acmqueue

December 1, 2016

Volume 14, issue 5

PDF

BBR: Congestion-Based Congestion Control

Measuring bottleneck bandwidth and round-trip propagation time

Neal Cardwell, Yuchung Cheng, C. Stephen Gunn, Soheil Hassas Yeganeh, Van Jacobson

By all accounts, today's Internet is not moving data as well as it should. Most of the world's cellular users experience delays of seconds to minutes; public Wi-Fi in airports and conference venues is often worse. Physics and climate researchers need to exchange petabytes of data with global collaborators but find their carefully engineered multi-Gbps infrastructure often delivers at only a few Mbps over intercontinental distances.[6]

These problems result from a design choice made when TCP congestion control was created in the 1980s—interpreting packet loss as "congestion."[13] This equivalence was true at the time but was because of technology limitations, not first principles. As NICs (network interface controllers) evolved from Mbps to Gbps and memory chips from KB to GB, the relationship between packet loss and congestion became more tenuous.

Today TCP's loss-based congestion control—even with the current best of breed, CUBIC[11]—is the primary cause of these problems. When bottleneck buffers are large, loss-based congestion control keeps them full, causing bufferbloat. When bottleneck buffers are small, loss-based congestion control misinterprets loss as a signal of congestion, leading to low throughput. Fixing these problems requires an alternative to loss-based congestion control. Finding this alternative requires an understanding of where and how network congestion originates.

Congestion and Bottlenecks

At any time, a (full-duplex) TCP connection has exactly one slowest link or *bottleneck* in each direction. The bottleneck is important because:

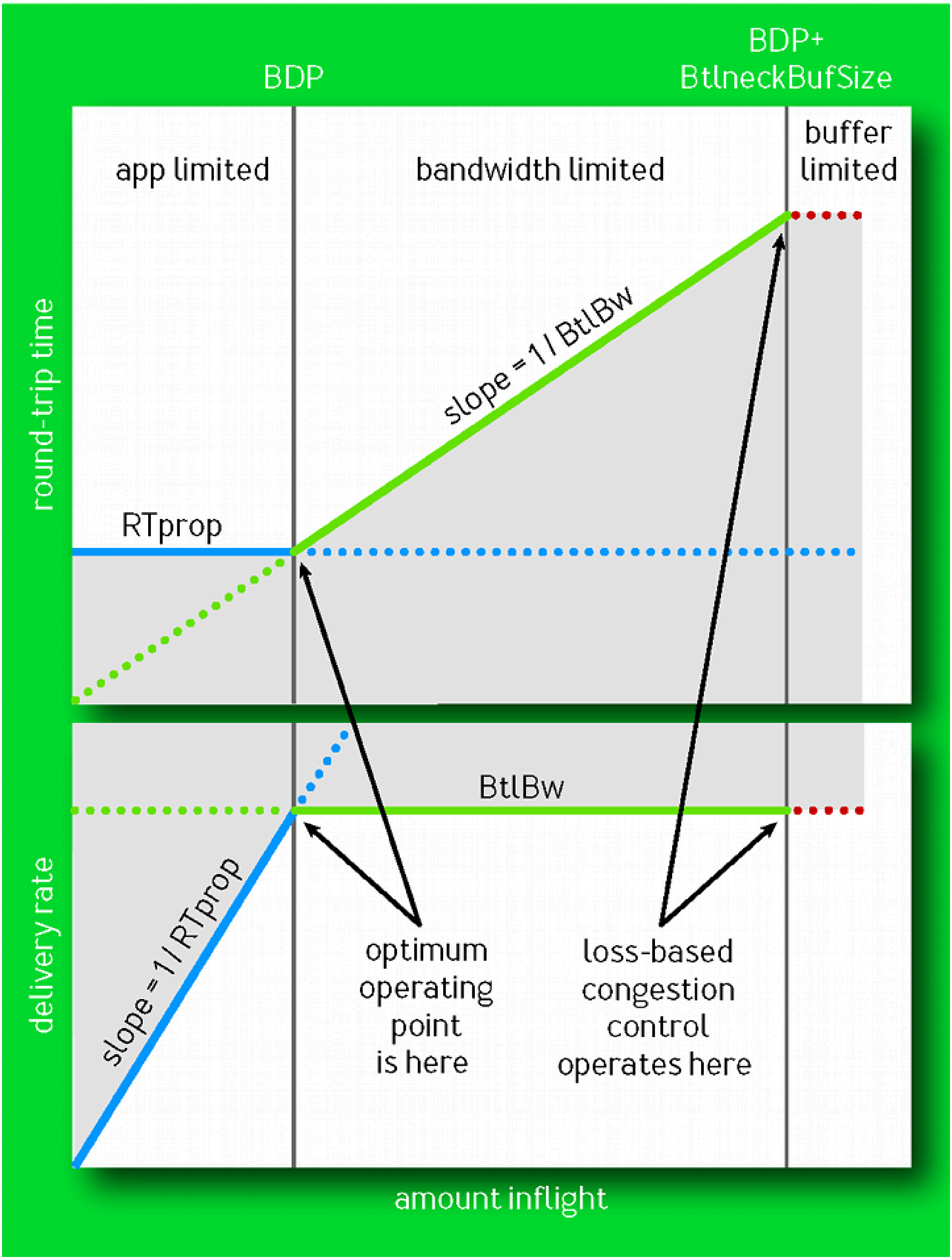
- It determines the connection's maximum data-delivery rate. This is a general property of incompressible flow (e.g., picture a six-lane freeway at rush hour where an accident has reduced one short section to a single lane. The traffic upstream of the accident moves no faster than the traffic through that lane).

- It's where persistent queues form. Queues shrink only when a link's departure rate exceeds its arrival rate. For a connection running at maximum delivery rate, all links upstream of the bottleneck have a faster departure rate so their queues migrate to the bottleneck.

Regardless of how many links a connection traverses or what their individual speeds are, from TCP's viewpoint an arbitrarily complex path behaves as a single link with the same RTT (round-trip time) and bottleneck rate. Two physical constraints, *RTprop* (round-trip propagation time) and *BtlBw* (bottleneck bandwidth), bound transport performance. (If the network path were a physical pipe, RTprop would be its length and BtlBw its minimum diameter.)

Figure 1 shows RTT and delivery rate variation with the amount of data *in flight* (data sent but not yet acknowledged). Blue lines show the RTprop constraint, green lines the BtlBw constraint, and red lines the bottleneck buffer. Operation in the shaded regions isn't possible since it would violate at least one constraint. Transitions between constraints result in three different regions (app-limited, bandwidth-limited, and buffer-limited) with qualitatively different behavior.

FIGURE 1: **DELIVERY RATE AND ROUND-TRIP TIME VS. INFLIGHT**



When there isn't enough data in flight to fill the pipe, RTprop determines behavior; otherwise, BtlBw dominates. Constraint lines intersect at $inflight = BtlBw \times RTprop$, a.k.a. the pipe's BDP (bandwidth-delay product). Since the pipe is full past this point, the *inflight - BDP* excess creates a queue at the bottleneck, which results in the linear dependence of RTT on inflight data shown in the upper graph. Packets are dropped when the excess exceeds the buffer capacity. *Congestion* is just sustained operation to the right of the BDP line, and *congestion control* is some scheme to bound how far to the right a connection operates on average.

Loss-based congestion control operates at the right edge of the bandwidth-limited region, delivering full bottleneck bandwidth at the cost of high delay and frequent packet loss. When memory was expensive buffer sizes were only slightly larger than the BDP, which minimized loss-based congestion control's excess delay. Subsequent memory price decreases resulted in buffers orders of magnitude larger than ISP link BDPs, and the resulting bufferbloat yielded RTTs of seconds instead of milliseconds.[9]

The left edge of the bandwidth-limited region is a better operating point than the right. In 1979 Leonard Kleinrock[16] showed this operating point was optimal, maximizing delivered bandwidth while minimizing delay and loss, both for individual connections and for the network as a whole[8]. Unfortunately, around the same time Jeffrey M. Jaffe[14] proved it was impossible to create a distributed algorithm that converged to this operating point. This result changed the direction of research from finding a distributed algorithm that achieved Kleinrock's optimal operating point to investigating different approaches to congestion control.

Our group at Google spends hours each day examining TCP packet header captures from all over the world, making sense of behavior anomalies and pathologies. Our usual first step is finding the essential path characteristics, RTprop and BtlBw. That these can be inferred from traces suggests that Jaffe's result might not be as limiting as it once appeared. His result rests on fundamental measurement ambiguities (e.g., whether a measured RTT increase is caused by a path-length change, bottleneck bandwidth decrease, or queuing delay increase from another connection's traffic). Although it is impossible to disambiguate any single measurement, a connection's behavior over time tells a clearer story, suggesting the possibility of measurement strategies designed to resolve ambiguity.

Combining these measurements with a robust servo loop using recent control systems advances[12] could result in a distributed congestion-control protocol that reacts to actual congestion, not packet loss or transient queue delay, and converges with high probability to Kleinrock's optimal operating point. Thus began our three-year quest to create a congestion control based on measuring the two parameters that characterize a path: bottleneck bandwidth and round-trip propagation time, or BBR.

Characterizing the Bottleneck

A connection runs with the highest throughput and lowest delay when (rate balance) the bottleneck packet arrival rate equals BtlBw and (full pipe) the total data in flight is equal to the BDP ($= BtlBw \times RTprop$).

