Haoyuan(Aaron) Shan

ECS152 Project 3

04 December 2024

Code Name (New Updated with the project instructions please check folder (code_withnogpt)

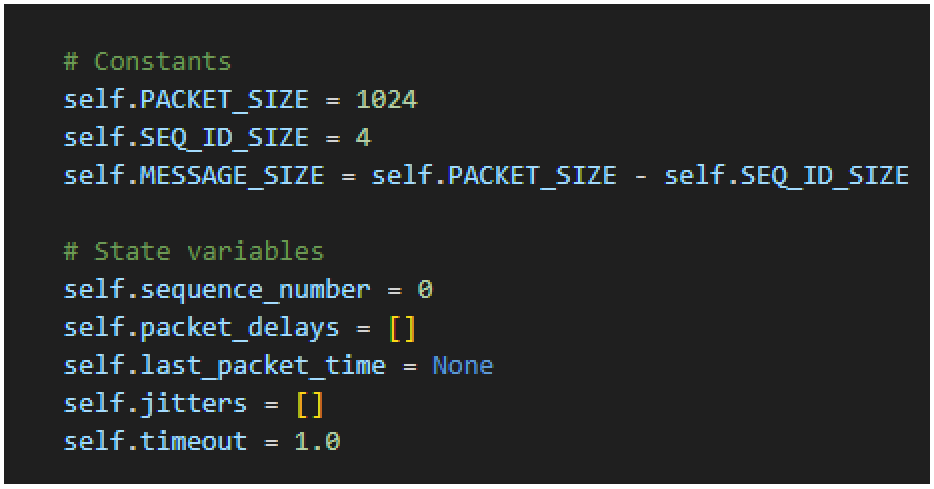
Section 1: Stop-and-Wait Protocol

Overview:

Stop-and-wait_protocol.py contains a class *StopAndWaitSender* and a main function:

Class StopAndWaitSender

Fields:



```
# Constants
self.PACKET_SIZE = 1024
self.SEQ_ID_SIZE = 4
self.MESSAGE_SIZE = self.PACKET_SIZE - self.SEQ_ID_SIZE

# State variables
self.sequence_number = 0
self.packet_delays = []
self.last_packet_time = None
self.jitters = []
self.timeout = 1.0
```

Methods:

- Constructor: create socket and initialize fields.
- *create_packet(self, data)*: adding sequence number to data, return data with sequence number
- *send_file(self, file_path)*: divide the file in file path into packets and use Stop and Wait protocol to send them to the receiver
- *send_fin_sequence(self)*: send the final packet to receiver

Function Main()

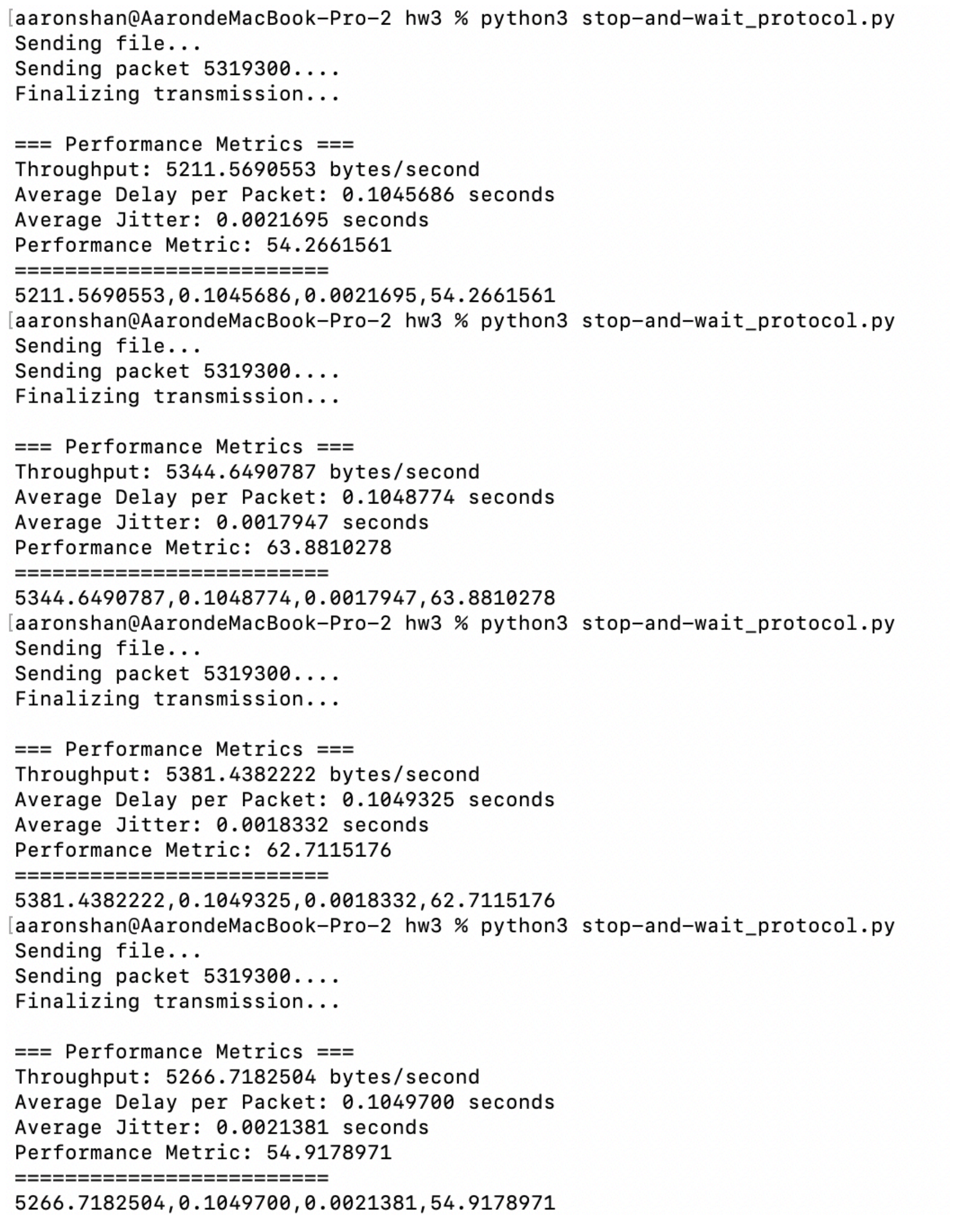
- Use *StopAndWaitSender* to send the file.
- Print formatted performance metrics

Details of protocol:

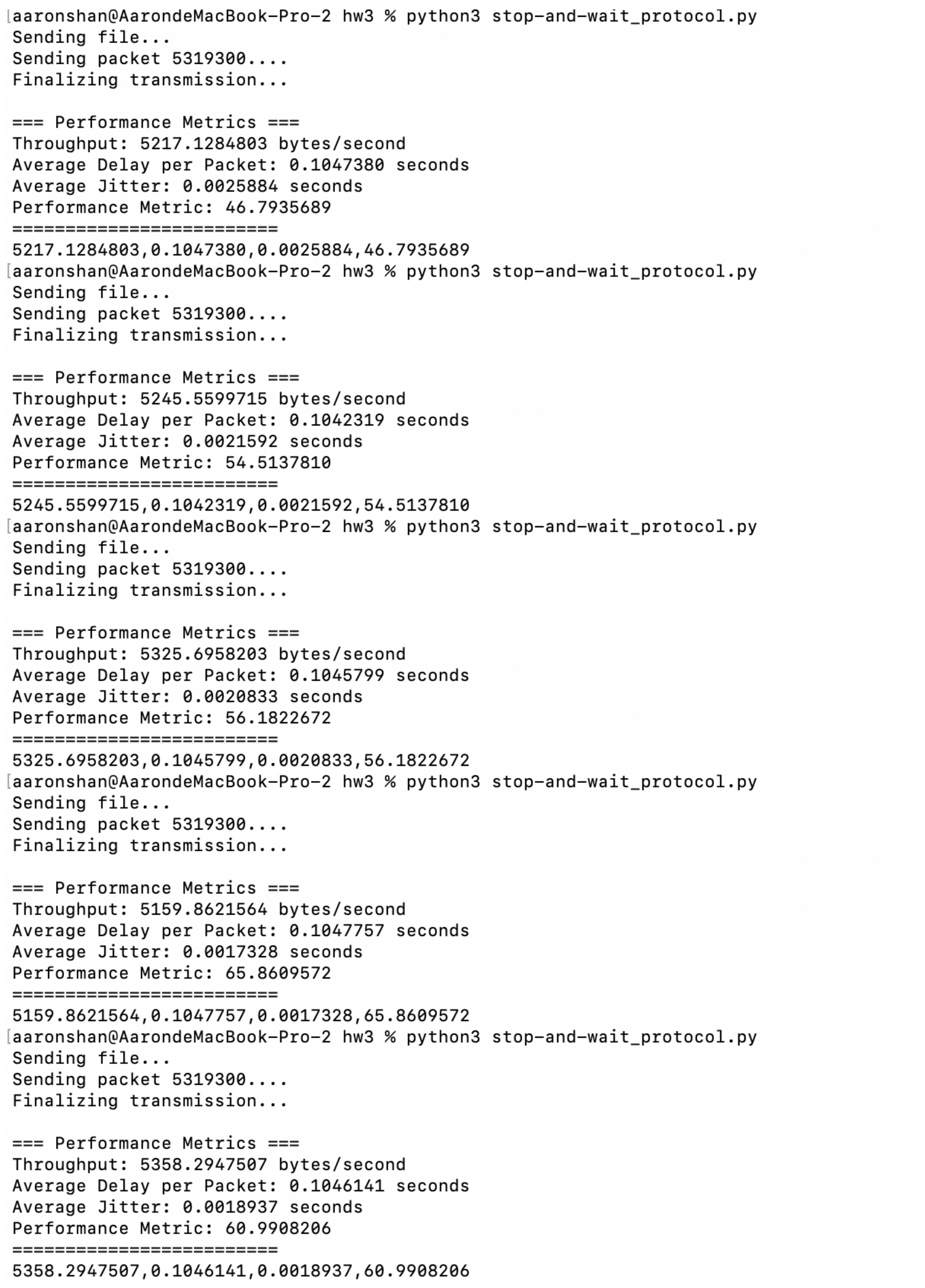
Most parts of protocol is in *send_file(self, file_path):*

