CS 244 - PA2 - Congestion-Control Contest

Praveen Kumar

April 27, 2017

1 A - Vary window size

1.1 Experiment

We vary the window size in controller.cc and record the throughput and delay statistics. Fig. 1 shows the 2D graph of throughput vs. 95-percentile signal delay for different values of window size.

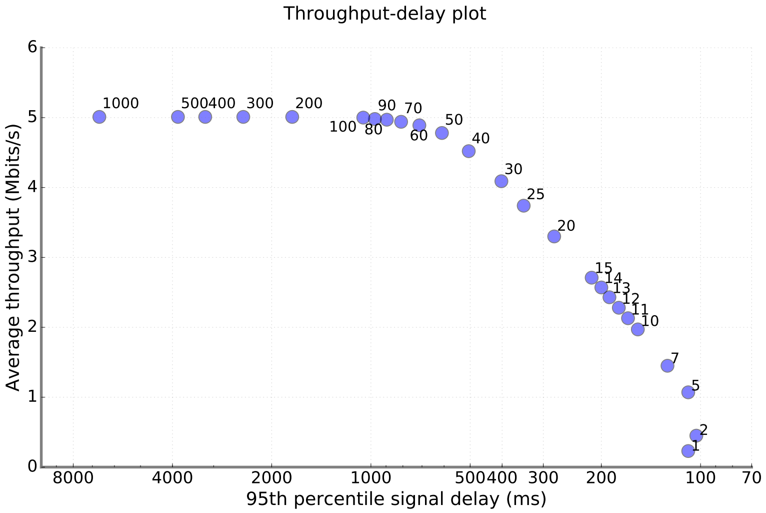

Figure 1: Throughput vs. 95-percentile signal delay for different values of window size

1.2 Best window size

To find the best window size, we plot the value of score defined as:

$$score = \frac{throughput}{95\ percentile\ signal\ delay}$$

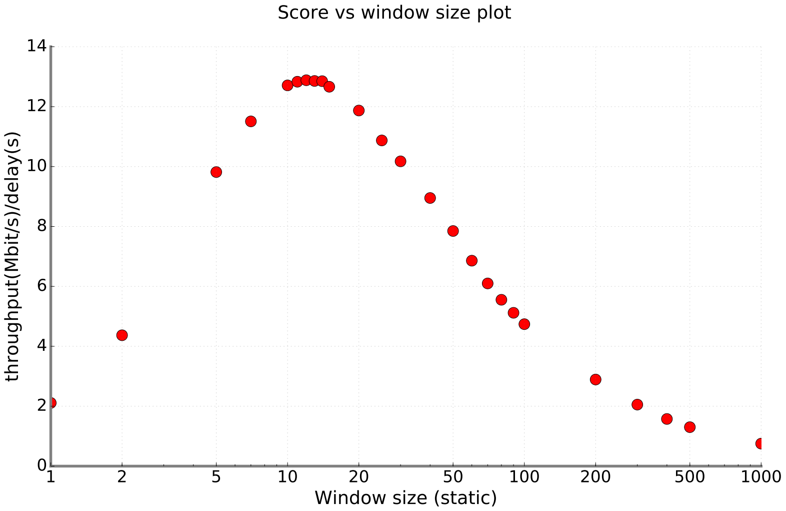

Figure 2: Score vs window size

in Fig. 2.

Fig. 3 shows essentially the same data[1]. From these graphs, we can infer that the optimum static window size is 12.

1.3 Repeatability

We repeated the experiments with window size = 12 five times.
Throughput: $\mu = 2.282\,\text{Mbits/s}, \sigma = 0.004\,\text{Mbits/s}$
95th percentile signal delay: $\mu = 177.2ms, \sigma = 0.748ms$
So, the experiments are highly repeatable.