The first condition guarantees that the bottleneck can run at 100 percent utilization. The second guarantees there is enough data to prevent bottleneck starvation but not overfill the pipe. The rate balance condition alone *does not* ensure there is no queue, only that it can't change size (e.g., if a connection starts by sending its 10-packet Initial Window into a five-packet BDP, then runs at exactly the bottleneck rate, five of the 10 initial packets fill the pipe so the excess forms a standing queue at the bottleneck that cannot dissipate). Similarly, the full pipe condition does not guarantee there is no queue (e.g., a connection sending a BDP in BDP/2 bursts gets full bottleneck utilization, but with an average queue of BDP/4). The only way to minimize the queue at the bottleneck and all along the path is to meet both conditions simultaneously.

BtlBw and RTprop vary over the life of a connection, so they must be continuously estimated. TCP currently tracks RTT (the time interval from sending a data packet until it is acknowledged) since it's required for loss detection. At any time *t*,

$$RTT_t = RTprop_t + \eta_t$$

where $\boldsymbol{\eta} \geq 0$ represents the "noise" introduced by queues along the path, the receiver's delayed ack strategy, ack aggregation, etc. RTprop is a physical property of the connection's path and changes only when the path changes. Since path changes happen on time scales » *RTprop,* an unbiased, efficient estimator at time *T* is

$$\widehat{RTprop} = RTprop + \min(\eta_t) = \min(RTT_t) \quad \forall t \in [T - W_R, T]$$

(i.e., a running min over time window W_R (which is typically tens of seconds to minutes).

Unlike RTT, nothing in the TCP spec requires implementations to track bottleneck bandwidth, but a good estimate results from tracking delivery rate. When the ack for some packet arrives back at the sender, it conveys that packet's RTT and announces the delivery of data inflight when that packet departed. Average delivery rate between send and ack is the ratio of data delivered to time elapsed: *deliveryRate = Δdelivered/Δt*. This rate must be $\leq$ the bottleneck rate (the arrival amount is known exactly so all the uncertainty is in the *Δt*, which must be $\geq$ the true arrival interval; thus, the ratio must be $\leq$ the true delivery rate, which is, in turn, upper-bounded by the bottleneck capacity). Therefore, a windowed-max of delivery rate is an efficient, unbiased estimator of BtlBw:

$$\widehat{BtlBw} = \max(deliveryRate_t) \quad \forall t \in [T - W_B, T]$$

where the time window W_B is typically six to ten RTTs.

TCP must record the departure time of each packet to compute RTT. BBR augments that record with the total data delivered so each ack arrival yields both an RTT and a delivery rate measurement that the filters convert to RTprop and BtlBw estimates.

Note that these values are completely independent: RTprop can change (for example, on a route change) but still have the same bottleneck, or BtlBw can change (for example, when a wireless link changes rate) without the path changing. (This independence is why both constraints have to be known to match sending behavior to delivery path.) Since RTprop is visible only to the left of BDP and BtlBw only to the right in figure 1, they obey an uncertainty principle: whenever one can be measured, the other cannot. Intuitively, this is because the pipe has to be overfilled to find its capacity, which creates a queue that obscures the length of the pipe. For example, an application running a request/response protocol might never send enough data to fill the pipe and observe BtlBw. A multi-hour bulk data transfer might spend its entire lifetime in the bandwidth-limited region and have only a single sample of RTprop from the first packet's RTT. This intrinsic uncertainty means that in addition to estimators to recover the two path parameters, there must be states that track both what can be learned at the current operating point and, as information becomes stale, how to get to an operating point where it can be relearned.

Matching the Packet Flow to the Delivery Path

The core BBR algorithm has two parts:

When an ack is received

Each ack provides new RTT and average delivery rate measurements that update the RTprop and BtlBw estimates:

```
function onAck(packet)
   rtt = now - packet.sendtime
   update_min_filter(RTpropFilter, rtt)
   delivered += packet.size
   delivered_time = now
   deliveryRate = (delivered - packet.delivered) / (delivered_time -
packet.delivered_time)
    if (deliveryRate > BtlBwFilter.currentMax || ! packet.app_limited)
        update_max_filter(BtlBwFilter, deliveryRate)
    if (app_limited_until > 0)
        app_limited_until = app_limited_until - packet.size
```

The if checks address the uncertainty issue described in the last paragraph: senders can be application limited, meaning the application runs out of data to fill the network. This is quite common because of request/response traffic. When there is a send opportunity but no data to send, BBR marks the corresponding bandwidth sample(s) as application limited (see send() pseudocode to follow). The code here decides which samples to include in the bandwidth model so it reflects network, not application, limits. BtlBw is a hard upper bound on the delivery rate so a measured delivery rate larger than the current BtlBw estimate must mean the estimate is too low, whether or not the sample was app-limited. Otherwise, application-limited samples are discarded. (Figure 1 shows that in the app-limited region deliveryRate underestimates BtlBw. These checks prevent filling the BtlBw filter with underestimates which would cause data to be sent too slowly.)

When data is sent

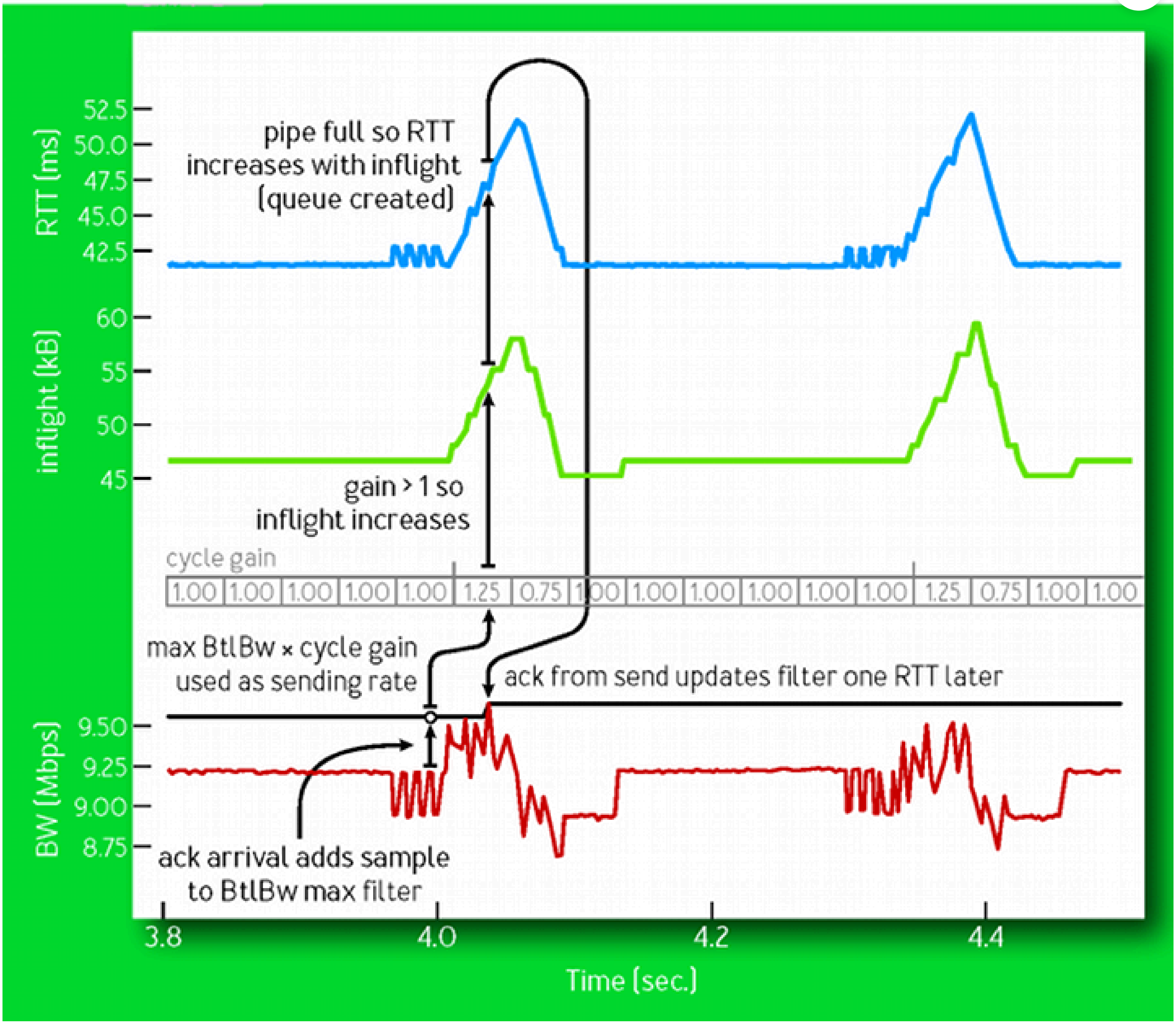
To match the packet-arrival rate to the bottleneck link's departure rate, BBR paces every data packet. (BBR must match the bottleneck *rate,* which means pacing is integral to the design and fundamental to operation—pacing_rate is BBR's primary control parameter. A secondary parameter, cwnd_gain, bounds inflight to a small multiple of the BDP to handle common network and receiver pathologies (see the later section on Delayed and Stretched ACKs). Conceptually, the TCP send routine looks like the following code. (In Linux, sending uses the efficient FQ/pacing queuing discipline,[4] which gives BBR line-rate single-connection performance on multigigabit links and handles thousands of lower-rate paced connections with negligible CPU overhead.)

```
function send(packet)
    bdp = BtlBwFilter.currentMax × RTpropFilter.currentMin
    if (inflight >= cwnd_gain × bdp)
        // wait for ack or retransmission timeout
        return
    if (now >= nextSendTime)
        packet = nextPacketToSend()
        if (! packet)
            app_limited_until = inflight
            return
        packet.app_limited = (app_limited_until > 0)
        packet.sendtime = now
        packet.delivered = delivered
        packet.delivered_time = delivered_time
        ship(packet)
        nextSendTime = now + packet.size / (pacing_gain × BtlBwFilter.currentMax)
    timerCallbackAt(send, nextSendTime)
```