- Start the timer for measuring throughput
- Create a bool *packet_sent* to record if the packet is sent. If so, we create a new packet from data and try to send it. If not, we retransmit the same packet.
- While *packet_send* is false, we start a timer, send a packet from data, and try to receive an ACK from the receiver.
  - If the ACK is received, we record packet delay, and turn *packet_send* to true
  - If timeout, we loop in while and resend the same packet
- After send all packets, record total time, calculate throughput, per-packet delay, and jitter

Conclusion for Section 1:

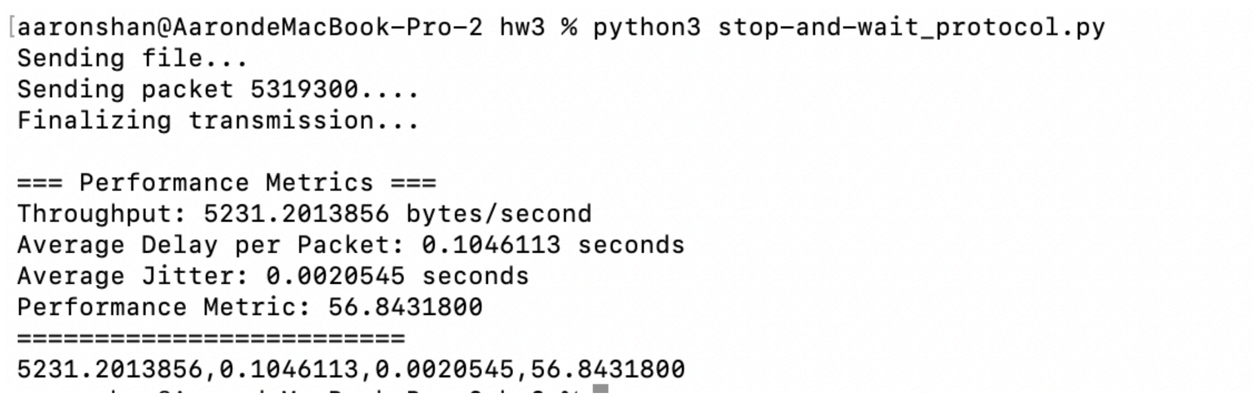
1. 

```
aaronshan@AarondeMacBook-Pro-2 hw3 % python3 stop-and-wait_protocol.py
Sending file...
Sending packet 5319300....
Finalizing transmission...

=== Performance Metrics ===
Throughput: 5211.5690553 bytes/second
Average Delay per Packet: 0.1045686 seconds
Average Jitter: 0.0021695 seconds
Performance Metric: 54.2661561
=========================
5211.5690553,0.1045686,0.0021695,54.2661561
aaronshan@AarondeMacBook-Pro-2 hw3 % python3 stop-and-wait_protocol.py
Sending file...
Sending packet 5319300....
Finalizing transmission...

=== Performance Metrics ===
Throughput: 5344.6490787 bytes/second
Average Delay per Packet: 0.1048774 seconds
Average Jitter: 0.0017947 seconds
Performance Metric: 63.8810278
=========================
5344.6490787,0.1048774,0.0017947,63.8810278
aaronshan@AarondeMacBook-Pro-2 hw3 % python3 stop-and-wait_protocol.py
Sending file...
Sending packet 5319300....
Finalizing transmission...

=== Performance Metrics ===
Throughput: 5381.4382222 bytes/second
Average Delay per Packet: 0.1049325 seconds
Average Jitter: 0.0018332 seconds
Performance Metric: 62.7115176
=========================
5381.4382222,0.1049325,0.0018332,62.7115176
aaronshan@AarondeMacBook-Pro-2 hw3 % python3 stop-and-wait_protocol.py
Sending file...
Sending packet 5319300....
Finalizing transmission...

=== Performance Metrics ===
Throughput: 5266.7182504 bytes/second
Average Delay per Packet: 0.1049700 seconds
Average Jitter: 0.0021381 seconds
Performance Metric: 54.9178971
=========================
5266.7182504,0.1049700,0.0021381,54.9178971
```



```
aaronshan@AarondeMacBook-Pro-2 hw3 % python3 stop-and-wait_protocol.py
Sending file...
Sending packet 5319300....
Finalizing transmission...

=== Performance Metrics ===
Throughput: 5217.1284803 bytes/second
Average Delay per Packet: 0.1047380 seconds
Average Jitter: 0.0025884 seconds
Performance Metric: 46.7935689
=========================
5217.1284803,0.1047380,0.0025884,46.7935689
aaronshan@AarondeMacBook-Pro-2 hw3 % python3 stop-and-wait_protocol.py
Sending file...
Sending packet 5319300....
Finalizing transmission...

=== Performance Metrics ===
Throughput: 5245.5599715 bytes/second
Average Delay per Packet: 0.1042319 seconds
Average Jitter: 0.0021592 seconds
Performance Metric: 54.5137810
=========================
5245.5599715,0.1042319,0.0021592,54.5137810
aaronshan@AarondeMacBook-Pro-2 hw3 % python3 stop-and-wait_protocol.py
Sending file...
Sending packet 5319300....
Finalizing transmission...

=== Performance Metrics ===
Throughput: 5325.6958203 bytes/second
Average Delay per Packet: 0.1045799 seconds
Average Jitter: 0.0020833 seconds
Performance Metric: 56.1822672
=========================
5325.6958203,0.1045799,0.0020833,56.1822672
aaronshan@AarondeMacBook-Pro-2 hw3 % python3 stop-and-wait_protocol.py
Sending file...
Sending packet 5319300....
Finalizing transmission...

=== Performance Metrics ===
Throughput: 5159.8621564 bytes/second
Average Delay per Packet: 0.1047757 seconds
Average Jitter: 0.0017328 seconds
Performance Metric: 65.8609572
=========================
5159.8621564,0.1047757,0.0017328,65.8609572
aaronshan@AarondeMacBook-Pro-2 hw3 % python3 stop-and-wait_protocol.py
Sending file...
Sending packet 5319300....
Finalizing transmission...

=== Performance Metrics ===
Throughput: 5358.2947507 bytes/second
Average Delay per Packet: 0.1046141 seconds
Average Jitter: 0.0018937 seconds
Performance Metric: 60.9908206
=========================
5358.2947507,0.1046141,0.0018937,60.9908206
```

2.

3. 

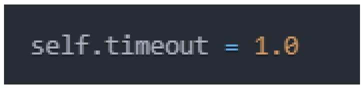
```
[aaronshan@AarondeMacBook-Pro-2 hw3 % python3 stop-and-wait_protocol.py
Sending file...
Sending packet 5319300....
Finalizing transmission...

=== Performance Metrics ===
Throughput: 5231.2013856 bytes/second
Average Delay per Packet: 0.1046113 seconds
Average Jitter: 0.0020545 seconds
Performance Metric: 56.8431800
=========================
5231.2013856,0.1046113,0.0020545,56.8431800
```

4. We collect 10 times data when we run it. The data are very similar. Through our code, we set our timeout value to 1.0 to ensure the sending process can be successful. At the beginning of the testing, we set the timeout to be 0.1. After we were doing the comparison, if we set the timeout to 1.0, the answer will be more trustable and it will also let the sending process be more efficient.
5. Due to many different reasons and affections, the answer might not be the most efficient as TA Vijeth mentions.
6. Calculations: We need to calculate the average and the standard deviation:

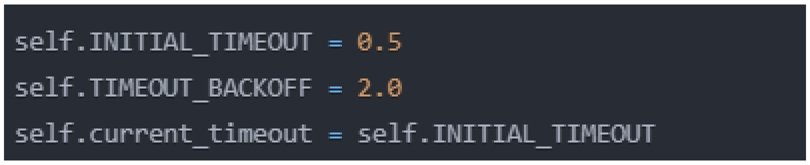
	Average (in 10 trials)	Standard Deviation (in 10 trials)
Throughput	5274.2117171	70.1176860
Per-packet delay	0.1045899	0.00035135787
Performance metric	57.69611735	5.4076080655568

Based on the gpt, what we get, we improve the efficient of how the code can handle the more throughput, less delay and jitter.

