EECS 557 Project Report
Adaptive Low-Latency Communication Control for Wireless Communications

Frank Zlomek
James Brisson
Jayanth Srinivasan Prabakaran
Winston Wang

May 15, 2023

1 Introduction

Currently, wireless networks use modern variants of TCP-Reno and TCP-Vegas which use packet loss to determine congestion. This made sense in the early years of the internet when small buffer sizes meant that congestion would quickly lead to packet loss [1]. However, modern throughput and the queues needed to store them have both increased dramatically. As a result, the loss is no longer a good indicator of congestion. When bottleneck buffers are large, loss-based congestion control will fill them until packets are dropped, leading to buffer bloat and increased latency throughout the network. This is unacceptable for next-generation IoT applications such as self-driving cars, virtual/augmented reality, and remote medicine.

To solve this, in 2016 Neal Cardwell et al. developed Google BBR, a congestion control algorithm that sought to minimize the standing size of queues while utilizing all available bandwidth. This is achieved by periodically probing to estimate the bottleneck bandwidth and round trip time (RTprop), and then keeping the total inflight traffic close to the optimal bottleneck bandwidth-delay product (BDP). On unstable wireless flows, we have found that BBR's periodic probing can cause spikes in transmission that prevent the total inflight data from converging to the optimal BDP. We will present a novel variation of TCP-BBR which adaptively probes the network based on the stability of the BDP. This reduced probing achieves a lower worst-case latency on wireless channels at minimal processing overhead and throughput.

1.1 Project results and conclusions

The main output of our project results can be found in Section 4, but overall we have significantly decreased the worst case RTprop at the expense of a slight decrease in the throughput of the overall network. When the bottleneck bandwidth of a wireless link spikes, our algorithm does not spike its transmission rate. This avoids forming queues due to an overestimate of the bottleneck bandwidth of the flow. This is mainly due to the fact, that BBR-APG is less reactive to short-term changes in bandwidth than BBR. Additionally, BBR-APG probes less often when in steady-state operation so it misses a lot of these spikes in throughput and therefore operates smoother in the long run. Overall BBR-APG decreases worst-case (95th percentile) RTprop by up to 60% in some cases with only approximately 1% decrease in throughput.

1.2 TCP BBR

In 1978, Leonard Kleinrock identified that there existed an optimal operating point that achieved maximum throughput while minimizing delay and loss [2]. He found that this point could be achieved by transmitting at the bottleneck bandwidth. This corresponds to keeping the total inflight data close to the BDP. Figure 1 illustrates this relationship between inflight and RTprop. Loss-based congestion control operates at the right edge of the bandwidth-limited region, where it fills the bottleneck queues until they overflow. This delivers full throughput at the cost of high latency and frequent packet loss. Conversely, performance on the left edge delivers data at the maximum capacity of the bottleneck, without creating a queue that eventually overflows. This results in the same high throughput at a significantly lower RTprop. Thinking of the bottleneck as a pipe, this corresponds to keeping the pipe full, but never fuller [3].

BBR is a congestion control algorithm using the bottleneck bandwidth (BtlBw) and propagation time (RTprop) as the feedback to determine the window size and throughput of a link. To operate at the BDP, BBR does not use packet loss to determine the capacity of the system. Rather, it successively estimates the bottleneck bandwidth and round trip time to optimize the flow of packets through the network [1]. Since BtlBw and RTprop are constantly changing, they must be continuously estimated in order to keep the system optimized. To estimate the bandwidth, TCP-BBR periodically spikes its transmission rate to see if it will increase the delivery rate. This periodicity is hard coded into the system, regardless of system dynamics.

