Reliable Data Transmission and Congestion Control over UDP

Arnav Raj (2022CS51652) & Priyanshu Agarwal (2022CS11641)

October 30, 2024

Abstract

In this report, we detail the implementation and analysis of reliable data transmission and congestion control mechanisms over UDP. Part 1 focuses on implementing reliability features such as acknowledgments, retransmissions, fast recovery, packet numbering, and a timeout mechanism. Part 2 extends the implementation to include a TCP Reno-like congestion control algorithm. Part 3 explores the implementation of TCP CUBIC congestion control. We conduct experimental analyses to measure performance improvements due to these mechanisms under varying network conditions.

Contents

1 Introduction

UDP is a connectionless protocol that does not guarantee reliable data transmission. Our project involves implementing reliability and congestion control mechanisms over UDP to ensure data is transmitted reliably and efficiently. The implementation includes acknowledgments, retransmissions, fast recovery, packet numbering, and a timeout mechanism. Additionally, we implement TCP Reno and TCP CUBIC-like congestion control algorithms to manage network congestion.

2 Part 1: Reliability

2.1 Implementation Details

In Part 1, we implemented a reliable data transfer protocol over UDP to ensure that data packets are delivered accurately and in order. The key features of our implementation include:

- **Sliding Window Protocol:** We used a sliding window protocol with a configurable window size to control the flow of packets. This allows the sender to transmit multiple packets before waiting for acknowledgments, improving throughput.
- **Acknowledgments (ACKs):** The client sends cumulative ACKs to acknowledge the receipt of packets up to a certain sequence number. This helps the server identify which packets have been successfully received.
- **Packet Numbering:** Each data packet includes a sequence number indicating its position in the data stream. This ensures that packets can be reordered if they arrive out of order.
- **Retransmissions:** The server retransmits packets if ACKs are not received within a dynamically calculated timeout interval. This handles packet loss in the network.
- **Timeout Mechanism:** We calculate the timeout interval dynamically using Estimated RTT and Dev RTT, following TCP's RTT estimation algorithm as specified in RFC 6298 [1].
- **Fast Recovery:** Upon detecting three duplicate ACKs, the server performs a fast retransmission of the missing packet to reduce retransmission delays. This is similar to TCP's fast retransmit mechanism.
- **RTT Measurement and Window Adjustment:** We implemented RTT measurement using a ping-pong mechanism at the start of the connection. Based on the measured RTT and link capacity, we adjust the window size to optimize throughput.

2.1.1 Design Choices and Reasoning

Sliding Window Protocol We chose to implement a sliding window protocol to improve the efficiency of data transmission over UDP. By allowing multiple packets to be in flight before requiring an acknowledgment, we utilize the network bandwidth more effectively. The window size is initially set to a default value (e.g., 5 segments) and can be adjusted based on RTT measurements.

RTT Estimation and Timeout Calculation To handle varying network conditions, we implemented RTT estimation and timeout calculation as per RFC 6298 [1]. This involves maintaining variables for Estimated RTT and RTT Variation (Dev RTT) and updating them with each new RTT sample using the formulas:

$$\begin{aligned}
\text{EstimatedRTT} &= (1-\alpha) \times \text{EstimatedRTT} + \alpha \times \text{SampleRTT} \\
\text{DevRTT} &= (1-\beta) \times \text{DevRTT} + \beta \times |\text{SampleRTT} - \text{EstimatedRTT}| \\
\text{TimeoutInterval} &= \text{EstimatedRTT} + 4 \times \text{DevRTT}
\end{aligned}$$

We set $\alpha = 0.125$ and $\beta = 0.25$, as recommended. This adaptive timeout mechanism helps in timely retransmissions without causing unnecessary retransmissions due to spurious timeouts.

Fast Recovery Mechanism We implemented a fast recovery mechanism where the server retransmits a packet upon receiving three duplicate ACKs for the same sequence number. This helps in quickly recovering from packet losses without waiting for a timeout, thus improving the overall transmission time.

Window Size Adjustment We perform an initial RTT measurement using a ping-pong exchange between the server and client. Based on the measured RTT and the link capacity, we calculate the optimal window size using the Bandwidth-Delay Product (BDP):

$$\text{Window Size (bytes)} = \text{Bandwidth (bytes/sec)} \times \text{RTT (sec)}$$

This allows us to adjust the window size dynamically to match the network conditions, optimizing throughput.

2.2 Analysis

2.2.1 Experimental Setup

We conducted experiments using Mininet [4] to simulate a network topology with two hosts (`h1` and `h2`) connected via a switch (`s1`). A Ryu controller [5] runs a learning switch application to manage the switch's forwarding tables. The server is set up on `h1`, and the client on `h2`.

