# University of Dhaka

Department of Computer Science and Engineering

CSE-3111 : Computer Networking Lab

Lab Report 6 : Implementation of TCP Reno congestion control algorithm.

**Submitted By:**

Afser Adil Olin

Roll No : AE-47

Anika Tabassum

Roll No : Rk-61

**Submitted On :**

February 28, 2023

**Submitted To :**

Dr. Md. Abdur Razzaque

Md Mahmudur Rahman

Md. Ashraful Islam

Md. Fahim Arefin

## Contents

**1 Introduction** **2**
1.1 Objectives . . . . . 2

**2 Theory** **2**
2.1 TCP Reno . . . . . 2
2.2 Comparison between TCP Tahoe and TCP Reno . . . . . 3
2.2.1 Similarities: . . . . . 3
2.2.2 Differences: . . . . . 3

**3 Methodology** **4**

**4 Implementation** **5**
4.1 Preliminary work . . . . . 5
4.2 TCP Reno Congestion Control . . . . . 5
4.3 EWMA: Estimated Weighted Mean Average . . . . . 5
4.4 Check Sum . . . . . 5
4.5 Error handling . . . . . 6

**5 Experimental Result** **6**
5.1 Implement TCP Reno Congestion Control . . . . . 6

**6 Analyzation** **7**
6.1 Congestion Window . . . . . 7
6.2 Estimated_RTT & Sample_RTT & Deviation_RTT . . . . . 8
6.3 Throughput . . . . . 8
6.4 Packet Loss . . . . . 9
6.5 Comparison Between Tcp Reno and Tcp Tahoe Congestion Window . . . . . 9

**7 Experience** **10**

## 1 Introduction

The objective of implementing the TCP Reno congestion control algorithm is to simulate and evaluate its performance in a network environment. The implementation aims to provide a better understanding of how the algorithm works and its effectiveness in controlling network congestion and compare it with TCP Tahoe.

### 1.1 Objectives

- To understand the principles and mechanics of TCP Reno congestion control algorithm and how it reacts to congestion in a network.
- To develop and implement the TCP Reno algorithm in a simulated network environment.
- To evaluate the performance of the algorithm in terms of its ability to effectively control network congestion, throughput, and delay.
- To compare the performance of TCP Reno with other congestion control algorithms, such as TCP Tahoe, TCP Vegas, and TCP NewReno, and identify the strengths and weaknesses of each algorithm.

## 2 Theory

### 2.1 TCP Reno

TCP Reno is a congestion control algorithm used in TCP (Transmission Control Protocol) networks to regulate the flow of data and prevent network congestion. The TCP Reno algorithm is based on the principle of detecting network congestion by monitoring packet loss and adjusting the transmission rate accordingly. The algorithm works as follows:

1. **Slow start :**
   At the beginning of a transmission, the TCP sender starts by sending a small number of packets and gradually increases the number of packets sent until it detects congestion. This phase is known as slow start.
2. **Congestion avoidance:**
   Once congestion is detected, the TCP sender reduces its transmission rate to avoid further congestion. In TCP Reno, the sender enters a congestion avoidance phase where it linearly increases its transmission rate until another congestion event occurs.
3. **Fast retransmit:**
   If the TCP sender detects that a packet has been lost, it immediately retransmits the packet without waiting for a timeout. This is known as fast retransmit and it helps to reduce the time taken to recover from packet loss.
4. **Fast recovery:**
   After fast retransmit, the sender enters a fast recovery phase where it reduces its transmission rate and waits for an acknowledgment of the retransmitted packet. Once the acknowledgment is received, the sender increases its transmission rate by one packet per round trip time (RTT).

5. **Timeout:**
   If a packet is not acknowledged within a certain period of time (known as the timeout interval), the sender assumes that the packet has been lost and retransmits the packet. The sender then enters the slow start phase again.

TCP Reno uses a combination of slow start, congestion avoidance, fast retransmit, and fast recovery to control congestion in the network and provide reliable data transmission.

