QUIC
Internet-Draft
Intended status: Standards Track
Expires: October 19, 2018

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

April 17, 2018

QUIC Loss Detection and Congestion Control
draft-ietf-quic-recovery-11

Abstract

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

Note to Readers

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

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

Status of This Memo

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

1.1. Notational Conventions

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.

2. Design of the QUIC Transmission Machinery

All transmissions in QUIC are sent with a packet-level header, which includes a packet sequence number (referred to below as a packet number). These packet numbers never repeat in the lifetime of a connection, and are monotonically increasing, which prevents ambiguity. This fundamental design decision obviates the need for disambiguating between transmissions and retransmissions and eliminates significant complexity from QUIC's interpretation of TCP loss detection mechanisms.

Every packet may contain several frames. We outline the frames that are important to the loss detection and congestion control machinery below.

- o Retransmittable frames are those that count towards bytes in flight and need acknowledgement. The most common are STREAM frames, which typically contain application data.

- o Retransmittable packets are those that contain at least one retransmittable frame.

- o Crypto handshake data is sent on stream 0, and uses the reliability machinery of QUIC underneath.

- o ACK frames contain acknowledgment information. ACK frames contain one or more ranges of acknowledged packets.

2.1. 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.

2.1.1. Monotonically Increasing Packet Numbers

TCP conflates transmission sequence number at the sender with delivery sequence number at the receiver, which results in retransmissions of the same data carrying the same sequence number, and consequently to problems caused by "retransmission ambiguity". QUIC separates the two: QUIC uses a packet number for transmissions, and any data that is to be delivered to the receiving application(s) is sent in one or more streams, with delivery order determined by stream offsets encoded within STREAM frames.

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

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

2.1.2. 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.

2.1.3. 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.

2.1.4. Explicit Correction For Delayed ACKs

QUIC ACKs explicitly encode the delay incurred at the receiver between when a packet is received and when the corresponding ACK is sent. This allows the receiver of the ACK to adjust for receiver delays, specifically the delayed ack timer, when estimating the path RTT. This mechanism also allows a receiver to measure and report the

delay from when a packet was received by the OS kernel, which is useful in receivers which may incur delays such as context-switch latency before a userspace QUIC receiver processes a received packet.

3. Loss Detection

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

3.1. Computing the RTT estimate

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

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

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

3.2. Ack-based Detection

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

3.2.1. Fast Retransmit

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

The RECOMMENDED initial value for kReorderingThreshold is 3.

We derive this default from recommendations for TCP loss recovery [RFC5681] [RFC6675]. It is possible for networks to exhibit higher degrees of reordering, causing a sender to detect spurious losses. Detecting spurious losses leads to unnecessary retransmissions and may result in degraded performance due to the actions of the congestion controller upon detecting loss. Implementers MAY use algorithms developed for TCP, such as TCP-NCR [RFC4653], to improve QUIC's reordering resilience, though care should be taken to map TCP specifics to QUIC correctly. Similarly, using time-based loss detection to deal with reordering, such as in PR-TCP, should be more readily usable in QUIC. Making QUIC deal with such networks is important open research, and implementers are encouraged to explore this space.

3.2.2. Early Retransmit

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

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

An endpoint SHOULD set the alarm such that a packet is marked as lost no earlier than 1.25 * max(SRTT, latest_RTT) since when it was sent.

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

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

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

The 1.25 multiplier increases reordering resilience. Implementers MAY experiment with using other multipliers, bearing in mind that a lower multiplier reduces reordering resilience and increases spurious retransmissions, and a higher multipler increases loss recovery delay.

This mechanism is based on Early Retransmit for TCP [RFC5827]. However, [RFC5827] does not include the alarm described above. Early Retransmit is prone to spurious retransmissions due to its reduced reordering resilence without the alarm. This observation led Linux TCP implementers to implement an alarm for TCP as well, and this document incorporates this advancement.

3.3. Timer-based Detection

Timer-based loss detection implements a handshake retransmission timer that is optimized for QUIC as well as the spirit of TCP's Tail Loss Probe and Retransmission Timeout mechanisms.

3.3.1. Handshake Timeout

Handshake packets, which contain STREAM frames for stream 0, are critical to QUIC transport and crypto negotiation, so a separate alarm is used for them.

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

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

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

When a handshake packet is sent, the sender SHOULD set an alarm for the handshake timeout period.

When the alarm fires, the sender MUST retransmit all unacknowledged handshake data, by calling RetransmitAllUnackedHandshakeData(). On each consecutive firing of the handshake alarm, the sender SHOULD double the handshake timeout and set an alarm for this period.

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

Handshake data may be cancelled by handshake state transitions. In particular, all non-protected data SHOULD no longer be transmitted once packet protection is available.

