QUIC   J. Iyengar, Ed.
Internet-Draft   Fastly
Intended status: Standards Track   I. Swett, Ed.
Expires: September 12, 2019   Google
                                                           March 11, 2019

QUIC Loss Detection and Congestion Control
draft-ietf-quic-recovery-19

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].

Table of Contents

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 existing 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 one or more ACK frame(s).

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

Ack-eliciting Frames: All frames besides ACK or PADDING are considered ack-eliciting.

Ack-eliciting Packets: Packets that contain ack-eliciting frames elicit an ACK from the receiver within the maximum ack delay and are called ack-eliciting packets.

Crypto Packets: Packets containing CRYPTO data sent in Initial or Handshake packets.

Out-of-order Packets: Packets that do not increase the largest received packet number for its packet number space by exactly one.

Packets arrive out of order when earlier packets are lost or delayed.

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:

o All packets are acknowledged, though packets that contain no ack-eliciting frames are only acknowledged along with ack-eliciting packets.

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

o Packets that contain only ACK frames do not count toward congestion control limits and are not considered in-flight.

o 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 round-trip time (RTT) measurement are unified across packet number spaces.

3.1.2. Monotonically Increasing Packet Numbers

TCP conflates transmission order at the sender with delivery order at the receiver, which results in retransmissions of the same data carrying the same sequence number, and consequently leads to "retransmission ambiguity". QUIC separates the two: QUIC uses a packet number to indicate transmission order, 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 within a packet number space, and directly encodes transmission order. A higher packet number signifies that the packet was sent later, and a lower packet number signifies that the packet was sent earlier. When a packet containing ack-eliciting frames is detected 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. Generating Acknowledgements

QUIC SHOULD delay sending acknowledgements in response to packets, but MUST NOT excessively delay acknowledgements of ack-eliciting packets. 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 ack-eliciting packet. QUIC recovery algorithms do not assume the peer sends an ACK immediately when receiving a second ack-eliciting packet.

In order to accelerate loss recovery and reduce timeouts, the receiver SHOULD send an immediate ACK after it receives an out-of-order packet. It could send immediate ACKs for in-order packets for a period of time that SHOULD NOT exceed 1/8 RTT unless more out-of-order packets arrive. If every packet arrives out-of- order, then an immediate ACK SHOULD be sent for every received packet.

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 the receiver can determine whether an immediate or delayed acknowledgement should be generated after processing incoming packets.

4.1. Crypto Handshake Data

In order to quickly complete the handshake and avoid spurious retransmissions due to crypto retransmission timeouts, crypto packets SHOULD use a very short ack delay, such as the local timer granularity. ACK frames MAY be sent immediately when the crypto stack indicates all data for that packet number space has been received.

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.

4.3. 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 and reordering, 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.

5. Computing the RTT estimate

Round-trip time (RTT) is calculated when an ACK frame arrives by computing the difference between the current time and the time the largest acked packet was sent. An RTT sample MUST NOT be taken for a packet that is not newly acknowledged or not ack-eliciting.

When RTT is calculated, the ack delay field from the ACK frame SHOULD be limited to the max_ack_delay specified by the peer. Limiting ack_delay to max_ack_delay ensures a peer specifying an extremely small max_ack_delay doesn't cause more spurious timeouts than a peer that correctly specifies max_ack_delay. It 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.

A sender calculates both smoothed RTT (SRTT) and RTT variance (RTTVAR) similar to those specified in [RFC6298], see Appendix A.6.

A sender takes an RTT sample when an ACK frame is received that acknowledges a larger packet number than before (see Appendix A.6). A sender will take multiple RTT samples per RTT when multiple such ACK frames are received within an RTT. When multiple samples are generated within an RTT, the smoothed RTT and RTT variance could retain inadequate history, as suggested in [RFC6298]. Changing these computations is currently an open research question.

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.

6. Loss Detection

QUIC senders use both ack information and timeouts to detect lost packets, and this section provides a description of these algorithms.

If a packet is lost, the QUIC transport needs to recover from that loss, such as by retransmitting the data, sending an updated frame, or abandoning the frame. For more information, see Section 13.2 of [QUIC-TRANSPORT].