## 2.2 Comparison between TCP Tahoe and TCP Reno

TCP Tahoe and TCP Reno are two popular congestion control algorithms used in TCP (Transmission Control Protocol) networks. Both algorithms aim to regulate the flow of data and prevent network congestion, but there are some similarities and differences between them.

### 2.2.1 Similarities:

1. **Slow start:**
   Both TCP Tahoe and TCP Reno use a slow start phase at the beginning of a transmission to gradually increase the number of packets sent until they detect congestion.
2. **Congestion avoidance:**
   Both algorithms enter a congestion avoidance phase where they regulate the transmission rate to avoid further congestion.
3. **Fast retransmit:**
   Both algorithms use a fast retransmit mechanism to retransmit lost packets without waiting for a timeout.

### 2.2.2 Differences:

1. **Reaction to congestion:**
   TCP Tahoe responds to congestion by reducing the transmission rate to roughly half of its current value and then slowly increasing it again. In contrast, TCP Reno reduces its transmission rate to roughly half of its current value but then linearly increases it until another congestion event occurs.
2. **Fast recovery:**
   TCP Reno has a fast recovery phase where it reduces its transmission rate and waits for an acknowledgment of the retransmitted packet before increasing its transmission rate by one packet per round trip time (RTT). TCP Tahoe, on the other hand, does not have a fast recovery phase and instead enters the slow start phase again.
3. **Timeout:**
   TCP Tahoe sets a long timeout interval, which means that it takes longer to detect and recover from lost packets. TCP Reno uses a shorter timeout interval, which helps to reduce the time taken to recover from packet loss.

In summary, both TCP Tahoe and TCP Reno use similar congestion control mechanisms, but TCP Reno has some improvements over TCP Tahoe, such as a shorter timeout interval and a fast recovery phase, which helps to improve network performance and reduce recovery time after congestion events.

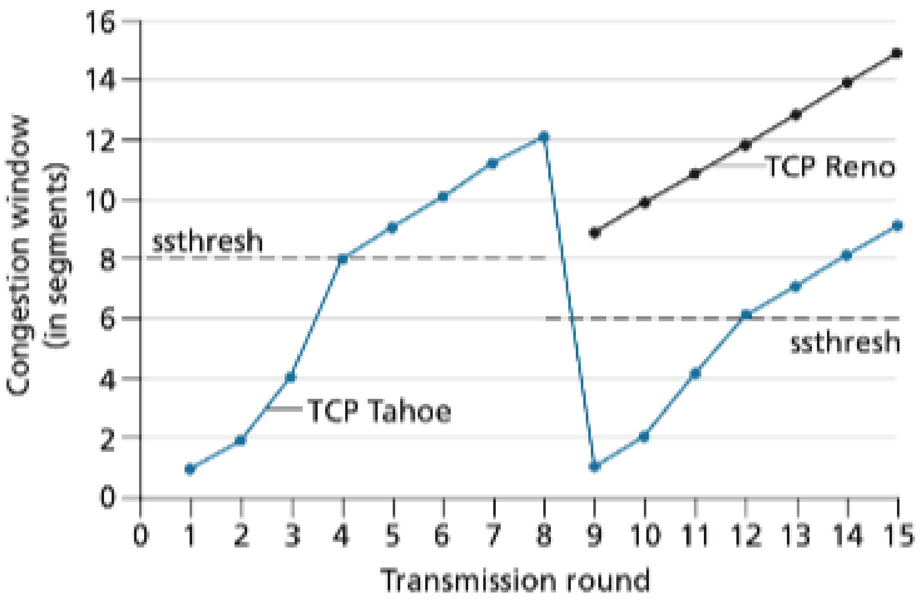


Figure 1: Evalution of TCP Reno and TCP Tahoe congestion window

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

- We need to study and understand the TCP Reno algorithm.
- We need to select a suitable network simulation tool.
- We need to develop the implementation of TCP Reno which includes include the TCP Reno congestion control algorithm.
- After the implementation is complete, we need to test the algorithm to ensure that it functions correctly.
- Once the simulations have been run, the results must be analyzed to evaluate the performance of the implementation.
- We need to refine the implementation, based on the results of the analysis,

- We need the implementation of TCP Reno should be documented. This includes documenting the modifications made to the TCP implementation, the simulation environment used, and the results of the testing and analysis.

