# QUIC Loss Detection and Congestion Control

## 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.

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

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

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

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

This Internet-Draft will expire on 25 July 2020.

## Copyright Notice

**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 congestion control and
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 are not ACK-only, and they are not acknowledged, declared lost, or abandoned along with old keys.

**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 ack delay and are called ack-eliciting packets.

## 3.  Design of the QUIC Transmission Machinery

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

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

QUIC packets can contain multiple frames of different types. The recovery mechanisms ensure that data and frames that need reliable delivery are acknowledged or declared lost and sent in new packets

as necessary. The types of frames contained in a packet affect
recovery and congestion control logic:

   *All packets are acknowledged, though packets that contain 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.

## 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.  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.

### 3.1.4.  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.

### 3.1.5.  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.

### 3.1.6.  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.  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).

## 4.1. 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 ACK
delays for only the largest acknowledged packet in an ACK frame.
While the reported ACK 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 4.3.

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
does not send an ACK frame on receiving only non-ack-eliciting
packets, so an ACK frame that is subsequently sent can include an
arbitrarily large Ack Delay field. 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.

## 4.2. Estimating min_rtt

min_rtt is the minimum RTT observed for a given network path.
min_rtt is set to the latest_rtt on the first RTT sample, and to the

lesser of min_rtt and latest_rtt on subsequent samples. In this document, min_rtt is used by loss detection to reject implausibly small rtt samples.

An endpoint uses only locally observed times in computing the min_rtt and does not adjust for ACK 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 4.3](#)), 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, the min_rtt will not adapt to it, allowing future RTT samples that are smaller than the new RTT be included in smoothed_rtt.

### 4.3.  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 path latency after adjusting RTT samples for ACK delays. For packets sent in the ApplicationData packet number space, a peer limits any delay in sending an acknowledgement for an ack-eliciting packet to no greater than the value it advertised in the max_ack_delay transport parameter. Consequently, when a peer reports an Ack Delay that is greater than its max_ack_delay, the delay is attributed to reasons out of the peer's control, such as scheduler latency at the peer or loss of previous ACK frames. Any delays beyond the peer's max_ack_delay are therefore considered effectively part of path delay and incorporated into the smoothed_rtt estimate.

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

  *MUST ignore the Ack Delay field of the ACK frame for packets sent in the Initial and Handshake packet number space.

  *MUST use the lesser of the value reported in Ack Delay field of the ACK frame and the peer's max_ack_delay transport parameter.

  *MUST NOT apply the adjustment if the resulting RTT sample is smaller than the min_rtt. This limits the underestimation that a misreporting peer can cause to the smoothed_rtt.

On the first RTT sample for a network path, the smoothed_rtt is set to the latest_rtt.

smoothed_rtt and rttvar are computed as follows, similar to
[RFC6298]. On the first RTT sample for a network path:

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

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

```
ack_delay = min(Ack Delay in ACK Frame, 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
```

## 5.  Loss Detection

QUIC senders use acknowledgements to detect lost packets, and a
probe time out Section 5.2 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 abandoning the frame. For more information, see Section 13.3 of
[QUIC-TRANSPORT].

## 5.1.  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.

   *Either its packet number is kPacketThreshold smaller than an
    acknowledged packet (Section 5.1.1), or it was sent long enough
    in the past (Section 5.1.2).

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 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.

### 5.1.1.  Packet Threshold

The RECOMMENDED initial value for the packet reordering threshold
(kPacketThreshold) is 3, based on best practices for TCP loss
detection [RFC5681] [RFC6675]. Implementations SHOULD NOT use a
packet threshold less than 3, to keep in line with TCP [RFC5681].

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.

### 5.1.2.  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 kGranularity. 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.

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.

## 5.2.  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 handshake has not been completed. 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. 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].

### 5.2.1.  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

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 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 is 0, as specified in 13.2.5 of [QUIC-TRANSPORT].

