Leif Olson
Saul Duran
ECS-152A

Project 2.2 - Congestion Control

Project Files Submitted

- **Congestion_Control**
    - sender_stop_and_wait_[Leif_Good-Olson]_[921489807]_[Saul_Duran]_[922748576].py
    - sender_fixed_sliding_window_[Leif_Good-Olson]_[921489807]_[Saul_Duran]_[922748576].py
    - sender_reno_[Leif_Good-Olson]_[921489807]_[Saul_Duran]_[922748576].py

Implementation Details & Tables

- **Stop & Wait**
    - The implementation of *stop and wait* is, as to be expected, simpler than the rest. I used my code for *udp_client.py* from project 1 as a base, then reviewed how *receiver.py* expected data to be transmitted to it. We first define our packet size, sequence ID size, server address, and timeout values to match *receiver.py*. We then define a function that creates a packet based on the next sequence ID and the chunk of data passed to it, converting those int values to a bytestring. The second function defined is the main function, which opens a UDP socket, along with the file that is passed to it. It reads this file in chunks, creating a packet for each chunk. When each packet is made, the *send_file* function attempts to send it to the predefined server address, saving the time at which that packet is sent and the time at which an acknowledgement of the packet is received. If an acknowledgement is not received, the packet is sent again. Once it is successful, the next packet is sent. Nearly done now! When all packets are exhausted (no more data in the file), we calculate the total transmission time, throughput, average packet delay, and the performance metric, printing each. Finally, the termination signal is sent to the server and we wait for the final acknowledgement, closing the socket when it is received.

Test Number	Throughput	Average Packet Delay	Performance Metric
1	2306029.655	0.0004387	691.8092037
2	2299327.025	0.00044	689.7984155
3	2308500.949	0.0004382	692.5505915

4	2330079.924	0.0004341	699.0242811
5	2305129.587	0.0004389	691.5391832
6	2185223.863	0.0004629	655.567483
7	2182949.108	0.0004634	654.8850567
8	2324701.64	0.0004352	697.4107967
9	2286198.195	0.0004425	685.8597684
10	2293300.189	0.0004411	687.9903653
Average	***2282144.014***	***0.0004435***	***684.6435145***
Standard Dev	***53304.00134***	***0.00001065061292***	***15.99119294***

- **Fixed Sliding Window**
    - The implementation of *fixed sliding window* is built upon *stop & wait*, though it becomes a bit more complicated. Instead of sending one packet at a time and waiting for acknowledgement, we read 100 chunks of data from the file and store them in a list. We will then iterate through this list of chunks, creating packets with them and sending them to the server address. Because we are sending so many packets at once, it becomes a little more difficult to track the delay of each one. Before we could simply check on each send/receive. Now, we will store packet send times in a dictionary, with the key being their sequence ID. Upon receiving acknowledgements for the packets, we will calculate the delay for each and store those values in a single variable so that we can average the delays. If any packets are not acknowledged, we will simply send them again until they are. This process is repeated for each window (100 packets each). Once all packets are depleted and successfully acknowledged, we can calculate our throughput, average packet delay, and performance metric, printing all 3 as before and sending the same termination signal. One thing I regret in this implementation is storing so many values - I don't believe it is particularly efficient, but I was indecisive about how to best implement it and consulted ChatGPT on how I should best store/calculate packet delay times. That is essentially what it recommended.

Test Number	Throughput	Average Packet Delay	Performance Metric
1	7049578.565	0.0071816	2114.878597
2	6889006.237	0.0073695	2066.70703
3	6975956.848	0.0072203	2092.792109
4	7024130.249	0.0071986	2107.244114

5	6479236.197	0.0077688	1943.776297
6	6579685.102	0.0077165	1973.910932
7	6713081.401	0.0075848	2013.92973
8	6842876.381	0.0073629	2052.868068
9	6613706.302	0.0076027	1984.117212
10	7001702.579	0.0072314	2100.515836
Average	***6816895.986***	***0.00742371***	***2045.073992***
Standard Dev	***206764.111***	***0.0002253608487***	***62.02907702***

- **TCP Reno**
  - The implementation of *TCP Reno* builds further upon *fixed sliding window* and *stop & wait*. In fact, most of the code is identical to that of *fixed sliding window*, however, we have added a few new parameters: *cwnd* (congestion window value), *ssthresh* (slow start threshold value), and *dup_acks* (tracks number of duplicate acknowledgements). Using the same *create_packet* function and a modified *send_file* function, we open the socket and initialize our variables, opening the file and saving chunks of its data to our *file_data* list. We will then send as many packets as our congestion window size will allow, tracking the time at which the transmission begins so that we can later calculate overall transmission time and therefore throughput. We also track packet delay in a similar manner as before. This is where *TCP Reno* diverges, however. When receiving acknowledgements for packets, we will adjust our parameter values so that we can increase the rate at which packets are sent, or slow them down. If our congestion window remains smaller than our slow start threshold, then we will incrementally increase our congestion window, resulting in an exponential increase in output. Otherwise, to avoid congestion, we will increment the congestion window by a much smaller value (1/cwnd). Now, if duplicate acknowledgements are received (3 in this case), we begin fast retransmit, sending the packets again and reducing our slow start threshold. If we time out while attempting to send a packet, we begin slow start, drastically decreasing the slow start threshold and resetting the congestion window so that we are slowly outputting packets. This helps strike a balance between throughput and error checking.

Test Number	Throughput	Average Packet Delay	Performance Metric
1	5285745.197	0.0001955	1585.723696

2	5442550.251	0.0001891	1632.765208
3	5301564.833	0.0001472	1590.469553
4	5310313.898	0.0001485	1593.094273
5	5577152.897	0.000162	1673.145983
6	5475456.757	0.000163	1642.637141
7	5520782.879	0.0001558	1656.234973
8	5129653.952	0.0001522	1538.896292
9	5336689.263	0.0001638	1601.006894
10	5510038.282	0.0001576	1653.011595
Average	***5388994.821***	***0.00016347***	***1616.698561***
Standard Dev	***138345.6446***	***0.00001631094043***	***41.50369423***