Design, Implementation, and Testing of a Pipelined Reliable Transfer Protocol

Neel Sadafule and Dylan Cantafio

December 2, 2024

1 Introduction

In this project, we designed and implemented a reliable data transfer protocol over UDP sockets in Python. The protocol incorporates flow control and congestion control mechanisms to ensure efficient and reliable communication over an unreliable network. Additionally, we simulated packet loss and corruption to test the robustness of our protocol under adverse network conditions.

2 Protocol Design and Implementation

2.1 Overview

Our protocol aims to provide reliable, connection-oriented communication on top of UDP, which is inherently unreliable and connectionless. To achieve this, we implemented features commonly found in TCP, such as sequence numbering, acknowledgments, checksums for error detection, flow control using a sliding window, and congestion control mechanisms.

2.2 Packet Structure

Each packet in our protocol consists of:

- **Packet Type (1 byte):** Indicates the type of packet:
  - `DATA` (0)
  - `SYN` (1)
  - `ACK` (2)
  - `FIN` (3)
- **Sequence Number (4 bytes):** A 32-bit number used to order packets.
- **Checksum (2 bytes):** A 16-bit checksum for error detection.
- **Payload (variable length):** The actual data being transmitted.

2.3 Connection Management

2.3.1 Connection Establishment

We implemented a three-way handshake to establish a connection:

1. **SYN:** The sender initiates the connection by sending a `SYN` packet with sequence number 0.
2. **SYN-ACK:** The receiver responds with a `SYN` packet (serving as `SYN-ACK`).
3. **ACK:** The sender sends an `ACK` packet to complete the handshake.

2.3.2 Connection Termination

Connection termination is handled using a two-way handshake:

1. **FIN:** The sender sends a `FIN` packet to initiate termination.
2. **FIN-ACK:** The receiver responds with a `FIN` packet (`FIN-ACK`).

2.4 Reliability Mechanisms

2.4.1 Sequence Numbers and Acknowledgments

- **Sequence Numbers:** Each data packet is assigned a unique sequence number to ensure proper ordering.
- **Acknowledgments:** The receiver sends `ACK` packets with the sequence number of the received packet to confirm successful reception.

2.4.2 Checksums for Error Detection

We compute a checksum by summing the packet type, sequence number, and payload bytes. The receiver recalculates the checksum upon receiving a packet to detect any corruption.

2.4.3 Timeouts and Retransmissions

- The sender starts a timer for each packet sent.
- If an acknowledgment is not received within a specified timeout interval, the packet is retransmitted.

2.5 Flow Control

We implemented flow control using a sliding window protocol:

- The sender maintains a window of unacknowledged packets.
- The window size is controlled by the congestion window (`cwnd`), ensuring the sender does not overwhelm the receiver.
- Multiple in-flight packets are allowed within the sliding window, ensuring efficient utilization of the network while maintaining reliability. Once an acknowledgment (`ACK`) is received, the window slides forward to include new packets.

2.6 Congestion Control

Our congestion control mechanism is inspired by TCP's slow start and congestion avoidance phases. The congestion window (`cwnd`) is dynamically adjusted based on network conditions to balance efficient data transfer and avoid congestion.

- **Slow Start Phase:** During the slow start phase, the `cwnd` increases exponentially as packets are successfully acknowledged by the receiver.
- **Congestion Avoidance Phase:** Once `cwnd` exceeds the slow start threshold (`ssthresh`), the protocol transitions to congestion avoidance. Here, `cwnd` increases linearly, ensuring network stability and avoiding excessive traffic.
- **Packet Loss Handling:** When packet loss is detected (e.g., through a timeout or duplicate ACKs), `ssthresh` is set to half of the current `cwnd`, and `cwnd` is reset to 1. This ensures a gradual recovery while minimizing the risk of further congestion.

2.7 Simulation of Packet Loss and Corruption

To test the protocol under adverse conditions, we simulated packet loss and corruption:

- **Packet Loss:** Implemented by randomly dropping packets with a certain probability (`LOSS_PROBABILITY`).
- **Packet Corruption:** Simulated by randomly altering packets with a certain probability (`ERROR_PROBABILITY`). Corrupted packets are detected using checksums and discarded by the receiver.

