COL334 Assignment 4

Abhinav Rajesh Shripad Entry Number: 2022CS11596
Jahnabi Roy Entry Number: 2022CS11094

November 1, 2024

Part 1

Code Overview

The server initiates the process by sequentially numbering packets and sending them to the client. The client acknowledges receipt of each packet using cumulative acknowledgments (ACKs), which are designed to inform the server of the next expected sequence number, thereby confirming the successful delivery of previous packets. A timeout mechanism triggers retransmission if the server fails to receive an ACK within the expected timeframe or detects three duplicate ACKs. When fast recovery is enabled, the server responds to duplicate ACKs by quickly retransmitting lost packets, allowing the transfer to continue without waiting for a full timeout. To dynamically adapt to network conditions, the timeout duration is calculated based on observed round-trip times (RTTs).

Analysis of Trends

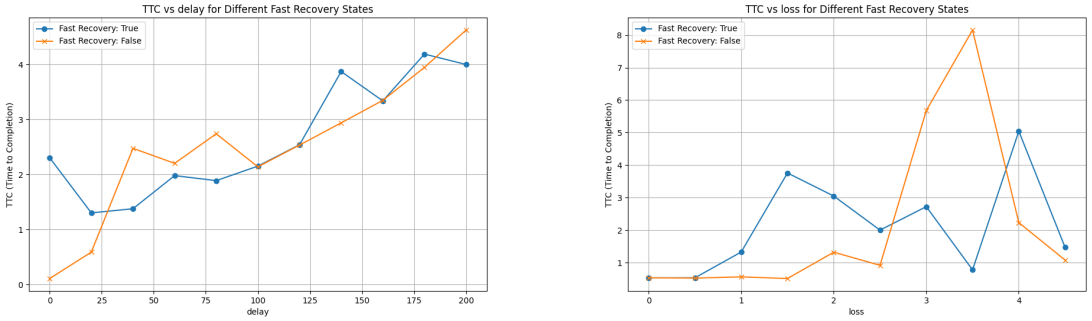

Figure 1. Analysis of trends for different values of loss and delays

The above data has been collected by running experiments for different values of delay at constant 1% loss and for different values of loss ranging from 0.5-5 at constant delay of 20ms (`p1_exp.py`).

Analysis of the observed trends:

- **Delay Experiments:** The delay experiments revealed that TTC generally increases with delay, showing variations depending on whether fast recovery is enabled. Fast recovery tends to mitigate severe delays by quickly retransmitting lost packets, resulting in a smoother trend line. Outliers with TTC exceeding 20 seconds were removed for clarity in presenting the results. As observed from the execution of the code, the high bumps in the graph is mostly due to the server achieving timeout and thus, causing an overhead in adjusting back to normal conditions. Observed trends are as follows:
  - *Low Delay Region (0-50ms)* : With Fast Recovery, TTC remains relatively low and stable, indicating the protocol's ability to handle minor delays effectively. Without Fast Recovery, TTC starts low but rises more sharply, showing a susceptibility to delay-induced timeouts.
  - *Medium Delay Region (50-125ms)* : Both approaches exhibit a gradual increase in TTC, though the performance gap narrows with increasing delay. By around 125ms, the effects of fast recovery become less pronounced.

  - *High Delay Region (125-200ms)* : TTC continues to increase, with both approaches performing similarly. However, fast recovery exhibits fewer spikes, underscoring its benefit in maintaining stability at high delays.

- **Loss Experiments:** In the loss experiments, increasing packet loss significantly impacted TTC, especially in the absence of fast recovery. Fast recovery reduces the impact of packet loss by rapidly retransmitting lost packets, which helps to smooth out TTC fluctuations. Outliers with TTC exceeding 10 seconds were removed for clarity in presenting the results. Fast Recovery helps maintain more predictable behavior in high-loss scenarios.

