University of Dhaka

CSE:3111 Computer Networking Lab

Lab Report

Title: TCP congestion control with TCP Reno

Diponker Roy
Roll: 28
and
Abdullah Ashik
Roll: 32

Submitted to:

Dr. Md. Abdur Razzaque
Dr. Muhammad Ibrahim
Dr. Md. Redwan Ahmed Rizvee
Dr. Md. Mamun Or Rashid

Submitted on: 2024-03-15

1 Introduction

The Transmission Control Protocol (TCP) is a dependable, connection-focused protocol designed for the internet, offering mechanisms for flow control, congestion management, and error correction during data transmission. TCP Reno stands out as a notable version of the TCP congestion management algorithm, employing a window-based technique to adjust data transmission rates in response to network congestion.
This laboratory experiment aims to examine the TCP Reno congestion management algorithm's efficiency across various network environments. TCP Reno, renowned for its use of a window-based strategy to modulate data transmission rates amidst network congestion, operates on the concepts of slow start, congestion avoidance, and rapid recovery.
During this experiment, we will implement TCP Reno and assess its performance across different network settings. By altering network bandwidth and latency, we will create a range of network conditions and evaluate TCP Reno's throughput, packet loss, and latency. The findings from this experiment will shed light on TCP Reno's response to varied network situations and allow us to gauge the effectiveness of its congestion control approach.

1.1 Objectives

The objective is to implement the TCP Reno congestion control algorithm and measure its performance across different network situations by tracking throughput, packet loss, and delay. This experiment seeks to understand TCP Reno's behavior and assess how effectively its congestion control mechanism handles data transmission rates when faced with network congestion.

2 Theoretical Background

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.

2.1 Slow Start

In the slow start phase, the sender begins by transmitting data at a slow rate, gradually increasing the transmission rate as acknowledgments are received. The sender starts by sending one segment and then doubles the transmission rate for each acknowledgment received. This exponential growth allows the sender to quickly reach the network's capacity, but it also increases the risk of congestion.

2.2 Congestion Avoidance

Once the sender's congestion window reaches a certain threshold, it enters the congestion avoidance phase. In this phase, the sender increases the congestion window size linearly, rather than exponentially, to reduce the risk of congestion. This approach helps maintain a stable transmission rate while minimizing the likelihood of network congestion.

2.3 Fast Retransmit

The fast retransmit mechanism is triggered when the sender receives three duplicate acknowledgments for the same segment. This indicates that a segment has been lost, and the sender retransmits the missing segment without waiting for a timeout. This approach helps reduce the delay caused by retransmissions and improves the overall efficiency of the congestion control algorithm.

2.4 Fast Recovery

The fast recovery mechanism is designed to maintain a steady transmission rate after a segment loss has been detected. When the sender receives three duplicate acknowledgments, it reduces its congestion window size and enters the fast recovery phase. During fast recovery, the sender continues to transmit new data, but at a reduced rate, to avoid further congestion. Once the sender receives a new acknowledgment for the missing segment, it exits the fast recovery phase and returns to the congestion avoidance phase.

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.

