Experimental Evaluation of BBR Congestion Control

Mario Hock, Roland Bless, Martina Zitterbart
Karlsruhe Institute of Technology
Karlsruhe, Germany
E-Mail: mario.hock@kit.edu, bless@kit.edu, zitterbart@kit.edu

***Abstract*—BBR is a recently proposed congestion control. Instead of using packet loss as congestion signal, like many currently used congestion controls, it uses an estimate of the available bottleneck link bandwidth to determine its sending rate. BBR tries to provide high link utilization while avoiding to create queues in bottleneck buffers. The original publication of BBR shows that it can deliver superior performance compared to CUBIC TCP in some environments. This paper provides an independent and extensive experimental evaluation of BBR at higher speeds. The experimental setup uses BBR's Linux kernel 4.9 implementation and typical data rates of 10 Gbit/s and 1 Gbit/s at the bottleneck link. The experiments vary the flows' round-trip times, the number of flows, and buffer sizes at the bottleneck. The evaluation considers throughput, queuing delay, packet loss, and fairness. On the one hand, the intended behavior of BBR could be observed with our experiments. On the other hand, some severe inherent issues such as increased queuing delays, unfairness, and massive packet loss were also detected. The paper provides an in-depth discussion of BBR's behavior in different experiment setups.**

I. Introduction

Congestion control protects the Internet from persistent overload situations. Since its invention and first Internet-wide introduction congestion control has evolved a lot [1], but is still a topic of ongoing research [10], [15]. In general, congestion control mechanisms try to determine a suitable amount of data to transmit at a certain point in time in order to utilize the available transmission capacity, but to avoid a persistent overload of the network. The bottleneck link is fully utilized if the amount of *inflight data* $D^{inflight}$ matches exactly the bandwidth delay product $bdp = b_r \cdot RTT_{min}$, where b_r is the available bottleneck data rate (i.e., the smallest data rate along a network path between two TCP end systems) and RTT_{min} is the minimal round-trip time (without any queuing delay). A fundamental difficulty of congestion control is to calculate a suitable amount of inflight data without exact knowledge of the current bdp. Usually, acknowledgments as feedback help to create estimates for the bdp. If $D^{inflight}$ is larger than bdp, the bottleneck is overloaded, and any excess data is filled into a buffer at the bottleneck link or dropped if the buffer capacity is exhausted. If this overload situation persists the bottleneck becomes congested. If $D^{inflight}$ is smaller than bdp, the bottleneck link is not fully utilized and bandwidth is wasted.

Loss-based congestion controls (such as CUBIC TCP [12] or TCP Reno [2]) use packet loss as congestion signal. They tend to completely fill the available buffer capacity at a bottleneck link, since most buffers in network devices still apply a tail drop strategy. A filled buffer implies a *large queuing delay* that adversely affects everyone's performance on the Internet: the inflicted latency is unnecessarily high. This also highly impacts interactive applications (e.g., Voice-over-IP, multiplayer online games), which often have stringent requirements to keep the one way end-to-end delay below 100 ms. Similarly, many transaction-based applications suffer from high latencies.

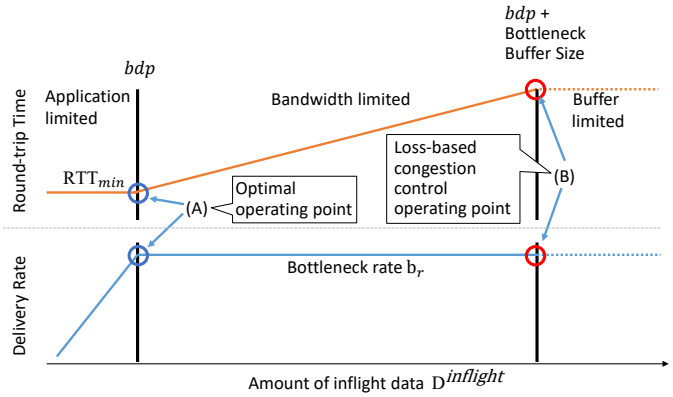


Fig. 1: Congestion control operating points: delivery rate and round-trip time vs. amount of inflight data, based on [5]

Recently, BBR was proposed by a team from Google [5] as new congestion control. It is called "congestion-based" congestion control in contrast to loss-based or delay-based congestion control. The fundamental difference in their mode of operation is illustrated in fig. 1 (from [5]), which shows round-trip time and delivery rate in dependence of $D^{inflight}$ for a single sender at a bottleneck. If the amount of inflight data $D^{inflight}$ is just large enough to fill the available bottleneck link capacity (i.e., $D^{inflight} = bdp$), the bottleneck link is fully utilized and the queuing delay is still zero or close to zero. This is the optimal operating point (A), because the bottleneck link is already fully utilized at this point. If the amount of inflight data is increased any further, the bottleneck buffer gets filled with the excess data. The delivery rate, however, does not increase anymore. The data is not delivered any faster since the bottleneck does not serve packets any faster and the throughput stays the same for the sender: the amount of inflight data is larger, but the round-trip time increases by the corresponding

978-1-5090-6501-1/17/$31.00

amount. Excess data in the buffer is useless for throughput gain and a queuing delay is caused that rises with an increasing amount of inflight data. Loss-based congestion controls shift the point of operation to (B) which implies an unnecessary high end-to-end delay, leading to "bufferbloat" [11] in case the buffer sizes are large. BBR now tries to shift the operating point of congestion control to the left toward (A).

BBR requires sender-side modifications only and is already used in Google's B4 wide-area network as well as at Google.com and at YouTube servers in the Internet. Moreover, the BBR team works toward an "End goal: improve BBR to enable it to be the default congestion control for the Internet." [8]. Therefore, it is of interest to understand better how it behaves in certain environments and situations. This paper provides an independent and extensive experimental evaluation of BBR at higher speeds. Section II provides an overview of the BBR mechanisms and section III explains some of BBR's inherent properties that were observed in the conducted experiments, too. The experimental setup (described in detail in section IV) employs 10 Gbit/s and 1 Gbit/s bottleneck links using the freely available Linux kernel 4.9 implementation. The experiments described in section V vary the flows' round-trip times, the number of flows, and the bottleneck buffer sizes. The evaluation considers achieved throughput, queuing delay, packet losses, and fairness. The latter is investigated in a homogeneous setting (i.e., among BBR flows) as well as in a heterogeneous setting, here among BBR and CUBIC TCP flows. Section VI briefly describes related work and section VII summarizes the results of this evaluation.

II. Overview of BBR

This section provides a brief overview of the main concepts behind BBR, further details (e.g., built-in traffic policing detection) are provided in [5], [7], [9]. The following description assumes a single BBR flow at the bottleneck, the behavior of multiple BBR flows is discussed in section III.

A coarse summary list of BBR features is as follows:

- BBR uses estimates for the available bottleneck data rate b_r and the minimal round-trip time RTT_{min} to calculate a path's available bdp. The estimate for b_r is based on a windowed maximum filter of the delivery rate that the receiver experiences.
- A BBR sender controls its transmission rate s_r with the help of pacing and an estimated data rate b_r, i.e., it is rate-based. It does not use a congestion window or ACK clocking to control the amount of inflight data, but uses an inflight data limit of $2bdp$. [5] mentions delayed and aggregated acknowledgments as reason for choosing $2bdp$ instead of only one bdp.
- BBR probes for more bandwidth by increasing its sending rate s_r to $1.25 s_{r0}$ for an RTT and directly reducing it again to $0.75 s_{r0}$, where s_{r0} is the current estimate of the available data rate. The reduction aims at draining a potential queue that was possibly created by the higher rate.
- BBR uses a special *ProbeRTT* phase that tries to drain the queue completely in order to measure RTT_{min}. Ideally, all BBR flows enter this phase together after some time.
- BBR is neither delay-based nor loss-based and it ignores packet loss as congestion signal. It also does not explicitly react to congestion, whereas congestion window-based approaches often use a multiplicative decrease strategy to reduce $D^{inflight}$.

