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:

Name: Muztoba Hasan Sinha

Roll No : 15

Name: Bholanath Das Niloy

Roll No : 22

Submitted On :

February 24^{th}, 2023

Submitted To :

Dr. Md. Abdur Razzaque

Md Mahmudur Rahman

Md. Ashraful Islam

Md. Fahim Arefin

Contents

1 Introduction

We are tasked with implementing flow control for tcp, then implementing congestion control using tcp tahoe. We are also tasked with adding timers everywhere so as to simulate realistic environments where there will be a lot of delay. Not only do we have to implement timers but we need to forcibly introduce error into the bit stream so that retransmission of lost/erronous packets can be checked. We also have to add the **EWMA** (Exponential Weighted Moving Average) equation to the sender side to have an estimate for the **RTT** (Round Trip Time).

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

2.1 What is TCP flow control?

TCP flow control is a mechanism used in the TCP protocol to manage the rate of data transmission between two devices over a network connection. It is designed to prevent the receiver from being overwhelmed by a flood of incoming data, and to ensure that data is transmitted smoothly and without errors.

TCP flow control works by using a sliding window protocol, where the receiver advertises the amount of data it is able to receive by sending a message containing the number of bytes of available buffer space in its receive window. The sender then limits the amount of data it sends to the receiver based on this advertised window size.

For example, if the receiver advertises a receive window size of 10,000 bytes, the sender will transmit up to 10,000 bytes of data and then wait for an acknowledgement from the receiver before sending additional data. This process is repeated until all data has been transmitted.

TCP flow control helps prevent network congestion and reduces the likelihood of packet loss or corruption. It is an essential component of the TCP protocol, ensuring that data is transmitted efficiently and reliably across a network connection.

2.2 What is TCP Tahoe?

TCP Tahoe is one of the early versions of the TCP congestion control algorithm, which is a crucial component of the TCP/IP protocol used for data transmission over the Internet. It was developed by Van Jacobson and his team at Lawrence Berkeley National Laboratory in the late 1980s.

TCP Tahoe uses a simple approach to congestion control, which involves reducing the sending rate of data packets when network congestion is detected. When a packet is lost or delayed, TCP Tahoe assumes that congestion has occurred and reduces the sending rate by cutting the congestion window size in half. This algorithm then slowly increases the sending rate again until congestion is detected, at which point it reduces the sending rate again.

TCP Tahoe's congestion control mechanism is designed to be conservative, which means that it responds quickly to network congestion, but it may not fully utilize the available network bandwidth. It was widely used in the early days of the Internet and formed the basis for later TCP congestion control algorithms, such as TCP Reno and TCP New Reno.

2.3 What is Congestion Control?

Congestion control is a mechanism used in networking to manage the flow of data across a network and prevent network congestion. Network congestion occurs when there is more traffic on the network than the network can handle, which can lead to packet loss, delays, and reduced network performance. Congestion control algorithms aim to prevent this by controlling the rate of data transmission to avoid overloading the network.

In TCP (Transmission Control Protocol), congestion control is implemented as part of the protocol itself. TCP uses a variety of congestion control algorithms to manage the flow of data and prevent network congestion. These algorithms work by dynamically adjusting the rate of data transmission based on network conditions, such as delays and packet loss.

There are several congestion control algorithms used in TCP, such as TCP Tahoe, TCP Reno, TCP New Reno, TCP Vegas, and TCP BIC, each with its own approach to managing network congestion. Some algorithms, such as TCP Reno, use a "slow start" approach, where the rate of data transmission is gradually increased until congestion is detected, at which point the rate is reduced. Other algorithms, such as TCP Vegas, use a "just-in-time" approach, where the rate of data transmission is adjusted based on real-time measurements of network delay.

Overall, congestion control is an essential mechanism in networking to ensure efficient and reliable data transfer. It helps prevent network congestion, reduce packet loss and delays, and optimize network performance, ensuring that the network can handle traffic efficiently and effectively.

2.4 How is TCP Tahoe Implemented?

1. **Initialization:** When the TCP connection is established, the sender initializes its congestion window to one Maximum Segment Size (MSS) and sends one segment to the receiver.

2. **Slow Start:** The sender increases its congestion window size by one MSS for each acknowledged segment it receives from the receiver. This phase is called "slow start" because the congestion window size grows exponentially with each acknowledgment, leading to a rapid increase in data transmission rate.

3. **Congestion Detection:** If a segment is lost or delayed, the receiver sends a duplicate acknowledgment to indicate that it has not received the missing data. If the sender receives three duplicate acknowledgments for the same segment, it assumes that the segment has been lost and reduces its congestion window size to half of its current value. This is done to prevent further congestion.

