Congestion-Control Algorithms on an Emulated Cellular Network

CS 244 PA2 Writeup

Teamname: `P4B4';DROP TABLE glad_you_have_sanitization`

Github Repo: `https://github.com/jxguan/cs244-pa2`

Hope Casey-Allen
hcaseyal@stanford.edu

Jiaxin Guan
jxguan@stanford.edu

1 Fixed Window Size

1.1 Varying the Window Sizes

We created a script to vary the fixed window size and output results for average capacity, average throughput, 95th percentile per-packet queueing delay, and 95th percentile signal delay (see `datagrump/warmup-a.sh`). We took measurements with windows sizes of 1, 2, 5, 10, 20, 50, 100, 200, 500 and took them over 3 runs each to estimate the repeatability of the measurements. The results we had are shown in figure 1.

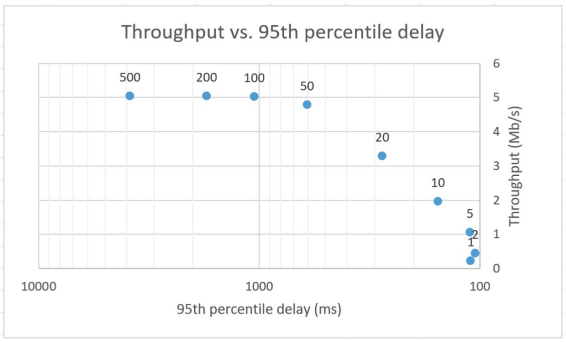


Figure 1: The performance for different fixed window sizes. Data Labels represent the corresponding fixed window size.

We can observe an overall pattern that by increasing the window size, we're able to obtain better network utilization, but at the same time getting higher delay. This matches our expectation as we would expect the queue to be full and the network fully utilized for large fixed window sizes.

Another interesting observation is that when increasing the window size from 1 to 2, the throughput increases, while at the same time the delay also slightly decreases. This is interesting in that normally we would expect the delay to increase as we increase the window size. We suspect that it is because with a larger window size 2, which is still way smaller than the network capacity, more packets can be sent through the network when the network is not at all congested. In this way, we have more packets with low delay for window size 2, and hence a lower average delay.

Notice that this reasoning only applies to window sizes that are way smaller than the max network capacity, because with relatively large window sizes, the network is likely often fully utilized - increasing the window size would not lead to more packets being delivered with low delay, but rather it will lead to more packets waiting in the queue when the network is congested, and hence a higher average delay.

1.2 Best Fixed Window Size

By examining the results from section 1.1, we observed that the window size that maximized the overall score was between 5 and 20. We then reran the script, but this time varying the input window size parameter between 5 and 20 to perform a ternary search to find the exact best window size. The resulting best fixed window size that we found is 12, with a score of 12.84 using the `throughput/delay` function on the contest page (or 3.68 if using the log(`throughput/delay`) function on the assignment handout). This is not a great result, but it is more than 10 times better than the performance for fixed window size 1.

1.3 Repeatability

We calculated the standard deviation for our results for the best fixed window size over 10 runs (see table 1). We can see that the standard deviations are very low when compared to the averages, showing that the measurements are extremely repeatable. This matches our expectation because all of the code is deterministic. The only source of variations is likely the fluctuation of the machine performance or different scheduling decisions.

	StdDev	Avg
Throughput (Mbits/s)	0.005	2.263
Delay (ms)	0.333	175.889
Score	0.046	12.868

Table 1: The standard deviations and averages of the throughput, delay and overall score.

2 Additive Increase, Multiplicative Decrease (AIMD)

2.1 Implementation

We implemented the AIMD scheme in `datagrump/controller.cc`. We made a slight modification that we perform a multiplicative decrease whenever the RTT exceeds a specific timeout. This simulates the situations where packets get lost in transmission and are resent after a timeout (conventionally $2 * \texttt{RTT}$). We derived the actual RTT by using the timestamps in the `ack_received` function per the Piazza post. We tested the implementation first with arbitrarily set constants, which gave us a score around 18, which is a significant improvement over the fixed window size algorithm.

2.2 Choosing the Constants

In order to choose the constants that yield the best performance, we first wrote a script to try different constants of additive increase size and multiplicative decrease factor. We were able to get the result shown in table 2.

	1	2	4	8	16
0.2	18.52	18.55	14.97	12.91	10.65
0.4	18.71	18.75	14.74	12.79	11.06
0.6	17.37	18.67	14.95	12.98	10.63
0.8	17.99	18.33	15.30	13.24	10.76

Table 2: The scores for different additive increase sizes and multiplicative decrease factors. The rows are the different factors for the multiplicative decrease, and the columns are the factors for the additive increase.

