COL334 Assignment 4

Rachit Bhalani (2023CS10961), Sabhya Arora (2023CS11177)

Reliable UDP File Transfer – Design Report

1. Header Structure

Each packet has a fixed 20-byte header:

- **Sequence Number (4 bytes):** For data packets, it is an identifier for the packet; for ACKs, it denotes the sequence number of the next packet that the client expects
- **Reserved/SACK (16 bytes):** Used in ACKs to encode up to two SACK blocks, each represented as (start, end).

Total size $\leq$ 1200 bytes (20-byte header + 1180-byte data).
ACK format: [next_expected(4)][start1(4)][end1(4)][start2(4)][end2(4)].

2. Protocol Overview

The system provides reliable file transfer over UDP using TCP-like mechanisms.

Client:

- Sends a request, receives data, and maintains a buffer for out-of-order packets.
- Sends cumulative ACK with up to two SACK blocks upon receipt of each packet. The first SACK block is the range of seq numbers received by the client such that he received packet lies in it. The second SACK block is the first range after the expected seq number which has been received by the client.
- Upon receiving EOF, it sends 5 ACKS and closes connection.

Server:

- Splits file into 1180-byte chunks and transmits using a byte-based sliding window (SWS).
- Maintains timers for each unacked packet.
- For each newly ACKed packet, it sends a new packet in flight so that the total number of bytes in flight is never less than the SWS (this can be done because the client buffer is assumed to be infinite)
- Upon receiving a triple duplicate ACK, the server retransmits the unacked packet with the minimum sequence number
- Upon timeout (no ACK received for a certain period of time), the server checks for timed out packets among the unacked packets and retransmits them.

- Estimates RTT and updates RTO dynamically:

$$\text{RTO} = \text{EstimatedRTT} + 4 \times \text{DevRTT}.$$

  bounded in [0.05, 5.0] s.

3. Enhancements and Design Choices

- **Selective Acknowledgment:** Two SACK blocks reduce unnecessary retransmissions.
- **Adaptive Timeout:** Dynamic RTO based on RTT samples.
- **Reliable EOF:** EOF marker retransmitted until acknowledged.

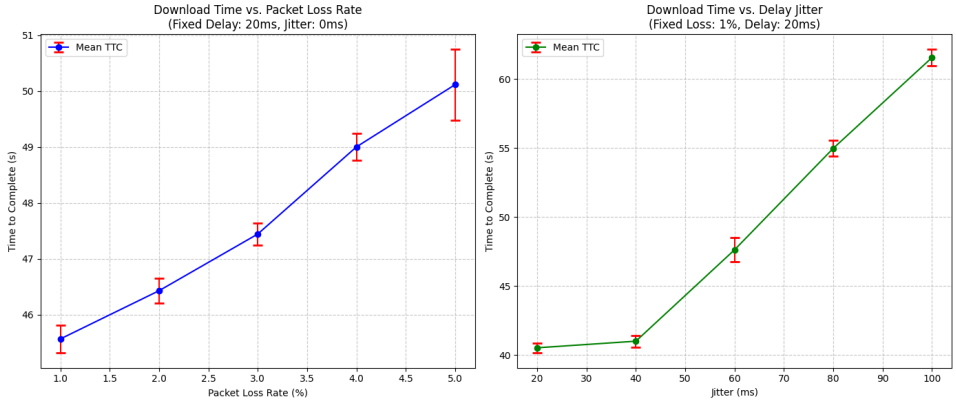

Figure 1: Plots for Part 1.

4. Observations

Based on the plots in Figure 1 (infinite bandwidth):

Impact of Packet Loss (Left Plot)

- **Linear Correlation:** There is a clear, linear relationship between packet loss and download time. As loss increases from 1% to 5%, the Time to Complete (TTC) steadily increases from $\approx$ 45s to $>$ 50s.
- **High Consistency:** TTC for 5 % packet loss has a large confidence interval due to more noise in the results.

Impact of Delay Jitter (Right Plot)

- **Significant Negative Impact:** Increasing jitter has a substantial negative effect on download time, particularly at higher values.

