sheen

CS244 Congestion Control Contest Report
Spring 2017

Authors:

Kate Stowell <kstowell@stanford.edu>
Travis Lanham <tlanham@stanford.edu>

Results:

```
Average capacity: 5.04 Mbits/s
Average throughput: 3.45 Mbits/s (68.5% utilization)
95th percentile per-packet queueing delay: 56 ms
95th percentile signal delay: 108 ms
```

Power = 31.94

Code:

https://github.com/lanhamt/sheen

Note: each warmup exercise is on a different branch for convenience - there should be four branches: warmup_a, warmup_b, warmup_c, and master which has our contest algorithm.

Warmup A:

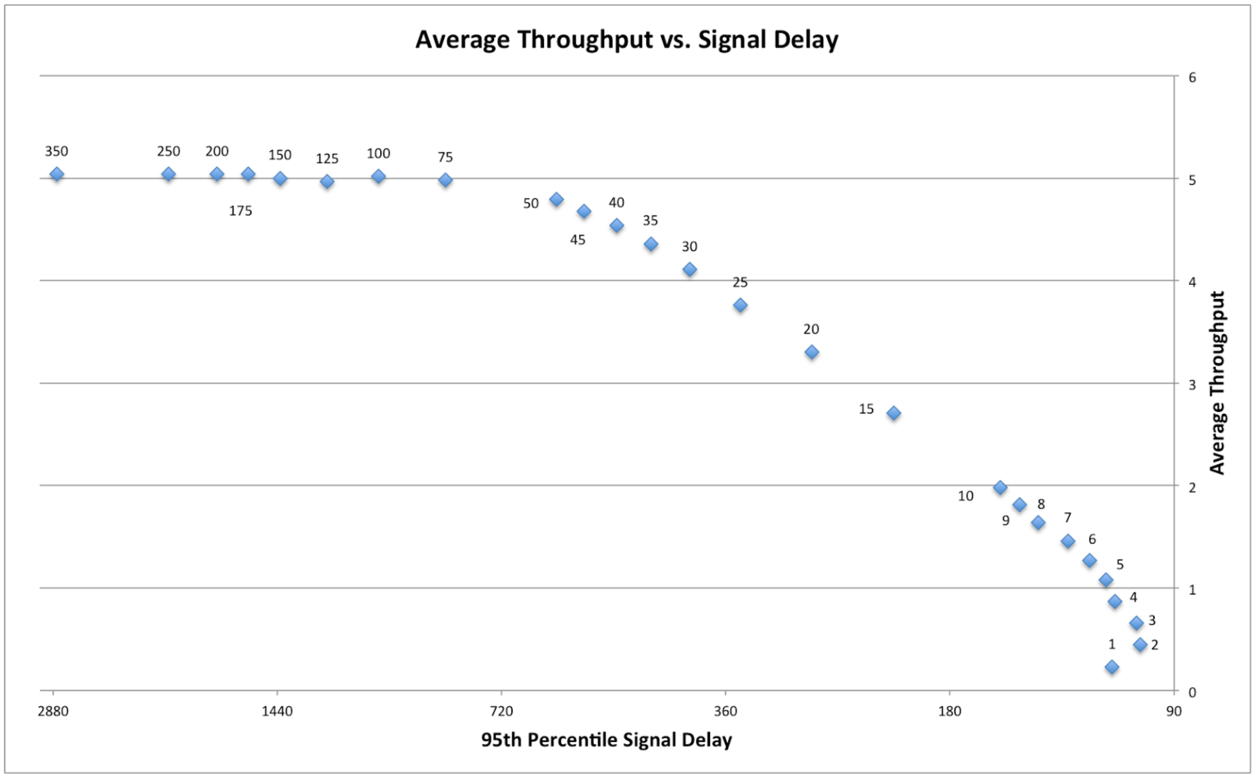
https://github.com/lanhamt/sheen/tree/warmup_a

A window size of 10 produced a power score of 12.85714286, which was the largest power score across the 26 window sizes that were tested. Generally, the larger the window size, the greater the throughput but with larger delay. The average throughput plateaus at 5.04, which is also the average capacity, somewhere between a window size of 150 and 175.

