The RACK-TLP loss detection algorithm for TCP
draft-ietf-tcpm-rack-14

Abstract

   This document presents the RACK-TLP loss detection algorithm for TCP.
   RACK-TLP uses per-segment transmit timestamps and selective
   acknowledgements (SACK) and has two parts: RACK ("Recent
   ACKnowledgment") starts fast recovery quickly using time-based
   inferences derived from ACK feedback. TLP ("Tail Loss Probe")
   leverages RACK and sends a probe packet to trigger ACK feedback to
   avoid retransmission timeout (RTO) events. Compared to the widely
   used DUPACK threshold approach, RACK-TLP detects losses more
   efficiently when there are application-limited flights of data, lost
   retransmissions, or data packet reordering events. It is intended to
   be an alternative to the DUPACK threshold approach.

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 June 4, 2021.

Copyright Notice

Table of Contents

1. Terminology

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. In this document, these words will appear
with that interpretation only when in UPPER CASE. Lower case uses of
these words are not to be interpreted as carrying [RFC2119]
significance.

2. Introduction

This document presents RACK-TLP, a TCP loss detection algorithm that
improves upon the widely implemented DUPACK counting approach in
[RFC5681] [RFC6675], and that is RECOMMENDED to be used as an
alternative to that earlier approach. RACK-TLP has two parts: RACK
("Recent ACKnowledgment") detects losses quickly using time-based
inferences derived from ACK feedback. TLP ("Tail Loss Probe")
triggers ACK feedback by quickly sending a probe segment, to avoid
retransmission timeout (RTO) events.

2.1. Background

In traditional TCP loss recovery algorithms [RFC5681] [RFC6675], a
sender starts fast recovery when the number of DUPACKs received
reaches a threshold (DupThresh) that defaults to 3 (this approach is
referred to as DUPACK-counting in the rest of the document). The
sender also halves the congestion window during the recovery. The
rationale behind the partial window reduction is that congestion does
not seem severe since ACK clocking is still maintained. The time
elapsed in fast recovery can be just one round-trip, e.g. if the
sender uses SACK-based recovery [RFC6675] and the number of lost
segments is small.

If fast recovery is not triggered, or triggers but fails to repair
all the losses, then the sender resorts to RTO recovery. The RTO
timer interval is conservatively the smoothed RTT (SRTT) plus four
times the RTT variation, and is lower bounded to 1 second [RFC6298].

Upon RTO timer expiration, the sender retransmits the first
unacknowledged segment and resets the congestion window to the LOSS
WINDOW value (by default 1 full-size segment [RFC5681]). The
rationale behind the congestion window reset is that an entire flight
of data was lost, and the ACK clock was lost, so this deserves a
cautious response. The sender then retransmits the rest of the data
following the slow start algorithm [RFC5681]. The time elapsed in
RTO recovery is one RTO interval plus the number of round-trips
needed to repair all the losses.

2.2. Motivation

Fast Recovery is the preferred form of loss recovery because it can
potentially recover all losses in the time scale of a single round
trip, with only a fractional congestion window reduction. RTO
recovery and congestion window reset should ideally be the last
resort, only used when the entire flight is lost. However, in
addition to losing an entire flight of data, the following situations
can unnecessarily resort to RTO recovery with traditional TCP loss
recovery algorithms [RFC5681] [RFC6675]:

1. Packet drops for short flows or at the end of an application data
    flight. When the sender is limited by the application (e.g.
    structured request/response traffic), segments lost at the end of
    the application data transfer often can only be recovered by RTO.
    Consider an example of losing only the last segment in a flight
    of 100 segments. Lacking any DUPACK, the sender RTO expires and
    reduces the congestion window to 1, and raises the congestion
    window to just 2 after the loss repair is acknowledged. In
    contrast, any single segment loss occurring between the first and
    the 97th segment would result in fast recovery, which would only
    cut the window in half.

2. Lost retransmissions. Heavy congestion or traffic policers can
    cause retransmissions to be lost. Lost retransmissions cause a
    resort to RTO recovery, since DUPACK-counting does not detect the
    loss of the retransmissions. Then the slow start after RTO
    recovery could cause burst losses again that severely degrades
    performance [POLICER16].

3. Packet reordering. Link-layer protocols (e.g., 802.11 block
    ACK), link bonding, or routers' internal load-balancing (e.g.,
    ECMP) can deliver TCP segments out of order. The degree of such
    reordering is usually within the order of the path round trip
    time. If the reordering degree is beyond DupThresh, the DUPACK-
    counting can cause a spurious fast recovery and unnecessary
    congestion window reduction. To mitigate the issue, [RFC4653]
    adjusts DupThresh to half of the inflight size to tolerate the

higher degree of reordering. However if more than half of the
inflight is lost, then the sender has to resort to RTO recovery.

