Workgroup: CCWG
Internet-Draft: draft-cardwell-ccwg-bbr-00
Published: 19 July 2024
Intended Status: Experimental
Expires: 20 January 2025
Authors: N. Cardwell, Ed. I. Swett, Ed. J. Beshay, Ed.
Google Google Meta

BBR Congestion Control

Abstract

This document specifies the BBR congestion control algorithm. BBR ("Bottleneck Bandwidth and Round-trip propagation time") uses recent measurements of a transport connection's delivery rate, round-trip time, and packet loss rate to build an explicit model of the network path. BBR then uses this model to control both how fast it sends data and the maximum volume 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 [RFC9438], 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"). BBR can be implemented in any transport protocol that supports packet-delivery acknowledgment. Thus far, open source implementations are available for TCP [RFC9293] and QUIC [RFC9000]. This document specifies version 3 of the BBR algorithm, BBRv3.

Status of This Memo

This Internet-Draft is submitted in full conformance with the provisions of BCP 78 and BCP 79.

Internet-Drafts are working documents of the Internet Engineering Task Force (IETF). Note that other groups may also distribute working documents as Internet-Drafts. The list of current Internet-Drafts is at https://datatracker.ietf.org/drafts/current/.

Internet-Drafts are draft documents valid for a maximum of six months and may be updated, replaced, or obsoleted by other documents at any time. It is inappropriate to use Internet-Drafts as reference material or to cite them other than as "work in progress."

This Internet-Draft will expire on 20 January 2025.

Copyright Notice

Table of Contents

1. Introduction

The Internet has traditionally used loss-based congestion control algorithms like Reno ([Jac88], [Jac90], [WS95] [RFC5681]) and CUBIC ([HRX08], [RFC9438]). These algorithms worked well for many years because they were sufficiently well-matched to the prevalent range of bandwidth-delay products and degrees of buffering in Internet paths. As the Internet has evolved, loss-based congestion control is increasingly problematic in several important scenarios:

1. Shallow buffers: In shallow buffers, packet loss can happen even when a link has low utilization. With high-speed, long-haul links employing commodity switches with shallow buffers, loss-based congestion control can cause abysmal throughput because it

overreacts, making large multiplicative decreases in sending rate upon packet loss (by 50% in Reno [RFC5681] or 30% in CUBIC [RFC9438]), and only slowly growing its sending rate thereafter. This can happen even if the packet loss arises from transient traffic bursts when the link is mostly idle.

2. Deep buffers: At the edge of today's Internet, loss-based congestion control can cause the problem of "bufferbloat", by repeatedly filling deep buffers in last-mile links and causing high queuing delays.

3. Dynamic traffic workloads: With buffers of any depth, dynamic mixes of newly-entering flows or flights of data from recently idle flows can cause frequent packet loss. In such scenarios loss-based congestion control can fail to maintain its fair share of bandwidth, leading to poor application performance.

In both the shallow-buffer (1.) or dynamic-traffic (3.) scenarios mentioned above it is difficult to achieve full throughput with loss-based congestion control in practice: for CUBIC, sustaining 10Gbps over 100ms RTT needs a packet loss rate below 0.000003% (i.e., more than 40 seconds between packet losses), and over a 100ms RTT path a more feasible loss rate like 1% can only sustain at most 3 Mbps [RFC9438]. These limitations apply no matter what the bottleneck link is capable of or what the connection's fair share is. Furthermore, failure to reach the fair share can cause poor throughpout and poor tail latency for latency-sensitive applications.

The BBR ("Bottleneck Bandwidth and Round-trip propagation time") congestion control algorithm is a model-based algorithm that takes an approach different from loss-based congestion control: BBR uses recent measurements of a transport connection's delivery rate, round-trip time, and packet loss rate to build an explicit model of the network path, including its estimated available bandwidth, bandwidth-delay product, and the maximum volume of data that the connection can place in-flight in the network without causing excessive queue pressure. It then uses this model in order to guide its control behavior in seeking high throughput and low queue pressure.

This document describes the current version of the BBR algorithm, BBRv3. The original version of the algorithm, BBRv1, was described previously at a high level [CCGHJ16][CCGHJ17]. The implications of BBR in allowing high utilization of high-speed networks with shallow buffers have been discussed in other work [MM19]. Active work on the BBR algorithm is continuing.

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 state variables and constants for the BBR algorithm.

The variables starting with C, P, or rs not defined below are defined in Section 4.5.2.1, "Delivery Rate Samples".

2.1. Transport Connection State

C.delivered: The total amount of data (tracked in octets or in packets) delivered so far over the lifetime of the transport connection C.

SMSS: The Sender Maximum Segment Size.

is_cwnd_limited: True if the connection has fully utilized its cwnd at any point in the last packet-timed round trip.

InitialCwnd: The initial congestion window set by the transport protocol implementation for the connection at initialization time.

2.2. Per-Packet State

packet.delivered: C.delivered when the given packet was sent by transport connection C.

packet.departure_time: The earliest pacing departure time for the given packet.

packet.tx_in_flight: The volume of data that was estimated to be in flight at the time of the transmission of the packet.

2.3. Per-ACK Rate Sample State

rs.delivered: The volume of data delivered between the transmission of the packet that has just been ACKed and the current time.

rs.delivery_rate: The delivery rate (aka bandwidth) sample obtained from the packet that has just been ACKed.

rs.rtt: The RTT sample calculated based on the most recently-sent segment of the segments that have just been ACKed.

rs.newly_acked: The volume of data cumulatively or selectively acknowledged upon the ACK that was just received. (This quantity is referred to as "DeliveredData" in [RFC6937].)

rs.newly_lost: The volume of data newly marked lost upon the ACK that was just received.

rs.tx_in_flight: The volume of data that was estimated to be in flight at the time of the transmission of the packet that has just been ACKed (the most recently sent segment among segments ACKed by the ACK that was just received).

rs.lost: The volume of data that was declared lost between the transmission and acknowledgement of the packet that has just been ACKed (the most recently sent segment among segments ACKed by the ACK that was just received).

2.4. Output Control Parameters

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

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.

2.5. Pacing State and Parameters

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

BBRPacingMarginPercent: The static discount factor of 1% used to scale BBR.bw to produce BBR.pacing_rate.

BBR.next_departure_time: The earliest pacing departure time for the next packet BBR schedules for transmission.

BBRStartupPacingGain: A constant specifying the minimum gain value for calculating the pacing rate that will allow the sending rate to double each round (4 * ln(2) ~= 2.77) [BBRStartupPacingGain]; used in Startup mode for BBR.pacing_gain.

BBRDrainPacingGain: A constant specifying the pacing gain value used in Drain mode, to attempt to drain the estimated queue at the bottleneck link in one round-trip or less. As noted in [BBRDrainPacingGain], any value at or below 1 / BBRStartupCwndGain = 1 / 2 = 0.5 will theoretically achieve this. BBR uses the value 0.35,

which has been shown to offer good performance on YouTube, when compared with other alternatives.

2.6. cwnd State and Parameters

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

BBRDefaultCwndGain: A constant specifying the minimum gain value that allows the sending rate to double each round (2) [BBRStartupCwndGain]. Used by default in most phases for BBR.cwnd_gain.

2.7. General Algorithm State

BBR.state: The current state of a BBR flow in the BBR state machine.

BBR.round_count: Count of packet-timed round trips elapsed so far.

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.

BBR.idle_restart: A boolean that is true if and only if a connection is restarting after being idle.

2.8. Core Algorithm Design Parameters

BBRLossThresh: The maximum tolerated per-round-trip packet loss rate when probing for bandwidth (the default is 2%).

BBRBeta: The default multiplicative decrease to make upon each round trip during which the connection detects packet loss (the value is 0.7).

BBRHeadroom: The multiplicative factor to apply to BBR.inflight_hi when calculating a volume of free headroom to try to leave unused in the path (e.g. free space in the bottleneck buffer or free time slots in the bottleneck link) that can be used by cross traffic (the value is 0.15).

BBRMinPipeCwnd: The minimal cwnd value BBR targets, to allow pipelining with TCP endpoints that follow an "ACK every other packet" delayed-ACK policy: 4 * SMSS.

2.9. Network Path Model Parameters

2.9.1. Data Rate Network Path Model Parameters

The data rate model parameters together estimate both the sending rate required to reach the full bandwidth available to the flow (BBR.max_bw), and the maximum pacing rate control parameter that is consistent with the queue pressure objective (BBR.bw).

BBR.max_bw: The windowed maximum recent bandwidth sample, obtained using the BBR delivery rate sampling algorithm in Section 4.5.2.1, measured during the current or previous bandwidth probing cycle (or during Startup, if the flow is still in that state). (Part of the long-term model.)

BBR.bw_lo: The short-term maximum sending bandwidth that the algorithm estimates is safe for matching the current network path delivery rate, based on any loss signals in the current bandwidth probing cycle. This is generally lower than max_bw (thus the name). (Part of the short-term model.)

BBR.bw: The maximum sending bandwidth that the algorithm estimates is appropriate for matching the current network path delivery rate, given all available signals in the model, at any time scale. It is the min() of max_bw and bw_lo.

2.9.2. Data Volume Network Path Model Parameters

The data volume model parameters together estimate both the volume of in-flight data required to reach the full bandwidth available to the flow (BBR.max_inflight), and the maximum volume of data that is consistent with the queue pressure objective (cwnd).

BBR.min_rtt: The windowed minimum round-trip time sample measured over the last MinRTTFilterLen = 10 seconds. This attempts to estimate the two-way propagation delay of the network path when all connections sharing a bottleneck are using BBR, but also allows BBR to estimate the value required for a BBR.bdp estimate that allows full throughput if there are legacy loss-based Reno or CUBIC flows sharing the bottleneck.

BBR.bdp: The estimate of the network path's BDP (Bandwidth-Delay Product), computed as: BBR.bdp = BBR.bw * BBR.min_rtt.

BBR.extra_acked: A volume of data that is the estimate of the recent degree of aggregation in the network path.

BBR.offload_budget: The estimate of the minimum volume of data necessary to achieve full throughput when using sender (TSO/GSO) and receiver (LRO, GRO) host offload mechanisms.

BBR.max_inflight: The estimate of the volume of in-flight data required to fully utilize the bottleneck bandwidth available to the flow, based on the BDP estimate (BBR.bdp), the aggregation estimate (BBR.extra_acked), the offload budget (BBR.offload_budget), and BBRMinPipeCwnd.

BBR.inflight_hi: The long-term maximum volume of in-flight data that the algorithm estimates will produce acceptable queue pressure, based on signals in the current or previous bandwidth probing cycle, as measured by loss. That is, if a flow is probing for bandwidth, and observes that sending a particular volume of in-flight data causes a loss rate higher than the loss rate objective, it sets inflight_hi to that volume of data. (Part of the long-term model.)

BBR.inflight_lo: Analogous to BBR.bw_lo, the short-term maximum volume of in-flight data that the algorithm estimates is safe for matching the current network path delivery process, based on any loss signals in the current bandwidth probing cycle. This is generally lower than max_inflight or inflight_hi (thus the name). (Part of the short-term model.)

2.10. State for Responding to Congestion

BBR.bw_latest: a 1-round-trip max of delivered bandwidth (rs.delivery_rate).

BBR.inflight_latest: a 1-round-trip max of delivered volume of data (rs.delivered).

2.11. Estimating BBR.max_bw

BBR.MaxBwFilter: The filter for tracking the maximum recent rs.delivery_rate sample, for estimating BBR.max_bw.

MaxBwFilterLen: The filter window length for BBR.MaxBwFilter = 2 (representing up to 2 ProbeBW cycles, the current cycle and the previous full cycle).

BBR.cycle_count: The virtual time used by the BBR.max_bw filter window. Note that BBR.cycle_count only needs to be tracked with a single bit, since the BBR.MaxBwFilter only needs to track samples from two time slots: the previous ProbeBW cycle and the current ProbeBW cycle.

2.12. Estimating BBR.extra_acked

BBR.extra_acked_interval_start: the start of the time interval for estimating the excess amount of data acknowledged due to aggregation effects.

BBR.extra_acked_delivered: the volume of data marked as delivered since BBR.extra_acked_interval_start.

BBR.ExtraACKedFilter: the max filter tracking the recent maximum degree of aggregation in the path.

BBRExtraAckedFilterLen = The window length of the BBR.ExtraACKedFilter max filter window in steady-state: 10 (in units of packet-timed round trips).

2.13. Startup Parameters and State

BBR.full_bw_reached: A boolean that records whether BBR estimates that it has ever fully utilized its available bandwidth over the lifetime of the connection.

BBR.full_bw_now: A boolean that records whether BBR estimates that it has fully utilized its available bandwidth since it most recetly started looking.

BBR.full_bw: A recent baseline BBR.max_bw to estimate if BBR has "filled the pipe" in Startup.

BBR.full_bw_count: The number of non-app-limited round trips without large increases in BBR.full_bw.

2.14. ProbeRTT and min_rtt Parameters and State

2.14.1. Parameters for Estimating BBR.min_rtt

BBR.min_rtt_stamp: The wall clock time at which the current BBR.min_rtt sample was obtained.

MinRTTFilterLen: A constant specifying the length of the BBR.min_rtt min filter window, MinRTTFilterLen is 10 secs.

2.14.2. Parameters for Scheduling ProbeRTT

BBRProbeRTTCwndGain = A constant specifying the gain value for calculating the cwnd during ProbeRTT: 0.5 (meaning that ProbeRTT attempts to reduce in-flight data to 50% of the estimated BDP).

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

ProbeRTTInterval: A constant specifying the minimum time interval between ProbeRTT states: 5 secs.

BBR.probe_rtt_min_delay: The minimum RTT sample recorded in the last ProbeRTTInterval.

BBR.probe_rtt_min_stamp: The wall clock time at which the current BBR.probe_rtt_min_delay sample was obtained.

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

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. High-Level Design Goals

The high-level goal of BBR is to achieve both:

1. The full throughput (or approximate fair share thereof) available to a flow

    *Achieved in a fast and scalable manner (using bandwidth in O(log(BDP)) time).

