ECS 152A: Computer Networks Project 3

Congestion Control

Professor Zubair Shafiq

Krystal Chau, SID: 920918540 Jacob Feenstra, SID: 921423591

December 7, 2023

Name of custom protocol: TCP Vegas Lite

1 File Submissions

1. `sender_stop_and_wait.py`
2. `sender_fixed_sliding_window.py`
3. `sender_tahoe.py`
4. `sender_reno.py`
5. `sender_custom.py`

2 sender_stop_and_wait.py

Stop and wait is inherently a protocol that does exactly as it is named. The sender should try and send a packet and wait until it's `ACK` has been received before moving on to the next packet. In the case that the `ACK` of a packet times out (starts at 1 second) or take too long to arrive, the protocol will resend the packet while also doubling it's timeout duration and continues to wait for the packet's `ACK`.

Since this protocol stops and waits for a single packet at a time (does not implement Cumulative Acknowledgements), there no use for storing any kind of identifier for previous messages. Our implementation simply constructs a packet, sends it, waits to receive it, and sends the empty message once the sequence id has exceeded the length of total data to finish the loop.

Run	Throughput	Average Packet Delay	Performance Metric
1	10093.71	0.1	100144.15
2	10075.02	0.1	99814.64
3	10096.34	0.1	100187.67
4	10104.09	0.1	100337.38
5	10102.18	0.1	100277.88
6	10106.45	0.1	100373.3
7	10104.9	0.1	100336.3
8	10103.79	0.1	100320.51
9	10099.61	0.1	100240.19
10	10091.86	0.1	100103.51
Average	10097.79	0.1	100213.55
Standard Deviation	9.41	1.46e-17	166.31

Table 1: sender_stop_and_wait.py

Run	Throughput	Average Packet Delay	Performance Metric
1	79161.97	1.25	63304.78
2	80529.37	1.21	66553.08
3	82504.48	1.19	69312.5
4	82087.59	1.21	67642.62
5	82006.15	1.21	67941.01
6	82873.6	1.18	70182.29
7	82669.06	1.16	71191.29
8	87429.97	1.14	76466.69
9	85932.66	1.16	74231.43
10	81811.56	1.22	67080.23
Average	82700.64	1.19	69390.59
Standard Deviation	2398.31	0.03	3843.66

Table 2: sender_fixed_sliding_window.py

3 sender_fixed_sliding_window.py

The Sliding Window protocol is supposed to consistently have a window of 100 packets in transit at all times. This basically means that as the `ACK` of a packet is received, the window will "slide" and next packet at the end of the window size.

This concept can be thought as managing resources. The total number of resources we have is the window size where the resources are used up if a packet is in transit and regained when the `ACK` is received.

An issue that we can encounter in Sliding Window that we did not face in Stop and Wait is possible timeouts within the window. To deal with this, we save the contents of the message to send and whether or not the `ACK` of a message has been received or not. We do this so that we can simply check if a message's `ACK` has not been received and re-transmits it easily. We can also encounter duplicate acknowledgements, however, these can be dealt with by checking if the `ack_id` received the same as the previous ACK's id and re-transmitting as needed. This re-transmit logic is the same as for timeout.

An issue we ran into here was that, in the case of duplicate acknowledgements, the `ack_id`'s received would not correspond with the message it was for. The work around was that when the re-transmitted acknowledgement finally does come, it will contain the `ack_id` of the next needed message id. This means that when we receive an `ACK` the is larger, all message ACKs that are less than that have been received. This way we can also avoid re-transmitting packets that were hidden under the duplicate `ACK`.

4 sender_tahoe.py

We implemented TCP Tahoe on top of the jumping window protocol since all that is required for Tahoe is the resizing of the congestion window (`cwnd`). Jumping window is to put simply, the easier version of Sliding window. It does everything Sliding window does, but instead of sliding with each received `ACK`, we only move on once we receive the acknowledgements of all packets in the window. With this we only really had to add a few features: starting from a `cwnd` of 1, incrementing the `cwnd` according to the slow-start threshold (`ssthresh`), and setting `cwnd` and `ssthresh` to accordingly in case of timeouts or duplicate acknowledgements.

To increment the `cwnd`, we need to check it against the `ssthresh`:

1. if `cwnd` is less than `ssthresh`, `cwnd` should double
2. if `cwnd` is more than or equal to `ssthresh`, `cwnd` will only increment by 1

Since we had already written timeout and duplicate acknowledgement logic for Sliding window, we can reuse that same logic and simply add that `ssthresh` will be set to `cwnd / 2` where it must be at least 1 and then `cwnd` will be set to 1.