After this simplified overview, more details are provided along its different operational phases: steady state operation (including *ProbeRTT* to measure RTT_{min} as well as *ProbeBW* to probe for more bandwidth) and startup behavior.

A. Steady State Operation

First, a BBR sender tries to determine the bdp of a network path by getting estimates for b_r and RTT_{min} from measurements (denoted as $\widehat{b_r}$ and $\widehat{RTT}_{min}$, respectively). It uses TCP acknowledgments to calculate the observed *delivery rate* as estimate $\widehat{b_r}$ and also to get an estimate of RTT_{min} (in [5] this is called RTT_{prop} – round-trip propagation time). The delivery rate is calculated by dividing the amount of delivered data (as indicated by acknowledgments) by the period Δt in which the measurement took place. More specifically, $\widehat{b_r} = \max(delivery\_rate_t)\ \forall t \in [T - W_B, T]$, where W_B is typically six to ten RTTs [5] (the Linux kernel v4.9 implementation uses $W_B = 10\,RTTs$). The $delivery\_rate_t$ is updated by every received acknowledgment.

In contrast to other approaches, BBR's sending rate is not determined by the so-called "ACK clocking", but by the estimate for the available bottleneck rate $\widehat{b_r}$. It uses *pacing* for every data packet at this rate in order to control the amount of sent data, thus BBR is not a window-based congestion control. However, it uses a *cap of inflight data* to $2\widehat{bdp}$ (with $\widehat{bdp} = \widehat{b_r}\widehat{RTT}_{min}$) to cover "pathological cases" [5].

1) ProbeRTT*:* The required estimate $\widehat{RTT}_{min}$ is calculated by using a minimum filter over a window W_R as $\widehat{RTT}_{min} = \min(RTT_t)\ \forall t \in [T - W_R, T]$ with $W_R = 10\,s$. In order to measure RTT_{min}, BBR uses a periodically occuring phase which is called *ProbeRTT* (the following explanation also considers multiple flows). *ProbeRTT* is entered when $\widehat{RTT}_{min}$ has not been updated by a lower measured value for several seconds (default: 10 s). In *ProbeRTT*, the sender abruptly limits its amount of inflight data to 4 packets for $\max(RTT, 200\,ms)$ and then returns to the previous state. This should drain the queue completely under the assumption that only BBR flows are present, so $\widehat{RTT}_{min} = RTT_{min}$ since no queuing delay exists.

RTT_{min} cannot be correctly measured in *ProbeRTT* if other flows are creating a queuing delay. However, large flows that enter *ProbeRTT* will drain many packets from the queue, so other flows will update their $\widehat{RTT}_{min}$. This lets their $\widehat{RTT}_{min}$ expire at the same time so that these flows enter *ProbeRTT* nearly simultaneously, which again may create a larger drain effect, letting other flows see a new $\widehat{RTT}_{min}$ and so on. This synchronized behavior increases the probability to actually measure RTT_{min}. It is important to note that $\widehat{RTT}_{min}$ only

modifies the inflight cap of $2\widehat{bdp}$, the sending rate is otherwise determined just by the recent maximum measured delivery rate.

2) ProbeBW *Phase:* BBR probes for more bandwidth by increasing the sending rate by a certain factor ($pacing\_gain = 1.25$) for an estimated RTT_{min}, then decreasing the sending rate by $pacing\_gain = 0.75$ in order to compensate a potential excess of inflight data. Thus, if this excess amount filled the bottleneck queue, the queue should be drained by the same amount directly afterwards. Moreover, the sending rate is varied in an eight-phase cycle using a $pacing\_gain$ of 5/4,3/4,1,1,1,1,1,1, where each phase lasts for an $\widehat{RTT}_{min}$. The start of the cycle is randomly chosen with 3/4 being excluded as initial phase. If the increased sending rate showed an increased delivery rate, the newly measured maximum delivery rate $\widehat{b_r}$ is immediately used as new sending rate, otherwise the previous rate is maintained.

B. Startup Behavior

In its startup phase BBR nearly doubles its sending rate every RTT as long as the delivery rate is increasing. This is achieved by using a $pacing\_gain$ of $2/\ln 2 = 2.885$ and an inflight cap of $3bdp$, i.e., it may create up to $2bdp$ of excess queue. BBR tries to determine whether it has saturated the bottleneck link by looking at the development of the delivery rate. If for several (three) rounds attempts to double the sending rate results only in a small increase of the delivery rate (less than 25%), BBR has found the current limit of $\widehat{b_r}$ and exits the startup phase. Then it enters a drain phase in order to reduce the amount of excess data that may have led to a queue. It uses the inverse of the startup's gain to reduce the excess queue and enters the *ProbeBW* phase once $D^{inflight} = bdp$ holds.

III. Analysis of BBR's Behavior

The following analysis is mainly based on the concepts described in [5], [7], some details were taken from the Linux implementation. While BBR's mechanisms work well for a single flow (confirmed by the results presented in section V-A), the situation is different when multiple flows share the bottleneck. BBR's model in fig. 1 reflects the aggregate behavior of all flows at the bottleneck, but not the perspective of an individual sender if multiple flows are traversing the bottleneck. As a result, each sender overestimates the available bandwidth leading to a too high total $D^{inflight}$. This analysis explains that BBR's mechanisms lead to inherent properties such as increased queuing delays, high packet losses, and unfair flow rate shares in certain settings. These properties were also observed in the evaluation results presented in section V. The following analysis assumes that the flows' sending rates are only limited by the congestion control. If the sending rate is application-limited the observed behavior will differ.

A. Multiple BBR Flows Steadily Overload the Bottleneck

A BBR flow i uses the observed maximum filtered delivery rate $\widehat{b_{r_i}}$ as its sending rate s_{r_i}. BBR's assumption is that $\widehat{b_{r_i}}$ of a flow i does not grow anymore beyond passing operating point (A) in fig. 1, which does not necessarily hold if multiple flows are sharing the bottleneck. In the latter case, an individual BBR flow i does not estimate the bottleneck bandwidth b_r, but its *available share* b_{r_i} of the bottleneck bandwidth, with $\sum_i b_{r_i} = b_r$. The maximum filtered delivery rate $\widehat{b_{r_i}}$ of flow i serves as estimate of b_{r_i}, which corresponds to the actual delivery rate of flow i. However, there is an important difference between $\sum_i \widehat{b_{r_i}}$ and $\widehat{b_r}$ from the model described above.

While the condition $\widehat{b_r} \leq b_r$ holds for a single flow, because the observed delivery rate $\widehat{b_r}$ cannot be larger than b_r in this particular case, it is possible (and often happens) that $\widehat{b_{r_i}} > b_{r_i}$ and also even $\sum_i \widehat{b_{r_i}} > b_r$. Since BBR uses $\widehat{b_{r_i}}$ as sending rate s_{r_i} of a flow i (i.e., $s_{r_i} := \widehat{b_{r_i}}$) it follows that $\sum_i s_{r_i} > b_r$, i.e., all flows together send faster than the bottleneck rate, thereby overloading the bottleneck.

So how come $\sum_i \widehat{b_{r_i}} > b_r$? This can be explained by the combination of BBR's rate-based approach, $\widehat{b_{r_i}}$ being the windowed maximum filter over b_{r_i}, and the *ProbeBW* phase.