## 4 Implementation

We have followed the lab instructions to implement a TCP Reno connection with congestion control and checksums according to spec.

### 4.1 Preliminary work

We have declared a custom header of size 20 bytes which contain

1. Sequence number of 6 bytes
2. Acknowledgement number of 6 bytes
3. Receive window size information of 6 bytes
4. Checksum of data 2 bytes

### 4.2 TCP Reno Congestion Control

TCP Reno is a congestion control algorithm used in computer networks to prevent network congestion and ensure fair sharing of network bandwidth.

The algorithm we implemented works by using a ”slow start” mechanism to ramp up the sending rate of the sender until it reaches a point where packet loss occurs. When a packet is lost, TCP Reno reduces the sending rate by cutting the congestion window in half, which decreases the number of packets being sent.

Additionally, it uses a ”congestion avoidance” mechanism that increases the sending rate by one packet for every successful acknowledgement received from the receiver. This allows the sender to gradually increase its sending rate, but not so quickly as to cause congestion and packet loss.

When a packet loss occurs, TCP Reno enters a ”Fast Retransmit phase” phase where it enters the congestion avoidance phase and gradually increases its sending rate again.

### 4.3 EWMA: Estimated Weighted Mean Average

This part is briefly discussed above. We have used the EWMA to increase throughput by reducing the sender timeout interval.

### 4.4 Check Sum

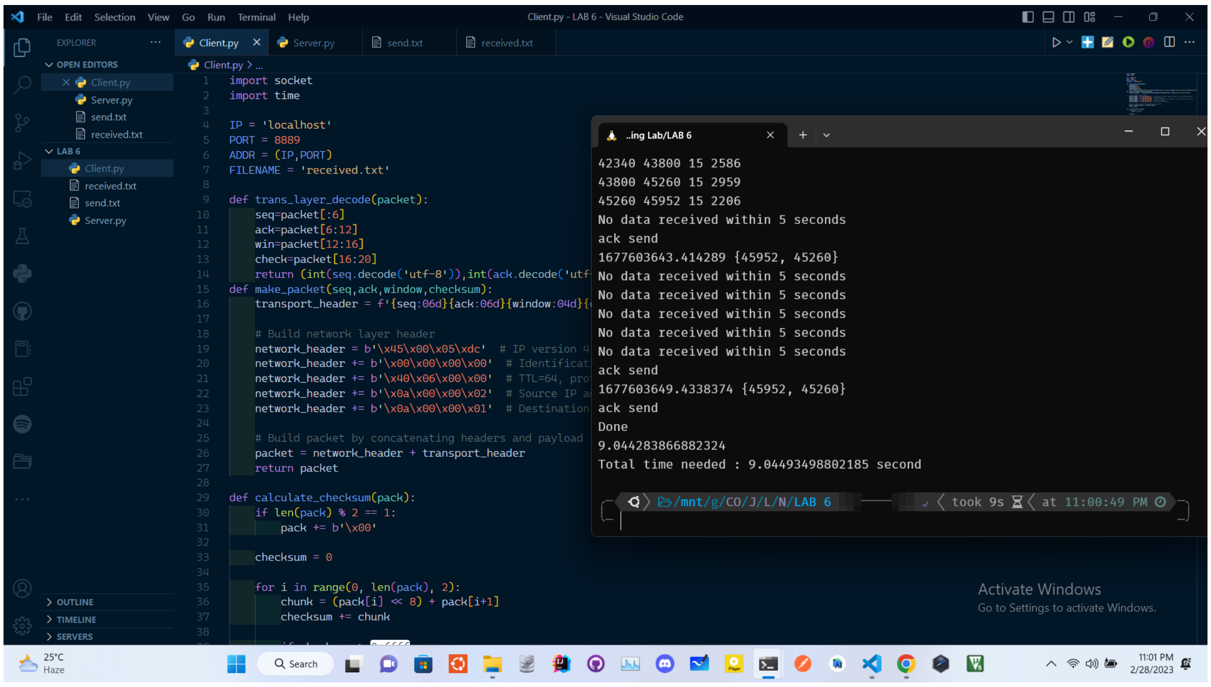
We have implemented a checksum function for 16 bit number.The sender first calculate for the checksum for the ”*payload*” which contains a packet of data via that function and send it by the header.In the client side, The checksum is checked if the data is mismatched.

