CS244 Proj 2

Eric Theis & Nathan James Tindall

April 2016

1 Warmup A: Window Size

Window Size	Avg. Throughput	95% delay	score
200	5.04	1733	2.91
200	5.04	1733	2.91
100	5.02	1052	4.77
100	5.02	1052	4.77
50	4.79	608	7.88
50	4.79	609	7.87
25	3.77	345	10.93
25	3.77	344	10.96
10	1.98	156	12.69
10	1.98	155	12.69

Our best window size (maximizing the score) was 10. This is probably because delay increased exponentially with throughput (at least when the window is constant). We saw very little variation between our runs of the same window size, so our results are very repeatable.

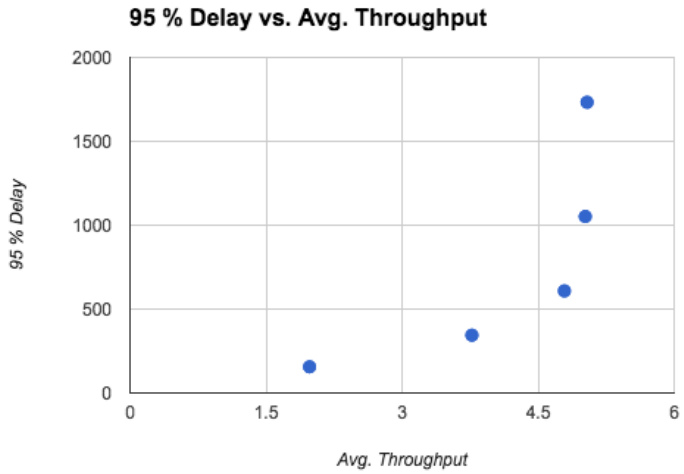
2 Warmup B: Simple AIMD

We incremented the cwnd by 1/cwnd on each ack, and added a function that informs the controller when a timeout occured, setting cwnd to k * cwnd, where $k < 1$. The initial cwnd size was 1.

We tested many different values of k, but even setting cwnd to 1 after each timeout, but had poor results since the buffer size is unbounded.

In an attempt to improve performance, we tested adding an upper bound on the cwnd, never allowing it to increase past 20, and setting the cwnd to 1 whenever we would normally increment past 20 (emulating a sawtooth-AIMD type behavior).

We additionally tried tweaking the timeout value from the initial value of 1000ms. By setting the timeout to 50ms, we observed a power score of 16, our best so far (with no upper bound on cwnd.)



AIMD is not performing very well, failing to get a power score above about 16. We suspect this is because a linear increase in cwnd is too slow to take full advantage of available throughput.

3 Warmup C: Simple RTT

Our simple delay triggered scheme used a running average of previous RTTs, linearly (1/cwnd) increasing when our RTTs were below the average and linearly decreasing otherwise. We initially thought to find an equilibrium RTT, quickly realizing that large variations in available throughput would prevent this. Thus, we also tried using a weighted average (weighing the most recent measurement more than the existing samples). The less we weighted previous RTTs, the better we performed. However, none of our attempts performed better than our AIMD runs.

4 Exercise D: The Contest

One of the biggest problems with our method in part C was that our reaction to delay became worse as delay became worse. Thus, we used a heap to track all un-aed packets, allowing us to track the current delay on said packets. We primarily used the delay on the oldest un-acked packet to determine whether to increase or decrease window size, hence our decision to use a min-heap on send-timestamp as our data structure.

Upon sending a packet, we add the packet's send timestamp to the heap. Upon receiving an ack, we remove the corresponding send timestamp from the heap. Each time a packet is sent, we retrieve the min timestamp from the map. Depending on the difference between the send timestamp and the min

timestamp, we multiplicatively decrease by a factor (smaller deltas for smaller differences). If the difference is less than 50ms (chosen due to half of the timeout length), we additively increase by a small factor.

Additionally, we intended to smooth the updates to cwnd by only updating at most every 12.5ms. However, due to an implementation error, we only updated if 12.5ms had passed since the last time the threshold for reduction had passed. For example, let packets A, B, and C, be consecutively sent from the network interface such that their timestamps are 10ms, 15ms, and 30ms, respectively. Additionally, suppose that they are all sent during a period of time where the min timestamp of all outstanding packets is 1ms, and that the difference threshold (between the packet timestamp and the min timestamp) to cause a decreasing action is 5ms. So when packet A is sent, it would cause a decreasing action, and the last_time_decreased value would be set to packet A's timestamp, 5ms. Packet B would also cause a decreasing action, but does not because not enough time has passed since a packet has passed a threshold. The last_time_decreased value would be updated to packet B's timestamp (even though no decreasing action occured). Finally, when packet C is sent, it would cause a decreasing action, since more than 12.5ms had passed since the last packet that could have caused decreasing action was sent.

It turns out that our erroneous implementation performed better than our corrected one, so we kept it. We suspect it does better because it performs a correction only after waiting to see if its last correction had any effect. Since most of the corrections are minor corrections (the majority of the time differences we observe fall within the fourth row of the chart below), mutating the cwnd by a very small multiplicative factor. This prevents our algorithm from making too many minor adjustments, overreacting to brief latency induced by spikes in network bandwidth.

Cutoff (T = timeout)	Delta	Num. Observed
$4 * T$	$0.5 * cwnd$	27
$2 * T$	$0.8 * cwnd$	38
$1 * T$	$0.95 * cwnd$	251
$\frac{1}{2} * T$	$0.97 * cwnd$	2539

For example, if the min value is more than $2 * T$ older than the current packet, then cwnd would be set to 0.8. The third column represents the number of each delta that we observed.

We initially wanted our algorithm to make very small adjustments that would aggregate the appropriate amount of change over time. Our tests showed that this was not reactive enough to rapid spikes in network connectivity. By adding additional difference cutoffs, we get the benefits of minor adjustment and the ability to scale back drastically when there are a lot of old packets that are outstanding in the network.

5 Exercise E

nutmeg - official submission (Amazon EC2 instance)
bloop - our virtual machine

`https://github.com/ntindall/nutmeg.git`