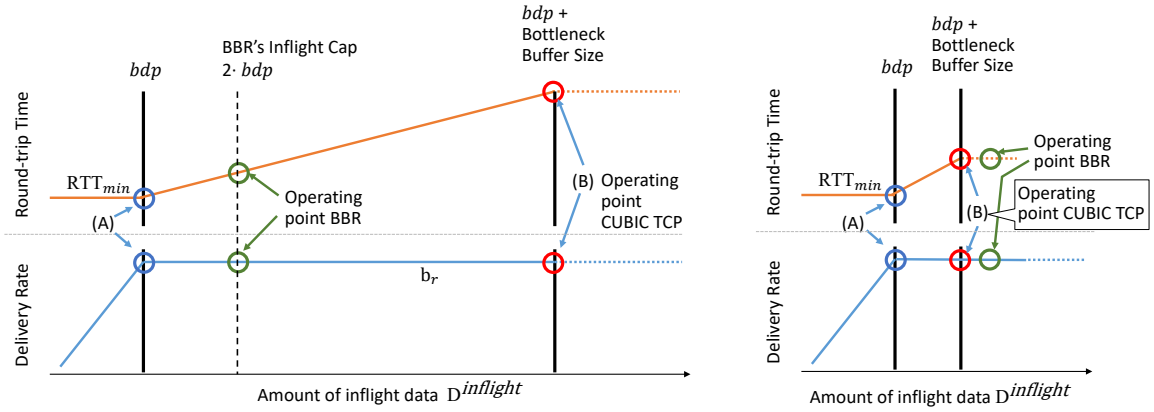
Let us assume that at some point in time, some of the flows (let flow i be one of them) will start to probe for more bandwidth, by sending with $p_g\widehat{b_{r_i}}$ (with p_g:=5/4=1.25 being the maximum probing gain) for an RTT. In case the bottleneck input rate $I = \sum_j s_{r_j}$ is larger than b_r, every flow j will get only $s_{r_j} \cdot b_r/I$ through the bottleneck (the excess is either queued or dropped). The delivery rate during probing will be $b_{r_i} = p_g\widehat{b_{r_i}} \cdot \min(b_r/I_p, 1)$, with I_p being the bottleneck input rate (including the increased rates of the probing flows). The updated $\widehat{b'_{r_i}}$ (after probing) satisfies

$$\widehat{b'_{r_i}} \geq p_g\widehat{b_{r_i}} \cdot \min(b_r/I_p, 1). \tag{1}$$

Now, if $I_p < p_g b_r$ (i.e., the bottleneck overload is less than 25%), eq. (1) will evaluate to $\widehat{b'_{r_i}} > \widehat{b_{r_i}}$. This means each probing flow i will actually measure a higher delivery rate. This updates the maximum filter and will *immediately* be used as new s_{r_i} for at least *further* 10 RTTs.

Now we distinguish between the following initial states:

1) $\sum_j \widehat{b_{r_j}} < b_r$, i.e., the bottleneck is not overloaded yet. In this case it follows that $I_p < p_g b_r$ holds, because even if all flows probe $I_p \leq \sum_j p_g\widehat{b_{r_j}} = p_g \sum_j \widehat{b_{r_j}} < p_g b_r$. Consequently, in this case all probing flows can increase their sending rates.
2) $\sum_j \widehat{b_{r_j}} = b_r$, i.e., the bottleneck is already fully utilized. If all flows probe at the same time (or if there is only a single flow) $I_p = p_g b_r$, thus $\widehat{b'_{r_i}} = \widehat{b_{r_i}}$, i.e., the probing flows will not measure a higher delivery rate, but refresh their current $\widehat{b_{r_i}}$ in the maximum filter. If not all flows probe simultaneously, $I_p < p_g b_r$ follows and thus every probing flow i will increase its sending rate after probing. Thus, $\widehat{b'_{r_i}} > \widehat{b_{r_i}}$ for each probing flow i and $\widehat{b'_{r_l}} = \widehat{b_{r_l}}$ for each non-probing flow l due to the maximum filter. Thus, summing over all flows yields $\sum_j \widehat{b'_{r_j}} > b_r$.
3) $\sum_j \widehat{b_{r_j}} > b_r$, i.e., the bottleneck is already overloaded. In this case I_p may be larger than $p_g b_r$. Still, $\widehat{b'_{r_j}} \geq b_{r_j}$



(a) Large buffer (b) Small buffer

Fig. 2: Observed operating points of BBR and CUBIC TCP with large and small buffer

due to the maximum filter. This means $\sum_j \widehat{b'_{r_j}} \geq b_r$ since $\sum_j b_{r_j} = b_r$. So even if some of the flows cannot increase their sending rate during probing $\sum_j \widehat{b_{r_j}} \geq b_r$ still holds.

Note that even if $\sum_j \widehat{b_{r_j}} > b_r$, I_p depends on the sending rate of the probing flow set and is not necessarily larger than $p_g b_r$. From the reasoning above, it can be concluded that, unless all flows probe at the same time, $\exists X \in (b_r, p_g b_r)$ so that $I_p < p_g b_r$ even if $\sum_i \widehat{b_{r_i}} = X$. In other words: Flows can increase their sending rate even if the bottleneck is already overloaded.

To summarize: $\sum_i \widehat{b_{r_i}} \geq b_r$ can happen for sustained periods of time with a tendency toward $\sum_i \widehat{b_{r_i}} > b_r$, if multiple BBR flows share a bottleneck. Thus the input rate at the bottleneck is larger than the bottleneck capacity. The excess data is queued at the bottleneck if the bottleneck buffer is not already exhausted. Consequently, due to the rate mismatch, the amount of inflight data steadily increases and is not decreased until *ProbeRTT*.

B. *Large Buffers* – BBR *Operates at its Inflight Cap*

In large buffers the increase of inflight data is limited by the inflight cap. Let D_i^{cap} be the inflight cap of flow i and assume that all flows have the same and correctly measured $\widehat{RTT}_{min}$. Then the amount of inflight data is bounded by $\sum_i D_i^{cap} = 2 \cdot \sum_i \widehat{b_{r_i}} \cdot \widehat{RTT}_{min}$. As discussed in the previous section, it is likely that $\sum_i \widehat{b_{r_i}} \in [b_r, 1.25 b_r)$, thus, $2bdp \leq \sum_i D_i^{cap} < 2.5bdp$. Since multiple BBR flows steadily increase the amount of inflight data, as shown in section III-A it can be expected that the inflight cap is regularly reached. This means that multiple BBR flows typically create a queuing delay of about one to 1.5 times the RTT. Experimental results shown in section V confirm this expectation. Thus, if multiple BBR flows share a bottleneck, BBR does not operate at point (A), as illustrated in fig. 2a.

If flows with different RTTs share a bottleneck, a flow i with a larger RTT than a flow j will usually get a larger rate share than j. Since D_i^{cap} depends on the RTT, $D_i^{cap} > D_j^{cap}$ follows. Usually, about one half of the allowed inflight data is "on the wire", the other half can potentially be queued at the bottleneck. This means flow i can usually queue more data at the bottleneck than flow j before reaching the inflight cap. This directly results in a larger rate share for i, which further increases D_i^{cap}.

C. *Small Buffers – Massive Packet Loss*

If the bottleneck buffer is smaller than a bdp (note that slightly smaller suffices, it does not have to be shallow) the bottleneck buffer is exhausted before the inflight cap is reached. This means point (B) is reached and packet loss occurs (fig. 2b). In order to handle non-congestion related packet loss, BBR does not back off if packet loss is detected. But in this case the packet loss *is* caused by congestion. Since BBR has no means to distinguish congestion related from non-congestion related loss, point (B) is actually crossed, which can lead to massive amounts of packet loss, as shown in section V-E. If flows with loss-based congestion control (e.g., CUBIC TCP) share the same bottleneck, they interpret the sustained high packet loss rates (correctly) as a sign of massive congestion and back off to very low transmission rates. Furthermore, [5] contains no explicit statement regarding congestion collapse prevention. In order to assess this issue, the loss recovery mechanisms of BBR have to be investigated further. But this is out of scope of this paper.