6.1. Acknowledgement-based Detection

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

A packet is declared lost if it meets all the following conditions:

- The packet is unacknowledged, in-flight, and was sent prior to an acknowledged packet.

- o Either its packet number is kPacketThreshold smaller than an acknowledged packet (Section 6.1.1), or it was sent long enough in the past (Section 6.1.2).

The acknowledgement indicates that a packet sent later was delivered, while the packet and time thresholds provide some tolerance for packet reordering.

Spuriously declaring packets as lost leads to unnecessary retransmissions and may result in degraded performance due to the actions of the congestion controller upon detecting loss. Implementations that detect spurious retransmissions and increase the reordering threshold in packets or time MAY choose to start with smaller initial reordering thresholds to minimize recovery latency.

6.1.1. Packet Threshold

The RECOMMENDED initial value for the packet reordering threshold (kPacketThreshold) is 3, based on best practices for TCP loss detection [RFC5681] [RFC6675].

Some networks may exhibit higher degrees of reordering, causing a sender to detect spurious losses. Implementers MAY use algorithms developed for TCP, such as TCP-NCR [RFC4653], to improve QUIC's reordering resilience.

6.1.2. Time Threshold

Once a later packet has been acknowledged, an endpoint SHOULD declare an earlier packet lost if it was sent a threshold amount of time in the past. The time threshold is computed as kTimeThreshold * max(SRTT, latest_RTT). If packets sent prior to the largest acknowledged packet cannot yet be declared lost, then a timer SHOULD be set for the remaining time.

The RECOMMENDED time threshold (kTimeThreshold), expressed as a round-trip time multiplier, is 9/8.

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

- o the latest RTT sample is lower than the SRTT, perhaps due to reordering where the acknowledgement encountered a shorter path;

- o 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.

Implementations MAY experiment with absolute thresholds, thresholds from previous connections, adaptive thresholds, or including RTT variance. Smaller thresholds reduce reordering resilience and increase spurious retransmissions, and larger thresholds increase loss detection delay.

6.2. Crypto Retransmission Timeout

Data in CRYPTO frames is critical to QUIC transport and crypto negotiation, so a more aggressive timeout is used to retransmit it.

The initial crypto retransmission 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 an acknowledgement is received, a new RTT is computed and the timer SHOULD be set for twice the newly computed smoothed RTT.

When a crypto packet is sent, the sender MUST set a timer for the crypto timeout period. This timer MUST be updated when a new crypto packet is sent. Upon timeout, the sender MUST retransmit all unacknowledged CRYPTO data if possible.

Until the server has validated the client's address on the path, the amount of data it can send is limited, as specified in [QUIC-TRANSPORT]. If not all unacknowledged CRYPTO data can be sent, then all unacknowledged CRYPTO data sent in Initial packets should be retransmitted. If no data can be sent, then no alarm should be armed until data has been received from the client.

Because the server could be blocked until more packets are received, the client MUST start the crypto retransmission timer even if there is no unacknowledged CRYPTO data. If the timer expires and the client has no CRYPTO data to retransmit and does not have Handshake keys, it SHOULD send an Initial packet in a UDP datagram of at least 1200 bytes. If the client has Handshake keys, it SHOULD send a Handshake packet.

On each consecutive expiration of the crypto timer without receiving an acknowledgement for a new packet, the sender SHOULD double the crypto retransmission timeout and set a timer for this period.

When crypto packets are in flight, the probe timer (Section 6.3) is not active.

6.2.1. Retry and Version Negotiation

A Retry or Version Negotiation packet causes a client to send another Initial packet, effectively restarting the connection process and resetting congestion control and loss recovery state, including resetting any pending timers. Either packet indicates that the Initial was received but not processed. Neither packet can be treated as an acknowledgment for the Initial.

The client MAY however compute an RTT estimate to the server as the time period from when the first Initial was sent to when a Retry or a Version Negotiation packet is received. The client MAY use this value to seed the RTT estimator for a subsequent connection attempt to the server.