The measurements were fairly consistent over several runs, generally with average throughput ±0.05 Mbps and more variable delay of ±10 ms. For example, the following is the data collected from five runs with a window size of 10:

Average throughput: 1.98 Mbits/s, 1.98 Mbits/s, 1.98 Mbits/s, 1.98 Mbits/s, 1.98 Mbits/s
95th percentile signal delay: 154 ms, 155 ms, 154 ms, 155 ms, 156 ms

The collected results are shown in Appendix A and the associated graph is shown here (points are labeled with the window size used for that trial):



Warmup B:

https://github.com/lanhamt/sheen/tree/warmup_b

We chose to add 1 to the window for every RTT which is more aggressive than traditional TCP congestion avoidance (which adds 1 to the window for each completely ACK'd window, or 1/cwnd for each RTT). This worked better on average than adjusting only the window size, however the best power score for AIMD was worse than the best power score for adjusting window size alone.

We found that using 4 as the multiplicative decrease constant provided the best score. When the window size was cut by four, the results were an average throughput of 2.50 Mbits/s and 95th percentile signal delay of 298 ms (score: 8.389).

Multiplicative Decrease constant	Average Throughput	95th percentile signal delay	Power Score
2	2.48	298	8.322
3	2.50	304	8.256
4	2.51	298	8.389
5	2.53	304	8.32
6	2.51	303	8.28

The timeout value was set to a constant. We tried 1000 and 100 with a multiplicative decrease constant of 2. There was a significant increase in performance with the lower timeout. With a timeout of 1000 the score was 1.09 compared to the 8.32 score with 100.

Warmup C:

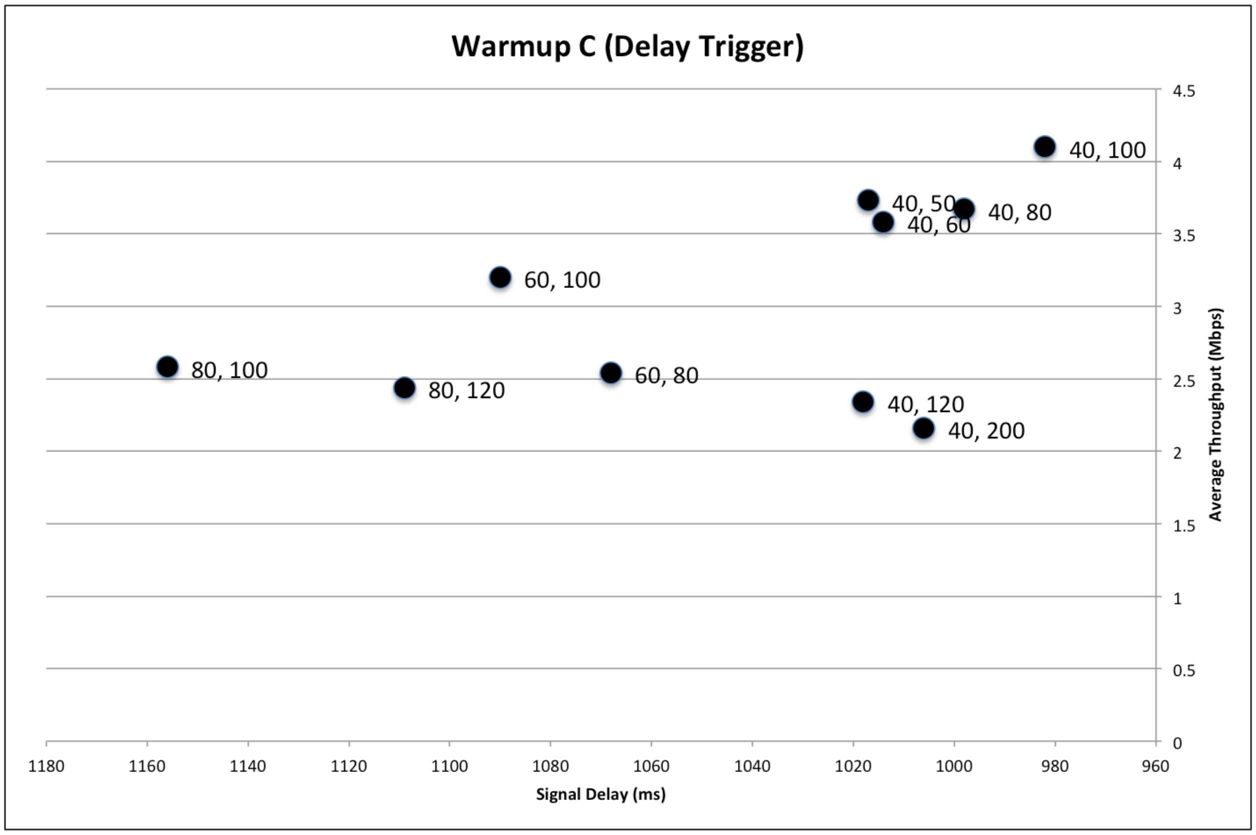
https://github.com/lanhamt/sheen/tree/warmup_c

