University of Dhaka

Department of Computer Science and Engineering

CSE-3111 : Computer Networking Lab

Lab Report 6:

TCP Congestion Control with TCP Reno

Submitted By:

1. Syed Mumtahin Mahmud, Roll: 50

2. Nazira Jesmin Lina, Roll: 55

Submitted On :

February 28, 2023

Submitted To :

Dr. Md. Abdur Razzaque

Md Mahmudur Rahman

Md. Ashraful Islam

Md. Fahim Arefin

1 Introduction

Transmission Control Protocol (TCP) is a reliable, connection-oriented protocol that provides flow control, congestion control, and error recovery mechanisms for data transmission over the internet. TCP Reno is a popular implementation of TCP congestion control algorithm that uses a window-based approach to regulate the rate of data transmission in the presence of network congestion.

The purpose of this lab experiment is to study the performance of TCP Reno congestion control algorithm under different network conditions TCP Reno is a widely used congestion control algorithm that uses a window-based approach to regulate the rate of data transmission in the presence of network congestion. The algorithm is based on the principle of slow-start, congestion avoidance, and fast recovery.

In this experiment, we will implement TCP Reno and evaluate its performance under different network conditions. We will vary the network bandwidth and delay to simulate different network scenarios and measure the throughput, packet loss, and delay of TCP Reno. The results of this experiment will provide insights into the behavior of TCP Reno under different network conditions and help us understand the effectiveness of its congestion control mechanism.

1.1 Objectives

- To implement the TCP Reno congestion control algorithm and evaluate its performance under different network conditions by measuring the throughput, packet loss, and delay.
- The experiment aims to provide insights into the behavior of TCP Reno and the effectiveness of its congestion control mechanism in regulating the rate of data transmission in the presence of network congestion.

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 quot;fast recoveryquot; 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 3 is added to it, 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. This means that the congestion window size is increased exponentially until the slow start threshold is reached, and then increased linearly. In TCP Reno, a fast recovery phase is added. When a packet loss is detected, the congestion window is halved, 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. This can lead to a significant decrease in throughput, especially in high-bandwidth networks. 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. In summary, the main difference between TCP Tahoe and TCP Reno is the way they handle packet loss. Figure 3.52 illustrates the evolution of TCP's congestion window for both TCP Tahoe and Reno.

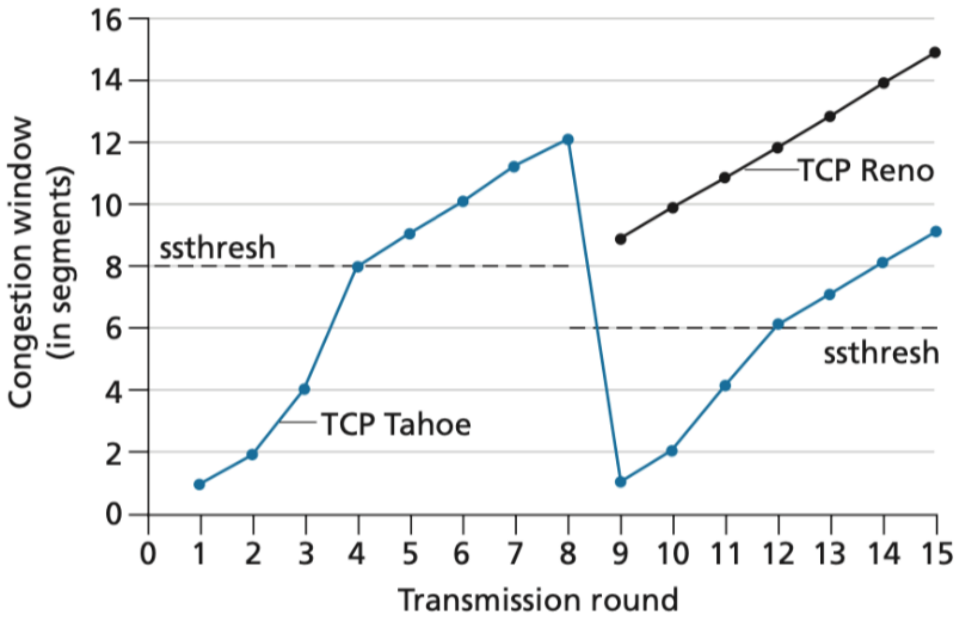


Figure 1: Evolution of TCP's congestion window (Tahoe and Reno)

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

1. Set up a network of computers with sockets for TCP communication. The computers should be connected via a LAN or WAN depending on the scope of the experiment.
2. Implement TCP Reno flow control and congestion control algorithms in the source code of the client and server applications using socket programming.
3. Configure the sliding window mechanism for flow control and implement the TCP Reno congestion control algorithm by defining the four phases: slow start, congestion avoidance, fast retransmit, and fast recovery.
4. Define network scenarios by varying network conditions such as bandwidth, latency, and packet loss. For each scenario, define a sender and receiver and set the TCP Reno parameters accordingly.
5. Run the experiments by transmitting data between the sender and receiver over the TCP connection. Record data transfer rates and network conditions at regular intervals.
6. Collect and analyze the 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 evaluate its effectiveness in controlling flow and congestion.

