Insights into BBRv3's Performance and Behavior by Experimental Evaluation

1st Roland Bless
Institute of Telematics
Karlsruhe Institute of Technology
Karlsruhe, Germany
ORCID: 0000-0002-1651-1548

2nd Lukas Lihotzki
Student at Institute of Telematics
Karlsruhe Institute of Technology
Karlsruhe, Germany

3rd Martina Zitterbart
Institute of Telematics
Karlsruhe Institute of Technology
Karlsruhe, Germany
ORCID: 0000-0003-0088-6289

***Abstract*—This paper investigates the performance of BBRv3. BBRv3 aims to achieve high throughput with low latency and low packet loss rates. We evaluate BBRv3's performance in a Linux-based testbed and also identify and explain causes of the observed behavior. In contrast to existing work, we investigate BBRv3's behavior at bottleneck data rates of 100 Mbit/s–10 Gbit/s and delve deeper into the issue of self-induced queuing delay. We also study the impact of delay jitter on performance. Moreover, we investigate the performance of various concurrent short flows taken from a real-world traffic trace. BBRv3 does not really achieve its low delay goal: while it can limit queuing delay in large buffers, it regularly adds 0.95 RTTmin queuing delay in a single flow scenario and more than 1 RTTmin over 50% of the time with multiple BBR flows (RTTmin: round-trip time without queuing delay). We also observe fairness problems and slow convergence that already existed with BBRv2. BBRv3 flows are not strongly susceptible to delay jitter, although it adversely affects queuing delay and fairness behavior to a certain degree (but not starvation). In our experiments with short flows from a real-world traffic trace, BBRv3 performs only slightly, but consistently better than CUBIC.**

***Index Terms*—Congestion control, BBRv3**

I. Introduction

Distributed Congestion Control (CC) has been an active area of research and development since its introduction in the 1980s. One driving factor for this continuity is that application demands have changed over the years making short round-trip times and low end-to-end delay crucial for application performance, especially for interactive and transactional web traffic. Traditional loss-based congestion control inherently creates self-induced queuing delay that can lead to the Bufferbloat problem [1] if large buffers are present at bottlenecks. Therefore, besides using Active Queue Management (AQM) mechanisms in the network, employing an improved CC is a further option to control queuing delay. However, achieving *high data rates* across a wide range of data rates while keeping *queuing delay low* and achieving *fairness* (typically flow rate fairness) is a very challenging task for congestion controls.

Introduced by Google in 2016, *Bottleneck-Bandwidth and Round-Trip Propagation Time (BBR)* congestion control strives to improve the performance of data transfer in the Internet. Its main goal is to improve performance compared to current loss-based congestion controls such as CUBIC TCP [2]. BBR aims at high data rates in the presence of occasional packet losses that are not caused by persistent congestion while keeping the *queuing delay low*. The third version, BBRv3, seeks to improve performance over the first two versions, which are now considered obsolete [3]. It was recently adopted by IETF's CCWG [4] as a CC proposal and may well replace CUBIC as the new Internet-wide CC standard.

ISBN 978-3-903176-72-0 © 2025 IFIP

Therefore, several groups have recently examined the performance of BBRv3 [5]–[8]. However, most of them focus on fairness issues and have not evaluated self-induced queuing delay, except in a simulation [8] or for a single flow at 20 ms round-trip time (RTT) [5]. The BBR specification [4] lists achieving *low delay* as one of its major goals besides achieving high throughput and low packet loss rates. Although modern CCs for the Internet are expected to work over a broad range of transmission data rates (up to hundreds of Gbit/s), none of the above evaluations has used real Ethernet bottleneck links at data rates above 1 Gbit/s. Similarly, no other evaluations have investigated the impact of delay jitter on BBRv3's performance yet, although various jitter sources in the Internet exist and [9], [10] suggest severe impacts for BBR.

In this paper we evaluate BBRv3's behavior in a dedicated testbed and systematically study various influences on its performance. Moreover, we *identify and explain the causes* of the behavior shown. The contributions of this paper include:

- First extensive investigation of BBRv3's self-induced queuing delay and explanation of its root causes (backed by traces of BBR's internal variables). One of BBR's major goals was to achieve low queuing delay and we show that it systematically adds queuing delay in the order of an RTT, even with a single flow (which BBRv1 did not).
- First investigation of the influence of delay-jitter on BBRv3's performance and fairness since research [9], [10] suggested that BBR is strongly susceptible to jitter.
- Comparison of different methods to generate application-limited traffic from real-world traces and using them for evaluation of BBRv3's performance with short flows.
- Investigation of BBRv3's behavior with *bottleneck* link data rates ranging from 100 Mbit/s to 10 Gbit/s, whereas other research mainly uses 100 Mbit/s or 1 Gbit/s at most.

- The use of real hardware with Google's reference Linux implementation, i.e., no virtualization, no simulation, or emulation. The experiments were repeated 30 times (except for tests with real-world traffic) in a thoroughly validated testbed.

Out of scope were evaluations using AQM and/or Explicit Congestion Notification as well as experiments across the Internet or with wireless links.

II. BBRv3 in a Nutshell

In this section we give a brief overview of BBRv3. For more details, the reader should refer to [4]. BBRv3 inherits its core mechanisms from BBRv1 [11] and BBRv2. It can be basically characterized as a rate-based CC that also controls the amount of in-flight data according to an internal model of the network path characteristics. A BBR sender uses paced sending at the target rate that is calculated by the model. BBR's core feature is to let a flow sender estimate the Bandwidth Delay Product (BDP), that is $b_r \cdot RTT_{min}$, where b_r is the current bottleneck bandwidth share and RTT_{min} the round-trip time (RTT) without any queuing delay. BBR uses a maximum filter of the delivery rate and RTT measurements to estimate the BDP. A BBR flow is in one of the states *StartUp*, *Drain*, *ProbeBW*, or *ProbeRTT* (see Fig. 1). Measurements for RTT_{min} are performed in the *ProbeRTT* state where BBR senders reduce their in-flight data to 0.5 BDP to remove the bottleneck queue. *ProbeRTT* starts if the RTT_{min} estimate has not been updated for more than 5 s. This condition automatically leads to a (self-)synchronized behavior of all BBR flows at the bottleneck (at most 10 s after an individual flow started). The so synchronized flows simultaneously execute *ProbeRTT* every 5 s thereby increasing the probability to actually emptying the queue during this phase (i.e., a period in a certain state). BBR leaves *ProbeRTT* after an RTT (or at least 200 ms) and enters *ProbeBW* if it estimates that the bottleneck link capacity has already been saturated or *StartUp* otherwise.

