QUIC
Internet-Draft
Intended status: Standards Track
Expires: April 6, 2019

J. Iyengar, Ed.
Fastly
I. Swett, Ed.
Google

October 03, 2018

QUIC Loss Detection and Congestion Control
draft-ietf-quic-recovery-15

Abstract

This document describes loss detection and congestion control mechanisms for QUIC.

Note to Readers

Discussion of this draft takes place on the QUIC working group mailing list (quic@ietf.org), which is archived at https://mailarchive.ietf.org/arch/search/?email_list=quic [1].

Working Group information can be found at https://github.com/quicwg [2]; source code and issues list for this draft can be found at https://github.com/quicwg/base-drafts/labels/-recovery [3].

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 April 6, 2019.

Copyright Notice

Copyright (c) 2018 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 4
2. Conventions and Definitions 4
3. Design of the QUIC Transmission Machinery 4
3.1. Relevant Differences Between QUIC and TCP 5
3.1.1. Separate Packet Number Spaces 5
3.1.2. Monotonically Increasing Packet Numbers 5
3.1.3. No Reneging 6
3.1.4. More ACK Ranges 6
3.1.5. Explicit Correction For Delayed ACKs 6
4. Loss Detection 7
4.1. Computing the RTT estimate 7
4.2. Ack-based Detection 7
4.2.1. Fast Retransmit 7
4.2.2. Early Retransmit 8
4.3. Timer-based Detection 9
4.3.1. Crypto Handshake Timeout 9
4.3.2. Tail Loss Probe 10
4.3.3. Retransmission Timeout 11
4.4. Generating Acknowledgements 12
4.4.1. Crypto Handshake Data 12
4.4.2. ACK Ranges 13
4.4.3. Receiver Tracking of ACK Frames 13
4.5. Pseudocode 13
4.5.1. Constants of interest 13
4.5.2. Variables of interest 14
4.5.3. Initialization 15
4.5.4. On Sending a Packet 16
4.5.5. On Receiving an Acknowledgment 17
4.5.6. On Packet Acknowledgment 19
4.5.7. Setting the Loss Detection Timer 19
4.5.8. On Timeout 21
4.5.9. Detecting Lost Packets 22
4.6. Discussion 23
5. Congestion Control 23
5.1. Explicit Congestion Notification 24
5.2. Slow Start 24

5.3. Congestion Avoidance 24
5.4. Recovery Period 24
5.5. Tail Loss Probe 25
5.6. Retransmission Timeout 25
5.7. Pacing 25
5.8. Pseudocode 26
5.8.1. Constants of interest 26
5.8.2. Variables of interest 26
5.8.3. Initialization 27
5.8.4. On Packet Sent 27
5.8.5. On Packet Acknowledgement 27
5.8.6. On New Congestion Event 28
5.8.7. Process ECN Information 28
5.8.8. On Packets Lost 28
5.8.9. On Retransmission Timeout Verified 29
6. Security Considerations 29
6.1. Congestion Signals 29
6.2. Traffic Analysis 29
6.3. Misreporting ECN Markings 29
7. IANA Considerations 30
8. References 30
8.1. Normative References 30
8.2. Informative References 30
8.3. URIs 31
Appendix A. Change Log 32
A.1. Since draft-ietf-quic-recovery-14 32
A.2. Since draft-ietf-quic-recovery-13 32
A.3. Since draft-ietf-quic-recovery-12 32
A.4. Since draft-ietf-quic-recovery-11 32
A.5. Since draft-ietf-quic-recovery-10 32
A.6. Since draft-ietf-quic-recovery-09 33
A.7. Since draft-ietf-quic-recovery-08 33
A.8. Since draft-ietf-quic-recovery-07 33
A.9. Since draft-ietf-quic-recovery-06 33
A.10. Since draft-ietf-quic-recovery-05 33
A.11. Since draft-ietf-quic-recovery-04 33
A.12. Since draft-ietf-quic-recovery-03 33
A.13. Since draft-ietf-quic-recovery-02 33
A.14. Since draft-ietf-quic-recovery-01 34
A.15. Since draft-ietf-quic-recovery-00 34
A.16. Since draft-iyengar-quic-loss-recovery-01 34
Acknowledgments 34
Authors' Addresses 34

1. Introduction

QUIC is a new multiplexed and secure transport atop UDP. QUIC builds on decades of transport and security experience, and implements mechanisms that make it attractive as a modern general-purpose transport. The QUIC protocol is described in [QUIC-TRANSPORT].

QUIC implements the spirit of known TCP loss recovery mechanisms, described in RFCs, various Internet-drafts, and also those prevalent in the Linux TCP implementation. This document describes QUIC congestion control and loss recovery, and where applicable, attributes the TCP equivalent in RFCs, Internet-drafts, academic papers, and/or TCP implementations.

2. Conventions and Definitions

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "NOT RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in BCP 14 [RFC2119] [RFC8174] when, and only when, they appear in all capitals, as shown here.

Definitions of terms that are used in this document:

ACK-only: Any packet containing only an ACK frame.

In-flight: Packets are considered in-flight when they have been sent and neither acknowledged nor declared lost, and they are not ACK-only.

Retransmittable Frames: All frames besides ACK or PADDING are considered retransmittable.

Retransmittable Packets: Packets that contain retransmittable frames elicit an ACK from the receiver and are called retransmittable packets.

3. Design of the QUIC Transmission Machinery

All transmissions in QUIC are sent with a packet-level header, which indicates the encryption level and includes a packet sequence number (referred to below as a packet number). The encryption level indicates the packet number space, as described in [QUIC-TRANSPORT]. Packet numbers never repeat within a packet number space for the lifetime of a connection. Packet numbers monotonically increase within a space, preventing ambiguity.

This design obviates the need for disambiguating between transmissions and retransmissions and eliminates significant complexity from QUIC's interpretation of TCP loss detection mechanisms.

QUIC packets can contain multiple frames of different types. The recovery mechanisms ensure that data and frames that need reliable delivery are acknowledged or declared lost and sent in new packets as necessary. The types of frames contained in a packet affect recovery and congestion control logic:

- All packets are acknowledged, though packets that contain only ACK and PADDING frames are not acknowledged immediately.

- Long header packets that contain CRYPTO frames are critical to the performance of the QUIC handshake and use shorter timers for acknowledgement and retransmission.

- Packets that contain only ACK frames do not count toward congestion control limits and are not considered in-flight. Note that this means PADDING frames cause packets to contribute toward bytes in flight without directly causing an acknowledgment to be sent.

