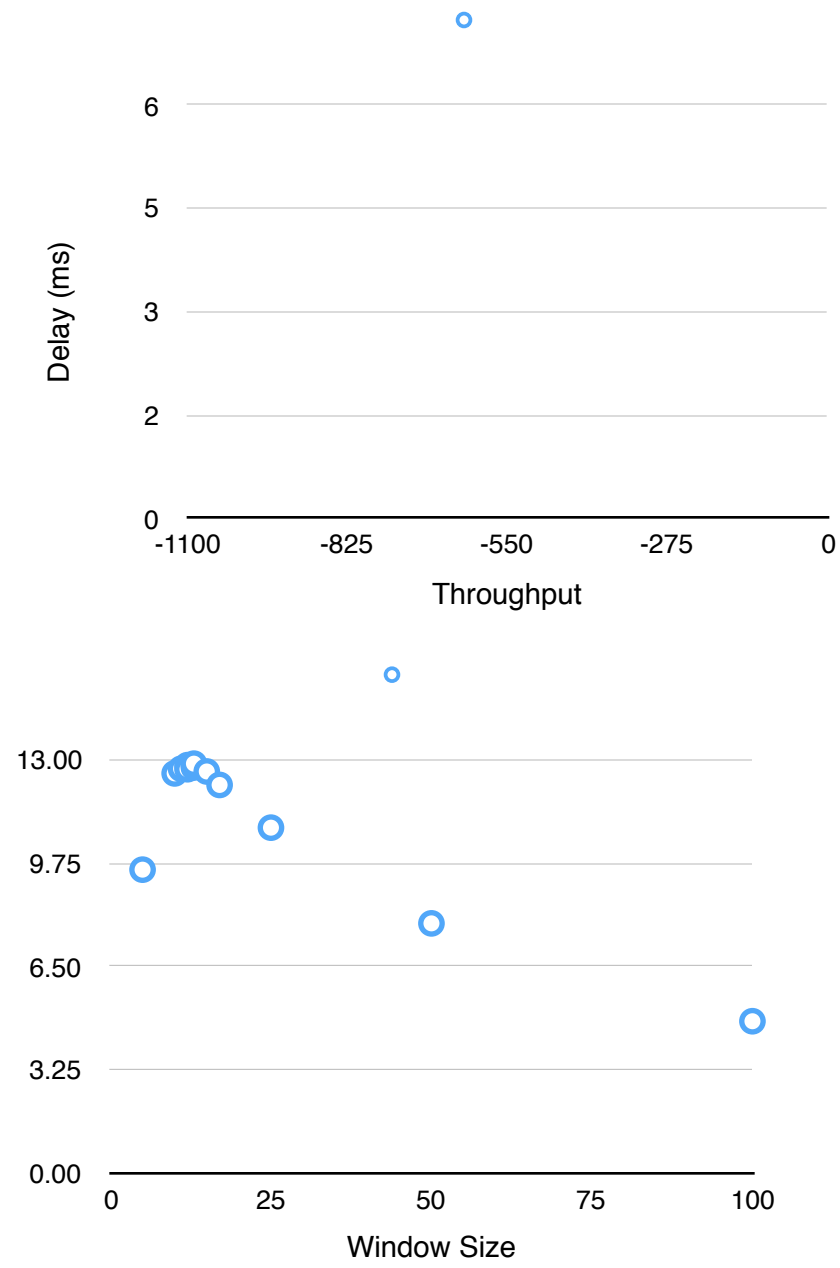
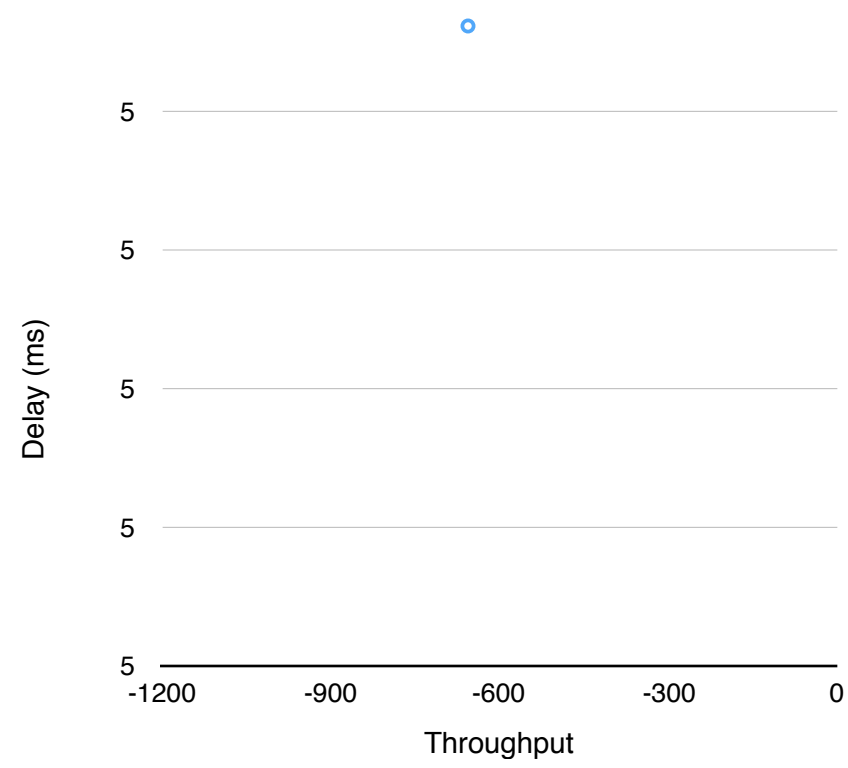
