Problem Set 2: Congestion Control Contest
6.829 Computer Networks
Sam DeLaughter, Brandon Karpf
25 October 2016

Warm-up Exercise A

1. The best constant window size to maximize the throughput/delay score is a window of 14 packets. The throughput/delay score for a window of size 14 was 12.85 Mbit/s^2.
2. The results are highly repeatable. We found almost no variation in the throughput/delay score when repeating the experiment with the same window size.

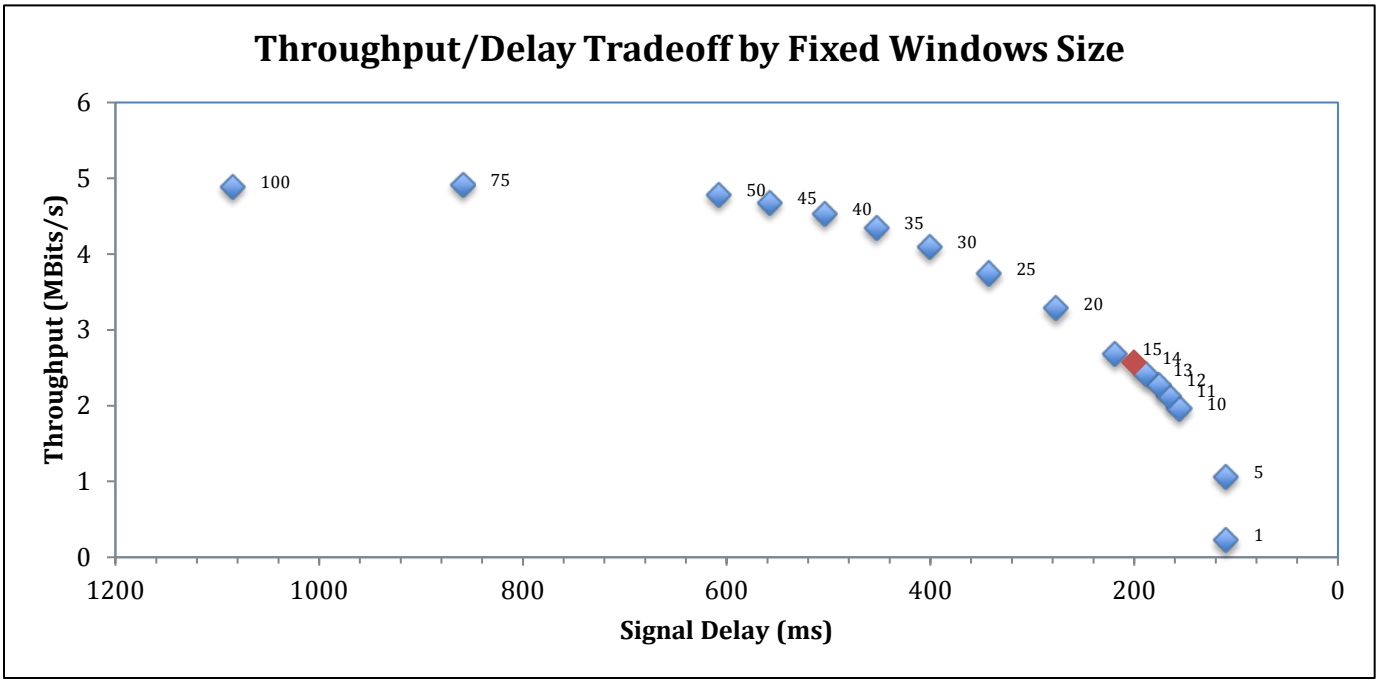

Figure 1. Graph of Throughput/Delay of the fixed window size scheme for congestion control

