QUIC J. Iyengar, Ed.
Internet-Draft Fastly
Intended status: Standards Track I. Swett, Ed.
Expires: 23 April 2021 Google
20 October 2020

QUIC Loss Detection and Congestion Control
draft-ietf-quic-recovery-32

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 (mailto:quic@ietf.org)), which is archived at https://mailarchive.ietf.org/arch/search/?email_list=quic.

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

Status of This Memo

Copyright Notice

Table of Contents

1. Introduction

QUIC is a new multiplexed and secure transport protocol atop UDP, specified in [QUIC-TRANSPORT]. This document describes congestion control and loss recovery for QUIC. Mechanisms described in this document follow the spirit of existing TCP congestion control and loss recovery mechanisms, described in RFCs, various Internet-drafts, or academic papers, and also those prevalent in 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-eliciting frames: All frames other than ACK, PADDING, and CONNECTION_CLOSE are considered ack-eliciting.

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

In-flight: Packets are considered in-flight when they are ack-eliciting or contain a PADDING frame, and they have been sent but are not acknowledged, declared lost, or discarded along with old keys.

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 are sent in monotonically increasing order within a space, preventing ambiguity.

This design obviates the need for disambiguating between transmissions and retransmissions; this 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 no ack-eliciting frames are only acknowledged along with ack-eliciting packets.

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

* Packets containing frames besides ACK or CONNECTION_CLOSE frames count toward congestion control limits and are considered in-flight.

* PADDING frames cause packets to contribute toward bytes in flight without directly causing an acknowledgment to be sent.

4. 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. However, protocol differences between QUIC and TCP contribute to algorithmic differences. These protocol differences are briefly described below.

4.1. 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.

4.2. 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. Application data is sent in one or more streams and delivery order is 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 includes 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.

4.3. Clearer Loss Epoch

QUIC starts a loss epoch when a packet is lost and ends one when any packet sent after the epoch starts is acknowledged. TCP waits for the gap in the sequence number space to be filled, and so if a segment is lost multiple times in a row, the loss epoch may not end for several round trips. Because both should reduce their congestion windows only once per epoch, QUIC will do it once for every round trip that experiences loss, while TCP may only do it once across multiple round trips.

4.4. No Reneging

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

4.5. 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.

4.6. Explicit Correction For Delayed Acknowledgements

QUIC endpoints measure the delay incurred between when a packet is received and when the corresponding acknowledgment is sent, allowing a peer to maintain a more accurate round-trip time estimate; see Section 13.2 of [QUIC-TRANSPORT].

4.7. Probe Timeout Replaces RTO and TLP

QUIC uses a probe timeout (PTO; see Section 6.2), with a timer based on TCP's RTO computation. QUIC's PTO includes the peer's maximum expected acknowledgement delay instead of using a fixed minimum timeout. QUIC does not collapse the congestion window until persistent congestion (Section 7.6) is declared, unlike TCP, which collapses the congestion window upon expiry of an RTO. Instead of collapsing the congestion window and declaring everything in-flight lost, QUIC allows probe packets to temporarily exceed the congestion window whenever the timer expires.

In doing this, QUIC avoids unnecessary congestion window reductions, obviating the need for correcting mechanisms such as F-RTO ([RFC5682]). Since QUIC does not collapse the congestion window on a PTO expiration, a QUIC sender is not limited from sending more in-flight packets after a PTO expiration if it still has available congestion window. This occurs when a sender is application-limited and the PTO timer expires. This is more aggressive than TCP's RTO mechanism when application-limited, but identical when not application-limited.

A single packet loss at the tail does not indicate persistent congestion, so QUIC specifies a time-based definition to ensure one or more packets are sent prior to a dramatic decrease in congestion window; see Section 7.6.

4.8. The Minimum Congestion Window is Two Packets

TCP uses a minimum congestion window of one packet. However, loss of that single packet means that the sender needs to waiting for a PTO (Section 6.2) to recover, which can be much longer than a round-trip time. Sending a single ack-eliciting packet also increases the chances of incurring additional latency when a receiver delays its acknowledgement.

QUIC therefore recommends that the minimum congestion window be two packets. While this increases network load, it is considered safe, since the sender will still reduce its sending rate exponentially under persistent congestion (Section 6.2).

5. Estimating the Round-Trip Time

At a high level, an endpoint measures the time from when a packet was sent to when it is acknowledged as a round-trip time (RTT) sample. The endpoint uses RTT samples and peer-reported host delays (see Section 13.2 of [QUIC-TRANSPORT]) to generate a statistical description of the network path's RTT. An endpoint computes the following three values for each path: the minimum value observed over the lifetime of the path (min_rtt), an exponentially-weighted moving average (smoothed_rtt), and the mean deviation (referred to as "variation" in the rest of this document) in the observed RTT samples (rttvar).

5.1. Generating RTT samples

An endpoint generates an RTT sample on receiving an ACK frame that meets the following two conditions:

* the largest acknowledged packet number is newly acknowledged, and

* at least one of the newly acknowledged packets was ack-eliciting.

The RTT sample, latest_rtt, is generated as the time elapsed since the largest acknowledged packet was sent:

```
latest_rtt = ack_time - send_time_of_largest_acked
```

An RTT sample is generated using only the largest acknowledged packet in the received ACK frame. This is because a peer reports acknowledgment delays for only the largest acknowledged packet in an ACK frame. While the reported acknowledgment delay is not used by the RTT sample measurement, it is used to adjust the RTT sample in subsequent computations of smoothed_rtt and rttvar (Section 5.3).

To avoid generating multiple RTT samples for a single packet, an ACK frame SHOULD NOT be used to update RTT estimates if it does not newly acknowledge the largest acknowledged packet.

An RTT sample MUST NOT be generated on receiving an ACK frame that does not newly acknowledge at least one ack-eliciting packet. A peer usually does not send an ACK frame when only non-ack-eliciting packets are received. Therefore an ACK frame that contains acknowledgements for only non-ack-eliciting packets could include an arbitrarily large ACK Delay value. Ignoring such ACK frames avoids complications in subsequent smoothed_rtt and rttvar computations.

A sender might generate multiple RTT samples per RTT when multiple ACK frames are received within an RTT. As suggested in [RFC6298], doing so might result in inadequate history in smoothed_rtt and rttvar. Ensuring that RTT estimates retain sufficient history is an open research question.

5.2. Estimating min_rtt

min_rtt is the sender's estimate of the minimum RTT observed for a given network path. In this document, min_rtt is used by loss detection to reject implausibly small rtt samples.

min_rtt MUST be set to the latest_rtt on the first RTT sample. min_rtt MUST be set to the lesser of min_rtt and latest_rtt (Section 5.1) on all other samples.

An endpoint uses only locally observed times in computing the min_rtt and does not adjust for acknowledgment delays reported by the peer. Doing so allows the endpoint to set a lower bound for the smoothed_rtt based entirely on what it observes (see Section 5.3), and limits potential underestimation due to erroneously-reported delays by the peer.

The RTT for a network path may change over time. If a path's actual RTT decreases, the min_rtt will adapt immediately on the first low sample. If the path's actual RTT increases however, the min_rtt will not adapt to it, allowing future RTT samples that are smaller than the new RTT to be included in smoothed_rtt.