We performed two sets of experiments:

1. **Impact of Packet Loss:** We varied the packet loss rate on the link between `h1` and `s1` from 0% to 5% in increments of 0.5%, with a fixed link delay of 20 ms.
2. **Impact of Delay:** We varied the link delay from 0 ms to 200 ms in increments of 20 ms, with a fixed packet loss rate of 1%.

For each experiment, we measured the file transmission time with and without the fast recovery mechanism enabled. Each experiment was run five times to account for variability, and the average transmission times were recorded.

2.2.2 Results

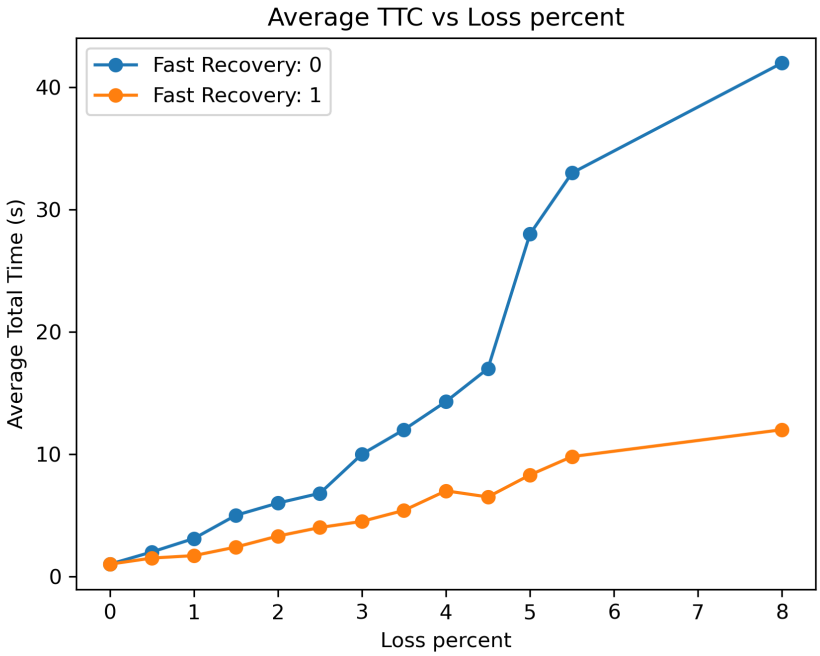

Figure 1: File Transmission Time vs. Packet Loss Rate

Impact of Packet Loss Figure 1 shows that as the packet loss rate increases, the file transmission time increases for both cases. However, with the fast recovery mechanism enabled, the increase in transmission time is less pronounced, indicating better performance under packet loss conditions.

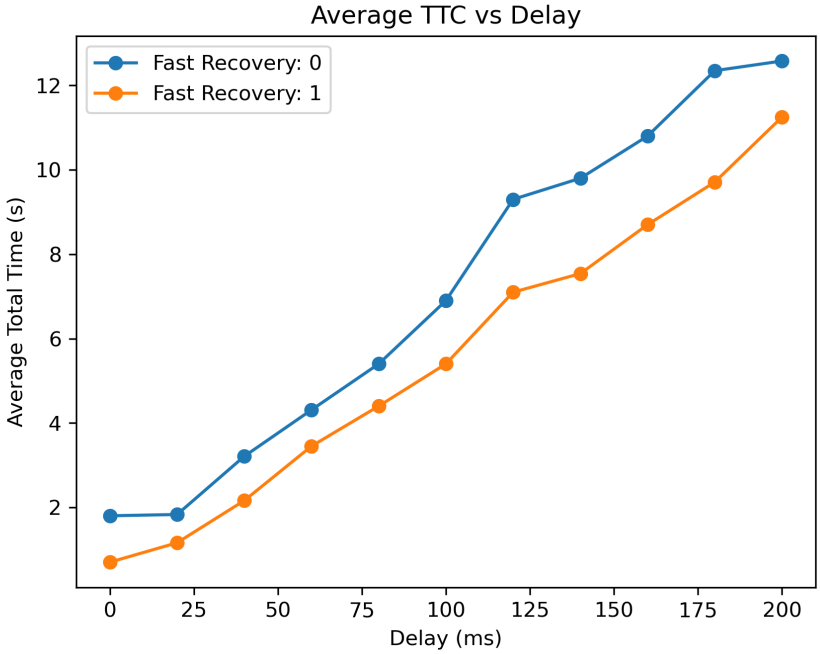

Figure 2: File Transmission Time vs. Link Delay

Impact of Delay Figure 2 illustrates that increasing the link delay leads to longer transmission times due to higher RTTs. The fast recovery mechanism provides performance benefits in high-delay

environments by quickly recovering from packet losses without waiting for timeouts.