6.2.2. Discarding Keys and Packet State

When packet protection keys are discarded (see Section 4.9 of [QUIC-TLS]), all packets that were sent with those keys can no longer be acknowledged because their acknowledgements cannot be processed anymore. The sender MUST discard all recovery state associated with those packets and MUST remove them from the count of bytes in flight.

Endpoints stop sending and receiving Initial packets once they start exchanging Handshake packets (see Section 17.2.2.1 of [QUIC-TRANSPORT]). At this point, recovery state for all in-flight Initial packets is discarded.

When 0-RTT is rejected, recovery state for all in-flight 0-RTT packets is discarded.

If a server accepts 0-RTT, but does not buffer 0-RTT packets that arrive before Initial packets, early 0-RTT packets will be declared lost, but that is expected to be infrequent.

It is expected that keys are discarded after packets encrypted with them would be acknowledged or declared lost. Initial secrets however might be destroyed sooner, as soon as handshake keys are available (see Section 4.10 of [QUIC-TLS]).

6.3. Probe Timeout

A Probe Timeout (PTO) triggers a probe packet when ack-eliciting data is in flight but an acknowledgement is not received within the expected period of time. A PTO enables a connection to recover from loss of tail packets or acks. The PTO algorithm used in QUIC implements the reliability functions of Tail Loss Probe [TLP] [RACK], RTO [RFC5681] and F-RTO algorithms for TCP [RFC5682], and the timeout

computation is based on TCP's retransmission timeout period [RFC6298].

6.3.1. Computing PTO

When an ack-eliciting packet is transmitted, the sender schedules a timer for the PTO period as follows:

PTO = smoothed_rtt + max(4*rttvar, kGranularity) + max_ack_delay

kGranularity, smoothed_rtt, rttvar, and max_ack_delay are defined in Appendix A.2 and Appendix A.3.

The PTO period is the amount of time that a sender ought to wait for an acknowledgement of a sent packet. This time period includes the estimated network roundtrip-time (smoothed_rtt), the variance in the estimate (4*rttvar), and max_ack_delay, to account for the maximum time by which a receiver might delay sending an acknowledgement.

The PTO value MUST be set to at least kGranularity, to avoid the timer expiring immediately.

When a PTO timer expires, the sender probes the network as described in the next section. The PTO period MUST be set to twice its current value. This exponential reduction in the sender's rate is important because the PTOs might be caused by loss of packets or acknowledgements due to severe congestion.

A sender computes its PTO timer every time an ack-eliciting packet is sent. A sender might choose to optimize this by setting the timer fewer times if it knows that more ack-eliciting packets will be sent within a short period of time.

6.3.2. Sending Probe Packets

When a PTO timer expires, the sender MUST send one ack-eliciting packet as a probe. A sender MAY send up to two ack-eliciting packets, to avoid an expensive consecutive PTO expiration due to a single packet loss.

Consecutive PTO periods increase exponentially, and as a result, connection recovery latency increases exponentially as packets continue to be dropped in the network. Sending two packets on PTO expiration increases resilience to packet drops, thus reducing the probability of consecutive PTO events.

Probe packets sent on a PTO MUST be ack-eliciting. A probe packet SHOULD carry new data when possible. A probe packet MAY carry

retransmitted unacknowledged data when new data is unavailable, when flow control does not permit new data to be sent, or to opportunistically reduce loss recovery delay. Implementations MAY use alternate strategies for determining the content of probe packets, including sending new or retransmitted data based on the application's priorities.

When the PTO timer expires multiple times and new data cannot be sent, implementations must choose between sending the same payload every time or sending different payloads. Sending the same payload may be simpler and ensures the highest priority frames arrive first. Sending different payloads each time reduces the chances of spurious retransmission.

When a PTO timer expires, new or previously-sent data may not be available to send and packets may still be in flight. A sender can be blocked from sending new data in the future if packets are left in flight. Under these conditions, a sender SHOULD mark any packets still in flight as lost. If a sender wishes to establish delivery of packets still in flight, it MAY send an ack-eliciting packet and re-arm the PTO timer instead.

6.3.3. Loss Detection

Delivery or loss of packets in flight is established when an ACK frame is received that newly acknowledges one or more packets.