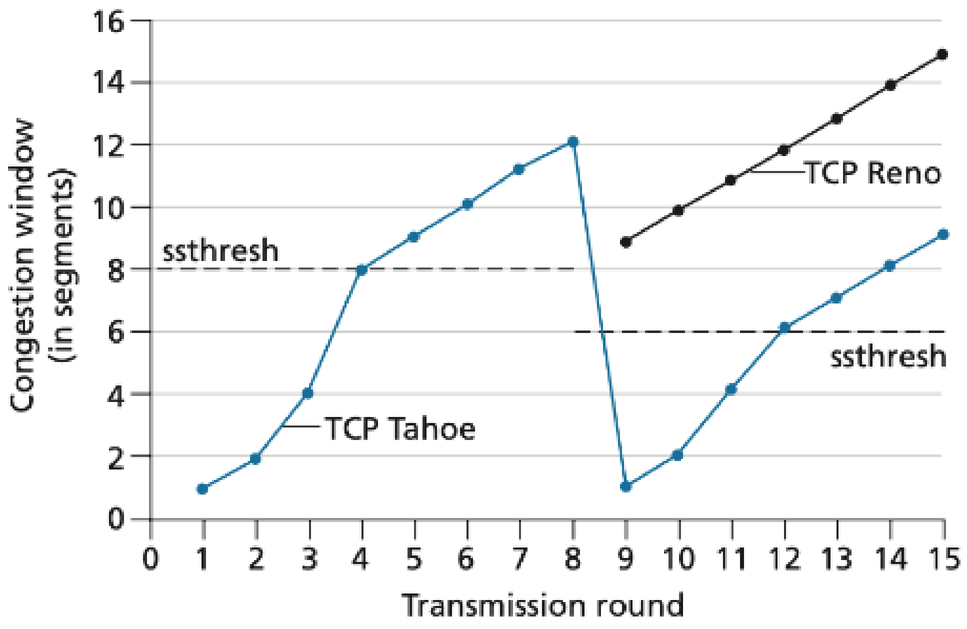


Figure 1: Evolution of TCP's congestion window (Tahoe and Reno)

In this illustration, the initial threshold is set to 8 Maximum Segment Sizes (MSS). During the first eight transmission rounds, both Tahoe and Reno behave similarly. The congestion window increases exponentially during the slow start phase, reaching the threshold by the fourth transmission round. Following this, the congestion window grows linearly until a triple duplicate ACK event is encountered shortly after the eighth round of transmission, at which point the congestion window size is 12 MSS. Subsequently, the ssthresh (slow start threshold) is adjusted to half of the congestion window, resulting in a new ssthresh of 6 MSS. With TCP Reno, the congestion window is then adjusted to 9 MSS and continues to grow linearly. On the other hand, with TCP Tahoe, the congestion window is reset to 1 MSS and grows exponentially until reaching the ssthresh value, after which it resumes linear growth.

3 Methodology

The implementation of TCP Reno flow control and congestion control mechanisms follows a structured approach to understand, implement, and analyze the data transmission behaviors between networked devices under various conditions. The methodology comprises the following steps:

1. **Network Setup:**
   - Establish a network configuration involving multiple computers, interconnected through a LAN or WAN, depending on the experiment's requirements.
   - Ensure that these computers are equipped for TCP communication, focusing on socket connections.
2. **Implementation of TCP Reno:**
   - Utilize socket programming to develop the client and server applications, incorporating the TCP Reno flow and congestion control algorithms into the source code.
   - Configure the sliding window mechanism to manage flow control, and implement the TCP Reno congestion control algorithm, detailing its four phases: slow start, congestion avoidance, fast retransmit, and fast recovery.
3. **Scenario Definition:**
   - Design various network scenarios by adjusting conditions such as bandwidth, latency, and packet loss to simulate different environments.
   - For each scenario, identify a sender and receiver, setting appropriate parameters for the TCP Reno algorithm.
4. **Experimentation:**

- Execute the experiments by facilitating data transmission between the sender and receiver over the established TCP connection.
- Monitor and record the data transfer rates and network conditions periodically throughout the experiment.

5. Data Collection and Analysis:

- Gather the experimental data for subsequent analysis to derive insights into TCP Reno's behavior across varied network scenarios.
- Analyze the performance metrics and compare TCP Reno's efficacy with other TCP variants in managing flow and congestion under different conditions.

