

Internet Congestion Control Research Group
Internet-Draft
Intended status: Experimental
Expires: 8 September 2022

N. Cardwell
Y. Cheng
S. Hassas Yeganeh
I. Swett
V. Jacobson
Google
7 March 2022

BBR Congestion Control
draft-cardwell-iccrbbr-congestion-control-02

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 [RFC8312], 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 [RFC793] and QUIC [RFC9000]. This document specifies version 2 of the BBR algorithm, also sometimes referred to as BBRv2 or bbr2.

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 8 September 2022.

Copyright Notice

Copyright (c) 2022 IETF Trust and the persons identified as the document authors. All rights reserved.

This document is subject to BCP 78 and the IETF Trust's Legal Provisions Relating to IETF Documents (<https://trustee.ietf.org/license-info>) in effect on the date of publication of this document. Please review these documents carefully, as they describe your rights and restrictions with respect to this document. Code Components extracted from this document must include Revised BSD License text as described in Section 4.e of the Trust Legal Provisions and are provided without warranty as described in the Revised BSD License.

Table of Contents

| | |
|---|----|
| 1. Introduction | 3 |
| 2. Terminology | 5 |
| 2.1. Transport Connection State | 5 |
| 2.2. Per-Packet State | 5 |
| 2.3. Per-ACK Rate Sample State | 5 |
| 2.4. Output Control Parameters | 6 |
| 2.5. Pacing State and Parameters | 6 |
| 2.6. cwnd State and Parameters | 7 |
| 2.7. General Algorithm State | 7 |
| 2.8. Core Algorithm Design Parameters | 7 |
| 2.9. Network Path Model Parameters | 8 |
| 2.9.1. Data Rate Network Path Model Parameters | 8 |
| 2.9.2. Data Volume Network Path Model Parameters | 8 |
| 2.10. State for Responding to Congestion | 9 |
| 2.11. Estimating BBR.max_bw | 10 |
| 2.12. Estimating BBR.extra_acked | 10 |
| 2.13. Startup Parameters and State | 10 |
| 2.14. ProbeRTT and min_rtt Parameters and State | 10 |
| 2.14.1. Parameters for Estimating BBR.min_rtt | 10 |
| 2.14.2. Parameters for Scheduling ProbeRTT | 11 |
| 3. Design Overview | 11 |
| 3.1. High-Level Design Goals | 11 |
| 3.2. Algorithm Overview | 12 |
| 3.3. State Machine Overview | 13 |
| 3.4. Network Path Model Overview | 13 |
| 3.4.1. High-Level Design Goals for the Network Path Model | 13 |
| 3.4.2. Time Scales for the Network Model | 14 |
| 3.5. Control Parameter Overview | 15 |
| 3.6. Environment and Usage | 15 |
| 4. Detailed Algorithm | 15 |
| 4.1. State Machine | 15 |
| 4.1.1. State Transition Diagram | 15 |

| | | |
|--------|--|----|
| 4.1.2. | State Machine Operation Overview | 16 |
| 4.1.3. | State Machine Tactics | 17 |
| 4.2. | Algorithm Organization | 18 |
| 4.2.1. | Initialization | 18 |
| 4.2.2. | Per-Transmit Steps | 18 |
| 4.2.3. | Per-ACK Steps | 19 |
| 4.2.4. | Per-Loss Steps | 19 |
| 4.3. | State Machine Operation | 19 |
| 4.3.1. | Startup | 19 |
| 4.3.2. | Drain | 22 |
| 4.3.3. | ProbeBW | 23 |
| 4.3.4. | ProbeRTT | 34 |
| 4.4. | Restarting From Idle | 39 |
| 4.4.1. | Setting Pacing Rate in ProbeBW | 39 |
| 4.4.2. | Checking for ProberRTT Completion | 40 |
| 4.4.3. | Logic | 40 |
| 4.5. | Updating Network Path Model Parameters | 40 |
| 4.5.1. | BBR.round_count: Tracking Packet-Timed Round Trips | 41 |
| 4.5.2. | BBR.max_bw: Estimated Maximum Bandwidth | 42 |
| 4.5.3. | BBR.min_rtt: Estimated Minimum Round-Trip Time | 44 |
| 4.5.4. | BBR.offload_budget | 46 |
| 4.5.5. | BBR.extra_acked | 47 |
| 4.5.6. | Updating the Model Upon Packet Loss | 48 |
| 4.6. | Updating Control Parameters | 52 |
| 4.6.1. | Summary of Control Behavior in the State Machine | 52 |
| 4.6.2. | Pacing Rate: BBR.pacing_rate | 53 |
| 4.6.3. | Send Quantum: BBR.send_quantum | 55 |
| 4.6.4. | Congestion Window | 55 |
| 5. | Implementation Status | 61 |
| 6. | Security Considerations | 62 |
| 7. | IANA Considerations | 62 |
| 8. | Acknowledgments | 63 |
| 9. | References | 63 |
| 9.1. | Normative References | 63 |
| 9.2. | Informative References | 64 |
| | Authors' Addresses | 66 |

1. Introduction

The Internet has traditionally used loss-based congestion control algorithms like Reno ([Jac88], [Jac90], [WS95] [RFC5681]) and CUBIC ([HRX08], [RFC8312]). 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, multiplicatively decreasing the sending rate upon packet loss, 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 [RFC8312]. 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 throughput 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, BBRv2. The previous 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 [draft-cheng-iccr-g-delivery-rate-estimation].

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.

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).

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

BBRStartupCwndGain: A constant specifying the minimum gain value for calculating the cwnd that will allow the sending rate to double each round (2.0); used in Startup mode for BBR.cwnd_gain.

BBR.packet_conservation: A boolean indicating whether BBR is currently using packet conservation dynamics to bound cwnd.

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 attempting to leave 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 (the value is 0.85).

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 [draft-cheng-iccrq-delivery-rate-estimation] - 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_hi: The long-term maximum sending bandwidth that the algorithm estimates will produce acceptable queue pressure, based on signals in the current or previous bandwidth probing cycle, as measured by loss. (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 or bw_hi (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, bw_hi, 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 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`: Analogous to `BBR.bw_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: 10 (in units of packet-timed round trips).

2.13. Startup Parameters and State

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

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 $O(\log(\text{BDP}))$ 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 $1.5 * \text{estimated_BDP}$
- * 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 simple state machine that aims for high throughput, low latency, 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 minimum data rate and data volume required to fully utilize the fair share of the bottleneck bandwidth available to the flow. This must incorporate estimates of the maximum available bandwidth (`BBR.max_bw`), the BDP of the path (`BBR.bdp`), and the requirements of any offload features on the end hosts or mechanisms on the network path that produce aggregation effects (summing up to `BBR.max_inflight`).