We made some changes for example:

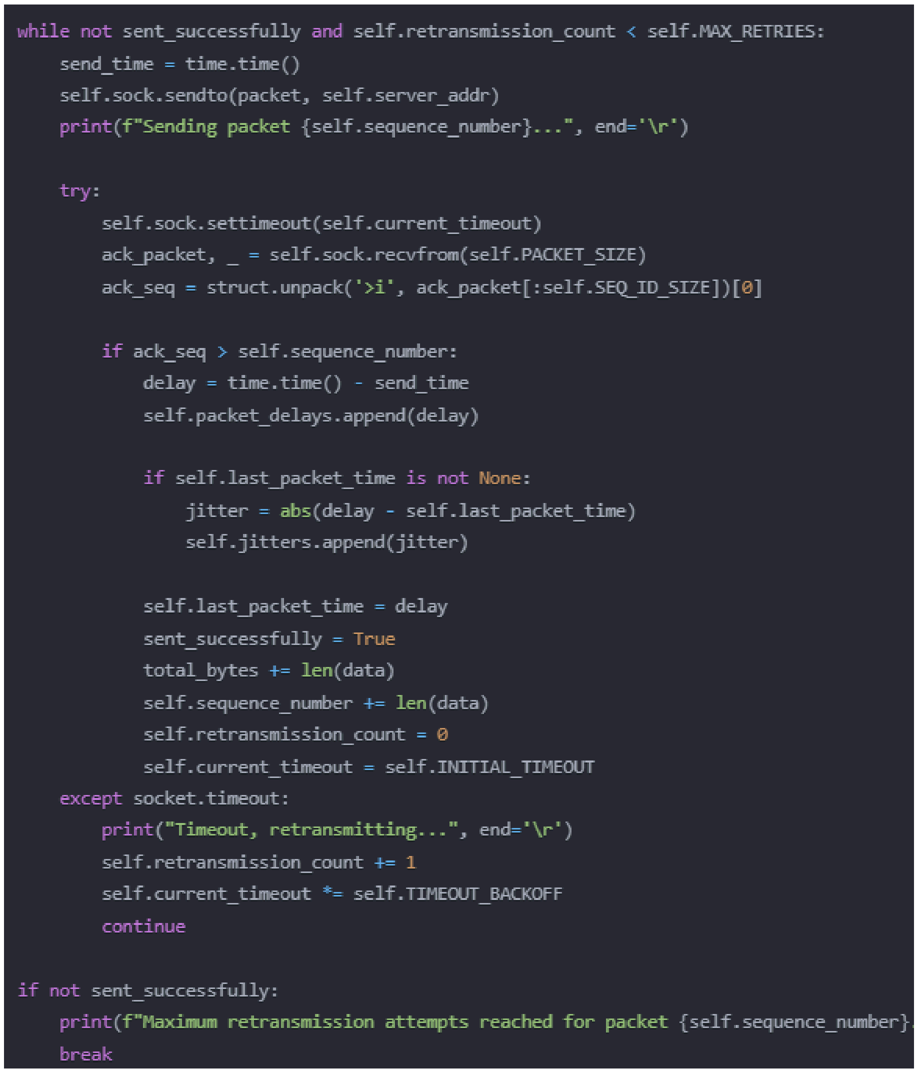


```
self.timeout = 1.0
```



```
self.INITIAL_TIMEOUT = 0.5
self.TIMEOUT_BACKOFF = 2.0
self.current_timeout = self.INITIAL_TIMEOUT
```

This was an general improve, we set the timeout to 1.0 and it be the constant value that we are going to use.

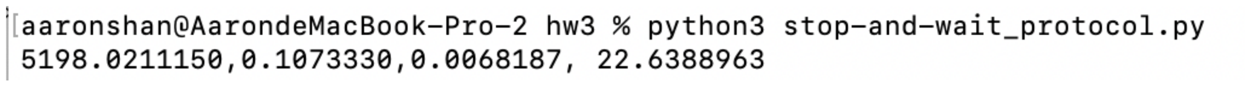


```
while not sent_successfully and self.retransmission_count < self.MAX_RETRIES:
    send_time = time.time()
    self.sock.sendto(packet, self.server_addr)
    print(f"Sending packet {self.sequence_number}...", end='\r')

    try:
        self.sock.settimeout(self.current_timeout)
        ack_packet, _ = self.sock.recvfrom(self.PACKET_SIZE)
        ack_seq = struct.unpack('>i', ack_packet[:self.SEQ_ID_SIZE])[0]

        if ack_seq > self.sequence_number:
            delay = time.time() - send_time
            self.packet_delays.append(delay)

            if self.last_packet_time is not None:
                jitter = abs(delay - self.last_packet_time)
                self.jitters.append(jitter)

            self.last_packet_time = delay
            sent_successfully = True
            total_bytes += len(data)
            self.sequence_number += len(data)
            self.retransmission_count = 0
            self.current_timeout = self.INITIAL_TIMEOUT
    except socket.timeout:
        print("Timeout, retransmitting...", end='\r')
        self.retransmission_count += 1
        self.current_timeout *= self.TIMEOUT_BACKOFF
        continue

if not sent_successfully:
    print(f"Maximum retransmission attempts reached for packet {self.sequence_number}
    break
```

The enhanced version adds MAX_RETRIES constant that sets threshold of how often one packet can be resent. It also stores the value of retransmission_count, and multiplies the current_timeout by the TIMEOUT_BACKOFF factor for every retransmission. If the maximum ‘retransmissions’ are exhausted, the enhanced version will terminate the packet transmission for the current packet it and proceed to the next packet.

There are some more changes by chatgpt so we just list here as examples

.Gpt assistance code for the output: just one example to make sure we can get our output



```
[aaronshan@AarondeMacBook-Pro-2 hw3 % python3 stop-and-wait_protocol.py
5198.0211150,0.1073330,0.0068187, 22.6388963
```

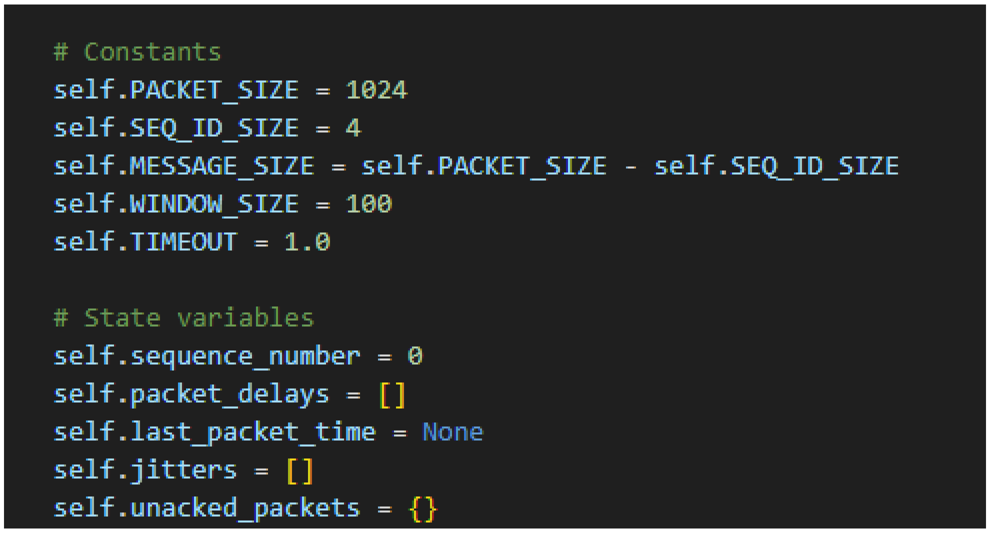
Section 2: Fixed Sliding Window Protocol with size 100 packets

Overview:

fsw_protocol.py contains a class *FixedWindowSender* and a function *main()*.

Class FixedWindowSender

Fields:



```
# Constants
self.PACKET_SIZE = 1024
self.SEQ_ID_SIZE = 4
self.MESSAGE_SIZE = self.PACKET_SIZE - self.SEQ_ID_SIZE
self.WINDOW_SIZE = 100
self.TIMEOUT = 1.0

# State variables
self.sequence_number = 0
self.packet_delays = []
self.last_packet_time = None
self.jitters = []
self.unacked_packets = {}
```

Methods:

- Constructor: create socket and initialize fields.

