BBR' – An Implementation of Bottleneck Bandwidth and Round-trip Time Congestion Control for ns-3

Mark Claypool
Worcester Polytechnic Institute
Worcester, MA
claypool@cs.wpi.edu

Jae Won Chung
Verizon Labs
Waltham, MA
jae.won.chung@verizon.com

Feng Li
Verizon Labs
Waltham, MA
feng.li@verizon.com

ABSTRACT

The dominant Internet protocol, TCP, does not work as well as it could over the wide-variety of networks facing today's applications. Bottleneck Bandwidth and Round-trip time (BBR) congestion control has been proposed as an improvement, with the promise of higher throughputs and lower delays as compared to other TCP congestion control algorithms. While BBR has been implemented for Linux, unfortunately, there is not yet an implementation for ns-3, a powerful, flexible and popular simulator used for network research. This paper presents BBR', an implementation of BBR for ns-3. BBR' extends ns-3 in a fashion similar to other TCP congestion control algorithms, re-using existing interconnection mechanisms and making BBR' extensible. Preliminary validation shows BBR' behaves and performs similarly to BBR, and preliminary performance evaluation shows BBR' has similar throughputs but significantly lower round-trip times than CUBIC in some wired and 4G LTE wireless scenarios.

ACM Reference Format:
Mark Claypool, Jae Won Chung, and Feng Li. 2018. BBR' – An Implementation of Bottleneck Bandwidth and Round-trip Time Congestion Control for ns-3. In *Proceedings of the 2018 Workshop on ns-3 (WNS3 2018), June 2018, Surathkal, India.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3199902.3199903

1 INTRODUCTION

The Transmission Control Protocol (TCP), the dominant network protocol in use on the Internet, was developed for traditional wired networks in an era with limited network resources (low bandwidths and small router queues). As such, TCP was intentionally designed so that lost packets indicate congestion, even though today's wireless networks may see packet loss due to signal corruption instead of congestion. In addition, traditional TCP determines congestion limits by filling router queues until they drop, but today's queues can be quite large, causing considerable delays when filled. Fortunately, TCP has proven adaptable to emerging networks, with many improvements to TCP's congestion control being proposed, implemented, evaluated and, eventually, widely deployed. This has proven successful through major TCP versions such as TCP NewReno [8] and TCP CUBIC [9], today's dominant congestion control algorithm for TCP.

A recently proposed congestion control algorithm for TCP is Bottleneck Bandwidth and Round Trip (BBR) [2, 3]. BBR seeks to keep router queues at the bottleneck link empty by sending at exactly the bottleneck link rate limit. To do so, the BBR sender infers the delivery rate at the receiver and uses this estimate as the bottleneck bandwidth. BBR also uses an estimated minimum round-trip time in order to keep exactly enough packets in flight to maximize throughput and minimize delay. Compared to loss-based congestion control algorithms such as Reno [1] or CUBIC [12], BBR has the potential to offer higher throughputs for bottlenecks with shallow buffers or random losses, and lower queuing delays for bottlenecks with deep buffers (avoiding "bufferbloat"). Google has already deployed BBR in its data centers, claiming significant throughput increases and latency reductions for internal backbone connections.

Despite a promising start, BBR has not been thoroughly vetted through the many network scenarios facing TCP connections in today's networks. In particular, the BBR has yet to be evaluated over 4G LTE wireless, and such networks have characteristics not faced by traditional wired networks, e.g., lossy channels, variable bitrates, potentially high latency, large per-user queues, and mobile end devices.

While BBR has recently been added to the Linux kernel,[1] many advances in network research have been made through simulation, specifically the family of *network simulators* (ns). Unfortunately, there is not yet an implementation of BBR for ns-3,[2] meaning the power and flexibility of ns-3 cannot be brought to bear on evaluating, and potentially improving, BBR.

This paper presents our design, implementation and evaluation of *BBR'*, an implementation of BBR for ns-3. Like other TCP congestion control algorithms in ns-3 (and Linux), BBR' is a separate module from the core TCP mechanisms, allowing full compatibility with all the existing TCP mechanisms (e.g., connections, retransmissions, and flow control), and interfacing with application and Internet layers as does any other version of TCP. Also like BBR, BBR' only requires changes to the sender side, not to the network nor to the receiver side.

Validation shows BBR' behaves as does BBR under controlled simulation conditions, and comparison with published BBR results [3] shows that BBR' in simulation performs similarly to BBR in the real world. Performance evaluation comparing BBR' to CUBIC over simulated wired and 4G LTE wireless networks shows BBR' achieves

WNS3 2018, June 2018, Surathkal, India
© 2018 Association for Computing Machinery.
ACM ISBN 978-1-4503-6413-3/18/06…$15.00
https://doi.org/10.1145/3199902.3199903

[1]https://patchwork.ozlabs.org/patch/671069/
[2]https://www.nsnam.org/overview/what-is-ns-3/

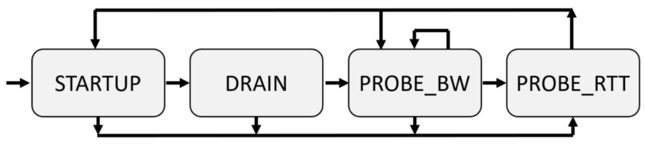


Figure 1: BBR State Transition Diagram

similar throughputs, but with dramatically lower round-trip times owing to BBR's ability to keep bottleneck queue occupancy low.