Endpoints SHOULD set the min_rtt to the newest RTT sample after persistent congestion is established. This is to allow a connection to reset its estimate of min_rtt and smoothed_rtt (Section 5.3) after a disruptive network event, and because it is possible that an increase in path delay resulted in persistent congestion being incorrectly declared.

Endpoints MAY re-establish the min_rtt at other times in the connection, such as when traffic volume is low and an acknowledgement is received with a low acknowledgement delay. Implementations SHOULD NOT refresh the min_rtt value too often, since the actual minimum RTT of the path is not frequently observable.

5.3. Estimating smoothed_rtt and rttvar

smoothed_rtt is an exponentially-weighted moving average of an endpoint's RTT samples, and rttvar is the variation in the RTT samples, estimated using a mean variation.

The calculation of smoothed_rtt uses RTT samples after adjusting them for acknowledgement delays. These delays are decoded from the ACK Delay field of ACK frames as described in Section 19.3 of [QUIC-TRANSPORT].

The peer might report acknowledgement delays that are larger than the peer's max_ack_delay during the handshake (Section 13.2.1 of [QUIC-TRANSPORT]). To account for this, the endpoint SHOULD ignore max_ack_delay until the handshake is confirmed (Section 4.1.2 of [QUIC-TLS]). When they occur, these large acknowledgement delays are likely to be non-repeating and limited to the handshake. The endpoint can therefore use them without limiting them to the max_ack_delay, avoiding unnecessary inflation of the RTT estimate.

Note however that a large acknowledgement delay can result in a substantially inflated smoothed_rtt, if there is either an error in the peer's reporting of the acknowledgement delay or in the endpoint's min_rtt estimate. Therefore, prior to handshake confirmation, an endpoint MAY ignore RTT samples if adjusting the RTT sample for acknowledgement delay causes the sample to be less than the min_rtt.

After the handshake is confirmed, any acknowledgement delays reported by the peer that are greater than the peer's max_ack_delay are attributed to unintentional but potentially repeating delays, such as scheduler latency at the peer or loss of previous acknowledgements. Excess delays could also be due to a non-compliant receiver. Therefore, these extra delays are considered effectively part of path delay and incorporated into the RTT estimate.

Therefore, when adjusting an RTT sample using peer-reported acknowledgement delays, an endpoint:

* MAY ignore the acknowledgement delay for Initial packets, since these acknowledgements are not delayed by the peer (Section 13.2.1 of [QUIC-TRANSPORT]);

* SHOULD ignore the peer's max_ack_delay until the handshake is confirmed;

* MUST use the lesser of the acknowledgement delay and the peer's max_ack_delay after the handshake is confirmed; and

* MUST NOT subtract the acknowledgement delay from the RTT sample if the resulting value is smaller than the min_rtt. This limits the underestimation of the smoothed_rtt due to a misreporting peer.

Additionally, an endpoint might postpone the processing of acknowledgements when the corresponding decryption keys are not immediately available. For example, a client might receive an acknowledgement for a 0-RTT packet that it cannot decrypt because 1-RTT packet protection keys are not yet available to it. In such cases, an endpoint SHOULD subtract such local delays from its RTT sample until the handshake is confirmed.

Similar to [RFC6298], smoothed_rtt and rttvar are computed as follows.

An endpoint initializes the RTT estimator during connection establishment and when the estimator is reset during connection migration; see Section 9.4 of [QUIC-TRANSPORT]. Before any RTT samples are available for a new path or when the estimator is reset, the estimator is initialized using the initial RTT; see Section 6.2.2.

smoothed_rtt and rttvar are initialized as follows, where kInitialRtt contains the initial RTT value:

```
smoothed_rtt = kInitialRtt
rttvar = kInitialRtt / 2
```

RTT samples for the network path are recorded in latest_rtt; see Section 5.1. On the first RTT sample after initialization, the estimator is reset using that sample. This ensures that the estimator retains no history of past samples.

On the first RTT sample after initialization, smoothed_rtt and rttvar are set as follows:

```
smoothed_rtt = latest_rtt
rttvar = latest_rtt / 2
```

On subsequent RTT samples, smoothed_rtt and rttvar evolve as follows:

```
ack_delay = decoded acknowledgement delay from ACK frame
if (handshake confirmed):
  ack_delay = min(ack_delay, max_ack_delay)
adjusted_rtt = latest_rtt
if (min_rtt + ack_delay < latest_rtt):
  adjusted_rtt = latest_rtt - ack_delay
smoothed_rtt = 7/8 * smoothed_rtt + 1/8 * adjusted_rtt
rttvar_sample = abs(smoothed_rtt - adjusted_rtt)
rttvar = 3/4 * rttvar + 1/4 * rttvar_sample
```

6. Loss Detection

QUIC senders use acknowledgements to detect lost packets, and a probe time out (see Section 6.2) to ensure acknowledgements are received. 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 discarding the frame. For more information, see Section 13.3 of [QUIC-TRANSPORT].

Loss detection is separate per packet number space, unlike RTT measurement and congestion control, because RTT and congestion control are properties of the path, whereas loss detection also relies upon key availability.

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.

* The packet was sent kPacketThreshold packets before 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, and 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 can detect spurious retransmissions and increase the reordering threshold in packets or time to reduce future spurious retransmissions and loss events. Implementations with adaptive time thresholds 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]). In order to remain similar to TCP, implementations SHOULD NOT use a packet threshold less than 3; see [RFC5681].

Some networks may exhibit higher degrees of packet reordering, causing a sender to detect spurious losses. Additionally, packet reordering could be more common with QUIC than TCP, because network elements that could observe and reorder TCP packets cannot do that for QUIC, because QUIC packet numbers are encrypted. Algorithms that increase the reordering threshold after spuriously detecting losses, such as RACK [RACK], have proven to be useful in TCP and are expected to be at least as useful in QUIC.

6.1.2. Time Threshold

Once a later packet within the same packet number space has been acknowledged, an endpoint SHOULD declare an earlier packet lost if it was sent a threshold amount of time in the past. To avoid declaring packets as lost too early, this time threshold MUST be set to at least the local timer granularity, as indicated by the kGranularity constant. The time threshold is:

```
max(kTimeThreshold * max(smoothed_rtt, latest_rtt), kGranularity)
```

If packets sent prior to the largest acknowledged packet cannot yet be declared lost, then a timer SHOULD be set for the remaining time.

Using max(smoothed_rtt, latest_rtt) protects from the two following cases:

* the latest RTT sample is lower than the smoothed RTT, perhaps due to reordering where the acknowledgement encountered a shorter path;

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

The RECOMMENDED time threshold (kTimeThreshold), expressed as a round-trip time multiplier, is 9/8. The RECOMMENDED value of the timer granularity (kGranularity) is 1ms.

Note: TCP's RACK ([RACK]) specifies a slightly larger threshold, equivalent to 5/4, for a similar purpose. Experience with QUIC shows that 9/8 works well.

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

6.2. Probe Timeout

A Probe Timeout (PTO) triggers sending one or two probe datagrams when ack-eliciting packets are not acknowledged within the expected period of time or the server may not have validated the client's address. A PTO enables a connection to recover from loss of tail packets or acknowledgements.

