Fairness of Congestion-Based Congestion Control: Experimental Evaluation and Analysis

Shiyao Ma*, Jingjie Jiang*, Wei Wang*, Bo Li*
*Department of Computer Science and Engineering, Hong Kong University of Science and Technology

***Abstract*—BBR is a new congestion-based congestion control algorithm proposed by Google. A BBR flow sequentially measures the bottleneck bandwidth and round-trip delay of the network pipe, and uses the measured results to govern its sending behavior, maximizing the delivery bandwidth while minimizing the delay. However, our deployment in geo-distributed cloud servers reveals a severe RTT fairness problem: a BBR flow with longer RTT dominates a competing flow with shorter RTT.**

Somewhat surprisingly, our deployment of BBR on the Internet and an in-house cluster unearthed a consistent bandwidth disparity among competing flows. Long BBR flows are bound to seize bandwidth from short ones. Intrigued by this unexpected behavior, we ask, is the phenomenon intrinsic to BBR? how's the severity? and what's the root cause? To this end, we conduct thorough measurements and develop a theoretical model on bandwidth dynamics. We find, as long as the competing flows are of different RTTs, bandwidth disparities will arise. With an RTT ratio of 10, even flow starvation can happen. We blame it on BBR's connivance at sending an excessive amount of data when probing bandwidth. Specifically, the amount of data is in proportion to RTT, making long RTT flows overwhelming short ones. Based on this observation, we design a derivative of BBR that achieves guaranteed flow fairness, at the meantime without losing any merits. We have implemented our proposed solution in Linux kernel and evaluated it through extensive experiments.

I. Introduction

BBR [1] (Bottleneck Bandwidth and RTT) emerges as a new congestion-based TCP congestion control algorithm that, for the first time, converges to Kleinrock's optimal operating point [2], maximizing delivery rate while minimizing round-trip time (RTT). Unlike traditional loss- or delay-based congestion control (e.g., [3]–[8]), BBR does not passively react to packet loss or delay. Instead, it takes an initiative stance by sequentially probing bottleneck bandwidth and minimum round-trip time, and using those measurements to deliver at full bottleneck bandwidth without creating an excess queue in the pipe. BBR has been added into Linux network stack since mainline 4.9. According to Google [1], the deployment of BBR has brought about up to 133× bandwidth improvement in B4 [9] and more than 80% reduction of the median RTT in YouTube.

Attracted by its promising performance advantages, we deployed BBR in our geo-distributed cloud servers, yet run into significant, unexpected RTT fairness issues. Specifically, a flow with longer RTT always overwhelms those with shorter RTTs. To confirm that such a phenomenon is not due to the noisy environments in the Internet, we have reproduced the same problem in our in-house cluster, where two flows, one with RTT of 10 ms and the other with RTT of 50 ms, compete on a bottleneck link of 100 Mbps. As shown in Fig. 1, the 10-ms RTT flow, when running alone in the beginning, delivers at full bandwidth. However, later when the 50-ms RTT flow joins, the 10-ms RTT flow quickly gives up bandwidth and settles on 6.3 Mbps most of the time.

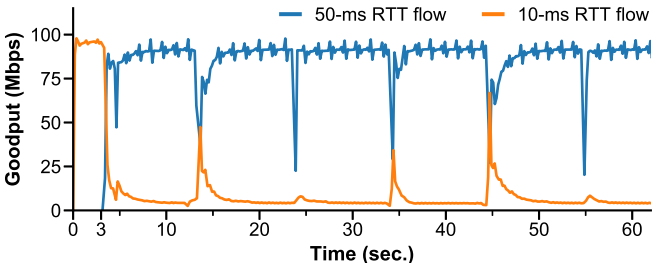

Fig. 1. The 50-ms RTT flow grabs most of the bandwidth. Excluding the startup phase, the 50-ms RTT flow has an average goodput of 87.3 Mbps, while the 10-ms RTT flow only has 6.3 Mbps.

This outcome is particularly surprising since traditional TCP congestion control algorithms, be it loss- or delay-based, all favor flows with shorter RTTs [5], [6], [10]–[12]. BBR, on the contrary, has a bias *against* them. Such a unique bias against flows with shorter RTTs has two serious consequences. First, it presents an unpleasant tradeoff between low latency and high delivery rate, unjustifying the decades of engineering efforts of bringing down end-to-end latency. For example, finding a route with the minimum RTT using the OSPF protocol would no longer be desirable, as flows along that route are easily overwhelmed when competing with others traversing along a suboptimal route with higher latency. Second, to make things worse, the advantage of a long RTT flow exposes a loophole which allows a strategic receiver to steal bandwidth from competing flows by artificially adding RTT latency to its inbound traffic.

Concerning about these consequences, in this paper, we seek to answer the following three questions:

1) How significant is the bias against short RTT? We performed comprehensive measurement studies in a clean network environment in an in-house cluster with 20 servers, and made the following three major observations.

First, BBR flows with shorter RTTs are always unfavored when competing against those with longer RTTs, irrespective of the bottleneck bandwidth, deployment of AQM strategies, RTT difference, and the number of competing flows.

Second, a small RTT disparity is sufficient to result in a

significant difference in throughput. In our experiments, a 10-ms RTT flow, when competing with a 15-ms RTT flow, ends up with $< 25\%$ of the bottleneck bandwidth. The larger the RTT difference is, the more salient the bandwidth disparity would be: when competing with a flow with RTT ≥ 30 ms, the 10-ms RTT flow is squeezed to $< 10\%$ of the bandwidth.

Third, the advantage of a long RTT flow scales. A 50-ms RTT flow occupies 50% of the bandwidth when competing with twenty 10-ms RTT flows.

2) What is the root cause of the bias? Through in-depth analysis of the behaviors of the two competing flows in Fig. 1, we show that the bias is introduced in two phases. (1) An excess queue forms on the bottleneck and grows quickly when BBR flows increase inflight to probe for more bandwidth. (2) A long RTT flow floods in a larger volume of excess traffic ($\texttt{inflight} - \texttt{BDP}$) than a short RTT flow, dominating the queue backlog as well as the delivery rate. The short RTT flow measures a lower delivery rate and slows down to match the measurement, making itself more of an underdog in the competition. Worse, the short RTT flow is susceptible to being CWND-bounded, thus unable to probe for more bandwidth.

3) How can we mitigate such a bias? Based on our answer to the previous question, we propose BBQ, a simple, yet effective solution that provides better RTT fairness without deviating from Kleinrock's optimal operating point. BBQ constantly detects the presence of an excess queue in the pipe. When a queue forms, BBQ prevents long RTT flows from pouring an overwhelming amount of excess traffic. To do so, BBQ enforces a small length of probing period to restrict flows from probing too long. Conversely, when the queue dissipates, BBQ switches to a longer probing period. This allows flows to quickly probe for the available bandwidth, ensuring high link utilization.

We implemented BBQ in Linux kernel 4.9.18 and evaluated its performance in our 20-machine cluster through comprehensive experiments. Evaluations show that compared with BBR, BBQ significantly improves RTT fairness, increasing the bandwidth share of a short RTT flow by up to $6.1\times$. Such a fairness improvement is achieved without compromising the full delivery bandwidth and low latency. Moreover, with BBQ, the average queueing delay is further reduced by 64.5%.

II. BBR: Congestion-Based Congestion Control

In this section, we give a brief introduction on how BBR works and what benefits it provides. For more details on the implementation, we refer the interested readers to Cardwell et al. [1].

