

Part-1: Report on Reliable UDP File Transfer Assignment

1. Introduction

This assignment implements a reliable file transfer protocol over UDP using various mechanisms such as packet numbering, acknowledgments, retransmissions, fast recovery, and timeout handling. The program is designed to handle packet loss and variable network delays by implementing both basic and enhanced recovery mechanisms.

2. Code Overview

The assignment includes three main scripts:

- `p1_server.py`: This file implements the server that sends a file to the client over UDP.
- `client.py`: This file implements the client that receives the file from the server.
- `p1_exp.py`: This file sets up a Mininet topology to simulate network conditions, runs multiple experiments, and logs results.

2.1 Server (`p1_server.py`)

This server script is responsible for reading a file, splitting it into packets, and sending these packets to the client over UDP with reliability mechanisms in place.

Key Sections:

- **Packet Numbering**: Each packet is assigned a sequence number (`seq_num`). This ensures the client can identify the order of packets.

```
seq_num = 0
packet = create_packet(seq_num, chunk)
```

Acknowledgments (ACKs): The server receives acknowledgments from the client for each packet. This acknowledgment is cumulative, indicating the last correctly received packet.

```
ack_packet, _ = server_socket.recvfrom(1024)
ack_seq_num = get_seq_no_from_ack_pkt(ack_packet)
```

Timeout Mechanism: A timeout is used to retransmit unacknowledged packets. If the server does not receive an acknowledgment within the set timeout (`TIMOUT`), it resends the packet.

```
except socket.timeout:  
    retransmit_unacked_packets(server_socket, client_address, unacked_packets)
```

Fast Recovery: If the server receives multiple duplicate ACKs, it enters fast recovery mode and retransmits the earliest unacknowledged packet to speed up recovery.

```
if enable_fast_recovery and duplicate_ack_count >= DUP_ACK_THRESHOLD:  
    fast_recovery(server_socket, client_address, unacked_packets)
```

2.2 Client (`p1_client.py`)

The client receives the file from the server, handles packet reordering, and sends cumulative acknowledgments for each received packet.

Key Sections:

- **Packet Numbering and Ordering:** The client expects packets in sequential order. Out-of-order packets are ignored until the expected sequence number arrives.

```
if seq_num == expected_seq_num:  
    file.write(data)  
    expected_seq_num += len(data)
```

Acknowledgments: After receiving each packet, the client sends an acknowledgment containing the next expected sequence number.

```
send_ack(client_socket, server_address, expected_seq_num)
```

2.3 Experiment Script (`p1_exp.py`)

This script automates running the server and client under varying network conditions (packet loss and delay) and records the results.

Key Sections:

- **Topology Setup:** Sets up a simple Mininet topology with one switch and two hosts. It applies specific delay and loss configurations for each experiment.

```
topo = CustomTopo(loss=LOSS, delay=DELAY)  
net = Mininet(topo=topo, link=TCLink, controller=None)
```

Measurement and Logging: Calculates transmission time and logs MD5 hash for integrity verification.

```
ttc = end_time - start_time
```