We can observe from the table that the optimal performance occurs with an additive increase size of 2 and a multiplicative decrease factor of 0.4. So we fixed these two constant and varied on the initial window size, which yielded the following result in figure 2.

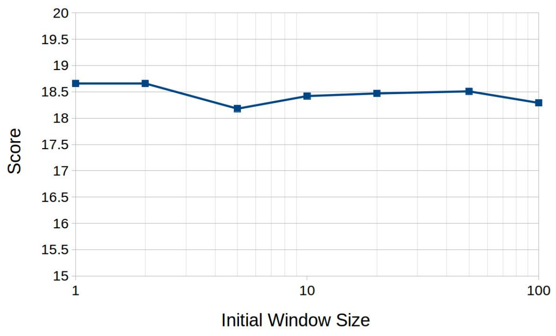


Figure 2: The distribution of scores over different initial window sizes.

We can see from the figure that the score is not influence much by the initial window size. It merely fluctuates around 18.5. This meets our expection, as the initial window size only matters for the very first few clock cycles. Hence, we picked the initial window size to be 2, which yielded the highest score in our testing.

Notice that we did not tune the timeout constant, mainly for two reasons:

1. In an AIMD scheme, the timeout is conventionally simply $2 * \texttt{RTT}$. With the average RTT estimation of 105 ms from section 1, we set the timeout to be 210 ms.

2. Notice that since our timeout is based on the timestamp when the ACK is received by the sender and the timestamp when the sender sends the packet, it essentially the delay used in delay-triggered schemes. Therefore, we will tune the threshold later in section 3.

Bringing these parts together, the final constants that we choose for the AIMD scheme are:

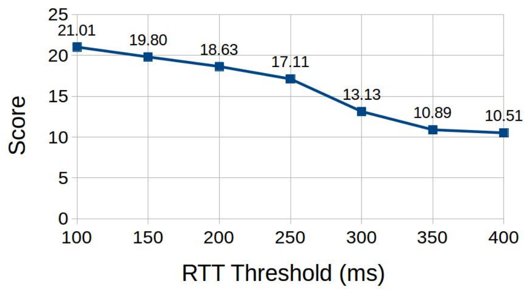


Figure 3: The score distribution over different RTT thresholds.

```
INITIAL_WINDOW_SIZE = 2
ADDITIVE_INCREASE_SIZE = 2
MULT_DECREASE_FACTOR = 0.4
MD_TIMEOUT = 210
```

These constant yielded a score of 18.66, which is around 1.5 times the fixed window size algorithm.

2.3 Analysis

Overall, the AIMD scheme that we implemented did quite good in utilizing the network with a network utilization of almost 90%. However, it's not doing so well in signal delay, the average of which is well above 200 ms. Since the score is calculated as `throughput/delay`, with such a high delay, even if we could achive 99.9% utilization, the score would still be capped at around 25. We would definitely need some other, not necessarily more sophisticated, algorithm to achieve high throughput and low delay.

3 Delay-Triggered Schemes

Since in the AIMD implementation we used received ack timestamps to determine if a timeout occurred, the AIMD scheme we have implemented in section 2 is essentially a delay-trigger scheme by itself. While we've tuned the AIMD constants in section 2, in this section, we will be tuning the RTT threshold for the delay, as well as exploring other options to update the window size.

3.1 Tuning the Threshold

Again, we set up a script to test the score under differnt thresholds. The results are shown in figure 3.

We can see an overall trend of decreasing score as the RTT threshold increases. Therefore, we also explicitly tested the case where the RTT threshold is 50 ms. Since the lowest possible RTT (propagation delay) is around 42 ms, the threshold of 50 ms yielded very bad utilization, as the AIMD scheme is constantly switching between the additive increase phase and the multiplicative decrease phase. It was easy to tell that the 50 ms threshold has a dissatisfying score (we did not even bother to wait for it to finish as we can already tell from the ongoing graph).

Therefore, in the end, we chose the RTT threshold to be 100 ms, which gave us the best result with a score of 21.01.

3.2 Additive Increase, Additive Decrease (AIAD)

We also experimented with different ways to update window size based on the delay.

One thing that we tried is an AIAD scheme with both the increase size and the decrease size being 2. We can see from the result in figure 4 that there is a very high utilization of 96.6%. But by using a additive decrease rather than multiplicative decrease, this algorithm's reaction to high delays is really slow, leading to an average signal delay of 361 ms. The overall score given by this algorithm is only 13.49, merely surpassing the fixed window size algorithm.

