ECS 152A: Computer Networks

Winter 2026

Project 1

Team: Julio Flores Cristerna & Kapila Mhaiskar

Stop and Wait Protocol:

a. Technical explanation

We first create a socket from the sender's side, bind it to a randomly selected port (2000), and set the timeout time for 1 second. In this implementation, the mp3 file is read and subdivided into chunks of size "MESSAGE_SIZE" and stored in an array/list "message_array".

Then, for each chunk, we create a while loop to iterate through "message_array", send packets to the receiver, and expect and acknowledge from the receiver. We create packets with a function "create_packet" similar to the function "create_acknowledgement" from the receiver code in order to create a packet consisting of the current sequence ID (starting at 0) and the payload (the chunk of the mp3 file we want to send).

After that, our sender will be expecting a message from the receiver, and if received successfully, we divide the message to get the sequence ID the receiver is expecting us to send next alongside with the acknowledgement message, implemented similarly like how the receiver receives a message and separates the data in the receiver code.

Our iterator "i" which iterates through "message_array" indicates which packet we have sent. To indicate the last sent packet was acknowledged, we check to see if the sequence ID sent by the receiver (the sequence ID the receiver expects to receive next) is greater than the sequence ID we previously sent. It is only then where we can increment our iterator "i" to indicate a successfully sent and acknowledged packet.

In the case the next expected sequence id "next_seq_id" is not greater than the current sequence id "curr_seq_id" (meaning they are equal), we check if our iterator has reached the end of "message_array" (meaning program should start terminating process) or the message from the receiver is the 'fin' message (to terminate program). If neither of those checks are true, it means we just received a duplicate packet and we do nothing with it and move forward to the next while loop iteration (without incrementing our iterator “i”). And in the case of a timeout, we just “continue” to the next iteration of the while loop as well.

We check if all packets have been sent if our iterator has reached “len(message_array)”, meaning all message chunks have been sent, to which we send an empty message to receiver with the last appropriate sequence ID, where the sender should now expect to see a final acknowledgement followed by a final message ‘fin’. If acknowledgements are not received due to timeout, we iterate through the while loop sending empty messages to the receiver until the last acknowledgement and final message are received by the sender. Otherwise, we send ‘==FINACK==’ to the receiver to close the receiver socket, we break out of the while loop and determine the metrics of the protocol.

For the throughput time, we get the start time using time.time() right after the sender socket is created and the end time right after we receive the last acknowledgement (second if statement in the while loop), and subtract the end time minus the start time. For the per packet delay average, we calculate the delay time of each packet, store it in an array and calculate the mean using statistics.mean(). Since we only want to get the start time the first time a packet is sent, we use “start_time_flag” as a way of knowing if the packet we’re sending has not been sent or if it's a packet resend. The end time is set after a successful acknowledgment (first if statement is true).

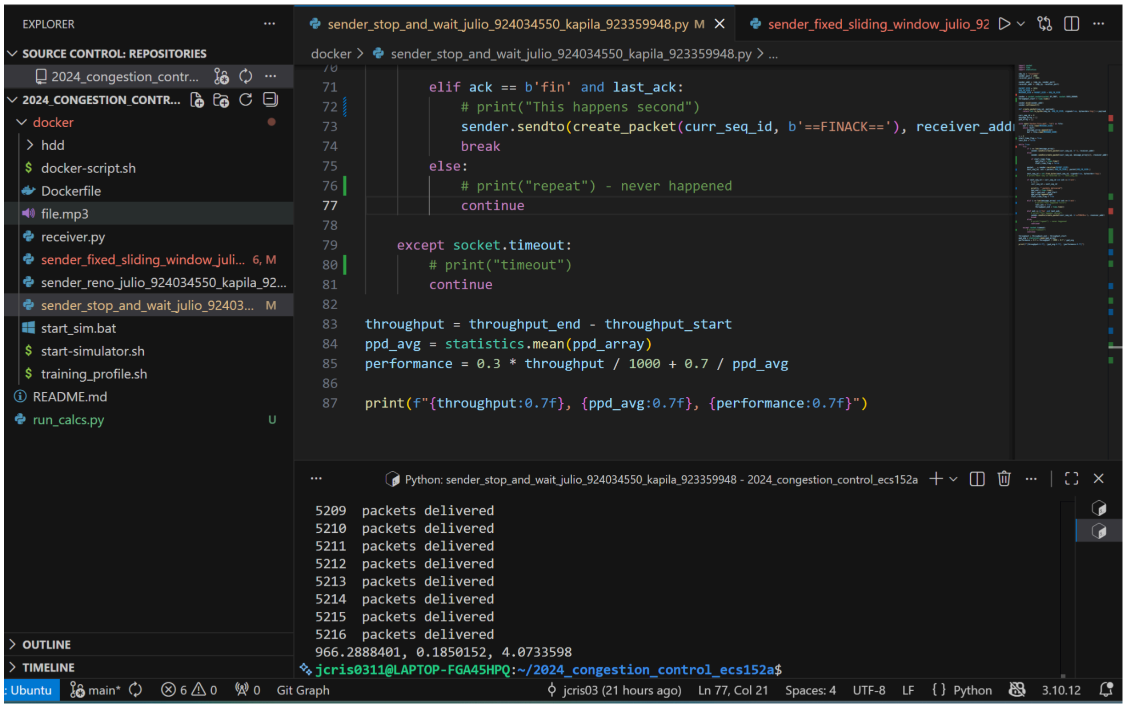
Our implementation only prints out one run of sending the mp3 file, so we run our sender 10 times, store results, and calculate the average and standard deviation of the metrics using python code.