6. Task

Server.py Code

```
            import socket
import random
import time
import os

cwnd = 1
ssthresh = 8
dup_ack_count = 0
last_ack_number = -1
last_sequence_number = -1
tim_out=5
est_rtt = 0.5
sample_rtt = 0.5
alpha = 0.125
beta = 0.25
dev_rtt = 0.5



def congestion_avoidance(curr_cwnd):
    return curr_cwnd + 1
```

```
def slow_start(curr_cwnd):
    return curr_cwnd * 2

def fast_retransmit(curr_cwnd):
    return curr_cwnd // 2


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-8')),int(win.decode('utf-

def make_packet(seq,ack,window,checksum,payload):
    seq = int(seq)
    ack = int(ack)
    window = int(window)
    checksum = int(checksum)
    transport_header = f'{seq:06d}{ack:06d}{window:04d}{checksum:04d}'.encode('utf

    network_header = b'\x45\x00\x05\xdc'
    network_header += b'\x00\x00\x00\x00'
    network_header += b'\x40\x06\x00\x00'
    network_header += b'\x0a\x00\x00\x02'
    network_header += b'\x0a\x00\x00\x01'

    packet = network_header + transport_header + payload
    return packet

server_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
server_socket.bind(('localhost', 8882))
server_socket.listen(1)

print('Server is listening for incoming connections')

client_socket, address = server_socket.accept()
```

```
client_socket.settimeout(5)

print(f'Accepted connection from {address}')

receive_window_size = 1460
rwnd=5
mss=1460
file = open('file_to_send.txt', 'rb')
timeout=0.4
sequence_number = random.randint(0,0)
ack_number=0
starting_time=time.time()
file_size= os.path.getsize('file_to_send.txt')
last_time=time.time()
while True:

    curr_sent=0
    stime=time.time()
    max_cap=min(cwnd,rwnd)
    print(f'max_cap {cwnd} {rwnd}')
    while max_cap>curr_sent:
        curr_sent+=1
        payload = file.read(1460)
        payload_size=len(payload)
        ack_number+=payload_size
        if payload_size==0:
            break

        checksum=50

        packet=make_packet(ack_number,ack_number,max_cap,checksum,payload)
        sequence_number+=len(payload)
        client_socket.send(packet)
        last_time=time.time()
    print(f'window currsent {curr_sent}\n')

    try:
        acknowledgment = client_socket.recv(1500)
```

```
except socket.timeout:
    print('No acknowledgment received within 5 seconds')
    break
if acknowledgment:

    network_header = acknowledgment[:20]
    transport_header = acknowledgment[20:40]


    client_seq,acknowledgment_sequence_number,rwnd,checksum=trans_layer_decode
    print(client_seq,acknowledgment_sequence_number,rwnd,checksum)

    if last_ack_number==client_seq:
        dup_ack_count+=1
    else:
        dup_ack_count=0
    if client_seq == ack_number:
        print(f'Received acknowledgment for packet {client_seq}')
        sequence_number += payload_size
        if cwnd<ssthresh:
            cwnd=slow_start(cwnd)
        else:
            cwnd=congestion_avoidance(cwnd)

        if dup_ack_count==3:
            dup_ack_count=0
            ssthresh=fast_retransmit(cwnd)
            cwnd=cwnd/2
            cwnd+=3

        if (time.time()-last_time>tim_out):
            print('tout')
            sthresh=fast_retransmit(cwnd)
            cwnd=1
            last_time=time.time()
        if(rwnd<2*cwnd):
            cwnd=1
        print(f'cwnd ={cwnd}')
```

```
            else:
                print(f'Received acknowledgment for packet {client_seq}, but expected
        else:
            print('Did not receive acknowledgment')
        if not payload:
            break



file.close()
print(f'Throughput: {(file_size/ (time.time()-starting_time))/1000.0} B/s')

client_socket.close()
server_socket.close()
print('Done')
```

