6.829 Pset 2 Writeup

Rati Gelashvili <gelash@mit.edu>
Henry Yuen <hyuen@mit.edu>

Our git repository: https://hyuentcs@bitbucket.org/hyuentcs/hyuen_pset2.git

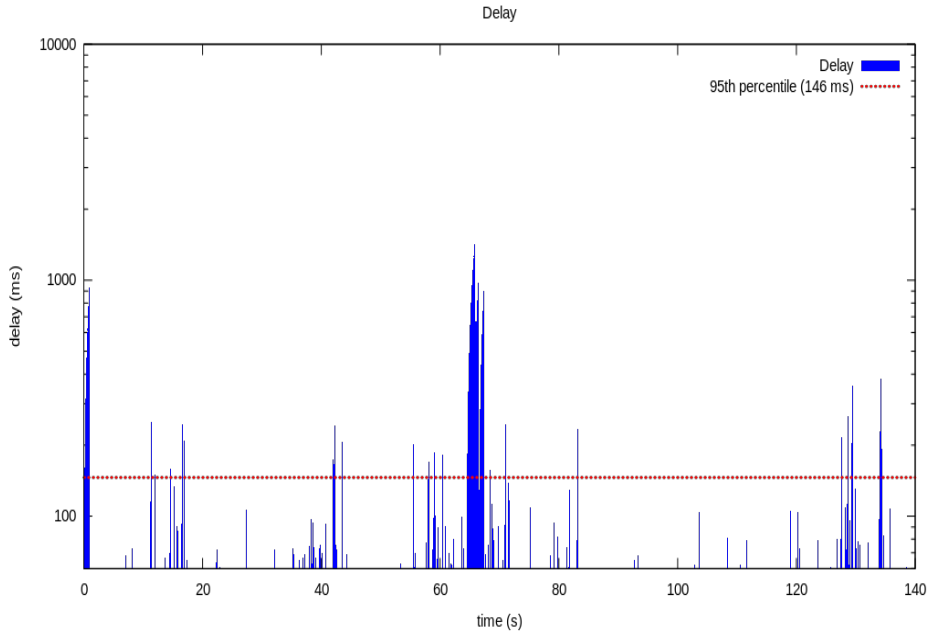
Note: the code on the git repository is our contest algorithm. The algorithms corresponding to fixed window size, AIMD, and Delay Trigger protocols can be found in older revisions in the git repository, if desired.

1. Fixed Window Size

We measured the throughput and delay of the protocol that simply uses a fixed window size. The results are below. We collected data for window sizes = 5, 10, 20, 40.

window size = 5

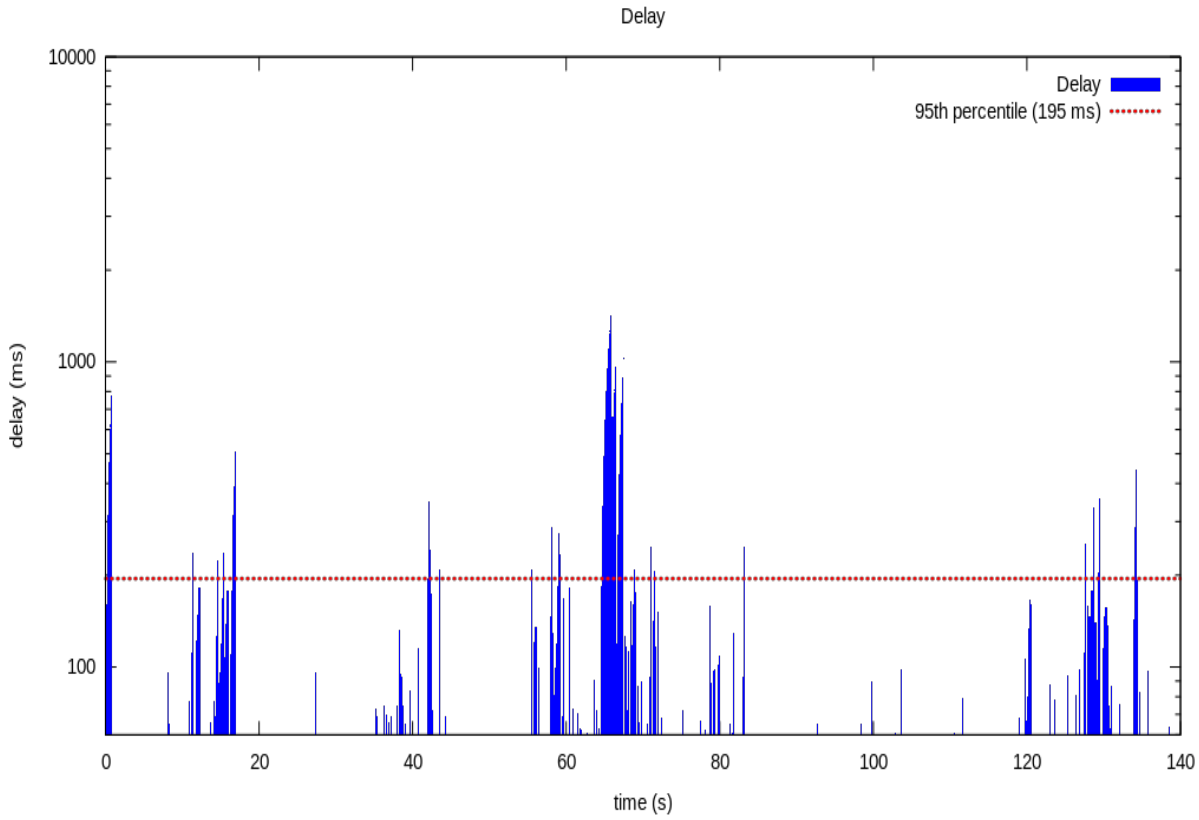
Delay graph:



log(throughput/delay) = -4.87
Delay (95 percentile): 146 ms
Protocol throughput: 1.12 Mbps