A PTO timer expiration event does not indicate packet loss and MUST NOT cause prior unacknowledged packets to be marked as lost. When an acknowledgement is received that newly acknowledges packets, loss detection proceeds as dictated by packet and time threshold mechanisms, see Section 6.1.

6.4. 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.

7. 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 Appendix B.2) beyond the available congestion window, unless the packet is a probe packet sent after a PTO timer expires, as described in Section 6.3.

Implementations MAY use other congestion control algorithms, such as Cubic [RFC8312], 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.

7.1. 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.

7.2. 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 PTO. While in slow start, QUIC increases the congestion window by the number of bytes acknowledged when each acknowledgment is processed.

7.3. 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.

7.4. 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 does not retransmit 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.

7.5. Ignoring Loss of Undecryptable Packets

During the handshake, some packet protection keys might not be available when a packet arrives. In particular, Handshake and 0-RTT packets cannot be processed until the Initial packets arrive, and 1-RTT packets cannot be processed until the handshake completes. Endpoints MAY ignore the loss of Handshake, 0-RTT, and 1-RTT packets that might arrive before the peer has packet protection keys to process those packets.

7.6. Probe Timeout

Probe packets MUST NOT be blocked by the congestion controller. A sender MUST however count these packets as being additionally in flight, since these packets add network load without establishing packet loss. Note that sending probe packets might cause the sender's bytes in flight to exceed the congestion window until an acknowledgement is received that establishes loss or delivery of packets.

7.7. Persistent Congestion

When an ACK frame is received that establishes loss of all in-flight packets sent over a long enough period of time, the network is considered to be experiencing persistent congestion. Commonly, this can be established by consecutive PTOs, but since the PTO timer is reset when a new ack-eliciting packet is sent, an explicit duration must be used to account for those cases where PTOs do not occur or are substantially delayed. This duration is the equivalent of kPersistentCongestionThreshold consecutive PTOs, and is computed as follows: ~~~ (smoothed_rtt + 4 * rttvar + max_ack_delay) * ((2 ^ kPersistentCongestionThreshold) - 1) ~~~

For example, assume:

smoothed_rtt = 1 rttvar = 0 max_ack_delay = 0
kPersistentCongestionThreshold = 2

If an eck-eliciting packet is sent at time = 0, the following scenario would illustrate persistent congestion:

```
+-----+------------------------+
| t=0 | Send Pkt #1 (App Data) |
+-----+------------------------+
| t=1 | Send Pkt #2 (PTO 1)    |
|     |                        |
| t=3 | Send Pkt #3 (PTO 2)    |
|     |                        |
| t=7 | Send Pkt #4 (PTO 3)    |
|     |                        |
| t=8 | Recv ACK of Pkt #4     |
+-----+------------------------+
```

The first three packets are determined to be lost when the ACK of packet 4 is received at t=8. The congestion period is calculated as the time between the oldest and newest lost packets: (3 - 0) = 3. The duration for persistent congestion is equal to: (1 * ((2 ^ kPersistentCongestionThreshold) - 1)) = 3. Because the threshold was reached and because none of the packets between the oldest and the newest packets are acknowledged, the network is considered to have experienced persistent congestion.

When persistent congestion is established, the sender's congestion window MUST be reduced to the minimum congestion window (kMinimumWindow). This response of collapsing the congestion window on persistent congestion is functionally similar to a sender's response on a Retransmission Timeout (RTO) in TCP [RFC5681] after Tail Loss Probes (TLP) [TLP].

7.8. 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).

7.9. Sending data after an idle period

A sender becomes idle if it ceases to send data and has no bytes in flight. A sender's congestion window MUST NOT increase while it is idle.

When sending data after becoming idle, a sender MUST reset its congestion window to the initial congestion window (see Section 4.1 of [RFC5681]), unless it paces the sending of packets. A sender MAY retain its congestion window if it paces the sending of any packets in excess of the initial congestion window.

A sender MAY implement alternate mechanisms to update its congestion window after idle periods, such as those proposed for TCP in [RFC7661].

7.10. Application Limited Sending

The congestion window should not be increased in slow start or congestion avoidance when it is not fully utilized. The congestion window could be under-utilized due to insufficient application data or flow control credit.