[1] Problem description said to use log

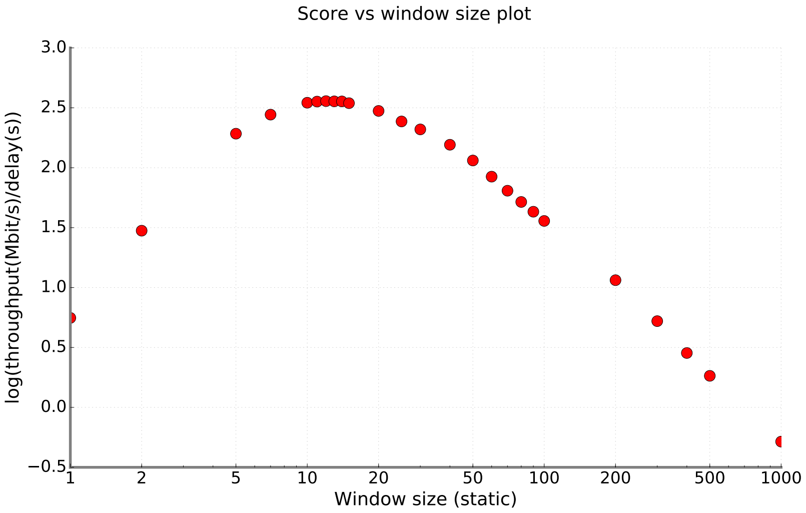

Figure 3: $\log(Score)$ vs window size

2 B - AIMD scheme

2.1 Implementation

We implemented a simple additive increase multiplicative decrease mechanism to dynamically adjust the window size as follows:

```
On ACK receipt:
  /* Additive increase */
  cwnd_ += aimd_inc_param_/cwnd_;

On timeout:
  /* Multiplicative decrease */
  if (cwnd_ > 1) {
    cwnd_ /= aimd_dec_param_;
  }
```

where `aimd_inc_param_` and `aimd_dec_param_` are the AIMD parameters.

2.2 Parameters

We tried two different values of starting window size – default (50) and the optimal static value of (12).
In terms of additive increase, we tried the usual value of 1 per RTT, along with more aggressive values of 5 and 10 as explained later. We kept the multiplicative decrease factor to 2 because it is good enough to backoff rapidly on congestion. We also tried different values of timeout parameter – default (1000) and a more aggressive (100).

2.3 Evaluation

Overall, we observe that AIMD doesn't perform well as explained next. Since, the window size gets changed dynamically, the starting value of window size doesn't affect the behavior much. Fig. 4 shows throughput vs delay for different values of the parameters. Similarly, Fig. 5 shows score for different parameter values.

An additive increase of 1 per RTT results in low throughput as we are not able to fill the pipe quickly when bandwidth is available. So, we increase this parameter to increase throughput, but this also causes more queueing delay and thus decreases score.

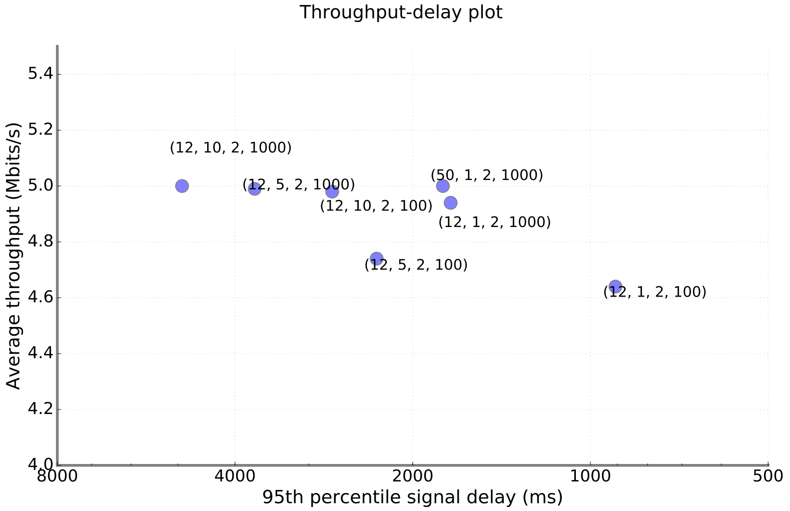

Figure 4: Throughput vs. 95-percentile signal delay for different values of parameters when using AIMD. Labels are: (initial window size, additive increases param, multiplicative decrease param, timeout interval)