As with loss detection, the probe timeout is per packet number space. That is, a PTO value is computed per packet number space.

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.

The PTO algorithm used in QUIC implements the reliability functions of Tail Loss Probe [RACK], RTO [RFC5681], and F-RTO algorithms for TCP [RFC5682]. The timeout computation is based on TCP's retransmission timeout period [RFC6298].

6.2.1. 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
```

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 variation in the estimate (4*rttvar), and max_ack_delay, to account for the maximum time by which a receiver might delay sending an acknowledgement.

When the PTO is armed for Initial or Handshake packet number spaces, the max_ack_delay in the PTO period computation is set to 0, since the peer is expected to not delay these packets intentionally; see 13.2.1 of [QUIC-TRANSPORT].

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

When ack-eliciting packets in multiple packet number spaces are in flight, the timer MUST be set to the earlier value of the Initial and Handshake packet number spaces.

An endpoint MUST NOT set its PTO timer for the application data packet number space until the handshake is confirmed. Doing so prevents the endpoint from retransmitting information in packets when either the peer does not yet have the keys to process them or the endpoint does not yet have the keys to process their acknowledgements. For example, this can happen when a client sends 0-RTT packets to the server; it does so without knowing whether the server will be able to decrypt them. Similarly, this can happen when a server sends 1-RTT packets before confirming that the client has verified the server's certificate and can therefore read these 1-RTT packets.

A sender SHOULD restart its PTO timer every time an ack-eliciting packet is sent or acknowledged, when the handshake is confirmed (Section 4.1.2 of [QUIC-TLS]), or when Initial or Handshake keys are discarded (Section 4.9 of [QUIC-TLS]). This ensures the PTO is always set based on the latest estimate of the round-trip time and for the correct packet across packet number spaces.

When a PTO timer expires, the PTO backoff MUST be increased, resulting in the PTO period being set to twice its current value. The PTO backoff factor is reset when an acknowledgement is received, except in the following case. A server might take longer to respond to packets during the handshake than otherwise. To protect such a server from repeated client probes, the PTO backoff is not reset at a client that is not yet certain that the server has finished validating the client's address. That is, a client does not reset the PTO backoff factor on receiving acknowledgements in Initial packets.

This exponential reduction in the sender's rate is important because consecutive PTOs might be caused by loss of packets or acknowledgements due to severe congestion. Even when there are ack-eliciting packets in-flight in multiple packet number spaces, the exponential increase in probe timeout occurs across all spaces to prevent excess load on the network. For example, a timeout in the Initial packet number space doubles the length of the timeout in the Handshake packet number space.

The total length of time over which consecutive PTOs expire is limited by the idle timeout.

The PTO timer MUST NOT be set if a timer is set for time threshold loss detection; see Section 6.1.2. A timer that is set for time threshold loss detection will expire earlier than the PTO timer in most cases and is less likely to spuriously retransmit data.

6.2.2. Handshakes and New Paths

Resumed connections over the same network MAY use the previous connection's final smoothed RTT value as the resumed connection's initial RTT. When no previous RTT is available, the initial RTT SHOULD be set to 333ms, resulting in a 1 second initial timeout, as recommended in [RFC6298].

A connection MAY use the delay between sending a PATH_CHALLENGE and receiving a PATH_RESPONSE to set the initial RTT (see kInitialRtt in Appendix A.2) for a new path, but the delay SHOULD NOT be considered an RTT sample.

Initial packets and Handshake packets could be never acknowledged, but they are removed from bytes in flight when the Initial and Handshake keys are discarded, as described below in Section 6.4. When Initial or Handshake keys are discarded, the PTO and loss detection timers MUST be reset, because discarding keys indicates forward progress and the loss detection timer might have been set for a now discarded packet number space.

6.2.2.1. Before Address Validation

Until the server has validated the client's address on the path, the amount of data it can send is limited to three times the amount of data received, as specified in Section 8.1 of [QUIC-TRANSPORT]. If no additional data can be sent, the server's PTO timer MUST NOT be armed until datagrams have been received from the client, because packets sent on PTO count against the anti-amplification limit. Note that the server could fail to validate the client's address even if 0-RTT is accepted.

Since the server could be blocked until more datagrams are received from the client, it is the client's responsibility to send packets to unblock the server until it is certain that the server has finished its address validation (see Section 8 of [QUIC-TRANSPORT]). That is, the client MUST set the probe timer if the client has not received an acknowledgement for one of its Handshake packets and the handshake is not confirmed (see Section 4.1.2 of [QUIC-TLS]), even if there are no packets in flight. When the PTO fires, the client MUST send a Handshake packet if it has Handshake keys, otherwise it MUST send an Initial packet in a UDP datagram with a payload of at least 1200 bytes.

6.2.3. Speeding Up Handshake Completion

When a server receives an Initial packet containing duplicate CRYPTO data, it can assume the client did not receive all of the server's CRYPTO data sent in Initial packets, or the client's estimated RTT is too small. When a client receives Handshake or 1-RTT packets prior to obtaining Handshake keys, it may assume some or all of the server's Initial packets were lost.

To speed up handshake completion under these conditions, an endpoint MAY, for a limited number of occasions per each connection, send a packet containing unacknowledged CRYPTO data earlier than the PTO expiry, subject to the address validation limits in Section 8.1 of [QUIC-TRANSPORT]. Doing so at most once for each connection is adequate to quickly recover from a single packet loss. Endpoints that do not cease retransmitting packets in response to unauthenticated data risk creating an infinite exchange of packets.

Endpoints can also use coalesced packets (see Section 12.2 of [QUIC-TRANSPORT]) to ensure that each datagram elicits at least one acknowledgement. For example, a client can coalesce an Initial packet containing PING and PADDING frames with a 0-RTT data packet and a server can coalesce an Initial packet containing a PING frame with one or more packets in its first flight.

6.2.4. Sending Probe Packets

When a PTO timer expires, a sender MUST send at least one ack-eliciting packet in the packet number space as a probe. An endpoint MAY send up to two full-sized datagrams containing ack-eliciting packets, to avoid an expensive consecutive PTO expiration due to a single lost datagram or transmit data from multiple packet number spaces. All probe packets sent on a PTO MUST be ack-eliciting.

In addition to sending data in the packet number space for which the timer expired, the sender SHOULD send ack-eliciting packets from other packet number spaces with in-flight data, coalescing packets if possible. This is particularly valuable when the server has both Initial and Handshake data in-flight or the client has both Handshake and Application Data in-flight, because the peer might only have receive keys for one of the two packet number spaces.

If the sender wants to elicit a faster acknowledgement on PTO, it can skip a packet number to eliminate the acknowledgment delay.

When the PTO timer expires, an ack-eliciting packet MUST be sent. An endpoint SHOULD include new data in this packet. Previously sent data MAY be sent if no new data can be sent. Implementations MAY use alternative strategies for determining the content of probe packets, including sending new or retransmitted data based on the application's priorities.

It is possible the sender has no new or previously-sent data to send. As an example, consider the following sequence of events: new application data is sent in a STREAM frame, deemed lost, then retransmitted in a new packet, and then the original transmission is acknowledged. When there is no data to send, the sender SHOULD send a PING or other ack-eliciting frame in a single packet, re-arming the PTO timer.

Alternatively, instead of sending an ack-eliciting packet, the sender MAY mark any packets still in flight as lost. Doing so avoids sending an additional packet, but increases the risk that loss is declared too aggressively, resulting in an unnecessary rate reduction by the congestion controller.

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.

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.

6.3. Handling Retry Packets

A Retry packet causes a client to send another Initial packet, effectively restarting the connection process. A Retry packet indicates that the Initial was received, but not processed. A Retry packet cannot be treated as an acknowledgment, because it does not indicate that a packet was processed or specify the packet number.

Clients that receive a Retry packet reset congestion control and loss recovery state, including resetting any pending timers. Other connection state, in particular cryptographic handshake messages, is retained; see Section 17.2.5 of [QUIC-TRANSPORT].

The client MAY 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 in place of its default for the initial RTT estimate.

6.4. 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. However, Initial secrets are discarded as soon as handshake keys are proven to be available to both client and server; see Section 4.9.1 of [QUIC-TLS].

7. Congestion Control

This document specifies a sender-side congestion controller for QUIC similar to TCP NewReno ([RFC6582]).

The signals QUIC provides for congestion control are generic and are designed to support different sender-side algorithms. A sender can unilaterally choose a different algorithm to use, such as Cubic ([RFC8312]).

If a sender uses a different controller than that specified in this document, the chosen controller MUST conform to the congestion control guidelines specified in Section 3.1 of [RFC8085].

Similar to TCP, packets containing only ACK frames do not count towards bytes in flight and are not congestion controlled. Unlike TCP, QUIC can detect the loss of these packets and MAY use that information to adjust the congestion controller or the rate of ACK-only packets being sent, but this document does not describe a mechanism for doing so.

The algorithm in this document specifies and uses the controller's congestion window in bytes.

An endpoint MUST NOT send a packet if it would cause bytes_in_flight (see Appendix B.2) to be larger than the congestion window, unless the packet is sent on a PTO timer expiration (see Section 6.2) or when entering recovery (see Section 7.3.2).

7.1. Explicit Congestion Notification

If a path has been validated to support ECN ([RFC3168], [RFC8311]), QUIC treats a Congestion Experienced (CE) codepoint in the IP header as a signal of congestion. This document specifies an endpoint's response when the peer-reported ECN-CE count increases; see Section 13.4.2 of [QUIC-TRANSPORT].

7.2. Initial and Minimum Congestion Window

QUIC begins every connection in slow start with the congestion window set to an initial value. Endpoints SHOULD use an initial congestion window of 10 times the maximum datagram size (max_datagram_size), limited to the larger of 14720 bytes or twice the maximum datagram size. This follows the analysis and recommendations in [RFC6928], increasing the byte limit to account for the smaller 8 byte overhead of UDP compared to the 20 byte overhead for TCP.

If the maximum datagram size changes during the connection, the initial congestion window SHOULD be recalculated with the new size. If the maximum datagram size is decreased in order to complete the handshake, the congestion window SHOULD be set to the new initial congestion window.

Prior to validating the client's address, the server can be further limited by the anti-amplification limit as specified in Section 8.1 of [QUIC-TRANSPORT]. Though the anti-amplification limit can prevent the congestion window from being fully utilized and therefore slow down the increase in congestion window, it does not directly affect the congestion window.

The minimum congestion window is the smallest value the congestion window can decrease to as a response to loss, increase in the peer-reported ECN-CE count, or persistent congestion. The RECOMMENDED value is 2 * max_datagram_size.

7.3. Congestion Control States

The NewReno congestion controller described in this document has three distinct states, as shown in Figure 1.

