Computer Networks

Student Name : Sahil Khan

Question 1: Ensuring Reliable Data Transmission over UDP

1 Introduction

In this Quesetion we implement reliability over the User Datagram Protocol (UDP), which inherently lacks features such as acknowledgments (ACKs), retransmissions, and packet sequencing. This implementation simulates key TCP mechanisms like cumulative ACKs, retransmissions, fast recovery, and adaptive timeout mechanisms to ensure reliable data transmission between a server and client using UDP. This report presents the structure of our solution and the experimental setup to measure performance improvements due to these mechanisms.

2 Reliability Mechanisms

2.1 Acknowledgments (ACKs)

We use cumulative ACKs at the client side to acknowledge the receipt of packets up to a particular sequence number. This approach reduces the number of ACKs and enhances efficiency. Additionally, delayed ACKs were implemented with a delay interval of 0.2 seconds, which further optimizes acknowledgment traffic under normal network conditions.

2.2 Retransmissions

The server implements a retransmission mechanism triggered by:

- A timeout on unacknowledged packets, calculated using an adaptive timeout mechanism based on round-trip time (RTT) estimates.
- Detection of 3 duplicate ACKs from the client, signaling the loss of a packet.

The retransmission mechanism ensures packets are resent until an ACK confirms successful delivery.

2.3 Fast Recovery

To reduce downtime due to packet loss, the server enters a fast recovery mode upon receiving 3 duplicate ACKs. This mechanism retransmits the missing packet without waiting for a timeout, significantly reducing the time required to recover from intermittent losses.

2.4 Packet Numbering and Sequencing

Each packet contains a unique sequence number, which allows the client to reorder packets as necessary and maintain data integrity. The sequence numbering begins from zero and increments according to the size of the data in each packet.

2.5 Timeout Mechanism

The server dynamically calculates an adaptive timeout interval based on RTT, using an exponential weighted moving average:

$$\text{Timeout Interval} = \text{Estimated RTT} + 4 \times \text{Dev RTT} \tag{1}$$

This adaptive mechanism allows the server to adjust to changing network conditions, reducing unnecessary retransmissions due to fluctuating RTTs.

3 Packet Format

To facilitate debugging and transmission, the packet is formatted using JSON with the following fields:

- `sequence_number`: Indicates the byte offset of the packet.
- `data_length`: Specifies the length of the data in bytes.
- `data`: The actual data, encoded in Base64 format for compatibility.

The ACK packet format includes only the `next_sequence_number` field, indicating the byte offset up to which data has been successfully received.

4 Experimental Setup

4.1 Topology

The experiments were conducted using Mininet with a simple two-node topology. The topology consists of two hosts (h1 and h2) connected through a switch (s1) controlled by a Ryu learning switch controller. The topology allows testing of the reliability mechanisms under controlled network conditions.

4.2 Experimental Parameters

To evaluate the performance improvements due to the fast recovery mechanism, we conducted two sets of experiments:

4.2.1 Loss Variation Experiment

We fixed the delay on the `h1-s1` link to 20 ms and varied the packet loss rate from 0% to 5% in increments of 0.5%. The `h2-s2` link was configured with 0% loss and 0 ms delay. We measured the time taken to download the file in both configurations (with and without fast recovery).

4.2.2 Delay Variation Experiment

We fixed the packet loss rate on the `h1-s1` link at 1% and varied the delay from 0 ms to 200 ms in increments of 20 ms. The `h2-s2` link was configured with 0% loss and 0 ms delay. As with the previous experiment, we recorded the file transfer time for both configurations (with and without fast recovery).

4.3 Data Collection

Each experiment was repeated 5 times to account for noise and provide average results. The performance was measured in terms of the file transfer completion time, focusing on the effectiveness of fast recovery under varying network conditions.

5 Results

5.1 Loss Variation Results

The following graph illustrates the effect of varying packet loss rates on file transfer time with and without fast recovery enabled.