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

A sender computes its PTO timer every time an ack-eliciting packet is sent. When ack-eliciting packets are in-flight in multiple packet number spaces, the timer MUST be set for the packet number space with the earliest timeout, except for ApplicationData, which MUST be ignored until the handshake completes; see Section 4.1.1 of [QUIC-TLS]. Not arming the PTO for ApplicationData prioritizes completing the handshake and prevents the server from sending a 1-RTT packet on a PTO before before it has the keys to process a 1-RTT packet.

When a PTO timer expires, the PTO period MUST be set to twice its current value. 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 life of a connection that is experiencing consecutive PTOs is limited by the endpoint's idle timeout.

The probe timer is not set if the time threshold Section 5.1.2 loss detection timer is set. The time threshold loss detection timer is expected to both expire earlier than the PTO and be less likely to spuriously retransmit data.

## 5.3.  Handshakes and New Paths

The initial probe timeout for a new connection or new path SHOULD be set to twice the initial RTT. 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, the initial RTT SHOULD be set to 500ms, 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.

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 data can be sent, then the PTO alarm MUST NOT be armed until datagrams have been received from the client.

Since the server could be blocked until more packets 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 or 1-RTT packets.

Prior to handshake completion, when few to none RTT samples have been generated, it is possible that the probe timer expiration is due to an incorrect RTT estimate at the client. To allow the client to improve its RTT estimate, the new packet that it sends MUST be ack-eliciting. If Handshake keys are available to the client, it MUST send a Handshake packet, and otherwise it MUST send an Initial packet in a UDP datagram of at least 1200 bytes.

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.

### 5.3.1.  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, unless there is no data available to send. 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.

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.

When the PTO timer expires, and there is new or previously sent unacknowledged data, it MUST be sent.

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.

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 alternative 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.

### 5.3.2. 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 5.1.

### 5.4. 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.

### 5.5. Discarding Keys and Packet State

When packet protection keys are discarded (see Section 4.10 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 4.10.1 of [QUIC-TLS]).

## 6.  Congestion Control

This document specifies a Reno congestion controller for QUIC
[RFC6582].

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

If an endpoint 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].

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 5.2).

### 6.1.  Explicit Congestion Notification

If a path has been verified 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 its peer receives packets with the Congestion Experienced
codepoint.

### 6.2.  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 any time the congestion window is less than ssthresh, which
only occurs after persistent congestion is declared. While in slow
start, QUIC increases the congestion window by the number of bytes
acknowledged when each acknowledgment is processed.

### 6.3. 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.

### 6.4. 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.

### 6.5. 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.

### 6.6. 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.

### 6.7. 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 computed as follows:

(smoothed_rtt + 4 * rttvar + max_ack_delay) *
    kPersistentCongestionThreshold

For example, assume:

smoothed_rtt = 1 rttvar = 0 max_ack_delay = 0
kPersistentCongestionThreshold = 3

If an ack-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     |

Table 1

The first three packets are determined to be lost when the
acknowlegement 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 * kPersistentCongestionThreshold) = 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) [RACK].

## 6.8.  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 smoothed RTT 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.

Sending multiple packets into the network without any delay between
them creates a packet burst that might cause short-term congestion
and losses. Implementations MUST either use pacing or limit such
bursts to the initial congestion window, which is recommended to be
the minimum of 10 * max_datagram_size and max(2* max_datagram_size,
14720)), where max_datagram_size is the current maximum size of a
datagram for the connection, not including UDP or IP overhead.

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).

## 6.9.  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 credit.

A sender MAY use the pipeACK method described in section 4.3 of
[RFC7661] to determine if the congestion window is sufficiently
utilized.

A sender that paces packets (see Section 6.8) 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.

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

## 7.  Security Considerations

## 7.1.  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.

## 7.2.  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.

## 7.3.  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 sent with
ECN-CE is not reported as having been CE marked when the packet is
acknowledged, then the sender SHOULD 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.