A. The Optimal Operating Point

For a TCP connection, there exists one slowest link at a time, known as the *bottleneck*. The bottleneck determines the connection's maximum delivery rate and is the only place where persistent queues build up. Ideally, the connection should (1) send at a rate matching the bandwidth available at the bottleneck (denoted as BtlBw), and (2) maintain the amount of data *in flight* that matches exactly one bandwidth-delay product (BDP), i.e., $\texttt{inflight} = \texttt{BtlBw} \cdot \texttt{RTprop}$, where RTprop is the round-trip propagation time. Kleinrock proved that this operating point maximizes the connection's throughput (fills the pipe) while minimizing RTT (keeping queues empty), and is optimal for both individual connections and the network as a whole [2].

However, prevalent TCP congestion control algorithms do not converge to Kleinrock's optimal operating point. Most of these algorithms use packet loss as a congestion signal (notably Reno [3] and its successor CUBIC [6]), delivering at full pipe bandwidth at the cost of *bufferbloat* [13]. When the buffer is deep, which is commonly observed on the last-mile links of today's Internet, the resulting bufferbloat can easily cause queueing delay of seconds.

Converging to the optimal operating point has long been a challenging problem, because BtlBw and Rtprop cannot be measured at the same time. To measure BtlBw, the pipe must be overfilled, and persistent queues form; to measure RTprop, on the other hand, all queues must be drained empty.

B. BBR's Principles and Benefits

The recently proposed BBR is a ground-up redesign of congestion control algorithm [1]. BBR addresses the challenge of finding the optimal operating point by sequentially measuring a connection's maximum delivery rate and minimum RTT.

Principles. In a nutshell, BBR estimates BtlBw as the maximum delivery rate in recent 10 round trips and RTprop as the minimum RTT measured in the past 10 seconds. The max-filtered bandwidth (MaxBw) and the min-filtered RTT (MinRTT) precisely model the pipe.

Based on this model, BBR governs its sending behavior through two control parameters: pacing rate and congestion window (CWND). BBR cycles through different pacing rates to probe for more bandwidth (i.e., pacing 25% faster than MaxBw), drain off excess queues (i.e., pacing 25% slower than MaxBw), and cruise at the current MaxBw to fully utilize bandwidth. BBR also sets CWND as a small multiple of BDP ($2\times$ by default), so as to bound the volume of inflight data without creating long, persistent queues.

Behaviors. When a BBR flow connects, it starts by performing an exponential search for the bottleneck bandwidth by increasing its sending rate by a factor of $2/\ln 2$ while the delivery rate is growing. Once BtlBw has been detected, the flow transitions into the Drain mode, clearing up the excess queue during the search. Then BBR exits this exponential-growth startup and enters the steady state.

In the steady state, a BBR flow alternates between ProbeBw and ProbeRTT mode to dynamically characterize the pipe's BtlBw and RTprop, based on the recently measured delivery rates and RTTs. A long-lived BBR flow spends the vast majority of its time in ProbeBW, pacing at different rates to fully probe and utilize the pipe's available bandwidth, while maintaining a small, bounded queue. If a flow has been continuously sending, and has not seen an RTT measurement

that matches or reduces its MinRTT for a long time (10 seconds by default), it will briefly enter the ProbeRTT mode to cut the inflight to a very small value (set CWND to four packets) to re-probe the round-trip propagation delay.

Benefits. BBR has quickly attracted a wide range of attention since its publication, due to the following promising benefits:

- *Easy to deploy:* BBR is a sender-based congestion control. It requires no modification of switches, nor does it need support for special functionalities, such as ECN [12], RDMA [8] or per-flow AQM [14].
- *Near-optimal latency:* BBR runs with nearly empty queue most of the time. Furthermore, BBR at most uses one BDP's worth of queue by explicitly bounding its inflight.
- *High throughput:* BBR quickly saturates high capacity links despite the existence of random losses and link errors. In lossy network environments, BBR can significantly improve throughput compared with loss-based congestion control.

Encouraged by these promising benefits, we deployed BBR in our geo-distributed cloud servers to speed up bulk transfer, yet encountered unexpected bias towards long RTT flows. We next reproduce these problems in a more controlled manner.

III. Bias towards Long RTT

In this section, we study BBR's RTT fairness performance through several benchmarks in an in-house cluster. Our measurement shows that flows with long RTTs are bound to overwhelm those with shorter RTTs. We then discuss the practical concerns about this bias.

A. Methodology

In order to eliminate the uncontrollable interferences in the wild Internet (e.g., background traffic, specialized token-bucket policers, AQM deployment, etc.), we purposely performed the measurements on an in-house cluster of 20 blade servers. Each server has a Xeon E5-1410 8-thread 2.8 GHz CPU with 24 GB RAM, and is interconnected through 1 Gbps NIC. We deployed Linux 4.9.18 with the latest BBR kernel module across the cluster. The sch_fq module is loaded for flow pacing as required by BBR.

Fig. 2 shows the network topology of our cluster. Sender n connects to receiver n with a configurable RTT governed by NetEm [15]. All flows share the same bottleneck link. A powerful 20-port blade server running on Linux acts as the bottleneck switch for controllable bandwidth. The switch buffer size per port is by default 2 MB, which is a moderate number among mainstream products such as Cisco Nexus 3064X, Arista 7050S-64, Juniper QFX3500, etc.

To examine BBR's RTT fairness in different scenarios, we developed a dedicated framework to coordinate and synchronize the measurement performed on each server. Specifically, we used Netperf [16] to generate flows, and used the Linux tc tool to control packet delay, bottleneck bandwidth, and AQM deployment. Our framework has been optimized so that TCP Small Queues [17], a flow control enhancement in modern Linux kernel, will not interfere with NetEm [15] in tc.

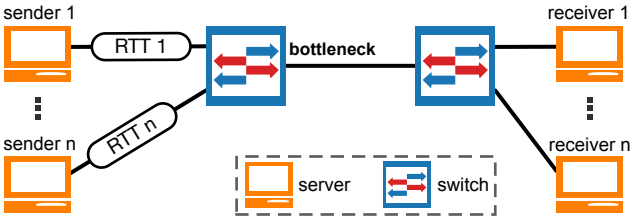

Fig. 2. The network topology of our in-house cluster.

B. Micro-benchmark

We are curious to know if the bias towards long RTT flows observed in Fig. 1 is an intrinsic problem of BBR, or merely an edge case occurring at some extreme operating parameters. To answer this question, we ran several micro-benchmarks under different network configurations. Unless otherwise stated, the RTT of a flow is configured to be either 10 ms or 50 ms; each flow lasts for 120 seconds for bulk transfer.

Bottleneck bandwidth. We show that BBR's bias towards long RTT persists across a wide range of bottleneck bandwidth. We gradually increase the bottleneck bandwidth from 10 Mbps to 1 Gbps. For each bandwidth setting, we initiated the same two flows as in the previous experiments, and measured their bandwidth share in the steady state. Fig. 3 depicts the results. While the bias against the short RTT flow is alleviated on low bandwidth bottleneck, the 10-ms RTT flow remains far below its fair share.