3 Testing and Results

3.1 Testing Environment

- **Hardware and Software:**
  - MacBook Pro running macOS.
  - Python 3.x for both `sender.py` and `receiver.py`.
- **Testing Tools:**
  - Wireshark for capturing and analyzing network traffic.
  - `tcpdump` for capturing packets on the loopback interface. (`sudo tcpdump -i lo0 -w lo0_capture.pcap`)
  - `wget` execute a large file download. (`wget http://speedtest.tele2.net/10MB.zip`)

3.2 Test Procedures

We conducted tests to verify the correctness and performance of our protocol under various network conditions.

1. **Baseline Test:**

- Set `LOSS_PROBABILITY` and `ERROR_PROBABILITY` to 0.
- Verified that data transfer occurs correctly without any loss or corruption.

2. **Packet Loss Simulation:**
   - Set `LOSS_PROBABILITY` to 0.1.
   - Observed the sender retransmitting packets upon timeouts.
   - Verified that all data packets were eventually received and acknowledged.
3. **Packet Corruption Simulation:**
   - Set `ERROR_PROBABILITY` to 0.05.
   - Observed the receiver detecting corrupted packets and discarding them.
   - Verified that the sender retransmitted the corrupted packets after timeouts.
4. **Combined Loss and Corruption:**
   - Tested with both `LOSS_PROBABILITY` and `ERROR_PROBABILITY` set to non-zero values.
   - Ensured the protocol handled both lost and corrupted packets effectively.

3.3 Results and Analysis

3.3.1 Connection Establishment and Termination

The three-way handshake successfully established a connection before data transfer, and connection termination was handled gracefully after all data was transmitted.

1	0.000000	127.0.0.1	127.0.0.1	UDP	39 53312 → 12345 Len=7
2	0.001612	127.0.0.1	127.0.0.1	UDP	39 12345 → 53312 Len=7
3	0.004011	127.0.0.1	127.0.0.1	UDP	39 53312 → 12345 Len=7

Figure 1: Wireshark Capture of the Three-Way Handshake. The capture highlights the SYN, SYN-ACK, and ACK packets exchanged between the sender (port 53312) and the receiver (port 12345).

449	65.412556	127.0.0.1	127.0.0.1	UDP	39 53312 → 12345 Len=7
450	65.412772	127.0.0.1	127.0.0.1	UDP	39 12345 → 53312 Len=7

Figure 2: Wireshark Capture of the Connection Termination. The capture shows the FIN packet sent by the sender and the FIN-ACK packet sent by the receiver to gracefully terminate the connection.

3.3.2 Data Transfer and Reliability

The sender adjusted the congestion window (`cwnd`) based on network conditions. The sliding window mechanism effectively managed flow control, and all data segments were eventually received in order, demonstrating reliability.

```
236 33.214498    127.0.0.1       127.0.0.1       UDP      54 53312 → 12345 Len=22
237 34.217745    127.0.0.1       127.0.0.1       UDP      54 53312 → 12345 Len=22
238 34.218855    127.0.0.1       127.0.0.1       UDP      54 53312 → 12345 Len=22
239 34.219204    127.0.0.1       127.0.0.1       UDP      39 12345 → 53312 Len=7
```

Figure 3: Illustration of the Sliding Window Protocol in Operation. Packets 236, 237, and 238 represent in-flight packets sent consecutively without waiting for acknowledgments. Packet 239 is an acknowledgment sent from the receiver confirming successful reception of earlier packets.

3.3.3 Congestion Control Behavior

We observed the congestion window (`cwnd`) increasing exponentially during the slow start phase and linearly during the congestion avoidance phase. Upon detecting packet loss, `cwnd` was reduced, and the protocol re-entered slow start.