4. **Congestion Avoidance:** Once the sender's congestion window reaches the size it had before congestion was detected, it enters the "congestion avoidance" phase. In this phase, the sender gradually increases its congestion window size by one MSS for every full round-trip time of the network, which is the time it takes for a segment to be sent from the sender to the receiver and for an acknowledgment to be received. This phase is designed to slow down the growth of the congestion window and prevent further congestion.

5. **Retransmission:** If the sender does not receive an acknowledgment from the receiver within a certain period of time, it assumes that the segment has been lost and retransmits the unacknowledged segment. The sender also reduces its congestion window size to one MSS to prevent further congestion.

6. **Repeat:** The sender repeats steps 2 to 5 for the duration of the data transfer, dynamically adjusting its congestion window size in response to network conditions.

2.5 The EWMA equation

2.5.1 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.

$$EMA(t) = \alpha * P(t) + (1 - \alpha) * EMA(t - 1)$$

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.

The smoothing factor, α, determines the weight of the current observation relative to the previous observations. A smaller α value will give more weight to the previous observations, while a larger α value will give more weight to the current observation.

The EWMA equation can be used to calculate a moving average of network measurements that has an exponentially decreasing weight for the past observations. This means that recent network measurements are given more weight than older network measurements, and the weight decreases exponentially as the measurements get older.

In networking, the EWMA equation is used in various applications, such as network performance monitoring and anomaly detection. For example, the EWMA equation can be used to estimate the average delay or traffic on a network link, and detect anomalies or sudden changes in the network behavior.

2.5.2 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:

$$Estimated\_RTT = \alpha * Sample\_RTT + (1 - \alpha) * Estimated\_RTT_{prev}$$

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
$Sample\_RTT$ is the RTT measured for a single packet.

$Estimated\_RTT_{prev}$ is the estimated RTT at the previous time period. The smoothing factor, α, determines the weight of the current observation relative to the previous observations. A smaller α value will give more weight to the previous RTT estimates, while a larger α value will give more weight to the current RTT estimate.

The EWMA equation can be used to estimate the RTT of a network connection over time. For each packet sent, the sender measures the RTT for that packet and updates the estimated RTT using the EWMA equation. This helps to smooth out variations in the RTT due to network congestion or other factors, and provides a more accurate estimate of the current RTT.

Overall, using the EWMA equation for estimating the RTT in networking can improve the accuracy of network performance measurements and help in identifying and diagnosing network problems.

2.5.3 Calculating the Dev_RTT

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

$$Deviation\_RTT = \alpha * |Sample\_RTT - Estimated\_RTT| + (1 - \alpha) * Deviation\_RTT_{prev}$$

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, this is usually 0.25.
$Sample\_RTT$ is the RTT measured for a single packet.

$Estimated\_RTT$ is the estimated RTT at the current time. $Deviation, RTT_{prev}$ is the estimated deviation of the RTT at the previous time period. The smoothing factor, α, determines the weight of the current observation relative to the previous observations. A smaller α value will give more weight to the previous deviation estimates, while a larger α value will give more weight to the current deviation estimate.

The EWMA equation can be used to estimate the deviation of the RTT of a network connection over time. For each packet sent, the sender measures the RTT for that packet and updates the estimated deviation using the EWMA equation. This helps to smooth out variations in the deviation due to network congestion or other factors, and provides a more accurate estimate of the current deviation.

Overall, using the EWMA equation for estimating the deviation of the RTT in networking can improve the accuracy of network performance measurements and help in identifying and diagnosing network problems.

2.6 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:

$$RTO = Estimated\_RTT + 4 * Deviation\_RTT$$

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.

The equation includes a factor of 4 multiplied by the deviation of the RTT to provide a buffer that takes into account the potential variation in the RTT that could occur between the time that the sender measures the RTT and the time that the retransmission timeout actually expires. This buffer helps to reduce the risk of premature retransmission due to small, temporary variations in the RTT.

Overall, incorporating both the estimated RTT and deviation of the RTT together in the calculation of the RTO can help to improve the reliability and efficiency of data transmission over a network.

3 Implementation

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

3.1 Preliminary work

We have declared a custom header of size 14 bytes which contain

1. Sequence number of 4 bytes
2. Acknowledgement number of 4 bytes
3. Ack bit represented by one byte

4. sf bit represented by one byte
5. Receive window size information of 2 bytes
6. Checksum of data 2 bytes

Parsing and packaging of these data are done by the *toHeader* and *fromHeader* function. We have explicitly declared a sender and receiver using python TCP sockets and we simulated TCP on it. We have a fixed MSS (Maximum Segment Size) among both the hosts so that the packet reading stays consistent. This is the abstraction of the TCP packet structure that we have used to simulate TCP.