A sender that paces packets (see Section 7.8) might delay sending packets and not fully utilize the congestion window due to this delay. A sender should not consider itself application limited if it would have fully utilized the congestion window without pacing delay.

8. Security Considerations

8.1. 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.

8.2. 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.

8.3. 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.

9. IANA Considerations

This document has no IANA actions. Yet.

10. References

10.1. Normative References

[QUIC-TLS]
Thomson, M., Ed. and S. Turner, Ed., "Using TLS to Secure QUIC", draft-ietf-quic-tls-19 (work in progress), March 2019.

[QUIC-TRANSPORT]
Iyengar, J., Ed. and M. Thomson, Ed., "QUIC: A UDP-Based Multiplexed and Secure Transport", draft-ietf-quic-transport-19 (work in progress), March 2019.

[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>.

10.2. Informative References

[FACK] Mathis, M. and J. Mahdavi, "Forward Acknowledgement: Refining TCP Congestion Control", ACM SIGCOMM , August 1996.

[RACK] Cheng, Y., Cardwell, N., Dukkipati, N., and P. Jha, "RACK: a time-based fast loss detection algorithm for TCP", draft-ietf-tcpm-rack-04 (work in progress), July 2018.

[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>.

[RFC7661] Fairhurst, G., Sathiaseelan, A., and R. Secchi, "Updating TCP to Support Rate-Limited Traffic", RFC 7661, DOI 10.17487/RFC7661, October 2015, <https://www.rfc-editor.org/info/rfc7661>.

[RFC8312] Rhee, I., Xu, L., Ha, S., Zimmermann, A., Eggert, L., and R. Scheffenegger, "CUBIC for Fast Long-Distance Networks", RFC 8312, DOI 10.17487/RFC8312, February 2018, <https://www.rfc-editor.org/info/rfc8312>.

[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.

10.3. 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. Loss Recovery Pseudocode

We now describe an example implementation of the loss detection mechanisms described in Section 6.

A.1. Tracking Sent Packets

To correctly implement congestion control, a QUIC sender tracks every ack-eliciting packet until the packet is acknowledged or lost. It is expected that implementations will be able to access this information by packet number and crypto context and store the per-packet fields (Appendix A.1.1) for loss recovery and congestion control.

After a packet is declared lost, it SHOULD be tracked for an amount of time comparable to the maximum expected packet reordering, such as 1 RTT. This allows for detection of spurious retransmissions.

Sent packets are tracked for each packet number space, and ACK processing only applies to a single space.

A.1.1. Sent Packet Fields

packet_number: The packet number of the sent packet.

ack_eliciting: A boolean that indicates whether a packet is ack-eliciting. If true, it is expected that an acknowledgement will be received, though the peer could delay sending the ACK frame containing it by up to the MaxAckDelay.

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

is_crypto_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.

time_sent: The time the packet was sent.

A.2. 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.

kPacketThreshold: Maximum reordering in packets before packet threshold loss detection considers a packet lost. The RECOMMENDED value is 3.

kTimeThreshold: Maximum reordering in time before time threshold loss detection considers a packet lost. Specified as an RTT multiplier. The RECOMMENDED value is 9/8.

kGranularity: Timer granularity. This is a system-dependent value. However, implementations SHOULD use a value no smaller than 1ms.

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

kPacketNumberSpace: An enum to enumerate the three packet number spaces. ~~~ enum kPacketNumberSpace { Initial, Handshake, ApplicationData, } ~~~

A.3. 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.

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

pto_count: The number of times a PTO has been sent without receiving an ack.

time_of_last_sent_ack_eliciting_packet: The time the most recent ack-eliciting packet was sent.

time_of_last_sent_crypto_packet: The time the most recent crypto packet was sent.

largest_acked_packet[kPacketNumberSpace]: The largest packet number acknowledged in the packet number space so far.

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.

loss_time[kPacketNumberSpace]: The time at which the next packet in that packet number space will be considered lost based on exceeding the reordering window in time.

sent_packets[kPacketNumberSpace]: An association of packet numbers in a packet number space to information about them. Described in detail above in Appendix A.1.