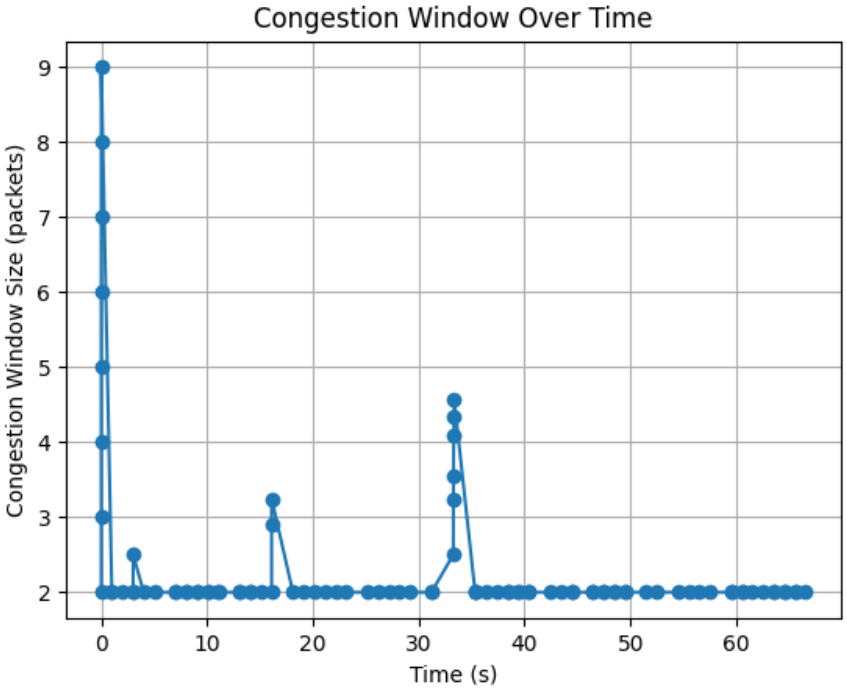

Figure 4: Congestion Window (`cwnd`) Over Time. The graph highlights exponential growth during slow start and linear growth during congestion avoidance.

- **Initial Growth:** The `cwnd` increases rapidly, demonstrating the slow start phase, where the congestion window grows exponentially until it reaches the slow start threshold (`ssthresh`).
- **Packet Loss Response:** After packet loss or congestion is detected (indicated by a timeout), the congestion window size drops sharply to its minimum and re-enters the slow start phase.
- **Periodic Spikes:** The periodic spikes in `cwnd` around 10 and 30 seconds reflect moments when the network briefly allows more packets before encountering congestion, at which point `cwnd` is reduced.

- **Stabilization:** Over time, `cwnd` stabilizes at lower levels, showing the protocol's adaptability to the network's capacity and its ability to prevent further congestion.

Additionally, we captured a screenshot of the congestion window behavior during slow start, highlighting the initial exponential growth:

```
13 1.008888    127.0.0.1    127.0.0.1    UDP    53 56597 → 12345 Len=21
14 1.009209    127.0.0.1    127.0.0.1    UDP    39 12345 → 56597 Len=7
15 1.010393    127.0.0.1    127.0.0.1    UDP    53 56597 → 12345 Len=21
16 1.010844    127.0.0.1    127.0.0.1    UDP    39 12345 → 56597 Len=7
17 1.011875    127.0.0.1    127.0.0.1    UDP    53 56597 → 12345 Len=21
18 1.012153    127.0.0.1    127.0.0.1    UDP    39 12345 → 56597 Len=7
19 1.012641    127.0.0.1    127.0.0.1    UDP    53 56597 → 12345 Len=21
20 1.012945    127.0.0.1    127.0.0.1    UDP    53 56597 → 12345 Len=21
21 2.016759    127.0.0.1    127.0.0.1    UDP    53 56597 → 12345 Len=21
22 2.017066    127.0.0.1    127.0.0.1    UDP    39 12345 → 56597 Len=7
23 2.017349    127.0.0.1    127.0.0.1    UDP    53 56597 → 12345 Len=21
24 2.017480    127.0.0.1    127.0.0.1    UDP    39 12345 → 56597 Len=7
25 2.018258    127.0.0.1    127.0.0.1    UDP    53 56597 → 12345 Len=21
26 2.018425    127.0.0.1    127.0.0.1    UDP    39 12345 → 56597 Len=7
```

Figure 5: Congestion Window (`cwnd`) During Slow Start. Packets 13–26 demonstrate exponential growth of `cwnd`, evidenced by the rapid transmission and minimal gaps in timestamps.