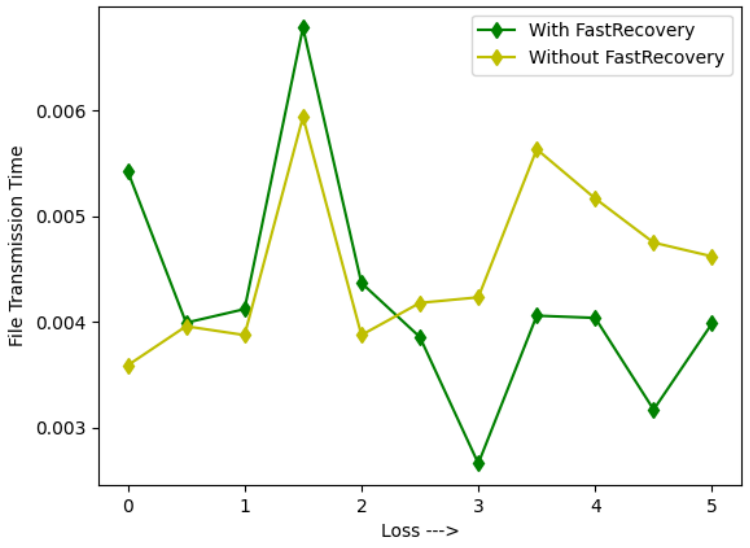

Figure 1: File Transfer Time vs. Packet Loss Rate (With and Without Fast Recovery)

5.2 Delay Variation Results

The graph below demonstrates the impact of varying link delay on file transfer time, comparing the configurations with and without fast recovery.

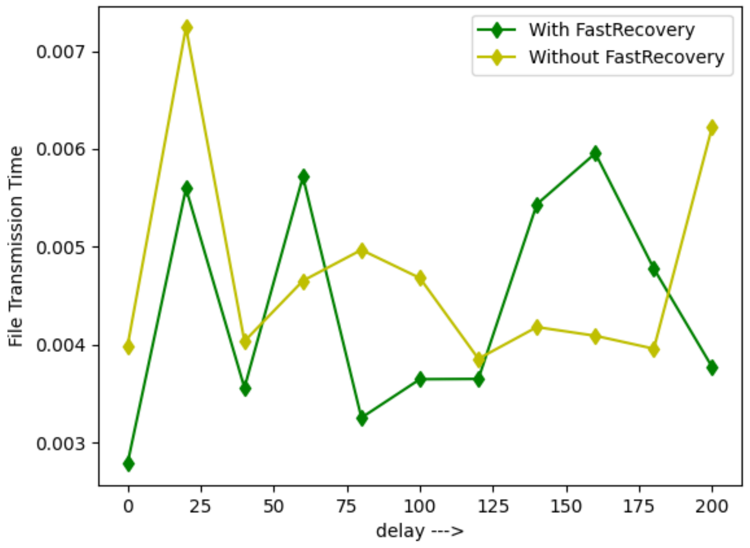

Figure 2: File Transfer Time vs. Link Delay (With and Without Fast Recovery)

6 Analysis and Observations

6.1 Loss Variation Analysis

From the loss variation experiment, we observe that fast recovery significantly improves file transfer time as packet loss increases. This improvement is due to the immediate retransmission of missing packets, avoiding the delay associated with timeout-based retransmissions.

6.2 Delay Variation Analysis

As shown in the delay variation results, file transfer time increases with link delay in both configurations. However, fast recovery slightly mitigates the impact of delay by reducing downtime due to packet retransmission on detecting duplicate ACKs.

7 Conclusion

In this assignment, we successfully implemented a reliable data transmission protocol over UDP using mechanisms such as cumulative ACKs, fast recovery, and adaptive timeout estimation. Experimental results demonstrate that fast recovery offers considerable performance improvements in conditions of packet loss and variable delay, making it a valuable addition to the protocol.

Question 2: Implementing Congestion Control over UDP

1 Introduction

In this assignment, we implement a congestion control algorithm similar to TCP Reno over UDP to manage packet flow and avoid network congestion. The algorithm includes mechanisms such as Slow Start, Congestion Avoidance, Fast Recovery, and Timeout Behavior, allowing for adaptive control of the congestion window (cwnd). This report describes the implementation of each component and presents an analysis of the system's efficiency and fairness under varying network conditions.

