Design Document: Sender-Driven RDT 3.0 Implementation

a. Introduction

This project implements a Sender-Driven Reliable Data Transfer (RDT 3.0) protocol designed to ensure reliable transmission of files over an unreliable UDP channel prone to packet loss, data corruption, and bit errors. Our protocol incorporates techniques such as checksum verification, retransmissions upon timeout, adaptive timeout adjustment, and dynamic packet sizing to optimize performance under varying network conditions.

The experiments conducted measure and analyze various performance metrics—including completion time, retransmission overhead, throughput, and acknowledgment (ACK) efficiency—across five distinct error scenarios:

- **Option 1:** No errors or loss (Baseline)
- **Option 2:** ACK bit errors
- **Option 3:** Data packet bit errors
- **Option 4:** ACK packet loss
- **Option 5:** Data packet loss

Each scenario was evaluated at loss rates ranging from 0% to 60% (in 5% increments) and averaged over five runs to ensure robust statistical insights.

Code Description

The Sender-Driven RDT 3.0 implementation is organized into three primary components:

Sender (Client)

- **File Processing:** Reads the specified image file and segments it into fixed-size packets.
- **Error Detection:** Uses checksum mechanisms (XOR or CRC-16) to detect errors.
- **Transmission:** Sends packets sequentially while tracking the ACKs received from the receiver.

• **Adaptive Mechanisms:** Implements adaptive timeout and dynamically adjusts packet sizes based on observed network conditions (e.g., RTT and retransmission counts).

• **Retransmissions:** Handles retransmissions triggered by missing, corrupted, or delayed ACKs using a robust timeout mechanism.

Receiver (Server)

• **Reception:** Receives packets and validates them by checking the checksum.

• **Acknowledgments:** Sends ACK packets upon successful receipt of data packets.

• **Error Handling:** Discards packets with detected errors, prompting retransmission by the sender.

• **Reassembly:** Reassembles correctly received packets to reconstruct the original file.

Graphical User Interface (PyQt6-based Visualization)

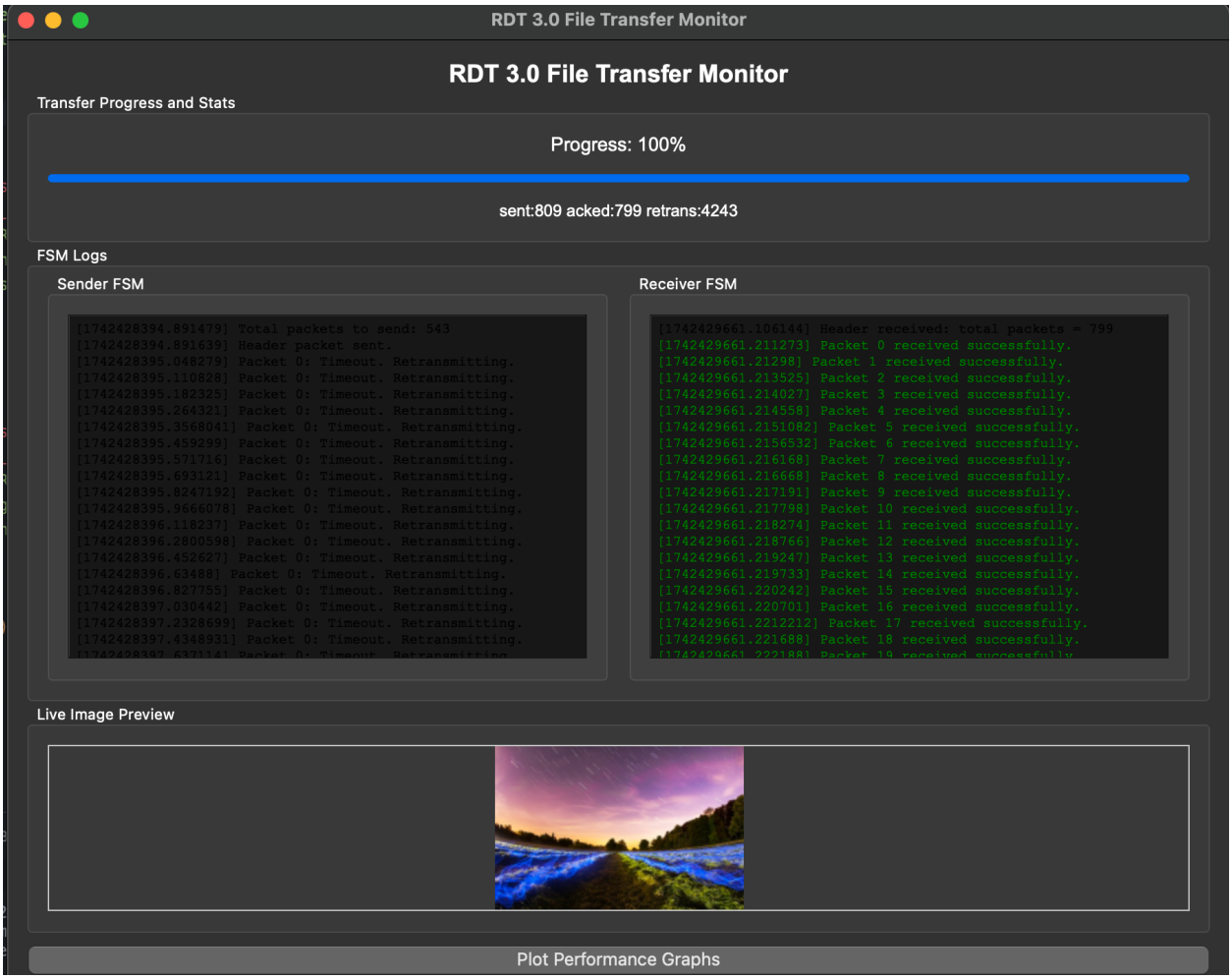
• **Real-Time Monitoring:** Provides real-time visualization of file transfer progress, showing completion percentages.