  - *Low Loss Region (0-1%)* : For both approaches, TTC remains stable with minimal variance. The low loss rate does not substantially impact completion times, and both setups perform comparably.
  - *Medium Loss Region (1-2%)* : With fast recovery, TTC increases moderately, reflecting the added overhead of retransmissions. However, performance remains stable. Without fast recovery, performance begins to degrade noticeably, with TTC experiencing sharper rises as the loss rate increases.
  - *High Loss Region (2-4%)* : With Fast Recovery, TTC increases but remains more predictable, with consistent retransmission intervals helping to maintain relative stability. Without Fast Recovery, TTC spikes dramatically, especially around 3% loss, indicating the effect of timeouts and delayed recovery processes.

Overall, the results underscore the utility of fast recovery in mitigating the impacts of both delay and packet loss. It maintains stability, particularly in moderate network conditions, by preventing the uncontrolled increase in TTC seen without it.

Part 2

Code Overview

The TCP Reno server implements a dynamic window size using the congestion window and slow start threshold variables to adapt to network conditions. Initially set to a predefined size, the congestion window increases during the slow start phase with each acknowledgment received, while transitioning to a more conservative growth rate in the congestion avoidance phase. Upon detecting timeouts, the server reduces the slow start threshold and resets the congestion window, effectively slowing down transmission to mitigate congestion. Additionally, the server responds to duplicate ACKs, entering fast recovery mode to adjust the window size.

Analysis of Trends

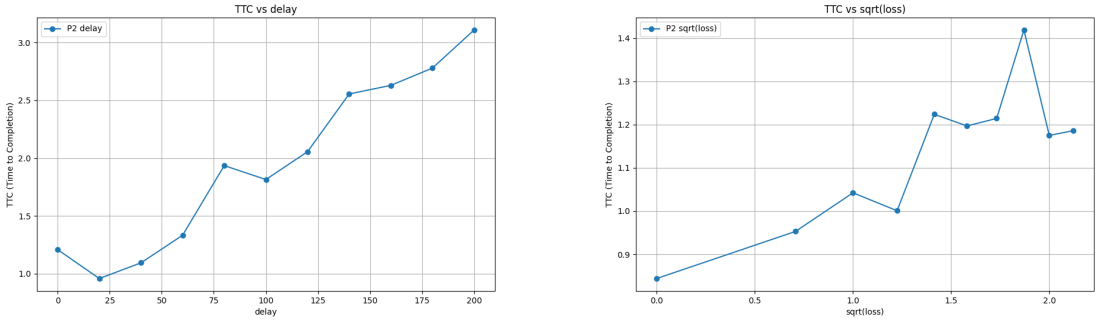

Figure 2. Analysis of trends for different values of delays and losses (experimented on similar topology in part 1) wrt TTC

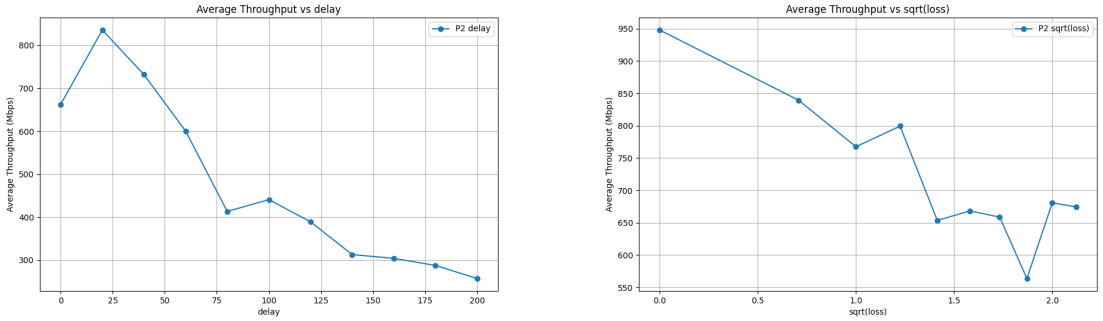

Figure 3. Analysis of trends for different values of delays and losses (experimented on similar topology in part 1) wrt Average Throughput