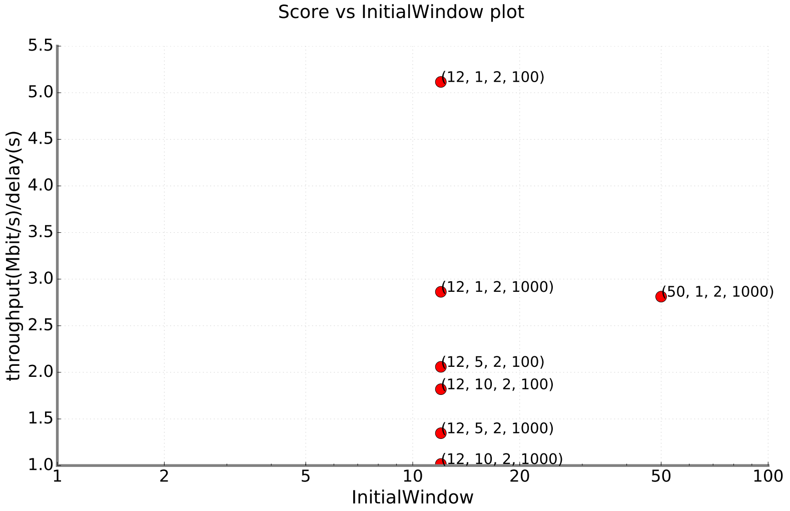

Figure 5: Score vs. Initial window for different values of parameters when using AIMD. Labels are: (initial window size, additive increases param, multiplicative decrease param, timeout interval)

3 C - delay-triggered scheme

We tried two different schemes – simple and AIMD based on RTT.

3.1 Simple

The simple approach is additive increase/decrease based on whether the latest measured RTT is less than or greater than a threshold (rtt_thresh).

```
if (rtt < rtt_thresh_):
  cwnd_ += 1/cwnd_;

if (rtt >= rtt_thresh_):
    cwnd_ -= 1/cwnd_;
```

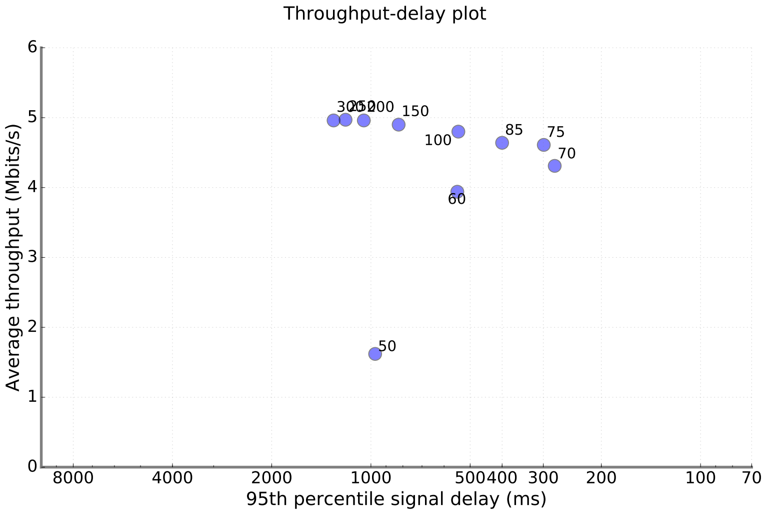

Figure 6: Throughput vs. 95-percentile signal delay for different values of rtt_thresh when using simple delay based approach. Labels show rtt_thresh

From Fig. 6, we can see that increasing *rtt_thresh* increases throughput but also increases latency. As seen in Fig. 7, the maximum score is achieved at $rtt\_thresh = 75ms$. Another issue with this approach is that the value of window can become less than 1, leading to timeouts before packet transmission resumes. This can be see for extremely low values of RTT threshold. We handle this in the next approach.