- * 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 queue 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 [RFC793] 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-iccr-g-ccid5].

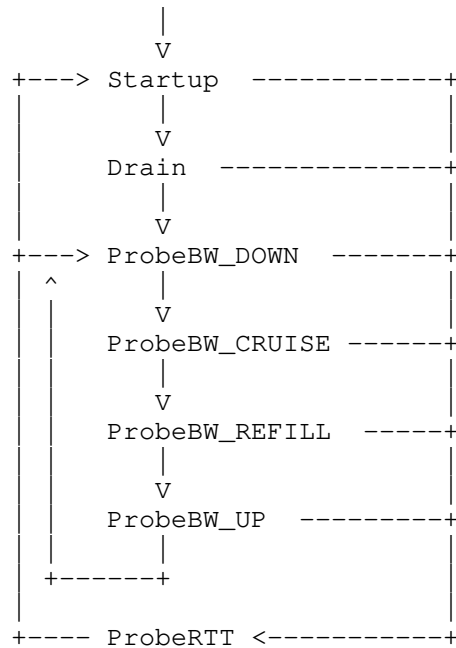
4. Detailed Algorithm

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 than the network is delivering data: to probe the maximum bandwidth available to the flow
- * 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
- * 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

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():
    init_windowed_max_filter(filter=BBR.MaxBwFilter, value=0, time=0)
    BBR.min_rtt = SRTT ? SRTT : Inf
    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
    BBRResetCongestionSignals()
    BBRResetLowerBounds()
    BBRInitRoundCounting()
    BBRInitFullPipe()
    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()
    BBRCheckStartupDone()
    BBRCheckDrain()
    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 200 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 in the smoothest possible fashion, 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 `BBRStartupCwndGain` (2).

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 = BBRStartupCwndGain
```

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 $(2.77 - 1) * estimated_BDP = 1.77 * estimated_BDP$. 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():
    BBRCheckStartupFullBandwidth()
    BBRCheckStartupHighLoss()
    if (BBR.state == Startup and BBR.filled_pipe)
        BBREnterDrain()
```

4.3.1.2. Exiting Startup Based on Bandwidth Plateau

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

Upon connection initialization the full pipe estimator runs:

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

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

```
BBRCheckStartupFullBandwidth():
    if BBR.filled_pipe or
        !BBR.round_start or rs.is_app_limited
        return /* no need to check for a full pipe now */
    if (BBR.max_bw >= BBR.full_bw * 1.25) /* still growing? */
        BBR.full_bw = BBR.max_bw /* record new baseline level */
        BBR.full_bw_count = 0
        return
    BBR.full_bw_count++ /* another round w/o much growth */
    if (BBR.full_bw_count >= 3)
        BBR.filled_pipe = true
```

BBR waits three rounds 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 is by looking at sustained packet losses. Specifically for a case where the following criteria are all met:

- * The connection has been in fast recovery for at least one full round trip.
- * The loss rate over the time scale of a single full round trip exceeds `BBRLossThresh` (2%).
- * There are at least `BBRStartupFullLossCnt=3` discontinuous sequence ranges lost in that round trip.

If these criteria are all met, then `BBRCheckStartupHighLoss()` sets `BBR.filled_pipe = true` and 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 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` that is the inverse of the value used during Startup, chosen to try to drain the queue in one round [`BBRDrainPacingGain`]:

```
BBREnterDrain():
    BBR.state = Drain
    BBR.pacing_gain = 1/BBRStartupCwndGain /* pace slowly */
    BBR.cwnd_gain = BBRStartupCwndGain    /* 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 has been fully drained, then BBR exits Drain and enters ProbeBW. To implement this, upon every ACK BBR executes:

```
BBRCheckDrain():
    if (BBR.state == Drain and packets_in_flight <= 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", above). It does this by switching to a `BBR.pacing_gain` of 0.9, sending at 90% of `BBR.bw`. The `pacing_gain` value of 0.9 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.

Exit conditions: The flow exits this 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 DOWN at least until the volume of in-flight data is less than or equal to `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 volume-based 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", above), 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, at which time it enters ProbeBW_REFILL (see "Time Scale for Bandwidth Probing", below).

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", above) 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 bw_hi 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 a loss signal. 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 $BBRHeadroom * inflight_hi$.

Exit conditions: The flow exits ProbeBW_REFILL after one packet-timed round trip, and enters 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 estimate permits. And 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 using a $BBR.pacing_gain$ of 1.25, sending faster than the current estimated available bandwidth.

If the flow has not set $BBR.inflight_hi$ or $BBR.bw_hi$, it tries to raise the volume of in-flight data to at least $BBR.pacing_gain * BBR.bdp = 1.25 * BBR.bdp$; note that this may take more than $BBR.min_rtt$ if $BBR.min_rtt$ is small (e.g. on a LAN).

If the flow has set $BBR.inflight_hi$ or $BBR.bw_hi$, it moves to an operating point based on those limits and then gradually increases the upper volume bound ($BBR.inflight_hi$) and rate bound ($BBR.bw_hi$) using the following approach:

- * bw_hi : The flow raises bw_hi to the latest measured bandwidth sample if the latest measured bandwidth sample is above bw_hi and the loss rate for the sample is not above the $BBRLossThresh$.

- * `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. 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 $O(1,000,000)$; even a quite typical BDP like $10\text{Gbps} * 100\text{ms}$ is 82,563 packets. 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 $O(\log(\text{BDP}))$ 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 any of the following conditions are met:

1. Estimated queue: The flow has been in `ProbeBW_UP` for at least $1 * \text{min_rtt}$, and the estimated queue is high enough that the flow judges it has robustly probed for available bandwidth ($\text{packets_in_flight} > 1.25 * \text{BBR.bdp}$).
2. Loss: The current loss rate 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: we want Reno/CUBIC to be able to perform well in 100M through 40G enterprise and datacenter Ethernet

- $BDP = 40 \text{ Gbps} * 20 \text{ us} / (1514 \text{ bytes}) \approx 66 \text{ packets}$
- * Public Internet last mile traffic: we want 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:
 - $BDP = 25\text{Mbps} * 30 \text{ ms} / (1514 \text{ bytes}) \approx 62 \text{ packets}$

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. Secondly, 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
 - $T_{\text{reno_bound}} = \text{pick_randomly_either}(\{62, 63\})$
 - $\text{reno_bdp} = \min(\text{BBR.bdp}, \text{cwnd})$
 - $T_{\text{reno}} = \min(\text{reno_bdp}, T_{\text{reno_bound}}) \text{ round trips}$
- * T_{probe} : The time between bandwidth probe UP phases:
 - $T_{\text{probe}} = \min(T_{\text{bbr}}, T_{\text{reno}})$

