University of Dhaka

Department of Computer Science and Engineering

CSE-3111 : Computer Networking Lab

Lab Experiment 6 : TCP Congestion Control with TCP Reno)

Submitted By:

Name: Muztoba Hasan Sinha

Roll No : 15

Name: Bholanath Das Niloy

Roll No : 22

Submitted On :

February 28^{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 Reno?

TCP Reno is a congestion control algorithm that is widely used in modern computer networks. It was introduced in 1990 as an enhancement to the original TCP congestion control algorithm, TCP Tahoe.

TCP Reno is designed to handle network congestion by monitoring the network and adjusting the rate at which data is transmitted based on the current state of the network. When congestion is detected, TCP Reno reduces the rate at which it sends data in order to avoid overwhelming the network.

TCP Reno uses a mechanism known as "fast recovery" to quickly recover from packet loss. When a packet is lost, TCP Reno assumes that the packet was dropped due to congestion and reduces the transmission rate. However, if a duplicate acknowledgement is received, TCP Reno enters fast recovery mode and increases the transmission rate until it reaches the previous rate before the packet loss occurred.

TCP Reno also includes a feature known as "slow start" which is used to gradually increase the transmission rate when a new connection is established or after a period of inactivity. Slow start is designed to avoid overwhelming the network with too much traffic too quickly.

Overall, TCP Reno is a highly effective congestion control algorithm that is widely used in modern computer networks. It has proven to be highly reliable and efficient, and has played a significant role in the growth and development of the Internet.

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?

The TCP Reno congestion control algorithm operates in several stages, which can be described in the following steps:

1. **Slow Start:** When a new TCP connection is established or after a period of inactivity, TCP Reno begins in the slow start phase. In this phase, the sending rate is gradually increased until a threshold called the congestion window (cwnd) is reached. The initial value of cwnd is typically set to a small value such as 1 MSS (Maximum Segment Size).

2. **Congestion Avoidance:** Once the slow start phase has completed, TCP Reno enters the congestion avoidance phase. In this phase, the cwnd is increased more slowly, typically by adding 1 MSS to the cwnd for every round-trip time (RTT) that elapses. This phase continues until congestion is detected.

3. **Fast Retransmit and Recovery:** When a packet is lost, TCP Reno detects the loss through the receipt of duplicate acknowledgements (ACKs) from the receiver. When three duplicate ACKs are received, TCP Reno enters fast retransmit mode, retransmitting the lost packet without waiting for a timeout. In addition, TCP Reno enters fast recovery mode, where the cwnd is reduced and increased by a smaller amount for each ACK received until the number of unacknowledged packets falls below a certain threshold.

4. **Timeout:** If a packet is not acknowledged within a certain timeout period, TCP Reno assumes that the packet has been lost and retransmits it. In addition, TCP Reno reduces the cwnd to the initial value of 1 MSS and enters the slow start phase.

5. **Congestion Detection:** TCP Reno uses various methods to detect congestion, including the receipt of duplicate ACKs, timeouts, and the detection of explicit congestion notification (ECN) marks from network routers. When congestion is detected, TCP Reno reduces the cwnd in order to avoid further congestion.

6. **Recovery:** After a period of congestion, TCP Reno gradually increases the cwnd again, starting with the slow start phase and then entering the congestion avoidance phase. This process continues as long as no further congestion is detected.

These are the main steps of the TCP Reno congestion control algorithm. They are designed to ensure reliable and efficient data transmission over the network while avoiding congestion and network overload.

2.5 Differences between TCP Tahoe & Reno

Feature	TCP Tahoe	TCP Reno
Fast Retransmit	No	Yes
Fast Recovery	No	Yes
Congestion Avoidance	Linear	Exponential
AIMD	Yes	Yes
Slow Start Threshold	Fixed (1 MSS)	Dynamic
Duplicate ACK Handling	Timeout	Fast Retransmit

Table 1: Differences between TCP Tahoe and Reno

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 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 following the **congestion avoidance** rule. In case of a timeout we set the congestion window to mss, halve the slow start threshold and restart from slow start.

3.3 EWMA: Estimated Weighted Mean Average

We have strictly followed the EWMA Standard provided in the previous assignment. We have used the EWMA to increase throughput by reducing the sender timeout interval.