    *Achieved with average packet loss rates of up to 1%.

2. Low queue pressure (low queuing delay and low packet loss).

These goals are in tension: sending faster improves the odds of achieving (1) but reduces the odds of achieving (2), while sending slower improves the odds of achieving (2) but reduces the odds of achieving (1). Thus the algorithm cannot maximize throughput or minimize queue pressure independently, and must jointly optimize both.

To try to achieve these goals, and seek an operating point with high throughput and low delay [K79] [GK81], BBR aims to adapt its sending process to match the network delivery process, in two dimensions:

1. data rate: the rate at which the flow sends data should ideally match the rate at which the network delivers the flow's data (the available bottleneck bandwidth)

2. data volume: the amount of unacknowledged data in flight in the network should ideally match the bandwidth-delay product (BDP) of the path

Both the control of the data rate (via the pacing rate) and data volume (directly via the congestion window or cwnd; and indirectly via the pacing rate) are important. A mismatch in either dimension can cause the sender to fail to meet its high-level design goals:

1. volume mismatch: If a sender perfectly matches its sending rate to the available bandwidth, but its volume of in-flight data exceeds the BDP, then the sender can maintain a large standing queue, increasing network latency and risking packet loss.

2. rate mismatch: If a sender's volume of in-flight data matches the BDP perfectly but its sending rate exceeds the available bottleneck bandwidth (e.g. the sender transmits a BDP of data in an unpaced fashion, at the sender's link rate), then up to a full BDP of data can burst into the bottleneck queue, causing high delay and/or high loss.

3.2. Algorithm Overview

Based on the rationale above, BBR tries to spend most of its time matching its sending process (data rate and data volume) to the network path's delivery process. To do this, it explores the 2-dimensional control parameter space of (1) data rate ("bandwidth" or "throughput") and (2) data volume ("in-flight data"), with a goal of finding the maximum values of each control parameter that are consistent with its objective for queue pressure.

Depending on what signals a given network path manifests at a given time, the objective for queue pressure is measured in terms of the most strict among:

* the amount of data that is estimated to be queued in the bottleneck buffer (data_in_flight - estimated_BDP): the objective is to maintain this amount at or below 1.5 * estimated_BDP

* the packet loss rate: the objective is a maximum per-round-trip packet loss rate of BBRLossThresh=2% (and an average packet loss rate considerably lower)

3.3. State Machine Overview

BBR varies its control parameters with a state machine that aims for high throughput, low latency, low loss, and an approximately fair sharing of bandwidth, while maintaining an up-to-date model of the network path.

A BBR flow starts in the Startup state, and ramps up its sending rate quickly, to rapidly estimate the maximum available bandwidth (BBR.max_bw). When it estimates the bottleneck bandwidth has been fully utilized, it enters the Drain state to drain the estimated

queue. In steady state a BBR flow mostly uses the ProbeBW states, to periodically briefly send faster to probe for higher capacity and then briefly send slower to try to drain any resulting queue. If needed, it briefly enters the ProbeRTT state, to lower the sending rate to probe for lower BBR.min_rtt samples. The detailed behavior for each state is described below.

3.4. Network Path Model Overview

3.4.1. High-Level Design Goals for the Network Path Model

At a high level, the BBR model is trying to reflect two aspects of the network path:

- Model what's required for achieving full throughput: Estimate the data rate (BBR.max_bw) and data volume (BBR.max_inflight) required to fully utilize the fair share of the bottleneck bandwidth available to the flow. This incorporates estimates of the maximum available bandwidth, the BDP of the path, and the requirements of any offload features on the end hosts or mechanisms on the network path that produce aggregation effects.

- Model what's permitted for achieving low queue pressure: Estimate the maximum data rate (BBR.bw) and data volume (cwnd) consistent with the queue pressure objective, as measured by the estimated degree of queuing and packet loss.

Note that those two aspects are in tension: the highest throughput is available to the flow when it sends as fast as possible and occupies as many bottleneck buffer slots as possible; the lowest que pressure is achieved by the flow when it sends as slow as possible and occupies as few bottleneck buffer slots as possible. To resolve the tension, the algorithm aims to achieve the maximum throughput achievable while still meeting the queue pressure objective.

3.4.2. Time Scales for the Network Model

At a high level, the BBR model is trying to reflect the properties of the network path on two different time scales:

3.4.2.1. Long-term model

One goal is for BBR to maintain high average utilization of the fair share of the available bandwidth, over long time intervals. This requires estimates of the path's data rate and volume capacities that are robust over long time intervals. This means being robust to congestion signals that may be noisy or may reflect short-term congestion that has already abated by the time an ACK arrives. This also means providing a robust history of the best recently-achievable performance on the path so that the flow can quickly and robustly aim

to re-probe that level of performance whenever it decides to probe the capacity of the path.

3.4.2.2. Short-term model

A second goal of BBR is to react to every congestion signal, including loss, as if it may indicate a persistent/long-term increase in congestion and/or decrease in the bandwidth available to the flow, because that may indeed be the case.

3.4.2.3. Time Scale Strategy

BBR sequentially alternates between spending most of its time using short-term models to conservatively respect all congestion signals in case they represent persistent congestion, but periodically using its long-term model to robustly probe the limits of the available path capacity in case the congestion has abated and more capacity is available.

3.5. Control Parameter Overview

BBR uses its model to control the connection's sending behavior. 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 maximum rate at which BBR sends data.

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

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

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 [RFC9293] and QUIC [RFC9000] transport protocols, and these implementations have been used in production for a large volume of Internet traffic. An open source implementation of BBR is also available for DCCP [RFC4340] [draft-romo-iccrg-ccid5].

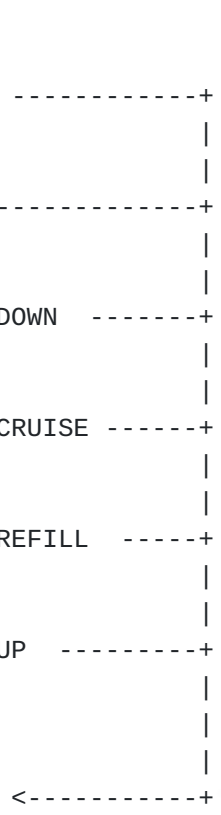
4. Detailed Algorithm

4.1. State Machine

BBR implements a state machine that uses the network path model to guide its decisions, and the control parameters to enact its decisions.

4.1.1. State Transition Diagram

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



4.1.2. State Machine Operation Overview

When starting up, BBR probes to try to quickly 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 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.min_rtt. Thus, BBR's state machine runs periodic, sequential experiments, sending faster to check for BBR.bw increases or sending slower to yield bandwidth, drain the queue, and check for BBR.min_rtt decreases. The frequency, magnitude, duration, and structure of these experiments differ

depending on what's already known (startup or steady-state) and application sending behavior (intermittent or continuous).

This state machine has several goals:

* Achieve high throughput by efficiently utilizing available bandwidth.

* Achieve low latency and packet loss rates by keeping queues bounded and small.

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

* Feed samples to the model estimators to refresh and update the model.

4.1.3. State Machine Tactics

In the BBR framework, at any given time the sender can choose one of the following tactics:

* Acceleration: Send faster then the network is delivering data: to probe the maximum bandwidth available to the flow

* Deceleration: Send slower than the network is delivering data: to reduce the amount of data in flight, with a number of overlapping motivations:

  - Reducing queuing delay: to reduce queuing delay, to reduce latency for request/response cross-traffic (e.g. RPC, web traffic).

  - Reducing packet loss: to reduce packet loss, to reduce tail latency for request/response cross-traffic (e.g. RPC, web traffic) and improve coexistence with Reno/CUBIC.

  - Probing BBR.min_rtt: to probe the path's BBR.min_rtt

  - Bandwidth convergence: to aid bandwidth fairness convergence, by leaving unused capacity in the bottleneck link or bottleneck buffer, to allow other flows that may have lower sending rates to discover and utilize the unused capacity

  - Burst tolerance: to allow bursty arrivals of cross-traffic (e.g. short web or RPC requests) to be able to share the bottleneck link without causing excessive queuing delay or packet loss

* Cruising: Send at the same rate the network is delivering data: try to match the sending rate to the flow's current available

bandwidth, to try to achieve high utilization of the available bandwidth without increasing queue pressure

Throughout the lifetime of a BBR flow, it sequentially cycles through all three tactics, to measure the network path and try to optimize its operating point.

BBR's state machine uses two control mechanisms. Primarily, it uses the pacing_gain (see the "Pacing Rate" section), which controls how fast packets are sent relative to BBR.bw. A pacing_gain > 1 decreases inter-packet time and increases inflight. A pacing_gain < 1 has the opposite effect, increasing inter-packet time and while aiming to decrease inflight. Second, if the state machine needs to quickly reduce inflight to a particular absolute value, it uses the cwnd.

4.2. Algorithm Organization

The BBR algorithm is an event-driven algorithm that executes steps upon the following events: connection initialization, upon each ACK, upon the transmission of each quantum, and upon loss detection events. All of the sub-steps invoked referenced below are described below.

4.2.1. Initialization

Upon transport connection initialization, BBR executes its initialization steps:

```
BBROnInit():
  InitWindowedMaxFilter(filter=BBR.MaxBwFilter, value=0, time=0)
  BBR.min_rtt = SRTT ? SRTT : Infinity
  BBR.min_rtt_stamp = Now()
  BBR.probe_rtt_done_stamp = 0
  BBR.probe_rtt_round_done = false
  BBR.prior_cwnd = 0
  BBR.idle_restart = false
  BBR.extra_acked_interval_start = Now()
  BBR.extra_acked_delivered = 0
  BBR.full_bw_reached = false
  BBRResetCongestionSignals()
  BBRResetLowerBounds()
  BBRInitRoundCounting()
  BBRResetFullBW()
  BBRInitPacingRate()
  BBREnterStartup()
```

4.2.2. Per-Transmit Steps

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

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

4.2.3. 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():
  BBRUpdateLatestDeliverySignals()
  BBRUpdateCongestionSignals()
  BBRUpdateACKAggregation()
  BBRCheckFullBWReached()
  BBRCheckStartupDone()
  BBRCheckDrainDone()
  BBRUpdateProbeBWCyclePhase()
  BBRUpdateMinRTT()
  BBRCheckProbeRTT()
  BBRAdvanceLatestDeliverySignals()
  BBRBoundBWForModel()

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

4.2.4. Per-Loss Steps

On every packet loss event, where some sequence range "packet" is marked lost, the BBR algorithm executes the following BBRUpdateOnLoss() steps in order to update its network path model

```
BBRUpdateOnLoss(packet):
  BBRHandleLostPacket(packet)
```

4.3. State Machine Operation

4.3.1. Startup

4.3.1.1. Startup Dynamics

When a BBR flow starts up, it performs its first (and most rapid) sequential probe/drain process in the Startup and Drain states. Network link bandwidths currently span a range of at least 11 orders

of magnitude, from a few bps to hundreds of Gbps. To quickly learn BBR.max_bw, 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.max_bw in O(log_2(BDP)) round trips.

To achieve this rapid probing smoothly, in Startup BBR uses the minimum gain values that will allow the sending rate to double each round: in Startup BBR sets BBR.pacing_gain to BBRStartupPacingGain (2.77) [BBRStartupPacingGain] and BBR.cwnd_gain to BBRDefaultCwndGain (2) [BBRStartupCwndGain].

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 = BBRStartupPacingGain
  BBR.cwnd_gain = BBRDefaultCwndGain
```

As BBR grows its sending rate rapidly, it obtains higher delivery rate samples, BBR.max_bw increases, and the pacing rate and cwnd both adapt by smoothly growing in proportion. Once the pipe is full, a queue typically forms, but the cwnd_gain bounds any queue to (cwnd_gain - 1) * estimated_BDP, which is approximately (2 - 1) * estimated_BDP = estimated_BDP. The immediately following Drain state is designed to quickly drain that queue.

During Startup, BBR estimates whether the pipe is full using two estimators. The first looks for a plateau in the BBR.max_bw estimate. The second looks for packet loss. The following subsections discuss these estimators.