window size = 10

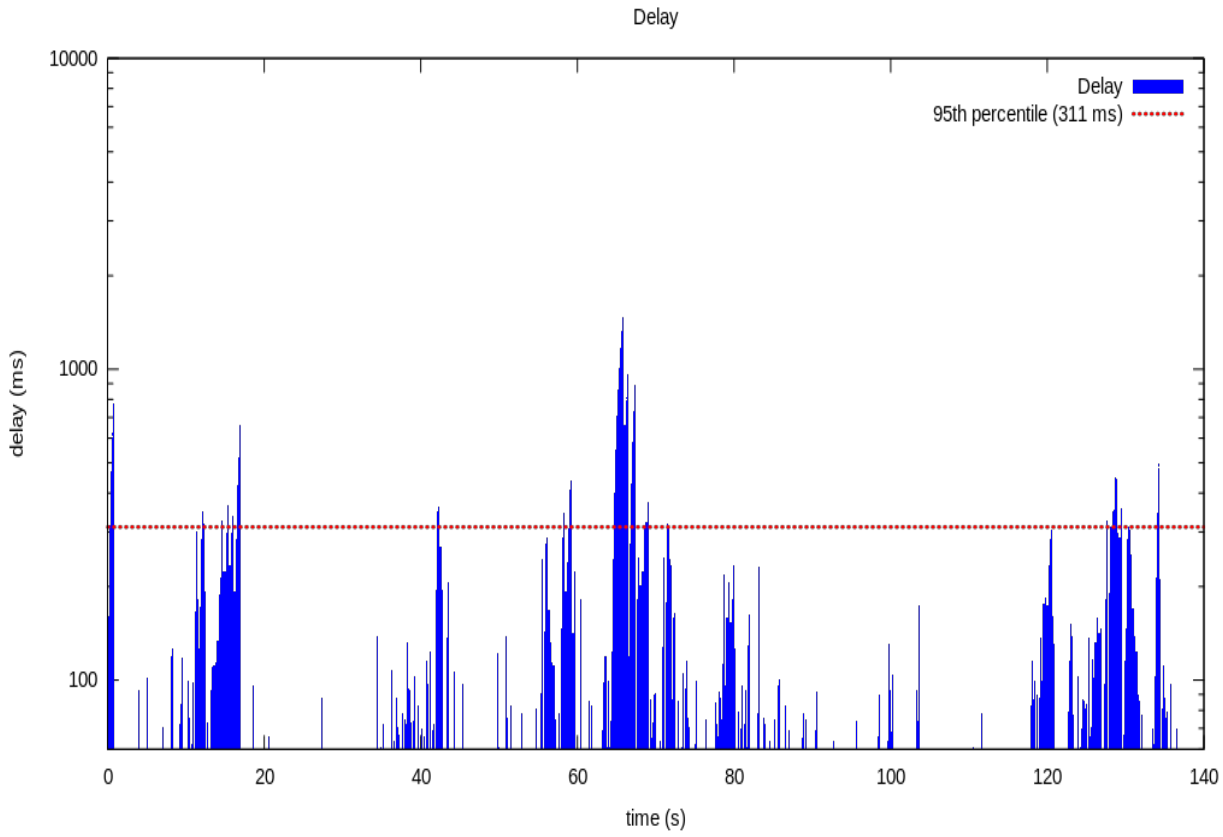
Delay graph:



log(throughput/delay) = -4.56
Delay (95 percentile): 195 ms
Protocol throughput: 2.04 Mbps

window size = 20

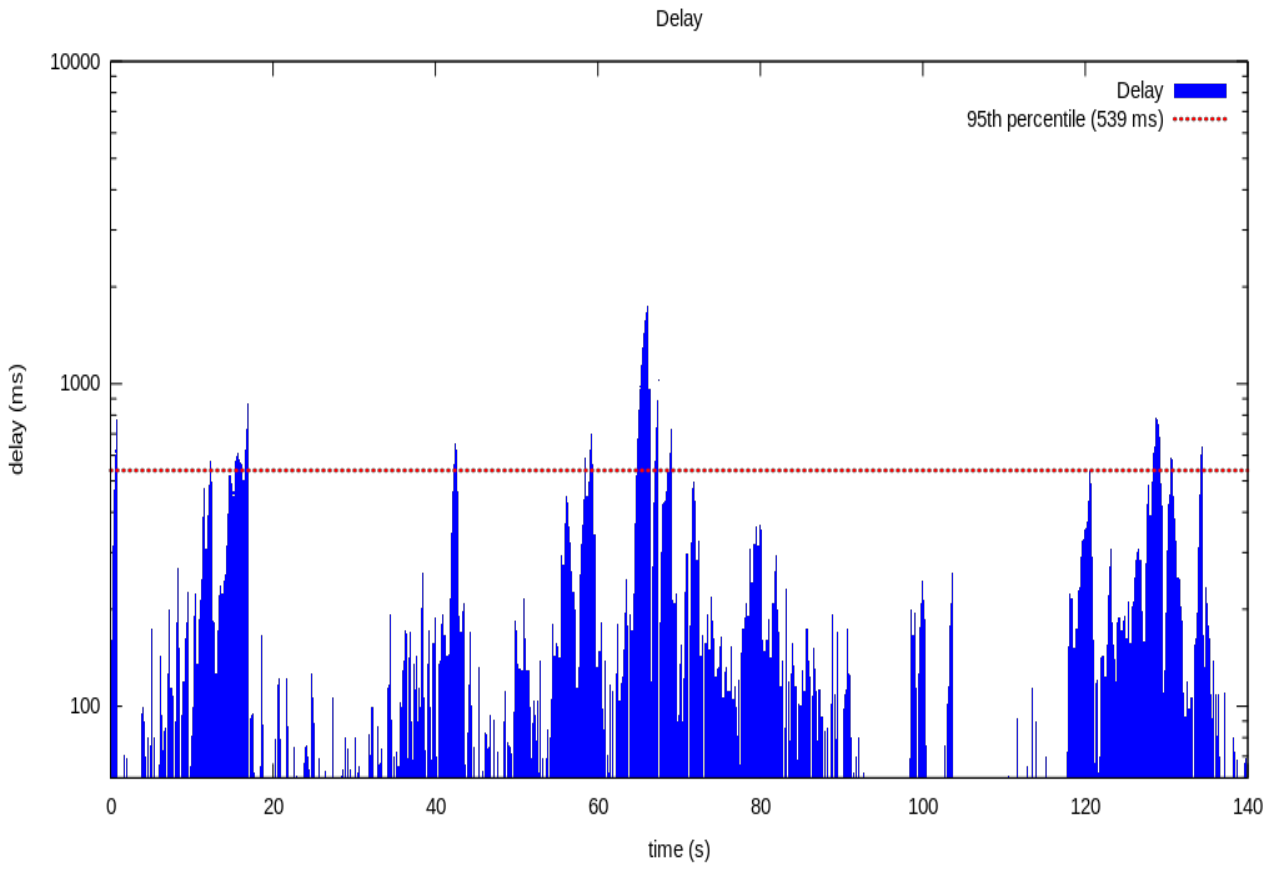
Delay graph:



log(throughput/delay) = -4.53
Delay (95 percentile): 311 ms
Protocol throughput: 3.36 Mbps

window size = 40

Delay graph:



log(throughput/delay) = -4.78
Delay (95 percentile): 539 ms
Protocol throughput: 4.54 Mbps

fixed window size discussion

The best score we achieved was -4.53 with window size = 20. The measurements are definitely repeatable, because the entire simulation process is deterministic, and the only variation should come due to minor fluctuations in, say, the latency within the simulator itself.

2. AIMD

We implemented a simple AIMD scheme. It worked as follows:

Upon receipt of an ACK: window size += ADDITIVE_INCREASE/window_size
Upon timeout of after TIMEOUT **ms**: window size /= MULTIPLICATIVE_DECREASE

where ADDITIVE_INCREASE, TIMEOUT, and MULTIPLICATIVE_DECREASE are constants that we varied.

