EEC 173A: Computer Networks
Project 3: Congestion Control

Professor Zubair

Adrian Rivera, SID: 921435699 Calvin Lau, SID: 919709340

December 1st, 2024

Python Scripts:

1. sender_stop_and_wait.py
2. sender_fixed_sliding_window.py
3. sender_tahoe.py
4. sender_reno.py

Protocol Analysis:

sender_stop_and_wait.py Analysis:

We ran our sender_stop_and_wait protocol 10 times and noted the throughput (in bytes per second), the average packet delay (in seconds), the average jitter (in seconds), and the performance metric in Table 1 below.

Our sender_stop_and_wait protocol constructs a message for each data packet. The packet is composed of a sequence ID followed by the corresponding chunk of the data. The sequence ID (seq_id) is updated after each successful packet transmission. If all data has been sent, an "empty" packet is sent to signal completion, marked by the length of the data. The sender sends the message through UDP using sendto() with the constructed sequence ID and data chunk. The packet is timestamped to measure delay and jitter. To handle ACKs (stop and wait), the sender waits for an ACK from the receiver. If an ACK is not received within the timeout period, the sender will retransmit the packet. The sender checks if the received ACK matches the expected sequence ID, confirming the successful receipt of the corresponding data packet. If the ACK is received, the sender proceeds to send the next packet. If the final ACK is received, the loop ends.

Run	**Throughput** (bytes/sec)	**Avg. Packet Delay (sec)**	**Avg. Jitter (sec)**	**Performance Metric**
1	5345.6412452	0.1903299	0.1588762	5.3672129

2	5206.6279695	0.1953683	0.164575	5.2231189
3	5483.0560553	0.1855099	0.1495944	5.5292191
4	5307.812793	0.1915601	0.1626062	5.3219986
5	5228.3279423	0.1944882	0.1671788	5.2343549
6	5416.3750461	0.1877329	0.1544804	5.4503431
7	5424.2218335	0.1874509	0.155009	5.4553289
8	5230.8666399	0.194397	0.1611365	5.2589689
9	5272.5274728	0.1928483	0.1628926	5.2894916
10	5307.6300047	0.1916094	0.1584413	5.3370715
Average	5322.3087002	0.19112949	0.15947904	5.34671084
Standard Deviation	88.9374582	0.00317148	0.0050354	0.0978991

Table 1: sender_stop_and_wait.py results after 10 runs

sender_fixed_sliding_window.py Analysis:

We ran our sender_fixed_sliding_window protocol 10 times and noted the throughput (in bytes per second), the average packet delay (in seconds), the average jitter (in seconds), and the performance metric in Table 2 below.

For the sliding window protocol, we observed that we were consistently reaching timeout on the first window of packets. We attempted to remedy this by trying to reduce the number of packets sent within the first window and by increasing the timeout. The following results were calculated using only 5 runs each, so the results could be related to variance.

We tried limiting the size of the first window to be half, this resulted in a noticeable improvement in performance metric (16.8479130). In this scenario, the first window of packets never reached timeout. In 3 out of our 5 runs, a different window reached timeout. However, on average, we were retransmitting less packets with the limited first window compared to always retranmitting the first 100. Doubling the timeout, from 1 second to 2 seconds, still resulted in the first window of 100 packets reaching timeout and did not lead to a better performance metric (14.6799800). Combining the limited first window and a larger timeout resulted in a small improvement in performance metric (15.4696282), which rarely dropped packets (once in the 5 runs). You can find these changes commented out since they are not required in the project details.

Run	Throughput (byte/sec)	Avg. Packet Delay (sec)	Avg. Jitter (sec)	Performance Metric
1	88354.4485594	1.0893249	0.0171012	15.4174025
2	87391.7226513	1.1227075	0.0183093	14.9134554
3	87906.2561629	1.1021004	0.0165704	15.5513778
4	76725.5412123	1.1056916	0.0168570	14.3283319
5	89187.5062169	1.0790099	0.0165616	15.6982364
6	94093.8499874	0.9958360	0.0151583	16.8097769
7	91256.3260672	0.9962514	0.0155562	16.3569377
8	83887.8007453	1.0532286	0.0171278	14.9868215
9	81944.0046874	1.1519721	0.0185446	14.2812653
10	78065.9739160	1.1840798	0.0187589	13.8130362
Average	85881.3430206	1.0880202	0.0170545	15.2156641
Standard Deviation	5330.5698038	0.0576574	0.0011430	0.8945112

Table 2: sender_fixed_sliding_window.py results after 10 runs

sender_tahoe.py Analysis:

We ran our sender_tahoe protocol 10 times and noted the throughput (in bytes per second), the average packet delay (in seconds), the average jitter (in seconds), and the performance metric in Table 3 below.