IV. TESTBED SETUP

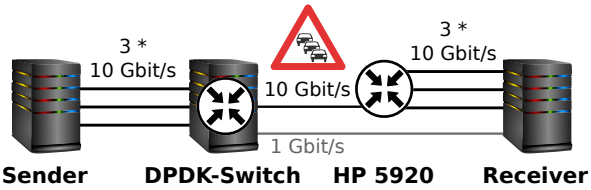


Fig. 3: Testbed setup

In order to evaluate BBR experimentally, we set up a physical testbed. The testbed has two configurations, 1) with a 10 Gbit/s bottleneck, and 2) with a 1 Gbit/s bottleneck, as shown in fig. 3. The 10 Gbit/s configuration has the commonly

used dumbbell topology. It consists of two 10 Gbit/s switches, a *sender* and a *receiver*, both equipped with three 10 Gbit/s interfaces. Due to interface speed limitations we could not use the same set-up to get a 1 Gbit/s bottleneck. Therefore, the 1 Gbit/s configuration uses a slightly modified topology with dedicated 1 Gbit/s NICs. If the 1 Gbit/s is enabled the 10 Gbit/s link is disabled and vice versa.

A *DPDK*-based software switch[1] was used at the bottleneck link. It provides detailed control over its buffers and also contains a delay emulator that was used to experiment with different RTTs. The emulator added artificial delay only in the direction to the sender, i.e., merely the ACKs were delayed. Since BBR does not use one-way delay measurements this does not influence the results. The Linux kernel module *netem* was not used as delay emulator, since it produced severe side effects like delay jitter and packet loss at speeds of 10 Gbit/s.

Sender, receiver, and the DPDK switch run on *Ubuntu 16.04* and are equipped with two Intel Xeon E5-2630 v3 CPUs and a 4-port Intel X710 10 Gbit/s NIC. For the 1 Gbit/s configuration, the DPDK switch and the receiver also have a dual-port Intel 82571EB Gigabit Ethernet Controller. In order to support BBR, the sender uses Linux kernel version 4.9, the other servers use version 4.4. At the sender, we enabled the *queuing discipline* `"fq"` for all network interfaces that were used for BBR flows. CUBIC TCP flows were always sent over different interfaces than BBR flows; `"fq"` was not enabled for flows of CUBIC TCP. BBR relies on `"fq"` to work properly, since it implements its packet pacing feature. While BBR determines the pacing rate, the actual pacing of the outgoing packets is carried out by `"fq"`. Traffic is generated with `iperf3`. RTT, CWnd and *goodput* measurements are collected at the sender with an open source tool[2] that is based on the kernel module `tcpprobe`; `tcpprobe` gives access to internal TCP state information of the Linux kernel. Throughput values are collected with another open source tool[3] that evaluates the counters of the network interfaces. Since we observed packet loss in the end system at speeds around 10 Gbit/s that were caused by a buffer overflow in the "RX ring" (between NIC and operating system), we increased the size of the RX rings of the 10 Gbit/s NICs at the receiver from 512 to 4096. This prevents the packet loss but increases the delay jitter induced by the end system. We also increased the maximum values for auto-tuning of TCP flow control, since the standard values are too small for 10 Gbit/s flows, at the used RTTs.

At the bottleneck, we used three different buffer sizes:

- 200 MByte: $\widehat{=}$ 160 ms queuing delay at 10 Gbit/s, when fully occupied.
- 20 MByte: $\widehat{=}$ 16 ms queuing delay at 10 Gbit/s, 160 ms at 1 Gbit/s
- 2 MByte: $\widehat{=}$ 16 ms queuing delay at 1 Gbit/s.

In the following we will also denote this buffer sizes as *"large"* and *"small"*, depending on whether they produce 160 ms or 16 ms queuing delay, respectively. For RTT_{min} =20 ms, this corresponds to $8bdp$ for large and $0.8bdp$ for small buffers.

For experiments with no more than three flows, each flow is sent over a different network interface. Otherwise, the flows are reasonably distributed among the interfaces (i.e., equally distributed, if possible; different interfaces for BBR and CUBIC TCP). In all of our experiments, the start times of the flows are chosen in a way that the time difference between two start times is no multiple of 10 s. Since BBR usually enters *ProbeRTT* every 10 s, this would induce unintended bias.

We repeated each experiment at least five times. In the following, we always show the results of a representatively chosen single run for clarity.

V. EVALUATION

A. BBR – *Intended Behavior*

The intended behavior of BBR can be nicely observed if only a single flow is active at the bottleneck link. The results of a corresponding experiment are depicted in fig. 4 with respect to throughput and RTT. The BBR flow can fully utilize the provided link capacity. However, throughput drops occur regularly (every 10s), caused by the *ProbeRTT* phase. After startup, the RTT increases up to 60 ms, corresponding to the inflight cap of three bdp during startup. After that the RTT is regularly increased to 25 ms – 27 ms during the *ProbeBW* phase. When $pacing\_gain = 1.25$ a queue builds up at the bottleneck that is drained in the subsequent phase with $pacing\_gain = 0.75$. The *ProbeRTT* phase can also be well observed in fig. 4c around *second 10.5* by the missing RTT spike. The minimal measured RTT is slightly above the delay of 20 ms which is induced by the delay emulator. The difference is the actual delay induced by the propagation delay in the testbed as well as by the processing times of the end systems and the switch.

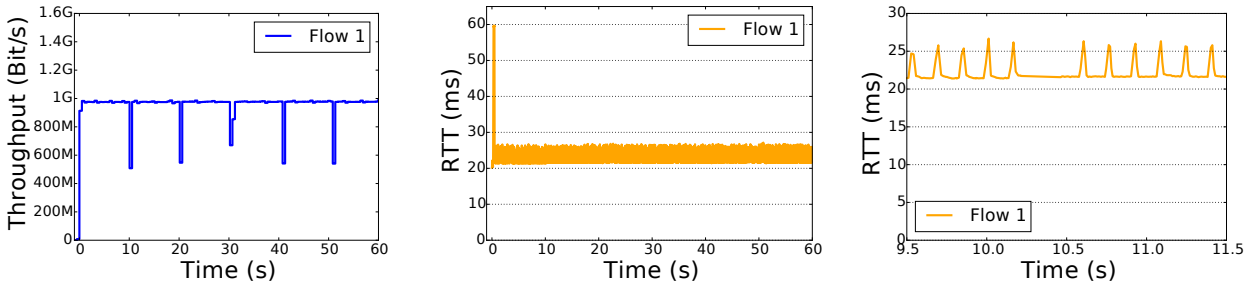
B. *Multiple* BBR *Flows*

BBR shows a different behavior if multiple flows share the same bottleneck link. Figure 5 depicts results of experiments with 1, 2, 4, and 6 BBR flows at the bottleneck. It can be observed that the RTT is increased to a value around 40 ms most of the time (as explained in sections III-A and III-B). The peak at the beginning is caused by the startup phase of BBR. In the experiments with 4 and 6 flows, the peak is significantly larger and longer lasting than in the case of one or two flows. If a BBR flow starts up when the RTT is already increased, the flow overestimates RTT_{min} and, thus, the bdp. Consequently, the inflight cap is set to a larger value than three times the (actual) bdp, causing larger queuing delays.

