Sharing but not Caring - Performance of TCP BBR and TCP CUBIC at the Network Bottleneck

Major Qualifying Project

Advisor: Craig Wills

Written By: Saahil Claypool

A Major Qualifying Project
WORCESTER POLYTECHNIC INSTITUTE

Submitted to the Faculty of the Worcester Polytechnic Institute in partial fulfillment of the requirements for the Degree of Bachelor of Science in Computer Science.

Project Number: MQP-CEW-1904

AUGUST 2018 - MARCH 2019

Abstract

Loss-based congestion control protocols such as TCP CUBIC can unnecessarily fill router buffers adding delays which degrade application performance. Newcomer TCP BBR uses estimates of the bottleneck bandwidth and round trip time (RTT) to try to operate at the theoretical optimum – just enough packets to fully utilize the network without excess queuing. We present detailed experimental results that show in practice, BBR can either over- or under-estimate the bottleneck bandwidth and RTT, causing high packet loss for shallow buffer routers and massive throughput variations when competing with TCP CUBIC flows. We suggest methods for improving BBR's estimation mechanisms to provide more stability and fairness.

Acknowledgments

I would like to thank Professor Mark Claypool, Dr. Jae Chung, and Dr. Feng Li for their insight and guidance during this project. Their years of experience and expertise in this area were invaluable in helping me make sense of this field.

Page

1 Introduction

Video streaming has become the largest contributor to Internet traffic. In 2017, video streaming accounted for 75 percent of all IP traffic, and this number is expected to grow to 82 percent by 2022 [1]. TCP is the most commonly used Internet protocol, accounting for 70% of all call and video streaming bytes sent [24]. Despite its prevalence, TCP is not optimized for video - video streaming has real-time constraints such that the next packet is often more important than a lost or in-flight packet. In contrast, TCP blocks the sender window progression until each packet is delivered in order, possibly incurring delay in the client video player [25]. This limitation is especially noticable when TCP is configured to use a loss-based congestion control protocol such as TCP CUBIC [14]. Loss based congestion control protocols equate loss with congestion and expand the congestion window (CWND) until a loss event occurs. This CWND adjustment provides high network utilization but can result in bufferbloat: large standing queues at bottleneck routers. These large queues increase the round trip times (RTTs) for TCP connections, which can greatly decrease video performance [13, 25]. This problem has been exacerbated in recent years by cheaper memory and thus larger router queues, further increasing the queueing delays. Networks that would have RTTs of milliseconds when uncongested can have RTTs of seconds. This increased RTT can reduce video streaming quality of service (QoS) because it can increase the time to detect and retransmit lost packets or to switch video to lower encoding rates [13].

Recent work seeks to combat bufferbloat by replacing TCP CUBIC with a congestion control protocol that is *not* loss based [8, 6, 4]. These approaches aim to minimize queueing delay while still maximizing throughput. BBR, a new congestion control protocol developed by Google [8], aims to combat bufferbloat by estimating the minimum round trip propagation time (RTprop) and the maximum bandwidth at the bottleneck (BtlBw) for a given connection to compute the bandwidth delay product (BDP). BBR then paces its sending rate at the estimated bottleneck bandwidth, and caps the inflight packets to a small multiple of the BDP.

In theory, a single BDP of packets inflight is the optimal operating point for a TCP connection as it minimizes delay while maximizing a connection's throughput [19], and in practice, BBR attempts to operate close to this optimal operating point. BBR has been successfully deployed in Google's Youtube edge servers, increasing quality-of-service [8]. Spotify AB, an audio streaming platform, has also tested BBR and found that BBR helps reduce streaming stutters [9].

Despite BBR's promising performance, BBR may not operate well on pathways with small bottleneck router queues, and when BBR is in direct competition with loss-based flows such as TCP CUBIC [23, 22, 8]. On shallow buffers, BBR creates a huge amount of loss due to its CWND cap and ignorance of loss. This behavior is especially problematic when it shares a bottleneck with loss-based congestion control protocols which treat this excess loss as a congestion signal. Even when router queues are not small, BBR can have significant throughput variation when sharing a bottleneck with CUBIC. Both of these problems together pose an issue to video streaming applications which are sensitive to high loss and inconsistent throughput.

Our paper seeks to better understand the performance of BBR in shallow buffers and when competing with CUBIC. We set up a hardware testbed for controlled experiments and create custom tools to conduct a wide variety of network performance tests. These tests vary: link capacities, network latencies, router queue lengths, TCP congestion control configurations, and number of TCP flows competing at a bottleneck.

Analysis of the results verifies prior work regarding BBR's performance in shallow buffers and in competition with CUBIC. It is crucial for experimental results to be independently reproduced by other researchers within the scientific community in order to generalize the knowledge beyond the experience of the individual scientist. Further, our wide variety of tests allow us to more precisely define BBR's behavior in different network conditions. Specifically, we find that BBR's high throughput variation and high loss in shallow buffers are due to a static CWND and erroneous RTprop estimations. We suggest heuristics to improve BBRs reliability in these conditions.

The rest of this paper is organized as follows: Section 2 discusses prior work in TCP congestion control and specific areas of improvement for TCP BBR; Section 3 describes our experimental setup to evaluate BBR in a hardware testbed; Section 4 provides validation of both our testbed and prior work on BBR; Section 5 presents our performance evaluations;

and finally, Section 6 summarizes our conclusions and presents possible future work.

2 Related Work

This section defines the optimal operating point (Section 2.1), describes recent TCP congestion control protocols (Section 2.2), and describes BBR specifically (Section 2.3).

2.1 The Optimal Operating Point

The theoretical optimal operating point for TCP congestion control is when a single BDP of packets is in flight in the network, and the arrival rate of packets at the bottleneck router is equal to the service rate (the limiting factor of the bandwidth) at that router [19, 18]. If these conditions are met, then the network will be fully utilized, but no packets will have extra queueing delay.

This relationship can be seen in Figure 1. If any more packets are added to the network, or they arrive in bursts at the bottleneck router, the packets incur queueing delay, thus moving that that operating point to the right of the optimal operating point. Conversely, if any fewer packets are added to the network, the bottleneck router will be underutilized, thus moving the operating point to the left of the optimal operating point. Thus, the optimal operating point achieves the maximize throughput *and* the minimum latency.

2.2 Recent Congestion Control Protocols

While 'Kleinrock's operating point' is provably the best theoretical operating point for TCP congestion control applications, it has also been proven impossible for a distributed algorithm to converge to this point [21]. Thus every congestion control protocol attempts to operate at some 'good' operating point. The different approaches include: loss-based protocols (Section 2.2.1) which maximize throughput, utility-based protocols (Section 2.2.2) which constantly take actions to improve their operating point based on some metric function, and measurement-based (Section 2.2.3) protocols which explicitly attempt to set their parameters to match the theoretical optimal operating point.

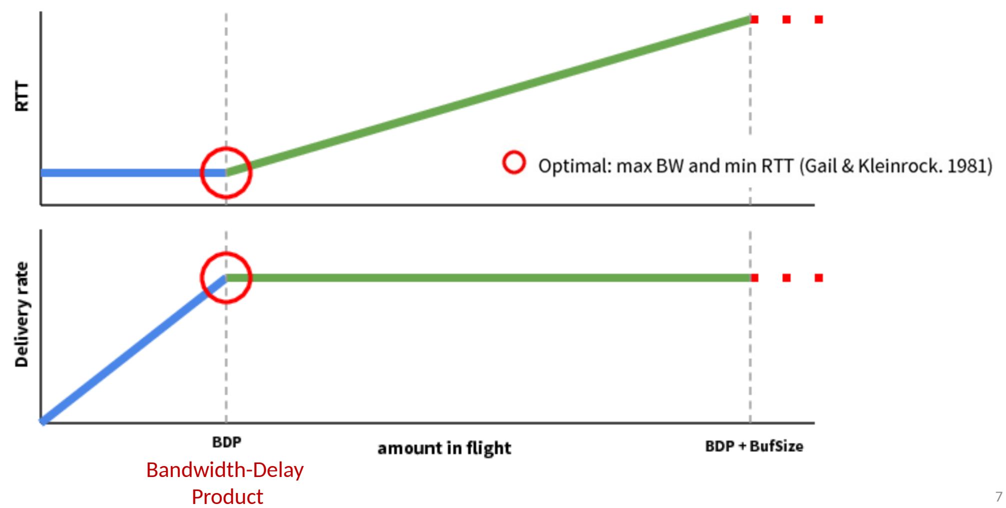


Figure 1: Optimal Operating Point [7]

2.2.1 Loss-based congestion control

Loss based congestion control protocols treat lost packets as congestion signals. These protocols typically work by expanding their CWND to utilize network bandwidth until loss occurs (indicating the network is saturated). Then, the protocols will decrease their CWND, and repeat the cycle. This cycle will result in the protocols constantly operating to right right of Kleinrock's theoretical optimal operating point, resulting in high throughput, but high latency.

TCP CUBIC [14], the defacto standard loss-based congestion control protocol, aims to maximize network utilization by controlling the congestion window (CWND) with a cubic function. The cubic function's convex nature allows the CWND to quickly grow to utilize available capacity. In the case of a congestion event (such as packet loss), the cubic function adjusts such that the CWND increases in a concave manner as it approaches the previous maximum. This concave plateau helps CUBIC to not overshoot the maximum network capacity, avoiding packet loss. In the absence of loss, CUBIC returns to the convex profile to rapidly fill available network capacity. Together, these two profiles seek high utilization and low loss.

2.2.2 Utility-based congestion control

Utility based congestion control protocols evaluate their performance or utility over time, and adjust their performance to maximize this value. This updating scheme requires two parts: a utility function to serve as an objective measure of performance, and the ability to take action to increase (or decrease) utility. Unlike loss-based congestion control (such as CUBIC), utility-based protocols do not have explicit responses for different congestion events, but rather just a general set of actions to take to maximize their utilities. Using just a utility function makes protocol design simple - the designer does not need to explicitly handle each and every condition, but rather just to let the algorithm adjust to maximize utility.

Performance-oriented Congestion Control (PCC) [11], one such utility-based congestion control protocol, works under the assumption that networks are too complicated to deterministically predict the effect of a given action. So, it is infeasible for a protocol to have a correct predefined action to a congestion event, as is the case in CUBIC, in order to achieve good performance. Thus, PCC treats the underlying network as a "black box" and empirically observes which actions provide the best utility by continuously conducting *experiments*. In an experiment, PCC reduces or increases sending rate and observes the utility created by this action. It uses this observed information to inform its next action through a gradient-ascent algorithm to adjust its sending rate towards the optimal. This decoupling of congestion events and actions allows PCC to perform well even in networks with high random loss such as WiFi. In comparison, traditional congestion control protocols such as CUBIC treat random loss as a congestion event and thus reduce their CWND (and throughput) accordingly.

Copa [4], another utility-based congestion protocol, uses a simple set of rules to update the sending rate and CWND towards an optimal utility value, creating no queuing at the bottleneck router if all flows are using a similar CWND updating method. Copa measures the queueing delay as the difference between the observed and minimum RTTs, and increases its sending rate until small queues are created at the bottleneck. Copa implements a "competitive mode" if it detects a competing buffer-filling flow. This addresses the problem seen in earlier TCP congestion control algorithms with utility based on delay, such as TCP Vegas [6], where full buffers cause such protocols to back off to reduce congestion, leading to unfairness [16].

2.2.3 Measurement-based congestion control

Loss-based congestion control protocols are reactive: they wait until loss is observed before taking an action to fix the loss. Utility-based congestion control protocols treat the network as a black box: they do not make an assumption about they *should* observe, but rather just take actions to maximize some utility function (which hopefully leads to good network performance). Measurement-based protocols attempt to estimate what the network *should* behave like. Measure-based protocols track the change in the network conditions and take predefined actions based on these measures. Examples of this include TCP Vegas, which we discuss in this section and TCP BBR, which we will discuss in Section 2.3, both of which attempt to measure network variables and explicitly adjust performance to maximize the network performance.

TCP Vegas [6], measures the observed throughput and RTT to estimate how much bloat (excess queued packets) it has created in the network. Vegas sets the *expected* throughput to $WindowSize/RTprop$ and compares this value to the actual observed throughput. If the *actual* throughput is lower than the *expected* throughput by some threshold α, Vegas assumes it has too small of a congestion window, and increases the CWND linearly for one RTT. Similarly, if the *actual* throughput is greater than the *expected* throughput by some threshold β, Vegas assumes the CWND is too large and decreases the CWND accordingly.

In theory, this scheme should let Vegas operate near the Kleinrock optimal, but in practice Vegas tends to be too conservative and lose out to loss-based congestion control protocols such as CUBIC because it decreases its CWND to minimize buffer bloat while loss-based protocols will increase CWNDs until loss occurs. This conservative CWND adjustment allows the loss-based protocols to dominate the bottleneck and results in Vegas receiving little of the network capacity.

