CS 244 **Spring 16-17**

Assignment 2: Congestion Control Contest

Professors: K. Winstein & S. Katti *Students: Jervis Muindi & Luke Hsiao*

Exercise A

Question 2.1 *Vary the fixed window size by editing controller.cc to see what happens. Make a 2D graph of throughput vs. 95-percentile signal delay (similar to what is seen on the contest analysis URLs) as you vary this value. What is the best single window size that you can find to maximize the overall "score" (log throughput/delay)? How repeatable are the measurements taken with the same window size over multiple runs?*

We tried several different windows sizes in increments of 5 datagrams. The raw numbers are shown in Table 2.1. Score is calculated as *throughput/delay*, and the best score is shown in bold.

Table 2.1: Throughput and Delay vs Fixed Window Size

Window Size	Throughput (Mbits/s)	95% Signal Delay (ms)	Score
5	1.05	109	9.63
10	1.93	155	12.45
15	**2.66**	**212**	**12.55**
20	3.26	277	11.77
25	3.73	343	10.87
30	4.07	401	10.15
35	4.32	453	9.54
40	4.51	504	8.95
45	4.65	557	8.35
50	4.76	607	7.84
55	4.85	652	7.44
60	4.91	711	6.91
65	4.94	763	6.48
70	4.96	808	6.13
75	4.98	855	5.82
80	4.99	896	5.57
85	5.00	935	5.34
90	5.00	972	5.14

These values are plotted as a 2D graph of throughput vs 95-percentile signal delay in Figure 2.1.

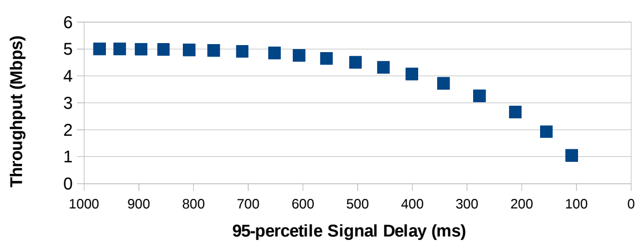

Figure 2.1: Throughput vs 95-percentile Signal Delay for varying fixed window size.

To answer the questions, we found that the best score was at a fixed window size of about 15 datagrams. The measurements taken with the same window size over multiple runs were very repeatable in the mahimahi environment. The throughput and delay only varied slightly (i.e. ± a few hundredths of Mbps or a few ms).

Exercise B

Question 2.2 *Implement a simple AIMD scheme, similar to TCP's congestion-avoidance phase. How well does this work? What constants did you choose?*

For this exercise, we implemented a simple AIMD protocol, which increments the `cwnd` by $alpha/cwnd$ on each ack, and decreases the window by `beta` on a loss event (i.e. $cwnd = cwnd/beta$). This mimics the AIMD behavior of TCP in congestion avoidance. In our mahimahi environment, there is no loss, and there is unbounded queues. Thus, to signal when a multiplicative decrease should occur, we use a fixed timeout value as a signal. We set the timeout to 80ms, which is approximately the 95-percentile signal delay on the top algorithms from the leaderboard. We also set the initial congestion window size to 10.

We tried a few of different values for `alpha` and `beta`, and recorded their performance in Table 2.2. We found that the typical values of `alpha` $= 1$ and `beta` $= 2$ perform worse than many of the fixed window sizes we tested in Exercise A. In particular, AIMD increased latency as the algorithm tries to slowly use all the throughput available. We then tweaked the constants to see how being more aggressive or timid is growth and decrease affected the power score.

Table 2.2: Throughput and Delay for various AIMD constants

Alpha	Beta	Throughput (Mbits/s)	95% Signal Delay (ms)	Score
1	2	4.72	847	5.57
1	3	4.68	765	6.12
1	4	4.65	734	6.34
1	5	4.64	724	6.41
1	10	4.59	705	6.51
0.1	1.1	3.80	377	10.08
0.1	**1.5**	**2.72**	**142**	**19.16**
0.1	2	2.43	186	13.06
0.5	2	4.41	570	7.74
0.2	4	3.23	267	12.10
2	2	4.88	1190	4.10
4	2	4.96	1655	3.00
4	10	4.95	1533	3.23

In general, we found that a slow additive increase (with `alpha` < 1), combined with a passive multiplicative decrease (`beta` < 2) provided a better power score on this cellular link that other configurations we tested. However, even in this best case, only about half of the available throughput was utilized. Even with the best constant values we found (shown in bold in the table) are not optimal.