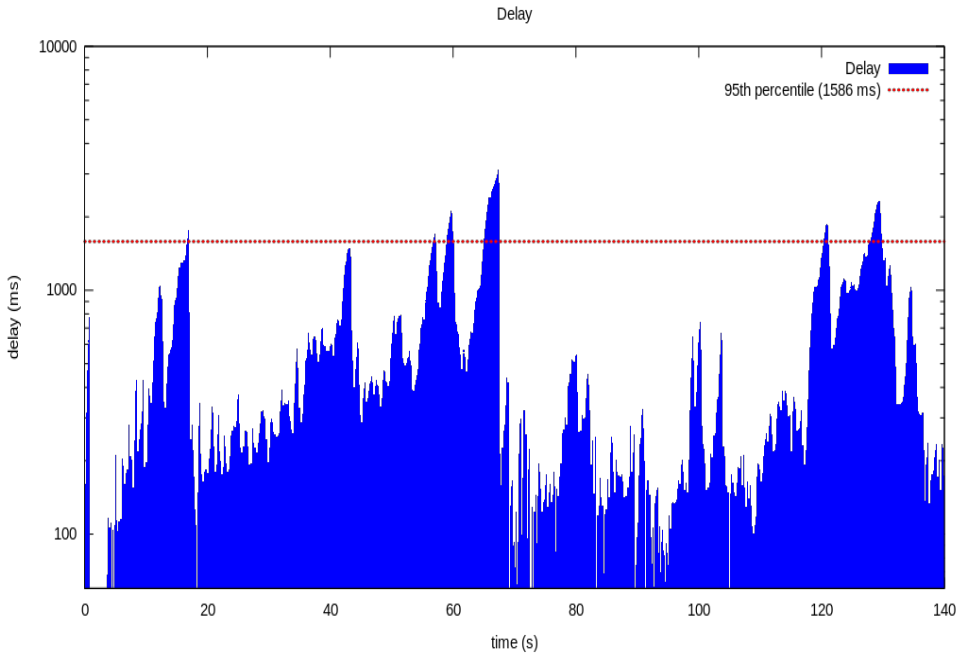
In a sense, this is not “true” AIMD, in that the multiplicative decreases are performed whenever there’s a signal of a dropped packet (because packets are never dropped in this system). We simulate this signal via the TIMEOUT.

Here are the various parameter settings we measured:

Experiment #1

TIMEOUT = 300ms	ADDITIVE_INCREASE = 1	MULTIPLICATIVE DECREASE = 5
log(throughput/delay) = -5.78	delay (%-ile): 1586 ms	throughput: 4.90 Mbps

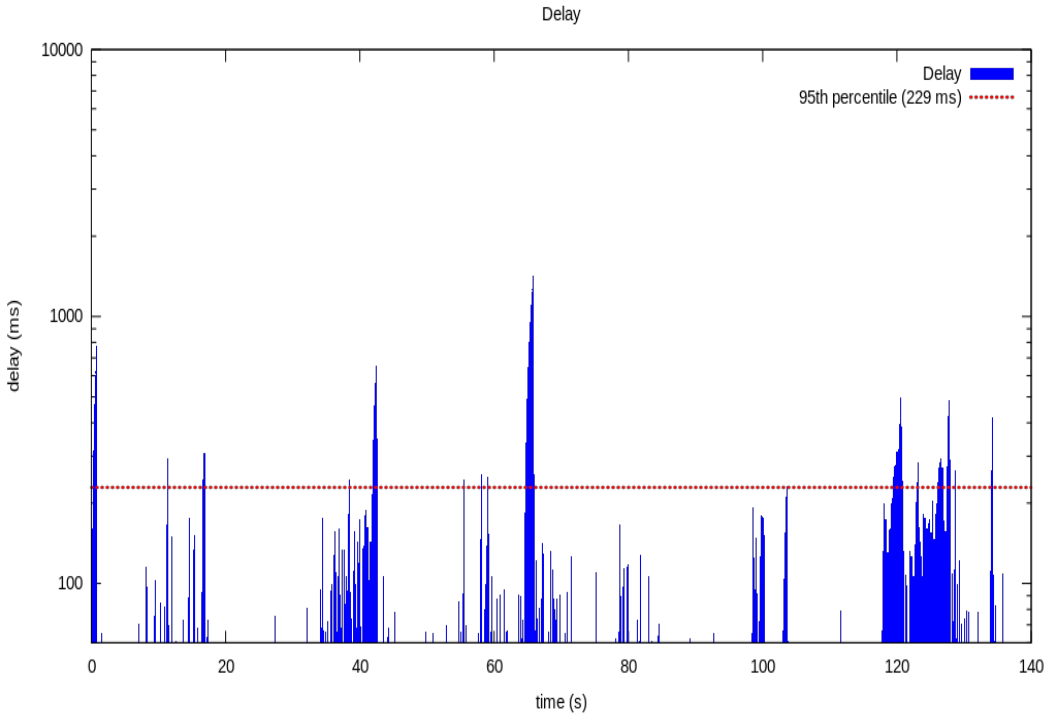
Delay graph



Experiment #2

TIMEOUT = 100ms	ADDITIVE_INCREASE = 0.1	MULTIPLICATIVE DECREASE = 5
log(throughput/delay) = -4.45	delay (%-ile): 229 ms	throughput: 2.68 Mbps

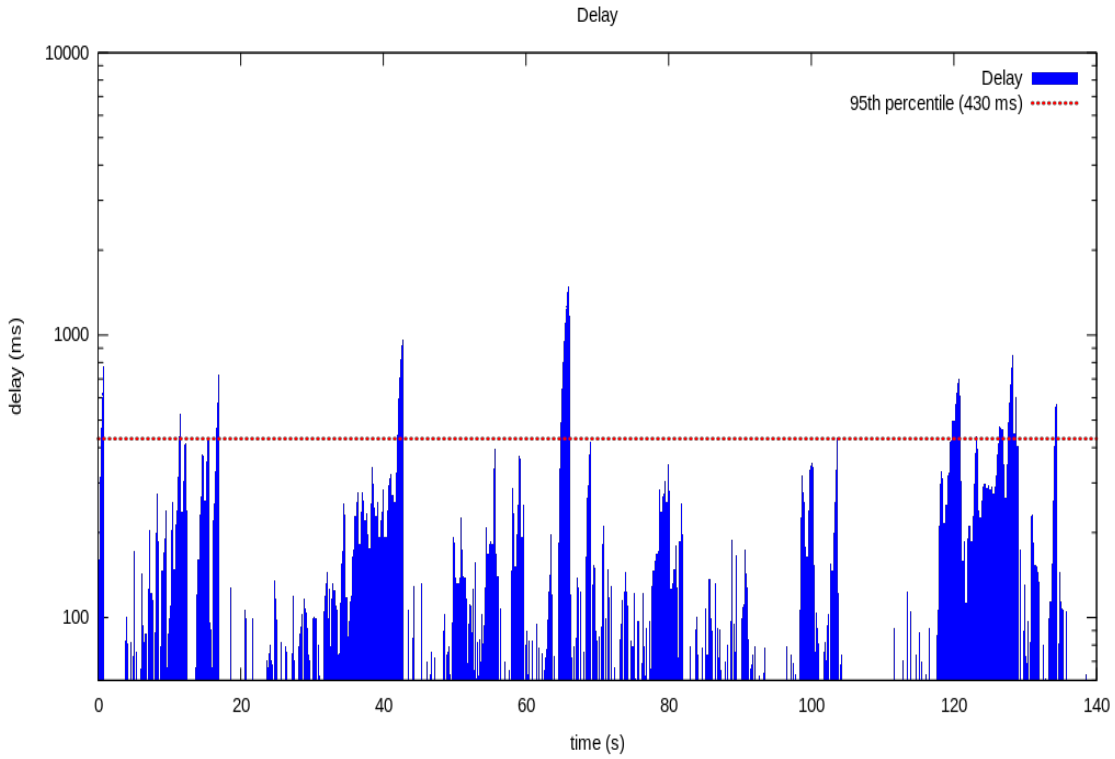
Delay graph



Experiment #3

TIMEOUT = 100ms	ADDITIVE_INCREASE = 0.2	MULTIPLICATIVE DECREASE = 5
log(throughput/delay) = -4.61	delay (%-ile): 430 ms	throughput: 4.26 Mbps

