                          BBR Congestion Control
                draft-cardwell-iccrg-bbr-congestion-control-00

Abstract

   This document specifies the BBR congestion control algorithm. BBR
   uses recent measurements of a transport connection's delivery rate
   and round-trip time to build an explicit model that includes both the
   maximum recent bandwidth available to that connection, and its
   minimum recent round-trip delay. BBR then uses this model to control
   both how fast it sends data and the maximum amount of data it allows
   in flight in the network at any time. Relative to loss-based
   congestion control algorithms such as Reno [RFC5681] or CUBIC
   [draft-ietf-tcpm-cubic], BBR offers substantially higher throughput
   for bottlenecks with shallow buffers or random losses, and
   substantially lower queueing delays for bottlenecks with deep buffers
   (avoiding "bufferbloat"). This algorithm can be implemented in any
   transport protocol that supports packet-delivery acknowledgment (thus
   far, open source implementations are available for TCP [RFC793] and
   QUIC [draft-ietf-quic-transport-00]).

Status of This Memo

Copyright Notice

Table of Contents

1. Introduction

   The Internet has largely used loss-based congestion control
   algorithms like Reno ([Jac88], [Jac90], [WS95] [RFC5681]) and CUBIC
   ([HRX08], [draft-ietf-tcpm-cubic]), which assume that packet loss is
   equivalent to congestion. These algorithms worked well for many
   years, not due to first principles but rather because Internet
   switches' and routers' queues were generally well-matched to the
   bandwidth of Internet links. As a result, queues tended to fill up
   and drop excess packets close to the moment when senders had really
   begun sending data too fast.

   But packet loss is not equivalent to congestion. Congestion can be
   thought of as a condition that occurs when a network path operates on
   a sustained basis with more data in flight than the bandwidth-delay
   product (BDP) of the path. As the Internet has evolved, loss-based
   congestion control is increasingly problematic because packet loss
   increasingly happens at moments that are divorced from the onset of
   congestion:

   1.   Shallow buffers: in shallow buffers, packet loss happens before
        congestion. With today's high-speed, long-haul links employing
        commodity switches with shallow buffers, loss-based congestion
        control can result in abysmal throughput because it overreacts,
        multiplicatively decreasing the sending rate upon packet loss,
        even if the packet loss comes from transient traffic bursts (this
        kind of packet loss can be quite frequent even when the link is
        mostly idle). Because of this dynamic, it is difficult to
        achieve full utilization with loss-based congestion control in
        practice: sustaining 10Gbps over 100ms RTT needs a packet loss
        rate below 0.000003%, and over a 100ms RTT path a more feasible
        loss rate like 1% can only sustain at most 3 Mbps (no matter what
        the bottleneck link is capable of) [draft-ietf-tcpm-cubic].

   2.   Deep buffers: at bottleneck links with deep buffers, congestion
        happens before packet loss. At the edge of today's Internet,
        loss-based congestion control causes the "bufferbloat" problem,
        by repeatedly filling the deep buffers in many last-mile links
        and causing up to seconds of needless queuing delay.

   The BBR congestion control algorithm takes a different approach:
   rather than assuming that packet loss is equivalent to congestion,

BBR builds a model of the network path in order to avoid and respond to actual congestion.

The BBR algorithm has been described previously at a high level [CCGHJ16][CCGHJ17], and active work on the BBR algorithm is continuing. This document describes the current BBR algorithm in detail.

This document is organized as follows. Section 2 provides various definitions that will be used throughout this document. Section 3 provides an overview of the design of the BBR algorithm, and section 4 describes the BBR algorithm in detail, including BBR's network path model, control parameters, and state machine. Section 5 describes the implementation status, section 6 describes security considerations, section 7 notes that there are no IANA considerations, and section 8 closes with Acknowledgments.

2. Terminology

This document defines the following state variables and constants for the BBR algorithm:

BBR.pacing_rate: The current pacing rate for a BBR flow, which controls inter-packet spacing.

BBR.send_quantum: The maximum size of a data aggregate scheduled and transmitted together.

cwnd: The transport sender's congestion window, which limits the amount of data in flight.

BBR.BtlBw: BBR's estimated bottleneck bandwidth available to the transport flow, estimated from the maximum delivery rate sample in a sliding window.

BBR.BtlBwFilter: The max filter used to estimate BBR.BtlBw.

BtlBwFilterLen: A constant specifying the length of the BBR.BtlBw max filter window for BBR.BtlBwFilter, BtlBwFilterLen is 10 packet-timed round trips.

BBR.RTprop: BBR's estimated two-way round-trip propagation delay of the path, estimated from the windowed minimum recent round-trip delay sample.

BBR.rtprop_stamp: The wall clock time at which the current BBR.RTProp sample was obtained.

BBR.rtprop_expired: A boolean recording whether the BBR.RTprop has
expired and is due for a refresh with an application idle period or a
transition into ProbeRTT state.

RTpropFilterLen: A constant specifying the length of the RTProp min
filter window, RTpropFilterLen is 10 secs.

BBR.pacing_gain: The dynamic gain factor used to scale BBR.BtlBw to
produce BBR.pacing_rate.

BBR.cwnd_gain: The dynamic gain factor used to scale the estimated
BDP to produce a congestion window (cwnd).

BBRHighGain: A constant specifying the minimum gain value that will
allow the sending rate to double each round (2/ln(2) ~= 2.89), used
in Startup mode for both BBR.pacing_gain and BBR.cwnd_gain.

BBR.filled_pipe: A boolean that records whether BBR estimates that it
has ever fully utilized its available bandwidth ("filled the pipe").

BBRMinPipeCwnd: The minimal cwnd value BBR tries to target using: 4
packets, or 4 * SMSS

BBR.round_count: Count of packet-timed round trips.

BBR.round_start: A boolean that BBR sets to true once per packet-
timed round trip, on ACKs that advance BBR.round_count.

BBR.next_round_delivered: packet.delivered value denoting the end of
a packet-timed round trip.

BBRGainCycleLen: the number of phases in the BBR ProbeBW gain cycle:
8.

ProbeRTTInterval: A constant specifying the minimum time interval
between ProbeRTT states: 10 secs.

ProbeRTTDuration: A constant specifying the minimum duration for
which ProbeRTT state holds inflight to BBRMinPipeCwnd or fewer
packets: 200 ms.

SMSS: The Sender Maximum Segment Size.

The variables starting with C, P, or rs are defined in
[draft-cheng-iccrg-delivery-rate-estimation].

The keywords "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT",
"SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this
document are to be interpreted as described in [RFC2119].

3. Design Overview

3.1. Network Path Model

BBR is a model-based congestion control algorithm: its behavior is
based on an explicit model of the network path over which a transport
flow travels. BBR's model includes explicit estimates of two
parameters:

1. BBR.BtlBw: the estimated bottleneck bandwidth available to the
    transport flow, estimated from the maximum delivery rate sample
    from a moving window.

2. BBR.RTprop: the estimated two-way round-trip propagation delay of
    the path, estimated from the the minimum round-trip delay sample
    from a moving window.

3.2. Target Operating Point

BBR uses its model to seek an operating point with high throughput
and low delay. To operate near the optimal operating point, the
point with maximum throughput and minimum delay [K79] [GK81], the
system needs to maintain two conditions:

1. Rate balance: the bottleneck packet arrival rate equals the
    bottleneck bandwidth available to the transport flow.

2. Full pipe: the total data in in fight along the path is equal to
    the BDP.

BBR uses its model to maintain an estimate of the optimal operating
point for the connection. To aim for rate balance BBR paces packets
at or near BBR.BtlBw, and to aim for a full pipe it modulates the
pacing rate to maintain an amount of data in flight near the
estimated bandwidth-delay product (BDP) of the path, or BBR.BtlBw *
BBR.RTprop.

