University of Dhaka

Department of Computer Science and Engineering

CSE-3111 : Computer Networking Lab

Lab Report 5 : Implementation of TCP flow control and congestion control algorithm (TCP Tahoe).

Submitted By:

Afser Adil Olin

Roll No : AE-47

Anika Tabassum

Roll No : Rk-61

Submitted On :

February 24, 2023

Submitted To :

Dr. Md. Abdur Razzaque

Md Mahmudur Rahman

Md. Ashraful Islam

Md. Fahim Arefin

Contents

1 Introduction

The objective of implementing TCP flow control and congestion control algorithm (TCP Tahoe) is to improve the reliability and efficiency of data transmission over a network by regulating the flow of data and preventing network congestion.

1.1 Objectives

- To gather knowledge about how TCP controls the flow of data between a sender and a receiver
- To learn how TCP controls and avoids the congestion of data when a sender or receiver detects a congestion in the link in-between them. (TCP Tahoe)

2 Theory

TCP is one of the protocols of the transport layer for network communication. TCP provides reliable, ordered, and error- checked delivery of a stream of bytes between applications running on hosts communicating via an IP network. TCP is connection-oriented, and a connection between client and server is established before data can be sent. The server must be listening (passive open) for connection requests from clients before a connection is established. Three-way handshake (active open), retransmission, and error-detection adds to reliability. Thus TCP can maintain various operations to establish perfect communications between a pair of hosts, e.g connection management, error detection, error recovery, congestion control, connection termination, flow control, etc. In this lab, we will have a look at the flow control mechanism and congestion control mechanisms of the TCP protocol.
TCP uses a sliding window flow control protocol. In each TCP segment, the receiver specifies in the receive window field the amount of additionally received data (in bytes) that it is willing to buffer for the connection. The sending host can send only up to that amount of data before it must wait for an acknowledgement and window update from the receiving host.

2.1 TCP Tahoe

TCP Tahoe is a congestion control algorithm that is used to manage the flow of data in a network and prevent congestion. It was one of the first congestion control algorithms implemented in TCP and is still widely used today.The congestion window is one of the factors that determines the number of bytes that can be outstanding at any time. It is a means of stopping a link between the sender and the receiver from getting overloaded with too much traffic and it is calculated by estimating how much congestion there is between the sender and receiver.
Slow-start is part of the congestion control strategy used by TCP. Slow-start is used in conjunction with other algorithms to avoid sending more data than the network is capable of transmitting, that is, to avoid causing network congestion.
The congestion window (CWND) is maintained by the sender. Note that this is not to be confused with the TCP window size which is maintained by the receiver.
A sample pictorial behavior of TCP Tahoe is given below :

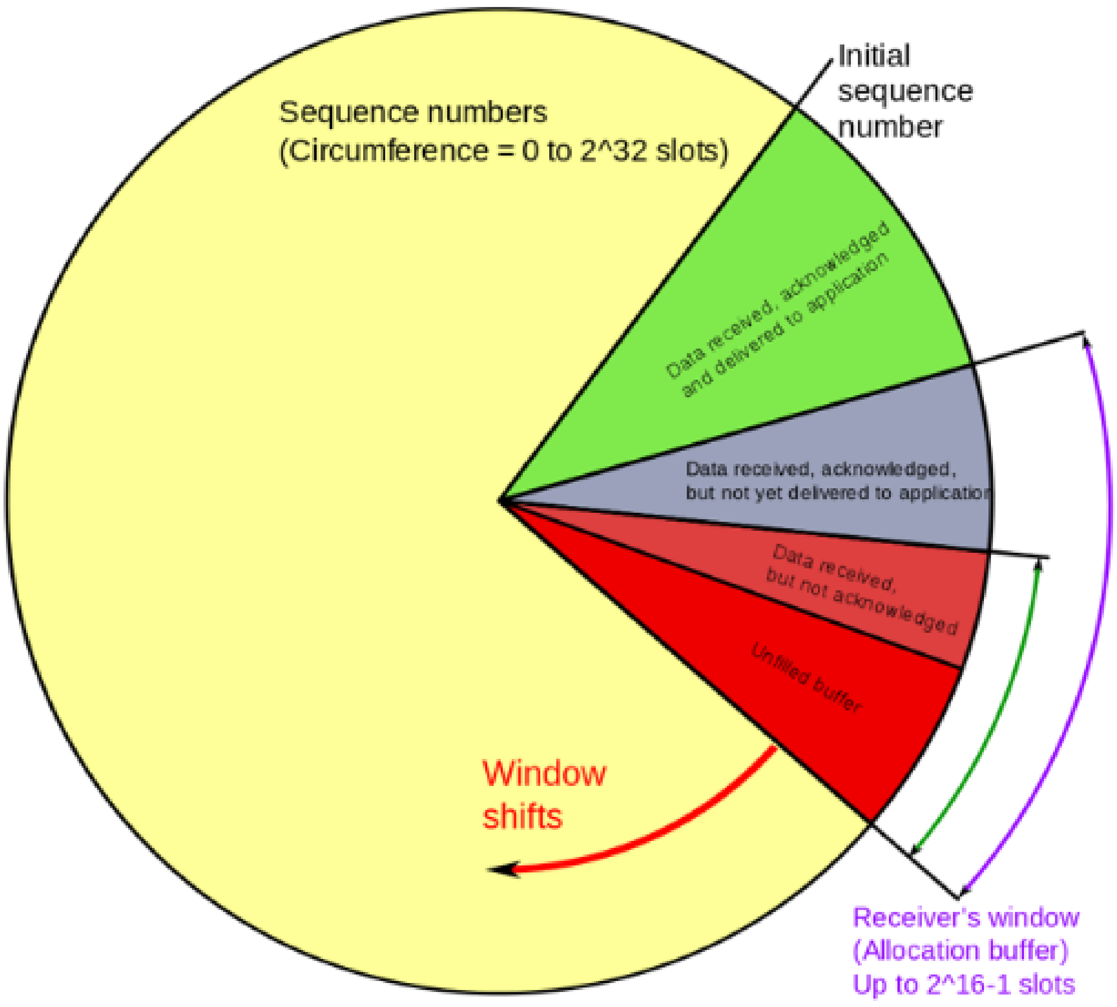

Sequence numbers
(Circumference = 0 to 2^32 slots)
Initial sequence number
Data received, acknowledged and delivered to application
Data received, acknowledged, but not yet delivered to application
Data received, but not acknowledged
Unfilled buffer
Window shifts
Receiver's window (Allocation buffer) Up to 2^16-1 slots

Figure 1: congestion control algorithm for TCP Tahoe

2.2 TCP Flow Control

TCP (Transmission Control Protocol) flow control is a mechanism that allows a sender and receiver to manage the amount of data being transmitted over a network connection. The purpose of flow control is to ensure that a sender does not overwhelm a receiver with too much data at once, leading to packet loss, congestion, or other network problems.