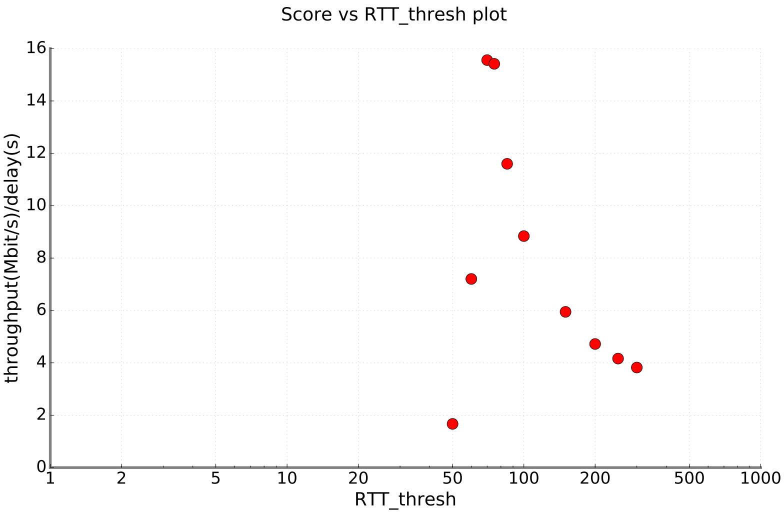

Figure 7: Score vs. rtt_thresh when using simple delay based approach

3.2 RTT + AIMD

In this approach, we modify the window in an AIMD manner based on whether the latest RTT is less than or greater than a threshold (rtt_thresh).

```
if (rtt < rtt_thresh_):
    cwnd_ += 1/cwnd_;

else:
    if (cwnd_ >= 2):
      cwnd_ = cwnd_/2;
```

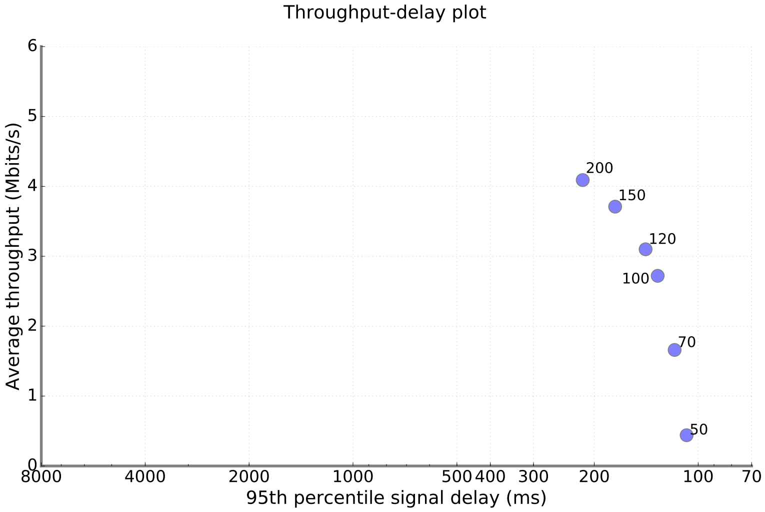

Figure 8: Throughput vs. 95-percentile signal delay for different values of rtt_thresh when using RTT+AIMD based approach. Labels show rtt_thresh

This improves on the previous method of simply increasing or decreasing the congestion window linearly based on whether the RTT was below or above a threshold. The performance becomes more predictable and follows a tradeoff curve. From Fig. 8, we see that increasing *rtt_thresh* increases throughput but also increases latency.

As seen in Fig. 9, the maximum score of 22 is achieved at $rtt\_thresh = 120ms$.