### 4.5 Error handling

We checked if any of the error occurred (via force), then we send an ack to the sender. Then the sender send the packet again. If the ack itself is dropped, the receiver will face a timeout and resend the last received seg as ack to prompt the sender to resend the file. The transfer terminates when the sender receives an acknowledgement equal or greater than the data length.

## 5 Experimental Result

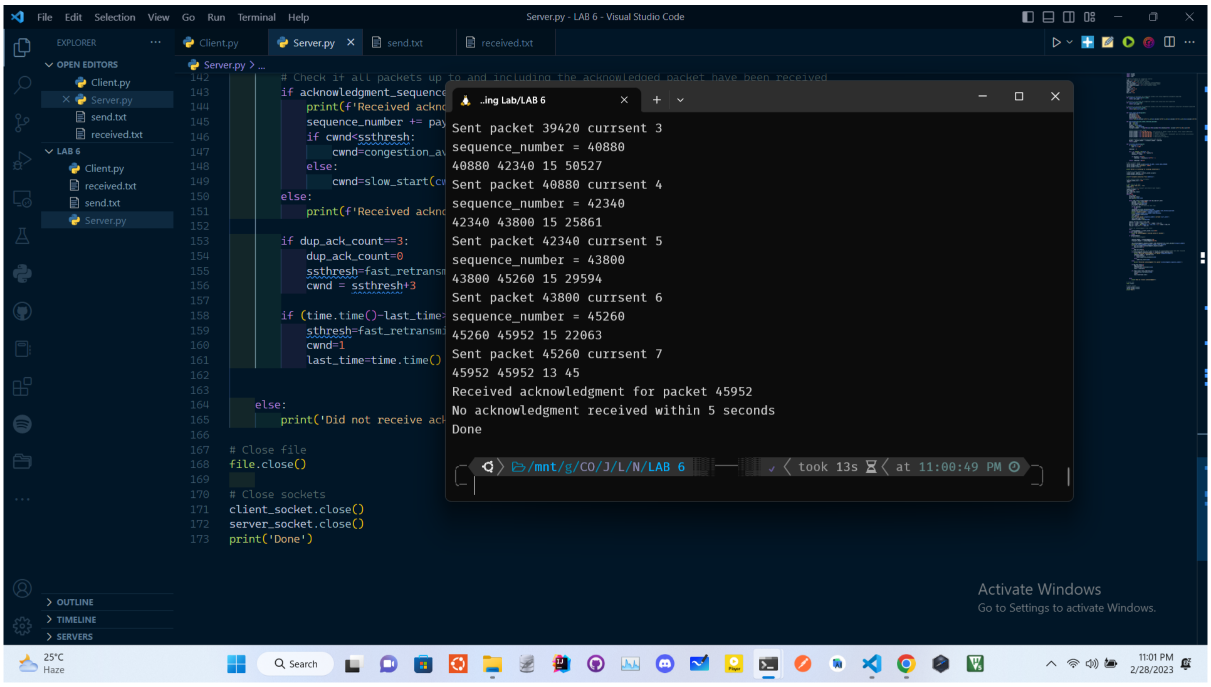
### 5.1 Implement TCP Reno Congestion Control



```python
import socket
import time

IP = 'localhost'
PORT = 8889
ADDR = (IP,PORT)
FILENAME = 'received.txt'

def trans_layer_decode(packet):
    seq=packet[:6]
    ack=packet[6:12]
    win=packet[12:16]
    check=packet[16:20]
    return (int(seq.decode('utf-8')),int(ack.decode('utf
def make_packet(seq,ack,window,checksum):
    transport_header = f'{seq:06d}{ack:06d}{window:04d}{

    # Build network layer header
    network_header = b'\x45\x00\x05\xdc'  # IP version 4
    network_header += b'\x00\x00\x00\x00'  # Identificat
    network_header += b'\x40\x06\x00\x00'  # TTL=64, pro
    network_header += b'\x0a\x00\x00\x02'  # Source IP a
    network_header += b'\x0a\x00\x00\x01'  # Destination

    # Build packet by concatenating headers and payload
    packet = network_header + transport_header
    return packet

def calculate_checksum(pack):
    if len(pack) % 2 == 1:
        pack += b'\x00'

    checksum = 0

    for i in range(0, len(pack), 2):
        chunk = (pack[i] << 8) + pack[i+1]
        checksum += chunk
```

```
42340 43800 15 2586
43800 45260 15 2959
45260 45952 15 2206
No data received within 5 seconds
ack send
1677603643.414289 {45952, 45260}
No data received within 5 seconds
No data received within 5 seconds
No data received within 5 seconds
No data received within 5 seconds
No data received within 5 seconds
ack send
1677603649.4338374 {45952, 45260}
ack send
Done
9.044283866882324
Total time needed : 9.04493498802185 second

/mnt/g/CO/J/L/N/LAB 6    took 9s  at 11:00:49 PM
```