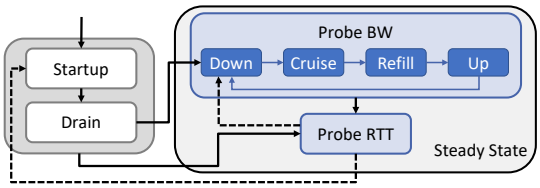


Fig. 1: BBR State Transition Diagram

A. Startup and Drain

A flow begins in *StartUp* and performs an exponential search for the available capacity: it uses a *pacing_gain* of 2.77 and *cwnd_gain* of 2. This means that its paced sending rate is 2.77 times higher than the current target rate and that the allowed congestion window is doubled during startup. *StartUp* is exited if one of two conditions is fulfilled: it has found a "bandwidth plateau" or detected certain packet loss indicators. A plateau is found if a less than 25% increase of the delivery rate is detected for three round trips without application limits. During *StartUp* a BBR sender may fill the bottleneck buffer in case the bottleneck link capacity has been saturated in the meantime. In order to quickly drain this excess data from the bottleneck queue a BBR sender transitions to *Drain* that uses a *pacing_gain* of 0.35 to drain the queue in less than one round trip. *Drain* is left when the amount of in-flight data is less than or equal to the estimated BDP.

B. Steady State – ProbeBW

Long-lived flows spend most of their time in *ProbeBW*. It has four sub-states: *DOWN*, *CRUISE*, *REFILL*, and *UP* (we omit their name prefix *ProbeBW_* for brevity). BBR cycles through these sub-states to decrease, keep, or increase its sending rate in order to either make room, to find a stable and fair operating point, or to detect free capacity. Figure 2 shows an example of a BBR flow in different phases over time.

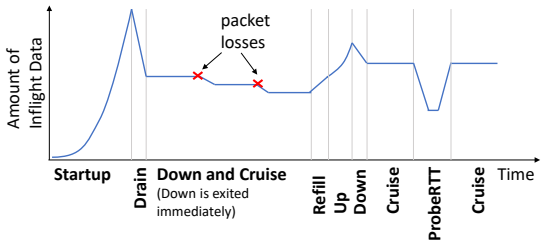


Fig. 2: Example of state transitions in different phases

UP probes for free capacity by setting the *pacing_gain* to 1.25 and the *cwnd_gain* to 2.25 (i.e., the allowed congestion window is *cwnd_gain* · estimated BDP). BBRv3 reacts to packet loss $\geq$2% or ECN as congestion signals. Both signals set an internal variable *inflight_longterm* that denotes a long-term maximum volume of in-flight data that creates "acceptable queue pressure" [4]. The corresponding *inflight_shortterm* value is considered as safe lower bound. If *inflight_longterm* has not been set, the volume of in-flight data increases to 1.25 BDP, thereby risking to cause queuing delay or packet loss. Otherwise, BBR increases *inflight_longterm* by adding an amount that is doubled each round trip. So, it starts slowly to increase the limit but then grows rapidly to be able to quickly utilize newly available bandwidth in networks with large BDPs. *UP* is exited if it determines that the flow has saturated the available bandwidth and reached a bandwidth plateau or that the packet loss within the last round trip has exceeded the 2% limit. BBR then enters the *DOWN* phase to reduce the queued data.

DOWN pursues a deceleration tactic using a *pacing_gain* of 0.9. *DOWN* is exited when there is "free headroom" or the volume of in-flight data is less than or equal to an estimated BDP. The headroom condition tries to leave free capacity for other flows (e.g., those newly starting or those trying to increase their share) by stopping at $0.85 \cdot$*inflight_longterm*, if the latter is set.

In *CRUISE*, BBR uses a *pacing_gain* of 1.0 to send data at the same rate the network is delivering data. In this state, it responds more sensitively to packet loss events (not only

beyond 2%), because the available bandwidth may have been reduced. Figure 2 shows an example of BBR's reaction to loss events during *CRUISE* as indicated by the two red cross marks. BBR reacts by adapting its bandwidth and *inflight_shortterm* estimates, e.g., by reducing them to 0.7 of their previous values.

BBR transitions from *CRUISE* to *REFILL* to begin probing for bandwidth after some calculated period that also considers the potential coexistence of CUBIC and Reno flows. In *REFILL*, BBR tries to fully utilize the bottleneck link without creating "queue pressure" by sending with *pacing_gain* = 1.0 for a round trip, after which it transitions to *UP*.

Major changes in BBRv3 addressed bandwidth and fairness convergence issues [3]. For example, the changes concerned exiting *UP* too early when limited by *inflight_longterm* as well as tuning parameter values for *cwnd_gain* in *UP*, *DOWN*, and for the Startup phase.

III. Testbed Setup and Measurement Methodology

In order to investigate the various performance aspects of BBRv3, especially at higher data rates beyond 1 Gbit/s bottleneck capacity, we set up a dedicated testbed (see Fig. 3). It allows to create different scenarios with varied RTTs and bottleneck link data rates in a controlled environment.

It consists of four Linux servers. Two servers are used as senders, one is used as (software-based) router and one as receiver. A software-based router allows one to flexibly create bottleneck link rate limits, artificial delay, different delay jitter patterns, and packet loss patterns. Moreover, it allows for detailed measurements of the bottleneck queue length. For data rates up to 1 Gbit/s, we use dedicated Ethernet links to a hardware switch (MikroTik CRS354-48G-4S+2Q+, not shown in Fig. 3) with VLANs to flexibly interconnect the servers. For higher data rates, dedicated and directly connected cables with 10 Gbit/s were used because there was no hardware switch. We carefully validated the testbed so that neither the hardware switch nor the router caused any side effects that noticeably impact the measurement results (more details below).

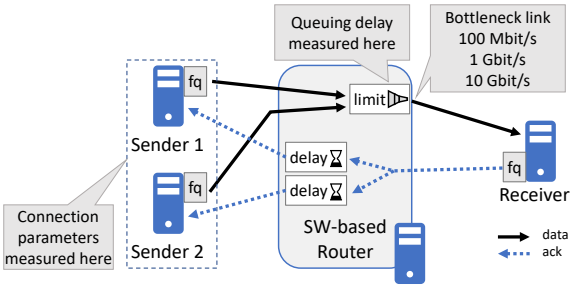


Fig. 3: Testbed setup

A. Used Hardware

Each server has two Intel Xeon Silver 4110 CPUs (2.1 GHz, 8 cores, 16 threads) and 96 GiB of RAM installed, split at half to every CPU's NUMA node. The servers have a dedicated management network interface card (NIC) on board and possess an Intel I350 NIC with four ports used for experiments with up to 1 Gbit/s as well as an Intel X550T NIC with two ports used for experiments at 10 Gbit/s. The bottleneck router has two X550T NICs installed to support the three required 10 Gbit/s links. Ethernet Flow Control was disabled.

