WPI-CS-TR-18-01 January 2018

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

by

Mark Claypool
Jae Chung
and Feng Li

Computer Science Technical Report Series

WORCESTER POLYTECHNIC INSTITUTE

Computer Science Department
100 Institute Road, Worcester, Massachusetts 01609-2280

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

Mark Claypool
Computer Science Department, Worcester Polytechnic Institute
100 Institute Road, Worcester, MA, 01609, USA

Jae Won Chung and Feng Li
Verizon Labs
60 Sylvan Rd, Waltham, MA, 02145, USA

March 16, 2018

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 current 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 TCP implementations in a fashion similar to other TCP congestion control algorithms, making BBR' extensible and re-using existing interconnection mechanisms. Preliminary validation shows BBR' behaves and performs similarly to BBR, and preliminary performance evaluation shows BBR' has slightly higher throughputs and significantly lower round-trip times than CUBIC in some wired and 4G LTE wireless scenarios.

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 packet loss in modern networks may not indicate congestion but rather corrupt signals over wireless. In addition, TCP determines

*This work is sponsored by the Verizon Labs and WPI. Opinions, interpretations, conclusions and recommendations are those of the authors.

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 modular enough to be 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 [Flo99] and TCP CUBIC [HRX08], today's dominant TCP version of TCP. Most versions of TCP have been thoroughly vetted through analysis, simulation and implementation before being widely deployed. Thus, there is an opportunity to improve TCP performance over today's varied networks by implementing a new congestion control algorithm and testing it in a variety of environments.

A recently proposed congestion control algorithm is Bottleneck Bandwidth and Round Trip (BBR) for TCP [CCG+16]. 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. Compared to loss-based congestion control algorithms such as Reno [MAB09] or CUBIC [RXH+17], 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 at `google.com` and YouTube Web servers.

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, and mobile end users.

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). The ns-family simulators are discrete-event simulators meant to provide for rapid, yet accurate, design, development and evaluation of network protocols. Ns version 2[2] (ns-2) has been the *de facto* simulator for network researchers for decades, while ns version 3[3] (ns-3) extends ns-2 by providing more fidelity to wireless links, such as 4G LTE. Unfortunately, there is not yet an implementation of BBR for ns-3, 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 module separate 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.

[1] `https://patchwork.ozlabs.org/patch/671069/`

[2] `http://nsnam.sourceforge.net/wiki/index.php/User_Information`

[3] `https://www.nsnam.org/overview/what-is-ns-3/`

Validation shows BBR' behaves as expected under controlled simulation conditions, and comparison with published BBR results [CCG+16, CCG+17] 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 slightly higher 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 **??** describes the latest BBR' code updates; Section 6 summarizes our conclusions; and Section 7 presents possible future work.

2 BBR

BBR [CCYJ17a] 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.

In order 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 the round-trip time, 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 the bottleneck bandwidth, BBR keeps a window of the estimates of the receiver delivery rate [CCYJ17b] 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, or the BDP via $BDP = B_{max} \times R_{min}$.

Every time BBR receives a packet acknowledgment, it estimates the round-trip (R_t) and the receiver delivery rate (B_t) for that packet.[4] It then adds R_t and B_t to the round-trip time and bandwidth windows, respectively.

BBR paces sending packets at a rate that matches estimated delivery rate at the receiver (B_{max}) – the pacing rate is BBR's primary control parameter. However, BBR also computes the BDP (based on sub-computations of R_{min} and B_{max}) and allows the TCP congestion window to grow to a multiple of the BDP BBR then compares the number of bytes in flight to the BDP.

BBR applies 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 does go 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.

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

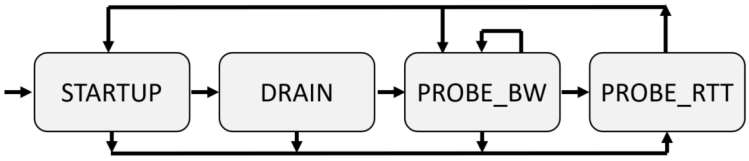


Figure 1: BBR State Transition Diagram

In general:

1. A BBR flow starts in the STARTUP state, and ramps up its sending rate quickly. It 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 built-up queue. While in DRAIN, BBR reduces the pacing rate to $B_{max} \times \frac{ln(2)}{2}$, but keeps the congestion window high. BBR keeps draining long enough to remove the built-up queue, then enters PROBE_BW.

3. In steady state, a BBR flow primarily uses the PROBE_BW state, sending at the bottleneck rate, but repeatedly probing and attempting to fully utilize and additional network bandwidth, all while maintaining a small, bounded queue. PROBE_BW does this by cycling 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. Each cycle phase lasts for one round-trip time. 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 matches or decreases the minimum round-trip time (R_{min}) for 10 seconds, then it briefly enters the PROBE_RTT state to quickly greatly reduce the packets inflight (by 98%) 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 greater.

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.

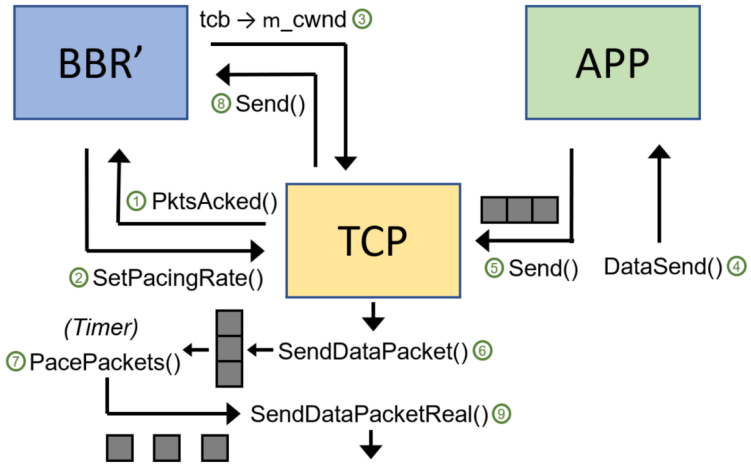


Figure 2: Overview of BBR' Control Flow

3 BBR'

BBR' (pronounced *BBR-prime*) is an implementation of BBR for ns-3.

This section (and all results in Section 4 and Section 5) are current as of BBR' version 1.6.

The most recent changes to the BBR' code are described in Section **??**.

3.1 Overview

For a visual overview of BBR', 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 ①calls 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}), ②sets the pacing rate via `SetPacingRate()`, and ③sets the TCP congestion window via `tcb->m_cWnd`.

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

TCP's `SendDataPacket()` ⑥enqueues 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 ⑦triggers `PacePackets()` at the pacing interval. `PacePackets()` first ⑧invokes BBR' `Send()` which records information needed to estimate the BW (in `PktsAcked()`). Then, `PacePackets()` ⑨sends the next packet in the queue via `SendDatapacketReal()`, resetting the timer to achieve paced sending.

In terms of code, BBR' sits at the same software layer as other TCP congestion control

"flavors", 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 – they all share the same TCP state machine, which allows for establishing connections through a three-way handshake, retransmissions upon 3 duplicated acknowledgments, flow control through acknowledgments, and connection termination. Moreover, BBR' flows can interface with lower layers, such as the IP layer, without requiring special code.

Where the TCP flavors, such as BBR', primarily differ is after a connection is established and the congestion control mechanism takes effect. As of version 3.26, ns-3 separates the TCP base class (`TcpSocketBase`) from the TCP congestion control class (`TcpCongestionOps`).[5] This allows the base class to maintain important TCP variables (e.g., the window of data segments) while the sub-classes of `TcpCongestionOps` can specify their individual responses to congestion (i.e., their congestion control algorithms). All operations that are delegated to congestion control are contained in the class `TcpCongestionOps`. This class mimics the Linux `tcp_congestion_ops` structure, defining the methods shown in Listing 1.

Listing 1: TcpCongestionOps Methods

```
// Get name of congestion control algorithm.
virtual std::string GetName() const;

// Get slow start threshold after loss event.
virtual uint32_t GetSsThresh(Ptr<const TcpSocketState> tcb,
                             uint32_t bytesInFlight);

// Change window during congestion avoidance.
virtual void IncreaseWindow(Ptr<TcpSocketState> tcb,
                            uint32_t segmentsAcked);

// Copy congestion control algorithm across socket (book-keeping).
virtual Ptr<TcpCongestionOps> Fork();

// Called every time ACK is received (optional).
virtual void PktsAcked(Ptr<TcpSocketState> tcb,
                       uint32_t segmentsAcked, const Time& rtt);

// Called before changing congestion state (mimics Linux, optional).
virtual void CongestionStateSet (Ptr<TcpSocketState> tcb,
                     const TcpSocketState::TcpCongState_t newState);
```

3.2 BBR' Class Definition

For syntax convention, BBR' variables use snake_case (e.g., `rtt_window`), BBR' methods use camelCase (e.g., `changeState()`), and BBR' attributes are prefixed with m_ (e.g., `m_state`).

BBR' also defines the namespace `bbr` used for BBR'-specific data structures and constants.

BBR' defines the protocol states as an `enum` in a separate namespace, as shown in Listing 2. STARTUP_STATE, DRAIN_STATE, PROBE_BW_STATE, and PROBE_RTT_STATE correspond to the respective BBR states (see Section 2).

[5] This modularity is similar to that in Linux.

Listing 2: BBR' Protocol States

```
namespace bbr {
  enum bbr_state {
    UNDEFINED_STATE=-1,
    STARTUP_STATE,
    DRAIN_STATE,
    PROBE_BW_STATE,
    PROBE_RTT_STATE,
  };
}
```

BBR's major parameters are defined via the constants shown in Listing 3. The parameters are fairly self-explanatory based on the BBR specification (see Section 2).