4 Experimental Result

Some Snapshots of the Client and Server Side queries can be seen in the following figures:

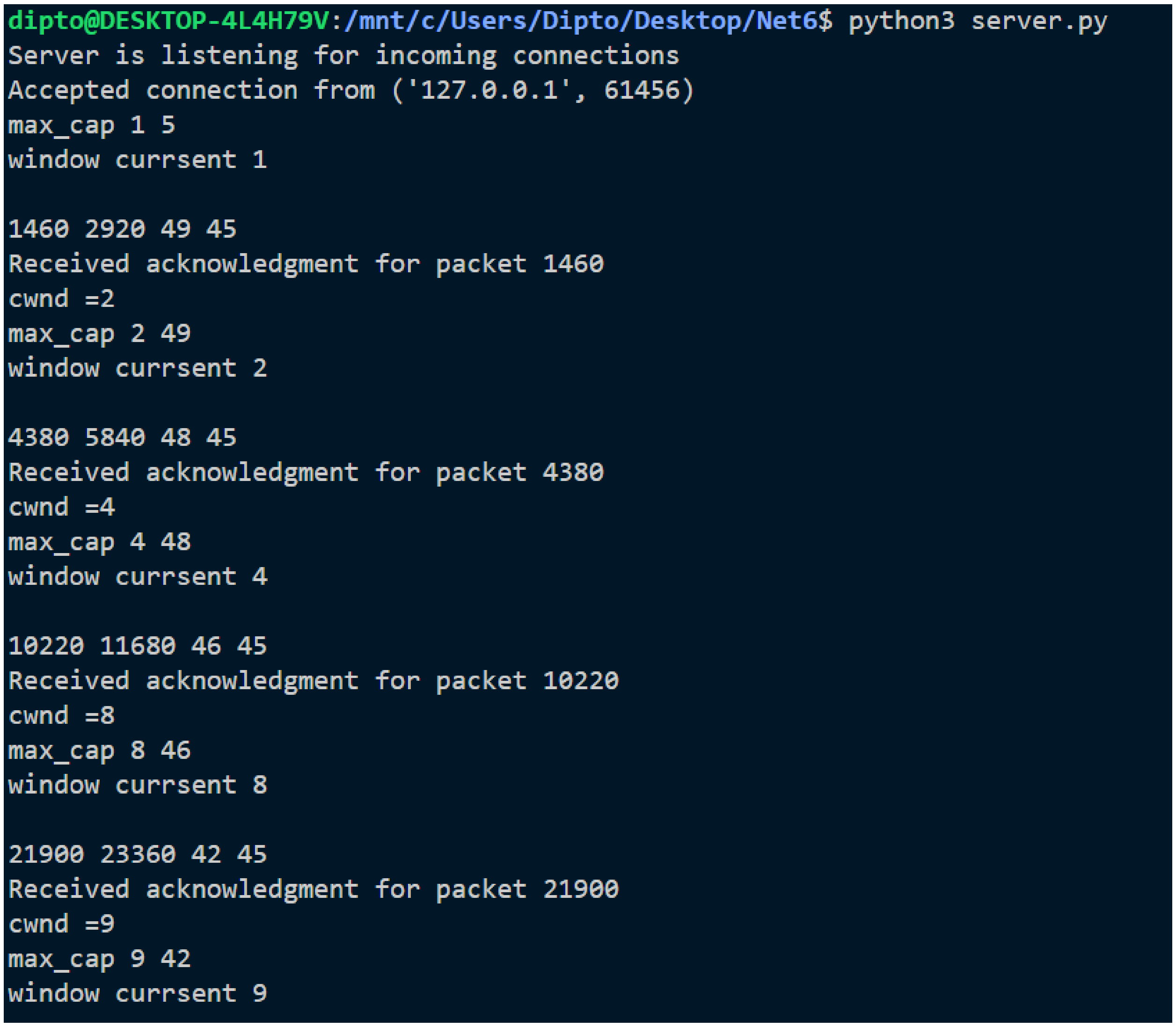
4.1 Server :

Once a connection is established, the server can receive window size. The receive window size specifies how much data the receiver is willing to accept before sending an acknowledgment. The size can be set to any value, but a common value is the maximum segment size (MSS) multiplied by a certain factor, such as 2 or 4. For example, if the MSS is 1460 bytes, the receive window size could be set to 2920 bytes or 5840 bytes.