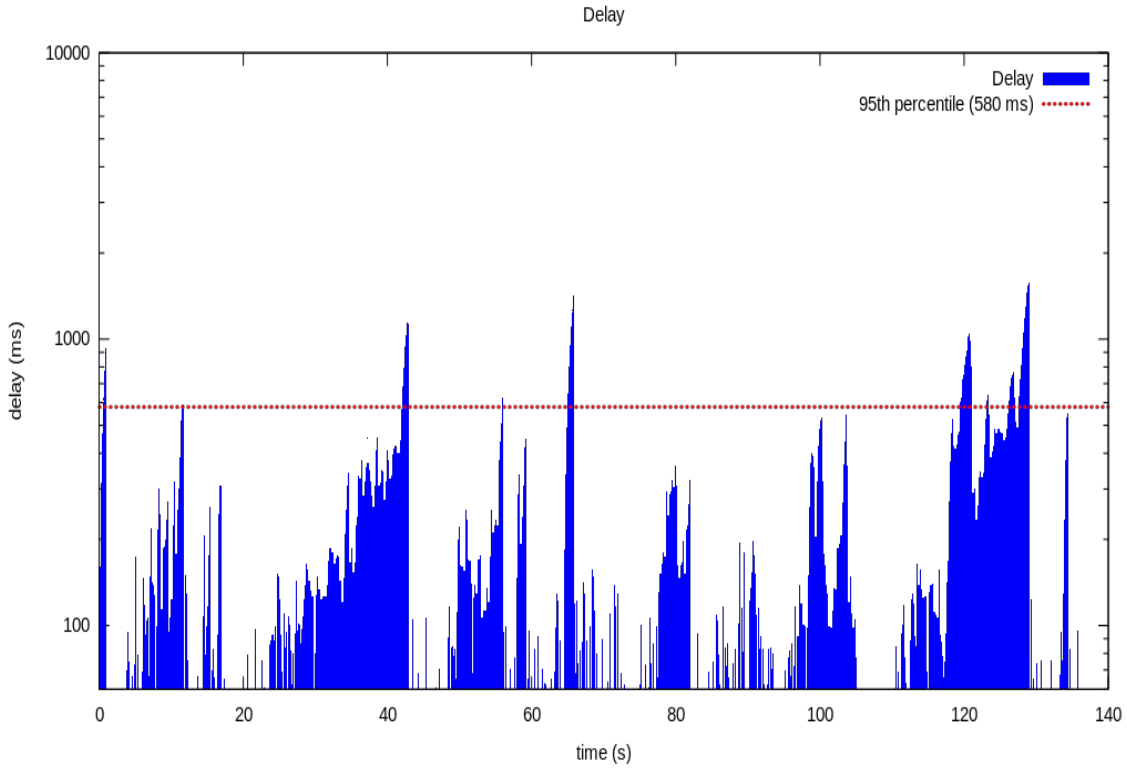
Delay graph



Experiment #4

TIMEOUT = 100ms	ADDITIVE_INCREASE = 0.5	MULTIPLICATIVE DECREASE = 10
log(throughput/delay) = -4.93	delay (%-ile): 580 ms	throughput: 4.22 Mbps

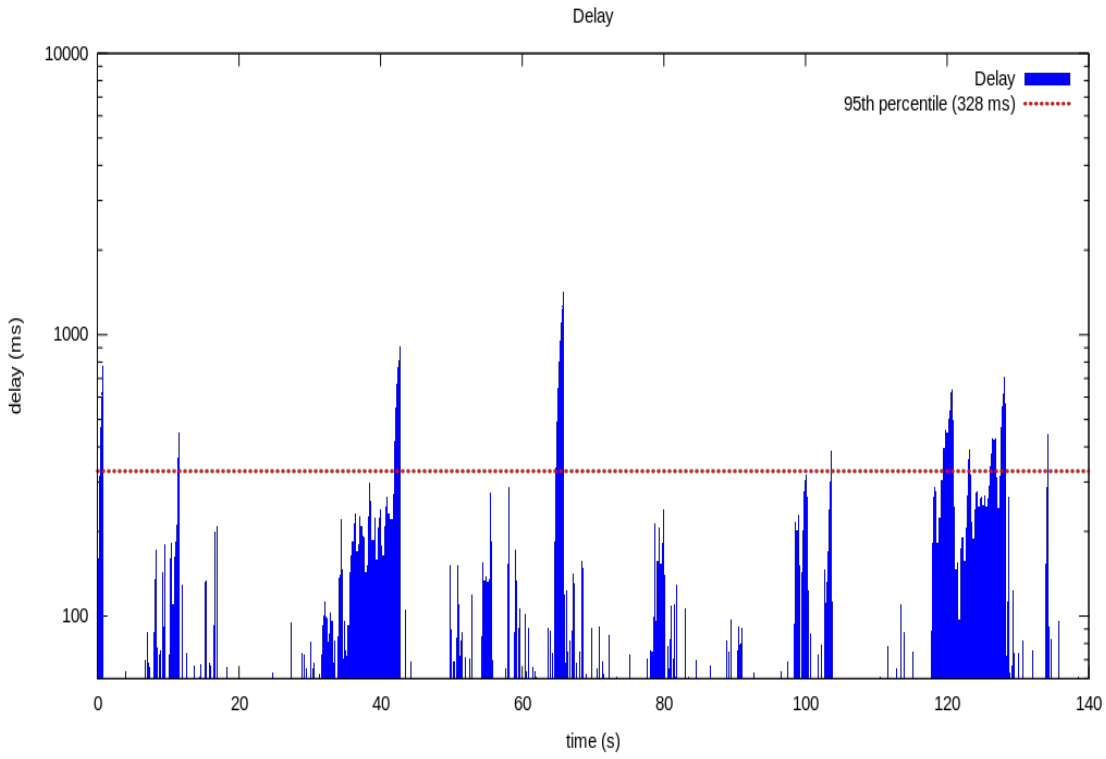
Delay graph



Experiment #5

TIMEOUT = 100ms	ADDITIVE_INCREASE = 0.2	MULTIPLICATIVE DECREASE = 10
log(throughput/delay) = -4.60	delay (%-ile): 328 ms	throughput: 3.30 Mbps

Delay graph



AIMD discussion

The best score we achieved with AIMD was -4.45, with average throughput 2.68 Mbps and 95 percentile delay of 229 ms. The constants were: TIMEOUT = 100ms, ADDITIVE_INCREASE = 0.1, and MULTIPLICATIVE_DECREASE = 5.

This method does not work very well, primarily because there is no “dropped packet” signal. The only signal of congestion comes from timeouts from not having received an ACK within TIMEOUT milliseconds, and first this signal occurs long after congestion has happened, and by the time the AIMD algorithm reacts to this, it’s too late (self inflicted buffering delays have happened). Also, because of the way the datagrump-sender works, every time there’s a timeout, it sends a packet (which does not help with congestion).

3. Delay Trigger

We implemented two versions of the Delay Trigger method.