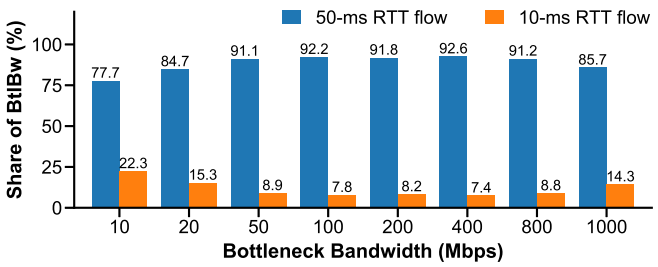

Fig. 3. Bandwidth share of two BBR flows competing on a bottleneck link of different capacities.

Deployment of AQM. Modern switches employ AQM (active queue management) schemes, notably RED [18] (random early detection), to alleviate the TCP collapse [19] problem. Fig. 4 shows that raising the drop probability or reducing the max-threshold (details in Table I) helps the short RTT flow to regain slightly more bandwidth share. However, if one tries to follow this trend and tune the parameters, an enormous amount of retransmissions will be incurred.

TABLE I
RED Parameters and Retransmissions

Scheme	Max	Min	Prob.	Retrans-50	Retrans-10
RED 1	0.50 MB	0.17 MB	2%	10178 pkts	2871 pkts
RED 2	0.50 MB	0.17 MB	10%	20583 pkts	6014 pkts
RED 3	0.33 MB	0.17 MB	2%	17193 pkts	5301 pkts

Disparity of RTT. In order to understand how bandwidth share changes with an increasing RTT disparity, we consider

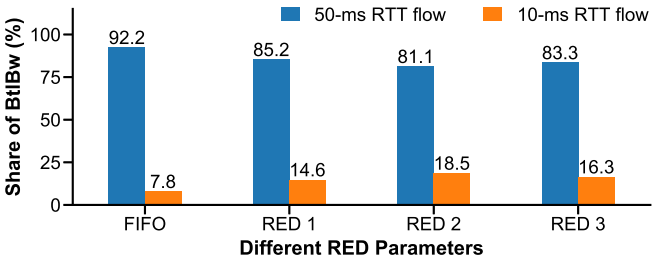

Fig. 4. Bandwidth share of two BBR flows competing on a bottleneck with different RED parameters (details in Table I).

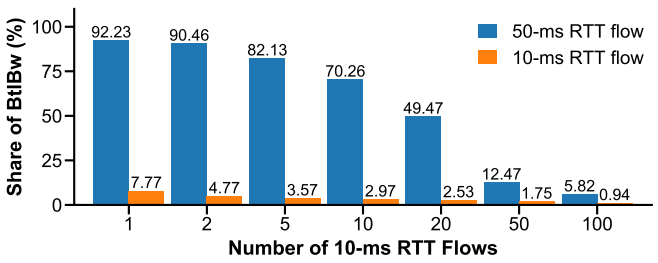

Fig. 6. Per-flow bandwidth share of flows with short and long RTTs. A 50-ms RTT flow competes with a varying number of 10-ms RTT flows.

the throughput of two competing BBR flows with short (flow A) and long (flow B) RTTs on a bottleneck link of 100 Mbps. Flow A has a fixed RTT of 10 ms; flow B has varying RTTs, ranging from 10 ms to 200 ms. This range of RTTs captures most LAN and long-haul connections. Fig. 5 shows the measured throughput of the two flows. When the two flows are of the same RTT, fairness is not a concern [1]. However, as the disparity of RTT increases, the bias towards long RTT becomes more pronounced. Specifically, it does not require a large RTT disparity to observe a salient throughput difference: The 15-ms RTT flow dominates the 10-ms RTT flow with 3× throughput.[1] This result is particularly disturbing. It suggests that a strategic receiver can steal bandwidth by artificially inflating its RTT. We shall discuss this point in detail in the next subsection.

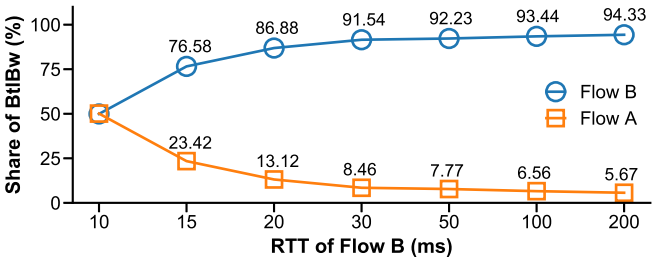

Fig. 5. The bandwidth disparity of two BBR flows becomes more salient as their RTT difference increases.

Number of competing flows. The number of competing flows is another critical factor affecting bandwidth share. To quantify its impact, we initiated a 50-ms RTT flow along with a varying number of 10-ms RTT flows. As shown in Fig. 6, the advantage of the long RTT flow diminishes quickly as the number of competing short RTT flows increases. This is because short RTT flows, even unfavored, can always grab some bandwidth in the end. Therefore, as their number surges, the long RTT flow would end up with less advantage. Nevertheless, the long RTT flow remains in favor with much more bandwidth than it deserves even when it is outnumbered by the 10-ms RTT competitors.

[1]Google has recently acknowledged the advantage of long RTT flows, but claimed that the problem is not of a significant concern of BBR [20]. We suspect that this is because Google's experiment was performed in a low-bandwidth environment (10 Mbps), where RTT bias is less severe (cf. Fig. 3).

Summary of findings. Our measurement confirms that the RTT fairness problem is intrinsic to BBR. Flows with long RTTs always have the upper hand. The longer the RTT is, the more bandwidth it will get.

C. Practical Concerns

BBR's bias towards long RTT flows is in stark contrast to the conventional wisdom of TCP congestion control. Traditional loss- and delay-based congestion control algorithms, such as Reno [3], CUBIC [6], DCTCP [21], and Vegas [7], all favor flows with short RTT [5], [10]–[12]. That BBR has a completely opposite bias *against* them raises two serious, practical concerns.

Unpleasant tradeoff. First, it enforces an unpleasant tradeoff between low latency and high delivery rate, which makes no sense in today's Internet. The networking community has tried for decades to reduce end-to-end latency. However, in the presence of the latency-bandwidth tradeoff, all those previous efforts will be unjustified. For instance, finding a route with the minimum RTT using routing algorithms such as OSPF and IS-IS could turn out undesirable, simply because flows along the shortest path are easily overwhelmed by others traversing long-haul.

Latency-cheating. Second, BBR's bias towards long RTTs can be easily manipulated by strategic receivers, who can steal bandwidth by artificially inflating its RTT (e.g., delaying inbound traffic). Because BBR is a sender-based congestion control, it would be very hard, if not impossible, for the sender to tell if the probed RTT has been manipulated by receivers.

The consequence of such "latency-cheating" can be more complicated. It does not appear to exist an equilibrium where all receivers are content about their bandwidth share. This can cause them to continuously game the network and run into a worse outcome.

To illustrate this "race-to-the-bottom" game, we initiated two competing BBR flows with the true RTT of 5 ms. The two flows alternately cheat. Each time a flow cheats, it delays the inbound traffic and inflates its RTT to 2× of the other. Fig. 7 shows the measured goodput of the two cheaters over time. Because a flow can always delay its traffic to steal more bandwidth from the other, the two keep doing so, causing the RTT growing exponentially.