A.4. Initialization

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

```
loss_detection_timer.reset()
crypto_count = 0
pto_count = 0
smoothed_rtt = 0
rttvar = 0
min_rtt = infinite
time_of_last_sent_ack_eliciting_packet = 0
time_of_last_sent_crypto_packet = 0
for pn_space in [ Initial, Handshake, ApplicatonData ]:
  largest_acked_packet[pn_space] = 0
  loss_time[pn_space] = 0
```

A.5. On Sending a Packet

After a packet is sent, information about the packet is stored. The parameters to OnPacketSent are described in detail above in Appendix A.1.1.

Pseudocode for OnPacketSent follows:

```
 OnPacketSent(packet_number, pn_space, ack_eliciting,
              in_flight, is_crypto_packet, sent_bytes):
   sent_packets[pn_space][packet_number].packet_number =
                                            packet_number
   sent_packets[pn_space][packet_number].time_sent = now
   sent_packets[pn_space][packet_number].ack_eliciting =
                                            ack_eliciting
   sent_packets[pn_space][packet_number].in_flight = in_flight
   if (in_flight):
     if (is_crypto_packet):
       time_of_last_sent_crypto_packet = now
     if (ack_eliciting):
       time_of_last_sent_ack_eliciting_packet = now
     OnPacketSentCC(sent_bytes)
     sent_packets[pn_space][packet_number].size = sent_bytes
     SetLossDetectionTimer()
```

A.6. 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, pn_space):
  largest_acked_packet[pn_space] =
      max(largest_acked_packet[pn_space], ack.largest_acked)

  // If the largest acknowledged is newly acked and
  // ack-eliciting, update the RTT.
  if (sent_packets[pn_space][ack.largest_acked] &&
      sent_packets[pn_space][ack.largest_acked].ack_eliciting):
    latest_rtt =
      now - sent_packets[pn_space][ack.largest_acked].time_sent
    UpdateRtt(latest_rtt, ack.ack_delay)

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

  // Find all newly acked packets in this ACK frame
  newly_acked_packets = DetermineNewlyAckedPackets(ack, pn_space)
  if (newly_acked_packets.empty()):
    return

  for acked_packet in newly_acked_packets:
    OnPacketAcked(acked_packet.packet_number, pn_space)

  DetectLostPackets(pn_space)

  crypto_count = 0
  pto_count = 0

  SetLossDetectionTimer()


UpdateRtt(latest_rtt, ack_delay):
  // min_rtt ignores ack delay.
  min_rtt = min(min_rtt, latest_rtt)
  // Limit ack_delay by max_ack_delay
  ack_delay = min(ack_delay, max_ack_delay)
  // 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
```

A.7. On Packet Acknowledgment

When a packet is acknowledged 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 acknowledged packets.

OnPacketAcked takes two parameters: acked_packet, which is the struct detailed in Appendix A.1.1, and the packet number space that this ACK frame was sent for.

Pseudocode for OnPacketAcked follows:

```
OnPacketAcked(acked_packet, pn_space):
  if (acked_packet.ack_eliciting):
    OnPacketAckedCC(acked_packet)
  sent_packets[pn_space].remove(acked_packet.packet_number)
```

A.8. Setting the Loss Detection Timer

QUIC loss detection uses a single timer for all timeout 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.

This algorithm may result in the timer being set in the past, particularly if timers wake up late. Timers set in the past SHOULD fire immediately.

Pseudocode for SetLossDetectionTimer follows:

```
// Returns the earliest loss_time and the packet number
// space it's from.  Returns 0 if all times are 0.
GetEarliestLossTime():
  time = loss_time[Initial]
  space = Initial
  for pn_space in [ Handshake, ApplicatonData ]:
    if loss_time[pn_space] != 0 &&
       (time == 0 || loss_time[pn_space] < time):
      time = loss_time[pn_space];
      space = pn_space
  return time, space

