ECS 152A: HW3 (Team Name: Nghetworking)

Nghi Dao (921147615), Bian Lee (920763430)

November 2024

Final submission: `sender_stop_and_wait.py`, `sender_fixed_sliding_window.py`, `sender_reno.py`, `sender_tahoe.py`, `proj3_NghiDao_921147615_BianLee_920763430_sender_custom.py`

Stop and Wait

Trial	Throughput (B/s)	Per-Packet Delay (s)	Jitter (s)	Metric
Trial 1	7314.0018657	0.1399498	0.0659052	7.9650685
Trial 2	7509.1577474	0.1363108	0.0590906	8.3121709
Trial 3	7317.5813929	0.1398805	0.0658115	7.9704184
Average	7380.2470020	0.1387137	0.0636024	8.0825526
Standard Deviation	91.1653752	0.0016993	0.0031906	0.1623793

- To implement a stop and wait protocol to transmit a file, we simply send 1020 bytes of the file along with a 4 byte sequence number at the beginning of the packet and wait until the packet is acknowledged. Before sending the next packet, the protocol checks to see if the previous packet has been acknowledged by comparing the last acknowledgement (`last_ack`) with expected (`expected[i]`) – if it is the case, it moves to the next packet.

Fixed Sliding Window

Trial	Throughput (B/s)	Per-Packet Delay (s)	Jitter (s)	Metric
Trial 1	78133.5354539	0.6062850	0.0243953	13.2320095
Trial 2	77232.4932655	0.5838673	0.0241912	13.2271543
Trial 3	72910.1242406	0.6145027	0.0246858	12.6437904
Average	76092.0509867	0.6015517	0.0244241	13.0343181
Standard Deviation	2279.8337102	0.0129469	0.0002029	0.2761519

- To implement a fixed sliding window protocol to transmit a file, we first choose a window size of 100 MSS. Since the acknowledgment is cumulative, the window will always be the 100 packets after the last acknowledged

packet (in other words, 100 packets can be sent without waiting for acknowledgement). Each packet is of size 1024 bytes, with 1020 bytes allocated for message and 4 bytes for sequence ID. By using the `ack` dictionary, we first mark them unacknowledged (by setting them `False`). Upon receiving an acknowledgement after the packets have been sent, the `ack_id` is then extracted, and all packets with sequence IDs less than this `ack_id` value is then marked acknowledged (by setting them `True`). This approach is valid since acknowledgements are cumulative. When all packets in the window (by checking all values in `ack`) are acknowledged, we shift the window. When there is a timeout, we retransmit all packets in the current window that are not acknowledged.

TCP Tahoe

Trial	**Throughput (B/s)**	**Per-Packet Delay (s)**	**Jitter (s)**	**Metric**
Trial 1	88959.5404415	0.6148084	0.0239217	14.3774794
Trial 2	84371.8865737	0.4012531	0.0236772	14.6544201
Trial 3	93373.0459800	0.5724175	0.0223338	15.2124103
Average	88901.4909984	0.5294930	0.0233109	14.7481033
Standard Deviation	3674.9371848	0.0923159	0.0006981	0.3472365

- To implement TCP Tahoe, we initialize the congestion window size (`cwnd`) to be 1 MSS, and slow start threshold (`ssthresh`) at 64. It uses dynamic window sizing mechanism in which the the window size is determined based on the current `cwnd` value. win_start and win_end describes the range of messages that need to be sent within the window.
- `ack_dict` is used to keep track of the count of acknowledgements for each sequence ID. After the packets in the window are sent, the protocol listens for acknowledgements where it extracts the `ack_id` and increments the corresponding count in the dictionary. We implemented a logic to check for duplicate ACKs by detecting triple duplicate ACKs in which case the `on_duplicate()` is called, resetting the `cwnd` to 1 and updates `ssthresh` to half of `cwnd`. The same logic is also applied when timeout occurs. If the ACKs are correctly received (no timeout or duplicate ACKs cases), `on_success` is called, which doubles the `cwnd` if the current `cwnd` is below the threshold (exponential growth). If not, the `cwnd` increments by 1, representing linear growth.