3.3. Control Parameters

BBR uses its model to control the connection's sending behavior, to
keep it near the target operating point. Rather than using a single
control parameter, like the cwnd parameter that limits the volume of
in-flight data in the Reno and CUBIC congestion control algorithms,
BBR uses three distinct control parameters:

1. pacing rate: the rate at which BBR sends data.

2. send quantum: the maximum size of any aggregate that the
    transport sender implementation may need to transmit in order to
    amortize per-packet transmission overheads.

3. cwnd: the maximum volume of data BBR allows in-flight in the
    network at any time.

3.4. State Machine Design Overview

BBR varies its three control parameters with a simple state machine.
The state machine aims for high throughput, low latency, and an
approximately fair sharing of bandwidth by alternating sequentially
between probing first for BBR.BtlBw and then for BBR.RTprop.

A BBR flow starts in the Startup state, and ramps up its sending rate
quickly. When it estimates the pipe is full, it enters the Drain
state to drain the queue. In steady state a BBR flow only uses the
ProbeBW state, to periodically briefly raise inflight to probe for
higher BBR.BtlBw samples, and (if needed) the ProbeRTT state, to
briefly lower inflight to probe for lower BBR.RTprop samples.

3.4.1. State Transition Diagram

The following state transition diagram summarizes the flow of control
and the relationship between the different states:

```
               |
               V
      +---> Startup  ----+
      |       |          |
      |       V          |
      |     Drain   ----+
      |       |          |
      |       V          |
      +---> ProbeBW -----+
      |       ^    |     |
      |       |    |     |
      |       +----+     |
      |                  |
      +---- ProbeRTT <---+
```

3.5. Algorithm Organization

The BBR algorithm executes steps upon connection initialization, upon
each ACK, and upon each transmission. All of the sub-steps invoked
referenced below are described below.

3.5.1. Initialization

   Upon transport connection initialization, BBR executes its
   initialization steps:

```
  BBROnConnectionInit():
    BBRInit()
```

3.5.2. Per-ACK Steps

   On every ACK, the BBR algorithm executes the following
   BBRUpdateOnACK() steps in order to update its network path model,
   update its state machine, and adjust its control parameters to adapt
   to the updated model:

```
  BBRUpdateOnACK():
    BBRUpdateModelAndState()
    BBRUpdateControlParameters()

  BBRUpdateModelAndState():
    BBRUpdateBtlBw()
    BBRCheckCyclePhase()
    BBRCheckFullPipe()
    BBRCheckDrain()
    BBRUpdateRTprop()
    BBRCheckProbeRTT()

  BBRUpdateControlParameters()
    BBRSetPacingRate()
    BBRSetSendQuantum()
    BBRSetCwnd()
```

3.5.3. Per-Transmit Steps

   When transmitting, BBR merely needs to check for the case where the
   flow is restarting from idle:

```
  BBROnTransmit():
    BBRHandleRestartFromIdle()
```

3.6. Environment and Usage

   BBR is a congestion control algorithm that is agnostic to transport-
   layer and link-layer technologies, requires only sender-side changes,
   and does not require changes in the network. Open source
   implementations of BBR are available for the TCP [RFC793] and QUIC
   [draft-ietf-quic-transport-00] transport protocols, and these

implementations have been used in production for a large volume of
Internet traffic.

4. Detailed Algorithm

4.1. Maintaining the Network Path Model

As noted above, BBR is a model-based congestion control algorithm: it
is based on an explicit model of the network path over which a
transport flow travels. This model includes two estimated
parameters: BBR.BtlBw, and BBR.RTprop.

4.1.1. BBR.BtlBw

BBR.BtlBw is BBR's estimate of the bottleneck bandwidth available to
data transmissions for the transport flow. At any time, a transport
connection's data transmissions experience exactly one slowest link
or bottleneck. The bottleneck's delivery rate determines the
connection's maximum data-delivery rate. BBR tries to closely match
its sending rate to this bottleneck delivery rate to help seek "rate
balance", where the flow's packet arrival rate at the bottleneck
equals BBR.BtlBw. The bottleneck rate varies over the life of a
connection, so BBR continually estimates BBR.BtlBw using recent
delivery rate samples.

4.1.1.1. Delivery Rate Samples for Estimating BBR.BtlBw

Since calculating delivery rate samples is subtle, and the samples
are useful independent of congestion control, the approach BBR uses
for measuring each single delivery rate sample is specified in a
separate Internet Draft [draft-cheng-iccrg-delivery-rate-estimation].

4.1.1.2. BBR.BtlBw Max Filter

Delivery rate samples tend to be below the bottleneck bandwidth
available to the flow, due to "noise" introduced by random variation
in physical transmission processes (e.g. radio link layer noise) or
queues along the network path. Thus to filter out these effects BBR
uses a max filter: BBR estimates BBR.BtlBw using the windowed maximum
recent delivery rate sample seen by the connection over that past
BtlBwFilterLen round trips.

The length of the BBR.BtlBw max filter window is BtlBwFilterLen = 10
round trips. This length is driven by trade-offs among several
considerations:

o The BtlBwFilterLen is long enough to cover an entire ProbeBW gain
   cycle (see the "ProbeBW" section below). This ensures that the

window contains at least some delivery rate samples that are the
result of data transmitted with a super-unity pacing_gain (a
pacing_gain larger than 1.0). Such super-unity delivery rate
samples are instrumental in revealing the path's underlying
available bandwidth even when there is noise from delivery rate
shortfalls due to aggregation delays, queuing delays from cross-
traffic, lossy link layers with uncorrected losses, or non-
congestive buffer drops (e.g., brief coincident bursts in a
shallow buffer).

o The BtlBwFilterLen aims to be long enough to cover short-term
   fluctuations in the network's delivery rate due to the
   aforementioned sources of noise. In particular, the delivery rate
   for radio link layers (e.g., wifi and cellular technologies) can
   be highly variable, and the filter window needs to be long enough
   to remember "good" delivery rate samples in order to be robust to
   such variations.

o The BtlBwFilterLen aims to be short enough to respond in a timely
   manner to real reductions in the bandwidth available to a flow,
   whether because new flows have started sharing the bottleneck, or
   the bottleneck link service rate has reduced due to physical
   changes, policy changes, or routing changes. In any of these
   cases, existing BBR flows traversing the bottleneck should, in a
   timely manner, reduce their BBR.BtlBw estimates and thus pacing
   rate and in-flight data, in order to match the sending behavior to
   the new available bandwidth.

4.1.1.3. Tracking Time for the BBR.BtlBw Max Filter

BBR tracks time for the BBR.BtlBw filter window using a virtual (non-
wall-clock) time tracked by BBR.round_count, a count of "packet-
timed" round-trips. BBR uses this virtual BBR.round_count because it
is more robust than using wall clock time. In particular, arbitrary
intervals of wall clock time can elapse due to variations in RTTs or
timer delays for retransmission timeouts, causing wall-clock-timed
bandwidth estimates to "time out" too quickly.