Figure 2: Client Side View



```
        # Check if all packets up to and including the acknowledged packet have been received
        if acknowledgment_sequence
            print(f'Received ackno
            sequence_number += pay
            if cwnd<ssthresh:
                cwnd=congestion_av
            else:
                cwnd=slow_start(cw
        else:
            print(f'Received ackno

        if dup_ack_count==3:
            dup_ack_count=0
            ssthresh=fast_retransm
            cwnd = ssthresh+3

        if (time.time()-last_time
            sthresh=fast_retransmi
            cwnd=1
            last_time=time.time()


    else:
        print('Did not receive ack

# Close file
file.close()

# Close sockets
client_socket.close()
server_socket.close()
print('Done')
```

```
Sent packet 39420 currsent 3
sequence_number = 40880
40880 42340 15 50527
Sent packet 40880 currsent 4
sequence_number = 42340
42340 43800 15 25861
Sent packet 42340 currsent 5
sequence_number = 43800
43800 45260 15 29594
Sent packet 43800 currsent 6
sequence_number = 45260
45260 45952 15 22063
Sent packet 45260 currsent 7
45952 45952 13 45
Received acknowledgment for packet 45952
No acknowledgment received within 5 seconds
Done

/mnt/g/CO/J/L/N/LAB 6 ── ✓ took 13s at 11:00:49 PM
```