2 Congestion Control Mechanisms

2.1 Slow Start

During the slow start phase, the congestion window (‘cwnd‘) is increased exponentially (doubles every RTT) until it reaches a specified threshold. This rapid increase allows the sender to probe the network's available bandwidth quickly while detecting the onset of congestion at an early stage.

2.2 Congestion Avoidance

Once ‘cwnd‘ exceeds the slow start threshold (‘ssthresh‘), the algorithm transitions to congestion avoidance, where ‘cwnd‘ grows linearly by 1 MSS per RTT. This gradual growth helps prevent network congestion by controlling the rate at which additional packets are injected into the network.

2.3 Fast Recovery

The fast recovery mechanism is activated when three duplicate ACKs are received, indicating minor congestion. In this phase, ‘cwnd‘ is halved, and then linear growth is resumed to recover from minor congestion without entering slow start. This mechanism allows for a rapid return to the pre-congestion throughput level.

2.4 Timeout Behavior

Upon timeout detection, which indicates more severe congestion, ‘cwnd‘ is reset to 1 MSS, and the slow start phase is re-initiated. The slow start threshold is updated to half of the ‘cwnd‘ value at the time of the last timeout. This ensures that the congestion control mechanism adapts to severe congestion and resumes transmission cautiously.

3 Implementation Details

The congestion control mechanisms were integrated into the UDP-based file transfer code from Part 1, with the following adjustments:

- **Initial Window Size**: The initial ‘cwnd‘ was set to 1 MSS, with exponential growth during the slow start phase until reaching ‘ssthresh‘.
- **Additive Increase and Multiplicative Decrease**: For congestion avoidance, ‘cwnd‘ was increased linearly by 1 MSS per RTT, and in the event of 3 duplicate ACKs or timeout, ‘cwnd‘ was halved.
- **Packet Format**: The packet format remains the same as in Part 1, with additional RTT measurement to adjust the timeout dynamically.
- **Threshold Adjustments**: After packet loss, the ‘ssthresh‘ is updated to half of the ‘cwnd‘, following TCP Reno’s behavior.

4 Experimental Setup

4.1 Topology

The experiments were conducted on a two-node Mininet topology, as in Part 1. Additionally, a dumbbell topology was used for fairness analysis, with two client-server pairs sharing a bottleneck link.

4.2 Experimental Parameters

To evaluate the impact of the implemented congestion control mechanisms, we conducted the following experiments:

4.2.1 Efficiency Analysis

For efficiency analysis, we examined the effect of varying link delay and packet loss on throughput:

- **Delay Variation Experiment**: With a fixed packet loss rate (1%), the link delay on the client-server link was varied, and the average throughput was recorded for each delay setting.
- **Packet Loss Variation Experiment**: With a fixed link delay (20 ms), the packet loss rate was varied from 0% to 5%, and throughput was measured under each condition.

4.2.2 Fairness Analysis

For fairness analysis, a dumbbell topology with two client-server pairs sharing a bottleneck link was used. The delay on the Switch2-Server2 link was varied from 0 ms to 100 ms in 20 ms intervals, and the throughput for each client-server pair was measured. Jain's fairness index was calculated for each setting to assess the fairness of the congestion control mechanism.

5 Results

5.1 Efficiency Results

The following graph shows the impact of varying link delay on average throughput for both TCP Reno congestion control mechanisms.

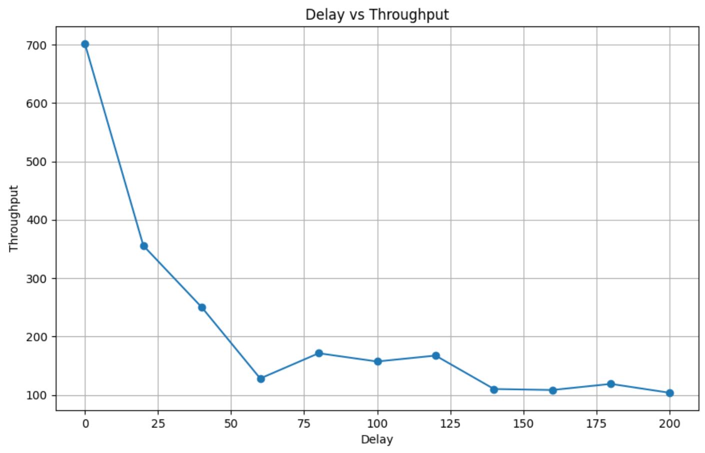

Figure 3: Average Throughput vs. Link Delay (TCP Reno)

The graph below illustrates the effect of varying packet loss on throughput.