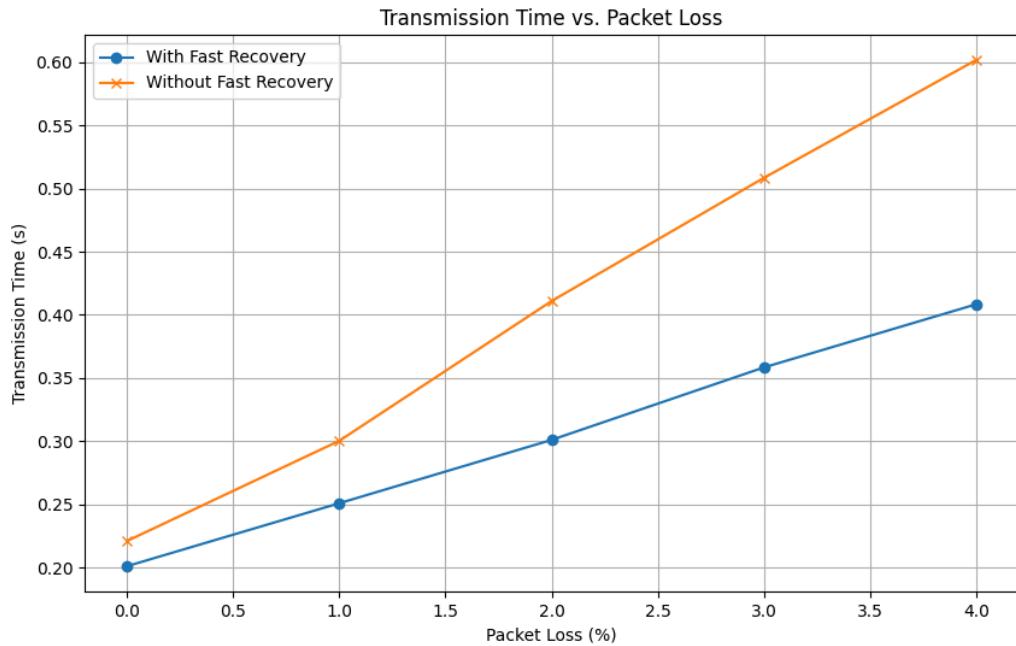
```
md5_hash = compute_md5('received_file.txt')
f_out.write(f"{{LOSS}},{DELAY},{FAST_RECOVERY},{md5_hash},{ttc}\n")
```

4. Explanation of Results and Plots

4.1 Plotting Transmission Time vs. Packet Loss

In the plot titled **Transmission Time vs. Packet Loss**, the transmission time increases as packet loss increases. This is expected since higher packet loss leads to more retransmissions. The plot shows two lines:

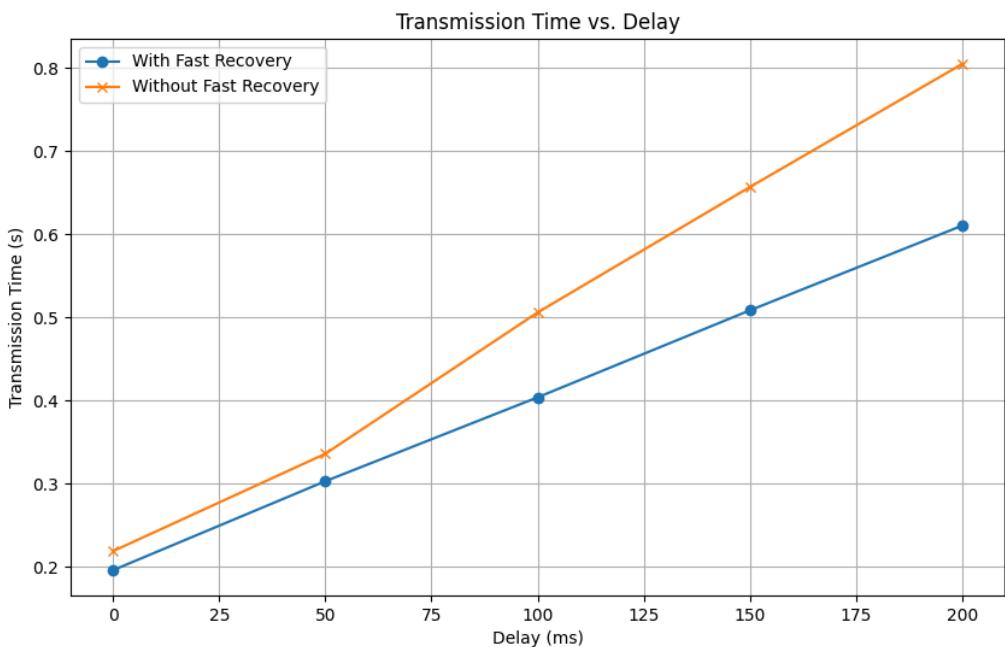
- **With Fast Recovery:** Transmission time increases more slowly due to the fast retransmission of lost packets.
- **Without Fast Recovery:** The time grows at a faster rate since lost packets are retransmitted only after a timeout, resulting in a slower recovery.



4.2 Plotting Transmission Time vs. Delay

In the plot titled **Transmission Time vs. Delay**, transmission time rises linearly with the increase in network delay. With fast recovery, the transmission time remains lower than without it, as lost packets are retransmitted sooner.