Exercise C

Question 2.3 *Implement a simple delay-triggered scheme, where the window rises or falls based on whether the round-trip-time crosses some threshold value. Experiment with a few thresholds or tweaks and report on what worked the best.*

We implemented a simple scheme that increases or decreases the window size based on an RTT threshold. On each ACK, we calculate the RTT of the particular datagram and additively increase the window if the RTT is less than the threshold, or additively decrease the window size if the measured RTT is greater than the threshold. The amount of increase and decrease are defined by the constants `alpha` and `beta`, respectively.

We experiment with several values of our three constants: `alpha`, `beta`, and `rtt_thresh`, and summarize the results in Table 2.3

Table 2.3: Performance of Delay-based Congestion Control

alpha	beta	rtt_thresh (ms)	Throughput (Mbits/s)	95% Signal Delay (ms)	Score
1	20	100	3.30	149	22.15
1	10	100	3.76	155	24.26
1	5	100	4.32	190	22.26
1	1	100	4.81	497	9.68
1	10	150	4.43	820	5.40
1	**10**	**90**	**3.57**	**137**	**26.06**
1	10	85	3.34	136	24.56
1	10	80	3.15	129	24.42
1	10	70	2.57	119	21.60
10	10	90	4.75	237	20.04
10	1	90	4.89	924	5.29
10	1	60	4.89	924	5.29

We found that using this approach, we achieved the best performance when the additive decrease occurred at a faster rate than the additive increase in order to quickly allow the queues to drain when a particular RTT is exceeded to reduce the queueing delay. We also found that the best RTT threshold was about 90ms, when combined with our selection of alpha and beta.

This approach produced the best score of any of our attempts thus far. We believe that even a simple-delay based algorithm was able to outperform AIMD or a fixed window scheme due to it's ability to react to congestion in the network (by observing RTTs of individual datagrams). In the fixed window scheme, no ajudstments were made for congestion, and in the AIMD scheme of the previous exercise, the algorithm only reacted to timeouts (since our mahimahi environment does not drop datagrams). Consequently, AIMD would still cause bufferbloat during additive increase, and had to wait for timeouts to stop its growth, rather than the growing queueing delay.

Exercise D

Question 2.4 *Try different approaches and work to maximize your score on the final evaluation. Be wary about "overtraining": after the contest is over, we will collect new network traces and then run everybody's entries over the newly-collected evaluation trace. In your report, please explain your approach, including the important decisions you had to make and how you made them. Include illustrative plots.*

Attempt 1: EWMA Receiver Rate & BW

In our first approach, we referred to the Sprout- EWMA[1] idea of using an exponentially weighted moving average to estimate the rate that the receiver was receiving datagrams. This served as a proxy of the bottleneck bandwidth capacity and varied over time. We then approximated the RTT to as the minimum RTT that was seen during the flow. Then, our algorithm tried to maintain the bandwidth delay product number of datagrams in flight at any given time as $RTT_{estimate} * BW_{estimate}$ using the estimates we calculated.

While implementing this approach, we found that there were interesting edge cases that required us to add minimum and maximum bounds on the window size. In particular, if no minimum was set, we found that it was possible for our algorithm to reach a state where the estimated bandwidth was a rate of 1 datagram per RTT, which caused our algorithm to send a single datagram at a time in an attempt to avoid bloating the buffers, when really the estimate itself was incorrect.

This approach used the following parameters: `max_window`, `min_window`, and `weight`. We initialized our estimated RTT to 80 ms, and our estimated receiver rate to 1 datagram per 2.4 ms (approximately 5Mbps). The weight is used in updating the average time between datagrams received by the receiver as $time = (weight * measured_time) + ((1 - weight) * time)$ where $measured_time$ is the time interval between datagrams received, and the rate is calculated as 1 datagram per $time$. This means that a larger weight gives more influence to the current measurement than past measurements. On our trace, the estimated RTT converges to the minimum RTT of 42ms.