The BBR.round_count counts packet-timed round trips by recording
state about a sentinel packet, and waiting for an ACK of any data
packet that was sent after that sentinel packet, using the following
pseudocode:

Upon connection initialization:

```
   BBRInitRoundCounting():
     BBR.next_round_delivered = 0
     BBR.round_start = false
     BBR.round_count = 0
```

Upon sending each packet transmission:

```
   packet.delivered = BBR.delivered
```

Upon receiving an ACK for a given data packet:

```
   BBRUpdateRound():
     BBR.delivered += packet.size
     if (packet.delivered >= BBR.next_round_delivered)
       BBR.next_round_delivered = BBR.delivered
       BBR.round_count++
       BBR.round_start = true
     else
       BBR.round_start = false
```

4.1.1.4. BBR.BtlBw and Application-limited Delivery Rate Samples

Transmissions can be application-limited, meaning the transmission
rate is limited by the application rather than the congestion control
algorithm. This is quite common because of request/response traffic.
When there is a transmission opportunity but no data to send, the
delivery rate sampler marks the corresponding bandwidth sample(s) as
application-limited [draft-cheng-iccrg-delivery-rate-estimation].
The BBR.BtlBw estimator carefully decides which samples to include in
the bandwidth model to ensure that BBR.BtlBw reflects network limits,
not application limits. By default, the estimator discards
application-limited samples, since by definition they reflect
application limits. However, the estimator does use application-
limited samples if the measured delivery rate happens to be larger
than the current BBR.BtlBw estimate, since this indicates the current
BBR.BtlBw estimate is too low.

4.1.1.5. Updating the BBR.BtlBw Max Filter

For every ACK that acknowledges some data packets as delivered, BBR
invokes BBRUpdateBtlBw() to update the BBR.BtlBw estimator as follows
(here packet.delivery_rate is the delivery rate sample obtained from
the "packet" that has just been ACKed, as specified in
[draft-cheng-iccrg-delivery-rate-estimation]):

```
   BBRUpdateBtlBw()
     BBRUpdateRound()
     if (rs.delivery_rate >= BBR.BtlBw || ! rs.is_app_limited)
         BBR.BtlBw = update_windowed_max_filter(
                       filter=BBR.BtlBwFilter,
                       value=rs.delivery_rate,
                       time=BBR.round_count,
                       window_length=BtlBwFilterLen)
```

4.1.2. BBR.RTprop

   BBR.RTprop is BBR's estimate of the round-trip propagation delay of
   the path over which a transport connection is sending. The path's
   round-trip propagation delay determines the minimum amount of time
   over which the connection must be willing to sustain transmissions at
   the BBR.BtlBw rate, and thus the minimum amount of data needed in-
   flight, for the connection to reach full utilization (a "Full Pipe").
   The round-trip propagation delay can vary over the life of a
   connection, so BBR continually estimates RTProp using recent round-
   trip delay samples.

4.1.2.1. Round-Trip Time Samples for Estimating BBR.RTprop

   For every data packet a connection sends, BBR calculates an RTT
   sample that measures the time interval from sending a data packet
   until that packet is acknowledged.

   For the most part, the same considerations and mechanisms that apply
   to RTT estimation for the purposes of retransmission timeout
   calculations [RFC6298] apply to BBR RTT samples. Namely, BBR does
   not use RTT samples based on the transmission time of retransmitted
   packets, since these are ambiguous, and thus unreliable. Also, BBR
   calculates RTT samples using both cumulative and selective
   acknowledgments (if the transport supports [RFC2018] SACK options or
   an equivalent mechanism), or transport-layer timestamps (if the
   transport supports [RFC7323] TCP timestamps or an equivalent
   mechanism).

   The only divergence from RTT estimation for retransmission timeouts
   is in the case where a given acknowledgment ACKs more than one data
   packet. In order to be conservative and schedule long timeouts to
   avoid spurious retransmissions, the maximum among such potential RTT
   samples is typically used for computing retransmission timeouts;
   i.e., SRTT is typically calculated using the data packet with the
   earliest transmission time. By contrast, in order for BBR to try to
   reach the minimum amount of data in flight to fill the pipe, BBR uses
   the minimum among such potential RTT samples; i.e., BBR calculates
   the RTT using the data packet with the latest transmission time.

4.1.2.2. BBR.RTprop Min Filter

RTT samples tend to be above the round-trip propagation delay of the
path, due to "noise" introduced by random variation in physical
transmission processes (e.g. radio link layer noise), queues along
the network path, the receiver's delayed ack strategy, ack
aggregation, etc. Thus to filter out these effects BBR uses a min
filter: BBR estimates BBR.RTprop using the minimum recent RTT sample
seen by the connection over that past RTpropFilterLen seconds. (Many
of the same network effects that can decrease delivery rate
measurements can increase RTT samples, which is why BBR's min-
filtering approach for RTTs is the complement of its max-filtering
approach for delivery rates.)

The length of the RTProp min filter window is RTpropFilterLen = 10
secs. This is driven by trade-offs among several considerations:

o The RTpropFilterLen is longer than ProbeRTTInterval, so that it
   covers an entire ProbeRTT cycle (see the "ProbeRTT" section
   below). This helps ensure that the window can contain RTT samples
   that are the result of data transmitted with inflight below the
   estimated BDP of the flow. Such RTT samples are important for
   helping to reveal the path's underlying two-way propagation delay
   even when the aforementioned "noise" effects can often obscure it.

o The RTpropFilterLen aims to be long enough to avoid needing to cut
   in-flight and throughput often. Measuring two-way propagation
   delay requires in-flight to be at or below BDP, which risks some
   amount of underutilization, so BBR uses a filter window long
   enough that such underutilization events can be rare.

o The RTpropFilterLen aims to be long enough that many applications
   have a "natural" moment of silence or low utilization that can cut
   in-flight below BDP and naturally serve to refresh the BBR.RTprop,
   without requiring BBR to force an artificial cut in in-flight.
   This applies to many popular applications, including Web, RPC,
   chunked audio or video traffic.

o The RTpropFilterLen aims to be short enough to respond in a timely
   manner to real increases in the two-way propagation delay of the
   path, e.g. due to route changes, which are expected to typically
   happen on time scales of 30 seconds or longer.

4.1.2.3. Updating the BBR.RTprop Min Filter

Upon transmitting each packet, BBR (or the associated transport
protocol) stores in per-packet data the wall-clock transmission time
of the packet (Now() returns the current wall-clock time):

```
   packet.send_time = Now()
```

For every ACK that acknowledges some data packets as delivered, BBR
(or the associated transport protocol) calculates an RTT sample "rtt"
as follows:

```
   packet.rtt = Now() - packet.send_time
```

A BBR implementation MAY use a generic windowed min filter to track
BBR.RTprop. However, a significant savings in space can be achieved
by using the same state to track BBR.RTprop and ProbeRTT timing, so
this document describes this combined approach. With this approach,
on every ACK that provides an RTT sample BBR updates the BBR.RTprop
estimator as follows:

```
   BBRUpdateRTprop()
     BBR.rtprop_expired =
       Now() > BBR.rtprop_stamp + RTpropFilterLen
     if (packet.rtt >= 0 and
         (packet.rtt <= BBR.RTprop or BBR.rtprop_expired))
       BBR.RTprop = packet.rtt
       BBR.rtprop_stamp = Now()
```

Here BBR.rtprop_expired is a boolean recording whether the BBR.RTprop
has expired and is due for a refresh, via either an application idle
period or a transition into ProbeRTT state.

4.2. BBR Control Parameters

BBR uses three distinct but interrelated control parameters: pacing
rate, send quantum, and congestion window (cwnd).

4.2.1. Pacing Rate

To help match the packet-arrival rate to the bottleneck link's
departure rate, BBR paces data packets. Pacing enforces a maximum
rate at which BBR schedules packets for transmission. Pacing is the
primary mechanism that BBR uses to control its sending behavior; BBR
implementations MUST implement pacing.