Listing 3: BBR' Constants

```
namespace bbr {
  const Time INIT_RTT = Time(10000000); // Nanoseconds (.010 sec).
  const double INIT_BW = 6.0;           // Mb/s.
  const int RTT_WINDOW_TIME = 10;       // In seconds.
  const int BW_WINDOW_TIME = 10;        // In RTTs.
  const int MIN_CWND = 4;               // In packets.
  const float PACING_FACTOR = 1.0;      // For pacing tuning.

  // PROBE_BW state:
  // Gain rate when BW probing: [1.25, 0.75, 1, 1, 1, 1, 1, 1]
  const float STEADY_FACTOR = 1.0;      // Steady rate adjustment.
  const float PROBE_FACTOR = 0.25;      // Add when probe.
  const float DRAIN_FACTOR = 0.25;      // Decrease when drain.

  // STARTUP state:
  const float STARTUP_THRESHOLD = 1.25; // Threshold to exit STARTUP.
  const float STARTUP_GAIN = 2.89;      // Roughly 2/ln(2).

  // To enter PROBE_RTT state:
  const float RTT_NOCHANGE_LIMIT = 10;  // In seconds.
}
```

BBR' defines the structure in Listing 4 for the book-keeping used in estimating the receiver delivery rate (bottleneck bandwidth) at the sender. The BBR' class definition records a `packet_struct` for each packet sent and received. BBR' computes the bottleneck bandwidth in `PktsAcked()` (see Section 3.4) and stores the result in a `struct bw_struct`. See Section 3.3 for details on the bandwidth estimation algorithm.

Listing 4: BBR' Bandwidth Estimation Structure

```
namespace bbr {
  // Structure for tracking TCP window for estimating BW.
  struct packet_struct {
    SequenceNumber32 acked;  // Last sequence number acked.
    SequenceNumber32 sent;   // Next sequence number sent.
    Time time;               // Time sent.
    int delivered;           // Delivered bytes.
  };
```

```
8
9     // Structure for storing BW estimates.
10    struct bw_struct {
11      Time time;                // Time stored.
12      int round;                // Virtual time stored.
13      double bw_est;            // Bandwidth estimate.
14    };
15  }
```

The main BBR' class definition shown in Figure 5, is, as described above, derived from the ns-3 class `TcpCongestionOps`.

The attributes from Lines 3-20 are primarily to manage the protocol state and the bandwidth and round-trip time windows over time. The attributes `m_min_rtt_change` and `m_bytes_in_flight` are used to determine when to enter PROBE_RTT and exit PROBE_RTT, respectively. The attribute `m_packet_conservation`, is used to modulate the congestion window during Fast Recovery. The attributes `m_in_retrans_seq` and `m_retrans_seq` are used to ignore retransmission sequences when doing bandwidth estimations. The attributes `m_round`, `m_delivered`, and `m_next_round_delivered` are used for computing packet-timed RTTs (versus wall-clock RTTs).

The `friend` class declarations from Lines 23-28 are to support the state machine and states that control the protocol behavior. See Section 3.8 for details.

The constructor `TcpBbr()` initializes parameters, mostly setting the round-trip time and bandwidth windows to zero length and the protocol state to `STARTUP_STATE`. Other parameters (e.g., `m_pacing_gain`), are all zeroed.

The method `Fork()` is only for ns-3 internal functionality, mechanically making a copy of the Object. Similarly, the ns-3 base class `TcpCongestionOps` requires definition of the methods `IncreaseWindow()` and `GetSsThresh()`, but these methods perform no meaningful functionality in BBR' since the congestion window is manually controlled when sending (see Section 3.5).

The methods `PktsAcked()`, `Send()` and `CongestionStateSet()` have the primary congestion-response duties, and are described in Section 3.4, Section 3.5, and Section 3.6, respectively.

The private methods `getRTT()`, `getBW()`, `cullRTTwindow()`, `cullBWwindow()`, `getBDP()` and `updateTargetCwnd()`, provide support for `PktsAcked()` and `Send()` (see Section 3.7).

The private method `checkProbeRTT()` checks if BBR' should transition to the PROBE_RTT state.

Listing 5: BBR' Class Definition (TcpBbr)

```
0 class TcpBbr : public TcpCongestionOps {
1
2  protected; // Attributes.
3    double m_pacing_gain;             // Scale estimated BDP for pacing.
4    double m_cwnd_gain;               // Scale estimated BDP for cwnd.
5    int m_round;                      // For recording virtual RTT time.
6    int m_delivered;                  // For computing virtual RTT rounds.
7    int m_next_round_delivered;       // For computing virtual RTT rounds.
8    std::map<Time, Time> m_rtt_window;// For computing min RTT.
9    std::vector<bbr::bw_struct> m_bw_window; // For computing max BW.
10   std::vector<bbr::packet_struct> m_pkt_window; // For est BW frm ACKs.
```

```
  uint32_t m_bytes_in_flight;      // Bytes in flight (in socket base).
  Time m_min_rtt_change;           // Last time min RTT changed.
  Time m_packet_conservation;      // Time to stop modulation.
  bool m_in_retrans_seq;           // True if in retrans seq.
  SequenceNumber32 m_retrans_seq;  // Retrans seq end.
  BbrStateMachine m_machine;       // State machine.
  BbrStartupState m_state_startup; // STARTUP state.
  BbrDrainState m_state_drain;     // DRAIN state.
  BbrProbeBWState m_state_probe_bw;// PROBE_BW state.
  BbrProbeRTTState m_state_probe_rtt; // PROBE_RTT state.

 public:  // Friends.
  friend class BbrState;
  friend class BbrStartupState;
  friend class BbrDrainState;
  friend class BbrProbeBWState;
  friend class BbrProbeRTTState;
  friend class BbrStateMachine;

 public:  // Methods.
  // Default constructor.
  TcpBbr();

  // On receiving ack:
  // - update congestion window
  // - store RTT
  // - compute and store estimated BW
  // - compute and set pacing rate
  virtual void PktsAcked(Ptr<TcpSocketState> tcb, uint32_t packets_acked,
                         const Time& rtt);

  // Before sending packet:
  // - Record information to estimate BW
  virtual void Send(Ptr<TcpSocketBase> tsb, Ptr<TcpSocketState> tcb,
                    SequenceNumber32 seq, bool isRetrans);

  // Copy BBR' congestion control across socket.
  Ptr<TcpCongestionOps> Fork();

  // BBR' ignores calls to increase window.
  void IncreaseWindow(Ptr<TcpSocketState> tcb, uint32_t segmentsAcked);

  // BBR' does not use ssthresh, so ignored.
  uint32_t GetSsThresh(Ptr<const TcpSocketState> tcb,
                       uint32_t bytesInFlight);

 private: // Methods.
  double getBDP() const;  // Return bandwidth-delay product (in Mbits).
  Time getRTT() const;    // Return RTT (min of window, in seconds).
  double getBW() const;   // Return bandwidth (max of window, in Mb/s).
  void cullBWwindow();    // Remove old BW estimates (10+ RTTs).
  void cullRTTwindow();   // Remove old RTT estimates (10+ sec).
  void updateTargetCwnd();// Compute target TCP cwnd.
  bool checkProbeRTT();   // Check if should enter PROBE_RTT state.
```

```
65
66 };
```

3.3 Bandwidth Estimation

In order to obtain an estimate of the bottleneck bandwidth, BBR' infers the receiver's delivery rate at the sender using an algorithm based on Cheng et al. [CCYJ17b]. Broadly, the BBR' bandwidth estimation algorithm works as follows:

- When sending a packet, BBR' (and TCP, in general) has a window of not-yet acknowledged packets [1...n], with W_a representing the latest packet successfully acknowledged and W_s the next packet to be sent. i.e., A window like: W_a $[W_{a+1}...W_{s-1}]$ W_s.
- For each packet W_s BBR' sends, BBR' records W_s, W_a and the current time, storing this information in a 3-tuple, one for each outstanding packet.
- When an acknowledgment for a packet arrives, BBR' computes how many bytes have been successfully delivered at the receiver since the packet was sent as well as the elapsed time (in seconds) since it was sent.
- This, in turn, is used to estimate the bandwidth at time t':

$$b_{t'} = \frac{W_s - W_a}{W_{t'} - W_t} \tag{1}$$

  where the units are bytes per second.
- Note, since TCP has ambiguous ACKs during retransmissions, BBR' skips bandwidth estimates for retransmitted sequence ranges (see Section 4.2 of Cheng et al. [CCYJ17b]).

3.4 PktsAcked()

`Bbr::PktsAcked()` does the main "work" in BBR', computing parameters necessary for congestion control.

In general, when a TCP BBR' session receives a TCP ACK, ns-3 calls `Bbr::PktsAcked()`, providing an estimate of the round-trip time in the parameter `rtt` a pointer to the TCP transmission control block (tcb). BBR' updates the congestion window, taking care if in Fast Recovery or RTO. The RTT is stored (used for computing R_{min}). The bandwidth is estimated (used for computing B_{max}). And the pacing rate is computed and set in the TCP transmission control.

Listing 6 depicts the BBR' implementation of `Bbr::PcktsAcked()`. Note, for this code listing and all subsequent listings, some non-essential functionality has been removed (e.g., logging functions and some error checking) to improve readability. See the associated `git` repository for the complete code [Cla18].

Lines 12-28 show the congestion window modifications. If in Fast Recovery, BBR' ensures packet conservation on the first round of recovery, and sends at no more than twice the

current delivery rate on later rounds while still in recovery. If not in Fast Recovery, BBR' computes the target congestion window (roughly, the BDP - see Listing 12).

Lines 34-37, check if the new RTT estimate coming in is less than the previous minimum. If so, it notes the time. This impacts when BBR' might enter the PROBE_RTT state (see Section 3.8).

Line 40 stores the round-trip time estimate in a window (a vector `m_rtt_window`), which, based on the BBR specification [CCYJ17a], stores the past 10 seconds worth of round-trip time estimates.

In Line 43, if this is the first RTT received by the BBR' session, the state machine is updated to schedule an ns-3 timer (see Section 3.8 for details on `update()`).

Lines 47-52 compute the virtual (non-wall-clock) time tracked by the round (`m_round`) giving a count of "packet-timed" round-trips, as per the BBR specification [CCYJ17a]. Packet-timed round trips are computed by recording state as a "sentinel" when a packet goes out, and then when it is ACKed, that provides a round-trip time (round).