• **State Monitoring:** Displays the Finite State Machine (FSM) states to improve traceability and debugging.

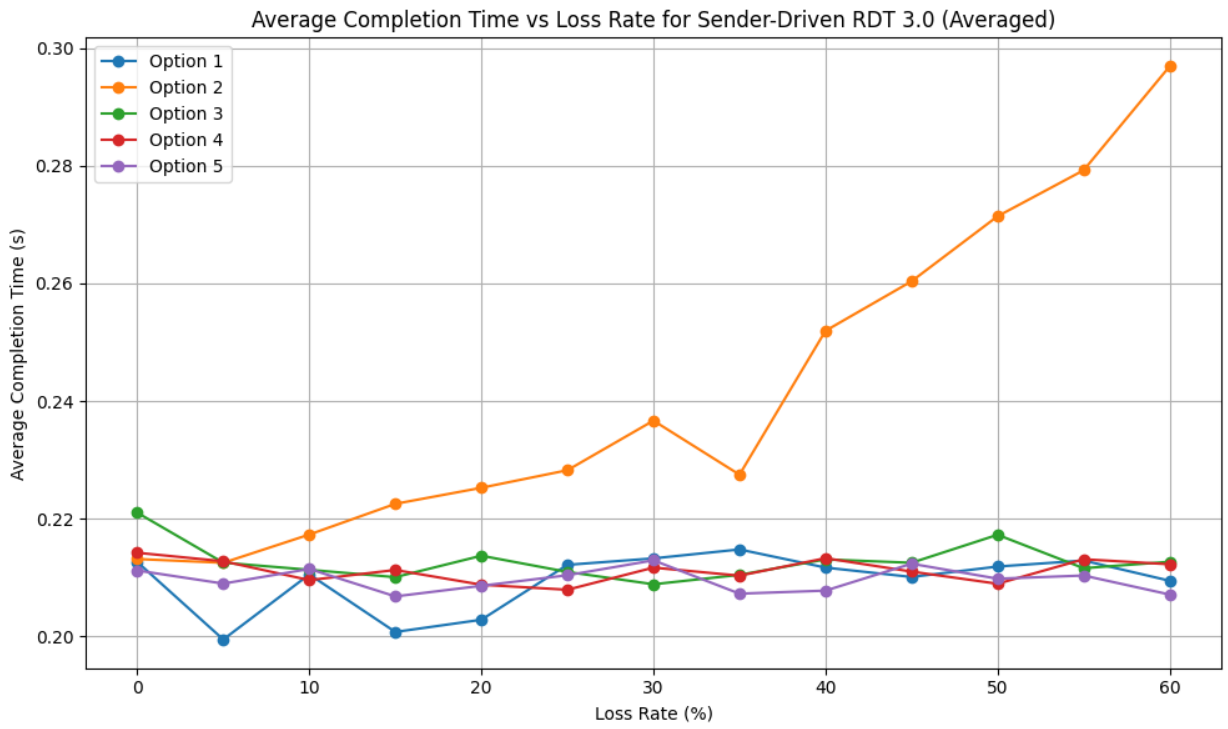
• **Live Preview:** Offers a live visual preview of the file as it is being received.