2.3 BBR: bottleneck bandwidth and round-trip propagation time

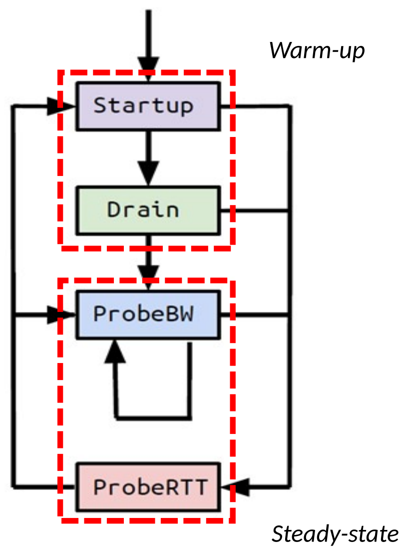


Figure 2: BBR's States [7]

BBR [8] is a congestion control protocol designed to replace loss based algorithms. Regardless of path, TCP views the path as a single link characterized by its bottleneck bandwidth (BtlBw) and the minimum RTT (RTprop) is the physical time it takes a packet to propagate through the network in the absence of queueing delay.

BBR [8] aims to operate near Klienrock's optimal operating point [18] by estimating the BtlBw and RTprop parameters and setting the sending rate and inflight packets accordingly. BBR estimates these parameters by switching through a series of states (Figure 2). During $ProbeBW$, which occurs every 8 RTTs, BBR increases its CWND (congestion window) and sending rate multiplier such that the sending rate is greater than the current BtlBw. The new estimated BtlBw is set to the maximum delivery rate observed during this probe. The RTprop estimation expires after 10 seconds, causing BBR to enter $ProbeRTT$ to re-estimate this value. In ProbeRTT, BBR reduces its inflight packets to just 4 per RTT, thus draining any queue that it had built up. The new RTprop estimate is set to the minimum RTT observed. Finally, during steady-state, BBR paces its sending rate at the estimated BtlBw and caps its inflight CWND to two times the estimated BDP. The CWND inflight cap is set to two rather than the theoretically optimal utility value of one to accommodate delayed and stretched ACKs in wireless networks. However, as we show in section 5, the larger CWND can cause high packet loss, instability, and unfairness.

Because BBR relies on *estimated* RTprop ($\widehat{RTprop}$) and BtlBw ($\widehat{BtlBw}$), BBR has inconsistent behavior when it mis-measures one or more of these values. Hock et al. [17] find that when multiple BBR flows share a bottleneck, BBR pathologically over-estimates its fair-share of the bandwidth since each flow measures the maximum available bandwidth over a time period. Because each flow takes a maximum, the sum of throughput's (derived from these estimates) is *always* greater than the bottleneck's actual maximum bandwidth, causing persistent queues to build at the bottleneck router until the inflight cap of 2 BDP is reached [17]. This persistent queue is especially problematic when the bottleneck router queue is smaller than a single BDP, whereupon BBR attempts to build a queue of 1 BDP, and ignores the massive packet loss caused by overwhelming the bottleneck queue.

Scholz et al. [23] and Miyazawa et al. [22] show that that BBR also produces inaccurate $\widehat{RTprop}$ estimates when it shares the bottleneck with buffer filling protocols such as CUBIC. When BBR over-estimates RTprop, it drastically changes its CWND and thus creates large amounts of loss. This loss and mis-measurement leads to a cyclic behavior where BBR and CUBIC each have constantly fluctuating throughput.

We confirm these prior results and explain these findings in depth in Section 5, as well as present possible fixes for BBR in Section 5.5.

2.4 Summary

Loss-based TCP congestion control does not operate at the optimal operating point, but instead fills the bottleneck with packets until loss is observed, resulting in high-latency connections or buffer bloat. More modern TCP congestion control schemes attempt to operate closer to the optimal operating point. These modern protocols can be split into two categories: utility-based congestion control protocols that treat the network as a black box and use a utility function to evaluate their own behavior, and measurement-based congestion control protocols that measure specific variables in the network to take specific actions to operate near the optimal. TCP BBR is one such measurement-based protocol that works by measuring the RTT and throughput, and uses these to calculate the BDP. Using the calculated BDP, BBR attempts to operate with around 1 to 2 BDP of packets in flight, and thus operate near the optimal.

While BBR has been successfully deployed by a number of companies, current research shows that BBR may not be stable in all network conditions. Specifically, because BBR relies on measured RTT and throughput values, it has very inconsistent behavior when it mis-measures these values. This mis-measuring happens when BBR competes for a network bottleneck with TCP CUBIC, and when BBR competes for a bottleneck with a small router queue. We confirm these prior results and explain these findings in depth in Section 5, as well as present possible fixes for BBR in Section 5.5.

3 Methodology

We setup a hardware testbed and develop a set of custom tools that enable a variety of network experiments for evaluating TCP CUBIC and TCP BBR in a controlled environment.

3.1 Hardware

Our testbed, named 'Panaderia' [1], is depicted in Figure 3 and consists of eight Raspberry Pi computers, two network switches, and one Linux PC functioning as a router ("Horno" [2]). The hardware is configured in a traditional dumbbell topology (Figure 4) - the Raspberry Pis are split into two subnets of four machines (a "churros" cluster and a "tartas" cluster).

Figure 3: Panaderia testbed

Each Raspberry Pi is a model 3B+ running the Linux kernel 4.17. Our experiments show that the individual Raspberry Pis have a maximum sending rate of roughly 225 Mb/s limited by the USB 2.0 bus speed. Below this throughput, we verified that the Raspberry Pis perform similarly to traditional Linux computers. We use a series of Python scripts to allow us to nimbly run experiments, vital for comparing BBR's behavior over a wide range of network conditions. We provide details on the setup specifics as well as access to our configuration scripts on GitHub[3].

The router is a Linux PC, configured with an Intel i7 CPU, 12 GB of RAM, and Broadcom BCM5722 Gigabit Ethernet PCI cards. The router uses NetEm [15] to add a fixed

[1]Panaderia means "bakery" or "bread shop" in Spanish
[2]Horno means "oven" in Spanish
[3]https://github.com/SaahilClaypool/rpi/tree/master/Config

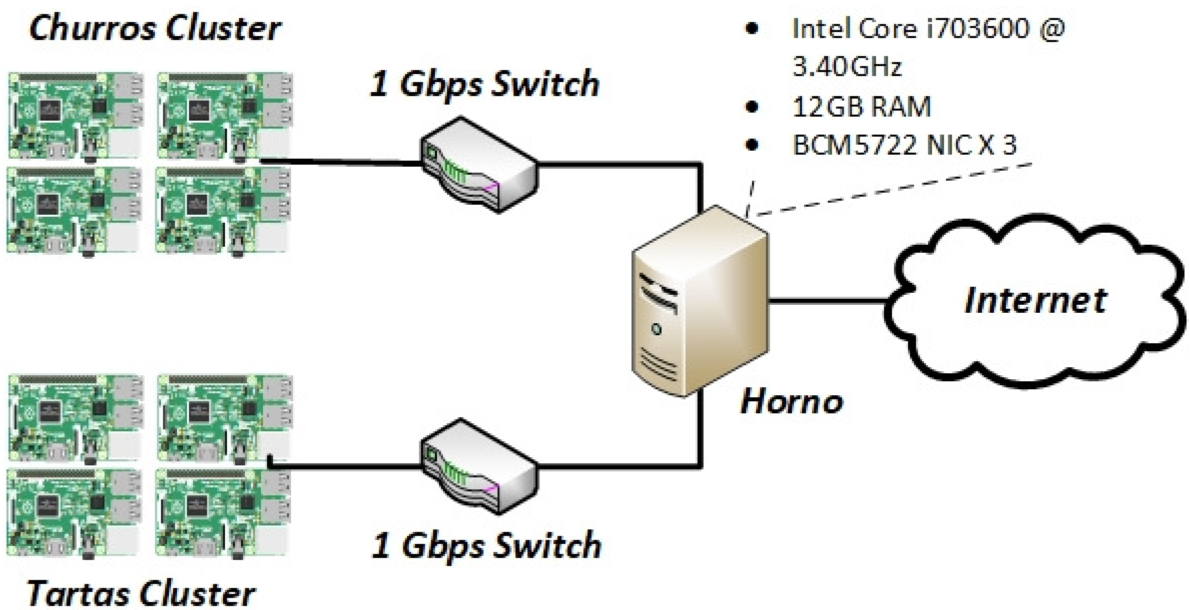


Figure 4: Hardware testbed topology

propagation delay of 24 ms giving a total propagation time of 25 ms to align with previous research [8]. The router also uses tc-tbf token bucket filters to control the bottleneck bandwidth [3]. Accumulated tokens can cause bursts of traffic interfering BBR's BtlBw estimate. Thus, we use *two* token bucket filters with small buckets to limit the the instantaneous burst [2]. Receiving and sending devices each collect packet captures (pcaps) which are then analyzed for round trip time, throughput, and inflight packets. Additionally, the router collects data on the total number of bytes queued, as well as the number of packets dropped.

3.2 Orchestrating Experiments

We use a series of Python scripts to allow us to nimbly run experiments, vital for comparing BBR's behavior with various network conditions. The data flow for a *single* experiment, depicted in the bottom half of Figure 5, is as follows: first, a configuration file is used to set up the servers and routers and start the experiment (Section 3.2.2) during which pcap files are recorded. The experiments use our custom data sender and server (Section 3.2.1) to create traffic in the network. During this time, another Python program polls the router for statistics, including queue length, drop rate, and total sent data, and stores this data in a

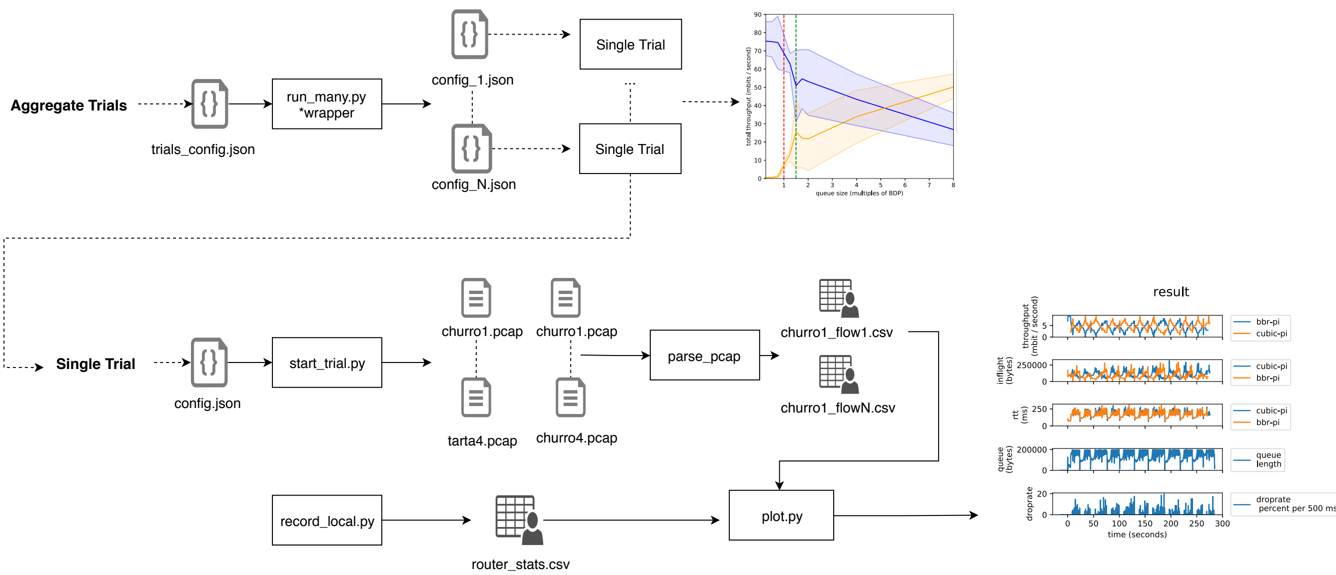


Figure 5: Experimental Data Flow

CSV file. Then, our parser program converts these pcap files into CSV files (Section 3.2.3). Finally, our plot.py program aggregates these CSV files into a single image such that it can be used to visually inspect the results of the trial.

Comparing BBR and CUBIC's performance over many different network conditions requires the ability to run many identical trials while only changing one condition. To run these trials, we wrote another tool to generate the configurations required to use our single-trial tools, and then summarize the results over the range of network conditions (Section 3.3).

3.2.1 Packet Sender Tool

To test the performance of different network congestion protocols, we needed a program to put load on the network. Originally, we used IPerf [12] which is commonly used for testing network performance. The IPerf program running as a server on one machine and connects a client to the server from another machine. The client sends data (over TCP) as quickly as it can, and the server ACKs each packet as quickly as the network transmits them. This setup is different from the traditional 'server' model where the server is usually the machine sending the majority of the data (such as when a phone is downloading a webpage from the server). While we used IPerf for much of our early work, IPerf only allows a single client and server per machine. This limitation is intentional as IPerf is designed to test the actual network connections, and allowing multiple clients on a single network could skew these results.

Testbed experiments often need to be able to run multiple flows from a single machine testing more than 1 flow per machine. So, we designed 'ServerSender[4]' program with an interface nearly identical to IPerf, but with the additional capability of starting multiple flows from a single client, and staggering the offset of these new flows by a set time. When run in 'server' mode, this program listens for a connection on a given port, and when it receives a 'client' stream on that port, it starts a new thread to ACK each packet received. In the 'client' mode, the program starts a number of threads specified by the user, and for each of these threads it makes a connection to the server. Each thread sends 1,000 byte chunks of data as quickly as possible, so the only limiting factor is the sender CWND and network capacity. This process continues for the number of seconds specified by the user.