Lines 60-62 check if the incoming ACK is less than the first (and the smallest, since the list is sorted) in the bandwidth estimation window (`m_est_window`). This primarily happens during a retransmission sequence, but can also happen if the `TcpSocketBase` method `SendPendingData()` is invoked without going through the BBR' `Send()` method that records the latest ACK information for the outgoing packet (see Section 3.5). When this happens, BBR' is unable to determine a bandwidth estimate so it is ignored (see Section 3.9.2 for details on pacing).

Lines 65-75 find the matching ACK in the bandwidth estimation window and then remove all entries up to and including this one.

Lines 77-82, if not in a retransmission sequence, compute the estimated delivery rate at the receiver based on the time elapsed between sending a packet and it's ACK and the amount of data acknowledged in the interim. See Section 3.3 for a description of the algorithm.

BBR' stores the bandwidth estimate (`bw_est`) in a window (`m_bw_window`, a map that records the time-bandwidth pair), which, based on the BBR specification [CCYJ17a], includes 10 round-trip times worth of estimates.

Lastly, BBR' computes the pacing rate, adjusted by the gain (Line 92), and then sets the pacing rate (Line 93) in the TCP socket state (see Section 3.9.2 for details).

Listing 6: TcpBbr::PktsAcked()

```
0  // On receiving ack:
1  // - update congestion window
2  // - store RTT
3  // - compute and store estimated BW
4  // - compute and set pacing rate
5  void TcpBbr::PktsAcked(Ptr<TcpSocketState> tcb, uint32_t packets_acked,
6                         const Time &rtt) {
7
8    ////////////////////////////////////////
9    // UPDATE TCP CONGESTION WINDOW
10
11   // If in Fast Recovery, target cwnd was set in CongestionStateSet().
12   if (tcb->m_congState == TcpSocketState::CA_RECOVERY) {
13     // If in first RTT of Fast Recovery, modulate cwnd.
```

```
    if (m_packet_conservation > Simulator::Now()) {
      if ((m_bytes_in_flight + bytes_delivered) > m_cwnd)
        m_cwnd = m_bytes_in_flight + packets_acked * 1500;
    }
  } else {
    // Not in Fast Recovery, so re-compute target cwnd.
    updateTargetCwnd();
  }

  // If growing cwnd, do so conservatively.
  if (tcb -> m_cWnd < m_cwnd)
    tcb -> m_cWnd = tcb -> m_cWnd + bytes_delivered;
  else
    // If shrinking cwnd, adjust immediately.
    tcb -> m_cWnd = m_cwnd;

  ////////////////////////////////////////
  // STORE RTT

  // See if changed minimum (track to decide when to PROBE_RTT).
  Time now = Simulator::Now();
  Time min_rtt = getRTT();
  if (rtt < getRTT())
    m_min_rtt_change = now;

  // Add current RTT to window.
  m_rtt_window.push_back(rtt);

  // Upon first RTT, call update() to initialize timer.
  if (m_rtt_window.size() == 1)
    m_machine.update();

  // Update packet-timed RTT.
  m_delivered += tcb->m_segmentSize;
  auto packet = m_pkt_window.begin();
  if (packet.delivered >= m_next_round_delivered) {
    m_next_round_delivered = m_delivered;
    m_round++;
  }

  ////////////////////////////////////////
  // ESTIMATE BW.
  SequenceNumber32 ack = tcb->m_lastAckedSeq; // W_s
  now = Simulator::Now();                     // W_t'

  // If ACK earlier than first, unknown when sent so ignore.
  auto first = m_bw_window.begin()->sent;
  if (ack < first)
    return;  // Nothing more to do.

  // Find newest ACK in window, <= current.  Note, window is sorted.
  bw_struct packet;
  for (auto it = m_bw_window.begin(); it != m_bw_window.end(); it++)
    if (it->sent <= ack)                      // W_a
```

```
68        packet = *it;
69
70    // Remove all ACKs <= current from window.
71    for (unsigned int i=0; i < m_pkt_window.size(); )
72      if (m_pkt_window[i].sent <= packet.sent)
73        m_pkt_window.erase(m_pkt_window.begin() + i);
74      else
75        i++;
76
77    // Estimate BW: bw = (W_s - W_a) / (W_t' - W_t)
78    if (!m_in_retrans_seq) {
79      double bw_est = (ack - packet.acked) /
80                      (now.GetSeconds() - packet.time.GetSeconds());
81      bw_est *= 8;           // Convert to b/s.
82      bw_est /= 1000000;     // Convert to Mb/s.
83
84      // Add current <Time,BW> to window.
85      m_bw_window[now] = bw_est;
86    }
87
88    ////////////////////////////////////////
89    // COMPUTE AND SET PACING RATE.
90
91    // Set pacing rate (in Mb/s), adjusted by gain.
92    double pacing_rate = getBW() * m_pacing_gain;
93    tsb -> SetPacingRate(pacing_rate);
94  }
```

3.5 Send()

In general, before a TCP BBR' session sends a packet, ns-3 calls `Bbr::Send()`. `Send()` basically stores information about the outgoing sequence and latest ACK so it can be used to compute the BW estimate in `PktsAcked()`.

Listing 7 shows the BBR' implementation of `Bbr::Send()`.

Line 6 records the number of bytes in flight which is used in STARTUP to decide when to exit that state.

Lines 9-lst:bbr-snd-retrans-b do book-keeping to note the start of a retransmission sequence.

Lines 14-21, if not in a retransmission sequence, BBR' does some book-keeping to update the bandwidth estimation data (see Section 3.3 for details on the algorithm).

Listing 7: TcpBbr::Send()

```
0  // Before sending packet:
1  // - Record information to estimate BW
2  void TcpBbr::Send(Ptr<TcpSocketBase> tsb, Ptr<TcpSocketState> tcb,
3                    SequenceNumber32 seq, bool isRetrans) {
4
5    // Get the bytes in flight (needed for STARTUP).
6    m_bytes_in_flight = tsb -> BytesInFlight();
7
```

```
  // If retransmission, start sequence.
  if (isRetrans) {
    m_in_retrans_seq = true;
    m_retrans_seq = seq;
  }

  // If not in retrans sequence, record info for BW est.
  if (!m_in_retrans_seq) {
    // Get last sequence number acked.
    bbr::packet_struct p;
    p.acked = tcb -> m_lastAckedSeq;
    p.sent = tcb -> m_nextTxSequence;
    p.delivered = m_delivered;
    m_pkt_window.push_back(p);
  }
}
```

3.6 CongestionStateSet()

The `CongestionStateSet()` method, in general, performs calculations specific to congestion control algorithm as the protocol changes states. Ns-3 manages the states: CA_OPEN (normal), CA_DISORDER (normal, but has seen some SACKs or dupACKs), CA_RECOVERY (in Fast Recovery) and CA_LOSS (in RTO).

BBR's `CongestionStateSet()` method is shown in Listing 8.

When modifying the congestion window, BBR' sets the member attribute `m_cwnd`, actually changing the TCP congestion window in `PktsAcked()` (see Section 3.4).

When entering Fast Recovery (CA_RECOVERY) or an RTO (CA_LOSS), BBR' saves the current congestion window as the last known "good" value into `m_prior_cwnd`. When BBR' later exits either state, it restores the congestion window to this value.

During an RTO (entering CA_LOSS), BBR' minimizes the congestion window to 1.

When entering Fast Recovery (CA_RECOVERY), BBR' reduces the congestion window to exactly the number of bytes in flight, and modulates the congestion window to not grow too quickly for one round-trip time (controlled by `m_packet_conservation`).

Listing 8: TcpBbr::CongestionStateSet()

```
// Events/calculations specific to BBR' congestion state.
void TcpBbr::CongestionStateSet(Ptr<TcpSocketState> tcb,
          const TcpSocketState::TcpCongState_t new_state) {

  auto old_state = tcb->m_congState;

  // Enter RTO --> minimal cwnd.
  if (new_state == TcpSocketState::CA_LOSS) {
    m_prior_cwnd = m_cwnd;
    m_cwnd = 1;
  }

  // Enter Fast Recovery --> save cwnd, modulate for 1 RTT.
  if (new_state == TcpSocketState::CA_RECOVERY) {
```

```
    m_prior_cwnd = m_cwnd;
    m_cwnd = m_bytes_in_flight + 1;
    m_packet_conservation = Simulator::Now() + getRTT();
  }

  // Exit RTO or Fast Recovery --> restore cwnd.
  if ((old_state == TcpSocketState::CA_RECOVERY ||
       old_state == TcpSocketState::CA_LOSS) &&
      (new_state != TcpSocketState::CA_RECOVERY &&
       new_state != TcpSocketState::CA_LOSS)) {
    m_packet_conservation = Simulator::Now(); // Stop packet conservation
        .
    if (m_prior_cwnd > m_cwnd)
      m_cwnd = m_prior_cwnd;
  }
}
```

3.7 Support Methods

BBR' has several methods to support `PktsAcked()` and `Send()`.

Listing 9 shows `getBDP()` that computes and returns the bandwidth-delay product (BDP). In most cases, the BDP computation is simply the maximum observed bandwidth multiplied by the minimum observed round-trip time. However, in the case of the first call to compute the BDP (see Section 3.5), there is not yet an estimate for round-trip time nor the bandwidth, so each is given a small, preset value (see Section 4.2.1 of [CCYJ17a]). All subsequent calls use the round-trip times and bandwidth estimates obtained after the first ACK.

Listing 9: TcpBbr::getBDP()

```
// Return bandwidth-delay product (in Mbits).
double TcpBbr::getBDP() const {

  Time rtt = getRTT();
  if (rtt.IsNegative())
    rtt = bbr::INIT_RTT;

  double bw = getBW();
  if (bw < 0)
    bw = bbr::INIT_BW;

  return (double) (rtt.GetSeconds() * bw);
}
```

Listing 10 shows `getRTT()` that computes and returns the round-trip time. In most cases, the round-trip time is computed as the minimum value in the round-trip time estimate window (`m_rtt_window`). The exception is when there are no round-trip time estimates yet, and then -1.0 is returned.