3.4 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.

3.5 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

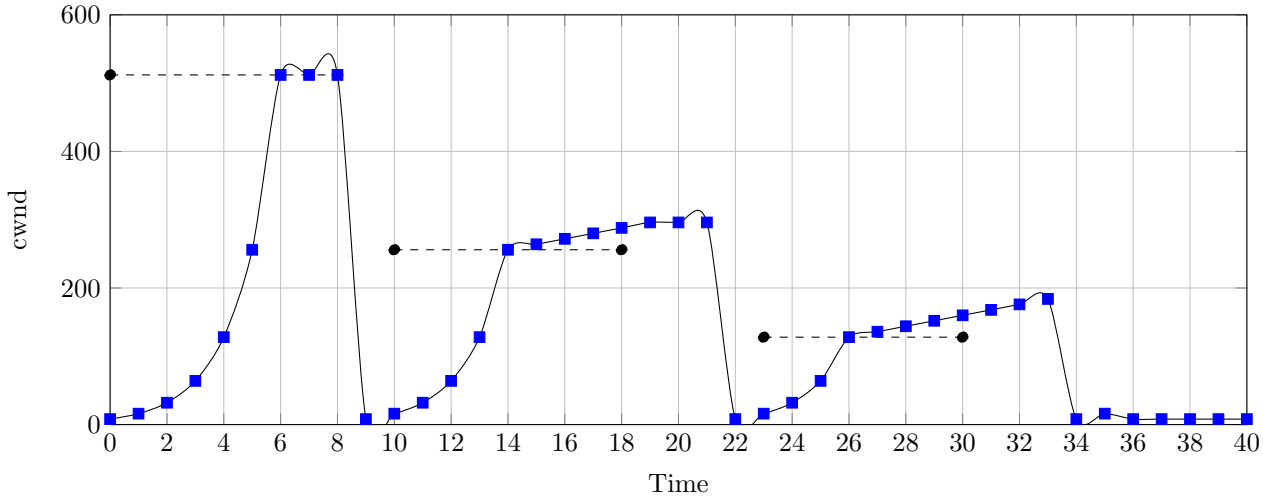


Figure 1: Congestion Window Timing Graph with the ssthresholds

This is the old results from tcp tahoe.

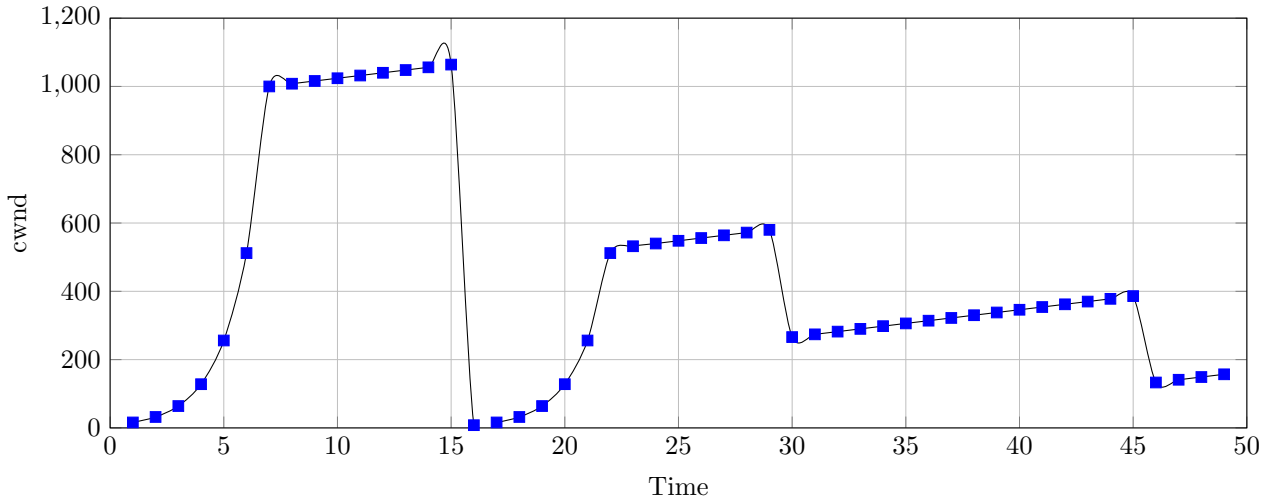


Figure 2: Congestion Window Timing Graph with the ssthresholds for TCP Reno

New results for tcp reno.