3.2.2 Running a single Trial

The most important program in our setup is the 'start_trial.py' program which allows us to set up the router and each of the servers and clients from a single configuration file, as can be seen in the bottom half of Figure 5. This is what enables quick changes between different router and client configurations; the only changes required to vary the number of clients and servers or the throughput and round trip time are a couple of line changes in a single file. This program works by reading in a JSON configuration file that contains the following: A name (to uniquely prefix the graph outputs in the next section), a time that specifies the length of each connection, a 'setup' or pre-experiment setup section, a 'run' section to kick-off the experiment, and the 'finish' section to remove any side effects from the trial.

In the *setup* section, each host is configured with a JSON object. This section is mostly used to specify the router configuration, and to set the congestion control protocol used for each sever and client. The router is configured by loading a shell script file to set the parameters for the token bucket filters (limiting throughput) and netem (to inflate the round trip time). Each key in the configuration file is automatically run over SSH on the

[4] https://github.com/SaahilClaypool/NetworkTools/tree/master/ServerSender

host, allowing the configuration of the remote hosts from this single central configuration file. Additionally, a 'tcpdump' program is started on each client and server during this phase to capture the network performance over the course of the experiment. It is important to start tcpdump on *both* the client and server machines as the observed throughput may be asymmetric - the sender application may observe a high throughput as it sends a large window of data, but this data may just be queueing at the bottleneck router and never observed at the receiving application. Finally, each of the designated 'server' applications have to begin the server process, using the tool described in the previous section.

The *run* section is where the actual trial is started. Again, each host is configured with a JSON object where each command is automatically run over SSH on the remote client. In this step, each of the clients launches the sender application with the specified number of flows.

The *finish* phase is run after all processes end in the run phase. This phase kills any hanging servers and tcpdump programs.

After running each of the phases, the 'start_trial.py' program copies each packet capture file from the remote hosts to the local machine, and names them according to their hostname. This consistent, programmatically defined naming is vital for allowing us to automate the parsing and visualization of each network trial without any manual intervention (discussed in the next section).

3.2.3 Automated parsing and plotting of pcap files

Once all of the pcap files are copied to the local machine, our tool automatically parses them into CSV files, and a Python script generates plots for that trial. This allows the visual inspection of every trial without manually writing scripts for plotting every trial.

The parser program works as follows: given a directory of pcap files, it identifies the 'sender' and 'receiver' applications by their host names, and then reads the binary pcap format inspect the packets send and received. For the receiving hosts, it measures the observed throughput as the number of bytes received over a specified time interval (the goodput). For each sender host, the program records the current inflight data, and round trip time observed over the trial. It combines these statistics into CSV files for easy use by

Python. This parsing could have been done by a program such as TShark (a command line program provided by WireShark), but writing a custom tool automates the identification of sender and client applications based on our consistent naming schemes.

Once the CSV files summarizing the network traces are written, a Python program takes the set of files generated and create a stacked plot of the throughput, round trip time, router queue length and loss (collected and parsed by another program). Again, because we relied on convention over configuration for our file structure and naming schemes, these graphs could be generated identically for each and every trial we ran. While the resulting graphs are not 'journal quality', they are vital in allowing us to quickly compare network statistics over a number of different severs and clients in a single view, and thus allowing us to determine which configurations were worth investigating further. An example plot showing our initial look at cyclic BBR and CUBIC performance can be seen in Figure 6. Again, these graphs are purely for exploration - we use these to identify which results should be pursued further. For the graphs generated in Section 5, we used the same pcaps generated from these trials, but manually recreated the graphs for clarity. In total, we were able to run over 1,200 1 to 5 minute trials comparing BBR and CUBIC, corresponding to roughly 100 hours of network experiments, providing over 100 Gigabytes of packet captures, and, most importantly, over 6,000 graphs automatically generated such that that we could visually inspect each trial without further programming. This efficiency allowed us to test the wide range of conditions needed to understand BBR's performance and responses over the wide range of network conditions explored.

3.3 Evaluating BBR and CUBIC

To compare BBR and CUBIC or a larger range of network conditions, we create another Python program to generate configuration files for the program 'start_trial.py' described in Section 3.2.2. The dataflow for this tool can be seen in the top half of Figure 5. Listing 1 demonstrates a sample configuration file used to run a series of trials used to create Figure 13. This configuration sets the throughput to 80 Mb/s, and the RTprop to 25 ms. Then, for each BDP (defining the router queue length in terms of BDP), our scripts will run 3 trials for 5 minutes collecting network and end-host statistics.

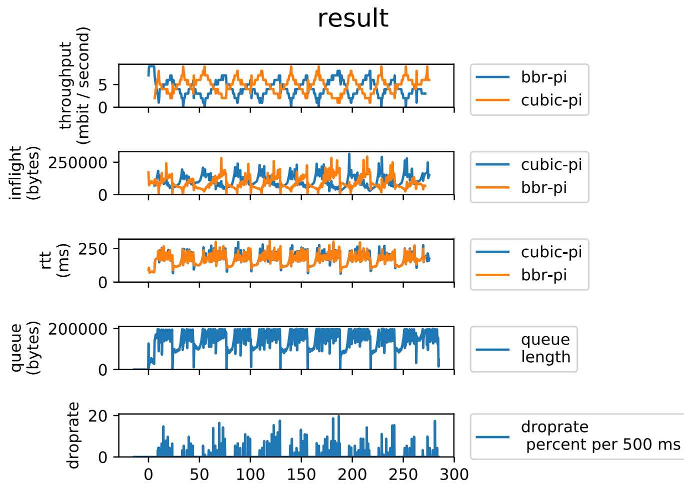


Figure 6: Rough plot of cyclic BBR and CUBIC performance

Listing 1: Configuration Example

```
{
  'throughput': 80,
  'delay': 24,
  'BDP': [
      0.25, 0.5, 0.75, 1.25, 1.5, 1.75, 2, 4, 8
  ],
  'trials': 3,
  'time': 300
}
```

A single configuration file minimizes the chance of configuration errors across runs, and automatically couples setup and configuration with experimental data.

After preliminary tests to validate our testbed, we use the tools above to get a deeper look at the cyclic performance exhibited by BBR and CUBIC. In each experiment trial, we set 2 to 4 computers as receivers, simply acknowledging packets, and 2 to 4 computers to be senders, sending data as fast as possible. We focus on varying the bottleneck capacity from 40, 80 and 120 Mb/s, and router queue size as a function of the BDP from $\frac{1}{4}$ to 8 BDP. Additionally, we vary the number of BBR and CUBIC flows that are competing at the bottleneck. These parameters can be seen in Table 1.

Table 1: Experiment Parameters

Parameters	Values
Capacity	40, 80, 120 Mb/s
Network Congestion Control Protocol	CUBIC, BBR
Router Queue Length	0.25, 0.50, 0.75, 1.00, 1.25, 1.50, 1.75, 2.00, 4.00, 8.00 BDP
Flows	2, 4, 8

3.4 Summary

We create a hardware testbed composed of 8 raspberry pi computers and a number of software tools to allow us to automate much of the experimentation process. This allows us to run a huge amount of network experiments analyzing BBR's behavior in various network conditions. Specifically, we set up experiments to test how BBR and CUBIC's performance changes when competing for a bottleneck router with varying flow numbers, queue lengths, and throughputs.

4 Validation

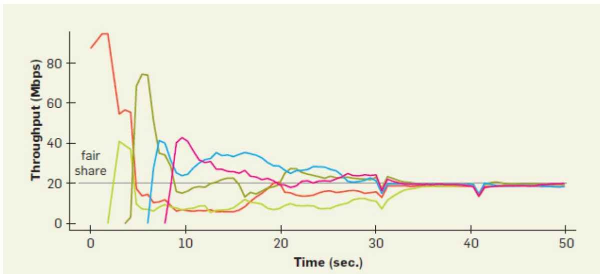
When creating a novel testing environment, it is important that we validate its behavior against known results. Further, it is vital that confirm that prior work is reproducible to ensure that we - and other - researchers are building upon a solid foundation. We validate the TCP performance of the Raspberry Pis by confirming the behavior of BBR follows the behavior seen by Cardwell et. al [8].

This includes: 1) synchronization of multiple flows, and 2) adjustments to the CWND corresponding to increased or decreased throughput.

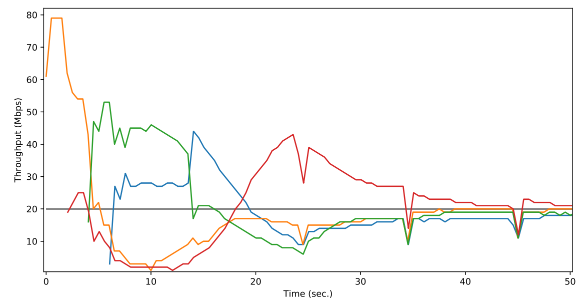
4.1 Synchronizations of Multiple Flows

A key criteria for a congestion control protocol is that it achieves a fair and stable operating point when it is competing with flows of the same type. Cardwell et. al. [8] indicate that when multiple BBR flows compete for a single bottleneck, the flows should obtain a fair share of the bandwidth. Further, their RTT estimates should expire at the same time, causing them to enter ProbeRTT at the same time, and thus simultaneously obtain accurate RTprop estimates. This accurate agreement in RTprop is vital in ensuring the BBR flows receive a fair share of the bandwidth because, as we discuss in Section 5, the CWND dictates the observed fairness of congestion control protocols, and BBR's CWND is derived from this RTprop estimate.

Figure 7a depicts the throughputs of 5 competing BBR flows with staggered start times competing for a 100 MB/s bottleneck with a RTprop of 10ms, as evaluated by Cardwell et. al. [8]. As seen in this figure, each of the flows obtains a fair share of the bandwidth, 20 MB/s, and each flow enters ProbeRTT at the same time, as seen around time 30, and 10 seconds later at time 40. We confirm this behavior in similar conditions in our Panaderia testbed by running four competing staggered BBR flows for an 80 MB/s link (Figure 7b). Here we use 4 flows such that each Raspberry Pi has only a single client or server running to ensure no confounding factors from having more than one flow per machine. Similar to Cardwell et. al., we confirm that the BBR flows do in fact synchronize at a fair share. Further, this supports that the *Panaderia* produces 'reliable' results comparable to other hardware testbeds, thus supporting the validity of our other findings as well as furthering Cardwell et. al.'s work by showing it can be reproduced.



(a) Synchronization of BBR, as shown in Figure 8 of Cardwell et. al. 2017 [8]

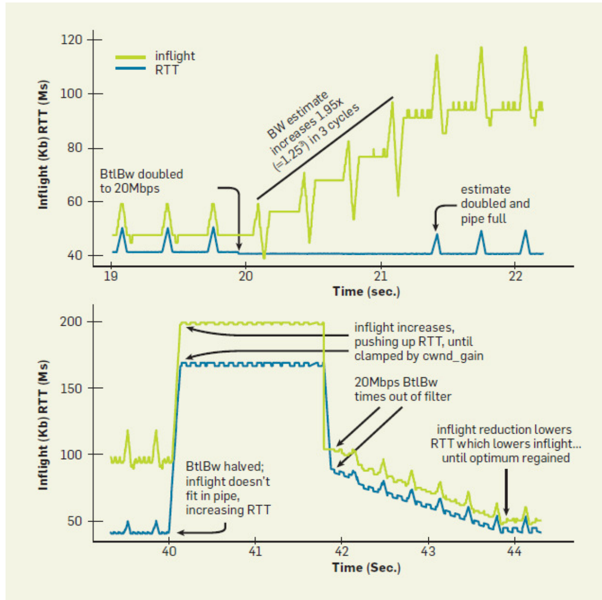


(b) Synchronization of BBR in the Panaderia

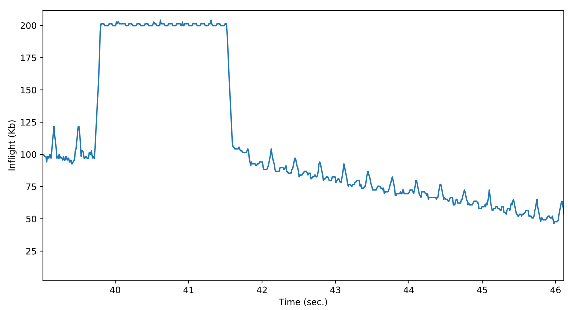
Figure 7: Comparison of Cardwell et. al. 2017 to the our Panaderia testbed

4.2 Dynamic CWND

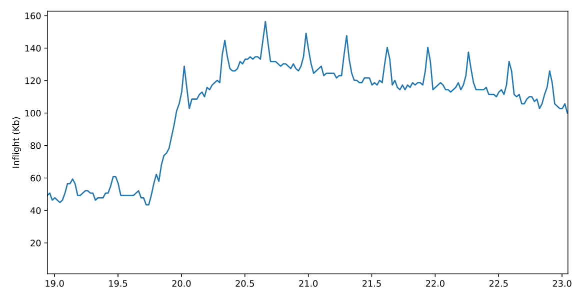
BBR is designed to adjust its BtlBw at an exponential rate to capitalize on available bandwidth. Cardwell et. al. [8] demonstrate this (Figure 8a) by running at single BBR flow through a 10 MB/s 40ms bottleneck and at time 20 abruptly doubling the bandwidth to 20 MB/s for 20 seconds and finally dropping the bandwidth back to 10 MB/s. Again, we run an identical test in our Panaderia testbed, shown in Figure 8c and Figure 8b. While the the observed throughput of BBR looks nearly identical when the throughput is decreasing (Figure 8b), the behavior when the bandwidth *increases* is somewhat different between Figure 8c and Figure 8a. Rather than increasing the bottleneck bandwidth through probes over a series of RTT, this testbed shows BBR increases its bottleneck bandwidth at a smooth exponential rate. We do not believe this is an artifact of our testbed, but rather because we are using a slightly different *version* of BBR. We evaluate our tests on the Linux kernel 4.17 which includes changes to the original BBR specification. In fact, our observed behavior closely matches the rendered proposals in [7], which discuss increasing BBR's speed to acquire available bandwidth. Although these proposals lacked a similar graph showing ramp up behavior from a real-world test, their 'rendered' behavior (Figure 8d) is very close to our observed ramp-up behavior. Thus, our test both reproduces this new BBR behavior and confirms that our testbed produces results that align with prior work.



