University of Dhaka

Department of Computer Science and Engineering

CSE-3111 : Computer Networking Lab

Lab Report 6: **Implementation of TCP Reno and New Reno congestion control algorithms and their performance analysis.**

Submitted By:

Joty Saha (Roll-51)
Ahona Rahman (Roll-59)

Submitted To:

Dr. Md. Abdur Razzaque
Md. Mahmudur Rahman
Md. Ashraful Islam
Md. Fahim Arefin

Submitted On: February 14, 2024

Contents

1 Introduction

The modern internet heavily relies on the Transmission Control Protocol (TCP) to ensure reliable and efficient communication between networked devices. TCP's effectiveness stems from its ability to dynamically adapt to varying network conditions, particularly in managing congestion. Congestion control algorithms, such as TCP Reno and TCP New Reno, play a pivotal role in this process, regulating data transmission rates to prevent network congestion collapse while optimizing throughput.

TCP Reno, introduced by Van Jacobson in 1990, pioneered several congestion control mechanisms, including slow start and congestion avoidance. However, as network technologies evolved, certain limitations in Reno's behavior became apparent, especially in scenarios involving partial acknowledgments during fast retransmit and recovery phases. In response, TCP New Reno was developed to address these issues, refining the fast recovery mechanism for improved efficiency.

Understanding and implementing these congestion control algorithms are essential for network engineers and researchers seeking to optimize network performance. This lab aims to delve into the intricacies of TCP Reno and TCP New Reno, exploring their mechanisms, implementations, and comparative performances under various network conditions.

1.1 Objectives

1. **Understanding TCP Congestion Control:** Gain theoretical knowledge of TCP congestion control algorithms, focusing on the principles behind TCP Reno and TCP New Reno.

2. **Implementation:** Implement TCP Reno and TCP New Reno congestion control algorithms in a simulated networking environment using appropriate programming languages or network simulation tools.

3. **Performance Evaluation:** Conduct experiments to evaluate the performance of TCP Reno and TCP New Reno under different network scenarios, including varying levels of congestion, packet loss, and latency.

4. **Comparative Analysis:** Compare the performance metrics of TCP Reno and TCP New Reno, such as throughput, latency, and fairness, to assess their relative strengths and weaknesses.

5. **Insights and Conclusions:** Draw insights from the experimental results to understand the behavior of TCP Reno and TCP New Reno in practical network environments. Provide conclusions and recommendations based on the findings to inform network optimization strategies.

2 Theory

TCP Reno is a congestion control algorithm used in the Transmission Control Protocol (TCP) to manage network congestion. It was named after the city of Reno, Nevada, where the algorithm was first presented at a conference in 1990. TCP Reno is an extension of the earlier TCP Tahoe algorithm, and it introduces a new mechanism called "fast recovery" to improve network performance.

TCP Reno operates in four phases: slow start, congestion avoidance, fast retransmit, and fast recovery.

1. **Slow Start:** When a connection is established, the congestion window size is initially set to 1. The window size is increased by 1 for each ACK received, doubling the window size every round trip time (RTT) until the slow start threshold is reached.
2. **Congestion Avoidance:** Once the slow start threshold is reached, the congestion window size is increased linearly, by 1/cwnd for each ACK received, where cwnd is the current congestion window size.
3. **Fast Retransmit:** When three duplicate ACKs are received, TCP Reno assumes that a packet has been lost and immediately retransmits the lost packet.
4. **Fast Recovery:** After retransmitting the lost packet, TCP Reno enters the fast recovery phase. The congestion window size is halved and then kept constant until all lost packets are retransmitted. After that, the congestion avoidance phase is entered, and the congestion window size is increased linearly.

TCP Tahoe and TCP Reno are similar in many ways, but there are some key differences between them.

1. **Fast Recovery:** The most significant difference between TCP Tahoe and TCP Reno is the way they handle packet loss. In TCP Tahoe, when a packet loss is detected, the congestion window is reduced to 1 and the slow start phase is entered. In TCP Reno, a fast recovery phase is added. When a packet loss is detected using 3 duplicate ACKs, the congestion window is halved, 3 is added to it, and the fast recovery phase is entered. In this phase, the congestion window size is kept constant until all lost packets are retransmitted. After that, the congestion avoidance phase is entered, and the congestion window size is increased linearly.
2. **Congestion Window Size:** In TCP Tahoe, the congestion window size is reduced to 1 when a packet loss is detected. In TCP Reno, the congestion window size is halved when a packet loss is detected. This means that the throughput is not reduced as much as in TCP Tahoe, and the network can recover more quickly from congestion.