4.2 Estimated_RTT & Sample_RTT & Deviation_RTT

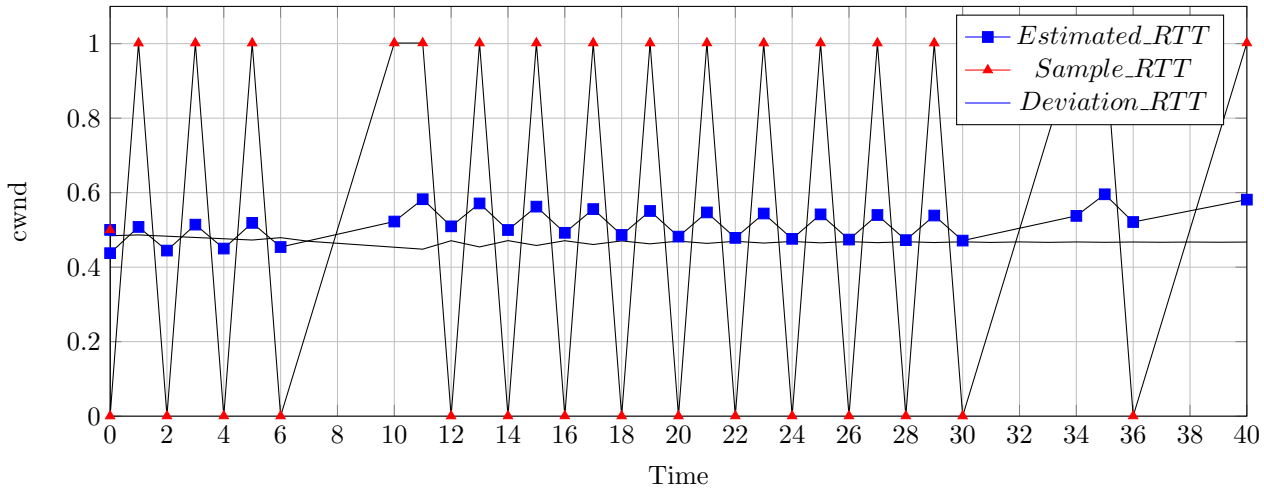
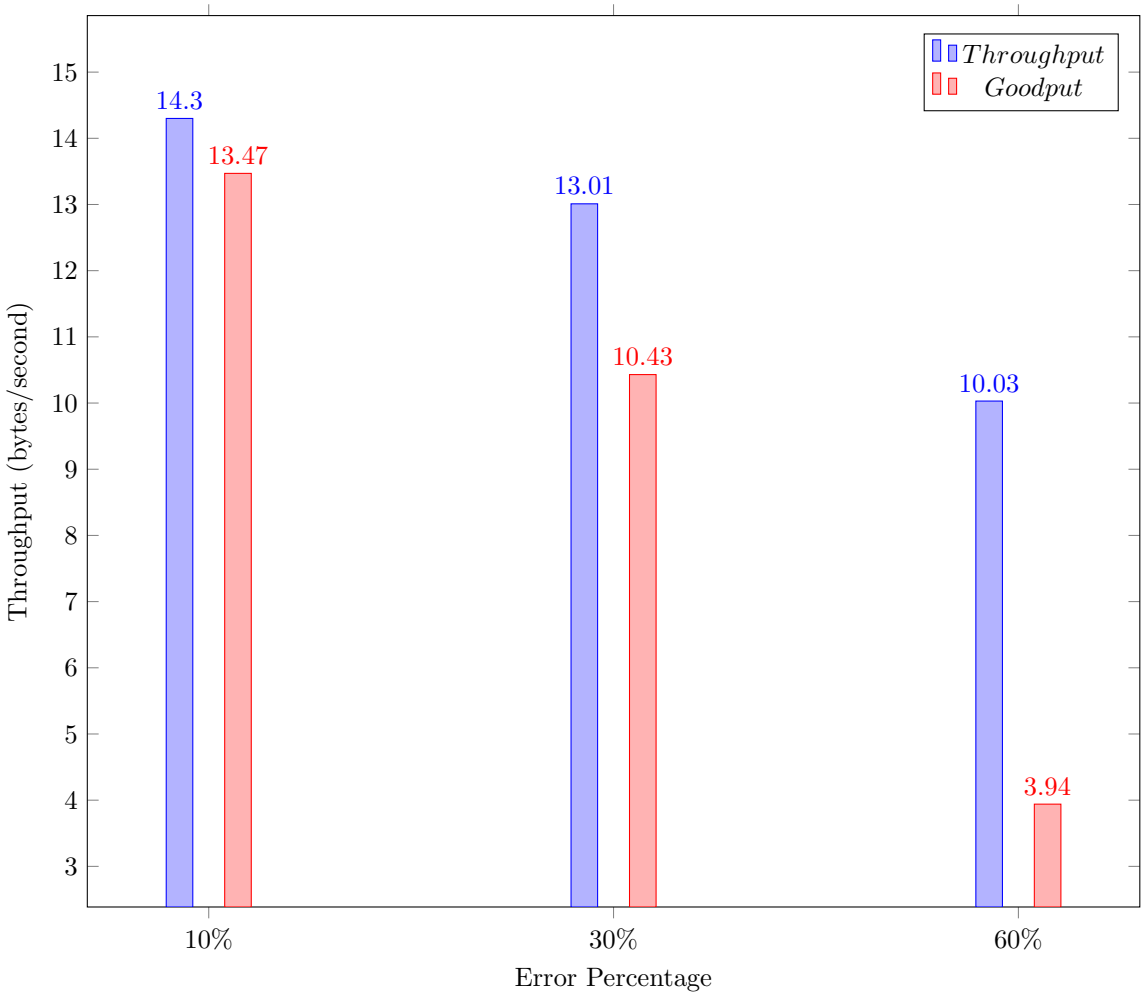