Table 2.4: Performance of EWMA-estimated Rate & Min RTT

max_window	min_window	weight	Throughput (Mbits/s)	95% Signal Delay (ms)	Score
100	0	0.25	0.01	1027	0.01
100	2	0.25	0.43	103	4.17
100	3	0.25	2.85	199	14.32
100	5	0.25	3.18	213	14.93
100	3	0.50	4.28	370	11.57
100	5	0.50	4.32	362	11.93
100	5	0.75	4.30	667	11.93
100	5	0.10	1.16	112	10.36
200	5	0.40	3.88	1421	2.73
80	5	0.30	3.69	218	16.93

As shown in Table 2.4, we were unable to break above a power score of 20. Note that when `min_window` was set to 0, we see dismal scores to do our algorithm getting in the undesirable state of sending only a single datagram at a time as described above. Furthermore, increasing this minimum to 2 datagrams resulted in similar behavior, where 1 datagram would be sent per $RTT/2$, and not provide an accurate feedback signal for our estimates. Once the minimum window was set to 3 datagrams, we were able to get better feedback, and break out of the negative feedback states that occurred in the previous two settings when throughput of the link suddenly dropped significantly. However, even with this minimum, our algorithm was slow to recover from these large drops, and would often take 5 or more seconds before utilizing the available bandwidth.

Ultimately, the primary flaw of this approach is that if the algorithm reached some steady state at a low throughput, and was unable to quickly break out of this low-throughput state to utilize the available bandwidth, and when it did break out of this state, it often overshot the available bandwidth, causing large queuing delays. We worked to address this in our following attempt.

[1] `http://alfalfa.mit.edu/`

Rocinante: Delay-Based Additive Increase, Additive Decrease

We decided build off of the delay-based algorithm from Exercise C. Specifically we built on the idea of using an additive-increase and additive decrease. However, unlike the previous attempt, which used a naive RTT threshold, we now use feedback from the acknowledged datagrams to set this threshold.

In this algorithm, we use an additive increase (controlled by `alpha`) and an additive decrease (controlled by `beta`). At all times, the algorithm is either increasing cwnd or decreasing it, based on an exponentially weighted moving average of the RTTs seen, and the current RTT of a particular datagram. At a high level, when the RTTs being seen are "stable", or within a specified range of the average RTT, the congestion window is additively increased. However, when a RTT is reported that exceeds this allowance around the average RTT, it signals congestion, and the algorithm decreases the congestion window to avoid bloating the bottleneck queues. When the network is congested, the exponentially weighted moving average is also reset to the estimated round trip propagation time, which serves as a baseline of the expected RTT in an non-congested network.

Several observations motivated the design of this algorithm. First, we learned from Exercise A, B, and C, that in general, an algorithm that reacts to delay rather than datagram loss or timeout is able to respond to congestion much more quickly, and is able to keep the queues relatively small. Second, from our first attempt, we realized that our algorithm must be able to avoid getting stuck in an undesirable stable state. In our case, we saw that having two simple states (increasing or decreasing) was an intuitive way ensure that the algorithm was responsive. This was just a choice we made for simplicity, other algorithms like BBR use more complex states (e.g. ProbeBW) to ensure that the available throughput is utilized.

With this general design of our algorithm in place, we moved on to testing the performance of our various parameters: (1) `alpha`, which defines the amount of additive increase to apply to cwnd; (2) `beta`, which defines the amount of additive decrease for cwnd; (3) `rtt_allowance`, which defines how much a RTT measurement is allowed to exceed the average RTT while still being considered "stable"; (4) `ewma_weight`, which defined the weight applied to the current measurement of RTT in the EWMA of RTT; and `timeout`, which defined the RTO in milliseconds before another datagram is sent. The some of the results of our exploration are shown in Table 2.5.

Table 2.5: Performance of Rocinante (Delay-based AIAD)

alpha	beta	rtt_allowance	ewma_weight	timeout	Throughput (Mbits/s)	95% Signal Delay (ms)	Score
1	2	1.4	0.15	1000	4.94	2064	2.39
1	4	1.4	0.15	1000	4.85	1902	2.54
1	5	1.4	0.15	1000	4.85	1896	2.56
1	10	1.4	0.15	1000	4.67	1788	2.61
2	5	1.4	0.15	1000	4.98	2807	1.77
2	5	1.4	0.01	1000	3.25	120	27.08
2	5	1.4	0.00	1000	2.92	114	25.61
5	2	1.4	0.01	1000	4.71	1853	2.54
2	5	1.1	0.01	1000	0.42	111	3.78
2	5	1.5	0.01	1000	4.00	643	6.22
2	5	1.4	0.01	100	3.29	105	31.33
2	5	1.4	0.01	50	3.24	83	39.04
2	5	1.4	0.01	25	3.47	86	40.35
2	5	1.4	0.01	15	3.55	98	36.22