Figure 3: Server Side View

# 6 Analyzation

## 6.1 Congestion Window

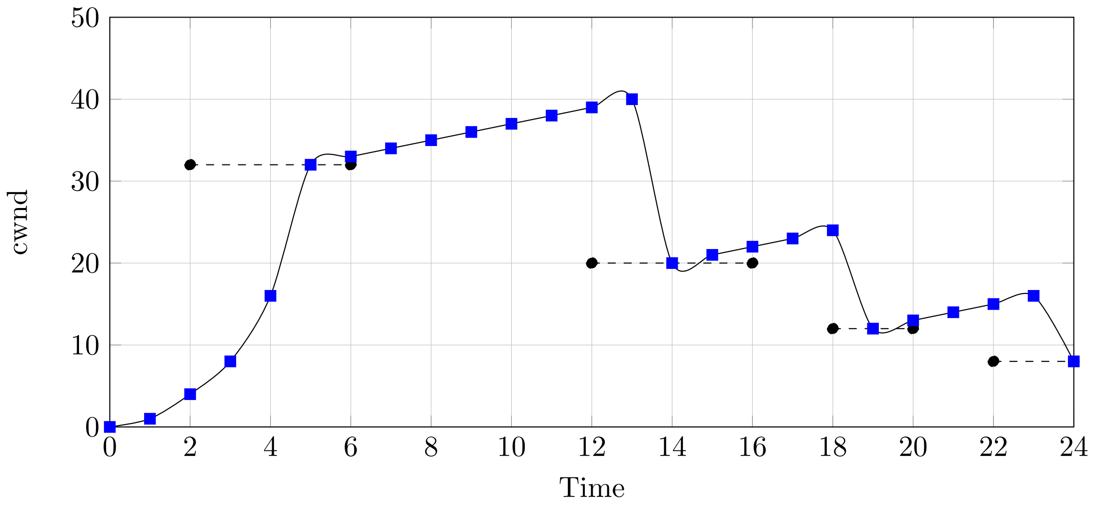


Figure 4: Congestion Window Timing Graph with the ssthresholds

