QUIC
Internet-Draft
Intended status: Standards Track
Expires: September 14, 2017

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

March 13, 2017

QUIC Loss Detection and Congestion Control
draft-ietf-quic-recovery-02

Abstract

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

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

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.

This document first describes pre-requisite parts of the QUIC transmission machinery, then discusses QUIC's default congestion control and loss detection mechanisms, and finally lists the various TCP mechanisms that QUIC loss detection implements (in spirit.)

1.1. Notational Conventions

The words "MUST", "MUST NOT", "SHOULD", and "MAY" are used in this document. It's not shouting; when they are capitalized, they have the special meaning defined in [RFC2119].

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 makes duplicate detection trivial. 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.

- Retransmittable frames are frames requiring reliable delivery. The most common are STREAM frames, which typically contain application data.

- Crypto handshake data is also sent as STREAM data, and uses the reliability machinery of QUIC underneath.

- ACK frames contain acknowledgment information. QUIC uses a SACK-based scheme, where acks express up to 256 ranges. The ACK frame also includes a receive timestamp for each packet newly acked.

2.1. Relevant Differences Between QUIC and TCP

There are some notable differences between QUIC and TCP which are important for reasoning about the differences between the loss recovery mechanisms employed by the two protocols. We briefly describe these 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 sequence number (referred to as the "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 stream offsets encoded within STREAM frames inside of packets that determine delivery order.

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 equivalent 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 up to 256 ACK ranges, opposed to TCP's 3 SACK ranges. In high loss environments, this speeds recovery.

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

We now describe QUIC's loss detection as functions that should be called on packet transmission, when a packet is acked, and timer expiration events.

3.1. Constants of interest

Constants used in loss recovery and 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.

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 sapce before time based loss detection considers a packet lost. In fraction of an RTT.

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.

3.2. Variables of interest

We first describe the variables required to implement the loss detection mechanisms 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.

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

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

initial_rtt: The initial RTT used before any RTT measurements have been made.

reordering_threshold: The largest delta between the largest acked retransmittable packet and a packet containing retransmittable frames before it’s 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, 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.

3.3. 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 (UsingTimeLossDetection())
  reordering_threshold = infinite
  time_reordering_fraction = kTimeReorderingFraction
else:
  reordering_threshold = kReorderingThreshold
  time_reordering_fraction = infinite
loss_time = 0
smoothed_rtt = 0
rttvar = 0
initial_rtt = kDefaultInitialRtt
```

3.4. 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_retransmittble: A boolean that indicates whether the packet contains at least one frame requiring reliable deliver. The retransmittability of various QUIC frames is described in [QUIC-TRANSPORT]. If false, it is still acceptable for an ack to be received for this packet. However, a caller MUST NOT set is_retransmittable to true if an ack is not expected.

- sent_bytes: The number of bytes sent in the packet.

Pseudocode for OnPacketSent follows:

```
OnPacketSent(packet_number, is_retransmittable, sent_bytes):
  sent_packets[packet_number].packet_number = packet_number
  sent_packets[packet_number].time = now
  if is_retransmittable:
    sent_packets[packet_number].bytes = sent_bytes
    SetLossDetectionAlarm()
```

3.5. On Ack Receipt

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

Pseudocode for OnAckReceived and UpdateRtt follow:

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

  DetectLostPackets(ack.largest_acked_packet)
  SetLossDetectionAlarm()


UpdateRtt(rtt_sample):
  // Based on {{RFC6298}}.
  if (smoothed_rtt == 0):
    smoothed_rtt = rtt_sample
    rttvar = rtt_sample / 2
  else:
    rttvar = 3/4 * rttvar + 1/4 * (smoothed_rtt - rtt_sample)
    smoothed_rtt = 7/8 * smoothed_rtt + 1/8 * rtt_sample
```

3.6. 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 packet number of the newly acked packet, and returns a list of packet numbers that are detected as lost.

Pseudocode for OnPacketAcked follows:

```
OnPacketAcked(acked_packet_number):
  handshake_count = 0
  tlp_count = 0
  rto_count = 0
  sent_packets.remove(acked_packet_number)
```

3.7. 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.

3.7.1. Handshake Packets

The initial flight has no prior RTT sample. A client SHOULD remember the previous RTT it observed when resumption is attempted and use that for an initial RTT value. If no previous RTT is available, the initial RTT defaults to 200ms. Once an RTT measurement is taken, it MUST replace initial_rtt.