3. RACK-TLP high-level design

RACK-TLP allows senders to recover losses more effectively in all
three scenarios described in the previous section. There are two
design principles behind RACK-TLP. The first principle is to detect
losses via ACK events as much as possible, to repair losses at round-
trip time-scales. The second principle is to gently probe the
network to solicit additional ACK feedback, to avoid RTO expiration
and subsequent congestion window reset. At a high level, the two
principles are implemented in RACK and TLP, respectively.

3.1. RACK: time-based loss inferences from ACKs

The rationale behind RACK is that if a segment is delivered out of
order, then the segments sent chronologically before that were either
lost or reordered. This concept is not fundamentally different from
[RFC5681] [RFC6675] [FACK]. RACK's key innovation is using per-
segment transmission timestamps and widely-deployed SACK [RFC2018]
options to conduct time-based inferences, instead of inferring losses
by counting ACKs or SACKed sequences. Time-based inferences are more
robust than DUPACK-counting approaches because they have no
dependence on flight size, and thus are effective for application-
limited traffic.

Conceptually, RACK puts a virtual timer for every data segment sent
(including retransmissions). Each timer expires dynamically based on
the latest RTT measurements plus an additional delay budget to
accommodate potential packet reordering (called the reordering
window). When a segment's timer expires, RACK marks the
corresponding segment lost for retransmission.

In reality, as an algorithm, RACK does not arm a timer for every
segment sent because it's not necessary. Instead the sender records
the most recent transmission time of every data segment sent,
including retransmissions. For each ACK received, the sender
calculates the latest RTT measurement (if eligible) and adjusts the
expiration time of every segment sent but not yet delivered. If a
segment has expired, RACK marks it lost.

Since the time-based logic of RACK applies equally to retransmissions
and original transmissions, it can detect lost retransmissions as
well. If a segment has been retransmitted but its most recent
(re)transmission timestamp has expired, then after a reordering
window it's marked lost.

3.2. TLP: sending one segment to probe losses quickly with RACK

   RACK infers losses from ACK feedback; however, in some cases ACKs are
   sparse, particularly when the inflight is small or when the losses
   are high. In some challenging cases the last few segments in a
   flight are lost. With [RFC5681] or [RFC6675] the sender's RTO would
   expire and reset the congestion window, when in reality most of the
   flight has been delivered.

   Consider an example where a sender with a large congestion window
   transmits 100 new data segments after an application write, and only
   the last three segments are lost. Without RACK-TLP, the RTO expires,
   the sender retransmits the first unacknowledged segment, and the
   congestion window slow-starts from 1. After all the retransmits are
   acknowledged the congestion window has been increased to 4. The
   total delivery time for this application transfer is three RTTs plus
   one RTO, a steep cost given that only a tiny fraction of the flight
   was lost. If instead the losses had occurred three segments sooner
   in the flight, then fast recovery would have recovered all losses
   within one round-trip and would have avoided resetting the congestion
   window.

   Fast Recovery would be preferable in such scenarios; TLP is designed
   to trigger the feedback RACK needed to enable that. After the last
   (100th) segment was originally sent, TLP sends the next available
   (new) segment or retransmits the last (highest-sequenced) segment in
   two round-trips to probe the network, hence the name "Tail Loss
   Probe". The successful delivery of the probe would solicit an ACK.
   RACK uses this ACK to detect that the 98th and 99th segments were
   lost, trigger fast recovery, and retransmit both successfully. The
   total recovery time is four RTTs, and the congestion window is only
   partially reduced instead of being fully reset. If the probe was
   also lost then the sender would invoke RTO recovery resetting the
   congestion window.

3.3. RACK-TLP: reordering resilience with a time threshold

3.3.1. Reordering design rationale

   Upon receiving an ACK indicating an out-of-order data delivery, a
   sender cannot tell immediately whether that out-of-order delivery was
   a result of reordering or loss. It can only distinguish between the
   two in hindsight if the missing sequence ranges are filled in later
   without retransmission. Thus a loss detection algorithm needs to
   budget some wait time -- a reordering window -- to try to
   disambiguate packet reordering from packet loss.

The reordering window in the DUPACK-counting approach is implicitly
defined as the elapsed time to receive acknowledgements for
DupThresh-worth of out-of-order deliveries. This approach is
effective if the network reordering degree (in sequence distance) is
smaller than DupThresh and at least DupThresh segments after the loss
are acknowledged. For cases where the reordering degree is larger
than the default DupThresh of 3 packets, one alternative is to
dynamically adapt DupThresh based on the FlightSize (e.g., the sender
adjusts the DUPTHRESH to half of the FlightSize). However, this does
not work well with the following two types of reordering:

1. Application-limited flights where the last non-full-sized segment
    is delivered first and then the remaining full-sized segments in
    the flight are delivered in order. This reordering pattern can
    occur when segments traverse parallel forwarding paths. In such
    scenarios the degree of reordering in packet distance is one
    segment less than the flight size.