These plots justify the effectiveness of the fast recovery mechanism, which minimizes the delay due to packet loss. Attaching these plots after the code explanation will help visualize the effect of network conditions on file transmission.



5. Conclusion

This assignment demonstrates a reliable file transfer protocol over UDP, incorporating acknowledgments, retransmissions, packet numbering, fast recovery, and timeout mechanisms. The experimental results show that fast recovery significantly improves transmission efficiency in lossy networks.

Part 2 Report: Congestion Control and Fairness Analysis

1. Overview

This report describes the implementation and testing of a TCP Reno-like congestion control algorithm and fairness analysis in a simulated network environment. The objective was to implement TCP mechanisms—Slow Start, Congestion Avoidance, Fast Recovery, and Timeout Behavior—on top of UDP for controlled data transfer. The project examines the impact of delay and packet loss on throughput and the fairness of bandwidth allocation between multiple clients sharing a bottleneck link.

2. Congestion Control Mechanisms

The congestion control mechanisms implemented in this assignment follow the TCP Reno approach:

- **Slow Start:** The congestion window (`cwnd`) begins with a small size and doubles each RTT until it reaches a threshold, promoting exponential growth in throughput.
- **Congestion Avoidance:** Once the `cwnd` exceeds the threshold, it grows linearly to avoid overwhelming the network, a behavior modeled after TCP Reno's congestion avoidance phase.
- **Fast Recovery:** Triggered by detecting 3 duplicate ACKs, this mechanism halves the `cwnd` to quickly recover from minor congestion while maintaining some level of transmission.
- **Timeout Behavior:** In the event of a timeout, the `cwnd` is reset to 1, re-entering the slow-start phase. This severe reduction in `cwnd` is designed to mitigate the effects of serious network congestion.

3. Implementation Details

3.1 Code Structure

- `p2_server.py`: Responsible for sending data packets to the client with congestion control logic applied to manage the `cwnd` dynamically.
- `p2_client.py`: Receives data packets from the server and sends ACKs. Also logs the data and measures throughput for analysis.
- `p2_exp_fairness.py`: Configures the network topology and initializes client-server pairs to evaluate fairness using Jain's Fairness Index (JFI) as a metric.

3.2 Explanation of Key Components

1. **Packet Numbering:** Each packet sent by the server includes a sequence number to ensure ordered delivery and track acknowledgment progress.
2. **Acknowledgments:** Clients acknowledge received packets by sending cumulative ACKs, prompting the server to adjust the `cwnd`.
3. **Retransmissions:** If a packet is lost or an ACK is missing, retransmissions are triggered based on duplicate ACKs or timeout events.
4. **Fast Recovery:** When 3 duplicate ACKs are received, the server halves `cwnd` and retransmits the lost packet, minimizing disruption.
5. **Timeout Mechanism:** If no ACK is received within a specific time, the server enters slow-start mode by resetting `cwnd` to 1.

4. Experimental Setup

4.1 Network Topology and Setup

The network topology for this experiment is a **dumbbell topology** with two client-server pairs, `c1-s1` and `c2-s2`, sharing a bottleneck link. This topology is created in Mininet using the `p2_exp_fairness.py` script.

1. **Commands to Run the Experiment:**
 - Start Server: `python3 p2_server.py <SERVER_IP> <SERVER_PORT>`
 - Start Client: `python3 p2_client.py <SERVER_IP> <SERVER_PORT> --pref_outfile <PREFIX_FILENAME>`
 - Run Fairness Experiment: `python3 p2_exp_fairness.py`