The sending host implements pacing by maintaining inter-packet
spacing at the time each packet is scheduled for transmission,
calculating the next transmission time for a packet for a given flow
(here "next_send_time") as a function of the most recent packet size
and the current pacing rate, as follows:

```
   next_send_time = Now() + packet.size / pacing_rate
```

To adapt to the bottleneck, in general BBR sets the pacing rate to be proportional to BBR.BtlBw, with a dynamic gain, or scaling factor of proportionality, called pacing_gain.

When a BBR flow starts it has no BBR.BtlBw estimate. So in this case it sets an initial pacing rate based on the transport sender implementation's initial congestion window ("InitialCwnd", e.g. from [RFC6298]), the initial SRTT (smoothed round-trip time) after the first non-zero RTT sample, and the initial pacing_gain:

```
BBRInitPacingRate():
  nominal_bandwidth = InitialCwnd / (SRTT ? SRTT : 1ms)
  BBR.pacing_rate =  BBR.pacing_gain * nominal_bandwidth
```

After initialization, on each data ACK BBR updates its pacing rate to be proportional to BBR.BtlBw, as long as it estimates that it has filled the pipe (BBR.filled_pipe is true; see the "Startup" section below for details), or doing so increases the pacing rate. Limiting the pacing rate updates in this way helps the connection probe robustly for bandwidth until it estimates it has reached its full available bandwidth ("filled the pipe"). In particular, this prevents the pacing rate from being reduced when the connection has only seen application-limited samples. BBR updates the pacing rate on each ACK by executing the BBRSetPacingRate() step as follows:

```
BBRSetPacingRateWithGain(pacing_gain):
  rate = pacing_gain * BBR.BtlBw
  if (BBR.filled_pipe || rate > BBR.pacing_rate)
    BBR.pacing_rate = rate

BBRSetPacingRate():
  BBRSetPacingRateWithGain(BBR.pacing_gain)
```

4.2.2. Send Quantum

In order to amortize per-packet host overheads involved in the sending process, high-performance transport sender implementations often schedule an aggregate containing multiple packets (multiple MSS) worth of data as a single quantum (using TSO, GSO, or other offload mechanisms [DC13]). The BBR congestion control algorithm makes this control decision explicitly, dynamically calculating a BBR.send_quantum control parameter that specifies the maximum size of these transmission aggregates. This decision is based on a trade- off: a smaller BBR.send_quantum is preferred a lower data rates because it results in shorter packet bursts, shorter queues, lower queueing delays, and lower rates of packet loss; a bigger BBR.send_quantum can be required at higher data rates because it results in lower CPU overheads at the sending and receiving hosts,

   who can ship larger amounts of data with a single trip through the
   networking stack.

   On each ACK, BBR runs BBRSetSendQuantum() to update BBR.send_quantum
   as follows:

```
     BBRSetSendQuantum():
       if (BBR.pacing_rate < 1.2 Mbps)
         BBR.send_quantum = 1 * MSS
       else if (BBR.pacing_rate < 24 Mbps)
         BBR.send_quantum  = 2 * MSS
       else
         BBR.send_quantum  = min(BBR.pacing_rate * 1ms, 64KBytes)
```

   A BBR implementation MAY use alternate approaches to select a
   BBR.send_quantum, as appropriate for the CPU overheads anticipated
   for senders and receivers, and buffering considerations anticipated
   in the network path. However, for the sake of the network and other
   users, a BBR implementation SHOULD attempt to use the smallest
   feasible quanta.

4.2.3. Congestion Window

   The congestion window, or cwnd, controls the maximum volume of data
   BBR allows in flight in the network at any time. BBR's adapts the
   cwnd based on its model of the network path and the state machine's
   decisions about how to probe that path.

   By default, BBR grows its cwnd to meet its target cwnd,
   BBR.target_cwnd, which is scaled to adapt to the estimated BDP
   computed from its path model. But BBR's selection of cwnd is
   designed to explicitly trade off among competing considerations that
   dynamically adapt to various conditions. So in loss recovery BBR
   more conservatively adjusts its sending behavior based on more recent
   delivery samples, and if BBR needs to re-probe the current BBR.RTprop
   of the path then it cuts its cwnd accordingly. The following
   sections describe the various considerations that impact cwnd.

4.2.3.1. Initial cwnd

   At its core, BBR is about using measurements to build a model of the
   network path and then adapting control decisions to the path based on
   that model. As such, the selection of the initial cwnd is considered
   to be outside the scope of the BBR algorithm, since at initialization
   there are no measurements yet upon which BBR can operate. Thus, at
   initialization, BBR uses the transport sender implementation's
   initial congestion window (e.g. from [RFC6298] for TCP).

4.2.3.2. Target cwnd

   The BBR BBR.target_cwnd is the upper bound on the volume of data BBR
   allows in flight. This bound is always in place, and dominates when
   all other considerations have been satisfied: the flow is not in loss
   recovery, does not need to probe BBR.RTprop, and has accumulated
   confidence in its model parameters by receiving enough ACKs to
   gradually grow the current cwnd to meet the BBR.target_cwnd.

   On each ACK, BBR calculates the BBR.target_cwnd as follows:

     BBRInflight(gain):
       if (BBR.RTprop == Inf)
         return InitialCwnd /* no valid RTT samples yet */
       quanta = 3*BBR.send_quantum
       estimated_bdp = BBR.BtlBw * BBR.RTprop
       return gain * estimated_bdp + quanta

     BBRUpdateTargetCwnd():
       BBR.target_cwnd = BBRInflight(BBR.cwnd_gain)

   The "estimated_bdp" term allows enough packets in flight to fully
   utilize the estimated BDP of the path, by allowing the flow to send
   at BBR.BtlBw for a duration of BBR.RTprop. Scaling up the BDP by
   cwnd_gain, selected by the BBR state machine to be above 1.0 at all
   times, bounds in-flight data to a small multiple of the BDP, in order
   to handle common network and receiver pathologies, such as delayed,
   stretched, or aggregated ACKs [A15]. The "quanta" term allows enough
   quanta in flight on the sending and receiving hosts to reach full
   utilization even in high-throughput environments using offloading
   mechanisms.

4.2.3.3. Minimum cwnd for Pipelining

   Normally (except in the immediate aftermath of entering loss
   recovery), BBR imposes a floor of BBRMinPipeCwnd (4 packets, i.e. 4 *
   MSSS). This floor helps ensure that even at very low BDPs, and with
   a transport like TCP where a receiver may ACK only every alternate
   MSS of data, there are enough packets in flight to maintain full
   pipelining. In particular BBR tries to allow at least 2 data packets
   in flight and ACKs for at least 2 data packets on the path from
   receiver to sender.

4.2.3.4. Modulating cwnd in Loss Recovery

   BBR interprets loss as a hint that there may be recent changes in
   path behavior that are not yet fully reflected in its model of the
   path, and thus it needs to be more conservative.

Upon a retransmission timeout, meaning the sender thinks all in-
flight packets are lost, BBR conservatively reduces cwnd to one
packet (1 MSS) and sends a single packet. Then BBR gradually
increases cwnd using the normal approach outlined below in "Core cwnd
Adjustment Mechanism".

When a BBR sender detects packet loss but there are still packets in
flight, on the first round of the loss-repair process BBR temporarily
reduces the cwnd to match the current delivery rate as ACKs arrive.
On second and later rounds of loss repair, it ensures the sending
rate never exceeds twice the current delivery rate as ACKs arrive.

When BBR exits loss recovery it restores the cwnd to the "last known
good" value that cwnd held before entering recovery. This applies
equally whether the flow exits loss recovery because it finishes
repairing all losses or because it executes an "undo" event after
inferring that a loss recovery event was spurious.