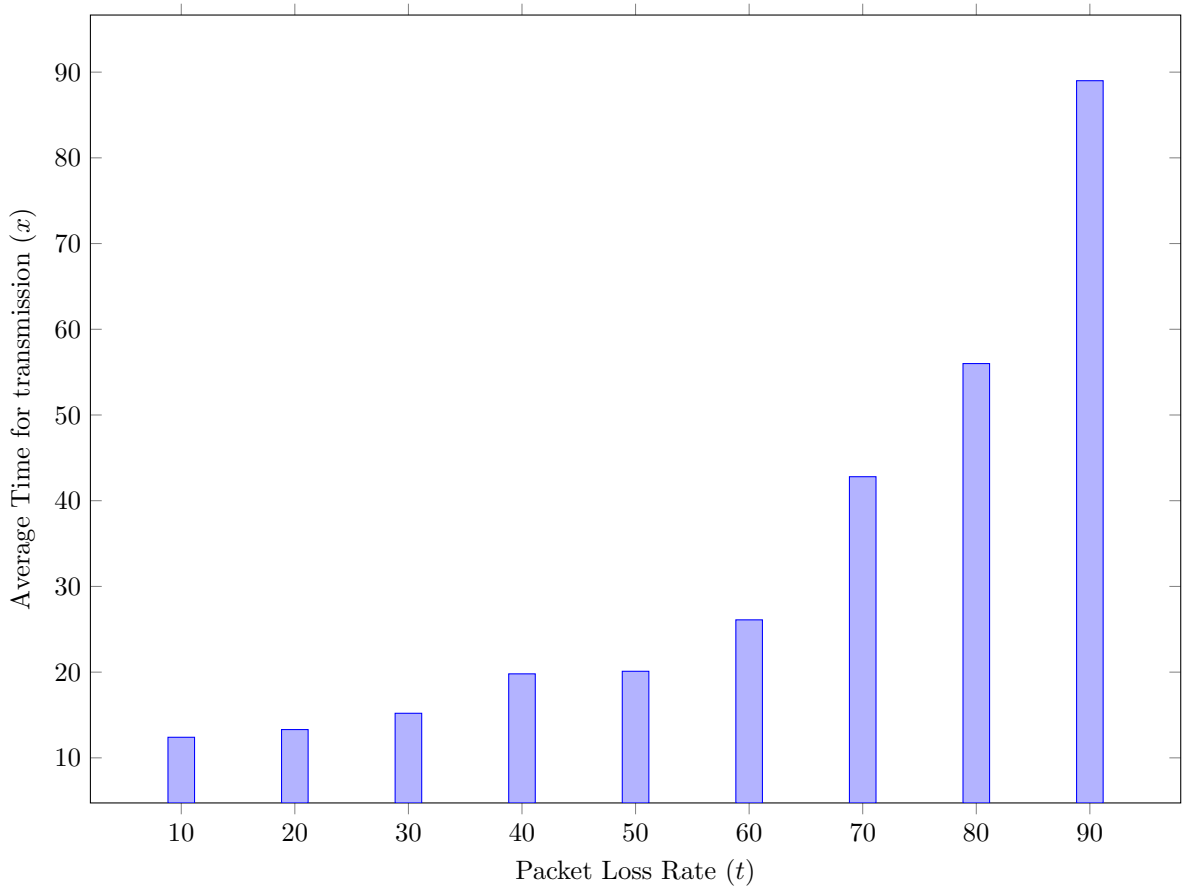
Figure 3: Estimated RTT & Sample RTT & Deviation RTT Timing