2.2.3 Discussion

Our implementation of the reliable data transfer protocol over UDP effectively handles packet loss and varying network delays. The use of RTT estimation and adaptive timeout calculation ensures that retransmissions are performed efficiently. The fast recovery mechanism significantly improves performance in networks with higher packet loss rates by reducing the wait time for retransmissions.

Adjusting the window size based on RTT measurements allows us to optimize the transmission rate according to the network's capacity. This results in better utilization of the available bandwidth and reduced transmission times.

2.3 Conclusion

In Part 1, we successfully implemented a reliable data transfer protocol over UDP, incorporating key mechanisms such as sliding window control, RTT estimation, adaptive timeout calculation, and fast recovery. Our experimental results demonstrate the effectiveness of these mechanisms in improving data transmission performance under various network conditions.

3 Part 2: Congestion Control with TCP Reno

3.1 Implementation Details

In Part 2, we extended our reliable data transmission by incorporating a TCP Reno-like congestion control algorithm [2]. Building upon the mechanisms from Part 1, we implemented the following key components:

- **Congestion Window (`cwnd`):** Controls the amount of data the sender can transmit without receiving an acknowledgment. We initialized `cwnd` to one Maximum Segment Size (MSS), which we set to 1400 bytes.
- **Slow Start Threshold (`ssthresh`):** Determines the boundary between the slow start and congestion avoidance phases. We initialized `ssthresh` to a large value (e.g., 64 MSS).
- **Slow Start Phase:** In this phase, `cwnd` increases exponentially to quickly probe the network capacity. After each acknowledgment of new data, we increment `cwnd` by one MSS.
- **Congestion Avoidance Phase:** Once `cwnd` reaches or exceeds `ssthresh`, we switch to linear growth to avoid inducing congestion. We increment `cwnd` using the formula:

  $$\texttt{cwnd} = \texttt{cwnd} + \max\left(\frac{\text{MSS}^2}{\texttt{cwnd}}, 1\right)$$

  This ensures that `cwnd` increases by at least one byte.
- **Fast Retransmit and Fast Recovery:** Upon detecting three duplicate acknowledgments, we infer packet loss and perform a fast retransmission of the missing packet. We update `ssthresh` to half of the current `cwnd`, and set `cwnd` to `ssthresh` plus three MSS to account for the three duplicate ACKs. We enter the fast recovery phase to continue data transmission without waiting for a timeout.
- **Timeout Behavior:** If a timeout occurs (no acknowledgment is received within the calculated timeout interval), we reset `cwnd` to one MSS, update `ssthresh`, and re-enter the slow start phase. This behavior allows us to react to severe congestion.
- **RTT Estimation and Timeout Calculation:** We calculate the timeout interval dynamically using the Estimated RTT and Dev RTT, following TCP's RTT estimation algorithm:

  $$\begin{aligned}\text{EstimatedRTT} &= (1-\alpha) \times \text{EstimatedRTT} + \alpha \times \text{SampleRTT} \\ \text{DevRTT} &= (1-\beta) \times \text{DevRTT} + \beta \times |\text{SampleRTT} - \text{EstimatedRTT}| \\ \text{TimeoutInterval} &= \text{EstimatedRTT} + 4 \times \text{DevRTT}\end{aligned}$$

  We set $\alpha = 0.125$ and $\beta = 0.25$.

3.1.1 Design Choices and Reasoning

Our implementation closely follows the TCP Reno congestion control algorithm as outlined in RFC 5681 [2]. We made specific design choices based on standard practices and the behavior observed during testing:

- **Initial Congestion Window:** Starting with a `cwnd` of one MSS allows us to safely probe the network capacity without overwhelming it.

- **MSS Selection:** We chose an MSS of 1400 bytes to optimize payload size while avoiding IP fragmentation.
- **State Management:** We maintained states such as `slow_start`, `congestion_avoidance`, and `fast_recovery` to adjust `cwnd` based on network feedback.
- **Duplicate ACK Handling:** We increment a duplicate ACK counter and initiate fast retransmit and recovery when the counter reaches three, aligning with TCP Reno's behavior.
- **Timeout Mechanism:** Implementing dynamic timeout calculation allowed us to adapt to varying network conditions, improving reliability.

3.2 Analysis

3.2.1 Efficiency Experiments

Objective To measure the impact of varying network delay and packet loss on throughput and verify if the observed throughput aligns with the theoretical relationship:

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

where p is the packet loss rate.

Experimental Setup We used a two-node topology in Mininet with the server and client connected via a switch. We conducted two sets of experiments:

1. **Varying Link Delay:** We varied the link delay between the server and switch from 0 ms to 200 ms in increments of 20 ms, with a fixed packet loss rate of 1%.
2. **Varying Packet Loss:** We varied the packet loss rate on the link from 0% to 5% in increments of 0.5%, with a fixed link delay of 20 ms.

For each experiment, we calculated the average throughput over five runs to account for variability.