In TCP, flow control is achieved through the use of a sliding window protocol. The receiver advertises a window size to the sender, indicating the amount of data that can be sent before the receiver's buffer becomes full. The sender then sends data up to the advertised window size, and waits for an acknowledgement from the receiver before sending more data.

As the receiver processes the data, it frees up space in its buffer, and advertises a larger window size to the sender. This allows the sender to continue sending data without overwhelming the receiver. If the receiver's buffer becomes full, it advertises a smaller window size, causing the

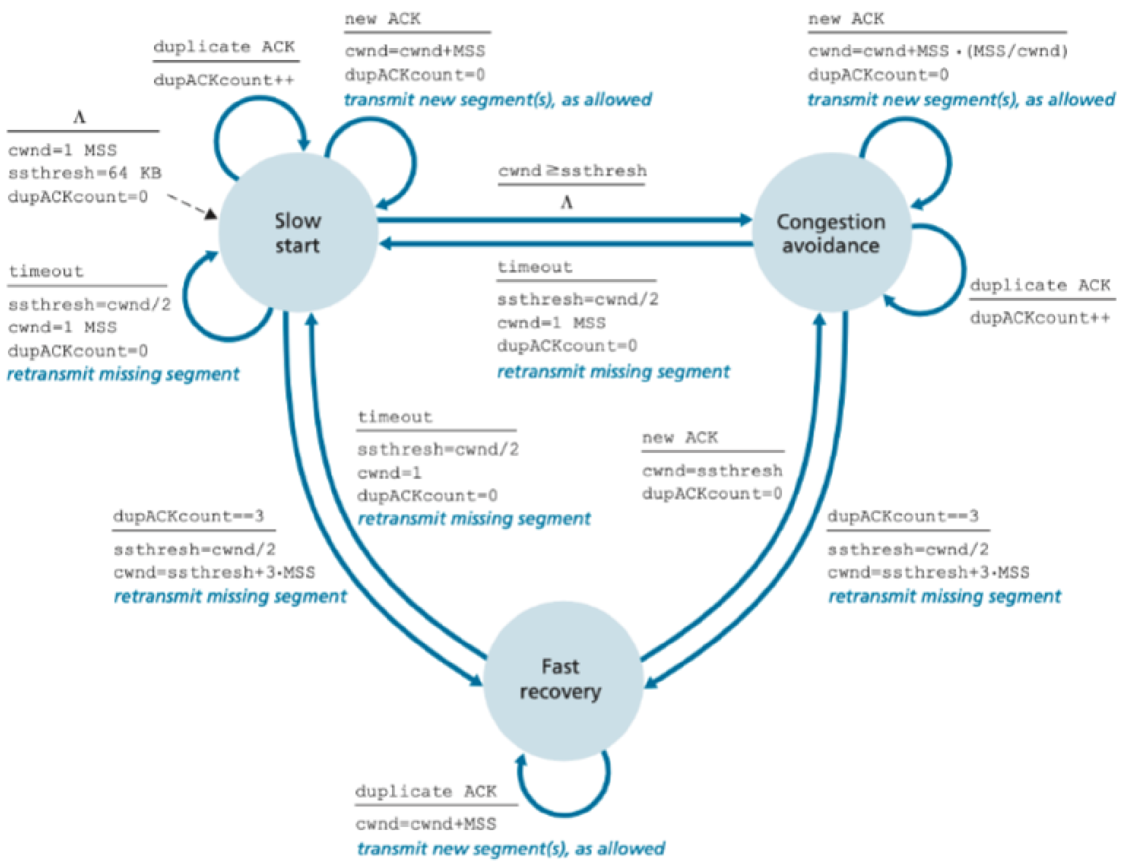


Figure 2: FSM description of TCP Congestion Control

sender to slow down its transmission rate.

TCP flow control is essential for maintaining the reliability and efficiency of network communication. Without flow control, a sender could flood a receiver with data, causing packet loss and congestion, leading to poor network performance.

2.3 TCP Congestion Control

TCP Tahoe is a congestion control algorithm that uses three mechanisms to manage the flow of data in a network: slow start, congestion avoidance, and fast retransmit. The following is a simplified version of the TCP Tahoe algorithm:

In the server side when a connection is established, server send Maximum Segment Size. Then waits for client so that client can send data.

$$\boxed{\text{TCP Tahoe} = \text{Slow Start} + \text{AIMD} + \text{Fast Retransmit}}$$

2.3.1 Initialization

- Set the congestion window (cwnd) to 1 Maximum Segment Size (MSS).
- Set the slow start threshold (ssthresh) to a large value, e.g., the size of the receive window.

- Set the duplicate ACK counter to 0.

2.3.2 Slow Start phase

It lasts until congestion window size reaches up to "Slow Start Threshold (ssthresh)". The slow start algorithm doubles the congestion window size(cwnd) in one RTT. Initially, ssthresh is set to infinite. Subsequently, it adapts depending on the packet loss events. When cwnd becomes equal to ssthresh, slow start stops. After that AIMD phase takeovers.

- For each ACK received, increase cwnd by 1 MSS.
- If cwnd $>=$ ssthresh, switch to congestion avoidance.
- If a packet loss is detected (either by receiving 3 duplicate ACKs or a timeout), set ssthresh to cwnd/2, set cwnd to 1 MSS, and go to step 4 (fast retransmit).

2.3.3 AIMD phase

It starts when the slow start stops. Additive Increase increases cwnd by 1 and Multiplicative Decrease reduces ssthresh to 50 percent of cwnd. Note that cwnd is 'not' reduced by 50 percent but ssthresh.

- For each ACK received, increase cwnd by (MSS * MSS) / cwnd.
- If a packet loss is detected (either by receiving 3 duplicate ACKs or a timeout), set ssthresh to cwnd/2, set cwnd to 1 MSS, and go to step 4 (fast retransmit).

The TCP Tahoe algorithm repeats these steps for the duration of the connection, adjusting the congestion window and retransmitting lost packets as necessary to manage the flow of data and prevent congestion in the network.

2.3.4 Fast Retransmit Phase

It is the loss detection algorithm. It is triggered by 3 duplicate acknowledgments. On packet loss detection, it resets cwnd to initcwnd.