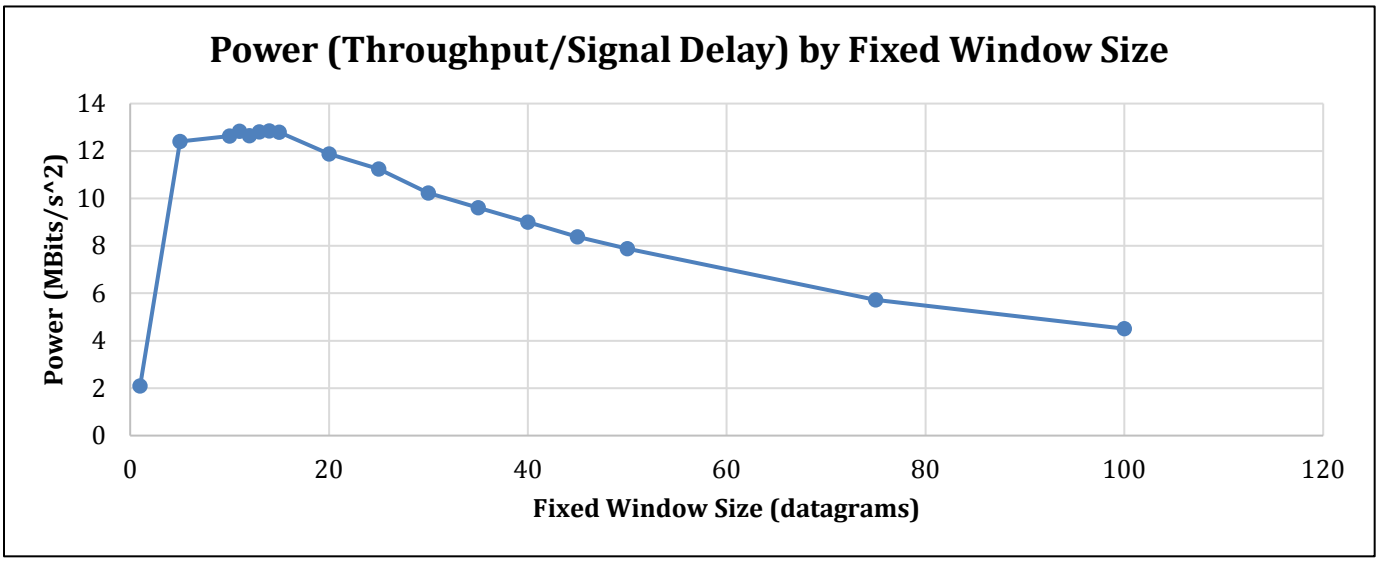

Figure 2. Graph of the power (throughput/delay) of the fixed window size scheme for congestion control

Warm-up Exercise B

1. The AIMD scheme did not work well at all. We had nearly 100% throughput but awful delays. This was somewhat expected, as it resembles the performance of TCP Cubic seen in the Sprout paper, which uses a scheme similar to AIMD.
2. We stuck with the traditional *window_size += 1* every successful RTT within the threshold and *window_size *= 0.75* on our congestion signal. Our congestion signal was caused by a timeout of 50 milliseconds. Therefore, any delay beyond 50ms caused the window size to decrease by half. We chose such a low delay threshold because anything higher than that caused delays of up to 50 seconds. With those values, we obtained 99.5% utilization with 16055 ms delay.

Warm-up Exercise C

1. The best threshold we found for packet delay was 70 milliseconds. We programmed the window to constantly increase by 1 packet until there was delay above 70 milliseconds. At each RTT with delay above 70 milliseconds, we decreased the window by 10. We also tried decreasing the window by 25% for every time the RTT showed a delay greater than 70 milliseconds, which performed slightly better than the linear decrease.
2. With the above scheme, our best result was 47.8% utilization with 139ms delay.

Exercise D

The system in question is inherently nonlinear with no constant steady state value for delay, link speed, or packet throughput. Further, the issue with only communicating RTTs is that there is

no way to determine a characteristic equation for the system without over-fitting. Therefore, there is no way to determine an appropriate transfer function, and subsequently no way to determine an ideal controller based on traditional control theory. The only observable control input is RTT, and the only functional control output is window size. Our transfer function (and characteristic equation) for the system would be modeled such that the $T(s) = \frac{RTT(s)}{cwnd(s)}$, and there really is no way to do that with this system. Further, due to the nonlinear nature of the system and the fact that we cannot linearize around any single constant desired steady state, traditional digital control schemes are a bit tricky to implement.