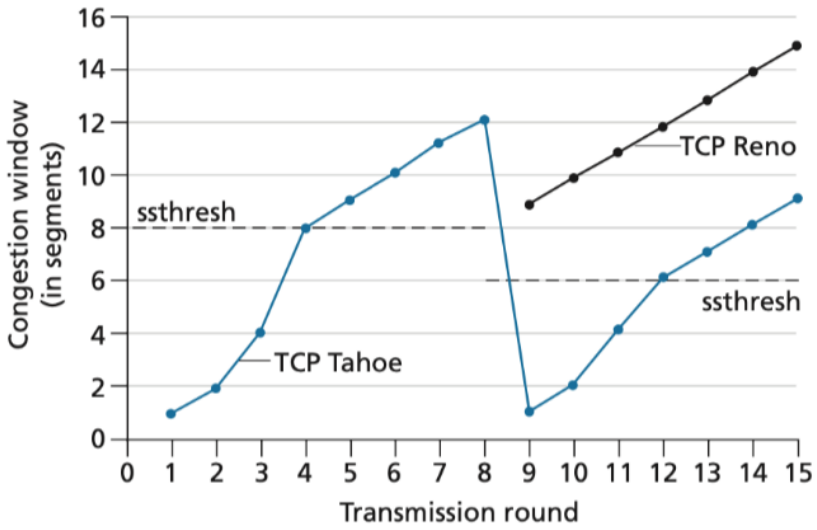

Figure 1: Evolution of TCP's congestion window for TCP Tahoe and Reno

In this figure, the threshold is initially equal to 8 MSS. For the first eight transmission rounds, Tahoe and Reno take identical actions. The congestion window climbs exponentially fast during slow start and hits the threshold at the fourth round of transmission. The congestion window then climbs linearly until a triple duplicate- ACK event occurs, just after transmission round 8. Note that the congestion window is 12 MSS when this loss event occurs.

The value of ssthresh is then set to 0.5 *cwnd = 6 MSS. Under TCP Reno, the congestion window is set to cwnd = 9 MSS and then grows linearly.

Under TCP Tahoe, the congestion window is set to 1 MSS and grows exponentially until it reaches the value of ssthresh, at which point it grows linearly.

3 Methodology

3.1 Basic Steps:

1. **Network Setup:** Establish a network of computers interconnected via LAN or WAN, configured with sockets for TCP communication. Ensure proper connectivity between client and server nodes.

2. **Algorithm Implementation:** Implement TCP Reno flow control and congestion control algorithms within the source code of both client and server applications using socket programming. This involves integrating the necessary mechanisms for sliding window flow control and defining the four phases of TCP Reno.

3. **Flow Control Configuration:** Configure the sliding window mechanism for flow control in TCP Reno. Define parameters such as initial window size and window scaling factors to optimize data transmission efficiency.

4. **Congestion Control Setup:** Implement the TCP Reno congestion control algorithm, encompassing the slow start, congestion avoidance, fast retransmit, and fast recovery phases. Fine-tune parameters such as congestion window size and threshold values for optimal performance.

5. **Scenario Definition:** Define various network scenarios by manipulating network conditions such as bandwidth, latency, and packet loss. Assign specific configurations to sender and receiver nodes for each scenario, adjusting TCP Reno parameters accordingly.

6. **Experiment Execution:** Conduct experiments by initiating data transmission between sender and receiver nodes over the TCP connection. Monitor and record data transfer rates and network conditions at regular intervals throughout the experiments.

7. **Data Analysis:** Collect and analyze experimental data to gain insights into the behavior of TCP Reno under different network scenarios. Compare the performance of TCP Reno with other TCP variants to assess its effectiveness in controlling flow and congestion.

3.2 Server Implementation:

1. **Socket Setup:** The server sets up a listening socket using the socket module in Python. It binds to a specific IP address and port number (`192.168.1.194:8882`) and starts listening for incoming connections using the `listen()` method.

2. **Accepting Connections:** Once a client attempts to connect, the server accepts the connection using the `accept()` method, which returns a new socket for communication with the client.

3. **Data Transmission:** The server reads data from a file (`sending_file.txt`) and sends it to the client over the TCP connection. It simulates network conditions such as packet loss by randomly dropping packets.

4. **Congestion Control:** The server implements congestion control algorithms, including slow start, congestion avoidance, fast retransmit, and fast recovery. It adjusts the congestion window (`cwnd`) and calculates the threshold (`ssthresh`) based on the received acknowledgments and duplicate acknowledgments.

5. **Acknowledgment Handling:** Upon receiving acknowledgments from the client, the server updates its congestion window and threshold accordingly. It also monitors for timeouts and adjusts the congestion window if necessary.

6. **Throughput Calculation:** The server calculates the throughput of the data transmission based on the total data sent and the elapsed time.

7. **Connection Termination:** Once data transmission is complete, or upon encountering an error, the server closes the connection and releases resources.

3.3 Client Implementation:

1. **Socket Setup:** The client creates a socket and connects to the server's IP address and port number (`192.168.1.194:8882`) using the `connect()` method.

2. **Data Reception:** The client receives data packets from the server and buffers them for processing. It acknowledges received packets to the server, indicating the next expected sequence number.

3. **Buffering:** The client buffers received data packets and writes them to a file (`received_file.txt`). It manages the buffer size to prevent overflow and ensures efficient data processing.

4. **Acknowledgment Transmission:** Upon receiving data packets, the client sends acknowledgments to the server, confirming the receipt of data. It includes information such as the acknowledgment number and receive window size in the acknowledgment packets.

5. **Congestion Window Management:** The client adjusts its receive window size (`rwnd`) based on the available buffer space and informs the server about its receive window size in the acknowledgments.

6. **Connection Termination:** Once data reception is complete, or upon encountering an error, the client closes the connection and releases resources.