2. A flight of segments that are delivered partially out of order.
    One cause for this pattern is wireless link-layer retransmissions
    with an inadequate reordering buffer at the receiver. In such
    scenarios, the wireless sender sends the data packets in order
    initially, but some are lost and then recovered by link-layer
    retransmissions; the wireless receiver delivers the TCP data
    packets in the order they are received, due to the inadequate
    reordering buffer. The random wireless transmission errors in
    such scenarios cause the reordering degree, expressed in packet
    distance, to have highly variable values up to the flight size.

In the above two cases the degree of reordering in packet distance is
highly variable. This makes DUPACK-counting approach ineffective
including dynamic adaptation variants like [RFC4653]. Instead the
degree of reordering in time difference in such cases is usually
within a single round-trip time. This is because the packets either
traverse slightly disjoint paths with similar propagation delays or
are repaired quickly by the local access technology. Hence, using a
time threshold instead of packet threshold strikes a middle ground,
allowing a bounded degree of reordering resilience while still
allowing fast recovery. This is the rationale behind the RACK-TLP
reordering resilience design.

Specifically, RACK-TLP introduces a new dynamic reordering window
parameter in time units, and the sender considers a data segment S
lost if both conditions are met:

1. Another data segment sent later than S has been delivered

2. S has not been delivered after the estimated round-trip time plus
    the reordering window

Note that condition (1) implies at least one round-trip of time has
elapsed since S has been sent.

3.3.2. Reordering window adaptation

The RACK reordering window adapts to the measured duration of
reordering events, within reasonable and specific bounds to
disincentivize excessive reordering. More specifically, the sender
sets the reordering window as follows:

1. The reordering window SHOULD be set to zero if no reordering has
    been observed on the connection so far, and either (a) three
    segments have been delivered out of order since the last recovery
    or (b) the sender is already in fast or RTO recovery. Otherwise,
    the reordering window SHOULD start from a small fraction of the
    round trip time, or zero if no round trip time estimate is
    available.

2. The RACK reordering window SHOULD adaptively increase (using the
    algorithm in "Step 4: Update RACK reordering window", below) if
    the sender receives a Duplicate Selective Acknowledgement (DSACK)
    option [RFC2883]. Receiving a DSACK suggests the sender made a
    spurious retransmission, which may have been due to the
    reordering window being too small.

3. The RACK reordering window MUST be bounded and this bound SHOULD
    be SRTT.

Rules 2 and 3 are required to adapt to reordering caused by dynamics
such as the prolonged link-layer loss recovery episodes described
earlier. Each increase in the reordering window requires a new round
trip where the sender receives a DSACK; thus, depending on the extent
of reordering, it may take multiple round trips to fully adapt.

For short flows, the low initial reordering window helps recover
losses quickly, at the risk of spurious retransmissions. The
rationale is that spurious retransmissions for short flows are not
expected to produce excessive additional network traffic. For long
flows the design tolerates reordering within a round trip. This
handles reordering in small time scales (reordering within the round-
trip time of the shortest path).

However, the fact that the initial reordering window is low, and the
reordering window's adaptive growth is bounded, means that there will

continue to be a cost to reordering that disincentivizes excessive
reordering.

3.4. An Example of RACK-TLP in Action: fast recovery

The following example in figure 1 illustrates the RACK-TLP algorithm
in action:

```
Event  TCP DATA SENDER                          TCP DATA RECEIVER
_____  _____
  1.   Send P0, P1, P2, P3            -->
       [P1, P2, P3 dropped by network]

  2.                                  <--         Receive P0, ACK P0

  3a.  2RTTs after (2), TLP timer fires
  3b.  TLP: retransmits P3            -->

  4.                                  <--         Receive P3, SACK P3

  5a.  Receive SACK for P3
  5b.  RACK: marks P1, P2 lost
  5c.  Retransmit P1, P2             -->
       [P1 retransmission dropped by network]

  6.                                  <--    Receive P2, SACK P2 & P3

  7a.  RACK: marks P1 retransmission lost
  7b.  Retransmit P1                  -->

  8.                                  <--         Receive P1, ACK P3
```

Figure 1. RACK-TLP protocol example

Figure 1, above, illustrates a sender sending four segments (P1, P2,
P3, P4) and losing the last three segments. After two round-trips,
TLP sends a loss probe, retransmitting the last segment, P3, to
solicit SACK feedback and restore the ACK clock (event 3). The
delivery of P3 enables RACK to infer (event 5b) that P1 and P2 were
likely lost, because they were sent before P3. The sender then
retransmits P1 and P2. Unfortunately, the retransmission of P1 is
lost again. However, the delivery of the retransmission of P2 allows
RACK to infer that the retransmission of P1 was likely lost (event
7a), and hence P1 should be retransmitted (event 7b).