- Retransmit the lost packet.
- Set the duplicate ACK counter to 0.
- Return to step 2.4.1 (slow start).

2.4 The EWMA equation

2.4.1 Defining the EWMA equation

The EWMA (Exponential Weighted Moving Average) equation is a mathematical formula used to calculate a weighted moving average of a time series data. The EWMA is commonly used in finance, statistics, and engineering to smooth out noise in data and estimate trends.

$$\boxed{\text{EMA(t)} = \alpha * \text{P(t)} + (1 - \alpha) \quad * \quad \text{EMA(t-1)}}$$

where: EMA(t) is the exponential moving average at time t. α(alpha) is the smoothing factor or weight, which is a value between 0 and 1. P(t) is the value of the network measurement at time t. EMA(t -1) is the exponential moving average at the previous time period

2.4.2 Calculating the Estimated RTT

For our purpose we are going to use this to estimate the RTT and the average DevRTT (Deviation from RTT). The EWMA equation for estimating the RTT is as follows:

$$\boxed{\text{Estimated\_RTT} = \alpha * \text{Sample\_RTT} + (1 - \alpha) \quad * \quad \text{Estimated\_RTT}_{pre}}$$

where, Estimated_RTT is the estimated RTT at the current time. α is the smoothing factor or weight, which is a value between 0 and 1, this is usually 0.125 SampleRTT is the RTT measured for a single packet.

2.4.3 Calculating the Dev RTT

The EWMA equation for estimating the deviation of the RTT is as follows:

$$\boxed{\text{Deviation\_RTT} = \alpha * \|Sample\_RTT - \quad \text{Estimated\_RTT}\| + (1\alpha) * \quad \text{Deviation\_RTT}_{prev}}$$

where, Deviation_RTT is the estimated deviation of the RTT at the current time. α is the smoothing factor or weight, which is a value between 0 and 1. Sample_RTT is the RTT measured for a single packet.

2.5 Finding the Timeout Value

Incorporating both the estimated Round Trip Time (RTT) and the deviation of the RTT can help in more accurately estimating the appropriate retransmission timeout (RTO) value. The RTO value is the amount of time that the sender waits before retransmitting a packet that has not been acknowledged. One way to incorporate both the estimated RTT and deviation of the RTT is to use the following equation to calculate the RTO:

$$\boxed{\text{RTO} = \text{Estimated\_RTT} + \quad 4 * \quad \text{Deviation\_RTT}}$$

This equation takes into account the estimated RTT, which represents the average time it takes for a packet to travel from the sender to the receiver and back, and the deviation of the RTT, which represents the variability in the RTT due to network congestion or other factors.

3 Methodology

- We need to study and understand the TCP Tahoe algorithm.
- We need to select a suitable network simulation tool:
- We need to develop the implementation of TCP Tahoe which includes include the TCP Tahoe flow control and 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 Tahoe 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 Tahoe connection with Flow control, 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 Flow Control

At first , we send a packet of data from server to client. If the sequence is a valid sequence , then we continued to get more consecutive packet for 500 ms. If we get any packet that is out of sequence or the buffer get full , we immediately break the loop. After the loop , we send an acknowledgement about the last packet we got which was in sequence. We take an buffer memory to store those incoming consecutive packet. After sending acknowledgement we take the data from the buffer and write it down to the file. We continued the process until the server stop sending data for some time until we get a timeout. Then the client take for granted that, the data sending is completed.

4.3 TCP Congestion Control

TCP Tahoe 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 Tahoe 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 Tahoe enters a ”fast recovery” phase where it tries to recover from the loss as quickly as possible by sending a small number of packets, known as a ”burst,” to determine if the network is still congested. If no further packet loss occurs, TCP Tahoe re-enters the congestion avoidance phase and gradually increases its sending rate again.

4.4 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.

4.5 Check Sum

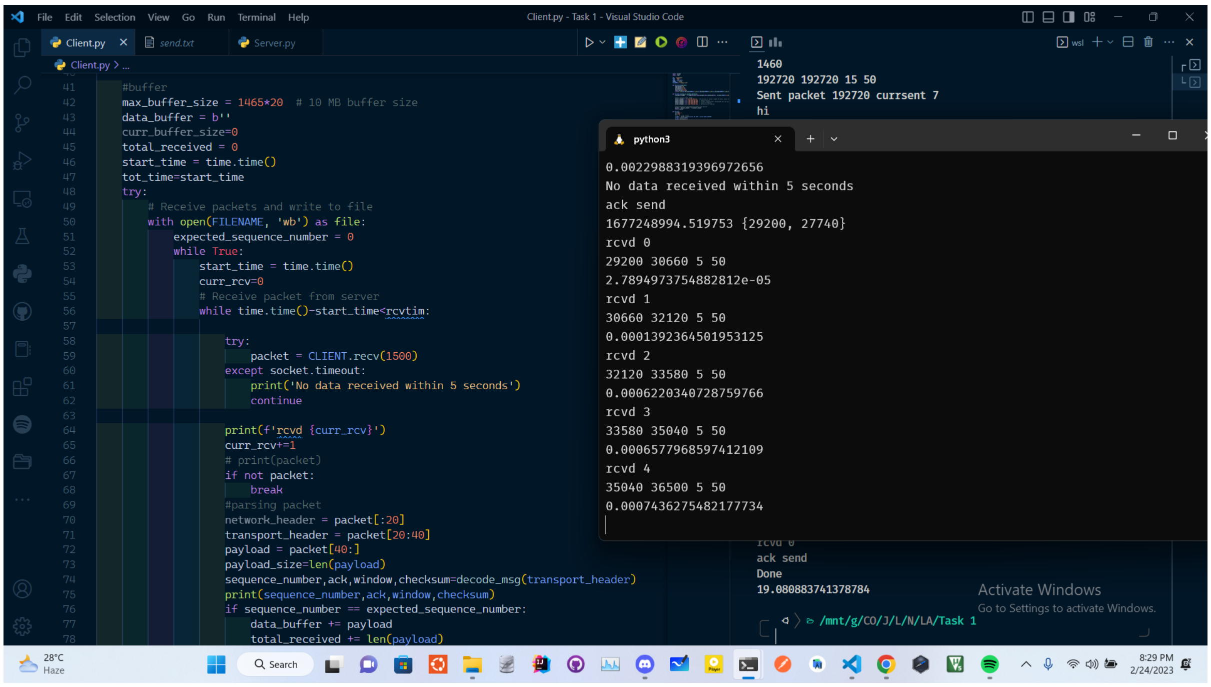
We have implemented a checksum function for 16 bit number.The sender first calculate for the checksum for the ”*payload*” which contains a packet of data via that function and send it by the header.In the client side, The checksum is checked if the data is mismatched.