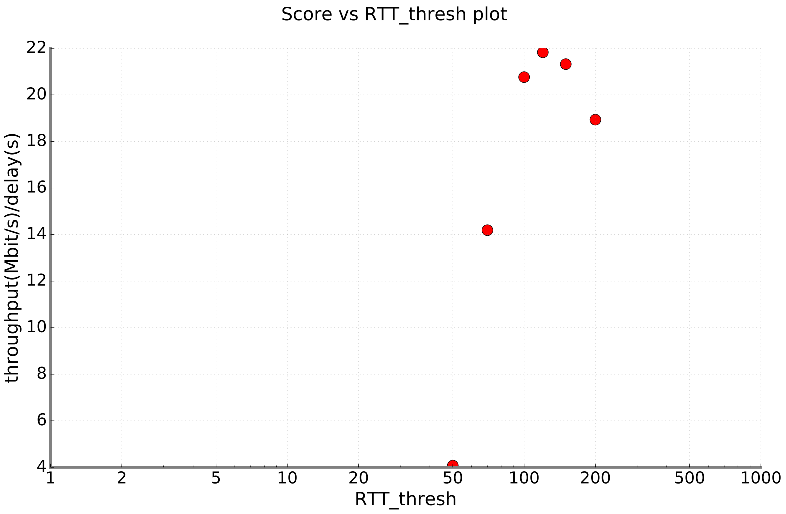

Figure 9: Score vs. rtt_thresh when using RTT+AIMD based approach

4 D - The Contest

The network trace provided is highly unstable. So, our system tries to estimate the steady state whenever the network is stable, and reacts quickly on detecting congestion.

4.1 Latte - Design Approaches Considered

4.1.1 Steady state

For the steady state behaviour, we measure the RTT and bottleneck bandwidth in manner similar to BBR [1]. Let rtt_t be an RTT sample received at time t. We compute min_rtt as the minimum value of rtt_t as observed in time window of W_R. Similarly, whenever an ACK arrives, we update the amount of data that has been delivered. We keep a sliding window of amount of the data delivered. From this, we can estimate the delivery rate. Unlike BBR, we don't use the ACK to bring this information because the current packet format in the framework doesn't have such a field. The bottleneck bandwidth, similarly, is the maximum delivery rate in a window W_B.

Updating congestion window: From the estimated RTT and bandwidth, we can compute the BDP. We set the congestion window = $\lambda \times BDP$, where λ is a configurable parameter.

$$cwnd = \lambda \times max\_bw \times min\_rtt$$

Sending packets: Whenever the congestion window allows packets to be sent, we schedule packets to be transmitted at regular intervals = γ / *bandwidth*, where γ is configurable. In BBR, γ is also modulated periodically by *cycle gain* to detect any increase in bandwidth.
λ and γ are similar to *cwd_gain* and *pacing_gain* in BBR.
This works pretty well if the network is in a steady state. But, the trace shows that the network capacity keeps changing rapidly.

4.1.2 Fast reaction to congestion

Observation The bottleneck bandwidth for cellular networks is volatile, and BBR's approach leads to high latency when this bandwidth decreases. Consider Figure 10 from the the original paper. When the bandwidth has decreased, it causes increased RTT as the sender is still sending at a higher rate. Because RTT estimation happens before this bottleneck is detected, this increases the number of inflight packets and further increases queueing delay. RTT starts decreasing only after the maximum bottleneck window that caused high BDP goes out of bandwidth window, which can be of the order of a few RTTs. So, for this entire duration, the network experiences high delay.