B. Used Software

The servers run Ubuntu Server 22.04.4 LTS. The BBRv3 kernel[1] at commit `7542cc7` (kernel 6.4.0) is installed on the end systems. Segmentation offloading is enabled on the end systems. The maximum read buffer size is set to 500 MBytes and the maximum write buffer size to 1000 MBytes. The RX ring size at the receiver was increased from its default value of 512 to 32768 to prevent packet loss at data rates of 10 Gbit/s. Furthermore, we used the `fq` queuing discipline at the end systems as indicated in Fig. 3 to support the more efficient pacing required by BBR. Without `fq`, BBR employs an internal pacing that uses a per-socket high resolution timer. This is less precise and more resource intensive, especially if many flows need to be transmitted.

C. Bottleneck Router Setup

The software-based bottleneck router uses regular Linux Kernel packet forwarding for data rates up to 1 Gbit/s. Up to these speeds, NetEm is used to limit the bottleneck link rate or to create packet loss (see 'limit' box in Fig. 3) and to add artificial delay ('delay' boxes) in the reverse direction for ACK packets. Delaying ACKs is less resource intensive than delaying data packets, thus avoiding adverse side effects at high data rates. For data rates beyond 1 Gbit/s we implemented a simple XDP-based router since we detected occasional (but reproducible) packet losses with standard Linux Kernel forwarding. Since these packet losses would be absent with a properly dimensioned hardware router, we wanted to exclude them as they clearly showed an impact on fairness and convergence to fairness. Queuing and delaying were implemented in user space with AF_XDP, because XDP alone cannot provide these functions. In addition, interrupts were pinned to a single NUMA node and to dedicated cores to improve throughput and minimize jitter. We also disabled CPU frequency scaling to prevent noise in the effective RTT observed at 10 Gbit/s. Moreover, receive side scaling, GRO (Generic Receive Offload), and Adaptive Interrupt Coalescence were disabled to improve performance at high data rates and let it more behave like a hardware router.

D. Measurement Methodology

To measure the state of individual TCP connections, TCP-Log[2] is used at the senders (at the default measurement rate of 10 Hz). TCPLog tracks the goodput, the RTT, the congestion window, and amount of data transferred over time. It uses the tcpinfo socket diagnostics feature to expose TCP information to user space over a netlink socket. We extended TCPLog to also report the internal variables of BBR. The evaluation uses CPUnetLOG [12] on all servers to measure

[1] https://github.com/google/bbr/tree/v3

[2] https://gitlab.kit.edu/kit/tm/telematics/congestion-control/logging/tcplog

the throughput on each NIC used in the experiment and CPU usage to identify processing bottlenecks (measurement rate 10 Hz). Using CPUnetLOG at the receiver is important to rule out any processing bottlenecks on this end caused by CPU load. This is especially important for short flow experiments, where both senders have to handle a potentially large number of open connections (see Section VI).

To measure the queuing delay, the amount of data and the number of frames in the bottleneck queue are captured with 10 Hz at the bottleneck router using the output of `tc` (used in batch mode to avoid spawning lots of processes) with a timestamp for each sample. When the XDP router is used, the current buffer occupancy stored in a shared memory segment is sampled at 10 Hz and time-stamped by a separate program. The amount of data is then divided by the actual bottleneck bandwidth to calculate the queuing delay. All experiments were repeated 30 times and lasted 60 s (except for tests with real-world traffic).

IV. Self-induced Queuing Delay

A major goal of BBRv3 is to achieve "low queue pressure", which is defined as *low queuing delay* and low packet loss [4, Sec. 3.1]. BBR's design aims to fill the queue only during startup and the *UP* phases and to converge "with high probability to Kleinrock's optimal operating point" [11]. We investigate how low the queuing delay that BBRv3 creates actually is and we also explain the causes of the observed behavior. BBR flows that share the same bottleneck with other delay-sensitive flows (e.g., those from highly interactive applications such as multiplayer online games) may noticeably affect the latter. For example, a BBR flow with an RTT of 100 ms can create queuing delay of this order, disturbing the concurrent flows because they also suffer from this flow's self-induced queuing delay (unless AQM is used).

A. Queuing Delay caused by a Single Flow

In contrast to BBRv1, even a single BBRv3 flow creates a non-negligible Queuing Delay (QD) with higher peaks. There are several causes that contribute to the creation of QD in this case:

1) One cause is the *UP* phase that increases the amount of queued data at the bottleneck over the duration of three round trips in many situations.
2) Another cause is that BBR sometimes overestimates the available bandwidth because its bandwidth estimator is susceptible to measurement noise. This causes a queue to build even during the *CRUISE* phase that should conceptually *not* increase "queue pressure" [4, Sec. 4.3.3.2].

1) 1st Cause – UP Phase: In order to illustrate the first cause, we calculate the maximum QD produced by a single application-unlimited flow (i.e., the application always has a packet to transmit) after starting. BBR's pacing gain during *UP* is 1.25. The *UP* phase lasts three round trips if no loss signals occur. Assuming that BBR measures the bandwidth without error, the resulting QD corresponds to a quarter of the duration of the *UP* phase. When calculating the total duration of these three round trips, one must consider that the duration of a round trip is extended by QD produced in previous *UP* round trips. Therefore, the QD (in RTT_{min}) generated in all three round trips is calculated as: $(1.25)^3 - 1 \approx 0.95$. The QD takes $0.95 RTT_{min}/0.25 = 3.8\ RTT_{min}$ to build up. Then, this queue is drained in the *DOWN* phase using a pacing gain of 0.90. Therefore, the QD disappears after $0.95\ RTT_{min}/0.10 = 9.5\ RTT_{min}$.

Figure 4 clearly shows the effect at 50 ms RTT_{min} and the rate limit of 100 Mbit/s with the corresponding delay peaks (*REFILL* and *UP* phases are highlighted by vertical lines). In contrast, BBRv1 caused only a maximum QD of $0.25 RTT_{min}$ (indicated by the red horizontal line) in its *ProbeBW* phase for a short period of time ($< RTT_{min}$) for a single flow.

In Fig. 5 one can see the differences between BBRv3 (dashed lines) and BBRv1 (solid lines) quite clearly (for RTT_{min}=50 ms and different bottleneck bandwidths): although in 70–80% of the cases the QD is similar between BBRv3 and BBRv1, BBRv3 achieves much higher peak QD values whereas BBRv1 is limited to around 0.25–0.3RTT_{min}.