```
BBRCheckStartupDone():
  BBRCheckStartupHighLoss()
  if (BBR.state == Startup and BBR.full_bw_reached)
    BBREnterDrain()
```

4.3.1.2. Exiting Acceleration Based on Bandwidth Plateau

In phases where BBR is accelerating to probe the available bandwidth - Startup and ProbeBW_UP - BBR runs a state machine to estimate whether an accelerating sending rate has saturated the available per-flow bandwidth ("filled the pipe") by looking for a plateau in the measured rs.delivery_rate.

BBR tracks the status of the current full-pipe estimation process in the boolean BBR.full_bw_now, and uses BBR.full_bw_now to exit ProbeBW_UP. BBR records in the boolean BBR.full_bw_reached whether BBR estimates that it has ever fully utilized its available bandwidth

(over the lifetime of the connection), and uses BBR.full_bw_reached to decide when to exit Startup and enter Drain.

The full pipe estimator works as follows: if BBR counts several (three) non-application-limited rounds where attempts to significantly increase the delivery rate actually result in little increase (less than 25 percent), then it estimates that it has fully utilized the per-flow available bandwidth, and sets both BBR.full_bw_now and BBR.full_bw_reached to true.

Upon starting a full pipe detection process, the following initialization runs:

```
BBRResetFullBW():
  BBR.full_bw = 0
  BBR.full_bw_count = 0
  BBR.full_bw_now = 0
```

While running the full pipe detection process, upon an ACK that acknowledges new data, and when the delivery rate sample is not application-limited (see Section 4.5.2.1), BBR runs the "full pipe" estimator:

```
BBRCheckFullBWReached():
  if (BBR.full_bw_now or rs.is_app_limited)
    return  /* no need to check for a full pipe now */
  if (rs.delivery_rate >= BBR.full_bw * 1.25)
    BBRResetFullBW()       /* bw is still growing, so reset */
    BBR.full_bw = rs.delivery_rate  /* record new baseline bw */
    return
  if (!BBR.round_start)
    return
  BBR.full_bw_count++   /* another round w/o much growth */
  BBR.full_bw_now = (BBR.full_bw_count >= 3)
  if (BBR.full_bw_now)
    BBR.full_bw_reached = true
```

BBR waits three rounds 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 auto-tuning to open up the receive window and for the BBR sender to realize that BBR.max_bw should be higher: in the first round the receive-window auto-tuning 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.

4.3.1.3. Exiting Startup Based on Packet Loss

A second method BBR uses for estimating the bottleneck is full in Startup is by looking at packet losses. Specifically, BBRCheckStartupHighLoss() checks whether all of the following criteria are all met:

- The connection has been in fast recovery for at least one full packet-timed round trip.
- The loss rate over the time scale of a single full round trip exceeds BBRLossThresh (2%).
- There are at least BBRStartupFullLossCnt=6 discontiguous sequence ranges lost in that round trip.

If these criteria are all met, then BBRCheckStartupHighLoss() takes the following steps. First, it sets BBR.full_bw_reached = true. Then it sets BBR.inflight_hi to its estimate of a safe level of in-flight data suggested by these losses, which is max(BBR.bdp, BBR.inflight_latest), where BBR.inflight_latest is the max delivered volume of data (rs.delivered) over the last round trip. Finally, it exits Startup and enters Drain.

The algorithm waits until all three criteria are met to filter out noise from burst losses, and to try to ensure the bottleneck is fully utilized on a sustained basis, and the full bottleneck bandwidth has been measured, before attempting to drain the level of in-flight data to the estimated BDP.

4.3.2. Drain

Upon exiting Startup, BBR enters its Drain state. In Drain, BBR aims to quickly drain any queue at the bottleneck link that was created in Startup by switching to a pacing_gain well below 1.0, until any estimated queue has been drained. It uses a pacing_gain of BBRDrainPacingGain = 0.35, chosen via analysis [BBRDrainPacingGain] and experimentation (on YouTube) to try to drain the queue in less than one round-trip:

```
BBREnterDrain():
  BBR.state = Drain
  BBR.pacing_gain = BBRDrainPacingGain    /* pace slowly */
  BBR.cwnd_gain = BBRDefaultCwndGain      /* maintain cwnd */
```

In Drain, when the amount of data in flight is less than or equal to the estimated BDP, meaning BBR estimates that the queue at the bottleneck link has been fully drained, then BBR exits Drain and enters ProbeBW. To implement this, upon every ACK BBR executes:

```
BBRCheckDrainDone():
  if (BBR.state == Drain and C.pipe <= BBRInflight(1.0))
    BBREnterProbeBW()  /* BBR estimates the queue was drained */
```

4.3.3. ProbeBW

Long-lived BBR flows tend to spend the vast majority of their time in the ProbeBW states. In the ProbeBW states, a BBR flow sequentially accelerates, decelerates, and cruises, to measure the network path, improve its operating point (increase throughput and reduce queue pressure), and converge toward a more fair allocation of bottleneck bandwidth. To do this, the flow sequentially cycles through all three tactics: trying to send faster than, slower than, and at the same rate as the network delivery process. To achieve this, a BBR flow in ProbeBW mode cycles through the four Probe bw states (DOWN, CRUISE, REFILL, and UP) described below in turn.

4.3.3.1. ProbeBW_DOWN

In the ProbeBW_DOWN phase of the cycle, a BBR flow pursues the deceleration tactic, to try to send slower than the network is delivering data, to reduce the amount of data in flight, with all of the standard motivations for the deceleration tactic (discussed in "State Machine Tactics" in Section 4.1.3). It does this by switching to a BBR.pacing_gain of 0.90, sending at 90% of BBR.bw. The pacing_gain value of 0.90 is derived based on the ProbeBW_UP pacing gain of 1.25, as the minimum pacing_gain value that allows bandwidth-based convergence to approximate fairness, and validated through experiments.

Exit conditions: The flow exits the ProbeBW_DOWN phase and enters CRUISE when the flow estimates that both of the following conditions have been met:

* There is free headroom: If inflight_hi is set, then BBR remains in ProbeBW_DOWN at least until the volume of in-flight data is less than or equal to a target calculated based on (1 - BBRHeadroom)*BBR.inflight_hi. The goal of this constraint is to ensure that in cases where loss signals suggest an upper limit on the volume of in-flight data, then the flow attempts to leave some free headroom in the path (e.g. free space in the bottleneck buffer or free time slots in the bottleneck link) that can be used by cross traffic (both for convergence of bandwidth shares and for burst tolerance).

* The volume of in-flight data is less than or equal to BBR.bdp, i.e. the flow estimates that it has drained any queue at the bottleneck.

4.3.3.2. ProbeBW_CRUISE

In the ProbeBW_CRUISE phase of the cycle, a BBR flow pursues the "cruising" tactic (discussed in "State Machine Tactics" in Section 4.1.3), attempting to send at the same rate the network is delivering data. It tries to match the sending rate to the flow's current available bandwidth, to try to achieve high utilization of the available bandwidth without increasing queue pressure. It does this by switching to a pacing_gain of 1.0, sending at 100% of BBR.bw. Notably, while in this state it responds to concrete congestion signals (loss) by reducing BBR.bw_lo and BBR.inflight_lo, because these signals suggest that the available bandwidth and deliverable volume of in-flight data have likely reduced, and the flow needs to change to adapt, slowing down to match the latest delivery process.

Exit conditions: The connection adaptively holds this state until it decides that it is time to probe for bandwidth (see "Time Scale for Bandwidth Probing", in Section 4.3.3.5), at which time it enters ProbeBW_REFILL.

4.3.3.3. ProbeBW_REFILL

The goal of the ProbeBW_REFILL state is to "refill the pipe", to try to fully utilize the network bottleneck without creating any significant queue pressure.

To do this, BBR first resets the short-term model parameters bw_lo and inflight_lo, setting both to "Infinity". This is the key moment in the BBR time scale strategy (see "Time Scale Strategy", Section 3.4.2.3) where the flow pivots, discarding its conservative short-term bw_lo and inflight_lo parameters and beginning to robustly probe the bottleneck's long-term available bandwidth. During this time the estimated bandwidth and inflight_hi, if set, constrain the connection.

During ProbeBW_REFILL BBR uses a BBR.pacing_gain of 1.0, to send at a rate that matches the current estimated available bandwidth, for one packet-timed round trip. The goal is to fully utilize the bottleneck link before transitioning into ProbeBW_UP and significantly increasing the chances of causing loss. The motivating insight is that, as soon as a flow starts acceleration, sending faster than the available bandwidth, it will start building a queue at the bottleneck. And if the buffer is shallow enough, then the flow can cause loss signals very shortly after the first accelerating packets arrive at the bottleneck. If the flow were to neglect to fill the pipe before it causes this loss signal, then these very quick signals of excess queue could cause the flow's estimate of the path's capacity (i.e. inflight_hi) to significantly underestimate. In particular, if the flow were to transition directly from

ProbeBW_CRUISE to ProbeBW_UP, the volume of in-flight data (at the time the first accelerating packets were sent) may often be still very close to the volume of in-flight data maintained in CRUISE, which may be only (1 - BBRHeadroom)*inflight_hi.

Exit conditions: The flow exits ProbeBW_REFILL after one packet-timed round trip, and enters ProbeBW_UP. This is because after one full round trip of sending in ProbeBW_REFILL the flow (if not application-limited) has had an opportunity to place as many packets in flight as its BBR.bw and inflight_hi permit. Correspondingly, at this point the flow starts to see bandwidth samples reflecting its ProbeBW_REFILL behavior, which may be putting too much data in flight.

4.3.3.4. ProbeBW_UP

After ProbeBW_REFILL refills the pipe, ProbeBW_UP probes for possible increases in available bandwidth by raising the sending rate, using a BBR.pacing_gain of 1.25, to send faster than the current estimated available bandwidth. It also raises the cwnd_gain to 2.25, to ensure that the flow can send faster than it had been, even if cwnd was previously limiting the sending process.

If the flow has not set BBR.inflight_hi, it implicitly tries to raise the volume of in-flight data to at least BBR.pacing_gain * BBR.bdp = 1.25 * BBR.bdp.

If the flow has set BBR.inflight_hi and encounters that limit, it then gradually increases the upper volume bound (BBR.inflight_hi) using the following approach:

* inflight_hi: The flow raises inflight_hi in ProbeBW_UP in a manner that is slow and cautious at first, but increasingly rapid and bold over time. The initial caution is motivated by the fact that a given BBR flow may be sharing a shallow buffer with thousands of other flows, so that the buffer space available to the flow may be quite tight (even just a single packet or less). The increasingly rapid growth over time is motivated by the fact that in a high-speed WAN the increase in available bandwidth (and thus the estimated BDP) may require the flow to grow the volume of its inflight data by up to O(1,000,000) packets; even a quite typical high-speed WAN BDP like 10Gbps * 100ms is around 83,000 packets (with a 1500-byte MTU). BBR takes an approach where the additive increase to BBR.inflight_hi exponentially doubles each round trip; in each successive round trip, inflight_hi grows by 1, 2, 4, 8, 16, etc, with the increases spread uniformly across the entire round trip. This helps allow BBR to utilize a larger BDP in O(log(BDP)) round trips, meeting the design goal for scalable utilization of newly-available bandwidth.

Exit conditions: The BBR flow ends ProbeBW_UP bandwidth probing and transitions to ProbeBW_DOWN to try to drain the bottleneck queue when either of the following conditions are met:

1. Bandwidth saturation: BBRIsTimeToGoDown() (see below) uses the "full pipe" estimator (see Section 4.3.1.2) to estimate whether the flow has saturated the available per-flow bandwidth ("filled the pipe"), by looking for a plateau in the measured rs.delivery_rate. If, during this process, the volume of data is constrained by BBR.inflight_hi (the flow becomes cwnd-limited while cwnd is limited by BBR.inflight_hi), then the flow cannot fully explore the available bandwidth, and so it resets the "full pipe" estimator by calling BBRResetFullBW().

2. Loss: The current loss rate, over the time scale of the last round trip, exceeds BBRLossThresh (2%).

4.3.3.5. Time Scale for Bandwidth Probing

Choosing the time scale for probing bandwidth is tied to the question of how to coexist with legacy Reno/CUBIC flows, since probing for bandwidth runs a significant risk of causing packet loss, and causing packet loss can significantly limit the throughput of such legacy Reno/CUBIC flows.

4.3.3.5.1. Bandwidth Probing and Coexistence with Reno/CUBIC

BBR has an explicit strategy for coexistence with Reno/CUBIC: to try to behave in a manner so that Reno/CUBIC flows coexisting with BBR can continue to work well in the primary contexts where they do today:

* Intra-datacenter/LAN traffic: the goal is to allow Reno/CUBIC to be able to perform well in 100M through 40G enterprise and datacenter Ethernet:

  - BDP = 40 Gbps * 20 us / (1514 bytes) ~= 66 packets

* Public Internet last mile traffic: the goal is to allow Reno/CUBIC to be able to support up to 25Mbps (for 4K Video) at an RTT of 30ms, typical parameters for common CDNs for large video services:

  - BDP = 25Mbps * 30 ms / (1514 bytes) ~= 62 packets

The challenge in meeting these goals is that Reno/CUBIC need long periods of no loss to utilize large BDPs. The good news is that in the environments where Reno/CUBIC work well today (mentioned above), the BDPs are small, roughly ~100 packets or less.

4.3.3.5.2. A Dual-Time-Scale Approach for Coexistence

The BBR strategy has several aspects:

1. The highest priority is to estimate the bandwidth available to the BBR flow in question.

2. Secondarily, a given BBR flow adapts (within bounds) the frequency at which it probes bandwidth and knowingly risks packet loss, to allow Reno/CUBIC to reach a bandwidth at least as high as that given BBR flow.

To adapt the frequency of bandwidth probing, BBR considers two time scales: a BBR-native time scale, and a bounded Reno-conscious time scale:

*T_bbr: BBR-native time-scale

  -T_bbr = uniformly randomly distributed between 2 and 3 secs

*T_reno: Reno-coexistence time scale

  -T_reno_bound = pick_randomly_either({62, 63})

  -reno_bdp = min(BBR.bdp, cwnd)

  -T_reno = min(reno_bdp, T_reno_bound) round trips

*T_probe: The time between bandwidth probe UP phases:

  -T_probe = min(T_bbr, T_reno)

This dual-time-scale approach is similar to that used by CUBIC, which has a CUBIC-native time scale given by a cubic curve, and a "Reno emulation" module that estimates what cwnd would give the flow Reno-equivalent throughput. At any given moment, CUBIC choose the cwnd implied by the more aggressive strategy.

We randomize both the T_bbr and T_reno parameters, for better mixing and fairness convergence.

4.3.3.5.3. Design Considerations for Choosing Constant Parameters

We design the maximum wall-clock bounds of BBR-native inter-bandwidth-probe wall clock time, T_bbr, to be:

*Higher than 2 sec to try to avoid causing loss for a long enough time to allow Reno flow with RTT=30ms to get 25Mbps (4K video) throughput. For this workload, given the Reno sawtooth that raises cwnd from roughly BDP to 2*BDP, one SMSS per round trip, the

inter-bandwidth-probe time must be at least: BDP * RTT = 25Mbps * .030 sec / (1514 bytes) * 0.030 sec = 1.9secs

* Lower than 3 sec to ensure flows can start probing in a reasonable amount of time to discover unutilized bw on human-scale interactive time-scales (e.g. perhaps traffic from a competing web page download is now complete).

The maximum round-trip bounds of the Reno-coexistence time scale, T_reno, are chosen to be 62-63 with the following considerations in mind:

* Choosing a value smaller than roughly 60 would imply that when BBR flows coexisted with Reno/CUBIC flows on public Internet broadband links, the Reno/CUBIC flows would not be able to achieve enough bandwidth to show 4K video.

* Choosing a value that is too large would prevent BBR from reaching its goal of tolerating 1% loss per round trip. Given that the steady-state (non-bandwidth-probing) BBR response to a non-application-limited round trip with X% packet loss is to reduce the sending rate by X% (see "Updating the Model Upon Packet Loss" in Section 4.5.10), this means that the BBR sending rate after N rounds of packet loss at a rate loss_rate is reduced to (1 - loss_rate)^N. A simplified model predicts that for a flow that encounters 1% loss in N=137 round trips of ProbeBW_CRUISE, and then doubles its cwnd (back to BBR.inflight_hi) in ProbeBW_REFILL and ProbeBW_UP, we expect that it will be able to restore and reprobe its original sending rate, since: (1 - loss_rate)^N * 2^2 = (1 - .01)^137 * 2^2 ~= 1.01. That is, we expect the flow will be able to fully respond to packet loss signals in ProbeBW_CRUISE while also fully re-measuring its maximum achievable throughput in ProbeBW_UP. However, with a larger number of round trips of ProbeBW_CRUISE, the flow would not be able to sustain its achievable throughput.

The resulting behavior is that for BBR flows with small BDPs, the bandwidth probing will be on roughly the same time scale as Reno/CUBIC; flows with large BDPs will intentionally probe more rapidly/frequently than Reno/CUBIC would (roughly every 62 round trips for low-RTT flows, or 2-3 secs for high-RTT flows).

The considerations above for timing bandwidth probing can be implemented as follows:

```
  /* Is it time to transition from DOWN or CRUISE to REFILL? */
  BBRIsTimeToProbeBW():
    if (BBRHasElapsedInPhase(BBR.bw_probe_wait) ||
        BBRIsRenoCoexistenceProbeTime())
      BBRStartProbeBW_REFILL()
      return true
    return false

  /* Randomized decision about how long to wait until
   * probing for bandwidth, using round count and wall clock.
   */
  BBRPickProbeWait():
    /* Decide random round-trip bound for wait: */
    BBR.rounds_since_bw_probe =
      random_int_between(0, 1); /* 0 or 1 */
    /* Decide the random wall clock bound for wait: */
    BBR.bw_probe_wait =
      2 + random_float_between(0.0, 1.0) /* 0..1 sec */ secs

  BBRIsRenoCoexistenceProbeTime():
    reno_rounds = BBRTargetInflight()
    rounds = min(reno_rounds, 63)
    return BBR.rounds_since_bw_probe >= rounds

  /* How much data do we want in flight?
   * Our estimated BDP, unless congestion cut cwnd. */
  BBRTargetInflight()
    return min(BBR.bdp, cwnd)
```

4.3.3.6. ProbeBW Algorithm Details

BBR's ProbeBW algorithm operates as follows.

Upon entering ProbeBW, BBR executes:

```
  BBREnterProbeBW():
    BBR.cwnd_gain = BBRDefaultCwndGain
    BBRStartProbeBW_DOWN()
```

The core logic for entering each state:

```
BBRStartProbeBW_DOWN():
  BBRResetCongestionSignals()
  BBR.probe_up_cnt = Infinity /* not growing inflight_hi */
  BBRPickProbeWait()
  BBR.cycle_stamp = Now()  /* start wall clock */
  BBR.ack_phase  = ACKS_PROBE_STOPPING
  BBRStartRound()
  BBR.state = ProbeBW_DOWN

BBRStartProbeBW_CRUISE():
  BBR.state = ProbeBW_CRUISE

BBRStartProbeBW_REFILL():
  BBRResetLowerBounds()
  BBR.bw_probe_up_rounds = 0
  BBR.bw_probe_up_acks = 0
  BBR.ack_phase = ACKS_REFILLING
  BBRStartRound()
  BBR.state = ProbeBW_REFILL

BBRStartProbeBW_UP():
  BBR.ack_phase = ACKS_PROBE_STARTING
  BBRStartRound()
  BBRResetFullBW()
  BBR.full_bw = rs.delivery_rate
  BBR.state = ProbeBW_UP
  BBRRaiseInflightHiSlope()
```

BBR executes the following BBRUpdateProbeBWCyclePhase() logic on each ACK that ACKs or SACKs new data, to advance the ProbeBW state machine:

```
/* The core state machine logic for ProbeBW: */
BBRUpdateProbeBWCyclePhase():
  if (!BBR.full_bw_reached)
    return  /* only handling steady-state behavior here */
  BBRAdaptUpperBounds()
  if (!IsInAProbeBWState())
    return /* only handling ProbeBW states here: */

  switch (state)

  ProbeBW_DOWN:
    if (BBRIsTimeToProbeBW())
      return /* already decided state transition */
    if (BBRIsTimeToCruise())
      BBRStartProbeBW_CRUISE()

  ProbeBW_CRUISE:
    if (BBRIsTimeToProbeBW())
      return /* already decided state transition */

  ProbeBW_REFILL:
    /* After one round of REFILL, start UP */
    if (BBR.round_start)
      BBR.bw_probe_samples = 1
      BBRStartProbeBW_UP()

  ProbeBW_UP:
    if (BBRIsTimeToGoDown())
      BBRStartProbeBW_DOWN()
```

The ancillary logic to implement the ProbeBW state machine:

```
IsInAProbeBWState()
  state = BBR.state
  return (state == ProbeBW_DOWN or
          state == ProbeBW_CRUISE or
          state == ProbeBW_REFILL or
          state == ProbeBW_UP)

/* Time to transition from DOWN to CRUISE? */
BBRIsTimeToCruise():
  if (inflight > BBRInflightWithHeadroom())
    return false /* not enough headroom */
  if (inflight <= BBRInflight(BBR.max_bw, 1.0))
    return true  /* inflight <= estimated BDP */

/* Time to transition from UP to DOWN? */
BBRIsTimeToGoDown():
  if (is_cwnd_limited and cwnd >= BBR.inflight_hi)
    BBRResetFullBW()   /* bw is limited by inflight_hi */
    BBR.full_bw = rs.delivery_rate
  else if (BBR.full_bw_now)
    return true  /* we estimate we've fully used path bw */
  return false

BBRHasElapsedInPhase(interval):
  return Now() > BBR.cycle_stamp + interval

/* Return a volume of data that tries to leave free
 * headroom in the bottleneck buffer or link for
 * other flows, for fairness convergence and lower
 * RTTs and loss */
BBRInflightWithHeadroom():
  if (BBR.inflight_hi == Infinity)
    return Infinity
  headroom = max(1*SMSS, BBRHeadroom * BBR.inflight_hi)
  return max(BBR.inflight_hi - headroom,
             BBRMinPipeCwnd)

/* Raise inflight_hi slope if appropriate. */
BBRRaiseInflightHiSlope():
  growth_this_round = 1*SMSS << BBR.bw_probe_up_rounds
  BBR.bw_probe_up_rounds = min(BBR.bw_probe_up_rounds + 1, 30)
  BBR.probe_up_cnt = max(cwnd / growth_this_round, 1)

/* Increase inflight_hi if appropriate. */
BBRProbeInflightHiUpward():
  if (!is_cwnd_limited or cwnd < BBR.inflight_hi)
    return  /* not fully using inflight_hi, so don't grow it */
 BBR.bw_probe_up_acks += rs.newly_acked
 if (BBR.bw_probe_up_acks >= BBR.probe_up_cnt)
```

```
    delta = BBR.bw_probe_up_acks / BBR.probe_up_cnt
    BBR.bw_probe_up_acks -= delta * BBR.bw_probe_up_cnt
    BBR.inflight_hi += delta
  if (BBR.round_start)
    BBRRaiseInflightHiSlope()

/* Track ACK state and update BBR.max_bw window and
 * BBR.inflight_hi. */
BBRAdaptUpperBounds():
  if (BBR.ack_phase == ACKS_PROBE_STARTING and BBR.round_start)
    /* starting to get bw probing samples */
    BBR.ack_phase = ACKS_PROBE_FEEDBACK
  if (BBR.ack_phase == ACKS_PROBE_STOPPING and BBR.round_start)
    /* end of samples from bw probing phase */
    if (IsInAProbeBWState() and !rs.is_app_limited)
      BBRAdvanceMaxBwFilter()

  if (!IsInflightTooHigh())
    /* Loss rate is safe. Adjust upper bounds upward. */
    if (BBR.inflight_hi == Infinity)
      return /* no upper bounds to raise */
    if (rs.tx_in_flight > BBR.inflight_hi)
      BBR.inflight_hi = rs.tx_in_flight
    if (BBR.state == ProbeBW_UP)
      BBRProbeInflightHiUpward()
```

4.3.4. ProbeRTT

4.3.4.1. ProbeRTT Overview

To help probe for BBR.min_rtt, on an as-needed basis BBR flows enter the ProbeRTT state to try to cooperate to periodically drain the bottleneck queue, and thus improve their BBR.min_rtt estimate of the unloaded two-way propagation delay.

A critical point is that before BBR raises its BBR.min_rtt estimate (which would in turn raise its maximum permissible cwnd), it first enters ProbeRTT to try to make a concerted and coordinated effort to drain the bottleneck queue and make a robust BBR.min_rtt measurement. This allows the BBR.min_rtt estimates of ensembles of BBR flows to converge, avoiding feedback loops of ever-increasing queues and RTT samples.

The ProbeRTT state works in concert with BBR.min_rtt estimation. Up to once every ProbeRTTInterval = 5 seconds, the flow enters ProbeRTT, decelerating by setting its cwnd_gain to BBRProbeRTTCwndGain = 0.5 to reduce its volume of inflight data to half of its estimated BDP, to try to measure the unloaded two-way propagation delay.

There are two main motivations for making the MinRTTFilterLen roughly twice the ProbeRTTInterval. First, this ensures that during a ProbeRTT episode the flow will "remember" the BBR.min_rtt value it measured during the previous ProbeRTT episode, providing a robust BDP estimate for the cwnd = 0.5*BDP calculation, increasing the likelihood of fully draining the bottleneck queue. Second, this allows the flow's BBR.min_rtt filter window to generally include RTT samples from two ProbeTT episodes, providing a more robust estimate.

The algorithm for ProbeRTT is as follows:

Entry conditions: In any state other than ProbeRTT itself, if the BBR.probe_rtt_min_delay estimate has not been updated (i.e., by getting a lower RTT measurement) for more than ProbeRTTInterval = 5 seconds, then BBR enters ProbeRTT and reduces the BBR.cwnd_gain to BBRProbeRTTCwndGain = 0.5.

Exit conditions: After maintaining the volume of in-flight data at BBRProbeRTTCwndGain*BBR.bdp for at least ProbeRTTDuration (200 ms) and at least one packet-timed round trip, BBR leaves ProbeRTT and transitions to ProbeBW if it estimates the pipe was filled already, or Startup otherwise.

4.3.4.2. ProbeRTT Design Rationale

BBR is designed to have ProbeRTT sacrifice no more than roughly 2% of a flow's available bandwidth. It is also designed to spend the vast majority of its time (at least roughly 96 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 diverse flows (e.g., flows with different RTTs or lower rates and thus longer inter-packet gaps) to have overlapping ProbeRTT states, while still being short enough to bound the throughput penalty of ProbeRTT's cwnd capping to roughly 2%, with the average throughput targeted at:

```
throughput = (200ms*0.5*BBR.bw + (5s - 200ms)*BBR.bw) / 5s
           = (.1s + 4.8s)/5s * BBR.bw = 0.98 * BBR.bw
```

As discussed above, BBR's BBR.min_rtt filter window, MinRTTFilterLen, and time interval between ProbeRTT states, ProbeRTTInterval, work in concert. BBR uses a MinRTTFilterLen equal to or longer than ProbeRTTInterval to allow the filter window to include at least one ProbeRTT.

To allow coordination with other BBR flows, each BBR flow MUST use the standard ProbeRTTInterval of 5 secs.

A ProbeRTTInterval of 5 secs is short enough to allow quick convergence if traffic levels or routes change, but long enough so that interactive applications (e.g., Web, remote procedure calls,

video chunks) often have natural silences or low-rate periods within the window where the flow's rate is low enough for long enough to drain its queue in the bottleneck. Then the BBR.probe_rtt_min_delay filter opportunistically picks up these measurements, and the BBR.probe_rtt_min_delay estimate 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.

As an optimization, when restarting from idle and finding that the BBR.probe_rtt_min_delay has expired, BBR does not enter ProbeRTT; the idleness is deemed a sufficient attempt to coordinate to drain the queue.

4.3.4.3. Calculating the rs.rtt RTT Sample

Upon transmitting each packet, BBR (or the associated transport protocol) stores in per-packet data the wall-clock scheduled transmission time of the packet in packet.departure_time (see "Pacing Rate: BBR.pacing_rate" in Section 4.6.2 for how this is calculated).

For every ACK that newly acknowledges some data (whether cumulatively or selectively), the sender's BBR implementation (or the associated transport protocol implementation) attempts to calculate an RTT sample. The sender MUST consider any potential retransmission ambiguities that can arise in some transport protocols. If some of the acknowledged data was not retransmitted, or some of the data was retransmitted but the sender can still unambiguously determine the RTT of the data (e.g. if the transport supports [RFC7323] TCP timestamps or an equivalent mechanism), then the sender calculates an RTT sample, rs.rtt, as follows:

```
rs.rtt = Now() - packet.departure_time
```

4.3.4.4. ProbeRTT Logic

On every ACK BBR executes BBRUpdateMinRTT() to update its ProbeRTT scheduling state (BBR.probe_rtt_min_delay and BBR.probe_rtt_min_stamp) and its BBR.min_rtt estimate:

```
BBRUpdateMinRTT()
  BBR.probe_rtt_expired =
    Now() > BBR.probe_rtt_min_stamp + ProbeRTTInterval
  if (rs.rtt >= 0 and
      (rs.rtt < BBR.probe_rtt_min_delay or
       BBR.probe_rtt_expired))
     BBR.probe_rtt_min_delay = rs.rtt
     BBR.probe_rtt_min_stamp = Now()

  min_rtt_expired =
    Now() > BBR.min_rtt_stamp + MinRTTFilterLen
  if (BBR.probe_rtt_min_delay < BBR.min_rtt or
      min_rtt_expired)
    BBR.min_rtt       = BBR.probe_rtt_min_delay
    BBR.min_rtt_stamp = BBR.probe_rtt_min_stamp
```

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

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