The rest of this paper is organized as follows: Section 2 gives an overview of BBR, including the protocol's states; Section 3 details our BBR' implementation, with code explained for the major functionality; Section 4 validates BBR' through analysis of protocol behavior and comparison with previously published BBR results; Section 5 evaluates BBR' compared with CUBIC in basic wireless and 4G LTE wireless scenarios; Section 6 summarizes our conclusions; and Section 7 presents possible future work.

2 BBR

BBR [4] attempts to run a TCP connection at the bottleneck bandwidth rate with minimal delay. This happens only when the total data in flight is equal to the bandwidth-delay product ($bandwidth \times delay$), or BDP.

To compute the BDP, BBR determines the minimum round-trip time (R_{min}) and the maximum delivery rate (called the bottleneck bandwidth, B_{max}) on the path from the sender to the receiver.

To determine R_{min}, BBR keeps a window of round-trip time estimates for the past 10 seconds. R_{min} is then selected as the smallest value in this window.

To determine B_{max}, BBR keeps a window of receiver delivery rate estimates (bandwidth estimates) [5] for the past 10 round-trip times. B_{max} is then selected as the largest value in this window.

BBR uses B_{max} and R_{min} to determine the number of bytes to have in flight (the BDP) via $BDP = B_{max} \times R_{min}$, allowing the TCP congestion window to grow to a small multiple of the BDP. BBR paces sending packets at a rate that matches the bottleneck bandwidth (B_{max}) – the pacing rate is BBR's primary control parameter.

Every time BBR receives a packet[3] acknowledgment, it estimates the round-trip (R_t) and bandwidth (B_t) for that packet. It then adds R_t and B_t to the round-trip time and bandwidth windows, respectively.

BBR uses the above behavior (estimate round-trip time and bottleneck bandwidth, pace packets, and have only a BDP-multiple inflight) at all times, allowing a shared code-base for all implementation aspects. However, BBR goes through 4 distinct phases that govern adjustments to the pacing rate and congestion window in order to quickly reach steady state conditions and to probe for any network changes to bottleneck bandwidth and/or round-trip time. BBR's state transition diagram is shown in Figure 1.

(1) A BBR flow begins in the STARTUP state and quickly ramps up its sending rate. While in STARTUP, BBR sets the pacing rate and the congestion window to the BDP $\times \frac{2}{ln(2)}$, roughly doubling the bitrate each round-trip time.

(2) When a BBR flow estimates the network pipe is full (the maximum bandwidth has not increased by more than 25% for the past 3 round-trip times), it enters the DRAIN state to drain the queue built up during STARTUP. While in DRAIN, BBR reduces the pacing rate to $B_{max} \times \frac{ln(2)}{2}$, but keeps the congestion window high. BBR drains long enough to remove the built-up queue, then enters PROBE_BW.

(3) A long-lived BBR flow remains primarily in the PROBE_BW state, sending at the bottleneck rate, but repeatedly probing and attempting to fully utilize any additional network bandwidth, all while maintaining a small, bounded queue. PROBE_BW does this by cycling, once per round-trip time, through a series of 8 gain values: `[1.25,0.75,1,1,1,1,1,1]`, where the gain values are applied as multiples to the bottleneck rate. For example, when the gain is 1.25, BBR deliberately sends 25% more packets than the BDP for one round-trip time. If B_{max} increases prior to this phase, the BDP, and thus overall sending rate, increases correspondingly. But if B_{max} is unchanged, the gain of 0.75 in the subsequent phase drains any queue build-up caused by the previous higher gain.

(4) If BBR has not received an RTT sample that decreases the minimum round-trip time (R_{min}) for 10 seconds, then it briefly enters the PROBE_RTT state to quickly, greatly reduce (by 98%) the packets inflight in order to re-probe the path's two-way propagation delay. The BBR flow stops probing after one round-trip time or 200 milliseconds, whichever is longer.

(5) When a BBR flow exits the PROBE_RTT state, if the full bandwidth estimate of the pipe has been reached, then it enters PROBE_BW; otherwise, it enters STARTUP to try to re-fill the pipe.

3 BBR'

BBR' (pronounced *BBR-prime*) is an implementation of BBR for ns-3. Figure 2 depicts the control flow for BBR' in relation to other components in ns-3 with which BBR' interacts. The large colored boxes, TCP, APP and BBR', represent major components. TCP corresponds to `TcpSocketBase` and `TcpSocketState` objects in ns-3, BBR' to `TcpBbr` and the BBR' state objects, and APP to any throughput-intensive application layer object (e.g., `BulkSendApplication`).

Each time TCP receive an ACK, it ① calls BBR's `PktsAcked()` which computes and updates the congestion window, stores the RTT estimate (for computing R_{min}), computes and stores the estimated BW (for computing B_{max}), ② sets the pacing rate via `SetPacingRate()`, and ③ sets the TCP congestion window via `tcb->m_cWnd`.

When an ns-3 virtual device is ready to enqueue a packet, a `DataSend()` callback ④ is made to the App. The App then ⑤ calls `Send()` one or more times to give packets to TCP for transmission.

TCP's `SendDataPacket()` ⑥ enqueues the packet for sending, returning to the App with an indication that packet is on the way.

The packet is actually transmitted based on the pacing rate where TCP sets an ns-3 timer that ⑦ triggers `PacePackets()` at the pacing interval. `PacePackets()` first ⑧ invokes BBR' `Send()` which records information needed to estimate the BW (in `PktsAcked()`). Then,

[3] Actually, a *segment*, but packet is used synonymously in this document.