3.3 Multiplicative Increase, Multiplicative Decrease (MIMD)

As a complement to section 3.2, we also experimented with an MIMD scheme with the multiplicative increase factor set to be 1.1 and the multiplicative decrease factor set to be 0.5. From figure 5, we can see that, as we have expected, the multiplicative increase is too aggressive, sending way too many packet than what the network can handle, leading to a very high signal delay. Also from figure 6, we can clearly see that all the high delays are caused by the spikes of sent packets during the multiplicative increase phase. However, the utilization seems quite good with a 90.6% utilization rate, because there are always tons of packets waiting to be delivered in the queue. But since the delays are so high, the overall score for this MIMD scheme is only 0.1, by far the third to last on the "leaderboard".

3.4 Lesson Learned

Neither the AIAD and the MIMD scheme seems ideal, so in order to achieve a better performance,

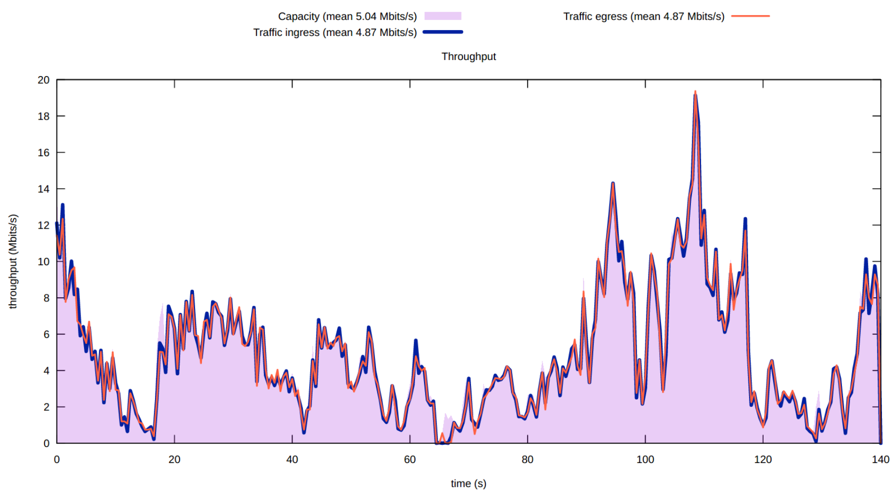


Figure 4: The result of an AIAD scheme with both additive increase size and additive decrease size equal to 2.

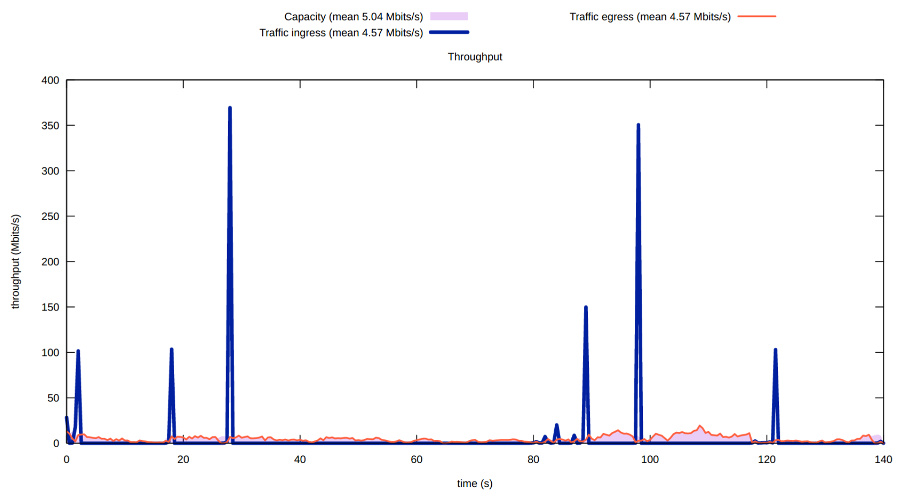


Figure 5: The result of an MIMD scheme with multiplicative increase factor 1.1 and multiplicative decrease factor 0.5.

we would probably want to start from the AIMD scheme and see what optimizations we can add to it.

4 The Contest

4.1 Starting Out: Researching Related Work

We started our approach by watching the Sprout video and skimming the Sprout source code and the Sprout paper (Winstein et al., 2013). We really liked the idea that Sprout presented of predicting network capacity changes in order to achieve high throughput and low delay.

