6.829 Problem Set 2

Updated 04/18/13

Alan Ho, Zach Kabelac

Overview

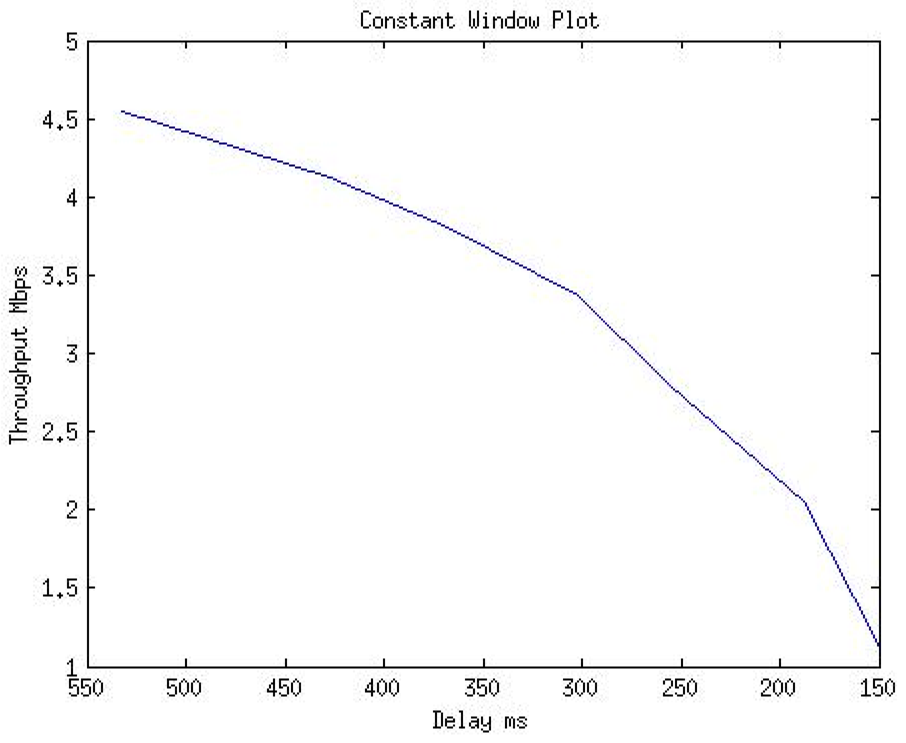
In this report, we discuss the problem of creating a congestion-control protocol that is designed for use in cellular wireless networks, where throughput varies significantly with time, and packet delivery is almost guaranteed (too reliable) so that clients cannot rely on packet drops to detect congestion. This report covers a few solutions and evaluates them based on a metric defined to be the overall throughput in Mbps divided by the 95th percentile of delay in ms. In the first section, we discuss the limitations of a fixed congestion window size approach. Second, we implement and evaluate an AIMD (additive increase, multiplicative decrease) scheme similar to that of TCP congestion control.

Fixed Window Size Evaluation

We started testing the link by using a constant window size, and increasing the window size on each subsequent run. From our tests, we found that both the delay and throughput increased for each window size, and the score peaked around a constant window size of 20. We also found that the measurements were fairly consistent (we did not record multiple runs at the same windows size, but we did try them and observed that they did not change the score by more than 0.05).

In the following plot, we can see a clear tradeoff between throughput and delay: as we increase the window size (from the right side of the plot to the left side), the throughput increases at the cost of delay. The score peaks approximately at the point where a tangent line would touch the curve, around cwnd=20.

The fixed window size scheme achieves a maximum score of about -4.50, much worse than the Sprout algorithm.



AIMD Scheme

As instructed in the assignment, we started by implementing an AIMD (additive increase, multiplicative decrease) scheme similar to that of TCP. We used similar constants. In particular, the additive increase of the window size was by one packet per round trip time, implemented by accumulating fractional window size increases for each ACK. The constant for multiplicative decrease was 2, or halving our windows size whenever we detected congestion.