3.1. Relevant Differences Between QUIC and TCP

Readers familiar with TCP's loss detection and congestion control will find algorithms here that parallel well-known TCP ones. Protocol differences between QUIC and TCP however contribute to algorithmic differences. We briefly describe these protocol differences below.

3.1.1. Separate Packet Number Spaces

QUIC uses separate packet number spaces for each encryption level, except 0-RTT and all generations of 1-RTT keys use the same packet number space. Separate packet number spaces ensures acknowledgement of packets sent with one level of encryption will not cause spurious retransmission of packets sent with a different encryption level. Congestion control and RTT measurement are unified across packet number spaces.

3.1.2. Monotonically Increasing Packet Numbers

TCP conflates transmission sequence number at the sender with delivery sequence number at the receiver, which results in retransmissions of the same data carrying the same sequence number, and consequently to problems caused by "retransmission ambiguity".

QUIC separates the two: QUIC uses a packet number for transmissions, and any application data is sent in one or more streams, with delivery order determined by stream offsets encoded within STREAM frames.

QUIC's packet number is strictly increasing, and directly encodes transmission order. A higher QUIC packet number signifies that the packet was sent later, and a lower QUIC packet number signifies that the packet was sent earlier. When a packet containing frames is deemed lost, QUIC rebundles necessary frames in a new packet with a new packet number, removing ambiguity about which packet is acknowledged when an ACK is received. Consequently, more accurate RTT measurements can be made, spurious retransmissions are trivially detected, and mechanisms such as Fast Retransmit can be applied universally, based only on packet number.

This design point significantly simplifies loss detection mechanisms for QUIC. Most TCP mechanisms implicitly attempt to infer transmission ordering based on TCP sequence numbers - a non-trivial task, especially when TCP timestamps are not available.

3.1.3. No Reneging

QUIC ACKs contain information that is similar to TCP SACK, but QUIC does not allow any acked packet to be reneged, greatly simplifying implementations on both sides and reducing memory pressure on the sender.

3.1.4. More ACK Ranges

QUIC supports many ACK ranges, opposed to TCP's 3 SACK ranges. In high loss environments, this speeds recovery, reduces spurious retransmits, and ensures forward progress without relying on timeouts.

3.1.5. Explicit Correction For Delayed ACKs

QUIC ACKs explicitly encode the delay incurred at the receiver between when a packet is received and when the corresponding ACK is sent. This allows the receiver of the ACK to adjust for receiver delays, specifically the delayed ack timer, when estimating the path RTT. This mechanism also allows a receiver to measure and report the delay from when a packet was received by the OS kernel, which is useful in receivers which may incur delays such as context-switch latency before a userspace QUIC receiver processes a received packet.

4. Loss Detection

QUIC senders use both ack information and timeouts to detect lost packets, and this section provides a description of these algorithms. Estimating the network round-trip time (RTT) is critical to these algorithms and is described first.

4.1. Computing the RTT estimate

RTT is calculated when an ACK frame arrives by computing the difference between the current time and the time the largest newly acked packet was sent. If no packets are newly acknowledged, RTT cannot be calculated. When RTT is calculated, the ack delay field from the ACK frame SHOULD be subtracted from the RTT as long as the result is larger than the Min RTT. If the result is smaller than the min_rtt, the RTT should be used, but the ack delay field should be ignored.

Like TCP, QUIC calculates both smoothed RTT and RTT variance similar to those specified in [RFC6298].

Min RTT is the minimum RTT measured over the connection, prior to adjusting by ack delay. Ignoring ack delay for min RTT prevents intentional or unintentional underestimation of min RTT, which in turn prevents underestimating smoothed RTT.

4.2. Ack-based Detection

Ack-based loss detection implements the spirit of TCP's Fast Retransmit [RFC5681], Early Retransmit [RFC5827], FACK, and SACK loss recovery [RFC6675]. This section provides an overview of how these algorithms are implemented in QUIC.

4.2.1. Fast Retransmit

An unacknowledged packet is marked as lost when an acknowledgment is received for a packet that was sent a threshold number of packets (kReorderingThreshold) and/or a threshold amount of time after the unacknowledged packet. Receipt of the acknowledgement indicates that a later packet was received, while the reordering threshold provides some tolerance for reordering of packets in the network.

The RECOMMENDED initial value for kReorderingThreshold is 3, based on TCP loss recovery [RFC5681] [RFC6675]. Some networks may exhibit higher degrees of reordering, causing a sender to detect spurious losses. Spuriously declaring packets lost leads to unnecessary retransmissions and may result in degraded performance due to the actions of the congestion controller upon detecting loss.

Implementers MAY use algorithms developed for TCP, such as TCP-NCR [RFC4653], to improve QUIC's reordering resilience.

QUIC implementations can use time-based loss detection to handle reordering based on time elapsed since the packet was sent. This may be used either as a replacement for a packet reordering threshold or in addition to it. The RECOMMENDED time threshold, expressed as a fraction of the round-trip time (kTimeReorderingFraction), is 1/8.

4.2.2. Early Retransmit

Unacknowledged packets close to the tail may have fewer than kReorderingThreshold retransmittable packets sent after them. Loss of such packets cannot be detected via Fast Retransmit. To enable ack-based loss detection of such packets, receipt of an acknowledgment for the last outstanding retransmittable packet triggers the Early Retransmit process, as follows.

If there are unacknowledged in-flight packets still pending, they should be marked as lost. To compensate for the reduced reordering resilience, the sender SHOULD set a timer for a small period of time. If the unacknowledged in-flight packets are not acknowledged during this time, then these packets MUST be marked as lost.

An endpoint SHOULD set the timer such that a packet is marked as lost no earlier than 1.125 * max(SRTT, latest_RTT) since when it was sent.

Using max(SRTT, latest_RTT) protects from the two following cases:

- the latest RTT sample is lower than the SRTT, perhaps due to reordering where packet whose ack triggered the Early Retransit process encountered a shorter path;

- the latest RTT sample is higher than the SRTT, perhaps due to a sustained increase in the actual RTT, but the smoothed SRTT has not yet caught up.

The 1.125 multiplier increases reordering resilience. Implementers MAY experiment with using other multipliers, bearing in mind that a lower multiplier reduces reordering resilience and increases spurious retransmissions, and a higher multiplier increases loss recovery delay.

This mechanism is based on Early Retransmit for TCP [RFC5827]. However, [RFC5827] does not include the timer described above. Early Retransmit is prone to spurious retransmissions due to its reduced reordering resilence without the timer. This observation led Linux

TCP implementers to implement a timer for TCP as well, and this document incorporates this advancement.