- **Non-Linear Effect:** The impact is not linear. TTC is stable for low jitter (20-40ms, ≈40-41s), but degrades sharply after 40ms.
- **Performance Threshold:** A performance threshold appears to be crossed after 40ms, with TTC rising rapidly to over 61s at 100ms jitter. This suggests high RTT variability severely impacts the protocol's efficiency.
- The confidence interval at 60ms is notably wider, indicating more noise at this value of jitter.

Congestion Control Algorithm

Overview. The server implements TCP CUBIC-inspired congestion control to set its send window ('cwnd') in bytes. Growth follows CUBIC's cubic function of time since the last congestion epoch; on loss it reduces 'cwnd' and enters recovery behaviour. The implementation uses an MSS = 1180 B (DATA_SIZE) as the unit for window adjustments. Initial cwnd is 1 MSS.

Core CUBIC equations.

$$K \;=\; \left(\frac{W_{\max}\cdot\beta}{C}\right)^{1/3} \quad \text{(computed at epoch start)}$$

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

The sender sets

$$\text{cwnd} \leftarrow \max(10\text{*MSS},\ \min(W(t),\ W_{\text{cap}}))$$

In the code: $C = 10000$, $\beta = 0.3$, $W_{\max}$ is the last pre-congestion cwnd, and cwnd is clamped between 10 MSS and a configured maximum.

Epoch management. When a congestion event occurs the epoch is reset. Upon receiving an ack beyond the loss episode (i.e., when we exit from recovery) the epoch starts, K is computed and time t measured from that start for the cubic growth.

Loss / congestion handling.

- **Timeout (severe):** Treat as severe congestion — set $W_{\max} \leftarrow$ cwnd/2, reduce 'cwnd' to 10 MSS, reset epoch.
- **Triple duplicate ACK (fast retransmit):** On 3 duplicate ACKs the server:
  - Fast-retransmits the oldest unacked packet.
  - Sets $W_{\max} \leftarrow$ cwnd and reduces 'cwnd' multiplicatively (factor CWND_DEC, here 0.5).
  - Enters a recovery mode where cubic growth is suspended.

Recovery While 'in_recovery':

- The standard cubic update is skipped (no growth from the cubic function).

- checks for timeout of unacked packets upon the receipt of every 100th ACK; and retransmits the ones that are timed out
- The code allows limited incremental increases (adds one MSS per duplicate ACK) to probe for delivery during fast recovery.
- The sender exits recovery once cumulative ACKs acknowledge beyond lossy episode.

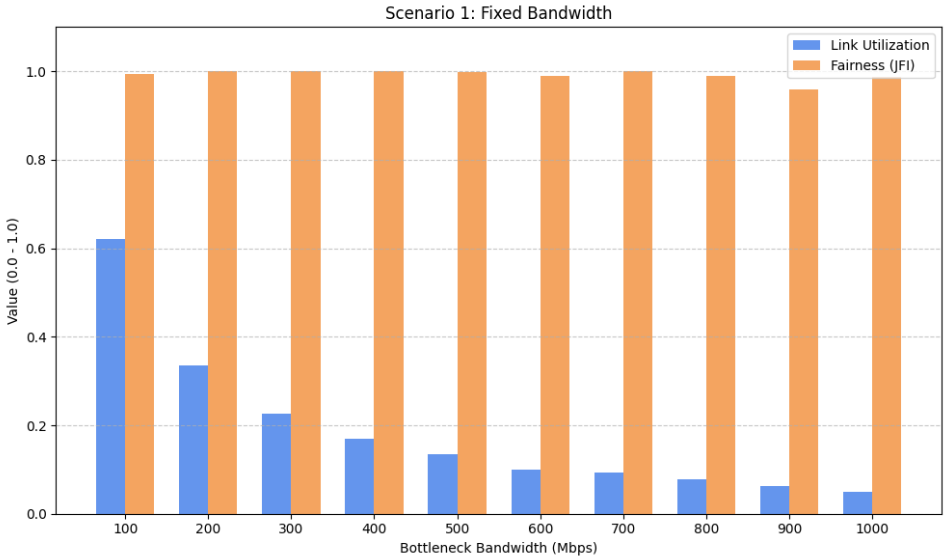

Figure 2: Plots for fixed bandwidth scenario.