We decided to use a PID controller with a few modifications that implemented Adaptive Control techniques to correct for (1) the nonlinearity of our system combined and (2) the tricky control signals. Because we have no characteristic equation, we took a roundabout way to implementing the controller that included thresholds and tedious trial and error tuning. In short, it was tedious and we did not have the time to fully optimize our scheme. However, we will discuss a few techniques later in this review for perfecting this control method. The potential for an optimized version of this controller cannot be understated. It is far simpler than Sprout, tunable, adaptable, and our non-optimized version came close to Sprout's power score.

The fundamental control algorithm is as follows:

$$error_i = RTT_i - RTT_{i-1}$$
$$error_{change} = error_i - error_{i-1}$$
$$error_{sum} = \sum_{0}^{i} error_i$$
$$w_i = K_P(error_i) + K_D\left(error_{change}\right) + K_I(error_{sum})$$
$$Window_i = Window_{i-1} + w$$

Where K_P, K_D, and K_I are the proportional, derivative, and integral gains respectively. These gains have negative values, which means the controller will attempt to stabilize the system response. Due to the fact that we have no characteristic equation, these control gains must be determined experimentally. For our test, we chose gain values of K_P = -0.065, K_D = -0.007, and K_I = -0.00003.

We then added a threshold for the actual error sum to +/- 600. This functions as a low-pass filter and ensures the integral term of the control signal (w) stays within reasonable bounds if the data is particularly noisy. It is important to note that we noticed our controller has trouble handling extremely noisy signals, particularly when there is a large increase between two RTTs. This characteristic is a potential limitation and could be resolved with further testing.

The next important technique we used was a threshold for the target RTT. Because the system is nonlinear with no constant steady state, it is impossible to linearize around one specific desired RTT. Therefore, we set a threshold for max and min RTT and linearized the system about that range. The use of these terms is found in the $error_i$ term calculation:

If RTT_i is greater than the max RTT threshold,
Then $error_i = (RTT_i - Max_{RTT})$
If RTT_i is less than the min RTT threshold,
Then $error_i = (RTT_i - Min_{RTT})$

If RTT_i is between the max RTT threshold and the min RTT threshold,
Then $error_i = (RTT_i - RTT_{i-1})$

This algorithm tells the controller, w, to find a steady state within the defined RTT threshold values, which is equivalent to linearizing the system about those points. The RTT threshold we chose was Min_{RTT} = 62ms and Max_{RTT} = 105ms.

The final modification we made to the traditional PID controller was an attempt to quickly implement a highly simplified version of Adaptive Control (AC). AC is good for nonlinear systems, particularly systems with streams. Mechatronic examples would be the flow of gases, liquids, and powders to name a few. Packet flow seems to fall into this category. A notable characteristic of these systems is the diminishing effectiveness of any traditional controller as the control signal moves away from the linearization point. In terms of our system, as RTT goes beyond the threshold (62-105 ms), the effectiveness of our PID controller diminishes rapidly. This does not matter as much in regards to the training data because we know that the system is most stable with an RTT between 62-105 milliseconds. However, another data set may have a more effective RTT range that differs from the range we chose, and therefore our PID control gains will be suboptimal. In a word, any traditional control scheme like PID will be over-fit when used to control a nonlinear system with a dynamic steady state.

AC corrects for that problem. Formal AC dynamically tunes the control gains as the system operates. It uses a scheduling algorithm and the observable system states to test and modify the control gains. Unfortunately, our test-bed does not allow for this application of AC because of the limited scope of the test data. In a real-world scenario, complete AC applied to a PID controller should be highly effective for congestion control. Simplified AC uses a pre-determined equation for each control gain that characterizes the optimal gain value as a function of control signal. As the system changes states, so do the control gains. The control gains are therefore limited to a discrete set of values based on where the system currently is in relation to its operating parameters (usually with max and min threshold values). We implemented a highly simplified version of this simplified AC with the two following algorithms:

AC Algorithm A:
If RTT_i is greater than the max RTT threshold,
Then $error_i = (RTT_i - Max_{RTT} - Nudge_1)$
If RTT_i is less than the min RTT threshold,
Then $error_i = (RTT_i - Min_{RTT} + Nudge_2)$
If RTT_i is between the max RTT threshold and the min RTT threshold,
Then $error_i = (RTT_i - RTT_{i-1})$

Instead of modifying each individual control gain, we decided to modify the control signal. The Nudge term shifts the gain values when the RTT is outside of the threshold. We chose a nudge of 4 for both nudges, although those values can and should be tuned independently. By implementing AC in this way, we alter the control, w, linearly when the RTT is beyond our linearized threshold without having to recalculate each individual gain.