Steady-state behavior

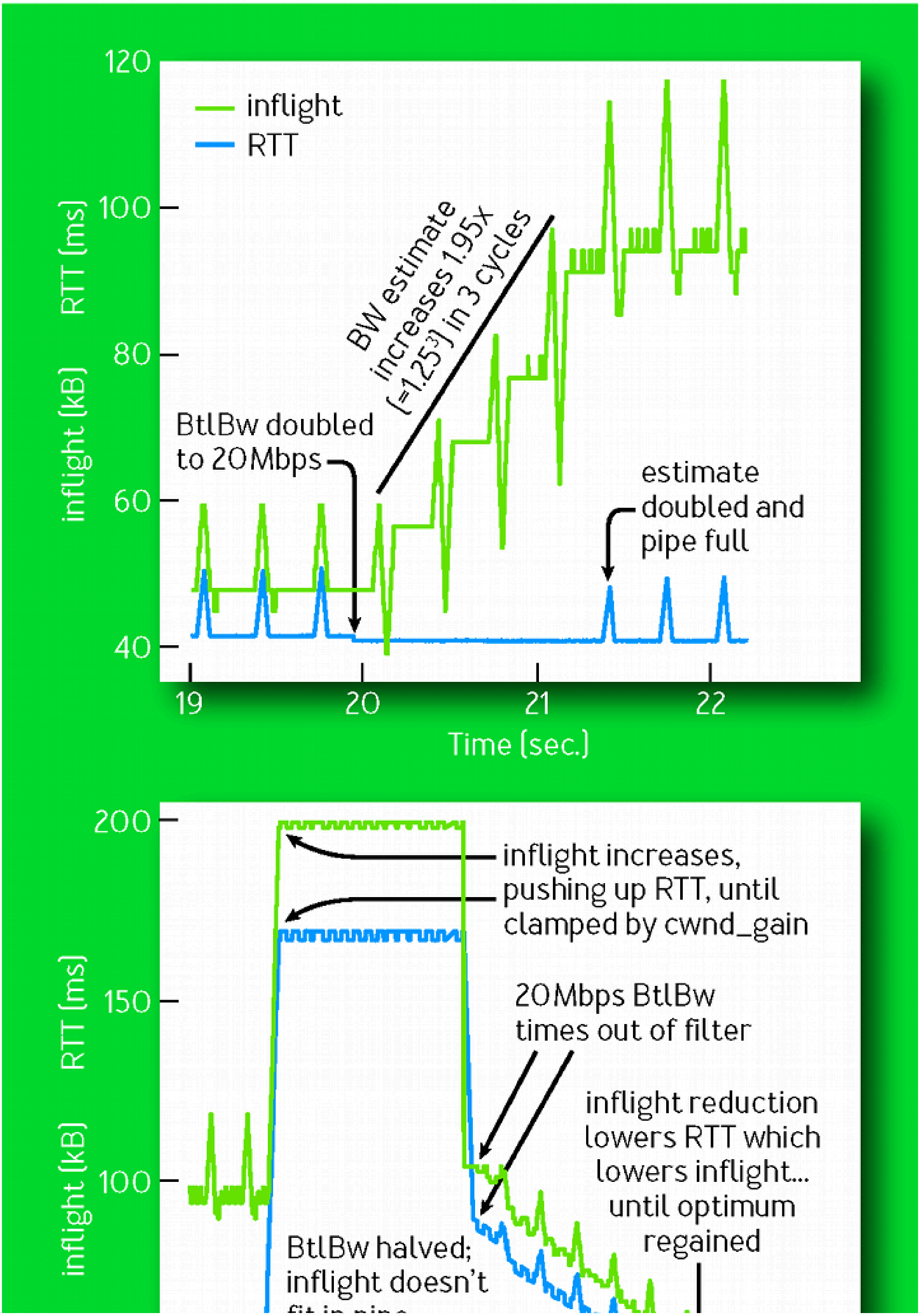
The rate and amount BBR sends is solely a function of the estimated BtlBw and RTprop, so the filters control adaptation in addition to estimating the bottleneck constraints. This creates the novel control loop shown in figure 2, which illustrates the RTT (blue), inflight (green) and delivery rate (red) detail from 700 ms of a 10-Mbps, 40-ms flow. The thick gray line above the delivery-rate data is the state of the BtlBw max filter. The triangular structures result from BBR cycling pacing_gain to determine if BtlBw has increased. The gain used for each part of the cycle is shown time-aligned with the data it influenced. The gain is applied an RTT earlier, when the data is sent. This is indicated by the horizontal jog in the event sequence description running up the left side.

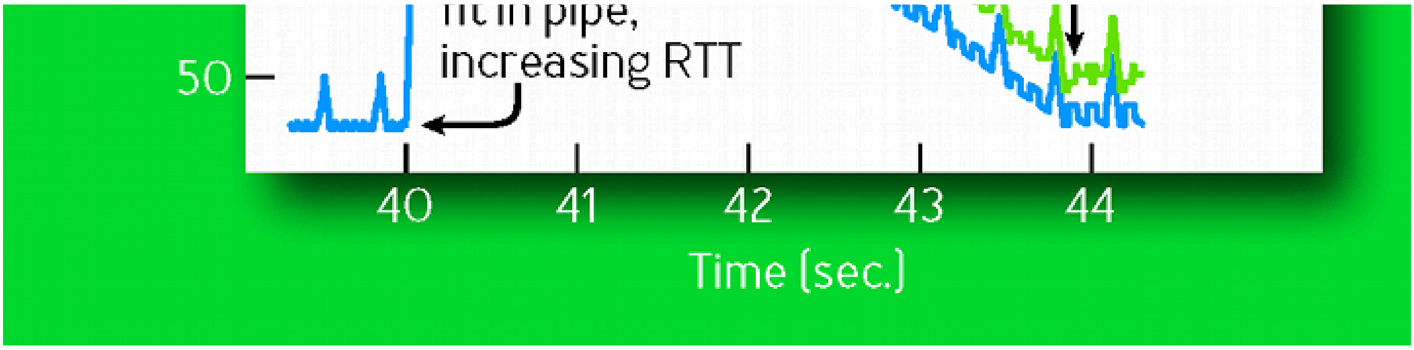
FIGURE 2: RTT (BLUE), INFLIGHT (GREEN) AND DELIVERY RATE (RED) DET



BBR minimizes delay by spending most of its time with one BDP in flight, paced at the BtlBw estimate. This moves the bottleneck to the sender so it can't observe BtlBw increases. Consequently, BBR periodically spends an RTprop interval at a pacing_gain > 1, which increases the sending rate and inflight. If BtlBw hasn't changed, then a queue is created at the bottleneck, increasing RTT, which keeps deliveryRate constant. (This queue is removed by sending at a compensating pacing_gain < 1 for the next RTprop.) If BtlBw has increased, deliveryRate increases and the new max immediately increases the BtlBw filter output, increasing the base pacing rate. Thus, BBR converges to the new bottleneck rate exponentially fast. Figure 3 shows the effect on a 10-Mbps, 40-ms flow of BtlBw abruptly doubling to 20 Mbps after 20 seconds of steady operation (left graph) then dropping to 10 Mbps after another 20 seconds of steady operation at 20 Mbps (right graph).

FIGURE 3: **BANDWIDTH CHANGE**





(BBR is a simple instance of a Max-plus control system, a new approach to control based on nonstandard algebra.[12] This approach allows the adaptation rate [controlled by the *max* gain] to be independent of the queue growth [controlled by the *average* gain]. Applied to this problem, it results in a simple, implicit control loop where the adaptation to physical constraint changes is automatically handled by the filters representing those constraints. A conventional control system would require multiple loops connected by a complex state machine to accomplish the same result.)

Single BBR Flow Startup Behavior

Existing implementations handle events such as startup, shutdown, and loss recovery with event-specific algorithms and many lines of code. BBR uses the code detailed earlier (in the previous section, Matching the Packet Flow to the Delivery Path) for everything, handling events by sequencing through a set of "states" that are defined by a table containing one or more fixed gains and exit criteria. Most of the time is spent in the ProbeBW state described in the section on Steady-state Behavior. The Startup and Drain states are used at connection start (figure 4). To handle Internet link bandwidths spanning 12 orders of magnitude, Startup implements a binary search for BtlBw by using a gain of 2/ln2 to double the sending rate while delivery rate is increasing. This discovers BtlBw in log_2BDP RTTs but creates up to *2BDP* excess queue in the process. Once Startup finds BtlBw, BBR transitions to Drain, which uses the inverse of Startup's gain to get rid of the excess queue, then to ProbeBW once the inflight drops to a BDP.