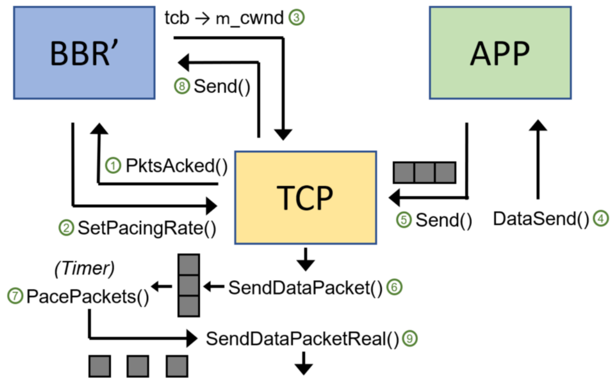


Figure 2: Overview of BBR' Control Flow

PacePackets() ⑨ sends the next packet in the queue via SendDataPacketReal(), resetting the timer to achieve paced sending.

3.1 BBR' Code

In terms of code, BBR' resides at the same software layer as other TCP congestion control versions, such as TCP NewReno, TCP Westwood, TCP Vegas, and TCP BIC. This allows BBR' flows to use the same TCP code as do all other ns-3 TCP versions and BBR' flows can interface with lower layers, such as the IP layer, without requiring special code. Where the TCP versions, including BBR', primarily differ is after a connection is established and the congestion control mechanism takes effect. In ns-3, this is defined in the TcpCongestionOps class,[4] sub-classed for each TCP version. The most important BBR' method in this class is PktsAcked() which, when TCP receives an ACK, stores the estimated round-trip time and bandwidth and then computes and sets the pacing rate.

The interested reader (and developer) is encouraged to read the technical report [7] for significantly more details on BBR' and the git repository [6] for the actual source code.

3.2 ns-3 Code Modifications

While BBR' was designed to avoid modifications to the existing ns-3 code base as much as possible, some slight modifications are required to support the unique features required by BBR. In total, the changes to support BBR' require about 75 new lines of code to 3 different files. This section describes the required changes, with patches available for ns-3.27 available in the associated git repository [6].

In order to support controlling the cwnd and pacing rate via BBR' Send(), the TcpCongestionOps class is extended with a virtual Send() method that is called at the top of TcpSocketBase::SendDataPacket(). This code hook invokes a Send() method each time a TCP socket sends data, with BBR' (and any other TCP version that provides a custom Send() method) invoking a custom Send() that performs appropriate congestion actions before sending. BBR's Send() records information to estimate the bandwidth.

[4]This class mimics the Linux tcp_congestion_ops structure

Listing 1: SendDataPacket() (in tcp-socket-base.cc)

```
// If pacing, queue until time to send else send now.
uint32_t TcpSocketBase::SendDataPacket (SeqNum32 seq,
                    uint32_t maxSize, bool withAck)

  // If not pacing, send now.
  if (m_pacing_rate == 0.0)
    return SendDataPacketReal(seq, maxSize, withAck);

  // Store packet.
  tcp_pacing_struct packet{seq, maxSize, withAck};
  m_pacing_packets.push(packet);

  // If no pending event, immediately schedule.
  if (m_pacing_event.IsExpired())
    m_pacing_event = Simulator::ScheduleNow(
             &TcpSocketBase::PacePackets, this);
```

Listing 2: SendDataPacketReal() (in tcp-socket-base.cc)

```
// Really send the data packet.
uint32_t TcpSocketBase::SendDataPacketReal (
  SeqNum32 seq, uint32_t maxSize, bool withAck) {

  // Hook to do congestion control actions.
  m_congestionControl->Send(this, m_tcb);

  // Rest is same as original SendDataPacket()...
```

An unrelated change is needed to make public the TcpSocketBase method BytesInFlight() so that BBR' can decide when to exit the DRAIN state.

Packet pacing, as required by the BBR specification [4], requires a somewhat more substantial change to the ns-3 code. In general, packet pacing to support BBR' in ns-3 is implemented by "hijacking" the normal TCP send and, instead of sending the packet, putting that packet in a queue. Packets are removed from this queue and sent with a fixed time-gap (i.e., paced), with the inter-packet time computed from the pacing rate and controlled by an ns-3 timer.

TcpSocketBase is extended with a structure for the packets queued for pacing (tcp_pacing_struct), stored in a queue (m_pacing_packets). The inter-packet timing is stored in an ns-3 event (m_pacing_event).

The packet is actually hijacked in SendDataPacket() by storing the packet rather than sending it, shown in Listing 1.

The SendDataPacketReal() method, shown in Listing 2, does the "real" sending by pulling the next packet off the queue and sending it, effectively doing the work of the original TCP SendDataPacket() method.

The pacing rate (once computed by BBR') is stored as an attribute of the TcpSocketState class with methods to get and set it.

The actual pacing of packets is done by a PacePackets() method that is repeatedly triggered with an ns-3 timer every inter-packet interval, sending a packet, as shown in Listing 3.

