6.829 Problem Set 2

Victor Domene
victordomene@college.harvard.edu

Gabriel Guimaraes
gabrielguimaraes@college.harvard.edu

October 26, 2016

1 Fixed Window Size

1.1 Varying the Window Size

On an Ubuntu machine in AWS, we ran the contest experiment three times for several values of the window size between 1 and 50. The scores were computed as

$$\ln\left(\frac{\text{Throughput}}{\text{Signal Delay}}\right)$$

obtaining the results below (Figure 1), where the values of the throughput and signal delays have been averaged over the three runs for each window size. We conducted more measurements in the 10 to 20 range, since it seemed more promising than lower or higher values, as the plot suggests. The maximum score was obtained with a window size of 12.

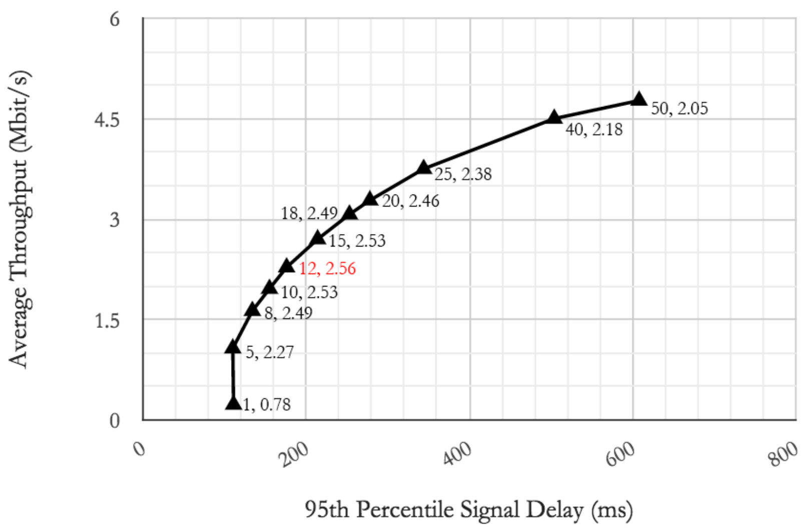

Figure 1: Average Throughput vs. 95th Percentile of Signal Delay.

The graph below (Figure 2) shows the behavior of the score as we increase the window size. We notice that there is a maximum around 12, and that the score deteriorates if we decrease the window size (due to low throughput) as well as if we increase the window size (due to high delay).

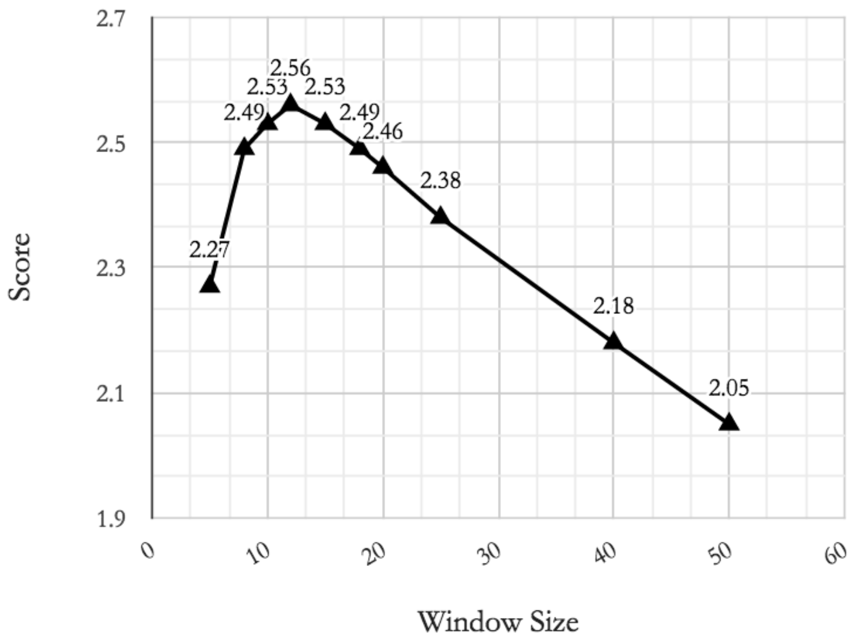

Figure 2: Score, as defined in the beginning of this section, corresponding to different fixed window sizes.