FIGURE 4: FIRST SECOND OF A 10-MBPS, 40-MS BBR FLOW

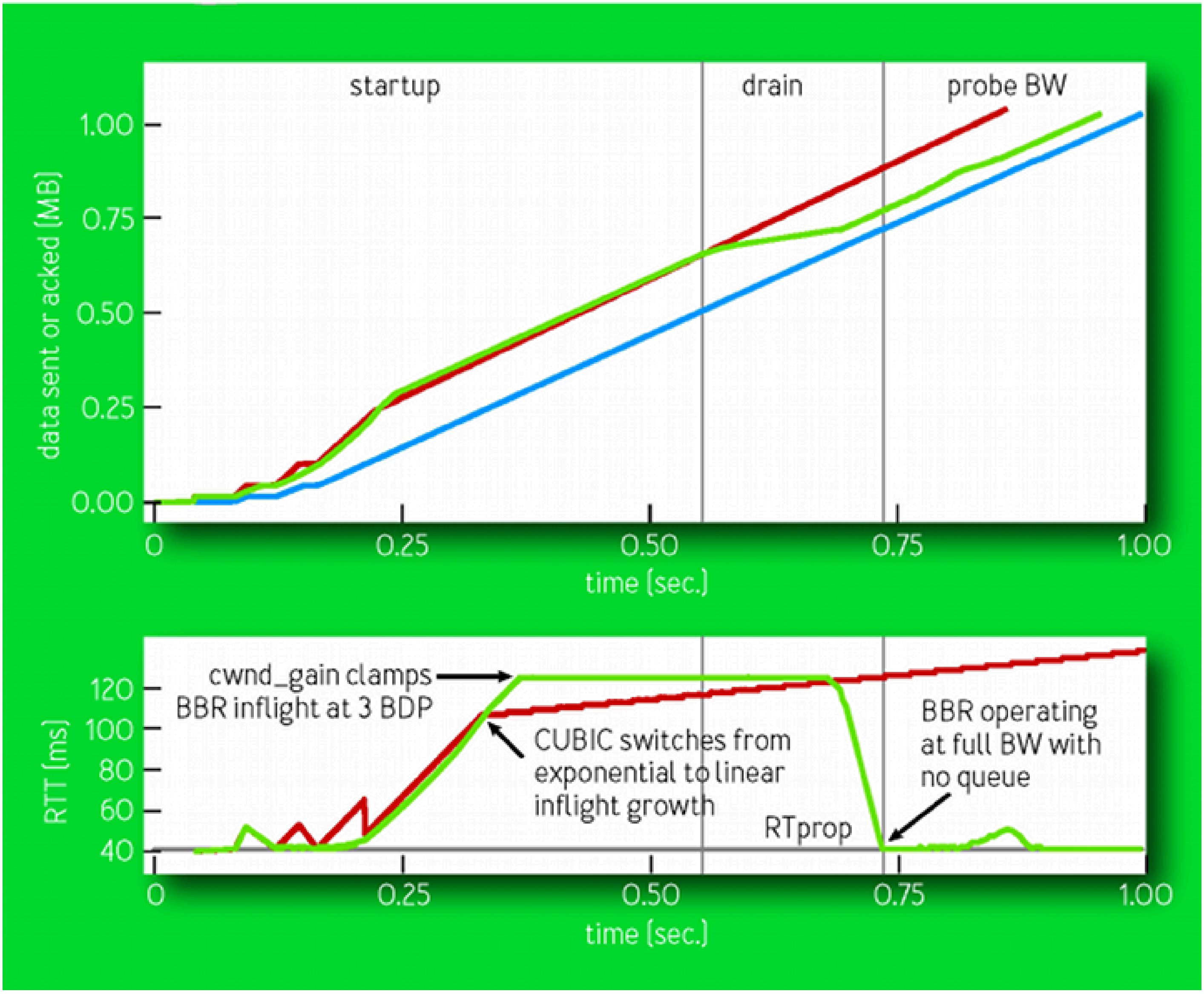
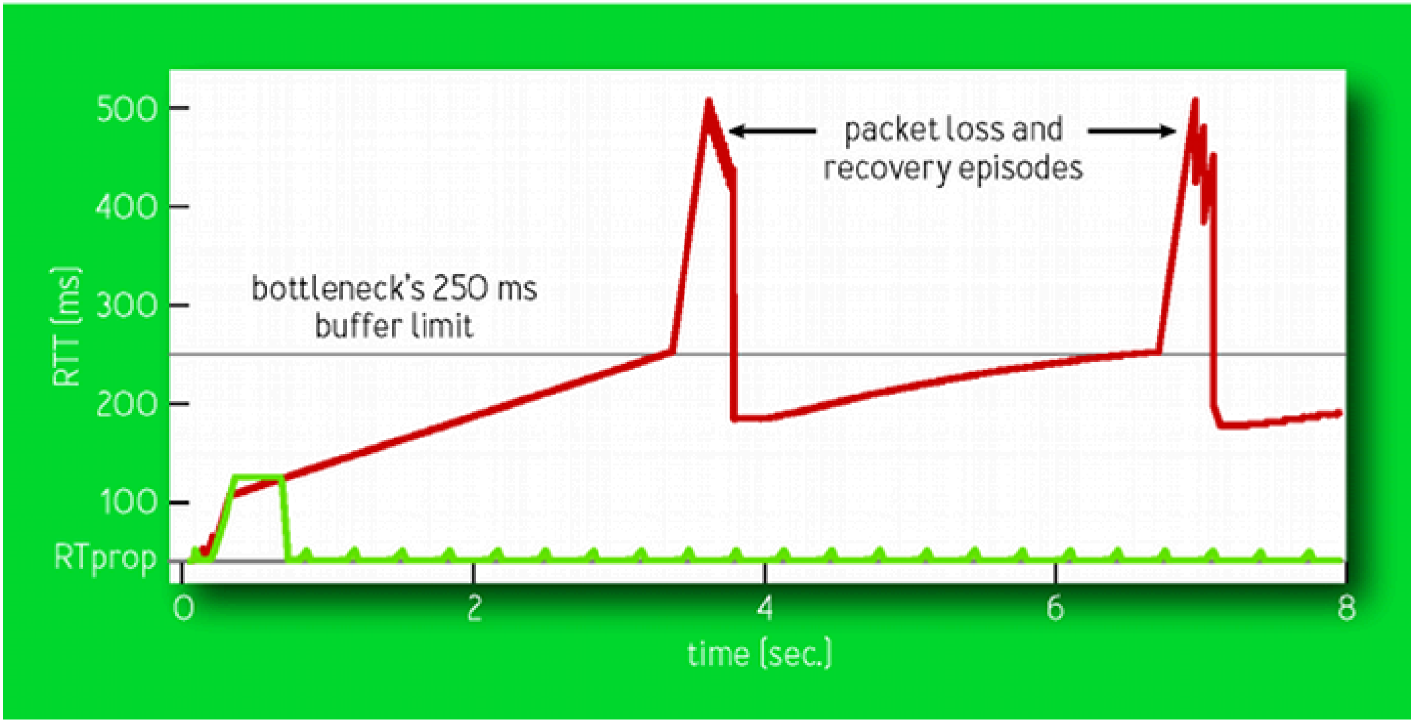


Figure 4 shows the first second of a 10-Mbps, 40-ms BBR flow. The time/sequence plot shows the sender (green) and receiver (blue) progress vs. time. The red line shows a CUBIC sender under identical conditions. Vertical gray lines mark BBR state transitions. The lower figure shows the RTT of the two connections vs. time. Note that the time reference for this data is ack arrival (blue) so, while they appear to be time shifted, events are shown at the point where BBR learns of them and acts.

The lower graph of figure 4 contrasts BBR and CUBIC. Their initial behavior is similar, but BBR completely drains its startup queue while CUBIC can't. Without a path model to tell it how much of the inflight is excess, CUBIC makes inflight growth less aggressive, but growth continues until either the bottleneck buffer fills and drops a packet or the receiver's inflight limit (TCP's receive window) is reached.

Figure 5 shows RTT behavior during the first eight seconds of the connections shown in figure 4. CUBIC (red) fills the available buffer, then cycles from 70 to 100 percent full every few seconds. After startup, BBR (green) runs with essentially no queue.