- **Delay Experiments**: The graph shows Time to Completion (TTC) versus delay. Since TTC is inversely proportional to throughput (as also shown in Figure 3), we analyze the results in terms of the TTC vs. Delay plot.
  - Theory suggests that throughput $\propto \frac{1}{\text{RTT}}$, implying that TTC $\propto$ RTT.
  - An overall increasing trend in TTC is observed as delay increases. The relationship appears approximately linear, with an initial dip at 25ms, likely due to experimental variance.
  - The graph indicates that higher delays increasingly impact performance negatively, aligning with theoretical expectations, as increased RTT (delay) extends completion times.
- **Loss Experiments**: The graph shows Time to Completion (TTC) versus the square root of packet loss. Since TTC is inversely proportional to throughput (as also shown in Figure 3), we analyze the results in terms of the TTC vs. $\sqrt{\text{loss}}$ plot.
  - Theory suggests that throughput $\propto \frac{1}{\sqrt{p}}$, implying that TTC $\propto \sqrt{p}$.
  - A general upward trend in TTC is observed as $\sqrt{\text{loss}}$ increases, aligning with theoretical expectations.
  - The relationship is not perfectly smooth, showing some fluctuations, with a notable spike around loss = 3.5, likely indicating points where congestion control mechanisms struggle to maintain efficient transmission.
  - Overall, the trend supports the theoretical relationship, though with more variation compared to the delay case.

Both graphs demonstrate that the implemented congestion control mechanism reflects theoretical TCP behavior, with performance degrading as delay (RTT) and packet loss (in terms of square root proportionality) increase.

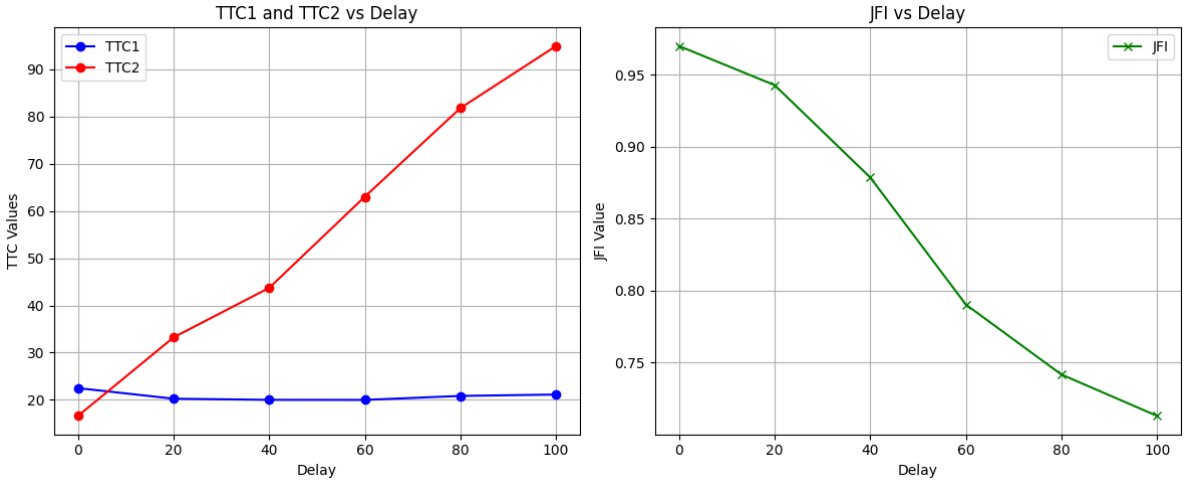

Figure 4. Analysis of trends for different values of delays with TTC and Jain's Fairness Index