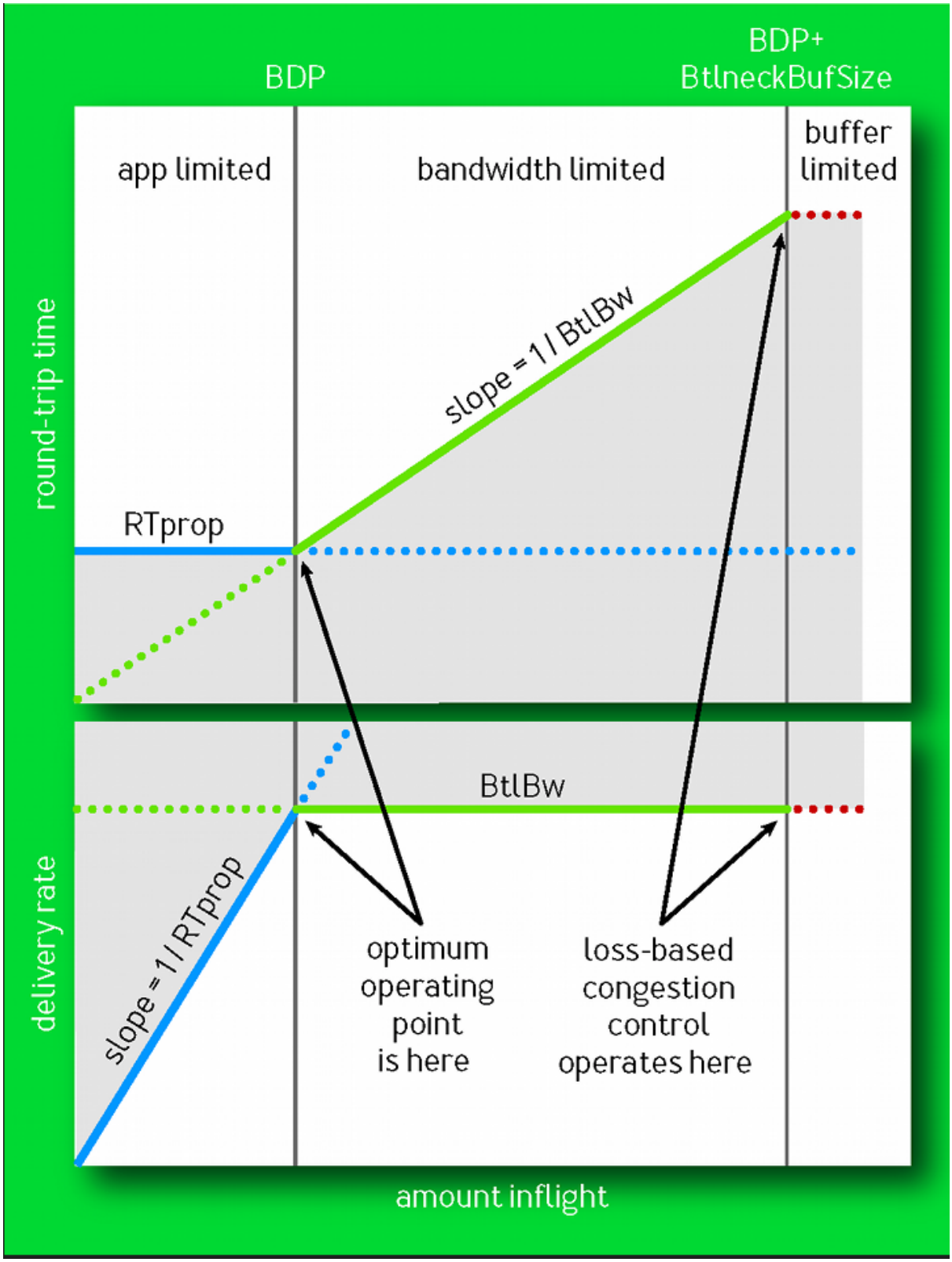

Figure 1: Delivery Rate and Round-Trip Time vs. Inflight

1.2.1 Characterizing the Bottleneck

Similar to other congestion control algorithms, the RTprop is calculated by taking the smallest time it takes for a sent packet to receive an ACK. The RTprop at any time t can be found by

$$RTprop_t = RTprop_t + \eta_t \tag{1}$$

where

$$\eta \geq 0 \tag{2}$$

is the noise inherent in the system. This leads to an unbiased estimator at time T as

$$\widehat{RTpropprop} = RTprop + \min \eta_t = \min(RTprop_t) \forall t \in [T - W_r, T] \tag{3}$$

To estimate bottleneck bandwidth, the algorithm tracks the total amount of data delivered in each ACK, and divides it by the time it took to receive the ACK. This delivery rate is less than or equal to the bottleneck rate and therefore becomes an unbiased estimator of the BtlBw [1].

$$\widehat{BtlBw} = \max(deliverRate_t) \forall t \in [T - W_B, T] \tag{4}$$

When operating at this point, RTprop and BtlBw are independent of each other. As a result, it is impossible to estimate both quantities at the same time. BBR cycles between different modes to probe for increased bandwidth or decreased RTprop.

1.2.2 BBR Steady-State Behavior

To control the amount of inflight data, BBR controls values called pacing_gain. A pacing_gain > 1 increases inflight and decreases packet inter-arrival time, whereas a pacing_gain 1 has the opposite effect. When in the steady state of the algorithm, ProbeBW, BBR periodically probes to find increases in bottleneck bandwidth. This is implemented by cycling through the following pacing_gains: [5/4, 3/4, 1, 1, 1, 1, 1, 1]. When an increase in bandwidth is encountered, Figure 2 demonstrates that in each period the inflight will increase by 5/4. This allows BBR to converge to higher bottleneck rates exponentially. However, in a steady state, BBR drains queues by sending at 99% of the BDP. As a result, queues drain linearly, as demonstrated on the right side of Figure 2. This weakness to decrease in bottleneck bandwidth is a key weakness of BBR.

2 Novel Aspects of our Algorithm

In ProbeBW mode (steady state), the periodic increase in inflight will causes queues if the bottleneck bandwidth is less than or equal to BBR's estimate. These queues increase RTprop and are suboptimal for constant connections. Furthermore, there is no mention of design intent when determining the pacing gain cycle length of 8. Our plan is to learn the frequency of probing through a feedback mechanism. Our updated BBR algorithm with adaptive pacing gain (BBR-APG) addresses this while keeping BBR's fundamental operations intact. The current BBR probe implementation cycles through the following pacing: [5/4, 3/4, 1, 1, 1, 1, 1, 1]. Our novel change is to vary the number of ones in this array. If our currently estimated BDP is significantly different from the previous BDP, we decrease the cycle length (number of ones) by 4, down to a minimum of 4. If it is not significantly different, we increase the cycle length by 4, up to a maximum of 16. For stable bottlenecks, the BBR-APG will probe half as often, whereas for stable networks it will probe twice as often. This behavior is particularly useful when working with wireless bottlenecks, where bandwidth will occasionally spike very high. If BBR sees these spikes when probing and increases throughput, it will create a large queue when the bandwidth settles back down. This will lead to linear recovery time as mentioned previously. BBR-APG avoids this by decreasing its cycle length. This increases the likelihood that our algorithm hits these peaks and increases inflight significantly. However, for stable increases in capacity, our algorithm converges faster than BBR.