Listing 10: TcpBbr::getRTT()

```
// Return round-trip time (minimum of window, in seconds).
// Return -1 if no RTT estimates yet.
```

```
Time TcpBbr::getRTT() const {
  Time min_rtt = Time::Max();

  if (m_rtt_window.size() == 0)
    min_rtt = Time(-1.0); // If no RTT estimates yet.
  else
    // Find minimum RTT in window.
    for (auto it = m_rtt_window.begin(); it != m_rtt_window.end(); it++)
      min_rtt = std::min(min_rtt, *it);

  return min_rtt;
}
```

Listing 11 shows `getBW()` that computes and returns the bandwidth (BW). In most cases, the BW is computed as the maximum value in the BW estimate window (`m_bw_window`). The exception is when there are no BW estimates yet and -1.0 is returned.

Listing 11: TcpBbr::getBW()

```
// Return bandwidth (maximum of window, in Mb/s).
// Return -1 if no BW estimates yet.
double TcpBbr::getBW() const {
  double max_bw = 0;

  if (m_bw_window.size() == 0)
    max_bw = -1.0;  // If no BW estimates yet.
  else
    // Find max BW in window.
    for (auto it = m_bw_window.begin(); it != m_bw_window.end(); it++)
      max_bw = std::max(max_bw, it->second);

  return max_bw;
}
```

Listing 12 shows `updateTargetCwnd()` that computes the target congestion window (called by `PktsAcked()`, see Section 3.4). Basically, the target congestion window (`m_cwnd`) is set to the BDP, modified by the congestion window gain (`m_cwnd_gain`). A final check makes sure the congestion window is not too small.

Listing 12: TcpBbr::updateTargetCwnd()

```
// Compute target TCP cwnd (m_cwnd) based on BDP and gain.
void TcpBbr::updateTargetCwnd() {

  double bdp = getBDP();
  m_cwnd = bdp * m_cwnd_gain;
  m_cwnd = (m_cwnd * 1000000 / 8); // Mbits to bytes.

  // Make sure cwnd not too small (roughly, 4 packets).
  if ((m_cwnd / 1500) < bbr::MIN_CWND)
    m_cwnd = bbr::MIN_CWND * 1500; // In bytes.
}
```

Listing 13 shows `checkProbeRTT()` that determines if BBR' should enter the PROBE_RTT state. Basically, if currently in the PROBE_BW state and the minimum RTT has not

changed in RTT_NOCHANGE_LIMIT second (default is 10), this method returns true. See Section 3.8 for how this method is used in state transitions.

Listing 13: TcpBbr::checkProbeRTT()

```
// Return true if should enter PROBE_RTT state.
bool TcpBbr::checkProbeRTT() {

  // If in PROBE_BW and min RTT hasn't changed in 10 seconds.
  Time now = Simulator::Now();
  if (m_machine.getStateType() == bbr::PROBE_BW &&
      (now.GetSeconds() - m_min_rtt_change.GetSeconds()) >
       RTT_NOCHANGE_LIMIT) {
    m_min_rtt_change = now;
    return true;
  }

  return false;
}
```

3.8 BBR' States

BBR' transitions through the BBR states as specified in Section 2, with the slight exception of transitions to and from the PROBE_RTT state. BBR' only transitions to/from the PROBE_BW state, whereas BBR may transition to the PROBE_BW state at any time and may also return to the STARTUP state under some conditions (see Future work in Section 7).

A state machine class, BbrStateMachine, drives BBR' through state transitions and executions, shown in Listing 14.

Listing 14: BbrStateMachine.h

```
class BbrStateMachine : public Object {

 public:
  static TypeId GetTypeId();       // Get type id.
  std::string GetName() const;     // Get name of object.
  BbrStateMachine();               // Default constructor.
  BbrStateMachine(TcpBbr *owner); // Constructor with BBR' owner.
  bbr_state getStateType() const; // Get type of current state.
  void changeState(BbrState *p_new_state); // Change state.
  void update();                   // Update by executing current state.

 private:
  BbrState *m_state;               // Current state.
  TcpBbr *m_owner;                 // BBR' flow that owns machine.
};
```

The state machine update() method, shown in Listing 15, is invoked every RTT. First, BBR' checks if it should enter the PROBE_RTT state, as detailed in Listing 13. Then, the current state is executed, followed by book-keeping to update (cull) the round-trip time and bandwidth windows. Lastly, BBR' schedules the next update() method call, typically once per RTT (or when the first ACK arrives, providing the first RTT).

Listing 15: BbrStateMachine::update()

```
// Update by executing current state.
void BbrStateMachine::update() {

  // Check if should enter PROBE_RTT.
  if (m_owner -> checkProbeRTT())
    changeState(&m_owner -> m_state_probe_rtt);

  // Execute current state.
  m_state -> execute();

  // Cull RTT window.
  m_owner -> cullRTTwindow();

  // Cull BW window.
  m_owner -> cullBWwindow();

  // Schedule next event (if we can).
  Time rtt = m_owner -> getRTT();
  if (!rtt.IsNegative())
    Simulator::Schedule(rtt, &BbrStateMachine::update, this);
  // else update() called in PktsAcked() upon getting first ACK.
}
```

The state machine `changeState()` method, shown in Listing 16, calls `exit()` on the old state, changes the state (`m_state`), and calls `enter()` on the new state.

Listing 16: BbrStateMachine::changeState()

```
// Change current state to new state.
void BbrStateMachine::changeState(BbrState *new_state) {
  m_state -> exit();    // Exit old state.
  m_state = new_state; // Change to new state.
  m_state -> enter();   // Enter new state.
}
```

The state machine drives classes derived from the base `BbrState` class shown in Listing 17.

Listing 17: BbrState.h

```
class BbrState : public Object {

 public:
  static TypeId GetTypeId();              // Get type id.
  virtual std::string GetName() const; // Get name of object.
  BbrState();                             // Default constructor.
  BbrState(TcpBbr *owner);                // Constructor with BBR' owner.
  virtual ~BbrState();                    // Destructor.
  virtual bbr_state getType() const=0; // Get state type.
  virtual void enter();                   // When state first entered.
  virtual void execute()=0;               // When state updated.
  virtual void exit();                    // When state exited.

 protected:
  TcpBbr *m_owner;                        // BBR' flow that owns state.
```

```
};
```

BBR' derives a separate class for each state type: `BbrStartupState`, `BbrDrainState`, `BbrProbeBWState`, and `BbrProbeRTTState`. For brevity, those class definitions are not shown here.

BBR' STARTUP is intended to act much like TCP slowstart, quickly ramping up to the bottleneck bitrate. It does this by setting the gain rate so as to approximately double the bandwidth each round-trip time.

Listing 18 depicts the actions BBR' takes when entering the STARTUP state, which is to set the gain rates to $\frac{2}{ln(2)}$, as per the BBR specification [CCYJ17a].

Listing 18: BbrStartupState::enter()

```
// Invoked when state first entered.
void BbrStartupState::enter() {
  // Set gains to 2/ln(2).
  m_owner -> m_pacing_gain = bbr::STARTUP_GAIN;
  m_owner -> m_cwnd_gain = bbr::STARTUP_GAIN;
}
```

When operating in STARTUP (`BbrStartupState::execute()`), shown in Listing 19, BBR' checks if 3+ consecutive bandwidth estimates have increased by less than 25%. If so, BBR' exits the STARTUP state by setting the next state to be the DRAIN state. Otherwise, BBR' remains in the STARTUP state.

Listing 19: BbrStartupState::execute()

```
// Invoked when state updated.
void BbrStartupState::execute() {

  // Get current BW (new).
  double new_bw = m_owner -> getBW();

  // Still growing?
  if (new_bw > m_full_bw * bbr::STARTUP_THRESHOLD) {
    m_full_bw = new_bw;
    m_full_bw_count = 0;
    return;
  }

  // If 3+ rounds w/out much growth, STARTUP --> DRAIN.
  m_full_bw_count++;
  if (m_full_bw_count >= 3)
    m_owner -> m_machine.changeState(&m_owner -> m_state_drain);
}
```

BBR' DRAIN is intended to drain the excess queue that was built up during the STARTUP state as the sending rate was increased beyond the bottleneck rate.

Listing 20 depicts the actions BBR' takes when entering the DRAIN state, which is to set the pacing gain rate (`m_pacing_gain`) to $\frac{ln(2)}{2}$ while maintaining a high cwnd gain (`m_cwnd_gain`), as per the BBR specification [CCYJ17a].

Next, since BBR' will have over-shot the bandwidth target, BBR' obtains the lower target number of inflight bytes. It uses this to determine when to exit the DRAIN state when executing.

Listing 20: BbrDrainState::enter()

```
// Invoked when state first entered.
void BbrDrainState::enter() {

  // Set gain to 1/[2/ln(2)].
  m_owner -> m_pacing_gain = 1 / bbr::STARTUP_GAIN;
  m_owner -> m_cwnd_gain = bbr::STARTUP_GAIN;

  // Get BDP for target inflight limit.
  double bdp = m_owner -> getBDP();
  bdp = bdp * 1000000 / 8; // Convert to bytes.
  m_inflight_limit = (uint32_t) bdp;
  m_round_count = 0;
}
```

When operating in DRAIN (`BbrDrainState::execute()`), shown in Listing 21, BBR' checks if DRAIN should exit based on two conditions. The primary condition is to exit when the number of bytes in flight is less than the limit (computed when DRAIN started). A secondary condition is after 5 rounds (RTTs). While the BBR' specification [CCYJ17a] indicates any built up queue should be drained in one round (RTT), the 1 / STARTUP_GAIN drain rate means up to 5 RTTs[6]) are likely required.

Upon exiting, BBR' enters the PROBE_BW state.

Listing 21: BbrDrainState::execute()

```
// Invoked when state updated.
void BbrDrainState::execute() {

  m_round_count++;

  // Exit when byte-in-flight under limit OR 5 rounds, whichever is first
     .
  if (m_owner -> m_bytes_in_flight < m_inflight_limit ||
      m_round_count == 5)
    m_owner -> m_machine.changeState(&m_owner -> m_state_probe_bw);
}
```

BBR' PROBE_BW periodically (once every 8 RTTs) increases the data rate to see if the delivery rate (BW) has increased.