We also read *Timely: Rtt-based Congestion Control for the Datacenter* (Mittal et al., 2015) and liked the idea of using delay gradients to adjust transmission rate. We also appreciated how it is simlar to the AIMD scheme by endorsing the additive increase and multiplicative decrease phases. However, we were aware that we may not be able to make RTT measurements with microsecond accuracy and thus the delay gradients may not be totally sufficient to estimate switch queueing.

4.2 Delay Gradient Scheme

The first scheme we tried was based off of the delay gradient scheme presented in the *Timely* paper (Mittal et al., 2015). We took advantage of the following pseudocode from the paper:

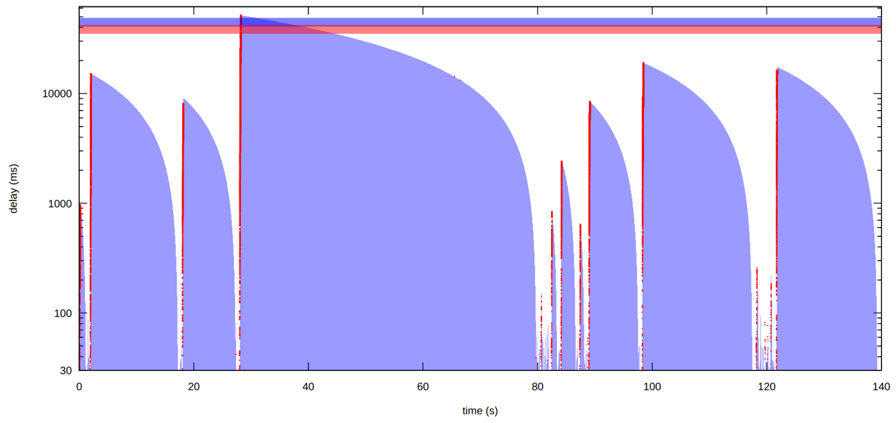


Figure 6: The delay over time for the MIMD scheme with multiplicative increase factor 1.1 and multiplicative decrease factor 0.5.

Algorithm 1: TIMELY congestion control.

```
Data: new_rtt
Result: Enforced rate
new_rtt_diff = new_rtt - prev_rtt ;
prev_rtt = new_rtt ;
rtt_diff = (1 - $\alpha$) · rtt_diff + $\alpha$ · new_rtt_diff ;
                                        ▷ $\alpha$: EWMA weight parameter
normalized_gradient = rtt_diff / minRTT ;
if new_rtt < $T_{low}$ then
    rate ← rate + $\delta$ ;
                                        ▷ $\delta$: additive increment step
    return;
if new_rtt > $T_{high}$ then
    rate ← rate · $\left(1 - \beta \cdot \left(1 - \frac{T_{high}}{new\_rtt}\right)\right)$ ;
                                        ▷ $\beta$: multiplicative decrement factor
    return;
if normalized_gradient $\leq 0$ then
    rate ← rate + N · $\delta$ ;
        ▷ N = 5 if gradient<0 for five completion events
    (HAI mode); otherwise N = 1
else
    rate ← rate · (1 - $\beta$ · normalized_gradient)
```

The Delay Gradient Algorithm described in *Timely: Rtt-based Congestion Control for the Datacenter* (Mittal et al., 2015)

4.2.1 Initial Implementation

The Delay Gradient scheme works by looking at the change in RTT over time. There are 4 main states.

1. If the change in RTT, the delay gradient, is negative or 0, then this means that that the queue is draining, and we can try to send more packets by increasing our window size.

2. If the gradient is positive, then the network is becoming congested as packets are spending longer and longer time in the queue, so we decrease our sending rate. The last two stages are for the extremes of a very empty queue or a very congested queue.

3. If the most recent rtt of a packet is less than some threshold T_{low}, then we aggressively increase the sending rate.

4. If the most recent rtt is greater than some threshold T_{high}, which is the highest RTT that we will accept, then we aggressively decrease the sending rate.

In the very original run right after we implemented the delay gradient scheme with arbitrarily picked constant, we got a score of 12.51 (with HAI enabled), which is even worse than the AIAD scheme. However, we understand the importance of tuning, so we were quite optimistic with the delay gradient scheme.

4.2.2 Tuning the Parameters

We observed that the delay plays an important role in the scoring function and that the highest scoring teams had the lowest delay, so we prioritized that when adjusting parameters. Specifically, we decreased our threshold high parameter.