4 VALIDATION

To validate BBR', key protocol statistics are observed under a basic bottleneck condition (Section 4.1). Then, simulated BBR' performance results are compared with published BBR performance results [3]. For the latter, three scenarios are examined: steady state

Listing 3: `PacePackets()` (in tcp-socket-base.cc)

```
// Send packet in queue and set timer for next send.
void TcpSocketBase::PacePackets()

  // Get next packet to send.
  tcp_pacing_struct p = m_pacing_packets.front();
  m_pacing_packets.pop();

  // Send it.
  SendDataPacketReal(p.seq, p.maxSize, p.withAck);

  // Get size for computing pacing interval.
  double size = p.maxSize;

  // Schedule next send event.
  if (m_pacing_rate > 0) {
    size *= 8 / 1000000.0;   // to Mbits.
    double delta = size / m_pacing_rate; // to sec.
    delta *= 1000000000;   // to nanosec.
    m_pacing_event = Simulator::Schedule(Time(delta),
                   &TcpSocketBase::PacePackets, this);
  }
```

where the bottleneck bandwidth is unchanging (Section 4.2), steady state with an abrupt increase in the bottleneck bandwidth (Section 4.3), and steady state with an abrupt decrease in the bottleneck bandwidth (Section 4.4). Additional validation analysis and scenarios are shown in the technical report [7].

4.1 Basic Bottleneck

A bulk-download over BBR' was run over a single bottleneck (10 Mb/s, 22 ms RTT, queue size 100 packets) for 12 seconds, with hooks to record round-trip time estimates, bandwidth estimates, and protocol states. Traces were analyzed to measure the bottleneck queue occupancy and throughput.

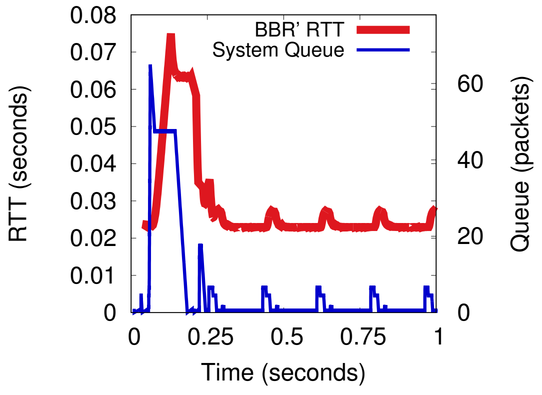
BBR' went through it's states (see Section 2) at the following times:

State	Time (seconds)
STARTUP	[0.000, 0.134)
DRAIN	[0.134, 0.200)
PROBE_BW	[0.200, 10.001)
PROBE_RTT	[10.001, 10.205)
PROBE_BW	[10.205, 12.000]

As expected, given the relatively short round-time and modest bottleneck bandwidth, BBR' spends little time in the STARTUP and DRAIN states (about 150 milliseconds total). Most of the time is spent in the PROBE_BW state (about 14.5 out of 15 seconds). Since the base RTT never changes, BBR' enters the PROBE_RTT state once about 10 seconds in, stays there for 200 milliseconds, and then returns to the PROBE_BW state.

Figure 3 graphs the two key BBR' statistics, estimated RTT (top) and estimated BW (bottom), for the first second (the x-axis). In both graphs, the thick lines in red are the BBR' predictions at the sender-side transport layer, while the thinner blue lines are the actual system-level measurements of the queue occupancy (top) and the bottleneck throughput (bottom). The system-level measurements are to assess whether or not BBR' accurately predicts the underlying network state.

In Figure 3-top, the round-trip time during BBR' STARTUP overshoots the bottleneck bandwidth and fills the bottleneck queue, but after draining the queue built up during the DRAIN state, at about



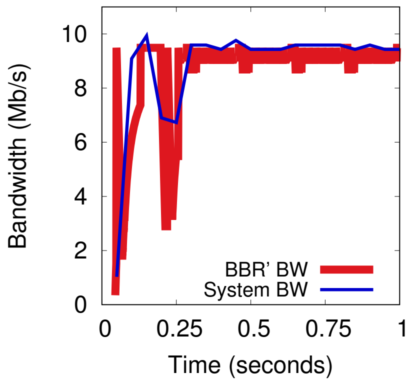


Figure 3: BBR' RTT and BW Estimates

0.15 seconds the round-trip returns to near the minimum of 0.022 seconds. BBR settles into PROBE_BW where the small "spikes" in the round-trip times are due to the gain rate cycling. BBR's round-trip time measurements closely match the actual queue occupancy. The minimum round-trip time (not shown), picked out by BBR' over all round-trip time estimates over the last 10 seconds, stays fixed near the minimum 0.022 seconds.

In Figure 3-bottom, the bandwidth estimates during STARTUP and DRAIN vary between about 0.3 and 9.7 Mb/s and during PROBE-_BW vary between 9.4 and 9.6 Mb/s. BBR's BW measurements closely match the actual delivery rates. The maximum bandwidth (not shown), picked out by BBR' over all bandwidth estimates over the past 10 round-trip times, stays fixed at about 9.7 Mb/s.

4.2 Steady State, No Bandwidth Change

Cardwell et al. published results depicting BBR steady-state behavior (i.e., in the PROBE_BW state) of a 700 ms of a 10 Mb/s BBR flow with a round-trip time of 40 ms, shown in Figure 4 of [3]. Figure 4 shows their graph on the left, with a graph for an equivalent BBR' flow on the right. The BBR graphs were annotated by the authors to illustrate protocol behavior.