There are several ways to implement this high-level design for
updating cwnd in loss recovery. One is as follows:

Upon retransmission timeout (RTO):

```
  BBR.prior_cwnd = BBRSaveCwnd()
  cwnd = 1
```

Upon entering Fast Recovery, set cwnd to the number of packets still
in flight (allowing at least one for a fast retransmit):

```
  BBR.prior_cwnd = BBRSaveCwnd()
  cwnd = packets_in_flight + max(packets_delivered, 1)
  BBR.packet_conservation = true
```

Upon every ACK in Fast Recovery, run the following
BBRModulateCwndForRecovery() steps, which help ensure packet
conservation on the first round of recovery, and sending at no more
than twice the current delivery rate on later rounds of recovery
(given that "packets_delivered" packets were newly marked ACKed or
SACKed and "packets_lost" were newly marked lost):

```
  BBRModulateCwndForRecovery():
    if (packets_lost > 0)
      cwnd = max(cwnd - packets_lost, 1)
    if (BBR.packet_conservation)
      cwnd = max(cwnd, packets_in_flight + packets_delivered)
```

After one round-trip in Fast Recovery:

```
   BBR.packet_conservation = false
```

Upon exiting loss recovery (RTO recovery or Fast Recovery), either by
repairing all losses or undoing recovery, BBR restores the best-known
cwnd value we had upon entering loss recovery:

```
   BBR.packet_conservation = false
   BBRRestoreCwnd()
```

The BBRSaveCwnd() and BBRRestoreCwnd() helpers help remember and
restore the last-known good cwnd (the latest cwnd unmodulated by loss
recovery or ProbeRTT), and is defined as follows:

```
   BBRSaveCwnd():
     if (not InLossRecovery() and BBR.state != ProbeRTT)
       return cwnd
     else
       return max(BBR.prior_cwnd, cwnd)

   BBRRestoreCwnd():
     cwnd = max(cwnd, BBR.prior_cwnd)
```

4.2.3.5. Modulating cwnd in ProbeRTT

If BBR decides it needs to enter the ProbeRTT state (see the
"ProbeRTT" section below), its goal is to quickly reduce the volume
of in-flight data and drain the bottleneck queue, thereby allowing
measurement of BBR.RTprop. To implement this mode, BBR bounds the
cwnd to BBRMinPipeCwnd, the minimal value that allows pipelining (see
the "Minimum cwnd for Pipelining" section, above):

```
   BBRModulateCwndForProbeRTT():
     if (BBR.state == ProbeRTT)
       cwnd = min(cwnd, BBRMinPipeCwnd)
```

4.2.3.6. Core cwnd Adjustment Mechanism

The network path and traffic traveling over it can make sudden
dramatic changes. To adapt to these changes smoothly and robustly,
and reduce packet losses in such cases, BBR uses a conservative
strategy. When cwnd is above the BBR.target_cwnd derived from BBR's
path model, BBR cuts the cwnd immediately to the target. When cwnd
is below BBR.target_cwnd, BBR raises the cwnd gradually and
cautiously, increasing cwnd by no more than the amount of data
acknowledged (cumulatively or selectively) upon each ACK.

Specifically, on each ACK that acknowledges "packets_delivered"
packets as newly ACKed or SACKed, BBR runs the following BBRSetCwnd()
steps to update cwnd:

```
BBRSetCwnd():
  BBRUpdateTargetCwnd()
  BBRModulateCwndForRecovery()
  if (not BBR.packet_conservation) {
    if (BBR.filled_pipe)
      cwnd = min(cwnd + packets_delivered, BBR.target_cwnd)
    else if (cwnd < BBR.target_cwnd || BBR.delivered < InitialCwnd)
      cwnd = cwnd + packets_delivered
    cwnd = max(cwnd, BBRMinPipeCwnd)
  }
  BBRModulateCwndForProbeRTT()
```

There are several considerations here. If BBR has measured enough
samples to achieve confidence that it has filled the pipe (see the
description of BBR.filled_pipe in the "Startup" section below), then
it increases its cwnd based on the number of packets delivered, while
bounding its cwnd to be no larger than the BBR.target_cwnd adapted to
the estimated BDP. Otherwise, if the cwnd is below the target, or
the sender has marked so little data delivered (less than
InitialCwnd) that it does not yet judge its BBR.BtlBw estimate and
BBR.target_cwnd as useful, then it increases cwnd without bounding it
to be below the target. Finally, BBR imposes a floor of
BBRMinPipeCwnd in order to allow pipelining even with small BDPs (see
the "Minimum cwnd for Pipelining" section, above).

4.3. State Machine

BBR implements a simple state machine that uses the network path
model described above to guide its decisions, and the control
parameters described above to enact its decisions. At startup, BBR
probes to build a model of the network path; to adapt to later
changes to the path or its traffic, BBR must continue to probe to
update its model. If the available bottleneck bandwidth increases,
BBR must send faster to discover this. Likewise, if the actual
round-trip propagation delay changes, this changes the BDP, and thus
BBR must send slower to get inflight below the new BDP in order to
measure the new BBR.RTprop. Thus, BBR's state machine runs periodic,
sequential experiments, sending faster to check for BBR.BtlBw
increases or sending slower to check for BBR.RTprop decreases. The
frequency, magnitude, duration, and structure of these experiments
differ depending on what's already known (startup or steady-state)
and sending application behavior (intermittent or continuous).

This state machine has several goals:

o Achieve high throughput by efficiently utilizing available
   bandwidth.

o Achieve low latency by keeping queues bounded and small.

o Periodically send multiplicatively faster to probe the network to
   quickly discover newly-available bandwidth (in time proportional
   to the logarithm of the newly-available bandwidth).

o If needed, send slower in order to drain the bottleneck queue and
   probe the network to try to estimate the network's two-way
   propagation delay.

o Share bandwidth with other flows in an approximately fair manner.

o Feed samples to the BBR.BtlBw and BBR.RTprop estimators to refresh
   and update the model.

BBR's state machine uses two control mechanisms. First and foremost,
it uses the pacing_gain (see the "Pacing Rate" section), which
controls how fast packets are sent relative to BBR.BtlBw and is thus
key to BBR's ability to learn. A pacing_gain > 1 decreases inter-
packet time time and increases inflight. A pacing_gain < 1 has the
opposite effect, increasing inter-packet time and generally
decreasing inflight. Second, if the state machine needs to quickly
reduce inflight to a particular absolute value, it uses the cwnd.

The following sections describe each state in turn.

4.3.1. Initialization Steps

Upon transport connection initialization, BBR executes the following
steps:

```
  BBRInit():
    init_windowed_max_filter(filter=BBR.BtlBwFilter, value=0, time=0)
    BBR.rtprop = SRTT ? SRTT : Inf
    BBR.rtprop_stamp = Now()
    BBR.probe_rtt_done_stamp = 0
    BBR.probe_rtt_round_done = false
    BBR.packet_conservation = false
    BBR.prior_cwnd = 0
    BBR.idle_restart = false
    BBRInitRoundCounting()
    BBRInitFullPipe()
    BBRInitPacingRate()
    BBREnterStartup()
```

4.3.2. **Startup**

4.3.2.1. **Startup Dynamics**

   When a BBR flow starts up, it performs its first (and most rapid)
   sequential probe/drain process. Network link bandwidths currently
   span a range of at least 11 orders of magnitude, from a few bps to
   100 Gbps. To quickly learn BBR.BtlBw, given this huge range to
   explore, BBR's Startup state does an exponential search of the rate
   space, doubling the sending rate each round. This finds BBR.BtlBw in
   $O(\log_2(BDP))$ round trips.