b. External Sources

- How to skip iterations in a loop?: https://stackoverflow.com/questions/549674/how-to-skip-iterations-in-a-loop
- Socket Programming in Python (Guide): https://realpython.com/python-sockets/
- Socket Programming in Python: https://www.geeksforgeeks.org/python/socket-programming-python/
- TCP vs UDP Sockets in Python: https://www.youtube.com/watch?v=esLgiMLbRkI

c. Results

Screenshot of results for one run


EXPLORER
SOURCE CONTROL: REPOSITORIES
2024_congestion_contr...
2024_CONGESTION_CONTR...
docker
hdd
docker-script.sh
Dockerfile
file.mp3
receiver.py
sender_fixed_sliding_window_juli... 6, M
sender_reno_julio_924034550_kapila_92...
sender_stop_and_wait_julio_92403... M
start_sim.bat
start-simulator.sh
training_profile.sh
README.md
run_calcs.py U
OUTLINE
TIMELINE
sender_stop_and_wait_julio_924034550_kapila_923359948.py M
sender_fixed_sliding_window_julio_92
docker > sender_stop_and_wait_julio_924034550_kapila_923359948.py > ...
Python: sender_stop_and_wait_julio_924034550_kapila_923359948 - 2024_congestion_control_ecs152a
Ubuntu
main*
6 0
0
Git Graph
jcris03 (21 hours ago)
Ln 77, Col 21
Spaces: 4
UTF-8
LF
{ } Python
3.10.12

```python
            elif ack == b'fin' and last_ack:
                # print("This happens second")
                sender.sendto(create_packet(curr_seq_id, b'==FINACK=='), receiver_add
                break
            else:
                # print("repeat") - never happened
                continue

        except socket.timeout:
            # print("timeout")
            continue

    throughput = throughput_end - throughput_start
    ppd_avg = statistics.mean(ppd_array)
    performance = 0.3 * throughput / 1000 + 0.7 / ppd_avg

    print(f"{throughput:0.7f}, {ppd_avg:0.7f}, {performance:0.7f}")
```

```
5209  packets delivered
5210  packets delivered
5211  packets delivered
5212  packets delivered
5213  packets delivered
5214  packets delivered
5215  packets delivered
5216  packets delivered
966.2888401, 0.1850152, 4.0733598
jcris0311@LAPTOP-FGA45HPQ:~/2024_congestion_control_ecs152a$
```

Metrics Computation Over 10 Iterations

	Throughput	Per Packed Delay Avg	Performance
1	966.2888401	0.1850152	4.0733598
2	968.6239021	0.1854684	4.0648159
3	995.1880946	0.1905691	3.9717639
4	961.8505008	0.1841576	4.0896465
5	987.2919190	0.1890042	3.9998085
6	1009.3388352	0.1932499	3.9250546
7	976.3835568	0.1869321	4.0375894
8	998.8854973	0.1912448	3.9598949
9	959.4354711	0.1836857	4.0986871
10	987.4859211	0.1890804	3.9983756

Fixed Sliding Window Protocol:

a. Technical explanation

We implemented the code for the fixed sliding window protocol similarly to the stop and wait protocol code. After setting up the sender socket and dividing the mp3 file into chunks, we set up three indexes to help us record where the window is: the index of the last acknowledgement chunk “last_ack_index”, the index of the last sent chunk “las_sent_index”, and the index of the end of the window “window_index”, where “last_ack_index” and “window_index” should be 100 ints apart non-inclusive..