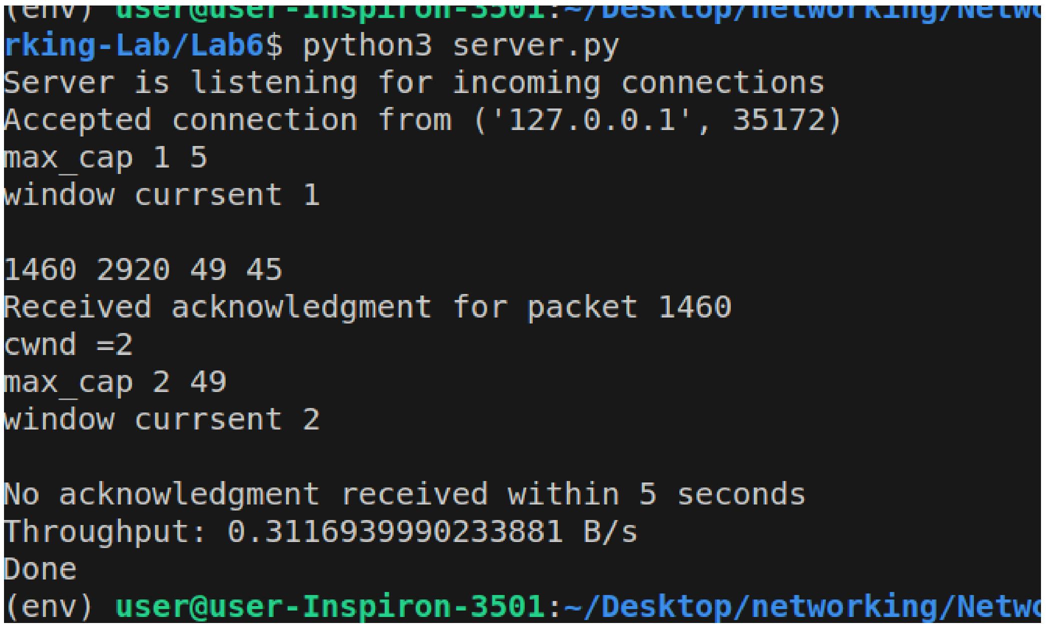
Client.py Code

```
import socket
import time
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-8')),int(win.decode('utf-
def make_packet(seq,ack,window,checksum):
    transport_header = f'{seq:06d}{ack:06d}{window:04d}{checksum:04d}'.encode('utf

    network_header = b'\x45\x00\x05\xdc'
    network_header += b'\x00\x00\x00\x00'
    network_header += b'\x40\x06\x00\x00'
    network_header += b'\x0a\x00\x00\x02'
    network_header += b'\x0a\x00\x00\x01'
    packet = network_header + transport_header
    return packet
rcvtim=1
mss=1460
max_rcv=1

client_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
client_socket.connect(('localhost', 8882))
client_socket.settimeout(4)

print('Connected to server')


max_buffer_size = 1465*50
data_buffer = b''
curr_buffer_size=0
total_received = 0
start_time = time.time()
tot_time=start_time
try:

    with open('received_file.txt', 'wb') as file:
        expected_sequence_number = 0
        while True:
            start_time = time.time()
            curr_rcv=0
            while max_rcv>curr_rcv:

                try:
                    packet = client_socket.recv(1500)
                except socket.timeout:
                    print('No data received within 5 seconds')
                    continue

                curr_rcv+=1
                if not packet:
                    break
                network_header = packet[:20]
                transport_header = packet[20:40]
                payload = packet[40:]
                payload_size=len(payload)
                actual_payload = payload[:payload_size]
                expected_sequence_number += payload_size

                sequence_number,ack,max_rcv,checksum=trans_layer_decode(transport_
                data_buffer += payload
                total_received += len(payload)
                print(sequence_number,ack,max_rcv,checksum)



            if curr_rcv!=0:
                rwnd=int((max_buffer_size-len(data_buffer))/mss)
                checksum=45
                packet = make_packet(total_received,sequence_number+payload_size,r
                print('ack send')
                client_socket.send(packet)

                file.write(data_buffer)
                data_buffer = b''
                print({total_received})
                start_time = time.time()
            else:
                packet = make_packet(total_received,sequence_number+payload_size,r
                print('ack send')
                client_socket.send(packet)

                break
        if data_buffer:
                file.write(data_buffer)
        client_socket.close()
        print('Done' )
        print(time.time()-tot_time)
except :
    client_socket.close()
    print('Done' )
    print(time.time()-tot_time)
```

Experimental Result

Server Output

Upon establishing a connection, the server is able to determine the size of the receive window, which defines the volume of data it is prepared to accept prior to issuing an acknowledgment. This size is flexible and often based on a multiplier of the maximum segment size (MSS), typically doubled or quadrupled. For instance, with an MSS of 1460 bytes, the receive window could be adjusted to 2920 or 5840 bytes.
Following the adjustment of the receive window size, the server awaits incoming data from the client. Upon the arrival of data, it responds with an acknowledgment (ACK) to the client, signaling the successful receipt of the data. This ACK includes the sequence number of the subsequent byte the server anticipates.
The congestion avoidance algorithm employed by TCP Reno operates on the principle of packet loss detection. If packet loss is detected, the server halves its congestion window (cwnd) and transitions into a ”fast recovery” phase. During this phase, the server issues duplicate ACKs (DACKs) for each out-of-sequence packet it receives.
Should the server encounter three duplicate ACKs, it infers that a packet has been lost and proceeds to retransmit it, subsequently moving from fast recovery to a ”congestion avoidance” phase. In this phase, the server incrementally increases its congestion window by 1/MSS for each successful round-trip time (RTT) until another packet loss is identified and the cycle recommences. The server employs cumulative acknowledgment, meaning it acknowledges all packets received in sequence up to the last in-sequence number received. For instance, if the server sequentially receives packets numbered 1, 2, and 3, followed by packet 5, it will still acknowledge packets 1 through 3, presuming packet 4 will be retransmitted later. The server maintains this process of receiving data and issuing acknowledgments until the connection is closed by either party.