4.3. Timer-based Detection

Timer-based loss detection recovers from losses that cannot be handled by ack-based loss detection. It uses a single timer which switches between a handshake retransmission timer, a Tail Loss Probe timer and Retransmission Timeout mechanisms.

4.3.1. Crypto Handshake Timeout

Data in CRYPTO frames is critical to QUIC transport and crypto negotiation, so a more aggressive timeout is used to retransmit it. Below, the term "handshake packet" is used to refer to packets containing CRYPTO frames, not packets with the specific long header packet type Handshake.

The initial handshake timeout SHOULD be set to twice the initial RTT.

At the beginning, there are no prior RTT samples within a connection. Resumed connections over the same network SHOULD use the previous connection's final smoothed RTT value as the resumed connection's initial RTT.

If no previous RTT is available, or if the network changes, the initial RTT SHOULD be set to 100ms.

When CRYPTO frames are sent, the sender SHOULD set a timer for the handshake timeout period. Upon timeout, the sender MUST retransmit all unacknowledged CRYPTO data by calling RetransmitAllUnackedHandshakeData(). On each consecutive expiration of the handshake timer without receiving an acknowledgement for a new packet, the sender SHOULD double the handshake timeout and set a timer for this period.

When CRYPTO frames are outstanding, the TLP and RTO timers are not active unless the CRYPTO frames were sent at 1-RTT encryption.

When an acknowledgement is received for a handshake packet, the new RTT is computed and the timer SHOULD be set for twice the newly computed smoothed RTT.

4.3.1.1. Retry and Version Negotiation

A Retry or Version Negotiation packet causes a client to send another Initial packet, effectively restarting the connection process.

Either packet indicates that the Initial was received but not processed. Neither packet can be treated as an acknowledgment for the Initial, but they MAY be used to improve the RTT estimate.

4.3.2. Tail Loss Probe

The algorithm described in this section is an adaptation of the Tail Loss Probe algorithm proposed for TCP [TLP].

A packet sent at the tail is particularly vulnerable to slow loss detection, since acks of subsequent packets are needed to trigger ack-based detection. To ameliorate this weakness of tail packets, the sender schedules a timer when the last retransmittable packet before quiescence is transmitted. Upon timeout, a Tail Loss Probe (TLP) packet is sent to evoke an acknowledgement from the receiver.

The timer duration, or Probe Timeout (PTO), is set based on the following conditions:

- PTO SHOULD be scheduled for max(1.5*SRTT+MaxAckDelay, kMinTLPTimeout)

- If RTO (Section 4.3.3) is earlier, schedule a TLP in its place. That is, PTO SHOULD be scheduled for min(RTO, PTO).

QUIC includes MaxAckDelay in all probe timeouts, because it assumes the ack delay may come into play, regardless of the number of packets outstanding. TCP's TLP assumes if at least 2 packets are outstanding, acks will not be delayed.

A PTO value of at least 1.5*SRTT ensures that the ACK is overdue. The 1.5 is based on [TLP], but implementations MAY experiment with other constants.

To reduce latency, it is RECOMMENDED that the sender set and allow the TLP timer to fire twice before setting an RTO timer. In other words, when the TLP timer expires the first time, a TLP packet is sent, and it is RECOMMENDED that the TLP timer be scheduled for a second time. When the TLP timer expires the second time, a second TLP packet is sent, and an RTO timer SHOULD be scheduled Section 4.3.3.

A TLP packet SHOULD carry new data when possible. If new data is unavailable or new data cannot be sent due to flow control, a TLP packet MAY retransmit unacknowledged data to potentially reduce recovery time. Since a TLP timer is used to send a probe into the network prior to establishing any packet loss, prior unacknowledged packets SHOULD NOT be marked as lost when a TLP timer expires.

A sender may not know that a packet being sent is a tail packet. Consequently, a sender may have to arm or adjust the TLP timer on every sent retransmittable packet.

4.3.3. Retransmission Timeout

A Retransmission Timeout (RTO) timer is the final backstop for loss detection. The algorithm used in QUIC is based on the RTO algorithm for TCP [RFC5681] and is additionally resilient to spurious RTO events [RFC5682].

When the last TLP packet is sent, a timer is set for the RTO period. When this timer expires, the sender sends two packets, to evoke acknowledgements from the receiver, and restarts the RTO timer.

Similar to TCP [RFC6298], the RTO period is set based on the following conditions:

- When the final TLP packet is sent, the RTO period is set to max(SRTT + 4*RTTVAR + MaxAckDelay, kMinRTOTimeout)

- When an RTO timer expires, the RTO period is doubled.

The sender typically has incurred a high latency penalty by the time an RTO timer expires, and this penalty increases exponentially in subsequent consecutive RTO events. Sending a single packet on an RTO event therefore makes the connection very sensitive to single packet loss. Sending two packets instead of one significantly increases resilience to packet drop in both directions, thus reducing the probability of consecutive RTO events.

QUIC's RTO algorithm differs from TCP in that the firing of an RTO timer is not considered a strong enough signal of packet loss, so does not result in an immediate change to congestion window or recovery state. An RTO timer expires only when there's a prolonged period of network silence, which could be caused by a change in the underlying network RTT.

QUIC also diverges from TCP by including MaxAckDelay in the RTO period. Since QUIC corrects for this delay in its SRTT and RTTVAR computations, it is necessary to add this delay explicitly in the TLP and RTO computation.

When an acknowledgment is received for a packet sent on an RTO event, any unacknowledged packets with lower packet numbers than those acknowledged MUST be marked as lost. If an acknowledgement for a packet sent on an RTO is received at the same time packets sent prior

to the first RTO are acknowledged, the RTO is considered spurious and standard loss detection rules apply.

A packet sent when an RTO timer expires MAY carry new data if available or unacknowledged data to potentially reduce recovery time. Since this packet is sent as a probe into the network prior to establishing any packet loss, prior unacknowledged packets SHOULD NOT be marked as lost.

A packet sent on an RTO timer MUST NOT be blocked by the sender's congestion controller. A sender MUST however count these bytes as additional bytes in flight, since this packet adds network load without establishing packet loss.

4.4. Generating Acknowledgements

QUIC SHOULD delay sending acknowledgements in response to packets, but MUST NOT excessively delay acknowledgements of packets containing frames other than ACK. Specifically, implementations MUST attempt to enforce a maximum ack delay to avoid causing the peer spurious timeouts. The maximum ack delay is communicated in the "max_ack_delay" transport parameter and the default value is 25ms.