```
                 New Path or      +------------+
            persistent congestion |   Slow     |
        (O)---------------------->|   Start    |
                                  +------------+
                                        |
                                Loss or |
                        ECN-CE increase |
                                        v
    +------------+     Loss or      +------------+
    | Congestion |  ECN-CE increase |  Recovery  |
    | Avoidance  |----------------->|   Period   |
    +------------+                  +------------+
              ^                            |
              |                            |
              +----------------------------+
                 Acknowledgment of packet
                   sent during recovery
```

Figure 1: Congestion Control States and Transitions

These states and the transitions between them are described in subsequent sections.

7.3.1. Slow Start

A NewReno sender is in slow start any time the congestion window is below the slow start threshold. A sender begins in slow start because the slow start threshold is initialized to an infinite value.

While a sender is in slow start, the congestion window increases by the number of bytes acknowledged when each acknowledgment is processed. This results in exponential growth of the congestion window.

The sender MUST exit slow start and enter a recovery period when a packet is lost or when the ECN-CE count reported by its peer increases.

A sender re-enters slow start any time the congestion window is less than the slow start threshold, which only occurs after persistent congestion is declared.

7.3.2. Recovery

A NewReno sender enters a recovery period when it detects the loss of a packet or the ECN-CE count reported by its peer increases. A sender that is already in a recovery period stays in it and does not re-enter it.

On entering a recovery period, a sender MUST set the slow start threshold to half the value of the congestion window when loss is detected. The congestion window MUST be set to the reduced value of the slow start threshold before exiting the recovery period.

Implementations MAY reduce the congestion window immediately upon entering a recovery period or use other mechanisms, such as Proportional Rate Reduction ([PRR]), to reduce the congestion window more gradually. If the congestion window is reduced immediately, a single packet can be sent prior to reduction. This speeds up loss recovery if the data in the lost packet is retransmitted and is similar to TCP as described in Section 5 of [RFC6675].

The recovery period aims to limit congestion window reduction to once per round trip. Therefore during a recovery period, the congestion window does not change in response to new losses or increases in the ECN-CE count.

A recovery period ends and the sender enters congestion avoidance when a packet sent during the recovery period is acknowledged. This is slightly different from TCP's definition of recovery, which ends when the lost segment that started recovery is acknowledged ([RFC5681]).

7.3.3. Congestion Avoidance

A NewReno sender is in congestion avoidance any time the congestion window is at or above the slow start threshold and not in a recovery period.

A sender in congestion avoidance uses an Additive Increase Multiplicative Decrease (AIMD) approach that MUST limit the increase to the congestion window to at most one maximum datagram size for each congestion window that is acknowledged.

The sender exits congestion avoidance and enters a recovery period when a packet is lost or when the ECN-CE count reported by its peer increases.

7.4. Ignoring Loss of Undecryptable Packets

During the handshake, some packet protection keys might not be available when a packet arrives and the receiver can choose to drop the packet. 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 have arrived before the peer had packet protection keys to process those packets.

Endpoints MUST NOT ignore the loss of packets that were sent after the earliest acknowledged packet in a given packet number space.

7.5. 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.

7.6. Persistent Congestion

When a sender establishes loss of all in-flight packets sent over a long enough duration, the network is considered to be experiencing persistent congestion.

7.6.1. Duration

The persistent congestion duration is computed as follows:

```
(smoothed_rtt + max(4*rttvar, kGranularity) + max_ack_delay) *
    kPersistentCongestionThreshold
```

Unlike the PTO computation in Section 6.2, this duration includes the max_ack_delay irrespective of the packet number spaces in which losses are established.

This duration allows a sender to send as many packets before establishing persistent congestion, including some in response to PTO expiration, as TCP does with Tail Loss Probes ([RACK]) and a Retransmission Timeout ([RFC5681]).

Larger values of kPersistentCongestionThreshold cause the sender to become less responsive to persistent congestion in the network, which can result in aggressive sending into a congested network. Too small a value can result in a sender declaring persistent congestion unnecessarily, resulting in reduced throughput for the sender.

The RECOMMENDED value for kPersistentCongestionThreshold is 3, which results in behavior that is approximately equivalent to a TCP sender declaring an RTO after two TLPs.

This design does not use consecutive PTO events to establish persistent congestion, since application patterns impact PTO expirations. For example, a sender that sends small amounts of data with silence periods between them restarts the PTO timer every time it sends, potentially preventing the PTO timer from expiring for a long period of time, even when no acknowledgments are being received. The use of a duration enables a sender to establish persistent congestion without depending on PTO expiration.

7.6.2. Establishing Persistent Congestion

A sender establishes persistent congestion on receiving an acknowledgement if at least two ack-eliciting packets are declared lost, and:

* all packets, across all packet number spaces, sent between these two send times are declared lost;

* the duration between the send times of these two packets exceeds the persistent congestion duration (Section 7.6.1); and

* a prior RTT sample existed when both packets were sent.

The persistent congestion period SHOULD NOT start until there is at least one RTT sample. Before the first RTT sample, a sender arms its PTO timer based on the initial RTT (Section 6.2.2), which could be substantially larger than the actual RTT. Requiring a prior RTT sample prevents a sender from establishing persistent congestion with potentially too few probes.

Since network congestion is not affected by packet number spaces, persistent congestion SHOULD consider packets sent across packet number spaces. A sender that does not have state for all packet number spaces or an implementation that cannot compare send times across packet number spaces MAY use state for just the packet number space that was acknowledged.

When persistent congestion is declared, the sender's congestion window MUST be reduced to the minimum congestion window (kMinimumWindow), similar to a TCP sender's response on an RTO ([RFC5681]).

7.6.3. Example

The following example illustrates how a sender might establish persistent congestion. Assume:

```
smoothed_rtt + max(4*rttvar, kGranularity) + max_ack_delay = 2
kPersistentCongestionThreshold = 3
```

Consider the following sequence of events:

Time	Action
t=0	Send packet #1 (app data)
t=1	Send packet #2 (app data)
t=1.2	Recv acknowledgement of #1
t=2	Send packet #3 (app data)
t=3	Send packet #4 (app data)
t=4	Send packet #5 (app data)
t=5	Send packet #6 (app data)
t=6	Send packet #7 (app data)
t=8	Send packet #8 (PTO 1)
t=12	Send packet #9 (PTO 2)
t=12.2	Recv acknowledgement of #9

Table 1

Packets 2 through 8 are declared lost when the acknowledgement for packet 9 is received at t = 12.2.

The congestion period is calculated as the time between the oldest and newest lost packets: 8 - 1 = 7. The persistent congestion duration is: 2 * 3 = 6. Because the threshold was reached and because none of the packets between the oldest and the newest lost packets were acknowledged, the network is considered to have experienced persistent congestion.

While this example shows PTO expiration, they are not required for persistent congestion to be established.

7.7. Pacing

A sender SHOULD pace sending of all in-flight packets based on input from the congestion controller.

Sending multiple packets into the network without any delay between them creates a packet burst that might cause short-term congestion and losses. Senders MUST either use pacing or limit such bursts. Senders SHOULD limit bursts to the initial congestion window; see Section 7.2. A sender with knowledge that the network path to the receiver can absorb larger bursts MAY use a higher limit.

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.

Endpoints can implement pacing as they choose. A perfectly paced sender spreads packets exactly evenly over time. For a window-based congestion controller, such as the one in this document, that rate can be computed by averaging the congestion window over the round-trip time. Expressed as a rate in bytes:

```
rate = N * congestion_window / smoothed_rtt
```

Or, expressed as an inter-packet interval:

```
interval = smoothed_rtt * packet_size / congestion_window / N
```

Using a value for "N" that is small, but at least 1 (for example, 1.25) ensures that variations in round-trip time do not result in under-utilization of the congestion window.

Practical considerations, such as packetization, scheduling delays, and computational efficiency, can cause a sender to deviate from this rate over time periods that are much shorter than a round-trip time.

One possible implementation strategy for pacing uses a leaky bucket algorithm, where the capacity of the "bucket" is limited to the maximum burst size and the rate the "bucket" fills is determined by the above function.

7.8. Under-utilizing the Congestion Window

When bytes in flight is smaller than the congestion window and sending is not pacing limited, the congestion window is under-utilized. When this occurs, the congestion window SHOULD NOT be increased in either slow start or congestion avoidance. This can happen due to insufficient application data or flow control limits.

A sender that paces packets (see Section 7.7) 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.

A sender MAY implement alternative mechanisms to update its congestion window after periods of under-utilization, such as those proposed for TCP in [RFC7661].

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. To reduce leaked information, endpoints can bundle acknowledgments with other frames, or they can use PADDING frames at a potential cost to performance.

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 can detect suppression of reports by marking occasional packets that it sends with an ECN-CE marking. If a packet sent with an ECN-CE marking is not reported as having been CE marked when the packet is acknowledged, then the sender can disable ECN for that path by not setting ECT codepoints in subsequent packets sent on that path [RFC3168].

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. Congestion controllers respond to reports of ECN-CE by reducing their rate, but the response may vary. Markings can be treated as equivalent to loss ([RFC3168]), but other responses can be specified, such as ([RFC8511]) or ([RFC8311]).

9. IANA Considerations

This document has no IANA actions.

10. References

10.1. Normative References

[QUIC-TLS] Thomson, M., Ed. and S. Turner, Ed., "Using TLS to Secure QUIC", Work in Progress, Internet-Draft, draft-ietf-quic-tls-32, 20 October 2020, <https://tools.ietf.org/html/draft-ietf-quic-tls-32>.

[QUIC-TRANSPORT] Iyengar, J., Ed. and M. Thomson, Ed., "QUIC: A UDP-Based Multiplexed and Secure Transport", Work in Progress, Internet-Draft, draft-ietf-quic-transport-32, 20 October 2020, <https://tools.ietf.org/html/draft-ietf-quic-transport-32>.

[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>.

[RFC3168] Ramakrishnan, K., Floyd, S., and D. Black, "The Addition of Explicit Congestion Notification (ECN) to IP", RFC 3168, DOI 10.17487/RFC3168, September 2001, <https://www.rfc-editor.org/info/rfc3168>.

[RFC8085] Eggert, L., Fairhurst, G., and G. Shepherd, "UDP Usage Guidelines", BCP 145, RFC 8085, DOI 10.17487/RFC8085, March 2017, <https://www.rfc-editor.org/info/rfc8085>.

[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>.

10.2. Informative References

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

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

[RACK] Cheng, Y., Cardwell, N., Dukkipati, N., and P. Jha, "The RACK-TLP loss detection algorithm for TCP", Work in Progress, Internet-Draft, draft-ietf-tcpm-rack-11, 30 September 2020, <http://www.ietf.org/internet-drafts/draft-ietf-tcpm-rack-11.txt>.

[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>.

[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>.

[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>.

[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>.

[RFC8511] Khademi, N., Welzl, M., Armitage, G., and G. Fairhurst, "TCP Alternative Backoff with ECN (ABE)", RFC 8511, DOI 10.17487/RFC8511, December 2018, <https://www.rfc-editor.org/info/rfc8511>.

Appendix A. Loss Recovery Pseudocode

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

The pseudocode segments in this section are licensed as Code Components; see the copyright notice.

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, the endpoint can still maintain state for it for an amount of time to allow for packet reordering; see Section 13.3 of [QUIC-TRANSPORT]. This enables a sender to detect 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 max_ack_delay.

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

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.

kPacketThreshold: Maximum reordering in packets before packet threshold loss detection considers a packet lost. The value recommended in Section 6.1.1 is 3.

kTimeThreshold: Maximum reordering in time before time threshold loss detection considers a packet lost. Specified as an RTT multiplier. The value recommended in Section 6.1.2 is 9/8.

kGranularity: Timer granularity. This is a system-dependent value, and Section 6.1.2 recommends a value of 1ms.

kInitialRtt: The RTT used before an RTT sample is taken. The value recommended in Section 6.2.2 is 333ms.

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.

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 Section 5.3.

rttvar: The RTT variation, computed as described in Section 5.3.

min_rtt: The minimum RTT seen in the connection, ignoring acknowledgment delay, as described in Section 5.2.

first_rtt_sample: The time that the first RTT sample was obtained.

max_ack_delay: The maximum amount of time by which the receiver intends to delay acknowledgments for packets in the Application Data packet number space, as defined by the eponymous transport parameter (Section 18.2 of [QUIC-TRANSPORT]). Note that the actual ack_delay in a received ACK frame may be larger due to late timers, reordering, or loss.

loss_detection_timer: Multi-modal timer used for loss detection.

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

time_of_last_ack_eliciting_packet[kPacketNumberSpace]: The time the most recent ack-eliciting packet was sent.

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

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()
pto_count = 0
latest_rtt = 0
smoothed_rtt = kInitialRtt
rttvar = kInitialRtt / 2
min_rtt = 0
first_rtt_sample = 0
for pn_space in [ Initial, Handshake, ApplicationData ]:
  largest_acked_packet[pn_space] = infinite
  time_of_last_ack_eliciting_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, 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 (ack_eliciting):
      time_of_last_ack_eliciting_packet[pn_space] = now()
    OnPacketSentCC(sent_bytes)
    sent_packets[pn_space][packet_number].sent_bytes =
      sent_bytes
    SetLossDetectionTimer()
```