- **TTC1 & TTC2 vs Delay**
  - **TTC1 Stability:** TTC1 represents the time taken for file transfer completion on the baseline link in the dumbbell topology, which maintains a default delay of 5ms. This stability provides a control reference for the impact of increased delays on the alternate link.
  - **TTC2 Sensitivity to Delay:** As delay is added to the secondary link, TTC2 shows a near-linear increase, highlighting a direct relationship between time to completion and delay. As TTC2 rises, throughput decreases, as longer transfer times indicate reduced efficiency.
  - These observations align with TCP performance expectations, where increased delay hinders throughput due to slower congestion window growth, prolonged detection and recovery from packet loss, and extended feedback loops for congestion control.
- **Jain's Fairness Index (JFI)**
  - **Fairness Across Delay:** The JFI graph reflects the fairness of resource allocation between the two connections as delay varies. At 0ms delay, JFI starts near 0.97, indicating almost perfect fairness.
  - **Decline with Increasing Delay:** As delay grows, JFI decreases non-linearly, showing a moderate decline until around 40ms, followed by a sharper drop from 40ms to 60ms, eventually reaching approximately 0.72 at 100ms delay.
  - High initial fairness suggests the congestion control algorithm is effective under low-delay conditions. However, as delay increases, the growing throughput disparity mirrors the degradation in fairness. The non-linear decline indicates key delay thresholds where fairness is more significantly impacted.
  - Increased Round Trip Time (RTT) affects the congestion window growth rate, and asymmetric delays lead to imbalanced resource allocation, thus reducing fairness.

Part 3

Code Overview

For reliability, acknowledgments (ACKs), packet numbering, retransmissions, and a fast recovery mechanism, ensuring correct packet ordering and re-sending of lost packets is implemented. For congestion control, it employs both TCP Reno and TCP CUBIC algorithms. TCP Reno uses a sliding window with slow-start, congestion avoidance, and fast recovery phases, adjusting the window size based on network conditions. The TCP CUBIC algorithm introduces a cubic function to adjust the congestion window, optimizing performance on high-speed networks.

Analysis of Trends

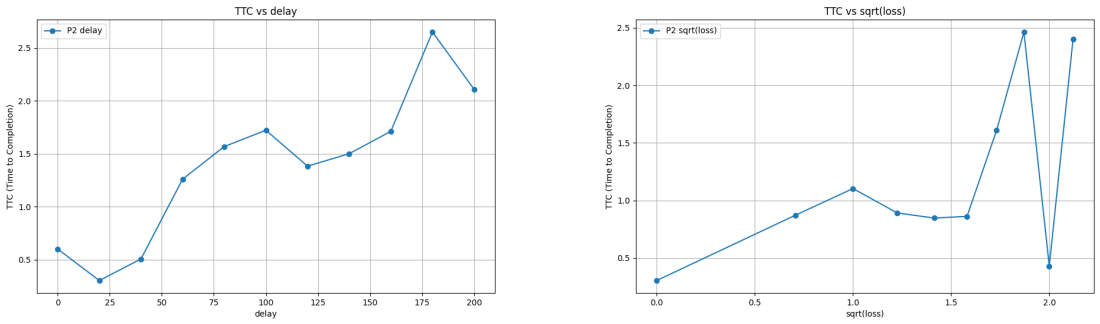

Figure 5. Analysis of trends for different values of delays and losses (experimented on similar topology in part 1) wrt TTC

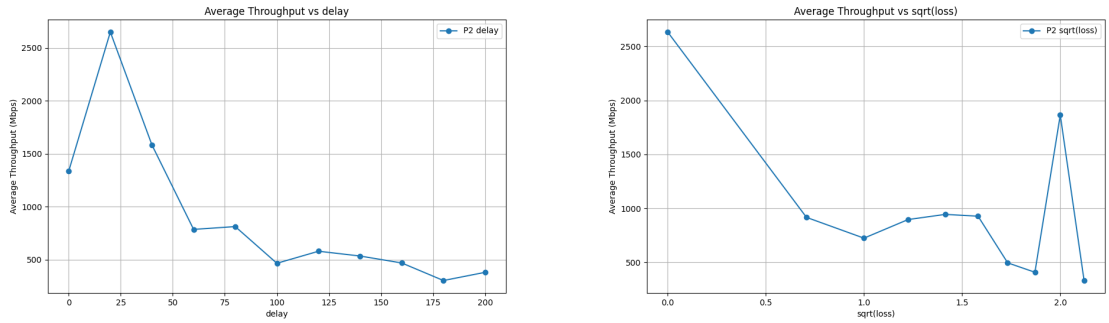

Figure 6. Analysis of trends for different values of delays and losses (experimented on similar topology in part 1) wrt Average Throughput