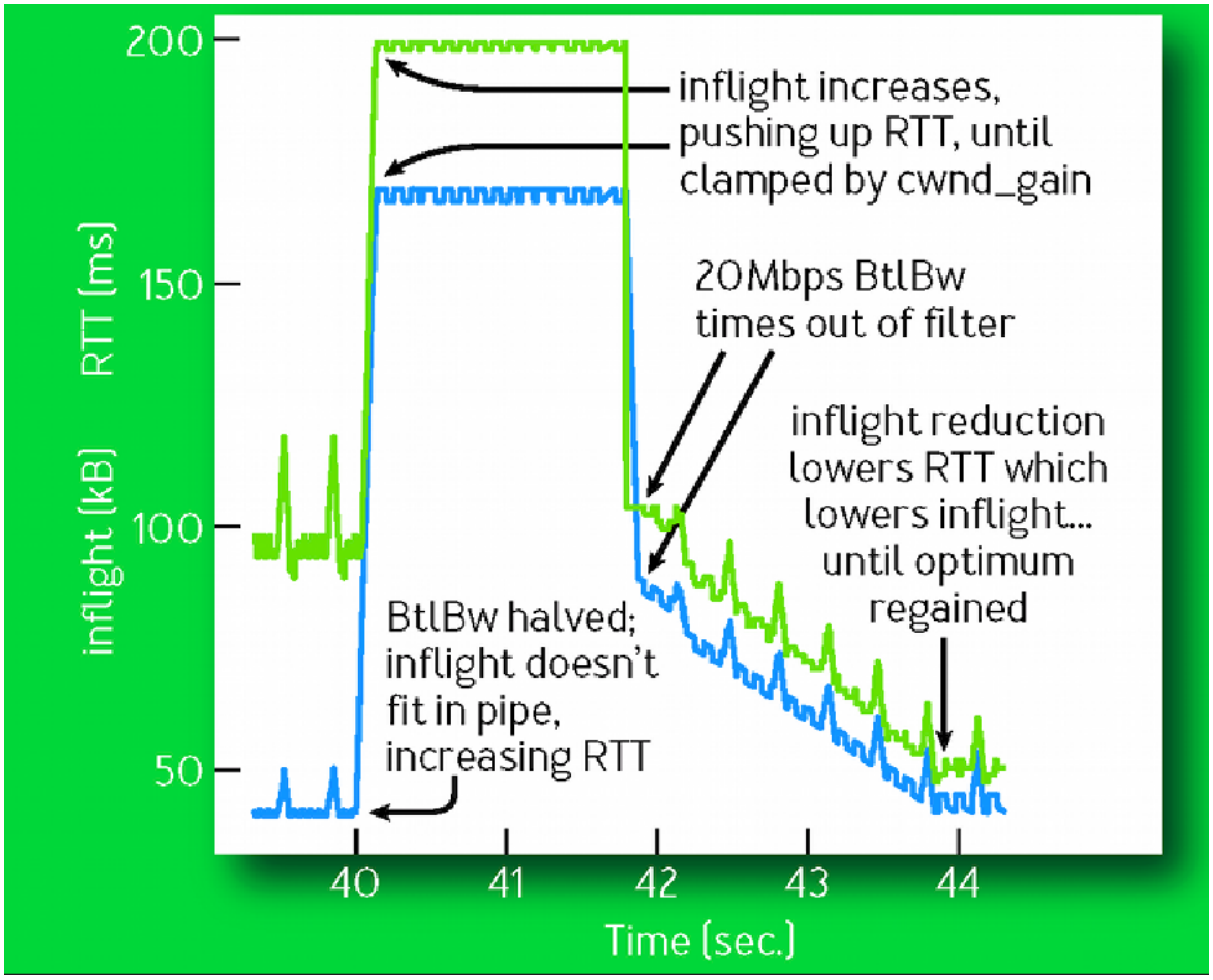

Figure 2: Bandwidth Change

Algorithm 1 Dynamic adjustment of cycle length

```
/* Cycle length initialized to 8 to match TCP BBR                    */
cycle_len ← 8
tol ← 1/4  stride ← 4  MIN_CYCLE_LEN ← 4  MAX_CYCLE_LEN ← 16
if |current_BDP − prev_BDP| > prev_BDP × tol then
  | cycle_len ← (cycle_len − stride < MIN_CYCLE_LEN) ? MIN_CYCLE_LEN : cycle_len − stride
end
else
  | cycle_len ← (cycle_len + stride > MAX_CYCLE_LEN) ? MAX_CYCLE_LEN : cycle_len + stride
end
```