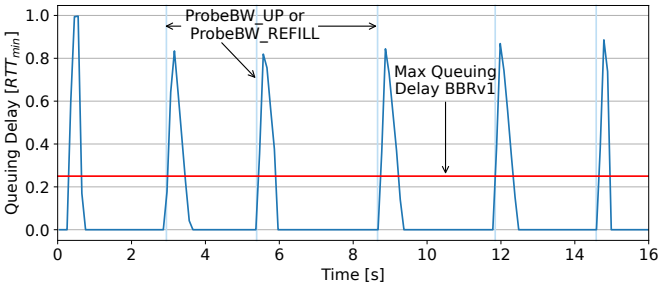


Fig. 4: Queuing Delay caused by a single flow in its *UP* phase (100 Mbit/s, RTT_{min} =50 ms)

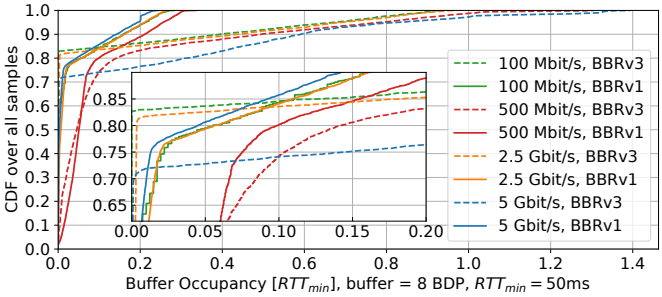
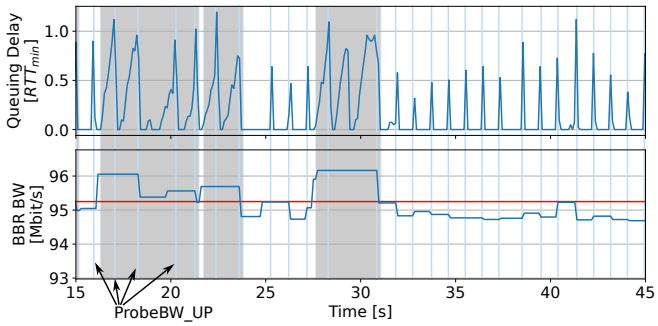


Fig. 5: Queuing Delay CDF of BBRv3 compared to BBRv1 (RTT_{min} =50 ms)

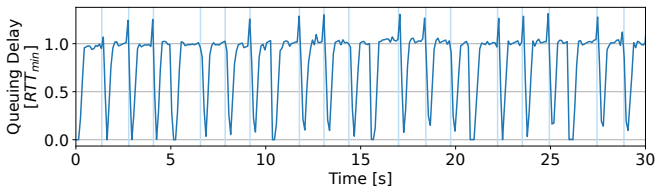
2) 2nd Cause – Bandwidth Overestimation: BBR sometimes overestimates bandwidth due to jitter, introduced by the bottleneck or end systems, which causes noise in the delivery rate samples. The maximum filter used by BBR's bandwidth estimator is sensitive to jitter, overestimating bandwidth in this case. For previous versions of BBR, this problem is discussed in the context of ACK aggregation rather than jitter [13]. BBR uses

a fixed margin of 1% to calculate the target pacing rate from the estimated bandwidth. When the bandwidth is overestimated by more than 1%, the pacing rate is higher than the bottleneck bandwidth in the *CRUISE* phase, so the bottleneck queue builds up during this phase. When the pacing rate is just a little above the actual bottleneck bandwidth, the queue builds up slowly until the pacing gain is reduced by the *ProbeRTT* or *DOWN* phase. Otherwise, the build-up of the queue is stopped by the congestion window.

This effect is illustrated by plotting BBR's bandwidth estimate (BBR BW) and the QD of a single run in Fig. 6a. The effect is well visible during certain time spans that are highlighted in light gray (where BBR BW is above the red horizontal line). Within these spans, the QD starts to build up in *CRUISE*, i.e., *even before* the start of *UP* shown as a vertical line. This effect is stronger when the RTT is low because there are more delivery rate samples in this case. With 1 Gbit/s, this effect *permanently* produces a QD during *CRUISE* as shown in Fig. 6b.



(a) 100 Mbit/s, RTT_{min} =50 ms, buffer=8 BDP



(b) 1 Gbit/s, RTT_{min} =10 ms, buffer=8 BDP

Fig. 6: Queuing Delay caused by a single flow due to jitter-induced bandwidth overestimation during *CRUISE*. Results in (a) show that the BBR BW estimator is too high, causing QD to increase even before *UP*, whereas in (b) QD is created permanently by the jitter-sensitive BW estimator.

This effect is easily achieved in practice, e.g., for a single flow with RTT_{min} =10 ms a jitter of 1 ms triggers the overestimation, so that a QD >10 ms occurs for bottleneck bandwidths of 500 Mbit/s and above. The CDF in Fig. 7 confirms this and shows that BBRv3 has basically more difficulties in limiting QD at higher data rates and lower RTTs due to the jitter sensitivity of BBR's bandwidth estimator. The flows with 500 Mbit/s and RTTs of 10 and 20 ms cause consistently higher QD than the flows at 100 Mbit/s and/or 50 ms RTT.

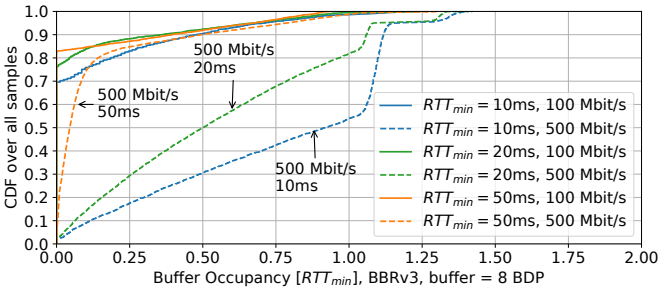


Fig. 7: Queuing Delay CDF of a single BBRv3 flow

Takeaways: BBRv3 can create a significant queuing delay around one RTT_{min} even for a single flow. A flow with smaller RTT and higher bandwidth typically creates higher queuing delay because BBR's bandwidth estimator is sensitive to jitter that occurs more often in these constellations.

B. Queuing Delay caused by Multiple Flows

Multiple application-unlimited BBR flows systematically produce an RTT_{min}-dependent queuing delay that is mainly caused by an overestimation of the available bandwidth. This behavior is inherent and present since BBRv1 [14]. The reason is that each flow outperforms the others in its *UP* phase. Assuming the correct estimates of bandwidth shares, the bottleneck queue will build up when a flow enters *UP*, temporarily decreasing the share of the other flows, because some of their packets will be queued instead of forwarded directly. However, the flow in *UP* can actually increase its bandwidth share by reducing the share of the other flows. As the bandwidth estimate uses a maximum filter, multiple flows estimate their own bandwidth share based on the sample from their respective *UP* phase. Therefore, the sum of these estimates can be larger than the available bottleneck bandwidth. Similarly to experiments with single flows, overestimating the bandwidth leads to the build-up of the bottleneck queue, limited to the BDP-dependent congestion window set by BBR.