(TODO: Work this section some more. Add text on client vs. server, and on stateless retry.)

3.3.2. Tail Loss Probe

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

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

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

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

- If RTO (Section 3.3.3) is earlier, schedule a TLP alarm in its place. That is, PTO SHOULD be scheduled for min(RTO, PTO).

MaxAckDelay is the maximum ack delay supplied in an incoming ACK frame. MaxAckDelay excludes ack delays that aren't included in an RTT sample because they're too large and excludes those which reference an ack-only packet.

QUIC diverges from TCP by calculating MaxAckDelay dynamically, instead of assuming a constant delayed ack timeout for all connections. QUIC includes this in all probe timeouts, because it assume the ack delay may come into play, regardless of the number of packets outstanding. TCP's TLP assumes if at least 2 packets are outstanding, acks will not be delayed.

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

To reduce latency, it is RECOMMENDED that the sender set and allow the TLP alarm to fire twice before setting an RTO alarm. In other words, when the TLP alarm fires the first time, a TLP packet is sent, and it is RECOMMENDED that the TLP alarm be scheduled for a second time. When the TLP alarm fires the second time, a second TLP packet is sent, and an RTO alarm SHOULD be scheduled Section 3.3.3.

A TLP packet SHOULD carry new data when possible. If new data is unavailable or new data cannot be sent due to flow control, a TLP packet MAY retransmit unacknowledged data to potentially reduce

recovery time. Since a TLP alarm is used to send a probe into the network prior to establishing any packet loss, prior unacknowledged packets SHOULD NOT be marked as lost when a TLP alarm fires.

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

3.3.3. Retransmission Timeout

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

When the last TLP packet is sent, an alarm is scheduled for the RTO period. When this alarm fires, the sender sends two packets, to evoke acknowledgements from the receiver, and restarts the RTO alarm.

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

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

o When an RTO alarm fires, the RTO period is doubled.

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

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

QUIC also diverges from TCP by including MaxAckDelay in the RTO period. QUIC is able to explicitly model delay at the receiver via the ack delay field in the ACK frame. Since QUIC corrects for this delay in its SRTT and RTTVAR computations, it is necessary to add this delay explicitly in the TLP and RTO computation.

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

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

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

3.4. Generating Acknowledgements

QUIC SHOULD delay sending acknowledgements in response to packets, but MUST NOT excessively delay acknowledgements of packets containing non-ack frames. Specifically, implementaions MUST attempt to enforce a maximum ack delay to avoid causing the peer spurious timeouts. The default maximum ack delay in QUIC is 25ms.

An acknowledgement MAY be sent for every second full-sized packet, as TCP does [RFC5681], or may be sent less frequently, as long as the delay does not exceed the maximum ack delay. QUIC recovery algorithms do not assume the peer generates an acknowledgement immediately when receiving a second full-sized packet.

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

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

3.4.1. ACK Ranges

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

ACK frames SHOULD always acknowledge the most recently received packets, and the more out-of-order the packets are, the more

important it is to send an updated ACK frame quickly, to prevent the peer from declaring a packet as lost and spuriusly retransmitting the frames it contains.

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

3.4.2. Receiver Tracking of ACK Frames

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

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

3.5. Pseudocode

3.5.1. Constants of interest

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

kMaxTLPs (default 2): Maximum number of tail loss probes before an RTO fires.

kReorderingThreshold (default 3): Maximum reordering in packet number space before FACK style loss detection considers a packet lost.

kTimeReorderingFraction (default 1/8): Maximum reordering in time space before time based loss detection considers a packet lost. In fraction of an RTT.

kUsingTimeLossDetection (default false): Whether time based loss detection is in use. If false, uses FACK style loss detection.

kMinTLPTimeout (default 10ms): Minimum time in the future a tail loss probe alarm may be set for.

kMinRTOTimeout (default 200ms): Minimum time in the future an RTO alarm may be set for.

kDelayedAckTimeout (default 25ms): The length of the peer's delayed ack timer.

kDefaultInitialRtt (default 100ms): The default RTT used before an RTT sample is taken.

3.5.2. Variables of interest

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

loss_detection_alarm: Multi-modal alarm used for loss detection.

handshake_count: The number of times the handshake packets have been retransmitted without receiving an ack.

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

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

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

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

time_of_last_sent_handshake_packet: The time the most recent packet containing handshake data was sent.

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

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

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

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

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

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

max_ack_delay: The maximum ack delay in an incoming ACK frame for this connection. Excludes ack delays for ack only packets and those that create an RTT sample less than min_rtt.

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

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

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

sent_packets: An association of packet numbers to information about them, including a number field indicating the packet number, a time field indicating the time a packet was sent, a boolean indicating whether the packet is ack only, and a bytes field indicating the packet's size. sent_packets is ordered by packet number, and packets remain in sent_packets until acknowledged or lost.