- *create_packet(self, data)*: adding sequence number to data, return data with sequence number
- *send_file(self, file_path)*: divide the file in file path into packets and use fixed sliding window protocol to send them to the receiver. This will also send the final packet and return throughout, pre packet delay, and jitter
- *close(self):* close the socket

Function main():

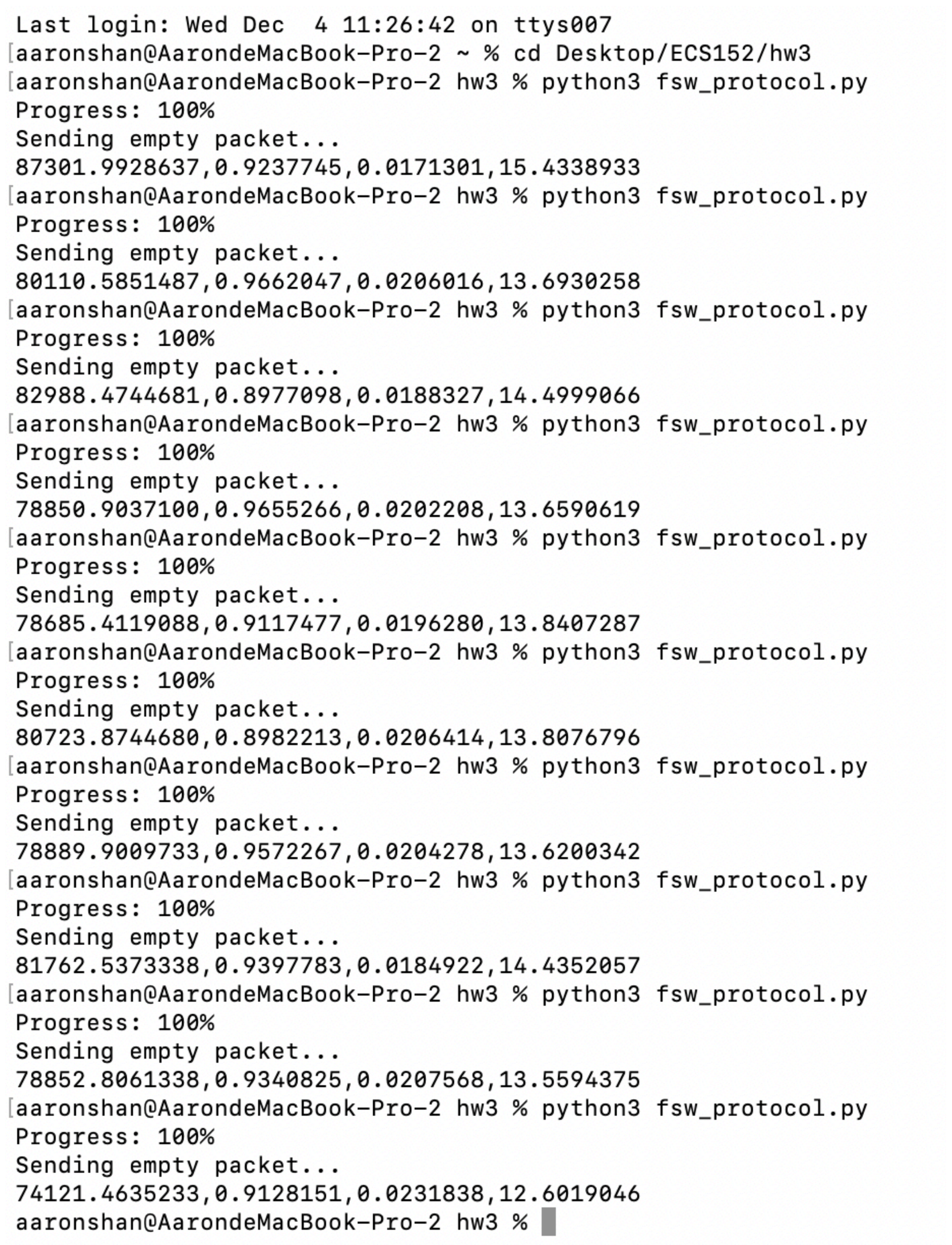
- Call the methods in *FixedWindowSender* to get throughput, pre-packet delay, and jitter and print them. Also calculate the matrix.

Details of protocol:

Most parts of protocol is in *send_file(self, file_path):*

- Start the timer for throughput.
- Use while to loop checking the number of unACKed packets. If the number of unACKed packets is smaller than the window size, we send more packets to fill the window.
  - Try to get ACKs in a while loop, if we get ACKs, we remove all the ACKed packets from the unACKed packets list and calculate delay.
  - If there is a timeout, we retransmit all unACKed packet
- Send the final packet after we send all the packets
- Calculate and return throughput, pre-packet delay, and jitter

Conclusion:



```
Last login: Wed Dec  4 11:26:42 on ttys007
[aaronshan@AarondeMacBook-Pro-2 ~ % cd Desktop/ECS152/hw3
[aaronshan@AarondeMacBook-Pro-2 hw3 % python3 fsw_protocol.py
Progress: 100%
Sending empty packet...
87301.9928637,0.9237745,0.0171301,15.4338933
[aaronshan@AarondeMacBook-Pro-2 hw3 % python3 fsw_protocol.py
Progress: 100%
Sending empty packet...
80110.5851487,0.9662047,0.0206016,13.6930258
[aaronshan@AarondeMacBook-Pro-2 hw3 % python3 fsw_protocol.py
Progress: 100%
Sending empty packet...
82988.4744681,0.8977098,0.0188327,14.4999066
[aaronshan@AarondeMacBook-Pro-2 hw3 % python3 fsw_protocol.py
Progress: 100%
Sending empty packet...
78850.9037100,0.9655266,0.0202208,13.6590619
[aaronshan@AarondeMacBook-Pro-2 hw3 % python3 fsw_protocol.py
Progress: 100%
Sending empty packet...
78685.4119088,0.9117477,0.0196280,13.8407287
[aaronshan@AarondeMacBook-Pro-2 hw3 % python3 fsw_protocol.py
Progress: 100%
Sending empty packet...
80723.8744680,0.8982213,0.0206414,13.8076796
[aaronshan@AarondeMacBook-Pro-2 hw3 % python3 fsw_protocol.py
Progress: 100%
Sending empty packet...
78889.9009733,0.9572267,0.0204278,13.6200342
[aaronshan@AarondeMacBook-Pro-2 hw3 % python3 fsw_protocol.py
Progress: 100%
Sending empty packet...
81762.5373338,0.9397783,0.0184922,14.4352057
[aaronshan@AarondeMacBook-Pro-2 hw3 % python3 fsw_protocol.py
Progress: 100%
Sending empty packet...
78852.8061338,0.9340825,0.0207568,13.5594375
[aaronshan@AarondeMacBook-Pro-2 hw3 % python3 fsw_protocol.py
Progress: 100%
Sending empty packet...
74121.4635233,0.9128151,0.0231838,12.6019046
aaronshan@AarondeMacBook-Pro-2 hw3 %
```

1.

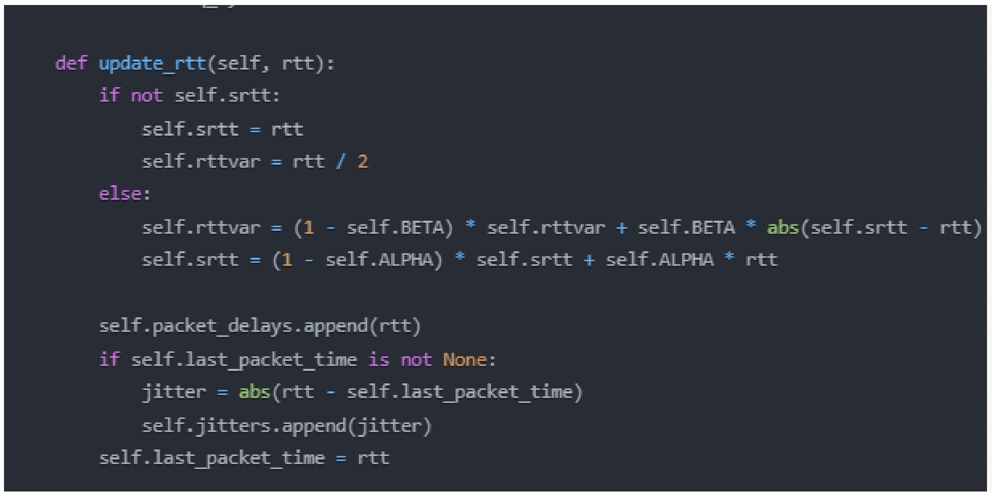
2. Based on the project requirements, we need to make sure the debug information is limited and we only need to print out throughput, delay, jitter, and metric, in the final submission, I will delete all the no-needed debug information.