An acknowledgement SHOULD be sent immediately upon receipt of a second packet but the delay SHOULD NOT exceed the maximum ack delay. QUIC recovery algorithms do not assume the peer generates an acknowledgement immediately when receiving a second full-packet.

Out-of-order packets SHOULD be acknowledged more quickly, in order to accelerate loss recovery. The receiver SHOULD send an immediate ACK when it receives a new packet which is not one greater than the largest received packet number.

Similarly, packets marked with the ECN Congestion Experienced (CE) codepoint in the IP header SHOULD be acknowledged immediately, to reduce the peer's response time to congestion events.

As an optimization, a receiver MAY process multiple packets before sending any ACK frames in response. In this case they can determine whether an immediate or delayed acknowledgement should be generated after processing incoming packets.

4.4.1. Crypto Handshake Data

In order to quickly complete the handshake and avoid spurious retransmissions due to handshake timeouts, handshake packets SHOULD use a very short ack delay, such as 1ms. ACK frames MAY be sent

immediately when the crypto stack indicates all data for that encryption level has been received.

4.4.2. ACK Ranges

When an ACK frame is sent, one or more ranges of acknowledged packets are included. Including older packets reduces the chance of spurious retransmits caused by losing previously sent ACK frames, at the cost of larger ACK frames.

ACK frames SHOULD always acknowledge the most recently received packets, and the more out-of-order the packets are, the more important it is to send an updated ACK frame quickly, to prevent the peer from declaring a packet as lost and spuriously retransmitting the frames it contains.

Below is one recommended approach for determining what packets to include in an ACK frame.

4.4.3. Receiver Tracking of ACK Frames

When a packet containing an ACK frame is sent, the largest acknowledged in that frame may be saved. When a packet containing an ACK frame is acknowledged, the receiver can stop acknowledging packets less than or equal to the largest acknowledged in the sent ACK frame.

In cases without ACK frame loss, this algorithm allows for a minimum of 1 RTT of reordering. In cases with ACK frame loss, this approach does not guarantee that every acknowledgement is seen by the sender before it is no longer included in the ACK frame. Packets could be received out of order and all subsequent ACK frames containing them could be lost. In this case, the loss recovery algorithm may cause spurious retransmits, but the sender will continue making forward progress.

4.5. Pseudocode

4.5.1. Constants of interest

Constants used in loss recovery are based on a combination of RFCs, papers, and common practice. Some may need to be changed or negotiated in order to better suit a variety of environments.

kMaxTLPs: Maximum number of tail loss probes before an RTO expires. The RECOMMENDED value is 2.

kReorderingThreshold: Maximum reordering in packet number space before FACK style loss detection considers a packet lost. The RECOMMENDED value is 3.

kTimeReorderingFraction: Maximum reordering in time space before time based loss detection considers a packet lost. In fraction of an RTT. The RECOMMENDED value is 1/8.

kUsingTimeLossDetection: Whether time based loss detection is in use. If false, uses FACK style loss detection. The RECOMMENDED value is false.

kMinTLPTimeout: Minimum time in the future a tail loss probe timer may be set for. The RECOMMENDED value is 10ms.

kMinRTOTimeout: Minimum time in the future an RTO timer may be set for. The RECOMMENDED value is 200ms.

kDelayedAckTimeout: The length of the peer's delayed ack timer. The RECOMMENDED value is 25ms.

kInitialRtt: The RTT used before an RTT sample is taken. The RECOMMENDED value is 100ms.

4.5.2. Variables of interest

Variables required to implement the congestion control mechanisms are described in this section.

loss_detection_timer: Multi-modal timer used for loss detection.

handshake_count: The number of times all unacknowledged handshake data has been retransmitted without receiving an ack.

tlp_count: The number of times a tail loss probe has been sent without receiving an ack.

rto_count: The number of times an RTO has been sent without receiving an ack.

largest_sent_before_rto: The last packet number sent prior to the first retransmission timeout.

time_of_last_sent_retransmittable_packet: The time the most recent retransmittable packet was sent.

time_of_last_sent_handshake_packet: The time the most recent packet containing a CRYPTO frame was sent.

largest_sent_packet: The packet number of the most recently sent packet.

largest_acked_packet: The largest packet number acknowledged in an ACK frame.

latest_rtt: The most recent RTT measurement made when receiving an ack for a previously unacked packet.

smoothed_rtt: The smoothed RTT of the connection, computed as described in [RFC6298]

rttvar: The RTT variance, computed as described in [RFC6298]

min_rtt: The minimum RTT seen in the connection, ignoring ack delay.

max_ack_delay: The maximum amount of time by which the receiver intends to delay acknowledgments, in milliseconds. The actual ack_delay in a received ACK frame may be larger due to late timers, reordering, or lost ACKs.

reordering_threshold: The largest packet number gap between the largest acknowledged retransmittable packet and an unacknowledged retransmittable packet before it is declared lost.

time_reordering_fraction: The reordering window as a fraction of max(smoothed_rtt, latest_rtt).

loss_time: The time at which the next packet will be considered lost based on early transmit or exceeding the reordering window in time.

sent_packets: An association of packet numbers to information about them, including a number field indicating the packet number, a time field indicating the time a packet was sent, a boolean indicating whether the packet is ack-only, a boolean indicating whether it counts towards bytes in flight, and a bytes field indicating the packet's size. sent_packets is ordered by packet number, and packets remain in sent_packets until acknowledged or lost. A sent_packets data structure is maintained per packet number space, and ACK processing only applies to a single space.

4.5.3. Initialization

At the beginning of the connection, initialize the loss detection variables as follows:

```
    loss_detection_timer.reset()
    handshake_count = 0
    tlp_count = 0
    rto_count = 0
    if (kUsingTimeLossDetection)
      reordering_threshold = infinite
      time_reordering_fraction = kTimeReorderingFraction
    else:
      reordering_threshold = kReorderingThreshold
      time_reordering_fraction = infinite
    loss_time = 0
    smoothed_rtt = 0
    rttvar = 0
    min_rtt = infinite
    largest_sent_before_rto = 0
    time_of_last_sent_retransmittable_packet = 0
    time_of_last_sent_handshake_packet = 0
    largest_sent_packet = 0
```

4.5.4. On Sending a Packet

After any packet is sent, be it a new transmission or a rebundled transmission, the following OnPacketSent function is called. The parameters to OnPacketSent are as follows:

- packet_number: The packet number of the sent packet.