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 MSS per round trip, the inter-bandwidth-probe time must be at least: $BDP * RTT = 25Mbps * .030 \text{ sec} / (1514 \text{ bytes}) * 0.030 \text{ sec} = 1.9\text{secs}$
- * 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 (e.g. Netflix Reno 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 larger than roughly 65 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 round trip with X% packet loss is to reduce the sending rate by X% (see the "Updating the Model Upon Packet Loss" section), this means that the BBR sending rate after N rounds of packet loss at a rate `loss_rate` is $(1 - \text{loss_rate})^N$. This means that for a flow that encounters 1% loss in 65 round trips of `ProbeBW_CRUISE`, and then doubles its `cwnd` (back to `BBR.inflight_hi`) in `ProbeBW_REFILL` and `ProbeBW_UP`, it will be able to restore and reprobe its original sending rate, since: $\text{BBW.max_bw} * (1 - \text{loss_rate})^N * 2 = \text{BBR.max_bw} * (1 - .01)^{65} \approx 1.04 * \text{BBR.max_bw}$. That is, 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`.

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? */
BBRCheckTimeToProbeBW():
    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 =
        2sec + random_float_between(0.0, 1.0) /* 0..1 sec */

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():
    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()
    BBR.cycle_stamp = Now() /* start wall clock */
    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.filled_pipe)
        return /* only handling steady-state behavior here */
    BBRAdaptUpperBounds()
    if (!IsInAProbeBWState())
        return /* only handling ProbeBW states here: */

    switch (state)

    ProbeBW_DOWN:
        if (BBRCheckTimeToProbeBW())
            return /* already decided state transition */
        if (BBRCheckTimeToCruise())
            BBRStartProbeBW_CRUISE()

    ProbeBW_CRUISE:
        if (BBRCheckTimeToProbeBW())
            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 (BBRHasElapsedInPhase(BBR.min_rtt) and
            inflight > BBRInflight(BBR.max_bw, 1.25))
            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? */
BBRCheckTimeToCruise():
    if (inflight > BBRInflightWithHeadroom())
        return false /* not enough headroom */
    if (inflight <= BBRInflight(BBR.max_bw, 1.0))
        return true /* inflight <= estimated BDP */

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, BBRHeadroom * BBR.inflight_hi)
    return max(BBR.inflight_hi - headroom,
               BBRMinPipeCwnd)

/* Raise inflight_hi slope if appropriate. */
BBRRaiseInflightHiSlope():
    growth_this_round = 1MSS << 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 and BBR.bw_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 (!CheckInflightTooHigh())
        /* Loss rate is safe. Adjust upper bounds upward. */
        if (BBR.inflight_hi == Infinity or BBR.bw_hi == Infinity)
            return /* no upper bounds to raise */
        if (rs.tx_in_flight > BBR.inflight_hi)
            BBR.inflight_hi = rs.tx_in_flight
```

```
if (rs.delivery_rate > BBR.bw_hi)
    BBR.bw_hi = rs.bw
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 allow the flow 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 ProbeRTT 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 $\text{BBRProbeRTTCwndGain} * \text{BBR.bdp}$ for at least ProbeRTTDuration (200 ms) and at least one 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 $\text{ProbeRTTDuration} = 200$ ms) to allow flows with different RTTs to have overlapping ProbeRTT states, while still being short enough to bound the throughput penalty of ProbeRTT 's cwnd capping to roughly 2%, with the average throughput targeted at:

$$\begin{aligned} \text{throughput} &= (200\text{ms} * 0.5 * \text{BBR.bw} + (5\text{s} - 200\text{ms}) * \text{BBR.bw}) / 5\text{s} \\ &= (.1\text{s} + 4.8\text{s}) / 5\text{s} * \text{BBR.bw} = 0.98 * \text{BBR.bw} \end{aligned}$$

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 flow MUST use the standard ProbeRTTInterval of 5 secs.

An 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 $\text{BBR.probe_rtt_min_delay}$ filter opportunistically picks up these measurements, and the $\text{BBR.probe_rtt_min_delay}$ 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 $\text{BBR.probe_rtt_min_delay}$ 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 the "Pacing Rate: `BBR.pacing_rate`" section 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
        packets_in_flight <= 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 + packets_in_flight) ? : 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.filled_pipe)
        BBRStartProbeBW_DOWN()
        BBRStartProbeBW_CRUISE()
    else
        BBREnterStartup()
```

4.4. Restarting From Idle

4.4.1. Setting Pacing Rate in ProbeBW

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 at the moment the flow sends one or more packets BBR sets `BBR.pacing_rate` to exactly `BBR.bw`. More precisely, the BBR algorithm takes the following steps in `BBRHandleRestartFromIdle()` before sending a packet for a flow.

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 $(\log_2(\text{BDP}) * \text{RTT})$ 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.

4.4.2. Checking for ProberRTT Completion

As an optimization, 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.

4.4.3. Logic

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

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

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 the BBR algorithm 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 [draft-cheng-iccr-g-delivery-rate-estimation] 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 [draft-cheng-iccr-g-delivery-rate-estimation] 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_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 for Estimating BBR.max_bw

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

4.5.2.2. 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.

4.5.2.3. 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 [draft-cheng-iccr-g-delivery-rate-estimation]. 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.

4.5.2.4. 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 [draft-cheng-iccr-g-delivery-rate-estimation]):

```
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)
```

4.5.2.5. 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++
```

4.5.3. 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.

4.5.3.1. 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.

4.5.3.2. 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.