What we are doing in the algorithm is when a substantial increase or decrease in the bandwidth-delay product is experienced we decrease the cycle length window size to a size of 4 from its previous size of 8 and then let it decrease to the new optimal state. If the system is running optimally, we update the window size to 16, allowing for a greater increase in pulse lengths and fewer packets being sent through the system. See 3 To reiterate what we are doing, the novel aspect of our project will not be in optimizing the basic functionality of the BBR algorithm but in taking a piece of the algorithm that is used to estimate its optimal state and use that to stay in the optimal state for longer periods of time. This will allow fewer packets to be going through the system, decreasing the network's overall latency. We believe this will be very beneficial for wireless networks since they can send at intermittent bandwidths. We can use this updated algorithm to check bandwidth changes infrequently but still stay on top of changes by sending these probe packers through the system, just at an increased time period when compared too BBR. In the case that a user is streaming or in a video conference we can use BBR-

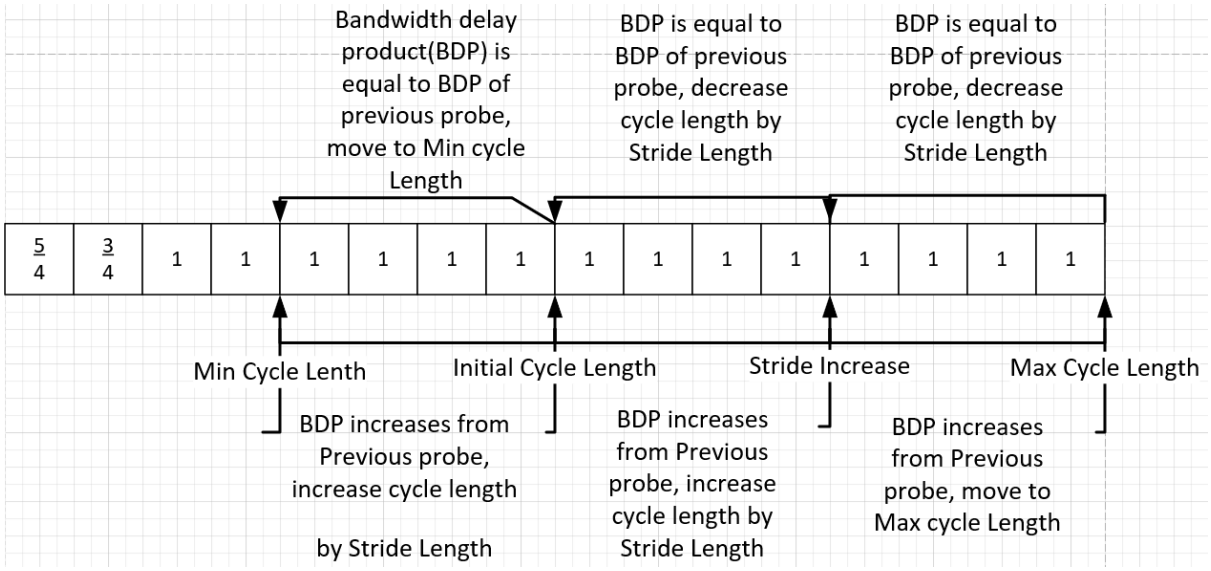

Figure 3: Algorithm Visualization

APG to shadow the bandwidth and follow through the time it takes him to watch that video. Once the streaming event is over a probe pulse will be sent out in a certain amount of time to check that event and then re-calibrate the intersection point. All while keeping the number of own packets going through the network at a minimum.

3 Methodology

3.1 Implementation

In order to prove the novelty of our algorithm we will be using a virtual machine with Ubuntu and version 4.15.0-1141-gcp of the Linux kernel, which has a BRR module running in it [4]. We then use MahiMahi to emulate the network and to allow us to control the bottleneck bandwidths, RTprops, and the loss rates [5]. MahiMahi is a network emulation tool that is used to effectively evaluate the ways you can make the web perform faster over different network conditions [5]. With MahiMahi we then use a custom Python TCP server and client, that perfectly encapsulates the sender and receiver and creates a network channel with traffic for our algorithm to work in a real-world network condition. with MahiMahi we are able to specify our own loss, RTprop, and capacity in the network and change it at will as we see fit [5]. Once MahiMahi was up and running, we used it to test out the design of the BBR algorithm in order to prove the baseline operation of our system before adding additional changes to it. In addition, for our experiment we are using a link with an infinite-sized buffer, this allows us to simplify our test environment as well as helps us to keep everything the same so we can prove that our novel idea is working to increase throughput through the link [4]. Once the BBR-APG code for our simulation was working correctly and we were getting images that matched the BBR paper, we then went to work on editing the simulation to include our intended change. In the original simulation, the pulse timing was hard-coded into the BBR algorithm. Since our change was to learn when to send a pulse based on the available bandwidth we needed to write additional code in Python in order to accomplish this. Once this Python code was written we then had to implement it into the simulation and then do some work to debug and test it. With this code in place, we were able to start our experimentation.