- ack_only: A boolean that indicates whether a packet contains only ACK or PADDING frame(s). If true, it is still expected an ack will be received for this packet, but it is not retransmittable.

- in_flight: A boolean that indicates whether the packet counts towards bytes in flight.

- is_handshake_packet: A boolean that indicates whether the packet contains cryptographic handshake messages critical to the completion of the QUIC handshake. In this version of QUIC, this includes any packet with the long header that includes a CRYPTO frame.

- sent_bytes: The number of bytes sent in the packet, not including UDP or IP overhead, but including QUIC framing overhead.

Pseudocode for OnPacketSent follows:

```
 OnPacketSent(packet_number, ack_only, in_flight,
              is_handshake_packet, sent_bytes):
   largest_sent_packet = packet_number
   sent_packets[packet_number].packet_number = packet_number
   sent_packets[packet_number].time = now
   sent_packets[packet_number].ack_only = ack_only
   sent_packets[packet_number].in_flight = in_flight
   if !ack_only:
     if is_handshake_packet:
       time_of_last_sent_handshake_packet = now
     time_of_last_sent_retransmittable_packet = now
     OnPacketSentCC(sent_bytes)
     sent_packets[packet_number].bytes = sent_bytes
     SetLossDetectionTimer()
```

4.5.5. On Receiving an Acknowledgment

When an ACK frame is received, it may newly acknowledge any number of packets.

Pseudocode for OnAckReceived and UpdateRtt follow:

```
OnAckReceived(ack):
  largest_acked_packet = ack.largest_acked
  // If the largest acknowledged is newly acked,
  // update the RTT.
  if (sent_packets[ack.largest_acked]):
    latest_rtt = now - sent_packets[ack.largest_acked].time
    UpdateRtt(latest_rtt, ack.ack_delay)

  // Find all newly acked packets in this ACK frame
  newly_acked_packets = DetermineNewlyAckedPackets(ack)
  for acked_packet in newly_acked_packets:
    OnPacketAcked(acked_packet.packet_number)

  if !newly_acked_packets.empty():
    // Find the smallest newly acknowledged packet
    smallest_newly_acked =
      FindSmallestNewlyAcked(newly_acked_packets)
    // If any packets sent prior to RTO were acked, then the
    // RTO was spurious. Otherwise, inform congestion control.
    if (rto_count > 0 &&
          smallest_newly_acked > largest_sent_before_rto):
      OnRetransmissionTimeoutVerified(smallest_newly_acked)
    handshake_count = 0
    tlp_count = 0
    rto_count = 0

  DetectLostPackets(ack.largest_acked_packet)
  SetLossDetectionTimer()

  // Process ECN information if present.
  if (ACK frame contains ECN information):
     ProcessECN(ack)


UpdateRtt(latest_rtt, ack_delay):
  // min_rtt ignores ack delay.
  min_rtt = min(min_rtt, latest_rtt)
  // Adjust for ack delay if it's plausible.
  if (latest_rtt - min_rtt > ack_delay):
    latest_rtt -= ack_delay
  // Based on {{RFC6298}}.
  if (smoothed_rtt == 0):
    smoothed_rtt = latest_rtt
    rttvar = latest_rtt / 2
  else:
    rttvar_sample = abs(smoothed_rtt - latest_rtt)
    rttvar = 3/4 * rttvar + 1/4 * rttvar_sample
    smoothed_rtt = 7/8 * smoothed_rtt + 1/8 * latest_rtt
```

4.5.6. On Packet Acknowledgment

When a packet is acked for the first time, the following OnPacketAcked function is called. Note that a single ACK frame may newly acknowledge several packets. OnPacketAcked must be called once for each of these newly acked packets.

OnPacketAcked takes one parameter, acked_packet, which is the struct of the newly acked packet.

If this is the first acknowledgement following RTO, check if the smallest newly acknowledged packet is one sent by the RTO, and if so, inform congestion control of a verified RTO, similar to F-RTO [RFC5682].

Pseudocode for OnPacketAcked follows:

```
OnPacketAcked(acked_packet):
  if (!acked_packet.is_ack_only):
    OnPacketAckedCC(acked_packet)
  sent_packets.remove(acked_packet.packet_number)
```

4.5.7. Setting the Loss Detection Timer

QUIC loss detection uses a single timer for all timer-based loss detection. The duration of the timer is based on the timer's mode, which is set in the packet and timer events further below. The function SetLossDetectionTimer defined below shows how the single timer is set.

4.5.7.1. Handshake Timer

When a connection has unacknowledged handshake data, the handshake timer is set and when it expires, all unacknowledgedd handshake data is retransmitted.

When stateless rejects are in use, the connection is considered immediately closed once a reject is sent, so no timer is set to retransmit the reject.

Version negotiation packets are always stateless, and MUST be sent once per handshake packet that uses an unsupported QUIC version, and MAY be sent in response to 0-RTT packets.

4.5.7.2. Tail Loss Probe and Retransmission Timer

Tail loss probes [TLP] and retransmission timeouts [RFC6298] are timer based mechanisms to recover from cases when there are outstanding retransmittable packets, but an acknowledgement has not been received in a timely manner.

The TLP and RTO timers are armed when there is no unacknowledged handshake data. The TLP timer is set until the max number of TLP packets have been sent, and then the RTO timer is set.

4.5.7.3. Early Retransmit Timer

Early retransmit [RFC5827] is implemented with a 1/4 RTT timer. It is part of QUIC's time based loss detection, but is always enabled, even when only packet reordering loss detection is enabled.

4.5.7.4. Pseudocode

Pseudocode for SetLossDetectionTimer follows:

```
    SetLossDetectionTimer():
       // Don't arm timer if there are no retransmittable packets
       // in flight.
       if (bytes_in_flight == 0):
         loss_detection_timer.cancel()
         return

       if (handshake packets are outstanding):
         // Handshake retransmission timer.
         if (smoothed_rtt == 0):
           timeout = 2 * kInitialRtt
         else:
           timeout = 2 * smoothed_rtt
         timeout = max(timeout, kMinTLPTimeout)
         timeout = timeout * (2 ^ handshake_count)
         loss_detection_timer.set(
           time_of_last_sent_handshake_packet + timeout)
         return;
       else if (loss_time != 0):
         // Early retransmit timer or time loss detection.
         timeout = loss_time -
           time_of_last_sent_retransmittable_packet
       else:
         // RTO or TLP timer
         // Calculate RTO duration
         timeout =
           smoothed_rtt + 4 * rttvar + max_ack_delay
         timeout = max(timeout, kMinRTOTimeout)
         timeout = timeout * (2 ^ rto_count)
         if (tlp_count < kMaxTLPs):
           // Tail Loss Probe
           tlp_timeout = max(1.5 * smoothed_rtt
                             + max_ack_delay, kMinTLPTimeout)
           timeout = min(tlp_timeout, timeout)

       loss_detection_timer.set(
         time_of_last_sent_retransmittable_packet + timeout)
```

4.5.8. On Timeout

QUIC uses one loss recovery timer, which when set, can be in one of several modes. When the timer expires, the mode determines the action to be performed.