All of these values were measured with 10% loss in mind. Note that we did not see any significant changes in RTT amongst TCP Reno and TCP Tahoe. Which makes sense as the message is passing in the local machine itself

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.

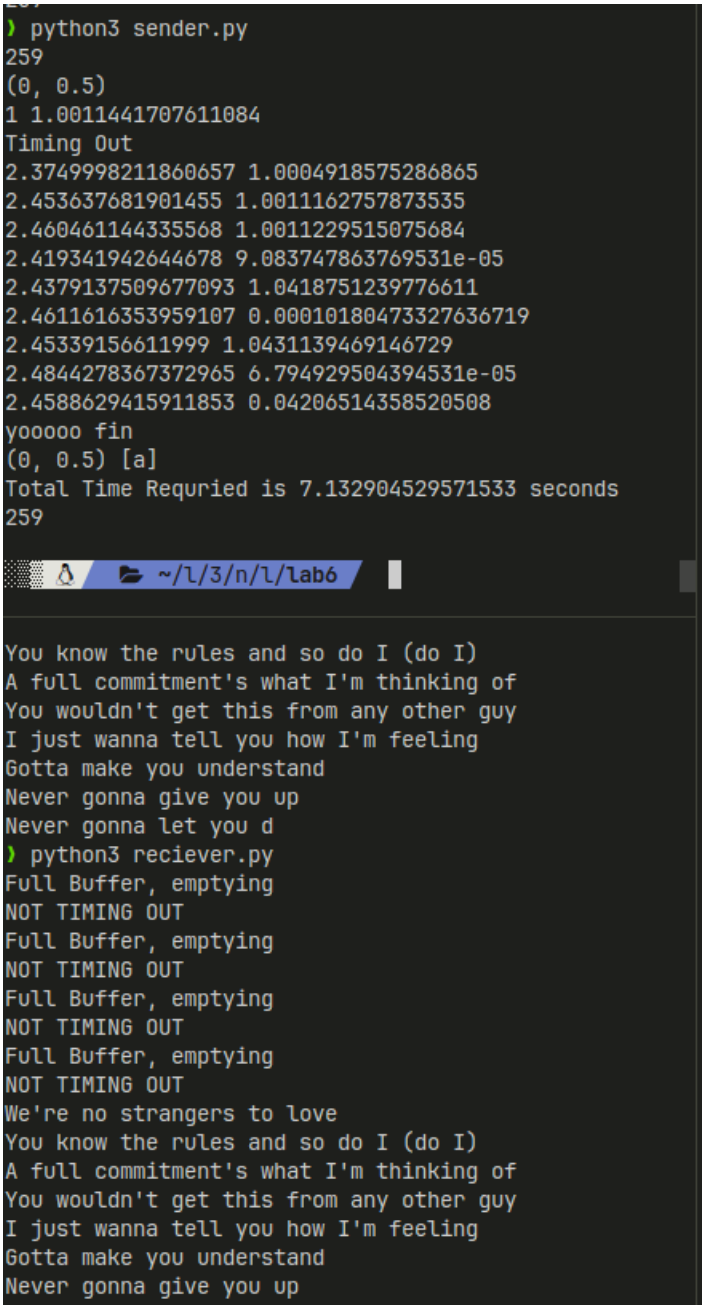


Figure 4: Running the program

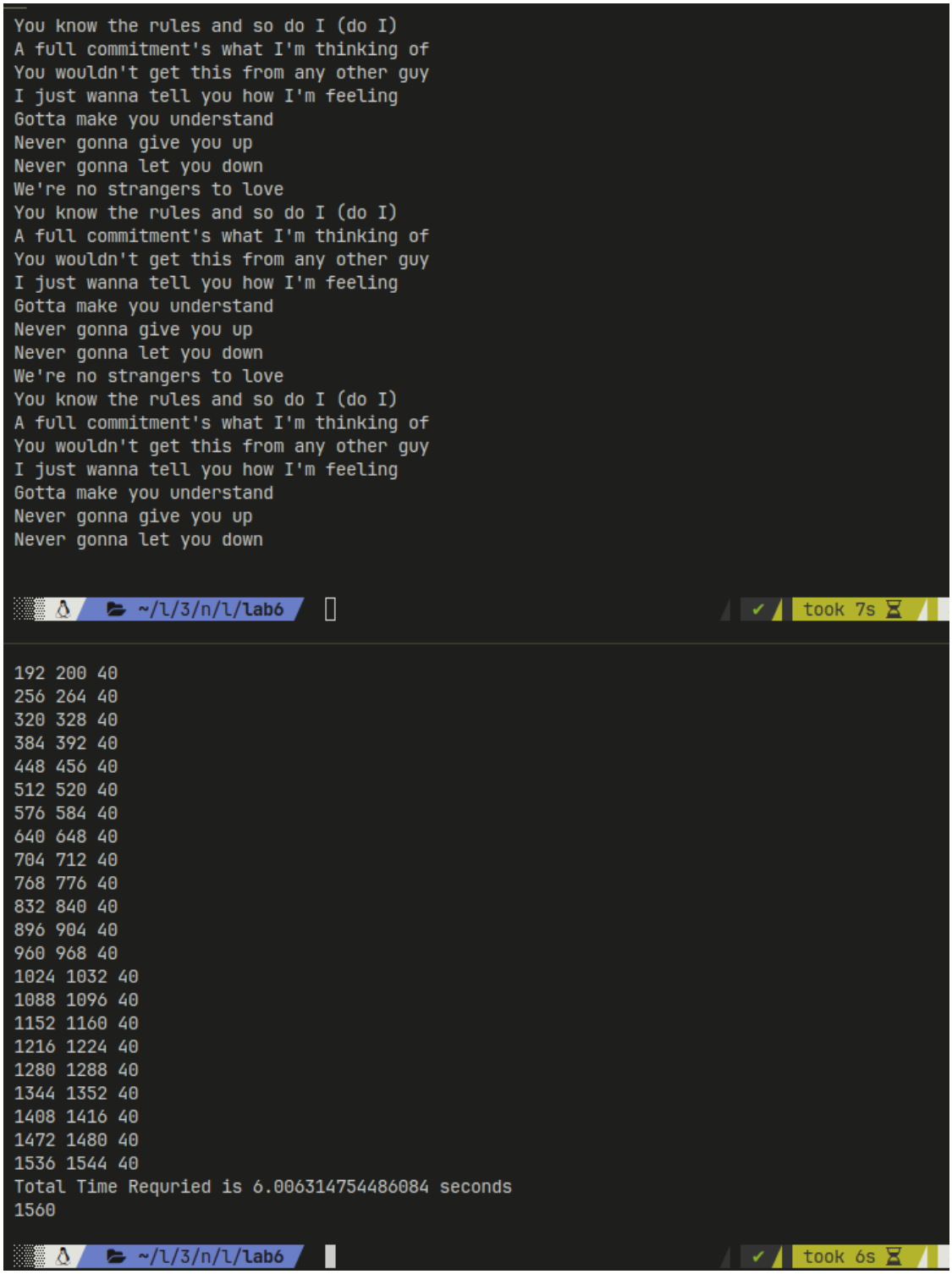