Listing 22 depicts the actions BBR' takes when entering the PROBE_BW state – namely, to pick a random, non-"low" phase for the gain cycles. As described in the BBR specification [CCYJ17a], this randomization is to avoid having BBR flows that enter PROBE_BW simultaneously all having a the same "high" gain. Based on the phase, the `m_pacing_gain` is set accordingly, followed by the `m_cwnd_gain`.

Listing 22: BbrProbeBWState::enter()

[6]The time to drain the overshoot, $\frac{2.89}{(1-\frac{1}{2.89})}$, is about 4.5

```
// Invoked when state first entered.
void BbrProbeBWState::enter() {
  // Pick random cycle phase (except "low") to start to avoid
  // flows that enter PROBE_BW at same time all being "high".
  do {
    m_gain_cycle = rand() % 8;
  } while (m_gain_cycle == 1);  // Phase 1 is "low" cycle.

  // Set pacing gain according to cycle.
  if (m_gain_cycle == 0) // Phase 0 is "high" cycle.
    m_owner -> m_pacing_gain = bbr::STEADY_FACTOR + bbr::PROBE_FACTOR;
  else
    m_owner -> m_pacing_gain = bbr::STEADY_FACTOR;
  m_owner -> m_cwnd_gain = 2 * bbr::STEADY_FACTOR;
}
```

When operating in PROBE_BW (`BbrProbeBWState::execute()`), shown in Listing 23, as per the BBR specification [CCYJ17a], BBR' cycles through `m_gain` rates, looking for potential bandwidth changes. The increase ("high") probes for more bandwidth followed immediately by a decrease ("low") to drain off any queue occupancy incurred if there was no increase and then 6 periods of steady state ("stdy").

Listing 23: BbrProbeBWState::execute()

```
// Invoked when state updated.
void BbrProbeBWState::execute() {
  // Set gain rate: [high, low, stdy, stdy, stdy, stdy, stdy, stdy]
  if (m_gain_cycle == 0)
    m_owner -> m_gain = bbr::STEADY_FACTOR + bbr::PROBE_FACTOR;
  else if (m_gain_cycle == 1)
    m_owner -> m_gain = bbr::STEADY_FACTOR - bbr::DRAIN_FACTOR;
  else
    m_owner -> m_gain = bbr::STEADY_FACTOR;

  // Move to next cycle, wrapping.
  m_gain_cycle++;
  if (m_gain_cycle > 7)
    m_gain_cycle = 0;
}
```

BBR' PROBE_RTT is intended to: 1) check for a new minimum RTT that perhaps cannot be seen by the sender at the BDP rates, and 2) provide fairness for simultaneous BBR' flows that arrive later.

Listing 24 depicts the actions BBR' takes when entering the PROBE_RTT state. Specifically, BBR' sets the pacing gain (`m_pacing_gain`) and cwnd gain (`m_cwnd_gain`) to a steady value (the actual sending rate is set in `Send()` – see Listing 7). Then, BBR' computes how long to remain in PROBE_RTT, either one RTT or 0.2 seconds, whichever is larger.

Listing 24: BbrProbeRTTState::enter()

```
// Invoked when state first entered.
void BbrProbeRTTState::enter() {
  // Set gain (Send() will minimize window);
```

```
3     m_owner -> m_pacing_gain = bbr::STEADY_FACTOR;
4     m_owner -> m_cwnd_gain = bbr::STEADY_FACTOR;
5
6     // Schedule when exit: max (0.2 seconds, min RTT).
7     Time rtt = m_owner -> getRTT();
8     if (rtt.GetSeconds() > 0.2)
9       m_probe_rtt_time = rtt;
10    else
11      m_probe_rtt_time = Time(0.2 * 1000000000);
12    m_probe_rtt_time = m_probe_rtt_time + Simulator::Now();
13  }
```

When operating in PROBE_RTT (`BbrProbeRTTState::execute()`), shown in Listing 25, BBR' merely checks whether enough time has elapsed (comparing the current time to `m_probe_rtt_time`). If so, BBR' changes state back to the PROBE_BW state.

Listing 25: BbrProbeRTTState::execute()

```
0  // Invoked when state updated.
1  void BbrProbeRTTState::execute() {
2
3    // If enough time elapsed, PROBE_RTT --> PROBE_BW.
4    Time now = Simulator::Now();
5    if (now > m_probe_rtt_time)
6      m_owner -> m_machine.changeState(&m_owner -> m_state_probe_bw);
7  }
```

3.9 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 require a total of about 75 new lines of code to three different files. This section describes the required changes, with patches available for ns-3.27 available in the associated `git` repository [Cla18].

3.9.1 Send Support

In order to support controlling the cwnd and pacing rate via BBR' `Send()`, the `TcpCongestionOps` class needs a slight modification. Specifically, an empty virtual method is added to the class definition:

Listing 26: Addition to TcpCongestionOps in tcp-congestion-ops.h (line 125)

```
0    // Enable congestion control-specific Send() functionality
1    // (invoked in TcpSocketBase::SendDataPacketReal()).
2    virtual void Send(Ptr<TcpSocketBase> tsb, Ptr<TcpSocketState> tcb,
3                      SequenceNumber32 seq, bool isRetrans)
4    { /* Empty*/  }
```

and a call to `Send()` added at the top of `TcpSocketBase::SendDataPacketReal()` (see Section 3.9.2):

Listing 27: Addition to TcpCongestionOps in tcp-socket-base.cc

```
  // Hook for congestion control-specific Send() method.
  // (Added for BBR' support.)
  m_congestionControl->Send(this, m_tcb, seq, isRetransmission);
```

Together, this code hook invokes a `Send()` method each time a TCP socket sends data, with BBR' (and any other TCP flavor that provides a custom `Send()` method) invoking a custom `Send()` that performs appropriate congestion actions before sending (see Listing 7).

An unrelated change is needed to make the TcpSocketBase method `BytesInFlight()` `public` so that BBR' can use it in the DRAIN state:

Listing 28: Modification to TcpSocketBase in tcp-socket-base.h (line 824)

```
 public:
  virtual uint32_t BytesInFlight (void) const;
```

3.9.2 Pacing Support

Packet pacing, indicated as a "must" in the BBR specification [CCYJ17a], 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 the normal TCP send and, instead of sending the packet, putting that packet in a queue. Packets are then 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.

The pacing rate itself (once computed by BBR') is stored as an attribute of the TCP transmission control block (TCB), where basic information is passed between the socket and the congestion control algorithm. In ns-3, the TCB is contained in the `TcpSocketState` class with methods to get and set it.

Listing 29: Addition to TcpSocketState in tcp-socket-base.h (line 195)

```
protected:
  double m_pacing_rate;                       // Pacing rate (in Mb/s).

public:
  void SetPacingRate (double pacing_rate);  // Get rate (in Mb/s).
  double GetPacingRate () const;            // Set rate (in Mb/s).
```

A structure is created to store the packets queued for pacing:

Listing 30: Addition of pacing structure in tcp-socket-base.h (line 191)

```
// Structure for tracking packets to send for pacing.
struct tcp_pacing_struct {
  SequenceNumber32 seq;     // Sequence location in TCP buffer.
  uint32_t maxSize;         // Bytes to extract.
  bool withAck;             // Include ACK or not.
};
```

The `SendDataPacket()` method is hijacked by first creating the method that does the actual ("real") sending:

Listing 31: Addition to TcpSocketBase in tcp-socket-base.h (line 665)

```
  uint32_t SendDataPacketReal (SequenceNumber32 seq, uint32_t maxSize,
                               bool withAck);
```

A `std::queue` is used to store the hijacked packets and an attribute for the next scheduled `SendDataPacketReal()` call, stored in an ns-3 event (`m_pacing_event`):

Listing 32: Addition to TcpSocketBase in tcp-socket-base.h (line 1087)

```
 protected:
  EventId m_pacing_event;                              // Pacing event.
  std::queue<tcp_pacing_struct> m_pacing_packets;  // Pacing packets.
```

The functionality for `SendDataPacket()` is then replaced with code that stores the packet rather than sending it:

Listing 33: Addition to TcpSocketBase in tcp-socket-base.cc (line 2671)

```
// If pacing, queue until time to send else send now.
uint32_t TcpSocketBase::SendDataPacket (SequenceNumber32 seq,
                                        uint32_t maxSize, bool withAck) {
  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);

  // Return size that would have been sent so app knows it's scheduled.
  Ptr<Packet> p = m_txBuffer->CopyFromSequence(maxSize, seq);
  uint32_t sz = p->GetSize(); // Size of packet
  return sz;
}
```

Whereas `SendDataPacketReal()` actually sends the packet after it is pulled off the queue, effectively doing the work of the original TCP `SendDataPacket()` method:

Listing 34: Addition to TcpSocketBase in tcp-socket-base.cc (line 2741)

```
// Really send the data packet.
// Extract at most maxSize bytes from the TxBuffer at sequence seq,
// add the TCP header, and send to TcpL4Protocol.
uint32_t TcpSocketBase::SendDataPacketReal (SequenceNumber32 seq,
                                    uint32_t maxSize, bool withAck) {

  m_congestionControl->Send(this, m_tcb);

  // Rest of the method is the same as SendDataPacket...
```

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:

Listing 35: Addition to TcpSocketBase in tcp-socket-base.h (line 1087)

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

Pacing, calling `SendDataPacketReal()`, is done in `PacePackets()` via:

Listing 36: PacePackets() addition to tcp-socket-base.cc (line 2800)

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

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

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

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

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

3.9.3 Alternate Configurations

Up to this point, there has been but one configuration for BBR' presented. However, the current code base (see the `git` repository [Cla18]) provides for alternate configurations for pacing (Section 3.9.3) and the round-trip time used in culling the bandwidth window (Section 3.9.3).

Pacing Configurations

BBR' provides three possible configurations for pacing:

1. TCP_PACING. Packet pacing is done in TCP, as described in this document, and is used by BBR'. This is the default configuration.