Pseudocode for OnLossDetectionTimeout follows:

```
OnLossDetectionTimeout():
  if (handshake packets are outstanding):
    // Handshake timeout.
    RetransmitAllUnackedHandshakeData()
    handshake_count++
  else if (loss_time != 0):
    // Early retransmit or Time Loss Detection
    DetectLostPackets(largest_acked_packet)
  else if (tlp_count < kMaxTLPs):
    // Tail Loss Probe.
    SendOnePacket()
    tlp_count++
  else:
    // RTO.
    if (rto_count == 0)
      largest_sent_before_rto = largest_sent_packet
    SendTwoPackets()
    rto_count++

  SetLossDetectionTimer()
```

4.5.9. Detecting Lost Packets

Packets in QUIC are only considered lost once a larger packet number in the same packet number space is acknowledged. DetectLostPackets is called every time an ack is received and operates on the sent_packets for that packet number space. If the loss detection timer expires and the loss_time is set, the previous largest acked packet is supplied.

4.5.9.1. Pseudocode

DetectLostPackets takes one parameter, acked, which is the largest acked packet.

Pseudocode for DetectLostPackets follows:

```
DetectLostPackets(largest_acked):
  loss_time = 0
  lost_packets = {}
  delay_until_lost = infinite
  if (kUsingTimeLossDetection):
    delay_until_lost =
      (1 + time_reordering_fraction) *
          max(latest_rtt, smoothed_rtt)
  else if (largest_acked.packet_number == largest_sent_packet):
    // Early retransmit timer.
    delay_until_lost = 9/8 * max(latest_rtt, smoothed_rtt)
  foreach (unacked < largest_acked.packet_number):
    time_since_sent = now() - unacked.time_sent
    delta = largest_acked.packet_number - unacked.packet_number
    if (time_since_sent > delay_until_lost ||
        delta > reordering_threshold):
      sent_packets.remove(unacked.packet_number)
      if (!unacked.is_ack_only):
        lost_packets.insert(unacked)
    else if (loss_time == 0 && delay_until_lost != infinite):
      loss_time = now() + delay_until_lost - time_since_sent

  // Inform the congestion controller of lost packets and
  // lets it decide whether to retransmit immediately.
  if (!lost_packets.empty()):
    OnPacketsLost(lost_packets)
```

4.6. Discussion

The majority of constants were derived from best common practices among widely deployed TCP implementations on the internet. Exceptions follow.

A shorter delayed ack time of 25ms was chosen because longer delayed acks can delay loss recovery and for the small number of connections where less than packet per 25ms is delivered, acking every packet is beneficial to congestion control and loss recovery.

The default initial RTT of 100ms was chosen because it is slightly higher than both the median and mean min_rtt typically observed on the public internet.

5. Congestion Control

QUIC's congestion control is based on TCP NewReno [RFC6582]. NewReno is a congestion window based congestion control. QUIC specifies the congestion window in bytes rather than packets due to finer control and the ease of appropriate byte counting [RFC3465].

QUIC hosts MUST NOT send packets if they would increase bytes_in_flight (defined in Section 5.8.2) beyond the available congestion window, unless the packet is a probe packet sent after the TLP or RTO timer expires, as described in Section 4.3.2 and Section 4.3.3.

Implementations MAY use other congestion control algorithms, and endpoints MAY use different algorithms from one another. The signals QUIC provides for congestion control are generic and are designed to support different algorithms.

5.1. Explicit Congestion Notification

If a path has been verified to support ECN, QUIC treats a Congestion Experienced codepoint in the IP header as a signal of congestion. This document specifies an endpoint's response when its peer receives packets with the Congestion Experienced codepoint. As discussed in [RFC8311], endpoints are permitted to experiment with other response functions.

5.2. Slow Start

QUIC begins every connection in slow start and exits slow start upon loss or upon increase in the ECN-CE counter. QUIC re-enters slow start anytime the congestion window is less than ssthresh, which typically only occurs after an RTO. While in slow start, QUIC increases the congestion window by the number of bytes acknowledged when each ack is processed.

5.3. Congestion Avoidance

Slow start exits to congestion avoidance. Congestion avoidance in NewReno uses an additive increase multiplicative decrease (AIMD) approach that increases the congestion window by one maximum packet size per congestion window acknowledged. When a loss is detected, NewReno halves the congestion window and sets the slow start threshold to the new congestion window.

5.4. Recovery Period

Recovery is a period of time beginning with detection of a lost packet or an increase in the ECN-CE counter. Because QUIC retransmits stream data and control frames, not packets, it defines the end of recovery as a packet sent after the start of recovery being acknowledged. This is slightly different from TCP's definition of recovery, which ends when the lost packet that started recovery is acknowledged.

The recovery period limits congestion window reduction to once per round trip. During recovery, the congestion window remains unchanged irrespective of new losses or increases in the ECN-CE counter.

5.5. Tail Loss Probe

A TLP packet MUST NOT be blocked by the sender's congestion controller. The sender MUST however count these bytes as additional bytes-in-flight, since a TLP adds network load without establishing packet loss.

Acknowledgement or loss of tail loss probes are treated like any other packet.

5.6. Retransmission Timeout

When retransmissions are sent due to a retransmission timeout timer, no change is made to the congestion window until the next acknowledgement arrives. The retransmission timeout is considered spurious when this acknowledgement acknowledges packets sent prior to the first retransmission timeout. The retransmission timeout is considered valid when this acknowledgement acknowledges no packets sent prior to the first retransmission timeout. In this case, the congestion window MUST be reduced to the minimum congestion window and slow start is re-entered.

5.7. Pacing

This document does not specify a pacer, but it is RECOMMENDED that a sender pace sending of all in-flight packets based on input from the congestion controller. For example, a pacer might distribute the congestion window over the SRTT when used with a window-based controller, and a pacer might use the rate estimate of a rate-based controller.