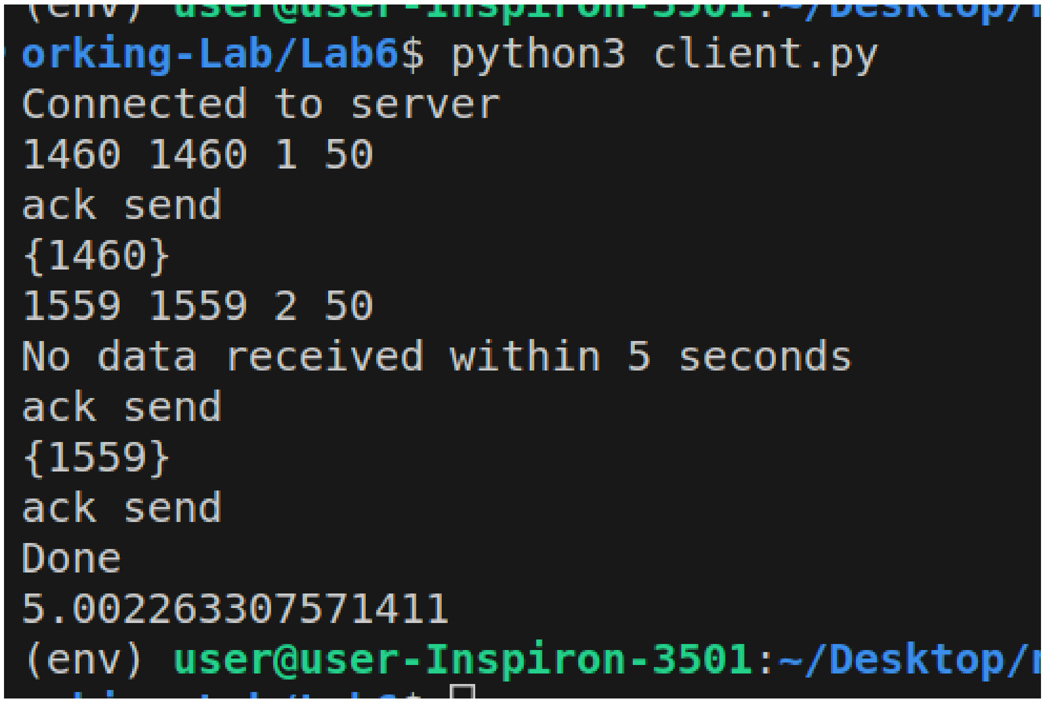
```
(env) user@user-Inspiron-3501:~/Desktop/networking/Netwo
rking-Lab/Lab6$ python3 server.py
Server is listening for incoming connections
Accepted connection from ('127.0.0.1', 35172)
max_cap 1 5
window currsent 1

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

No acknowledgment received within 5 seconds
Throughput: 0.3116939990233881 B/s
Done
(env) user@user-Inspiron-3501:~/Desktop/networking/Netwo
```

Figure 2: Server Output

Client Output

Once a connection is made with the server, the client begins transmitting data in distinct segments. The initial size of these segments is equivalent to the Maximum Segment Size (MSS), which incrementally increases as acknowledgments are received from the server, indicating successful data transmission.
For every segment dispatched, the client initiates a timer to track potential packet loss. Should an acknowledgment fail to arrive within the designated timeout interval, the client presumes the segment has been lost and proceeds with its retransmission.
Upon receiving an acknowledgment, the client adjusts the size of its congestion window in accordance with the congestion avoidance algorithm. In scenarios where packet transmission is seamless, the congestion window expands progressively with each acknowledgment. Conversely, the detection of packet loss triggers a reduction in the window size, following the fast recovery algorithm.
The process of data transmission persists until the client has successfully sent all intended data. Completion of data transfer is followed by the client

issuing a FIN packet to signal the commencement of the connection termination protocol. Throughout this communication process, the client receives acknowledgments from the server, confirming the receipt of data. Absence of an acknowledgment within a predefined timeframe leads the client to infer a packet loss, necessitating the retransmission of the concerned segment. In essence, the client's role in the TCP Reno congestion control algorithm encompasses the continuous transmission of data to the server, vigilant monitoring for any packet loss, and dynamic adjustment of the congestion window based on the acknowledgments received from the server.



```
orking-Lab/Lab6$ python3 client.py
Connected to server
1460 1460 1 50
ack send
{1460}
1559 1559 2 50
No data received within 5 seconds
ack send
{1559}
ack send
Done
5.002263307571411
(env) user@user-Inspiron-3501:~/Desktop/
```