We stress that the two concerns above are unique to BBR due to its congestion-based congestion control. The presence

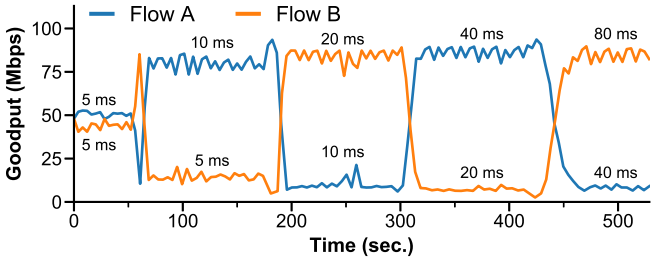

Fig. 7. Two flows with 5-ms true RTT alternately cheat on a 100-Mbps link. Each time a flow cheats, it inflates its RTT (annotated in the figure) to 2× of the other. The two flows race to the bottom with RTT growing exponentially.

of these concerns clouds the performance advantages of this promising new algorithm. As a first step to address this problem, we next analyze why BBR favors flows with long RTTs.

IV. The Anatomy of Bias

In this section, we analyze the root cause of BBR's bias against short RTT by diving deep into the behaviors of two competing flows in Fig. 1. We generalize our findings and show that the bias is developed through two phases.

A. Deep Dive

At a first glimpse, BBR's bias against short RTT flows appears counter-intuitive. BBR flows probe for more bandwidth once in every eight round trips [1]. When additional bandwidth becomes available, flows with shorter RTTs would respond to this environmental change more quickly, and hence have the upper hand to claim more available bandwidth in advance.

To understand why this first-mover advantage fails to sustain, we first refer back to Fig. 1 and focus on the steady-state goodput of the two flows. Fig. 8 provides a zoom-in view to illustrate more details. For the 50-ms RTT flow, when its `MinRTT` expires, it transitions into `ProbeRTT`: in order to learn the true `RTprop`, it reduces the inflight to four packets to drain the excess queue. As the queue dips and the bandwidth occupied by the 50-ms RTT flow yields, the 10-ms RTT flow detects a new `MaxBw` and reclaims all the available bandwidth. However, this situation does not last long. When the 50-ms RTT flow returns to `ProbeBw`, the 10-ms RTT flow is quickly overwhelmed.

Why does the short RTT flow yield the acquired bandwidth to the long RTT flow so easily? Our **first finding** attributes this outcome to *the persistent queue developed on the bottleneck.* Note that when the long RTT flow returns to `ProbeBw`, it resumes with its previous `MaxBw` (close to `BtlBw`). Because the short RTT flow has detected a much higher `MaxBw` when the long RTT flow was in `ProbeRTT`, the aggregated `MaxBw` of the two flows exceeds the bottleneck capacity, and thus an excess queue develops. Fig. 9 confirms this theory by tracking the RTT changes of the two flows over time. After the 50-ms RTT flow exits `ProbeRTT`, the RTTs of both flows explode until the surging inflight is bounded by `CWND`—a clear sign of a rapid-growing queue forming on the bottleneck.

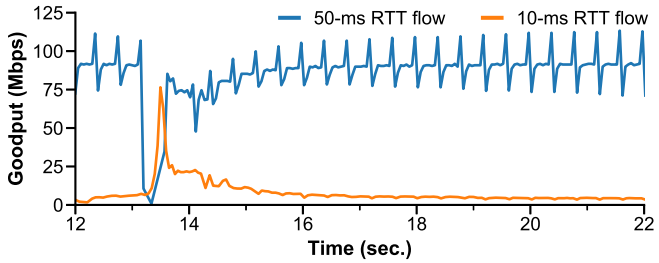

Fig. 8. Goodput of the two BBR flows in Fig. 1 from 12 to 22 sec.

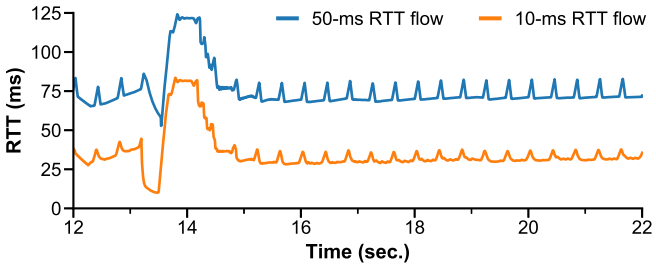

Fig. 9. RTT of the two BBR flows in Fig. 1 from 12 to 22 sec.

Interestingly, the development of the excess queue has drastically different impacts on the goodput of the two flows. Referring back to Fig. 8, for the short RTT flow, its goodput falls off the cliff as soon as the queue forms, whereas the goodput of the long RTT flow is decreased by only a small amount. Why does this happen? Queueing theory tells us that in the presence of a persistent queue, the bandwidth share of competing flows is determined by their backlog. This motivated us to measure the instantaneous queue backlog of the two flows by instrumenting the kernel functions on the bottleneck using `SystemTap` [22]. We see in Fig. 10 that the 10-ms RTT flow quickly occupies all queue slots when its competitor enters `ProbeRTT`, meaning that it has probed for more bandwidth. The 10-ms RTT flow then drains off queues and tries to cruise at the new bandwidth. However, when the 50-ms RTT flow returns to `ProbeBw`, the 10-ms RTT flow quickly recedes, and never finds a chance to come back.

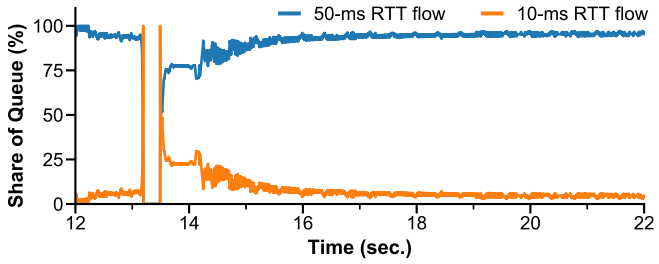

Fig. 10. Queue share of the two BBR flows in Fig. 1 from 12 to 22 sec.

We now reach our **second finding**: *as the queue develops, the backlog of the flow with shorter RTT is diminishing, so is the bandwidth share.* To explain this outcome, let us replay what has happened since the return of the 50-ms RTT flow from `ProbeRTT`. Both flows enter `ProbeBw`, probing for more bandwidth by periodically pouring slightly more inflight than

their own estimated BDP (1.25 BDP) into the pipe. In the beginning, because both flows detect roughly the same MaxBw (≈BtlBw), they send at the same rate, and the inflight−BDP excess (0.25 BDP) is proportional to the flow's MinRTT. Since the 50-ms RTT flow has a larger MinRTT, it deposits much more packets into the queue for a longer period than the 10-ms RTT flow. The increased queue share allows it to operate at a higher delivery rate than its competitor. Unable to measure a higher bandwidth, the 10-ms RTT flow quickly expires its MaxBw and replaces it with a smaller value. This, in turn, forces the 10-ms flow to send less data, which results in an even smaller queue share and a lower delivery rate. A positive feedback loop establishes and drives the 10-ms RTT flow to the bottom.

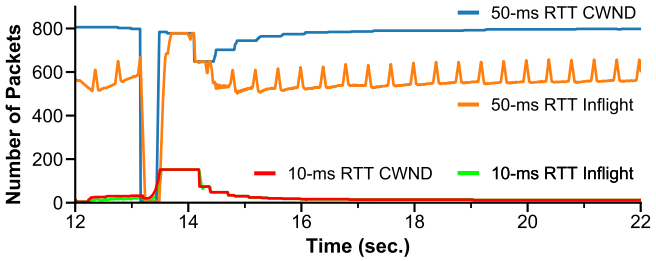

Fig. 11. CWND and inflight of the two BBR flows in Fig. 1 from 12 to 22 sec.