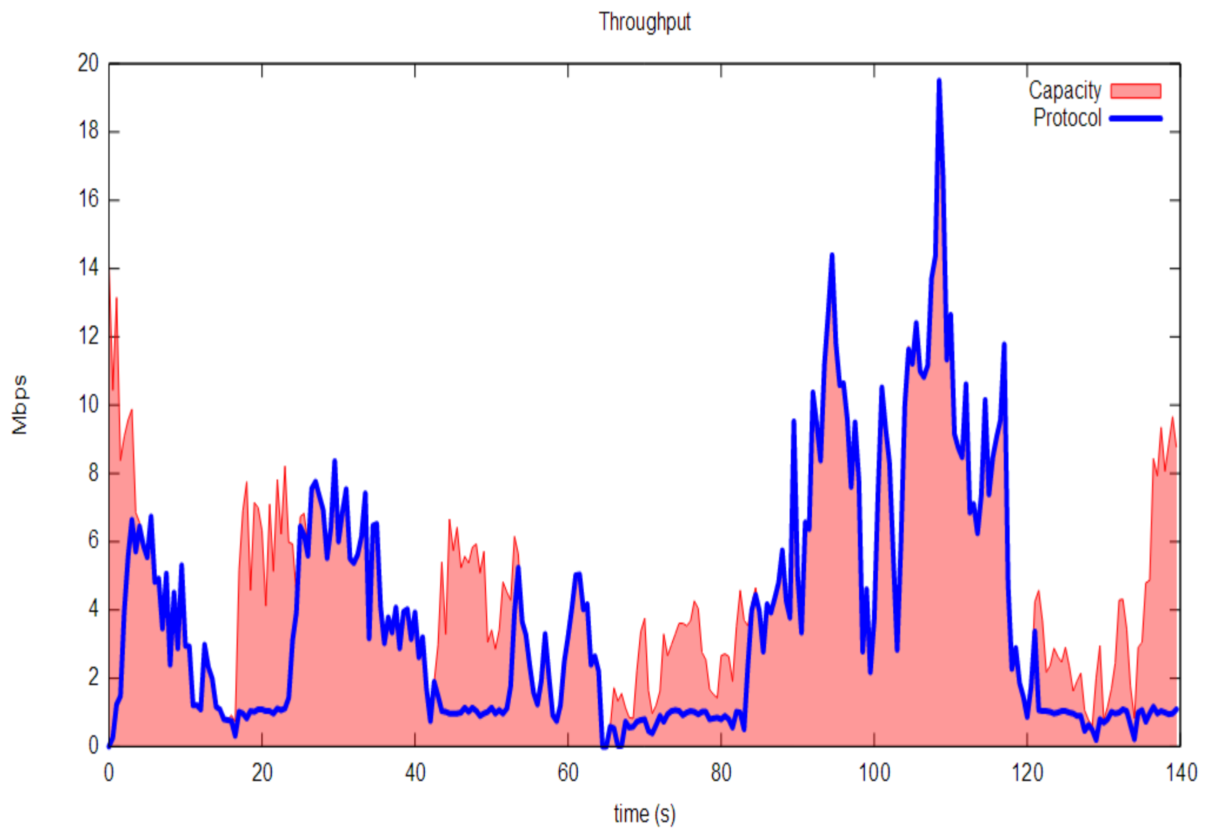
This scheme alone does not work at all if you strictly adhere to the TCP definition of congestion for a client, which is whenever there is a dropped packet. Because of the reliability guarantees of cellular networks (99.99% packet delivery), the system never sees any packet drops and therefore never adjusts the window. Instead, we look at the last RTT and compare to a threshold to indicate congestion. We used a threshold of 100 ms.

We did a few runs with AIMD, tweaking the alpha and beta values (additive increase constant, multiplicative decrease constant, respectively). On initial runs, we followed TCP's parameters and used alpha=1 (increase window size by 1/cwnd per ACK), and beta=2 (halve cwnd on congestion). This resulted in throughput of 4.01 Mbps and delay (95th percentile) of 666 ms. The delays were high because the cellular network had moments where the capacity dropped from 5 Mbps to almost 0 Mbps in a matter of a few seconds. Since the window was so high when the capacity was 5 Mbps, there were too many outstanding packets and the slow connection created long delays for those packets in the queue. By the time we detect congestion via the RTT threshold, the system already had too many outstanding packets on the queue and link and no amount of multiplicative increase could change this. This is the “self-inflicted delay” problem.