From our experiments, and by observing the behavior of the throughput and queueing delay graphs paired with debug outputs, we saw that out initial choice of EWMA weight (0.15) allowed the average to drift too quickly, and thus allowed the algorithm to bloat the buffers. Instead, lowering this weight such that the average stayed relatiely close to our estimate of the propagation delay significantly reduced the 95-percentile signal delay. However, we also found that eliminating the weight entirely could negatively affect our score. An alternative approach may have been to eliminate the EWMA average and use some statistical variance to determine what RTT's to consider "stable", but we did not explore this alternative.

Next, we found that it was important in our AIAD algorithm to ensure that additive decrease occurred at a rate that was faster than additive increase. Intuitively, this allows the algorithm to more quickly close the window when congestion is detected. If we used an even more aggressive decrease (e.g. exponential), we found that throughput was wasted as just a few decreases could almost completely close the window.

When we experimented with the RTT allowance, which defines how much a measured RTT can exceed the average while still being considered "stable", we initially selected a vale of 1.4 due to our observations from an experiment with a fixed window size of 1 datagram. In particular, we saw that the minimum RTT was 42 ms, and that individual packet RTTs could vary significantly from about 40ms up to about 60ms. Further experiments with a small allowance shows that small allowance is too limiting, and hurts throughput performance. Consequently, we selected an allowance of about 1.4 (i.e. a measured RTT value that is 1.4× larger than the average can be considered stable). This seemed to work well due the nature of the cellular network trace, which resulted in large (> 1.4×) spikes in RTT when the throughput of the link dropped. Similar to the RTT average, the purpose of this is just to define what a "stable" RTT looks like, and an alternative approach could be using statistics on the RTTs seen, rather than a fixed allowance as we have implemented here.

We found that tuning the first four parameters resulted in a score of slightly less than 30, but we were unable to exceed this. Unexpectedly, lowering the timeout value significantly improved our score. We suspect that this is due to our initial misunderstanding of "signal delay". We had previously assumed that we were trying to minimize the delay of individual datagrams. However, "signal delay" is also negatively affected by dead time, when no datagrams are being sent. By lowering timeout, we ensure that there is a minimum trickle of datagrams being sent, as a rate that is low enough to not negatively bloat buffers, but frequently enough to ensure that our signal delay is not affected by seconds of dead time.

Overall, our algorithm was most sensitive to changes to the RTT allowance, EWMA weighting, and timeout. The choice of alpha and beta had less affect as long as both values were relatively small, and beta was larger than alpha.

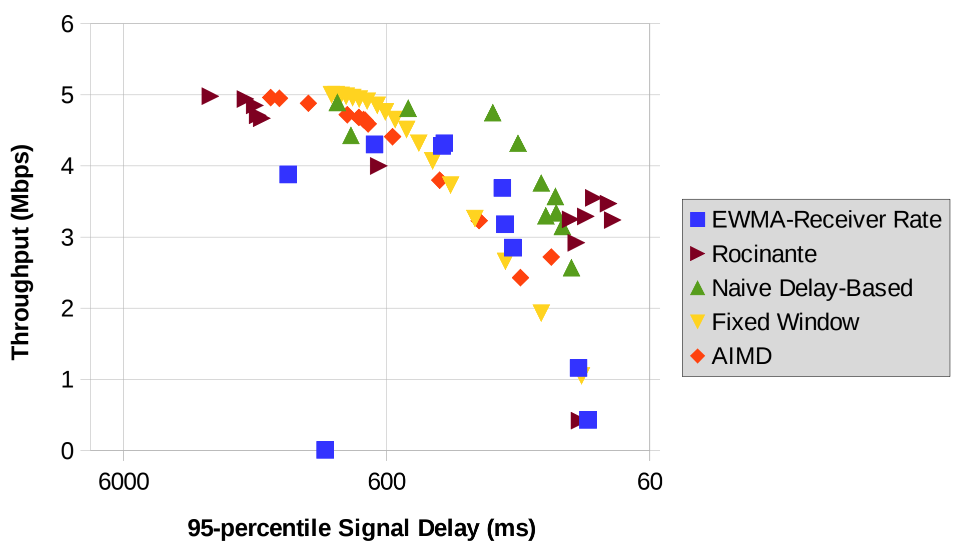

Figure 2.2: Throughput vs 95-percentile Signal Delay for all Exercises. We found that delay-based schemes produced the highest score and improved on our algorithm from Exercise C for Exercise D.