To make things worse, when the large flow returns to ProbeBw, the excess queue quickly builds up, and if the queue grows too fast, the inflight of the short RTT flow would be CWND-bounded (inflight=CWND) due to the long queueing delay. Meaning, the flow's sending rate is ACK clocked [3] and cannot be further increased. Being CWND-bounded essentially refrains the short RTT flow from injecting more packets into the pipe, speeding up the race to the bottom. Fig. 11 confirms this problem. The 10-ms RTT flow has its inflight CWND-bounded, whereas the 50-ms RTT flow can still inflate inflight (shown as the small spikes) to probe for more bandwidth.

B. The Development of Bias in Two Phases

Our findings based on the experiment of Fig. 1 highlight that the bias against short RTT is developed in two phases: (1) a persistent queue forms on the bottleneck, and (2) the backlog of the short RTT flow diminishes. We now generalize our findings to two competing flows with different RTTs and discuss the root cause behind. We limit our discussions to the period during which both flows stay in ProbeBw.

Phase 1. *When the two flows enter* ProbeBw*, a persistent queue forms on the bottleneck. The queue keeps growing until one flow is* CWND*-bounded.*

We explain how this happens through a step-by-step analysis of the behaviors of two competing flows A and B.

Step 1. To probe for more bandwidth, one flow, say flow-A, paces slightly faster than its BtlBw estimate (MaxBw), inflating its inflight to 1.25 BDP. An excess queue hence forms on the bottleneck, at least in one round trip.

Step 2. The increased inflight raises the queue share of flow-A,[2] which, in turn, increases its delivery rate. Flow-A measures a new maximum bandwidth MaxBw.

Step 3.1. Flow-A drains the excess queue after probing. It uses the newly updated MaxBw and reduces its inflight to a BDP higher than the previous estimate. Meaning, the excess queue due to probing cannot be drained empty.

Step 3.2. Flow-B maintains its sending rate and inflight as long as its MaxBw has not expired. The queue keeps growing until one flow expires its MaxBw, after which the aggregated sending rate falls to match the bottleneck capacity, and the queue sustains.

Repeating the entire process, we see that the queue continues growing after each probe-and-drain, until one flow is CWND-bounded and stops probing.

Phase 2. *As the queue develops, the backlog of the flow with shorter RTT diminishes, and the flow is overwhelmed.*

Once a persistent queue forms on the bottleneck, the throughput of a flow is determined by its queue share. Two factors come into play, respectively corresponding to BBR's two control parameters, pacing rate and CWND.

Factor 1 (Positive feedback loop)**.** *The flow with shorter RTT tends to contribute less queue backlog due to a smaller BDP estimate. This reduces its queue share and triggers a positive feedback loop with even less share.*

We give an intuitive explanation. A BBR flow periodically paces faster than the current BtlBw estimate to probe for more bandwidth, depositing 0.25 BDP worth of excess traffic to the queue. Intuitively, a flow with shorter RTT has a smaller BDP estimate than its competitor, and injects much less excess traffic into the pipe. This drives its queue share down, followed by a lower delivery rate. The persistently low delivery rate tricks the flow to lower the BtlBw estimate and sends at a lower rate, which further reduces its queue share, triggering a positive feedback loop.

Factor 2 (CWND-bounded inflight)**.** *To make things worse, as the queue develops, the flow with shorter RTT tends to be* CWND*-bounded, and is unable to probe for more bandwidth.*

We explain how this happens in two steps.

Step 1. The flow with shorter RTT enters the CWND-bounded mode before its competitor. According to BBR [1], a flow is CWND-bounded if and only if the queueing delay is greater than the minRTT, i.e.,

$$\texttt{inflight} = \texttt{Bw} \cdot \texttt{RTT} > 2 \cdot \texttt{MaxBw} \cdot \texttt{MinRTT}. \quad (1)$$

Because the two flows share the same bottleneck queue, as the queue develops, the queueing delay keeps increasing and will exceed a smaller MinRTT first. Meaning, a flow with shorter RTT is CWND-bounded first.

Step 2. Once entering the CWND-bounded mode, the flow will stay in it throughout ProbeBw. This is because as the queue

[2]Flow-B is likely not probing at the same time and maintains its current backlog in the queue, if any.

develops, the CWND-bounded flow measures increasing RTT. It hence has no chance to find a way out as its queueing delay is always greater than its MinRTT.

Now that the short RTT flow is CWND-bounded, it loses control of its sending rate which is ACK clocked [3] (matching the delivery rate). Since then, the queue stops growing, and the queueing delay remains unchanged. This suggests that the long RTT flow will never be CWND-bounded (its MinRTT is always greater than the queueing delay).

V. Improving RTT Fairness using BBQ

In this section, we close the gap between flows with different RTTs using a BBR improvement algorithm. Our design goal is to *provide better RTT fairness than BBR without deviating from Kleinrock's optimal operating point*, i.e., maximizing delivery rate while minimizing latency.

Our previous discussions reveal that the source of bias against short RTT flows originates from the rapid growth of a persistent queue when flows are probing for more bandwidth. Therefore, preventing a queue from building up too fast in ProbeBw is the key to achieving better RTT fairness.

As a first-cut solution, we consider a simple approach that completely gets rid of the queue once it forms after the probing, before it causes any damage.

A. Too Late to Drain after Probing

BBR already drains queues after probing. However, this is performed only in best efforts, and the queue may not be drained empty. Specifically, in the ProbeBw mode, BBR has a drain period following the probing period, during which a flow paces slower than its MaxBw, keeping the inflight to 0.75 BDP estimate (setting pacing gain to 0.75). The drain period ends when the inflight falls below one BDP (no excess queue), *or* the period has spanned one MinRTT, whichever comes first [23]. If it is the latter that comes first, the inflight − BDP excess is not fully cleared up.

Instead of trying best-effort, our implementation forces each flow to drain inflight to *exactly one* up-to-date BDP estimate. This way, the flow leaves no backlog in the queue, so that long RTT flows cannot squeeze out short ones.

Contrary to our expectation, such a drain-after-probing approach is of little help in improving RTT fairness. Applying this approach to the previous experiment in Fig. 1, we found that the short RTT flow ends up with even lower bandwidth share than that using BBR. Curious about why this happened, we dig into the pacing cycles of the two flows. As shown in Fig. 12, the 10-ms RTT flow spends the majority of its time in the drain periods, and when it proceeds to cruising (setting pacing gain to 1), its MaxBw has long expired and is replaced with a smaller estimate. This forces the short RTT flow to send at a lower rate, hence triggering a positive feedback loop.

Why is the 10-ms RTT flow trapped in the draining period for so long? In general, with a pacing gain of 0.75, each flow can at best drain inflight to $0.75 \cdot$MaxBw$\cdot$(MinRTT + Queueing). Therefore, if the queueing delay is persistently larger than $\frac{1}{3}$MinRTT, the flow is unable to drain its inflight to one BDP. As

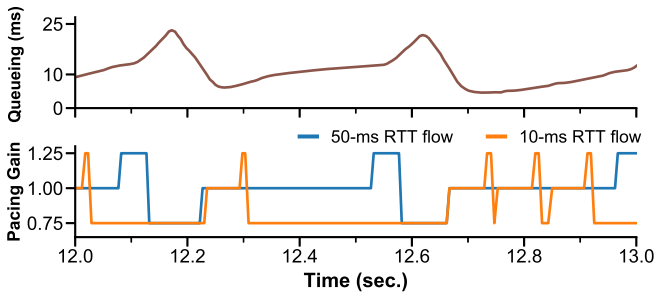

Fig. 12. The queueing delay and gain cycling of a 50-ms RTT flow and a 10-ms RTT flow under the first-cut solution.

illustrated in Fig. 12, while the 10-ms RTT flow keeps trying to drain its inflight, the queueing delay is still increasing due to the long RTT flow probing at the same time. The 10-ms RTT flow is then trapped in the drain periods until its competitor finishes probing and starts to drain.