Algorithm #1
Upon receipt of an ACK:

```
calculate rtt = time ack received - time packet sent

if rtt >= DELAY_UPPER_BOUND,
        window_size = CONSERVATIVE_WINDOW_SIZE
else if rtt <= DELAY_LOWER_BOUND
        window_size = BASELINE_WINDOW_SIZE
```

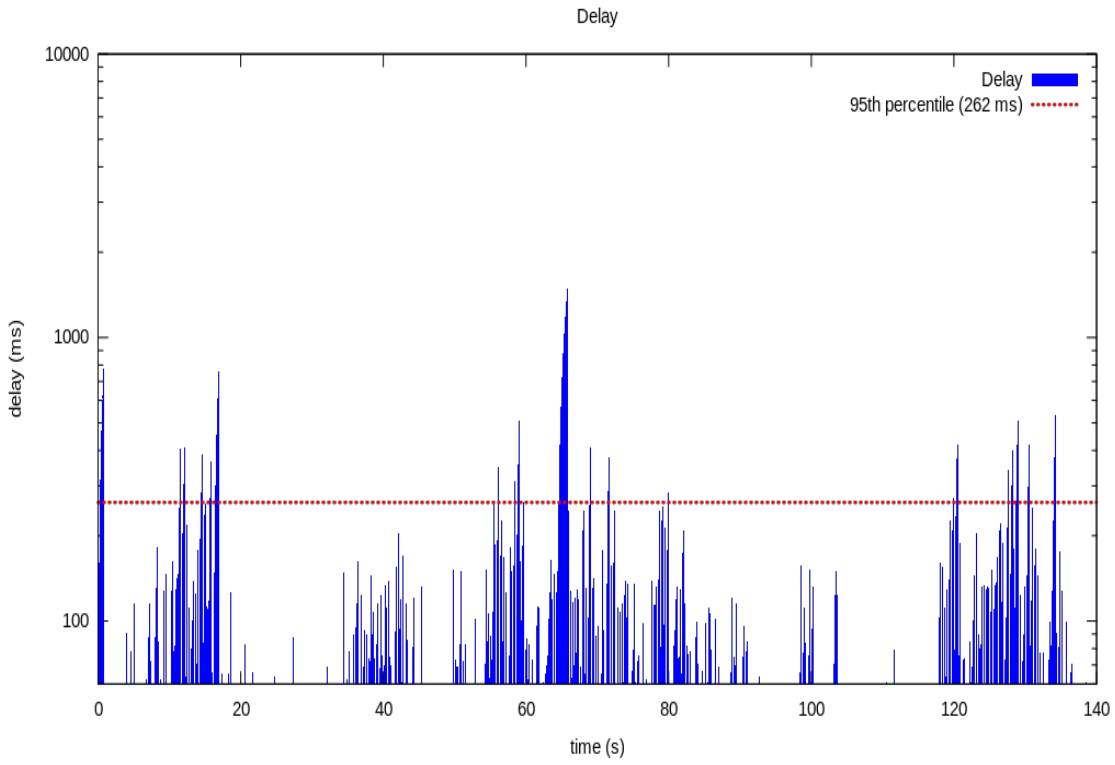
where DELAY_UPPER_BOUND, DELAY_LOWER_BOUND, CONSERVATIVE_WINDOW_SIZE, and BASELINE_WINDOW_SIZE are constants that we vary.

The following two experiments were based off of Algorithm #1

Experiment #1

DELAY_UPPER_BOUND	150ms
DELAY_LOWER_BOUND	50ms
CONSERVATIVE_WINDOW_SIZE	1
BASELINE_WINDOW_SIZE	25
log(throughput/delay)	-4.30
Delay (95 %-ile)	262ms
Throughput	3.55 Mbps

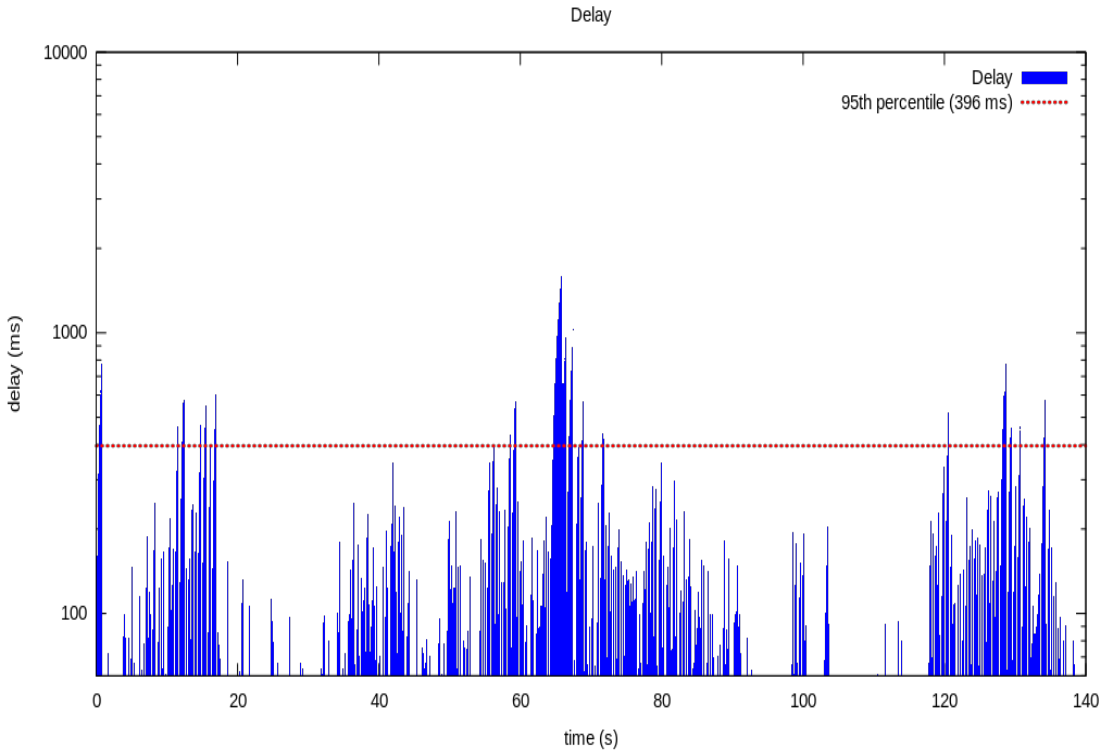
Delay graph



Experiment #2

DELAY_UPPER_BOUND	150ms
DELAY_LOWER_BOUND	50ms
CONSERVATIVE_WINDOW_SIZE	1
BASELINE_WINDOW_SIZE	35
log(throughput/delay)	-4.61
Delay (95 %-ile)	396ms
Throughput	3.96 Mbps

Delay graph



Algorithm #2

In the previous algorithm, the window size was set to fixed values based on whether it was above or below the delay bounds. Here, if the delay is above or below the bounds, the window will rise or fall. If the delay is high, the window will experience a multiplicative decrease. if the delay is low, the window will increase an additive increase.

Upon receipt of an ACK:

```
calculate rtt = time ack received - time packet sent

if rtt >= DELAY_UPPER_BOUND
        window_size -= DELAY_TRIGGER_DECREASE
        if window_size <= CONSERVATIVE_WINDOW_SIZE
                window_size = CONSERVATIVE_WINDOW_SIZE

else if rtt <= DELAY_LOWER_BOUND
     window_size += DELAY_TRIGGER_INCREASE/the_window_size
```