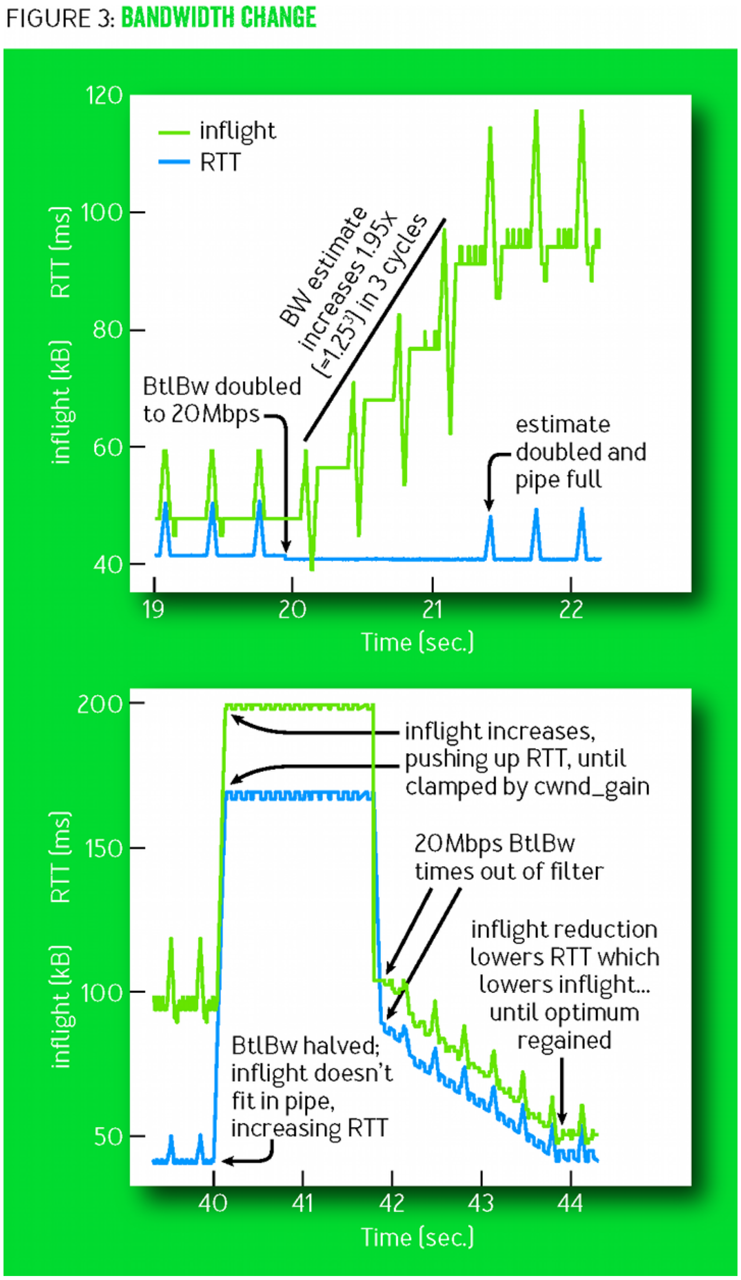

Figure 10: BBR's response to change in bandwidth. (Source: BBR paper [1] Fig 3)

Solution To react to such increase in RTT, we want to quickly cut down the number of packets in flight that may cause queueing delays. So, whenever the sender sees a RTT higher than *min_rtt*, one approach that cuts down the congestion window by a factor exponential to the relative increase in RTT could be:

$$cwnd = \frac{cwnd}{e^{\left(\frac{rtt_t}{min\_rtt}-1\right)}}$$

This approach is robust to noise in RTT measurements because even if the sender receives a bad sample with high RTT, the congestion window changes for a very small interval. It will be updated when the next ACK is received. Also, this exponential penalty doesn't affect the congestion window much when measured RTT is close to *min_rtt*. There are other functions that can be used instead of the exponential function. But, the key takeaway is to *avoid build-up of queueing delay on detecting high RTT when bottleneck bandwidth decreases.*

4.1.3 RTT gradient approach (TIMELY)

Another way to detect build up of congestion could be to look at the RTT gradient instead of just RTT values [2]. This approach could detect that that RTT is increasing and react to it before RTT reaches a high value. We estimated the gradient as:

$$rtt\_grad_t = \frac{rtt_t - last\_rtt_t}{min\_rtt}$$

$$rtt\_grad = (1-\alpha) * rtt\_grad + \alpha * rtt\_grad_t;$$

where α is EWMA smoothing factor. On detecting that that RTT is increasing, or equivalently, RTT gradient is positive, the sender reacts by cutting down the congestion window multiplicatively.

$$cwnd = cwnd \times (1 - \beta \times rtt\_grad)$$

We found that this approach didn't result in any significant improvement for the given trace.