To summarize, once a long, persistent queue has formed, it will be too late to get rid of it.

B. BBQ

The failure of the drain-after-probing approach prompts us that the algorithm must react to the queue development early on. Specifically, the algorithm should constantly detect if an excess queue is forming on the bottleneck. When it happens, the algorithm quickly intervenes, refraining long RTT flows from pouring too much excess traffic into the pipe, so as to protect short RTT flows from being squeezed out.

Our solution, which we call BBQ, follows exactly this intuition. We start to focus on how BBQ regulates excess traffic poured into the pipe. We then discuss how BBQ detects the presence of a persistent queue.

Regulating excess traffic. In order to probe for more bandwidth, a BBR flow periodically pours excess traffic into the pipe when it comes to the probing period. A probing period spans one MinRTT, during which the flow paces 25% faster (sets pacing gain 1.25) than the measured MaxBw. Because a long RTT flow probes for a longer time (cf. Fig. 12), it pours more excess traffic into the pipe and thus dominates short RTT flows.

To eliminate this advantage for long RTT flows, we impose a *cap* to the span of a probing period. Instead of probing for MinRTT time, our solution, BBQ, employs the following length for the probing period:

$$\rho = \min\{\texttt{MinRTT}, \alpha\},$$

where α is a parameter that caps the probing period. Imposing a cap to the probing period enables BBQ to regulate excess traffic, improving RTT fairness from two perspectives.

First, a long RTT flow with MinRTT $> \alpha$ now probes for at most α time and will not flood in an overwhelming amount of excess inflight to edge out the coexisting short RTT flows. With a small α (e.g., 3 ms), flows probe for the same period of time, making nearly equal contributions to the bottleneck queue with fair bandwidth share (details in Sec. VI).

Second, regulating excess traffic prevents a queue from growing too fast or too long. The reduced queueing delay lowers the chance of having a short RTT flow CWND-bounded.

We stress that the cap is imposed to a probing period only, while the length of the other periods (draining and cruising) remains unchanged. Similar to BBR, BBQ never over-drains inflight, and stops draining at one BDP or when the period has span one MinRTT, whichever comes first. BBQ hence delivers at the maximum bandwidth with better RTT fairness than BBR.

Regulating the excess traffic by capping the probing period is not without its downside. In general, it slows down the probing for more bandwidth. In case that more bandwidth becomes available (e.g., a route update or the departure of a competing flow), flows cannot quickly ramp up and may take a long time to saturate the bottleneck.

BBQ addresses this problem by judiciously capping the probing period only when the bottleneck is fully utilized with a persistent queue. In the absence of a persistent queue, BBQ simply reduces to BBR, allowing flows to quickly probe for more bandwidth. This requires BBQ to constantly detect the presence of a non-zero queue.

Queue detection. Existing works [7], [8] interpret a negative delay gradient measured through RTT signals ($\frac{\mathrm{d}\,q(t)}{\mathrm{d}\,t} = \frac{\mathrm{d}\,\mathtt{RTT}}{\mathrm{d}\,t}$) as an indicator for the decreasing contention on the bottleneck, provided that a non-zero queue exists in the network. However, RTT gradient cannot be used as an evidence for the presence/absence of a persistent queue here because the pacing gain cycling in BBR incurs frequent queue size changes. In this regard, we turn to a direct RTT measurement instead. Specifically, BBQ considers the pipe underutilized if the RTT measurement drops below a threshold:

$$\mathtt{RTT} < (1+\beta)\mathtt{MinRTT},$$

where β is a slack factor used in account of inaccurate measurement or unstable link states. BBQ reduces to BBR when the pipe is underutilized.

Summary of the algorithm. BBQ flows keep detecting the presence of a persistent queue through RTT measurements. When the pipe is full, BBQ flows probe for approximately the same period of time α, so as to have an equal share of the queue backlog. When the pipe is underutilized, BBQ simply reduces to BBR. This way, flows can quickly probe for the available bandwidth and utilize it as fast as possible.

C. Guidelines for Choosing Parameters

Cap for probing period. Having a large cap α for a probing period is less beneficial to improving RTT fairness. For example, if we choose $\alpha = 15$ ms for a 10-ms RTT flow coexisting with a 50-ms RTT flow, the former can at most probe for 10 ms, whereas the latter can probe for 15 ms. The 50-ms RTT flow still has the upper hand, though less advantageous compared with BBR. In general, to optimize RTT fairness, it is desirable to have α smaller than the typical propagation delay of Internet connections. On the other hand, having a too small α results in a slow convergence to the stable bandwidth share. We recommend $\alpha = 3$ ms and will show in Sec. VI-C that such a choice fits most LAN and long-haul connections with propagation delays ranging from 5 ms to 300 ms.

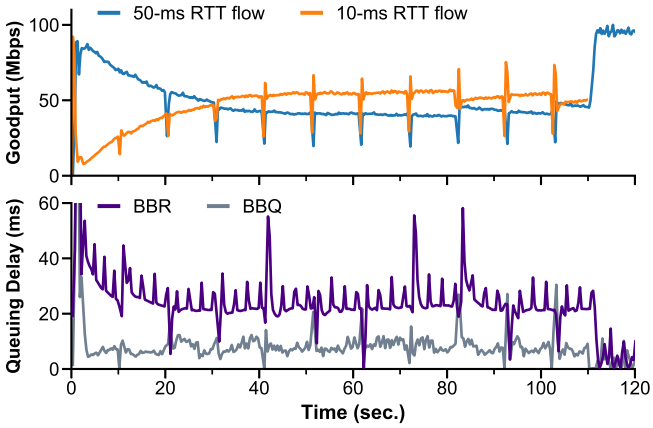

Fig. 13. The 10-ms RTT flow has a goodput of 51.4 Mbps in steady state, and the 50-ms RTT flow has 42.5 Mbps. After the 10-ms RTT flow leaves, the 50-ms RTT flow promptly saturates the link. Besides, BBQ flows experience a smaller queueing delay compared with BBR.

Slack factor. The slack factor β must be small enough to avoid false detection of pipe under-utilization, but meanwhile not too small so that high link utilization is ensured. For wired networks, we recommend $\beta \in [0.5\%, 1\%]$.

D. Limitations

Similar to BBR, BBQ employs a constant CWND gain to bound the inflight of each flow to 2 BDP. A short RTT flow hence has a smaller CWND than a long RTT flow, provided that both flows have the same estimation of the bottleneck bandwidth. This suggests that a short RTT flow will be CWND-bounded earlier, and will gradually yield bandwidth to the long RTT flow. Can we address this problem by raising the CWND bound to a larger multiple of BDP? In our experiments, by setting CWND to 4 BDP, we did observe a higher bandwidth share for short RTT flows, yet at a price of a significantly longer queueing delay. For this reason, BBQ chooses to prioritize low latency by retaining CWND=2 BDP. How to adaptively adjust the CWND gain to navigate the tradeoff between RTT fairness and queueing delay is left for future work.