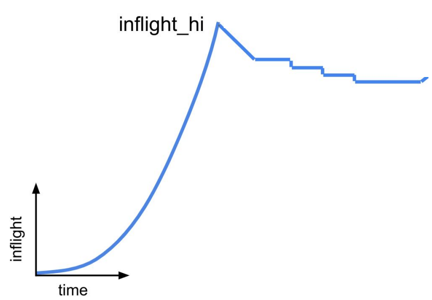
(a) BBR quickly maximizes throughput, as shown in Figure 5 Cardwell et. al [8]



(b) Decreasing throughput, seen in the Panaderia



(c) Increasing throughput, as seen in the Panaderia



(d) BBR Proposed ramp up changes, as seen in Slide 11 of IETF 102 [7]

Figure 8: BBR quickly adjusts to lower throughput

4.3 Summary

We demonstrate that our testbed is capable of producing results nearly identical to those published by Cardwell et. al. [8]. These results support the validity of our other findings as our testbed provides identical behavior to the test environments used by prior work. Further, we confirm that the prior work is in fact reproducible, which is vital for ensuring that we and other researchers are building upon a solid foundation.

5 Results

We validated BBR's behavior in our testbed based on prior work (Section 5.1). We then extend previous work by evaluating BBR's behavior in shallow buffers (Section 5.2) and as a function of the router queue length (Section 5.3). Next, we evaluate BBR's interplay with CUBIC, again focusing on the relationship with router queue length (Section 5.4). Finally, we propose mechanisms to improve BBR's performance in adverse conditions (Section 5.5).

5.1 Standard BBR Behavior

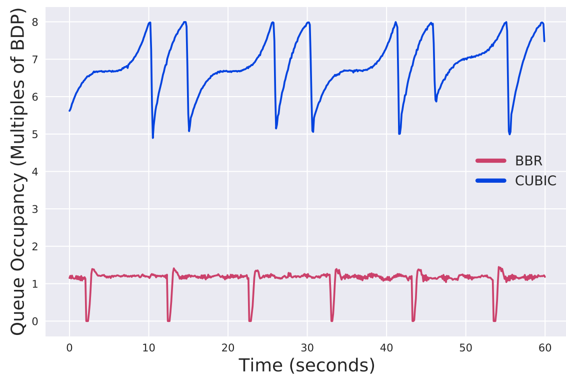
Prior work has shown that when there is more than one flow competing for the same bottleneck, BBR tends to create a 1 to 1.5 BDP standing queue [17]. We verify this behavior by running 2, 4, and 8 BBR flows for 5 minutes at 40, 80, and 120 Mb/s with a 25 ms RTprop and a large bottleneck queue.

Figure 9 depicts 4 BBR flows competing for an 80 Mb/s link with a maximum router queue to 2 Mbytes ($8BDP$). We show only the steady-state behavior (1 minute of a 5 minute trial). For reference, we run and show an identical and independent trail using TCP CUBIC to compare their behavior. CUBIC, as a loss-based protocol, continues to fill the queue until the 8 BDP maximum. BBR creates a consistent queue of 1.1 BDP increasing to roughly 1.5 BDP during each ProbeRTT phase.

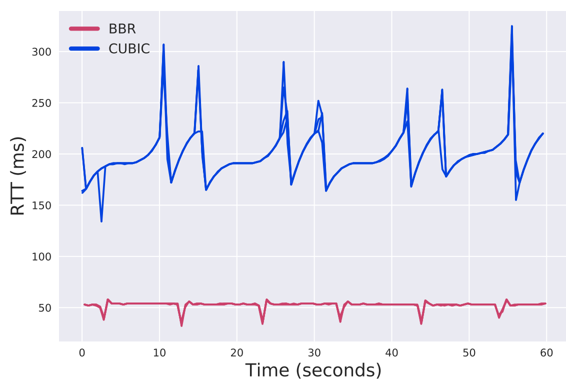
In each of these scenarios, our results confirm that BBR does create a standing queue of 1 to 1.5 BDP. When the queue is large enough to contain this excess 1.5 BDP, BBR behaves as expected - it exhibits relatively low RTT and high utilization. This behavior can be seen in Figure 10 Note that, while the RTT is low (roughly one-fourth of CUBIC's), it is around *double* the RTprop. One BDP queued at the router takes a full RTT to process ($BDP = BtlBw \times RTprop$).

5.2 BBR in Shallow Buffers

Since multiple BBR flows competing at a bottleneck create a 1 to 1.5 BDP queue at the bottleneck router, when routers have shallow queues they cannot hold the excess packets, and thus BBR creates a huge amount of loss.



(a) Queue Occupancy



(b) Round trip time

Figure 9: BBR and CUBIC in a large queue

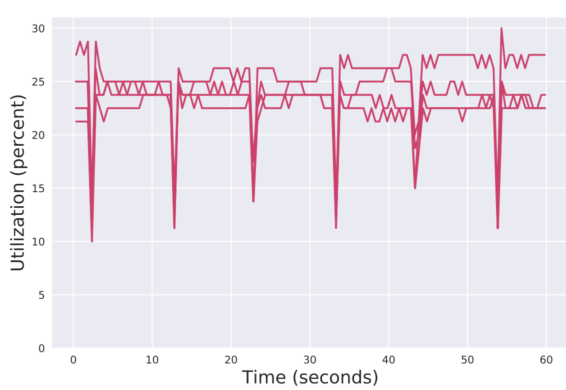


Figure 10: BBR's throughput in a large queue

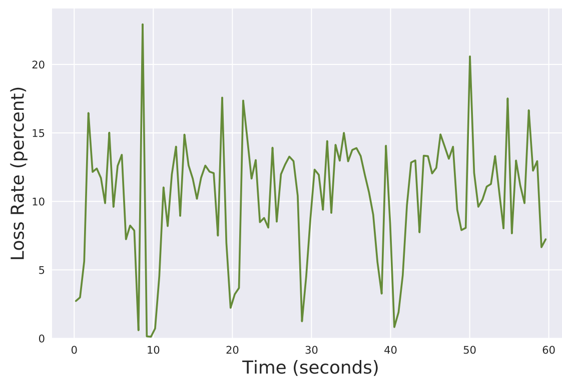


Figure 11: BBR's loss rate in a small queue

Figure 12: BBR's loss versus router queue length

We demonstrate this behavior by running a similar experiment to Section 5.1 with a bottleneck bandwidth of 80 Mb/s, a RTprop of 25 ms, but a bottleneck router queue of just 0.5 the BDP. Figure 11 depicts the packet loss averaged over half second intervals during the course of the flow. In this case, when the router queue is small, the excess packets are dropped by the router, causing persistent, high packet loss.

5.3 BBR over Different Router Queue Lengths

We evaluate BBR's loss rate over a range of router queue lengths. Specifically, we run 3 identical trials at 40, 80, and 120 Mb/s, all at 25 ms RTprop for 5 minutes for each given queue size: 0.25, 0.50, 0.75, 1.25, 1.50, 2.00, 4.00, and 8.00 BDP. We use the recorded packet captures at each host and the queue statistics at the bottleneck router to determine the

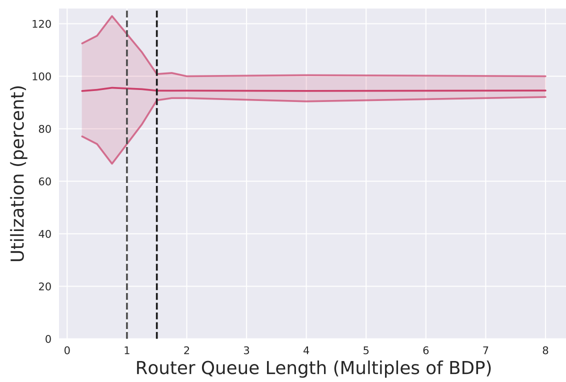


Figure 13: BBR's network utilization versus router queue length

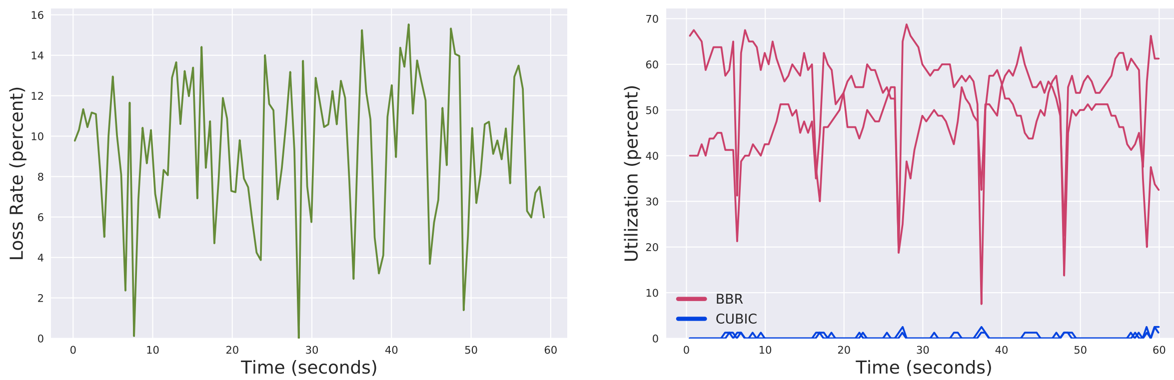
aggregate behavior of BBR given the bottleneck queue size.

Figure 12 depicts the loss rates of these trials. The y-axis depicts the loss rate during steady state (minutes 2 to 4 of a 5 minute connection) averaged over the 3 trials. The x-axis depicts the maximum router queue length in terms of the BDP.

Figure 13 depicts the network utilization over the same trials. The y-axis depicts the total utilization by all BBR flows during steady state (minutes 2 to 4 of a 5 minute connection) averaged over the 3 trials. The x-axis depicts the maximum router queue length in terms of the BDP.

There is a an extremely high loss rate when the queue length is less than a BDP. In fact, the loss remains high until the queue length is at least 1.5 times the BDP. An extra 1.5 BDP of packets are enqueued at the bottleneck router, causing loss when the router has a queue any smaller than 1.5 times the BDP. This result confirms the findings of Hock et al. that BBR's inflight in practice is 2.5 times the BDP [17].

Thus, as BBR does not respond to loss as a congestion signal, the queue occupancy will always grows to 1.5 BDP. BBR's throughput remains relatively high despite high loss. From here on, we refer to any buffer less than 1.5 BDP as shallow, buffers 1.5 BDP to 4 BDP as medium, and buffers greater than 4 BDP as deep. Note that all of these are relative to the BDP - in a high throughput, high RTT connection, a 'shallow' buffer of 1 BDP could be large in terms of bytes.



(a) Loss rate

(b) Throughput utilization

Figure 14: BBR against CUBIC in a shallow buffer

5.4 BBR's interplay with CUBIC

Another concern with BBR is its interplay with CUBIC and other loss-based congestion control protocols. BBR's mechanism for controlling the bottleneck bandwidth is at odds with CUBIC's - CUBIC only adjusts its CWND to minimize loss, while BBR mostly ignores loss as a congestion signal. This difference presents itself uniquely in each of the conditions discussed above - shallow, medium and deep buffers.

5.4.1 BBR and CUBIC in Shallow Buffers

By itself in shallow buffers, BBR creates high amounts of 'ambient' loss by growing its CWND beyond what the network can handle, as seen in Figure 11. Because CUBIC treats this loss as congestion in the same scenario, CUBIC shrinks its CWND to reduce loss. Figure 14b depicts the relative network utilizations of BBR and CUBIC competing over a shallow ($\frac{1}{2}$ BDP) buffer in an 80 Mb/s and 25 ms connection. The drop rates look similar to Figure 11, averaging about 10 percent. As seen in Figure 18, the relative network utilization for CUBIC is *much* lower than BBR because, again, there is high loss created by BBR.