Generally, the performance results for BBR and BBR' look quite similar, with nearly the same round-trip times, bandwidths and bytes inflight. More specifically, both sets of graphs have the same

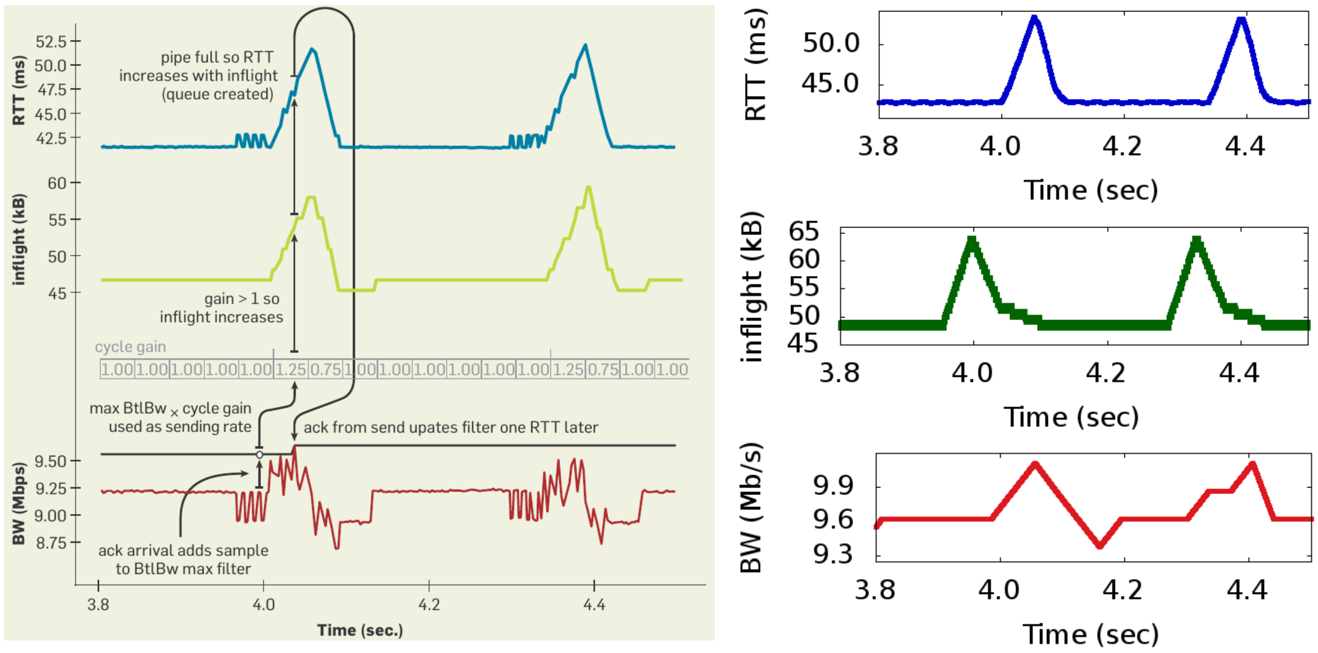


Figure 4: Comparison of BBR (left, from [3]) with BBR' (right). Graphs are round-trip time in blue, inflight in green and delivery rate in red

vertical structures resulting from the PROBE_BW cycling to determine if the bottleneck bandwidth has increased. In both scenarios, the bottleneck capacity does not change, so the increase in the data rate when the pacing gain is 1.25 results in a buildup of the bottleneck queue and an increase in the round-trip time. The immediately following cycle with a reduced pacing gain of 0.75 lowers the data rate and drains the queue, before returning to a gain rate of 1.0 until the next increase. Since the round-trip time is about 40 milliseconds, these vertical "spikes" are about 320 ($8 \times 40 = 320$) milliseconds apart.

4.3 Bandwidth Increase

Cardwell et al. published results (Figure 5-top of [3]) depicting a 10 Mb/s, 40 ms RTT network with a long-lived BBR flow where there is a sudden doubling of bandwidth (up to 20 Mb/s) at time 20 seconds. Figure 5 shows their graph on the left, with graphs for an equivalent BBR' scenario on the right. Again, the BBR graphs were annotated by the authors to illustrate protocol behavior.

From the graphs, BBR and BBR' generally behave the same, detecting the change in bandwidth at time 20 seconds and quickly doubling the bytes inflight to utilize the new found capacity. For both BBR and BBR', the round-trip time stays low, near the channel minimum, with only occasional increases during PROBE_BW gain cycles.

4.4 Bandwidth Decrease

Cardwell et al. also published results (Figure 5-bottom of [3]) of the same flow in Section 4.3 with a sudden halving of bandwidth (from 20 Mb/s down to 10 Mb/s) at time 40 seconds. Figure 6 shows their graph (and annotations) on the left, with graphs for an equivalent BBR' scenario on the right.

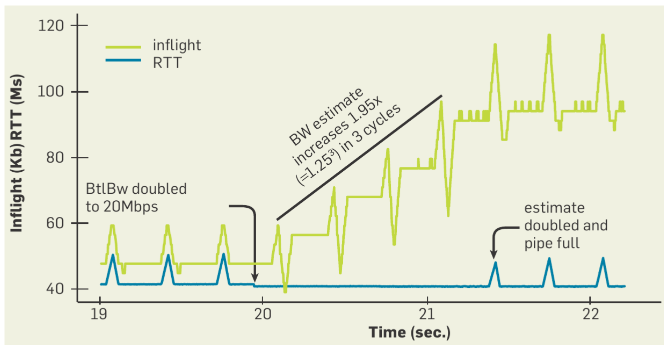
From the graphs, again, BBR and BBR' generally behave the same. The abrupt decrease in bandwidth at time 40 results in a marked increase in the round-trip time as the bytes inflight fill the bottleneck queue. Once the high bandwidth readings are pushed out of the bandwidth window at around time 41.75, BBR and BBR' both settle into the new inflight amount, draining the built up queue and returning the round-trip times to near the channel minimum.