4.6 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 Flow Control

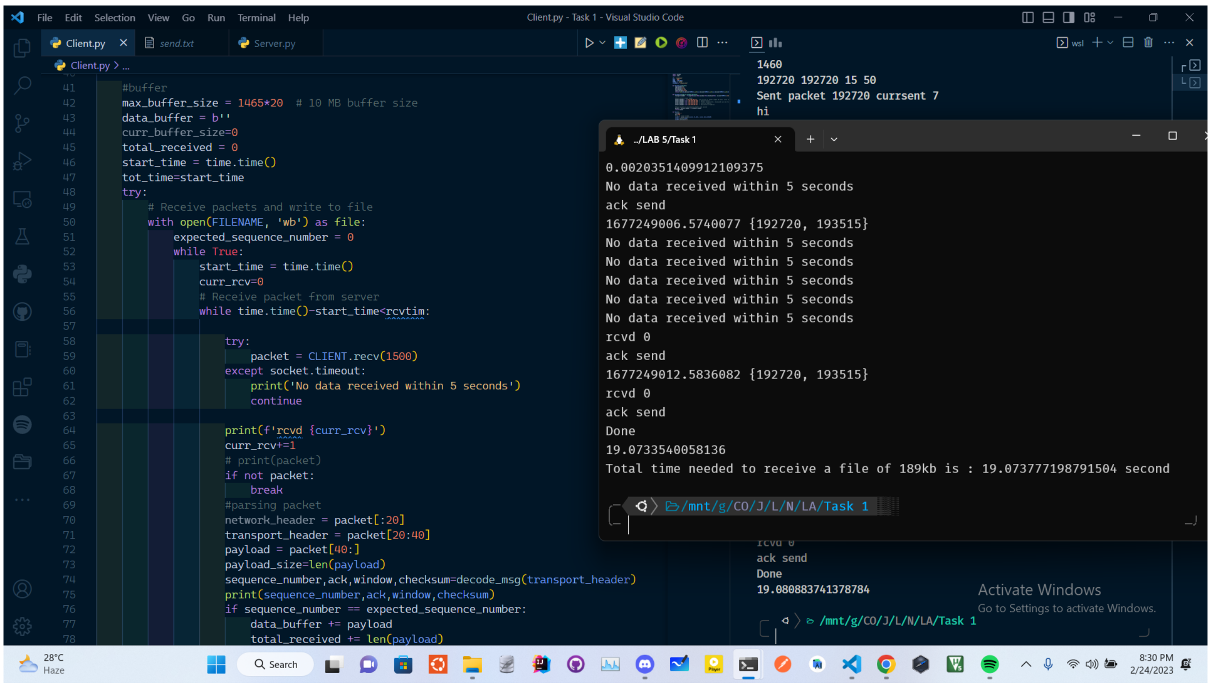


```python
#buffer
max_buffer_size = 1465*20  # 10 MB buffer size
data_buffer = b''
curr_buffer_size=0
total_received = 0
start_time = time.time()
tot_time=start_time
try:
    # Receive packets and write to file
    with open(FILENAME, 'wb') as file:
        expected_sequence_number = 0
        while True:
            start_time = time.time()
            curr_rcv=0
            # Receive packet from server
            while time.time()-start_time<rcvtim:

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

                print(f'rcvd {curr_rcv}')
                curr_rcv+=1
                # print(packet)
                if not packet:
                    break
                #parsing packet
                network_header = packet[:20]
                transport_header = packet[20:40]
                payload = packet[40:]
                payload_size=len(payload)
                sequence_number,ack,window,checksum=decode_msg(transport_header)
                print(sequence_number,ack,window,checksum)
                if sequence_number == expected_sequence_number:
                    data_buffer += payload
                    total_received += len(payload)
```

```
1460
192720 192720 15 50
Sent packet 192720 currsent 7
hi
```

```
0.0022988319396972656
No data received within 5 seconds
ack send
1677248994.519753 {29200, 27740}
rcvd 0
29200 30660 5 50
2.7894973754882812e-05
rcvd 1
30660 32120 5 50
0.0001392364501953125
rcvd 2
32120 33580 5 50
0.0006220340728759766
rcvd 3
33580 35040 5 50
0.0006577968597412109
rcvd 4
35040 36500 5 50
0.0007436275482177734
```

```
ack send
Done
19.080883741378784

/mnt/g/CO/J/L/N/LA/Task 1
```