As a comparison, we show attempts from each of the exercises all together in Figure 2.2. Much like the congestion control competition itself, we were also able to explore the various realizable algorithms within the Pareto frontier. We noticed that using this enhanced delay-based AIAD approach allowed us to significantly reduce the 95-percentile signal delay while maintaining reasonable throughput than our other attempts, which lead to the best power scores.

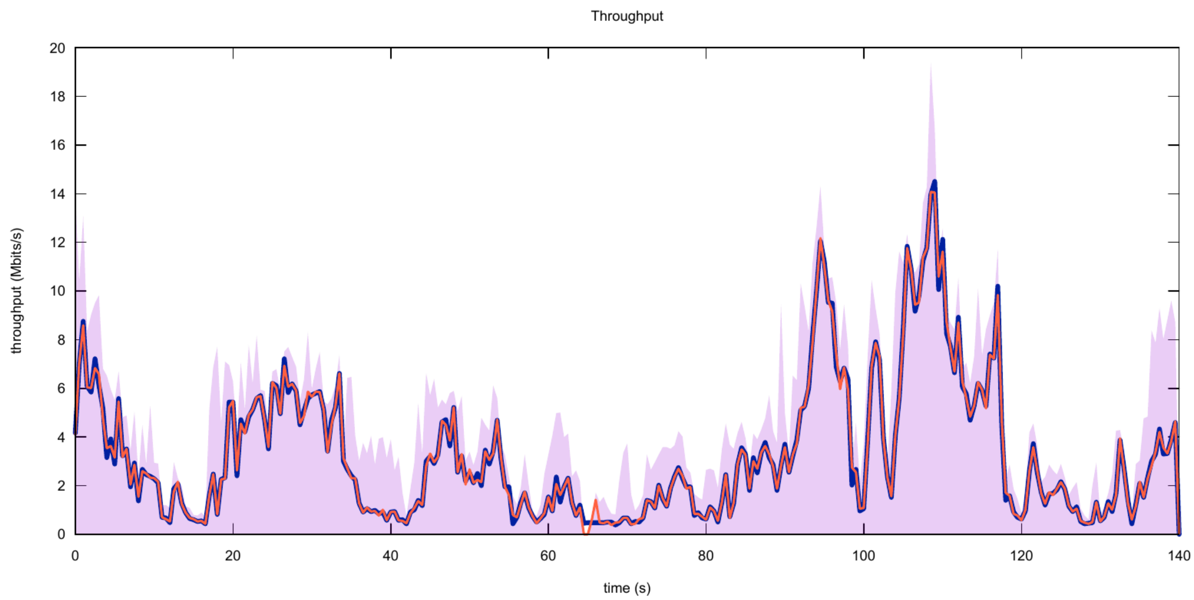

Figure 2.3: Rocinante on the Verizon trace

In Figure 2.3, we see that our algorithm is conservative, in that it often does not fully utilize the available bandwidth. Furthermore, because we are simply doing additive increase, we see that our algorithm can also be a little but slow to grow. For example, at about 70 seconds into the trace, available bandwidth increases, but our algorithm additively increases its congestion window and misses out on some of the available throughput. One approach that may have addressed this would be to implement a short probing state like that in BBR, where BBR can converge exponentially fast to a new bottleneck bandwidth.

Exercise E

Question 2.5 *Pick a cool name for your scheme!*

We call our approach `Rocinante`, named after the light, agile attack ship in The Expanse. Our code is available on GitHub here:

`https://github.com/jervisfm/cs244-project2`

Summary Statistics

As requested in the submission guidelines, our final submission's summary statistics are shown below.

Average capacity: 5.04 Mbits/s
Average throughput: 3.31 Mbits/s (65.6% utilization)
95th percentile per-packet queueing delay: 39 ms
95th percentile signal delay: 79 ms
Power Score: 41.90