4.1.4 BBR's cycle gain

BBR uses cycle gain to periodically increase and decrease the pacing rates to probe for increase in bandwidth. We experimented with various gain factors and concluded that the net benefit of cycle gain was overshadowed by setting an optimal value of λ.

Timeout We found that setting a good value of timeout is critical to reducing the signal delay. We find empirically that a value of $1.5 \times min\_rtt$ works well. Once a timeout happens, we transmit one packet to act as a probe and reset the timer.

4.2 Latte: Final design

Our final design consists of dynamic bottleneck bandwidth and minimum RTT detection, augmented with aggressive backoff on high RTT detection.

In order to keep the system simple, we removed any approaches (e.g. cycle gain) that we tested and didn't result in significant improvement. Latte considers the latest RTT sample and compares with min_rtt. In our system, since the congestion window is updated on every ACK, based on measured bandwidth and RTT, we can quickly adjust the congestion window on every ACK. Whenever we receive an ACK with higher RTT, we decrease the congestion window by a factor of rtt_t/min_rtt. Thus, the congestion window effectively becomes:

$$\lambda \times bandwidth \times min\_rtt^2/rtt_t$$

when rtt_t is high. This ensures that we do not create congestion in the steady state and rtt_t are close to min_rtt.

4.2.1 Performance

Overfit Based on experimenting with different values, we set $\lambda = 1.7$ and $\gamma = 0.8$. This worked really well for the given trace, and after optimizing the control parameters, we were able to achieve a **power score of 42.56** on the given trace.

`http://cs344g.keithw.org/report/?Latte-1492997567-eezahmoy`

```
Average capacity: 5.04 Mbits/s
Average throughput: 3.83 Mbits/s (75.9% utilization)
95th percentile per-packet queueing delay: 50 ms
95th percentile signal delay: 90 ms
Power Score: 42.56
```

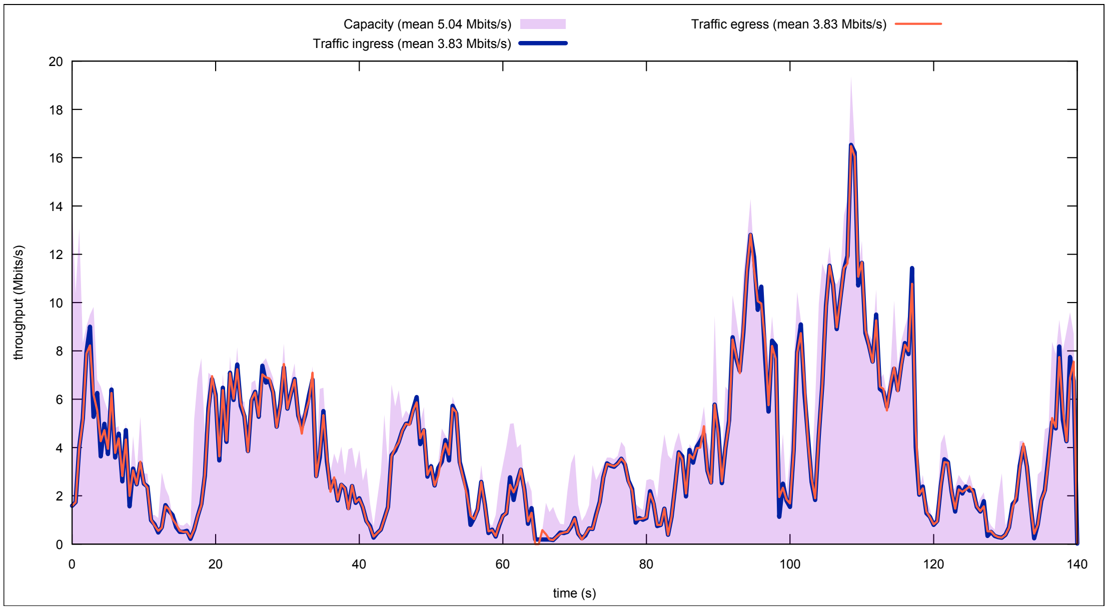

Figure 11: Throughput - Latte on verizon trace

Generalization However, the above parameters seem like overfitting for the particular trace. So, we added a logic to set λ dynamically on the value of *cwnd* that i) keeps λ high at low congestion window to prevent stalling, and ii) decreases λ as congestion window increases to decrease the probability (and amount) of overshooting capacity. As a result, the performance on the given trace decreases. The results are at:
`http://cs344g.keithw.org/report/?Latte-1493317091-ifaikaix`