   To achieve this rapid probing in the smoothest possible fashion, upon
   entry into Startup state BBR sets BBR.pacing_gain and BBR.cwnd_gain
   to BBRHighGain = $2/\ln(2)$ ~= 2.89, the minimum gain value that will
   allow the sending rate to double each round. Hence, when
   initializing a connection, or upon any later entry into Startup mode,
   BBR executes the following BBREnterStartup() steps:

     BBREnterStartup():
       BBR.state = Startup
       BBR.pacing_gain = BBRHighGain
       BBR.cwnd_gain = BBRHighGain

   As BBR grows its sending rate rapidly, it obtains higher delivery
   rate samples, BBR.BtlBw increases, and the pacing rate and cwnd both
   adapt by smoothly growing in proportion. Once the pipe is full, a
   queue generally forms, but the cwnd_gain bounds any queue to
   (cwnd_gain - 1) * BDP, and the immediately following Drain state is
   designed to quickly drain that queue.

4.3.2.2. **Estimating When Startup has Filled the Pipe**

   During Startup, BBR estimates whether the pipe is full by looking for
   a plateau in the BBR.BtlBw estimate. The output of this "full pipe"
   estimator is tracked in BBR.filled_pipe, a boolean that records
   whether BBR estimates that it has ever fully utilized its available
   bandwidth ("filled the pipe"). If BBR notices that there are several
   (three) rounds where attempts to double the delivery rate actually
   result in little increase (less than 25 percent), then it estimates
   that it has reached BBR.BtlBw, sets BBR.filled_pipe to true, exits
   Startup and enters Drain.

   Upon connection initialization the full pipe estimator runs:

```
      BBRInitFullPipe():
        BBR.filled_pipe = false
        BBR.full_bw = 0
        BBR.full_bw_count = 0
```

Once per round trip, upon an ACK that acknowledges new data, and when the delivery rate sample is not application-limited (see [draft-cheng-iccrg-delivery-rate-estimation]), BBR runs the "full pipe" estimator, if needed:

```
      BBRCheckFullPipe():
        if BBR.filled_pipe or
           not BBR.round_start or rs.is_app_limited
          return  // no need to check for a full pipe now
        if (BBR.BtlBw >= BBR.full_bw * 1.25)  // BBR.BtlBw still growing?
          BBR.full_bw = BBR.BtlBw    // record new baseline level
          BBR.full_bw_count = 0
          return
        BBR.full_bw_count++ // another round w/o much growth
        if (BBR.full_bw_count >= 3)
          BBR.filled_pipe = true
```

BBR waits three rounds in order to have solid evidence that the sender is not detecting a delivery-rate plateau that was temporarily imposed by the receive window. Allowing three rounds provides time for the receiver's receive-window autotuning to open up the receive window and for the BBR sender to realize that BBR.BtlBw should be higher: in the first round the receive-window autotuning algorithm grows the receive window; in the second round the sender fills the higher receive window; in the third round the sender gets higher delivery-rate samples. This three-round threshold was validated by YouTube experimental data.

4.3.3. Drain

In Startup, when the BBR "full pipe" estimator estimates that BBR has filled the pipe, BBR switches to its Drain state. In Drain, BBR aims to quickly drain any queue created in Startup by switching to a pacing_gain well below 1.0; specifically, it uses a pacing_gain that is the inverse of the value used during Startup, which drains the queue in one round:

```
      BBREnterDrain():
        BBR.state = Drain
        BBR.pacing_gain = 1/BBRHighGain  // pace slowly
        BBR.cwnd_gain = bbr_high_gain    // maintain cwnd
```

In Drain, when the number of packets in flight matches the estimated
BDP, meaning BBR estimates that the queue has been fully drained but
the pipe is still full, then BBR leaves Drain and enters ProbeBW. To
implement this, upon every ACK BBR executes:

```
  BBRCheckDrain():
    if (BBR.state == Startup and BBR.filled_pipe)
      BBREnterDrain()
    if (BBR.state == Drain and packets_in_flight <= BBRInflight(1.0))
      BBREnterProbeBW()  // we estimate queue is drained
```

4.3.4. ProbeBW

4.3.4.1. Gain Cycling Dynamics

BBR flows spend the vast majority of their time in ProbeBW state,
probing for bandwidth using an approach called gain cycling, which
helps BBR flows reach high throughput, low queuing delay, and
convergence to a fair share of bandwidth. With gain cycling, BBR
cycles through a sequence of values for the pacing_gain. It uses an
eight-phase cycle with the following pacing_gain values: 5/4, 3/4, 1,
1, 1, 1, 1, 1. Each phase normally lasts for roughly BBR.RTprop.

In phase 0 of the gain cycle, BBR probes for more bandwidth by using
a pacing_gain of 5/4, which gradually raises inflight. It does this
until the elapsed time in the phase has been at least BBR.RTprop and
either inflight has reached 5/4 * estimated_BDP (which may take
longer than BBR.RTprop if BBR.RTprop is low) or some packets have
been lost (suggesting that perhaps the path cannot actually hold 5/4
* estimated_BDP in flight).

Next, phase 1 of the gain cycle is designed to drain any queue at the
bottleneck (the likely outcome of phase 0 if the pipe was full) by
using a pacing_gain of 3/4, chosen to be the same distance below 1
that 5/4 is above 1. This phase lasts until either a full BBR.RTprop
has elapsed or inflight drops below estimated_BDP (suggesting that
the queue has drained earlier than expected, perhaps because of
application-limited behavior).

Finally, phases 2 through 7 of the gain cycle are designed to cruise
with a short queue and full utilization by using a pacing_gain of
1.0. Each of these phases lasts for the BBR.RTprop.

The average gain across all phases is 1.0 because ProbeBW aims for
its average pacing rate to equal BBR.BtlBw, the estimated available
bandwidth, and thus maintain high utilization, while maintaining a
small, well-bounded queue.

   Note that while gain cycling varies the pacing_gain value, during
   ProbeBW the cwnd_gain stays constant at two, since delayed,
   stretched, or aggregated acks can strike at any time (see the "Target
   cwnd" section).

4.3.4.2. Gain Cycling Randomization

   To improve mixing and fairness, and to reduce queues when multiple
   BBR flows share a bottleneck, BBR randomizes the phases of ProbeBW
   gain cycling by randomly picking an initial phase, from among all but
   the 3/4 phase, when entering ProbeBW.

   Why not start cycling with 3/4? The main advantage of the 3/4
   pacing_gain is to drain any queue that can be created by running a
   5/4 pacing_gain when the pipe is already full. When exiting Drain or
   ProbeRTT and entering ProbeBW, there is typically no queue to drain,
   so the 3/4 gain does not provide that advantage. Using 3/4 in those
   contexts only has a cost: a link utilization for that round of 3/4
   instead of 1. Since starting with 3/4 would have a cost but no
   benefit, and since entering ProbeBW happens at the start of any
   connection long enough to have a Drain, BBR uses this small
   optimization.

4.3.4.3. Gain Cycling Algorithm

   BBR's ProbeBW gain cycling algorithm operates as follows.