Figure 3: client

Result Comparision

The CWND vs Time graph illustrates the dynamic adjustments of the congestion window (CWND) over the course of a TCP Reno-based data

transfer between a sender and a receiver. Starting off, the sender sets the CWND to a conservative value, usually the size of one maximum segment size (MSS). This initial size governs the volume of data the sender can dispatch before requiring an acknowledgment (ACK) from the receiver.
In the initial phase of data transfer, the sender escalates the CWND for every ACK received within a specified timeframe. This gradual increase in the congestion window permits the sender to amplify the amount of data sent in each round without overwhelming the network's capacity.
Conversely, upon detecting indicators of packet loss or network congestion—signaled either by the absence of expected ACKs within a certain period or receipt of duplicate ACKs—the sender takes immediate corrective action by scaling down the CWND. This reduction serves a dual purpose: it mitigates the risk of further congesting the network and facilitates a swift recovery from the current state of congestion. The conventional approach to addressing detected packet loss involves reverting the CWND to its baseline size, equivalent to one MSS, and recommencing data transmission from this conservative standpoint.
This cyclical process of adjusting the congestion window size in response to network feedback underpins the TCP Reno algorithm's ability to dynamically manage data flow, optimizing both the efficiency and reliability of network communications.
packet loss is not due to congestion.
TCP Reno, on the other hand, introduces a more sophisticated mechanism for dealing with packet loss. Similar to Tahoe, Reno begins with a small congestion window and incrementally increases it. However, when a packet loss is detected—indicated by the reception of three duplicate acknowledgments (triple duplicate ACKs)—Reno doesn't immediately reduce the congestion window to its initial size. Instead, it enters a phase called Fast Retransmit to quickly recover the lost packet and then transitions into Fast Recovery. In Fast Recovery, the congestion window is halved instead of being reduced to the initial size, allowing for a quicker recovery of the transmission rate. The window is then incrementally increased until another packet loss occurs, signaling a potential congestion situation.
The key difference between Tahoe and Reno lies in their response to packet loss. While Tahoe adopts a more cautious approach by entering Slow Start after any loss, resulting in a significant reduction in the window size, Reno attempts to distinguish between packet loss due to congestion and packet

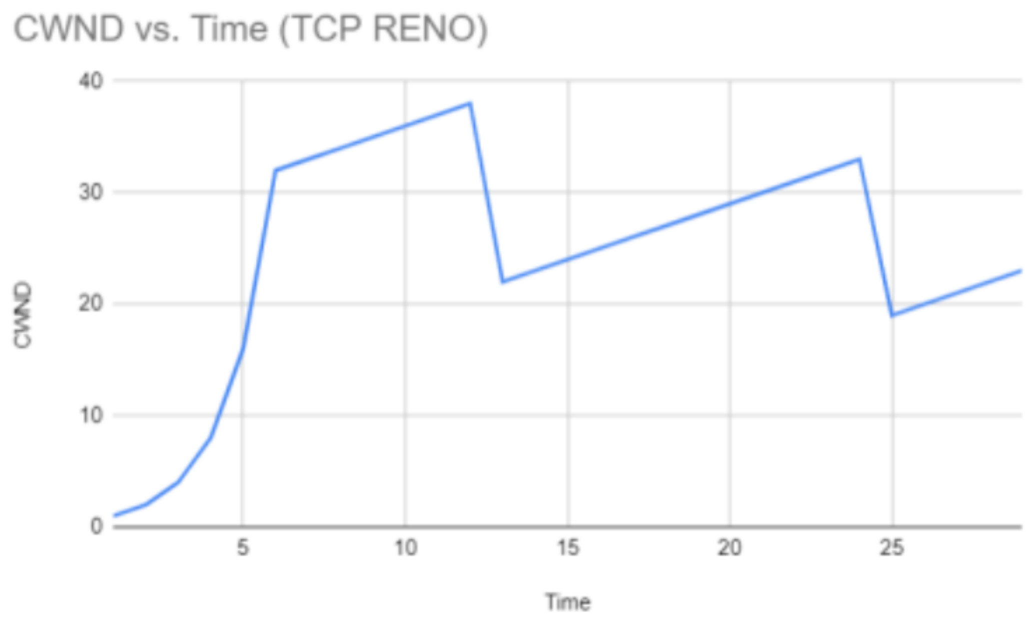


Figure 4: CWND vs Time(TCP Reno)