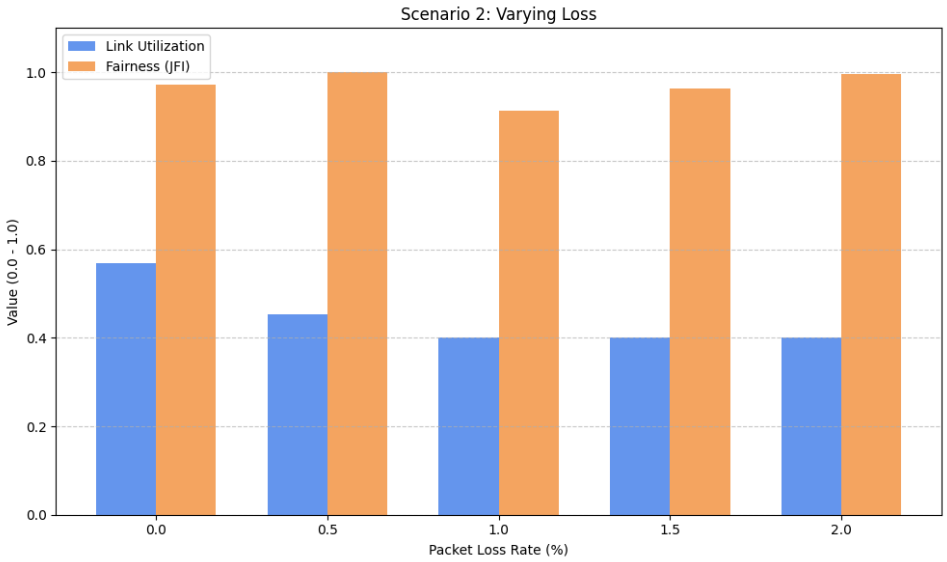

Figure 3: Plots for varying loss scenario.

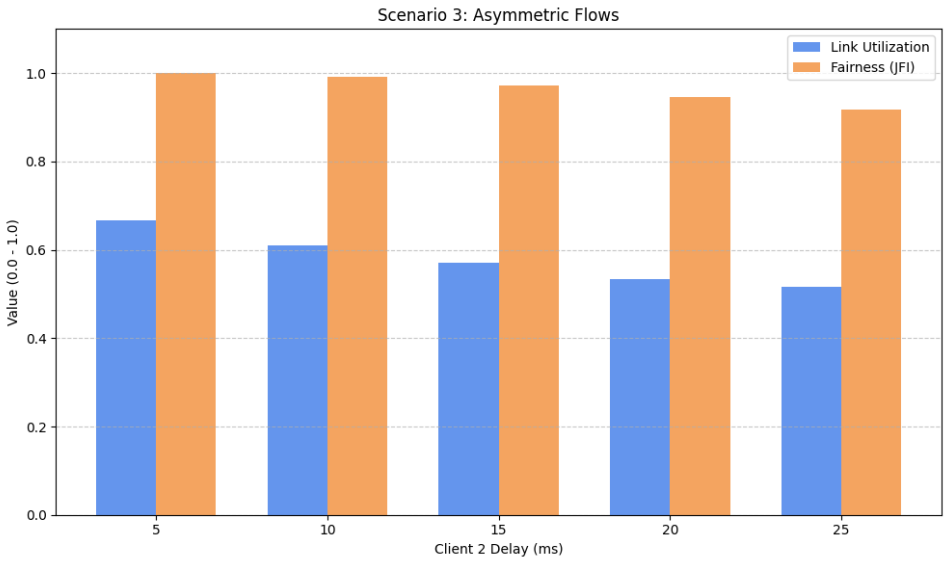

Figure 4: Plots for asymmetric flows scenario.

Observations

Scenario 1: Fixed Bandwidth (Figure 2)

- **Link Utilization:** Utilization is highest (0.62) at the 100 Mbps bottleneck and decreases as bandwidth increases. This is because the file size is fixed. At 100 Mbps, the link is the bottleneck, allowing CUBIC time to ramp up and utilize the link. At higher bandwidths (e.g., 1000 Mbps), the transfer completes before the CUBIC growth function ($W(t)$), which is time-dependent, can expand the congestion window large enough to saturate the pipe.

- **Fairness:** JFI is excellent (> 0.98) because both flows start simultaneously and face the same conditions. They grow together in the CUBIC concave region without significant congestion, leading to equal bandwidth sharing.