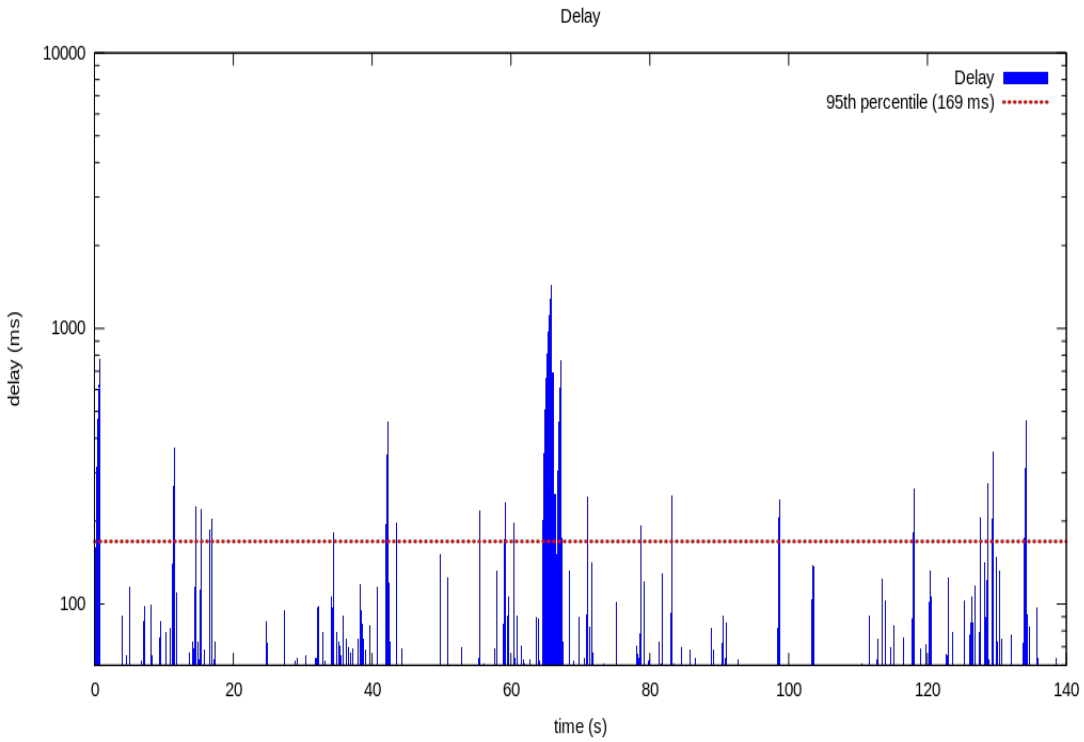
where DELAY_UPPER_BOUND, DELAY_LOWER_BOUND, CONSERVATIVE_WINDOW_SIZE, DELAY_TRIGGER_DECREASE and DELAY_TRIGGER_INCREASE are constants that we vary.

The next four experiments are based off of Algorithm #2.

Experiment #3

DELAY_UPPER_BOUND	150ms
DELAY_LOWER_BOUND	50ms
CONSERVATIVE_WINDOW_SIZE	1
DELAY_TRIGGER_DECREASE	1
DELAY_TRIGGER_INCREASE	0.4
log(throughput/delay)	-4.00
Delay (95 %-ile)	169ms
Throughput	3.10 Mbps

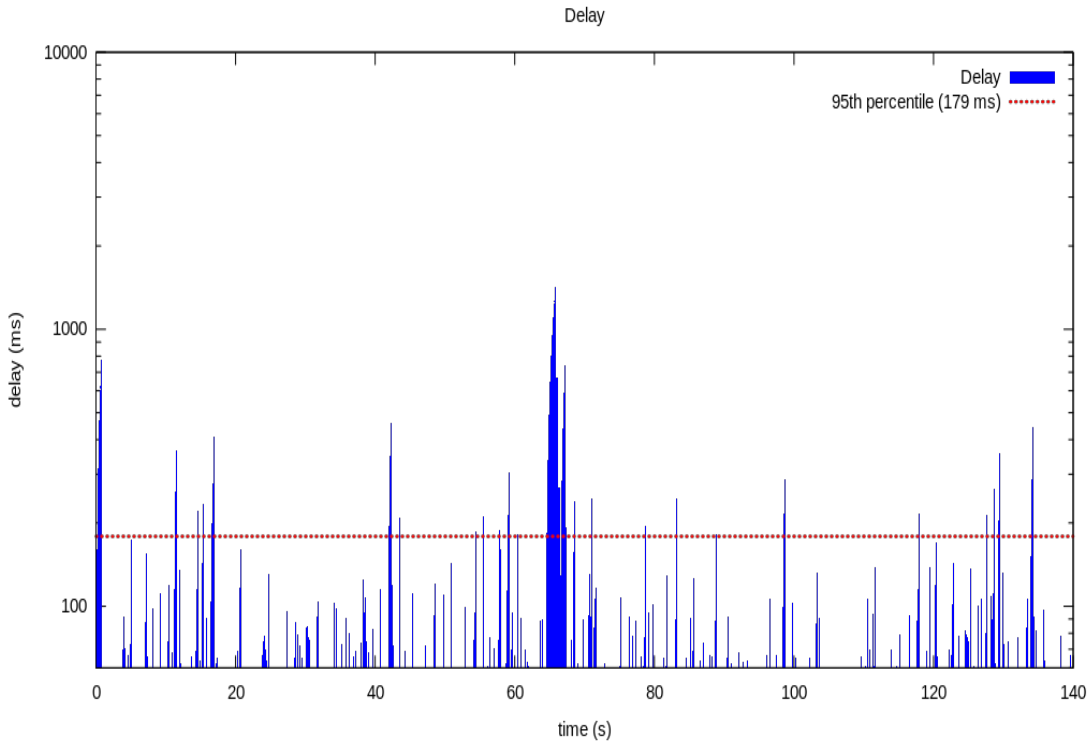
Delay graph



Experiment #4

DELAY_UPPER_BOUND	120ms
DELAY_LOWER_BOUND	80ms
CONSERVATIVE_WINDOW_SIZE	1
DELAY_TRIGGER_DECREASE	1
DELAY_TRIGGER_INCREASE	0.8
log(throughput/delay)	-3.94
Delay (95 %-ile)	179ms
Throughput	3.48 Mbps