4.5.4. 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

4.5.5. 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)
BBR.extra_acked =
    update_windowed_max_filter(
        filter=BBR.ExtraACKedFilter,
        value=extra,
        time=BBR.round_count,
        window_length=BBRExtraAckedFilterLen)
```

4.5.6. 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.

4.5.6.1. Probing for Bandwidth In Startup

In Startup, if the congestion signals meet the Startup exit criteria, the flow exits Startup and enters Drain.

4.5.6.2. 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. */
```

```
CheckInflightTooHigh():
```

```
  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 packet. To estimate this value, we can solve for "lost_prefix" in the following equation, where `inflight_prev` represents the volume of in-flight data preceding this packet, `lost_prev` represents the data lost among that previous in-flight data:

```

lost           /  inflight           >= BBRLossThresh
(lost_prev + lost_prefix) / (inflight_prev + lost_prefix) >= BBRLossThresh
/* solving for lost_prefix we arrive at: */
lost_prefix = (BBRLossThresh * inflight_prev - lost_prev) / (1 - BBRLossThresh
)

```

In pseudocode:

```

BBRHandleLostPacket(packet):
    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

```

4.5.6.3. 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 (rs.losses > 0)
        BBR.loss_in_round = 1
    if (!BBR.loss_round_start)
        return /* wait until end of round trip */
    BBRAdaptLowerBoundsFromCongestion()
    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, BBR.bw_hi)

```

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" subsection above: "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 | | |
| Drain | decel | 0.5 | 2 | bw_hi, bw_lo | inflight_hi, inflight_lo |
| ProbeBW_DOWN | decel | 0.9 | 2 | bw_hi, bw_lo | inflight_hi, inflight_lo |
| ProbeBW_CRUISE | cruise | 1.0 | 2 | bw_hi, bw_lo | 0.85*inflight_hi inflight_lo |
| ProbeBW_REFILL | accel | 1.0 | 2 | bw_hi | inflight_hi |
| ProbeBW_UP | accel | 1.25 | 2 | bw_hi | inflight_hi |
| ProbeRTT | decel | 1.0 | 0.5 | bw_hi, 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.filled_pipe 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.filled_pipe || 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():
    if (BBR.pacing_rate < 1.2 Mbps)
        floor = 1 * SMSS
    else
        floor = 2 * SMSS
    BBR.send_quantum = min(BBR.pacing_rate * 1ms, 64KBytes)
    BBR.send_quantum = max(BBR.send_quantum, floor)
```

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 == Inf)
        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 && BBR.cycle_idx == ProbeBW_UP)
        inflight += 2
    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 "Core cwnd Adjustment Mechanism".

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

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

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

Upon retransmission timeout (RTO):

```
BBROnEnterRTO():
    BBR.prior_cwnd = BBRSaveCwnd()
    cwnd = packets_in_flight + 1
```

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

```
BBROnEnterFastRecovery():
    BBR.prior_cwnd = BBRSaveCwnd()
    cwnd = packets_in_flight + max(rs.newly_acked, 1)
    BBR.packet_conservation = true
```

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

```
BBRModulateCwndForRecovery():
    if (rs.newly_lost > 0)
        cwnd = max(cwnd - rs.newly_lost, 1)
    if (BBR.packet_conservation)
        cwnd = max(cwnd, packets_in_flight + rs.newly_acked)
```

After one round-trip in Fast Recovery:

```
BBR.packet_conservation = false
```

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

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

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)
        return cwnd
    else
        return 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. Core cwnd Adjustment Mechanism

The network path and traffic traveling over it can make sudden dramatic changes. To adapt to these changes smoothly and robustly, and reduce packet losses in such cases, BBR uses a conservative strategy. When cwnd is above the BBR.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()
  BBRModulateCwndForRecovery()
  if (!BBR.packet_conservation) {
    if (BBR.filled_pipe)
      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.filled_pipe in the "Startup" section below), then it increases its cwnd based on the number of packets delivered, while bounding its cwnd to be no larger than the BBR.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 BBR have been publicly released:

- * Linux TCP
 - Source code URL:
 - o <https://github.com/google/bbr/blob/v2alpha/README.md>
 - o https://github.com/google/bbr/blob/v2alpha/net/ipv4/tcp_bbr2.c

- Source: Google
- Maturity: production
- License: dual-licensed: GPLv2 / BSD
- Contact: <https://groups.google.com/d/forum/bbr-dev>
- Last updated: August 21, 2021

* QUIC

- Source code URLs:
 - o https://cs.chromium.org/chromium/src/net/third_party/quiche/src/quic/core/congestion_control/bbr2_sender.cc
 - o https://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

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, and Adrian Zapletal for feedback and suggestions on earlier versions of this document.

9. References

9.1. Normative References

- [RFC793] Postel, J., "Transmission Control Protocol", September 1981.
- [RFC2018] Mathis, M. and J. Mahdavi, "TCP Selective Acknowledgment Options", RFC 2018, October 1996,
<<http://www.rfc-editor.org/rfc/rfc2018.txt>>.
- [RFC7323] Borman, D., Braden, B., Jacobson, V., and R. Scheffenegger, "TCP Extensions for High Performance", September 2014.
- [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>>.
- [RFC5226] Narten, T. and H. Alvestrand, "Guidelines for Writing an IANA Considerations Section in RFCs", May 2008.
- [RFC6298] Paxson, V., "Computing TCP's Retransmission Timer", RFC 6298, June 2011,
<<https://wiki.tools.ietf.org/html/rfc6298>>.
- [RFC5681] Allman, M., Paxson, V., and E. Blanton, "TCP Congestion Control", RFC 5681, September 2009,
<<https://tools.ietf.org/html/rfc5681>>.

- [RFC7942] Sheffer, Y. and A. Farrel, "Improving Awareness of Running Code: The Implementation Status Section", July 2016.
- [RFC8312] Rhee, I., Xu, L., Ha, S., Zimmermann, A., Eggert, L., and R. Scheffenegger, "CUBIC for Fast Long-Distance Networks", February 2018, <<https://tools.ietf.org/html/rfc8312>>.
- [RFC8985] Cheng, Y., Cardwell, N., Dukkkipati, 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>>.
- [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>>.

9.2. Informative References

- [draft-cheng-iccr-g-delivery-rate-estimation]
Cheng, Y., Cardwell, N., Hassas Yeganeh, S., and V. Jacobson, "Delivery Rate Estimation", Work in Progress, Internet-Draft, draft-cheng-iccr-g-delivery-rate-estimation, November 2021, <<https://tools.ietf.org/html/draft-cheng-iccr-g-delivery-rate-estimation>>.
- [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>>.
- [MM19] Mathis, M. and J. Mahdavi, "Deprecating The TCP Macroscopic Model", Computer Communication Review, vol. 49, no. 5, pp. 63-68 , October 2019.

- [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>.
- [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>.
- [draft-romo-iccr-g-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-iccr-g-ccid5, 25 October 2021, <<https://tools.ietf.org/html/draft-romo-iccr-g-ccid5>>.
- [DC13] Dumazet, E. and Y. Cheng, "TSO, fair queuing, pacing: three's a charm", IETF 88 , November 2013, <<https://www.ietf.org/proceedings/88/slides/slides-88-tcpm-9.pdf>>.
- [A15] Abrahamsson, M., "TCP ACK suppression", IETF AQM mailing list , November 2015, <<https://www.ietf.org/mail-archive/web/aqm/current/msg01480.html>>.
- [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>>.
- [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.
- [B15] Brakmo, L., "TCP-NV: An Update to TCP-Vegas", , August 2015, <https://docs.google.com/document/d/1o-53jbO_xH-m9g2YCgjaf5bK8vePjWP6Mk0rYiRLK-U/edit>.
- [WS95] Wright, G. and W. Stevens, "TCP/IP Illustrated, Volume 2: The Implementation", Addison-Wesley , 1995.

- [HRX08] Ha, S., Rhee, I., and L. Xu, "CUBIC: A New TCP-Friendly High-Speed TCP Variant", ACM SIGOPS Operating System Review , 2008.
- [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>>.
- [K79] Kleinrock, L., "Power and deterministic rules of thumb for probabilistic problems in computer communications", Proceedings of the International Conference on Communications 1979.

Authors' Addresses

Neal Cardwell
Google
Email: ncardwell@google.com

Yuchung Cheng
Google
Email: ycheng@google.com

Soheil Hassas Yeganeh
Google
Email: soheil@google.com

Ian Swett
Google
Email: ianswett@google.com

Van Jacobson
Google
Email: vanj@google.com

CoRE Working Group
Internet-Draft
Intended status: Experimental
Expires: January 8, 2020

I. Jarvinen
M. Kojo
I. Raitahila
University of Helsinki
Z. Cao
Huawei
July 7, 2019

Fast-Slow Retransmission Timeout and Congestion Control Algorithm for
CoAP
draft-jarvinen-core-fasor-02

Abstract

This document specifies an alternative retransmission timeout and congestion control back off algorithm for the CoAP protocol, called Fast-Slow RTO (FASOR).

The algorithm specified in this document employs an appropriate and large enough back off of Retransmission Timeout (RTO) as the major congestion control mechanism to allow acquiring unambiguous RTT samples with high probability and to prevent building a persistent queue when retransmitting. The algorithm also aims to retransmit quickly using an accurately managed retransmission timeout when link-errors are occurring, basing RTO calculation on unambiguous round-trip time (RTT) samples.

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 January 8, 2020.

Copyright Notice

Copyright (c) 2019 IETF Trust and the persons identified as the document authors. All rights reserved.

This document is subject to BCP 78 and the IETF Trust's Legal Provisions Relating to IETF Documents (<https://trustee.ietf.org/license-info>) in effect on the date of publication of this document. Please review these documents carefully, as they describe your rights and restrictions with respect to this document. Code Components extracted from this document must include Simplified BSD License text as described in Section 4.e of the Trust Legal Provisions and are provided without warranty as described in the Simplified BSD License.

Table of Contents

| | |
|--|----|
| 1. Introduction | 2 |
| 2. Conventions | 3 |
| 3. Problems with Existing CoAP Congestion Control Algorithms . . | 3 |
| 4. FASOR Algorithm | 4 |
| 4.1. Computing Normal RTO (FastRTO) | 4 |
| 4.2. Slow RTO | 5 |
| 4.3. Retransmission Timeout Back Off Logic | 6 |
| 4.3.1. Overview | 6 |
| 4.3.2. Retransmission State Machine | 7 |
| 4.4. Retransmission Count Option | 9 |
| 4.5. Alternatives for Exchanging Retransmission Count Information | 11 |
| 5. Security Considerations | 11 |
| 6. IANA Considerations | 11 |
| 7. References | 11 |
| 7.1. Normative References | 11 |
| 7.2. Informative References | 12 |
| Appendix A. Pseudocode for Basic FASOR without Dithering | 13 |
| Authors' Addresses | 15 |

1. Introduction

CoAP senders use retransmission timeout (RTO) to infer losses that have occurred in the network. For such a heuristic to be correct, the RTT estimate used for calculating the retransmission timeout must match to the real end-to-end path characteristics. Otherwise, unnecessary retransmission may occur. Both default RTO mechanism for CoAP [RFC7252] and CoCoA [I-D.ietf-core-cocoa] have issues in dealing with unnecessary retransmissions and in the worst-case the situation can persist causing congestion collapse [JRCK18a].

This document specifies FASOR retransmission timeout and congestion control algorithm [JRCK18b]. FASOR algorithm ensures unnecessary retransmissions that a sender may have sent due to an inaccurate RTT estimate will not persist avoiding the threat of congestion collapse. FASOR also aims to quickly restore the accuracy of the RTT estimate. Armed with an accurate RTT estimate, FASOR not only handles congestion robustly but also can quickly infer losses due to link errors.

2. Conventions

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in BCP 14, RFC 2119 [RFC2119].

3. Problems with Existing CoAP Congestion Control Algorithms

Correctly inferring losses requires the retransmission timeout (RTO) to be longer than the real RTT in the network. Under certain circumstances the RTO may be incorrectly small. If the real end-to-end RTT is larger than the retransmission timeout, it is impossible for the sender to avoid making unnecessary retransmissions that duplicate data still existing in the network because the sender cannot receive any feedback in time. Unnecessary retransmissions cause two basic problems. First, they increase the perceived end-to-end RTT if the bottleneck has buffering capacity, and second, they prevent getting unambiguous RTT samples. Making unnecessary retransmissions is also a pre-condition for the congestion collapse [RFC0896], which may occur in the worst case if retransmissions are not well controlled [JRCK18a]. Therefore, the sender retransmission timeout algorithm should actively attempt to prevent unnecessary retransmissions from persisting under any circumstance.

Karn's algorithm [KP87] has prevented unnecessary retransmission from turning into congestion collapse for decades due to robust RTT estimation and retransmission timeout backoff handling. The recent CoAP congestion control algorithms, however, diverge from the principles of Karn's algorithm in significant ways and may pose a threat to the stability of the Internet due to those differences.