We implemented two main while loops to send and receive packets. The outer while loop will first send 100 packets, from “last_ack_index” to “window_index” (which should be 100 ints apart at all times with the exception of when we are sending the last 100 packets), using “last_sent_index” as our iterator. The tricky part is knowing which sequence ID to send, since in the stop and wait protocol, we just used the sequence id the receiver had sent previously. In this case, we analyzed the receiver code and noticed the next sequence ID’s are increments of “MESSAGE_SIZE” depending on which part of the mp3 file has been received.

Once all 100 packets are sent, we reach our inner loop, which is implemented similarly as the loop in the stop and wait protocol code. Each loop will try to receive a message from the receiver. We check if the sequence ID received from the receiver “next_seq_id” is bigger than the sequence ID of the packet that is next to be acknowledged. If that is the case, more than one packet can be acknowledged here, so we calculate how many packets were acknowledged (explained in next section), adjust the window accordingly by N, and then we reach another while loop which will send N packets. Once N packets are sent, we iterate back to the inner while loop to receive another packet from the receiver. In case of a timeout, we go back to the beginning of the outer loop and send 100 packets as explained in the previous paragraph.

Once the last_ack_index reaches one integer before “len(message_array)”, we send our blank message to initiate the termination process and exit the outer loop once the last acknowledgement is received. After that, we reach one final while loop which will keep sending empty messages to the receiver until the sender receives the ‘fin’ message, to which we can close the receiver socket.

For the throughput time, we calculate it similarly to stop and wait; get the start time after creating the socket and get the end time after the last packet acknowledgement is received. For the packet delays, we get the start time after sending a packet, and the end time after an acknowledgment is received, but if we get an acknowledgement for multiple packets, all those packets will have the same end time.

b. Window adjustment technique explanation

As previously mentioned, we sometimes can get an acknowledgement for multiple packets. We use the sequence IDs to figure out how many packets were acknowledged, by subtracting "next_seq_id" by "curr_seq_id", and since these sequence ID are multiples of "MESSAGE_SIZE", we divide the subtraction by "MESSAGE_SIZE" using "//" instead of "/" since the former results in an int with floor division (the latter results in a float). We store the result in "pack_acks", and this result is the amount of integers we need to increase "last_ack_index" and "window_index" (unless "window_index" has reached the end of "len(message_array)").