3.2 Flow Control

The idea of flow control is that the sender should not send so much data that it overwhelms the receiver. It is fully dependent on the behaviour of the network. In an actual web application the TCP buffer is cleared by the application layer randomly when needed thus this capacity to accept data changes wildly. Here to abstract this we have implemented a scheme for the cumulative ack. The reciever sends an ack if one these two events occur

1. If we have filled the buffer.
2. Timer on the receiver to runs out.

In both cases we have sent an ack, acknowledging the latest in order data. Along with this ack we calculate the empty space in the buffer. this was calculated by subtracting the received data in the buffer from the received buffer size. This value is then used to restrict the flow of data dynamically on the sender size. The sender window is restricted to the maximum the receiver can handle.

3.3 TCP Congestion Control

TCP congestion control controls the flow of data considering the bottleneck as any link in the network except the receiver (handled by the flow control). While the sender and receiver might be capable somewhere along the network chain there may arise a weak link. We need to match our sending rate to account for this or unnecessary congestion or even dropped packets might be more frequent along the route. We do this by starting transmission cautiously by sending 1 packet per window and quickly ramping up by increasing the window size as mentioned in the specification document, this is the **slow start** phase. When this window exceeds the *ssthresh* we approach cautiously again, according to the specified document thus simulating the **congestion avoidance** phase. In case of a dropped/unacked packet or an out of order packet (indicated by a timeout and triple acknowledgement respectively), we halve the *ssthreshold* so that we increase our rate more cautiously next time and set the window to zero so that we don't hit another packet loss as fast.

3.4 EWMA: Estimated Weighted Mean Average

We have strictly followed the EWMA Standard broadly described Section 2.5. We have used the EWMA to increase throughput by reducing the sender timeout interval.

3.5 Checksum

We have implemented a simple 16bit checksum function *calculate_checksum* which given a bytestream returns the 16bit checksum. This is calculated by the sender and sent via the header. The receiver after getting the packet calls the same function on the payload and compares the checksum. In case of mismatch the receiver discards the packet. The missing packet is handled by our error handling subroutines.

3.6 Error handling

In case of a packet drop the receiver will either get an out of order packet which triggers sending an ack packet. The sender sees that ack corrects the segment number and starts re-sending from unordered packet. In case if the packet is not an end packet this will trigger a triple ack and sender will respond immediately. 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.

4 Graphs

4.1 Congestion Window

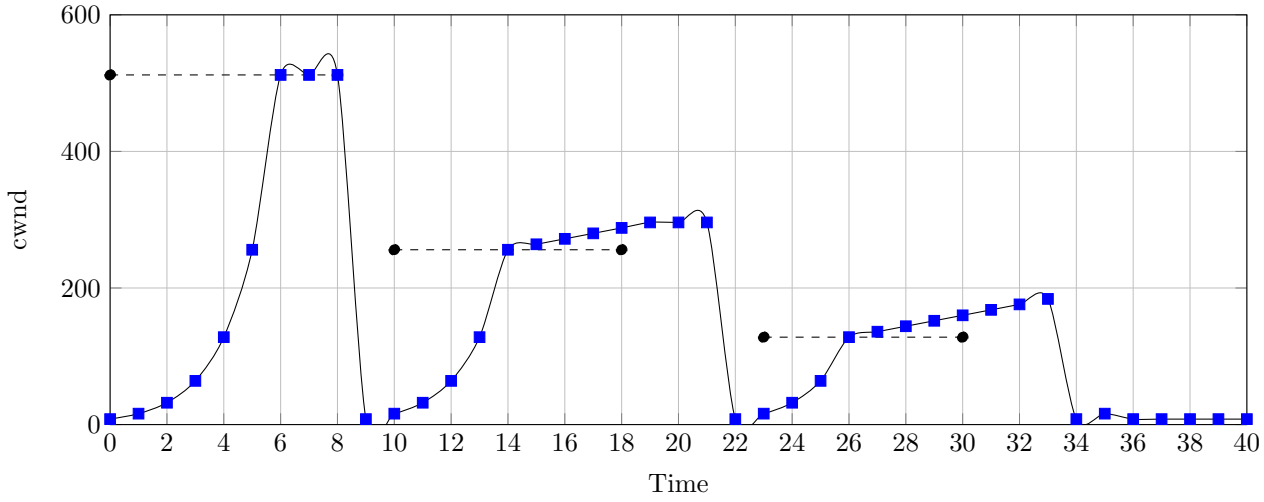


Figure 1: Congestion Window Timing Graph with the ssthresholds

Since we are sending cumulative acks, we assuming the case where $cwnd \leq ssthresh$ we are doubling the *cwnd* value as opposed to what was written in the documentation for the lab 5. But this was what sir wanted for us to do in, as was said in the zoom meeting before the start of the lab. Due to conflicting instructions from sir and what was written in the doc, we opted to listen to the instructions sir gave in the zoom meeting.