3. Based on the requirements, we need to set our window size to be 100.

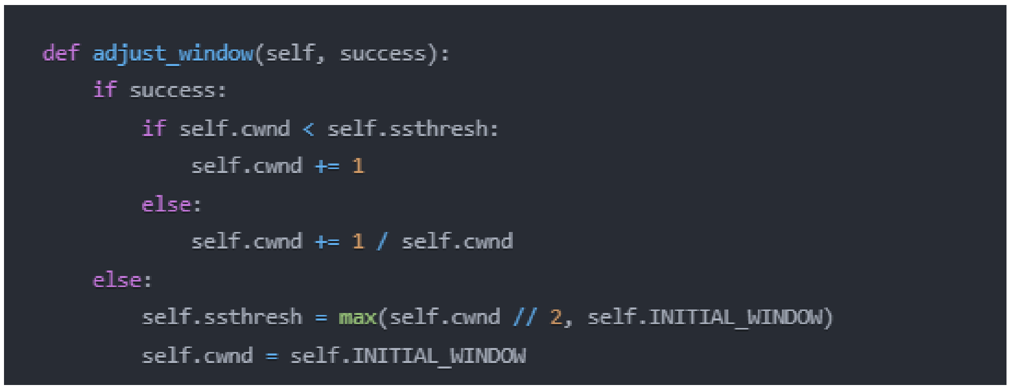
	Average (in 10 trials)	Standard Deviation (in 10 trials)
Throughput	80228.79505315	3248.4380507
Pre-packet delay	0.9307087	0.02478152
Performance metric	13.9150877	0.7063445

For gpt assistance, we change some parts to make sure that it might help us to make our code to be more faster

Here are some changes we did (examples)



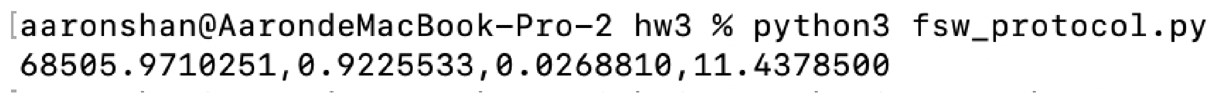
```
def update_rtt(self, rtt):
    if not self.srtt:
        self.srtt = rtt
        self.rttvar = rtt / 2
    else:
        self.rttvar = (1 - self.BETA) * self.rttvar + self.BETA * abs(self.srtt - rtt)
        self.srtt = (1 - self.ALPHA) * self.srtt + self.ALPHA * rtt

    self.packet_delays.append(rtt)
    if self.last_packet_time is not None:
        jitter = abs(rtt - self.last_packet_time)
        self.jitters.append(jitter)
    self.last_packet_time = rtt
```



```
def adjust_window(self, success):
    if success:
        if self.cwnd < self.ssthresh:
            self.cwnd += 1
        else:
            self.cwnd += 1 / self.cwnd
    else:
        self.ssthresh = max(self.cwnd // 2, self.INITIAL_WINDOW)
        self.cwnd = self.INITIAL_WINDOW
```

Based on the gpt mentions, we add rtt idea to our code and try to let the code be more efficients and try to get more throughput.

Here is an example of result was generated by the code with GPT assistance



```
aaronshan@AarondeMacBook-Pro-2 hw3 % python3 fsw_protocol.py
68505.9710251,0.9225533,0.0268810,11.4378500
```

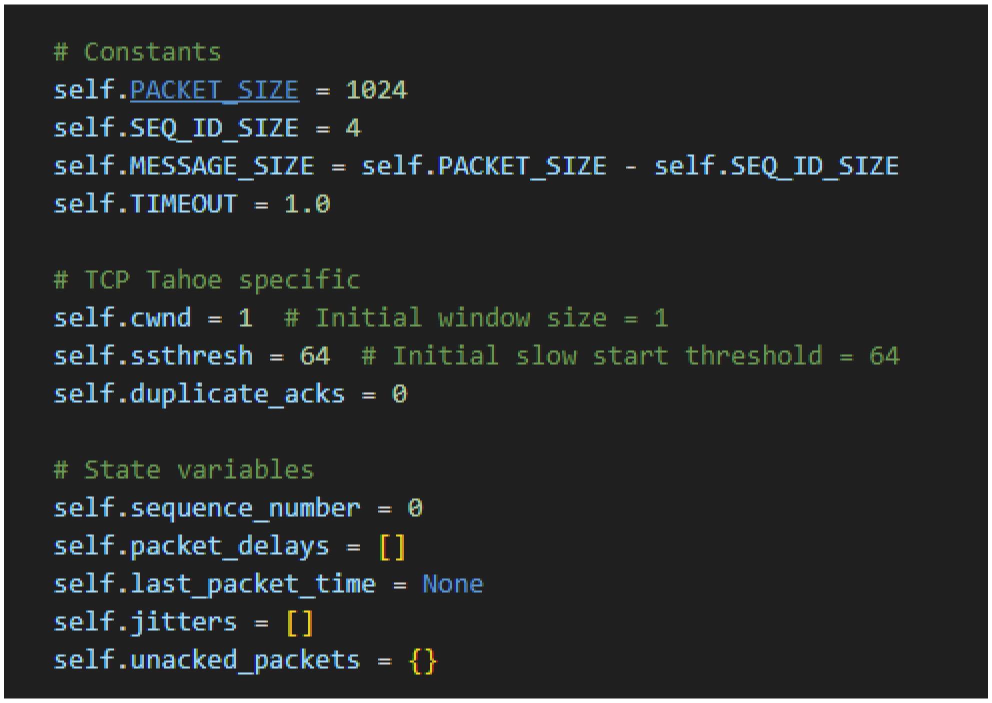
Section 3: TCP Tahoe

Overview:

tcp_tahoe.py contains a class *TCPTahoeSender* and a function *main()*.

Class TCPTahoeSender:

Fields:



```
# Constants
self.PACKET_SIZE = 1024
self.SEQ_ID_SIZE = 4
self.MESSAGE_SIZE = self.PACKET_SIZE - self.SEQ_ID_SIZE
self.TIMEOUT = 1.0

# TCP Tahoe specific
self.cwnd = 1  # Initial window size = 1
self.ssthresh = 64  # Initial slow start threshold = 64
self.duplicate_acks = 0

# State variables
self.sequence_number = 0
self.packet_delays = []
self.last_packet_time = None
self.jitters = []
self.unacked_packets = {}
```

Methods:

- Constructor: create socket and initialize fields.

- *create_packet(self, data)*: adding sequence number to data, return data with sequence number
- *handle_duplicate_ack(self):* set current window size to 1 when duplicate ACK detected. Also reset the duplicate ACK counter
- *handle_timeout(self):* set current window size to 1 when a timeout occurs
- *send_file(self, file_path)*: divide the file in file path into packets and use TCP Tahoe protocol to send them to the receiver. This will also send the final packet and return throughout, pre packet delay, and jitter
- *close(self):* close the socket

Function main():

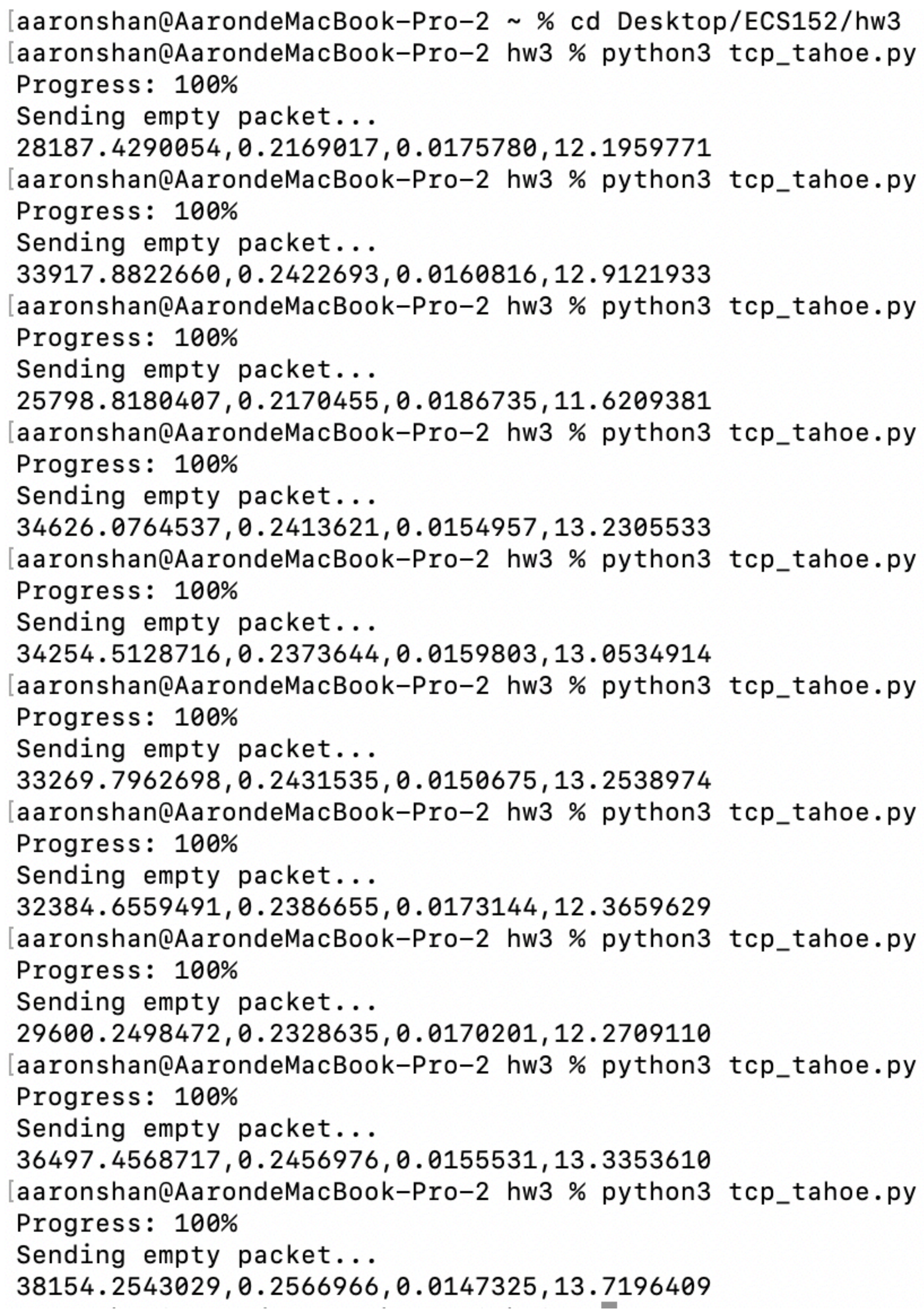
- Call the methods in *TCPTahoeSender* to get throughput, pre-packet delay, and jitter and print them. Also calculate the matrix.