We initially used Hyperactive increase (HAI) to more aggressively increase the sending rate after a period of slow growth (where the gradient is negative for 5 completion events). This enabled us to get higher throughput at the cost of higher delay. After doing some math on the scoring function, we decided that the best strategy is to maintain a relatively low delay and then try to maximize the throughput. Thus, we ended up disabling HAI for a less aggressive increase to achieve a lower delay.

By looking at the graphs, we saw that there were times when the packet transmission rate dipped to near 0, so we lower bounded the packet transmission rate by 1.

We measured the lowest RTT time observed in the network to be 42 ms, and correspondingly set the `minRTT` parameter to be 50 ms. We set the initial rate to be 50 so that the algorithm can get a faster head start in transmitting packets. We set initial `rtt_diff` to be 0 so that it can be updated more quickly to reflect the gradient of the first few packets.

The rest of the parameters left for us to set are α which controls the EWMA weight, β which is the mutiplicative decrease factor, δ which is the additive increase size and T_{low} and T_{high} the two delay thresholds.

After adjusting these parameters several times, we were able to reach a score of 30.15 by setting $\alpha = 0.25$, $\beta = 0.4$, $\delta = 0.1$, $T_{low} = 80$ and $T_{high} = 150$.

We then removed the "lowerbounding the packet transmission rate by 1" part, but our score decreased because we had long periods of time of 0 packet transmission, which led to a slightly lower throughput with approximately the same delay.

We then ran a script overnight to vary δ, T_{low}, and T_{high}, and picked the constants producing the best score. One intersting thing that we noticed was that (as shown in figure 7) the distribution of score over δ follows a similar pattern for different choices of T_{low} and T_{high}, but the optimal choice of δ depends on our choices of T_{low}, and T_{high}.

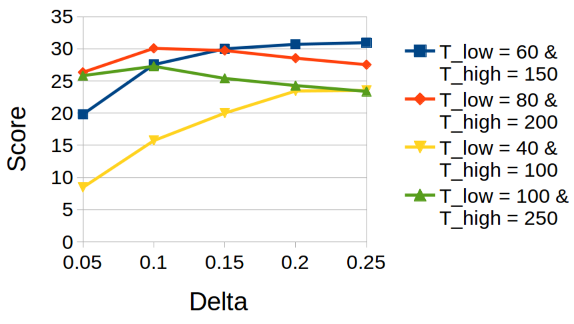


Figure 7: The distribution of score over δ for fixed choices of T_{low} and T_{high}.

We also noticed that the max score achieved is at the boundary value of what we tried with $\delta = 0.25$ and it is likely that the optimal choice for δ is greater than 0.25. Therefore, we went ahead and fixed T_{low} and T_{high}, and adjusted δ from 0.1 all the way to 0.9. As shown in figure 8, indeed we reached the maximum score between $\delta = 0.2$ and $\delta = 0.3$. Hence, we picked $\delta = 0.25$ as our choice.

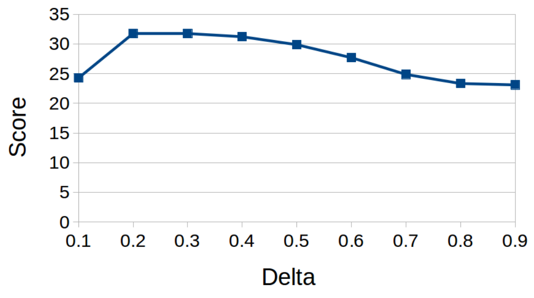


Figure 8: The distribution of score over δ for the optimal choice of T_{low} and T_{high}.

After getting the optimal choices with $\delta = 0.25$, $T_{low} = 60$ and $T_{high} = 150$, which yielded us a best score of 30.95, we fixed these three parameters, and started tweaking α and β using `datagrump/tweak_timely_constants2.sh`. This gave the best choices for $\alpha = 0.8$ and $\beta = 0.3$ with an overall score of 32.85.

We then noticed that the distribution of score over these five parameters is likely a mesh in a five-dimensional space. Picking the maximum value by adjusting only two or three parameters would not give us the best result, but rather we need to tweak the parameters in multiple iterations. So we started to tune the different parameters alternatively until the max score becomes stable and no longer increasing. Interestingly, it turned out that the choices we had were already the optimals.

This brought us to our final choice of the parameters with $\alpha = 0.8$, $\beta = 0.3$, $\delta = 0.25$, $T_{low} = 60$ and $T_{high} = 150$, which gave us a score of 32.85.