4.2 Estimated_RTT & Sample_RTT & Deviation_RTT

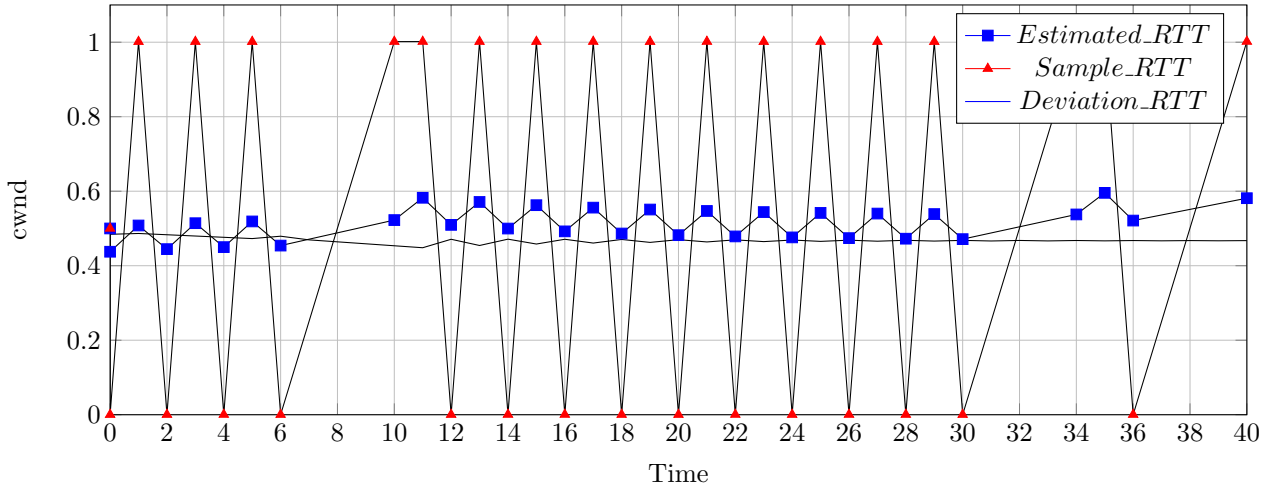
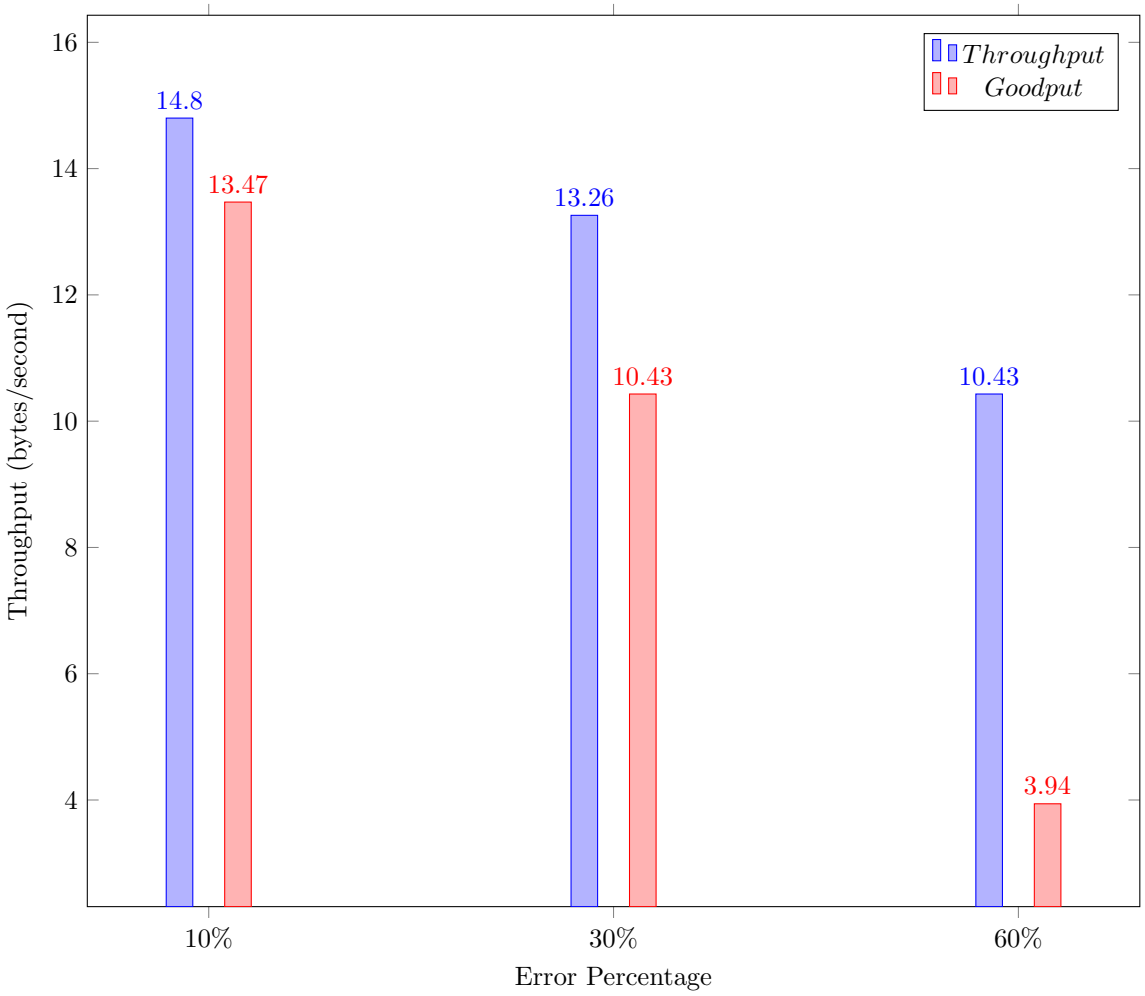