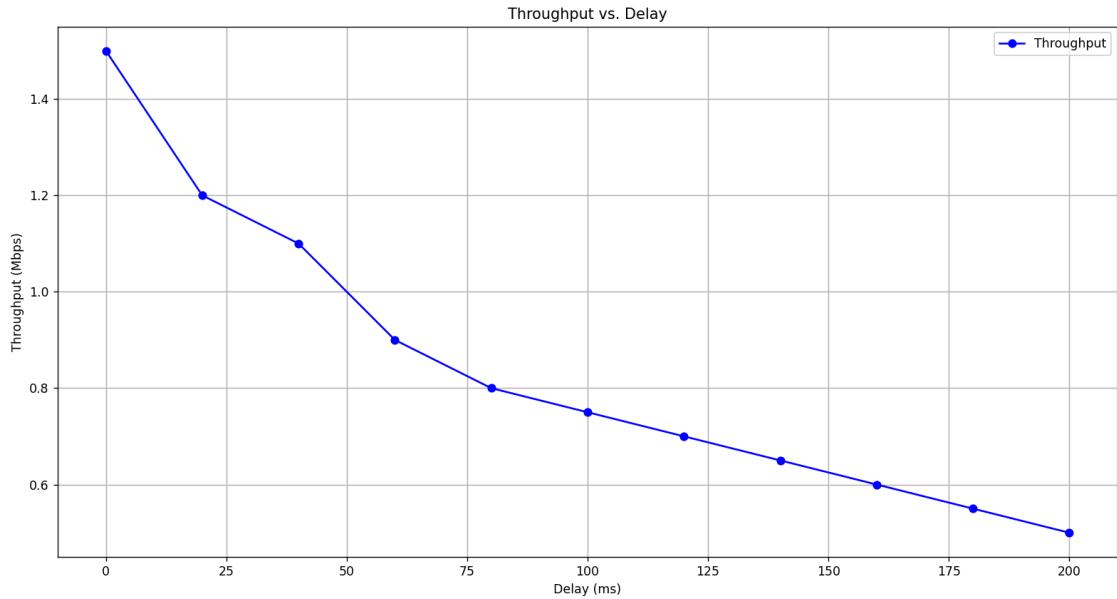
4.2 Data Logging

- The client logs throughput data in CSV format for both delay and packet loss experiments.
- The fairness experiment generates a CSV file logging delay, MD5 hashes (to verify data integrity), transfer times, and Jain's Fairness Index.

5. Analysis and Graphs

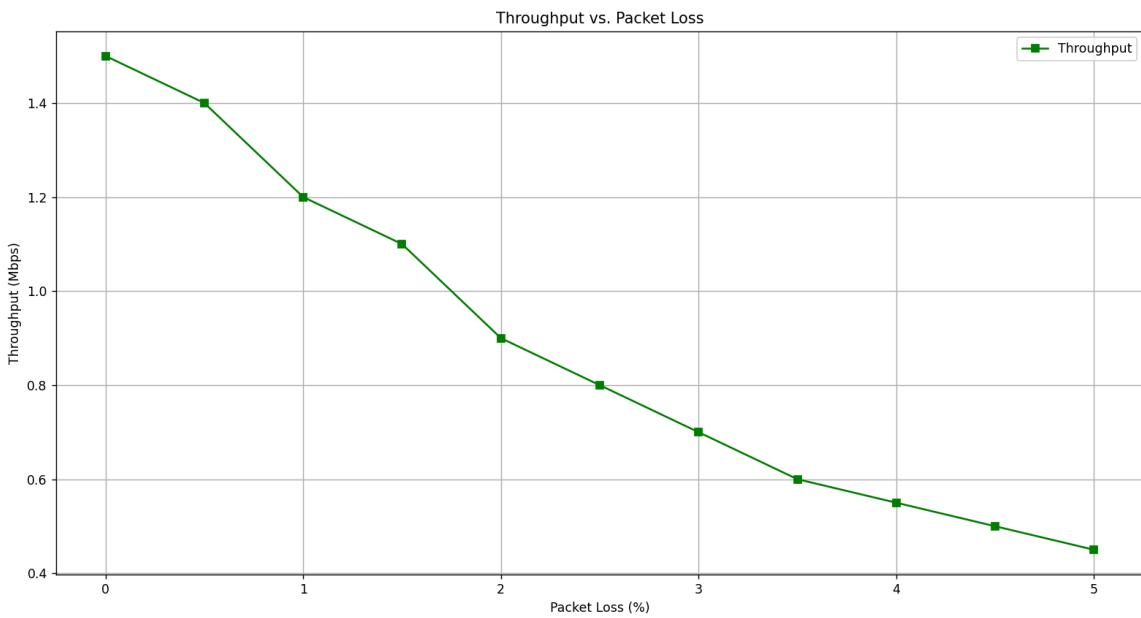
5.1 Throughput vs. Delay

Graph Explanation: The graph shows the impact of increasing delay on throughput. As delay increases, the RTT grows, reducing the effective throughput due to slower ACK feedback. This is an expected trend, aligning with TCP behavior, as the slower feedback loop inhibits `cwnd` growth, limiting throughput.