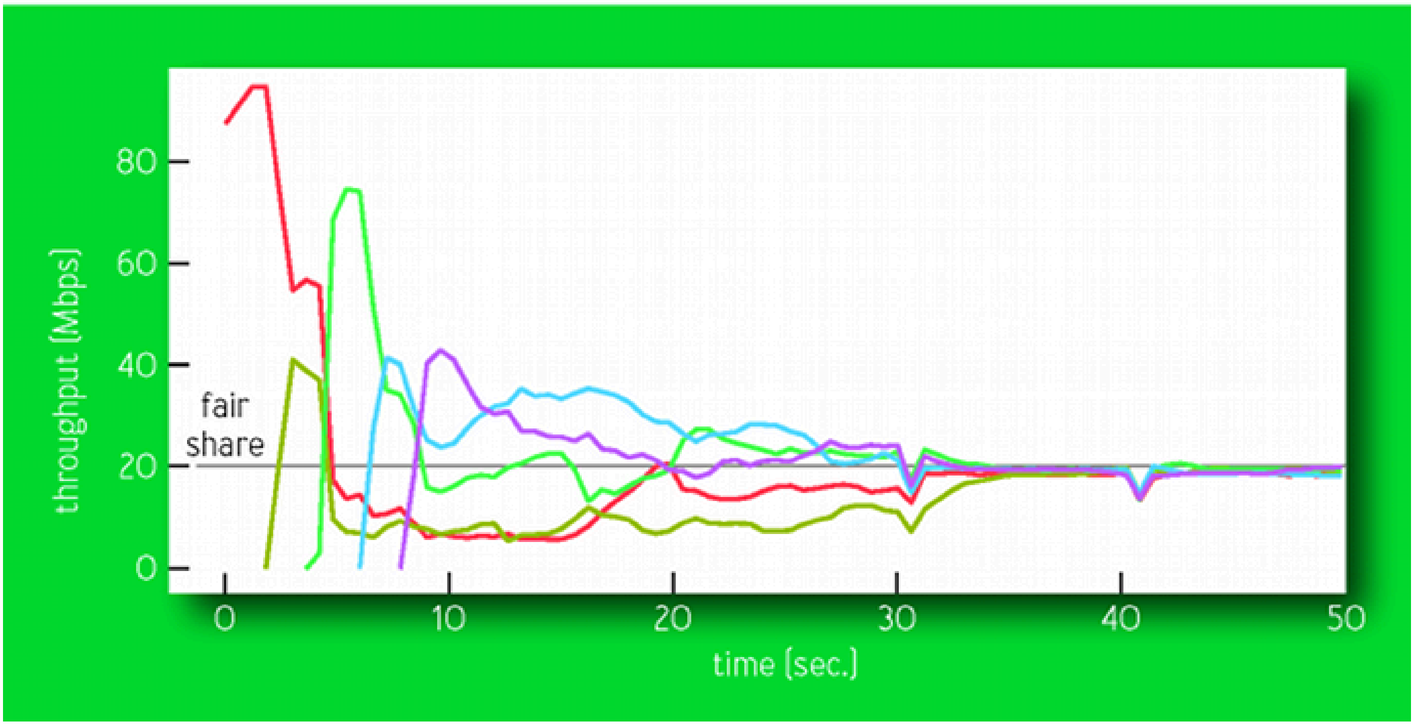
FIGURE 5: FIRST 8 SECONDS OF 10-MBPS, 40-MS CUBIC AND BBR FLOWS



Behavior of Multiple BBR Flows Sharing a Bottleneck

Figure 6 shows how individual throughputs for several BBR flows sharing a 100-Mbps/10-ms bottleneck converge to a fair share. The downward facing triangular structures are connection ProbeRTT states whose self-synchronization accelerates final convergence.

FIGURE 6: THROUGHPUTS OF 5 BBR FLOWS SHARING A BOTTLENECK



ProbeBW gain cycling (figure 2) causes bigger flows to yield bandwidth to smaller flows, resulting in each learning its fair share. This happens fairly quickly (a few ProbeBW cycles), though unfairness can persist when late starters overestimate RTprop as a result of starting when other flows have (temporarily) created a queue.

To learn the true RTProp, a flow moves to the left of BDP using ProbeRTT state: when the RTProp estimate has not been updated (i.e., by measuring a lower RTT) for many seconds, BBR enters ProbeRTT, which reduces the inflight to four packets for at least one round trip, then returns to the previous state. Large flows entering ProbeRTT drain many packets from the queue, so several flows see a new RTprop (new minimum RTT). This makes their RTprop estimates expire at the same time, so they enter ProbeRTT together, which makes the total queue dip larger and causes more flows to see a new RTprop, and so on. This distributed coordination is the key to both fairness and stability.

BBR synchronizes flows around the desirable event of an empty bottleneck queue. By contrast, loss-based congestion control synchronizes around the undesirable events of periodic queue growth and overflow, amplifying delay and packet loss.

Google B4 WAN Deployment Experience

Google's B4 network is a high-speed WAN (wide-area network) built using commodity switches.[15] Losses on these shallow-buffered switches result mostly from coincident arrivals of small traffic bursts. In 2015 Google started switching B4 production traffic from CUBIC to BBR. No issues or regressions were experienced, and since 2016 all B4 TCP traffic uses BBR. Figure 7 shows one reason for switching: BBR's throughput is consistently 2 to 25 times greater than CUBIC's. We had expected even more improvement but discovered that 75 percent of BBR connections were limited by the kernel's TCP receive buffer, which the network operations team had deliberately set low (8 MB) to prevent CUBIC flooding the network with megabytes of excess inflight (8-MB/200-ms intercontinental RTT ⇒ 335-Mbps max throughput). Manually raising the receive buffer on one US-Europe path caused BBR immediately to reach 2 Gbps, while CUBIC remained at 15 Mbps—the 133x relative improvement predicted by Mathis et al.[17]

FIGURE 7: BBR VS. CUBIC RELATIVE THROUGHPUT IMPROVEMENT

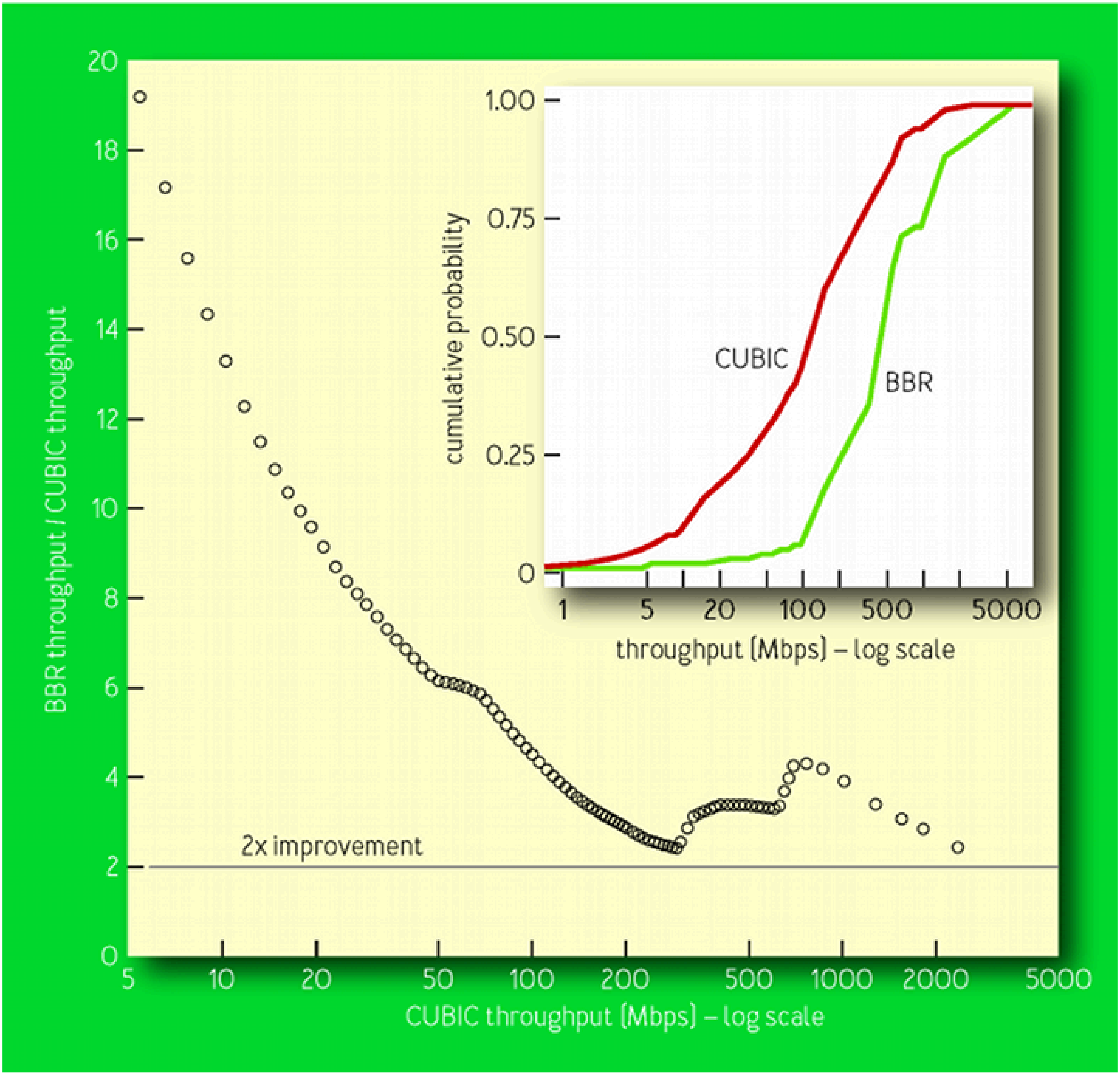


Figure 7 shows BBR vs. CUBIC relative throughput improvement; the inset shows throughput CDFs (cumulative distribution functions). Measures are from an active prober service that opens persistent BBR and CUBIC connections to remote data centers, then transfers 8 MB of data every minute. Probers communicate via many B4 paths within and between North America, Europe, and Asia.

The huge improvement is a direct consequence of BBR *not* using loss as a congestion indicator. To achieve full bandwidth, existing loss-based congestion controls require the loss rate to be less than the inverse square of the BDP[17] (e.g., < one loss per 30 million packets for a 10-Gbps/100-ms path). Figure 8 compares measured goodput at various loss rates. CUBIC's loss tolerance is a structural property of the algorithm, while BBR's is a configuration parameter. As BBR's loss rate approaches the ProbeBW peak gain, the probability of measuring a delivery rate of the true BtlBw drops sharply, causing the max filter to underestimate.