Figure 5: Reno finishing time

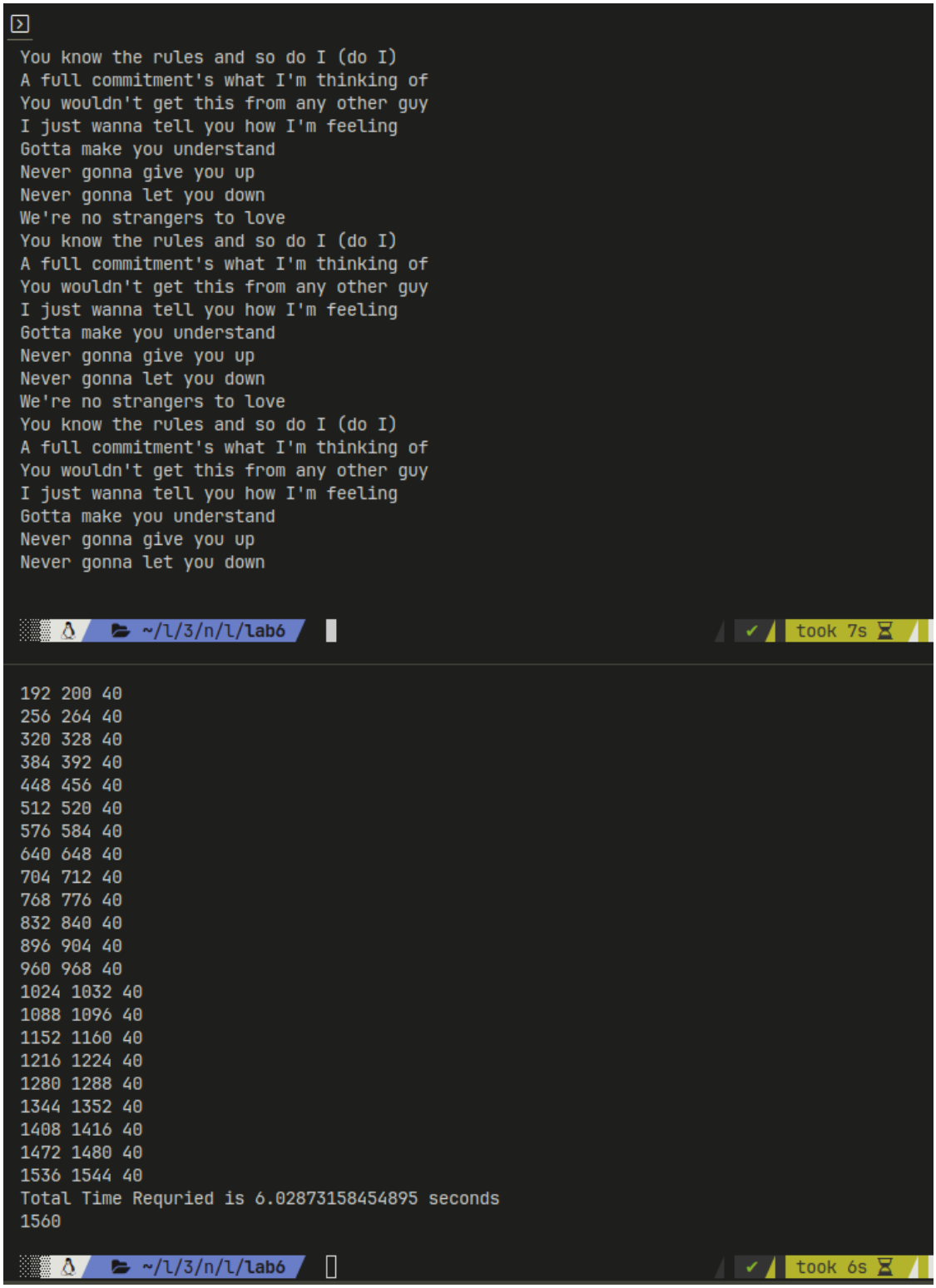


Figure 6: Tahoe being marginally slower than Reno

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].