## 6.2 Estimated_RTT & Sample_RTT & Deviation_RTT

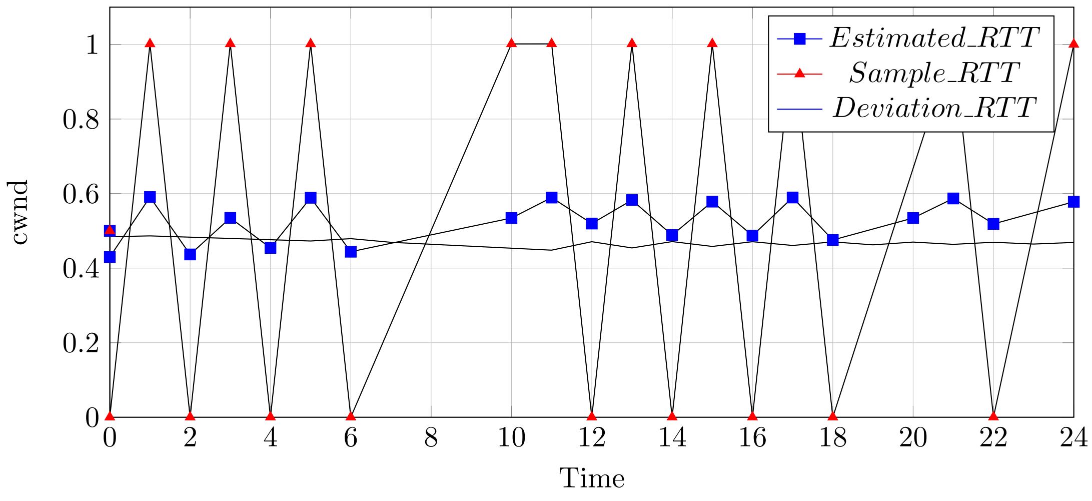
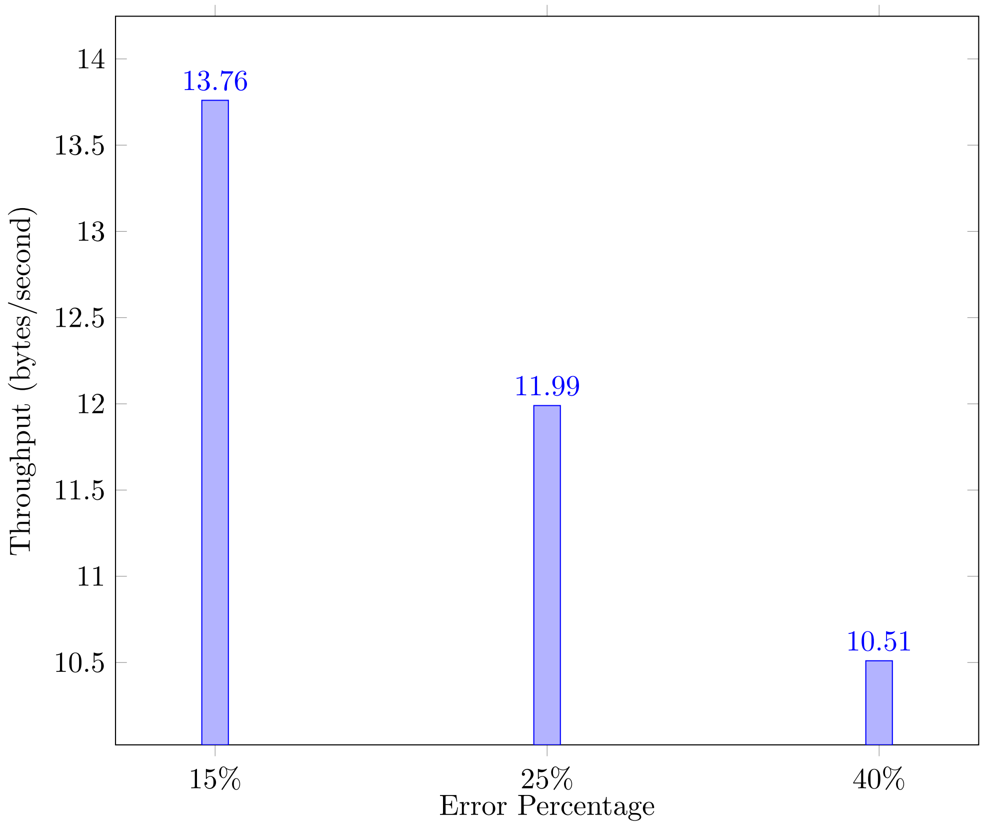
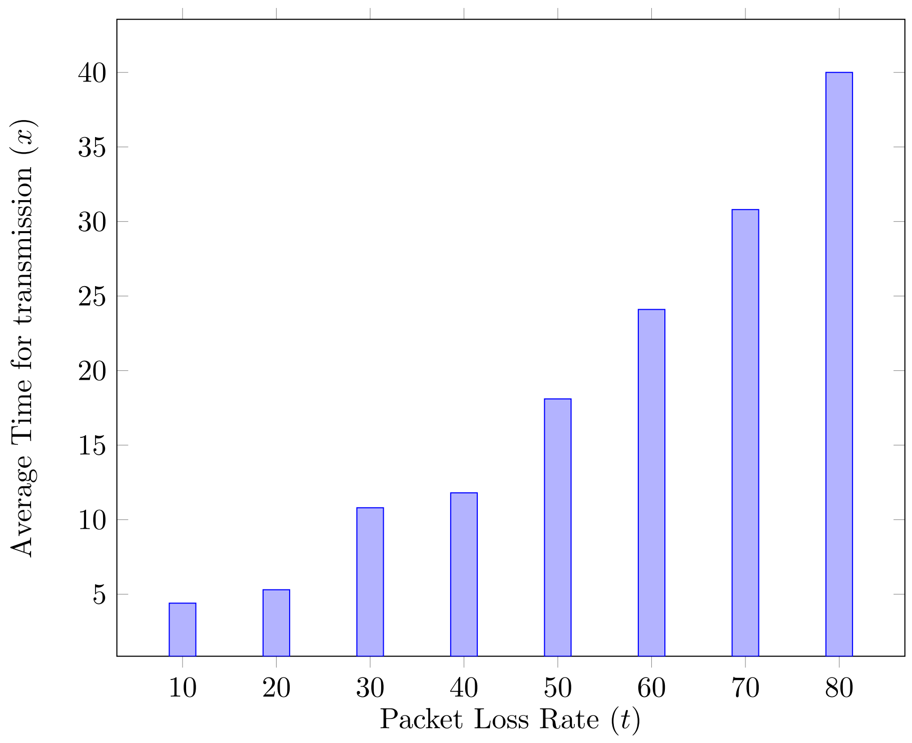