3.2 Experimentation

For the experiment, we use the MahiMahi emulator to allow us to simulate a wireless mobile network. We specify our "trace" or capacity of the network and then it simulates a mobile network by using data on how packets have been sent through a real-world network. Our MahiMahi emulator is using

packet information from a Verizon LTE and T-Mobile LTE network and uses the RTprop and Bandwidth information from that network to send the packets between the sender and receiver in our network. We used the following 4 network testing examples in order to test BBR-APG: 1. Constant bandwidth through the entire time frame. 2. Step increase in the bandwidth of the network 3. A graph of a mobile network. We then ran the BBR and BBR-APG algorithm through these test examples in MahiMahi to see how they reacted in each case. We then compared the data in a table for each experiment between how BBR performed and how BBR-APG performed in the same network conditions. The Testing Section below will show the output we got from our test and the table for each of the 4 experiments.

4 Testing and Results

4.1 Experiment One

In Experiment One, we conducted a comparison of the performance of TCP BBR and BBR-APG on a constant bandwidth trace of 12 Mbps. As shown in Figure 4, BBR-APG exhibits significantly less probing compared to BBR shown in Figure 5. This reduction in probing can be attributed to the adaptive cycle length of our algorithm, which recognizes the constant channel and adjusts the rate of probing accordingly.

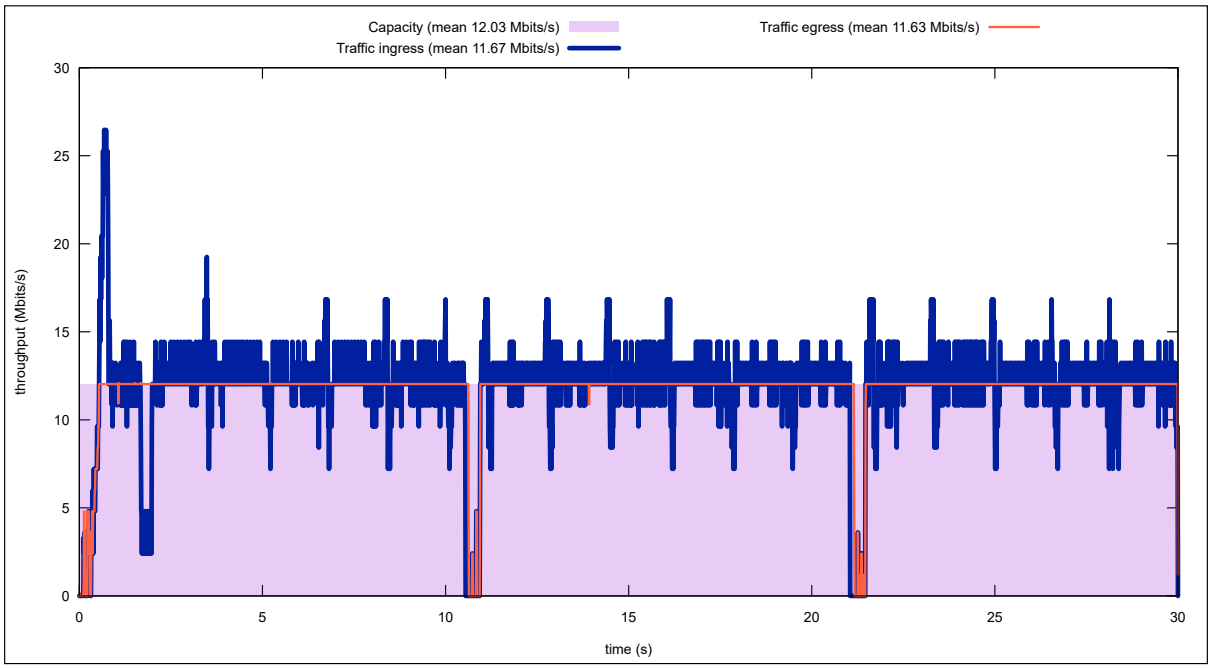

Figure 4: BBR-APG in constant bandwidth

4.2 Experiment Two

Experiment Two involved a comparison of the performance of TCP BBR-APG and BBR on a trace with a step increase in bandwidth. The bandwidth was increased from 12 Mbps to 24 Mbps after 3 seconds, providing the step increase. As shown in Figure 6, BBR-APG demonstrated a faster adaptation to the step increase compared to BBR shown in Figure 7. This can be attributed to the adaptive cycle length of our algorithm, which is able to sense the change in bandwidth and adjust the cycle length accordingly to suit the increased bandwidth. These findings underscore the potential benefits of using BBR-APG in scenarios where network capacity varies over time.