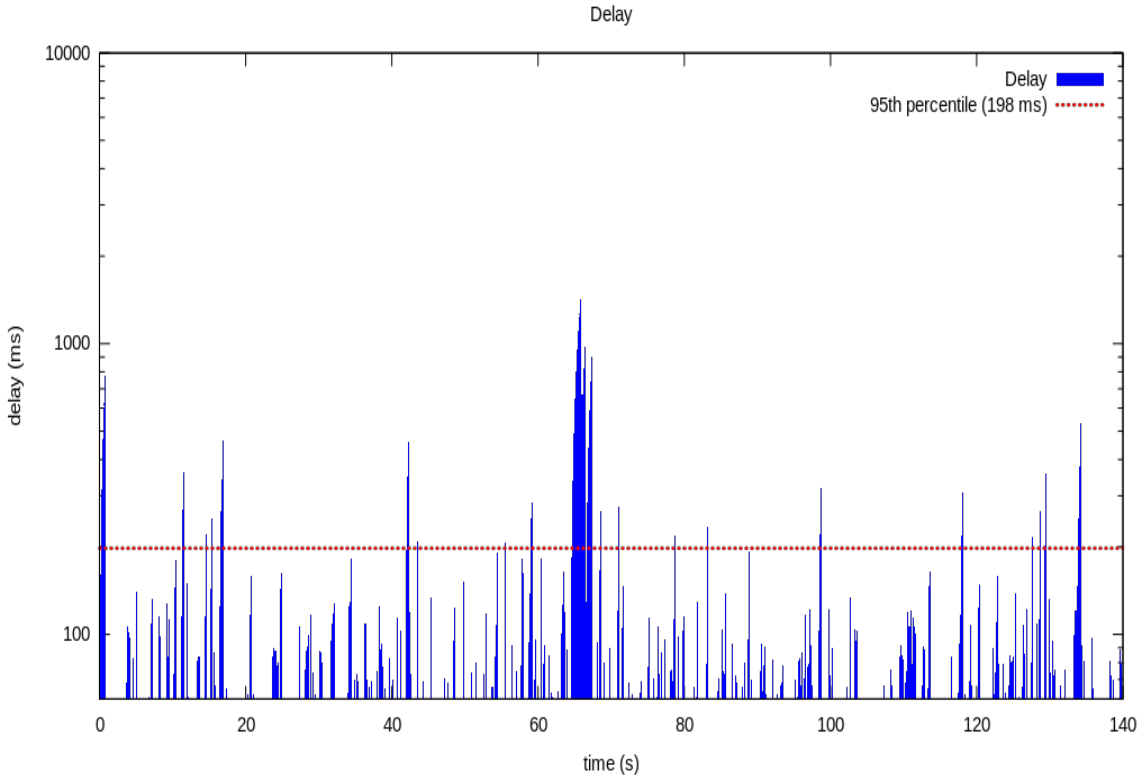
Delay graph



Experiment #5

DELAY_UPPER_BOUND	150ms
DELAY_LOWER_BOUND	80ms
CONSERVATIVE_WINDOW_SIZE	1
DELAY_TRIGGER_DECREASE	1
DELAY_TRIGGER_INCREASE	0.8
log(throughput/delay)	-3.93
Delay (95 %-ile)	198ms
Throughput	3.90 Mbps

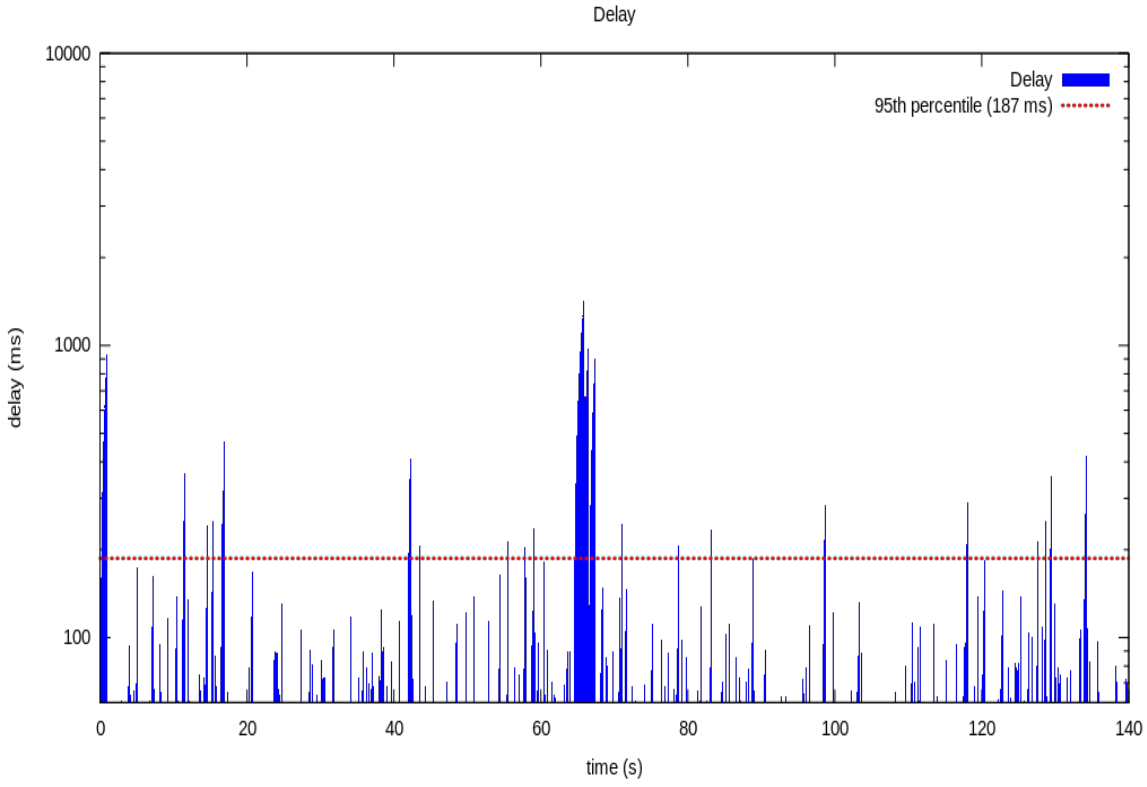
Delay graph



Experiment #6

DELAY_UPPER_BOUND	120ms
DELAY_LOWER_BOUND	80ms
CONSERVATIVE_WINDOW_SIZE	1
DELAY_TRIGGER_DECREASE	1
DELAY_TRIGGER_INCREASE	1
log(throughput/delay)	-3.94
Delay (95 %-ile)	187ms
Throughput	3.63 Mbps

Delay graph



Delay Trigger discussion

The Delay Trigger paradigm is clearly superior to the previous two protocols (fixed window size and AIMD). Algorithm #2 is better than Algorithm #1, simply because the window size's responses to changes in delay is smoother. Finally, the best parameter settings seemed to be when the DELAY UPPER BOUND is close to the DELAY LOWER BOUND, forcing the window size to settle into a near-optimal value around when the delay is about 100ms.

4. Contest Algorithm (yes that's the official name)

The algorithm:

For the competition, we started with the Delay Trigger algorithm and tried to push it to the limits. We observed that having the DELAY UPPER BOUND value, packets trigger the event of cutting down the window size when their acknowledgement comes late, but waiting for the acknowledgement might waste our most valuable resource - time, especially if the link is congested. What do we mean exactly? We know when the packet was sent, and after the DELAY UPPER BOUND time passes since the packet was sent, if the packet is not acknowledged, we know this packet will definitely trigger the cutting down the window size event at some point in the future. Knowing this, we would like to preemptive and act instantly, with the hope to react faster, thus improve the delay that we are imposing by sending more packets onto the congested link.

To implement the algorithm, we used a priority queue data-structure, that contained the pairs for each sent packet - the id of the packet and the time, after which the packet is destined to be have the round trip time larger than the DELAY UPPER BOUND. The data structure is sorted by this threshold times. We utilized the sender's timeout mechanism, to make sure that the top elements of the data structure (candidate late packets), were checked regularly - dequeued out of the top of the priority queue if their threshold time (to compare the time, we used the current timestamp generated the same way as the timestamps for the packets) was passed - and triggering the cutting off the window size event, if they were not acknowledged at this time.

The event of cutting down the window size was exactly the same as in the original Delay Trigger algorithm. However, we also maintained another state variable - late_packet_count indicating the number of late packets detected in the most recent round.

Window size is increased similarly as before, if a packet arrived with really fast round trip time. If it was smaller than the DELAY LOWER BOUND threshold, the window size would increase with a large constant per round-trip-time (by dividing the window size, as usual in the window based protocols), otherwise, if it was smaller than a larger DELAY LOWER BOUND2 constant, the increase in window size per round trip size decreased linearly based on just how large the round trip time was (starting with a constant at DELAY LOWER BOUND all the way down to 0 at DELAY LOWER BOUND 2).

However, the increase only happens if the late_packet_count was below a certain threshold, to avoid responding to outliers and being more robust.

Evaluation:

Tweaking the implementation by adjusting the parameters gave us a relatively respectable best score of -3.69 on the training data. We believe it could have been a bit better if not the delay spike (and subsequently low bandwidth utilization) at the beginning, that arises because the components are started up asynchronously and our window-based algorithm does not continue sending a lot of packets if the acknowledgements are not coming back.