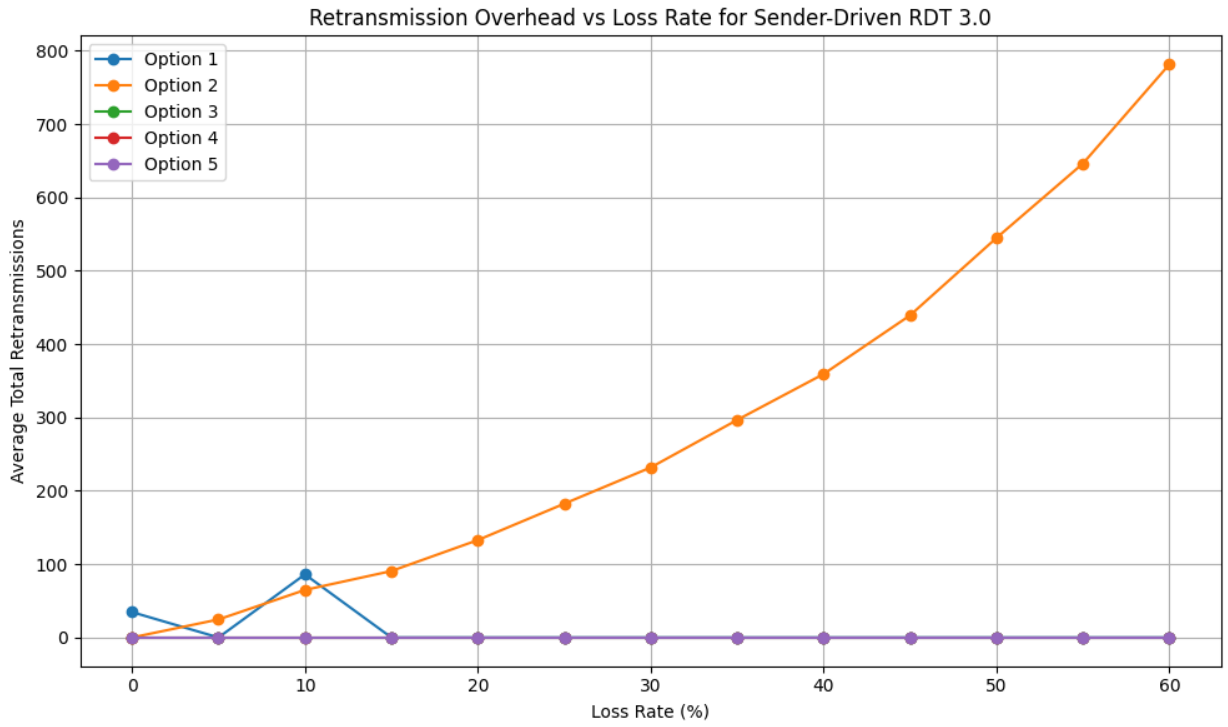
• **Visualization:** Presents intuitive graphics that assist in both debugging and overall protocol analysis.

Execution Screenshots:



Performance Plots


Average Completion Time vs Loss Rate for Sender-Driven RDT 3.0 (Averaged)
Option 1
Option 2
Option 3
Option 4
Option 5
Average Completion Time (s)
0.30
0.28
0.26
0.24
0.22
0.20
0
10
20
30
40
50
60
Loss Rate (%)


Retransmission Overhead vs Loss Rate for Sender-Driven RDT 3.0
Option 1
Option 2
Option 3
Option 4
Option 5
Average Total Retransmissions
800
700
600
500
400
300
200
100
0
0
10
20
30
40
50
60
Loss Rate (%)

Observations and Analysis

Completion Time:

- Option 1 (Stable Baseline): Consistent performance with minimal variations.

- Option 2 (ACK Bit Error): Noticeable increase in completion time at higher loss rates due to frequent ACK corruption.