After setting the receive window size, the server waits for the client to send data. Once data is received, the server sends an acknowledgment (ACK) back to the client to indicate that it has received the data. The ACK will contain the next expected sequence number, which is the sequence number of the next byte the server is expecting to receive.

TCP Reno uses a congestion avoidance algorithm that is based on detecting packet loss. When the server detects packet loss, it reduces its congestion window (cwnd) by half and enters a state known as "fast recovery". In this state, the server sends duplicate ACKs (DACKs) back to the client for each packet that is received out of order.

Once the server receives three duplicate ACKs, it assumes that the missing packet has been lost and retransmits it. The server then exits fast recovery and enters a state known as "congestion

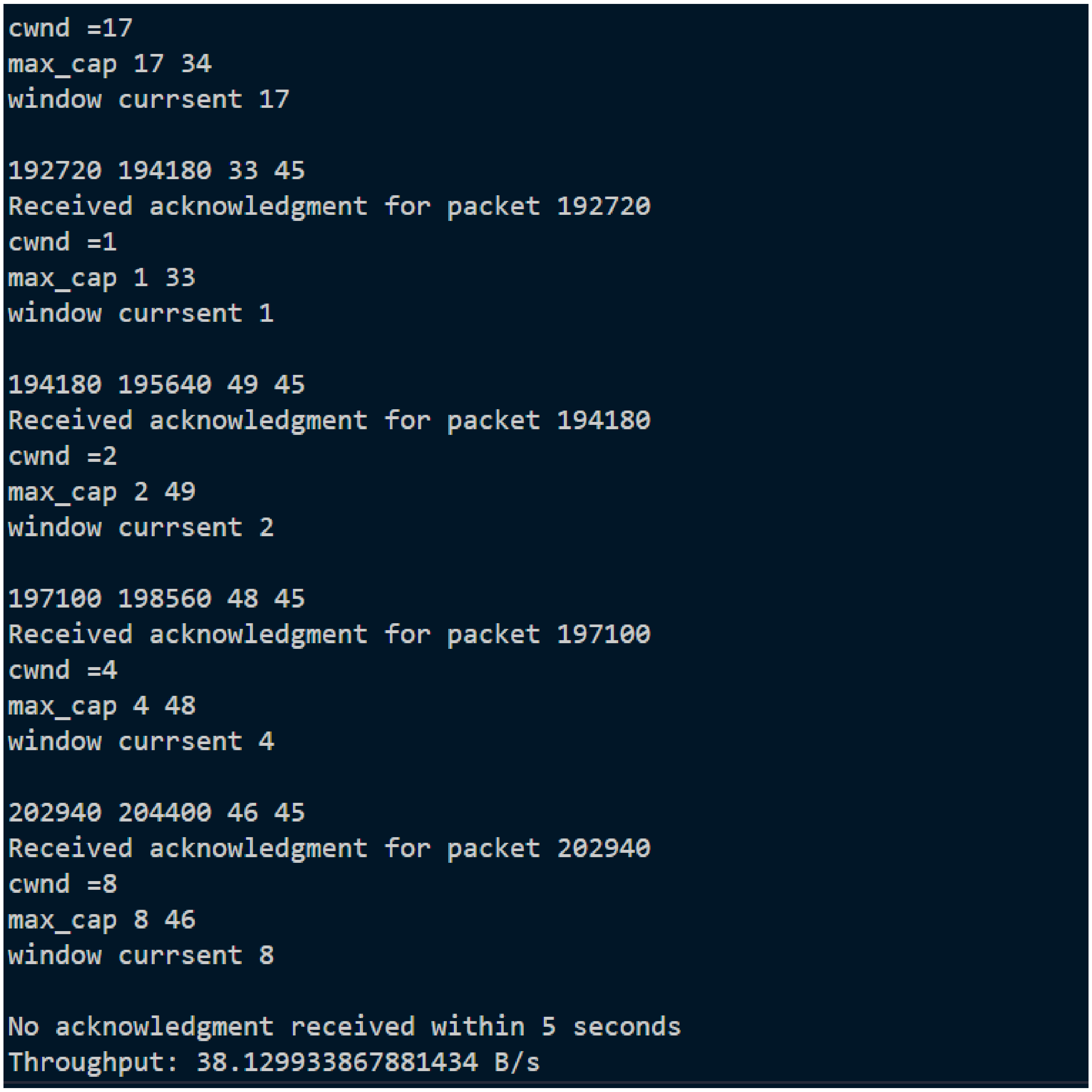


```
dipto@DESKTOP-4L4H79V:/mnt/c/Users/Dipto/Desktop/Net6$ python3 server.py
Server is listening for incoming connections
Accepted connection from ('127.0.0.1', 61456)
max_cap 1 5
window currsent 1

1460 2920 49 45
Received acknowledgment for packet 1460
cwnd =2
max_cap 2 49
window currsent 2

4380 5840 48 45
Received acknowledgment for packet 4380
cwnd =4
max_cap 4 48
window currsent 4

10220 11680 46 45
Received acknowledgment for packet 10220
cwnd =8
max_cap 8 46
window currsent 8

21900 23360 42 45
Received acknowledgment for packet 21900
cwnd =9
max_cap 9 42
window currsent 9
```