We ran this trial with two servers running CUBIC and two servers running BBR. We also ran similar trials with 120, 80, and 40 Mb/s connections, and with one of each flow type instead of two. Each of these resulted in a similar output once averaged over BDP and expected fair share, as can be seen in Section 6.2.

When CUBIC is competing with BBR through a shallow buffer, CUBIC observes a high loss rate. As a result, CUBIC frequently shrinks its CWND, which in turn causes CUBIC to receive an unfair share of the network capacity. BBR on the other hand continues to maximize its own throughput as it does not respond to loss.

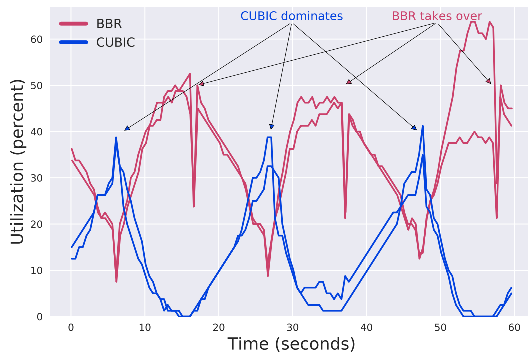
5.4.2 BBR and CUBIC in Medium to Large Buffers

In medium buffers, where BBR is *not* persistently inducing loss, BBR and CUBIC display a cyclic behavior. We demonstrate this by running BBR and CUBIC through a bottleneck configured as above with an 80 Mb/s bandwidth and 25 ms RTprop, but with a bottleneck router queue of 1.75 BDP. Figure 15 shows the results, annotated to match the explanations that follow. BBR and CUBIC exhibit cyclic performance - they alternate which flows dominate the connection over a very regular 20 second period, confirming prior results by Scholz et al. [23] and Miyazawa et al. [22]. We build upon this work by explaining the factors that cause this cyclic performance, as well as the aspects required to cause these cycles.

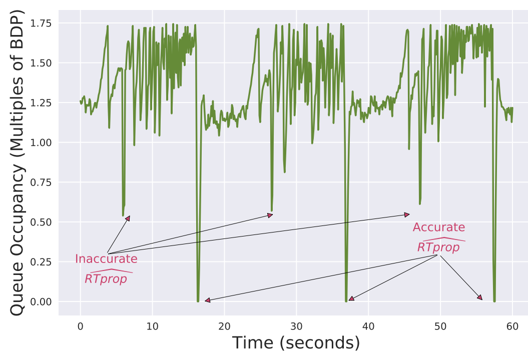
CUBIC Dominates. When BBR has an *accurate* estimate for the throughput and RTprop, it caps its inflight at 2 $\widehat{BDP}$. This means that BBR allows just 1 BDP of packets to queue at the bottleneck router for 8 RTTs. For 8 RTTs, CUBIC is thus able to expand its CWND to 0.75 BDP before seeing loss.

Since observed throughput is proportional to the fair share at the bottleneck router, as CUBIC gets more packets in queue, BBR observes a lower throughput, and thus further decreases its CWND, which is derived from the observed throughput. These responses create a positive feedback loop allowing CUBIC to continue increasing its CWND as BBR continues to back off as it observes a reduced throughput.

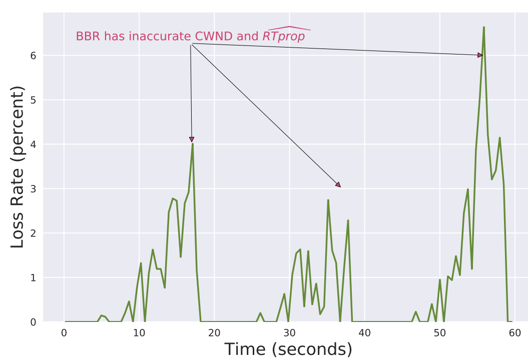
BBR takes over. Every 10 seconds without observing a new minimum RTT ($\widehat{RTprop}$), BBR probes for RTprop by reducing its inflight packets to just 4 packets to drain the router queue [8]. BBR uses the minimum observed RTT as the new $\widehat{RTprop}$ (time 28 of Figure 15a). However, the queue length, shown in Figure 15b, does not change significantly because most of the packets in the queue are from the CUBIC flows. This means that even when BBR decreases its inflight packets, the queue stays relatively filled. Figure 15d depicts the RTT over this period, where around time 28, the RTT is still much higher than the true RTprop:



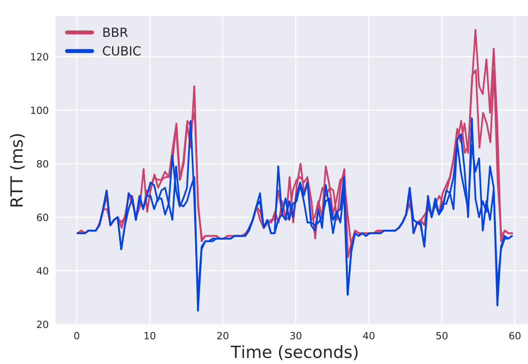
(a) Network Utilization for BBR and CUBIC. Cyclic.



(b) Queue Length. Not always draining



(c) Drop rate. High loss every 20 s



(d) RTT. Cyclic.

Figure 15: BBR against CUBIC shows cyclic performance in medium buffers

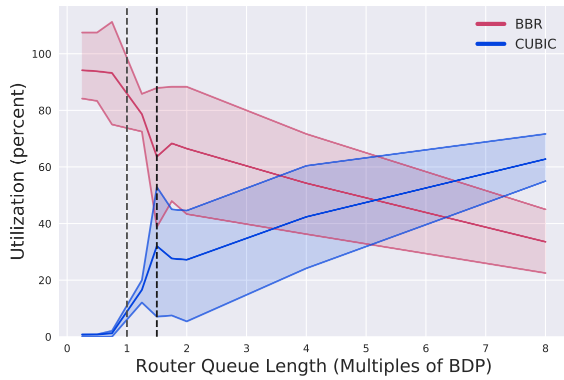


Figure 16: BBR and CUBIC's interplay versus router queue length

around 60 ms rather than 25 ms. This high RTT causes the $\widehat{RTprop}$ to be too large, and, because BBR's $\widehat{BDP}$ and thus CWND is derived from this RTprop, BBR greatly increases its CWND.

Because the router queue is already filled by CUBIC, this increased CWND causes a large amount of packet loss for both CUBIC and BBR. BBR ignores the loss, but CUBIC backs off, decreasing its inflight data. This loss can be seen at around time 10, 30, and 50 of Figure 15c, each of which corresponds to just after BBR increases its CWND after an inaccurate RTprop probe.

This behavior continues for 10 seconds, whereupon BBR again probes for RTprop. The probe obtains an accurate $\widehat{RTprop}$ of 25 ms, as seen at second 38 of Figure 15b, because the queue is fully drained, and thus BBR reduces its CWND accordingly. This allows CUBIC to grow its CWND, as discussed above, and the cycle repeats.

We visualize this cyclic behavior over a range of queue sizes: 0.25, 0.50, 0.75, 1.00, 1.75, 2.00, 4.00, and 8.00 BDP. We record the drop rate and the throughput averaged over both BBR flows and both CUBIC flows at 40, 80 and 120 Mb/s. These results for the 80 MB/s connection are depicted in Figure 16[5]. The x-axis depicts the varying queue length for each trial, and the y-axis the percent of throughput utilized by the flows, which are grouped by congestion control protocol. This figure shows the relative network utilization (mean) of BBR and CUBIC, as well as their 75th percentile and 25th percentile utilizations as averaged over half second intervals. As seen in the figure, the behavior drastically changes

[5]See Section 6.2 for graphs of the 40 and 120 MB/s connections

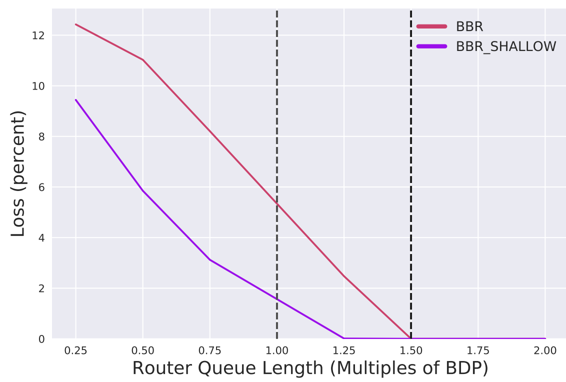


Figure 17: Modified BBR_Shallow's loss versus router queue length

at 1.5 BDP where BBR stops creating persistent loss. After this point, the interquartile range of utilization greatly increases, indicating the cyclic performance has begun within these flows.

As the bottleneck queue gets larger, BBR becomes more limited by its 2BDP CWND. This causes CUBIC to progressively obtain more of the throughput as its CWND grows beyond BBR's CWND limits.

5.5 Improving BBR's Performance

We have identified two weaknesses in BBR that affect performance: BBR's static 2BDP CWND, and BBR's inaccurate RTprop estimation. We discuss proposals to fix these issues below, with a proof of concept evaluation. These preliminary results are meant as an inspiration, not as a vigorous implementation.

5.5.1 CWND adjustment with a feedback loop

Currently, BBR caps the inflight packets at 2BDP, causing the 1 to 1.5 BDP of packets at the bottleneck router, and a high amount of loss in shallow buffers. These results indicate that this 2BDP is sometimes too large. However, when the bottleneck router queue is large and BBR competes with a loss-based congestion control protocol, then its CWND limits its queue share, indicating the CWND is too small. Thus, BBR needs a dynamic CWND to fit both of these circumstances.

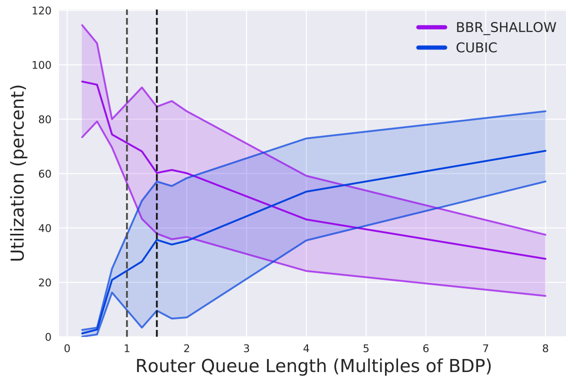


Figure 18: BBR_Shallow and CUBICs interplay over a range of queue lengths

We propose a feedback loop in Algorithm 1 to dynamically adjust BBR's CWND cap, rather than the current static 2BDP cap.

Algorithm 1: CWND Adjustment

Data: $loss, throughput, RTT, \widehat{BtlBw}, \widehat{RTprop}$

if $loss > threshold$ **then**

  Decrease CWND `// Must be over saturating the queue`

else if $throughput < \widehat{BtlBw}$ **then**

  `/* Must be underutilizing our share of bandwidth. Or, there are delayed acks. Need a larger CWND */`

  Increase CWND

else if $RTT < \widehat{RTprop}$ **then**

  Decrease CWND `// Must be over saturating the router`

else

  Maintain CWND

We demonstrate that BBR's inflight cap is responsible for its fair share behavior and shallow buffer loss by manually adjusting BBR's inflight cap to 1.5 BDP. We then rerun each of the above tests with this modified 'BBR_Shallow[6]'. As seen in Figure 17, which follows the same format as Figure 12, BBR_Shallow creates less loss in buffers that are below 1.5 BDP because fewer packets are attempting to be queued at the bottleneck router. When these flows are run against CUBIC, their fair share is much lower, as shown in Figure 18, following the format of Figure 16. Instead of having a 50 percent share when the bottleneck

[6] `https://github.com/SaahilClaypool/raspberry-linux/blob/bbr2-patched/net/ipv4/tcp_bbrshallow.c`

router is around 5 BDP, the fair share is now achieved when the bottleneck router is just 3 BDP. Thus, the CWND seems to determine inter-protocol fairness.

5.5.2 Improving BBR's RTprop estimation

The RTprop should be nearly constant for the same network path, only ever changing after a route change. However, when BBR competes with CUBIC, BBR *increases* the estimated RTprop when it is unable to drain the queue, causing BBR to greatly over estimate the BDP, and thus create high queueing and loss.

BBR could instead *always* use the minimum of any RTT it sees - basically it should never increase the estimated RTprop. Always using the minimum in this way would no longer handle route changes that result in a higher RTprop. Either BBR could ignore route changes completely (as they are uncommon, and most flows are short [10]) or BBR could only accept a higher RTprop if it is *consistently* higher for many RTprop probes or drastically higher for a single probe. Thus adjusting only to consistently higher RTprop probes might allow BBR to detect a route change without increasing its CWND erroneously.

5.6 Summary

First, we confirm that BBR creates a high amount of loss in shallow buffers. We evaluate BBR over a wide range of buffer queue lengths to find the percent of packets lost as a function of the router queue. We find that as long as the router queue is less than 1.5 BDP, BBR will create a high amount of loss, regardless of the throughput or the number of flows competing for the bottleneck.