2. APP_PACING. Packet pacing is *not* done in TCP. BBR' changes the pacing rate and still expects packets to be paced, but in this configuration, the application layer must do the pacing. The `git` repository [Cla18] application directory provides a `BulkSendApplication` that implements application-level pacing.

3. NO_PACING. Pacing is not done at all. In this configuration, BBR' behavior changes slightly. Instead of having a larger cwnd and having the pacing rate control the bandwidth, there is effectively no pacing rate and the cwnd instead is used to control the bandwidth. Thus, the cwnd is set to the BDP (with any needed gain adjustments) as the sole way of controlling the rate.

Pacing configurations are controlled via PACING_CONFIG, defined in tcp-socket-base.h. e.g.,

Listing 37: BBR' Pacing Configurations in tcp-socket-base.h

```
// TCP_PACING: Packet pacing is done in TCP (in socket-base.cc).
// APP_PACING: Packet pacing is NOT done in TCP, only in application.
// NO_PACING: No packet pacing is done (BBR' adjusts accordingly).
enum enum_pacing_config {TCP_PACING, APP_PACING, NO_PACING};

const enum_pacing_config PACING_CONFIG = NO_PACING;
```

Timing Configurations

BBR' provides two possible configurations for the round-trip time used in culling the bandwidth window.

1. PACKET_TIME. Bandwidth window culling is done with a count of "packet-timed" round-trips, as described in this document, and is used by BBR'. This is the default configuration.

2. WALLCLOCK_TIME. Bandwidth window culling is done using the wall-clock for timing. In this configuration, bandwidth estimates that are older than the minimum round-trip time are removed from the bandwidth window.

Time configurations for bandwidth culling are controlled via TIMING_CONFIG, defined in tcp-bbr.h. e.g.,

Listing 38: BBR' Time Configurations in tcp-bbr.h

```
// PACKET_TIME - Use packet-time RTT for culling BW window.
// WALLCLOCK_TIME - Use wall-clock RTT for culling BW window.
enum enum_time_config {WALLCLOCK_TIME, PACKET_TIME};

const enum_time_config TIME_CONFIG = PACKET_TIME;
```

3.9.4 Miscellaneous

BBR' (and BBR) infers both round-trip time and bandwidth from the server side only – no modifications are needed to the TCP client. However, for testing and evaluation, it may be useful to record the actual delivery rate on the client. In effect, BBR' is trying to infer delivery rate on the sender through the delivery rate estimation algorithm [CCYJ17b], with the actual delivery rate recorded on the receiver the "ground truth". One way to record

delivery data on the client in ns-3 is to add a hook to the code similar to that used for BBR' `Send()`. Specifically, make a slight modification to the `TcpCongestionOps` class with an empty `Recv()`:

Listing 39: Addition to TcpCongestionOps in tcp-congestion-ops.h

```
0  // Added for BBR' support.
1  virtual void Recv (const TcpHeader &tcp_header, Time recv_time,
2                     uint32_t size) { /* Empty */ }
```

and add a call to `Recv()` at the top of `TcpSocketBase::ReceivedData()`:

Listing 40: Addition to TcpCongestionOps in tcp-socket-base.cc

```
0  // Added for BBR' support.
1  Time t = Simulator::Now();
2  m_congestionControl->Recv(tcpHeader, t, p->GetSize());
```

Together, this code hook invokes a `Recv()` call each time a TCP socket receives data. Recording timing and data received there can be used to obtain the actual delivery rate for off-line analysis of performance.

4 Validation

In order to validate BBR', we first observe BBR' behavior under a basic bottleneck condition, examining the behavior of key protocol attributes (See Section 4.1). Next, we compare BBR' performance results with BBR performance results published by the BBR originators, N. Cardwell, Y. Cheng, C.S. Gunn, S.H. Yeganeh and V. Jacobson [CCG+16, CCG+17]. Three scenarios are examined: steady state 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).

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 3.8) 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, 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 about 10 seconds in, stays there for 200 milliseconds, and then returns to the PROBE_BW state.

Figure 3 depicts graphs of other key BBR' variables. Each graph shows the simulation time in seconds on the x-axis.

Graph (a) plots the estimated round-trip versus time. The round-trip time spikes initially during STARTUP as BBR' overshoots the bottleneck bandwidth and fills the bottleneck queue, but after draining the queue build-up at about 150 milliseconds the round-trip has returned to near the minimum of 0.022 seconds. BBR settles into the PROBE_BW state where the small "spikes" in the round-trip times are due to the gain rate cycling in the PROBE_BW phase.

Graph (b) shows that the minimum round-trip time, picked out by BBR' over all round-trip time estimates over the last 10 seconds, stays fixed near the minimum 0.022 seconds.

Graph (c) plots the estimated bandwidth versus time. The bandwidth estimates during STARTUP and DRAIN vary between about 0.3 and 9.7 Mb/s, but bandwidth estimates during PROBE_BW varies between 9.4 and 9.6 Mb/s. Graph (e) shows the accompanying cumulative density function (CDF) of all the bandwidth estimates.

Graph (d) shows that the maximum bandwidth, picked out by BBR' over all bandwidth estimates over the past 10 round-trip times, stays fixed at about 9.7 Mb/s.

Graph (f) plots the bytes inflight versus time. BBR' sends out bytes based on the controlled cwnd, which in turn is based on the BDP. The BDP is computed as the maximum bandwidth (from graph (d)) times the minimum round trip time (from graph (b)). During STARTUP, the inflight bytes soar past the steady state rate, reduces during DRAIN, then oscillates during PROBE_BW cycles. At time 10 seconds, the inflight bytes drop as BBR' enters PROBE_RTT, draining any queue build up and looking for an RTT change.

Graph (g) plots the queue occupancy at the router versus time, depicted in blue since this is computed from system logs and not by BBR'. The queue fluctuates with the pacing rate in graph (f) since BBR' puts additional packets in flight when probing for BW. Since the congestion window is bound to twice the BDP, the queue fills to a relatively low maximum, much lower than the queue capacity of 100.

Graph (h) plots the throughput (computed every 50 milliseconds) at the router versus time, also shown in blue since this is computed from the system logs and not by BBR'. The throughput quickly ramps up during STARTUP and DRAIN, before settling at a steady rate of about 9.7 Mb/s from about 150 milliseconds until the end. The one exception is during PROBE_RTT around 10 seconds in, where the throughput drops for about 200 milliseconds.

4.2 Steady State, No Bandwidth Change

Cardwell et al. published results depicting BBR steady-state behavior (i.e., in state PROBE_BW) of a 700 ms of a 10 Mb/s BBR flow with a round-trip time of 40 ms, shown in Figure 2 of [CCG+16] and Figure 4 of [CCG+17]. Figure 4 shows their graph on the left, with a graph of the performance results 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 packets inflight. More specifically, both sets

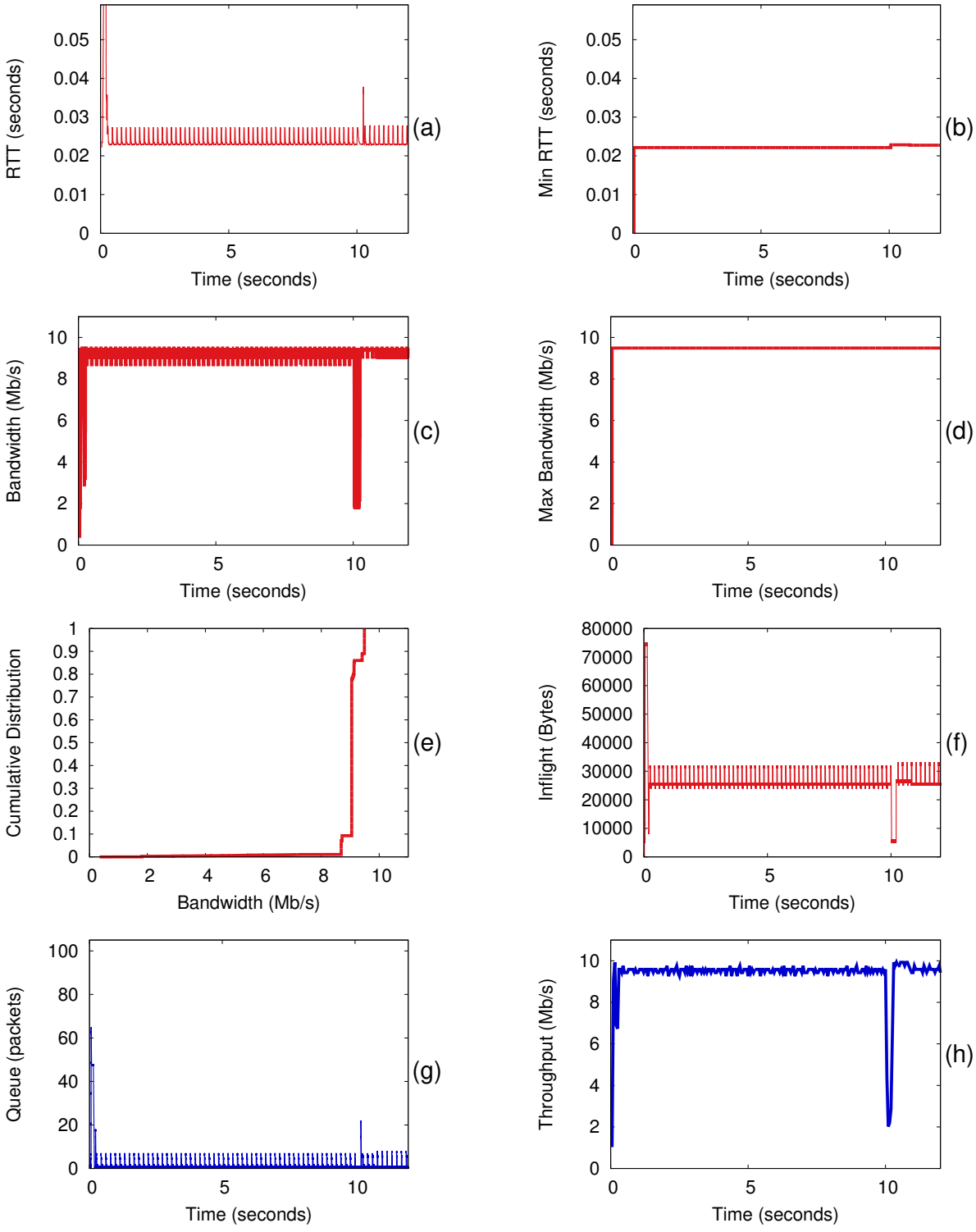


Figure 3: Basic BBR' Protocol Behavior.