The default RTO mechanism for CoAP [RFC7252] uses only an initial RTO dithered between 2 and 3 seconds, while CoCoA [I-D.ietf-core-cocoa] measures RTT both from unambiguous and ambiguous RTT samples and applies a modified version of the TCP RTO algorithm [RFC6298]. The algorithm in RFC 7252 lacks solution to persistent congestion. The binary exponential back off used for the retransmission timeout does not properly address unnecessary retransmissions when RTT is larger

than the default RTO (ACK_TIMEOUT). If the CoAP sender performs exchanges over an end-to-end path with such a high RTT, it persistently keeps making unnecessary retransmissions for every exchange wasting some fraction of the used resources (network capacity, battery power).

CoCoA [I-D.ietf-core-cocoa] attempts to improve scenarios with link-error related losses and solve persistent congestion by basing its RTO value on an estimated RTT. However, there are couple of exceptions when the RTT estimation is not available:

- At the beginning of a flow where initial RTO of 2 seconds is used.
- When RTT suddenly jumps high enough to trigger the rule in CoCoA that prevents taking RTT samples when more than two retransmissions are needed. This may also occur when the packet drop rate on the path is high enough.

When RTT estimate is too small, unnecessary retransmission will occur also with CoCoA. CoCoA being unable to take RTT samples at all is a particularly problematic phenomenon as it is similarly persisting state as with the algorithm outlined in RFC 7252 and the network remains in a congestion collapsed state due to persisting unnecessary retransmissions.

4. FASOR Algorithm

FASOR [JRCK18b] is composed of three key components: RTO computation, Slow RTO, and novel retransmission timeout back off logic.

4.1. Computing Normal RTO (FastRTO)

The FASOR algorithm measures the RTT for an CoAP message exchange over an end-to-end path and computes the RTO value using the TCP RTO algorithm specified in [RFC6298]. We call this normal RTO or FastRTO. In contrast to the TCP RTO mechanism, FASOR SHOULD NOT use 1 second lower-bound when setting the RTO because RTO is only a backup mechanisms for loss detection with TCP, whereas with CoAP RTO is the primary and only loss detection mechanism. A lower-bound of 1 second would impact timeliness of the loss detection in low RTT environments. The RTO value MAY be upper-bounded by at least 60 seconds. A CoAP sender using the FASOR algorithm SHOULD set initial RTO to 2 seconds. The computed RTO value as well as the initial RTO value is subject to dithering; they are dithered between $RTO + 1/4 \times SRTT$ and $RTO + SRTT$. For dithering initial RTO, SRTT is unset; therefore, SRTT is replaced with initial RTO / 3 which is derived from the RTO formula and equals to a hypothetical initial RTT that

would yield the initial RTO using the SRTT and RTTVAR initialization rule of RFC 6298. That is, for initial RTO of 2 seconds we use SRTT value of 2/3 seconds.

FastRTO is updated only with unambiguous RTT samples. Therefore, it closely tracks the actual RTT of the network and can quickly trigger a retransmission when the network state is not dubious. Retransmitting without extra delay is very useful when the end-to-end path is subject to losses that are unrelated to congestion. When the first unambiguous RTT sample is received, the RTT estimator is initialized with that sample as specified in [RFC6298] except RTTVAR that is set to $R/2K$.

4.2. Slow RTO

We introduce Slow RTO as a safe way to ensure that only a unique copy of message is sent before at least one RTT has elapsed. To achieve this the sender must ensure that its retransmission timeout is set to a value that is larger than the path end-to-end RTT that may be inflated by the unnecessary retransmission themselves. Therefore, whenever a message needs to be retransmitted, we measure Slow RTO as the elapsed time required for getting an acknowledgement. That is, Slow RTO is measured starting from the original transmission of the request message until the receipt of the acknowledgement, regardless of the number of retransmissions. In this way, Slow RTO always covers the worst-case RTT during which a number of unnecessary retransmissions were made but the acknowledgement is received for the original transmission. In contrast to computing normal RTO, Slow RTO is not smoothed because it is derived from the sending pattern of the retransmissions (that may turn out unnecessary). In order to drain the potential unnecessary retransmissions successfully from the network, it makes sense to wait for the time used for sending them rather than some smoothed value. However, Slow RTO is multiplied by a factor to allow some growth in load without making Slow RTO too aggressive (by default the factor of 1.5 is used). FASOR then applies Slow RTO as one of the backed off timer values used with the next request message.

Slow RTO allows rapidly converging towards stable operating point because 1) it lets the duplicate copies sent earlier to drain from the network reducing the perceived end-to-end RTT, and 2) allows enough time to acquire an unambiguous RTT sample for the RTO computation. Robustly acquiring the RTT sample ensures that the next RTO is set according to the recent measurement and further unnecessary retransmissions are avoided. Slow RTO itself is a form of back off because it includes the accumulated time from the retransmission timeout back off of the previous exchange. FASOR uses this for its advantage as the time included into Slow RTO is what is

needed to drain all unnecessary retransmissions possibly made during the previous exchange. Assuming a stable RTT and that all of the retransmissions were unnecessary, the time to drain them is the time elapsed from the original transmission to the sending time of the last retransmission plus one RTT. When the acknowledgement for the original transmission arrives, one RTT has already elapsed, leaving only the sending time difference still unaccounted for which is at minimum the value for Slow RTO (when an RTT sample arrives immediately after the last retransmission). Even if RTT would be increasing, the draining still occurs rapidly due to exponentially backed off frequency in sending the unnecessary retransmissions.

4.3. Retransmission Timeout Back Off Logic

4.3.1. Overview

FASOR uses normal RTO as the base for binary exponential back off when no retransmission were needed for the previous CoAP message exchange. When retransmission were needed for the previous CoAP message exchange, the algorithm rules, however, are more complicated than with the traditional RTO back off because Slow RTO is injected into the back off series to reduce high impact of using Slow RTO. FASOR logic chooses from three possible back off series alternatives:

FAST back off: Perform traditional RTO back off with the normal RTO as the base. Applied when the previous message was not retransmitted.

FAST_SLOW_FAST back off: First perform a probe using the normal RTO for the original transmission of the request message to improve cases with losses unrelated to congestion. If the probe for the original transmission of the request message is successful without retransmissions, continue with FAST back off for the next message exchange. If the request message needs to be retransmitted, continue by using Slow RTO for the first retransmission in order to respond to congestion and drain the network from the unnecessary retransmissions that were potentially sent for the previous exchange. If still further RTOs are needed, continue by backing off the normal RTO further on each timeout. FAST_SLOW_FAST back off is applied just once when the previous request message using FAST back off required one or more retransmissions.

SLOW_FAST back off: Perform Slow RTO first for the original transmission to respond to congestion and to acquire an unambiguous RTT sample with high probability. Then, if the original request needs to be retransmitted, continue with the normal RTO-based RTO back off serie by backing off the normal RTO

on each timeout. SLOW_FAST back off is applied when the previous request message using FAST_SLOW_FAST or SLOW_FAST back off required one or more retransmissions. Once an acknowledgement for the original transmission with unambiguous RTT sample is received, continue with FAST back off for the next message exchange.

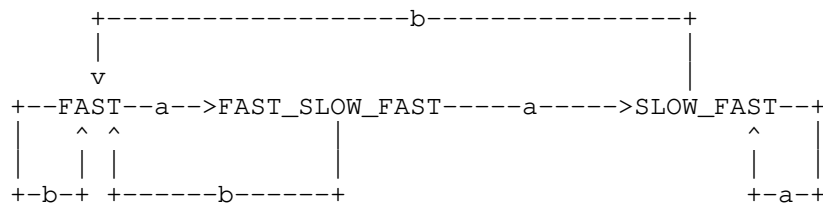
For the initial message, FAST is used with INITIAL_RTO as the FastRTO value. From there on, state is updated when an acknowledgement arrives. Following unambiguous RTT samples, FASOR always uses FAST. Whenever retransmissions are needed, the back off series selection is first downgraded to FAST_SLOW_FAST back off and then to SLOW_FAST back off if further retransmission are needed in FAST_SLOW_FAST.

When Slow RTO is used as the first RTO value, the sender is likely to acquire unambiguous RTT sample even when the network has high delay due to congestion because Slow RTO is based on a very recent measurement of the worst-case RTT. However, using Slow RTO may negatively impact the performance when losses unrelated to congestion are occurring. Due to its potential high cost, FASOR algorithm attempts to avoid using Slow RTO unnecessarily.

The CoAP protocol is often used by devices that are connected through a wireless network where non-congestion related losses are much more frequent than in their wired counterparts. This has implications for the retransmission timeout algorithm. While it would be possible to implement FASOR such that it immediately uses Slow RTO when a dubious network state is detected, which would handle congestion very well, it would do significant harm for performance when RTOs occur due to non-congestion related losses. Instead, FASOR uses first normal RTO for one transmission and only responds using Slow RTO if RTO expires also for that request message. Such a pattern quickly probes if the losses were unrelated to congestion and only slightly delays response if real congestion event is taking place. To ensure that an unambiguous RTT sample is also acquired on a congested network path, FASOR then needs to use Slow RTO for the original transmission of the subsequent packet if the probe was not successful.