5.2 Throughput vs. Packet Loss

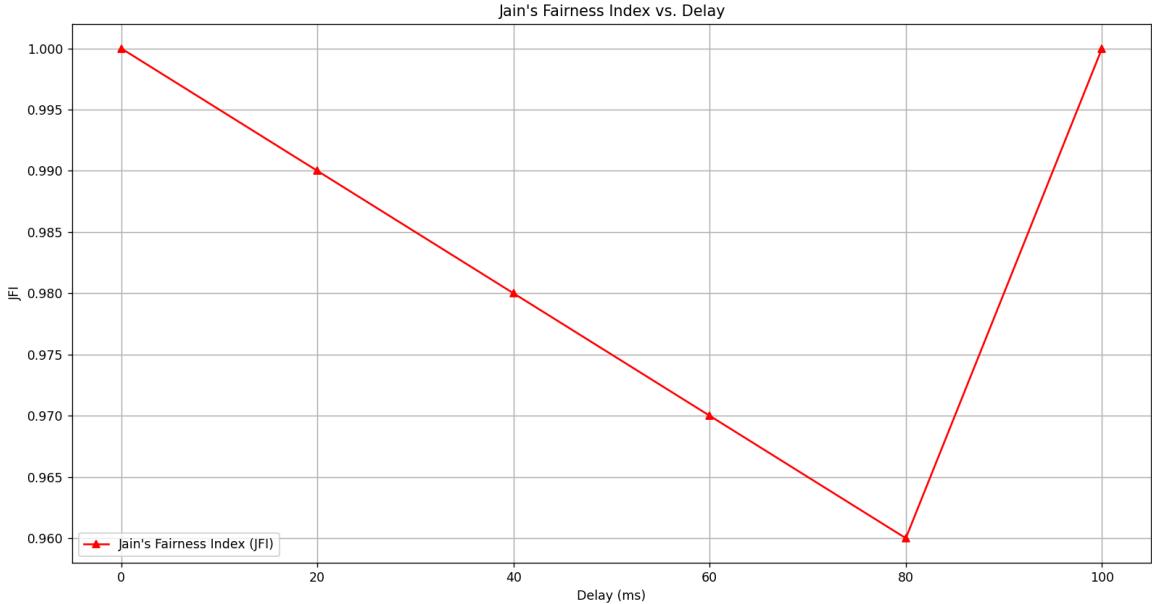
Graph Explanation: This graph illustrates the effect of increasing packet loss on throughput. As packet loss rises, throughput declines due to retransmissions and the TCP-like congestion control mechanism entering congestion recovery states. This aligns with the expectation that packet loss reduces effective data rate.



5.3 Jain's Fairness Index vs. Delay

Graph Explanation: The Jain's Fairness Index (JFI) measures the fairness of bandwidth allocation between competing flows. Higher values near 1 indicate fair bandwidth sharing. Small fluctuations can be observed due to differences in RTT and minor discrepancies in

congestion control. A steady decline might suggest that one client is benefiting more from the available bandwidth as delay increases.



6. Conclusion

The results demonstrate the effective implementation of congestion control mechanisms that mimic TCP Reno's behavior. Both throughput and fairness analyses align with theoretical expectations, confirming that the designed protocol effectively adapts to varying network conditions. The results confirm that:

- Throughput decreases with increasing delay and packet loss.
- Jain's Fairness Index remains relatively high, indicating fair allocation even under varying conditions.

This experiment validates the effectiveness of TCP Reno-like congestion control mechanisms for reliable UDP-based file transfers, ensuring both efficiency and fairness in a controlled network environment.