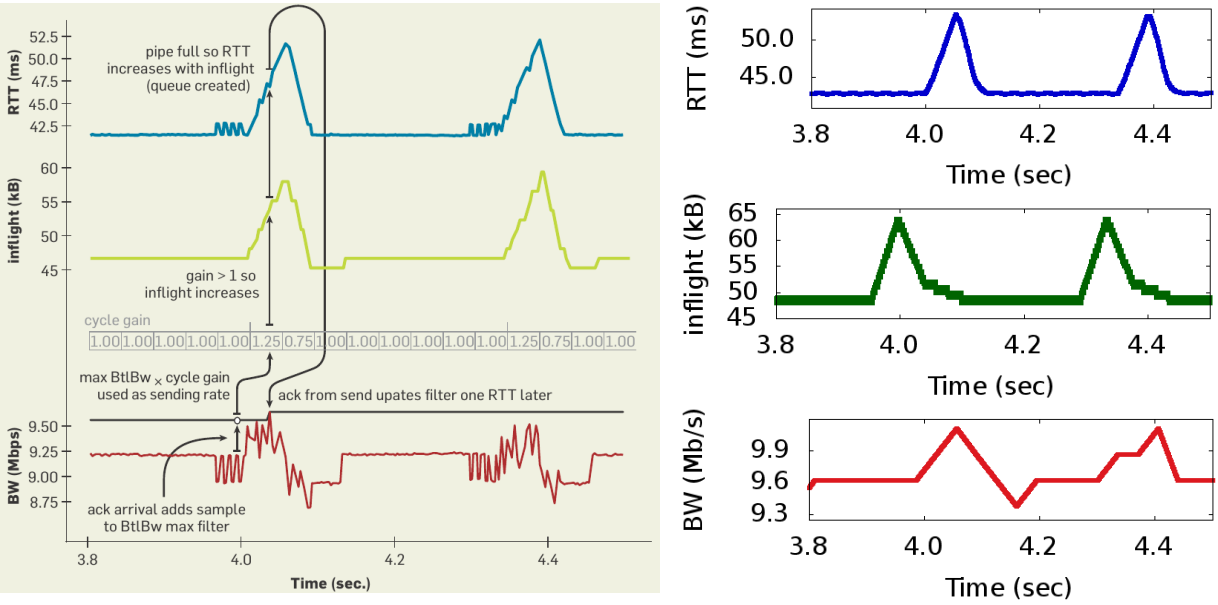


Figure 4: Comparison of BBR (left, from [CCG+17]) with BBR' (right). Graphs are round-trip time in blue, inflight in green and delivery rate in red.

of graphs have the same vertical structures resulting from the 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 immediate 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 RTT 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 3-top of [CCG+16] and Figure 5-top of [CCG+17]) depicting a long-lived BBR 10 Mb/s, 40 ms RTT flow where there is a sudden doubling of bottleneck capacity at time 20 seconds. Figure 5 shows their graph on the left, with a graph of the performance results for an equivalent BBR' scenario with the packets inflight (middle) and round-trip time (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 and quickly doubling the packets in flight 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 gain cycles.

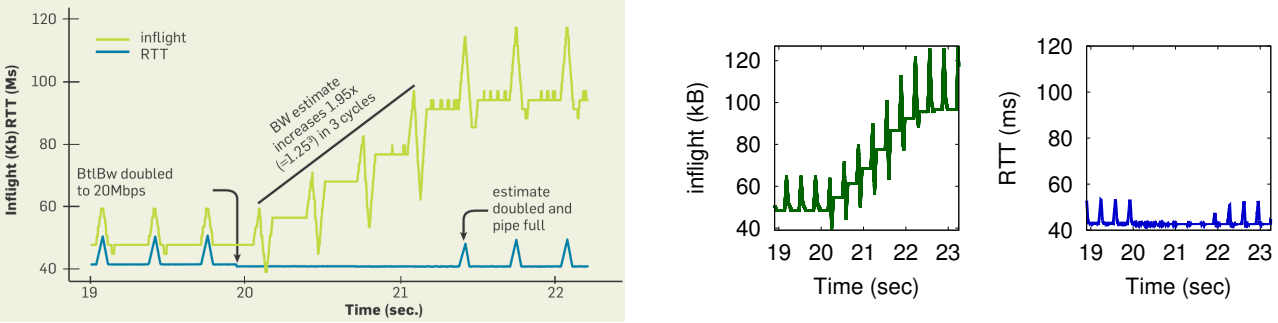


Figure 5: Comparison of BBR (left, from [CCG+17]) with BBR' (middle and right). Lines in green are packets inflight and lines in blue are round-trip time.

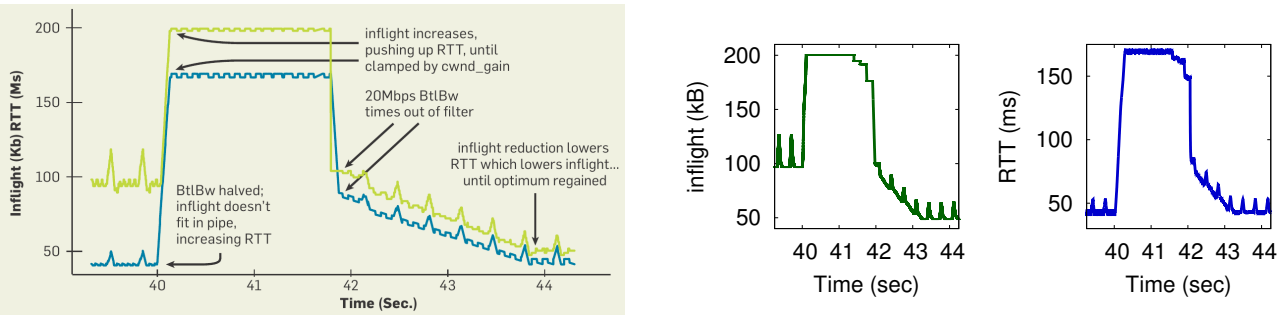


Figure 6: Comparison of BBR (left, from [CCG+17]) with BBR' (middle and right). Lines in green are packets inflight and lines in blue are round-trip time.

4.4 Bandwidth Decrease

Cardwell et al. published results (Figure 3-bottom of [CCG+16] and Figure 5-bottom of [CCG+17]) of the same flow as in Figure 4.3 where there is a sudden decrease in bottleneck capacity from 20 Mb/s back to 10 Mb/s at time 40 seconds. Figure 6 shows their graph on the left, with a graph of the performance results for an equivalent BBR' scenario with the packets in flight (middle) and round-trip time (right). As before, the BBR graphs were annotated by the authors to illustrate protocol behavior.

Again, BBR and BBR' generally behave the same, with the abrupt decrease in bandwidth resulting in a marked increase in the round-trip time. BBR and BBR' both settle into the new inflight rate after the high bandwidth readings are pushed out of the bandwidth window. Then, BBR and BBR' quickly drain the queue and settles down to a new BDP. Note, while the round-trip time is about 40 milliseconds, BBR and BBR' both use the packet-timed round-trip time which tracks the current round-trip time (about 170 milliseconds). Thus, it takes about 1.7 seconds for the 20 Mb/s maximum bandwidth estimate to time out of the window and then bring down the resulting BDP.

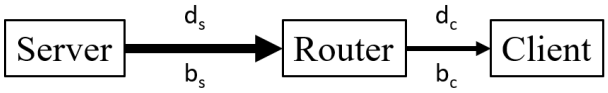


Figure 7: Wired Topology

4.5 Summary

In summary, observations of BBR' behavior in a basic, single-bottleneck with known parameters align as expected, giving some confidence in the implementation. Further visual comparison of published BBR results with equivalent BBR' results helps validate the BBR' implementation in that BBR' performs similarly to BBR. While the validation has only been undertaken for the steady-state behavior in the PROBE_BW state, long-lived, throughput intensive flows by far spend most of their time in this state.

5 Evaluation

This section evaluates BBR' 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).

5.1 Wired

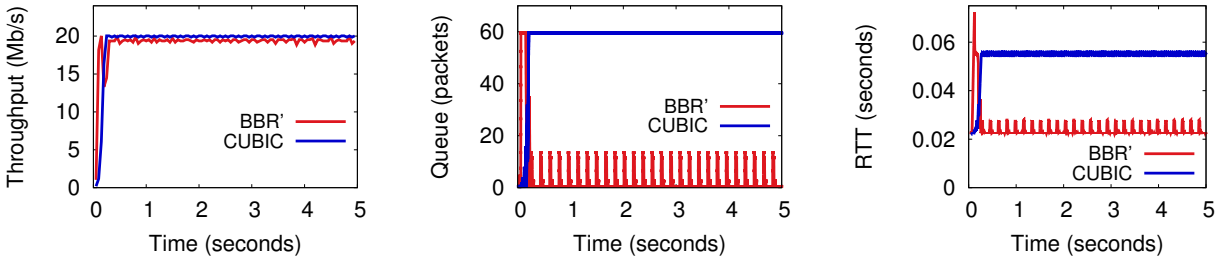
The intent is to represent the canonical congestion scenario of a network constrained by an interior bottleneck of a router between a client and a server. This typically means a server connected by a high capacity, modest latency connection to a router near a client (e.g., a head-end to a residential PC). The router, in turn, has a lower capacity, lower latency connection to the client. 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:

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

As indicated in the table, packets are 1000 bytes and the router queue size is 60 packets, which is about the bandwidth-delay product.