For the delay based trigger, we used the RTT to try and establish a floor and ceiling for window size. When RTT fell below an expansion threshold, we increased window size by 1, and when it rose above a contraction threshold, we decreased window size by 1. We tried different variations on this scheme, including different bounds for expansion and contraction, ranging from as low as 40 for the expansion bound and as high as 200 for the contraction bound. We observed the best result with an expansion threshold window size of 40 and contraction threshold of 100.

We also tried several variations on the delay trigger, including increasing/decreasing window size by more and less than one. The window size adjustment parameter yielded poor results, the window size would grow too fast or slow, resulting in almost 100% throughput but several thousand millisecond delay (up to ~20000ms).

Overall, the expansion threshold of 40 (segments) gave the best results with signal delay about 10% less than the other tests on average.

The below graph illustrates the results from different expansion/contraction thresholds - each data point is representative of a different configuration. The tuple for each data point is the (EXPAND_THRESH, CONTRACT_THRESH), the full data set can be seen in Appendix B.



Contest:

We went through several major revisions to our algorithm before settling on our final iteration. Generally, our development cycle consisted of starting with an algorithm and then iterating to maximize performance. We used logging of window size and RTT to inform our decisions; for example, we created a window size floor after noticing that there were periods when window size was persistently 0 but there was available bandwidth in the network (window size would stay 0 because the controller is somewhat slow to react to changes in the network, we observed about a 10-30 segment latency between a change in bandwidth and change in RTT).

Initially, we focused on the changes in RTT. We started by keeping track of two RTT values at a time. Every time a pack was received, the new RTT was compared to the last measured RTT. We tried increasing or decreasing the window by one when the RTT decreased or increased, respectively, to try and compensate for changing network conditions (when RTT is decreasing we cut back on window size, when it's increasing we increase as well to take advantage of additional bandwidth). We also tried introducing a floor on window size which improved the flow significantly, taking our power score to about 20. This also preformed better than scaling the increases and decreases in window size using the fractional change in RTT.

After recording two RTTs, we decided to keep track of the last n RTT values to better gauge the trend in RTT. We tried keeping a list of the last 100, 200, 500, 1000, 5000, and 10000 RTTs collected and using the mean RTT to determine whether window size should be changed. The list of 10000 RTT values had the best score (power score of 22), but the delay introduced in maintaining the list may have been partially responsible for this.

We tried using these collected RTTs to compare the most recent RTT to the median, mean, 75th percentile, and 25th percentile values. The most successful of these was comparison to the 25th and 75th percentile. If an RTT was less than the 25th percentile then the window size was increased by 1 and if it was larger than the 75th percentile then the window size was cut in half. A minimum window size of 5 was enforced and the timeout value returned was twice the last recorded 75th percentile RTT. This resulted in an average throughput of 2.85 Mbits/s and 95th percentile signal delay of 123 ms (score: 23.171).