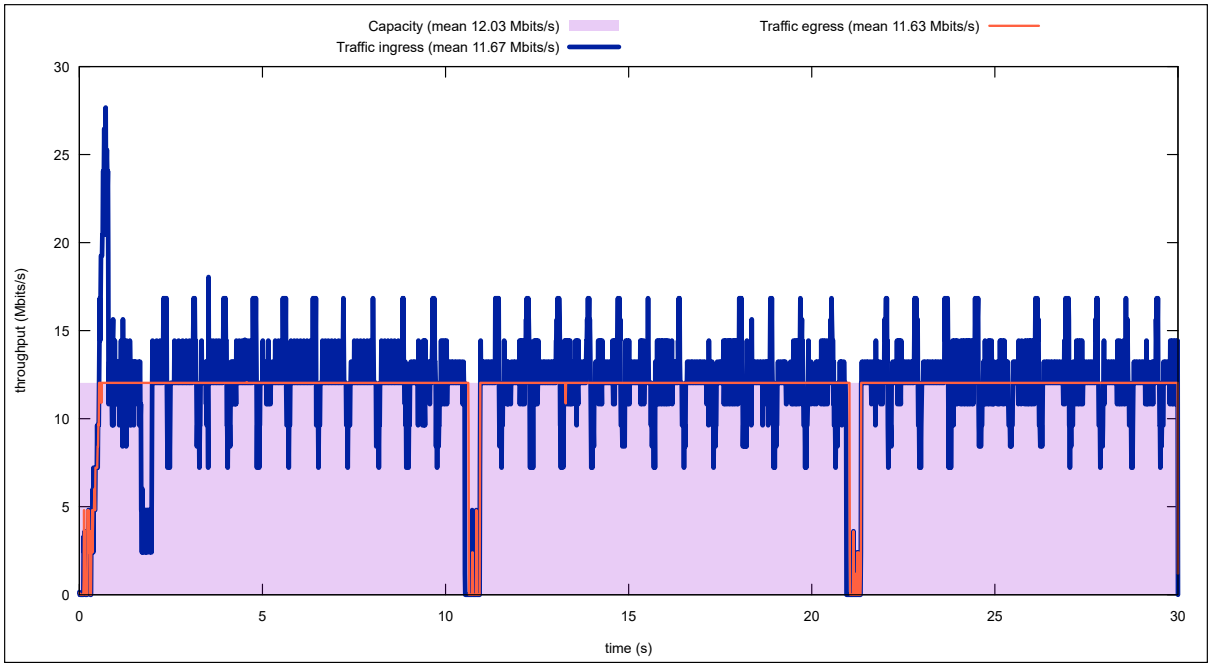

Figure 5: BBR in constant bandwidth

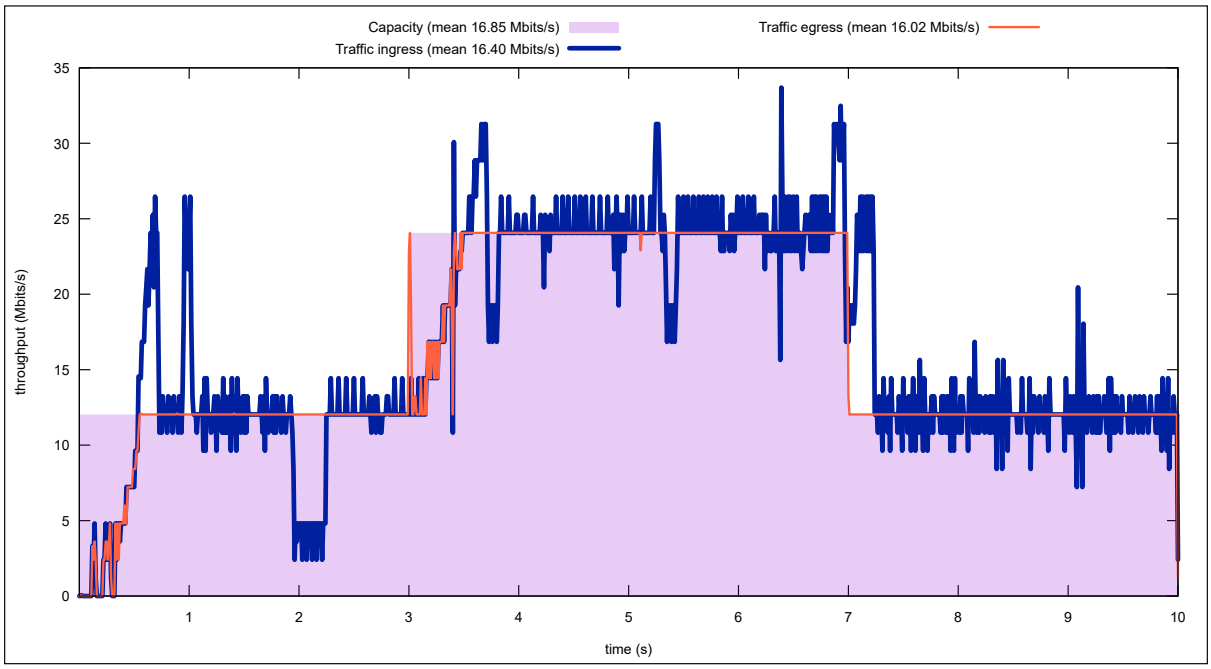

Figure 6: BBR-APG in Step BW increase

4.3 Experiment Four

It should be noted that the experiments conducted so far may not fully represent real-world scenarios. To address this, we conducted a comparison of BBR-APG and BBR using a Verizon LTE and T-Mobile LTE trace to simulate a more realistic network scenario. As shown in Figure 8 and Figure 9, the Verizon LTE and T-Mobile trace provided a more complex and dynamic network environment than the constant and step change traces used in previous experiments. These results provide a more accurate evaluation of the performance of BBR-APG compared to BBR in real-world scenarios.