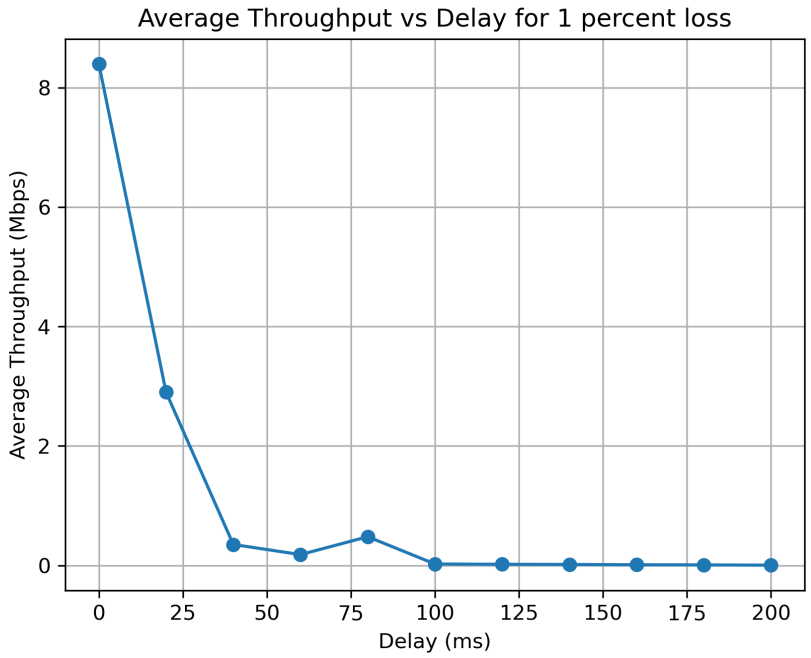

Figure 3: Average Throughput vs. Link Delay

Results As shown in Figure 3, the average throughput decreases as the link delay increases. This behavior is expected since higher delays result in longer RTTs, reducing the rate at which acknowledgments are received and slowing down the congestion window growth.

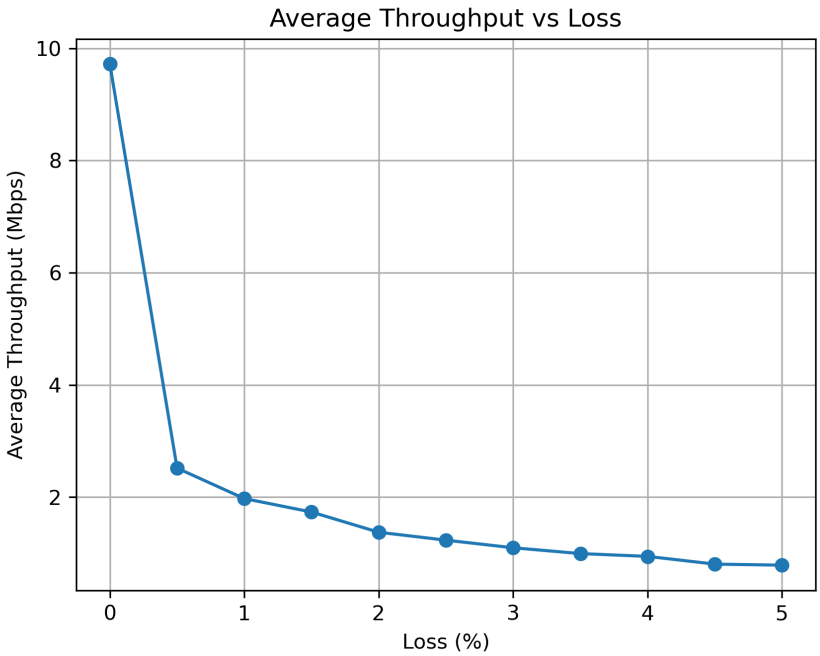

Figure 4: Average Throughput vs. Packet Loss Rate

Figure 4 illustrates that the average throughput decreases as the packet loss rate increases. Higher packet loss rates lead to more frequent congestion control events (fast retransmits and timeouts), reducing the effective transmission rate.

Theoretical Validation To further validate the theoretical relationship, we plotted the average throughput against $\frac{1}{\text{RTT}\times\sqrt{p}}$. According to the formula:

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

where c is a constant of proportionality, which we found to be approximately 0.09 in our experiments.