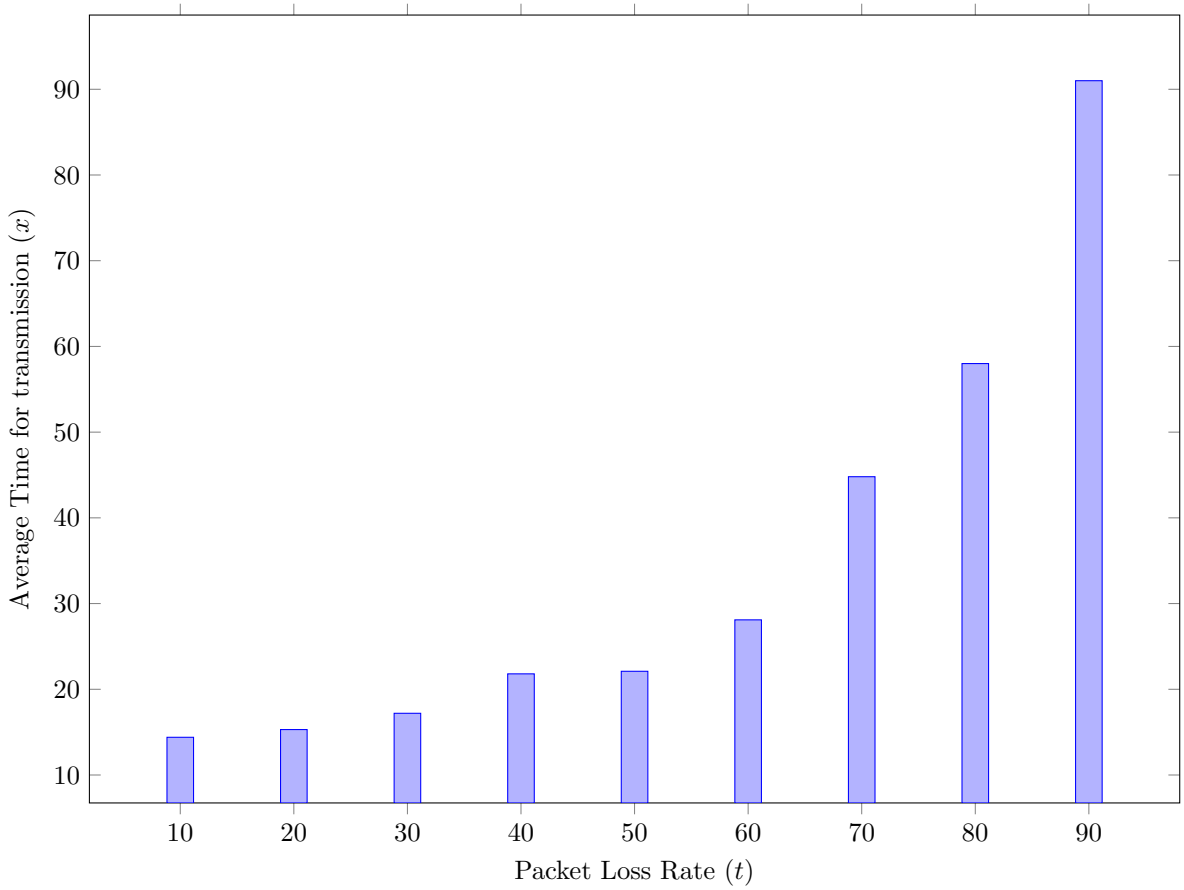
Figure 2: Estimated RTT & Sample RTT & Deviation RTT Timing

All of these values were measured with 10% loss in mind.