3.3.4 Latency (Round-Trip Time - RTT)

Latency, measured as the round-trip time (RTT) for each packet, was observed to increase during periods of congestion or retransmissions. Figure 6 shows RTT spikes corresponding to packet retransmissions and congestion events, reflecting the protocol's handling of adverse network conditions.

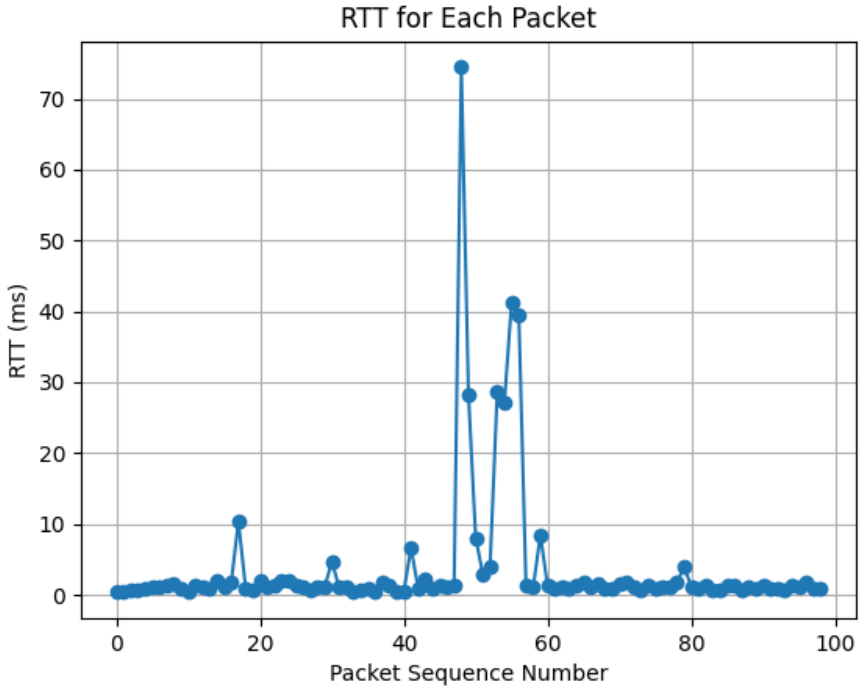

Figure 6: RTT for Each Packet. Spikes indicate increased latency during packet retransmissions and congestion.

- **Consistent RTTs:** For most packets, the RTT remains consistent and low, indicating smooth transmission conditions.
- **Spikes in RTT:** Spikes are observed for certain packets, especially around sequence numbers 40–60. These spikes likely correspond to:
  - **Retransmissions:** Due to packet loss or corruption, additional time is required to send and acknowledge a packet.
  - **Congestion:** Higher network congestion increases delays in receiving acknowledgments, leading to elevated RTT values.
- **Stabilization After Spikes:** After the spikes, the RTT stabilizes, demonstrating that the protocol efficiently adapts to network conditions and mitigates the effects of congestion or packet loss.

3.3.5 Throughput

The throughput, measured in bits per second (bps), shows how efficiently the protocol utilized the available network capacity over time. As illustrated in Figure 7, throughput varies with network conditions.