4.3.2. Retransmission State Machine

FASOR consists of the three states discussed above while making retransmission decisions, FAST, FAST_SLOW_FAST and SLOW_FAST. The state machine of the FASOR algorithm is depicted in Figure 1.



a: retransmission acknowledged, ambiguous RTT sample acquired;
b: no retransmission, unambiguous RTT sample acquired;

Figure 1: State Machine of FASOR

In the FAST state, if the original transmission of the message has not been acknowledged by the receiver within the time defined by `FastRTO`, the sender will retransmit it. If there is still no acknowledgement of the retransmitted packet within $2 * \text{FastRTO}$, the sender performs the second retransmission and if necessary, each further retransmission applying binary exponential back off of `FastRTO`. The retransmission interval in this state is defined as `FastRTO`, $2^1 * \text{FastRTO}$, ..., $2^i * \text{FastRTO}$.

When there is an acknowledgement after any retransmission, the sender will calculate `SlowRTO` value based on the algorithm defined in Section 4.2.

When there is an acknowledgement after any retransmission, the sender will also switch to the second state, `FAST_FLOW_FAST`. In this state, the retransmission interval is defined as `FastRTO`, $\text{Max}(\text{SlowRTO}, 2 * \text{FastRTO})$, $\text{FastRTO} * 2^1$, ..., $2^i * \text{FastRTO}$. The state will be switched back to the FAST state once an acknowledgement is returned within `FastRTO`, i.e., no retransmission happens for a message. This is reasonable because it shows the network has recovered from congestion or bloated queue.

If some retransmission has been made before the acknowledged arrives in the `FAST_SLOW_FAST` state, the sender updates the `SlowRTO` value, and moves to the third state, `SLOW_FAST`. The retransmission interval in the `SLOW_FAST` state is defined as `SlowRTO`, `FastRTO`, $\text{FastRTO} * 2^1$, ..., $2^i * \text{FastRTO}$.

In `SLOW_FAST` state, the sender switches back to the FAST state if an unambiguous acknowledgement arrives. Otherwise, the sender stays in the `SLOW_FAST` state if retransmission happens again.

4.4. Retransmission Count Option

When retransmissions are needed to deliver a CoAP message, it is not possible to measure RTT for the RTO computation as the RTT sample becomes ambiguous. Therefore, it would be beneficial to be able to distinguish whether an acknowledgement arrives for the original transmission of the message or for a retransmission of it. This would allow reliably acquiring an RTT sample for every CoAP message exchange and thereby compute a more accurate RTO even during periods of congestion and loss.

The Retransmission Count Option is used to distinguish whether an Acknowledgement message arrives for the original transmission or one of the retransmissions of a Confirmable message. However, the Retransmission Count Option cannot be used with an Empty Acknowledgement (or Reset) message because the CoAP protocol specification [RFC7252] does not allow adding options to an Empty message. Therefore, Retransmission Count Option is useful only for the common case of Piggybacked Response. In case of Empty Acknowledgements the operation of FASOR is the same as without the option.

| No. | C | U | N | R | Name | Format | Length | Default |
|-----|---|---|---|---|------------|--------|--------|---------|
| TBD | | | X | | Rexmit-Cnt | uint | 0-1 | 0 |

C=Critical, U=Unsafe, N=NoCacheKey, R=Repeatable

Table 1: Retransmission Count Option

Implementation of the Retransmission Count option is optional and it is identified as elective. However, when it is present in a CoAP message and a CoAP endpoint processes it, it MUST be processed as described in this document. The Retransmission Count option MUST NOT occur more than once in a single message.

The value of the Retransmission Count option is a variable-size (0 to 1 byte) unsigned integer. The default value for the option is the number 0 and it is represented with an empty option value (a zero-length sequence of bytes). However, when a client intends to use Retransmit Count option, it MUST reserve space for it by limiting the request message size also when the value is empty in order to fit the full-sized option into retransmissions.

The Retransmission Count option can be present in both the request and response message. When the option is present in a request it

indicates the ordinal number of the transmission for the request message.

If the server supports (implements) the Retransmission Count option and the option is present in a request, the server MUST echo the option value in its Piggybacked Response unmodified. If the server replies with an Empty Acknowledgement the server MUST silently ignore the option and MUST NOT include it in a later separate response to that request.

When Piggybacked Response carrying the Retransmission Count option arrives, the client uses the option to match the response message to the corresponding transmission of the request. In order to measure a correct RTT, the client must store the timestamp for the original transmission of the request as well as the timestamp for each retransmission, if any, of the request. The resulting RTT sample is used for the RTO computation. If the client retransmitted the request without the option but the response includes the option, the client MUST silently ignore the option.

The original transmission of a request is indicated with the number 0, except when sending the first request to a new destination endpoint. The first original transmission of the request to a new endpoint carries the number 255 (0xFF) and is interpreted the same as an original transmission carrying the number 0. Retransmissions, if any, carry the ordinal number of the retransmission. Once the first Piggybacked Response from the new endpoint arrives the client learns whether or not the other endpoint implements the option. If the first response includes the echoed option, the client learns that the other endpoint supports the option and may continue including the option to each retransmitted request. From this point on the original transmissions of requests implicitly include the option number 0 and a zero-byte integer will be sent according to the CoAP uint-encoding rules. If the first Piggybacked Response does not include the option, the client SHOULD stop including the option into the requests to that endpoint.

When the Retransmission Count option is in use, the client bases the retransmission timeout for the normal RTO in the back off series as follows:

$\max(\text{RTO}, \text{Previous-RTT-Sample})$

Previous-RTT-Sample is the RTT sample acquired from the previous message exchange. If no RTT sample was available with the previous message exchange (e.g., the server replied with an Empty Acknowledgement), RTO computed earlier is used like in case the Retransmission Count option is not in use.

4.5. Alternatives for Exchanging Retransmission Count Information

An alternative way of exchanging the retransmission count information between a client and server is to encode it in the Token. The Token is a client-local identifier and a client solely decides how it generates the Token. Therefore, including a varying Token value to retransmissions of the same request is all possible as long as the client can use the Token to differentiate between requests and match a response to the corresponding request. The server is required to make no assumptions about the content or structure of a Token and always echo the Token unmodified in its response.

How exactly a client encodes the retransmission count into a Token is an implementation issue. Note that the original transmission of a request may carry a zero-length Token given that the rules for generating a Token as specified in RFC 7252 [RFC7252] are followed. This allows reducing the overhead of including the Token into the requests in such cases where Token could otherwise be omitted. However, similar to Retransmit Count option the maximum request message size MUST be limited to accommodate the Token with retransmit count into the retransmissions of the request.

5. Security Considerations

6. IANA Considerations

This memo includes no request to IANA.

7. References

7.1. Normative References

- [RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, DOI 10.17487/RFC2119, March 1997, <<https://www.rfc-editor.org/info/rfc2119>>.
- [RFC6298] Paxson, V., Allman, M., Chu, J., and M. Sargent, "Computing TCP's Retransmission Timer", RFC 6298, DOI 10.17487/RFC6298, June 2011, <<https://www.rfc-editor.org/info/rfc6298>>.
- [RFC7252] Shelby, Z., Hartke, K., and C. Bormann, "The Constrained Application Protocol (CoAP)", RFC 7252, DOI 10.17487/RFC7252, June 2014, <<https://www.rfc-editor.org/info/rfc7252>>.

7.2. Informative References

- [I-D.ietf-core-cocoa]
Bormann, C., Betzler, A., Gomez, C., and I. Demirkol,
"CoAP Simple Congestion Control/Advanced", draft-ietf-
core-cocoa-03 (work in progress), February 2018.
- [JRCK18a] Jarvinen, I., Raitahila, I., Cao, Z., and M. Kojo, "Is
CoAP Congestion Safe?", Applied Networking Research
Workshop (ANRW'18), July 2018.
- [JRCK18b] Jarvinen, I., Raitahila, I., Cao, Z., and M. Kojo, "FASOR
Retransmission Timeout and Congestion Control Mechanism
for CoAP?", Proceedings of IEEE Global Communications
Conference (Globecom 2018), to appear, December 2018.
- [KP87] Karn, P. and C. Partridge, "Improving Round-trip Time
Estimates in Reliable Transport Protocols", SIGCOMM'87
Proceedings of the ACM Workshop on Frontiers in Computer
Communications Technology, August 1987.
- [RFC0896] Nagle, J., "Congestion Control in IP/TCP Internetworks",
RFC 896, DOI 10.17487/RFC0896, January 1984,
<<https://www.rfc-editor.org/info/rfc896>>.

Appendix A. Pseudocode for Basic FASOR without Dithering