The QD measured for two flows at different data rates, RTTs, and buffer sizes is shown in Fig. 8. We chose buffer sizes of 0.8, 1, 2, and 8 BDP. 0.8 BDP is a buffer size that is smaller than the rule-of-thumb size of 1 BDP that is considered optimal for most loss-based CCs. The first version of BBR had problems with size 0.8 BDP [14]. 2 BDP is a size that BBR typically utilizes to some extent but rarely exceeds, while 8 BDP is considered to be a "deep buffer" size that would lead to bufferbloat with loss-based CCs. The boxplot shows the distribution of QD (in RTT_{min}) over time using 30 measurements of 1 min each (each box: central mark = median, bottom and top box edges = 25th and 75th percentiles, respectively; whiskers are at points that lie within 1.5× the inter-quartile range from bottom and top edges).

After the startup phase, the QD does not shift over time, so the samples mostly represent the steady state. With a buffer limit of 8 BDP, BBR generates QD that exceeds $1.5RTT_{min}$ temporarily, irrespectively of the data rate and the RTT. A QD

of $2RTT_{min}$ is only exceeded by outliers. For both 2 BDP and 8 BDP, the median QD exceeds $1.0RTT_{min}$. In contrast, BBR hits the buffer limit regularly when it is set to 0.8 BDP or 1.0 BDP, resulting in packet loss of up to 0.6%. With a data rate of 10 Gbit/s, the receiver's performance seems to be impacted by packet loss, leading to a throughput reduction of up to 1.6% and, in turn, an empty queue for a larger fraction of the time. In configurations where this effect does not occur, the median QD exceeds 0.4 RTT_{min} with 0.8 BDP buffer limit and 0.5 RTT_{min} with 1.0 BDP buffer limit.

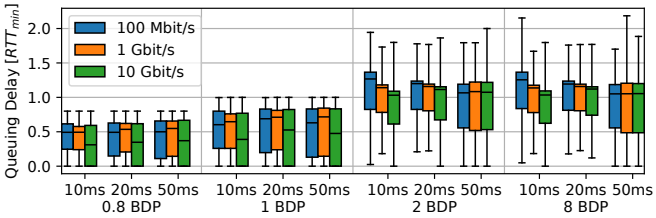


Fig. 8: Queuing Delay (in [RTT_{min}]) caused by two flows at varying data rates, RTTs, and buffer sizes (in [BDP])

The queuing delay when increasing the number of flows is shown in Fig. 9 (buffer size is 8 BDP). These results confirm that QD rises with an increasing number of flows, caused by increased bandwidth overestimation. For 100 Mbit/s and 10 ms RTT_{min}, the QD nearly doubles when increasing the number of flows from 16 to 32, despite no such observable change in the sum of bandwidth estimates, or in the RTT_{min} values. Therefore, this effect is not caused by the BDP estimation.

Instead, it must be caused by one of the other mechanisms that increase the congestion window, *extra acked* or *offload budget* [4, Sec. 4.5.8–9]. Extra acked increases the congestion window to accommodate jitter. The offload budget estimates the minimum volume of data necessary to achieve full throughput when using host offload mechanisms (e.g., TCP Segment Offload, GRO). It is constant during all measurements and accommodates segments that have yet to be reassembled by segmentation offload, for example. The affected configuration has the lowest per-flow BDP, so any constant amount that is added to the congestion window has the highest relative impact. To evaluate both mechanisms separately, this run (100 Mbit/s, 10 ms, 32 flows) was tested with a modified BBR version that disabled extra acked or the offload budget. With disabled offload budget, the median QD is reduced to 3.67 RTT_{min}. Disabling the extra acked mechanism reduced it to 2.73 RTT_{min}.

When some of the flows start with a delay, additional QD is produced because the BDP is overestimated. This effect is shown in Fig. 10 for $RTT_{min} = 10\ ms$ as an example, plotting the QD quartiles of 30 measurements for each point in time. In any case, this effect is caused by the maximum filter that considers the last 5 s. Therefore, the bandwidth estimate of the early flow that backs off lags behind. This is shown for three consecutive runs in Fig. 11. The bottom row shows BBR's bandwidth estimate (BBR BW) for each flow, whereas the top row shows their sum (BBW BW Sum)

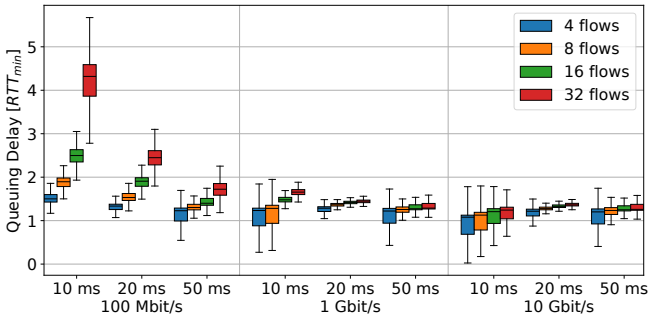


Fig. 9: Queuing Delay with multiple flows (buffer size = 8 BDP)

and the middle row shows the goodput of each flow. Although the goodput of the early flow drops instantly when the late flow starts, its *bandwidth estimate* (BBR BW) is not reduced at that point, leading to an increased sum of estimates (BBR BW Sum). Other reasons may increase the BDP estimate even more. Moreover, late flows do not participate in the first *ProbeRTT* phase. If a late flow enters *UP* instead during the *ProbeRTT* phase of the other flow, it increases its bandwidth estimate considerably without decreasing the bandwidth (and the corresponding estimate) of the early flow. This is visible in the third run of Fig. 11. Furthermore, QD produced by early flows increases the RTT_{min} estimate of late flows until *ProbeRTT*.

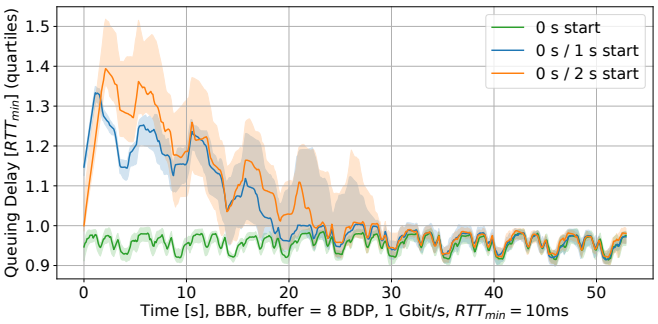


Fig. 10: Queuing Delay without and with delayed start (of 1 s or 2 s), note that the y-axis starts at $0.9RTT_{min}$

Takeaways: With multiple flows at the bottleneck, BBRv3 creates RTT-dependent queuing delay around 1 RTT_{min} in larger buffers ($\geq$ 2 BDP) often for more than 50% of the time. Moreover, the QD increases with an increasing number of flows at the bottleneck. The main reason is the common overestimation of the available bandwidth, especially during the UP phases, but there are further mechanisms that may additionally aggravate the QD.