AC Algorithm B:
If $error_i$ is greater than zero,
Then $w_i = \alpha * K_P(error_i) + \beta * K_D(error_{change}) + \gamma * K_I(error_{sum})$
If $error_i$ is less than zero,

$$Then\ w_i = \delta * K_P(error_i) + \varepsilon * K_D\left(error_{change}\right) + \theta * K_I(error_{sum})$$

If $error_i$ is zero,

$$Then\ w_i = K_P(error_i) + K_D\left(error_{change}\right) + K_I(error_{sum})$$

The algorithm above modifies each individual control gain based on the response of the system. The constants (α, β, γ, δ, ε, θ) have been determined experimentally and should not be altered. Although this method is a significant simplification of the AC concept, it still worked to improve our controller performance. It is important to note that with more time, a full implementation of AC will provide a highly effective, dynamic, and still simple PID controller for the problem of congestion control.

The final important change we made was a limit on the size of the control signal w. We limited the size of w to no less than $w = -0.5 * Window_i$. This ensured that we would never decrease the current window size by more than 50% in any single step. We added this term when we discovered that our controller functioned extremely well for consistent data and struggled with outliers. The control scheme we implemented is highly successful at maintaining a constant RTT except when there are extremely drastic shifts down in available link speed. Therefore, we wanted to dampen the controller's reaction to those negative change outliers.

The most effective test using this method on the training data had the following results:

Average capacity: 5.04 Mbits/s
Average throughput: 3.23 Mbits/s (64.0% utilization)
95th percentile per-packet queueing delay: 54 ms
95th percentile signal delay: 94 ms
Power: 34.36 MBits/s^2 (Average Throughput / Signal Delay)

It is important to note that we acknowledge that test was over-fit. Therefore, we backed off on many of the parameters in order to limit the effect of over-fitting. The final variables we used were:

K_P = -0.065
K_I = -0.00003
K_D = -0.007
min_rtt = 62
max_rtt = 105
error_sum_bound = 600
error_threshold_nudge = 0.395
error_threshold_nudge2 = 0.0

It is important to note that due to computational and time constraints, we could not fully optimize these variables. As one can see in Figure 3, there are many places where the link capacity is not fully utilized. Tuning the gains would resolve that issue. Further, with more time, a Monte Carlo simulation could be used to quickly determine the most optimal values for these variables. Finally, a complete implementation of AC would yield the most effective dynamic gain values and control.