C. *Impact of* RTT_{min}

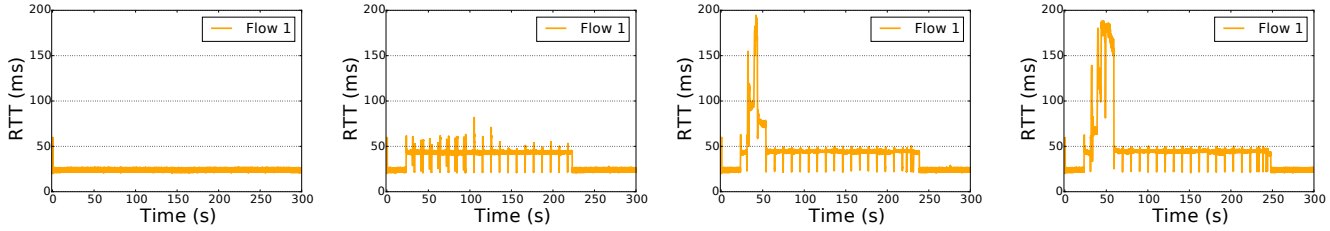
As just seen, multiple BBR flows are not able to drain the buffer as one would have expected. In fact, the RTT doubles if (at least) two BBR flows with the same RTT_{min} share a bottleneck as clearly shown in fig. 6. Here, RTT_{min} is varied from 5 ms to 80 ms, causing effective RTTs of about 10 ms to 160 ms. This doubling of the RTT corresponds to BBR's inflight cap of *two* bdp (as explained in section III-B), so multiple BBR flows reach this inflight cap most of the time.

[1] https://git.scc.kit.edu/TM/DPDK_AQM_Switch
[2] https://git.scc.kit.edu/CPUnetLOG/TCPlog
[3] https://git.scc.kit.edu/CPUnetLOG/CPUnetLOG



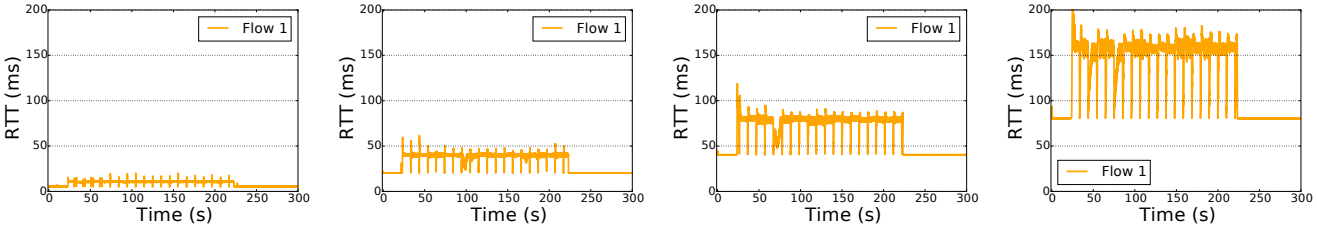
(a) Throughput (b) RTT (c) RTT (zoomed-in)

Fig. 4: A single BBR flow with 1 Gbit/s bottleneck



(a) 1 flow (b) 2 flows (c) 4 flows (d) 6 flows

Fig. 5: RTT of different numbers of competing BBR flows at a 1 Gbit/s bottleneck (large buffer), RTT_{min} =20 ms



(a) 5 ms (b) 20 ms (c) 40 ms (d) 80 ms

Fig. 6: RTT of two BBR flows at a 10 Gbit/s bottleneck (large buffer), comparison of different RTT_{min}

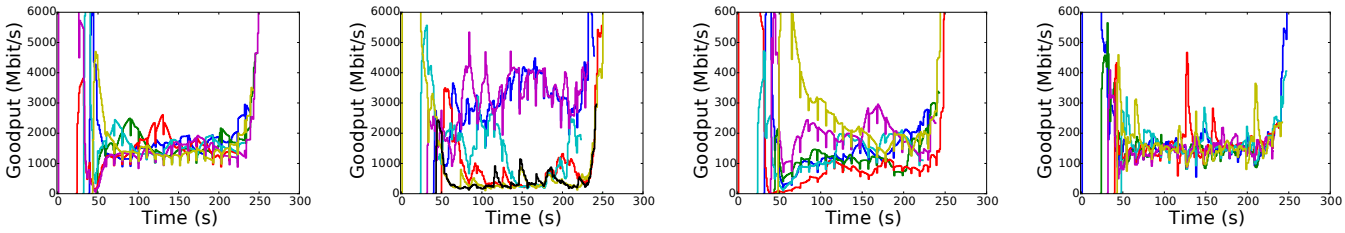
D. Intra-protocol Fairness

Multiple BBR flows with the same RTT_{min} can achieve a reasonable fair rate share in some of the tested scenarios. However, our experiments show that a sustained suppression of individual flows is also possible. Figure 7 shows the *goodput* of six BBR flows that are successively started (starting points: 0 s, 23 s, 31 s, 38 s, 42 s, 47 s) in scenarios that varied data rates (10 Gbit/s, 1 Gbit/s) and buffer sizes (large, small). Here, "goodput" denotes payload data transmitted by TCP, excluding retransmissions. In the scenario with a 10 Gbit/s bottleneck and small buffers (shown in fig. 7b) two of the flows get a significantly larger rate share than the remaining four flows. These flows, in turn, only achieve very small rates, for prolonged timespans. This behavior could be observed in all repetitions of this experiment. Section III-C suggests that fairness can be quite random until inflight caps are reached.

In the other scenarios all six flows get similar rate shares to some extent. However, (almost) identical rate shares are usually not achieved. As shown in fig. 7c, rate differences of around 100 Mbit/s between individual flows are not uncommon.

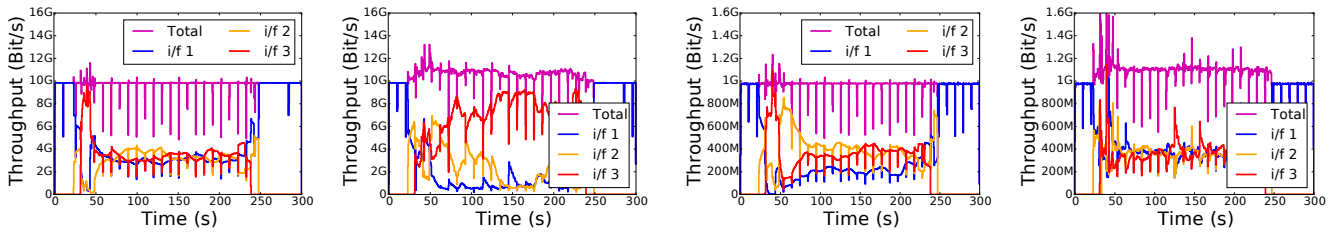
E. Packet Loss

While fig. 7 shows the goodput of the six BBR flows, fig. 8 shows the *outgoing data rate* of the three network interfaces (*if1*, *if2*, *if3*) of the sender (including headers and retransmissions). In addition to that, the sum of these data rates is also shown. Since there are less interfaces available than flows, two flows are sent via each interface. It can be seen that this sum is way above the bottleneck capacity, especially for small buffer sizes (10 Gbit/s in fig. 8b, 1 Gbit/s in fig. 8d). This leads to a severe network overload and results in an enormous amount of retransmissions (more than 16 million packets) due to packet loss, see fig. 12a. For comparison we conducted the same experiment with CUBIC TCP. As shown in fig. 9, the total output rate is close to 10 Gbit/s and 1 Gbit/s, respectively.