3.5. An Example of RACK-TLP in Action: RTO

In addition to enhancing fast recovery, RACK improves the accuracy of RTO recovery by reducing spurious retransmissions.

Without RACK, upon RTO timer expiration the sender marks all the unacknowledged segments lost. This approach can lead to spurious retransmissions. For example, consider a simple case where one segment was sent with an RTO of 1 second, and then the application writes more data, causing a second and third segment to be sent right before the RTO of the first segment expires. Suppose only the first segment is lost. Without RACK, upon RTO expiration the sender marks all three segments as lost and retransmits the first segment. When the sender receives the ACK that selectively acknowledges the second segment, the sender spuriously retransmits the third segment.

With RACK, upon RTO timer expiration the only segment automatically marked lost is the first segment (since it was sent an RTO ago); for all the other segments RACK only marks the segment lost if at least one round trip has elapsed since the segment was transmitted. Consider the previous example scenario, this time with RACK. With RACK, when the RTO expires the sender only marks the first segment as lost, and retransmits that segment. The other two very recently sent segments are not marked lost, because they were sent less than one round trip ago and there were no ACKs providing evidence that they were lost. When the sender receives the ACK that selectively acknowledges the second segment, the sender would not retransmit the third segment but rather would send any new segments (if allowed by congestion window and receive window).

In the above example, if the sender were to send a large burst of segments instead of two segments right before RTO, without RACK the sender may spuriously retransmit almost the entire flight. Note that the Eifel protocol [RFC3522] cannot prevent this issue because it can only detect spurious RTO episodes. In this example the RTO itself was not spurious.

3.6. Design Summary

To summarize, RACK-TLP aims to adapt to small time-varying degrees of reordering, quickly recover most losses within one to two round trips, and avoid costly RTO recoveries. In the presence of reordering, the adaptation algorithm can impose sometimes-needless delays when it waits to disambiguate loss from reordering, but the penalty for waiting is bounded to one round trip and such delays are confined to flows long enough to have observed reordering.

4. Requirements

   The reader is expected to be familiar with the definitions given in
   the TCP congestion control [RFC5681] and selective acknowledgment
   [RFC2018] and loss recovery [RFC6675] RFCs. RACK-TLP has the
   following requirements:

   1. The connection MUST use selective acknowledgment (SACK) options
       [RFC2018], and the sender MUST keep SACK scoreboard information
       on a per-connection basis ("SACK scoreboard" has the same meaning
       here as in [RFC6675] section 3).

   2. For each data segment sent, the sender MUST store its most recent
       transmission time with a timestamp whose granularity that is
       finer than 1/4 of the minimum RTT of the connection. At the time
       of writing, microsecond resolution is suitable for intra-
       datacenter traffic and millisecond granularity or finer is
       suitable for the Internet. Note that RACK-TLP can be implemented
       with TSO (TCP Segmentation Offload) support by having multiple
       segments in a TSO aggregate share the same timestamp.

   3. RACK DSACK-based reordering window adaptation is RECOMMENDED but
       is not required.

   4. TLP requires RACK.

5. Definitions

   The reader is expected to be familiar with the variables of SND.UNA,
   SND.NXT, SEG.ACK, and SEG.SEQ in [RFC793], SMSS, FlightSize in
   [RFC5681], DupThresh in [RFC6675], RTO and SRTT in [RFC6298]. A
   RACK-TLP implementation needs to store new per-segment and per-
   connection state, described below.

5.1. Per-segment variables

   Theses variables indicate the status of the most recent transmission
   of a data segment:

   "Segment.lost" is true if the most recent (re)transmission of the
   segment has been marked lost and needs to be retransmitted. False
   otherwise.

   "Segment.retransmitted" is true if the segment has ever been
   retransmitted. False otherwise.

   "Segment.xmit_ts" is the time of the last transmission of a data
   segment, including retransmissions, if any, with a clock granularity

specified in the Requirements section. A maximum value INFINITE_TS
indicates an invalid timestamp that represents that the Segment is
not currently in flight.

"Segment.end_seq" is the next sequence number after the last sequence
number of the data segment.

5.2. Per-connection variables

"RACK.segment". Among all the segments that have been either
selectively or cumulatively acknowledged, RACK.segment is the one
that was sent most recently (including retransmissions).

"RACK.xmit_ts" is the latest transmission timestamp of RACK.segment.

"RACK.end_seq" is the Segment.end_seq of RACK.segment.

"RACK.ack_ts" is the time when the full sequence range of
RACK.segment was selectively or cumulatively acknowledged.

"RACK.segs_sacked" returns the total number of segments selectively
acknowledged in the SACK scoreboard.

"RACK.fack" is the highest selectively or cumulatively acknowledged
sequence (i.e. forward acknowledgement).

"RACK.min_RTT" is the estimated minimum round-trip time (RTT) of the
connection.

"RACK.rtt" is the RTT of the most recently delivered segment on the
connection (either cumulatively acknowledged or selectively
acknowledged) that was not marked invalid as a possible spurious
retransmission.