```
BBRCheckProbeRTT():
  if (BBR.state != ProbeRTT and
      BBR.probe_rtt_expired and
      not BBR.idle_restart)
    BBREnterProbeRTT()
    BBRSaveCwnd()
    BBR.probe_rtt_done_stamp = 0
    BBR.ack_phase = ACKS_PROBE_STOPPING
    BBRStartRound()
  if (BBR.state == ProbeRTT)
    BBRHandleProbeRTT()
  if (rs.delivered > 0)
    BBR.idle_restart = false

BBREnterProbeRTT():
  BBR.state = ProbeRTT
  BBR.pacing_gain = 1
  BBR.cwnd_gain = BBRProbeRTTCwndGain  /* 0.5 */

BBRHandleProbeRTT():
  /* Ignore low rate samples during ProbeRTT: */
  MarkConnectionAppLimited()
  if (BBR.probe_rtt_done_stamp == 0 and
      C.pipe <= BBRProbeRTTCwnd())
    /* Wait for at least ProbeRTTDuration to elapse: */
    BBR.probe_rtt_done_stamp =
      Now() + ProbeRTTDuration
    /* Wait for at least one round to elapse: */
    BBR.probe_rtt_round_done = false
    BBRStartRound()
  else if (BBR.probe_rtt_done_stamp != 0)
    if (BBR.round_start)
      BBR.probe_rtt_round_done = true
    if (BBR.probe_rtt_round_done)
      BBRCheckProbeRTTDone()

BBRCheckProbeRTTDone():
  if (BBR.probe_rtt_done_stamp != 0 and
      Now() > BBR.probe_rtt_done_stamp)
    /* schedule next ProbeRTT: */
    BBR.probe_rtt_min_stamp = Now()
    BBRRestoreCwnd()
    BBRExitProbeRTT()

MarkConnectionAppLimited():
  C.app_limited =
    (C.delivered + C.pipe) ? : 1
```

4.3.4.5. Exiting ProbeRTT

When exiting ProbeRTT, BBR transitions to ProbeBW if it estimates the pipe was filled already, or Startup otherwise.

When transitioning out of ProbeRTT, BBR calls BBRResetLowerBounds() to reset the lower bounds, since any congestion encountered in ProbeRTT may have pulled the short-term model far below the capacity of the path.

But the algorithm is cautious in timing the next bandwidth probe: raising inflight after ProbeRTT may cause loss, so the algorithm resets the bandwidth-probing clock by starting the cycle at ProbeBW_DOWN(). But then as an optimization, since the connection is exiting ProbeRTT, we know that infligh is already below the estimated BDP, so the connection can proceed immediately to ProbeBW_CRUISE.

To summarize, the logic for exiting ProbeRTT is as follows:

```
BBRExitProbeRTT():
  BBRResetLowerBounds()
  if (BBR.full_bw_reached)
    BBRStartProbeBW_DOWN()
    BBRStartProbeBW_CRUISE()
  else
    BBREnterStartup()
```

4.4. Restarting From Idle

4.4.1. Actions when Restarting from Idle

When restarting from idle in ProbeBW states, BBR leaves its cwnd as-is and paces packets at exactly BBR.bw, 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 before the flow sends one or more packets BBR sets BBR.pacing_rate to exactly BBR.bw.

Also, when restarting from idle BBR checks to see if the connection is in ProbeRTT and has met the exit conditions for ProbeRTT. If a connection goes idle during ProbeRTT then often it will have met those exit conditions by the time it restarts, so that the connection can restore the cwnd to its full value before it starts transmitting a new flight of data.

More precisely, the BBR algorithm takes the following steps in BBRHandleRestartFromIdle() before sending a packet for a flow:

```
BBRHandleRestartFromIdle():
  if (C.pipe == 0 and C.app_limited)
    BBR.idle_restart = true
    BBR.extra_acked_interval_start = Now()
    if (IsInAProbeBWState())
      BBRSetPacingRateWithGain(1)
    else if (BBR.state == ProbeRTT)
      BBRCheckProbeRTTDone()
```

4.4.2. Comparison with Previous Approaches

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.bw, typically achieving approximate rate balance and a full pipe after only one BBR.min_rtt has elapsed.

4.5. Updating Network Path Model Parameters

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. The following is a summary of each parameter, including its meaning and how the algorithm calculates and uses its value. We can group the parameter into three groups:

- *core state machine parameters
- *parameters to model the data rate
- *parameters to model the volume of in-flight data

4.5.1. BBR.round_count: Tracking Packet-Timed Round Trips

Several aspects of BBR depend on counting the progress of "packet-timed" round trips, which start at the transmission of some segment, and then end at the acknowledgement of that segment. BBR.round_count is a count of the number of these "packet-timed" round trips elapsed so far. 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 application idleness, variations in RTTs, or timer delays for retransmission timeouts, causing wall-clock-timed model parameter estimates to "time out" or to be "forgotten" too quickly to provide robustness.

BBR 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, the rate estimation algorithm in Section 4.5.2.1 records the amount of data thus far acknowledged as delivered:

```
packet.delivered = C.delivered
```

Upon receiving an ACK for a given data packet, the rate estimation algorithm in Section 4.5.2.1 updates the amount of data thus far acknowledged as delivered:

```
  C.delivered += packet.size
```

Upon receiving an ACK for a given data packet, the BBR algorithm first executes the following logic to see if a round trip has elapsed, and if so, increment the count of such round trips elapsed:

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

BBRStartRound():
  BBR.next_round_delivered = C.delivered
```

4.5.2. BBR.max_bw: Estimated Maximum Bandwidth

BBR.max_bw is BBR's estimate of the maximum bottleneck bandwidth available to data transmissions for the transport flow. At any time, a transport connection's data transmissions experience some 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 the departure rate. The bottleneck rate varies over the life

of a connection, so BBR continually estimates BBR.max_bw using recent signals.

4.5.2.1. Delivery Rate Samples

This section describes a generic algorithm for a transport protocol sender to estimate the current delivery rate of its data on the fly. This technique is used by BBR to get fresh, reliable, and inexpensive delivery rate information.

At a high level, the algorithm estimates the rate at which the network delivered the most recent flight of outbound data packets for a single flow. In addition, it tracks whether the rate sample was application-limited, meaning the transmission rate was limited by the sending application rather than the congestion control algorithm.

Each acknowledgment that cumulatively or selectively acknowledges that the network has delivered new data produces a rate sample which records the amount of data delivered over the time interval between the transmission of a data packet and the acknowledgment of that packet. The samples reflect the recent goodput through some bottleneck, which may reside either in the network or on the end hosts (sender or receiver).

4.5.2.2. Delivery Rate Sampling Algorithm Overview

4.5.2.2.1. Requirements

This algorithm can be implemented in any transport protocol that supports packet-delivery acknowledgment (so far, implementations are available for TCP [RFC9293] and QUIC [RFC9000]). This algorithm requires a small amount of added logic on the sender, and requires that the sender maintain a small amount of additional per-packet state for packets sent but not yet delivered. In the most general case it requires high-precision (microsecond-granularity or better) timestamps on the sender (though millisecond-granularity may suffice for lower bandwidths). It does not require any receiver or network changes. While selective acknowledgments for out-of-order data (e.g., [RFC2018]) are not required, such a mechanism is highly recommended for accurate estimation during reordering and loss recovery phases.

4.5.2.2.2. Estimating Delivery Rate

A delivery rate sample records the estimated rate at which the network delivered packets for a single flow, calculated over the time interval between the transmission of a data packet and the acknowledgment of that packet. Since the rate samples only include packets actually cumulatively and/or selectively acknowledged, the sender knows the exact octets that were delivered to the receiver

(not lost), and the sender can compute an estimate of a bottleneck delivery rate over that time interval.

The amount of data delivered MAY be tracked in units of either octets or packets. Tracking data in units of octets is more accurate, since packet sizes can vary. But for some purposes, including congestion control, tracking data in units of packets may suffice.

4.5.2.2.2.1. ACK Rate

First, consider the rate at which data is acknowledged by the receiver. In this algorithm, the computation of the ACK rate models the average slope of a hypothetical "delivered" curve that tracks the cumulative quantity of data delivered so far on the Y axis, and time elapsed on the X axis. Since ACKs arrive in discrete events, this "delivered" curve forms a step function, where each ACK causes a discrete increase in the "delivered" count that causes a vertical upward step up in the curve. This "ack_rate" computation is the average slope of the "delivered" step function, as measured from the "knee" of the step (ACK) preceding the transmit to the "knee" of the step (ACK) for packet P.

Given this model, the ack rate sample "slope" is computed as the ratio between the amount of data marked as delivered over this time interval, and the time over which it is marked as delivered:

```
ack_rate = data_acked / ack_elapsed
```

To calculate the amount of data ACKed over the interval, the sender records in per-packet state "P.delivered", the amount of data that had been marked delivered before transmitting packet P, and then records how much data had been marked delivered by the time the ACK for the packet arrives (in "C.delivered"), and computes the difference:

```
data_acked = C.delivered - P.delivered
```

To compute the time interval, "ack_elapsed", one might imagine that it would be feasible to use the round-trip time (RTT) of the packet. But it is not safe to simply calculate a bandwidth estimate by using the time between the transmit of a packet and the acknowledgment of that packet. Transmits and ACKs can happen out of phase with each other, clocked in separate processes. In general, transmissions often happen at some point later than the most recent ACK, due to processing or pacing delays. Because of this effect, drastic over-estimates can happen if a sender were to attempt to estimate bandwidth by using the round-trip time.

The following approach computes "ack_elapsed". The starting time is "P.delivered_time", the time of the delivery curve "knee" from the

ACK preceding the transmit. The ending time is "C.delivered_time", the time of the delivery curve "knee" from the ACK for P. Then we compute "ack_elapsed" as:

```
ack_elapsed = C.delivered_time - P.delivered_time
```

This yields our equation for computing the ACK rate, as the "slope" from the "knee" preceding the transmit to the "knee" at ACK:

```
ack_rate = data_acked / ack_elapsed
ack_rate = (C.delivered - P.delivered) /
           (C.delivered_time - P.delivered_time)
```

4.5.2.2.2.2. Compression and Aggregation

For computing the delivery_rate, the sender prefers ack_rate, the rate at which packets were acknowledged, since this usually the most reliable metric. However, this approach of directly using "ack_rate" faces a challenge when used with paths featuring aggregation, compression, or ACK decimation, which are prevalent [A15]. In such cases, ACK arrivals can temporarily make it appear as if data packets were delivered much faster than the bottleneck rate. To filter out such implausible ack_rate samples, we consider the send rate for each flight of data, as follows.

4.5.2.2.2.3. Send Rate

The sender calculates the send rate, "send_rate", for a flight of data as follows. Define "P.first_sent_time" as the time of the first send in a flight of data, and "P.sent_time" as the time the final send in that flight of data (the send that transmits packet "P"). The elapsed time for sending the flight is:

```
send_elapsed = (P.sent_time - P.first_sent_time)
```

Then we calculate the send_rate as:

```
send_rate = data_acked / send_elapsed
```

Using our "delivery" curve model above, the send_rate can be viewed as the average slope of a "send" curve that traces the amount of data sent on the Y axis, and the time elapsed on the X axis: the average slope of the transmission of this flight of data.

4.5.2.2.2.4. Delivery Rate

Since it is physically impossible to have data delivered faster than it is sent in a sustained fashion, when the estimator notices that the ack_rate for a flight is faster than the send rate for the

flight, it filters out the implausible ack_rate by capping the delivery rate sample to be no higher than the send rate.

More precisely, over the interval between each transmission and corresponding ACK, the sender calculates a delivery rate sample, "delivery_rate", using the minimum of the rate at which packets were acknowledged or the rate at which they were sent:

```
delivery_rate = min(send_rate, ack_rate)
```

Since ack_rate and send_rate both have data_acked as a numerator, this can be computed more efficiently with a single division (instead of two), as follows:

```
delivery_elapsed = max(ack_elapsed, send_elapsed)
delivery_rate = data_acked / delivery_elapsed
```

4.5.2.2.3. Tracking application-limited phases

In application-limited phases the transmission rate is limited by the sending application rather than the congestion control algorithm. Modern transport protocol connections are often application-limited, either due to request/response workloads (e.g., Web traffic, RPC traffic) or because the sender transmits data in chunks (e.g., adaptive streaming video).

Knowing whether a delivery rate sample was application-limited is crucial for congestion control algorithms and applications to use the estimated delivery rate samples properly. For example, congestion control algorithms likely do not want to react to a delivery rate that is lower simply because the sender is application-limited; for congestion control the key metric is the rate at which the network path can deliver data, and not simply the rate at which the application happens to be transmitting data at any moment.

To track this, the estimator marks a bandwidth sample as application-limited if there was some moment during the sampled flight of data packets when there was no data ready to send.

The algorithm detects that an application-limited phase has started when the sending application requests to send new data, or the connection's retransmission mechanisms decide to retransmit data, and the connection meets all of the following conditions:

1. The transport send buffer has less than one SMSS of unsent data available to send.

2. The sending flow is not currently in the process of transmitting a packet.

3. The amount of data considered in flight is less than the congestion window (cwnd).

4. All the packets considered lost have been retransmitted.

If these conditions are all met then the sender has run out of data to feed the network. This would effectively create a "bubble" of idle time in the data pipeline. This idle time means that any delivery rate sample obtained from this data packet, and any rate sample from a packet that follows it in the next round trip, is going to be an application-limited sample that potentially underestimates the true available bandwidth. Thus, when the algorithm marks a transport flow as application-limited, it marks all bandwidth samples for the next round trip as application-limited (at which point, the "bubble" can be said to have exited the data pipeline).

4.5.2.2.3.1. Considerations Related to Receiver Flow Control Limits

In some cases receiver flow control limits (such as the TCP [RFC9293] advertised receive window, RCV.WND) are the factor limiting the delivery rate. This algorithm treats cases where the delivery rate was constrained by such conditions the same as it treats cases where the delivery rate is constrained by in-network bottlenecks. That is, it treats receiver bottlenecks the same as network bottlenecks. This has a conceptual symmetry and has worked well in practice for congestion control and telemetry purposes.