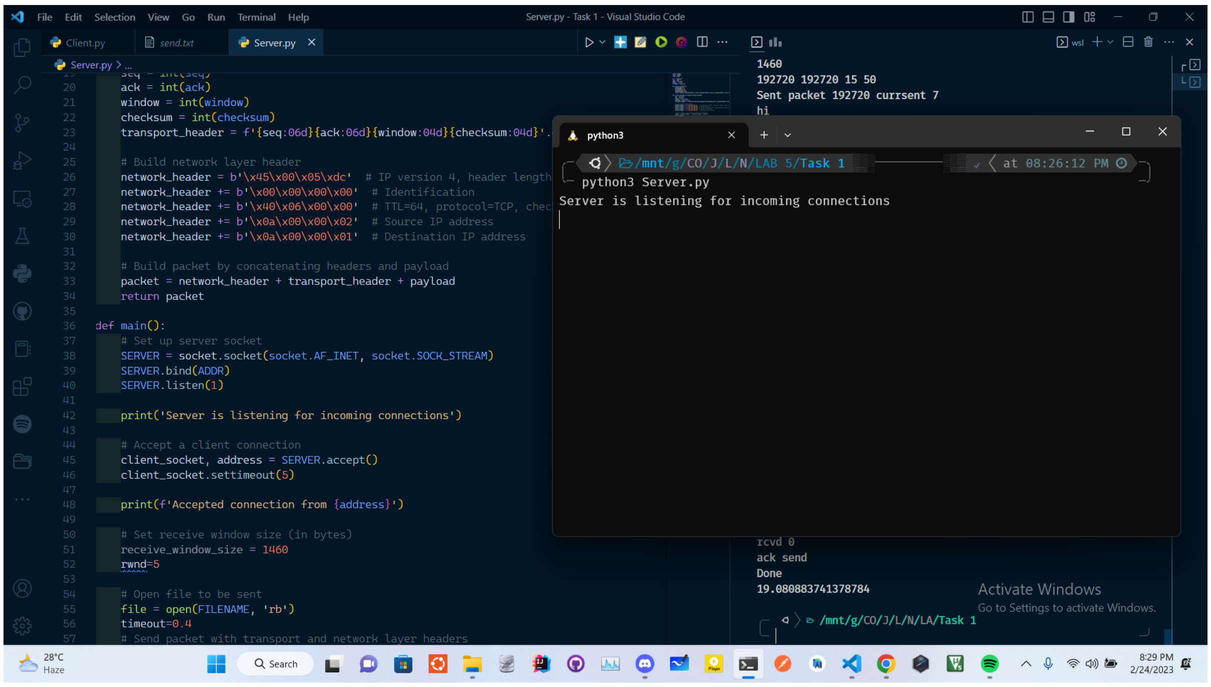
```python
#buffer
max_buffer_size = 1465*20  # 10 MB buffer size
data_buffer = b''
curr_buffer_size=0
total_received = 0
start_time = time.time()
tot_time=start_time
try:
    # Receive packets and write to file
    with open(FILENAME, 'wb') as file:
        expected_sequence_number = 0
        while True:
            start_time = time.time()
            curr_rcv=0
            # Receive packet from server
            while time.time()-start_time<rcvtim:

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

                print(f'rcvd {curr_rcv}')
                curr_rcv+=1
                # print(packet)
                if not packet:
                    break
                #parsing packet
                network_header = packet[:20]
                transport_header = packet[20:40]
                payload = packet[40:]
                payload_size=len(payload)
                sequence_number,ack,window,checksum=decode_msg(transport_header)
                print(sequence_number,ack,window,checksum)
                if sequence_number == expected_sequence_number:
                    data_buffer += payload
                    total_received += len(payload)
```

```
1460
192720 192720 15 50
Sent packet 192720 currsent 7
hi
```

```
0.0020351409912109375
No data received within 5 seconds
ack send
1677249006.5740077 {192720, 193515}
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
rcvd 0
ack send
1677249012.5836082 {192720, 193515}
rcvd 0
ack send
Done
19.0733540058136
Total time needed to receive a file of 189kb is : 19.073777198791504 second

/mnt/g/CO/J/L/N/LA/Task 1
```

```
ack send
Done
19.080883741378784

/mnt/g/CO/J/L/N/LA/Task 1
```

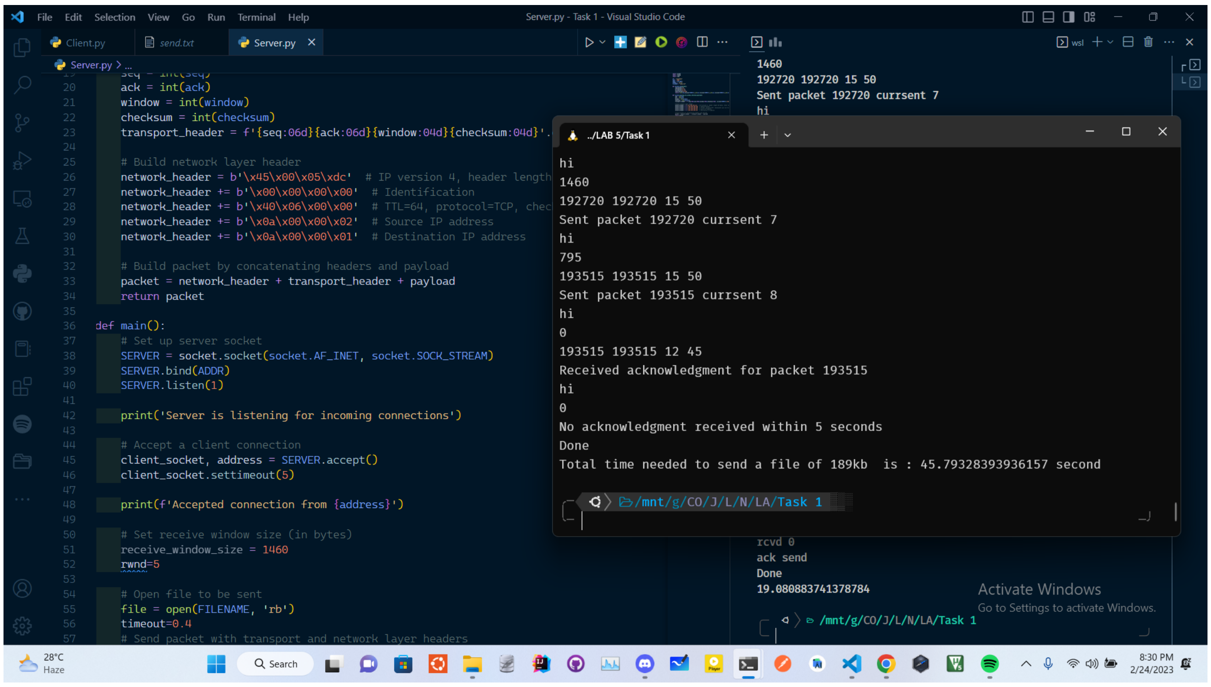
Figure 3: Client Side View



```python
    seq = int(seq)
    ack = int(ack)
    window = int(window)
    checksum = int(checksum)
    transport_header = f'{seq:06d}{ack:06d}{window:04d}{checksum:04d}'.

    # Build network layer header
    network_header = b'\x45\x00\x05\xdc'  # IP version 4, header length
    network_header += b'\x00\x00\x00\x00'  # Identification
    network_header += b'\x40\x06\x00\x00'  # TTL=64, protocol=TCP, chec
    network_header += b'\x0a\x00\x00\x02'  # Source IP address
    network_header += b'\x0a\x00\x00\x01'  # Destination IP address

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

def main():
    # Set up server socket
    SERVER = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    SERVER.bind(ADDR)
    SERVER.listen(1)

    print('Server is listening for incoming connections')

    # Accept a client connection
    client_socket, address = SERVER.accept()
    client_socket.settimeout(5)

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

    # Set receive window size (in bytes)
    receive_window_size = 1460
    rwnd=5

    # Open file to be sent
    file = open(FILENAME, 'rb')
    timeout=0.4
    # Send packet with transport and network layer headers
```

```
/mnt/g/CO/J/L/N/LAB 5/Task 1                    at 08:26:12 PM
python3 Server.py
Server is listening for incoming connections
```

```
1460
192720 192720 15 50
Sent packet 192720 currsent 7
hi
rcvd 0
ack send
Done
19.080883741378784
/mnt/g/CO/J/L/N/LA/Task 1
```