"RACK.reordering_seen" indicates whether the sender has detected data
segment reordering event(s).

"RACK.reo_wnd" is a reordering window computed in the unit of time
used for recording segment transmission times. It is used to defer
the moment at which RACK marks a segment lost.

"RACK.dsack_round" indicates if a DSACK option has been received in
the lastest round trip.

"RACK.reo_wnd_mult" is the multiplier applied to adjust RACK.reo_wnd.

"RACK.reo_wnd_persist" is the number of loss recoveries before
resetting RACK.reo_wnd.

"RACK.rtt_seq" is the SND.NXT when RACK.rtt is updated.

"TLP.is_retrans": a boolean indicating whether there is an
unacknowledged TLP retransmission.

"TLP.end_seq": the value of SND.NXT at the time of sending a TLP
retransmission.

"TLP.max_ack_delay": sender's maximum delayed ACK timer budget.

Per-connection timers

"RACK reordering timer": a timer that allows RACK to wait for
reordering to resolve, to try to disambiguate reordering from loss,
when some out-of-order segments are marked as SACKed.

"TLP PTO": a timer event indicating that an ACK is overdue and the
sender should transmit a TLP segment, to solicit SACK or ACK
feedback.

These timers augment the existing timers maintained by a sender,
including the RTO timer [RFC6298]. A RACK-TLP sender arms one of
these three timers -- RACK reordering timer, TLP PTO timer, or RTO
timer -- when it has unacknowledged segments in flight. The
implementation can simplify managing all three timers by multiplexing
a single timer among them with an additional variable to indicate the
event to invoke upon the next timer expiration.

6. RACK Algorithm Details

6.1. Upon transmitting a data segment

Upon transmitting a new segment or retransmitting an old segment,
record the time in Segment.xmit_ts and set Segment.lost to FALSE.
Upon retransmitting a segment, set Segment.retransmitted to TRUE.

```
RACK_transmit_new_data(Segment):
        Segment.xmit_ts = Now()
        Segment.lost = FALSE

RACK_retransmit_data(Segment):
        Segment.retransmitted = TRUE
        Segment.xmit_ts = Now()
        Segment.lost = FALSE
```

6.2. Upon receiving an ACK

   Step 1: Update RACK.min_RTT.

   Use the RTT measurements obtained via [RFC6298] or [RFC7323] to
   update the estimated minimum RTT in RACK.min_RTT. The sender SHOULD
   track a windowed min-filtered estimate of recent RTT measurements
   that can adapt when migrating to significantly longer paths, rather
   than a simple global minimum of all RTT measurements.

   Step 2: Update state for most recently sent segment that has been
   delivered

   In this step, RACK updates the states that track the most recently
   sent segment that has been delivered: RACK.segment; RACK maintains
   its latest transmission timestamp in RACK.xmit_ts and its highest
   sequence number in RACK.end_seq. These two variables are used, in
   later steps, to estimate if some segments not yet delivered were
   likely lost. Given the information provided in an ACK, each segment
   cumulatively ACKed or SACKed is marked as delivered in the
   scoreboard. Since an ACK can also acknowledge retransmitted data
   segments, and retransmissions can be spurious, the sender needs to
   take care to avoid spurious inferences. For example, if the sender
   were to use timing information from a spurious retransmission, the
   RACK.rtt could be vastly underestimated.

   To avoid spurious inferences, ignore a segment as invalid if any of
   its sequence range has been retransmitted before and either of two
   conditions is true:

   1. The Timestamp Echo Reply field (TSecr) of the ACK's timestamp
       option [RFC7323], if available, indicates the ACK was not
       acknowledging the last retransmission of the segment.

   2. The segment was last retransmitted less than RACK.min_rtt ago.

   The second check is a heuristic when the TCP Timestamp option is not
   available, or when the round trip time is less than the TCP Timestamp
   clock granularity.

   Among all the segments newly ACKed or SACKed by this ACK that pass
   the checks above, update the RACK.rtt to be the RTT sample calculated
   using this ACK. Furthermore, record the most recent Segment.xmit_ts
   in RACK.xmit_ts if it is ahead of RACK.xmit_ts. If Segment.xmit_ts
   equals RACK.xmit_ts (e.g. due to clock granularity limits) then
   compare Segment.end_seq and RACK.end_seq to break the tie.