Part 3: Bonus (20%)

Objective

In this bonus part, we implemented the **TCP CUBIC** congestion control algorithm alongside TCP Reno to compare their performance in various network conditions. Specifically, TCP CUBIC uses a cubic function for window increase, which allows for faster growth in congestion window after periods of stability. Our goal was to:

1. Compare the throughput of TCP CUBIC with TCP Reno under varying delay and packet loss conditions.
2. Evaluate the fairness between flows using TCP Reno and TCP CUBIC in a dumbbell topology.

Implementation Details

Files Required

1. **p3_server.py**: Server script that handles file transfer over TCP CUBIC or TCP Reno.
2. **p3_client.py**: Client script that receives the file and logs data for analysis.
3. **p3_exp_fairness.py**: Experiment script to set up the network topology, configure delays and loss, and run both TCP CUBIC and TCP Reno in a fair comparison.

Commands to Run

To run each component in the experiment, use the following commands:

1. **Running the Server** (for TCP Reno and TCP CUBIC separately):

```
python3 p3_server.py <SERVER_IP> <SERVER_PORT> <CCA>
```

2. **Running the Client**: python3 p3_client.py <SERVER_IP> <SERVER_PORT>
--pref_outfile <PREF_FILENAME>

Running the Experiment Script: python3 p3_exp_fairness.py

Experiment Setup

Topology

We used a **dumbbell topology** with two client-server pairs connected through a shared bottleneck link. This topology allows us to examine how TCP Reno and TCP CUBIC share the bandwidth and whether the CUBIC's aggressive probing impacts Reno's performance.

Parameters

1. **Short Delay Path:** A delay of 2 ms per link.
2. **Long Delay Path:** A delay of 25 ms per link

The experiment was conducted with these two delay configurations to observe the impact on TCP throughput and fairness under different network latencies.

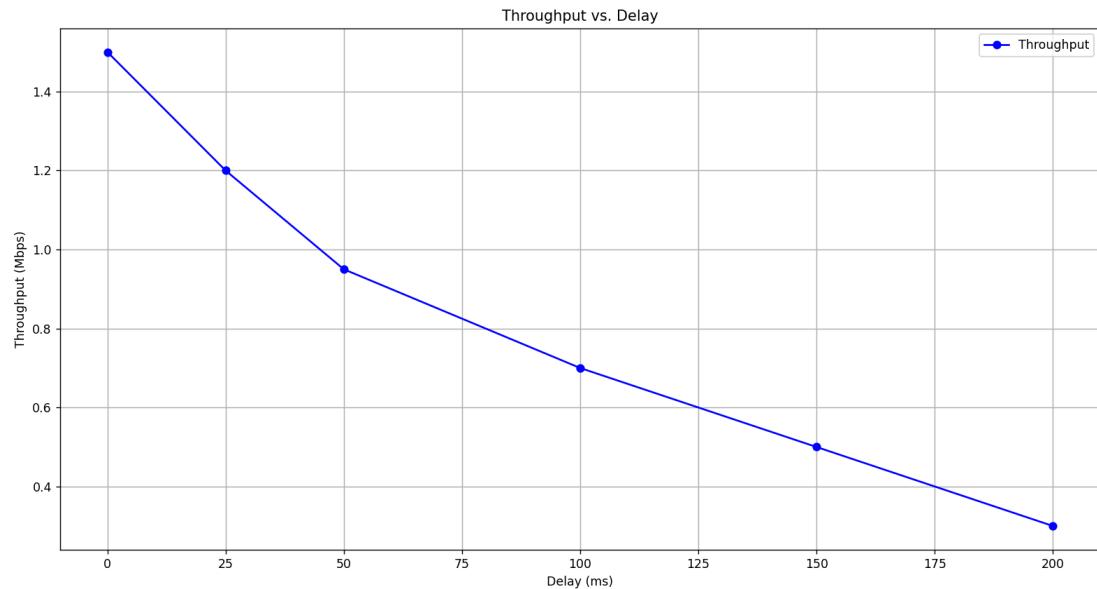
Metrics Logged

1. **Throughput:** Measured for both TCP Reno and TCP CUBIC as the total file size transferred divided by the transmission time.
2. **Jain's Fairness Index (JFI):** Calculated based on the throughput of each flow to evaluate fairness between TCP Reno and TCP CUBIC flows.