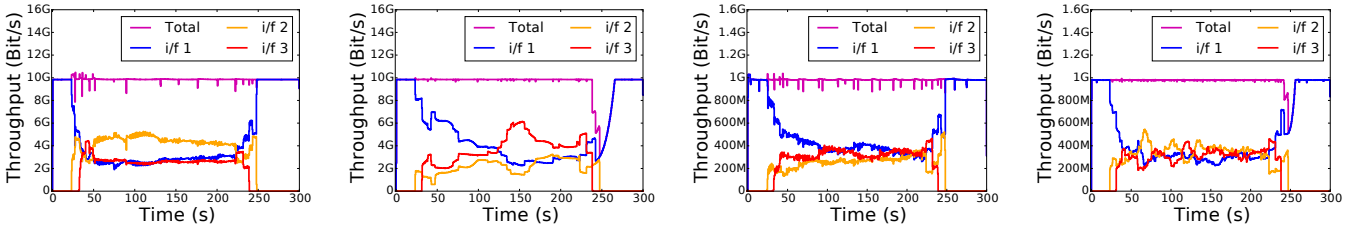
(a) 10 Gbit/s large buffer (b) 10 Gbit/s small buffer (c) 1 Gbit/s large buffer (d) 1 Gbit/s small buffer

Fig. 7: Goodput of six BBR flows in different scenarios, RTT_{min} =20 ms



(a) 10 Gbit/s large buffer (b) 10 Gbit/s small buffer (c) 1 Gbit/s large buffer (d) 1 Gbit/s small buffer

Fig. 8: BBR – Outgoing data of sender interfaces (same experiments as in fig. 7)



(a) 10 Gbit/s large buffer (b) 10 Gbit/s small buffer (c) 1 Gbit/s large buffer (d) 1 Gbit/s small buffer

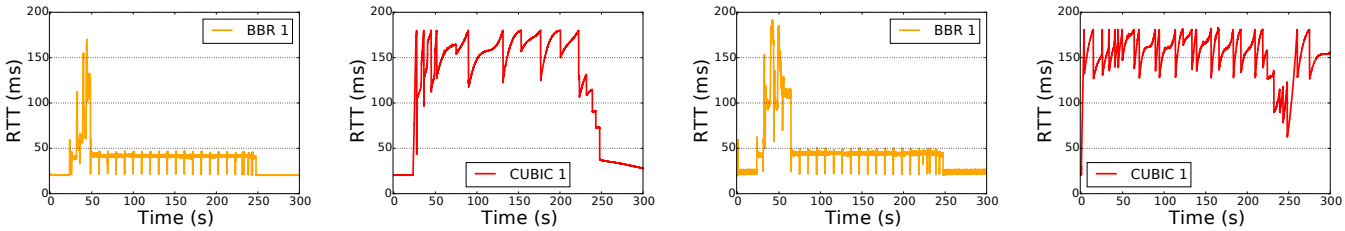
Fig. 9: CUBIC TCP – Outgoing data of sender interfaces (same setting as in fig. 8)

Still around 2 000 and 12 000 retransmissions occur. But this is several orders of magnitude lower than those caused by BBR.

With large buffers, the number of retransmissions of the BBR flows is significantly lower than with small buffers. The reason for this was already given in section III-B and illustrated in fig. 2: The inflight cap limits the buffer utilization to around one bdp. With large buffers this avoids packet loss most of the time (except during the startup phases of the BBR flows). But even with $0.8\,bdp$ as "small buffer" size, the inflight cap is too large to avoid a persistent overload of the bottleneck buffer. In contrast to CUBIC TCP and most other TCP congestion controls, BBR ignores packet loss as a congestion signal. However, in the just described experiments, the huge amount of packet loss is clearly a result of persisting congestion.

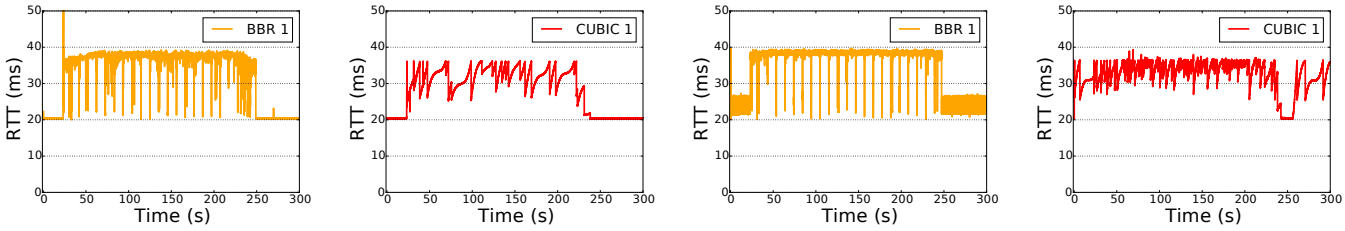
The amount of inflight data can be indirectly seen by the RTT of the flows. If there is more than one bdp of data in flight, the RTT increases due to queuing delay. Figure 10 shows the RTTs of flow 1 in the respective experiments with large buffers. Since all flows experience about the same queuing delay, we only show the RTT of a single flow for clarity. CUBIC TCP regularly fills the bottleneck buffer up to exhaustion and backs off afterwards. BBR increases the RTT to a similar amount during startup, after that the RTT is reduced to about twice RTT_{min}. This means that about one bdp is queued in the bottleneck buffer.

With small buffers, operating point (B) (cf. fig. 2) lies further to the left and thus CUBIC TCP's operating point as well (illustrated by the plots of the RTT in figs. 11b and 11d). Again, CUBIC TCP fills the bottleneck buffer up to exhaustion and backs off afterwards. With BBR the observed RTT is almost constantly at the maximum level, just below 40 ms (20 ms delay emulator + 16 ms maximum queuing delay + actual delay, i.e., processing times, etc.). Since the size of the small buffer is less than one bdp, the buffer overflows before the inflight cap is reached. This means that BBR's operating point is at least at (B). Additionally, the large number of retransmissions (fig. 12) and the increased total sending rate (figs. 8b and 8d) that is constantly significantly above the bottleneck bandwidth (10 Gbit/s / 1 Gbit/s, respectively) confirm that operating point (B) is actually crossed.



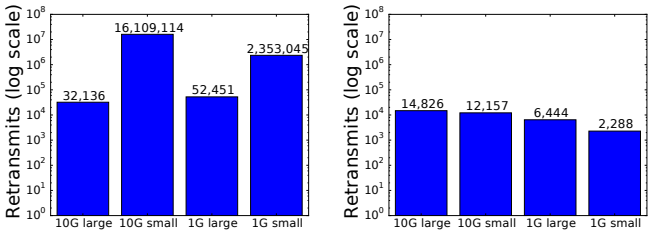
(a) 10 Gbit/s BBR (b) 10 Gbit/s CUBIC TCP (c) 1 Gbit/s BBR (d) 1 Gbit/s CUBIC TCP

Fig. 10: RTTs corresponding to figs. 8 and 9 – large buffer, RTT_{min} =20 ms



(a) 10 Gbit/s BBR (b) 10 Gbit/s CUBIC TCP (c) 1 Gbit/s BBR (d) 1 Gbit/s CUBIC TCP

Fig. 11: RTTs corresponding to figs. 8 and 9 – small buffer, RTT_{min} =20 ms



(a) BBR (b) CUBIC TCP

Fig. 12: Retransmissions (six flows)

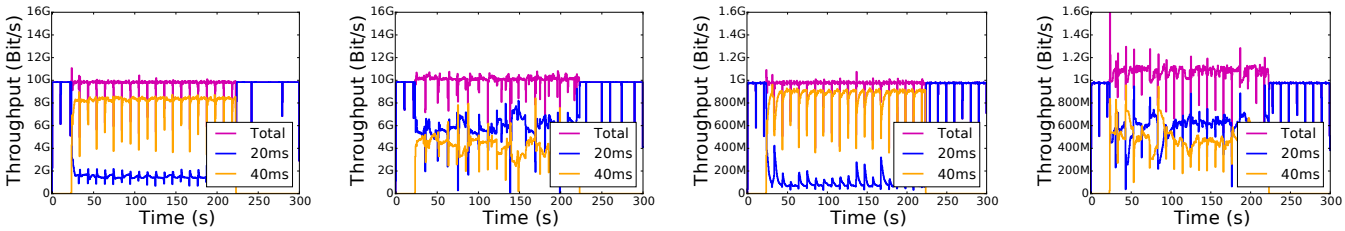
F. Fairness among Flows with Different RTTs