FIGURE 8: BBR VS. CUBIC GOODPUT UNDER LOSS

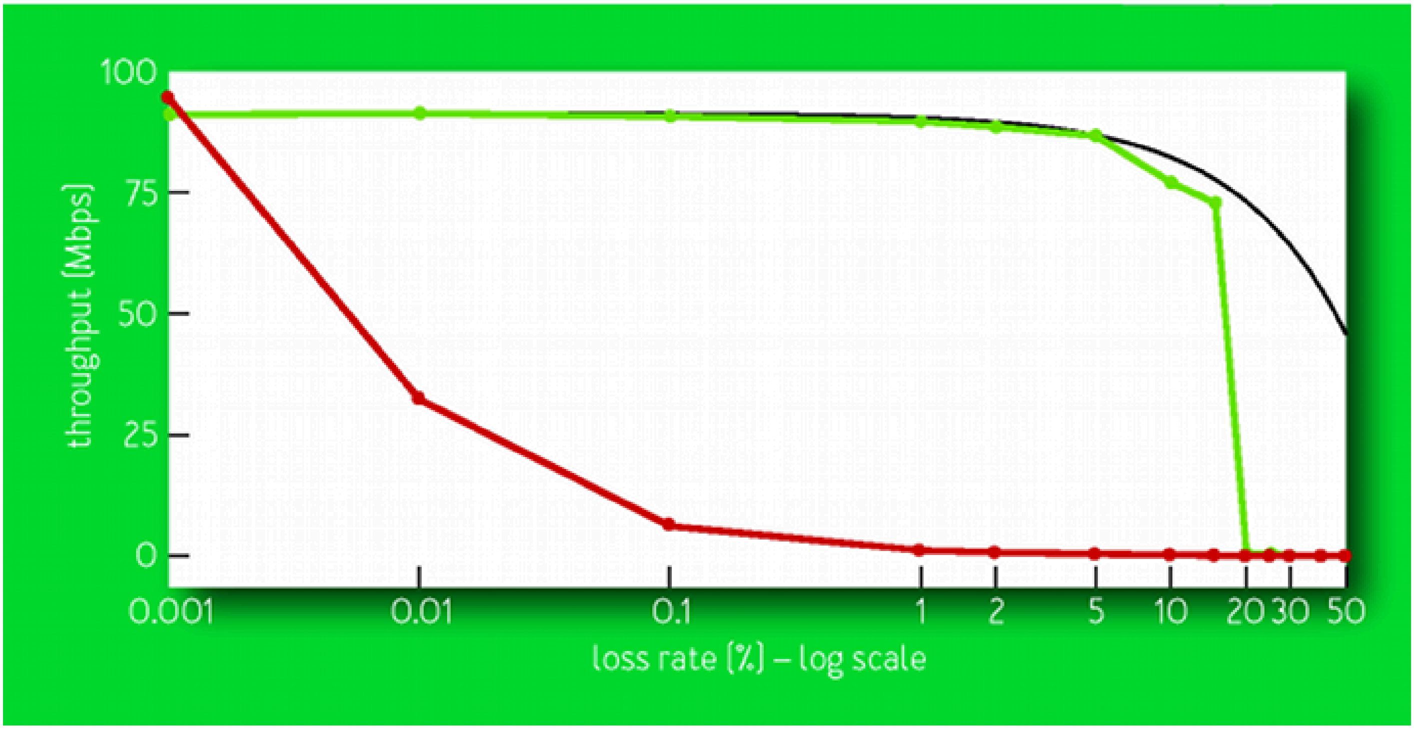


Figure 8 shows BBR vs. CUBIC goodput for 60-second flows on a 100-Mbps/100-ms link with 0.001 to 50 percent random loss. CUBIC's throughput decreases by 10 times at 0.1 percent loss and totally stalls above 1 percent. The maximum possible throughput is the link rate times fraction delivered (= *1 - lossRate*). BBR meets this limit up to a 5 percent loss and is close up to 15 percent.

YouTube Edge Deployment Experience

BBR is being deployed on Google.com and YouTube video servers. Google is running small-scale experiments in which a small percentage of users are randomly assigned either BBR or CUBIC. Playbacks using BBR show significant improvement in all of YouTube's quality-of-experience metrics, possibly because BBR's behavior is more consistent and predictable. BBR only slightly improves connection throughput because YouTube already adapts the server's streaming rate to well below BtlBw to minimize bufferbloat and rebuffer events. Even so, BBR reduces median RTT by 53 percent on average globally and by more than 80 percent in the developing world.

Figure 9 shows BBR vs. CUBIC median RTT improvement from more than 200 million YouTube playback connections measured on five continents over a week. More than half of the world's 7 billion mobile Internet subscriptions connect via 8- to 114-kbps 2.5 G systems,[5] which suffer well-documented problems because of loss-based congestion control's buffer-filling propensities.[3] The bottleneck link for these systems is usually between the SGSN (serving GPRS support node)[18] and mobile device. SGSN software runs on a standard PC platform with ample memory, so there are frequently megabytes of buffer between the Internet and mobile device. Figure 10 compares (emulated) SGSN Internet-to-mobile delay for BBR and CUBIC. The horizontal lines mark one of the more serious consequences: TCP adapts to long RTT delay except on the connection initiation SYN packet, which has an OS-dependent fixed timeout. When the mobile device is receiving bulk data (e.g., from automatic app updates) via a large-buffered SGSN, the device can't connect to anything on the Internet until the queue empties (the SYN ACK accept packet is delayed for longer than the fixed SYN timeout).

FIGURE 9: BBR VS. CUBIC MEDIAN RTT IMPROVEMENT

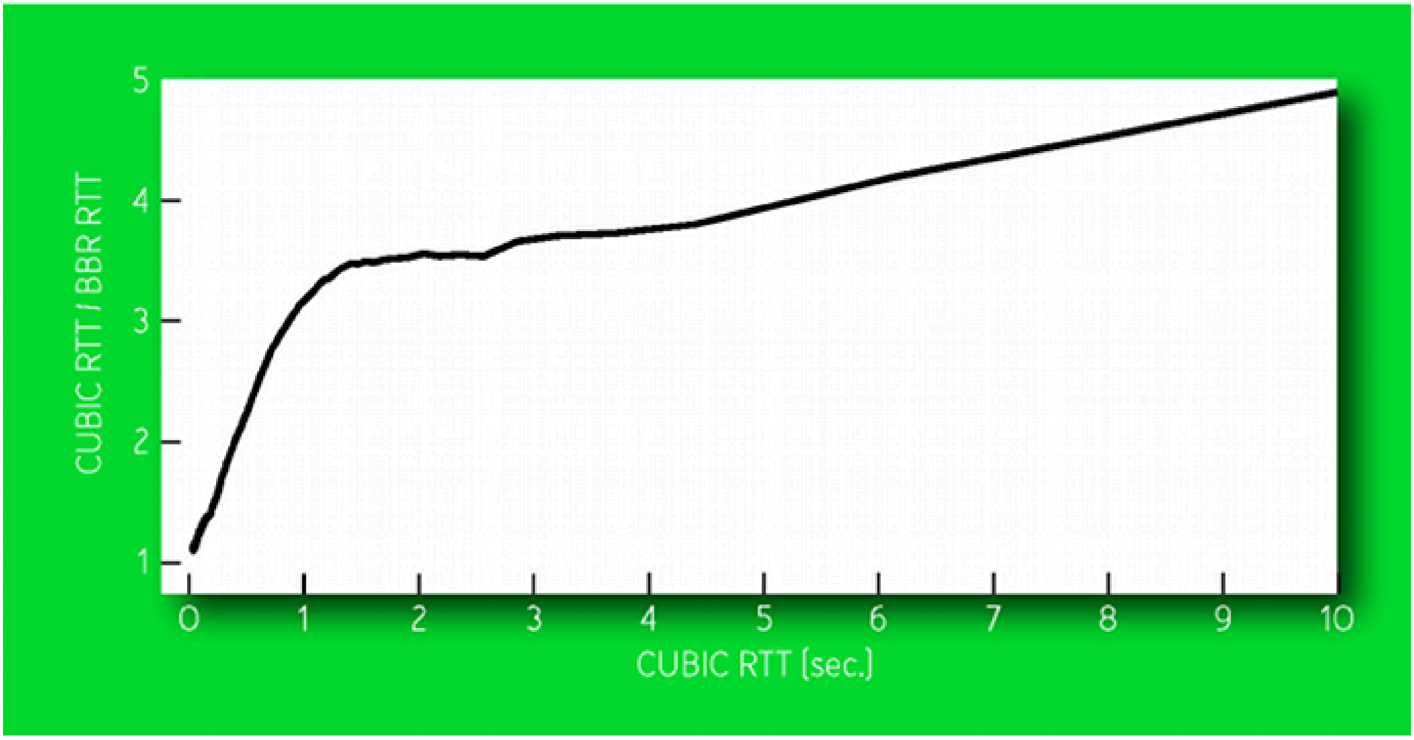
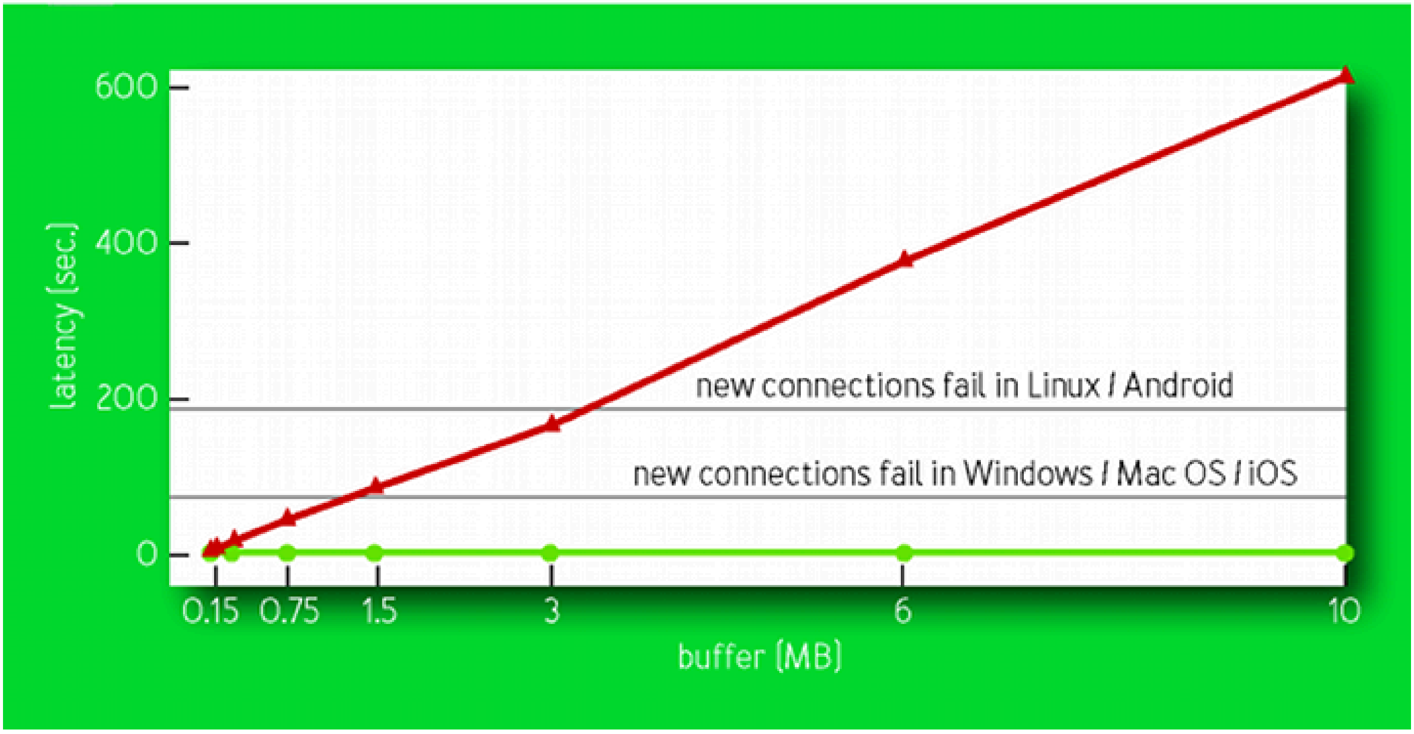