1.2 Repeatability of Experiments

We did not observe significant variation in signal delay, average throughput or the final computed score between each run. Between the three repeated experiments, the maximum difference in score was 0.016, which happened for the window size of 50. Since the window size was fixed, there was no time dependency of any sort. Due to this, combined with the fact that measurements were made over the exact same network traces and in the same environment, we expected this predictability of results.

2 TCP-like AIMD Scheme

In the TCP-like experiment, we implemented an AIMD scheme based on ACK timeouts as a congestion signal: whenever the sender sees a timeout, it triggers a callback in the `Controller`, which will then do the multiplicative decrease portion of AIMD. We intially thought about using duplicate ACKs to help detect congestion, but this experiment is set up in a way that duplicates do not occur. In our best result, we had a score (as calculated by the link at the contest, and not the log-scale score from the previous section) of 22.82. We experimented with the additive increase constant and the multiplicative increase constant, as well as the threshold used to determine a timeout. We describe the results of varying these constants in the sections below.

As in TCP, the additive increase constant is only applied after the number of ACKs received reach the current window size. The multiplicative decrease is applied on every timeout

2.1 Varying AIMD Constants

To assess the influence of the AIMD constants, we fixed the timeout at 30ms. Changing this timeout makes AIMD more responsive to packet drops, which means that it will be able to react to fluctuations faster. Still, it is not too effective – the results are summarized below for different constants.

Add. Inc	**Mult. Dec**	**Throughput**	**95th Signal Delay**	**Score**
0.5	2	0.36	0.68	5.21
1	2	2.94	194	21.94
1	3	2.34	116	20.94
1	4	2.26	99	22.82
2	2	4.13	226	18.27
2	4	3.78	206	18.34

As expected, by increasing the MD constant, the algorithm is more conservative on the signal delay, which yields generally a better score. The trade-off is to yield a smaller, but yet acceptable, throughput. We also require a minimum AI constant, so that the throughput doesn't get damaged unnecessarily.

2.2 Varying Timeouts

To verify the influence of the timeout threshold, we fixed the additive increase constant to 1 and the multiplicative decrease constant to 4 (the best score from the previous section). We discovered that, as expected, changing the timeout used by the sender to continue sending packets significantly affected the score. The pattern observed in delay and throughput graphs suggests that this is because the sender will not be waiting for a packet that has probably been lost, and we can immediately reduce the window size to avoid congestion. In other words, changing the timeout threshold made the system more responsive to packet drops. The results can be observed in Figure 3.

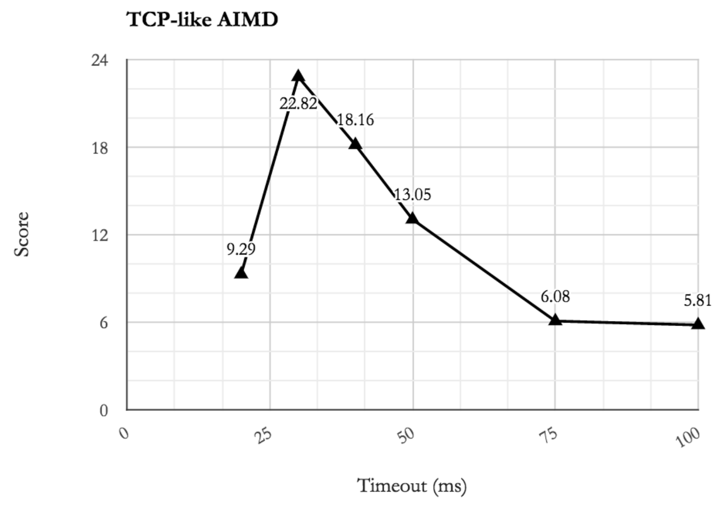

Figure 3: Score corresponding to different timeouts in TCP-based AIMD.