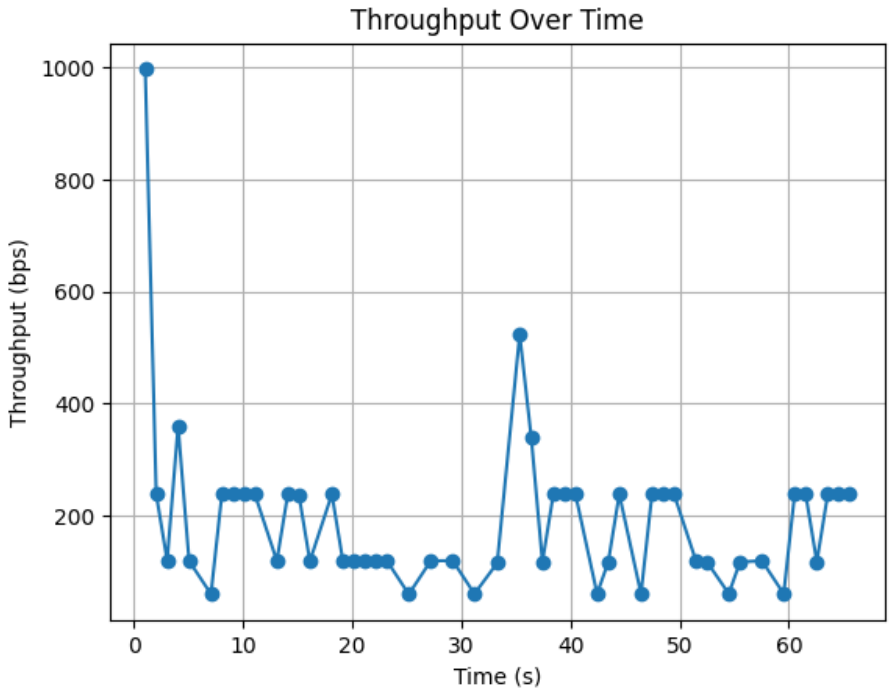

Figure 7: Throughput Over Time. The graph shows fluctuations caused by network conditions and retransmissions.

- **High Initial Throughput:** The initial throughput is very high, reflecting the rapid transmission of data during the early stages when `cwnd` is increasing.

- **Drops Due to Retransmissions:** The drops in throughput correspond to periods of packet loss or retransmissions, where effective transmission temporarily decreases as the protocol compensates for lost or corrupted packets.
- **Spikes Corresponding to `cwnd` Expansion:** Spikes in throughput, such as around 30 seconds, align with moments when the congestion window briefly expands, allowing more packets to be sent and acknowledged successfully.
- **Stabilization Over Time:** The overall trend shows a decrease in throughput over time as the protocol stabilizes in response to network conditions, maintaining reliable communication rather than maximizing speed.

3.3.6 Handling of Packet Loss and Corruption

The protocol successfully detected and handled packet loss through retransmissions. Packet corruption was detected using checksums, and corrupted packets were retransmitted.

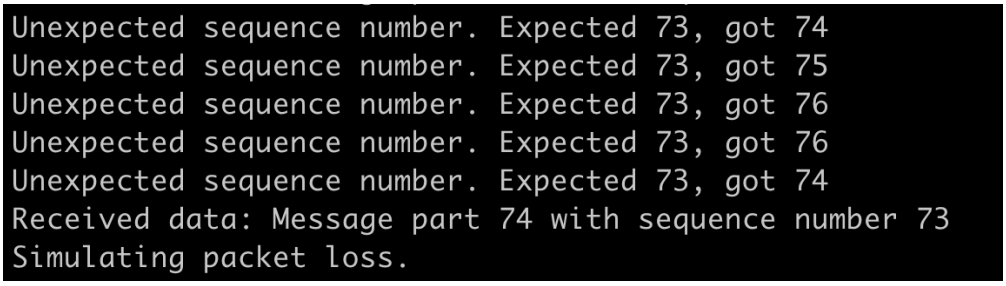

Figure 8: Logs showing the receiver simulating packet loss for sequence number 73.

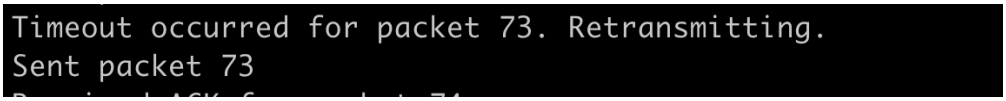

Figure 9: Logs showing the sender retransmitting packet 73 after a timeout.

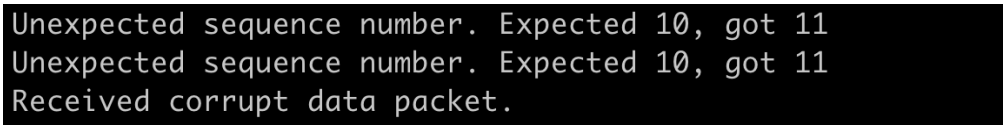

Figure 10: Logs showing the receiver discarding a corrupted packet with sequence number 10.

```
Timeout occurred for packet 10. Retransmitting.
Sent packet 10
Timeout occurred for packet 11. Retransmitting.
Sent packet 11
Timeout occurred for packet 10. Retransmitting.
Sent packet 10
Received ACK for packet 10
```

Figure 11: Logs showing the sender retransmitting packet 10 after detecting corruption.