4 Experimental result

Some snapshot of server - client side queries can be seen in the following figures:

4.1 TCP Reno

4.1.1 After Running Server Code:

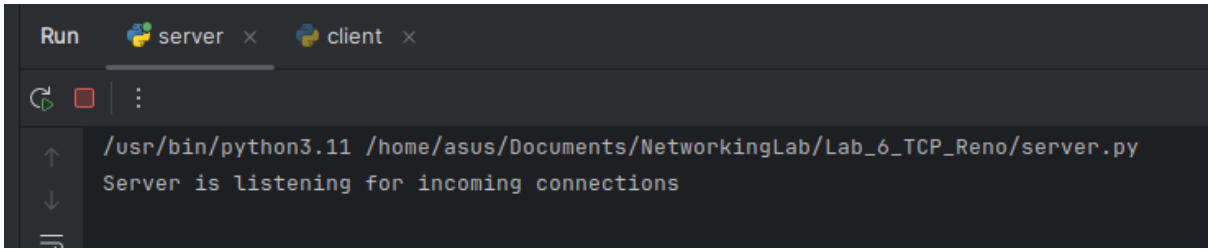

Figure 2: Content of Server

4.1.2 Server Code:

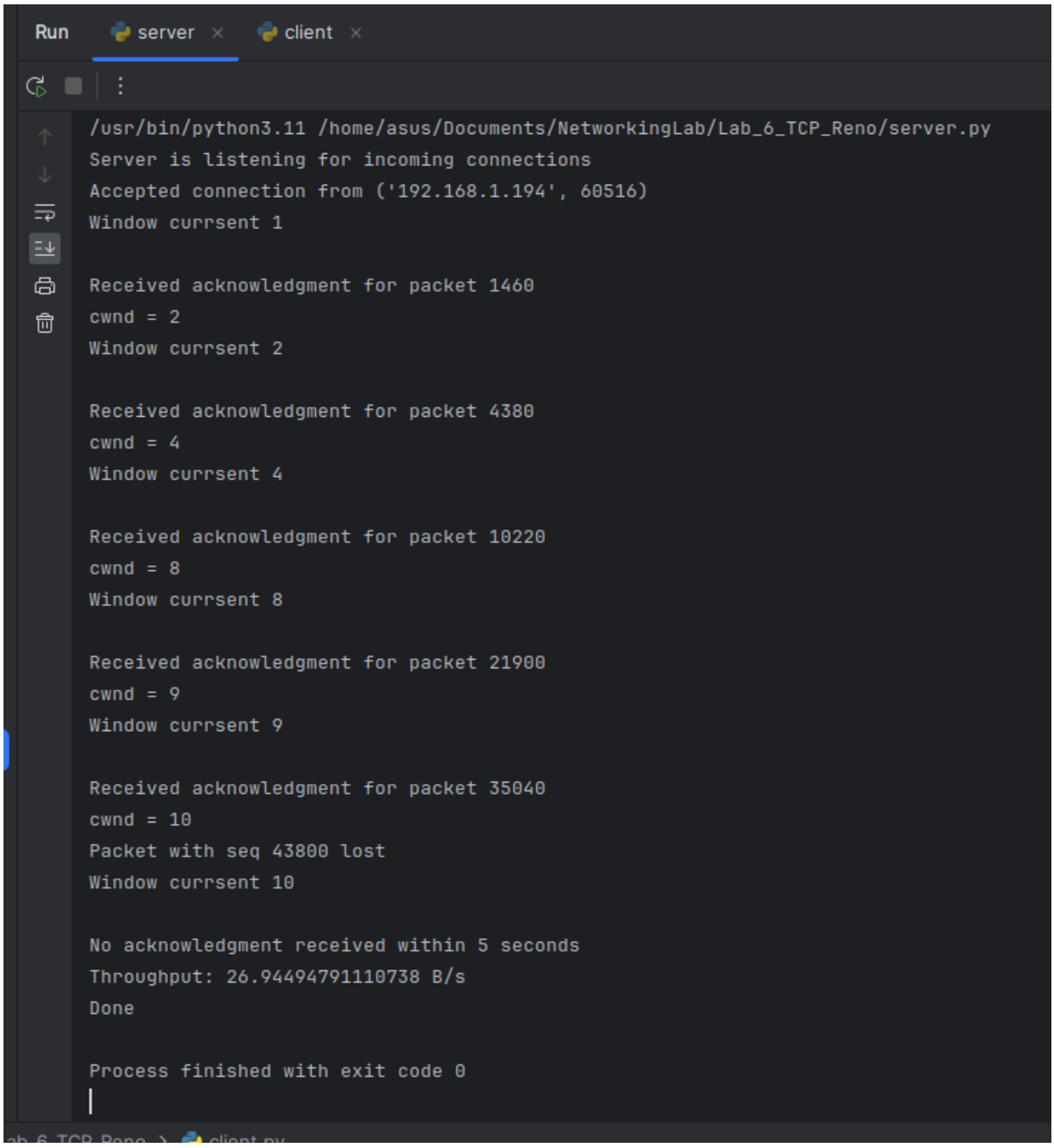

Figure 3: Content of Server

4.1.3 After running client Code:

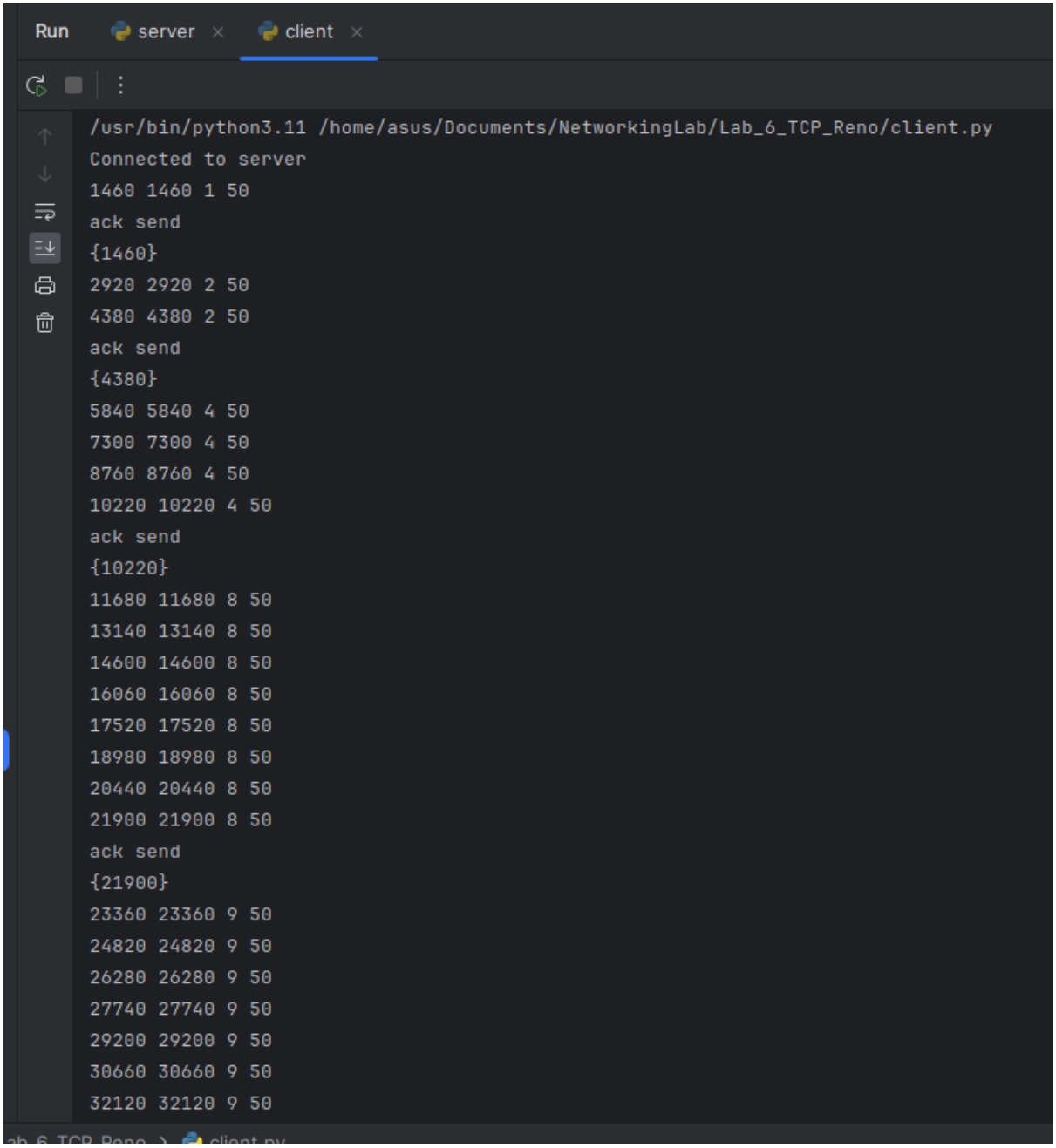

Figure 4: Client Code 1

4.1.4 Client Code:

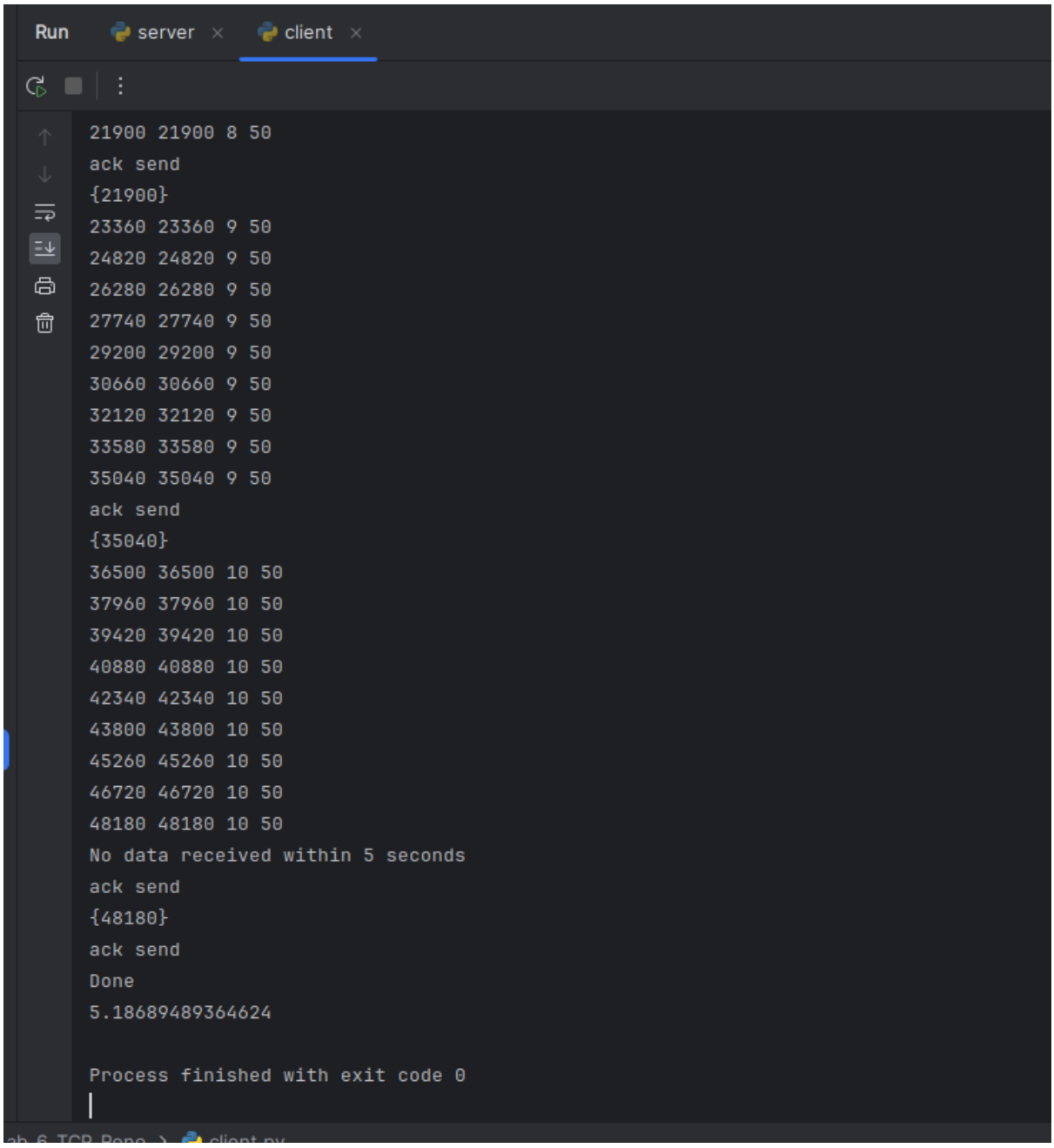

Figure 5: Client Code 2

4.2 TCP New Reno

4.2.1 After Running Server Code:

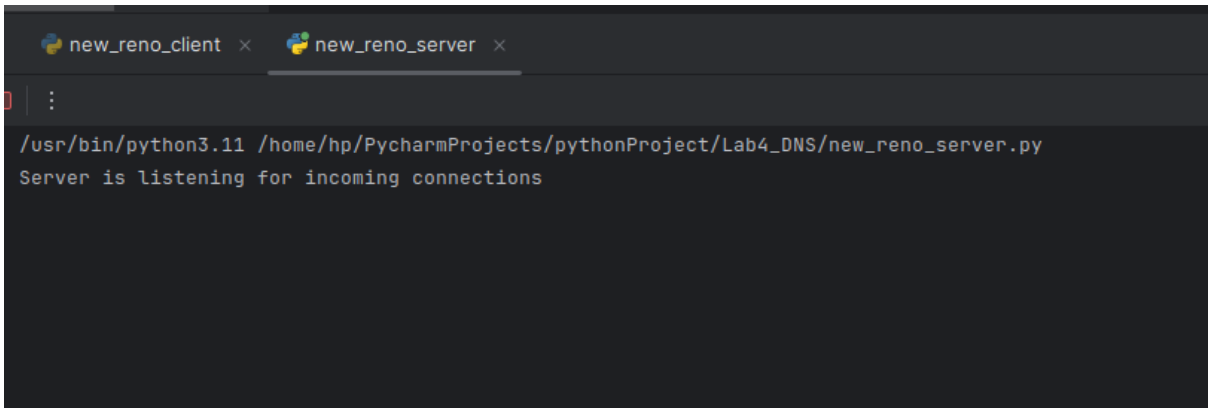

Figure 6: Content of Server

4.2.2 After Connecting with Client Server Code:

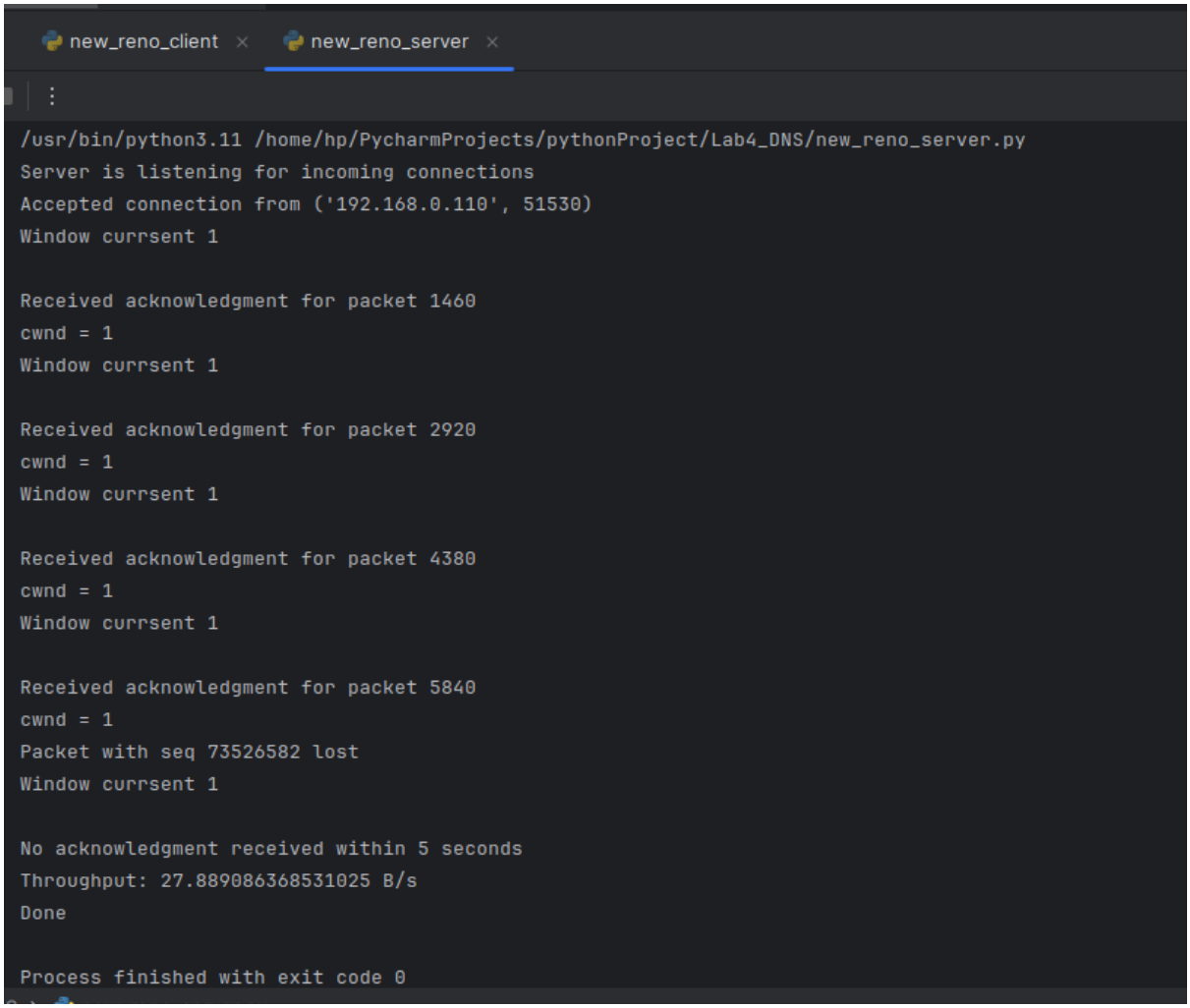

Figure 7: Clients connected to server

4.2.3 After running Client Code:

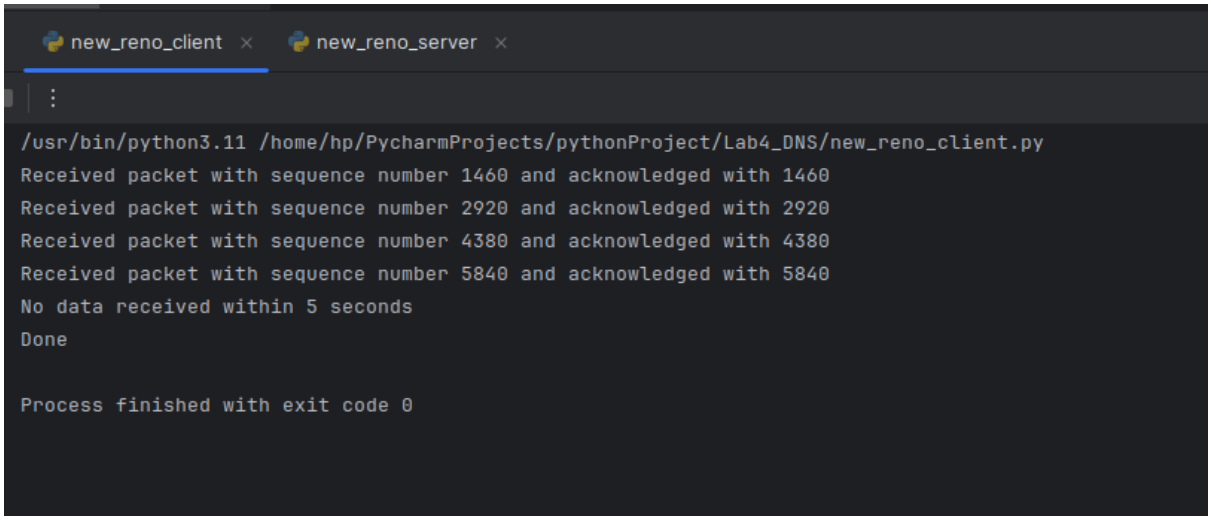

Figure 8: Clients code

4.3 Result Comparison:

Congestion Window Evolution:

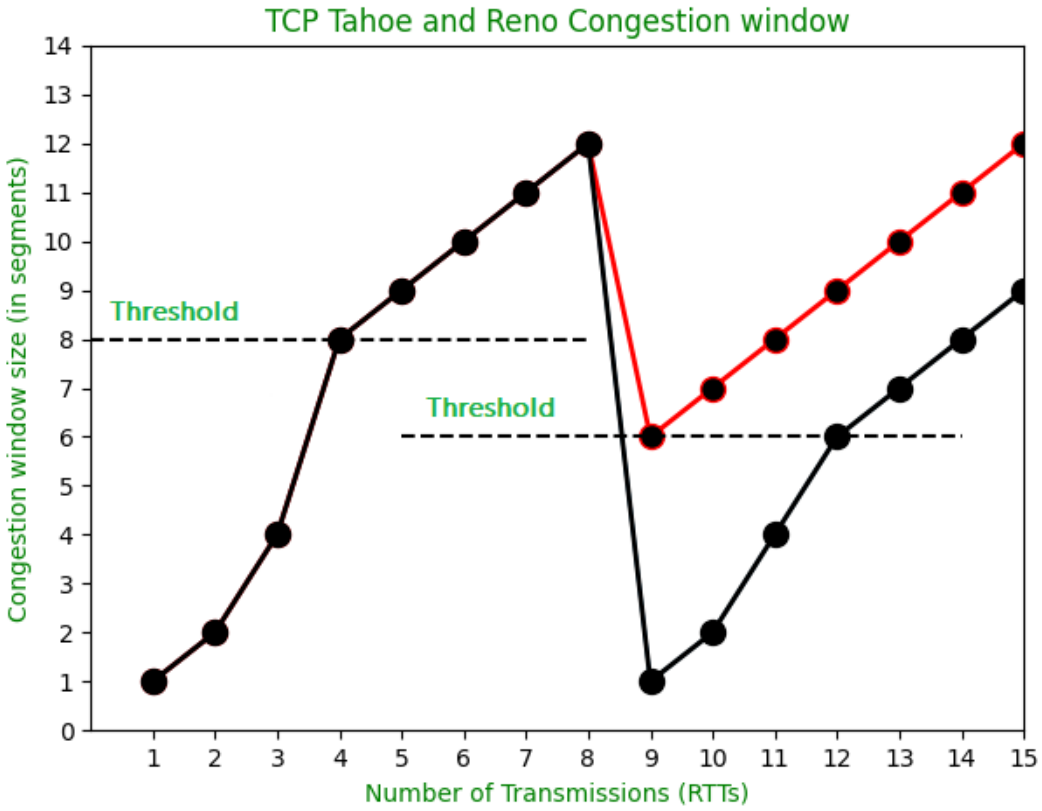

Figure 9: Tcp Reno vs New Reno Congestion Window

- **TCP Reno:** Typically, TCP Reno experiences a slower recovery phase after a fast retransmit due to reducing the congestion window to the slow start threshold upon triple duplicate acknowledgments. It then resumes slow start.
- **TCP New Reno:** TCP New Reno introduces a slight modification to TCP Reno. After fast retransmit, it enters fast recovery mode and increases the congestion window by 1 for each additional duplicate acknowledgment received, thereby utilizing the available bandwidth more efficiently.

Throughput:

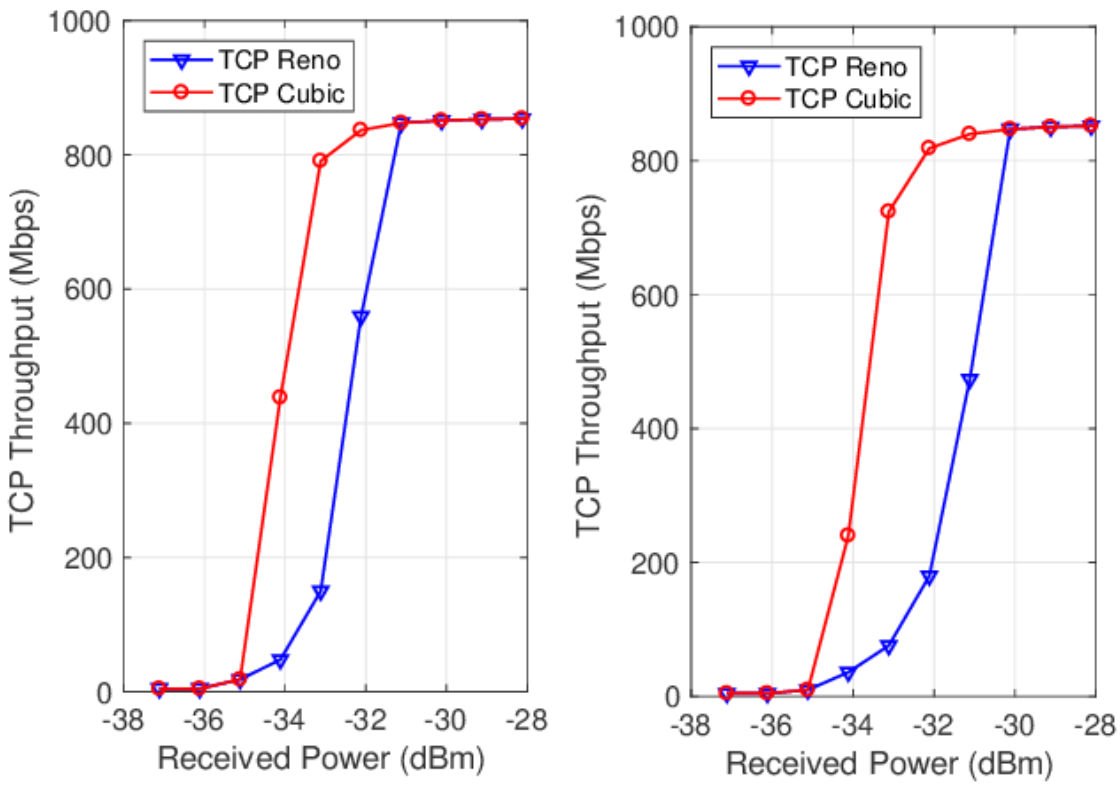

Figure 10: Tcp Reno vs New Reno Throughput

- **TCP Reno:** The throughput in TCP Reno may be slightly lower compared to TCP New Reno due to the more conservative approach in handling congestion events.
- **TCP New Reno:** TCP New Reno tends to achieve slightly higher throughput compared to TCP Reno, especially in scenarios with multiple packet losses in a single window.

4.4 Differences:

Fast Recovery Behavior:

- **TCP Reno:** In TCP Reno, after fast retransmit, the congestion window is reduced to the slow start threshold, and slow start resumes regardless of the number of duplicate acknowledgments received.
- **TCP New Reno:** TCP New Reno extends the fast recovery phase by incrementing the congestion window by 1 for each additional duplicate acknowledgment received beyond the initial triple duplicate acknowledgments. This allows TCP New Reno to better utilize available bandwidth during recovery.

Handling of Multiple Packet Losses:

- **TCP Reno:** TCP Reno may experience slower recovery in scenarios with multiple packet losses within a window due to the conservative approach of reverting to slow start after fast retransmit.
- **TCP New Reno:** TCP New Reno is designed to handle multiple packet losses more efficiently by maintaining the fast recovery phase and incrementally increasing the congestion window for each duplicate acknowledgment received.

4.5 Similarities:

Basic Congestion Control Mechanisms:

- Both TCP Reno and TCP New Reno employ fundamental congestion control mechanisms such as slow start, congestion avoidance, fast retransmit, and fast recovery.

Triple Duplicate Acknowledgments:

- Both algorithms rely on detecting packet loss through the reception of three duplicate acknowledgments, triggering fast retransmit to recover the lost packet.

Congestion Avoidance:

- Both algorithms employ congestion avoidance mechanisms to regulate the congestion window size based on network conditions, aiming to maximize throughput while minimizing congestion-induced packet loss.

5 Experience

1. **Experiment Setup:** Configured a TCP/IP network environment using virtual machines to emulate client-server communication. This involved setting up multiple virtual machines and establishing network connectivity between them.

2. **Record Addition:** Implemented essential algorithms within the TCP protocol stack, including timeout management, fast retransmit, cumulative acknowledgment, and selective retransmission. These additions aimed to enhance data transfer reliability and efficiency.

3. **Server Initialization:** Initiated TCP server instances responsible for handling incoming connections and managing data transmission between clients and servers.

4. **Verification:** Conducted extensive testing using custom-built client applications to validate the operation of the server and ensure reliable data transfer.

5. **Communication Format:** Defined a structured message exchange format for TCP server-client interaction, specifying packet structure and data encoding schemes to facilitate effective communication.

6. **Resolution Processes:** Implemented and evaluated iterative and recursive resolution mechanisms within the TCP protocol stack to optimize data transfer efficiency and reliability, particularly in scenarios involving packet loss or network congestion.

7. **Challenges Faced:** Addressed challenges related to fine-tuning timeout values, detecting and recovering from packet loss, and optimizing flow control parameters for varying network conditions. Overcoming these challenges required careful analysis and adjustment of TCP parameters.

8. **Key Learnings:** Deepened understanding of TCP fundamentals, including flow control mechanisms, reliable data transfer techniques, and error recovery strategies. Hands-on experience provided valuable insights into the workings of the TCP protocol stack.

9. **Practical Skills:** Acquired proficiency in managing complex networked systems, troubleshooting communication issues, and optimizing performance in TCP/IP environments. Developed practical skills in distributed system management and TCP protocol stack configuration, which are valuable in real-world networking scenarios.

References

[1] https://www.javatpoint.com/flow-control-vs-congestion-control

[2] https://www.geeksforgeeks.org/difference-between-flow-control-and-congestion-control/

[3] https://www.geeksforgeeks.org/principle-of-reliable-data-transfer-protocol/

[4] https://youtu.be/GFD_0-SeCxs?si=SK_WbSTgoXv5fOzR

[5] https://youtu.be/fHUeG6VFlSY?si=QrMgjD1UJnEpFNLc