V. Fairness – The Impact of Jitter

Existing work shows that latecomer unfairness, RTT unfairness, and inter-protocol unfairness also apply to BBRv3 [6]. While these kinds of unfairness showed also up in our measurements we investigate the *impact of jitter* on BBR's

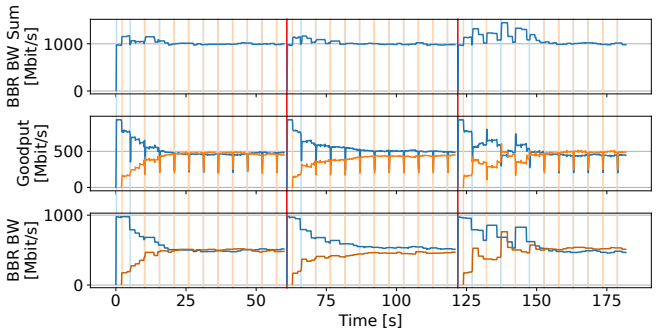


Fig. 11: Queuing Delay caused by bandwidth overestimation with delayed start (2nd flow starts after 2 s), 3 consecutive runs of 60 s

fairness. Jitter occurs naturally in networks at different time scales. There are several sources of non-congestive delay that contribute to perceivable jitter in the range of a few up to tens of milliseconds. Examples for such sources comprise delayed ACKs, ACK aggregation, end-host or in-network scheduling and delays at the physical and MAC layers. For the latter, especially access to a shared medium (e.g., broadband cable or wireless media such as Wi-Fi) causes perceivable jitter.

Claims that BBR is sensitive to jitter, possibly causing total starvation [9] were never experimentally evaluated. In addition, [9] evaluates an older version of BBR, although the mechanisms discussed are still present in the current version. Therefore, the impact of jitter is interesting to evaluate experimentally.

The experiments use two flows with RTT_{min}=10 ms. For one of them, *non-reordering* jitter (always ≥ 0) is added. Both per-packet and slotted jitter are evaluated. In *per-packet jitter*, the delay is chosen independently for each packet. To prevent reordering, packets are at least delayed until all preceding packets are sent. In *slotted jitter*, the delay is chosen at the beginning of a new slot. Packets that arrive when the slot starts or is still used by preceding packets are appended to the slot immediately. The slot ends when it runs out of packets. Slotted jitter is more realistic because it resembles the behavior of media access or ACK aggregation.

Because NetEm cannot reliably handle 10 Gbit/s with multiple senders and our simple XDP router implementation does not support adding artificial jitter, only bandwidths of 100 Mbit/s and 1 Gbit/s were evaluated.

The results are shown in Table I as ratios between the jitter and non-jitter flows. It is observed that the flow without jitter prevails (ratio <0.5) in all configurations with buffer sizes of 0.8 BDP and 1.0 BDP. This is caused by BBR's reaction to packet loss that reduces the congestion window. The congestion window needs to accommodate jitter, so this reaction affects flows that experience more jitter. As expected, this effect is noticeably stronger with higher jitter. The results do not differ clearly between the buffer sizes 0.8 BDP and 1.0 BDP.

With 8 BDP, the results differ by the type of jitter. For per-packet jitter, the flow without jitter prevails. The reason is the underestimation of extra acked, and, in turn, the congestion

TABLE I: Ratio of transferred data (fair = 0.5) in the last 30 s of a BBR flow with the specified jitter to a concurrent BBR flow without jitter (both RTT_{min} =10 ms), median of 30 runs

Buffer Size	slotted jitter						per-packet jitter					
	Link rate						**Link rate**					
	100 Mbit/s			1 Gbit/s			100 Mbit/s			1 Gbit/s		
	Jitter [ms]			**Jitter [ms]**			**Jitter [ms]**			**Jitter [ms]**		
	10	20	40	10	20	40	10	20	40	10	20	40
0.8 BDP	0.43	0.35	0.27	0.32	0.21	0.13	0.44	0.39	0.24	0.41	0.23	0.09
1.0 BDP	0.44	0.36	0.28	0.34	0.24	0.13	0.45	0.38	0.27	0.36	0.22	0.10
2.0 BDP	0.56	0.47	0.36	0.64	0.49	0.32	0.46	0.36	0.26	0.39	0.22	0.12
8.0 BDP	0.71	0.77	0.77	0.68	0.68	0.65	0.46	0.32	0.27	0.39	0.24	0.14

window. With two flows, this leads to occupation of a smaller part of the bottleneck queue, which in turn reduces the bandwidth. The expected extra acked value in MSS depends on the bandwidth reached. For easier reasoning, *BBR Jitter* is calculated as the ratio of BBR's extra acked value in MSS and the smoothed goodput in Mbit/s. The MSS is constant, so the resulting unit of $\frac{MSS}{Mbit/s}$ can be converted to units of ms. As an example, one can look at the configuration with 1 Gbit/s and 20 ms jitter shown in Fig. 12. The estimated jitter only reaches 10 ms, despite of 20 ms per-packet jitter actually added at the bottleneck. Therefore, extra acked underestimation with per-packet jitter is confirmed.

In contrast, a flow with slotted jitter prevails regardless of RTT_{min} or the bottleneck bandwidth, as long as the buffer size is set to 8.0 BDP. For slotted jitter, BBR's jitter estimate matches the configured value (see right part of Fig. 12). Therefore, the congestion window is correctly calculated. The bandwidth gain is caused by an increase in round-trip times due to slotted jitter, leading to the known RTT unfairness. With a buffer size of 2 BDP and slotted jitter, the result depends on the configured jitter. With 10 ms jitter, the flow with jitter prevails, which is the result of bandwidth overestimation. With higher jitter values, the flow without jitter prevails due to an insufficient congestion window. As expected, this effect is stronger for higher jitter values, just as with buffer sizes of 0.8 BDP and 1.0 BDP.

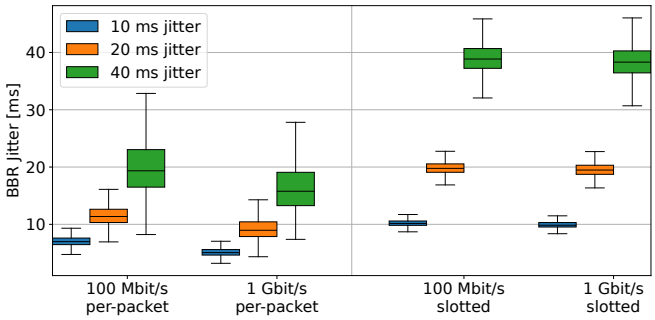


Fig. 12: Jitter estimated by BBR, ten consecutive runs of 60 s each, buffer size = 8 BDP, two flows with RTT_{min}=10 ms, one with artificially added jitter of 10, 20, or 40 ms

Takeaways: The concept of 'non-deterministic jitter' [9] is rather broad and although both jitter models match the

definition of 'non-deterministic jitter', the results between them differed significantly. Jitter impacts BBR's fairness and unfairness increases with higher jitter and higher data rates. Even when considering less realistic artificial jitter models, total starvation as indicated by [9] was not observed. Instead, the resulting impact results from differences in measured RTTs caused by jitter, resulting in RTT-based unfairness.