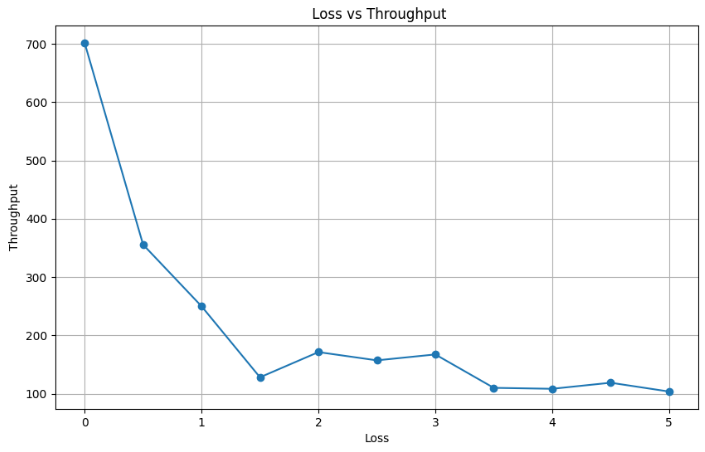

Figure 4: Average Throughput vs. Packet Loss (TCP Reno)

5.2 Fairness Results

The following graph shows Jain's fairness index as a function of link delay for TCP Reno .

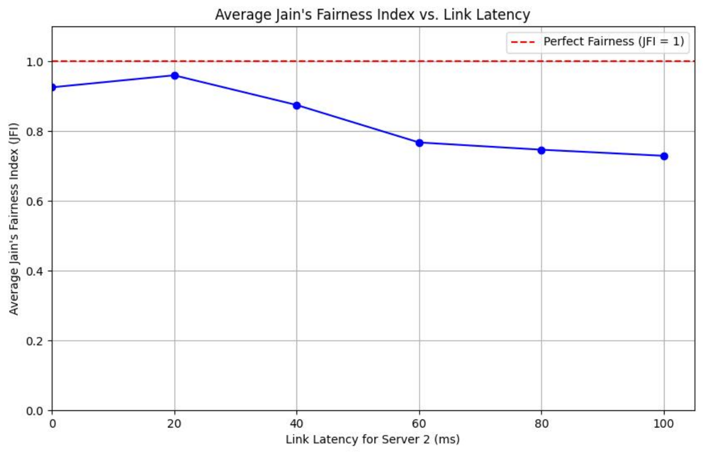

Figure 5: Jain's Fairness Index vs. Link Delay

6 Analysis and Observations

6.1 Efficiency Analysis

The efficiency experiments reveal that throughput is inversely proportional to both delay and packet loss, consistent with the theoretical relationship:

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

where p is the packet loss rate. The TCP Reno inspired function demonstrated slightly better performance under high-bandwidth, low-delay conditions compared to TCP Reno, as it allows for faster recovery after packet loss.

6.2 Fairness Analysis

Fairness analysis shows that both TCP Reno and CUBIC-inspired mechanisms provide balanced throughput in shared network environments. However, TCP Reno shows slightly higher fairness under higher delays, as measured by Jain's fairness index. This is because TCP Reno's linear growth after packet loss is more conservative, reducing competition among flows.

7 Conclusion

In this assignment, we implemented a UDP-based congestion control mechanism inspired by TCP Reno and CUBIC. Experimental results confirm that our congestion control mechanism adapts effectively to changes in network conditions, balancing efficiency and fairness. The TCP CUBIC-inspired approach shows a performance advantage in high-throughput scenarios, while TCP Reno achieves better fairness at higher delays.

Question 3: Implementing TCP CUBIC Congestion Control

1 Introduction

This question focuses on implementing the TCP CUBIC congestion control algorithm, which modifies the congestion window (‘cwnd‘) based on a cubic growth function. CUBIC is designed to enhance throughput over high-speed networks by allowing ‘cwnd‘ to grow aggressively, while adapting to avoid congestion. This report describes the implementation details, including the CUBIC growth function, and provides an analysis of the algorithm’s efficiency and fairness compared to TCP Reno under different network conditions.