Figure 2: Content of server (1)

avoidance". In this state, the server increases its congestion window by 1/MSS for each successful round-trip time (RTT), until it detects another packet loss and repeats the process.

The server should use cumulative acknowledgment, which means that it will acknowledge all received packets up to the highest sequence number it has received in order. For example, if the server receives packets with sequence numbers 1, 2, and 3 in order, and then receives packet 5, it will still acknowledge packets 1-3 because it assumes that packet 4 was lost and will be re transmitted later.



```
cwnd =17
max_cap 17 34
window currsent 17

192720 194180 33 45
Received acknowledgment for packet 192720
cwnd =1
max_cap 1 33
window currsent 1

194180 195640 49 45
Received acknowledgment for packet 194180
cwnd =2
max_cap 2 49
window currsent 2

197100 198560 48 45
Received acknowledgment for packet 197100
cwnd =4
max_cap 4 48
window currsent 4

202940 204400 46 45
Received acknowledgment for packet 202940
cwnd =8
max_cap 8 46
window currsent 8

No acknowledgment received within 5 seconds
Throughput: 38.129933867881434 B/s
```

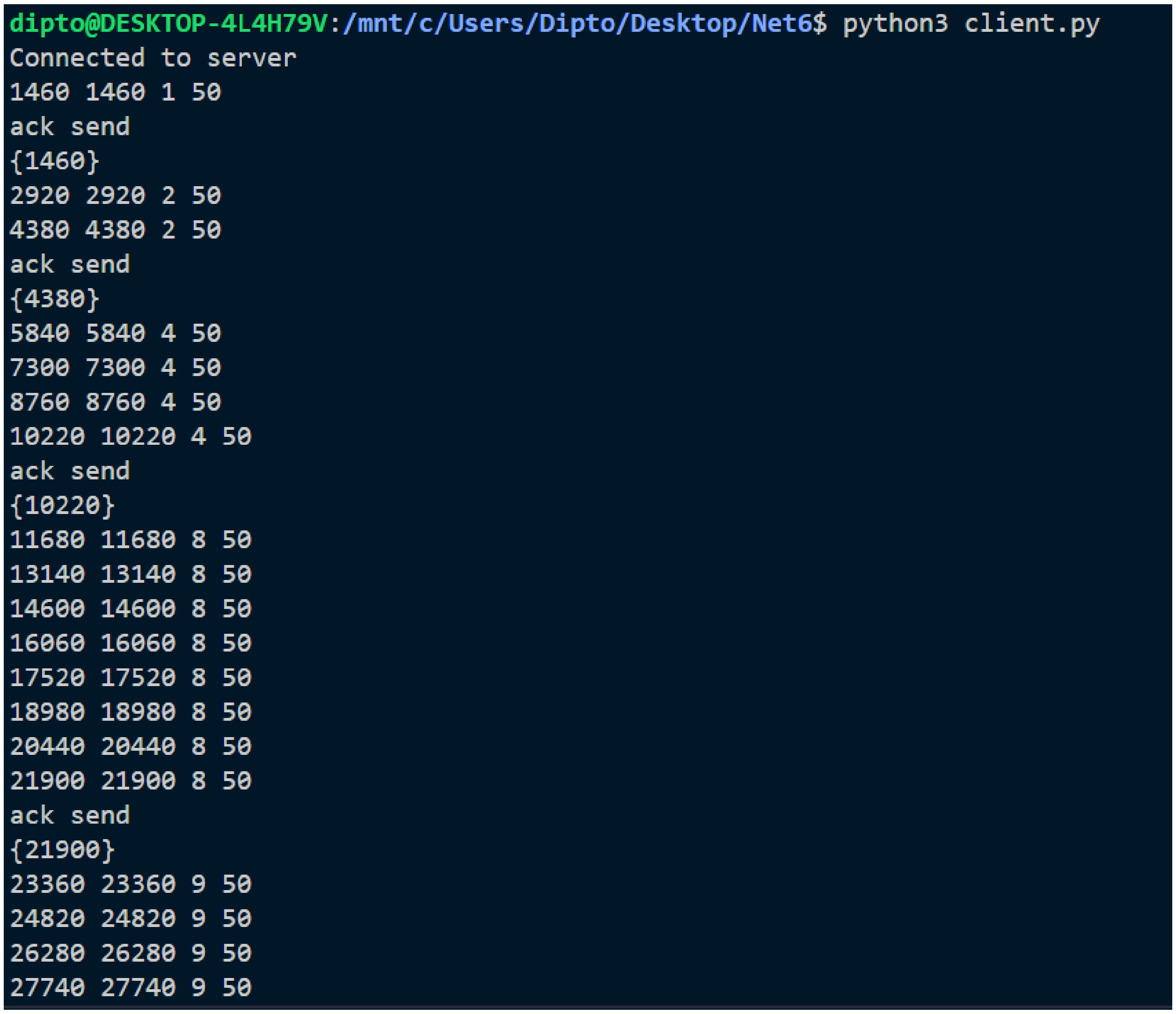
Figure 3: Content of server (2)