This worked fairly well for this set of experiments, but we expect it not to work so well for different network traces: it is likely an overfit to the packet drop rate of this particular trace. The best score of 22.82 was achieved with a timeout of 30ms.

3 Delay-based Scheme

This scheme works as follows: whenever the measured RTT of a packet is above a threshold, we decrease the window size. Otherwise, we increase the window size. The main design choices here are, then, the policies that determine how to increase/decrease the window size, and the RTT threshold value.

3.1 Varying Control Mechanisms

First, we fixed the RTT threshold to 80ms, inspired by Sprout's threshold and trying to be a little more conservative, and we consider two different policies for controlling the window size:

- AIAD (Additive Increase, Additive Decrease) with constants 1 and -0.1.
- AIMD (Additive Increase, Multiplicative Decrease) with constants 3 and 1.02.
- MIMD (Multiplicative Increase, Multiplicative Decrease) with constants 1.5 and 1.05.

The reasoning behind these constants is that the feedback will be given per RTT, rather than on an aggregate of ACKs, in the TCP-like scheme. Then, the multiplicative decrease constant must be a lot smaller, to avoid damping the throughput too much.

Before the experiments, we expected MIMD to give poor performance, since it "jumps" up and down too quickly, and thus will tend to have higher delay. We conjectured that AIMD would yield lower delay and higher throughput than AIAD. The results of our experiments are summarized below.

Model	**Throughput**	**Signal Delay**	**Score**
AIAD	4.18	142	29.43
MIMD	2.82	152	18.52
AIMD	4.01	112	35.71

Our expectations were mostly met, except that MIMD provided reasonable delay. This is probably due to our cautious choice of constants. We did not spend too much time trying to improve the constants for each scheme, since this will probably lead to overfit.

3.2 Varying RTT Thresholds

We fixed the constants of our AIMD delay-based approach from the previous subsection, namely, the additive increase constant set to 3 and the multiplicative decrease constant set to 1.02. We then varied the RTT threshold used as a congestion signal. The results are summarized in Figure 4.

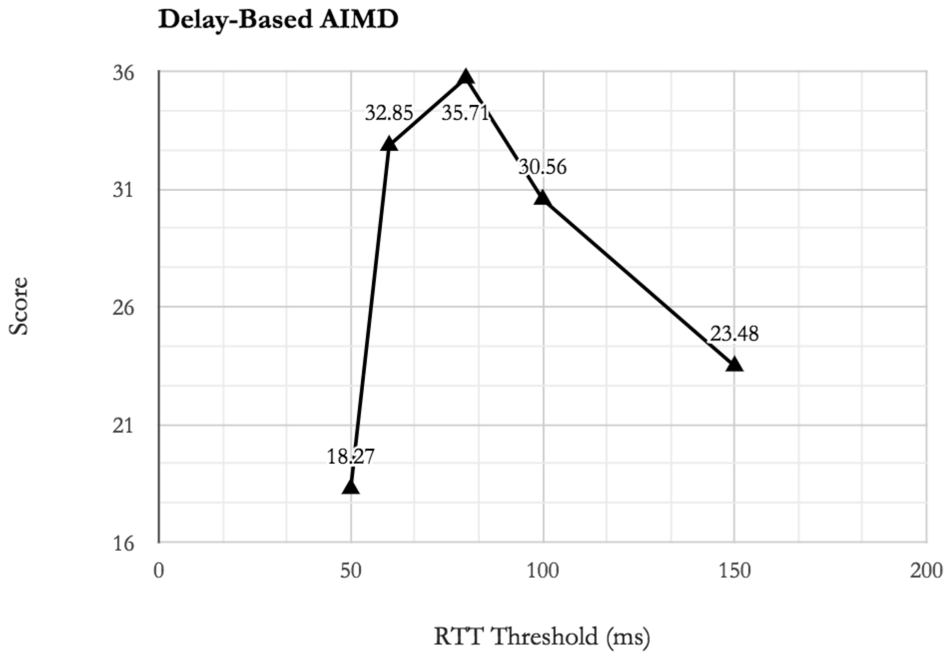

Figure 4: Comparison of AIMD performance across RTT thresholds.