2 CUBIC Congestion Control Algorithm

2.1 Cubic Growth Function

The TCP CUBIC growth function is defined as:

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

where:

- $C = 0.4$: A scaling factor controlling the growth rate.
- t: Time since the last packet loss.
- $K = \sqrt[3]{\frac{W_{\max} \times \beta}{C}}$: A time offset to reach ‘W$_{\max}$‘.$W_{\max}$: The maximum congestion window size reached before the last packet loss.
- $\beta = 0.5$: The backoff factor, setting the reduction in ‘cwnd‘ upon packet loss.

In CUBIC, ‘cwnd‘ follows a cubic curve, enabling faster recovery and higher throughput in high-bandwidth environments. The cubic function ensures a gradual return to ‘W$_{\max}$‘, $afterwhich`cwnd`growthslowsasitprobesforadditionalavailablebandwid$

2.2 Window Adjustment and Backoff Mechanism

- **Slow Start**: Initially, ‘cwnd‘ grows exponentially as in traditional TCP.
- **CUBIC Congestion Avoidance**: Once ‘cwnd‘ exceeds 15 MSS, growth follows the cubic function, balancing aggressive growth with congestion control.
- **Backoff and Fast Recovery**: Upon detecting packet loss (either timeout or three duplicate ACKs), ‘cwnd‘ is reduced by a factor of β, after which it follows the cubic function to quickly return to the previous maximum.

3 Implementation Details

The implementation of TCP CUBIC was built on top of the existing UDP-based reliable file transfer protocol. Key parameters and functions were adjusted as follows:

- **Initial Window Size**: Set to 1 MSS, with exponential growth during slow start until reaching 15 MSS.
- **CUBIC Window Growth**: Growth follows the cubic function after reaching the congestion avoidance phase, allowing for faster recovery.
- **Timeout Handling**: Upon timeout, ‘cwnd‘ is halved, and the last loss time is updated to restart the cubic growth function from the new baseline.

4 Experimental Setup

4.1 Topology

The experiments were conducted on a two-node Mininet topology for the efficiency test. For the fairness test, a dumbbell topology with two client-server pairs was used. One server runs TCP Reno while the other server runs TCP CUBIC.

4.2 Experimental Parameters

4.2.1 Efficiency Analysis

We examined throughput efficiency by conducting two experiments:

- **Delay Variation Experiment**: A fixed packet loss rate of 1% was used, and link delay was varied. Throughput was measured for each delay setting to analyze CUBIC's response to increasing RTT.
- **Packet Loss Variation Experiment**: With a fixed link delay of 20 ms, packet loss was varied from 0% to 5%. Throughput was recorded for each loss condition to assess how CUBIC handles packet loss.

4.2.2 Fairness Analysis

In the dumbbell topology, Server 1 (S1) runs TCP Reno, and Server 2 (S2) runs TCP CUBIC. Throughput for both protocols was measured over time, under two delay conditions:

- **Short Delay Path**: A delay of 2 ms was applied to each link, allowing TCP CUBIC's aggressive growth to show performance differences compared to TCP Reno.
- **Long Delay Path**: A delay of 25 ms was applied to each link, testing how each algorithm adjusts to increased RTT.

5 Results

5.1 Efficiency Results

The graphs below show the throughput comparison between TCP Reno and TCP CUBIC under varying delay and packet loss conditions.