The server should continue to receive data from the client and send acknowledgments until the connection is terminated by either the server or the client.

4.2 Client :

After establishing a connection with the server, the client sends data to the server in segments. The size of each segment is determined by the congestion window size, which is initially set to the MSS and increases as the client receives acknowledgments from the server.

As data is sent to the server, the client sets a timer for each segment to monitor for packet loss. If an acknowledgment is not received within the timeout period, the client assumes that the packet has been lost and re-transmits the segment.



```
dipto@DESKTOP-4L4H79V:/mnt/c/Users/Dipto/Desktop/Net6$ python3 client.py
Connected to server
1460 1460 1 50
ack send
{1460}
2920 2920 2 50
4380 4380 2 50
ack send
{4380}
5840 5840 4 50
7300 7300 4 50
8760 8760 4 50
10220 10220 4 50
ack send
{10220}
11680 11680 8 50
13140 13140 8 50
14600 14600 8 50
16060 16060 8 50
17520 17520 8 50
18980 18980 8 50
20440 20440 8 50
21900 21900 8 50
ack send
{21900}
23360 23360 9 50
24820 24820 9 50
26280 26280 9 50
27740 27740 9 50
```

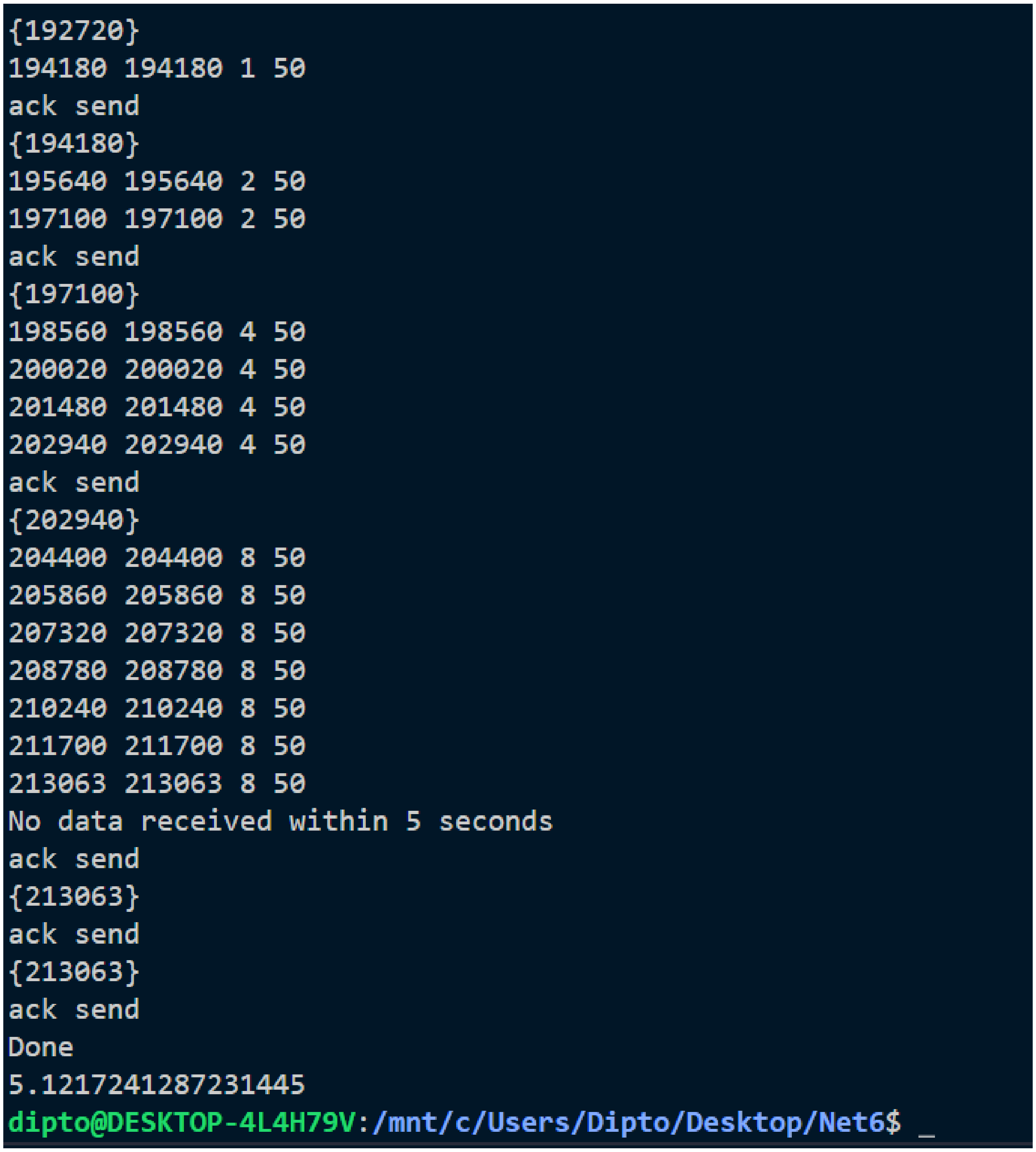
Figure 4: Content of Client (1)

When the client receives an acknowledgment from the server, it updates the congestion window size using the congestion avoidance algorithm. If no packets are lost, the window size is increased linearly for each acknowledgment received. If packets are lost, the window size is reduced using the fast recovery algorithm.

The client continues to send data to the server until all data has been transmitted. Once all data has been sent, the client sends a FIN packet to initiate the connection termination process.