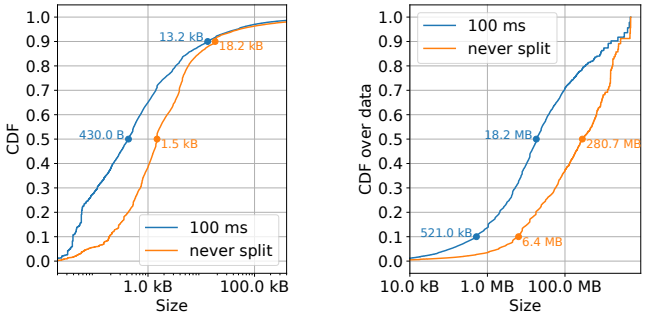
VI. Performance for Short Flows with Real-World Traffic

The evaluations in the previous sections used continuous *application-unlimited* flows. In this case, the sender application sends new data continuously. In contrast, a *short* flow only sends a limited amount of data and then terminates, which is true for most Web traffic flows.

To evaluate CC with short flows, the load generator needs a model to open new flows and send data. The results on CC performance are expected to be more applicable to the real world if this model is closer to real-world flows. To get a diverse set of real-world flows, a packet capture of an Internet backbone router is used: MAWI 2020/06/10 [15] (also used in [16]). We derive three different configurations from this trace as follows. First, the capture is divided into individual unidirectional flows according to the addresses and ports. Second, each flow is optionally split into multiple contiguous *transmission chunks* that are separated by transmission pauses. Our chosen split policy ignores pauses shorter than a threshold, so that a new chunk starts when no data is sent for at least 100 ms. For each transmission chunk, the amount of data transferred is summed up until the next pause threshold. In addition, the pause duration before each chunk is analyzed.

In the *Replay* configuration, the transmission chunks and pauses are replayed as determined by the split policy. In the *Collapsed* configuration, each flow is represented as a single contiguous transmission, skipping pauses between transmission chunks in a single flow. The *Synthetic* configuration recreates the traffic pattern used in previous work by drawing the inter-arrival time and the data amount *independently*, based on the Collapsed configuration [16]. A closer analysis showed that the Synthetic configuration eliminates some burstiness that is preserved in the Collapsed configuration. The resulting chunk size distributions of the MAWI trace with and without splitting (then the size of the single chunk corresponds to the flow size) can be seen in Fig. 13. Although only 10% of the flows are larger than 18.2 kBytes, 90% of the transmitted *data* belongs to flows larger than 6.4 MBytes and 50% of the transmitted data belongs to flows larger than 280.7 MBytes.

As the MAWI packet capture has a duration of 15 minutes, each run of one of those models also takes 15 minutes. Flows from the trace are assigned to both senders in a round-robin fashion and CPUnetLOG showed no CPU limitations. Experiments are carried out with a bottleneck bandwidth of 3 Gbit/s (using NetEm and verified absence of packet loss due to exceeding the buffer), which is significantly higher than needed to transfer the total data amount of the model for the total duration of the run (≈ 1.25 Gbit/s), so it is expected that



(a) Transmission Chunk Size CDF when choosing a random flow. Corresponds to flow size if 'never split'.

(b) Transmission Chunk Size CDF when choosing the flow corresponding to a random transmitted payload byte.

Fig. 13: Cumulative Distribution Function (CDF) of transmission chunk sizes from the MAWI trace [15]

there is enough headroom for transmissions to complete before other transmissions start. Moreover, 3 Gbit/s produces enough congestion to show different results for CC behavior, providing meaningful results. With $RTT_{min} = 10\,ms$, around 170 flows were concurrently active at the maximum. Figure 14 shows the average slowdown values (i.e., normalized flow completion time [16]) to transmit flows from the MAWI trace using BBRv3, CUBIC, and a theoretical SRPT (Shortest Remaining Processing Time) scheduler. The latter knows the flow sizes a priori and has been shown to deliver near-optimal performance [16].

BBR shows consistently slightly lower average slowdown values than CUBIC for all different RTTs and configurations. This results from the shorter queuing delay that BBR produces when the link is congested, compared to CUBIC. This behavior is shown in Fig. 15 where the CDF for QD is shown for BBRv3 and CUBIC. Because only some flows experience congestion, the CDF starts at 0.65 on the Y-axis to show the relevant part. In Fig. 15a an increased slope in the range of 1 BDP to 1.5 BDP is visible, caused by BBR's congestion window limitation of 2 BDP. This effect shows that the BDP estimation works in some cases. However, in the long tail up to the limit of 8 BDP, BBR probably overestimates the BDP, for example, by short-lived flows that have not yet passed a *ProbeRTT* phase. The CDF for CUBIC in Fig. 15b does not show these pronounced differences, i.e., the QD is more evenly distributed during congestion.

Takeaways: For short flows, BBR achieves only a slightly lower average slowdown than CUBIC, but it does so consistently for all different RTTs and configurations. This is the result of BBR's design goal of avoiding completely filling *deep buffers.*

VII. Conclusions

One of BBR's original goals was to reduce queuing delay and to achieve "low delay" is still stated as an objective. Our evaluation showed that a *single BBRv3 flow* creates higher