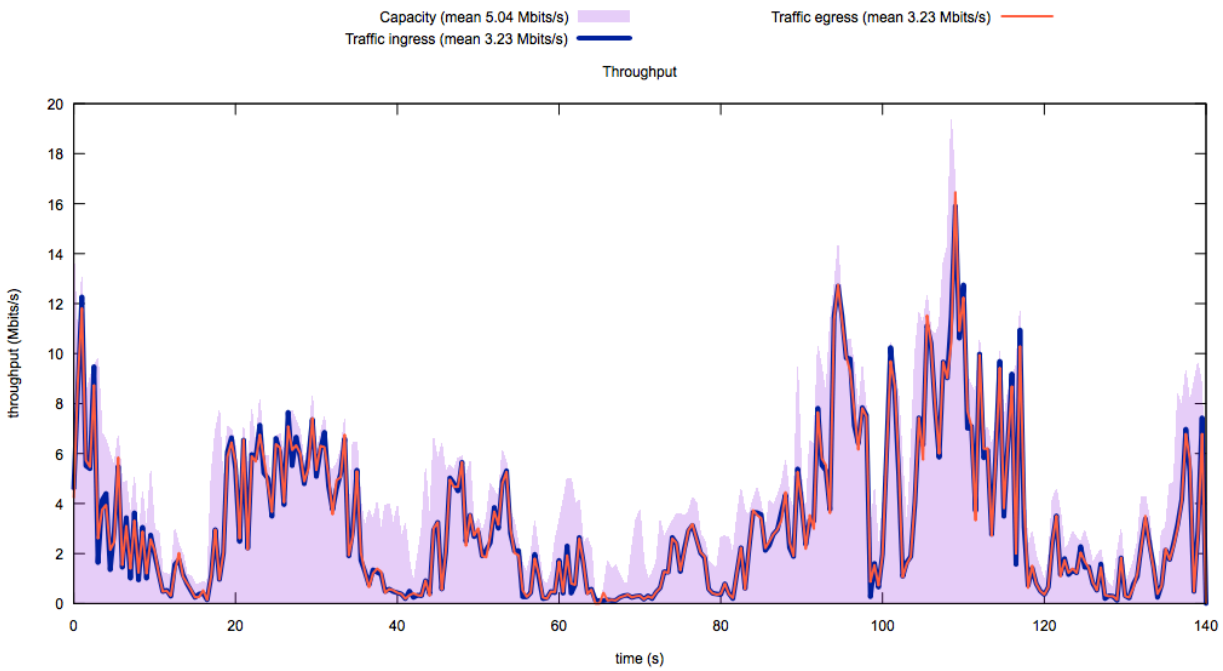

Figure 3. Signal throughput using a PID controller with simplified AC

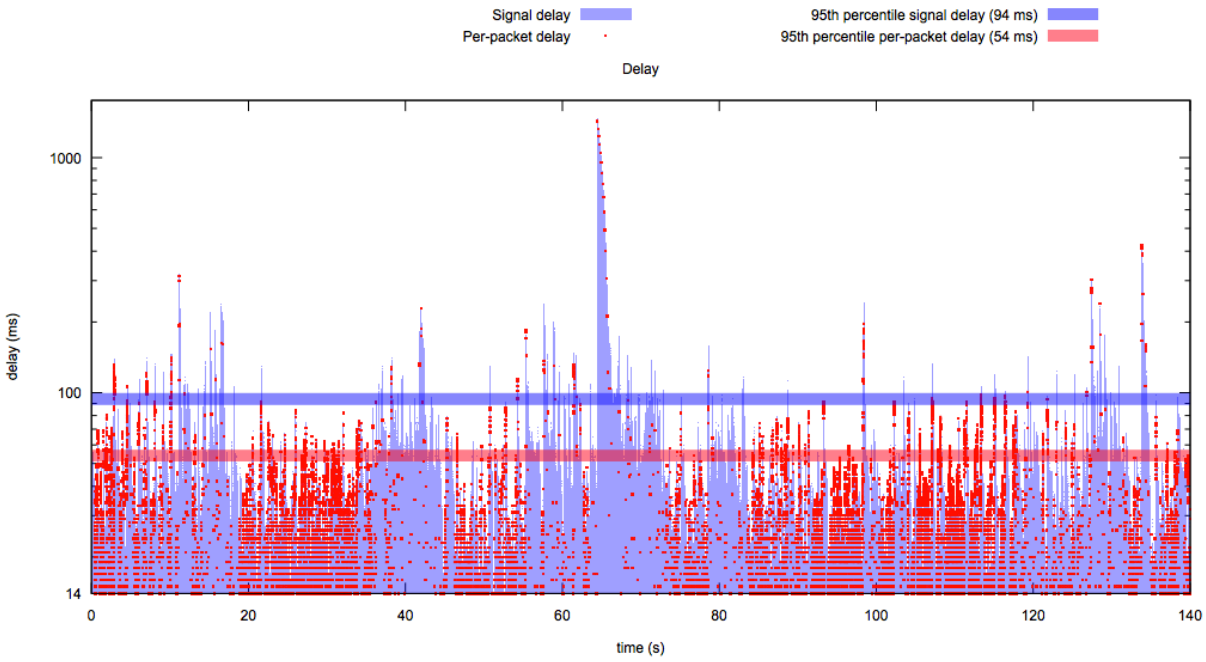

Figure 4. Signal delay using a PID controller with simplified AC