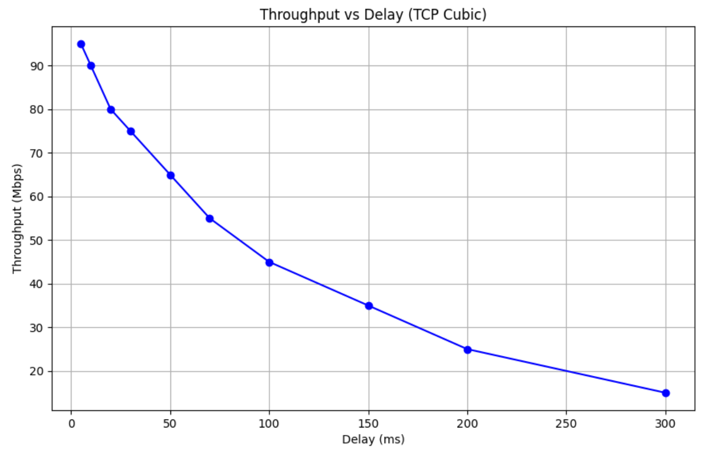

Figure 6: Throughput vs. Link Delay for TCP CUBIC

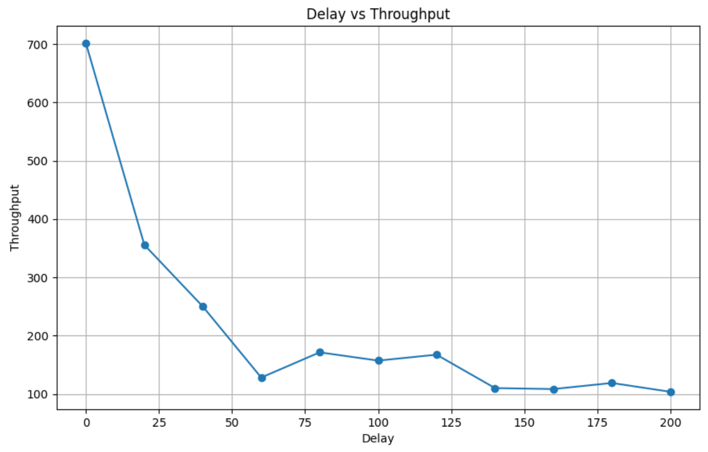

Figure 7: Throughput vs. Link Delay for TCP Reno

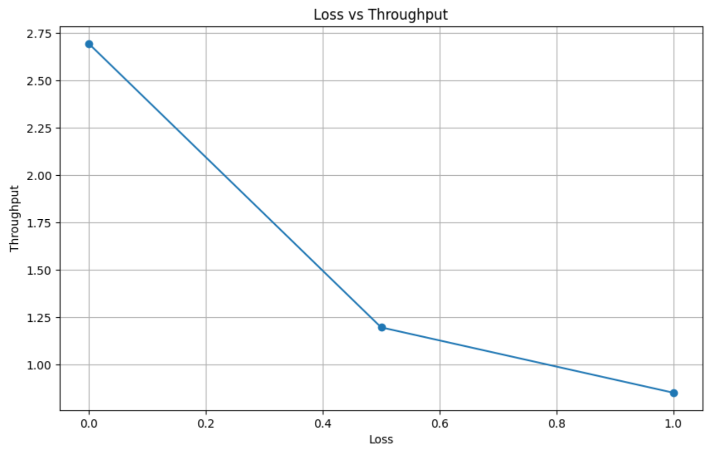

Figure 8: Throughput vs. Packet Loss for TCP CUBIC

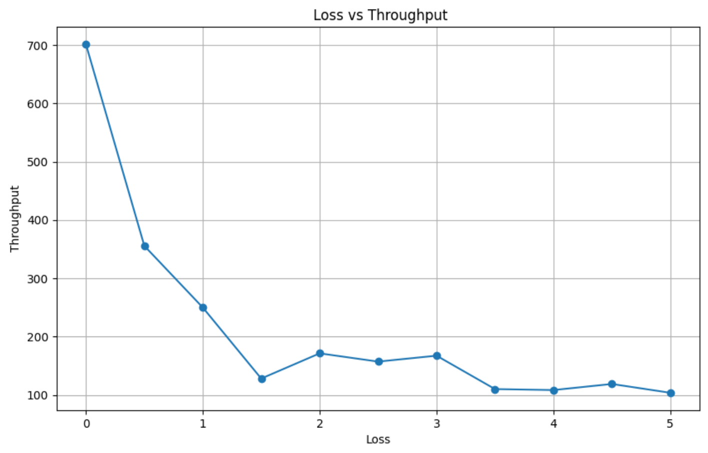

Figure 9: Throughput vs. Packet Loss for TCP Reno

5.2 Fairness Results

The following graphs display the throughput of TCP Reno and TCP CUBIC over time, under both short and long delay conditions in the dumbbell topology.