   Upon entering ProbeBW, BBR (re)starts gain cycling with the
   following:

```
     BBREnterProbeBW():
       BBR.state = ProbeBW
       BBR.pacing_gain = 1
       BBR.cwnd_gain = 2
       BBR.cycle_index = BBRGainCycleLen - 1 - random_int_in_range(0..6)
       BBRAdvanceCyclePhase()
```

   On each ACK BBR runs BBRCheckCyclePhase(), to see if it's time to
   advance to the next gain cycle phase:

```
   BBRCheckCyclePhase():
     if (BBR.sate == ProbeBW and BBRIsNextCyclePhase())
       BBRAdvanceCyclePhase()

   BBRAdvanceCyclePhase():
     BBR.cycle_stamp = Now()
     BBR.cycle_index = (BBR.cycle_index + 1) % BBRGainCycleLen
     pacing_gain_cycle = [5/4, 3/4, 1, 1, 1, 1, 1, 1]
     BBR.pacing_gain = pacing_gain_cycle[BBR.cycle_index]

   BBRIsNextCyclePhase():
     is_full_length = (Now() - BBR.cycle_stamp) > BBR.RTprop
     if (BBR.pacing_gain == 1)
       return is_full_length
     if (BBR.pacing_gain > 1)
       return is_full_length and
                 (packets_lost > 0 or
                  prior_inflight >= BBRInflight(BBR.pacing_gain))
     else  //  (BBR.pacing_gain < 1)
       return is_full_length or
                 prior_inflight <= BBRInflight(1)
```

   Here, "prior_inflight" is the amount of data that was in flight
   before processing this ACK.

4.3.4.4. Restarting From Idle

   When restarting from idle, BBR leaves its cwnd as-is and paces
   packets at exactly BBR.BtlBw, aiming to return as quickly as possible
   to its target operating point of rate balance and a full pipe.
   Specifically, if the flow's BBR.state is ProbeBW, and the flow is
   application-limited, and there are no packets in flight currently,
   then at the moment the flow sends one or more packets BBR sets
   BBR.pacing_rate to exactly BBR.BtlBw. More precisely, the BBR
   algorithm takes the following steps in BBRHandleRestartFromIdle()
   before sending a packet for a flow:

```
   BBRHandleRestartFromIdle():
     if (packets_in_flight == 0 and C.app_limited)
       BBR.idle_start = true
       if (BBR.state == ProbeBW)
         BBRSetPacingRateWithGain(1)
```

   The "Restarting Idle Connections" section of [RFC5681] suggests
   restarting from idle by slow-starting from the initial window.
   However, this approach was assuming a congestion control algorithm
   that had no estimate of the bottleneck bandwidth and no pacing, and
   thus resorted to relying on slow-starting driven by an ACK clock.

The long (log_2(BDP)*RTT) delays required to reach full utilization
with that "slow start after idle" approach caused many large
deployments to disable this mechanism, resulting in a "BDP-scale
line-rate burst" approach instead. Instead of these two approaches,
BBR restarts by pacing at BBR.BtlBw, typically achieving approximate
rate balance and a full pipe after only one BBR.RTprop has elapsed.

4.3.5. ProbeRTT

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

BBR is designed to spend the vast majority of its time (about 98
percent) in ProbeBW and the rest in ProbeRTT, based on a set of
tradeoffs. ProbeRTT lasts long enough (at least ProbeRTTDuration =
200 ms) to allow flows with different RTTs to have overlapping
ProbeRTT states, while still being short enough to bound the
throughput penalty of ProbeRTT's cwnd capping to roughly 2 percent
(200 ms/10 seconds).

As discussed in the "BBR.RTprop Min Filter" section above, BBR's
BBR.RTprop filter window, RTpropFilterLen, and time interval between
ProbeRTT states, ProbeRTTInterval, work in concert. A BBR
implementation MUST use a RTpropFilterLen equal to or longer than
ProbeRTTInterval, and to allow coordination with other BBR flows MUST
use the standard ProbeRTTInterval of 10 secs. It is RECOMMENDED to
use 10 secs for both RTpropFilterLen and ProbeRTTInterval. An
RTpropFilterLen of 10 secs is short enough to allow quick convergence
if traffic levels or routes change, but long enough so that
interactive applications (e.g., Web pages, remote procedure calls,
video chunks) often have natural silences or low-rate periods within
the window where the flow's rate is low enough or long enough to
drain its queue in the bottleneck. Then the BBR.RTprop filter
opportunistically picks up these BBR.RTprop measurements, and RTProp
refreshes without requiring ProbeRTT. This way, flows typically need
only pay the 2 percent throughput penalty if there are multiple bulk
flows busy sending over the entire ProbeRTTInterval window.

On every ACK BBR executes BBRCheckProbeRTT() to handle the steps
related to the ProbeRTT state as follows:

```
BBRCheckProbeRTT():
  if (BBR.state != ProbeRTT and
      BBR.rtprop_expired and
      not BBR.idle_restart)
    BBREnterProbeRTT()
    BBRSaveCwnd()
    BBR.probe_rtt_done_stamp = 0
  if (BBR.state == ProbeRTT)
    BBRHandleProbeRTT()
  BBR.idle_restart = false

BBREnterProbeRTT():
  BBR.state = ProbeRTT
  BBR.pacing_gain = 1
  BBR.cwnd_gain = 1

BBRHandleProbeRTT():
  /* Ignore low rate samples during ProbeRTT: */
  C.app_limited =
    (BW.delivered + packets_in_flight) ? : 1
  if (BBR.probe_rtt_done_stamp == 0 and
      packets_in_flight <= BBRMinPipeCwnd)
    BBR.probe_rtt_done_stamp =
      Now() + ProbeRTTDuration
    BBR.probe_rtt_round_done = false
    BBR.next_round_delivered = BBR.delivered
  else if (BBR.probe_rtt_done_stamp != 0)
    if (BBR.round_start)
      BBR.probe_rtt_round_done = true
    if (BBR.probe_rtt_round_done and
        Now() > BBR.probe_rtt_done_stamp)
      BBR.rtprop_stamp = Now()
      BBRRestoreCwnd()
      BBRExitProbeRTT()

BBRExitProbeRTT():
  if (BBR.filled_pipe)
    BBREnterProbeBW()
  else
    BBREnterStartup()
```

When restarting from idle (BBR.idle_restart is true) and finding that
the BBR.RTprop has expired, BBR does not enter ProbeRTT; the idleness
is deemed a sufficient attempt to coordinate to drain the queue.

The critical point is that before BBR raises its BBR.RTprop estimate (which will in turn raise its cwnd), it must first enter ProbeRTT to make a concerted and coordinated effort to drain the bottleneck queue and measure the BBR.RTprop. This allows the BBR.RTprop estimates of ensembles of BBR flows to stay well-anchored in reality, avoiding feedback loops of ever-increasing queues and RTT samples.

5. Implementation Status

This section records the status of known implementations of the algorithm defined by this specification at the time of posting of this Internet-Draft, and is based on a proposal described in [RFC7942]. The description of implementations in this section is intended to assist the IETF in its decision processes in progressing drafts to RFCs. Please note that the listing of any individual implementation here does not imply endorsement by the IETF. Furthermore, no effort has been spent to verify the information presented here that was supplied by IETF contributors. This is not intended as, and must not be construed to be, a catalog of available implementations or their features. Readers are advised to note that other implementations may exist.

According to [RFC7942], "this will allow reviewers and working groups to assign due consideration to documents that have the benefit of running code, which may serve as evidence of valuable experimentation and feedback that have made the implemented protocols more mature. It is up to the individual working groups to use this information as they see fit".