## 8.  IANA Considerations

This document has no IANA actions. Yet.

## 9.  References

## 9.1.  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-25, 22 January 2020, <https://tools.ietf.org/html/
            draft-ietf-quic-tls-25>.

[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-25,
            22 January 2020, <https://tools.ietf.org/html/draft-ietf-
            quic-transport-25>.

[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>.

[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>.

## 9.2.  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", Work in Progress, Internet-Draft, draft-ietf-tcpm-
            rack-05, 26 April 2019, <http://www.ietf.org/internet-
            drafts/draft-ietf-tcpm-rack-05.txt>.

[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>.

[RFC4653]   Bhandarkar, S., Reddy, A. L. N., 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>.

[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>.

## Appendix A.  Loss Recovery Pseudocode

We now describe an example implementation of the loss detection mechanisms described in Section 5.

### 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 track it 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.

**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 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 500ms.

**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 [RFC6298]

**rttvar:**  The RTT variation, 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 for packets in the
   ApplicationData packet number space. The actual ack_delay in a
   received ACK frame may be larger due to late timers, reordering,
   or lost ACK frames.

**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_sent_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 = 0
rttvar = 0
min_rtt = 0
max_ack_delay = 0
for pn_space in [ Initial, Handshake, ApplicationData ]:
  largest_acked_packet[pn_space] = infinite
  time_of_last_sent_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_sent_ack_eliciting_packet[pn_space] = 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):
  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)

  // Nothing to do if there are no newly acked packets.
  newly_acked_packets = DetermineNewlyAckedPackets(ack, pn_space)
  if (newly_acked_packets.empty()):
    return

  // If the largest acknowledged is newly acked and
  // at least one ack-eliciting was newly acked, update the RTT.
  if (sent_packets[pn_space].contains(ack.largest_acked) &&
      IncludesAckEliciting(newly_acked_packets)):
    latest_rtt =
      now - sent_packets[pn_space][ack.largest_acked].time_sent
    ack_delay = 0
    if (pn_space == ApplicationData):
      ack_delay = ack.ack_delay
    UpdateRtt(ack_delay)

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

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

  DetectLostPackets(pn_space)

  pto_count = 0

  SetLossDetectionTimer()


UpdateRtt(ack_delay):
  // First RTT sample.
  if (smoothed_rtt == 0):
    min_rtt = latest_rtt
    smoothed_rtt = latest_rtt
    rttvar = latest_rtt / 2
    return

  // 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 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.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.in_flight):
    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:

```
GetEarliestTimeAndSpace(times):
  time = times[Initial]
  space = Initial
  for pn_space in [ Handshake, ApplicationData ]:
    if (times[pn_space] != 0 &&
        (time == 0 || times[pn_space] < time) &&
        # Skip ApplicationData until handshake completion.
        (pn_space != ApplicationData ||
          IsHandshakeComplete())):
      time = times[pn_space];
      space = pn_space
  return time, space

PeerNotAwaitingAddressValidation():
  # 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 ||
          has received 1-RTT ACK

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

  if (no ack-eliciting packets in flight &&
      PeerNotAwaitingAddressValidation()):
    loss_detection_timer.cancel()
    return

  // Use a default timeout if there are no RTT measurements
  if (smoothed_rtt == 0):
    timeout = 2 * kInitialRtt
  else:
    // Calculate PTO duration
    timeout = smoothed_rtt + max(4 * rttvar, kGranularity) +
      max_ack_delay
  timeout = timeout * (2 ^ pto_count)

  sent_time, _ = GetEarliestTimeAndSpace(
    time_of_last_sent_ack_eliciting_packet)
  loss_detection_timer.update(sent_time + 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 =
    GetEarliestTimeAndSpace(loss_time)
  if (earliest_loss_time != 0):
    // Time threshold loss Detection
    DetectLostPackets(pn_space)
    SetLossDetectionTimer()
    return

  if (endpoint is client without 1-RTT keys):
    // 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()
  else:
    // PTO. Send new data if available, else retransmit old data.
    // If neither is available, send a single PING frame.
    _, pn_space = GetEarliestTimeAndSpace(
      time_of_last_sent_ack_eliciting_packet)
    SendOneOrTwoAckElicitingPackets(pn_space)

  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.

Pseudocode for DetectLostPackets follows:

```
DetectLostPackets(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.
    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)

  // 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 6.