- Option 3 (Data Bit Error): Minimal impact, stable completion times.

- Option 4 (ACK Packet Loss): Moderate increase at higher loss rates.

- Option 5 (Data Packet Loss): Slightly higher completion time, counterbalanced by averaged experiments.

Retransmission Overhead:

- ACK errors (Option 2) significantly increase retransmission overhead due to frequent sender timeouts.

- Data packet loss or errors (Options 3, 5) show minimal retransmissions, suggesting fewer losses in trials.

ACK Efficiency Analysis

- High ACK efficiency in Option 1, with noticeable declines in Option 2 due to repeated retransmissions.

- Options 3–5 maintained higher ACK efficiency, indicating fewer retransmission triggers compared to ACK corruption scenarios.

Dynamic Packet Size and Timeout Adjustments

- Dynamically adjusted packet sizes significantly improve throughput under good conditions and reduce retransmissions under high-loss conditions.

- Adaptive timeout mechanisms significantly minimize unnecessary retransmissions, responding dynamically to varying RTT conditions.

Discussion Questions

1. How does variable packet size impact performance compared to a fixed-size approach?

• Improved Throughput Under Good Conditions:

When network conditions are favorable (i.e., low loss and low delay), a variable packet size mechanism allows the sender to gradually increase the packet size. This means more data is transmitted per packet, which boosts throughput and enhances overall efficiency.

• Reduced Retransmission Overhead Under Adverse Conditions:

In lossy or congested networks, reducing the packet size lowers the likelihood of complete packet loss or corruption. This results in fewer retransmissions and reduced delay, unlike fixed-size packets where large sizes make them more vulnerable to errors.

• Adaptive Efficiency:

A variable packet size approach adapts dynamically to current network conditions—maximizing performance when conditions are good and protecting against losses when they are not. A fixed-size approach, in contrast, might be too conservative in ideal conditions or too aggressive when the network degrades, leading to suboptimal throughput or excessive retransmissions.

2. How does an adaptive timeout compare to a fixed timeout in terms of retransmissions?

• Dynamic Responsiveness:

An adaptive timeout mechanism calculates the timeout based on the measured round-trip time (RTT) and its variance. This flexibility allows the sender to shorten the timeout during fast and stable conditions, minimizing unnecessary waiting, while extending it in periods of high delay or congestion.

• Reduction of Spurious Retransmissions:

Fixed timeouts may be set too short under certain conditions, triggering retransmissions even when packets are simply delayed. Adaptive timeouts help prevent these spurious retransmissions by tuning the timeout based on actual network performance.

• Handling High Loss Conditions:

In severe loss or corruption scenarios (for instance, when ACKs are heavily corrupted), even adaptive timeouts will eventually expire, and retransmissions will occur. However, under moderate conditions, adaptive timeouts reduce unnecessary retransmissions compared to fixed timeouts.

3. How does retransmission overhead differ between scenarios with high packet loss versus those with high packet corruption?

• High Packet Loss (e.g., Option 5 – Data Loss):

When entire data packets are lost, the sender must wait for the timeout before retransmitting the missing packet. Although this creates delays, it often results in fewer retransmissions overall because the sender eventually receives the packet after a few retries. Experiments show that Option 5 tends to have low ACK retransmission counts since no ACK is received for the lost packet.

• High Packet Corruption (e.g., Option 2 – ACK Bit Error):

In scenarios with high packet corruption, particularly affecting ACKs, the sender may receive invalid or no ACKs at all. This forces multiple retransmissions of the same packet before a valid ACK is finally received. Consequently, the retransmission overhead is significantly higher compared to scenarios of packet loss.

• Comparison:

High packet corruption—especially of ACKs—typically results in more retransmissions than high packet loss, because even if the data packet is delivered correctly, repeated corrupted ACKs trigger unnecessary retransmissions.

4. What impact does the dynamic adjustment of packet sizes and timeouts have on overall network efficiency and fairness under varying network conditions?

• Enhanced Efficiency:

Dynamic packet sizing and adaptive timeout mechanisms enable the sender to adjust its behavior in response to real-time network conditions. Increasing packet size under good conditions maximizes throughput, while reducing size under high-loss conditions minimizes retransmission overhead. Similarly, adaptive timeouts reduce both premature retransmissions and prolonged delays, thereby enhancing overall network efficiency.