An implementation should take care to architect its congestion controller to work well with a pacer. For instance, a pacer might wrap the congestion controller and control the availability of the congestion window, or a pacer might pace out packets handed to it by the congestion controller. Timely delivery of ACK frames is important for efficient loss recovery. Packets containing only ACK frames should therefore not be paced, to avoid delaying their delivery to the peer.

As an example of a well-known and publicly available implementation of a flow pacer, implementers are referred to the Fair Queue packet scheduler (fq qdisc) in Linux (3.11 onwards).

5.8. Pseudocode

5.8.1. Constants of interest

Constants used in congestion control are based on a combination of RFCs, papers, and common practice. Some may need to be changed or negotiated in order to better suit a variety of environments.

kMaxDatagramSize: The sender's maximum payload size. Does not include UDP or IP overhead. The max packet size is used for calculating initial and minimum congestion windows. The RECOMMENDED value is 1200 bytes.

kInitialWindow: Default limit on the initial amount of outstanding data in bytes. Taken from [RFC6928]. The RECOMMENDED value is the minimum of 10 * kMaxDatagramSize and max(2* kMaxDatagramSize, 14600)).

kMinimumWindow: Minimum congestion window in bytes. The RECOMMENDED value is 2 * kMaxDatagramSize.

kLossReductionFactor: Reduction in congestion window when a new loss event is detected. The RECOMMENDED value is 0.5.

5.8.2. Variables of interest

Variables required to implement the congestion control mechanisms are described in this section.

ecn_ce_counter: The highest value reported for the ECN-CE counter by the peer in an ACK frame. This variable is used to detect increases in the reported ECN-CE counter.

bytes_in_flight: The sum of the size in bytes of all sent packets that contain at least one retransmittable or PADDING frame, and have not been acked or declared lost. The size does not include IP or UDP overhead, but does include the QUIC header and AEAD overhead. Packets only containing ACK frames do not count towards bytes_in_flight to ensure congestion control does not impede congestion feedback.

congestion_window: Maximum number of bytes-in-flight that may be sent.

end_of_recovery: The largest packet number sent when QUIC detects a loss. When a larger packet is acknowledged, QUIC exits recovery.

ssthresh: Slow start threshold in bytes. When the congestion window is below ssthresh, the mode is slow start and the window grows by the number of bytes acknowledged.

5.8.3. Initialization

At the beginning of the connection, initialize the congestion control variables as follows:

```
congestion_window = kInitialWindow
bytes_in_flight = 0
end_of_recovery = 0
ssthresh = infinite
ecn_ce_counter = 0
```

5.8.4. On Packet Sent

Whenever a packet is sent, and it contains non-ACK frames, the packet increases bytes_in_flight.

```
OnPacketSentCC(bytes_sent):
  bytes_in_flight += bytes_sent
```

5.8.5. On Packet Acknowledgement

Invoked from loss detection's OnPacketAcked and is supplied with acked_packet from sent_packets.

```
InRecovery(packet_number):
  return packet_number <= end_of_recovery

OnPacketAckedCC(acked_packet):
  // Remove from bytes_in_flight.
  bytes_in_flight -= acked_packet.bytes
  if (InRecovery(acked_packet.packet_number)):
    // Do not increase congestion window in recovery period.
    return
  if (congestion_window < ssthresh):
    // Slow start.
    congestion_window += acked_packet.bytes
  else:
    // Congestion avoidance.
    congestion_window += kMaxDatagramSize * acked_packet.bytes
        / congestion_window
```

5.8.6. On New Congestion Event

Invoked from ProcessECN and OnPacketsLost when a new congestion event is detected. Starts a new recovery period and reduces the congestion window.

```
CongestionEvent(packet_number):
  // Start a new congestion event if packet_number
  // is larger than the end of the previous recovery epoch.
  if (!InRecovery(packet_number)):
    end_of_recovery = largest_sent_packet
    congestion_window *= kLossReductionFactor
    congestion_window = max(congestion_window, kMinimumWindow)
    ssthresh = congestion_window
```

5.8.7. Process ECN Information

Invoked when an ACK frame with an ECN section is received from the peer.

```
ProcessECN(ack):
  // If the ECN-CE counter reported by the peer has increased,
  // this could be a new congestion event.
  if (ack.ce_counter > ecn_ce_counter):
    ecn_ce_counter = ack.ce_counter
    // Start a new congestion event if the last acknowledged
    // packet is past the end of the previous recovery epoch.
    CongestionEvent(ack.largest_acked_packet)
```

5.8.8. On Packets Lost

Invoked by loss detection from DetectLostPackets when new packets are detected lost.

```
OnPacketsLost(lost_packets):
  // Remove lost packets from bytes_in_flight.
  for (lost_packet : lost_packets):
    bytes_in_flight -= lost_packet.bytes
  largest_lost_packet = lost_packets.last()

  // Start a new congestion epoch if the last lost packet
  // is past the end of the previous recovery epoch.
  CongestionEvent(largest_lost_packet.packet_number)
```

5.8.9. On Retransmission Timeout Verified

QUIC decreases the congestion window to the minimum value once the retransmission timeout has been verified and removes any packets sent before the newly acknowledged RTO packet.

```
OnRetransmissionTimeoutVerified(packet_number)
  congestion_window = kMinimumWindow
  // Declare all packets prior to packet_number lost.
  for (sent_packet: sent_packets):
    if (sent_packet.packet_number < packet_number):
      bytes_in_flight -= lost_packet.bytes
      sent_packets.remove(sent_packet.packet_number)
```

6. Security Considerations

6.1. Congestion Signals

Congestion control fundamentally involves the consumption of signals - both loss and ECN codepoints - from unauthenticated entities. On-path attackers can spoof or alter these signals. An attacker can cause endpoints to reduce their sending rate by dropping packets, or alter send rate by changing ECN codepoints.

6.2. Traffic Analysis

Packets that carry only ACK frames can be heuristically identified by observing packet size. Acknowledgement patterns may expose information about link characteristics or application behavior. Endpoints can use PADDING frames or bundle acknowledgments with other frames to reduce leaked information.

6.3. Misreporting ECN Markings

A receiver can misreport ECN markings to alter the congestion response of a sender. Suppressing reports of ECN-CE markings could cause a sender to increase their send rate. This increase could result in congestion and loss.

A sender MAY attempt to detect suppression of reports by marking occasional packets that they send with ECN-CE. If a packet marked with ECN-CE is not reported as having been marked when the packet is acknowledged, the sender SHOULD then disable ECN for that path.