Details of protocol:

Most parts of protocol is in *send_file(self, file_path):*

We reuse the code of Fixed Sliding Window Protocol. In Tahoe, instead of fixed window size in Fixed Sliding Window Protocol, the methods *handle_duplicate_ack(self)* and *handle_timeout(self)* will change the window size when duplicate ACK or a timeout happen.

Conclusion:

1. 

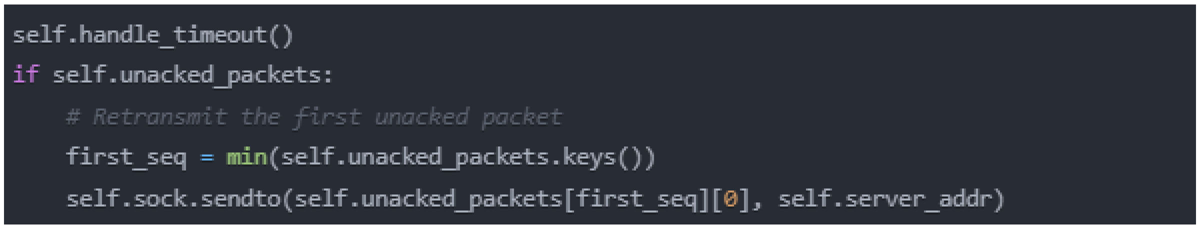
```
[aaronshan@AarondeMacBook-Pro-2 ~ % cd Desktop/ECS152/hw3
[aaronshan@AarondeMacBook-Pro-2 hw3 % python3 tcp_tahoe.py
Progress: 100%
Sending empty packet...
28187.4290054,0.2169017,0.0175780,12.1959771
[aaronshan@AarondeMacBook-Pro-2 hw3 % python3 tcp_tahoe.py
Progress: 100%
Sending empty packet...
33917.8822660,0.2422693,0.0160816,12.9121933
[aaronshan@AarondeMacBook-Pro-2 hw3 % python3 tcp_tahoe.py
Progress: 100%
Sending empty packet...
25798.8180407,0.2170455,0.0186735,11.6209381
[aaronshan@AarondeMacBook-Pro-2 hw3 % python3 tcp_tahoe.py
Progress: 100%
Sending empty packet...
34626.0764537,0.2413621,0.0154957,13.2305533
[aaronshan@AarondeMacBook-Pro-2 hw3 % python3 tcp_tahoe.py
Progress: 100%
Sending empty packet...
34254.5128716,0.2373644,0.0159803,13.0534914
[aaronshan@AarondeMacBook-Pro-2 hw3 % python3 tcp_tahoe.py
Progress: 100%
Sending empty packet...
33269.7962698,0.2431535,0.0150675,13.2538974
[aaronshan@AarondeMacBook-Pro-2 hw3 % python3 tcp_tahoe.py
Progress: 100%
Sending empty packet...
32384.6559491,0.2386655,0.0173144,12.3659629
[aaronshan@AarondeMacBook-Pro-2 hw3 % python3 tcp_tahoe.py
Progress: 100%
Sending empty packet...
29600.2498472,0.2328635,0.0170201,12.2709110
[aaronshan@AarondeMacBook-Pro-2 hw3 % python3 tcp_tahoe.py
Progress: 100%
Sending empty packet...
36497.4568717,0.2456976,0.0155531,13.3353610
[aaronshan@AarondeMacBook-Pro-2 hw3 % python3 tcp_tahoe.py
Progress: 100%
Sending empty packet...
38154.2543029,0.2566966,0.0147325,13.7196409
```

2. Calculation: We need to calculate the Average of each value and

3.

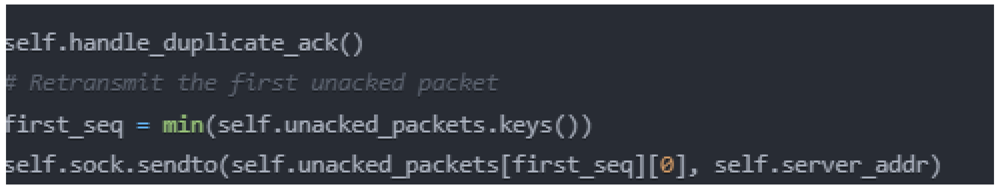
	Average (in 10 trials)	Standard Deviation (in 10 trials)
Throughput	32669.1131878	3602.6617495
Pre-packet delay	0.2372019	0.01170468
Performance metric	12.7958926	0.61836722

We get gpt assistance to try to improve the process time as needed, here are some examples of changes by the idea was memotions by gpt:



```
self.handle_timeout()
if self.unacked_packets:
    # Retransmit the first unacked packet
    first_seq = min(self.unacked_packets.keys())
    self.sock.sendto(self.unacked_packets[first_seq][0], self.server_addr)
```

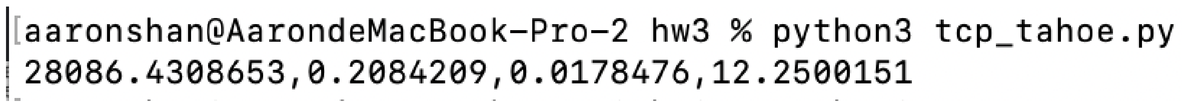
This makes the retransmission handling more straightforward by extending the handle_timeout() method to include the retransmission of the first unacked packet, instead of having a block of code that does it.



```
self.handle_duplicate_ack()
# Retransmit the first unacked packet
first_seq = min(self.unacked_packets.keys())
self.sock.sendto(self.unacked_packets[first_seq][0], self.server_addr)
```

Unlike in the original implementation, the handle_duplicate_ack() method of the TCPTahoeSender class only contains a comments line and three lines providing a simple set of rules for updating ssthresh, cwnd, and the number of the received duplicate acknowledgments.

Here is a result as example:



```
[aaronshan@AarondeMacBook-Pro-2 hw3 % python3 tcp_tahoe.py
28086.4308653,0.2084209,0.0178476,12.2500151
```

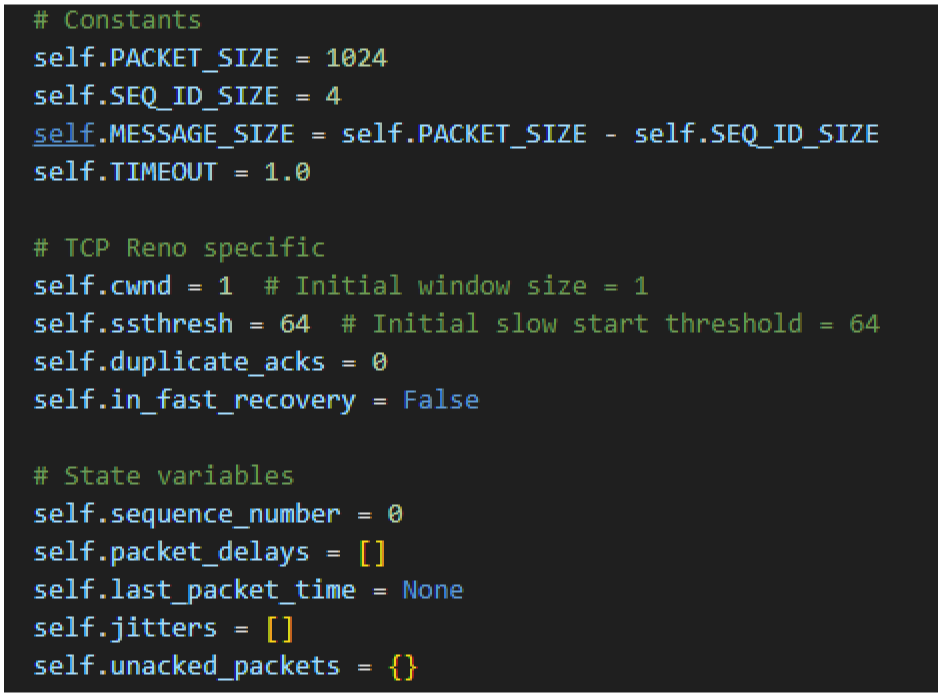
Section 4: TCP Reno

Overview:

tcp_reno.py contains a class *TCPRenoSender* and a function *main()*.