4.5.2.3. Detailed Delivery Rate Sampling Algorithm

4.5.2.3.1. Variables

4.5.2.3.1.1. Per-connection (C) state

This algorithm requires the following new state variables for each transport connection:

C.delivered: The total amount of data (measured in octets or in packets) delivered so far over the lifetime of the transport connection. This does not include pure ACK packets.

C.delivered_time: The wall clock time when C.delivered was last updated.

C.first_sent_time: If packets are in flight, then this holds the send time of the packet that was most recently marked as delivered. Else, if the connection was recently idle, then this holds the send time of most recently sent packet.

C.app_limited: The index of the last transmitted packet marked as application-limited, or 0 if the connection is not currently application-limited.

We also assume that the transport protocol sender implementation tracks the following state per connection. If the following state variables are not tracked by an existing implementation, all the following parameters MUST be tracked to implement this algorithm:

C.write_seq: The data sequence number one higher than that of the last octet queued for transmission in the transport layer write buffer.

C.pending_transmissions: The number of bytes queued for transmission on the sending host at layers lower than the transport layer (i.e. network layer, traffic shaping layer, network device layer).

C.lost_out: The number of packets in the current outstanding window that are marked as lost.

C.retrans_out: The number of packets in the current outstanding window that are being retransmitted.

C.pipe: The sender's estimate of the amount of data outstanding in the network (measured in octets or packets). This includes data packets in the current outstanding window that are being transmitted or retransmitted and have not been SACKed or marked lost (e.g. "pipe" from [RFC6675]). This does not include pure ACK packets.

4.5.2.3.1.2. Per-packet (P) state

This algorithm requires the following new state variables for each packet that has been transmitted but not yet ACKed or SACKed:

P.delivered: C.delivered when the packet was sent from transport connection C.

P.delivered_time: C.delivered_time when the packet was sent.

P.first_sent_time: C.first_sent_time when the packet was sent.

P.is_app_limited: true if C.app_limited was non-zero when the packet was sent, else false.

P.sent_time: The time when the packet was sent.

4.5.2.3.1.3. Rate Sample (rs) Output

This algorithm provides its output in a RateSample structure rs, containing the following fields:

rs.delivery_rate: The delivery rate sample (in most cases rs.delivered / rs.interval).

rs.is_app_limited: The P.is_app_limited from the most recent packet delivered; indicates whether the rate sample is application-limited.

rs.interval: The length of the sampling interval.

rs.delivered: The amount of data marked as delivered over the sampling interval.

rs.prior_delivered: The P.delivered count from the most recent packet delivered.

rs.prior_time: The P.delivered_time from the most recent packet delivered.

rs.send_elapsed: Send time interval calculated from the most recent packet delivered (see the "Send Rate" section above).

rs.ack_elapsed: ACK time interval calculated from the most recent packet delivered (see the "ACK Rate" section above).

4.5.2.3.2. Transmitting or retransmitting a data packet

Upon transmitting or retransmitting a data packet, the sender snapshots the current delivery information in per-packet state. This will allow the sender to generate a rate sample later, in the UpdateRateSample() step, when the packet is (S)ACKed.

If there are packets already in flight, then we need to start delivery rate samples from the time we received the most recent ACK, to try to ensure that we include the full time the network needs to deliver all in-flight packets. If there are no packets in flight yet, then we can start the delivery rate interval at the current time, since we know that any ACKs after now indicate that the network was able to deliver those packets completely in the sampling interval between now and the next ACK.

After each packet transmission, the sender executes the following steps:

```
SendPacket(Packet P):
  if (SND.NXT == SND.UNA)  /* no packets in flight yet? */
    C.first_sent_time  = C.delivered_time = P.sent_time

  P.first_sent_time = C.first_sent_time
  P.delivered_time  = C.delivered_time
  P.delivered       = C.delivered
  P.is_app_limited  = (C.app_limited != 0)
```

4.5.2.3.3. Upon receiving an ACK

When an ACK arrives, the sender invokes GenerateRateSample() to fill in a rate sample. For each packet that was newly SACKed or ACKed, UpdateRateSample() updates the rate sample based on a snapshot of connection delivery information from the time at which the packet was last transmitted. UpdateRateSample() is invoked multiple times when a stretched ACK acknowledges multiple data packets. In this case we use the information from the most recently sent packet, i.e., the packet with the highest "P.delivered" value.

```
/* Upon receiving ACK, fill in delivery rate sample rs. */
GenerateRateSample(RateSample rs):
  for each newly SACKed or ACKed packet P
    UpdateRateSample(P, rs)

  /* Clear app-limited field if bubble is ACKed and gone. */
  if (C.app_limited and C.delivered > C.app_limited)
    C.app_limited = 0

  if (rs.prior_time == 0)
    return false  /* nothing delivered on this ACK */

  /* Use the longer of the send_elapsed and ack_elapsed */
  rs.interval = max(rs.send_elapsed, rs.ack_elapsed)

  rs.delivered = C.delivered - rs.prior_delivered

  /* Normally we expect interval >= MinRTT.
   * Note that rate may still be overestimated when a spuriously
   * retransmitted skb was first (s)acked because "interval"
   * is under-estimated (up to an RTT). However, continuously
   * measuring the delivery rate during loss recovery is crucial
   * for connections that suffer heavy or prolonged losses.
   */
  if (rs.interval <  MinRTT(tp))
    rs.interval = -1
    return false  /* no reliable sample */

  if (rs.interval != 0)
    rs.delivery_rate = rs.delivered / rs.interval

  return true;  /* we filled in rs with a rate sample */

/* Update rs when a packet is SACKed or ACKed. */
UpdateRateSample(Packet P, RateSample rs):
  if (P.delivered_time == 0)
    return /* P already SACKed */

  C.delivered += P.data_length
  C.delivered_time = Now()

  /* Update info using the newest packet: */
  if (!rs.has_data or IsNewestPacket(P, rs))
    rs.has_data         = true
    rs.prior_delivered  = P.delivered
    rs.prior_time       = P.delivered_time
    rs.is_app_limited   = P.is_app_limited
    rs.send_elapsed     = P.sent_time - P.first_sent_time
    rs.ack_elapsed      = C.delivered_time - P.delivered_time
    rs.last_end_seq     = P.end_seq
```

```
      C.first_sent_time   = P.sent_time

    /* Mark the packet as delivered once it's SACKed to
     * avoid being used again when it's cumulatively acked.
     */
    P.delivered_time = 0

  /* Is the given Packet the most recently sent packet
   * that has been delivered? */
  IsNewestPacket(Packet P, RateSample rs):
    return (P.sent_time > C.first_sent_time or
            (P.sent_time == C.first_sent_time and
             after(P.end_seq, rs.last_end_seq))
```

4.5.2.3.4. Detecting application-limited phases

An application-limited phase starts when the connection decides to send more data, at a point in time when the connection had previously run out of data. Some decisions to send more data are triggered by the application writing more data to the connection, and some are triggered by loss detection (during ACK processing or upon the triggering of a timer) estimating that some sequence ranges need to be retransmitted. To detect all such cases, the algorithm calls CheckIfApplicationLimited() to check for application-limited behavior in the following situations:

- The sending application asks the transport layer to send more data; i.e., upon each write from the application, before new application data is enqueued in the transport send buffer or transmitted.

- At the beginning of ACK processing, before updating the estimated number of packets in flight, and before congestion control modifies the cwnd or pacing rate.

- At the beginning of connection timer processing, for all timers that might result in the transmission of one or more data segments. For example: RTO timers, TLP timers, RACK reordering timers, or Zero Window Probe timers.

When checking for application-limited behavior, the connection checks all the conditions previously described in the "Tracking application-limited phases" section, and if all are met then it marks the connection as application-limited:

```
  CheckIfApplicationLimited():
    if (C.write_seq - SND.NXT < SND.MSS and
        C.pending_transmissions == 0 and
        C.pipe < cwnd and
        C.lost_out <= C.retrans_out)
      C.app_limited = (C.delivered + C.pipe) ? : 1
```

4.5.2.4. Delivery Rate Sampling Discussion

4.5.2.4.1. Offload Mechanisms

If a transport sender implementation uses an offload mechanism (such as TSO, GSO, etc.) to combine multiple SMSS of data into a single packet "aggregate" for the purposes of scheduling transmissions, then it is RECOMMENDED that the per-packet state be tracked for each packet "aggregate" rather than each SMSS. For simplicity this document refers to such state as "per-packet", whether it is per "aggregate" or per SMSS.

4.5.2.4.2. Impact of ACK losses

Delivery rate samples are generated upon receiving each ACK; ACKs may contain both cumulative and selective acknowledgment information. Losing an ACK results in losing the delivery rate sample corresponding to that ACK, and generating a delivery rate sample at later a time (upon the arrival of the next ACK). This can underestimate the delivery rate due the artificially inflated "rs.interval". As with any effect that can cause underestimation, it is RECOMMENDED that applications or congestion control algorithms using the output of this algorithm apply appropriate filtering to mitigate the impact of this effect.

4.5.2.4.3. Impact of packet reordering

This algorithm is robust to packet reordering; it makes no assumptions about the order in which packets are delivered or ACKed. In particular, for a particular packet P, it does not matter which packets are delivered between the transmission of P and the ACK of packet P, since C.delivered will be incremented appropriately in any case.

4.5.2.4.4. Impact of packet loss and retransmissions

There are several possible approaches for handling cases where a delivery rate sample is based on an ACK or SACK for a retransmitted packet.

If the transport protocol supports unambiguous ACKs for retransmitted data sequence ranges (as in QUIC [RFC9000]) then the algorithm is

perfectly robust to retransmissions, because the starting packet, P, for the sample can be unambiguously retrieved.

If the transport protocol, like TCP [RFC9293], has ambiguous ACKs for retransmitted sequence ranges, then the following approaches MAY be used:

1. The sender MAY choose to filter out implausible delivery rate samples, as described in the GenerateRateSample() step in the "Upon receiving an ACK" section, by discarding samples whose rs.interval is lower than the minimum RTT seen on the connection.

2. The sender MAY choose to skip the generation of a delivery rate sample for a retransmitted sequence range.

4.5.2.4.5. Connections without SACK support

If the transport connection does not use SACK (i.e., either or both ends of the connections do not accept SACK), then this algorithm can be extended to estimate approximate delivery rates using duplicate ACKs (much like Reno and [RFC5681] estimates that each duplicate ACK indicates that a data packet has been delivered). The details of this extension will be described in a future version of this draft.

4.5.3. BBR.max_bw Max Filter

Delivery rate samples are often below the typical 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 or along the network path. To filter these effects BBR uses a max filter: BBR estimates BBR.max_bw using the windowed maximum recent delivery rate sample seen by the connection over recent history.

The BBR.max_bw max filter window covers a time period extending over the past two ProbeBW cycles. The BBR.max_bw max filter window length is driven by trade-offs among several considerations:

* It is long enough to cover at least one entire ProbeBW cycle (see the "ProbeBW" section). 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 variable cross-traffic, lossy link layers with uncorrected losses, or short-term buffer exhaustion (e.g., brief coincident bursts in a shallow buffer).

* It 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.

* It aims to be short enough to respond in a timely manner to sustained reductions in the bandwidth available to a flow, whether this is because other flows are using a larger share of the bottleneck, or the bottleneck link service rate has reduced due to layer 1 or layer 2 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.max_bw estimates and thus pacing rate and in-flight data, in order to match the sending behavior to the new available bandwidth.

4.5.4. BBR.max_bw 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 Section 4.5.2.1. The BBR.max_bw estimator carefully decides which samples to include in the bandwidth model to ensure that BBR.max_bw 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.max_bw estimate, since this indicates the current BBR.Max_bw estimate is too low.

4.5.5. Updating the BBR.max_bw Max Filter

For every ACK that acknowledges some data packets as delivered, BBR invokes BBRUpdateMaxBw() to update the BBR.max_bw estimator as follows (here rs.delivery_rate is the delivery rate sample obtained from the ACK that is being processed, as specified in Section 4.5.2.1):

```
BBRUpdateMaxBw()
  BBRUpdateRound()
  if (rs.delivery_rate >= BBR.max_bw || !rs.is_app_limited)
      BBR.max_bw = update_windowed_max_filter(
                     filter=BBR.MaxBwFilter,
                     value=rs.delivery_rate,
                     time=BBR.cycle_count,
                     window_length=MaxBwFilterLen)
```

4.5.6. Tracking Time for the BBR.max_bw Max Filter

BBR tracks time for the BBR.max_bw filter window using a virtual (non-wall-clock) time tracked by counting the cyclical progression through ProbeBW cycles. Each time through the Probe bw cycle, one round trip after exiting ProbeBW_UP (the point at which the flow has its best chance to measure the highest throughput of the cycle), BBR increments BBR.cycle_count, the virtual time used by the BBR.max_bw filter window. Note that BBR.cycle_count only needs to be tracked with a single bit, since the BBR.max_bw filter only needs to track samples from two time slots: the previous ProbeBW cycle and the current ProbeBW cycle:

```
BBRAdvanceMaxBwFilter():
  BBR.cycle_count++
```

4.5.7. BBR.min_rtt: Estimated Minimum Round-Trip Time

BBR.min_rtt 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.bw 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 BBR.min_rtt using recent round-trip delay samples.

4.5.7.1. Round-Trip Time Samples for Estimating BBR.min_rtt

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.

4.5.7.2. BBR.min_rtt 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.min_rtt using the minimum recent RTT sample seen by the connection over that past MinRTTFilterLen 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 BBR.min_rtt min filter window is MinRTTFilterLen = 10 secs. This is driven by trade-offs among several considerations:

* The MinRTTFilterLen 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.

* The MinRTTFilterLen 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.

* The MinRTTFilterLen 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.min_rtt, 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.

*The MinRTTFilterLen 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 longer time scales.

A BBR implementation MAY use a generic windowed min filter to track BBR.min_rtt. However, a significant savings in space and improvement in freshness can be achieved by integrating the BBR.min_rtt estimation into the ProbeRTT state machine, so this document discusses that approach in the ProbeRTT section.