The results demonstrate that the protocol successfully handles both packet loss and corruption:

- **Robustness to Packet Loss:** The sliding window mechanism ensures that lost packets are retransmitted, maintaining reliability.
- **Error Detection and Correction:** Checksums effectively detect corrupted packets, and retransmissions ensure data integrity.
- **Adaptation to Network Conditions:** The congestion control algorithm effectively adjusts the sending rate based on network feedback, balancing efficiency and fairness.

3.4 Fairness and Coexistence

To evaluate the fairness of our protocol, we conducted additional tests where it operated concurrently with standard TCP traffic generated by separate ‘wget‘ commands downloading the file from [Speedtest](http://speedtest.tele2.net/10MB.zip). This setup allowed us to observe how our protocol behaves under realistic network conditions with competing traffic.

3.4.1 Congestion Window Over Time

The plot **”Congestion Window Over Time”** (Figure 12) reflects the dynamic adjustments of the congestion window (`cwnd`) as the protocol interacts with both network conditions and competing TCP traffic from three ‘wget‘ commands.

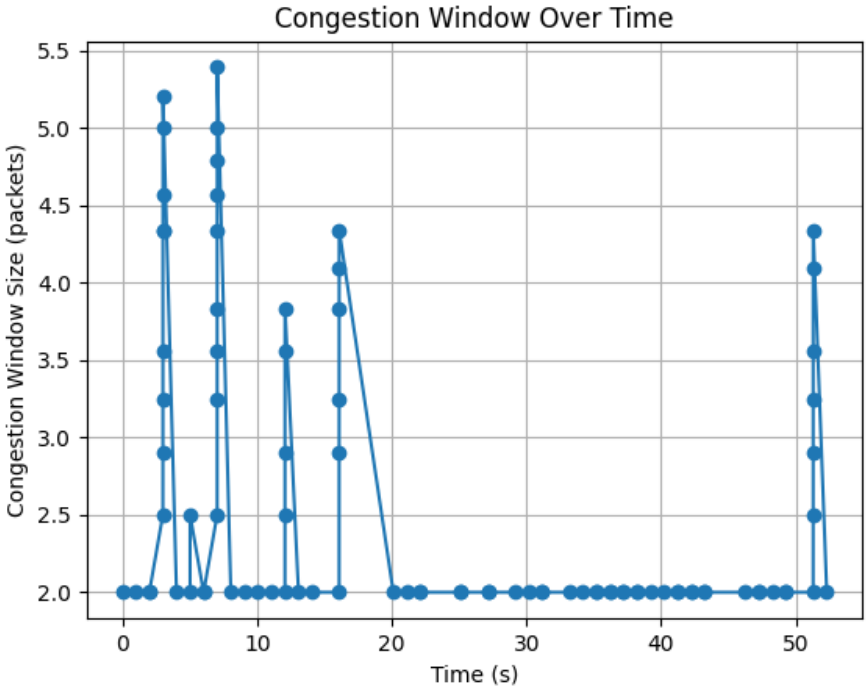

Figure 12: Congestion Window (`cwnd`) Over Time with Concurrent TCP Traffic. The plot shows how `cwnd` adjusts in response to competing ‘wget‘ traffic, demonstrating fairness in bandwidth sharing.

- **Initial Behavior and Slow Start Phase:** At the beginning of the transfer, the congestion window increases exponentially during the slow start phase, peaking at approximately 5.5 packets. This phase occurs before any significant network contention from the first ‘wget‘ command, allowing the protocol to probe the network's available bandwidth effectively.
- **Impact of the First wget Command:** Around the 5-second mark, the congestion window begins to drop sharply. This corresponds to the first ‘wget‘ download introducing TCP traffic into the network. The congestion window stabilizes between 2 and 4 packets, indicating that the protocol has detected increased network congestion and transitioned to the congestion avoidance phase.
- **Subsequent TCP Traffic:** Each successive ‘wget‘ command causes similar drops in the congestion window, showcasing the protocol's responsiveness to bursts of competing traffic. Between these drops, the congestion window remains stable at a lower level, allowing fair coexistence with the TCP flows.
- **Recovery and Final Behavior:** After the last ‘wget‘ command completes, the congestion window increases again, peaking toward the end of the session. This recovery demonstrates the protocol's ability to reclaim bandwidth when network conditions improve.