4.2.3 Analysis

By looking at the result of our Delay Gradient Scheme, we find out the bottleneck to a higher score is likely the delay. We still had a signal delay of 124 ms, while the other contestant on top of the leaderboard had signal delays of 80 ms. By examining figure 9, we noticed that our high delay was caused by sudden drops of the network capacity. In order to deal with that, we would want our algorithm to adjust more quickly to sudden congestions in the network. We realized that the feedback loop for the delay is quite long, and hence we started to look into other ways to adjust to the network capacity. We believed that the delay is not

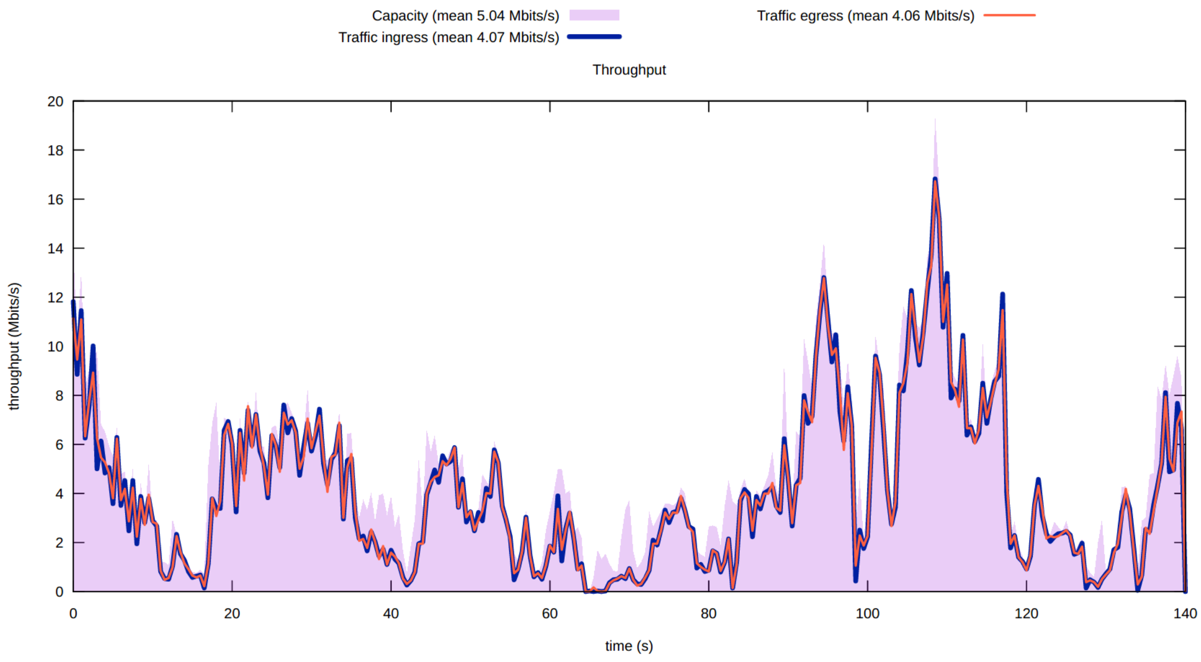


Figure 9: The result of our delay gradient scheme with the optimal paramters.

a straightforward discription of the network status, but rather it is the number of packets in the queue that really matters. Bearing that in mind, we turned to the Sprout paper (Winstein et al., 2013) for optimization ideas.

4.3 Inspirations from Sprout

We drew inspiration from Sprout to use the number of packets in the queue as an indicator of how congested the network is. This is in correspondence to Sprout's algorithm where the estimations are based off the number of bytes sent and received. In our case, all of the packets are fixed sized, so simpling using the number of packets is sufficient. We achieved that by calculating the difference between the seq number in the latest ack and the the seq number of last sent packet.

We started off by simply replacing the delay gradient algorithm with a num-packets-in-queue gradient algorithm. Instead calculating the gradient of the RTTs, we calculated the gradient of the number of packets in flight. Without any tuning of the parameters, we were able to get the result as shown in figure 10. We can see that it has an extremely high utilization of 97.5%, but also a very high signal delay of over 700 ms.

We then tuned the parameters (the two thresholds to be exact) and tested how this scheme works under different choices of the parameters. It turned out that whenever we are able to obtain a lower delay, the throughput also drops accordingly. In that way it is very hard for us to make any improvements on the score by simply tweaking the parameters.

We noticed that a major issue in this scheme is the two thresholds. While the thresholds make sense for bounding the delay within a certain region, it does not make sense to bound the queue size to a specific region, as the network capacity can vary from 0 to very high.

Therefore, we modified our scheme to still use the two thresholds for the delay, only using the number of packets in queue for the gradient. This managed to give us an extremely low delay of only 78 ms, but the utilization is also terribly low (only 13.8%). This portion of code is checked in at the `hybrid` branch of our Github Repo.