- **Delay Experiments**: The graph shows Time to Completion (TTC) versus delay. Since TTC is inversely proportional to throughput (as also shown in Figure 6), we analyze the results in terms of the TTC vs. Delay plot.
  - As with TCP-Reno, TCP-Cubic also follows a similar trend, with a few experimental variations.
  - However the throughput reaches 2500 Mbps for TCP-Cubic while throughput is nearly 900 Mbps for TCP-Reno, thus showing a significant improvement for TCP-Cubic. TCP CUBIC, optimized for high-speed networks, uses cubic function for window growth, allowing rapid scaling of throughput after loss recovery. This non-linear growth enables CUBIC to quickly utilize available bandwidth, demonstrating its advantage over Reno in high-performance network environments.
- **Loss Experiments**: The graph shows Time to Completion (TTC) versus the square root of packet loss. Since TTC is inversely proportional to throughput (as also shown in Figure 3), we analyze the results in terms of the TTC vs. $\sqrt{\text{loss}}$ plot.
  - Similarly for loss experiments, the trend is similar to TCP-Reno and aligns theoretical with some experimental variance.
  - As seen in delay experiments above, the throughput is quite high for TCP-Cubic (upto 2500 Mbps) in comparison to TCP-Reno.

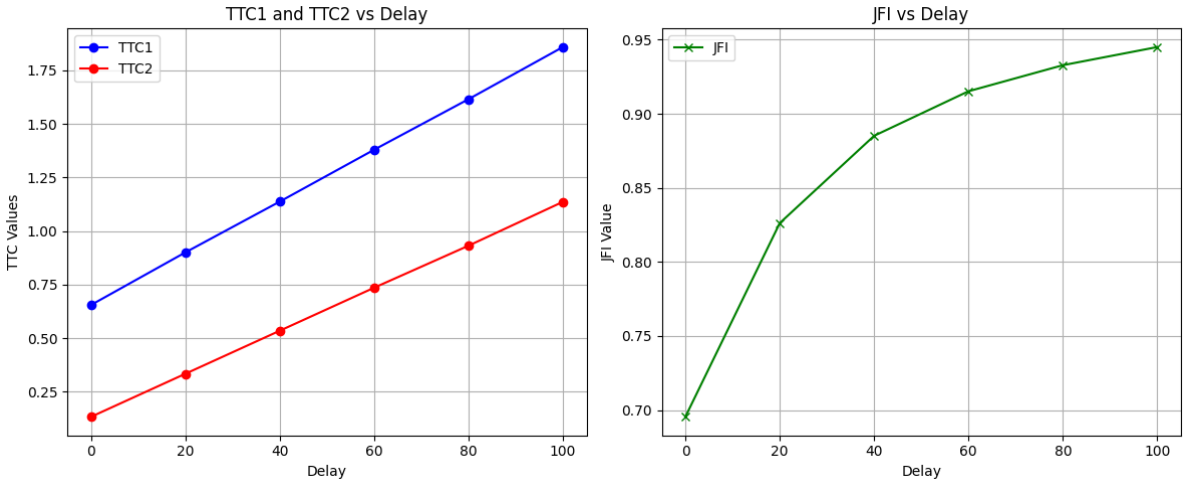

Figure 7. Analysis of trends for different values of delays with TTC and Jain's Fairness Index for TCP Reno vs TCP Cubic

- **TTC (Time to Completion) vs. Delay:** TCP Reno exhibits higher TTC values than TCP Cubic across all delays. As delay increases, both protocols show an upward trend in TTC, indicating a decrease in throughput with higher delays. Notably, TCP Cubic consistently achieves faster completion times than TCP Reno.

- **Jain's Fairness Index (JFI):** At 0 ms delay, JFI is approximately 0.70, suggesting moderate unfairness. However, JFI steadily improves as delay increases, reaching around 0.94 at 100 ms. This trend reflects that with increased delay, TCP Reno and TCP Cubic tend to share network bandwidth more equitably.
- **Aggressiveness and Fairness:** At lower delays, TCP Cubic is more aggressive, capturing a larger share of the available bandwidth. As delay increases, the competition between TCP Reno and TCP Cubic becomes more balanced, resulting in fairer resource sharing between the two protocols.