Endpoints MUST retransmit handshake frames if not acknowledged within a time limit. This time limit will start as the largest of twice the rtt value and MinTLPTimeout. Each consecutive handshake retransmission doubles the time limit, until an acknowledgement is received.

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

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 per handshake packet that uses an unsupported QUIC version, and MAY be sent in response to 0RTT packets.

3.7.2. Tail Loss Probe and Retransmission Timeout

Tail loss probes [I-D.dukkipati-tcpm-tcp-loss-probe] 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.

3.7.3. Early Retransmit

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.

3.7.4. Pseudocode

Pseudocode for SetLossDetectionAlarm follows:

```
 SetLossDetectionAlarm():
    if (retransmittable packets are not outstanding):
      loss_detection_alarm.cancel();
      return

    if (handshake packets are outstanding):
      // Handshake retransmission alarm.
      if (smoothed_rtt == 0):
        alarm_duration = 2 * initial_rtt
      else:
        alarm_duration = 2 * smoothed_rtt
      alarm_duration = max(alarm_duration, kMinTLPTimeout)
      alarm_duration = alarm_duration << handshake_count
    else if (loss_time != 0):
      // Early retransmit timer or time loss detection.
      alarm_duration = loss_time - now
    else if (tlp_count < kMaxTLPs):
      // Tail Loss Probe
      if (retransmittable_packets_outstanding = 1):
        alarm_duration = 1.5 * smoothed_rtt + kDelayedAckTimeout
      else:
        alarm_duration = kMinTLPTimeout
      alarm_duration = max(alarm_duration, 2 * smoothed_rtt)
    else:
      // RTO alarm
      if (rto_count = 0):
        alarm_duration = smoothed_rtt + 4 * rttvar
        alarm_duration = max(alarm_duration, kMinRTOTimeout)
      else:
        alarm_duration = loss_detection_alarm.get_delay() << 1

    loss_detection_alarm.set(now + alarm_duration)
```

3.8. 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.
    RetransmitAllHandshakePackets();
    handshake_count++;
  // TODO: Clarify early retransmit and time loss.
  else if (loss_time != 0):
    // Early retransmit or Time Loss Detection
    DetectLostPackets(largest_acked_packet)
  else if (tlp_count < kMaxTLPs):
    // Tail Loss Probe.
    if (HasNewDataToSend()):
      SendOnePacketOfNewData()
    else:
      RetransmitOldestPacket()
    tlp_count++
  else:
    // RTO.
    RetransmitOldestTwoPackets()
    rto_count++

  SetLossDetectionAlarm()
```

3.9. 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.

3.9.1. Handshake Packets

The receiver MUST ignore unprotected packets that ack 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.

3.9.2. 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 (time_reordering_fraction != infinite):
    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 = 9/8 * max(latest_rtt, smoothed_rtt)
  foreach (unacked less than largest_acked.packet_number):
    time_since_sent = now() - unacked.time_sent
    packet_delta = largest_acked.packet_number - unacked.packet_number
    if (time_since_sent > delay_until_lost):
      lost_packets.insert(unacked)
    else if (packet_delta > reordering_threshold)
      lost_packets.insert(unacked)
    else if (loss_time == 0 && delay_until_lost != infinite):
      loss_time = delay_until_lost - time_since_sent

  // Inform the congestion controller of lost packets and
  // lets it decide whether to retransmit immediately.
  OnPacketsLost(lost_packets)
  foreach (packet in lost_packets)
    sent_packets.remove(packet.packet_number)
```

4. Congestion Control

(describe NewReno-style congestion control [RFC6582] for QUIC.) (describe appropriate byte counting.) (define recovery based on packet numbers.) (describe min_rtt based hystart.) (describe how QUIC's F-RTO [RFC5682] delays reducing CWND.) (describe PRR [RFC6937])

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".

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

6.2. Informative References

[I-D.dukkipati-tcpm-tcp-loss-probe] 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.

[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, <http://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, <http://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, <http://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, <http://www.rfc-editor.org/info/rfc6582>.

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

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-01

- 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.2. Since draft-ietf-quic-recovery-00:

- Improved description of constants and ACK behavior

B.3. 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)
Google

Email: jri@google.com

Ian Swett (editor)
Google

Email: ianswett@google.com