Class TCPRenoSender:

Fields:



```
# Constants
self.PACKET_SIZE = 1024
self.SEQ_ID_SIZE = 4
self.MESSAGE_SIZE = self.PACKET_SIZE - self.SEQ_ID_SIZE
self.TIMEOUT = 1.0

# TCP Reno specific
self.cwnd = 1  # Initial window size = 1
self.ssthresh = 64  # Initial slow start threshold = 64
self.duplicate_acks = 0
self.in_fast_recovery = False

# State variables
self.sequence_number = 0
self.packet_delays = []
self.last_packet_time = None
self.jitters = []
self.unacked_packets = {}
```

Methods:

- Constructor: create socket and initialize fields.
- *create_packet(self, data)*: adding sequence number to data, return data with sequence number

- *handle_duplicate_ack(self):* calculate *ssthresh and* set current window size to *ssthresh* + 3 when duplicate ACK detected. Also reset the duplicate ACK counter and set *in_fast_recovery* to true
- *handle_timeout(self):* set current window size to 1 when a timeout occurs. Also reset the duplicate ACK counter
- *send_file(self, file_path)*: divide the file in file path into packets and use TCP Reno protocol to send them to the receiver. This will also send the final packet and return throughout, pre packet delay, and jitter
- *close(self):* close the socket

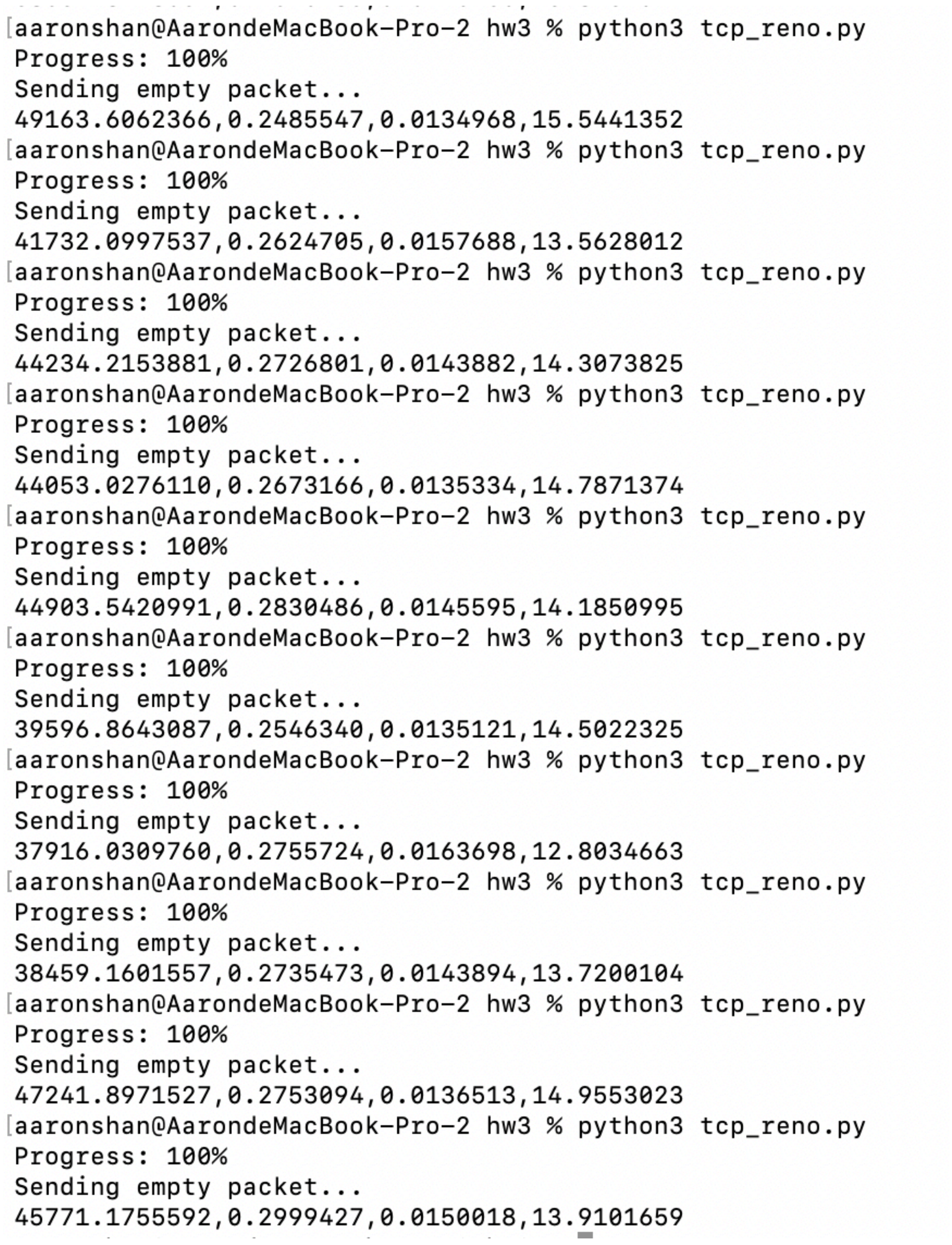
Function main():

- Call the methods in *TCPRenoSender* to get throughput, pre-packet delay, and jitter and print them. Also calculate the matrix.

Details of protocol:

Most parts of protocol is in *send_file(self, file_path):*

We reuse the code of TCP Tahoe Protocol. In Reno, instead of set window size to 1 in TCP Tahoe when duplicate ACK detected, the methods *handle_duplicate_ack(self)* will change the window size to ½ * current window size when duplicate ACK happens.

1. 

```
aaronshan@AarondeMacBook-Pro-2 hw3 % python3 tcp_reno.py
Progress: 100%
Sending empty packet...
49163.6062366,0.2485547,0.0134968,15.5441352
aaronshan@AarondeMacBook-Pro-2 hw3 % python3 tcp_reno.py
Progress: 100%
Sending empty packet...
41732.0997537,0.2624705,0.0157688,13.5628012
aaronshan@AarondeMacBook-Pro-2 hw3 % python3 tcp_reno.py
Progress: 100%
Sending empty packet...
44234.2153881,0.2726801,0.0143882,14.3073825
aaronshan@AarondeMacBook-Pro-2 hw3 % python3 tcp_reno.py
Progress: 100%
Sending empty packet...
44053.0276110,0.2673166,0.0135334,14.7871374
aaronshan@AarondeMacBook-Pro-2 hw3 % python3 tcp_reno.py
Progress: 100%
Sending empty packet...
44903.5420991,0.2830486,0.0145595,14.1850995
aaronshan@AarondeMacBook-Pro-2 hw3 % python3 tcp_reno.py
Progress: 100%
Sending empty packet...
39596.8643087,0.2546340,0.0135121,14.5022325
aaronshan@AarondeMacBook-Pro-2 hw3 % python3 tcp_reno.py
Progress: 100%
Sending empty packet...
37916.0309760,0.2755724,0.0163698,12.8034663
aaronshan@AarondeMacBook-Pro-2 hw3 % python3 tcp_reno.py
Progress: 100%
Sending empty packet...
38459.1601557,0.2735473,0.0143894,13.7200104
aaronshan@AarondeMacBook-Pro-2 hw3 % python3 tcp_reno.py
Progress: 100%
Sending empty packet...
47241.8971527,0.2753094,0.0136513,14.9553023
aaronshan@AarondeMacBook-Pro-2 hw3 % python3 tcp_reno.py
Progress: 100%
Sending empty packet...
45771.1755592,0.2999427,0.0150018,13.9101659
```

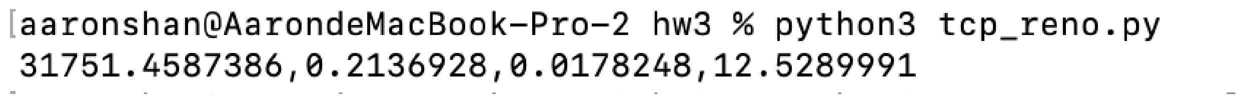
2.

	Average (in 10 trials)	Standard Deviation (in 10 trials)
Throughput	43307.1619240	3586.5047217

Pre-packet delay	0.2713066	0.01374346
Performance metric	14.2277733	0.7408115

We get some help from the gpt:

The originally idea and some steps that we did are very similar like what we did the change in tcp tahoe. But We provide the sample output front he TCP reno:



```
[aaronshan@AarondeMacBook-Pro-2 hw3 % python3 tcp_reno.py
 31751.4587386,0.2136928,0.0178248,12.5289991
```

Also, based on what we learn from the class, the TCP reno should faster and have more throughput and higher metric than TCP tahoe. Through the data, we can see, we are right!

Stage 4: Custom protocol:

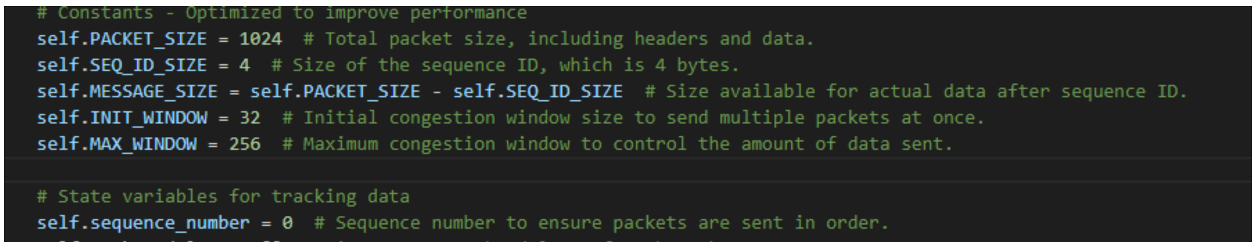
Overview

custom_protocol.py contains a class *OptimizedAdaptiveSender* and a function *main()*.