Next, we confirm that BBR and CUBIC exhibit cyclic performance when they compete with each other at the bottleneck. We are able to determine the exact mechanisms that cause this: BBR's inaccurate RTprop estimation and static CWND size. Thus we finish by proposing new mechanisms the improve BBR's RTprop estimation and a new feedback loop to adjust BBR's CWND to reduce loss and increase fairness dynamically.

6 Conclusions

In this section, we summarize our work and suggest new avenues of research for congestion control.

6.1 Summary

We present details on a hardware testbed and a set of custom tools to automate network congestion control experiments. This design allows us, and other researchers, to better analyze modern congestion control protocols such as BBR. Our testbed consists of: 8 Raspberry Pi's, two network switches, and a Linux PC router. Our testing infrastructure combines this setup with a series of Python scripts and other programs, which include tools to run and record the results of a single trial, scripts to run a series of trials over a set of parameters, and tools to automatically parse and plot each of these trials. In total, this configuration allows us to gather and analyze over 100 hours of experiments corresponding to 6,000 plots summarizing each trial. Specifically, this testbed enabled us to capture and analyze the performance of BBR and CUBIC over a wide range of BDP providing insight on the specific causes for the cyclic performance exhibited between these two protocols.

As BBR becomes more widely adopted, it is important that it provides consistent behavior when run with existing protocols such as CUBIC and over a range of router queue sizes. Currently, BBR's high loss rates and throughput variations under these scenarios could be disastrous for applications that rely on a stable network connections for good performance.

Analysis of the results provide a deep look into the behavior of TCP BBR in relation to shallow buffers and competition with CUBIC. Confirming prior work, we find that BBR creates high loss in shallow buffers, and cyclic throughput performance when competing with CUBIC through a bottleneck. Further, we identify the cause for these issues: inaccurate RTT and bottleneck estimations leading to problematic CWND settings. We provide potential heuristics to improve BBR's estimation and CWND. We propose a feedback loop to control BBR's CWND to dynamically adjust to network conditions rather than the current 2 BDP cap.

6.2 Future Work

Future work could implement and test our proposed changes to BBR. Additionally, as BBR gets wider adoption, BBR's behavior over a wider set of conditions such as 4G and satellite connections could be evaluated. Specifically, our hardware testbed could be expanded by adding more Raspberry Pi's. This addition would allow us to test more flows at a time without having multiple flows per machine (which may act as a confounding factor in some types of experiments). Additionally, our the Raspberry Pis are gated by a maximum throughput of 225 MB/s, which precludes them from being used to test datacenter-style network conditions. Alternatively, the testbed infrastructure could be reused with traditional Linux PCs to emulate a real datacenter (albeit at a higher cost), which would allow our infrastructure to be used to test more mission critical types of scenarios.

Our experiments focused on fairly reliable network conditions - low loss, stable capacities, and stable round trip times. These conditions are not necessarily typical of the real world, so it could be useful to expand our work to include higher variance capacities, loss rates, and round trip times to emulate wireless (4G, LTE, 5G) or satellite environments. These expanded conditions could also include simulations of moving devices such as phones or connected cars, similar to recent work studying BBR on high speed trains [26]. Moving endpoints will force handoffs between middleboxes, and thus incur periodic increases in round trip time, and it is still unclear how BBR, or our proposed adjustments, would handle these situations.

Finally, our testbed framework and analysis could be used beyond TCP. Recently, HTTP 3.0 proposal was published, and it does *not* propose to use TCP as the underlying transport protocol. Rather, it will use the UDP based QUIC protocol. This protocol can be configured to use a modified version of TCP CUBIC and TCP BBR, but it also includes many other minor changes. If this QUIC protocol is to receive wide adoption congestion control research approaches such as ours can be used to better understand QUIC. This research is especially important as preliminary work indicates that current TCP-based approaches are *not* always effective over QUIC [5] despite QUIC relying on many mechanisms similar to TCP. While companies have released 'web-scale' experimental validation of QUIC [20], our testbed infrastructure could be useful in provided a controlled environment for testing these new protocols in specific network conditions.

Appendices

1 BBR competing with BBR

Each of the figures shown below show the results of similar tests run with different configurations. Each of these graphs follow the same format as Figure 12 and Figure 13. These results show the all of the conditions not included in Section 5 due to space. For a full list of conditions tested, see Table 1. Together, these results demonstrate that our findings presented in Section 5 are not due to the specific configuration used as the same patterns are visible over all numbers of flows and throughput variations.

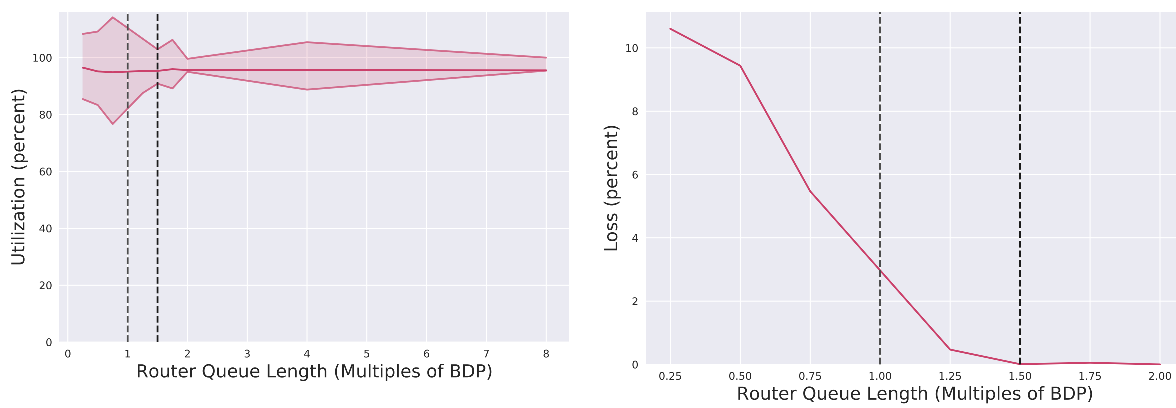


Figure 19: 2 BBR flows at 80 MB/s

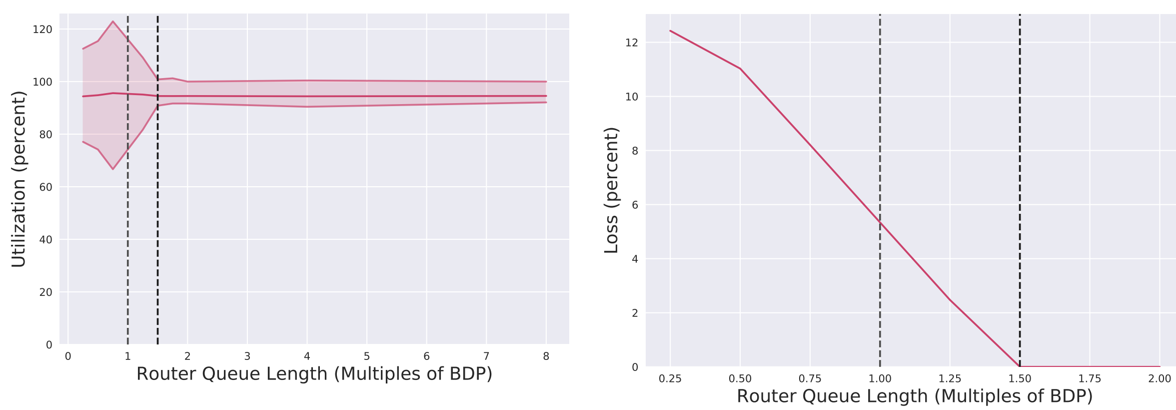


Figure 20: 4 BBR flows flows at 80 MB/s

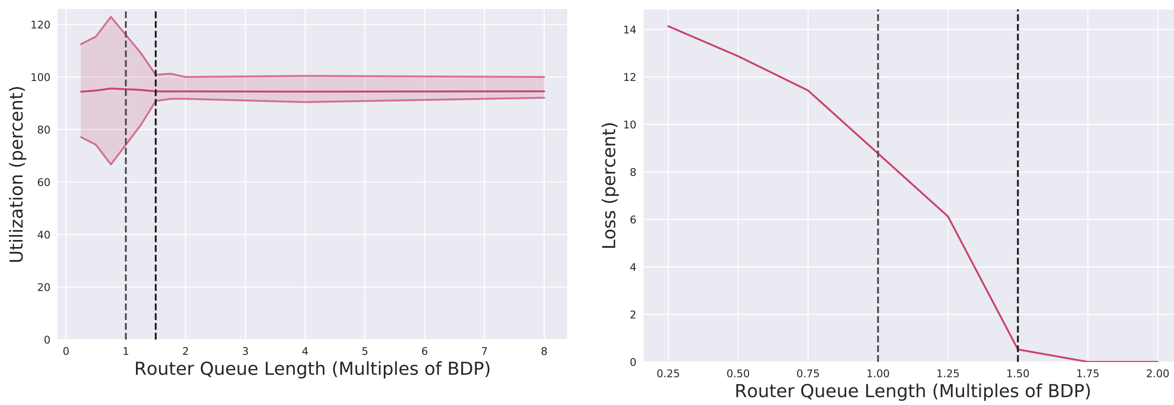


Figure 21: 8 BBR flows flows at 80 MB/s

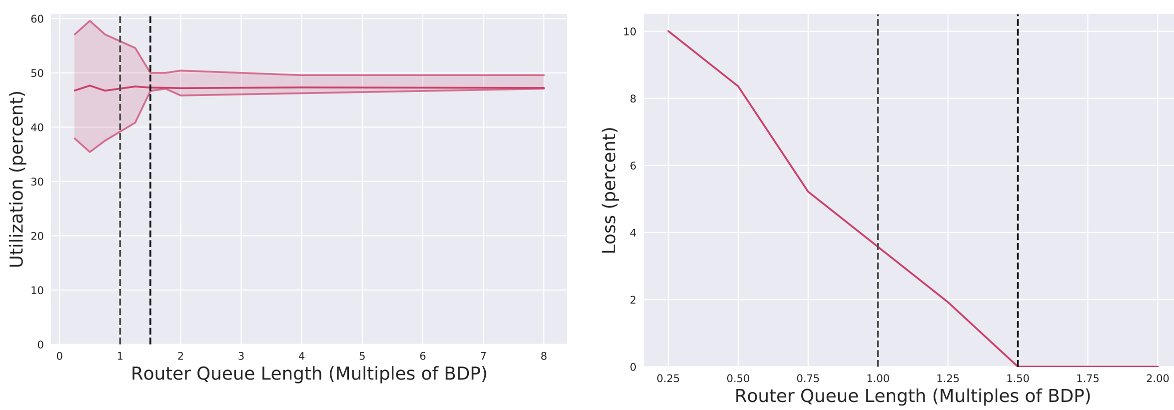


Figure 22: 2 BBR flows at 40 MB/s

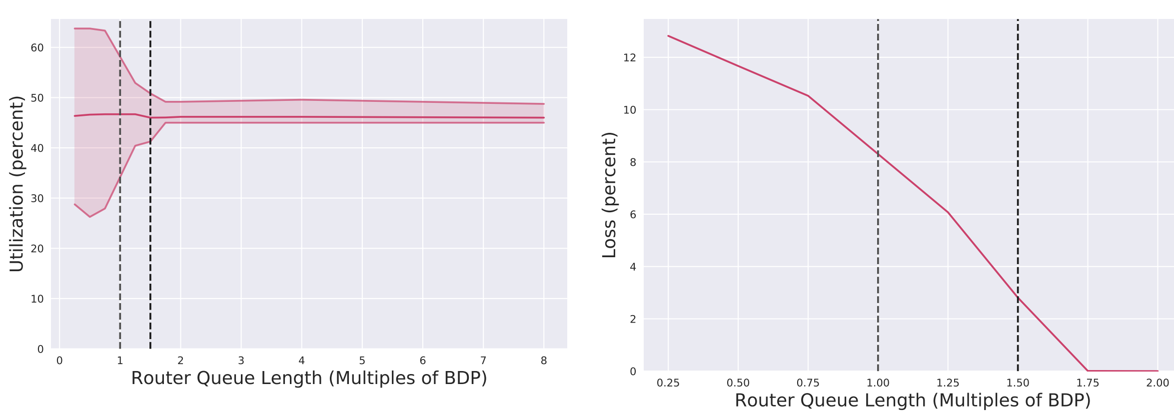
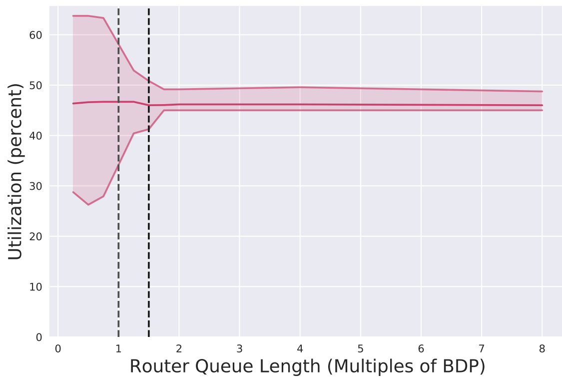