SetLossDetectionTimer():
  // Don't arm timer if there are no ack-eliciting packets
  // in flight.
  if (no ack-eliciting packets in flight):
    loss_detection_timer.cancel()
    return

  loss_time, _ = GetEarliestLossTime()
  if (loss_time != 0):
    // Time threshold loss detection.
    loss_detection_timer.update(loss_time)
    return

  if (crypto packets are in flight):
    // Crypto retransmission timer.
    if (smoothed_rtt == 0):
      timeout = 2 * kInitialRtt
    else:
      timeout = 2 * smoothed_rtt
    timeout = max(timeout, kGranularity)
    timeout = timeout * (2 ^ crypto_count)
    loss_detection_timer.update(
      time_of_last_sent_crypto_packet + timeout)
    return

  // Calculate PTO duration
  timeout =
    smoothed_rtt + max(4 * rttvar, kGranularity) + max_ack_delay
  timeout = timeout * (2 ^ pto_count)

  loss_detection_timer.update(
    time_of_last_sent_ack_eliciting_packet + timeout)
```

A.9. On Timeout

When the loss detection timer expires, the timer's mode determines the action to be performed.

Pseudocode for OnLossDetectionTimeout follows:

```
OnLossDetectionTimeout():
  loss_time, pn_space = GetEarliestLossTime()
  if (loss_time != 0):
    // Time threshold loss Detection
    DetectLostPackets(pn_space)
  // Retransmit crypto data if no packets were lost
  // and there are still crypto packets in flight.
  else if (crypto packets are in flight):
    // Crypto retransmission timeout.
    RetransmitUnackedCryptoData()
    crypto_count++
  else:
    // PTO
    SendOneOrTwoPackets()
    pto_count++

  SetLossDetectionTimer()
```

A.10. Detecting Lost Packets

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 acknowledged packet is supplied.

Pseudocode for DetectLostPackets follows:

```
DetectLostPackets(pn_space):
  loss_time[pn_space] = 0
  lost_packets = {}
  loss_delay = kTimeThreshold * max(latest_rtt, smoothed_rtt)

  // Packets sent before this time are deemed lost.
  lost_send_time = now() - loss_delay

  // Packets with packet numbers before this are deemed lost.
  lost_pn = largest_acked_packet[pn_space] - kPacketThreshold

  foreach unacked in sent_packets:
    if (unacked.packet_number > largest_acked_packet[pn_space]):
      continue

    // Mark packet as lost, or set time when it should be marked.
    if (unacked.time_sent <= lost_send_time ||
        unacked.packet_number <= lost_pn):
      sent_packets.remove(unacked.packet_number)
      if (unacked.in_flight):
        lost_packets.insert(unacked)
    else:
      if (loss_time[pn_space] == 0):
        loss_time[pn_space] = unacked.time_sent + loss_delay
      else:
        loss_time[pn_space] = min(loss_time[pn_space],
                                  unacked.time_sent + loss_delay)

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

Appendix B. Congestion Control Pseudocode

We now describe an example implementation of the congestion controller described in Section 7.

B.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 data in flight, in bytes. Taken from [RFC6928], but increased slightly to account for the smaller 8 byte overhead of UDP vs 20 bytes for TCP. The RECOMMENDED value is the minimum of 10 * kMaxDatagramSize and max(2* kMaxDatagramSize, 14720)).

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.

kPersistentCongestionThreshold: Number of consecutive PTOs required for persistent congestion to be established. The rationale for this threshold is to enable a sender to use initial PTOs for aggressive probing, as TCP does with Tail Loss Probe (TLP) [TLP] [RACK], before establishing persistent congestion, as TCP does with a Retransmission Timeout (RTO) [RFC5681]. The RECOMMENDED value for kPersistentCongestionThreshold is 2, which is equivalent to having two TLPs before an RTO in TCP.

B.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 ack-eliciting 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.

recovery_start_time: The time when QUIC first detects a loss, causing it to enter recovery. When a packet sent after this time 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.