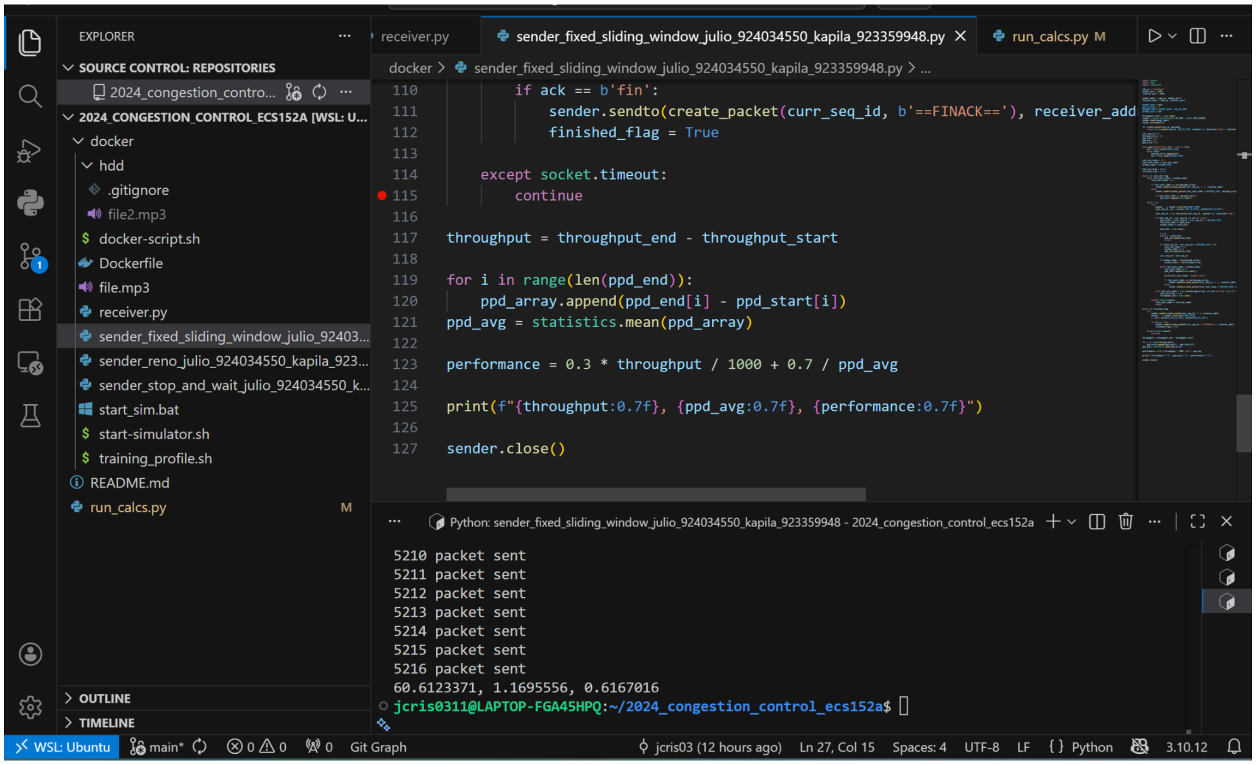
However, since the last packet won't be exactly the size of "MESSAGE_SIZE", we also check if the difference between the sequence IDs are not an exact multiple of "MESSAGE_SIZE" using modulo. If that is the case, that means we have acknowledged the last packet.

c. External sources

Same as Stop and Wait protocol

d. Results

Screenshot of results for one run


EXPLORER
SOURCE CONTROL: REPOSITORIES
2024_congestion_contro...
2024_CONGESTION_CONTROL_ECS152A [WSL: U...
docker
hdd
.gitignore
file2.mp3
docker-script.sh
Dockerfile
file.mp3
receiver.py
sender_fixed_sliding_window_julio_92403...
sender_reno_julio_924034550_kapila_923...
sender_stop_and_wait_julio_924034550_k...
start_sim.bat
start-simulator.sh
training_profile.sh
README.md
run_calcs.py
OUTLINE
TIMELINE
receiver.py
sender_fixed_sliding_window_julio_924034550_kapila_923359948.py
run_calcs.py M
docker > sender_fixed_sliding_window_julio_924034550_kapila_923359948.py > ...
Python: sender_fixed_sliding_window_julio_924034550_kapila_923359948 - 2024_congestion_control_ecs152a
WSL: Ubuntu
main*
Git Graph
jcris03 (12 hours ago)
Ln 27, Col 15
Spaces: 4
UTF-8
LF
Python
3.10.12

```python
        if ack == b'fin':
            sender.sendto(create_packet(curr_seq_id, b'==FINACK=='), receiver_add
            finished_flag = True

    except socket.timeout:
        continue

throughput = throughput_end - throughput_start

for i in range(len(ppd_end)):
    ppd_array.append(ppd_end[i] - ppd_start[i])
ppd_avg = statistics.mean(ppd_array)

performance = 0.3 * throughput / 1000 + 0.7 / ppd_avg

print(f"{throughput:0.7f}, {ppd_avg:0.7f}, {performance:0.7f}")

sender.close()
```

```
5210 packet sent
5211 packet sent
5212 packet sent
5213 packet sent
5214 packet sent
5215 packet sent
5216 packet sent
60.6123371, 1.1695556, 0.6167016
jcris0311@LAPTOP-FGA45HPQ:~/2024_congestion_control_ecs152a$
```

Metrics Computation Over 10 Iterations

	Throughput	**Per Packed Delay**	**Performance**
1	60.6123371	1.1695556	0.6167016
2	59.0753055	1.1315740	0.6363299
3	60.5363166	1.1604177	0.6213920
4	60.8001311	1.1732120	0.6148926
5	59.8971214	1.1494228	0.6269705
6	59.9257832	1.1559196	0.6235562
7	56.6331890	1.0925058	0.6577188
8	59.3523180	1.1428399	0.6303149
9	56.3041368	1.0502570	0.6833948
10	62.4335487	1.1837174	0.6100874