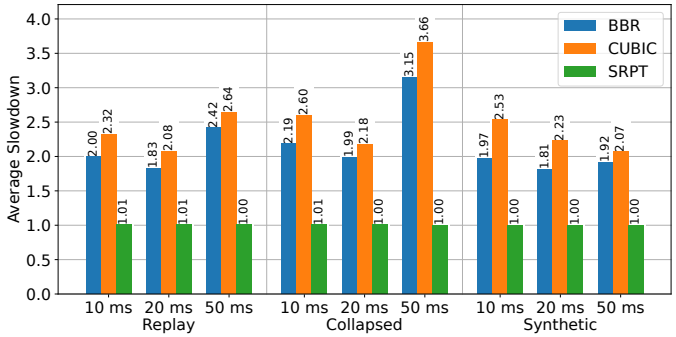
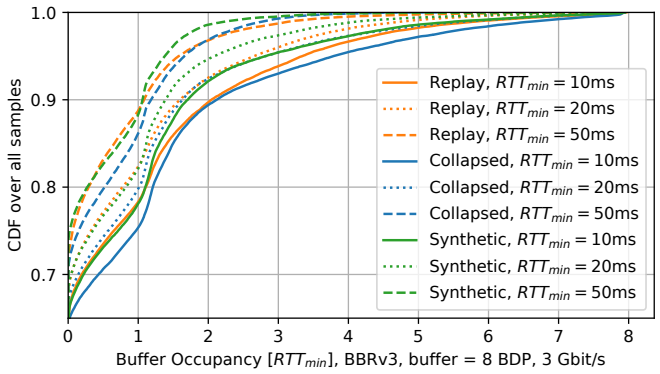
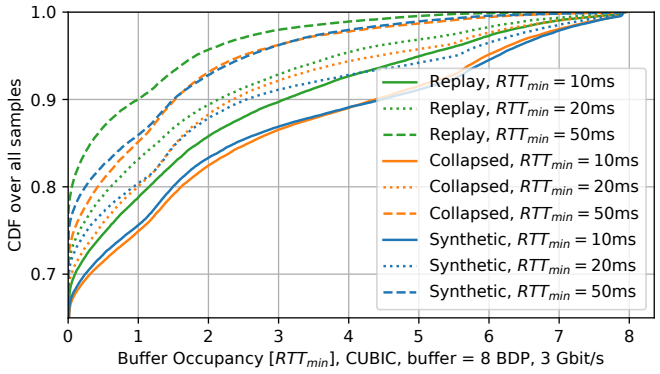


Fig. 14: Average Slowdown per flow using different RTT_{min} values and different traffic creation configurations



(a) BBRv3



(b) CUBIC

Fig. 15: CDF for relative Queuing Delay in RTT_{min}

self-induced queuing delay peaks than BBRv1. We regularly saw a queuing delay of $0.95RTT_{min}$. This results from the *UP* phase and bandwidth overestimation that is exacerbated by jitter in the network. The latter occurs naturally at higher data rates and lower RTTs resulting in persistently perceivable queuing delay.

The inherent problem of BBR of overestimating the bottleneck bandwidth with *multiple flows* due to its maximum filter for estimation of the delivery rate in its *UP* phases still exists in BBRv3. We identified the 'extra acked' and 'offload budget' mechanisms as further contributors to queuing delay. Queuing delay greater than $1\,RTT_{min}$ occurs in more than 50% of the time with two flows for all tested RTT_{min} values. Flows with high RTT_{min} especially negatively affect interactive flows that share the bottleneck.

BBR's fundamental fairness problems still exist in BBRv3: unfairness against CUBIC flows in shallow buffers and unfairness against short RTT BBRv3 flows as well as latecomer disadvantages. Our experiments to investigate the impact of additional jitter showed that BBRv3 is not strongly susceptible to delay jitter as suggested by [9]. However, jitter adversely affects fairness because of an insufficiently large congestion window.

Results for short flows taken from a real-world trace showed that the slowdown for BBRv3 is better than for CUBIC, but RTT unfairness can also be observed in these cases. Basically, BBRv3 works reasonably well across the tested data rate ranges, but there is still room to improve BBR, especially with respect to reducing queuing delay and convergence to fairness.

References

[1] J. Gettys and K. Nichols, "Bufferbloat: Dark Buffers in the Internet," *ACM Queue*, vol. 9, no. 11, pp. 40:40–40:54, Nov. 2011. [Online]. Available: http://doi.acm.org/10.1145/2063166.2071893

[2] L. Xu, S. Ha, I. Rhee, V. Goel, and L. Eggert (Ed.), "CUBIC for Fast and Long-Distance Networks," RFC 9438 (Proposed Standard), RFC Editor, Fremont, CA, USA, Aug. 2023. [Online]. Available: https://www.rfc-editor.org/rfc/rfc9438.txt

[3] N. Cardwell *et al.*, "BBRv3: Algorithm Bug Fixes and Public Internet Deployment," Presentation at IETF 117, IETF, Jul. 2023. [Online]. Available: https://datatracker.ietf.org/meeting/117/proceedings

[4] N. Cardwell, I. Swett, and J. Beshay, "BBR Congestion Control," Feb. 2025, work in Progress. [Online]. Available: https://datatracker.ietf.org/doc/draft-ietf-ccwg-bbr/02/

[5] J. Gomez, E. F. Kfoury, J. Crichigno, and G. Srivastava, "Evaluating TCP BBRv3 performance in wired broadband networks," *Computer Communications*, vol. 222, pp. 198–208, 2024.

[6] D. Zeynali, E. N. Weyulu, S. Fathalli, B. Chandrasekaran, and A. Feldmann, "Promises and Potential of BBRv3," in *Intern. Conf. on Passive and Active Network Measurement*. Springer, 2024, pp. 249–272.

[7] ——, "BBRv3 in the public Internet: a boon or a bane?" in *Proceedings of the 2024 Applied Networking Research Workshop*, 2024, pp. 97–99.

[8] A. Piotrowska, "Performance Evaluation of TCP BBRv3 in Networks with Multiple Round Trip Times." *Applied Sciences (2076-3417)*, vol. 14, no. 12, 2024.

[9] V. Arun, M. Alizadeh, and H. Balakrishnan, "Starvation in end-to-end congestion control," in *Proceedings of the ACM SIGCOMM 2022 Conference*. ACM, 2022, p. 177–192. [Online]. Available: https://doi.org/10.1145/3544216.3544223

[10] V. Arun, M. T. Arashloo, A. Saeed, M. Alizadeh, and H. Balakrishnan, "Toward formally verifying congestion control behavior," in *Proceedings of the 2021 ACM SIGCOMM 2021 Conference*, 2021, pp. 1–16.

[11] N. Cardwell, Y. Cheng, C. S. Gunn, S. H. Yeganeh, and V. Jacobson, "BBR: Congestion-Based Congestion Control," *ACM Queue*, vol. 14, no. 5, pp. 50:20–50:53, Oct. 2016.

[12] Institute of Telematics, "CPUnetLOG," https://gitlab.kit.edu/kit/tm/telematics/congestion-control/logging/cpunetlog, Karlsruhe Institute of Technology, 2024.

[13] B. Su *et al.*, "Rethinking the rate estimation of BBR congestion control," *Electronics Letters*, vol. 56, no. 5, pp. 239–241, 2020.

[14] M. Hock, R. Bless, and M. Zitterbart, "Experimental Evaluation of BBR Congestion Control," in *2017 IEEE 25th International Conference on Network Protocols (ICNP)*, Oct 2017.

[15] M. W. Group, "Packet traces from WIDE backbone, samplepoint G, 2020/06/10," 2020. [Online]. Available: https://mawi.wide.ad.jp/mawi/samplepoint-G/2020/202006101400.html

[16] A. Zapletal and F. Kuipers, "Slowdown as a Metric for Congestion Control Fairness," in *Proceedings of the 22nd ACM Workshop on Hot Topics in Networks*, 2023, pp. 205–212.