B.3. Initialization

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

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

B.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
```

B.5. On Packet Acknowledgement

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

```
InRecovery(sent_time):
  return sent_time <= recovery_start_time

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

B.6. On New Congestion Event

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

```
CongestionEvent(sent_time):
  // Start a new congestion event if the sent time is larger
  // than the start time of the previous recovery epoch.
  if (!InRecovery(sent_time)):
    recovery_start_time = Now()
    congestion_window *= kLossReductionFactor
    congestion_window = max(congestion_window, kMinimumWindow)
    ssthresh = congestion_window
```

B.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 was sent after the start of the previous
    // recovery epoch.
    CongestionEvent(sent_packets[ack.largest_acked].time_sent)
```

B.8. On Packets Lost

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

```
InPersistentCongestion(largest_lost_packet):
  pto = smoothed_rtt + max(4 * rttvar, kGranularity) +
    max_ack_delay
  congestion_period =
    pto * (2 ^ kPersistentCongestionThreshold - 1)
  // Determine if all packets in the window before the
  // newest lost packet, including the edges, are marked
  // lost
  return IsWindowLost(largest_lost_packet, congestion_period)

OnPacketsLost(lost_packets):
  // Remove lost packets from bytes_in_flight.
  for (lost_packet : lost_packets):
    bytes_in_flight -= lost_packet.size
  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.time_sent)

  // Collapse congestion window if persistent congestion
  if (InPersistentCongestion(largest_lost_packet)):
    congestion_window = kMinimumWindow
```

Appendix C. Change Log

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

Issue and pull request numbers are listed with a leading octothorp.

C.1. Since draft-ietf-quic-recovery-18

- Change IW byte limit to 14720 from 14600 (#2494)
- Update PTO calculation to match RFC6298 (#2480, #2489, #2490)
- Improve loss detection's description of multiple packet number spaces and pseudocode (#2485, #2451, #2417)
- Declare persistent congestion even if non-probe packets are sent and don't make persistent congestion more aggressive than RTO verified was (#2365, #2244)
- Move pseudocode to the appendices (#2408)
- What to send on multiple PTOs (#2380)

C.2. Since draft-ietf-quic-recovery-17

- After Probe Timeout discard in-flight packets or send another (#2212, #1965)

- Endpoints discard initial keys as soon as handshake keys are available (#1951, #2045)

- 0-RTT state is discarded when 0-RTT is rejected (#2300)

- Loss detection timer is cancelled when ack-eliciting frames are in flight (#2117, #2093)

- Packets are declared lost if they are in flight (#2104)

- After becoming idle, either pace packets or reset the congestion controller (#2138, 2187)

- Process ECN counts before marking packets lost (#2142)

- Mark packets lost before resetting crypto_count and pto_count (#2208, #2209)

- Congestion and loss recovery state are discarded when keys are discarded (#2327)

C.3. Since draft-ietf-quic-recovery-16

- Unify TLP and RTO into a single PTO; eliminate min RTO, min TLP and min crypto timeouts; eliminate timeout validation (#2114, #2166, #2168, #1017)

- Redefine how congestion avoidance in terms of when the period starts (#1928, #1930)

- Document what needs to be tracked for packets that are in flight (#765, #1724, #1939)

- Integrate both time and packet thresholds into loss detection (#1969, #1212, #934, #1974)

- Reduce congestion window after idle, unless pacing is used (#2007, #2023)

- Disable RTT calculation for packets that don't elicit acknowledgment (#2060, #2078)

- Limit ack_delay by max_ack_delay (#2060, #2099)

- Initial keys are discarded once Handshake are avaialble (#1951, #2045)

- Reorder ECN and loss detection in pseudocode (#2142)

- Only cancel loss detection timer if ack-eliciting packets are in flight (#2093, #2117)

C.4. Since draft-ietf-quic-recovery-14

- Used max_ack_delay from transport params (#1796, #1782)

- Merge ACK and ACK_ECN (#1783)

C.5. 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)

C.6. 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)

C.7. Since draft-ietf-quic-recovery-11

No significant changes.

C.8. 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)

C.9. Since draft-ietf-quic-recovery-09

No significant changes.

C.10. Since draft-ietf-quic-recovery-08

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

C.11. 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.

C.12. Since draft-ietf-quic-recovery-06

No significant changes.

C.13. Since draft-ietf-quic-recovery-05

- Add more congestion control text (#776)

C.14. Since draft-ietf-quic-recovery-04

No significant changes.

C.15. Since draft-ietf-quic-recovery-03

No significant changes.

C.16. 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)

C.17. 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

C.18. Since draft-ietf-quic-recovery-00

- Improved description of constants and ACK behavior

C.19. 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