VI. Evaluation

We have implemented BBQ in Linux kernel 4.9.18 and evaluated its performance in the same cluster environment as described in Sec. III. Highlights of our evaluations are summarized as follows.

- BBQ delivers at full bandwidth with near-optimal latency. Compared with BBR, BBQ reduces the queueing delay by 64.5% when a 10-ms RTT flow is competing with a 50-ms RTT flow (Sec. VI-A).
- BBQ consistently outperforms BBR with better RTT fairness, improving the bandwidth share of a short flow by up to 6.1× (Sec. VI-B).

- BBQ's performance advantage extends to a wide range of RTTs, from 5 ms to 300 ms (Sec. VI-C).

A. Micro-benchmark

We demonstrate the advantage of BBQ through a micro-benchmark in Fig. 13, where two flows, one with 10-ms RTT and the other with 50-ms RTT, compete on a bottleneck link of 100 Mbps. In the startup mode, while the 10-ms RTT flow occupies the full bandwidth first, it quickly yields to the 50-ms RTT flow, as the latter probes more aggressively during the startup. Later in the steady state, both flows enters `ProbeBw`. The short flow comes back and settles on 51.4 Mbps after the synchronization of `ProbeRTT` at 31 s, while the 50-ms RTT flow stabilizes at around 42.5 Mbps. When the 10-ms RTT flow departs at 110 s, BBQ instantly detects this and reduces to BBR. The 50-ms RTT flow quickly ramps up and takes the remaining bandwidth in just 1.7 seconds.

In addition to the significantly improved RTT fairness and high throughput, BBQ outperforms BBR with a lower delay. As shown in the lower graph of Fig. 13, BBQ reduces the average queueing delay to 8.3 ms, as opposed to 23.4 ms when using BBR, a 65% reduction on average.

B. Impact of Network Environment

For comparison purpose, we evaluate BBQ's RTT fairness performance using the same benchmarks as in Sec. III. Unless otherwise stated, we let a 10-ms RTT flow compete with a 50-ms RTT flow on a bottleneck link of 100 Mbps. We measure RTT fairness of a due algorithm (BBQ and BBR) using the bandwidth share received by the short RTT flow.

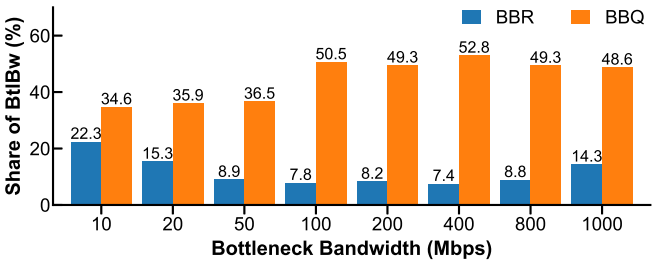

Fig. 14. BBQ improves the bandwidth share of the 10-ms RTT flow by 55.2% to 6.14× compared with BBR under different bottleneck capacities.

Bottleneck bandwidth. Fig. 14 compares the bandwidth share a short RTT flow receives using BBQ and BBR with bottleneck bandwidth spanning two orders of magnitude. In all cases, BBQ outperforms BBR with better RTT fairness, improving the bandwidth share of the 10-ms RTT flow by at least 55.2% (10 Mbps) and by up to 6.1× (400 Mbps). BBQ provides near-optimal RTT fairness for high-bandwidth link (≥ 100 Mbps), where the two flows equally share the bottleneck.

Disparity of RTT. In order to confirm that BBQ's improvement in fairness extends to different levels of RTT disparity, we let a 10-ms RTT flow compete with a flow with varying RTTs, ranging from 10 ms to 200 ms. Fig. 15 compares the bandwidth share of the 10-ms RTT flow using BBQ and BBR.

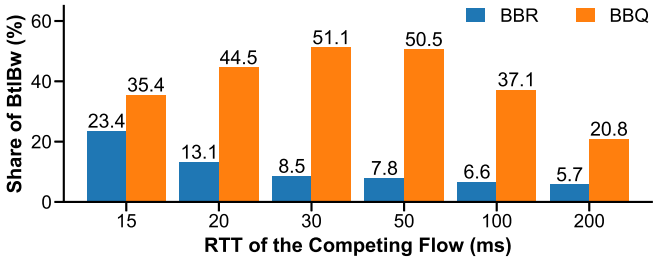

Fig. 15. BBQ improves the bandwidth share of the 10-ms RTT flow by up to 4.6× when competing with a flow with varying RTTs.

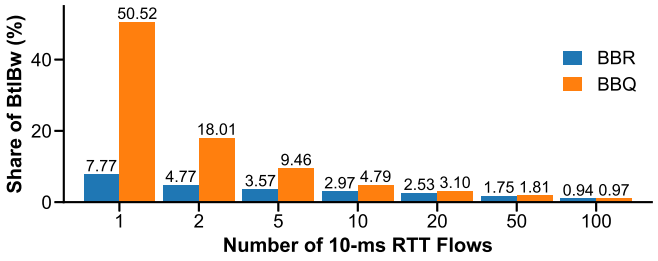

Fig. 16. The per-flow bandwidth share of each 10-ms RTT flow with different number of 10-ms RTT flows and one 50-ms RTT flow.

We see that for BBR flows, a larger RTT difference widens their throughput disparity. In comparison, BBQ successfully constrains this trend. Specifically, even when competing with a flow with 10× RTT (100ms), the 10-ms RTT flow can still retain 37.1% of the bottleneck bandwidth—a 4.62× fairness improvement over BBR.

Number of competing flows. We next evaluate if BBQ's improvement in fairness scales to more flows. We let a varying number of 10-ms RTT flows compete with one 50-ms RTT flow. Fig. 16 compares the bandwidth share of each 10-ms RTT flows using BBQ and BBR. As expected, in all cases, each 10-ms RTT flow improves its bandwidth share using BBQ. However, as their number increases, the benefits provided by BBQ become less salient, for two reasons. On one hand, the advantage of the 50-ms RTT flow diminishes when it is outnumbered by the 10-ms RTT competitors. On the other hand, with a growing number of competing flows, the queueing delay surges, and the 10-ms RTT flow are more likely `CWND`-bounded.

C. A One-Size-Fits-All Cap

We have recommended a small cap $\alpha = 3$ ms for a probing period in Sec. V-C. We now show that such a one-size-fits-all parameter works well for most Internet connections. We evaluate the situation where two flows with different RTTs, ranging from 5 ms to 300 ms, compete on a bottleneck link of 100 Mbps. Fig. 17 compares the bandwidth share of the flow with shorter RTT using BBQ and BBR. In all cases, by setting $\alpha = 3$ ms, BBQ consistently results in a fairer allocation, improving the bandwidth share of the short flow by at least 84.12% (200-300 ms) and by up to 4.4× (20-50 ms) as compared with BBR.

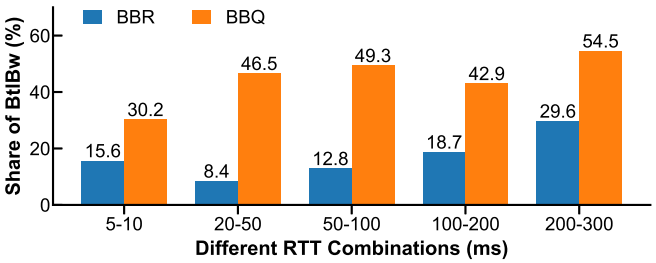

Fig. 17. A small α (3 ms) is a one-size-fits-all choice for typical Internet connections with RTTs ranging from 5 ms to 300 ms. The group is named after the RTTs of the two competing flows.