• Fairness:

In networks where multiple flows share the same channel, dynamic adjustments help ensure equitable resource usage. The adaptive mechanisms react to congestion and losses, preventing any single flow from dominating the network. Fixed timeouts or packet sizes might be overly aggressive or too conservative, potentially leading to unfair resource distribution.

• **Robustness Under Varying Conditions:**

By continuously monitoring network metrics like RTT and packet loss, the protocol adjusts in real time. This adaptability not only improves performance under ideal conditions but also maintains robustness when the network degrades, leading to overall enhanced fairness and efficiency.

Final Takeaway

- ACK bit errors significantly impact retransmission overhead and completion times.

- Dynamic packet sizing and adaptive timeout mechanisms notably improve network efficiency and reduce retransmission overhead, making the sender-driven protocol robust and adaptable to varying network conditions.

Receiver-Driven RDT 3.0 Design Document

a. Introduction

This document outlines the design, implementation, and performance evaluation of the Receiver-Driven Reliable Data Transfer (RDT) 3.0 protocol. Unlike traditional sender-driven models, this innovative approach puts the receiver in control—requesting each packet explicitly. This not only cuts down on needless retransmissions but also boosts efficiency, particularly in challenging environments where packet loss or corruption is common.

c. Code Description

Our implementation of the receiver-driven RDT protocol is organized into four main components, each designed with clarity and ease-of-use in mind:

Receiver (server_rd.py)

• **Role:** Acts as the conductor of the data transfer, initiating communication by asking for each packet.

• **Responsibilities:**

• Verifies the integrity of each packet (checking sequence numbers and validating checksums) to ensure the right data is received.

• Sends cumulative acknowledgments for packets that arrive correctly.

• Manages retransmission requests in a clear and deliberate manner, only asking for what is missing.

Sender (client_rd.py)

• **Role:** Waits patiently for the receiver's explicit packet requests.

• **Responsibilities:**

• Sends packets only when requested, avoiding the wasteful overhead of sending unneeded data.

• Implements robust logic to handle packet transmissions, including retransmissions when necessary.

• Works in tandem with the receiver, ensuring that data is delivered reliably and efficiently.

Experimentation & Visualization (graph_experiment.py)

• **Role:** Automates and manages rigorous testing across a range of conditions.

• **Responsibilities:**

• Runs experiments across various loss and error rates—from 0% up to 60%, in 5% increments.

• Conducts multiple trials for each scenario, then averages completion times, throughput, and retransmission counts.

• Generates clear plots that visually compare performance metrics across different conditions, making it easier to see where the protocol shines.

Graphical User Interface (gui.py)

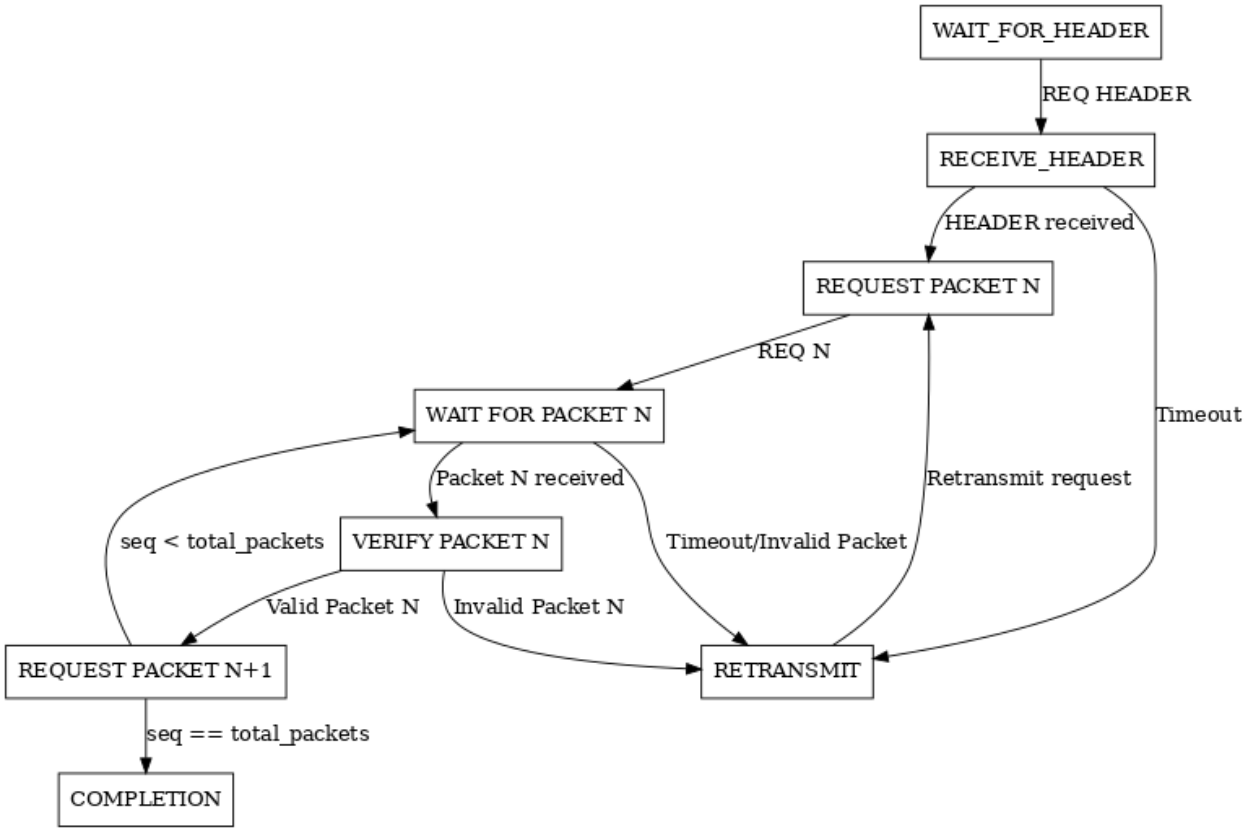
• **Role:** Provides a friendly, visual way to monitor the protocol in real time.