3.5.3. Initialization

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

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

3.5.4. On Sending a Packet

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

- packet_number: The packet number of the sent packet.
- is_ack_only: A boolean that indicates whether a packet only contains an ACK frame. If true, it is still expected an ack will be received for this packet, but it is not retransmittable.
- is_handshake_packet: A boolean that indicates whether a packet contains handshake data.
- sent_bytes: The number of bytes sent in the packet, not including UDP or IP overhead, but including QUIC framing overhead.

Pseudocode for OnPacketSent follows:

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

3.5.5. On Ack Receipt

When an ack is received, it may acknowledge 0 or more packets.

Pseudocode for OnAckReceived and UpdateRtt follow:

```
     OnAckReceived(ack):
       largest_acked_packet = ack.largest_acked
       // If the largest acked is newly acked, update the RTT.
       if (sent_packets[ack.largest_acked]):
         latest_rtt = now - sent_packets[ack.largest_acked].time
         UpdateRtt(latest_rtt, ack.ack_delay)
       // Find all newly acked packets.
       for acked_packet in DetermineNewlyAckedPackets():
         OnPacketAcked(acked_packet.packet_number)

       DetectLostPackets(ack.largest_acked_packet)
       SetLossDetectionAlarm()


     UpdateRtt(latest_rtt, ack_delay):
       // min_rtt ignores ack delay.
       min_rtt = min(min_rtt, latest_rtt)
       // Adjust for ack delay if it's plausible.
       if (latest_rtt - min_rtt > ack_delay):
         latest_rtt -= ack_delay
         // Only save into max ack delay if it's used
         // for rtt calculation and is not ack only.
         if (!sent_packets[ack.largest_acked].ack_only)
           max_ack_delay = max(max_ack_delay, ack_delay)
       // Based on {{RFC6298}}.
       if (smoothed_rtt == 0):
         smoothed_rtt = latest_rtt
         rttvar = latest_rtt / 2
       else:
         rttvar_sample = abs(smoothed_rtt - latest_rtt)
         rttvar = 3/4 * rttvar + 1/4 * rttvar_sample
         smoothed_rtt = 7/8 * smoothed_rtt + 1/8 * latest_rtt
```

3.5.6. On Packet Acknowledgment

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

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

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

Pseudocode for OnPacketAcked follows:

```
OnPacketAcked(acked_packet):
  if (!acked_packet.is_ack_only):
    OnPacketAckedCC(acked_packet)
  // If a packet sent prior to RTO was acked, then the RTO
  // was spurious.  Otherwise, inform congestion control.
  if (rto_count > 0 &&
      acked_packet.packet_number > largest_sent_before_rto)
    OnRetransmissionTimeoutVerified()
  handshake_count = 0
  tlp_count = 0
  rto_count = 0
  sent_packets.remove(acked_packet.packet_number)
```

3.5.7. **Setting the Loss Detection Alarm**

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

3.5.7.1. **Handshake Alarm**

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

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

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

3.5.7.2. **Tail Loss Probe and Retransmission Alarm**

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

The TLP and RTO timers are armed when there is not unacknowledged handshake data. The TLP alarm is set until the max number of TLP packets have been sent, and then the RTO timer is set.

3.5.7.3. Early Retransmit Alarm

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

3.5.7.4. Pseudocode

Pseudocode for SetLossDetectionAlarm follows:

```
 SetLossDetectionAlarm():
    // Don't arm the alarm if there are no packets with
    // retransmittable data in flight.
    if (bytes_in_flight == 0):
      loss_detection_alarm.cancel()
      return

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

    loss_detection_alarm.set(
      time_of_last_sent_retransmittable_packet + alarm_duration)
```

3.5.8. On Alarm Firing

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

Pseudocode for OnLossDetectionAlarm follows:

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

     SetLossDetectionAlarm()
```

3.5.9. Detecting Lost Packets

Packets in QUIC are only considered lost once a larger packet number is acknowledged. DetectLostPackets is called every time an ack is received. If the loss detection alarm fires and the loss_time is set, the previous largest acked packet is supplied.

3.5.9.1. Handshake Packets

The receiver MUST close the connection with an error of type OPTIMISTIC_ACK when receiving an unprotected packet that acks protected packets. The receiver MUST trust protected acks for unprotected packets, however. Aside from this, loss detection for handshake packets when an ack is processed is identical to other packets.

3.5.9.2. Pseudocode

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

Pseudocode for DetectLostPackets follows:

```
DetectLostPackets(largest_acked):
  loss_time = 0
  lost_packets = {}
  delay_until_lost = infinite
  if (kUsingTimeLossDetection):
    delay_until_lost =
      (1 + time_reordering_fraction) *
          max(latest_rtt, smoothed_rtt)
  else if (largest_acked.packet_number == largest_sent_packet):
    // Early retransmit alarm.
    delay_until_lost = 5/4 * max(latest_rtt, smoothed_rtt)
  foreach (unacked < largest_acked.packet_number):
    time_since_sent = now() - unacked.time_sent
    delta = largest_acked.packet_number - unacked.packet_number
    if (time_since_sent > delay_until_lost ||
        delta > reordering_threshold):
      sent_packets.remove(unacked.packet_number)
      if (!unacked.is_ack_only):
        lost_packets.insert(unacked)
    else if (loss_time == 0 && delay_until_lost != infinite):
      loss_time = now() + delay_until_lost - time_since_sent

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

3.6. Discussion

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

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

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

4. Congestion Control

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

QUIC hosts MUST NOT send packets if they would increase bytes_in_flight (defined in Section 4.7.2) beyond the available congestion window, unless the packet is a probe packet sent after the TLP or RTO alarm fires, as described in Section 3.3.2 and Section 3.3.3.

4.1. Slow Start

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

4.2. 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 MSS of bytes 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.

4.3. Recovery Period

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

During recovery, the congestion window is not increased or decreased. As such, multiple lost packets only decrease the congestion window once as long as they're lost before exiting recovery. This causes QUIC to decrease the congestion window multiple times if retransmisions are lost, but limits the reduction to once per round trip.

4.4. Tail Loss Probe

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

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

4.5. Retransmission Timeout

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

4.6. Pacing

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

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

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

4.7. Pseudocode

4.7.1. Constants of interest

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

kDefaultMss (default 1460 bytes): The default max packet size used for calculating default and minimum congestion windows.

kInitialWindow (default 10 * kDefaultMss): Default limit on the amount of outstanding data in bytes.

kMinimumWindow (default 2 * kDefaultMss): Default minimum congestion window.

kLossReductionFactor (default 0.5): Reduction in congestion window when a new loss event is detected.

4.7.2. Variables of interest

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

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

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

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

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

4.7.3. Initialization

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

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

4.7.4. 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
```

4.7.5. On Packet Acknowledgement

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

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

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

4.7.6. On Packets Lost

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

```
   OnPacketsLost(lost_packets):
     // Remove lost packets from bytes_in_flight.
     for (lost_packet : lost_packets):
       bytes_in_flight -= lost_packet.bytes
     largest_lost_packet = lost_packets.last()
     // Start a new recovery epoch if the lost packet is larger
     // than the end of the previous recovery epoch.
     if (!InRecovery(largest_lost_packet.packet_number)):
       end_of_recovery = largest_sent_packet
       congestion_window *= kLossReductionFactor
       congestion_window = max(congestion_window, kMinimumWindow)
       ssthresh = congestion_window
```

4.7.7. On Retransmission Timeout Verified

QUIC decreases the congestion window to the minimum value once the retransmission timeout has been verified.

```
   OnRetransmissionTimeoutVerified()
     congestion_window = kMinimumWindow
```

5. IANA Considerations

This document has no IANA actions. Yet.

6. References

6.1. Normative References

[QUIC-TRANSPORT]
Iyengar, J., Ed. and M. Thomson, Ed., "QUIC: A UDP-Based Multiplexed and Secure Transport", draft-ietf-quic-transport-11 (work in progress), April 2018.

[RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, DOI 10.17487/RFC2119, March 1997, <https://www.rfc-editor.org/info/rfc2119>.

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

6.2. Informative References

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

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

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

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

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

[RFC6298] Paxson, V., Allman, M., Chu, J., and M. Sargent, "Computing TCP's Retransmission Timer", RFC 6298, DOI 10.17487/RFC6298, June 2011, <https://www.rfc-editor.org/info/rfc6298>.

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

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

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

6.3. URIs

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

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

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

Appendix A. Acknowledgments

Appendix B. Change Log

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

B.1. 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)

B.2. Since draft-ietf-quic-recovery-09

No significant changes.

B.3. Since draft-ietf-quic-recovery-08

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

B.4. 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.

B.5. Since draft-ietf-quic-recovery-06

No significant changes.

B.6. Since draft-ietf-quic-recovery-05

- Add more congestion control text (#776)

B.7. Since draft-ietf-quic-recovery-04

No significant changes.

B.8. Since draft-ietf-quic-recovery-03

No significant changes.

B.9. 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)

B.10. 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

B.11. Since draft-ietf-quic-recovery-00

- Improved description of constants and ACK behavior

B.12. Since draft-iyengar-quic-loss-recovery-01

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

Authors' Addresses

Jana Iyengar (editor)
Fastly

Email: jri.ietf@gmail.com

Ian Swett (editor)
Google

Email: ianswett@google.com