In addition, our throughput was lower than it could be because additive increase with these parameters could not ramp up the congestion window fast enough to keep up with sudden rises in capacity. This is the key problem in cellular networks: the capacity changes dramatically within seconds.

Tweaking the parameters did not do much to change our scores. Delays remained in the hundreds of milliseconds while throughput remained high overall, but underutilized whenever the capacity spiked up from a “drought.”

The following is one plot showing the problems discussed. As you can see, the link is over-utilized immediately prior to a drought of capacity (i.e. right before the link’s capacity drops to 0 Mbps) and the link is underutilized whenever the capacity suddenly spikes.



Delay-Based Scheme

A reasonable next step is a simple delay-triggered scheme. We calculate the average RTT over the most recent packet transmissions to sense the current delay of the network. We chose to average over 10 measurements from the past 10 ACKs. This delay is correlated with the current capacity (throughput) of the network. We set a threshold of 70 ms. If the average RTT exceeds the threshold, we assume that the network is congested and perform AIMD on the congestion window size. Qualitatively, this resulted in much better delays while only marginally affecting throughput. We defer a quantitative discussion for the next section, which includes additional improvements that were necessary on top of the delay-triggered scheme.

The Zekalho Algorithm

Our algorithm uses the delay-based scheme to signal congestion and AIMD to adjust the window size. Here we discuss some additional incremental improvements that form the final Zekalho congestion control algorithm.

In the delay based scheme, if a packet returned with too high of a delay, the window size would be decreased. We decided that judgement from 1 packet was too sudden if that packet's delay was an outlier so we averaged the delay's of the most recent 4 packets to make sure the increase in delay was in fact significant. We set our threshold at 70 ms even though our desired average delay was around 150 ms because our window size would be large enough at the time that even though the channel was getting worse, we could not remove those packets from the channel, so we wanted to detect when the channel degraded much earlier in the process.

As for the AIMD scheme, one tweak we added was when the delay of packets was below 47 ms, meaning the channel was good, we increase the window size by 3 packets and when the delay of packets was above 47 ms, we only increased the window size by 1 packet. We did this because we wanted to bootstrap the window size so that we didn't lose out as much on a good channel, but when the channel started to become worse, we didn't want to increase the window size too aggressively anymore. We eventually got 47 ms, and 3 packet aggressive increase and 1 packet conservative increase through experimentation and testing.

A second tweak we made to AIMD was when we noticed that the congestion window would often collapse to 0 whenever there was some congestion, because it would be visible over a few ACKs. So we also enforced a "cooldown" period to ensure that the congestion window did not get halved too many times if there was a temporary intermittent drop in capacity. Once the window size was halved, we didn't let the window sized get halved again until the number of packets currently on the channel was below the halved window size.

An additional change we made to the AIMD and Delay-Based scheme was that we initially added upper and lower window bounds to the window size. We set the lower window size bound to 3 packets in a window. We found that 2 effectively helped to bootstrap the restart after a dramatic drop in window size, but at the same time, having 3 packets on the channel when the channel was severe did not add significantly to latency. We initially did not have an upper bound on the window size, but after multiple runs, our biggest issue was that when the channel crashed, we had too many packets on the channel and that made our latency spike. Given the unpredictable nature of the channel, we decided that it would be extremely difficult to predict one of these crashes so we decided to limit the negative effect one of these crashes would have on our system. That is how we came to the idea of having an upper limit on the window size. We calculated the window size by assumptions of latency on the channel, 70ms and the average bandwidth about 5MBPS a wireless channel should have. This gave us around a 40 packet limit. However, this didn't provide a significant advantage so we removed it in our final submitted version.

Future Work

Future work in developing congestion protocols for cellular networks needs to focus on two things. First, an ideal algorithm needs a good way of sensing the throughput capacity of the network in real-time. We used an average over RTT's. Others have found success in EWMA (exponentially weighted moving average). Secondly, an ideal algorithm needs to predict when the capacity collapses to 0 Mbps, and react accordingly, as fast as possible. This may be accomplished using some form of interpolation of most recent RTT's to detect a sudden drop in capacity.