A.6. On Receiving a Datagram

When a server is blocked by anti-amplification limits, receiving a datagram unblocks it, even if none of the packets in the datagram are successfully processed. In such a case, the PTO timer will need to be re-armed.

Pseudocode for OnDatagramReceived follows:

```
OnDatagramReceived(datagram):
  // If this datagram unblocks the server, arm the
  // PTO timer to avoid deadlock.
  if (server was at anti-amplification limit):
    SetLossDetectionTimer()
```

A.7. On Receiving an Acknowledgment

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

Pseudocode for OnAckReceived and UpdateRtt follow:

```
IncludesAckEliciting(packets):
  for packet in packets:
    if (packet.ack_eliciting):
      return true
  return false

OnAckReceived(ack, pn_space):
  if (largest_acked_packet[pn_space] == infinite):
    largest_acked_packet[pn_space] = ack.largest_acked
  else:
    largest_acked_packet[pn_space] =
        max(largest_acked_packet[pn_space], ack.largest_acked)

  // DetectAndRemoveAckedPackets finds packets that are newly
  // acknowledged and removes them from sent_packets.
  newly_acked_packets =
      DetectAndRemoveAckedPackets(ack, pn_space)
  // Nothing to do if there are no newly acked packets.
  if (newly_acked_packets.empty()):
    return

  // Update the RTT if the largest acknowledged is newly acked
  // and at least one ack-eliciting was newly acked.
  if (newly_acked_packets.largest().packet_number ==
          ack.largest_acked &&
      IncludesAckEliciting(newly_acked_packets)):
```

```
      latest_rtt =
        now() - newly_acked_packets.largest().time_sent
      UpdateRtt(ack.ack_delay)

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

    lost_packets = DetectAndRemoveLostPackets(pn_space)
    if (!lost_packets.empty()):
      OnPacketsLost(lost_packets)
    OnPacketsAcked(newly_acked_packets)

    // Reset pto_count unless the client is unsure if
    // the server has validated the client's address.
    if (PeerCompletedAddressValidation()):
      pto_count = 0
    SetLossDetectionTimer()


  UpdateRtt(ack_delay):
    if (first_rtt_sample == 0):
      min_rtt = latest_rtt
      smoothed_rtt = latest_rtt
      rttvar = latest_rtt / 2
      first_rtt_sample = now()
      return

    // min_rtt ignores acknowledgment delay.
    min_rtt = min(min_rtt, latest_rtt)
    // Limit ack_delay by max_ack_delay after handshake
    // confirmation. Note that ack_delay is 0 for
    // acknowledgements of Initial and Handshake packets.
    if (handshake confirmed):
      ack_delay = min(ack_delay, max_ack_delay)

    // Adjust for acknowledgment delay if plausible.
    adjusted_rtt = latest_rtt
    if (latest_rtt > min_rtt + ack_delay):
      adjusted_rtt = latest_rtt - ack_delay

    rttvar = 3/4 * rttvar + 1/4 * abs(smoothed_rtt - adjusted_rtt)
    smoothed_rtt = 7/8 * smoothed_rtt + 1/8 * adjusted_rtt
```

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 fire immediately.