(a) Throughput (b) Queue Size (c) Round-trip Time

Figure 8: Wired Network with CUBIC (blue) and BBR' (red).

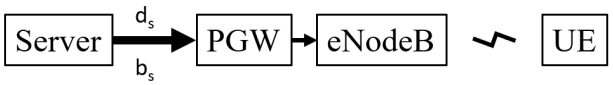


Figure 9: Wireless Topology

The scenario is repeated twice – the first time with a single BBR' flow doing a bulk download from server to client and the second time 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[7]) by Levasseur et al. [LCK14] is used.

Figure 8 depicts the results of both the BBR' and CUBIC runs plotted together, showing the first 5 seconds. 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 the maximum capacity of 20 Mb/s, BBR' slightly earlier.

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 3.8).

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 20 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 a UE, 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.

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

Parameter	Value
d_s	10 milliseconds
b_s	150 Mbits/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 first placed 5 kilometers from the UE. The results are depicted in Figure 10, with Figure 10a showing throughput and Figure 10b 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 at out about 11.5 Mb/s during steady-state. The 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.

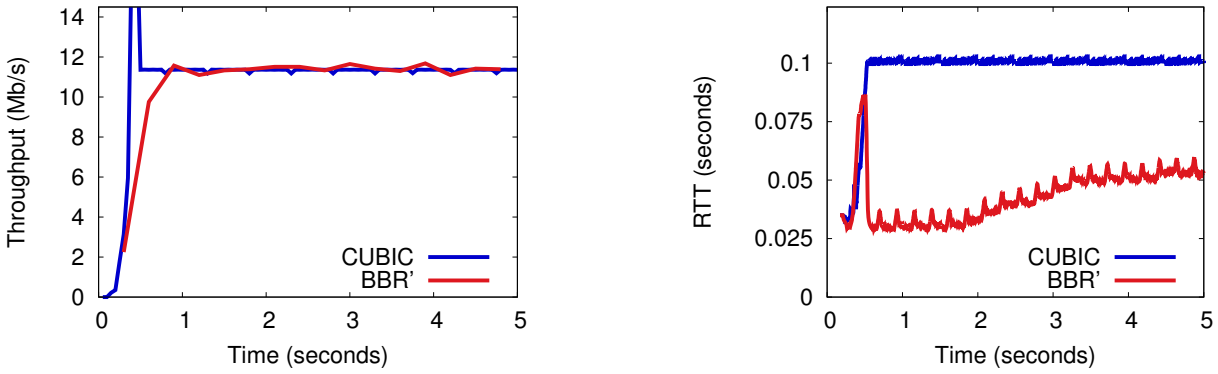
As in the wired network case (Section 5.1), there is a bigger difference in the round-trip times. In the beginning, both BBR' and CUBIC have a low round-trip time, about 30 milliseconds, which quickly increases for about 100 milliseconds as the flows startup during slow start for CUBIC and STARTUP for BBR'. However, after about 1 second, BBR' has settled into it's target low-queue, high-bandwidth condition with the round-trip times back to initial, minimal values whereas the queue (and, hence, round-trip time) for CUBIC remains high. Note, however, starting around 2 seconds, there is a slight rise in RTT for BBR', as it fills up the queue until capped by the congestion window limit (the `cwnd_gain` is 2, meaning the congestion window can grow to a maximum of twice the BDP) with an RTT of 0.05 seconds. This behavior is possibly due to B_{max} (not shown) being slightly too high, but detailed exploration is left as future work.

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).[8] Each simulation consisted of a single, 5-second bulk download, with separate runs for both CUBIC and BBR'.

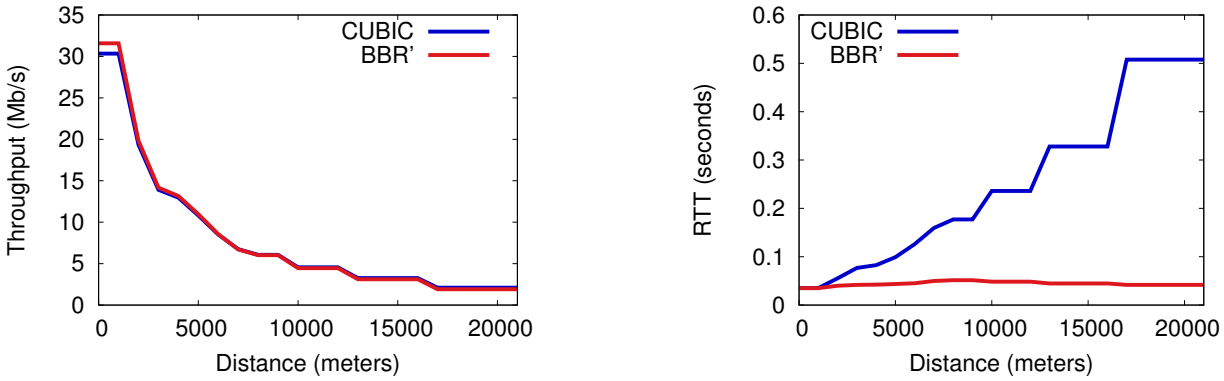
The results are depicted in Figure 10, with Figure 10a showing the average throughput and Figure 10b showing the average round-trip time. For both graphs, the x-axis is the distance in meters for the UE to the eNodeB.

[8] At 22 kilometers, neither TCP protocol was able to get any packets delivered in 5 seconds.



(a) Throughput (b) Round-trip Time

Figure 10: 4G LTE Network with CUBIC (blue) and BBR' (red).



(a) Throughput (b) Round-trip Time

Figure 11: 4G LTE Network versus UE Distance. CUBIC (blue) and BBR' (red).

From the throughout graph (Figure 11a), both CUBIC and BBR' achieve similar throughputs. However, BBR' gets slightly higher throughputs, particularly at close distances when the wireless channel is better.

The round-trip time graph (Figure 11a, however, shows 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.

6 Conclusion

The evolution of networks demands continued improvements to TCP, the prominent protocol on the Internet. Unfortunately, the predominant congestion control algorithm for TCP,

CUBIC [HRX08], saturates congested router queues, leading to dropped packets and higher than necessary round-trip times. The new congestion control algorithm BBR [CCG+16, CCG+17] promises to improve TCP performance compared to CUBIC by keeping exactly 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 New Reno, 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 [CCYJ17a] and performs similarly to previously published BBR results. Performance evaluation comparing BBR' with CUBIC shows that BBR' achieves comparable, perhaps slightly higher, 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 extension for 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 application-limited which could be incorporated (and validated and evaluated) into BBR'

As noted in Section 3.8, BBR' only transitions to/from PROBE_RTT to PROBE_BW while BBR has an additional transitions for PROBE_RTT. Future work could implement, validate and test the additional PROBE_RTT transitions for BBR'.

BBR' estimates the bottleneck bandwidth by computing the estimated receiver delivery rate based on Cheng et al. [CCYJ17b]. 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'.

BBR includes a "send quantum" (Section 4.2.2) [CCYJ17a] which is used to amortize per-packet host overheads involved in the sending process. Supposedly, this can be helpful at low rates with small packets. BBR' could incorporate and evaluate a send quantum parameter.

While pacing is indicated as mandatory, preliminary tests with BBR's alternate configuration, NO_PACING (see Section 3.9.3), suggest decent performance solely by controlling rates with the congestion window. Further evaluation can find when and where pacing benefits performance and where it does not.

Future work also includes BBR' evaluation over a wider-range of network conditions, including but not limited to capacities, topologies, protocols and application types. As suggested in the Introduction, particular attention is needed for TCP over modern wireless networks such as 4G LTE. Future work thus includes evaluation in additional 4G LTE scenarios, such as congested eNodeB's and UE mobility, with environments of mixed TCP versions and applications. For the most impact, ideally this evaluation would come after

ensuring BBR' is stable and does, in fact, fully represent the real-world implementation of BBR.

Acknowledgments

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

References

[CCG+16] N. Cardwell, Y. Cheng, C.S. Gunn, S.H. Yeganeh, and V. Jacobson. BBR: Congestion-Based Congestion Control. *ACM Queue*, October 2016.

[CCG+17] N. Cardwell, Y. Cheng, C.S. Gunn, S.H. Yeganeh, and V. Jacobson. BBR: Congestion-Based Congestion Control. *Communications of the ACM*, 60(2), February 2017.

[CCYJ17a] N. Cardwell, Y. Cheng, S. Hassas Yeganeh, and V. Jacobson. BBR Congestion Control. *IETF Draft draft-cardwell-iccrg-bbr-congestion-control-00*, July 2017.

[CCYJ17b] Y. Cheng, N. Cardwell, S. Hassas Yeganeh, and V. Jacobson. Delivery Rate Estimation. *IETF Draft cheng-iccrg-delivery-rate-estimation-00*, 2017.

[Cla18] Mark Claypool. BBR' - An Implementation of Bottleneck Bandwidth and Round-trip Time Congestion Control for ns-3. Git repository, `https://github.com/mark-claypool/bbr.git`, February 2018.

[Flo99] S. Floyd. The New Reno Modification to TCP's Fast Recovery Algorithm. *IETF Request for Comments (RFC) 2582*, April 1999.

[HRX08] S. Ha, I. Rhee, and L. Xu. CUBIC: a New TCP-friendly High-speed TCP Variant. *ACM SIGOPS Operating Systems Review*, 42(5):64–74, 2008.

[LCK14] Brett Levasseur, Mark Claypool, and Robert Kinicki. A TCP CUBIC Implementation in ns-3. In *Proceedings of the Workshop on ns-3 (WNS3)*, Atlanta, Georgia, USA, May 2014.

[MAB09] V. Paxson M. Allman and E. Blanton. TCP Congestion Control. *IETF Request for Comments (RFC) 5681*, September 2009.

[RXH+17] I. Rhee, L. Xu, S. Ha, A. Zimmermann, L. Eggert, and R. Scheffenegger. CUBIC for Fast Long-Distance Networks. *IETF Draft draft-zimmerman-tcpm-cubic-06*, September 2017.