3.4.2 Round-Trip Time (RTT) for Each Packet

The plot **"RTT for Each Packet"** (Figure 13) illustrates the variations in Round-Trip Time (RTT) for packets during the session with concurrent TCP traffic.

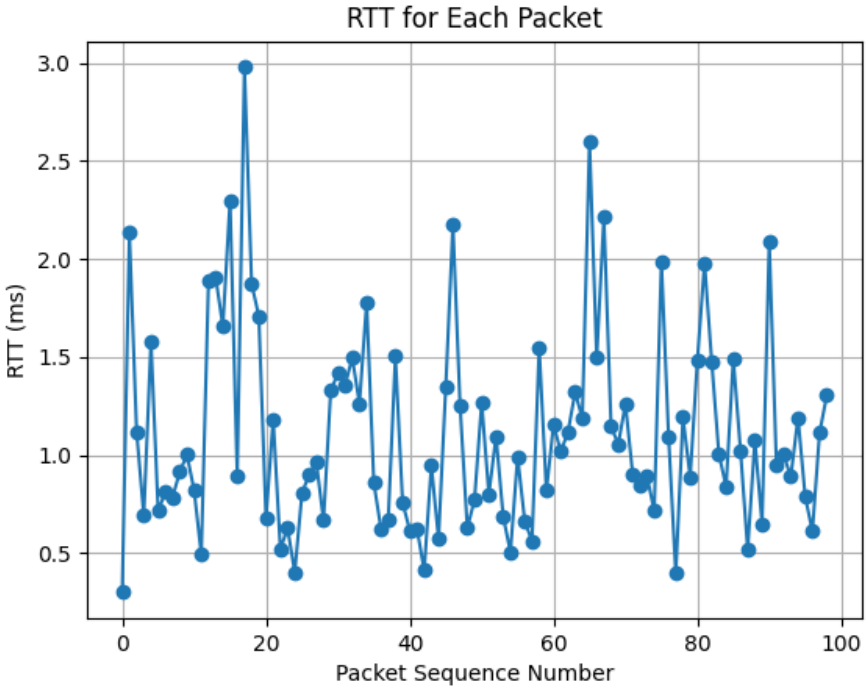

Figure 13: RTT for Each Packet with Concurrent TCP Traffic. The plot shows RTT spikes corresponding to periods of high network contention due to ‘wget‘ downloads.

- **Baseline RTT:** During the initial phase of the transfer (before the first ‘wget‘ command), RTT values are consistently low, indicating minimal network congestion.
- **Increased RTT During TCP Traffic:** Each ‘wget‘ command introduces competing traffic, causing RTT spikes of up to 3 ms. These spikes align with the drops in the congestion window in the first plot, confirming that the additional TCP traffic increased network contention.
- **Stabilization After TCP Traffic:** Once the competing ‘wget‘ traffic subsides, RTT values stabilize, with minor fluctuations. This demonstrates that the protocol recovers well after periods of contention, maintaining low latency when the network load decreases.

3.4.3 Throughput Over Time

The plot **"Throughput Over Time"** (Figure 14) highlights the protocol's ability to transfer data effectively while coexisting with competing TCP traffic.