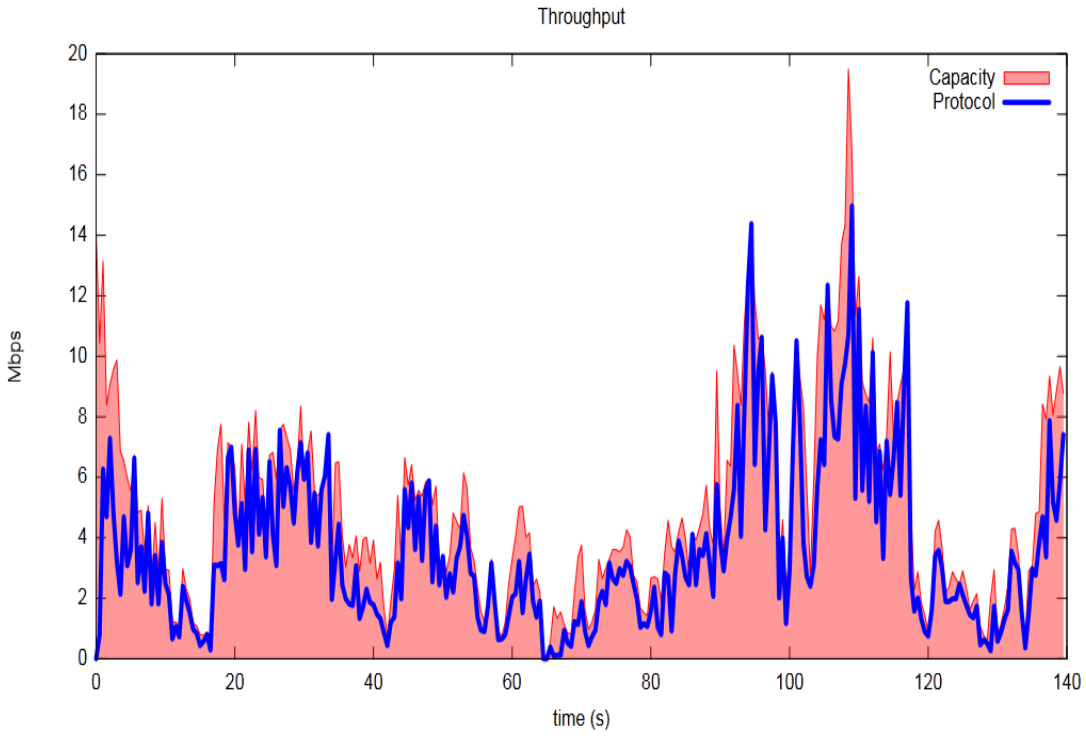
We have also submitted the link to the repository on Piazza. Here are the plots from the latest version (we chose to have this version because we thought it would be more reliable on the general data than the version with the score -3.69).

Here are the resulting statistics:

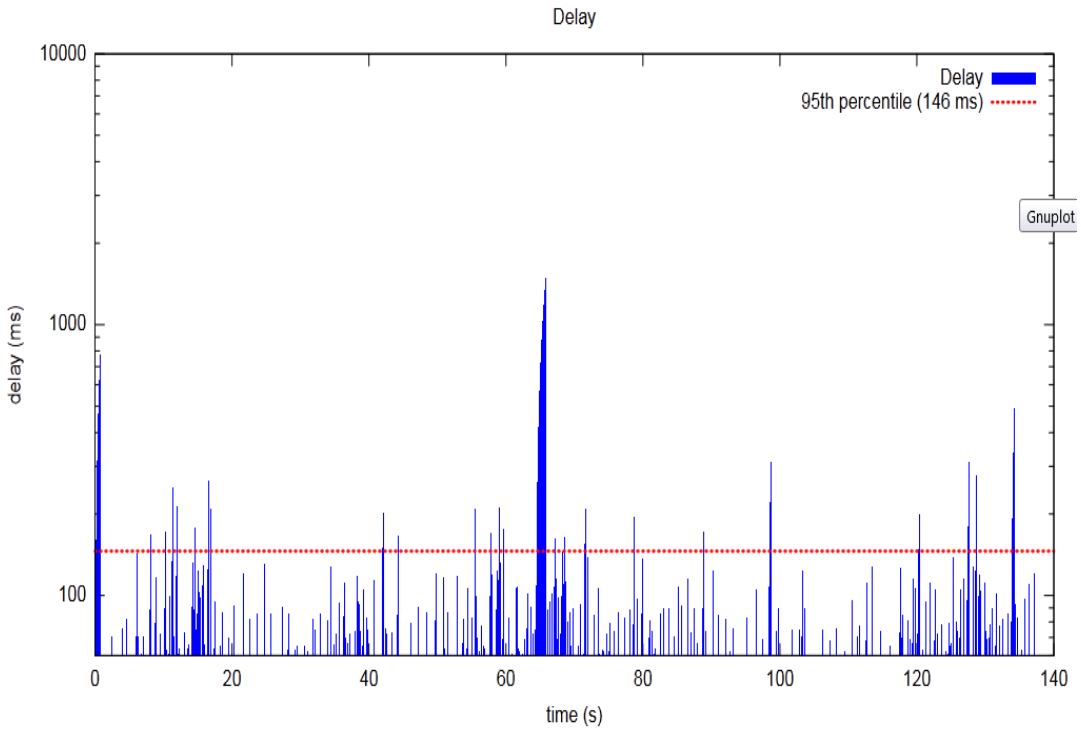
Summary statistics

log(throughput/delay) score:	**-3.70**
Skype's score:	-7.20
Sprout's score:	-3.45
Best possible score:	-2.31
Delay (95th percentile):	146 ms
Protocol throughput:	3.61 Mbps
Network capacity:	5.08 Mbps
Utilization:	71.2%

The throughput plot:



And the delay plot:



Thoughts:

The preemptive approach turned out to be respectable, but we did expect more out of it. Probably, we spent a lot of time optimizing its parameters than we should have, but we also faced several unforeseen phenomenon, due to some initial misconceptions and implementation bugs. Once we ironed them out, the performance kept improving, so we kept pushing forward.

We came up with an idea of a hybrid algorithm on Friday, that would essentially be not strictly window based, would maintain the estimate of average speed of buffer dequeue, and based on that, try to manage the buffer size. Based on the fastest observed round trip time and this speed estimate, we would know the baseline speed of number of packets that would not result in buffering, so the algorithm would send that, plus some overhead gradually (distributed over a period of time), that would over this period of time, if the speed remained constant, result in buffering and lead some desired total buffer size (we could easily estimate the actual buffer size based on the number of unacknowledged packets). The idea of having a desired buffer size for a given speed is that when the speed increases, there should be enough packets in the queue to avoid starvation before the algorithm adjusts, and if the speed decreases, the buffer is not too large before the algorithm cuts down the sending speed. We believe it is some modifications of this approach / algorithm that achieve very good scores on the problem, but unfortunately it required the complete redesign and reimplementation of our code, and it was too late and too risky.

What is very interesting though, is the potential to couple the preemptive reaction idea with this algorithm - in a sense, to let the preemptive algorithm govern the variance around the desired buffer size, which we have seen it is good at. The buffer-control algorithm, on the other hand, would give finer control over the throughput/delay threshold, because it would be operating with clear goals in mind - being almost optimal in the stable case, by having only a small desired buffer, on the other hand, having a safeguard in the form of this buffer to not incur high delay and low throughput in the hands of rapidly changing environment (starve if speed increases, congest if speed decreases, bound how much essentially). We believe that the hybrid algorithm has a very good potential, and we plan to try it out later independently after the contest.