4.5.8. BBR.offload_budget

BBR.offload_budget is the estimate of the minimum volume of data necessary to achieve full throughput using sender (TSO/GSO) and receiver (LRO, GRO) host offload mechanisms, computed as follows:

```
BBRUpdateOffloadBudget():
  BBR.offload_budget = 3 * BBR.send_quantum
```

The factor of 3 is chosen to allow maintaining at least:

*1 quantum in the sending host's queuing discipline layer

*1 quantum being segmented in the sending host TSO/GSO engine

*1 quantum being reassembled or otherwise remaining unacknowledged due to the receiver host's LRO/GRO/delayed-ACK engine

4.5.9. BBR.extra_acked

BBR.extra_acked is a volume of data that is the estimate of the recent degree of aggregation in the network path. For each ACK, the algorithm computes a sample of the estimated extra ACKed data beyond the amount of data that the sender expected to be ACKed over the timescale of a round-trip, given the BBR.bw. Then it computes BBR.extra_acked as the windowed maximum sample over the last BBRExtraAckedFilterLen=10 packet-timed round-trips. If the ACK rate falls below the expected bandwidth, then the algorithm estimates an aggregation episode has terminated, and resets the sampling interval to start from the current time.

The BBR.extra_acked thus reflects the recently-measured magnitude of data and ACK aggregation effects such as batching and slotting at shared-medium L2 hops (wifi, cellular, DOCSIS), as well as end-host

offload mechanisms (TSO, GSO, LRO, GRO), and end host or middlebox ACK decimation/thinning.

BBR augments its cwnd by BBR.extra_acked to allow the connection to keep sending during inter-ACK silences, to an extent that matches the recently measured degree of aggregation.

More precisely, this is computed as:

```
BBRUpdateACKAggregation():
  /* Find excess ACKed beyond expected amount over this interval */
  interval = (Now() - BBR.extra_acked_interval_start)
  expected_delivered = BBR.bw * interval
  /* Reset interval if ACK rate is below expected rate: */
  if (BBR.extra_acked_delivered <= expected_delivered)
      BBR.extra_acked_delivered = 0
      BBR.extra_acked_interval_start = Now()
      expected_delivered = 0
  BBR.extra_acked_delivered += rs.newly_acked
  extra = BBR.extra_acked_delivered - expected_delivered
  extra = min(extra, cwnd)
  if (BBR.full_bw_reached)
    filter_len = BBRExtraAckedFilterLen
  else
    filter_len = 1  /* in Startup, just remember 1 round */
  BBR.extra_acked =
    update_windowed_max_filter(
      filter=BBR.ExtraACKedFilter,
      value=extra,
      time=BBR.round_count,
      window_length=filter_len)
```

4.5.10. Updating the Model Upon Packet Loss

In every state, BBR responds to (filtered) congestion signals, including loss. The response to those congestion signals depends on the flow's current state, since the information that the flow can infer depends on what the flow was doing when the flow experienced the signal.

4.5.10.1. Probing for Bandwidth In Startup

In Startup, if the congestion signals meet the Startup exit criteria, the flow exits Startup and enters Drain (see Section 4.3.1.3).

4.5.10.2. Probing for Bandwidth In ProbeBW

BBR searches for the maximum volume of data that can be sensibly placed in-flight in the network. A key precondition is that the flow is actually trying robustly to find that operating point. To

implement this, when a flow is in ProbeBW, and an ACK covers data sent in one of the accelerating phases (REFILL or UP), and the ACK indicates that the loss rate over the past round trip exceeds the queue pressure objective, and the flow is not application limited, and has not yet responded to congestion signals from the most recent REFILL or UP phase, then the flow estimates that the volume of data it allowed in flight exceeded what matches the current delivery process on the path, and reduces BBR.inflight_hi:

```
/* Do loss signals suggest inflight is too high?
 * If so, react. */
IsInflightTooHigh():
  if (IsInflightTooHigh(rs))
    if (BBR.bw_probe_samples)
      BBRHandleInflightTooHigh()
    return true  /* inflight too high */
  else
    return false /* inflight not too high */

IsInflightTooHigh():
  return (rs.lost > rs.tx_in_flight * BBRLossThresh)

BBRHandleInflightTooHigh():
  BBR.bw_probe_samples = 0;  /* only react once per bw probe */
  if (!rs.is_app_limited)
    BBR.inflight_hi = max(rs.tx_in_flight,
                          BBRTargetInflight() * BBRBeta))
  If (BBR.state == ProbeBW_UP)
    BBRStartProbeBW_DOWN()
```

Here rs.tx_in_flight is the amount of data that was estimated to be in flight when the most recently ACKed packet was sent. And the BBRBeta (0.7x) bound is to try to ensure that BBR does not react more dramatically than CUBIC's 0.7x multiplicative decrease factor.

Some loss detection algorithms, including algorithms like RACK [RFC8985] that delay loss marking while waiting for potential reordering to resolve, may mark packets as lost long after the loss itself happened. In such cases, the tx_in_flight for the delivered sequence range that allowed the loss to be detected may be considerably smaller than the tx_in_flight of the lost packet itself. In such cases using the former tx_in_flight rather than the latter can cause BBR.inflight_hi to be significantly underestimated. To avoid such issues, BBR processes each loss detection event to more precisely estimate the volume of in-flight data at which loss rates cross BBRLossThresh, noting that this may have happened mid-way through some TSO/GSO offload burst (represented as a "packet" in the pseudocode in this document). To estimate this threshold volume of data, we can solve for "lost_prefix" in the following way, where

inflight_prev represents the volume of in-flight data preceding this packet, and lost_prev represents the data lost among that previous in-flight data.

First we start with:

```
lost / inflight >= BBRLossThresh
```

Expanding this, we get:

```
(lost_prev + lost_prefix) /    >= BBRLossThresh
(inflight_prev + lost_prefix)
```

Solving for lost_prefix, we arrive at:

```
lost_prefix >= (BBRLossThresh * inflight_prev - lost_prev) /
                  (1 - BBRLossThresh)
```

In pseudocode:

```
BBRNoteLoss()
  if (!BBR.loss_in_round)   /* first loss in this round trip? */
    BBR.loss_round_delivered = C.delivered
  BBR.loss_in_round = 1

BBRHandleLostPacket(packet):
  BBRNoteLoss()
  if (!BBR.bw_probe_samples)
    return /* not a packet sent while probing bandwidth */
  rs.tx_in_flight = packet.tx_in_flight /* inflight at transmit */
  rs.lost = C.lost - packet.lost /* data lost since transmit */
  rs.is_app_limited = packet.is_app_limited;
  if (IsInflightTooHigh(rs))
    rs.tx_in_flight = BBRInflightHiFromLostPacket(rs, packet)
    BBRHandleInflightTooHigh(rs)

/* At what prefix of packet did losses exceed BBRLossThresh? */
BBRInflightHiFromLostPacket(rs, packet):
  size = packet.size
  /* What was in flight before this packet? */
  inflight_prev = rs.tx_in_flight - size
  /* What was lost before this packet? */
  lost_prev = rs.lost - size
  lost_prefix = (BBRLossThresh * inflight_prev - lost_prev) /
                (1 - BBRLossThresh)
  /* At what inflight value did losses cross BBRLossThresh? */
  inflight = inflight_prev + lost_prefix
  return inflight
```

4.5.10.3. When not Probing for Bandwidth

When not explicitly accelerating to probe for bandwidth (Drain, ProbeRTT, ProbeBW_DOWN, ProbeBW_CRUISE), BBR responds to loss by slowing down to some extent. This is because loss suggests that the available bandwidth and safe volume of in-flight data may have decreased recently, and the flow needs to adapt, slowing down toward the latest delivery process. BBR flows implement this response by reducing the short-term model parameters, BBR.bw_lo and BBR.inflight_lo.

When encountering packet loss when the flow is not probing for bandwidth, the strategy is to gradually adapt to the current measured delivery process (the rate and volume of data that is delivered through the network path over the last round trip). This applies generally: whether in fast recovery, RTO recovery, TLP recovery; whether application-limited or not.

There are two key parameters the algorithm tracks, to measure the current delivery process:

BBR.bw_latest: a 1-round-trip max of delivered bandwidth (rs.delivery_rate).

BBR.inflight_latest: a 1-round-trip max of delivered volume of data (rs.delivered).

Upon the ACK at the end of each round that encountered a newly-marked loss, the flow updates its model (bw_lo and inflight_lo) as follows:

```
      bw_lo     = max(       bw_latest, BBRBeta *       bw_lo )
inflight_lo     = max( inflight_latest, BBRBeta * inflight_lo )
```

This logic can be represented as follows:

```
/* Near start of ACK processing: */
BBRUpdateLatestDeliverySignals():
  BBR.loss_round_start = 0
  BBR.bw_latest       = max(BBR.bw_latest,       rs.delivery_rate)
  BBR.inflight_latest = max(BBR.inflight_latest, rs.delivered)
  if (rs.prior_delivered >= BBR.loss_round_delivered)
    BBR.loss_round_delivered = C.delivered
    BBR.loss_round_start = 1

/* Near end of ACK processing: */
BBRAdvanceLatestDeliverySignals():
  if (BBR.loss_round_start)
    BBR.bw_latest       = rs.delivery_rate
    BBR.inflight_latest = rs.delivered

BBRResetCongestionSignals():
  BBR.loss_in_round = 0
  BBR.bw_latest = 0
  BBR.inflight_latest = 0

/* Update congestion state on every ACK */
BBRUpdateCongestionSignals():
  BBRUpdateMaxBw()
  if (!BBR.loss_round_start)
    return  /* wait until end of round trip */
  BBRAdaptLowerBoundsFromCongestion()  /* once per round, adapt */
  BBR.loss_in_round = 0

/* Once per round-trip respond to congestion */
BBRAdaptLowerBoundsFromCongestion():
  if (BBRIsProbingBW())
    return
  if (BBR.loss_in_round())
    BBRInitLowerBounds()
    BBRLossLowerBounds()

/* Handle the first congestion episode in this cycle */
BBRInitLowerBounds():
  if (BBR.bw_lo == Infinity)
    BBR.bw_lo = BBR.max_bw
  if (BBR.inflight_lo == Infinity)
    BBR.inflight_lo = cwnd

/* Adjust model once per round based on loss */
BBRLossLowerBounds()
  BBR.bw_lo       = max(BBR.bw_latest,
                        BBRBeta * BBR.bw_lo)
  BBR.inflight_lo = max(BBR.inflight_latest,
                        BBRBeta * BBR.inflight_lo)
```

```
BBRResetLowerBounds():
  BBR.bw_lo       = Infinity
  BBR.inflight_lo = Infinity

BBRBoundBWForModel():
  BBR.bw = min(BBR.max_bw, BBR.bw_lo)
```

4.6. Updating Control Parameters

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

4.6.1. Summary of Control Behavior in the State Machine

The following table summarizes how BBR modulates the control parameters in each state. In the table below, the semantics of the columns are as follows:

- State: the state in the BBR state machine, as depicted in the "State Transition Diagram" section above.
- Tactic: The tactic chosen from the "State Machine Tactics" in Section 4.1.3: "accel" refers to acceleration, "decel" to deceleration, and "cruise" to cruising.
- Pacing Gain: the value used for BBR.pacing_gain in the given state.
- Cwnd Gain: the value used for BBR.cwnd_gain in the given state.
- Rate Cap: the rate values applied as bounds on the BBR.max_bw value applied to compute BBR.bw.
- Volume Cap: the volume values applied as bounds on the BBR.max_inflight value to compute cwnd.

The control behavior can be summarized as follows. Upon processing each ACK, BBR uses the values in the table below to compute BBR.bw in BBRBoundBWForModel(), and the cwnd in BBRBoundCwndForModel():

State	Tactic	Pacing Gain	Cwnd Gain	Rate Cap	Volume Cap
Startup	accel	2.77	2	N/A	N/A
Drain	decel	0.5	2	bw_lo	inflight_hi, inflight_lo
ProbeBW_DOWN	decel	0.90	2	bw_lo	inflight_hi, inflight_lo
ProbeBW_CRUISE	cruise	1.0	2	bw_lo	0.85*inflight_hi inflight_lo
ProbeBW_REFILL	accel	1.0	2		inflight_hi
ProbeBW_UP	accel	1.25	2.25		inflight_hi
ProbeRTT	decel	1.0	0.5	bw_lo	0.85*inflight_hi inflight_lo

4.6.2. Pacing Rate: BBR.pacing_rate

To help match the packet-arrival rate to the bottleneck bandwidth available to the flow, BBR paces data packets. Pacing enforces a maximum rate at which BBR schedules quanta of packets for transmission.

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

```
BBR.next_departure_time = max(Now(), BBR.next_departure_time)
packet.departure_time = BBR.next_departure_time
pacing_delay = packet.size / BBR.pacing_rate
BBR.next_departure_time = BBR.next_departure_time + pacing_delay
```

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

When a BBR flow starts it has no bw estimate (bw is 0). So in this case it sets an initial pacing rate based on the transport sender implementation's initial congestion window ("InitialCwnd", e.g. from [RFC6928]), 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 =  BBRStartupPacingGain * nominal_bandwidth
```

After initialization, on each data ACK BBR updates its pacing rate to be proportional to bw, as long as it estimates that it has filled the pipe (BBR.full_bw_reached is true; see the "Startup" section 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 bandwidth samples. BBR updates the pacing rate on each ACK by executing the BBRSetPacingRate() step as follows:

```
BBRSetPacingRateWithGain(pacing_gain):
  rate = pacing_gain * bw * (100 - BBRPacingMarginPercent) / 100
  if (BBR.full_bw_reached || rate > BBR.pacing_rate)
    BBR.pacing_rate = rate

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

To help drive the network toward lower queues and low latency while maintaining high utilization, the BBRPacingMarginPercent constant of 1 aims to cause BBR to pace at 1% below the bw, on average.