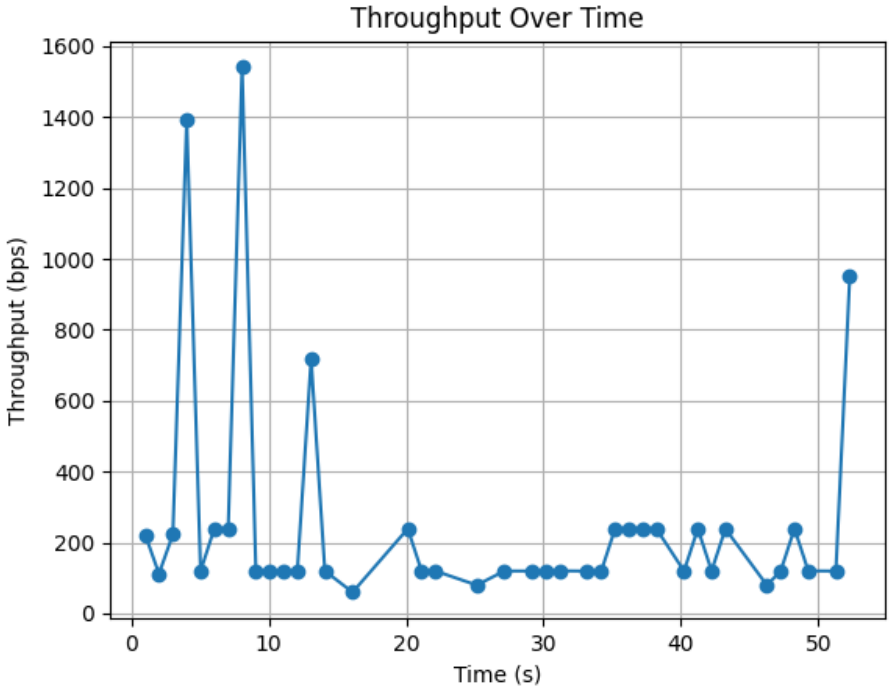

Figure 14: Throughput Over Time with Concurrent TCP Traffic. The plot shows how throughput fluctuates in response to competing ‘wget‘ traffic, reflecting the protocol’s fair bandwidth sharing.

- **Initial High Throughput:** At the start of the transfer, throughput peaks at around 1600 bps, corresponding to the slow start phase of the congestion window. This period occurs before any ‘wget‘ traffic is introduced, allowing the protocol to maximize its use of the network.
- **Impact of TCP Traffic:** Each ‘wget‘ command introduces significant reductions in throughput. These drops correspond to the congestion window decreases in the first plot, as the protocol reduces its sending rate to accommodate the competing TCP flows.
- **Recovery After Congestion:** Once the final ‘wget‘ download completes, the protocol’s throughput increases again, demonstrating its ability to recover bandwidth when the network contention decreases. The final peak in throughput aligns with the increase in the congestion window, indicating efficient utilization of available resources.

4 Conclusion

Our implementation of a pipelined reliable transfer protocol over UDP successfully achieved reliable, ordered, and efficient data transfer. The incorporation of flow control and congestion control mechanisms allowed the protocol to adapt to varying network conditions and maintain acceptable performance. Simulating packet loss and errors verified the robustness of the protocol, demonstrating its ability to handle adverse network scenarios.

Furthermore, we evaluated the fairness and coexistence capabilities of our protocol by running it concurrently with standard TCP traffic generated by separate `wget` commands downloading data

from `http://speedtest.tele2.net/10MB.zip`. The results from the congestion window (`cwnd`), Round-Trip Time (RTT), and throughput plots indicate that our protocol reduces its sending rate during periods of high contention from the `wget` commands. This behavior demonstrates fairness by allowing the competing TCP flows to utilize the network without being starved of bandwidth.

The protocol's responsiveness to network conditions is evident from its dynamic adjustments to the congestion window, RTT stabilization, and throughput recovery in response to the introduction and completion of competing traffic. During periods of concurrent `wget` traffic, the protocol avoids monopolizing bandwidth and shares network resources effectively. Once the competing traffic subsides, it increases its sending rate to utilize the available bandwidth efficiently.

These results provide strong evidence that the protocol is fair and capable of coexisting with standard TCP traffic. The dynamic behavior observed in the plots aligns with the principles of congestion control and fairness, demonstrating that the protocol can adapt to network conditions and share resources equitably.

In summary, our protocol not only ensures reliable and efficient data transfer over an unreliable network but also maintains fairness and adaptability in the presence of competing network traffic. This dual capability underscores the protocol's robustness and suitability for diverse networking environments.