The fairness among BBR flows with different RTTs strongly depends on the bottleneck buffer size. Figure 13 shows the throughput of two BBR flows with different RTT_{min} (20 ms vs. 40 ms) that compete at a bottleneck in different scenarios: 10 Gbit/s / 1 Gbit/s bottleneck bandwidth, large/small buffers. With small buffers (figs. 13b and 13d) flow 1 (smaller RTT_{min}) gets more bandwidth than flow 2. Still, flow 2 gets around 40% of the bottleneck bandwidth. As discussed above, both flows are most likely not limited by their inflight cap, due to the small buffer size. With large buffers (figs. 13a and 13c) flow 2 (larger RTT_{min}) gets significantly more bandwidth than flow 1, because the buffer size is large enough so that both flows are most likely limited by their inflight cap (cf. section III-B). In this case, both flows can put one *bdp* into the bottleneck buffer. Due to the larger RTT_{min}, the *bdp* of flow 2 is larger as well. This corresponds to a larger share of data in the bottleneck buffer, thereby resulting in a larger throughput for flow 2, as documented in figs. 13a and 13c.

Additional experiments with three competing flows at the bottleneck (RTT_{min}: 20 ms, 40 ms, 80 ms) show the severity of the problem. In figs. 14a and 14c, flow 1 (20 ms RTT_{min}) is almost entirely suppressed by the other two flows. Flow 3 (80 ms RTT_{min}) gets more than 80% of the bottleneck bandwidth. In small buffers, flows are not limited by their inflight cap in this particular case, so different RTT_{min} have not much impact on unfairness. This, however, leads to a large and persisting overload of the bottleneck, resulting in packet loss. Again, figs. 14b and 14d show that the total output rate of the senders is significantly above the bottleneck bandwidth.

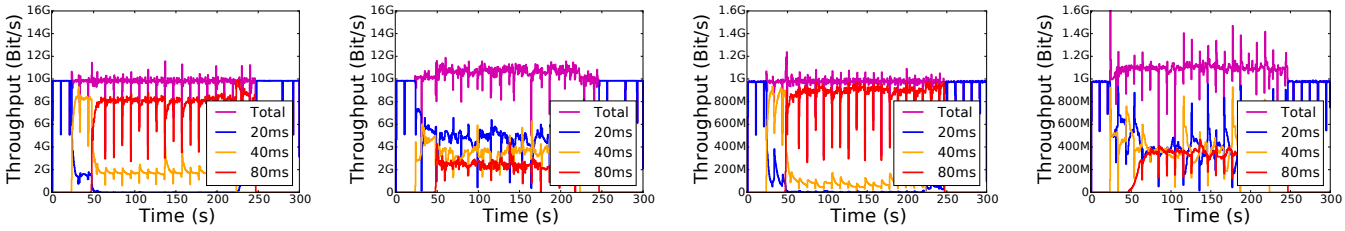
G. Inter Protocol Fairness with CUBIC TCP

In addition to the experiments above, which focused on BBR, we also conducted experiments on the interplay between BBR and CUBIC TCP. The fairness between a BBR flow and CUBIC TCP also depends on the size of the bottleneck buffer. Figures 15a and 15b show the throughput of one BBR (start: 0 s, end: 300 s) and one CUBIC TCP flow (start: 23 s, end: 223 s) that compete at a (10 Gbit/s / 1 Gbit/s) bottleneck with a large buffer. As loss-based congestion control, CUBIC TCP tends to fill the bottleneck buffer up to exhaustion, no matter how big the buffer is, whereas BBR limits its inflight data to two *bdp*. This means, the larger the bottleneck buffer, the larger the rate share of CUBIC TCP. However, since CUBIC TCP produces long lasting standing queues, a competing BBR flow may not be able to see the actual RTT_{min}, even during its *ProbeRTT* phase. During *ProbeRTT* the BBR flow reduces its own inflight data close to zero, however, the CUBIC TCP flow does not. Thus, the bottleneck buffer is usually not drained completely. In this case, the BBR flow assumes a higher RTT_{min} and, thus, also increases the inflight cap to a larger value. Both behaviors can be seen in figs. 15a and 15b. Occasionally, BBR



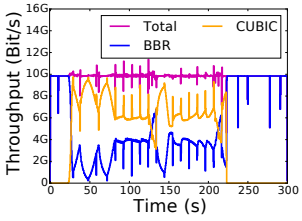
(a) 10 Gbit/s large buffer (b) 10 Gbit/s small buffer (c) 1 Gbit/s large buffer (d) 1 Gbit/s small buffer

Fig. 13: Two BBR flows with different RTT_{min} (20 ms, 40 ms)



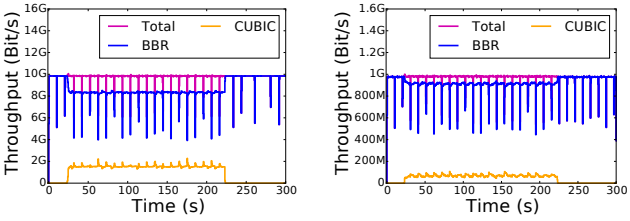
(a) 10 Gbit/s large buffer (b) 10 Gbit/s small buffer (c) 1 Gbit/s large buffer (d) 1 Gbit/s small buffer

Fig. 14: Three BBR flows with different RTT_{min} (20 ms, 40 ms, 80 ms)



(a) 10 Gbit/s – throughput (b) 1 Gbit/s – throughput

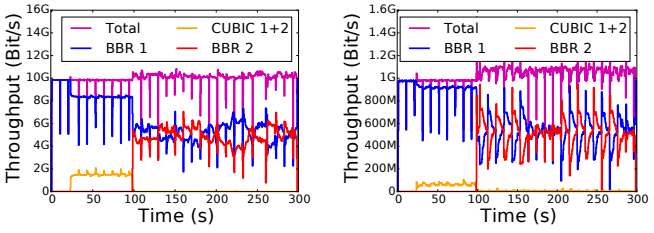
Fig. 15: BBR vs. CUBIC TCP (large buffer)



(a) 10 Gbit/s (b) 1 Gbit/s

Fig. 16: BBR vs. CUBIC TCP (small buffer)

gets a nearly fair share of the bottleneck bandwidth, at other times, most of the bottleneck bandwidth is occupied by the CUBIC TCP flow.



(a) 10 Gbit/s (b) 1 Gbit/s

Fig. 17: 2 BBR vs. 2 CUBIC TCP (small buffer)

With small buffers, BBR gets a vastly bigger rate share than CUBIC TCP (see fig. 16). However, there is a fundamental difference in the behavior depending on whether CUBIC TCP competes against a single or against multiple BBR flows. Therefore, we conducted further experiments with two BBR flows and two CUBIC TCP flows (shown in fig. 17). The experiments started the same way as the previous one. The additional flows were started at *second 98* (BBR) and *second 123* (CUBIC TCP), respectively. Once the two BBR flows are active, the throughput of the CUBIC TCP flow(s) drops close to zero. If the bottleneck buffer is smaller than one *bdp*, two competing BBR flows create large amounts of packet loss thereby also affecting the CUBIC TCP flows. While CUBIC TCP treats them as congestion signal and backs off, BBR keeps sending at a high rate, completely suppressing the CUBIC TCP flows.