TCP Reno

Trial	**Throughput (B/s)**	**Per-Packet Delay (s)**	**Jitter (s)**	**Metric**
Trial 1	84194.8942113	0.4088778	0.0223955	14.8412555
Trial 2	81008.9659068	0.4311589	0.0235109	14.2097038
Trial 3	77434.8113472	0.3005811	0.0223137	14.8865355
Average	80879.5571551	0.3802059	0.0227400	14.6458316
Standard Deviation	2761.3088746	0.0570333	0.0005461	0.3089425

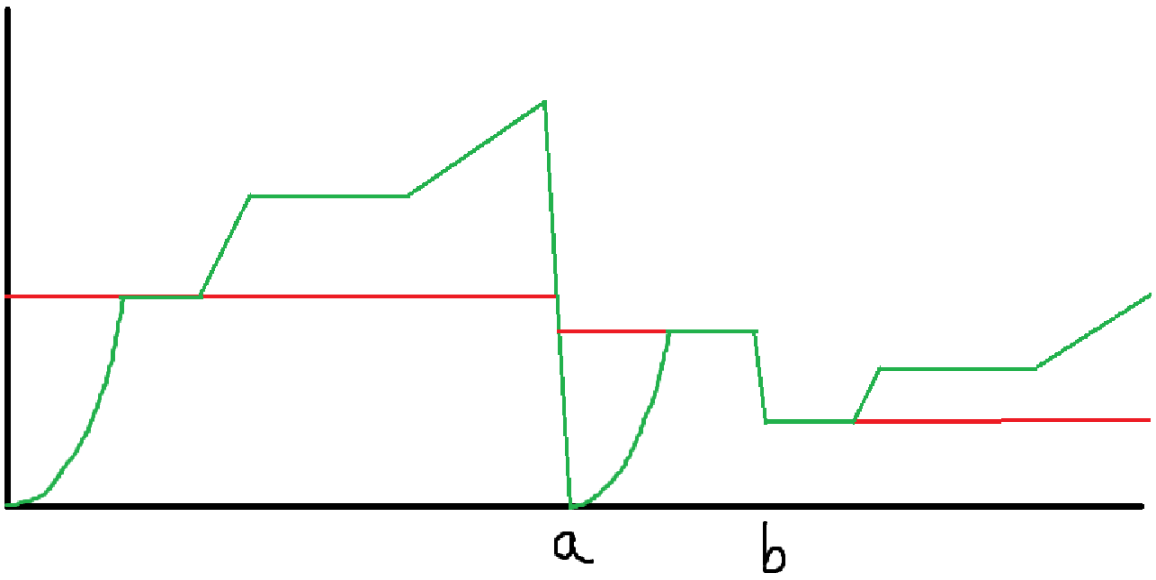
- Similar to TCP Reno, we initialize the congestion window size (`cwnd`) to be 1 MSS, and slow start threshold (`ssthresh`) at 64. It also uses dynamic window sizing, and same data structures like `ack_dict` to keep track

of count of ACKs, as well as same logic to check for triple duplicate ACKs. The difference arises with how the `cwnd` is sized in the case of the triple duplicate – instead of dropping it down to 1 like how it was done in Tahoe, we halve it (which also becomes the new `ssthresh`). This represents the fast recovery mechanism. If the packet loss is detected through a case of a timeout, the change in `ssthresh` and `cwnd` are same as that of Tahoe.

Custom

Trial	**Throughput (B/s)**	**Per-Packet Delay (s)**	**Jitter (s)**	**Metric**
Trial 1	82013.7472498	0.2790602	0.0202714	16.0011921
Trial 2	83565.7152654	0.3049463	0.0205101	15.8556215
Trial 3	88644.5082434	0.3503004	0.0198371	16.1892584
Average	84741.3235862	0.3114356	0.0202062	16.0153573
Standard Deviation	2831.7590565	0.0294434	0.0002786	0.1365745

- Our custom protocol builds on top of Reno. The window size will start off at 1 and doubles for each successful cycle until it reaches ssthresh. Once it reaches ssthresh, the window size will remain the same for 5 cycles if the acknowledgments are received with no timeout or duplicates. After this 5 cycles, the window size will increase by 2 until it reaches 1.5 ssthresh (assuming no duplicates or timeout). At this point, the window size will remain the same for another 10 cycles before increasing linearly by 1. An illustration for the protocol is shown below:



- The red line denotes ssthresh and at point (a) there is a timeout and at point (b) there is a duplicate acknowledgment. Similar to Reno and Tahoe, the initial increase in the window size is exponential before it stays the same for 5 cycles. The idea behind this is that we want to make sure that after an exponential increases, we are still able to transmit packets without any timeout or duplicates before we start increasing the

window size linearly. After we reach 1.5 ssthresh, we wait 10 clock cycles to make sure there is little congestion before we start increasing the window sizse once again. When a timeout occures at (a), the behviour is like Reno, where the window size resets to 1 and the ssthresh equals half of the last window size. At (b), a duplicate occurs and once again, the behavior is the same as Reno, where the window size and ssthresh is set to half of the previous window size.