Tables 1 and 2 show that BBR-APG has a significantly lower packet queue delay compared to BBR, with a 37% decrease for Verizon LTE and a 66% decrease for T-Mobile LTE. This reduction in delay can

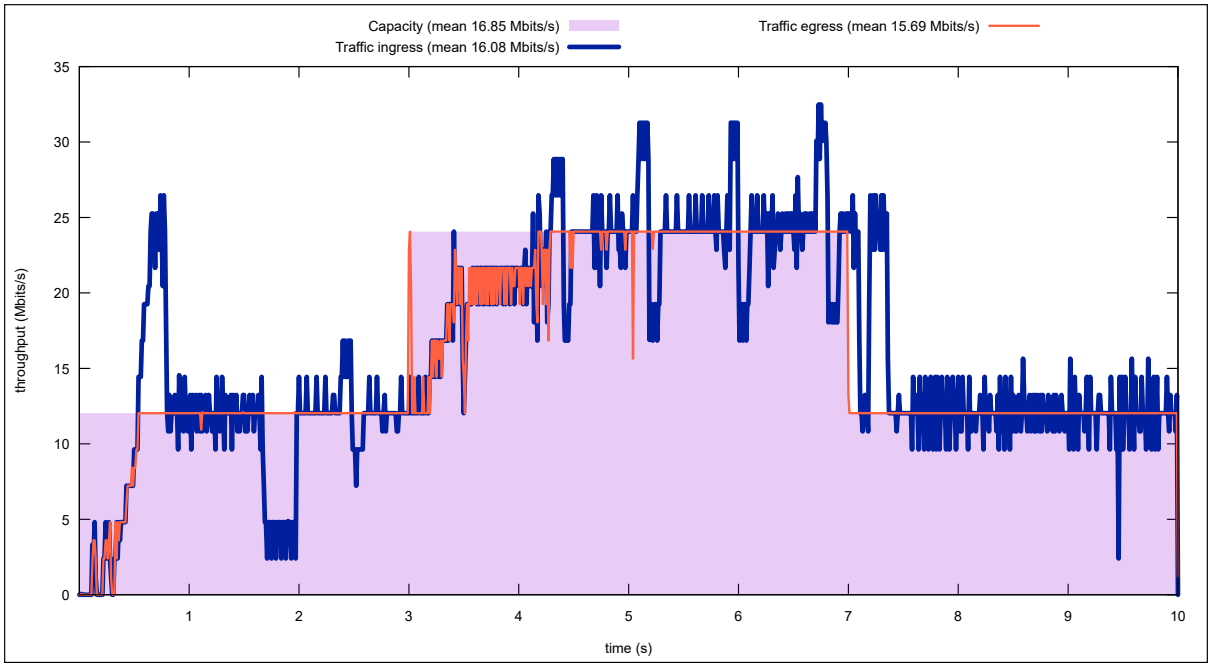

Figure 7: BBR in Step BW increase

be attributed to how BBR-APG adapts to the varying network conditions. BBR-APG has an adaptive probing interval that depends on the delay bandwidth product, which allows it to better avoid random peaks in the bandwidth and thereby results in less draining of the queues. In contrast, BBR rapidly compensates for the peaks in the network and thereby incurs queuing delays to drain the queues when the bandwidth goes down. These results suggest that the adaptive nature of BBR-APG improves the overall performance of the network.