### 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 amount of data in flight, in bytes. The RECOMMENDED value is the minimum of 10 * max_datagram_size and max(2 * max_datagram_size, 14720)). 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.

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

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

**kPersistentCongestionThreshold:** Period of time for persistent
congestion to be established, specified as a PTO multiplier. 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) [RACK], before establishing persistent congestion, as TCP
does with a Retransmission Timeout (RTO) [RFC5681]. The
RECOMMENDED value for kPersistentCongestionThreshold is 3, which
is approximately 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.

**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 PMTU (see Section 14.1 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 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.

**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.

## 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 OnPacketAcked and is supplied with the
acked_packet from sent_packets.

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

OnPacketAckedCC(acked_packet):
  // Remove from bytes_in_flight.
  bytes_in_flight -= acked_packet.size
  if (InCongestionRecovery(acked_packet.time_sent)):
    // Do not increase congestion window in recovery period.
    return
  if (IsAppOrFlowControlLimited()):
    // Do not increase congestion_window if application
    // limited or flow control limited.
    return
  if (congestion_window < ssthresh):
    // Slow start.
    congestion_window += acked_packet.size
  else:
    // Congestion avoidance.
    congestion_window += max_datagram_size * 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 packet was sent after the
  // start of the previous congestion recovery period.
  if (!InCongestionRecovery(sent_time)):
    congestion_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, 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
    CongestionEvent(sent_packets[ack.largest_acked].time_sent)
```

### B.8.  On Packets Lost

Invoked from DetectLostPackets when packets are deemed lost.

```
InPersistentCongestion(largest_lost_packet):
  pto = smoothed_rtt + max(4 * rttvar, kGranularity) +
    max_ack_delay
  congestion_period = pto * kPersistentCongestionThreshold
  // Determine if all packets in the time period before the
  // newest lost packet, including the edges, are marked
  // lost
  return AreAllPacketsLost(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()
  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-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.2.  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.3.  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.4.  Since draft-ietf-quic-recovery-21**

*No changes

**C.5.  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.6.  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.7.  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.8.  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.9.  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.10.  Since draft-ietf-quic-recovery-14**

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

*Merge ACK and ACK_ECN (#1783)

**C.11.  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.12.  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.13.  Since draft-ietf-quic-recovery-11**

No significant changes.

**C.14.  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.15.  Since draft-ietf-quic-recovery-09**

  No significant changes.

**C.16.  Since draft-ietf-quic-recovery-08**

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

**C.17.  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.18.  Since draft-ietf-quic-recovery-06**

  No significant changes.

**C.19.  Since draft-ietf-quic-recovery-05**

   *Add more congestion control text (#776)

**C.20.  Since draft-ietf-quic-recovery-04**

  No significant changes.

**C.21.  Since draft-ietf-quic-recovery-03**

  No significant changes.

**C.22.  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.23.  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.24.  Since draft-ietf-quic-recovery-00**

    *Improved description of constants and ACK behavior

**C.25.  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,     (Kazuho Oku), Lars Eggert, Magnus
   Westerlund, Marten Seemann, Martin Duke, Martin Thomson, Nick Banks,
   Praveen Balasubramaniam.

**Acknowledgments**

**Authors' Addresses**

   Jana Iyengar (editor)
   Fastly

   Email: [jri.ietf@gmail.com](mailto:jri.ietf@gmail.com)

   Ian Swett (editor)
   Google

Email: [ianswett@google.com](mailto:ianswett@google.com)