TCP Reno

a. **Technical Implementation and Explanation**

   i. **How was the slow-start threshold established?**

   The cwnd initially starts equal to 1 message size, and till it exceeds ssthresh, it is effectively doubled by adding to cwnd the count of the most recently acked bytes. Since cwnd bytes were sent, cwnd bytes are ack'd, and so cwnd + that count is effectively the same as cwnd * 2. Once cwnd exceeds ssthresh, the state is switched from slow start to congestion avoidance.

   ii. **Implementation of AIMD (Additive Increase Multiplicative Decrease) – How was the congestion window size increased?**

   While in the congestion avoidance state, the cwnd increases linearly according to the function f(ack'd bytes) = message_size * bytes ack'd / prev_cwnd + prev_cwnd. If all bytes are ack'd, then bytes ack'd == prev_cwnd and the increment to cwnd would be 1 MSS. If 3 duplicate acks are received, the fast retransmit & recovery state is entered, ssthresh is set to the max of half of cwnd and 1 MSS, and then cwnd is set to ssthresh + 3 MSS. This serves as the multiplicative decrease.

   iii. **How were fast retransmit and fast recovery implemented?**

   Once 3 duplicate acks are received, the fast retransmit process is started. Cwnd is multiplicatively decreased in the manner stated before, the state is changed to fast recovery, and 1 MSS from the last unack'd byte is retransmitted. In fast recovery, if another duplicate ack is received, cwnd is incremented by 1, if a new ack is received, cwnd is set to ssthresh, and the state is changed to congestion avoidance.

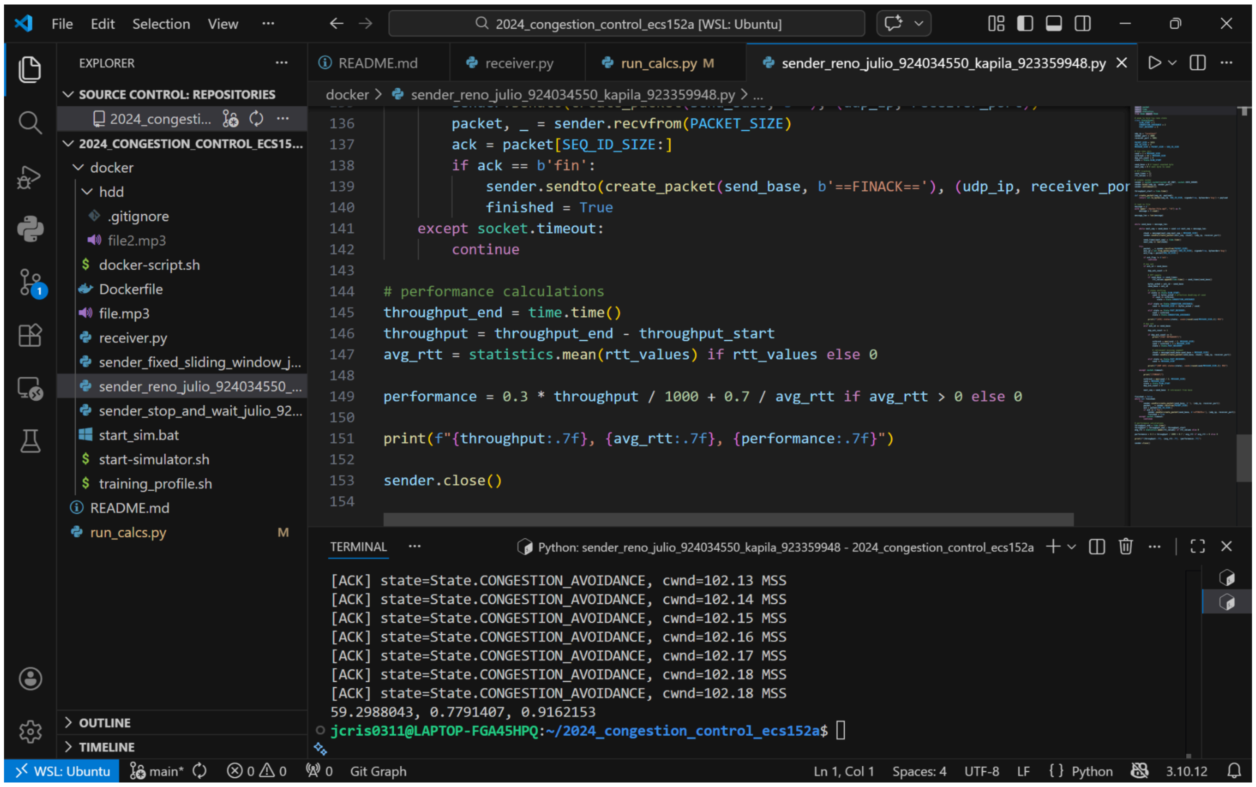
b. **Handling Congestion with Examples**