Figure 5: Estimated RTT & Sample RTT & Deviation RTT Timing

## 6.3 Throughput



## 6.4 Packet Loss



## 6.5 Comparison Between Tcp Reno and Tcp Tahoe Congestion Window

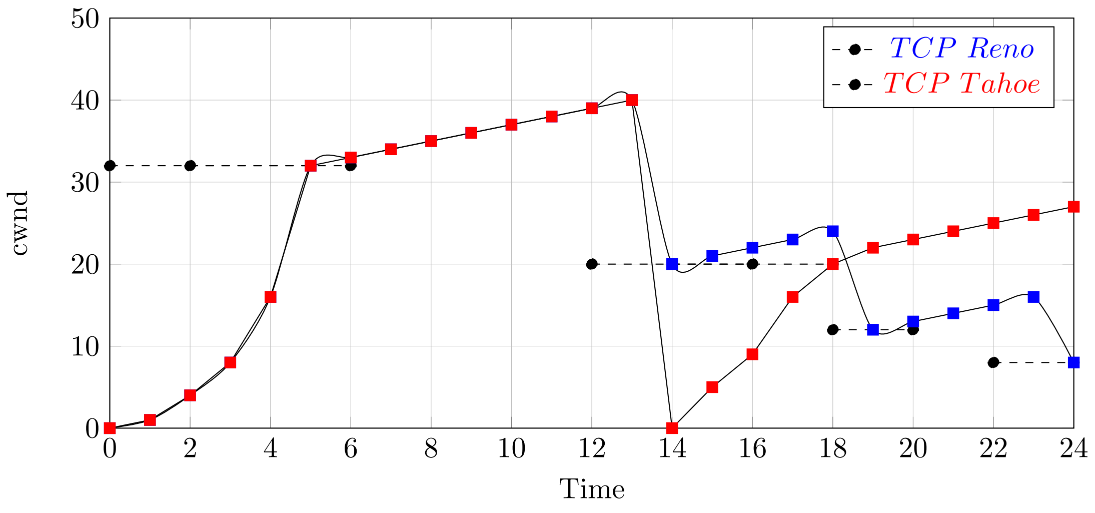


Figure 6: Comparison Between Tcp Reno and Tcp Tahoe Congestion Window

## 7 Experience

1. We had to investigate the effect of different values of the TCP Reno parameters on the performance of the algorithm.
2. We had to analyze the behavior of TCP Reno in different network conditions and compare its performance to other TCP algorithms.
3. We had to evaluate the data consistency mechanism of the distributed database by intentionally introducing inconsistencies into the database.
4. We had evaluated the performance of TCP Reno in scenarios with varying network topologies.
5. We had to examine the impact of packet drop rate on the performance of TCP Reno, and how the algorithm responds to packet loss events.
6. We had to investigate the stability and fairness of TCP Reno in scenarios with multiple competing flows and comparing its performance to other TCP algorithms.

## References

[1] Tcp Reno with example. *GeeksforGeeks*, feb 15 2022. [Online; accessed 2023-02-28].

[2] Tcp tahoe and TCP reno. *GeeksforGeeks*, feb 7 2022. [Online; accessed 2023-02-28].

[3] Contributors to Wikimedia projects. Tcp congestion control. https://en.wikipedia.org/wiki/TCP_congestion_control, feb 26 2023. [Online; accessed 2023-02-28].

[4] Wolfgang Richter. [Online; accessed 2023-02-28].