Figure 23: 4 BBR flows at 40 MB/s



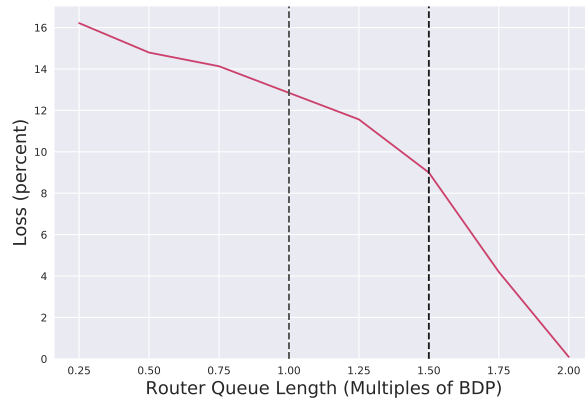
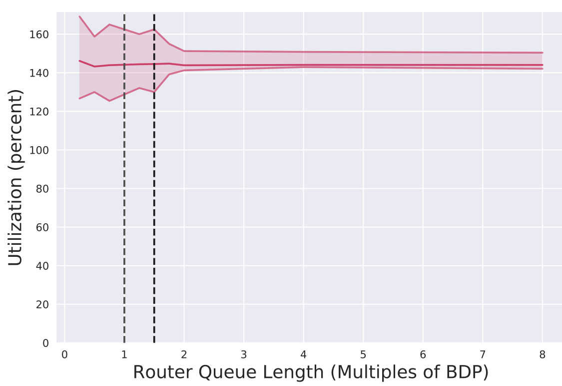


Figure 24: 8 BBR flows at 40 MB/s



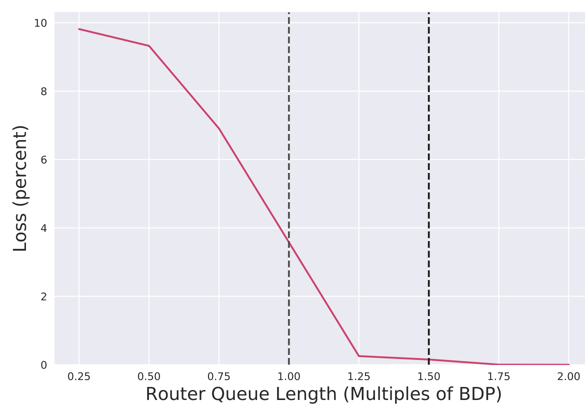
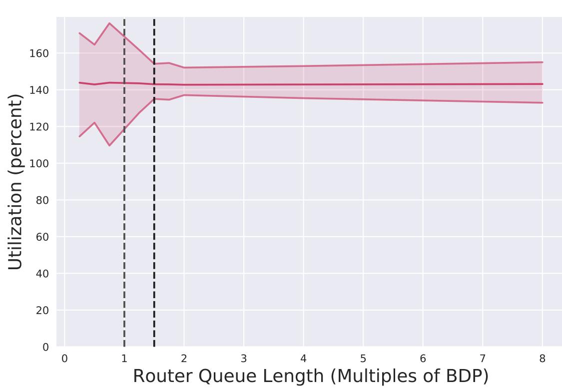


Figure 25: 2 BBR flows at 120 MB/s



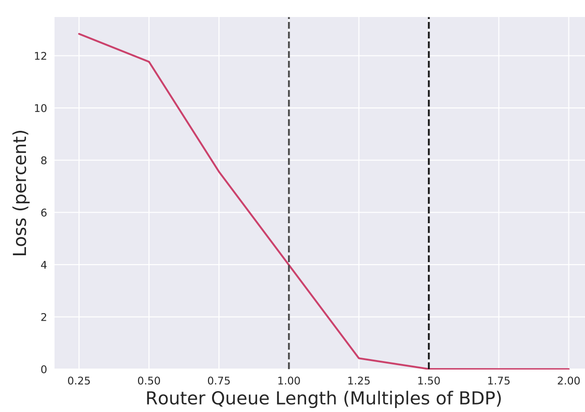


Figure 26: 4 BBR flows at 120 MB/s

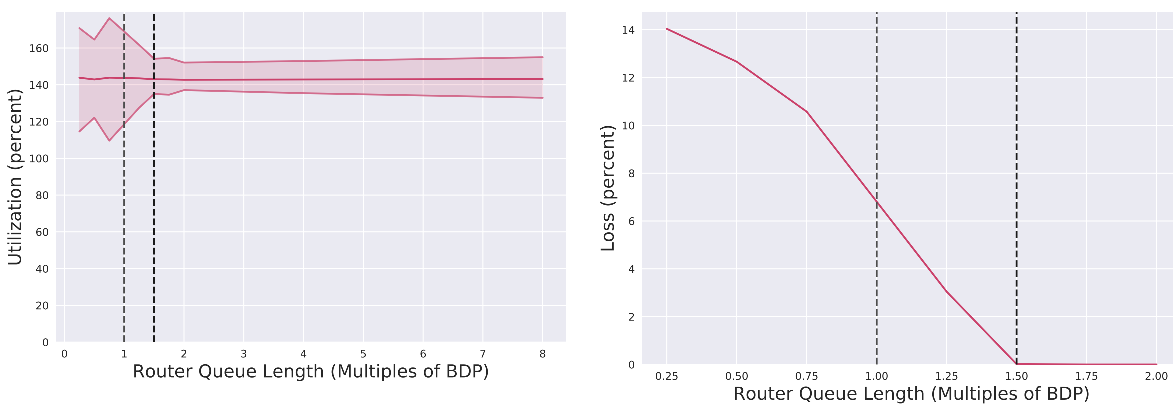


Figure 27: 8 BBR flows at 120 MB/s

2 BBR competing with CUBIC

These figures follow the same format as Appendix 1, but show BBR flows competing with CUBIC flows.