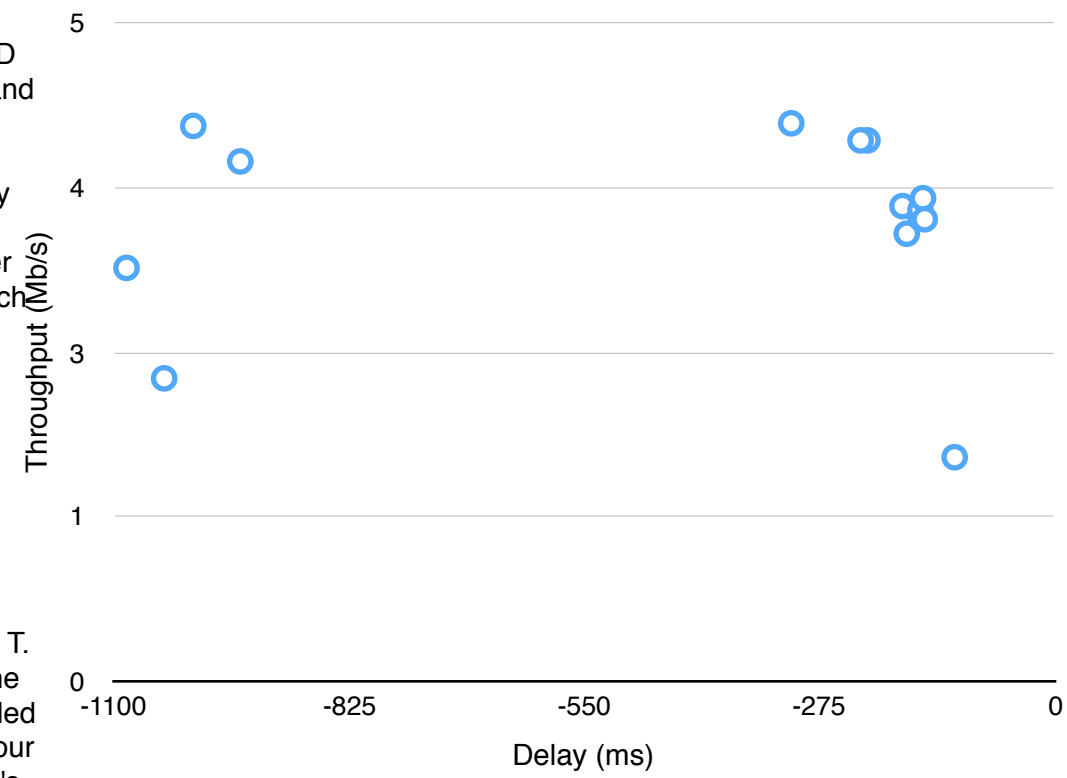