4.4 Future Work

Given that the tuning of the parameters takes so much time, we weren't able to tune our packets-in-the-queue algorithm to its best performance. We suspect that it could potetially perform better than the delay gradient algorithm, since we were able to reach a lowest-ever delay, but whether we were able to reach good utilizaiton with the scheme remains unknown. It would be interesting to tune out the parameters to see if this hybrid scheme could outperform Timely or Sprout.

5 Conclusions

We were able to reach the maximum score of 32.85 using the algorithm and parameters discussed in section 4.2. The actual code can be found at the `master` branch of our Github repo.

One reflection that we had when doing this project is how the scoring function could be not indicative. We observed that to reach a higher score, the best way is to maintain a low delay and a relatively moderate utilization of the network. Such might not be the case in real life. For example, if one is to do large file transfers over the network, a 100 ms delay and a 200 ms are not going to be huge impacts, but the scores are differed by the factor of 2. We believe that in order to better represent real-life scenarios, the scoring functions should be adpated correspondingly to actual use cases.

References

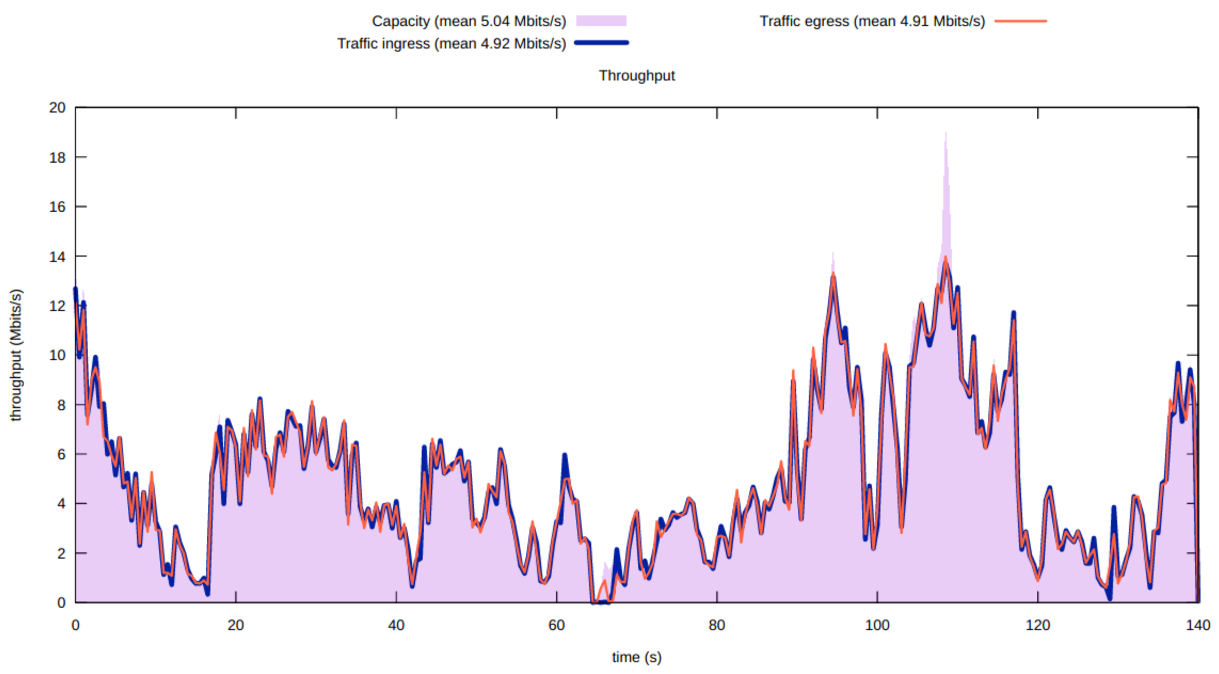
Radhika Mittal, Nandita Dukkipati, Emily Blem, Hassan Wassel, Monia Ghobadi, Amin Vahdat, Yaogong Wang, David Wetherall, David Zats, et al. 2015. Timely: Rtt-based congestion control for the datacenter. In *ACM SIGCOMM Computer Communication Review*. ACM, volume 45, pages 537–550.

Keith Winstein, Anirudh Sivaraman, Hari Balakrishnan, et al. 2013. Stochastic forecasts achieve high throughput and low delay over cellular networks. In *NSDI*. pages 459–471.

A Glossary

Here is a glossary of files and folders that we added to the `datagrump` folder.