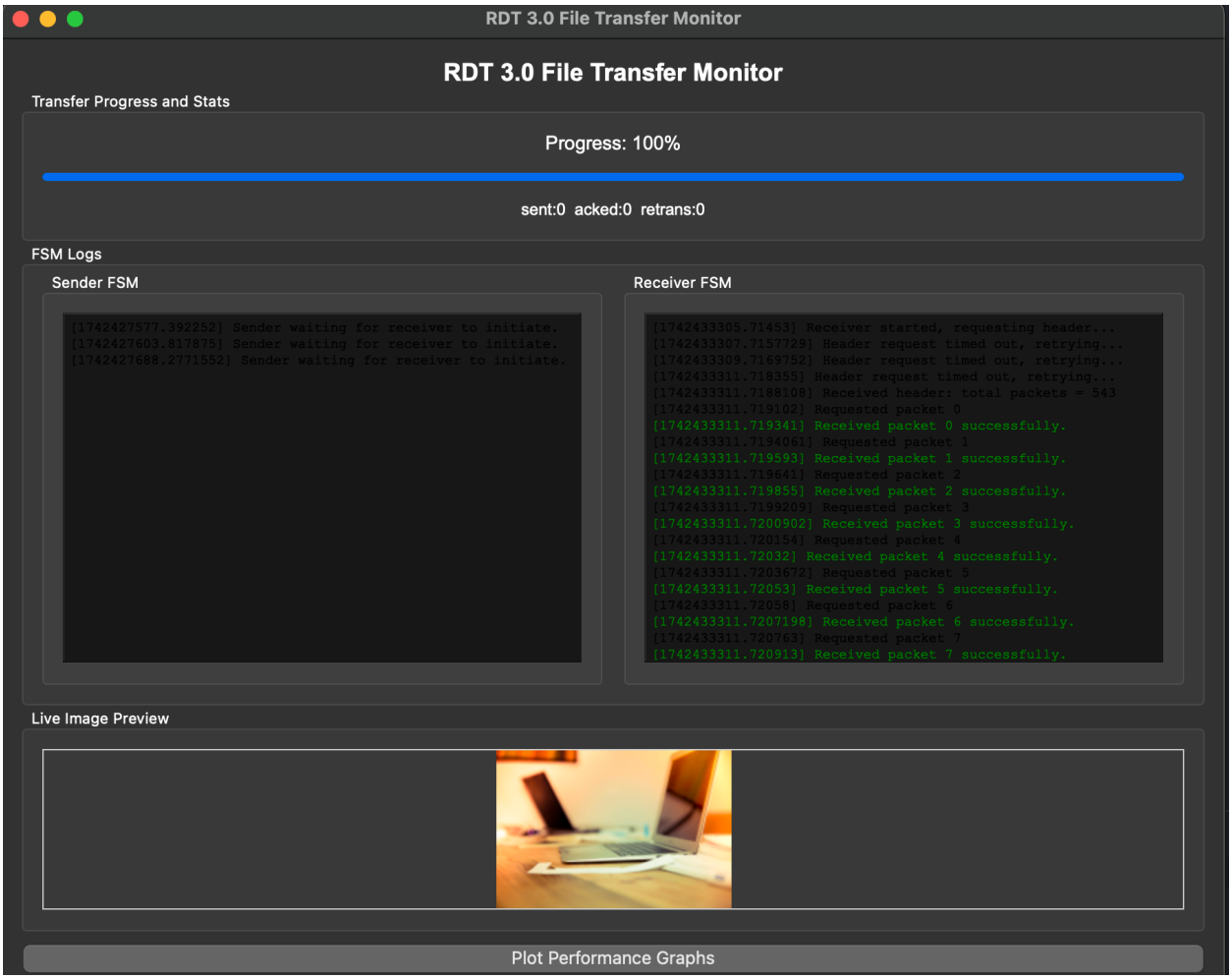
• **Responsibilities:**

• Offers live updates on packet transmissions, acknowledgments, and retransmissions.

• Visually represents progress and completion, making it simple for anyone—even those new to the protocol—to follow along.

d. Execution Screenshots


WAIT_FOR_HEADER
REQ HEADER
RECEIVE_HEADER
HEADER received
REQUEST PACKET N
REQ N
WAIT FOR PACKET N
Timeout
Packet N received
Retransmit request
seq < total_packets
VERIFY PACKET N
Timeout/Invalid Packet
Valid Packet N
Invalid Packet N
REQUEST PACKET N+1
RETRANSMIT
seq == total_packets
COMPLETION




```
yoseple@dhcp-10-250-227-202 src % make live
Starting server in live mode...
python3 server.py received_file.jpg sender_driven 10 1 > server.log 2>&1 &
Starting GUI...
python3 gui.py > gui.log 2>&1 &
Starting live client...
python3 client.py image.bmp 127.0.0.1 sender_driven 10 1
Total packets to send: 799
File transfer complete.
Completion time: 0.82 seconds
Data retransmissions: 10
ACK retransmissions: 4233
Throughput: 996615.85 bytes/s
```

Completion Time

• **Observation:** There's a slight initial overhead due to the nature of explicit packet requests.

• **Outcome:** Despite the initial delay, the performance is comparable or even better than sender-driven protocols in high-loss scenarios—thanks to fewer unnecessary retransmissions.

Retransmission Overhead

• **Observation:** The explicit request mechanism greatly minimizes needless retransmissions.

• **Outcome:** This results in significantly lower retransmission overhead compared to traditional sender-driven systems.

Throughput