4.5 Summary

In summary, observations of BBR' behavior in a basic, single-bottleneck with known parameters align with expectations, giving some confidence in the implementation. Further visual confirmation showing published BBR results are similar to equivalent BBR' results helps validate the BBR' implementation. While the validation has only been undertaken for the steady-state behavior in the PROBE_BW state, long-lived, throughput intensive flows (BBR's target) by far spend most of their time in this state.

5 EVALUATION

This section evaluates BBR' (v1.7) compared with CUBIC first for a basic wired connection with the bottleneck at the router (Section 5.1) followed by a basic 4G LTE configuration with the bottleneck at the eNodeB (Section 5.2).



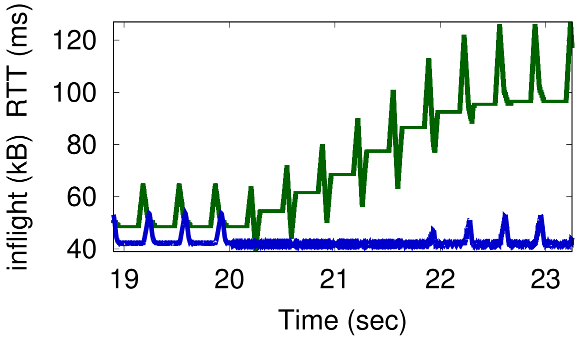
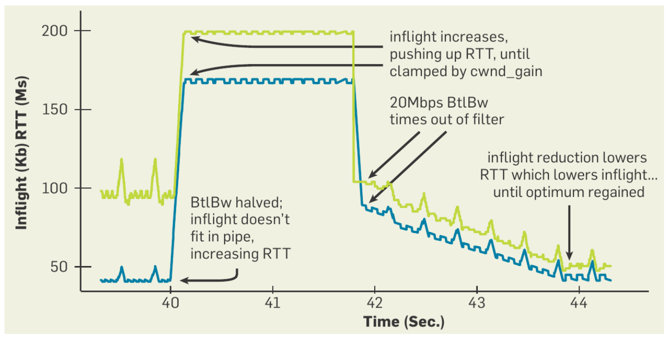


Figure 5: Comparison of BBR (left, from [3]) with BBR' (right). Green lines are bytes inflight and blue lines are RTT



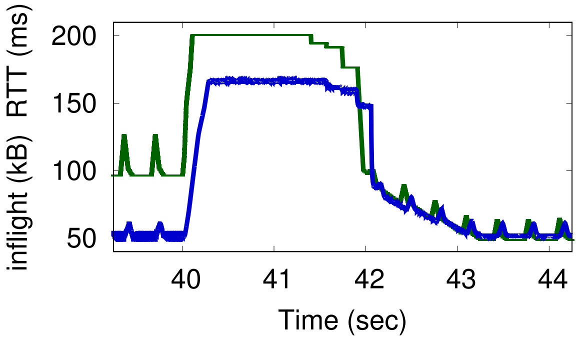


Figure 6: Comparison of BBR (left, from [3]) with BBR' (right). Green lines are bytes inflight and blue lines are RTT

5.1 Wired

The intent is to represent the canonical congestion scenario of a network constrained by an interior bottleneck of a router between a client and a server. This often means a server connected by a high capacity, modest latency connection to a router with a lower capacity, lower latency connection to the client (e.g., a head-end to a residential PC). A throughput-intensive flow runs down from the server to the client.

The topology used is shown in Figure 7. The variables d_s and d_c represent the one-way delay from the Server to the Router and from the Router to the Client, respectively. The variables b_s and b_c represent the bandwidth (capacity) from the Server to the Router and from the Router to the Client, respectively.

For this evaluation, the network conditions are as follows:

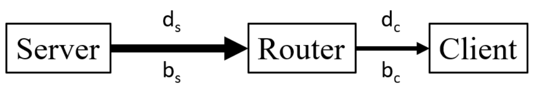


Figure 7: Wired Topology

Parameter	Value
d_s	10 milliseconds
d_c	1 millisecond
b_s	150 Mb/s
b_c	20 Mb/s
packet size	1000 bytes
queue size	60 packets

The router queue size of 60 packets is about the bandwidth-delay product.

The scenario is run first with a single BBR' flow doing a bulk download from server to client and then re-run, replacing the BBR' flow with a CUBIC flow. Since ns-3 does not come with a built-in version of TCP CUBIC, the latest TCP CUBIC (version 3.27[5]) by Levasseur et al. [10] is used.

Figure 8 depicts the results of both the BBR' and CUBIC runs plotted together, showing the first 3 seconds. The BBR' trendlines are all thick and red and the CUBIC trendlines are thin and blue. The horizontal axis for all graphs is the elapsed time in seconds.

The left graph shows the throughput, with both CUBIC and BBR' quickly reaching and maintaining the maximum capacity of nearly 20 Mb/s.

However, a major difference can be observed in the middle graph which shows queue occupancy. Here, CUBIC quickly saturates the router queue and keeps the queue persistently filled. BBR', on the other hand, after initially saturating the queue, drains the queue to nearly 0 and maintains low queue occupancy throughout. The small, periodic spikes in the queue for BBR' are due to the gain cycling in the PROBE_RTT state (see Section 2).

The effect of the router queue on round-trip time is observed in the right graph. CUBIC, by persistently filling the router queue, has a round-trip that is consistently about 35 milliseconds higher than that of BBR'.