c. **Citing External Sources**

   https://www.geeksforgeeks.org/computer-networks/tcp-tahoe-and-tcp-reno/

   Asking Chatgpt to clarify concepts

d. **Metrics Computation Over 10 Iterations**



```python
        packet, _ = sender.recvfrom(PACKET_SIZE)
        ack = packet[SEQ_ID_SIZE:]
        if ack == b'fin':
            sender.sendto(create_packet(send_base, b'==FINACK=='), (udp_ip, receiver_por
            finished = True
    except socket.timeout:
        continue

# performance calculations
throughput_end = time.time()
throughput = throughput_end - throughput_start
avg_rtt = statistics.mean(rtt_values) if rtt_values else 0

performance = 0.3 * throughput / 1000 + 0.7 / avg_rtt if avg_rtt > 0 else 0

print(f"{throughput:.7f}, {avg_rtt:.7f}, {performance:.7f}")

sender.close()
```

```
[ACK] state=State.CONGESTION_AVOIDANCE, cwnd=102.13 MSS
[ACK] state=State.CONGESTION_AVOIDANCE, cwnd=102.14 MSS
[ACK] state=State.CONGESTION_AVOIDANCE, cwnd=102.15 MSS
[ACK] state=State.CONGESTION_AVOIDANCE, cwnd=102.16 MSS
[ACK] state=State.CONGESTION_AVOIDANCE, cwnd=102.17 MSS
[ACK] state=State.CONGESTION_AVOIDANCE, cwnd=102.18 MSS
[ACK] state=State.CONGESTION_AVOIDANCE, cwnd=102.18 MSS
59.2988043, 0.7791407, 0.9162153
jcris0311@LAPTOP-FGA45HPQ:~/2024_congestion_control_ecs152a$
```

Throughput	Avg RTT	Performance Metric
59.1265697	0.7553030	0.9445183
61.1773221	1.1591652	0.6222361
59.1042626	1.1045337	0.6514829
57.1225247	1.0244642	0.7004207
58.7858531	1.1175969	0.6439797
59.2988043	0.7791407	0.9162153
56.0382938	0.9978592	0.7183132
54.3050275	0.9937882	0.7206669
53.5749304	0.9653215	0.7412195
53.0045104	0.9617490	0.7437420

Metrics Table

Protocol	**Throughput**		**Per Packet Delay Average (RTT Average for TCP Reno)**		**Performance**	
	Average	**Standard Deviation**	**Average**	**Standard Deviation**	**Average**	**Standard Deviation**
Stop & Wait	981.0773	17.08966	0.1878407	0.003273601	4.0219	0.05962905
Fixed Sliding Window	59.55702	1.872257	1.140942	0.04086098	0.6321359	0.02246052
TCP Reno	57.15381	2.799509	0.9858922	0.1333593	0.7402795	0.1086423

Submission Page

Include this signed page with your submission

I certify that all submitted work is my own work. I have completed all of the assignments on my own without assistance from others except as indicated by appropriate citation. I have read and understand the university policy on plagiarism and academic dishonesty. I further understand that official sanctions will be imposed if there is any evidence of academic dishonesty in this work. I certify that the above statements are true.

For Group Submissions,

Team Member 1's contributions to this assignment:

Short Summary:

Implemented stop and wait protocol code and started implementing fixed sliding window protocol code

Julio Flores Cristerna	JulioFC	2/13/26
Full Name	Signature	Date

Team Member 2's contributions to this assignment:

Short Summary:

Finished up the remainder of sliding window (logic to terminate only after receiving 2 fins and small fixes and tweaks to code. Whole of TCP reno.

Kapil Mhaiskar	KapilaM	2/13/26
Full Name	Signature	Date