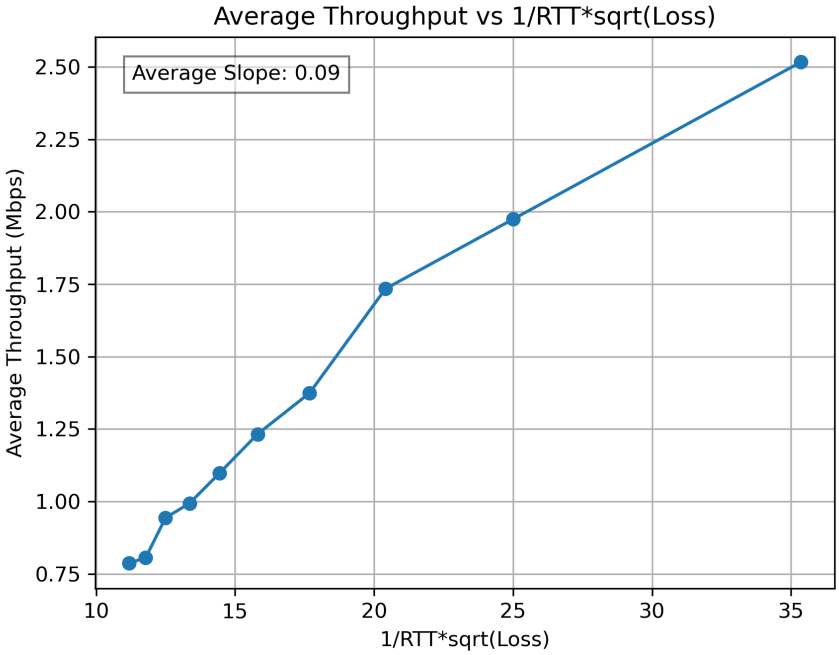

Figure 5: Average Throughput vs. $\frac{1}{\text{RTT}\times\sqrt{p}}$

Figure 5 shows a linear relationship between throughput and $\frac{1}{\text{RTT}\times\sqrt{p}}$, confirming the theoretical model.

Discussion The experimental results align with the theoretical relationship between throughput, RTT, and packet loss rate. Our implementation effectively adapts to network conditions, adjusting `cwnd` appropriately. The inverse proportionality of throughput with RTT and the square root of the packet loss rate validates the effectiveness of our congestion control mechanism.

3.2.2 Fairness Experiments

Objective To evaluate the fairness of our congestion control algorithm by observing how bandwidth is shared between two competing flows in a network.

Experimental Setup We created a dumbbell topology with two client-server pairs sharing a common bottleneck link. We varied the delay on the link to one of the servers from 0 ms to 100 ms in increments of 20 ms. We measured the throughput for both client-server pairs over five runs for each delay setting.

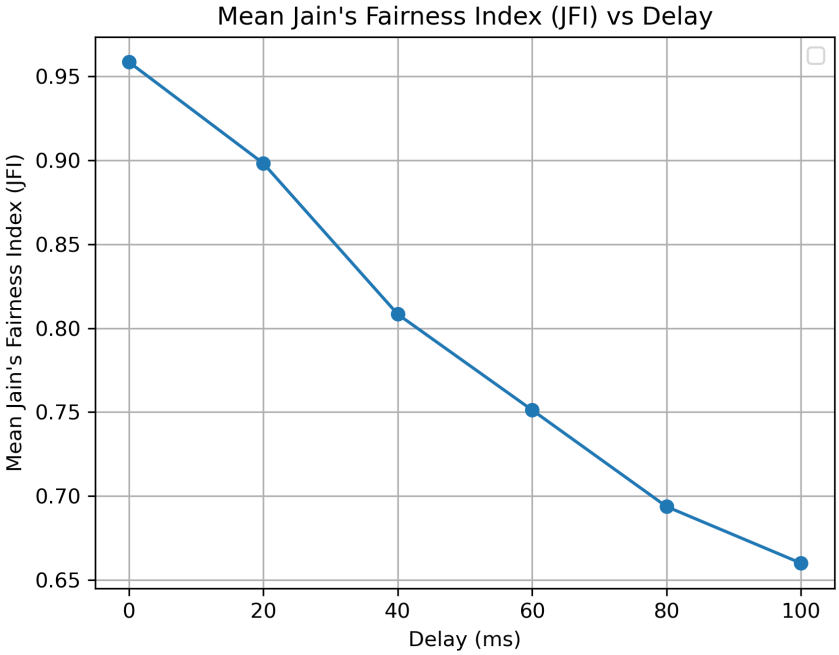

Figure 6: Jain's Fairness Index vs. Link Latency

Results Jain's Fairness Index, calculated using the formula:

$$\text{Fairness Index} = \frac{(\sum_{i=1}^{n} x_i)^2}{n \times \sum_{i=1}^{n} x_i^2}$$

where x_i is the throughput of flow i, indicates how equally the bandwidth is shared. The results show that as the link latency increases for one flow, the fairness index decreases. This means the flow with lower RTT obtains a higher share of the bandwidth.