5.2 4G LTE

For 4G LTE evaluation, a similar topology is used but the "last mile" is setup for LTE, shown in Figure 9. The router is replaced by an eNodeB and a packet gateway (PGW), the client with the User Equipment (UE, e.g., a mobile phone), and the final wired link becomes a 4G LTE connection. The UE is stationary, but positioned at different fixed distances from the eNodeB. The server to PGW bandwidth and latency are as for the wired setup in Section 5.1.

Parameter	Value
d_s	10 milliseconds
b_s	150 Mb/s
packet size	1000 bytes
mode	RLC AM
max tx buffer	512 Kbytes
resource blocks	50
HARQ	enabled
UE to eNodeB distance	varies

5.2.1 Medium Distance. For the first simulation, the UE is placed 5 kilometers from the eNodeB. The results are depicted in Figure 10, with the left graph showing throughput and the right graph showing round-trip time. For both graphs, the x-axis is the elapsed time in seconds. Unfortunately, the corresponding eNodeB queue occupancy is not readily available but, based on Figure 8, can be inferred from the round-trip time.

For throughput, both CUBIC and BBR' perform about the same, with both protocols achieving about 11.5 Mb/s during steady-state. The overall mean throughput for CUBIC is 10.8 Mb/s and the mean throughput for BBR' is 11.0 Mb/s. This slightly lower CUBIC throughput is due to BBR' more quickly ramping up to the bottleneck bandwidth during STARTUP versus CUBIC's slowstart.

As in the wired network case (Section 5.1), there is a bigger difference between CUBIC and BBR' in their round-trip times. In the beginning, both BBR' and CUBIC have a low round-trip time, about 30 milliseconds, which quickly increases for about 100 milliseconds as the flows startup during slow start for CUBIC and STARTUP for BBR'. However, after about 0.75 seconds, BBR' has settled into its target empty-queue, high-bandwidth condition with the round-trip times back to their initial, minimal values whereas the queue (and, hence, round-trip time) for CUBIC remains full.

5.2.2 Versus Distance. In order to analyze performance over a range of 4G LTE conditions, additional simulations were run with the UE at different distances from the eNodeB, ranging from extremely close (0 meters) to extremely far (over 20 kilometers).[6] Each simulation consisted of a single, 5-second bulk download, with separate runs for both CUBIC and BBR'.

The results are depicted in Figure 10, with the left graph showing the average throughput and the right graph showing the average round-trip time. For both graphs, the x-axis is the distance in meters from the UE to the eNodeB.

From the throughout graph, both CUBIC and BBR' achieve similar throughputs.

For the round-trip time graph, however, there are marked differences, with CUBIC having higher round-trip times for all distances above 1000 meters. Moreover, CUBIC's round-trip time increases fairly consistently with an increase in distance as sending packets from the saturated LTE queues takes longer as the channel conditions worsen. BBR', however, maintains a relatively low average round-trip time that is about the same at all distances since it keeps a relatively empty LTE queue regardless of the LTE channel conditions. Broadly the throughput similarities for CUBIC and BBR' and the round-trip time benefits for BBR' reflect real-life experiments comparing CUBIC and BBR over LTE [11].

6 CONCLUSION

The evolution of networks demands continued improvements to TCP, the dominant protocol on the Internet. Unfortunately, the predominant congestion control algorithm for TCP, CUBIC [9], saturates congested router queues, leading to dropped packets and higher than necessary round-trip times. The new congestion control algorithm BBR [2, 3] promises to improve TCP performance compared to CUBIC by keeping about one bandwidth-delay product of data in flight, maximizing receiver delivery rates while minimizing bottleneck queue occupancies. Despite this potential and claimed success in Google's own networks, BBR has yet to be fully vetted, particularly through simulation in ns-3, a popular, flexible simulator used for network research.

This paper presents BBR', an implementation of BBR for ns-3. BBR' integrates with TCP in ns-3 as do other congestion control algorithms, such as NewReno, Westwood and Vegas. This allows lower layers (e.g., IP) and higher layers (e.g., bulk-download applications) to use TCP BBR' as they would any other version of TCP, making it easy to deploy and test. Preliminary validation of BBR' shows the protocol behaves as per the BBR specification [4] and performs similarly to previously published BBR results [3]. Performance evaluation comparing BBR' with CUBIC shows that BBR'

[5]http://perform.wpi.edu/downloads/#cubic

[6]At 22 kilometers, neither TCP CUBIC nor TCP BBR' was able to get any packets delivered in 5 seconds.