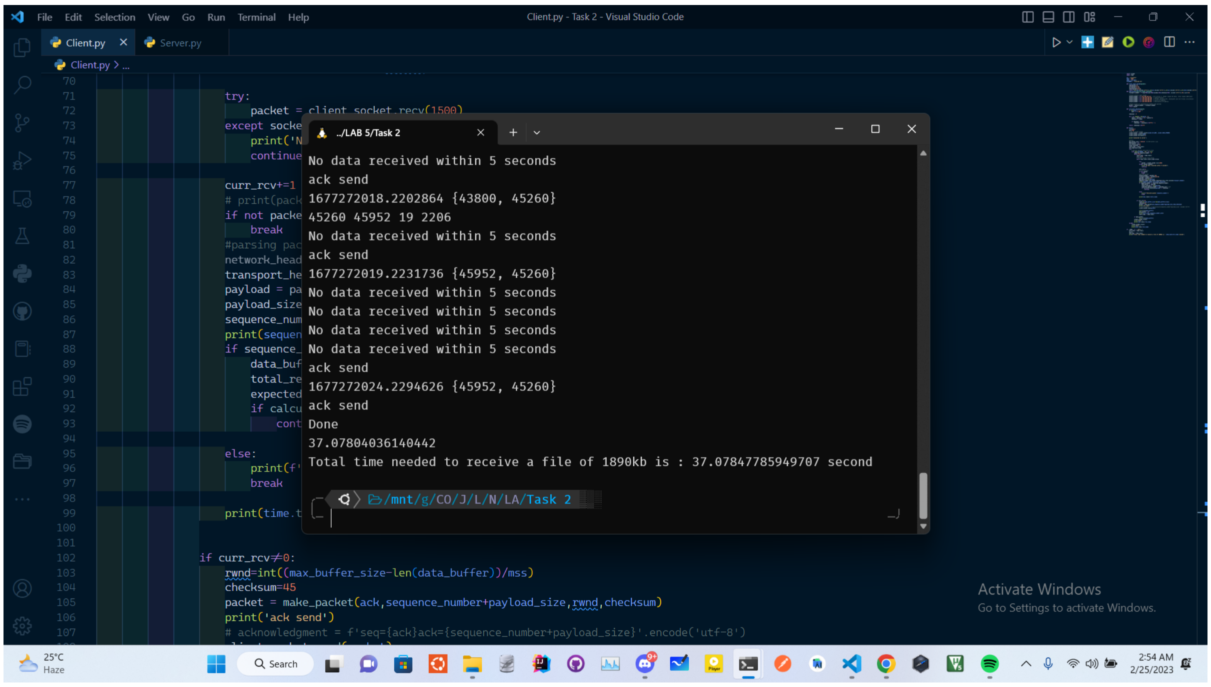


```python
    seq = int(seq)
    ack = int(ack)
    window = int(window)
    checksum = int(checksum)
    transport_header = f'{seq:06d}{ack:06d}{window:04d}{checksum:04d}'.

    # Build network layer header
    network_header = b'\x45\x00\x05\xdc'  # IP version 4, header length
    network_header += b'\x00\x00\x00\x00'  # Identification
    network_header += b'\x40\x06\x00\x00'  # TTL=64, protocol=TCP, chec
    network_header += b'\x0a\x00\x00\x02'  # Source IP address
    network_header += b'\x0a\x00\x00\x01'  # Destination IP address

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

def main():
    # Set up server socket
    SERVER = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    SERVER.bind(ADDR)
    SERVER.listen(1)

    print('Server is listening for incoming connections')

    # Accept a client connection
    client_socket, address = SERVER.accept()
    client_socket.settimeout(5)

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

    # Set receive window size (in bytes)
    receive_window_size = 1460
    rwnd=5

    # Open file to be sent
    file = open(FILENAME, 'rb')
    timeout=0.4
    # Send packet with transport and network layer headers
```

```
hi
1460
192720 192720 15 50
Sent packet 192720 currsent 7
hi
795
193515 193515 15 50
Sent packet 193515 currsent 8
hi
0
193515 193515 12 45
Received acknowledgment for packet 193515
hi
0
No acknowledgment received within 5 seconds
Done
Total time needed to send a file of 189kb  is : 45.79328393936157 second

/mnt/g/CO/J/L/N/LA/Task 1
```

Figure 4: Server Side View

5.2 Implement TCP Congestion Control

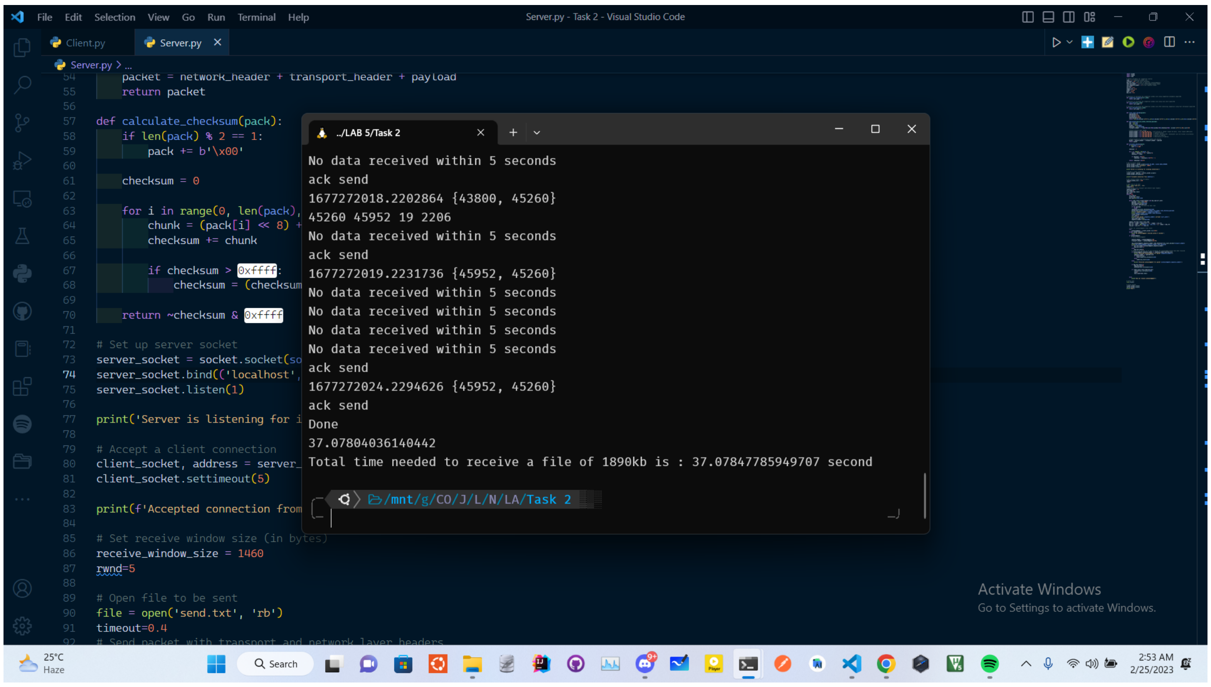


```
            try:
                packet = client_socket.recv(1500)
            except socke
                print('N
                continue

            curr_rcv+=1
            # print(pack
            if not packe
                break
            #parsing pac
            network_head
            transport_he
            payload = pa
            payload_size
            sequence_num
            print(sequen
            if sequence_
                data_buf
                total_re
                expected
                if calcu
                    cont

            else:
                print(f'
                break

            print(time.t


        if curr_rcv≠0:
            rwnd=int((max_buffer_size-len(data_buffer))/mss)
            checksum=45
            packet = make_packet(ack,sequence_number+payload_size,rwnd,checksum)
            print('ack send')
            # acknowledgment = f'seq={ack}ack={sequence_number+payload_size}'.encode('utf-8')
```

```
No data received within 5 seconds
ack send
1677272018.2202864 {43800, 45260}
45260 45952 19 2206
No data received within 5 seconds
ack send
1677272019.2231736 {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
ack send
1677272024.2294626 {45952, 45260}
ack send
Done
37.07804036140442
Total time needed to receive a file of 1890kb is : 37.07847785949707 second

/mnt/g/CO/J/L/N/LA/Task 2
```