VI. Related Work

BBR was originally described in [5], [7] and updates were presented by presentations [6], [8]. Recently, an Internet-Draft [9] complemented the description. Evaluation results shown in [5], [7] consist of scenarios with 10 Mbit/s and 100 Mbit/s bottleneck link bandwidth, RTTs are mostly 40 ms. The evaluation compares runs of a single BBR flow with runs of a single CUBIC TCP flow, but no interprotocol fairness. Intra-protocol fairness is shown with five BBR flows at 100 Mbit/s with 10 ms RTT. Additionally, results are presented from

measurements in Google's B4 WAN. [6] shows interprotocol fairness results with CUBIC TCP at 10 Mbit/s and an $8bdp$ buffer. While these results are confirmed by our measurements at higher speeds in section V-G (cf. fig. 15), this shows only the behavior of a single BBR flow against a single CUBIC TCP flow. Presentation [6] also shows that the BBR team is aware that BBR may suppress loss-based flows in a small bottleneck buffer (as confirmed by fig. 16). Moreover, [6] shows results for RTT fairness (also at 10 Mbit/s), but our observed unfairness seems to be larger (cf. fig. 13) at higher speeds. Further results were presented in [8] also showing interprotocol fairness to TCP Reno (also at 10 Mbit/s, 40 ms RTT) and comparing it to CUBIC TCP's fairness to TCP Reno. However, these results are only given for large buffers where BBR is mainly limited by its inflight cap. To best of our knowledge no independent evaluation results have been published as research paper yet.

Congestion control is still an active area of research. Several proposals have been made in the last years to achieve similar goals that BBR strives for. A quite old approach to avoid queuing delay (and thus operate at point (A) of fig. 1) is TCP Vegas [4]. However, TCP Vegas is being suppressed by loss-based congestion control, not scalable to higher speeds and has got issues to control the total queuing delay when multiple TCP Vegas flows share the same bottleneck. FAST TCP [16] is also based on TCP Vegas and aims toward high speed wide-area networks but does not have low queuing delay as design goal. YeAH TCP [3] and CDG [13] try also to keep the queuing delay low. They have mechanisms that increase their aggressiveness in case they are suppressed by other flows. CDG has a TCP Reno like additive increase that is not scalable for high-speed networks. YeAH TCP has a mechanism to drain the buffer while keeping a high link utilization at the bottleneck. TCP LoLa [14] is able to operate near (A) if one or multiple flows share a bottleneck. Furthermore, it incorporates a dedicated mechanism that provides a convergence to fairness among the flows. TCP LoLa, however, is not designed to compete with flows using a loss-based congestion control.

VII. Conclusion

This paper presents an extensive evaluation of the recently proposed congestion control BBR with 10 Gbit/s and 1 Gbit/s bottleneck links, multiple flows, and different RTTs. The results show that the concepts of BBR work quite well for a single flow at a bottleneck. However, the observed behavior of multiple flows does not meet BBR's original goal. BBR is based on a model that reflects the aggregate behavior of all flows at the bottleneck, but not the perspective of an individual sender. BBR's mechanisms inherently lead to a sustained overload of the bottleneck resulting in a steadily increasing amount of inflight data, queuing up at the bottleneck buffer. BBR has no mechanism to drain this unintentionally built-up queue, except *ProbeRTT* which is triggered at most every 10 s. This leads to the observed problems of increased queuing delays and RTT unfairness within large buffers as well as a massive amount of packet losses and unfairness to flows with loss-based congestion control in smaller buffers. BBR can keep throughput high even in case of packet loss, but ignoring loss caused by congestion can also lead to the observed cases with massive packet loss. Furthermore, BBR has no explicit mechanism to let multiple BBR flows converge to a fair share. BBR has neither an explicit congestion detection mechanism nor an explicit reaction to congestion. Since BBR is still under active development the presented evaluation results represent a snapshot of its current development state. However, we believe that the provided insights on BBR's mechanisms and their performance are of general value for congestion control research. Moreover, investigation of BBR's behavior with Active Queue Management mechanisms is interesting, since BBR does not react to packet loss as congestion signal.

Acknowledgment

This work was supported by the bwNET100G+ project, which is funded by the Ministry of Science, Research, and the Arts Baden-Württemberg (MWK). The authors alone are responsible for the content of this paper. Thanks to Polina Goltsman for her help with the testbed configuration.

References

[1] A. Afanasyev, N. Tilley, P. Reiher, and L. Kleinrock, "Host-to-Host Congestion Control for TCP," *Communications Surveys Tutorials, IEEE*, vol. 12, no. 3, pp. 304–342, May 2010.

[2] M. Allman, V. Paxson, and E. Blanton, "TCP Congestion Control," RFC 5681, IETF, Sep. 2009.

[3] A. Baiocchi, A. P. Castellani, and F. Vacirca, "YeAH-TCP: Yet Another Highspeed TCP," in *Int. Workshop on Protocols for Future, Large-Scale and Diverse Network Transports (PFLDNeT)*, vol. 7, 2007, pp. 37–42.

[4] L. S. Brakmo, S. W. O'Malley, and L. L. Peterson, "TCP Vegas: New Techniques for Congestion Detection and Avoidance," in *SIGCOMM '94*. New York, NY, USA: ACM, 1994, pp. 24–35.

[5] 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.

[6] ——, "BBR Congestion Control," Presentation in ICCRG at IETF 97th meeting, Nov. 2016. [Online]. Available: https://www.ietf.org/proceedings/97/slides/slides-97-iccrg-bbr-congestion-control-02.pdf

[7] ——, "BBR: Congestion-based Congestion Control," *Communnications of the ACM*, vol. 60, no. 2, pp. 58–66, Jan. 2017. [Online]. Available: http://doi.acm.org/10.1145/3009824

[8] ——, "BBR Congestion Control: An update," Presentation in ICCRG at IETF 98th meeting, Mar. 2017. [Online]. Available: https://www.ietf.org/proceedings/98/slides/slides-98-iccrg-an-update-on-bbr-congestion-control-00.pdf

[9] N. Cardwell, Y. Cheng, S. H. Yeganeh, and V. Jacobson, "BBR Congestion Control," Jul. 2017, Internet-Draft draft-cardwell-iccrg-bbr-congestion-control-00, IETF, work in progress.

[10] G. Fairhurst, B. Trammell, and M. Kuehlewind, "Services Provided by IETF Transport Protocols and Congestion Control Mechanisms," RFC 8095, IETF, Mar. 2017.

[11] J. Gettys and K. Nichols, "Bufferbloat: Dark Buffers in the Internet," *ACM Queue*, vol. 9, no. 11, pp. 40–54, Nov. 2011.

[12] S. Ha, I. Rhee, and L. Xu, "CUBIC: A New TCP-friendly High-speed TCP Variant," *SIGOPS Oper. Syst. Rev.*, vol. 42, no. 5, pp. 64–74, Jul. 2008.

[13] D. A. Hayes and G. Armitage, "Revisiting TCP Congestion Control Using Delay Gradients," in *NETWORKING'11*. Springer, 2011, pp. 328–341.

[14] M. Hock, F. Neumeister, M. Zitterbart, and R. Bless, "TCP LoLa: Congestion Control for Low Latencies and High Throughput," in *2017 IEEE 42nd Conference on Local Computer Networks*, Oct. 2017.

[15] D. Papadimitriou, M. Welzl, M. Scharf, and B. Briscoe, "Open Research Issues in Internet Congestion Control," RFC 6077, IETF, Feb. 2011.

[16] D. X. Wei, C. Jin, S. H. Low, and S. Hegde, "FAST TCP: Motivation, Architecture, Algorithms, Performance," *IEEE/ACM Transactions on Networking*, vol. 14, no. 6, pp. 1246–1259, Dec. 2006.