During the connection, the client also receives acknowledgments from the server indicating that data has been received successfully. If the client does not receive an acknowledgment within a certain period of time, it assumes that the packet has been lost and retransmits the segment.

Overall, the client implementation for TCP Reno congestion control algorithm involves sending data to the server, monitoring for packet loss, and updating the congestion window size based on acknowledgments received from the server.



```
{192720}
194180 194180 1 50
ack send
{194180}
195640 195640 2 50
197100 197100 2 50
ack send
{197100}
198560 198560 4 50
200020 200020 4 50
201480 201480 4 50
202940 202940 4 50
ack send
{202940}
204400 204400 8 50
205860 205860 8 50
207320 207320 8 50
208780 208780 8 50
210240 210240 8 50
211700 211700 8 50
213063 213063 8 50
No data received within 5 seconds
ack send
{213063}
ack send
{213063}
ack send
Done
5.1217241287231445
dipto@DESKTOP-4L4H79V:/mnt/c/Users/Dipto/Desktop/Net6$ _
```

Figure 5: Content of Client (2)

4.3 Result comparision:

he CWND vs Time graph shows how the congestion window changes over time during a data transfer between the sender and receiver in TCP Reno. Initially, the sender starts with a small congestion window size, which is typically equal to the maximum segment size (MSS). As the sender begins to transmit data, it waits for acknowledgments from the receiver. If it receives the acknowledgments within a certain time interval, it increases the congestion window size. This allows the sender to send more data without causing congestion in the network.

However, if the sender detects packet loss or congestion in the network, it reduces the congestion window size. This is done to prevent further congestion and to allow the network to recover. The sender uses a timeout mechanism to detect packet loss, and when it does, it reduces the congestion window size to the initial size and starts the transmission again.