File Edit Selection View Go Run Terminal Help
Client.py - Task 2 - Visual Studio Code
Client.py
Server.py
../LAB 5/Task 2
Activate Windows
Go to Settings to activate Windows.
25°C
Haze
Search
2:54 AM
2/25/2023

Figure 5: Client Side View



```
    packet = network_header + transport_header + payload
    return packet

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

    checksum = 0

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

        if checksum > 0xffff:
            checksum = (checksum

    return ~checksum & 0xffff

# Set up server socket
server_socket = socket.socket(so
server_socket.bind(('localhost',
server_socket.listen(1)

print('Server is listening for i

# Accept a client connection
client_socket, address = server_
client_socket.settimeout(5)

print(f'Accepted connection from

# Set receive window size (in bytes)
receive_window_size = 1460
rwnd=5

# Open file to be sent
file = open('send.txt', 'rb')
timeout=0.4
# Send packet with transport and network layer headers
```

```
No data received within 5 seconds
ack send
1677272018.2202864 {43800, 45260}
45260 45952 19 2206
No data received within 5 seconds
ack send
1677272019.2231736 {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
ack send
1677272024.2294626 {45952, 45260}
ack send
Done
37.07804036140442
Total time needed to receive a file of 1890kb is : 37.07847785949707 second

/mnt/g/CO/J/L/N/LA/Task 2
```

File Edit Selection View Go Run Terminal Help
Server.py - Task 2 - Visual Studio Code
Client.py
Server.py
../LAB 5/Task 2
Activate Windows
Go to Settings to activate Windows.
25°C
Haze
Search
2:53 AM
2/25/2023