6
5
3
2
0
Delay (ms)
-1100
-825
-550
-275
0
Throughput

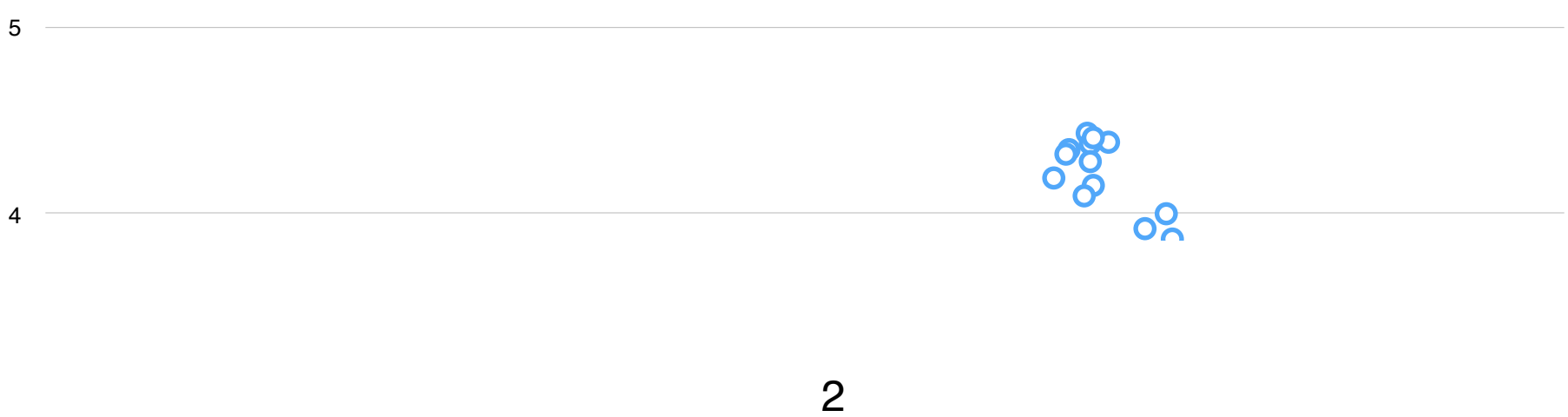
13.00
9.75
6.50
3.25
0.00
Power Score
0
25
50
75
100
Window Size