Run	Throughput	Average Packet Delay	Performance Metric
1	19242.88	0.08	233589.7
2	15705.21	0.07	229530.79
3	13249.83	0.05	264975.17
4	14959.82	0.06	255976.11
5	15073.13	0.07	223742.92
6	14082.21	0.06	248884.92
7	17098.56	0.08	205359.54
8	15376.06	0.07	210729.23
9	14313.89	0.06	227908.74
10	14871.93	0.06	248577.35
Average	15397.35	0.07	234927.45
Standard Deviation	1695.27	0.01	19413.99

Table 3: sender_tahoe.py

Run	Throughput	Average Packet Delay	Performance Metric
1	16293.94	0.07	218980.35
2	20431.21	0.08	243954.93
3	18658.08	0.01	178057.23
4	18030.07	0.08	217745.39
5	15265.85	0.07	205117.87
6	15287.31	0.06	243024.58
7	27454.06	0.13	215090.54
8	20840.92	0.11	190165.34
9	18221.81	0.09	209073.04
10	16497.97	0.08	216776.23
Average	18698.12	0.08	213798.55
Standard Deviation	3635.79	0.03	20412.93

Table 4: Caption

5 sender_reno.py

The implementation of TCP Reno is simply adding on to TCP Tahoe. TCP Reno's logic for resetting `cwnd` and `ssthresh` in the case of duplicate acknowledgements differs from TCP Tahoe in that `cwnd = ssthresh` instead of going back to 1.

6 sender_custom.py

TCP Vegas is a congestion control algorithm that is delay-based instead of loss-based. TCP Tahoe and TCP Reno alter their congestion window size based on a timeout or triple acknowledgement: they are reactionary algorithms. TCP Vegas is more dynamic, modifying the congestion window size based on RTT (and similar parameters)

There are three primary distinctions in the TCP Vegas algorithm:

1. Timely decision to retransmit a dropped segment - Record the system clock each time a segment is sent, read the clock again when the ACK corresponding to the segment arrives. Compute the difference. This furthers falls into two scenarios.

   (a) Duplicate ACK received. Vegas checks to see if difference between current time and timestamp recorded (RTT estimate) is greater than timeout value (RTO). If it is, retransmit without having to wait for 3xACKS.

   (b) When non-duplicate ACK is received, and it is the first or second one after a retransmission, Vegas again makes RTT/RTO comparison. If it is, retransmit. Note 3xACK logic is still applied, but the above algorithm is applied first.

2. 2) Give TCP the ability to anticipate congestion (and adjust transmission rate accordingly). Has a few conditions:
   (a) BaseRTT: the RTT of a segment when the connection is not congested. Vegas sets BaseRTT to the minimum of all measured round trip times (commonly this is the first segment sent by the connection, before the router queues increased).
   (b) $Expected \approx WindowSize/BaseRTT$
   (c) Actual Sending Rate is calculated by:
      i. Recording the sending time for a distinguished segment
      ii. Recording how many bytes are transmitted between this time sent and distinguished ACK is received.
      iii. Computing RTT for distinguished segment when ACK arrives
      iv. Dividing number of bytes transmitted by this sample RTT
      v. ASR is calculated once per RTT! So continuously updated
   (d) Diff: Compare Actual to Expected, where Diff = Expected − Actual. Then define two thresholds $\alpha < \beta$; roughly corresponds to too little and too much extra data. $Diff < \alpha$, Vegas increases cwnd linearly during the next RTT. When $Diff > \beta$, Vegas decreases cwnd linearly during next RTT. Vegas otherwise leaves cwnd unchanged if $\alpha < Diff < \beta$. Goal: keep between α and β extra bytes in the network.
   (e) α and β is described in terms of buffers, rather than extra bytes. W.r.t. BaseRTT and segment size, should be at-least n extra buffers (mapped to α), but no mare than k extra buffers (mapped to β). α is 2, and β is 4 (KB/s) during linear increase/decrease.
3. Modifies TCP's slow-start mechanism to avoid packet loss, while still trying to find available
   (a) Slow-start is very similar to TCP Tahoe/Reno, with some modifications
   (b) TCP Vegas allows slow-start only every other RTT.
   (c) In between, the cwnd is fixed to compare Expected and Actual (much like 2). If Actual falls below Expected by a certain amount— the γ threshold— Vegas changes from slow-start to congestion avoidance mode.

Beyond this TCP, Vegas has a similar paradigm to Tahoe and Reno. It begins in slow-start, then moves to congestion avoidance once an event is triggered. Then, it stays in congestion avoidance. We ran into problems into implementation that we believe explicitly affects our average packet delay, and thus the performance metric. The second component of TCP Vegas (the two conditions for fast retransmit given RTO) does not work. Our time() evaluations don't give us legitimate values. On average, the calculated RTT was around 12 seconds for a given packet. We spent some time trying to puzzle this out, but to no avail. TCP Vegas should have 40%-70% better performance than Tahoe/Reno. The code which implements Step 2 has been commented out in triple-quotations in `sender_custom.py`, because the RTT's are highly inaccurate.

Name: TCP VegasLite

Run	Throughput	Average Packet Delay	Performance Metric
1	77958.17	0.39	197626.43
2	77554.77	0.41	191213.57
3	81328.37	0.36	228383.06
4	78841.21	0.39	199963.91
5	79602.9	0.37	215912.53
6	82495.03	0.39	212131.72
7	65039.42	0.44	148783.28
8	78400.47	0.4	196070.25
9	80474.7	0.36	222217.71
10	80906.56	0.39	208288.84
Average	78260.16	0.39	202059.13
Standard Deviation	4907.96	0.02	22229.48

Table 5: sender_custom.py