Figure 6: Server Side View

6 Analyzation

6.1 Congestion Window

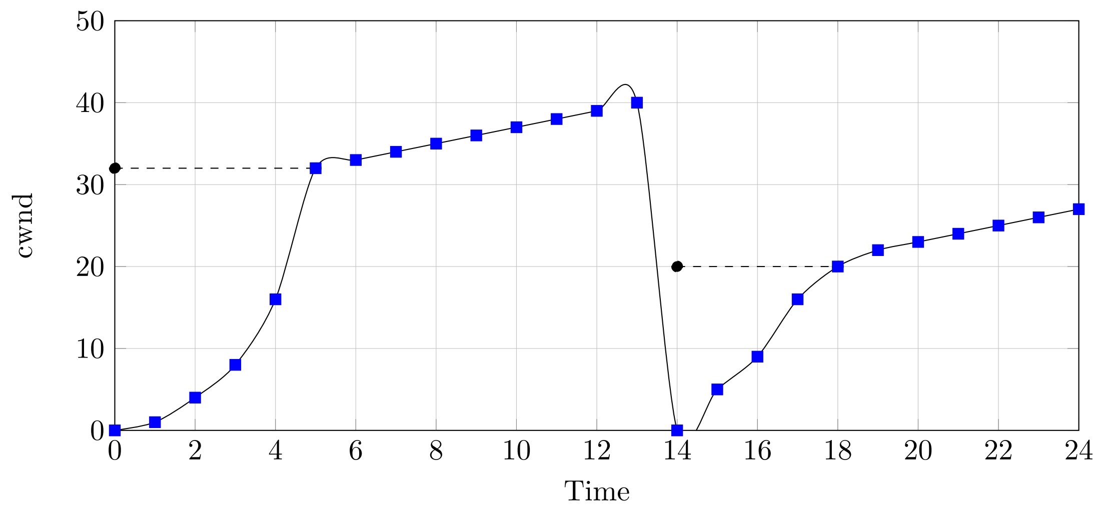


Figure 7: Congestion Window Timing Graph with the ssthresholds

6.2 Estimated_RTT & Sample_RTT & Deviation_RTT

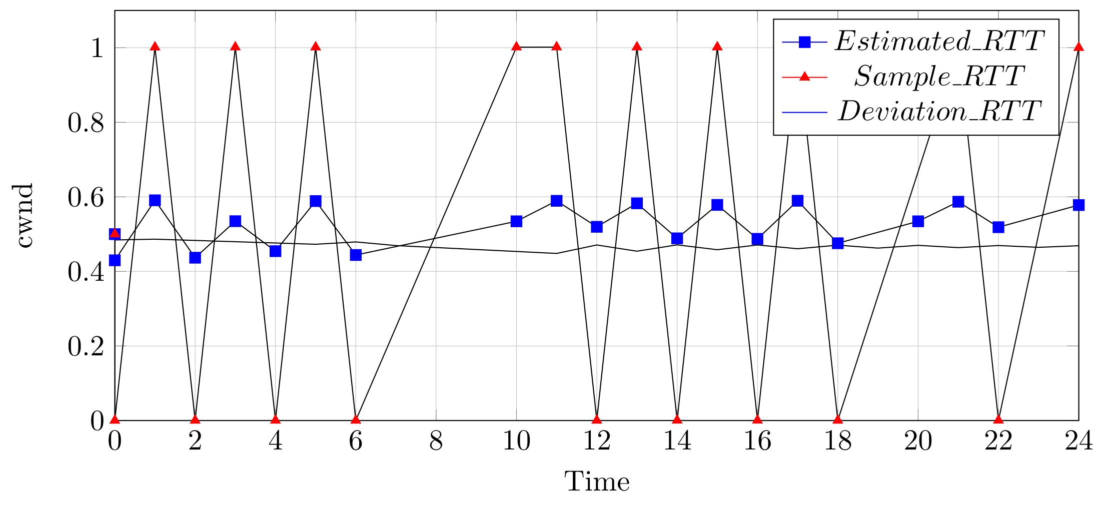
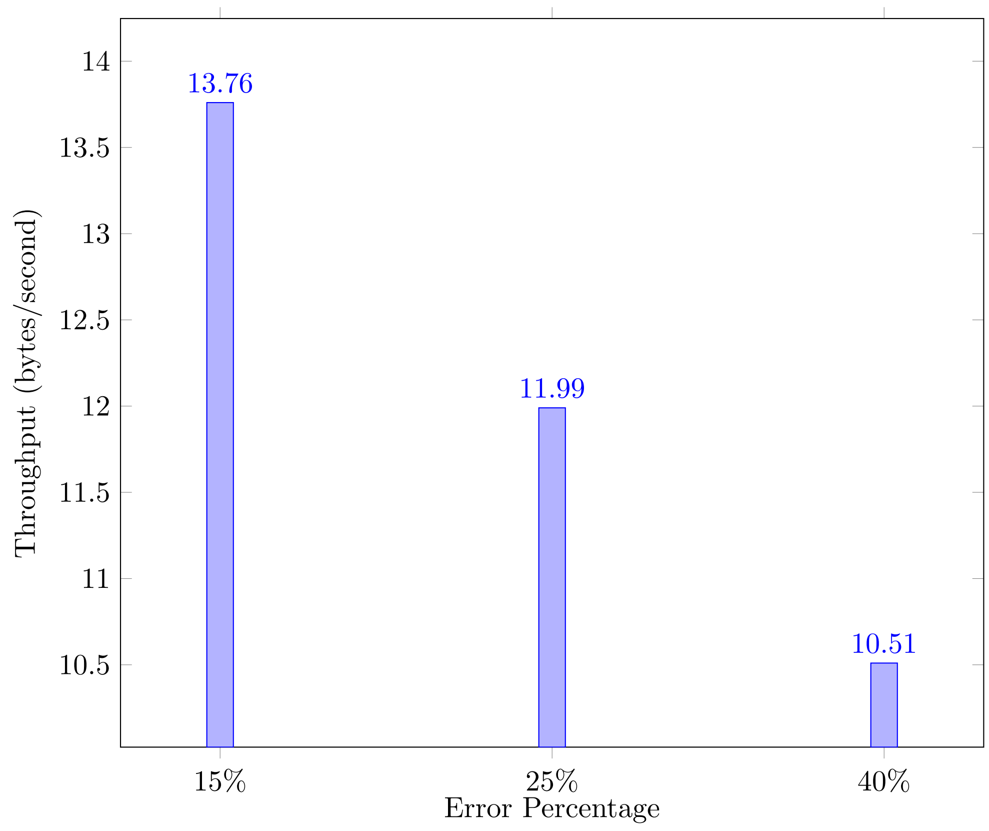
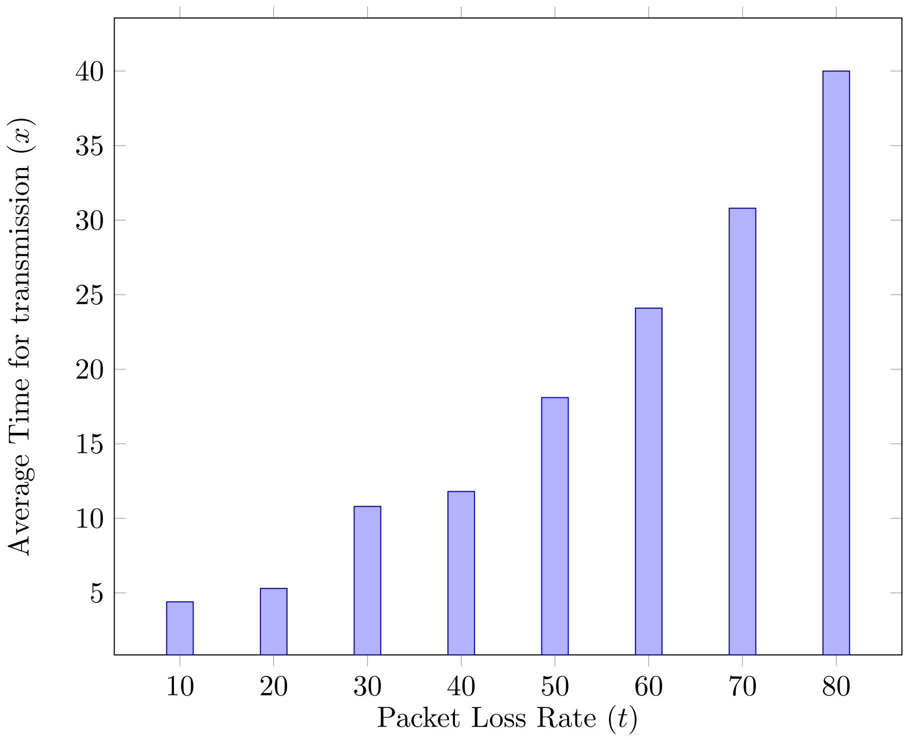


Figure 8: Estimated RTT & Sample RTT & Deviation RTT Timing

6.3 Throughput



6.4 Packet Loss



7 Experience

1. We had to investigate the effect of different values of the TCP Tahoe parameters on the performance of the algorithm.
2. We had to analyze the behavior of TCP Tahoe 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 Tahoe in scenarios with varying network topologies.
5. We had to examine the impact of packet drop rate on the performance of TCP Tahoe, and how the algorithm responds to packet loss events.
6. We had to investigate the stability and fairness of TCP Tahoe in scenarios with multiple competing flows and comparing its performance to other TCP algorithms.

References

[1] Difference between Flow Control and Congestion Control - Javatpoint. https://www.javatpoint.com/flow-control-vs-congestion-control. [Online; accessed 2023-02-24].

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