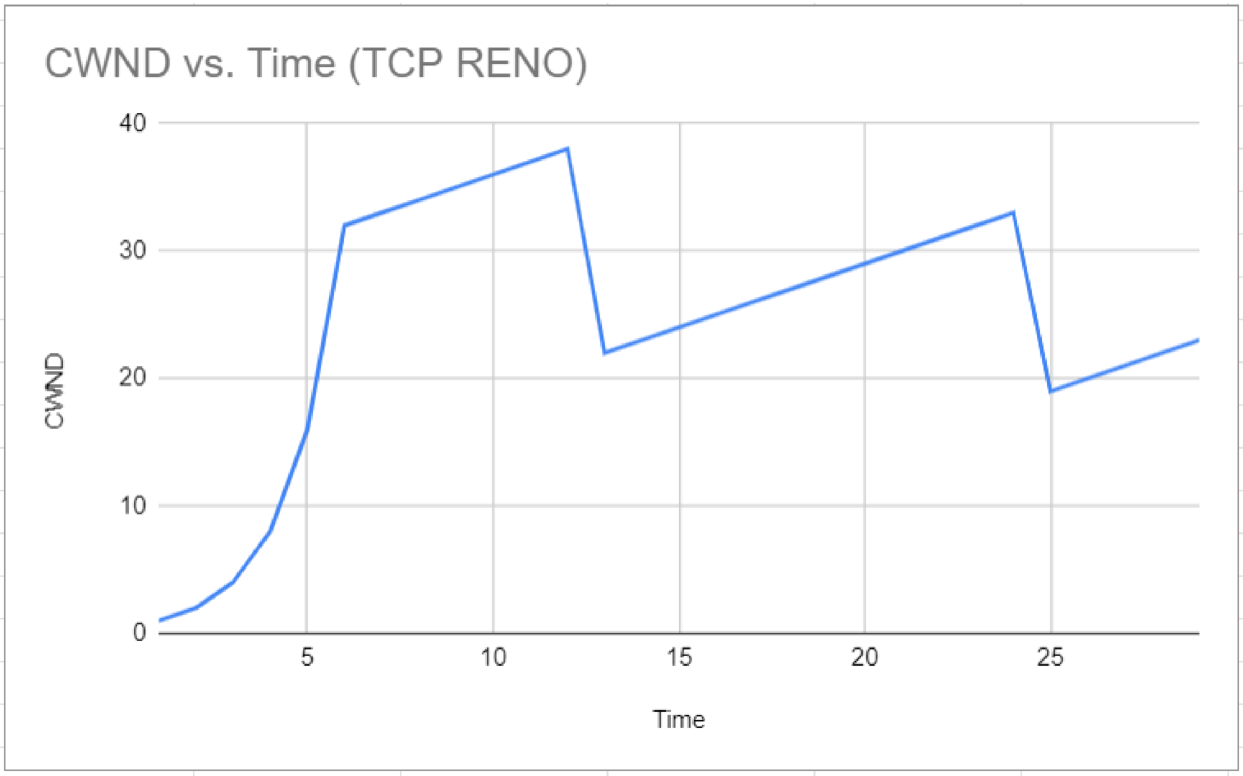


Figure 6: CWND vs Time(TCP Reno)

The CWND vs Time graph shows the dynamic changes in the congestion window size over time. It typically has a sawtooth shape, with the congestion window increasing and then decreasing periodically. The upswing in the sawtooth represents the time when the sender is increasing the congestion window size, and the downswing represents the time when the congestion window is being reduced.

By analyzing the CWND vs Time graph, we can gain insights into the behavior of TCP Reno congestion control algorithm under different network conditions. It helps in understanding the effectiveness of the algorithm in controlling congestion and improving the throughput of data transfer over a network.

TCP Tahoe and TCP Reno are two popular congestion control algorithms used in TCP. While both algorithms aim to achieve high network performance, there are some differences in their approaches.

TCP Tahoe uses a conservative approach to congestion control. It starts with a small window size, then gradually increases the window size until a packet is lost. When a packet loss occurs, Tahoe cuts the window size in half and enters a slow start phase, gradually increasing the window size again until it reaches the previous maximum value before the loss occurred. This approach helps Tahoe to avoid congestion, but it can result in lower network performance in situations where