```
var state = NORMAL_RTO

rfc6298_init(var fastrto, 2 secs)

var slowrto
SLOWRTO_FACTOR = 1.5

var original_sendtime
var retransmit_count

/*
 * Sending Original Copy and Retransmitting 'req'
 */
send_request(req) {
    original_sendtime = time.now
    retransmit_count = 0

    arm_rto(calculate_rto())
    send(req)
}

rto_for(req) {
    retransmit_count += 1

    arm_rto(calculate_rto())
    send(req)
}

/*
 * ACK Processings
 */
ack() {
    sample = time.now - original_sendtime
    if (retransmit_count == 0)
        unambiguous_ack(sample)
    else
        ambiguous_ack(sample)
}

unambiguous_ack(sample) {
    k = 4 // RFC6298 default K = 4
    if (rfc6298_is_first_sample(fastrto))
        k = 1
    rfc6298_update(fastrto, k, sample) // Normal RFC6298 processing
    state = NORMAL_RTO
}
```

```
ambiguous_nextstate = {
    [NORMAL_RTO] = FAST_SLOW_FAST_RTO,
    [FAST_SLOW_FAST_RTO] = SLOW_FAST_RTO,
    [SLOW_FAST_RTO] = SLOW_FAST_RTO
}

ambiguous_ack(sample) {
    slowrto = sample * SLOWRTO_FACTOR
    state = ambiguous_nextstate[state]
}

/*
 * RTO Calculations
 */
calculate_rto() {
    return <state>_rtoseries()
}

normal_rtoseries() {
    switch (retransmit_count) {
        case 0: return fastrto_series_init()
        default: return fastrto_series_backoff()
    }
}

fastslowfast_rtoseries() {
    switch (retransmit_count) {
        case 0: return fastrto_series_init()
        case 1: return MAX(slowrto, 2*fastrto)
        default: return fastrto_series_backoff()
    }
}

slowfast_rtoseries() {
    switch (retransmit_count) {
        case 0: return slowrto
        case 1: return fastrto_series_init()
        default: return fastrto_series_backoff()
    }
}

var backoff_series_timer

fastrto_series_init() {
    backoff_series_timer = fastrto
    return backoff_series_timer
}
```

```
fastrto_series_backoff() {  
    backoff_series_timer *= 2  
    return backoff_series_timer  
}
```

Figure 2

Authors' Addresses

Ilpo Jarvinen
University of Helsinki
P.O. Box 68
FI-00014 UNIVERSITY OF HELSINKI
Finland

EMail: ilpo.jarvinen@cs.helsinki.fi

Markku Kojo
University of Helsinki
P.O. Box 68
FI-00014 UNIVERSITY OF HELSINKI
Finland

EMail: markku.kojo@cs.helsinki.fi

Iivo Raitahila
University of Helsinki
P.O. Box 68
FI-00014 UNIVERSITY OF HELSINKI
Finland

EMail: iivo.raitahila@helsinki.fi

Zhen Cao
Huawei
Beijing
China

EMail: zhencao.ietf@gmail.com