Scenario 2: Varying Loss (Figure 3)

- **Link Utilization:** Utilization drops from 0.57 (0% loss) to 0.40 (2.0% loss). This is a direct result of CUBIC's 'on_congestion_event'. Each loss, likely detected via triple duplicate ACK, triggers a fast retransmit, setting $W_{\max}$ and reducing 'cwnd' by a factor of 0.5. More frequent loss events mean more frequent window reductions and recovery phases, leading to a lower average 'cwnd' and thus lower throughput.

- **Fairness:** JFI dips at 1.0% and 1.5% loss. This indicates that random loss can affect one flow more than the other, causing it to reduce its window while the other continues to grow, creating a temporary imbalance. At 2.0% loss, both flows are penalized frequently, paradoxically bringing their (reduced) throughputs closer together and improving fairness.

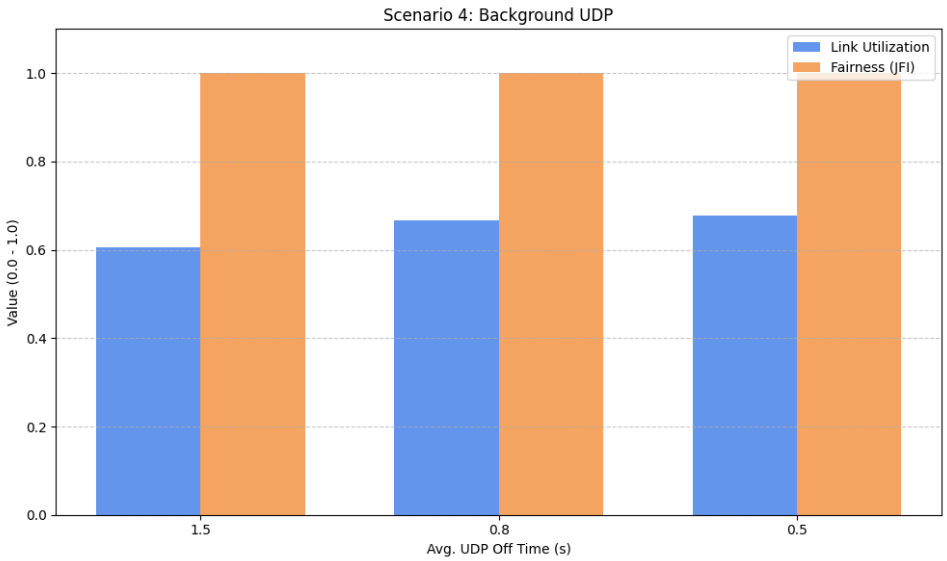

Figure 5: Plots for background udp scenario.

Scenario 3: Asymmetric Flows (Figure 4)

- **Link Utilization:** Overall utilization drops as the RTT for Client 2 increases. This is because the link is underutilized while waiting for the long-RTT flow (Client 2) to acknowledge data or recover from loss.

- **Fairness:** JFI (Fairness) degrades significantly, from ≈1.0 to ≈0.92, as RTT asymmetry increases. The short-RTT flow (Client 1) has a major advantage: its ACKs return faster, allowing it to detect triple duplicate ACKs and exit recovery much more quickly. The long-RTT flow is more likely to suffer a full RTO, which CUBIC treats as a *severe* event, resetting its 'cwnd' to 10 MSS. This stalls the long-RTT flow, allowing the short-RTT flow to dominate the link.

Scenario 4: Background UDP (Figure 5)

- **Link Utilization:** As the UDP "off time" increases (meaning the UDP traffic is less aggressive), the link utilization for our reliable flows improves, rising from 0.61 to 0.68. The aggressive UDP traffic (low off-time) induces packet loss, forcing our CUBIC flows into 'on_congestion_event' more often. Longer off-times provide quiet periods for CUBIC to execute its growth function and increase the 'cwnd'.

- **Fairness:** JFI remains near-perfect (≈1.0). The UDP traffic acts as random "noise" on the line, and this loss is distributed equally between the two reliable flows. Since both flows experience similar loss rates, their CUBIC algorithms are penalized and recover at roughly the same average rate, maintaining fairness.