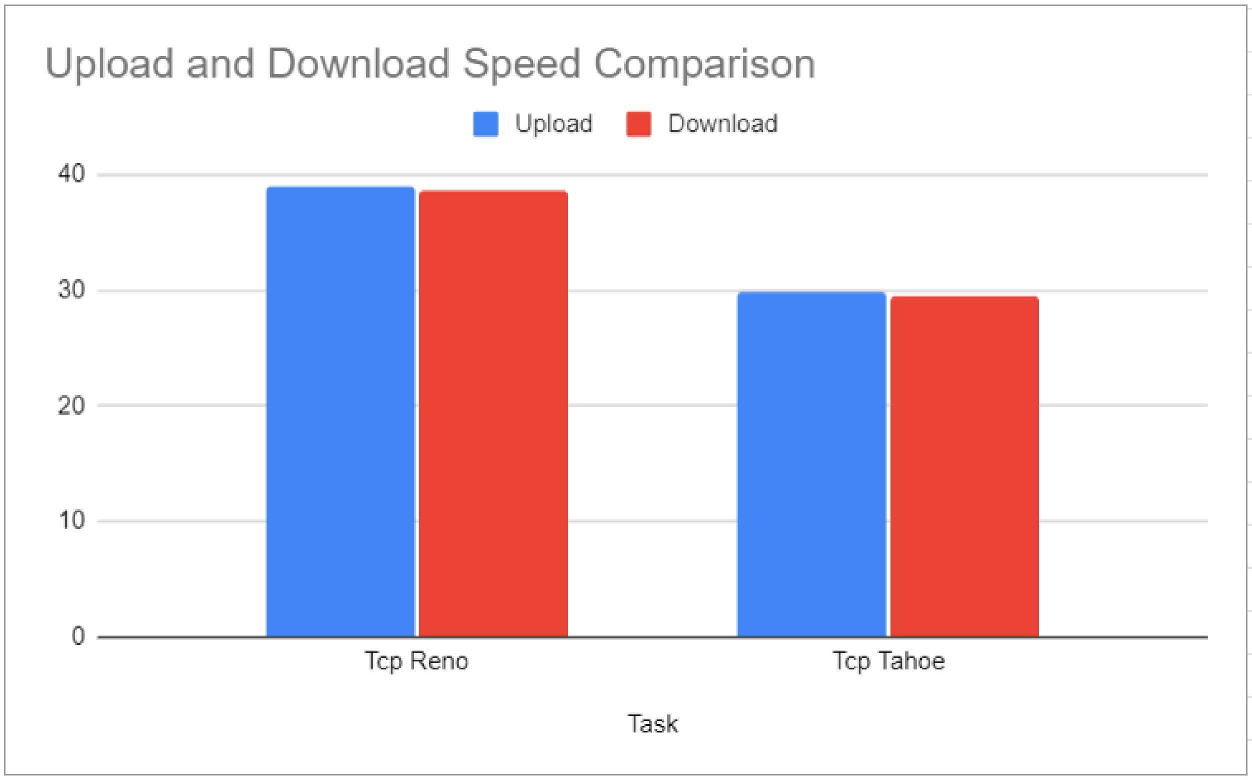


Figure 7: Throughput Comparison between Tcp Tahoe and Tcp Reno.

the available bandwidth is not fully utilized.

TCP Reno, on the other hand, uses a more aggressive approach to congestion control. Reno also starts with a small window size and gradually increases it until a packet is lost. However, when a packet loss occurs, Reno not only cuts the window size in half but also enters a fast recovery phase. During fast recovery, Reno sends additional packets to fill the network pipe, expecting that the lost packet will be retransmitted by the receiver. Once the lost packet is retransmitted and acknowledged, Reno exits fast recovery and resumes normal congestion control.

The difference in the congestion control approaches of Tahoe and Reno can be observed in their CWND (Congestion Window) vs Time graphs. The CWND vs Time graph for Tahoe shows a gradual increase in window size followed by a sharp drop in window size after a packet loss. The graph for Reno, on the other hand, shows a similar increase in window size followed by a smaller drop in window size during fast recovery, and then a gradual increase again.

In general, Reno is known to provide better network performance than Tahoe in high-bandwidth, high-delay networks. However, in low-bandwidth, low-delay networks, Tahoe may perform better due to its conservative approach to congestion control.

References

[1] Difference between Flow Control and Congestion Control - Javatpoint. https://www.javatpoint.com/flow-control-vs-congestion-control. [Online; accessed 2023-02-25].

[2] Difference between flow control and congestion control. *GeeksforGeeks*, may 24 2019. [Online; accessed 2023-02-25].

[3] Tcp Tahoe and TCP Reno. *GeeksforGeeks*, feb 7 2022. [Online; accessed 2023-02-25].

[4] James Kurose and Keith Ross. *Computer networking: A top-down approach, global edition.* Pearson Higher Ed, oct 23 2018. [Online; accessed 2023-02-16].