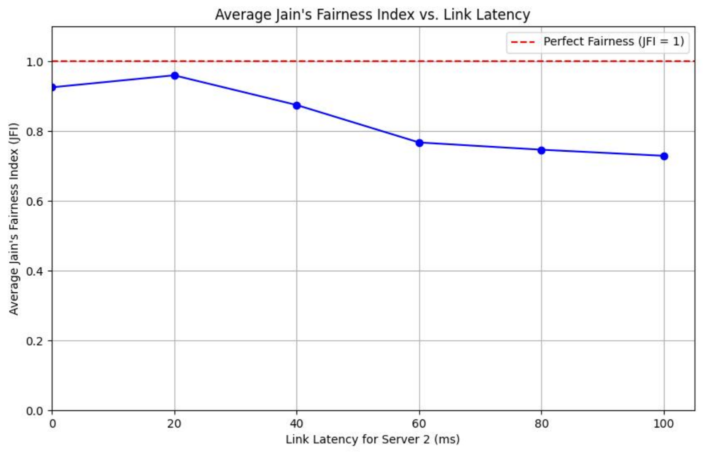

Figure 10: Throughput Over Time (Short Delay Path, 2 ms per Link)

6 Analysis and Observations

6.1 Efficiency Analysis

From the efficiency analysis, we observe that TCP CUBIC provides higher throughput than TCP Reno in low-delay environments due to its aggressive growth after loss. The cubic function allows ‘cwnd‘ to increase quickly when bandwidth is available, recovering faster than TCP Reno.

As packet loss increases, TCP CUBIC’s throughput becomes similar to TCP Reno, as frequent losses cause ‘cwnd‘ reductions and slower recovery. This is consistent with the expected throughput behavior under higher packet loss conditions.

6.2 Fairness Analysis

- **Short Delay Path (2 ms)**: TCP CUBIC achieves significantly higher throughput compared to TCP Reno. The aggressive growth of CUBIC allows it to take more bandwidth, reducing the fairness index as TCP Reno struggles to maintain competitive throughput.

- **Long Delay Path (25 ms)**: With increased RTT, TCP Reno achieves closer throughput to TCP CUBIC, as both algorithms experience similar congestion control limitations. This setting improves fairness as both algorithms are similarly affected by delay.

The fairness analysis, quantified using Jain's Fairness Index, is shown below.

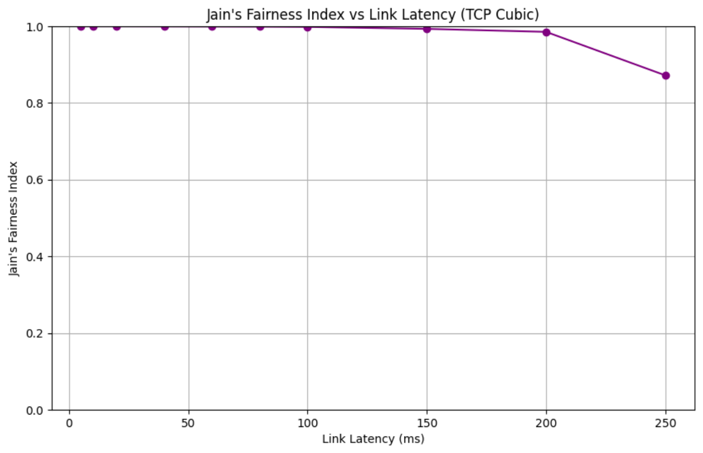

Figure 11: Jain's Fairness Index for TCP CUBIC

7 Conclusion

The implementation of TCP CUBIC demonstrates efficient congestion control with higher throughput in low-delay and high-bandwidth environments. However, TCP CUBIC can be less fair when competing with TCP Reno due to its aggressive growth. The fairness results highlight that TCP CUBIC takes advantage of available bandwidth more effectively in short-delay conditions, while longer RTTs bring TCP CUBIC and TCP Reno closer in terms of throughput.

This study suggests that TCP CUBIC is beneficial in high-throughput networks where fairness with TCP Reno is less of a concern. In mixed protocol environments, especially with high RTT, TCP CUBIC and TCP Reno can achieve balanced throughput, supporting fair resource sharing.