As of the time of writing, the following implementations of BBR have been publicly released:

o Linux TCP

   * Source code URL:

      + https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/tree/net/ipv4/tcp_bbr.c

   * Source: Google

   * Maturity: production

   * License: dual-licensed: GPLv2 / BSD

   * Contact: https://groups.google.com/d/forum/bbr-dev

   * Last updated: June 30, 2017

   o QUIC

      * Source code URLs:

         + https://chromium.googlesource.com/chromium/src/net/+/master/
            quic/core/congestion_control/bbr_sender.cc

         + https://chromium.googlesource.com/chromium/src/net/+/master/
            quic/core/congestion_control/bbr_sender.h

      * Source: Google

      * Maturity: production

      * License: BSD-style

      * Contact: https://groups.google.com/d/forum/bbr-dev

      * Last updated: June 30, 2017

6. Security Considerations

   This proposal makes no changes to the underlying security of
   transport protocols or congestion control algorithms. BBR shares the
   same security considerations as the existing standard congestion
   control algorithm [RFC5681].

7. IANA Considerations

   This document has no IANA actions. Here we are using that phrase,
   suggested by [RFC5226], because BBR does not modify or extend the
   wire format of any network protocol, nor does it add new dependencies
   on assigned numbers. BBR involves only a change to the congestion
   control algorithm of a transport sender, and does not involve changes
   in the network, the receiver, or any network protocol.

   Note to RFC Editor: this section may be removed on publication as an
   RFC.

8. Acknowledgments

   The authors are grateful to Len Kleinrock for his work on the theory
   underlying congestion control. We are indebted to Larry Brakmo for
   pioneering work on the Vegas [BP95] and New Vegas [B15] congestion
   control algorithms, which presaged many elements of BBR, and for
   Larry's advice and guidance during BBR's early development. The
   authors would also like to thank C. Stephen Gunn, Eric Dumazet, Ian
   Swett, Jana Iyengar, Victor Vasiliev, Nandita Dukkipati, Pawel

Jurczyk, Biren Roy, David Wetherall, Amin Vahdat, Leonidas
Kontothanassis, and the YouTube, google.com, Bandwidth Enforcer, and
Google SRE teams for their invaluable help and support.

9. References

9.1. Normative References

[RFC2018] Mathis, M. and J. Mahdavi, "TCP Selective Acknowledgment
           Options", RFC 2018, October 1996.

[RFC2119] Bradner, S., "Key words for use in RFCs to Indicate
           Requirement Levels", RFC 2119, March 1997.

[RFC5226] Narten, T. and H. Alvestrand, "Guidelines for Writing an
           IANA Considerations Section in RFCs", May 2008.

[RFC5681] Allman, M., Paxson, V., and E. Blanton, "TCP Congestion
           Control", RFC 5681, September 2009.

[RFC6298] Paxson, V., "Computing TCP's Retransmission Timer",
           RFC 6298, June 2011.

[RFC7323] Borman, D., Braden, B., Jacobson, V., and R.
           Scheffenegger, "TCP Extensions for High Performance",
           September 2014.

[RFC793]   Postel, J., "Transmission Control Protocol", September
           1981.

[RFC7942] Sheffer, Y. and A. Farrel, "Improving Awareness of Running
           Code: The Implementation Status Section", July 2016.

9.2. Informative References

[A15]   Abrahamsson, M., "TCP ACK suppression", IETF AQM mailing
           list , November 2015, <https://www.ietf.org/mail-
           archive/web/aqm/current/msg01480.html>.

[B15]   Brakmo, L., "TCP-NV: An Update to TCP-Vegas", , August
           2015, <https://docs.google.com/document/d/1o-53jbO_xH-
           m9g2YCgjaf5bK8vePjWP6Mk0rYiRLK-U/edit>.

[BP95]   Brakmo, L. and L. Peterson, "TCP Vegas: end-to-end
           congestion avoidance on a global Internet", IEEE Journal
           on Selected Areas in Communications 13(8): 1465-1480 ,
           October 1995.

   [CCGHJ16] Cardwell, N., Cheng, Y., Gunn, C., Hassas Yeganeh, S., and
              V. Jacobson, "BBR: Congestion-Based Congestion Control",
              ACM Queue Oct 2016, September-October 2016,
              <http://queue.acm.org/detail.cfm?id=3022184>.

   [CCGHJ17] Cardwell, N., Cheng, Y., Gunn, C., Hassas Yeganeh, S., and
              V. Jacobson, "BBR: Congestion-Based Congestion Control",
              Communications of the ACM Feb 2017, February 2017,
              <https://cacm.acm.org/magazines/2017/2/212428-bbr-
              congestion-based-congestion-control/pdf>.

   [DC13]   Dumazet, E. and Y. Cheng, "TSO, fair queuing, pacing:
              three's a charm", IETF 88 , November 2013,
              <https://www.ietf.org/proceedings/88/slides/slides-88-
              tcpm-9.pdf>.

   [draft-cheng-iccrg-delivery-rate-estimation]
              Cheng, Y., Cardwell, N., Hassas Yeganeh, S., and V.
              Jacobson, "Delivery Rate Estimation", draft-cheng-iccrg-
              delivery-rate-estimation-00 (work in progress), June 2017,
              <https://tools.ietf.org/html/draft-cheng-iccrg-delivery-
              rate-estimation-00>.

   [draft-ietf-quic-transport-00]
              Iyengar, J. and M. Thomson, "QUIC: A UDP-Based Multiplexed
              and Secure Transport", draft-ietf-quic-transport-00 (work
              in progress), Nov 2016, <https://tools.ietf.org/html/
              draft-ietf-quic-transport-00>.

   [draft-ietf-tcpm-cubic]
              Rhee, I., Xu, L., Ha, S., Zimmermann, A., Eggert, L., and
              R. Scheffenegger, "CUBIC for Fast Long-Distance Networks",
              draft-ietf-tcpm-cubic-04 (work in progress), February
              2017, <https://tools.ietf.org/html/draft-ietf-tcpm-cubic-
              04>.

   [GK81]   Gail, R. and L. Kleinrock, "An Invariant Property of
              Computer Network Power", Proceedings of the International
              Conference on Communications June, 1981,
              <http://www.lk.cs.ucla.edu/data/files/Gail/power.pdf>.

   [HRX08]   Ha, S., Rhee, I., and L. Xu, "CUBIC: A New TCP-Friendly
              High-Speed TCP Variant", ACM SIGOPS Operating System
              Review , 2008.

   [Jac88]   Jacobson, V., "Congestion Avoidance and Control", SIGCOMM
               1988, Computer Communication Review, vol. 18, no. 4, pp.
               314-329 , August 1988, <ftp://ftp.ee.lbl.gov/papers/
               congavoid.ps.Z>.

   [Jac90]   Jacobson, V., "Modified TCP Congestion Avoidance
               Algorithm", end2end-interest mailing list , April 1990,
               <ftp://ftp.isi.edu/end2end/end2end-interest-1990.mail>.

   [K79]   Kleinrock, L., "Power and deterministic rules of thumb for
               probabilistic problems in computer communications",
               Proceedings of the International Conference on
               Communications 1979.

   [WS95]   Wright, G. and W. Stevens, "TCP/IP Illustrated, Volume 2:
               The Implementation", Addison-Wesley , 1995.

Authors' Addresses

   Neal Cardwell
   Google, Inc
   76 Ninth Avenue
   New York, NY 10011
   USA

   Email: ncardwell@google.com

   Yuchung Cheng
   Google, Inc
   1600 Amphitheater Parkway
   Mountain View, California 94043
   USA

   Email: ycheng@google.com

   Soheil Hassas Yeganeh
   Google, Inc
   76 Ninth Avenue
   New York, NY 10011
   USA

   Email: soheil@google.com

   Van Jacobson
   Google, Inc
   1600 Amphitheater Parkway
   Mountain View, California 94043

   Email: vanj@google.com