   Step 2 may be summarized in pseudocode as:

```
   RACK_sent_after(t1, seq1, t2, seq2):
       If t1 > t2:
           Return true
       Else if t1 == t2 AND seq1 > seq2:
           Return true
       Else:
           Return false


   RACK_update():
       For each Segment newly acknowledged cumulatively or selectively
       in ascending order of Segment.xmit_ts:
           rtt = Now() - Segment.xmit_ts
           If Segment.retransmitted is TRUE:
               If ACK.ts_option.echo_reply < Segment.xmit_ts:
                   Continue
               If rtt < RACK.min_rtt:
                   Continue

           RACK.rtt = rtt
           If RACK_sent_after(Segment.xmit_ts, Segment.end_seq
                              RACK.xmit_ts, RACK.end_seq):
               RACK.xmit_ts = Segment.xmit_ts
               RACK.end_seq = Segment.end_seq
```

   Step 3: Detect data segment reordering

   To detect reordering, the sender looks for original data segments
   being delivered out of order. To detect such cases, the sender
   tracks the highest sequence selectively or cumulatively acknowledged
   in the RACK.fack variable. The name "fack" stands for the most
   "Forward ACK" (this term is adopted from [FACK]). If a never-
   retransmitted segment that's below RACK.fack is (selectively or
   cumulatively) acknowledged, it has been delivered out of order. The
   sender sets RACK.reordering_seen to TRUE if such segment is
   identified.

```
   RACK_detect_reordering():
       For each Segment newly acknowledged cumulatively or selectively
       in ascending order of Segment.end_seq:
           If Segment.end_seq > RACK.fack:
               RACK.fack = Segment.end_seq
           Else if Segment.end_seq < RACK.fack AND
                   Segment.retransmitted is FALSE:
               RACK.reordering_seen = TRUE
```

   Step 4: Update RACK reordering window

The RACK reordering window, RACK.reo_wnd, serves as an adaptive
allowance for settling time before marking a segment lost. This step
documents a detailed algorithm that follows the principles outlined
in the ``Reordering window adaptation'' section.

If no reordering has been observed, based on the previous step, then
one way the sender can enter Fast Recovery is when the number of
SACKed segments matches or exceeds DupThresh (similar to [RFC6675]).
Furthermore, when no reordering has been observed the RACK.reo_wnd is
set to 0 both upon entering and during Fast Recovery or RTO recovery.

Otherwise, if some reordering has been observed, then RACK does not
trigger Fast Recovery based on DupThresh.

Whether or not reordering has been observed, RACK uses the reordering
window to assess whether any segments can be marked lost. As a
consequence, the sender also enters Fast Recovery when there are any
number of SACKed segments as long as the reorder window has passed
for some non-SACKed segments.

When the reordering window is not set to 0, it starts with a
conservative RACK.reo_wnd of RACK.min_RTT/4. This value was chosen
because Linux TCP used the same factor in its implementation to delay
Early Retransmit [RFC5827] to reduce spurious loss detections in the
presence of reordering, and experience showed this worked reasonably
well [DMCG11].

However, the reordering detection in the previous step, Step 3, has a
self-reinforcing drawback when the reordering window is too small to
cope with the actual reordering. When that happens, RACK could
spuriously mark reordered segments lost, causing them to be
retransmitted. In turn, the retransmissions can prevent the
necessary conditions for Step 3 to detect reordering, since this
mechanism requires ACKs or SACKs for only segments that have never
been retransmitted. In some cases such scenarios can persist,
causing RACK to continue to spuriously mark segments lost without
realizing the reordering window is too small.

To avoid the issue above, RACK dynamically adapts to higher degrees
of reordering using DSACK options from the receiver. Receiving an
ACK with a DSACK option indicates a possible spurious retransmission,
suggesting that RACK.reo_wnd may be too small. The RACK.reo_wnd
increases linearly for every round trip in which the sender receives
some DSACK option, so that after N distinct round trips in which a
DSACK is received, the RACK.reo_wnd becomes (N+1) * min_RTT / 4, with
an upper-bound of SRTT.

If the reordering is temporary then a large adapted reordering window
would unnecessarily delay loss recovery later. Therefore, RACK
persists using the inflated RACK.reo_wnd for up to 16 loss
recoveries, after which it resets RACK.reo_wnd to its starting value,
min_RTT / 4. The downside of resetting the reordering window is the
risk of triggering spurious fast recovery episodes if the reordering
remains high. The rationale for this approach is to bound such
spurious recoveries to approximately once every 16 recoveries (less
than 7%).

To track the linear scaling factor for the adaptive reordering
window, RACK uses the variable RACK.reo_wnd_mult, which is
initialized to 1 and adapts with observed reordering.

The following pseudocode implements the above algorithm for updating
the RACK reordering window:

```
RACK_update_reo_wnd():

    /* DSACK-based reordering window adaptation */
    If RACK.dsack_round is not None AND
       SND.UNA >= RACK.dsack_round:
        RACK.dsack_round = None
    /* Grow the reordering window per round that sees DSACK.
       Reset the window after 16 DSACK-free recoveries */
    If RACK.dsack_round is None AND
       any DSACK option is present on latest received ACK:
        RACK.dsack_round = SND.NXT
        RACK.reo_wnd_mult += 1
        RACK.reo_wnd_persist = 16
    Else if exiting Fast or RTO recovery:
        RACK.reo_wnd_persist -= 1
        If RACK.reo_wnd_persist <= 0:
            RACK.reo_wnd_mult = 1

    If RACK.reordering_seen is FALSE:
        If in Fast or RTO recovery:
            Return 0
        Else if RACK.segs_sacked >= DupThresh:
            Return 0
    Return min(RACK.min_RTT / 4 * RACK.reo_wnd_mult, SRTT)
```

Step 5: Detect losses.