Figure 10 shows steady-state median RTT variation with link buffer size on a 128-Kbps/40-ms link with eight BBR (green) or CUBIC (red) flows. BBR keeps the queue near its minimum, independent of both bottleneck buffer size and number of active flows. CUBIC flows always fill the buffer, so the delay grows linearly with buffer size.

FIGURE 10: STEADY-STATE MEDIAN RTT VARIATION WITH LINK BUFFER SIZE



Mobile Cellular Adaptive Bandwidth

Cellular systems adapt per-subscriber bandwidth based partly on a demand estimate that uses the queue of packets destined for the subscriber. Early versions of BBR were tuned to create very small queues, resulting in connections getting stuck at low rates. Raising the peak ProbeBW pacing_gain to create bigger queues resulted in fewer stuck connections, indicating that it's possible to be too nice to some networks. With the current $1.25 \times BtlBw$ peak gain, no degradation is apparent compared with CUBIC on any network.

Delayed and Stretched ACKs

Cellular, Wi-Fi, and cable broadband networks often delay and aggregate ACKs.[1] When inflight is limited to one BDP, this results in throughput-reducing stalls. Raising ProbeBW's cwnd_gain to two allowed BBR to continue sending smoothly at the estimated delivery rate, even when ACKs are delayed by up to one RTT. This largely avoids stalls.

Token-Bucket Policers

BBR's initial YouTube deployment revealed that most of the world's ISPs mangle traffic with token-bucket policers.[7] The bucket is typically full at connection startup so BBR learns the underlying network's BtlBw, but once the bucket empties, all packets sent faster than the (much lower than BtlBw) bucket fill rate are dropped. BBR eventually learns this new delivery rate, but the ProbeBW gain cycle results in continuous moderate losses. To minimize the upstream bandwidth waste and application latency increase from these losses, we added policer detection and an explicit policer model to BBR. We are also actively researching better ways to mitigate the policer damage.

Competition with Loss-Based Congestion Control

BBR converges toward a fair share of the bottleneck bandwidth whether competing with other BBR flows or with loss-based congestion control. Even as loss-based congestion control fills the available buffer, ProbeBW still robustly moves the BtlBw estimate toward the flow's fair share, and ProbeRTT finds an RTProp estimate just high enough for tit-for-tat convergence to a fair share. Unmanaged router buffers exceeding several BDPs, however, cause long-lived loss-based competitors to bloat the queue and grab more than their fair share. Mitigating this is another area of active research.

Conclusion

Rethinking congestion control pays big dividends. Rather than using events such as loss or buffer occupancy, which are only weakly correlated with congestion, BBR starts from Kleinrock's formal model of congestion and its associated optimal operating point. A pesky "impossibility" result that the crucial parameters of delay and bandwidth cannot be determined simultaneously is sidestepped by observing they can be estimated sequentially. Recent advances in control and estimation theory are then used to create a simple distributed control loop that verges on the optimum, fully utilizing the network while maintaining a small queue. Google's BBR implementation is available in the open-source Linux kernel TCP and is described in detail in the appendix to this article.

BBR is deployed on Google's B4 backbone, improving throughput by orders of magnitude compared with CUBIC. It is also being deployed on Google and YouTube Web servers, substantially reducing latency on all five continents tested to date, most dramatically in developing regions. BBR runs purely on the sender and does not require changes to the protocol, receiver, or network, making it incrementally deployable. It depends only on RTT and packet-delivery acknowledgment, so can be implemented for most Internet transport protocols.

The authors can be contacted at bbr-dev@googlegroups.com.

Acknowledgments

The authors are grateful to Len Kleinrock for pointing out the right way to do congestion control. We are indebted to Larry Brakmo for pioneering work on Vegas[2] and New Vegas congestion control that presaged many elements of BBR, and for advice and guidance during BBR's early development. We would also like to thank Eric Dumazet, Nandita Dukkipati, Jana Iyengar, Ian Swett,

M. Fitz Nowlan, David Wetherall, Leonidas Kontothanassis, Amin Vahdat, and the Google BwE and YouTube infrastructure teams for their invaluable help and support.

Appendix - Detailed Description

A State Machine for Sequential Probing

The pacing_gain controls how fast packets are sent relative to BtlBw and is key to BBR's ability to learn. A pacing_gain > 1 increases inflight and decreases packet inter-arrival time, moving the connection to the right on figure 1. A pacing_gain < 1 has the opposite effect, moving the connection to the left.

BBR uses this pacing_gain to implement a simple sequential probing state machine that alternates between testing for higher bandwidths and then testing for lower round-trip times. (It's not necessary to probe for less bandwidth since that is handled automatically by the BtlBw max filter: new measurements reflect the drop, so BtlBw will correct itself as soon as the last old measurement times out of the filter. The RTprop min filter automatically handles path length increases similarly.)

If the bottleneck bandwidth increases, BBR must send faster to discover this. Likewise, if the actual round-trip propagation delay changes, this changes the BDP, and thus BBR must send slower to get inflight below BDP in order to measure the new RTprop. Thus, the only way to discover these changes is to run experiments, sending faster to check for BtlBw increases or sending slower to check for RTprop decreases. The frequency, magnitude, duration, and structure of these experiments differ depending on what's already known (startup or steady-state) and sending app behavior (intermittent or continuous).

Steady-State Behavior

BBR flows spend the vast majority of their time in ProbeBW state, probing for bandwidth using an approach called *gain cycling*, which helps BBR flows reach high throughput, low queuing delay, and convergence to a fair share of bandwidth. With gain cycling, BBR cycles through a sequence of values for the pacing_gain. It uses an eight-phase cycle with the following pacing_gain values: 5/4, 3/4, 1, 1, 1, 1, 1, 1. Each phase normally lasts for the estimated RTprop. This design allows the gain cycle first to probe for more bandwidth with a pacing_gain above 1.0, then drain any resulting queue with a pacing_gain an equal distance below 1.0, and then cruise with a short queue using a pacing_gain of 1.0. The average gain across all phases is 1.0 because ProbeBW aims for its average pacing rate to equal the available bandwidth and thus maintain high utilization, while maintaining a small, well-bounded queue. Note that while gain cycling varies the pacing_gain value, the cwnd_gain stays constant at two, since delayed and stretched acks can strike at any time (see the section on Delayed and Stretched Acks).