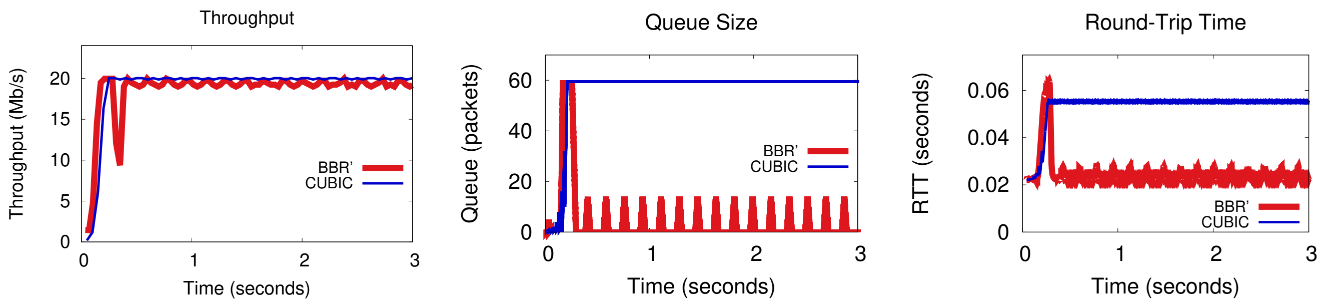


Figure 8: Wired Network

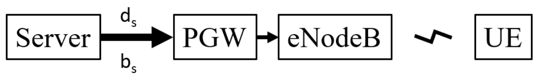


Figure 9: Wireless Topology

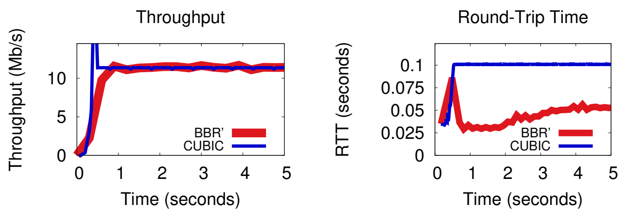


Figure 10: 4G LTE Network (UE Distance is 5k)

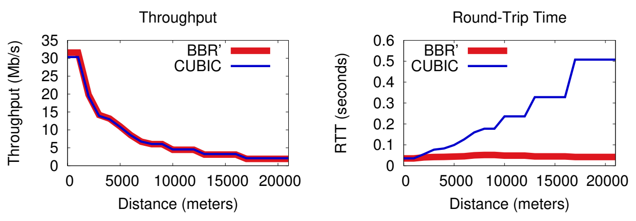


Figure 11: 4G LTE Network versus UE Distance

achieves comparable throughputs, while keeping congested queue occupancy's low thus having lower round-trip times.

7 FUTURE WORK

While a promising start, there are several areas of future work for extending BBR' in ns-3.

The current BBR' implementation assumes the application always has data to send. BBR has specifications that deal with flows that are rate-limited by the application which could be incorporated (and validated and evaluated) into BBR'.

BBR' estimates the bottleneck bandwidth by computing the estimated receiver delivery rate based on Cheng et al. [5]. While the core BBR' implementation appears to work as expected, challenges that face the technique, such as packet reordering, packet loss, and ACK loss, could be built and tested in BBR'.

While pacing is indicated as mandatory for BBR [4], preliminary tests with BBR' without pacing suggest decent performance solely by controlling rates with the congestion window. Further evaluation can help determine when pacing benefits performance and when it might not be needed. BBR' includes two alternate configuration options to disable pacing [6] that could be used for this.

Future work also includes BBR' evaluation over a wider-range of network conditions, including but not limited to capacities, topologies, protocols and application types. Evaluation on modern wireless networks such as 4G LTE, could include UEs with mobility, environments having mixed TCP versions and diverse applications.

Since many simulations target large scale networks, the efficiency of the BBR' ns-3 module could be analyzed and improved, as necessary.

ACKNOWLEDGMENTS

Thanks to Apoorva Lad and Chengle Zhang for early work on the initial approach and surrounding simulator setup.

REFERENCES

[1] M. Allman, V. Paxson, and E. Blanton. 2009. TCP Congestion Control. *IETF Request for Comments (RFC) 5681* (Sept. 2009).
[2] N. Cardwell, Y. Cheng, C.S. Gunn, S.H. Yeganeh, and V. Jacobson. 2016. BBR: Congestion-Based Congestion Control. *ACM Queue* (Oct. 2016).
[3] N. Cardwell, Y. Cheng, C.S. Gunn, S.H. Yeganeh, and V. Jacobson. 2017. BBR: Congestion-Based Congestion Control. *Commun. ACM* 60, 2 (Feb. 2017).
[4] N. Cardwell, Y. Cheng, S. Hassas Yeganeh, and V. Jacobson. 2017. BBR Congestion Control. *IETF Draft* (July 2017).
[5] Y. Cheng, N. Cardwell, S. Hassas Yeganeh, and V. Jacobson. 2017. Delivery Rate Estimation. *IETF Draft* (2017).
[6] M. Claypool. 2018. BBR' - An Implementation of Bottleneck Bandwidth and Round-trip Time Congestion Control for ns-3. Git repository, https://github.com/mark-claypool/bbr.git. (Feb. 2018).
[7] M. Claypool, J. Chung, and F. Li. 2018. *"BBR' – An Implementation of Bottleneck Bandwidth and Round-trip Time Congestion Control for ns-3"*. Technical Report WPI-CS-TR-18-01. Computer Science, Worcester Polytechnic Institute.
[8] S. Floyd. 1999. The New Reno Modification to TCP's Fast Recovery Algorithm. *IETF Request for Comments (RFC) 2582* (April 1999).
[9] S. Ha, I. Rhee, and L. Xu. 2008. CUBIC: a New TCP-friendly High-speed TCP Variant. *ACM SIGOPS Operating Systems Review* 42, 5 (2008), 64–74.
[10] B. Levasseur, M. Claypool, and R. Kinicki. 2014. A TCP CUBIC Implementation in ns-3. In *Proceedings of the Workshop on ns-3 (WNS3)*. Atlanta, Georgia, USA.
[11] F. Li, J. Chung, X. Jiang, and M. Claypool. 2018. TCP CUBIC versus BBR on the Highway. In *Proceedings of the Passive and Active Measurement Conference (PAM)*. Berlin, Germany.
[12] I. Rhee, L. Xu, S. Ha, A. Zimmermann, L. Eggert, and R. Scheffenegger. 2017. CUBIC for Fast Long-Distance Networks. *IETF Draft draft-zimmerman-tcpm-cubic-06* (Sept. 2017).