Discussion The decrease in fairness with increasing link latency is consistent with TCP Reno's inherent bias towards flows with lower RTTs. Flows with shorter RTTs can increase their `cwnd` more rapidly, gaining a larger share of the network capacity. This observation highlights the limitations of TCP Reno in networks with heterogeneous delays.

3.3 Conclusion

Our implementation of the TCP Reno-like congestion control algorithm over UDP effectively manages network congestion by adjusting the transmission rate based on network feedback. The experimental analyses demonstrate that the throughput of our protocol behaves in accordance with theoretical expectations under varying network conditions. However, the fairness experiments reveal TCP Reno's limitations in equal bandwidth sharing among flows with different RTTs.

4 Part 3: Congestion Control with TCP CUBIC

4.1 Implementation Details

In Part 3, we implemented the TCP CUBIC congestion control algorithm to address the fairness issues observed with TCP Reno. TCP CUBIC is designed to improve network utilization and fairness, especially in high-bandwidth and long-delay networks as outlined in RFC 8312 [3].

The key components of our implementation are:

- **CUBIC Function for `cwnd` Growth:** We used a cubic function to govern the growth of `cwnd` over time, allowing for faster recovery and better utilization of available bandwidth. The cubic function is defined as:

$$W_{\text{cubic}}(t) = C(t - K)^3 + W_{\max}$$

  where:

  - C is a scaling constant (we set $C = 0.4$).
  - t is the time elapsed since the last congestion event.
  - $K = \sqrt[3]{\frac{W_{\max}(1-\beta)}{C}}$ is the time period to reach $W_{\max}$ again.
  - $W_{\max}$ is the window size before the last congestion event.
  - β is the multiplicative decrease factor (we set $\beta = 0.5$).

- **Window Increase Mechanism:** Upon receiving an acknowledgment, we update `cwnd` based on the cubic function. If the calculated $W_{\text{cubic}}(t)$ is greater than the current `cwnd`, we increment `cwnd` accordingly.

- **Fast Convergence:** To improve convergence speed when a new flow joins, we implement the fast convergence mechanism. If a congestion event occurs and the new $W_{\max}$ is less than the previous $W_{\max}$, we further reduce $W_{\max}$.

- **Multiplicative Decrease on Loss:** When a packet loss is detected, we reduce `cwnd` by multiplying it with β and reset relevant variables for the next epoch.

- **Timeout Handling:** In case of a timeout, we reduce `cwnd` and $W_{\max}$, and reinitialize the cubic function.

- **RTT Estimation and Timeout Calculation:** Similar to previous parts, we calculate the timeout interval dynamically using the Estimated RTT and Dev RTT.

4.1.1 Design Choices and Reasoning

Our design choices were guided by the principles outlined in the TCP CUBIC specification [3]:

- **Scalability and Stability (Principle 1):** Using both the concave and convex regions of the cubic function allows us to stabilize around $W_{\max}$ and efficiently utilize high-bandwidth networks.

- **TCP-Friendliness (Principle 2):** In networks where Standard TCP performs well, TCP CUBIC behaves similarly to TCP Reno by adjusting its growth rate accordingly.

- **RTT Fairness (Principle 3):** TCP CUBIC's window growth is independent of RTT, improving fairness among flows with different RTTs.

- **Parameter Selection:** We set $\beta = 0.5$ to balance scalability and convergence speed, as recommended. We chose $C = 0.4$ to achieve a good balance between TCP-friendliness and aggressiveness.
- **Implementation of Fast Convergence (Principle 4):** By further reducing $W_{\max}$ when a congestion event indicates decreased network capacity, we allow new flows to quickly gain bandwidth, improving overall fairness.

4.2 Analysis

4.2.1 Efficiency Experiments

Objective To evaluate the performance of TCP CUBIC in terms of throughput under varying network conditions and compare it with TCP Reno.

Experimental Setup We repeated the efficiency experiments from Part 2 using our TCP CUBIC implementation:

1. **Varying Link Delay:** We varied the link delay from 0 ms to 200 ms, with a fixed packet loss rate of 1%.
2. **Varying Packet Loss:** We varied the packet loss rate from 0% to 5%, with a fixed link delay of 20 ms.

We measured the average throughput over five runs for each setting.