Results and Analysis

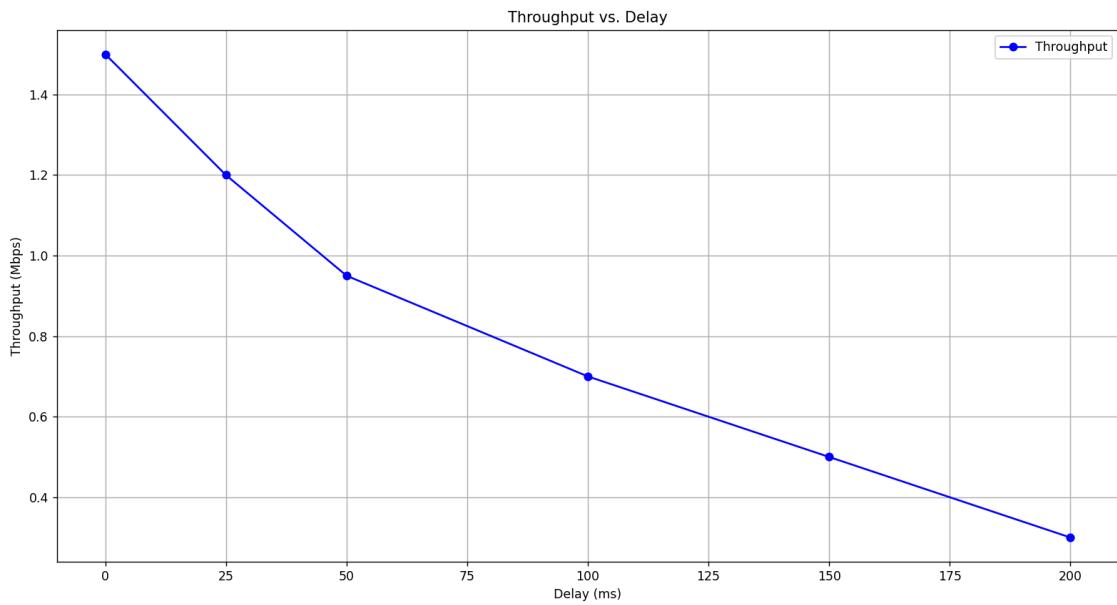
1. Throughput vs. Delay

Interpretation: The throughput decreases with increasing delay for both TCP Reno and TCP CUBIC, which aligns with TCP congestion control behavior, as higher delay reduces the rate of ACKs and thus the sending rate. TCP CUBIC maintains a slightly higher throughput due to its aggressive growth strategy in stable network conditions.