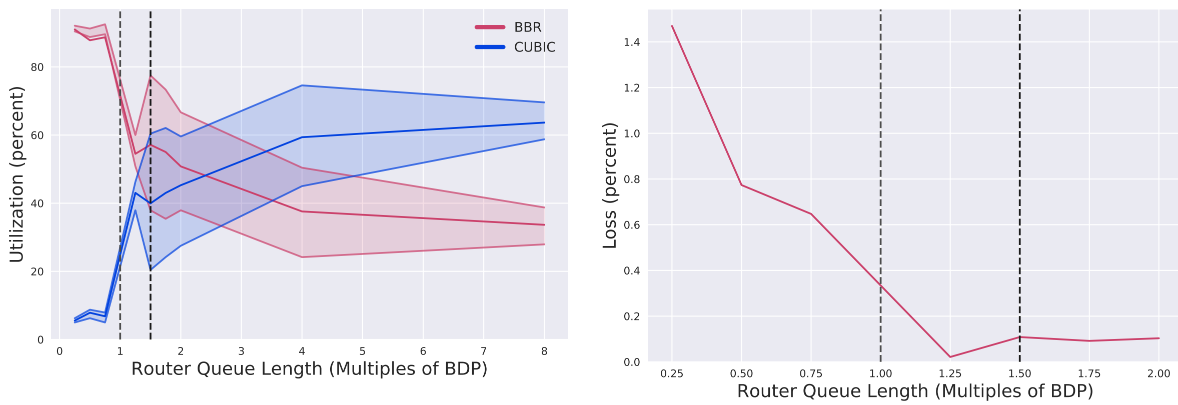


Figure 28: 2 flows, BBR vs CUBIC at 80 MB/s

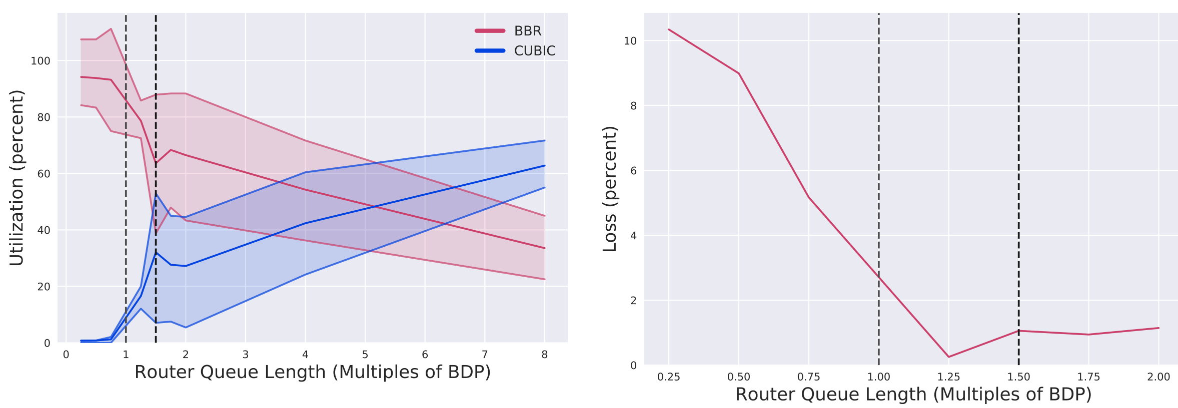


Figure 29: 4 flows, BBR vs CUBIC at 80 MB/s

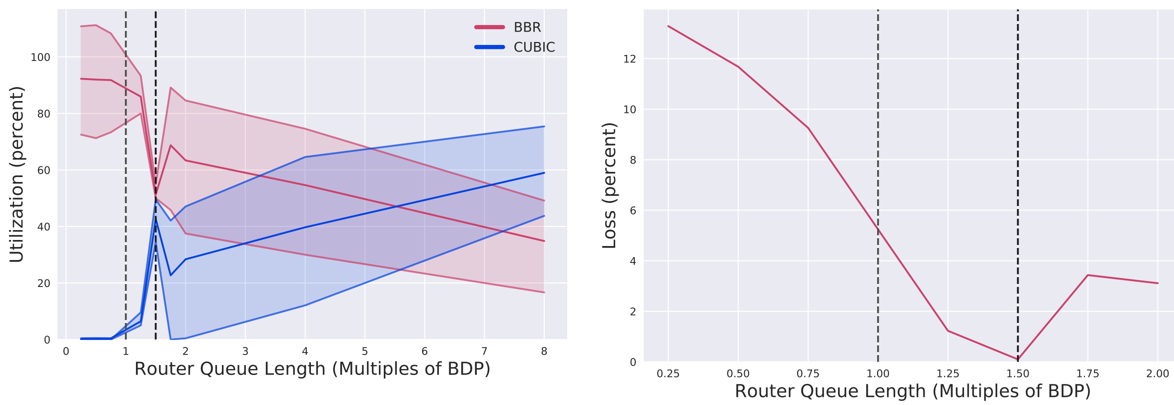


Figure 30: 8 flows, BBR vs CUBIC at 80 MB/s

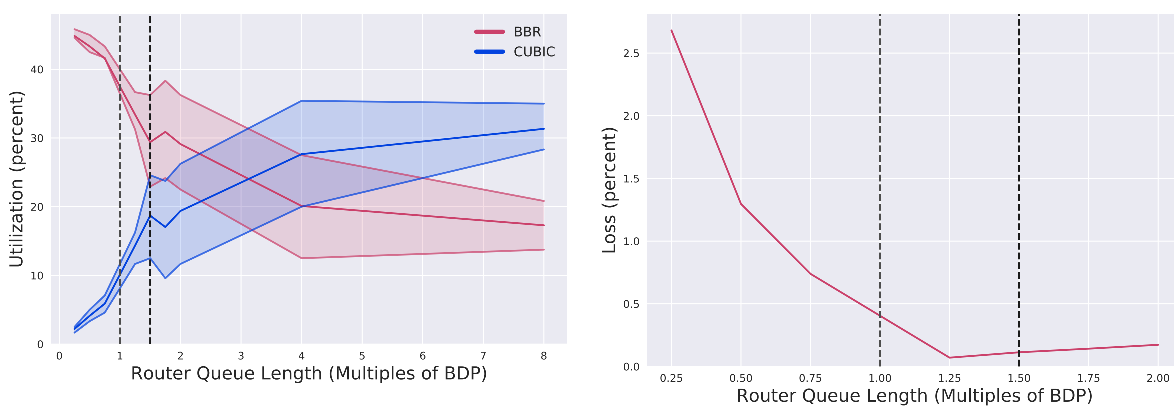


Figure 31: 2 flows, BBR vs CUBIC at 40 MB/s

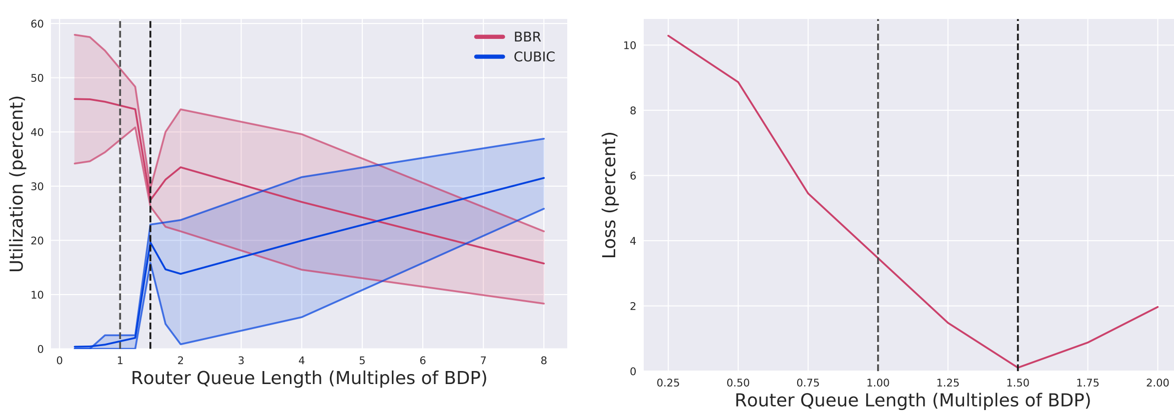


Figure 32: 4 flows, BBR vs CUBIC at 40 MB/s

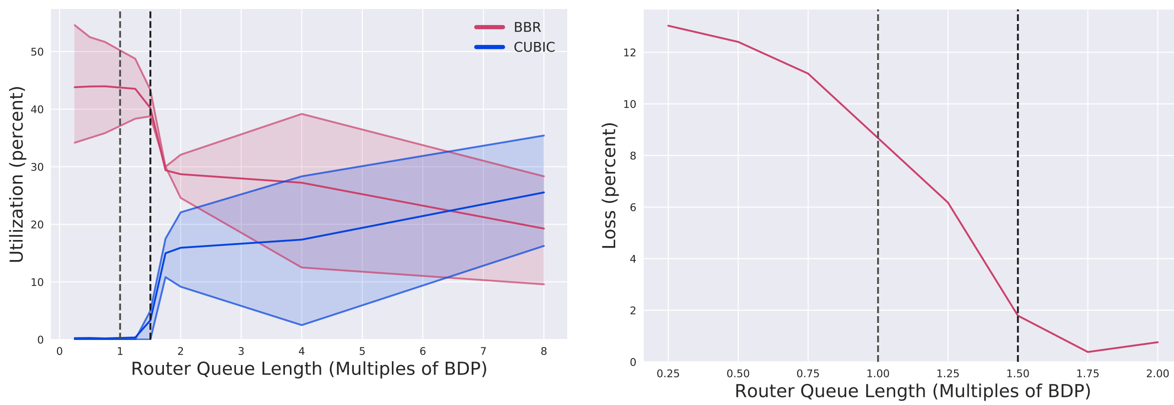


Figure 33: 8 flows, BBR vs CUBIC at 40 MB/s

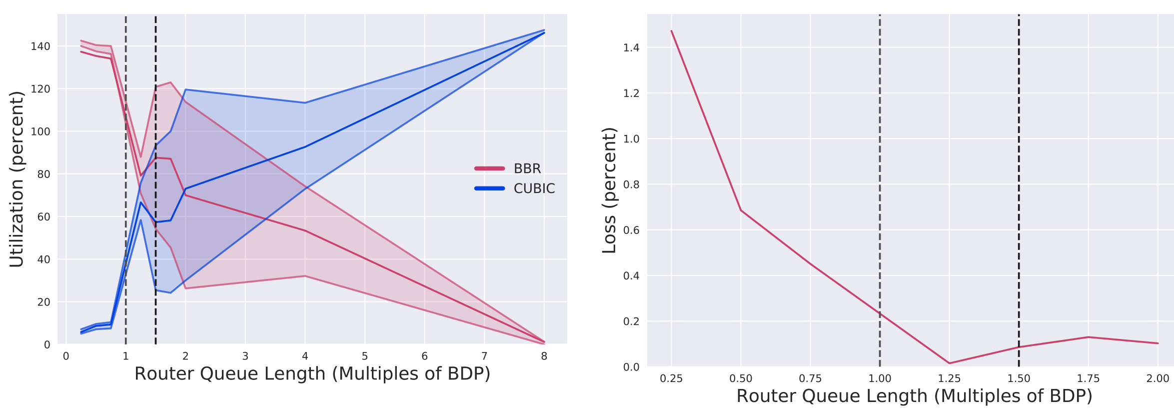


Figure 34: 2 flows, BBR vs CUBIC at 120 MB/s

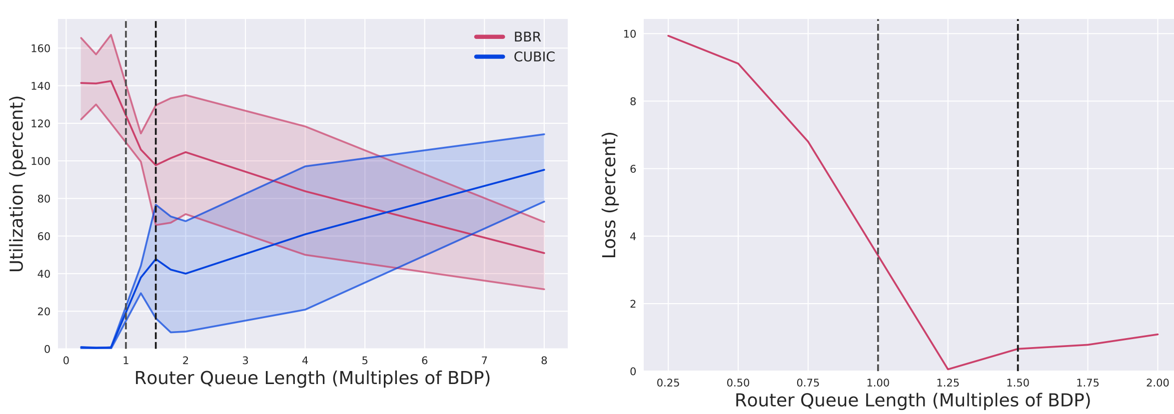


Figure 35: 4 flows, BBR vs CUBIC at 120 MB/s

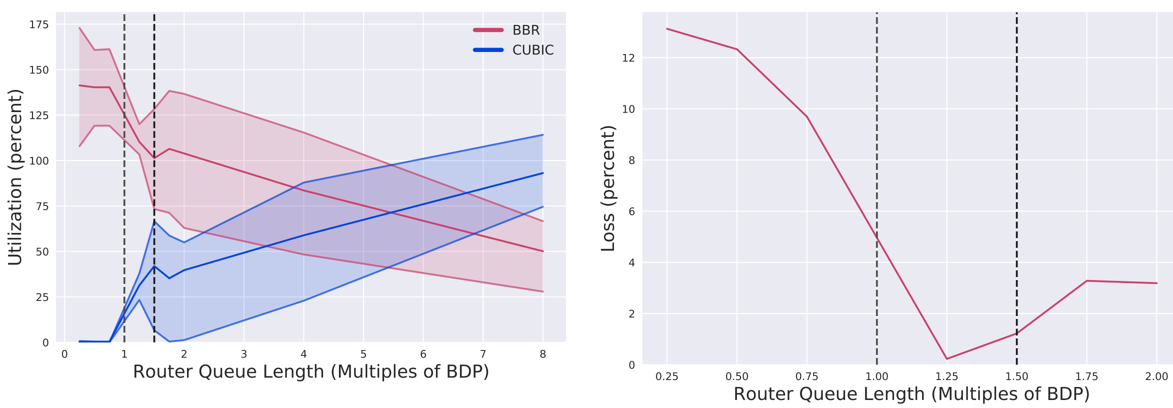


Figure 36: 8 flows, BBR vs CUBIC at 120 MB/s

Bibliography

[1] Cisco Visual Networking Index: Forecast and Trends, 2017–2022.

[2] Dual Token Bucket Algorithms - TechLibrary - Juniper Networks.
Accessed: Feburuary 24, 2019.

[3] tc-tbf(8): Token Bucket Filter - Linux man page.
Accessed: Feburuary 24, 2019.

[4] Venkat Arun and Hari Balakrishnan.
Copa: Practical Delay-Based Congestion Control for the Internet.
In *Proceedings of the Applied Networking Research Workshop on - ANRW '18*, pages 19–19, Montreal, QC, Canada, 2018. ACM Press.

[5] Divyashri Bhat, Amr Rizk, and Michael Zink.
Not so QUIC: A Performance Study of DASH over QUIC.
In *Proceedings of the 27th Workshop on Network and Operating Systems Support for Digital Audio and Video - NOSSDAV'17*, pages 13–18, Taipei, Taiwan, 2017. ACM Press.

[6] L.S. Brakmo and L.L. Peterson.
TCP Vegas: end to end congestion avoidance on a global Internet.
IEEE Journal on Selected Areas in Communications, 13(8):1465–1480, October 1995.

[7] Neal Cardwell, Yuchung Cheng, C. Stephen Gunn, Soheil Hassas YeganehIan Swett, Jana Iyengar, and Victor VasilievVan Jacobson.
Bbr congestion control:ietf 100 update: Bbr in shallow buffers.

[8] Neal Cardwell, Yuchung Cheng, C. Stephen Gunn, Soheil Hassas Yeganeh, and Van Jacobson.
BBR: congestion-based congestion control.
Communications of the ACM, 60(2):58–66, January 2017.

[9] Erik Carlsson and Eirini Kakogianni.
Smoother streaming with BBR, August 2018.
Spotify Labs. Accessed: Feburuary 24, 2019.

[10] Xuan Chen and John Heidemann.
Preferential treatment for short flows to reduce web latency.
Computer Networks, 41(6):779–794, April 2003.

[11] Mo Dong, Qingxi Li, Doron Zarchy, P Brighten Godfrey, and Michael Schapira.
PCC: Re-architecting congestion control for consistent high performance.
In *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*, pages 395–408, 2015.

[12] Jon Dugan.
IPerf - the ultimate speed test tool for TCP, UDP, and SCTP.

[13] Jim Gettys and Kathleen Nichols.
Bufferbloat: dark buffers in the Internet.
Communications of the ACM, 55(1):57, January 2012.

[14] Sangtae Ha, Injong Rhee, and Lisong Xu.
CUBIC: a new TCP-friendly high-speed TCP variant.
ACM SIGOPS Operating Systems Review, 42(5):64–74, July 2008.

[15] Stephen Hemminger et al.
Network emulation with netem.
In *Linux Australia*, pages 18–23, Sydney NSW 2001, Australia, 2005.

[16] Urs Hengartner, Jürg Bolliger, and Thomas Gross.
TCP Vegas revisited.

In *Proceedings IEEE Conference on Computer Communications (INFOCOM)*, volume 3, pages 1546–1555. IEEE, 2000.

[17] Mario Hock, Roland Bless, and Martina Zitterbart.
Experimental evaluation of BBR congestion control.
In *IEEE 25th International Conference on Network Protocols (ICNP)*, pages 1–10, Toronto, ON, October 2017. IEEE.

[18] Leonard Kleinrock.
Power and deterministic rules of thumb for probabilistic problems in computer communications.
In *Proceedings of the International Conference on Communications*, volume 43, pages 1–43, 1979.

[19] Leonard Kleinrock.
Internet congestion control using the power metric: Keep the pipe just full, but no fuller.
Ad Hoc Networks, 80:142–157, November 2018.

[20] Adam Langley, Janardhan Iyengar, Jeff Bailey, Jeremy Dorfman, Jim Roskind, Joanna Kulik, Patrik Westin, Raman Tenneti, Robbie Shade, Ryan Hamilton, Victor Vasiliev, Alistair Riddoch, Wan-Teh Chang, Zhongyi Shi, Alyssa Wilk, Antonio Vicente, Charles Krasic, Dan Zhang, Fan Yang, Fedor Kouranov, and Ian Swett.
The QUIC Transport Protocol: Design and Internet-Scale Deployment.
In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication - SIGCOMM '17*, pages 183–196, Los Angeles, CA, USA, 2017. ACM Press.

[21] Jeffrey M. Jaffe.
Flow control power is nondecentralizable.
Communications, IEEE Transactions on, 29:1301 – 1306, 10 1981.

[22] Kouto Miyazawa, Kanon Sasaki, Naoki Oda, and Saneyasu Yamaguchi.
Cycle and divergence of performance on TCP BBR.

In *IEEE 7th International Conference on Cloud Networking (CloudNet)*, pages 1–6. IEEE, 2018.

[23] Dominik Scholz, Benedikt Jaeger, Lukas Schwaighofer, Daniel Raumer, Fabien Geyer, and Georg Carle.
Towards a deeper understanding of TCP BBR congestion control.
In *Proceedings of IFIP Networking*, Zurich, Switzerland, 2018.

[24] Jacobus Van der Merwe, Subhabrata Sen, and Charles Kalmanek.
Streaming video traffic: Characterization and network impact.
In *Proceedings of the Seventh International Web Content Caching and Distribution Workshop*, Boulder, Colorado, USA, 2002.

[25] Bing Wang, Jim Kurose, Prashant Shenoy, and Don Towsley.
Multimedia streaming via TCP: An analytic performance study.
ACM Transactions on Multimedia Computing, Communications, and Applications (TOMM), 4(2):16, 2008.

[26] Jing Wang, Yufan Zheng, Yunzhe Ni, Chenren Xu, Feng Qian, Wangyang Li, Wantong Jiang, Yihua Cheng, Zhuo Cheng, Yuanjie Li, Xiufeng Xie, Yi Sun, and Zhongfeng Wang.
An Active-Passive Measurement Study of TCP Performance over LTE on High-speed Rails.
arXiv:1812.04823 [cs], December 2018.
arXiv: 1812.04823.