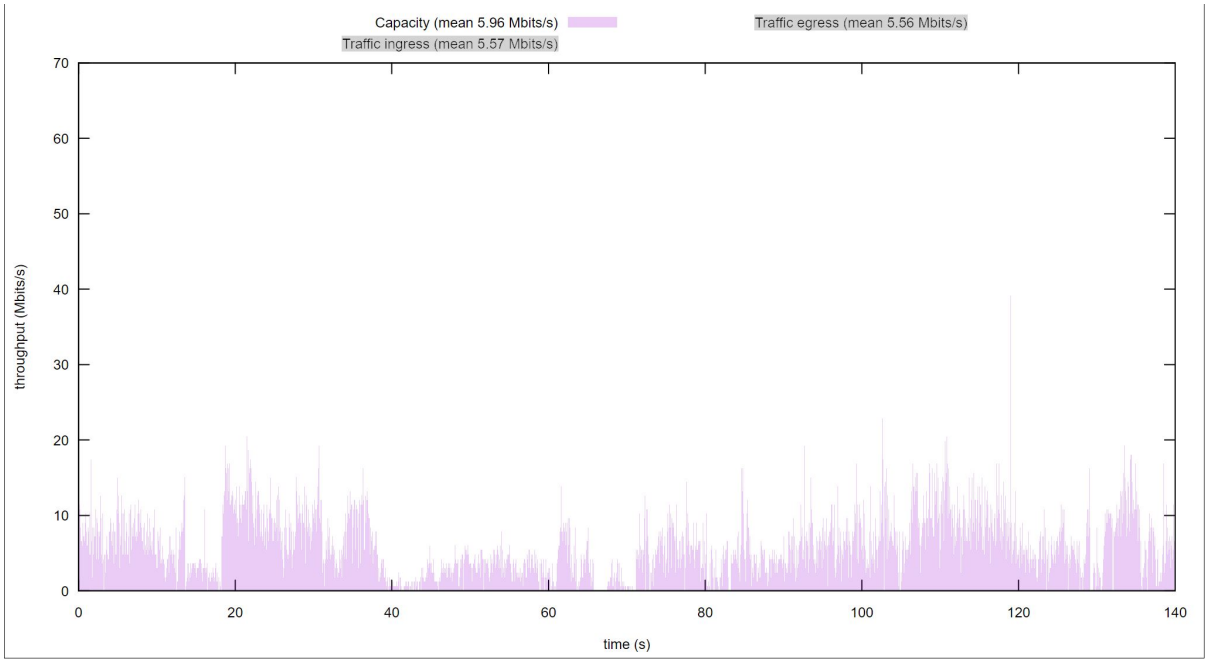

Figure 8: Verizon-LTE trace

5 Challenges

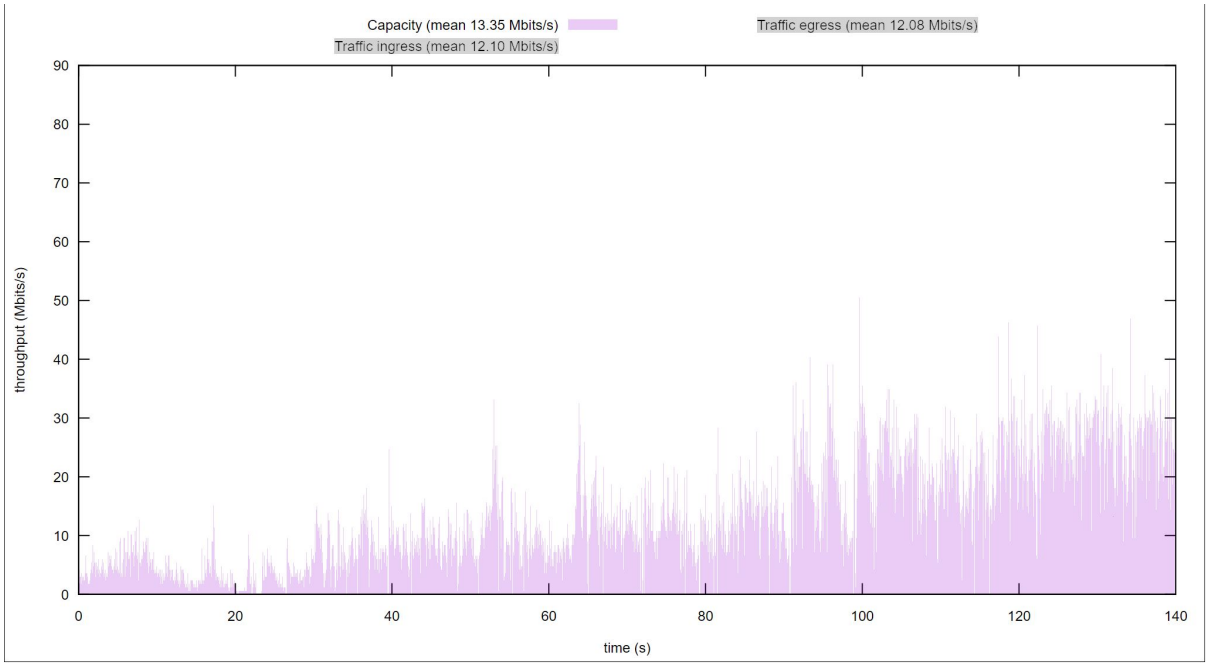

Figure 9: T-Mobile-LTE trace

Table 1: BBR-APG vs BBR for Verizon LTE

Congestion control	Loss rate	Goodput Mbps	RTprop ms	BW Mbps	Specified BW Mbps	Packet q delay	Signal delay
BBR-APG	0.001	5.56	100	5.96	100	491.0	1159.0
BBR	0.001	5.73	100	5.96	100	784.0	1923.0

Table 2: BBR-APG vs BBR for T-Mobile LTE

Congestion control	Loss rate	Goodput Mbps	RTprop ms	BW Mbps	Specified BW Mbps	Packet q delay	Signal delay
BBR-APG	0.001	12.08	100	13.35	100	273.0	523.0
BBR	0.001	12.18	100	13.35	100	807.0	1099.0

References

[1] C. Stephen Gunn Soheil Hassas Yeganeh Van Jacobson Neal Cardwell, Yuchung Cheng. Bbr: Congestion-based congestion control. *acmqueue*, 12 2016.

[2] Leonard Kleinrock. On flow control in computer networks," in conference record, international conference on communications. pages 27.2.1–27.2.5, Toronto, Ontario, 1978.

[3] Leonard Kleinrock. Internet congestion control using the power metric: Keep the pipe just full, but no fuller. *UCLA*, 6 2018.

[4] Luke Hsiao and Jervis Muindi. Cs244 2017: Rebbr: Reproducing bbr performance in lossy networks. *Stanford CS244*, 6 2017.

[5] Ravi Netravali, Anirudh Sivaraman, Keith Winstein, Somak Das, Ameesh Goyal, and Hari Balakrishnan. Mahimahi: A lightweight toolkit for reproducible web measurement. *MIT Computer Science and Artificial Intelligence Laboratory*, 8 2014.

[6] Taran Lynn and Dipak Ghosal. Tcp davis: A low latency first congestion control algorithm. *2022 IEEE International Conference on Networking, Architecture and Storage, NAS 2022 - Proceedings*, 2022.

[7] Junseon Kim Jihoon Lee Santae Ha Kyunghan Lee Shinik Park, Jinsung Lee. Exll: An extremely low-latency congestion control for mobile cellular networks. 13, 2018.