Reporting additional ECN-CE markings will cause a sender to reduce their sending rate, which is similar in effect to advertising reduced connection flow control limits and so no advantage is gained by doing so.

Endpoints choose the congestion controller that they use. Though congestion controllers generally treat reports of ECN-CE markings as equivalent to loss [RFC8311], the exact response for each controller could be different. Failure to correctly respond to information about ECN markings is therefore difficult to detect.

7. IANA Considerations

This document has no IANA actions. Yet.

8. References

8.1. Normative References

[QUIC-TRANSPORT] Iyengar, J., Ed. and M. Thomson, Ed., "QUIC: A UDP-Based Multiplexed and Secure Transport", draft-ietf-quic-transport-15 (work in progress), October 2018.

[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>.

[RFC8174] Leiba, B., "Ambiguity of Uppercase vs Lowercase in RFC 2119 Key Words", BCP 14, RFC 8174, DOI 10.17487/RFC8174, May 2017, <https://www.rfc-editor.org/info/rfc8174>.

[RFC8311] Black, D., "Relaxing Restrictions on Explicit Congestion Notification (ECN) Experimentation", RFC 8311, DOI 10.17487/RFC8311, January 2018, <https://www.rfc-editor.org/info/rfc8311>.

8.2. Informative References

[RFC3465] Allman, M., "TCP Congestion Control with Appropriate Byte Counting (ABC)", RFC 3465, DOI 10.17487/RFC3465, February 2003, <https://www.rfc-editor.org/info/rfc3465>.

[RFC4653] Bhandarkar, S., Reddy, A., Allman, M., and E. Blanton, "Improving the Robustness of TCP to Non-Congestion Events", RFC 4653, DOI 10.17487/RFC4653, August 2006, <https://www.rfc-editor.org/info/rfc4653>.

[RFC5681] Allman, M., Paxson, V., and E. Blanton, "TCP Congestion Control", RFC 5681, DOI 10.17487/RFC5681, September 2009, <https://www.rfc-editor.org/info/rfc5681>.

[RFC5682] Sarolahti, P., Kojo, M., Yamamoto, K., and M. Hata, "Forward RTO-Recovery (F-RTO): An Algorithm for Detecting Spurious Retransmission Timeouts with TCP", RFC 5682, DOI 10.17487/RFC5682, September 2009, <https://www.rfc-editor.org/info/rfc5682>.

[RFC5827] Allman, M., Avrachenkov, K., Ayesta, U., Blanton, J., and P. Hurtig, "Early Retransmit for TCP and Stream Control Transmission Protocol (SCTP)", RFC 5827, DOI 10.17487/RFC5827, May 2010, <https://www.rfc-editor.org/info/rfc5827>.

[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>.

[RFC6582] Henderson, T., Floyd, S., Gurtov, A., and Y. Nishida, "The NewReno Modification to TCP's Fast Recovery Algorithm", RFC 6582, DOI 10.17487/RFC6582, April 2012, <https://www.rfc-editor.org/info/rfc6582>.

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

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

[TLP] Dukkipati, N., Cardwell, N., Cheng, Y., and M. Mathis, "Tail Loss Probe (TLP): An Algorithm for Fast Recovery of Tail Losses", draft-dukkipati-tcpm-tcp-loss-probe-01 (work in progress), February 2013.

8.3. URIs

[1] https://mailarchive.ietf.org/arch/search/?email_list=quic

[2] https://github.com/quicwg

[3] https://github.com/quicwg/base-drafts/labels/-recovery

Appendix A. Change Log

RFC Editor's Note: Please remove this section prior to publication of a final version of this document.

A.1. Since draft-ietf-quic-recovery-14

- Used max_ack_delay from transport params (#1796, #1782)
- Merge ACK and ACK_ECN (#1783)

A.2. Since draft-ietf-quic-recovery-13

- Corrected the lack of ssthresh reduction in CongestionEvent pseudocode (#1598)
- Considerations for ECN spoofing (#1426, #1626)
- Clarifications for PADDING and congestion control (#837, #838, #1517, #1531, #1540)
- Reduce early retransmission timer to RTT/8 (#945, #1581)
- Packets are declared lost after an RTO is verified (#935, #1582)

A.3. Since draft-ietf-quic-recovery-12

- Changes to manage separate packet number spaces and encryption levels (#1190, #1242, #1413, #1450)
- Added ECN feedback mechanisms and handling; new ACK_ECN frame (#804, #805, #1372)

A.4. Since draft-ietf-quic-recovery-11

No significant changes.

A.5. Since draft-ietf-quic-recovery-10

- Improved text on ack generation (#1139, #1159)
- Make references to TCP recovery mechanisms informational (#1195)
- Define time_of_last_sent_handshake_packet (#1171)
- Added signal from TLS the data it includes needs to be sent in a Retry packet (#1061, #1199)

- Minimum RTT (min_rtt) is initialized with an infinite value (#1169)

A.6. Since draft-ietf-quic-recovery-09

No significant changes.

A.7. Since draft-ietf-quic-recovery-08

- Clarified pacing and RTO (#967, #977)

A.8. Since draft-ietf-quic-recovery-07

- Include Ack Delay in RTO(and TLP) computations (#981)
- Ack Delay in SRTT computation (#961)
- Default RTT and Slow Start (#590)
- Many editorial fixes.

A.9. Since draft-ietf-quic-recovery-06

No significant changes.

A.10. Since draft-ietf-quic-recovery-05

- Add more congestion control text (#776)

A.11. Since draft-ietf-quic-recovery-04

No significant changes.

A.12. Since draft-ietf-quic-recovery-03

No significant changes.

A.13. Since draft-ietf-quic-recovery-02

- Integrate F-RTO (#544, #409)
- Add congestion control (#545, #395)
- Require connection abort if a skipped packet was acknowledged (#415)
- Simplify RTO calculations (#142, #417)

A.14. Since draft-ietf-quic-recovery-01

- Overview added to loss detection
- Changes initial default RTT to 100ms
- Added time-based loss detection and fixes early retransmit
- Clarified loss recovery for handshake packets
- Fixed references and made TCP references informative

A.15. Since draft-ietf-quic-recovery-00

- Improved description of constants and ACK behavior

A.16. Since draft-iyengar-quic-loss-recovery-01

- Adopted as base for draft-ietf-quic-recovery
- Updated authors/editors list
- Added table of contents

Acknowledgments

Authors' Addresses

Jana Iyengar (editor)
Fastly

Email: jri.ietf@gmail.com

Ian Swett (editor)
Google

Email: ianswett@google.com