4.3 Throughput



4.4 Packet Loss



5 Screenshots of Working Code

In this short example we have consolidated all parts of the experiment. Here in the sender side we have printed the changes to the *cwnd*, sender_window and ssthreshold this shows that the part 1 and 2 that is the flow and congestion control are working as intended. Again we have printed the timing information that is *DevRTT, EstimatedRTT and SampleRTT* fluctuations in which show that it's working as intended. Lastly we have included the total time taken as well as the data send to the receiver, showing that the transmission was successful.

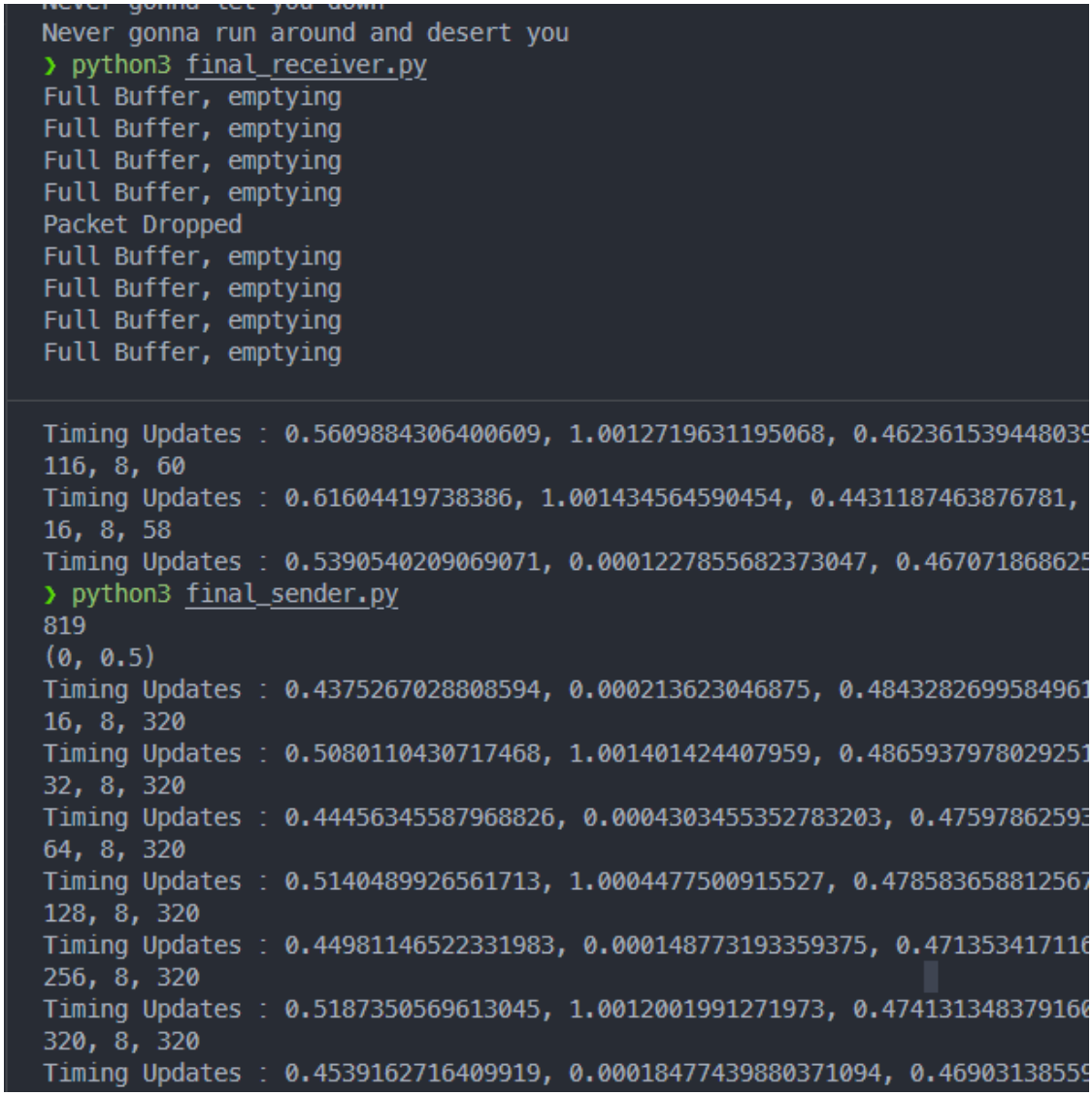


Figure 3: Running the program

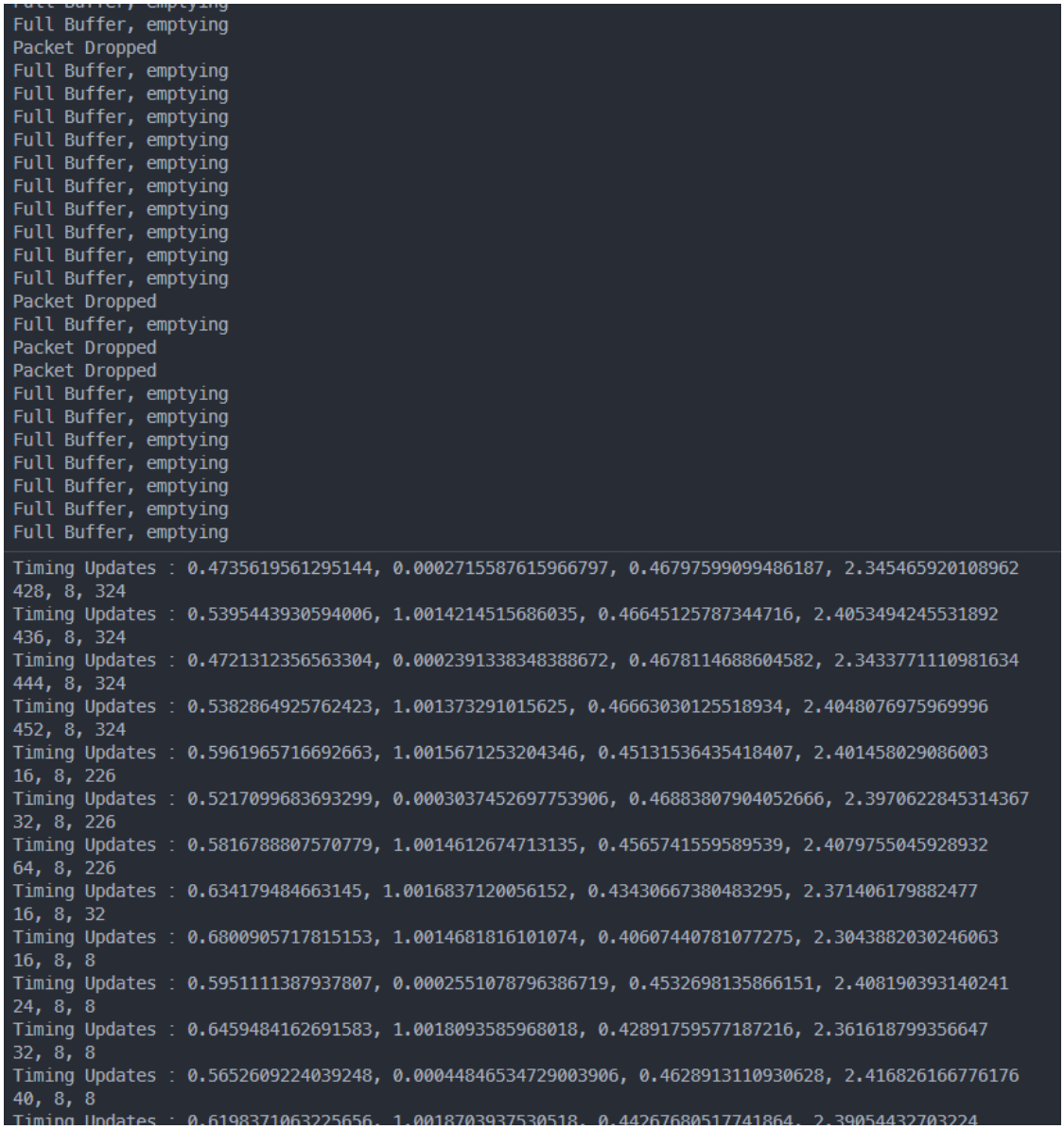


Figure 4: Showing Dropped Packets

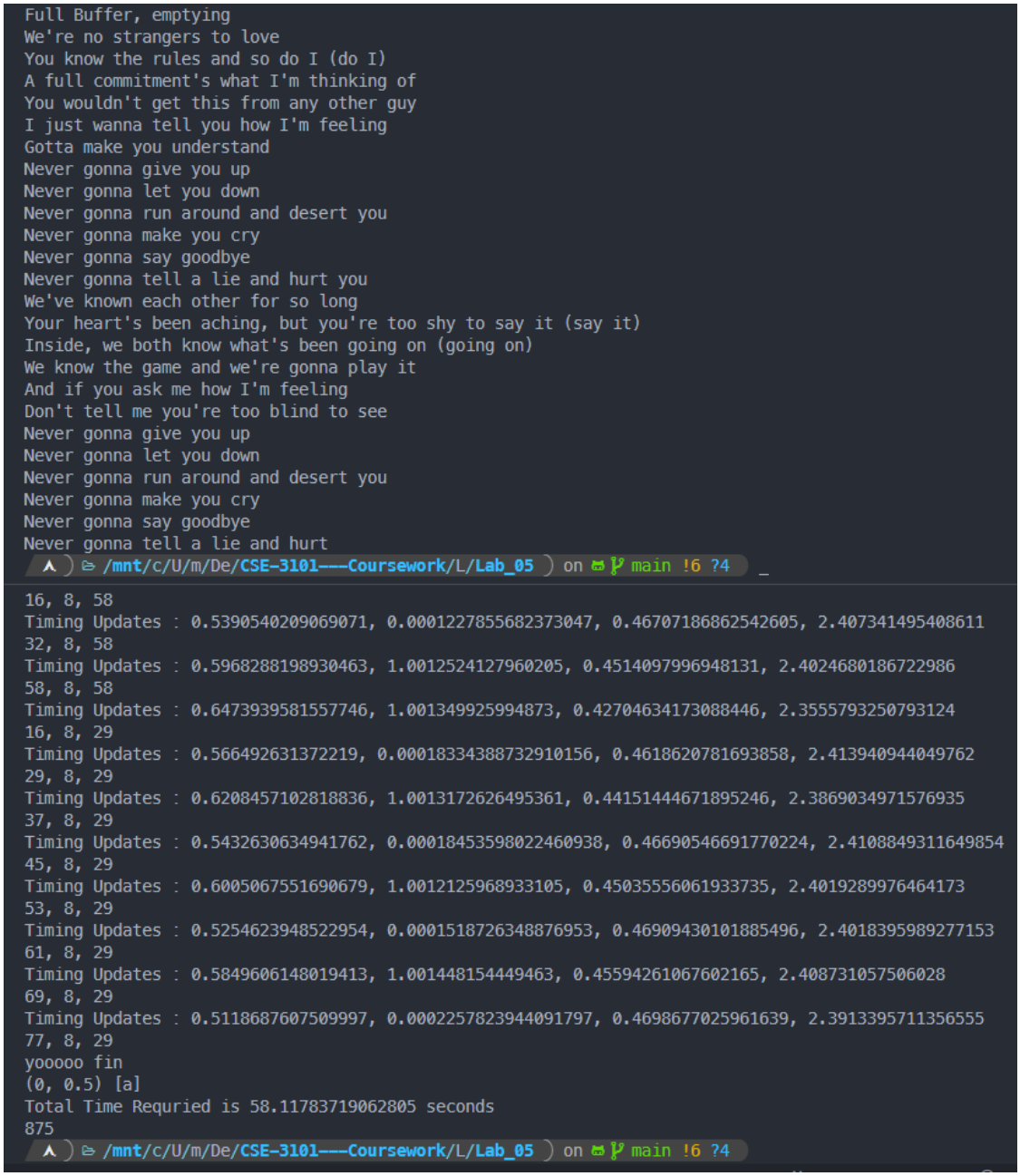


Figure 5: Final Output

References

[1] Tobías Chavarría. Exponentially weighted average - MLearning.ai - Medium. *MLearning.ai*, apr 9 2022. [Online; accessed 2023-02-24].

[2] Subham Datta. Flow control vs. congestion control in TCP. https://www.baeldung.com/cs/tcp-flow-control-vs-congestion-control, oct 24 2021. [Online; accessed 2023-02-24].

[3] fengkeyleaf. Algorithm/Implementing TCP with UDP.md at main · fengkeyleaf/Algorithm. https://github.com/fengkeyleaf/Algorithm/blob/main/Java/CSCI651/proj3/documentation/Implementing [Online; accessed 2023-02-24].

[4] James F. Kurose and Keith W. Ross. *Computer networking: A top-down approach*. Addison-Wesley Longman, 2013. [Online; accessed 2023-02-24].

[5] Mansi. Congestion control. https://www.scaler.com/topics/computer-network/tcp-congestion-control/, sep 12 2022. [Online; accessed 2023-02-24].