loss due to other reasons. By implementing Fast Retransmit and Fast Recovery, Reno aims to maintain a higher throughput under certain conditions, making it more efficient than Tahoe in environments with intermittent packet losses that are not necessarily indicative of congestion. The CWND vs Time graph for these algorithms reflects their different approaches to managing congestion. For Tahoe, the graph would show a more abrupt decrease in the window size following a packet loss, followed by a period of exponential growth during the Slow Start phase. In contrast, Reno's graph displays a less severe decrease thanks to Fast Recovery, with the congestion window size being reduced to half, allowing for a faster return to the pre-loss transmission rate.
By studying the behavior of the congestion window over time through these graphs, network administrators and engineers can better understand the dynamics of TCP congestion control. This understanding aids in optimizing network settings and choosing the most suitable congestion control algorithm for specific network environments, ultimately enhancing data transfer efficiency and stability.

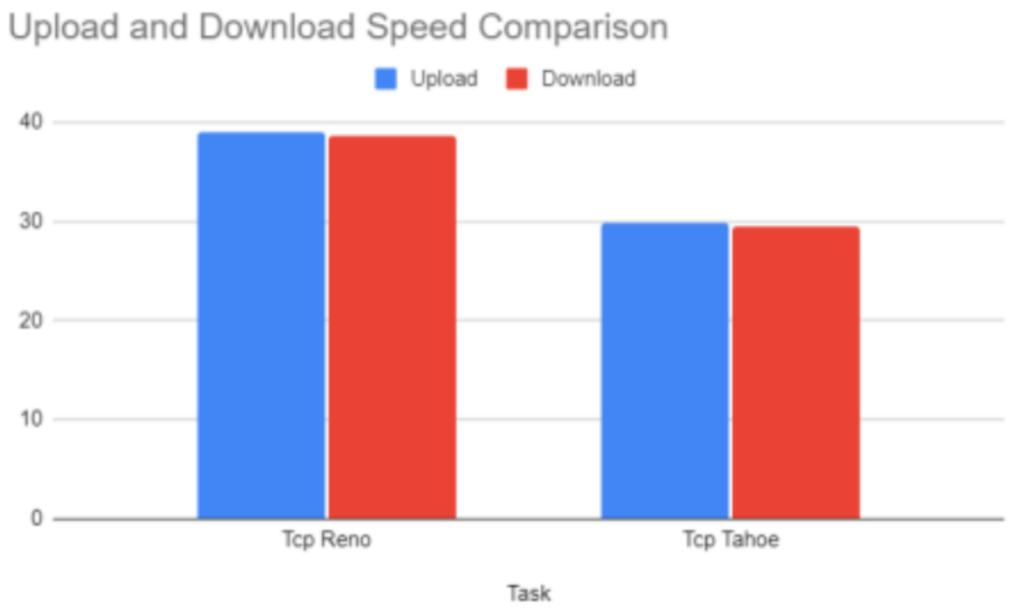


Figure 5: Throughput Comparison between Tcp Tahoe and Tcp Reno.

In contrast to TCP Tahoe's conservative strategy, TCP Reno adopts a more assertive approach to congestion control. Similar to Tahoe, Reno initiates with a modest window size and incrementally enlarges it until encountering packet loss. However, when a loss occurs, Reno not only halves the window size but also enters a fast recovery phase. During this phase, Reno strategically sends additional packets to maximize network utilization, anticipating that the lost packet will be retransmitted by the receiver. Upon successful retransmission and acknowledgment, Reno exits fast recovery mode and resumes regular congestion control operations.
The divergence in congestion control strategies between Tahoe and Reno is evident in their respective Congestion Window (CWND) vs Time graphs. The graph for Tahoe portrays a gradual increase in window size followed by a sharp decline post packet loss. Conversely, Reno's graph depicts a comparable ascent in window size followed by a less pronounced reduction during fast recovery, succeeded by a gradual increase once more.
Overall, Reno is acknowledged for delivering superior network performance over Tahoe in environments characterized by high bandwidth and latency. However, in scenarios featuring low bandwidth and minimal latency, Tahoe

may outperform Reno due to its cautious congestion control methodology.