Pseudocode for SetLossDetectionTimer follows:

```
GetLossTimeAndSpace():
  time = loss_time[Initial]
  space = Initial
  for pn_space in [ Handshake, ApplicationData ]:
    if (time == 0 || loss_time[pn_space] < time):
      time = loss_time[pn_space];
      space = pn_space
  return time, space

GetPtoTimeAndSpace():
  duration = (smoothed_rtt + max(4 * rttvar, kGranularity))
      * (2 ^ pto_count)
  // Arm PTO from now when there are no inflight packets.
  if (no in-flight packets):
    assert(!PeerCompletedAddressValidation())
    if (has handshake keys):
      return (now() + duration), Handshake
    else:
      return (now() + duration), Initial
  pto_timeout = infinite
  pto_space = Initial
  for space in [ Initial, Handshake, ApplicationData ]:
    if (no in-flight packets in space):
        continue;
    if (space == ApplicationData):
      // Skip Application Data until handshake confirmed.
      if (handshake is not confirmed):
        return pto_timeout, pto_space
      // Include max_ack_delay and backoff for Application Data.
      duration += max_ack_delay * (2 ^ pto_count)

    t = time_of_last_ack_eliciting_packet[space] + duration
    if (t < pto_timeout):
      pto_timeout = t
```

```
        pto_space = space
    return pto_timeout, pto_space

  PeerCompletedAddressValidation():
    // Assume clients validate the server's address implicitly.
    if (endpoint is server):
      return true
    // Servers complete address validation when a
    // protected packet is received.
    return has received Handshake ACK ||
         handshake confirmed

  SetLossDetectionTimer():
    earliest_loss_time, _ = GetLossTimeAndSpace()
    if (earliest_loss_time != 0):
      // Time threshold loss detection.
      loss_detection_timer.update(earliest_loss_time)
      return

    if (server is at anti-amplification limit):
      // The server's timer is not set if nothing can be sent.
      loss_detection_timer.cancel()
      return

    if (no ack-eliciting packets in flight &&
        PeerCompletedAddressValidation()):
      // There is nothing to detect lost, so no timer is set.
      // However, the client needs to arm the timer if the
      // server might be blocked by the anti-amplification limit.
      loss_detection_timer.cancel()
      return

    // Determine which PN space to arm PTO for.
    timeout, _ = GetPtoTimeAndSpace()
    loss_detection_timer.update(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():
  earliest_loss_time, pn_space = GetLossTimeAndSpace()
  if (earliest_loss_time != 0):
    // Time threshold loss Detection
    lost_packets = DetectAndRemoveLostPackets(pn_space)
    assert(!lost_packets.empty())
    OnPacketsLost(lost_packets)
    SetLossDetectionTimer()
    return

  if (bytes_in_flight > 0):
    // PTO. Send new data if available, else retransmit old data.
    // If neither is available, send a single PING frame.
    _, pn_space = GetPtoTimeAndSpace()
    SendOneOrTwoAckElicitingPackets(pn_space)
  else:
    assert(!PeerCompletedAddressValidation())
    // Client sends an anti-deadlock packet: Initial is padded
    // to earn more anti-amplification credit,
    // a Handshake packet proves address ownership.
    if (has Handshake keys):
      SendOneAckElicitingHandshakePacket()
    else:
      SendOneAckElicitingPaddedInitialPacket()

  pto_count++
  SetLossDetectionTimer()
```

A.10. Detecting Lost Packets

DetectAndRemoveLostPackets is called every time an ACK is received or the time threshold loss detection timer expires. This function operates on the sent_packets for that packet number space and returns a list of packets newly detected as lost.

Pseudocode for DetectAndRemoveLostPackets follows:

```
DetectAndRemoveLostPackets(pn_space):
  assert(largest_acked_packet[pn_space] != infinite)
  loss_time[pn_space] = 0
  lost_packets = {}
  loss_delay = kTimeThreshold * max(latest_rtt, smoothed_rtt)

  // Minimum time of kGranularity before packets are deemed lost.
  loss_delay = max(loss_delay, kGranularity)

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

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

    // Mark packet as lost, or set time when it should be marked.
    // Note: The use of kPacketThreshold here assumes that there
    // were no sender-induced gaps in the packet number space.
    if (unacked.time_sent <= lost_send_time ||
        largest_acked_packet[pn_space] >=
          unacked.packet_number + kPacketThreshold):
      sent_packets[pn_space].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)
  return lost_packets
```

A.11. Upon Dropping Initial or Handshake Keys

When Initial or Handshake keys are discarded, packets from the space are discarded and loss detection state is updated.

Pseudocode for OnPacketNumberSpaceDiscarded follows:

```
OnPacketNumberSpaceDiscarded(pn_space):
  assert(pn_space != ApplicationData)
  RemoveFromBytesInFlight(sent_packets[pn_space])
  sent_packets[pn_space].clear()
  // Reset the loss detection and PTO timer
  time_of_last_ack_eliciting_packet[pn_space] = 0
  loss_time[pn_space] = 0
  pto_count = 0
  SetLossDetectionTimer()
```

Appendix B. Congestion Control Pseudocode

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

The pseudocode segments in this section are licensed as Code Components; see the copyright notice.

B.1. Constants of interest

Constants used in congestion control are based on a combination of RFCs, papers, and common practice.

kInitialWindow: Default limit on the initial bytes in flight as described in Section 7.2.

kMinimumWindow: Minimum congestion window in bytes as described in Section 7.2.

kLossReductionFactor: Reduction in congestion window when a new loss event is detected. Section 7 recommends a value is 0.5.

kPersistentCongestionThreshold: Period of time for persistent congestion to be established, specified as a PTO multiplier. Section 7.6 recommends a value of 3.

B.2. Variables of interest

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

max_datagram_size: The sender's current maximum payload size. Does not include UDP or IP overhead. The max datagram size is used for congestion window computations. An endpoint sets the value of this variable based on its Path Maximum Transmission Unit (PMTU; see Section 14.2 of [QUIC-TRANSPORT]), with a minimum value of 1200 bytes.

ecn_ce_counters[kPacketNumberSpace]: The highest value reported for the ECN-CE counter in the packet number space by the peer in an ACK frame. This value 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 acknowledged 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.

congestion_recovery_start_time: The time when QUIC first detects congestion due to loss or ECN, causing it to enter congestion recovery. When a packet sent after this time is acknowledged, QUIC exits congestion 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.

The congestion control pseudocode also accesses some of the variables from the loss recovery pseudocode.

B.3. Initialization

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

```
congestion_window = kInitialWindow
bytes_in_flight = 0
congestion_recovery_start_time = 0
ssthresh = infinite
for pn_space in [ Initial, Handshake, ApplicationData ]:
  ecn_ce_counters[pn_space] = 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 OnAckReceived and is supplied with the newly acked_packets from sent_packets.

In congestion avoidance, implementers that use an integer representation for congestion_window should be careful with division, and can use the alternative approach suggested in Section 2.1 of [RFC3465].

```
InCongestionRecovery(sent_time):
  return sent_time <= congestion_recovery_start_time

OnPacketsAcked(acked_packets):
  for acked_packet in acked_packets:
    OnPacketAcked(acked_packet)

OnPacketAcked(acked_packet):
  // Remove from bytes_in_flight.
  bytes_in_flight -= acked_packet.sent_bytes
  // Do not increase congestion_window if application
  // limited or flow control limited.
  if (IsAppOrFlowControlLimited())
    return
  // Do not increase congestion window in recovery period.
  if (InCongestionRecovery(acked_packet.time_sent)):
    return
  if (congestion_window < ssthresh):
    // Slow start.
    congestion_window += acked_packet.sent_bytes
  else:
    // Congestion avoidance.
    congestion_window +=
      max_datagram_size * acked_packet.sent_bytes
      / congestion_window
```

B.6. On New Congestion Event

Invoked from ProcessECN and OnPacketsLost when a new congestion event is detected. If not already in recovery, this starts a recovery period and reduces the slow start threshold and congestion window immediately.