For each segment that has not been SACKed, RACK considers that
segment lost if another segment that was sent later has been
delivered, and the reordering window has passed. RACK considers the
reordering window to have passed if the RACK.segment was sent

sufficiently after the segment in question, or a sufficient time has
elapsed since the RACK.segment was S/ACKed, or some combination of
the two. More precisely, RACK marks a segment lost if:

```
 RACK.xmit_ts >= Segment.xmit_ts
         AND
 RACK.xmit_ts - Segment.xmit_ts + (now - RACK.ack_ts) >= RACK.reo_wnd
```

Solving this second condition for "now", the moment at which a
segment is marked lost, yields:

```
now >= Segment.xmit_ts + RACK.reo_wnd + (RACK.ack_ts - RACK.xmit_ts)
```

Then (RACK.ack_ts - RACK.xmit_ts) is the round trip time of the most
recently (re)transmitted segment that's been delivered. When
segments are delivered in order, the most recently (re)transmitted
segment that's been delivered is also the most recently delivered,
hence RACK.rtt == RACK.ack_ts - RACK.xmit_ts. But if segments were
reordered, then the segment delivered most recently was sent before
the most recently (re)transmitted segment. Hence RACK.rtt >
(RACK.ack_ts - RACK.xmit_ts).

Since RACK.RTT >= (RACK.ack_ts - RACK.xmit_ts), the previous equation
reduces to saying that the sender can declare a segment lost when:

```
now >= Segment.xmit_ts + RACK.reo_wnd + RACK.rtt
```

In turn, that is equivalent to stating that a RACK sender should
declare a segment lost when:

```
Segment.xmit_ts + RACK.rtt + RACK.reo_wnd - now <= 0
```

Note that if the value on the left hand side is positive, it
represents the remaining wait time before the segment is deemed lost.
But this risks a timeout (RTO) if no more ACKs come back (e.g., due
to losses or application-limited transmissions) to trigger the
marking. For timely loss detection, the sender is RECOMMENDED to
install a reordering timer. This timer expires at the earliest
moment when RACK would conclude that all the unacknowledged segments
within the reordering window were lost.

The following pseudocode implements the algorithm above. When an ACK
is received or the RACK reordering timer expires, call
RACK_detect_loss_and_arm_timer(). The algorithm breaks timestamp
ties by using the TCP sequence space, since high-speed networks often
have multiple segments with identical timestamps.

```
RACK_detect_loss():
    timeout = 0
    RACK.reo_wnd = RACK_update_reo_wnd()
    For each segment, Segment, not acknowledged yet:
        If RACK_sent_after(RACK.xmit_ts, RACK.end_seq,
                           Segment.xmit_ts, Segment.end_seq):
            remaining = Segment.xmit_ts + RACK.rtt +
                        RACK.reo_wnd - Now()
            If remaining <= 0:
                Segment.lost = TRUE
                Segment.xmit_ts = INFINITE_TS
            Else:
                timeout = max(remaining, timeout)
    Return timeout


RACK_detect_loss_and_arm_timer():
    timeout = RACK_detect_loss()
    If timeout != 0
        Arm the RACK timer to call
        RACK_detect_loss_and_arm_timer() after timeout
```

As an optimization, an implementation can choose to check only
segments that have been sent before RACK.xmit_ts. This can be more
efficient than scanning the entire SACK scoreboard, especially when
there are many segments in flight. The implementation can use a
separate doubly-linked list ordered by Segment.xmit_ts and inserts a
segment at the tail of the list when it is (re)transmitted, and
removes a segment from the list when it is delivered or marked lost.
In Linux TCP this optimization improved CPU usage by orders of
magnitude during some fast recovery episodes on high-speed WAN
networks.

6.3. Upon RTO expiration

Upon RTO timer expiration, RACK marks the first outstanding segment
as lost (since it was sent an RTO ago); for all the other segments
RACK only marks the segment lost if the time elapsed since the
segment was transmitted is at least the sum of the recent RTT and the
reordering window.

```
RACK_mark_losses_on_RTO():
    For each segment, Segment, not acknowledged yet:
        If SEG.SEQ == SND.UNA OR
            Segment.xmit_ts + RACK.rtt + RACK.reo_wnd - Now() <= 0:
             Segment.lost = TRUE
```

7. TLP Algorithm Details

7.1. Initializing state

   Reset TLP.is_retrans and TLP.end_seq when initiating a connection,
   fast recovery, or RTO recovery.