We achieved a highest score of 35.71, much better than the best score obtained in Section 2. The threshold used for that value was 35.71. This result is under the name *victordomene_rtt* on the server.

4 Beating Sprout!

4.1 Delay-based PID Scheme

This model can be found under *gabgl1_p* on the server.

Our first idea was to do a simple improvement of the RTT model, described in the previous section. That model seemed to work fairly well without tuning too many parameters, and so we thought we could get a better, yet robust result, by enhancing it. To do this, we looked at the very standard idea in control systems: Proportional Integral Derivative controllers (PID).

In simple terms, instead of using an AIMD or AIAD protocol to achieve the target RTT threshold, we would change the window size using the PID. This involves having a proportional term, which tends to act proportionally against the error, to decrease it; the derivative term, which tends to act as a "smoother" of the feedback signal (to avoid abrupt increases or decreases in the window size); and the integral term, which accounts for accumulating measurement imprecisions over time.

We did not spend too much time trying to tune the parameters of the PID. We chose $P = 0.2$, $I = 0.00001$ and $D = 0.2$. The integral constant is traditionally much smaller than P and D. This choice of parameters, along with an RTT threshold of 100ms, we got a throughput of 3.59Mbit/s, a signal delay of 145ms, so that the final score was 24.76 (named *gablg_p* on the Scoring Server). This is an improvement of roughly 2 points on top of the original delay-based scheme.

Since this approach was fairly simple, we considered that it would not be possible to improve it too much. We decided to moved on from this idea to try other protocols.

4.2 EWMA Link Speed Estimation & Interval Prediction

This model can be found under *victordomene_interval* on the server.

4.2.1 Overview

From our previous experiments, we observed that the delay is the real bottleneck to achieving higher scores. Throughput cannot be entirely sacrificed, otherwise the score will be too low, but it seems to be more useful to concentrate effort on reducing the 95th percentile signal delay. To do so, our approach must be as conservative as possible in taking actions that may cause congestion.

We came up with a simple way to do our predictions, based on Sprout EWMA and "using the past to predict the future". These steps are:

- Estimate the available link speed, using an exponentially weighted moving average;
- Based on a recent history of RTTs of the acknowledged packets, estimate an interval for which the estimated available link speed will be valid;
- Define a conservative window size based on these two predictions.

The difference to Sprout EWMA is that, rather than simply guessing that our prediction will work for the next 100ms, we look at the distribution of RTTs in the recent past (looking, for instance, at the variance and standard deviation of these distributions, as well as upwards or downwards trends), and decide for how long this estimate will be reasonable.

4.2.2 Link Speed Estimation

To estimate the link speed, we simply implemented a version of Sprout's EWMA. We consider a time unit of a tick, which we set to be 30ms. We count the number of ACKs received within this tick, and once we step into a new tick, we calculate an estimate of the link speed there. We use this tick estimate to update our total estimate by using an Exponential Weighted Moving Average, as follows:

$$\text{TotalEstimate} = \alpha \cdot \text{TotalEstimate} + (1 - \alpha) \cdot \text{TickEstimate}$$

where we tried to keep $\alpha = 0.8$ across experiments, preventing an overfitted result.

There is a fundamental issue with this approach as it is. It could be the case that we do not receive any feedback from the sender for a long time (for instance, for the duration of a timeout), which would make our estimate be inadequate. To prevent this, we can either implement a heart-beat which is sent to the server every tick (30ms), or simply reduce the sender's timeout. We chose to do the latter, since it was easier for testing purposes. The timeout was set to 100ms, which seemed to be good enough to keep a reasonable estimate.