Furthermore, to improve mixing and fairness, and to reduce queues when multiple BBR flows share a bottleneck, BBR randomizes the phases of ProbeBW gain cycling by randomly picking an initial phase—from among all but the 3/4 phase—when entering ProbeBW. Why not start cycling with 3/4? The main advantage of the 3/4 pacing_gain is to drain any queue that can be created by running a 5/4 pacing_gain when the pipe is already full. When exiting Drain or ProbeRTT and entering ProbeBW, there is no queue to drain, so the 3/4 gain does not provide that advantage. Using 3/4 in those contexts only has a cost: a link utilization for that round of 3/4 instead of 1. Since starting with 3/4 would have a cost but no benefit, and since entering ProbeBW happens at the start of any connection long enough to have a Drain, BBR uses this small optimization.

BBR flows cooperate to periodically drain the bottleneck queue using a state called ProbeRTT. In any state other than ProbeRTT itself, if the RTProp estimate has not been updated (i.e., by getting a lower RTT measurement) for more than 10 seconds, then BBR enters ProbeRTT and reduces the cwnd to a very small value (four packets). After maintaining this minimum number of packets in flight for at least 200 ms and one round trip, BBR leaves ProbeRTT and transitions to either Startup or ProbeBW, depending on whether it estimates the pipe was filled already.

BBR was designed to spend the vast majority of its time (about 98 percent) in ProbeBW and the rest in ProbeRTT, based on a set of tradeoffs. ProbeRTT lasts long enough (at least 200 ms) to allow flows with different RTTs to have overlapping ProbeRTT states, while still being short enough to bound the performance penalty of ProbeRTT's cwnd capping to roughly 2 percent (200 ms/10 seconds). The RTprop filter window (10 seconds) is short enough to allow quick convergence if traffic levels or routes change, but long enough so that interactive applications (e.g., Web pages, remote procedure calls, video chunks) often have natural silences or low-rate periods within the window where the flow's rate is low enough or long enough to drain its queue in the bottleneck. Then the RTprop filter opportunistically picks up these RTprop measurements, and RTProp refreshes without requiring ProbeRTT. This way, flows typically need only pay the 2 percent penalty if there are multiple bulk flows busy sending over the entire RTProp window.

Startup Behavior

When a BBR flow starts up, it performs its first (and most rapid) sequential probe/drain process. Network-link bandwidths span a range of 10^{12}—from a few bits to 100 gigabits per second. To learn BtlBw, given this huge range to explore, BBR does a binary search of the rate space. This finds BtlBw very quickly ($log_2 BDP$ round trips) but at the expense of creating a 2BDP queue on the final step of the search. BBR's Startup state does this search and then the Drain state drains the resulting queue.

First, Startup grows the sending rate exponentially, doubling it each round. To achieve this rapid probing in the smoothest possible fashion, in Startup the pacing_gain and cwnd_gain are set to 2/ln2, the minimum value that will allow the sending rate to double each round. Once the pipe is full, the cwnd_gain bounds the queue to (*cwnd_gain - 1*) × *BDP*.

During Startup, BBR estimates whether the pipe is full by looking for a plateau in the BtlBw estimate. If it notices that there are several (three) rounds where attempts to double the delivery rate actually result in little increase (less than 25 percent), then it estimates that it has reached BtlBw and exits Startup and enters Drain. BBR waits three rounds in order to have solid evidence that the sender is not detecting a delivery-rate plateau that was temporarily imposed by the receive window. Allowing three rounds provides time for the receiver's receive-window autotuning to open up the receive window and for the BBR sender to realize that BtlBw should be higher: in the first round the receive-window autotuning algorithm grows the receive window; in the second round the sender fills the higher receive window; in the third round the sender gets higher delivery-rate samples. This three-round threshold was validated by YouTube experimental data.

In Drain, BBR aims to quickly drain any queue created in Startup by switching to a pacing_gain that is the inverse of the value used during Startup, which drains the queue in one round. When the number of packets in flight matches the estimated BDP, meaning BBR estimates that the queue has been fully drained but the pipe is still full, then BBR leaves Drain and enters ProbeBW.

Note that BBR's Startup and CUBIC's slow start both explore the bottleneck capacity exponentially, doubling their sending rate each round; they differ in major ways, however. First, BBR is more robust in discovering available bandwidth, since it does not exit the search upon packet loss or (as in CUBIC's Hystart[10]) delay increases. Second, BBR smoothly accelerates its sending rate, while within every round CUBIC (even with pacing) sends a burst of packets and then imposes a gap of silence. Figure 4 demonstrates the number of packets in flight and the RTT observed on each acknowledgment for BBR and CUBIC.

Reacting to Transients

The network path and traffic traveling over it can make sudden dramatic changes. To adapt to these smoothly and robustly, and reduce packet losses in such cases, BBR uses a number of strategies to implement the core model. First, BBR treats cwnd_gain×BDP as a target that the current cwnd approaches cautiously from below, increasing cwnd by no more than the amount of data acknowledged at any time. Second, upon a retransmission timeout, meaning the sender thinks all in-flight packets are lost, BBR conservatively reduces cwnd to one packet and sends a single packet (just like loss-based congestion-control algorithms such as CUBIC). Finally, when the sender detects packet loss but there are still packets in flight, on the first round of the loss-repair process BBR temporarily reduces the sending rate to match the current delivery rate; on second and later rounds of loss repair it ensures the sending rate never exceeds twice the current delivery rate. This significantly reduces transient losses when BBR encounters policers or competes with other flows on a BDP-scale buffer.

References

1. Abrahamsson, M. 2015. TCP ACK suppression. IETF AQM mailing list; https://www.ietf.org/mail-archive/web/aqm/current/msg01480.html.

2. Brakmo, L. S., Peterson, L.L. 1995. TCP Vegas: end-to-end congestion avoidance on a global Internet. *IEEE Journal on Selected Areas in Communications* 13(8): 1465-1480.

3. Chakravorty, R., Cartwright, J., Pratt, I. 2002. Practical experience with TCP over GPRS. In *IEEE GLOBECOM*.

4. Corbet, J. 2013. TSO sizing and the FQ scheduler. LWN.net; https://lwn.net/Articles/564978/.

5. Ericsson. 2015 Ericsson Mobility Report (June); https://www.ericsson.com/res/docs/2015/ericsson-mobility-report-june-2015.pdf.

6. ESnet. Application tuning to optimize international astronomy workflow from NERSC to LFI-DPC at INAF-OATs; http://fasterdata.es.net/data-transfer-tools/case-studies/nersc-astronomy/.

7. Flach, T., Papageorge, P., Terzis, A., Pedrosa, L., Cheng, Y., Karim, T., Katz-Bassett, E., Govindan, R. 2016. An Internet-wide analysis of traffic policing. In ACM *SIGCOMM*: 468-482.

8. Gail, R., Kleinrock, L. 1981. An invariant property of computer network power. In *Conference Record, International Conference on Communications*: 63.1.1-63.1.5.

9. Gettys, J., Nichols, K. 2011. Bufferbloat: dark buffers in the Internet. *acmqueue* 9(11); http://queue.acm.org/detail.cfm?id=2071893.

10. Ha, S., Rhee, I. 2011. Taming the elephants: new TCP slow start. *Computer Networks* 55(9): 2092-2110.

11. Ha, S., Rhee, I., Xu, L. 2008. CUBIC: a new TCP-friendly high-speed TCP variant. *ACM SIGOPS Operating Systems Review* 42(5): 64-74.

12. Heidergott, B., Olsder, G. J., Van Der Woude, J. 2014. *Max Plus at Work: Modeling and Analysis of Synchronized Systems: a Course on Max-Plus Algebra and its Applications*. Princeton University Press.

13. Jacobson, V. 1988. Congestion avoidance and control. *ACM SIGCOMM Computer Communication Review* 18(4): 314-329.

14. Jaffe, J. 1981. Flow control power is nondecentralizable. *IEEE Transactions on Communications* 29(9): 1301-1306.

15. Jain, S., Kumar, A., Mandal, S., Ong, J., Poutievski, L., Singh, A., Venkata, S., Wanderer, J., Zhou, J., Zhu, M., et al. 2013. B4: experience with a globally-

deployed software defined WAN. *ACM SIGCOMM Computer Communication Review* 43(4): 3-14.

16. Kleinrock, L. 1979. Power and deterministic rules of thumb for probabilistic problems in computer communications. In *Conference Record, International Conference on Communications: 43.1.1-43.1.10.*

17. Mathis, M., Semke, J., Mahdavi, J., Ott, T. 1997. The macroscopic behavior of the TCP congestion avoidance algorithm. *ACM SIGCOMM Computer Communication Review* 27(3): 67-82.

18. Wikipedia. GPRS core network serving GPRS support node; https://en.wikipedia.org/wiki/GPRS_core_network#Serving_GPRS_support_node_.28SGSN.29.

Related Articles

Sender-side Buffers and the Case for Multimedia Adaptation
Aiman Erbad and Charles "Buck" Krasic
http://queue.acm.org/detail.cfm?id=2381998

You Don't Know Jack about Network Performance
Kevin Fall and Steve McCanne
http://queue.acm.org/detail.cfm?id=1066069

A Guided Tour through Data-center Networking
Dennis Abts and Bob Felderman
http://queue.acm.org/detail.cfm?id=2208919

The authors are members of Google's make-tcp-fast project, whose goal is to evolve Internet transport via fundamental research and open source software. Project contributions include TFO (TCP Fast Open), TLP (Tail Loss Probe), RACK loss recovery, fq/pacing, and a large fraction of the git commits to the Linux kernel TCP code for the past five years.