4.6.3. Send Quantum: BBR.send_quantum

In order to amortize per-packet overheads involved in the sending process (host CPU, NIC processing, and interrupt processing delays), high-performance transport sender implementations (e.g., Linux TCP) often schedule an aggregate containing multiple packets (multiple SMSS) worth of data as a single quantum (using TSO, GSO, or other offload mechanisms). The BBR congestion control algorithm makes this control decision explicitly, dynamically calculating a quantum control parameter that specifies the maximum size of these transmission aggregates. This decision is based on a trade-off:

- A smaller quantum is preferred at lower data rates because it results in shorter packet bursts, shorter queues, lower queueing delays, and lower rates of packet loss.

*A bigger 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():
  BBR.send_quantum = BBR.pacing_rate * 1ms
  BBR.send_quantum = min(BBR.send_quantum, 64 KBytes)
  BBR.send_quantum = max(BBR.send_quantum, 2 * SMSS)
```

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.

4.6.4. Congestion Window

The congestion window, or cwnd, controls the maximum volume of data BBR allows in flight in the network at any time. It is the maximum volume of in-flight data that the algorithm estimates is appropriate for matching the current network path delivery process, given all available signals in the model, at any time scale. BBR 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 BBR.max_inflight, which models what's required for achieving full throughput, and as such 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.min_rtt of the path then it cuts its cwnd accordingly. The following sections describe the various considerations that impact cwnd.

4.6.4.1. Initial cwnd

BBR generally uses measurements to build a model of the network path and then adapts 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).

4.6.4.2. Computing BBR.max_inflight

The BBR BBR.max_inflight 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.min_rtt, and has accumulated confidence in its model parameters by receiving enough ACKs to gradually grow the current cwnd to meet the BBR.max_inflight.

On each ACK, BBR calculates the BBR.max_inflight in BBRUpdateMaxInflight() as follows:

```
BBRBDPMultiple(gain):
  if (BBR.min_rtt == Infinity)
    return InitialCwnd /* no valid RTT samples yet */
  BBR.bdp = BBR.bw * BBR.min_rtt
  return gain * BBR.bdp

BBRQuantizationBudget(inflight)
  BBRUpdateOffloadBudget()
  inflight = max(inflight, BBR.offload_budget)
  inflight = max(inflight, BBRMinPipeCwnd)
    if (BBR.state == ProbeBW_UP)
    inflight += 2*SMSS
  return inflight

BBRInflight(gain):
  inflight = BBRBDPMultiple(gain)
  return BBRQuantizationBudget(inflight)

BBRUpdateMaxInflight():
  BBRUpdateAggregationBudget()
  inflight = BBRBDPMultiple(BBR.cwnd_gain)
  inflight += BBR.extra_acked
  BBR.max_inflight = BBRQuantizationBudget(inflight)
```

The "estimated_bdp" term tries to allow enough packets in flight to fully utilize the estimated BDP of the path, by allowing the flow to send at BBR.bw for a duration of BBR.min_rtt. Scaling up the BDP by BBR.cwnd_gain bounds in-flight data to a small multiple of the BDP, to handle common network and receiver behavior, such as delayed, stretched, or aggregated ACKs [A15]. The "quanta" term allows enough quanta in flight on the sending and receiving hosts to reach high throughput even in environments using offload mechanisms.

4.6.4.3. Minimum cwnd for Pipelining

For BBR.max_inflight, BBR imposes a floor of BBRMinPipeCwnd (4 packets, i.e. 4 * SMSS). This floor helps ensure that even at very low BDPs, and with a transport like TCP where a receiver may ACK only

every alternate SMSS 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.

4.6.4.4. 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 (RTO), BBR conservatively reduces cwnd to a value that will allow 1 SMSS to be transmitted. Then BBR gradually increases cwnd using the normal approach outlined below in "cwnd Adjustment Mechanism" in Section 4.6.4.6.

When a BBR sender is in Fast Recovery it uses the response described in "Updating the Model Upon Packet Loss" in Section 4.5.10.

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.

The high-level design for updating cwnd in loss recovery is as follows:

Upon retransmission timeout (RTO):

```
BBROnEnterRTO():
  BBRSaveCwnd()
  cwnd = C.pipe + 1
```

Upon entering Fast Recovery:

```
BBROnEnterFastRecovery():
  BBRSaveCwnd()
```

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:

```
BBRRestoreCwnd()
```

Note that exiting loss recovery happens during ACK processing, and at the end of ACK processing BBRBoundCwndForModel() will bound the cwnd based on the current model parameters. Thus the cwnd and pacing rate after loss recovery will generally be smaller than the values entering loss recovery.

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 (!InLossRecovery() and BBR.state != ProbeRTT)
    BBR.prior_cwnd = cwnd
  else
    BBR.prior_cwnd = max(BBR.prior_cwnd, cwnd)

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

4.6.4.5. 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.min_rtt. To implement this mode, BBR bounds the cwnd to BBRMinPipeCwnd, the minimal value that allows pipelining (see the "Minimum cwnd for Pipelining" section, above):

```
BBRProbeRTTCwnd():
  probe_rtt_cwnd = BBRBDPMultiple(BBR.bw, BBRProbeRTTCwndGain)
  probe_rtt_cwnd = max(probe_rtt_cwnd, BBRMinPipeCwnd)
  return probe_rtt_cwnd

BBRBoundCwndForProbeRTT():
  if (BBR.state == ProbeRTT)
    cwnd = min(cwnd, BBRProbeRTTCwnd())
```

4.6.4.6. 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.max_inflight derived from BBR's path model, BBR cuts the cwnd immediately to the BBR.max_inflight. When cwnd is below BBR.max_inflight, 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 "rs.newly_acked" packets as newly ACKed or SACKed, BBR runs the following BBRSetCwnd() steps to update cwnd:

```
BBRSetCwnd():
  BBRUpdateMaxInflight()
  if (BBR.full_bw_reached)
    cwnd = min(cwnd + rs.newly_acked, BBR.max_inflight)
  else if (cwnd < BBR.max_inflight || C.delivered < InitialCwnd)
    cwnd = cwnd + rs.newly_acked
  cwnd = max(cwnd, BBRMinPipeCwnd)
  BBRBoundCwndForProbeRTT()
  BBRBoundCwndForModel()
```

There are several considerations embodied in the logic above. If BBR has measured enough samples to achieve confidence that it has filled the pipe (see the description of BBR.full_bw_reached 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.max_inflight adapted to the estimated BDP. Otherwise, if the cwnd is below the BBR.max_inflight, or the sender has marked so little data delivered (less than InitialCwnd) that it does not yet judge its BBR.max_bw estimate and BBR.max_inflight as useful, then it increases cwnd without bounding it to be below BBR.max_inflight. Finally, BBR imposes a floor of BBRMinPipeCwnd in order to allow pipelining even with small BDPs (see the "Minimum cwnd for Pipelining" section, above).

4.6.4.7. Bounding cwnd Based on Recent Congestion

Finally, BBR bounds the cwnd based on recent congestion, as outlined in the "Volume Cap" column of the table in the "Summary of Control Behavior in the State Machine" section:

```
BBRBoundCwndForModel():
  cap = Infinity
  if (IsInAProbeBWState() and
      BBR.state != ProbeBW_CRUISE)
    cap = BBR.inflight_hi
  else if (BBR.state == ProbeRTT or
           BBR.state == ProbeBW_CRUISE)
    cap = BBRInflightWithHeadroom()

  /* apply inflight_lo (possibly infinite): */
  cap = min(cap, BBR.inflight_lo)
  cap = max(cap, BBRMinPipeCwnd)
  cwnd = min(cwnd, cap)
```

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 BBRv3 have been publicly released:

*Linux TCP

-Source code URL:

ohttps://github.com/google/bbr/blob/v3/README.md

ohttps://github.com/google/bbr/blob/v3/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: November 22, 2023

*QUIC

-Source code URLs:

ohttps://cs.chromium.org/chromium/src/net/third_party/quiche/src/quic/core/congestion_control/bbr2_sender.cc

ohttps://cs.chromium.org/chromium/src/net/third_party/quiche/src/quic/core/congestion_control/bbr2_sender.h

-Source: Google

-Maturity: production

-License: BSD-style

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

-Last updated: October 21, 2021

As of the time of writing, the following implementations of the delivery rate sampling algorithm have been publicly released:

*Linux TCP

-Source code URL:

oGPLv2 license: https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/tree/net/ipv4/tcp_rate.c

oBSD-style license: https://groups.google.com/d/msg/bbr-dev/X0LbDptlOzo/EVgkRjVHBQAJ

-Source: Google

-Maturity: production

-License: dual-licensed: GPLv2 / BSD-style

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

-Last updated: September 24, 2021

*QUIC

-Source code URLs:

ohttps://github.com/google/quiche/blob/main/quiche/quic/core/congestion_control/bandwidth_sampler.cc

ohttps://github.com/google/quiche/blob/main/quiche/quic/core/congestion_control/bandwidth_sampler.h

-Source: Google

-Maturity: production

-License: BSD-style

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

-Last updated: October 5, 2021

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 Kevin Yang, Priyaranjan Jha, Yousuk Seung, Luke Hsiao for their work on TCP BBR; Jana Iyengar, Victor Vasiliev, Bin Wu for their work on QUIC BBR; and Matt Mathis for his research work on the BBR algorithm and its implications [MM19]. We would also like to thank C. Stephen Gunn, Eric Dumazet, 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. We would like to thank Randall R. Stewart, Jim Warner, Loganaden Velvindron, Hiren Panchasara, Adrian Zapletal, Christian Huitema, Bao Zheng, Jonathan Morton, Matt Olson, and Junho Choi for feedback and suggestions on earlier versions of this document.

9. References

9.1. Normative References

[RFC2018] Mathis, M. and J. Mahdavi, "TCP Selective Acknowledgment Options", RFC 2018, October 1996, <http://www.rfc-editor.org/rfc/rfc2018.txt>.

[RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", RFC 2119, March 1997, <http://www.rfc-editor.org/rfc/rfc2119.txt>.

[RFC4340] Kohler, E., Handley, M., and S. Floyd, "Datagram Congestion Control Protocol (DCCP)", RFC 4340, DOI 10.17487/RFC4340, March 2006, <https://www.rfc-editor.org/info/rfc4340>.

[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, <https://tools.ietf.org/html/rfc5681>.

[RFC6298] Paxson, V., "Computing TCP's Retransmission Timer", RFC 6298, June 2011, <https://wiki.tools.ietf.org/html/rfc6298>.

[RFC6675] Blanton, E., Allman, M., Wang, L., Jarvinen, I., Kojo, M., and Y. Nishida, "A Conservative Loss Recovery Algorithm Based on Selective Acknowledgment (SACK) for TCP", RFC 6675, DOI 10.17487/RFC6675, August 2012, <https://www.rfc-editor.org/info/rfc6675>.

[RFC6928] Chu, J., Dukkipati, N., Cheng, Y., and M. Mathis, "Increasing TCP's Initial Window", RFC 6928, DOI 10.17487/RFC6928, April 2013, <https://www.rfc-editor.org/info/rfc6928>.

[RFC6937] Mathis, M., Dukkipati, N., and Y. Cheng, "Proportional Rate Reduction for TCP", RFC 6937, DOI 10.17487/RFC6937, May 2013, <https://www.rfc-editor.org/info/rfc6937>.

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

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

[RFC8985] Cheng, Y., Cardwell, N., Dukkipati, N., and P. Jha, "The RACK-TLP Loss Detection Algorithm for TCP", RFC 8985, DOI 10.17487/RFC8985, February 2021, <https://www.rfc-editor.org/info/rfc8985>.

[RFC9000] Iyengar, J., Ed. and M. Thomson, Ed., "QUIC: A UDP-Based Multiplexed and Secure Transport", RFC 9000, DOI 10.17487/

RFC9000, May 2021, <https://www.rfc-editor.org/info/rfc9000>.

[RFC9293] Eddy, W., Ed., "Transmission Control Protocol (TCP)", STD 7, RFC 9293, DOI 10.17487/RFC9293, August 2022, <https://www.rfc-editor.org/info/rfc9293>.

[RFC9438] Xu, L., Ha, S., Rhee, I., Goel, V., and L. Eggert, Ed., "CUBIC for Fast and Long-Distance Networks", RFC 9438, DOI 10.17487/RFC9438, August 2023, <https://www.rfc-editor.org/info/rfc9438>.

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>.

[BBRDrainPacingGain] Cardwell, N., Cheng, Y., Hassas Yeganeh, S., and V. Jacobson, "BBR Drain Pacing Gain: a Derivation", September 2021, <https://github.com/google/bbr/blob/master/Documentation/startup/gain/analysis/bbr_drain_gain.pdf>.

[BBRStartupCwndGain] Swett, I., Cardwell, N., Cheng, Y., Hassas Yeganeh, S., and V. Jacobson, "BBR Startup cwnd Gain: a Derivation", July 2018, <https://github.com/google/bbr/blob/master/Documentation/startup/gain/analysis/bbr_startup_cwnd_gain.pdf>.

[BBRStartupPacingGain] Cardwell, N., Cheng, Y., Hassas Yeganeh, S., and V. Jacobson, "BBR Startup Pacing Gain: a Derivation", June 2018, <https://github.com/google/bbr/blob/master/Documentation/startup/gain/analysis/bbr_startup_gain.pdf>.

[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, 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>.

[draft-romo-iccrg-ccid5] Romo, N., Kim, J., and M. Amend, "Profile for Datagram Congestion Control Protocol (DCCP) Congestion Control ID 5", Work in Progress, Internet-Draft, draft-romo-iccrg-ccid5, 25 October 2021, <https://tools.ietf.org/html/draft-romo-iccrg-ccid5>.

[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, <http://ee.lbl.gov/papers/congavoid.pdf>.

[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.

[MM19] Mathis, M. and J. Mahdavi, "Deprecating The TCP Macroscopic Model", Computer Communication Review, vol. 49, no. 5, pp. 63-68 , October 2019.

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

Authors' Addresses

Neal Cardwell (editor)
Google

Email: ncardwell@google.com

Ian Swett (editor)
Google

Email: ianswett@google.com

Joseph Beshay (editor)
Meta

Email: jbeshay@meta.com