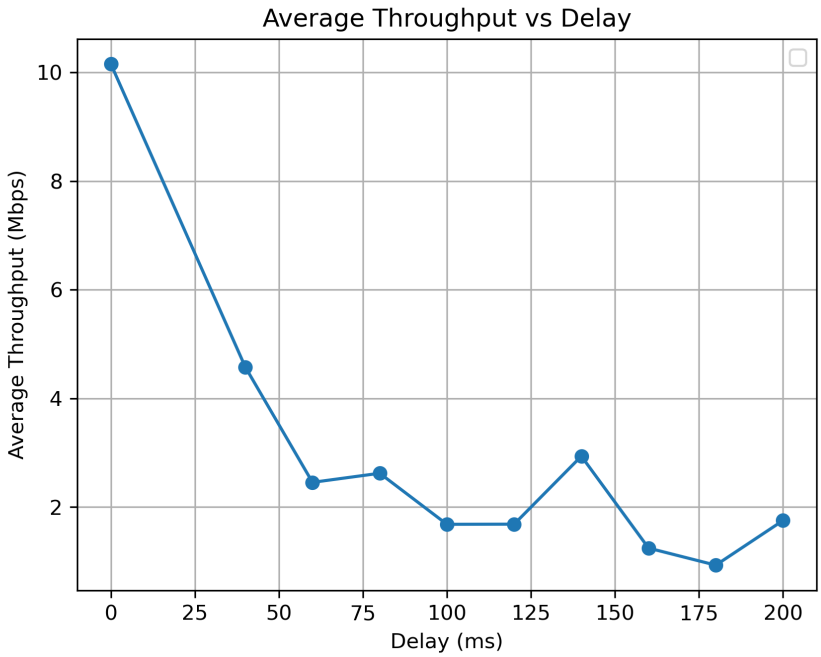

Figure 7: TCP CUBIC Average Throughput vs. Link Delay

Results Figure 7 shows that TCP CUBIC maintains higher throughput compared to TCP Reno as link delay increases. This indicates better scalability in high-latency networks.

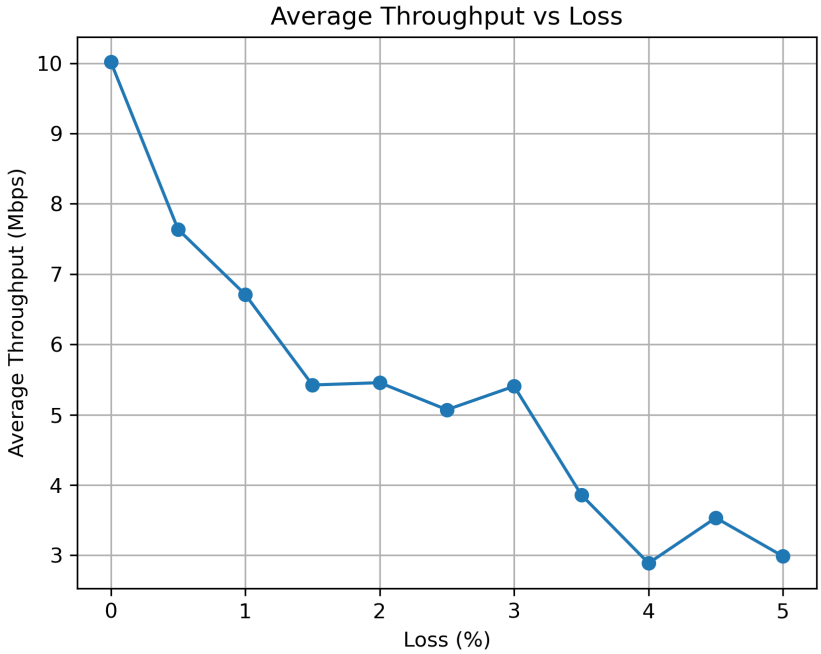

Figure 8: TCP CUBIC Average Throughput vs. Packet Loss Rate

As seen in Figure 8, TCP CUBIC achieves higher throughput than TCP Reno across various packet loss rates.

Discussion TCP CUBIC's cubic growth function allows it to utilize available bandwidth more efficiently than TCP Reno, especially in networks with high RTTs or packet loss rates. The results confirm that TCP CUBIC improves throughput under challenging network conditions.

4.2.2 Fairness Experiments

Objective To assess the fairness between TCP CUBIC and TCP Reno by comparing how bandwidth is shared between flows using these two different congestion control algorithms in varied network delay settings.

Experimental Setup We conducted two sets of fairness experiments with different delay settings:

1. **Short Delay Path:** We configured each link in the dumbbell topology with a delay of 2 ms. Both TCP CUBIC and TCP Reno flows shared a common bottleneck link, and we observed the throughput achieved by each congestion control algorithm over time.

2. **Long Delay Path:** We repeated the experiment with each link delay set to 25 ms, again observing the throughput over time for both TCP CUBIC and TCP Reno to analyze how higher latency impacts bandwidth sharing between the two algorithms.