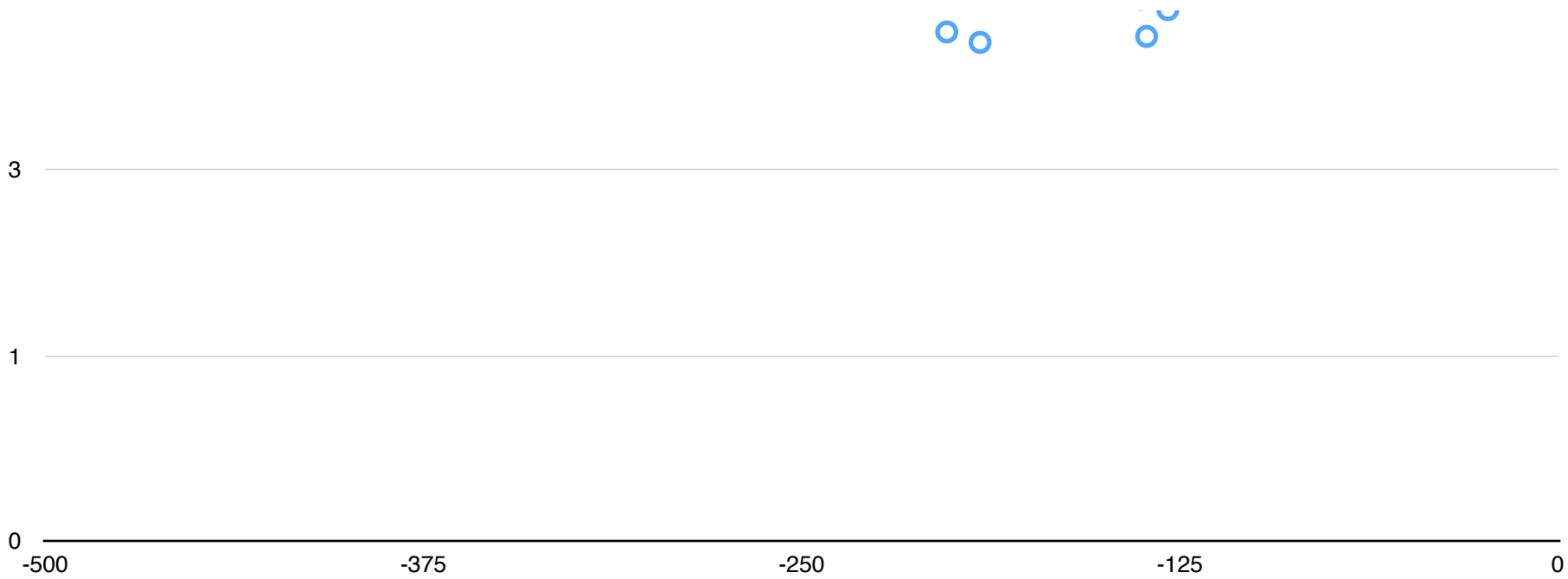

5
5
5
5
5
Delay (ms)
-1200
-900
-600
-300
0
Throughput

Exercise C:
We implemented a delay based congestion, initially with AIMD triggered by delay over/under a constant value of 100, 150, and 300ms. These initially scored fairly low, ~ 4.
Out initial modifications were inspired by BBR, in that we wanted to gain signal about network conditions rather than try to fill the window until packets get dropped, as TCP does.
- First we collected the delays (sender RTT received - sender packet sent time), and used this to estimate our minRTT, which on the test data was ~46ms. We decided to make the simplifying assumption that propagation delay was relatively static. Updating our minRTT estimate mid-flow would necessarily require that we have a period of sub-optimal throughput to probe that characteristic.
- For each ack received with a delay less than the min * a threshhold value T, we increase the window size by 1. This gives us
very good exponential growth to quickly find the bandwidth limit. Experimentally, we found 1.5 to be the optimal value for T.
- We tried multiple methods of recovery. Initially, we halved the window, but this behavior was too aggressive. Also, it discarded the information about at what size window the delay passes our threshhold. These approaches elevated the score into the 10's
- Out next approach was to preserve the that value as a "window_estimate", while backing the window size off by a multiplicative factor to allow the queue to drain. (we settled on a factor of .75). If we continue to have high RTT's while backing off the window (i.e. window < window_estimate), then we adjust our window estimate downward. We floor the window at a nonzero constant to prevent the backoff from choking off the flow.
- This window estimate was initially based on the time that the ack was received. We then shifted to storing the window size for every sent packet, and retrieving it based on the sequence number we received an ack for, for a more accurate estimate.
- Finally, we implemented a handler that notifies the controller if a timeout occurred, and use that to back off the window estimate.