- `calc_max_score.go`: A Golang script to compute the scores of outputs and output the best score.
- `contest_graph.ods`: A LibreOffice Calc file that we used to generate the graphs for the contest.
- `output_a/`: The folder containg the outputs generated when tuning the different parameters for warmup exercise A.
- `output_b/`: The folder containg the outputs generated when tuning the different parameters for warmup exercise B.
- `output_c/`: The folder containg the outputs generated when tuning the different parameters for warmup exercise C.
- `output_contest/`: The folder containg the outputs generated when tuning the different parameters for the contest.
- `PA2_partA_graph.xlsx`: A excel file that we used to generate some of the graphs in warmup exercise A.
- `tweak-timely-constants.sh`: A shell script to tweak the values of δ, T_{low} and T_{high} for the delay gradient algorithm.
- `tweak-timely-constants2.sh`: A shell script to tweak the values of α and β for the delay gradient algorithm.
- `tweak-timely-constants3.sh`: A shell script to tweak the values of T_{low} and T_{high} for the delay gradient algorithm.
- `tweak-timely-constants4.sh`: A shell script to tweak the value of δ on fixed T_{low} and T_{high} for the delay gradient algorithm.
- `warmup-a.sh`: A shell script to adjust the fixed window sizes for warmup exercise A.
- `warmup-b.sh`: A shell script to adjust the additive increase size and multiplicative decrease factor for the AIMD scheme in warmup exercise B.
- `warmup-b2.sh`: A shell script to adjust the initial window size for the AIMD scheme in warmup exercise B.
- `warmup-c.sh`: A shell script to adjust the delay threshold for the AIMD scheme as a delay-triggered scheme in warmup exercise C.
- `writeup/`: The LaTeX files and images to generate this writeup.



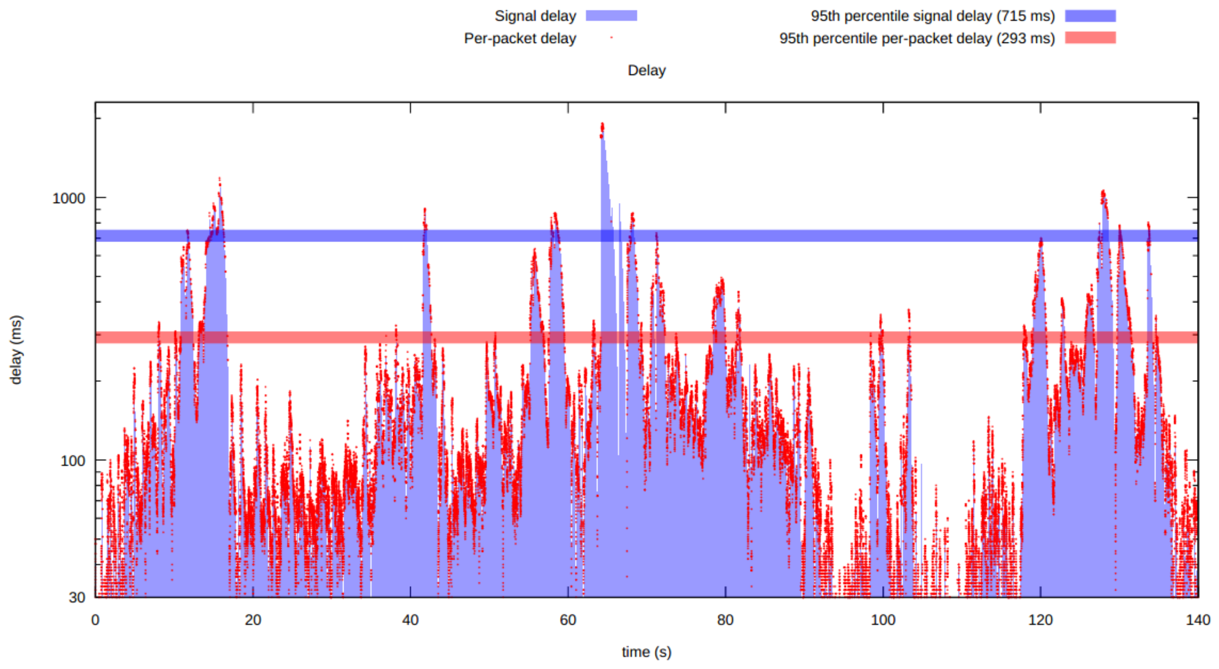


Figure 10: The throughput and delay graph for the baby num-packets-in-queue gradient algorithm.