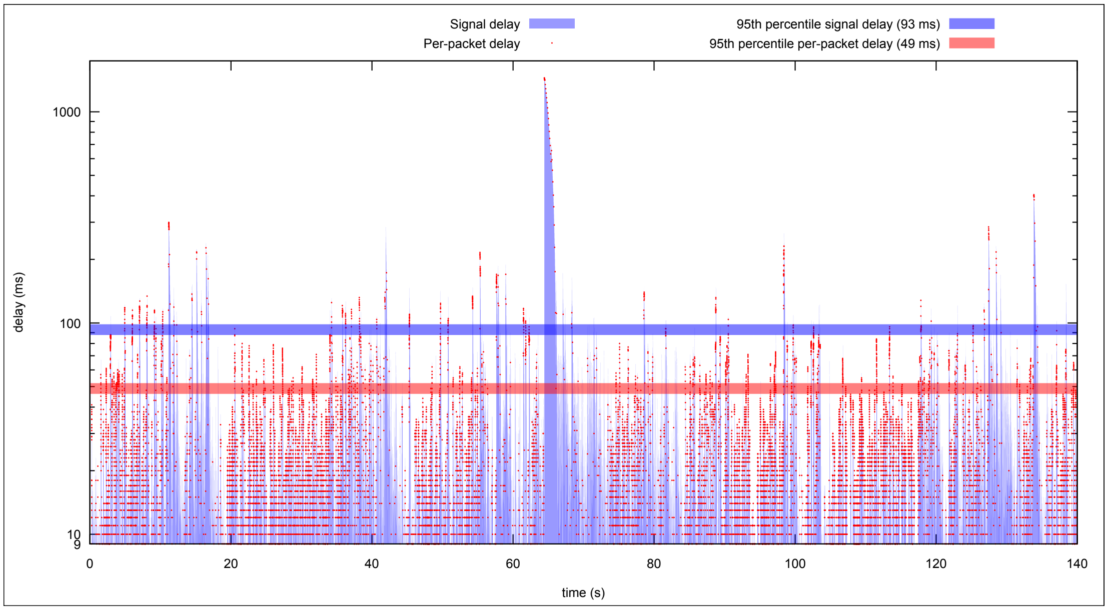

Figure 12: Signal and packet delay - latte on verizon trace

```
Average capacity: 5.04 Mbits/s
Average throughput: 3.83 Mbits/s (76.0% utilization)
95th percentile per-packet queueing delay: 49 ms
95th percentile signal delay: 93 ms
   *** Power score = 41.18 ***
```

The corresponding throughput and delay time series are shown in Figures 11 and 12.

4.3 Conclusion

Our key observation was that the network should perform close to Kleinrock's optimal point in network steady state, but react aggresively to increase in RTT to achieve high power score. We build Latte along this lines by following bottleneck bandwidth and RTT estimation similar to BBR and augment it with multiplicative decrease based on observed RTT.

References

[1] Neal Cardwell, Yuchung Cheng, C Stephen Gunn, Soheil Hassas Yeganeh, and Van Jacobson. BBR: Congestion-Based Congestion Control. *Queue*, 14(5):50, 2016.

[2] Radhika Mittal, Nandita Dukkipati, Emily Blem, Hassan Wassel, Monia Ghobadi, Amin Vahdat, Yaogong Wang, David Wetherall, David Zats, et al. TIMELY: RTT-based Congestion Control for the Datacenter. In *ACM SIGCOMM Computer Communication Review*, volume 45, pages 537–550. ACM, 2015.