Exercise D:
We tried several approaches in parallel, refining both the AIMD and delay-based controllers to scores in the low 20's. Our design was inspired by BBR, in that we wanted to interpret the singals we're receiving to estimate the condition of the flow and respond to avoid congestion. As such, we have three implicit states
  - exponential growth when we're receiving ack's with low delay
  - recovery (queue drain) when we cross a delay threshold
  - backoff if recovery fails to mitigate delay (i.e. throughput declining)
We also made an attempt to estimate bandwidth at the receiver based on the spacing of the recv_timestamp_acked, but did not come up with a predictive algorithm that substantially improved performance.
Our final submission takes the approach outlined in part C, and further improves upon it:
- We noticed that the delay-based increase would stall if recovering from a network outage. We would periodically resend on timeout, but the first ack's to received after the outage would have artifically long delay times (with the outage included), and we wouldn't begin exponential increase until the queue had cleared and we received ack's from the first packets to be sent after the outage. So the recovery would be delayed by the time to drain the queue.
- Because we actively avoid filling the queue, a timeout is more likely from a network outage or packet drop than a full queue. So on timeout, we save the previous window size, and temporarily decrease the timeout interval to probe for when the network returns. When it does, we restore the previous window, and increase it linearly while masking the delay signal until the pre-outage queue is clear.




3
1
0
-500
-375
-250
-125
0

Ex 1

Window Sz	Throughput	Signal delay	Power	Signal delay		
5	1.06	111	9.55	-111	5	
10	1.95	155	12.58	-155	10	
11	2.10	165	12.73	-165	11	
12	2.26	176	12.84	-176	12	0.05
12	2.26	176	12.84	-176	12	0.06
12	2.26	176	12.84	-176	12	12.82
12	2.26	176	12.84	-176	12	12.83
12	2.26	176	12.84	-176	12	
12	2.25	177	12.71	-177	12	
13	2.40	187	12.83	-187	13	
13	2.40	188	12.77	-188	13	
13	2.41	187	12.89	-187	13	
15	2.68	212	12.64	-212	15	
17	2.92	239	12.22	-239	17	
25	3.74	344	10.87	-344	25	
50	4.77	607	7.86	-607	50	
100	5.02	1051	4.78	-1051	100	

Ex 2

a	b	Throughput	Delay	
1/window	**1/2**	4.73	1137	-1137
	1/4	4.72	1133	-1133
	1/8	4.72	1135	-1135
1/2*wnd	**1/4**	4.63	942	-942

Ex 3

threshold	a	b	Throughput	Delay		

threshold	a	b	Throughput	Delay		
	1/cwnd	**1/4**	4.22	1007	-1007	4.19066534260179
	1/cwnd	**1/2**	4.22	1007	-1007	4.19066534260179
150ms	**1/cwnd**	**1/4**	3.14	1085	-1085	2.89400921658986
100ms	**1/cwnd**	**1/2**	2.30	1041	-1041	2.20941402497598
			1.7	117	-117	14.5299145299145
			4.24	308	-308	13.7662337662338
			3.4	173	-173	19.6531791907514
			3.61	178	-178	20.2808988764045
			3.52	157	-157	22.4203821656051
			3.58	157	-157	22.8025477707006
			3.67	154	-154	23.8311688311688
			3.51	152	-152	23.0921052631579
			4.11	219	-219	18.7671232876712
			4.11	227	-227	18.1057268722467
					0	
					0	

Ex 4

threshold	a	b	Throughput	Delay	Plot Delay	Power
			3.36	191	-191	17.5916230366492
			3.43	202	-202	16.980198019802
			4.29	157	-157	27.3248407643312
			4.18	163	-163	25.6441717791411
			4.22	156	-156	27.0512820512821
			4.15	164	-164	25.3048780487805
			3.65	138	-138	26.4492753623188
			3.75	131	-131	28.6259541984733
			3.58	129	-129	27.7519379844961
			3.4	136	-136	25
			4.23	150	-150	28.2
			4.10	156	-156	26.2820512820513
			4.26	155	-155	27.4838709677419
			3.94	155	-155	25.4193548387097
			3.87	158	-158	24.493670886076
			3.99	168	-168	23.75
					0	