4.2.3 Interval Prediction

We keep track of two histories both with size N: one history of the previous RTTS and another of the previous RTT deltas (current RTT minus previous RTT). Then, we analyze these histories to try to find trends and predict how confident we are on our link speed estimation and how confident we are that the link speed estimation won't change. Experimentally, we chose to focus on the history of the deltas and consider two measures:

- **Standard Deviation:** If the standard deviation is low, we have higher confidence that the link speed will continue the same. If the standard deviation is high, we have lower confidence that the link speed will remain the same for a long period of time, so we act more conservatively when updating the window size. We use the standard deviation instead of the variance since it has the same unit as the RTT itself.
- **Mean:** If the mean of the deltas is positive, that indicates a positive trend in RTT, which makes us want to be more conservative when updating the window size. Conversely, if the mean is negative, the delay has been decreasing so we can increase the window size for a longer interval, especially if the standard deviation calculated above is low.

At this point, we tried to come up with a formula that would encapsulate these aspects. We tried combinations of logistic functions, polynomials and other squashing functions. Experimentally, we realized that a polynomial yielded the best results. The final update formula for predicting the interval in which we can send at the estimated link speed. If the mean of the delta history is positive, we do

$$\tau = K * (0.00067 \cdot s^2 - 0.023 \cdot s + 0.88)$$

where s is the standard deviation of the delta history.

If the mean of the delta history is negative, we can be a bit more aggressive so we do

$$\tau = K * (0.00033 \cdot s^2 - 0.0116 \cdot s + 0.98)$$

where these constants come from a polynomial fit to a step function we chose based on intuitive values for the standard deviation. Since we considered only a finite amount of recent history, we assumed the standard deviation would be within a certain range (until 50), and guessed a reasonable step function without overfitting it to experimental results of the Verizon trace specifically. We chose $K = 65$ as a safe prediction.

4.2.4 Window Size Calculation

Given the two parameters above, it is easy to calculate what the window size should be:

$$W = \tau \cdot \text{TotalEstimate}$$

4.3 Mixture Model

This model can be found under *gabgl1_mixture* on the server.

As our final attempt, we decided to consider a model in which the window size was a weighted average between the window sizes produced by the two models, where the weights were 0.3 for the Interval Prediction and 0.7 for Delay-based.

5 Evaluating Delay-based, Interval Prediction & Mixture Model

Initially, we considered the results on the short Verizon trace. We obtained the results below, without tuning parameters specifically for this trace.

Model	Trace	Throughput	Delay	Score
Delay-based	**Verizon**	4	112	35.71
Interval Prediction	**Verizon**	3.34	128	26.09
Mixture	**Verizon**	4	122	33.03

We were intrigued by the fact that Interval Prediction did not seem to give better results than the Delay-based approach (in Section 3 of this report), and hypothesized that this was due to Delay-based overfitting to that trace. We decided to switch gears and run the same tests with the TMobile short traces, using the same set ups as above, without changing any parameter.

The results are summarized below. As expected, it seems like the AIMD constants in the Delay-based scheme are overfitted to the Verizon trace data. This is an indication that our scheme tends to perform better in the general case. More research in this scheme would be required to actually show that we are not overfitting to the specific traces.

Model	Trace	Throughput	Delay	Score
Delay-based	**TMobile**	7.8	194	40.20
Interval Prediction	**TMobile**	11.94	206	57.96
Mixture	**TMobile**	9.4	182	51.64

6 Naming

Our scheme has a very boring name: Interval Prediction. We couldn't come up with anything more interesting... But we used several names in the server, since we were trying different approaches. Usually we used names such as "test", and so on. For our final models, the names are cited in their respective section in this paper: *victordomene_interval*, *gablg1_mixture* and *gablg1_p*.

7 Conclusion

It seems like our Interval Prediction scheme may be an interesting idea to pursue, with perhaps more accurate models for interval prediction. Our initial approach was promising, and gave a reasonable score, while avoiding to overfit to the specific Verizon trace (differently from the Delay-based approach, which had its parameters tuned extensively). Our mixture model is also an interesting competitor in this approach, and doing different models and averaging them, or even choosing between models on demand, may be another direction for future investigation.