In our implementation of TCP Tahoe, slow start starts with a small cwnd, initially set to 1 MSS. During this phase, the cwnd grows exponentially, doubling in size for every RTT. This allows the sender to quickly discover the available bandwidth. Once the cwnd exceeds sshThresh, the growth becomes linear, incrementing by 1 MSS per RTT. This prevents congestion as the network approaches capacity. To handle new ACKs, when a new ACK is received, the cwnd increases exponentially in slow start and linearly in congestion avoidance. The sender also resets the duplicate ACK counter, as successful delivery is confirmed. If duplicate ACKs are received, the counter is increment. Upon reaching the duplicate ACK threshold (DUPE_ACK_THRESHOLD), the sender does a fast retransmit. When a timeout occurs, the sender halves the sshThresh, resets the cwnd to 1 MSS, and re-enters the slow start phase to probe the available bandwidth. Our sender also uses a sliding window mechanism to manage packets in flight. The cwnd dynamically adjusts based on ACKs and the current phase (slow start or congestion avoidance).

Our sender constructs packets with a sequence ID and data chunk and sends them over UDP. Unacknowledged packets are stored in_flight for retransmission if needed. After all packets are sent and ACK, the sender transmits a “FINACK” message to signal completion and outputs the performance statistics.

Run	**Throughput (byte/sec)**	**Avg. Packet Delay (sec)**	**Avg. Jitter (sec)**	**Performance Metric**
1	92504.956767	0.697917	0.0138203	17.6325098
2	15126.5727507	0.1207147	0.0335865	11.1172385
3	14847.6359596	0.1225839	0.0356441	10.8164166
4	90754.661178	0.7122007	0.013294	17.7209262
5	87330.5165297	0.7333351	0.0149802	16.4994503
6	91111.5329393	0.7070321	0.0134517	17.6766702
7	94984.5677332	0.6796313	0.0130659	18.3290677
8	16034.4423017	0.1177563	0.0263389	12.1938023
9	88082.1605641	0.7313065	0.0133225	17.4082605
10	103338.2717085	0.6217015	0.0111775	20.5671569
Average	69411.53184	0.52441791	0.01886816	15.9961499
Standard Deviation	37575.99844	0.2805822014	0.009300220372	3.369007674

Table 3: sender_tahoe.py results after 10 runs

sender_reno.py Analysis:

We ran our tcp_reno protocol 10 times and noted the throughput (in bytes per second), the average packet delay (in seconds), the average jitter (in seconds), and the performance metric in Table 4 below.

For our implementation of TCP Reno, we noticed that occasionally timeouts would occur for tens to hundreds of packets in a row causing poor performance. To remedy this, we doubled the timeout every time a packet timed out and reset it once the packet successfully returned an ACK. The consistent timeouts stopped and we got a much more consistent performance, which led to higher averages across the board.

Run	**Throughput (byte/sec)**	**Avg. Packet Delay (sec)**	**Avg. Jitter (sec)**	**Performance Metric**

1	110656.0764153	0.7104546	0.016079	17.3975028
2	170139.4480282	0.4762198	0.0094889	27.7206176
3	115715.563405	0.5554845	0.014227	18.7444522
4	118884.3371673	0.5709847	0.0209342	16.8054164
5	124283.0530923	0.6481966	0.0125829	20.4989938
6	104835.6029599	0.9192530	0.0217302	15.1724875
7	162100.0070585	0.4835024	0.0139498	23.5440129
8	139963.9411825	0.5442306	0.0168925	20.0631808
9	106343.3305493	0.7486438	0.0159396	17.0148917
10	127903.3130883	0.5299846	0.0130151	20.6246839
Average	128082.4672947	0.6186955	0.0154839	19.7586240
Standard Deviation	21531.0275715	0.1322023	0.0035447	3.5071222

Table 4: sender_reno.py results after 10 runs