• **Observation:** Throughput may start out a bit lower because of the extra overhead, but it stabilizes.

• **Outcome:** As loss rates increase, the protocol maintains more consistent and reliable throughput, making efficient use of available bandwidth.

ACK Efficiency

• **Observation:** By sending precise and deliberate requests, the protocol cuts down on redundant acknowledgments.

• **Outcome:** The receiver's efficient management of packet requests leads to higher acknowledgment efficiency, ensuring smoother data flow.

Discussion Questions

1. **Advantages and Disadvantages of Receiver-Driven vs. Sender-Driven Protocols**

    • **Advantages:**

    • Dramatically reduces retransmission overhead, saving bandwidth and reducing congestion.

    • Increases overall reliability and efficiency, especially in error-prone environments.

• **Disadvantages:**

• May incur slightly higher latency in ideal, low-loss conditions due to the extra handshaking.

• Introduces additional complexity with increased control overhead.

2. **Impact on Network Congestion**

• **Insight:** By putting the receiver in the driver's seat, the protocol efficiently manages data flow, cutting down on unnecessary transmissions and significantly easing network congestion.

3. **Real-world Applications**

• Ideal Scenarios: The receiver-driven protocol is particularly well-suited for environments such as satellite communications, mobile networks, and IoT sensor networks. In these settings, controlling bandwidth usage and reducing retransmission overhead are critical, making the protocol an attractive option for optimizing network resources even under challenging conditions.