The OptimizedAdaptiveSender class uses its own congestion control protocol in order to achieve a targeted average throughput, delay, and jitter. This protocol is expected to be able to respond to the fluctuations of the network and enhance the transfer of data. We check with the idea of protocol like BBR, quick Recovery to see it might be helpful or not. The key features of this custom protocol implementation are: Adaptive Congestion Window: Congestion control used in the class has an initial congestion window (INIT_WINDOW) of 32 packets and a maximum congestion window (MAX_WINDOW) of 256 packets. It dynamically adjusts the congestion window size based on the current network conditions, employing three distinct modes of operation: The parameter is set as “AGGRESSIVE” for high growth, “NORMAL” for moderate growth and “RECOVERY” for tackling losses. Enhanced RTT Tracking: The class has a deque of the last 20 values of the RTT samples in which we store the most recent samples at both front and end of the list. It employs Haworth’s Exponential Weighted Moving average (EWMA) to compute the average round trip time (SRTT) and the variation of round trip time (RTTVAR). These values are then used in order to set the timeout for retransmissions following the Jacobson Karels algorithm. Bandwidth Estimation: The class that estimates throughput is available bandwidth approximated from the recent bandwidth samples using EWMA. This enables the protocol to be more appropriate in tuning the congestion window since it is in a position to make better judgments with regard to the existent network capacity. Adaptive Pacing: The class also has a fine pacing control system that determines the rate of transmission of packets in the class. It records the time of the last packet which has been sent and also incorporates a pacing factor of 0.00005s so as to avoid congestion and jitter. Implementing these techniques enables the OptimizedAdaptiveSender to achieve better throughput, delay, and jitter with respect to the fundamental Stop-and-Wait, TCP Tahoe, and TCP Reno. By managing the congestion window, RTT/Bandwidth and by using adaptive pacing, the custom protocol has higher potential in sensing changes in the network and subsequently offers a more stable and efficient flow of data.

Class OptimizedAdaptiveSender:

Fields:



```
# Constants - Optimized to improve performance
self.PACKET_SIZE = 1024  # Total packet size, including headers and data.
self.SEQ_ID_SIZE = 4  # Size of the sequence ID, which is 4 bytes.
self.MESSAGE_SIZE = self.PACKET_SIZE - self.SEQ_ID_SIZE  # Size available for actual data after sequence ID.
self.INIT_WINDOW = 32  # Initial congestion window size to send multiple packets at once.
self.MAX_WINDOW = 256  # Maximum congestion window to control the amount of data sent.

# State variables for tracking data
self.sequence_number = 0  # Sequence number to ensure packets are sent in order.
```

Methods:

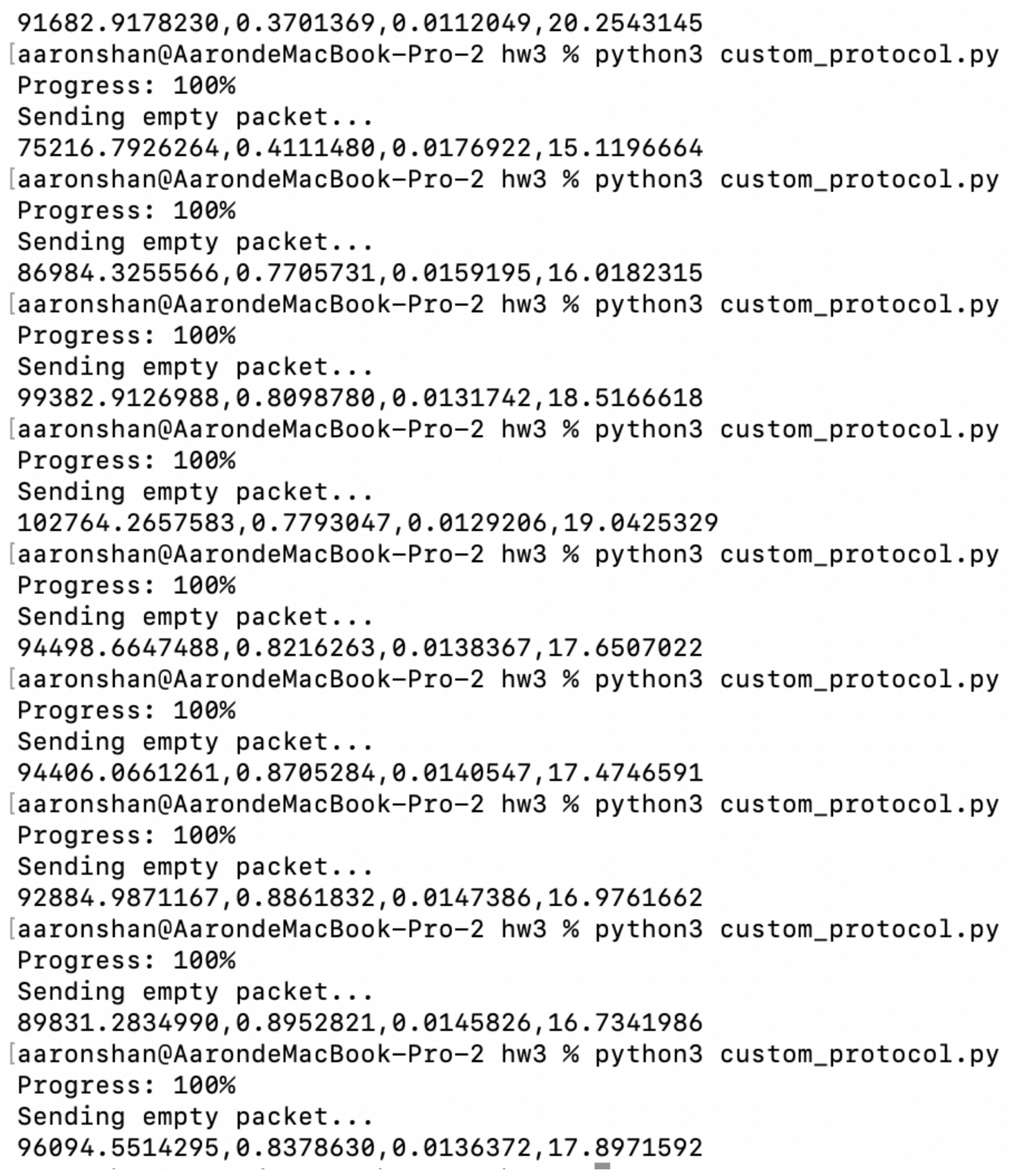
- Constructor: create socket and initialize fields.
- *create_packet(self, data)*: adding sequence number to data, return data with sequence number
- *update_rtt(self, rtt):* Calculate and update RTT using weighted moving average
- *estimate_bandwidth(self, bytes_acked, time_taken):* Estimate the bandwidth using exponential moving average
- *get_timeout(self):* Calculate the timeout value using the Jacobson/Karels algorithm
- *send_file(self, file_path):* divide the file in file path into packets and use Optimized Adaptive protocol to send them to the receiver. This will also send the final packet and return throughout, pre packet delay, and jitter
- *close(self):* close the socket

Function main():

- Call the methods in *OptimizedAdaptiveSender* to get throughput, pre-packet delay, and jitter and print them. Also calculate the matrix.

Details of protocol:

Most parts of protocol is in



```
91682.9178230,0.3701369,0.0112049,20.2543145
[aaronshan@AarondeMacBook-Pro-2 hw3 % python3 custom_protocol.py
Progress: 100%
Sending empty packet...
75216.7926264,0.4111480,0.0176922,15.1196664
[aaronshan@AarondeMacBook-Pro-2 hw3 % python3 custom_protocol.py
Progress: 100%
Sending empty packet...
86984.3255566,0.7705731,0.0159195,16.0182315
[aaronshan@AarondeMacBook-Pro-2 hw3 % python3 custom_protocol.py
Progress: 100%
Sending empty packet...
99382.9126988,0.8098780,0.0131742,18.5166618
[aaronshan@AarondeMacBook-Pro-2 hw3 % python3 custom_protocol.py
Progress: 100%
Sending empty packet...
102764.2657583,0.7793047,0.0129206,19.0425329
[aaronshan@AarondeMacBook-Pro-2 hw3 % python3 custom_protocol.py
Progress: 100%
Sending empty packet...
94498.6647488,0.8216263,0.0138367,17.6507022
[aaronshan@AarondeMacBook-Pro-2 hw3 % python3 custom_protocol.py
Progress: 100%
Sending empty packet...
94406.0661261,0.8705284,0.0140547,17.4746591
[aaronshan@AarondeMacBook-Pro-2 hw3 % python3 custom_protocol.py
Progress: 100%
Sending empty packet...
92884.9871167,0.8861832,0.0147386,16.9761662
[aaronshan@AarondeMacBook-Pro-2 hw3 % python3 custom_protocol.py
Progress: 100%
Sending empty packet...
89831.2834990,0.8952821,0.0145826,16.7341986
[aaronshan@AarondeMacBook-Pro-2 hw3 % python3 custom_protocol.py
Progress: 100%
Sending empty packet...
96094.5514295,0.8378630,0.0136372,17.8971592
```

	Average (in 10 trials)	Standard Deviation (in 10 trials)
Throughput	92374.6767383	7142.7651243
Pre-packet delay	0.7461523	0.1825148
Performance metric	17.5684292	1.4091132