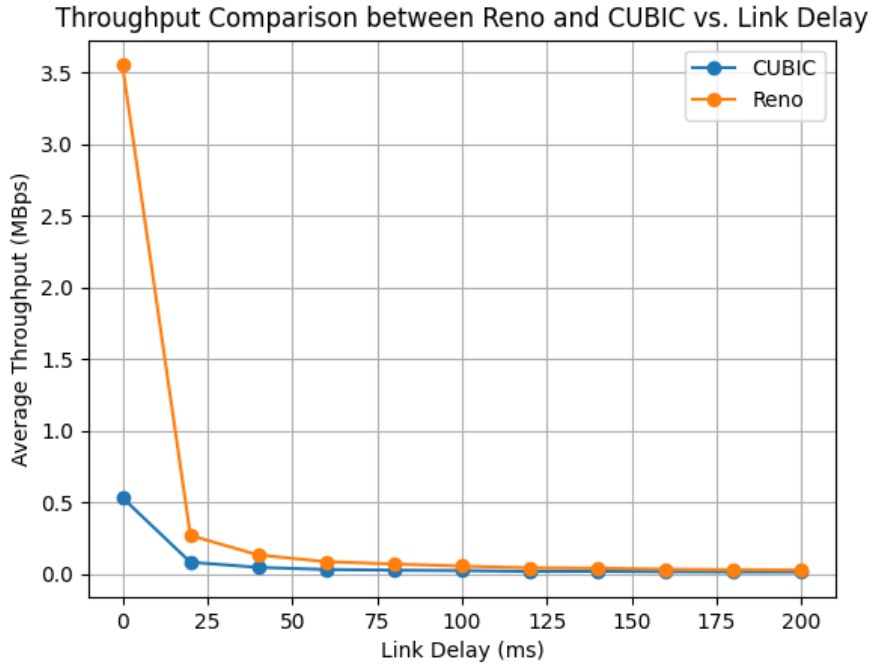

Figure 8: Throughput Comparison: TCP Reno and TCP Cubic vs Delay.

- In high-bandwidth, high-latency (long fat networks): CUBIC is optimized for these environments. Its cubic function allows the congestion window to grow more aggressively and recover quickly after losses, maintaining higher throughput.

- In lower-bandwidth or highly congested networks: TCP Reno might perform comparably or even slightly better than CUBIC, as it's more conservative and less likely to overwhelm limited network resources.
- This same was the case when we tested CUBIC on baadalVM and our personal computer. As baadalVM has more congested network, CUBIC is slower with respect to Reno on it while reverse is the case on our personal computer.

3.2 Fairness Comparison

Using a dumbbell topology with one link running TCP Reno and another running TCP Cubic, we compared fairness at 2 ms and 25 ms delays. Results indicate that TCP Cubic performs very good when delay is low or congestion is less, but as delay increases, Cubic's aggressive growth disrupts fair bandwidth sharing, particularly in high-delay conditions.

4 Conclusion

In this assignment it was demonstrated that UDP can be extended with reliability and congestion control using mechanisms similar to TCP. TCP Reno's linear growth is effective but limited in high-latency networks, while TCP Cubic's cubic growth provides higher efficiency at the expense of fairness in mixed-delay environments.

Assumptions

- **Use of "nohup"**: The use of the "nohup" command is assumed to be necessary to prevent server-client communication process from terminating and we make sure to wait for their procedure to end to flush the ttc, hash, etc. into the csv file correctly. This assumption ensures that processes, particularly long-running tasks, continue executing without interruption adn we wait for them. It is also assumed that "nohup" is available on the system where the commands will be executed or they will wait in each iteration of loops for the server-client transfer to end.
- **System Compatibility**: It is assumed that all scripts are compatible with the Unix/Linux operating system, given the use of commands like "nohup".