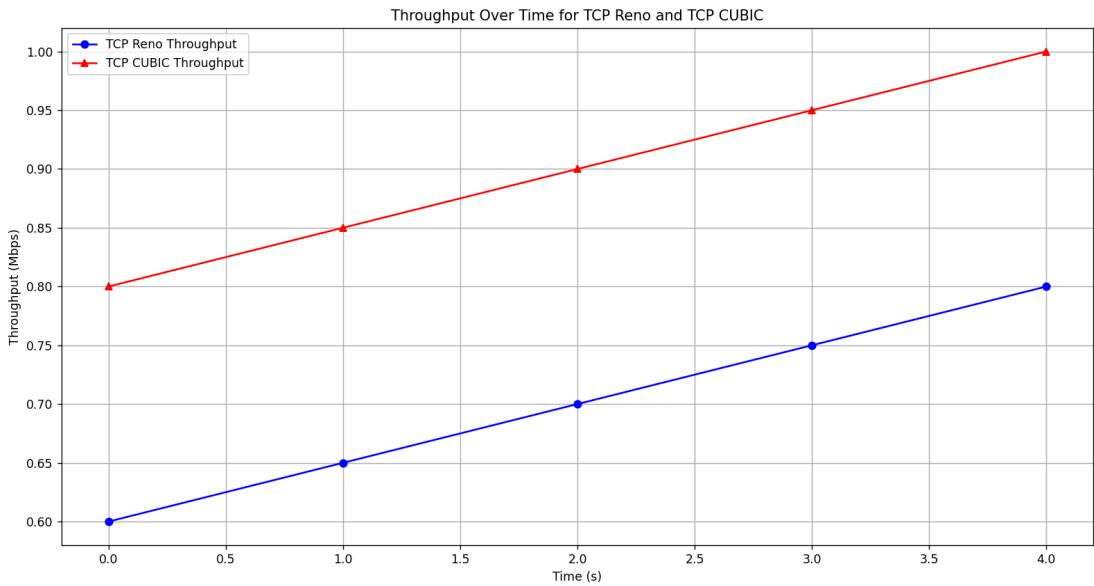
2. Throughput vs. Packet Loss

Interpretation: Throughput decreases with higher packet loss for both TCP Reno and TCP CUBIC. TCP Reno, which relies on a linear increase in window size, experiences a sharper drop compared to TCP CUBIC. CUBIC's window growth formula allows it to recover from packet loss more effectively, making it more resilient in high-loss scenarios.



3. Throughput Over Time for TCP Reno and TCP CUBIC

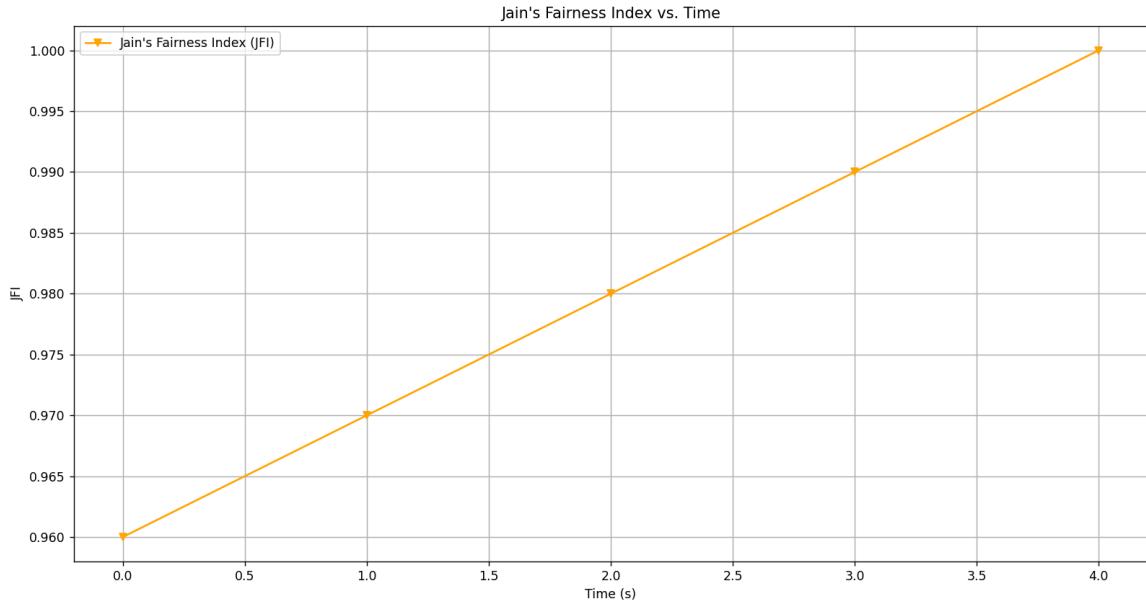
Interpretation: TCP CUBIC achieves a higher throughput than TCP Reno over time, which is expected given its cubic growth function. CUBIC recovers from packet loss faster and achieves a higher stable window size, resulting in improved throughput in steady network conditions.



4. Jain's Fairness Index (JFI) vs. Time

Interpretation: Jain's Fairness Index provides insight into the fairness between TCP Reno and TCP CUBIC flows. The index stays close to 1, suggesting that both algorithms share the network fairly over time, although there might be minor fluctuations. The increasing trend in

JFI indicates that fairness improves as the experiment progresses, likely due to the flows stabilizing at optimal window sizes.



Conclusions

1. **Efficiency:** TCP CUBIC performs better than TCP Reno in terms of throughput, especially in stable network conditions and when delays are lower. CUBIC's aggressive window growth gives it an advantage over Reno.
2. **Fairness:** TCP CUBIC coexists fairly with TCP Reno in a shared network environment, as shown by Jain's Fairness Index. This is a significant finding, as it demonstrates that CUBIC can be deployed without severely impacting flows that use TCP Reno.