```
OnCongestionEvent(sent_time):
  // No reaction if already in a recovery period.
  if (InCongestionRecovery(sent_time)):
    return

  // Enter recovery period.
  congestion_recovery_start_time = now()
  ssthresh = congestion_window * kLossReductionFactor
  congestion_window = max(ssthresh, kMinimumWindow)
  // A packet can be sent to speed up loss recovery.
  MaybeSendOnePacket()
```

B.7. Process ECN Information

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

```
ProcessECN(ack, pn_space):
  // If the ECN-CE counter reported by the peer has increased,
  // this could be a new congestion event.
  if (ack.ce_counter > ecn_ce_counters[pn_space]):
    ecn_ce_counters[pn_space] = ack.ce_counter
    sent_time = sent_packets[ack.largest_acked].time_sent
    OnCongestionEvent(sent_time)
```

B.8. On Packets Lost

Invoked when DetectAndRemoveLostPackets deems packets lost.

```
OnPacketsLost(lost_packets):
  // Remove lost packets from bytes_in_flight.
  for lost_packet in lost_packets:
    bytes_in_flight -= lost_packet.sent_bytes
  OnCongestionEvent(lost_packets.largest().time_sent)

  // Reset the congestion window if the loss of these
  // packets indicates persistent congestion.
  // Only consider packets sent after getting an RTT sample.
  if (first_rtt_sample == 0):
    return
  pc_lost = {}
  for lost in lost_packets:
    if lost.time_sent > first_rtt_sample:
      pc_lost.insert(lost)
  if (InPersistentCongestion(pc_lost)):
    congestion_window = kMinimumWindow
    congestion_recovery_start_time = 0
```

B.9. Removing Discarded Packets From Bytes In Flight

When Initial or Handshake keys are discarded, packets sent in that space no longer count toward bytes in flight.

Pseudocode for RemoveFromBytesInFlight follows:

```
RemoveFromBytesInFlight(discarded_packets):
  // Remove any unacknowledged packets from flight.
  foreach packet in discarded_packets:
    if packet.in_flight
      bytes_in_flight -= size
```

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-31

* Limit the number of Initial packets sent in response to unauthenticated packets (#4183, #4188)

C.2. Since draft-ietf-quic-recovery-30

Editorial changes only.

C.3. Since draft-ietf-quic-recovery-29

* Allow caching of packets that can't be decrypted, by allowing the reported acknowledgment delay to exceed max_ack_delay prior to confirming the handshake (#3821, #3980, #4035, #3874)

* Persistent congestion cannot include packets sent before the first RTT sample for the path (#3875, #3889)

* Recommend reset of min_rtt in persistent congestion (#3927, #3975)

* Persistent congestion is independent of packet number space (#3939, #3961)

* Only limit bursts to the initial window without information about the path (#3892, #3936)

* Add normative requirements for increasing and reducing the congestion window (#3944, #3978, #3997, #3998)

C.4. Since draft-ietf-quic-recovery-28

* Refactored pseudocode to correct PTO calculation (#3564, #3674, #3681)

C.5. Since draft-ietf-quic-recovery-27

* Added recommendations for speeding up handshake under some loss conditions (#3078, #3080)

* PTO count is reset when handshake progress is made (#3272, #3415)

* PTO count is not reset by a client when the server might be awaiting address validation (#3546, #3551)

* Recommend repairing losses immediately after entering the recovery period (#3335, #3443)

* Clarified what loss conditions can be ignored during the handshake (#3456, #3450)

* Allow, but don't recommend, using RTT from previous connection to seed RTT (#3464, #3496)

* Recommend use of adaptive loss detection thresholds (#3571, #3572)

C.6. Since draft-ietf-quic-recovery-26

No changes.

C.7. Since draft-ietf-quic-recovery-25

No significant changes.

C.8. Since draft-ietf-quic-recovery-24

* Require congestion control of some sort (#3247, #3244, #3248)

* Set a minimum reordering threshold (#3256, #3240)

* PTO is specific to a packet number space (#3067, #3074, #3066)

C.9. Since draft-ietf-quic-recovery-23

* Define under-utilizing the congestion window (#2630, #2686, #2675)

* PTO MUST send data if possible (#3056, #3057)

* Connection Close is not ack-eliciting (#3097, #3098)

* MUST limit bursts to the initial congestion window (#3160)

* Define the current max_datagram_size for congestion control (#3041, #3167)

C.10. Since draft-ietf-quic-recovery-22

* PTO should always send an ack-eliciting packet (#2895)

* Unify the Handshake Timer with the PTO timer (#2648, #2658, #2886)

* Move ACK generation text to transport draft (#1860, #2916)

C.11. Since draft-ietf-quic-recovery-21

* No changes

C.12. Since draft-ietf-quic-recovery-20

* Path validation can be used as initial RTT value (#2644, #2687)

* max_ack_delay transport parameter defaults to 0 (#2638, #2646)

* ACK delay only measures intentional delays induced by the implementation (#2596, #2786)

C.13. Since draft-ietf-quic-recovery-19

* Change kPersistentThreshold from an exponent to a multiplier (#2557)

* Send a PING if the PTO timer fires and there's nothing to send (#2624)

* Set loss delay to at least kGranularity (#2617)

* Merge application limited and sending after idle sections. Always limit burst size instead of requiring resetting CWND to initial CWND after idle (#2605)

* Rewrite RTT estimation, allow RTT samples where a newly acked packet is ack-eliciting but the largest_acked is not (#2592)

* Don't arm the handshake timer if there is no handshake data (#2590)

* Clarify that the time threshold loss alarm takes precedence over the crypto handshake timer (#2590, #2620)

* Change initial RTT to 500ms to align with RFC6298 (#2184)

C.14. 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.15. 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.16. 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 keys are available (#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.17. Since draft-ietf-quic-recovery-14

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

* Merge ACK and ACK_ECN (#1783)

C.18. 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.19. 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.20. Since draft-ietf-quic-recovery-11

No significant changes.

C.21. 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.22. Since draft-ietf-quic-recovery-09

No significant changes.

C.23. Since draft-ietf-quic-recovery-08

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

C.24. 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.25. Since draft-ietf-quic-recovery-06

No significant changes.

C.26. Since draft-ietf-quic-recovery-05

* Add more congestion control text (#776)

C.27. Since draft-ietf-quic-recovery-04

No significant changes.

C.28. Since draft-ietf-quic-recovery-03

No significant changes.

C.29. 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.30. 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.31. Since draft-ietf-quic-recovery-00

* Improved description of constants and ACK behavior

C.32. Since draft-iyengar-quic-loss-recovery-01

* Adopted as base for draft-ietf-quic-recovery

* Updated authors/editors list

* Added table of contents

Appendix D. Contributors

The IETF QUIC Working Group received an enormous amount of support from many people. The following people provided substantive contributions to this document:

* Alessandro Ghedini
* Benjamin Saunders
* Gorry Fairhurst
* å±±æ¬åå½¦ (Kazu Yamamoto)
* å¥¥ ä¸ç© (Kazuho Oku)
* Lars Eggert
* Magnus Westerlund
* Marten Seemann
* Martin Duke
* Martin Thomson
* Mirja KÃ¼hlewind
* Nick Banks
* Praveen Balasubramanian

Acknowledgments

Authors' Addresses

Jana Iyengar (editor)
Fastly

Email: jri.ietf@gmail.com

Ian Swett (editor)
Google

Email: ianswett@google.com