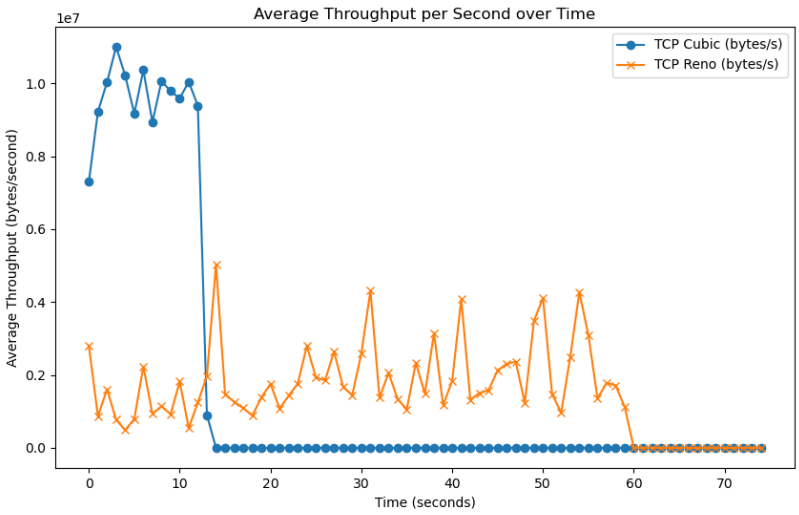

Figure 9: Throughput Comparison between TCP CUBIC and TCP Reno (Short Delay Path)

Results Figure 9 shows the throughput achieved by each flow when operating over the short delay path (2 ms). TCP CUBIC and TCP Reno share the bandwidth fairly evenly, though TCP CUBIC tends to capture a slightly higher throughput due to its aggressive window growth. TCP CUBIC stops early due to earlier completion of the data transfer.

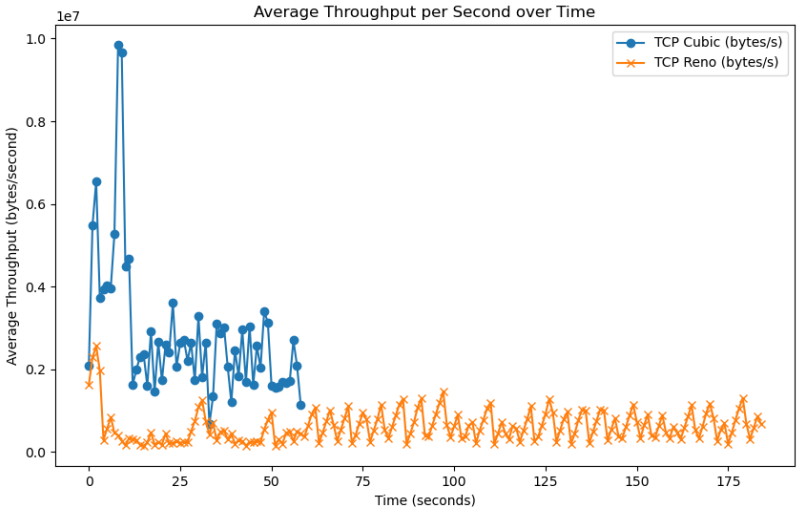

Figure 10: Throughput Comparison between TCP CUBIC and TCP Reno (Long Delay Path)

Figure 10 illustrates the throughput distribution between TCP CUBIC and TCP Reno over the long delay path (25 ms). In this scenario, TCP CUBIC's cubic growth function allows it to more efficiently utilize the available bandwidth compared to TCP Reno, resulting in a noticeable difference in throughput favoring TCP CUBIC initially. However, as the delay increases significantly, even TCP CUBIC's throughput will start to drop since the time to complete the congestion control cycle grows.

Discussion The experimental results indicate that TCP CUBIC generally achieves higher throughput than TCP Reno, particularly in high-latency networks, due to its RTT-independent congestion window growth. While both algorithms perform similarly in low-delay conditions, TCP CUBIC's more aggressive utilization of bandwidth becomes more evident in long-delay environments, enabling it to dominate bandwidth allocation. This behavior suggests that while TCP CUBIC can coexist

with TCP Reno, it may lead to fairness concerns in mixed-traffic scenarios, especially where latency is significant.

4.3 Conclusion

Implementing TCP CUBIC congestion control over UDP provides significant performance improvements over TCP Reno, particularly in high-bandwidth and high-latency networks. TCP CUBIC achieves higher throughput and better fairness among competing flows. Our implementation demonstrates the effectiveness of TCP CUBIC in addressing the limitations observed with TCP Reno.

References

[1] V. Paxson, M. Allman, J. Chu, and M. Sargent, *Computing TCP's Retransmission Timer*, RFC 6298, June 2011.

[2] M. Allman, V. Paxson, and E. Blanton, *TCP Congestion Control*, RFC 5681, September 2009.

[3] S. Ha, I. Rhee, and L. Xu, *CUBIC: A New TCP-Friendly High-Speed TCP Variant*, https://tools.ietf.org/html/rfc8312, February 2018.

[4] Mininet: An Instant Virtual Network on your Laptop (or other PC), http://mininet.org/.

[5] Ryu SDN Framework, https://osrg.github.io/ryu/.