Finally, we tried TCP based algorithms which gave the best results. We started with a modified TCP Vegas (slow start with congestion avoidance) and found the best results with a modified New Reno by incorporating some of the fast recovery ideas. At first we had only slow start and congestion avoidance and chose a fixed retransmit timeout. We experimented with several RTO algorithms but found that the constant worked far better. First we used the vanJacobson formula ($\beta RTO_k = \alpha RTT_{k-1} + (1 - \alpha)RTT_k$ where α=0.1 and β=2) which resulted in an order of magnitude decrease in throughput. Then we tried using modern RTO estimates as outlined in RFC6298 which performed even worse than the vanJacobson formula. We derived our final RTO and initial window size experimentally.

Since we could not use the loss-based congestion signals that TCP relies on, in particular duplicate ACKs, we experimented with reducing the effects of timeouts. We kept the parameters for timeout adjustment the same as TCP (setting ssthresh to half of cwnd and cwnd to this updated ssthresh + 3) after experimenting with different values for decreasing cwnd (we tried decrease by factor of 1.5, 3, 4, and 5) as well as different additive values to ssthresh (tried 1, 2, and 4) and finding that all were worse than the TCP parameters. Instead, we changed how often the timeout updates occur.

We have a timeout counter so the actual timeout updates (setting ssthresh to half of cwnd and cwnd to this updated ssthresh + 3) only occur every TIMEOUT_RETRY times. We did this to avoid an overcorrection to a timeout - for example, say the network is congested and the RTT for the next 5 segments will be greater than RTO (we observed this reaction latency), then instead of doing the adjustments for all of them, we restrict the number of adjustments. In loss-based TCP, overcorrection is avoided because duplicate ACKs will be received and a single timeout will occur, the sender will adjust, and the next packet to get through will signal that the network is less congested; however, for our simulator, we must be cautious not to overreact to RTO because we observed that the delay in the network is shorter than the time needed for changes in window size to take effect (the network could not adjust cwnd/2 before doing another time out and taking it down to a factor of 4 less than the original cwnd). This correction improved the power score from ~30 to ~32.

In summary, we start out in slow start where window size is increased by 1 for each RTT. When window size is greater than ssthresh, we move to the congestion avoidance state where cwnd is increased by 1/cwnd for each RTT. If a timeout occurs in congestion avoidance, we move to fast recovery which functions similarly to slow start except window size is increased by 2 for each RTT. When a timeout occurs, if TIMEOUT_RETRY (a configurable value) timeouts have occurred then we do the the timeout updates, otherwise we increment the counter.

Conclusions:

Overall, we affirmed many of the principles of congestion control including relying on RTT as an estimator of network congestion and additive increase and multiplicative decrease as a means of adjusting window size.

Appendices:

Appendix A - Window Size

Window Size	Average Throughput	95th percentile signal delay	Power Score
1	0.23	109	2.110091743
2	0.45	100	4.5
3	0.66	101	6.534653465
4	0.87	108	8.055555556
5	1.08	111	9.72972973
6	1.27	117	10.85470085
7	1.46	125	11.68
8	1.64	137	11.97080292
9	1.81	145	12.48275862
10	1.98	154	12.85714286
15	2.71	214	12.6635514
20	3.3	276	11.95652174
25	3.76	344	10.93023256
30	4.11	402	10.2238806
35	4.36	454	9.603524229
40	4.54	504	9.007936508
45	4.68	558	8.387096774
50	4.79	608	7.878289474
75	4.98	857	5.810968495
100	5.02	1053	4.767331434
125	4.97	1234	4.027552674
150	5	1428	3.50140056
175	5.04	1575	3.2
200	5.04	1734	2.906574394
250	5.04	2015	2.501240695
350	5.04	2848	1.769662921

Appendix B - Delay Trigger

EXPAND_THRESH	CONTRACT_THRESH	Average Throughput (Mbps)	Signal Delay (ms)	Power
40	50	3.73	1017	3.667649951
40	60	3.58	1014	3.530571992
40	80	3.67	998	3.677354709
40	100	4.1	982	4.175152749
40	120	2.34	1018	2.298624754
60	80	2.54	1068	2.378277154
60	100	3.2	1090	2.935779817
80	100	2.58	1156	2.23183391
80	120	2.44	1109	2.200180343
40	200	2.16	1006	2.147117296