VII. Related Work

Loss-based congestion control favors short RTT flows [10], [11]. Flows using the traditional AIMD algorithm [3], [4], [12] increase their window size by one packet per RTT. As a result, short RTT flows grab bandwidth more quickly and settle on a higher CWND in the steady state. By making the window control function independent of RTT, CUBIC [5], [6] achieves linear RTT fairness, meaning, the throughput achieved by a TCP flow is inversely proportional to its RTT. Delay-based congestion control [7], [8], [24] uses network latency as a signal of congestion. Those algorithms also favor short RTT flows, because the increase of window size [7] or the adjustment of transmission rate [8] is still inversely proportional to RTT.

BBR [1] is a new type of congestion control that proactively models a TCP connection, and paces its rate based on the estimate of the maximum bottleneck bandwidth and minimum RTT. Unlike loss- and delay-based congestion control, BBR presents an opposite bias *against* short RTT flows. To our knowledge, this paper is the first to identify and quantify such a severe RTT fairness problem for BBR. Our in-depth analysis reveals the root cause of BBR's bias against short RTT flows, based on which we propose our solution, BBQ.

The RTT fairness problem can also be addressed by deploying fair queueing algorithms [25] in switches, provided that the per-flow queues are available [14]. Unfortunately, switches in the Internet usually lack support for such fine-grained functionalities. Moreover, it is expensive to constantly tweak the fairness rules on all the switches to adapt to dynamic traffic. BBQ is a pure end-based solution, and can work with AQM for more diversified performance requirements.

VIII. Conclusion

In this paper, we systematically analyzed the extent and cause of BBR's RTT fairness problem through extensive measurement studies. We confirmed that a BBR flow with longer RTT dominates a flow with shorter RTT, irrespective of network configurations. We showed, through a deep dive into the flow behaviors, that such a bias is introduced in two phases: (1) when BBR flows probe for more bandwidth, a persistent queue forms on the bottleneck due to the excess traffic; (2) the long RTT flow probes for longer time than the short RTT flow, and hence floods in more excess traffic, dominating the queue backlog while overwhelming the short RTT flow. Based on these findings, we designed and implemented BBQ that can provide significantly better RTT fairness without compromising full delivery rate or low latency. BBQ constantly detects an excess queue, and when a queue forms, it enforces a short period of time for probing, so as to refrain long RTT flows from pouring too much excess traffics into the pipe. Evaluation shows that BBQ outperforms BBR with respect to RTT fairness by up to 6.1×, and achieves the same link utilization with even shorter end-to-end latency.

References

[1] N. Cardwell, Y. Cheng, C. S. Gunn, S. H. Yeganeh *et al.*, "BBR: congestion-based congestion control," *Commun. ACM*, vol. 60, no. 2, pp. 58–66, 2017.

[2] L. Kleinrock, "Power and deterministic rules of thumb for probabilistic problems in computer communications," in *IEEE ICC*, 1979, pp. 43.1.1–43.1.10.

[3] V. Jacobson, "Congestion avoidance and control," in *ACM SIGCOMM*, 1988, pp. 314–329.

[4] S. Floyd and T. Henderson, "The NewReno Modification to TCP's Fast Recovery Algorithm," https://tools.ietf.org/html/rfc2582, 1999.

[5] L. Xu, K. Harfoush, and I. Rhee, "Binary increase congestion control (BIC) for fast long-distance networks," in *IEEE INFOCOM*, 2004, pp. 2514–2524.

[6] S. Ha, I. Rhee, and L. Xu, "Cubic: a new tcp-friendly high-speed tcp variant," *ACM SIGOPS Oper. Syst. Rev.*, vol. 42, no. 5, pp. 64–74, 2008.

[7] L. S. Brakmo and L. L. Peterson, "TCP Vegas: End to end congestion avoidance on a global Internet," *IEEE J. Selected Areas in Commun.*, vol. 13, no. 8, pp. 1465–1480, 1995.

[8] R. Mittal, V. T. Lam, N. Dukkipati, E. Blem, H. Wassel, M. Ghobadi, A. Vahdat, Y. Wang, D. Wetherall, and D. Zats, "TIMELY: RTT-based Congestion Control for the Datacenter," *ACM SIGCOMM Computer Communication Review*, vol. 45, no. 4, pp. 537–550, 2015.

[9] S. Jain, A. Kumar, S. Mandal, J. Ong, L. Poutievski, A. Singh, S. Venkata, J. Wanderer, J. Zhou, M. Zhu *et al.*, "B4: Experience with a globally-deployed software defined wan," *ACM SIGCOMM Comput. Commun. Rev.*, vol. 43, no. 4, pp. 3–14, 2013.

[10] T. V. Lakshman and U. Madhow, "The performance of TCP/IP for networks with high bandwidth-delay products and random loss," *IEEE/ACM Trans. Netw.*, vol. 5, no. 3, pp. 336–350, 1997.

[11] P. Brown, "Resource sharing of TCP connections with different round trip times," in *IEEE INFOCOM*, vol. 3, 2000, pp. 1734–1741.

[12] M. Alizadeh, A. Javanmard, and B. Prabhakar, "Analysis of DCTCP: Stability, Convergence, and Fairness," in *ACM SIGMETRICS*, 2011, pp. 73–84.

[13] J. Gettys and K. Nichols, "Bufferbloat: dark buffers in the Internet," *ACM Queue*, vol. 9, no. 11, 2011.

[14] K. Nichols and V. Jacobson, "Controlling Queue Delay: A modern AQM is just one piece of the solution to bufferbloat." *ACM Queue*, vol. 10, no. 5, 2012.

[15] Netem. http://man7.org/linux/man-pages/man8/tc-netem.8.html.

[16] Netperf. http://www.netperf.org/netperf/.

[17] TCP small queue. https://lwn.net/Articles/506237/.

[18] S. Floyd and V. Jacobson, "Random early detection gateways for congestion avoidance," *IEEE/ACM Trans. Netw.*, vol. 1, no. 4, pp. 397–413, 1993.

[19] M. Allman, V. Paxson, and E. Blanton, "TCP congestion control," Tech. Rep., 2009.

[20] N. Cardwell, Y. Cheng, C. S. Gunn, S. H. Yeganeh, and V. Jacobson, "BBR congestion control," https://www.ietf.org/proceedings/97/slides/slides-97-iccrg-bbr-congestion-control-02.pd

[21] M. Alizadeh, A. Greenberg, D. A. Maltz, J. Padhye, P. Patel, B. Prabhakar, S. Sengupta, and M. Sridharan, "Data Center TCP (DCTCP)," in *ACM SIGCOMM*, 2010, pp. 63–74.

[22] "SystemTap," https://sourceware.org/systemtap/.

[23] "BBR congestion control algorithm," https://lwn.net/Articles/701149/.

[24] K. Tan, J. Song, Q. Zhang, and M. Sridharan, "A Compound TCP approach for high-speed and long distance networks," in *IEEE INFOCOM*, 2006, pp. 1–12.
[25] M. Shreedhar and G. Varghese, "Efficient fair queuing using deficit round-robin," *IEEE/ACM Trans. Netw.*, vol. 4, no. 3, pp. 375–385, 1996.