   TLP_init():
       TLP.end_seq = None
       TLP.is_retrans = false

7.2. Scheduling a loss probe

   The sender schedules a loss probe timeout (PTO) to transmit a segment
   during the normal transmission process. The sender SHOULD start or
   restart a loss probe PTO timer after transmitting new data (that was
   not itself a loss probe) or upon receiving an ACK that cumulatively
   acknowledges new data, unless it is already in fast recovery, RTO
   recovery, or the sender has segments delivered out-of-order (i.e.
   RACK.segs_sacked is not zero). These conditions are excluded because
   they are addressed by similar mechanisms, like Limited Transmit
   [RFC3042], the RACK reordering timer, and F-RTO [RFC5682].

   The sender calculates the PTO interval by taking into account a
   number of factors.

   First, the default PTO interval is 2*SRTT. By that time, it is
   prudent to declare that an ACK is overdue, since under normal
   circumstances, i.e. no losses, an ACK typically arrives in one SRTT.
   Choosing PTO to be exactly an SRTT would risk causing spurious
   probes, given that network and end-host delay variance can cause an
   ACK to be delayed beyond SRTT. Hence the PTO is conservatively
   chosen to be the next integral multiple of SRTT.

   Second, when there is no SRTT estimate available, the PTO SHOULD be 1
   second. This conservative value corresponds to the RTO value when no
   SRTT is available, per [RFC6298].

   Third, when FlightSize is one segment, the sender MAY inflate PTO by
   TLP.max_ack_delay to accommodate a potential delayed acknowledgment
   and reduce the risk of spurious retransmissions. The actual value of
   TLP.max_ack_delay is implementation-specific.

   Finally, if the time at which an RTO would fire (here denoted
   "TCP_RTO_expiration()") is sooner than the computed time for the PTO,
   then the sender schedules a TLP to be sent at that RTO time.

Summarizing these considerations in pseudocode form, a sender SHOULD
use the following logic to select the duration of a PTO:

```
TLP_calc_PTO():
    If SRTT is available:
        PTO = 2 * SRTT
        If FlightSize is one segment:
            PTO += TLP.max_ack_delay
    Else:
        PTO = 1 sec

    If Now() + PTO > TCP_RTO_expiration():
        PTO = TCP_RTO_expiration() - Now()
```

7.3. Sending a loss probe upon PTO expiration

When the PTO timer expires, the sender SHOULD transmit a previously
unsent data segment, if the receive window allows, and increment the
FlightSize accordingly. Note that FlightSize could be one packet
greater than the congestion window temporarily until the next ACK
arrives.

If such a segment is not available, then the sender SHOULD retransmit
the highest-sequence segment sent so far and set TLP.is_retrans to
true. This segment is chosen to deal with the retransmission
ambiguity problem in TCP. Suppose a sender sends N segments, and
then retransmits the last segment (segment N) as a loss probe, and
then the sender receives a SACK for segment N. As long as the sender
waits for the RACK reordering window to expire, it doesn't matter if
that SACK was for the original transmission of segment N or the TLP
retransmission; in either case the arrival of the SACK for segment N
provides evidence that the N-1 segments preceding segment N were
likely lost.

In the case where there is only one original outstanding segment of
data (N=1), the same logic (trivially) applies: an ACK for a single
outstanding segment tells the sender the N-1=0 segments preceding
that segment were lost. Furthermore, whether there are N>1 or N=1
outstanding segments, there is a question about whether the original
last segment or its TLP retransmission were lost; the sender
estimates whether there was such a loss using TLP recovery detection
(see below).

The sender MUST follow the RACK transmission procedures in the ''Upon
Transmitting a Data Segment'' section (see above) upon sending either
a retransmission or new data loss probe. This is critical for
detecting losses using the ACK for the loss probe. Furthermore,
prior to sending a loss probe, the sender MUST check that there is no

other previous loss probe still in flight. This ensures that at any
given time the sender has at most one additional packet in flight
beyond the congestion window limit. This invariant is maintained
using the state variable TLP.end_seq, which indicates the latest
unacknowledged TLP loss probe's ending sequence. It is reset when
the loss probe has been acknowledged or is deemed lost or irrelevant.
After attempting to send a loss probe, regardless of whether a loss
probe was sent, the sender MUST re-arm the RTO timer, not the PTO
timer, if FlightSize is not zero. This ensures RTO recovery remains
the last resort if TLP fails. The following pseudo code summarizes
the operations.

```
TLP_send_probe():

    If TLP.end_seq is None:
        TLP.is_retrans = false
        Segment = send buffer segment starting at SND.NXT
        If Segment exists and fits the peer receive window limit:
            /* Transmit the lowest-sequence unsent Segment */
            Transmit Segment
            RACK_transmit_data(Segment)
            TLP.end_seq = SND.NXT
            Increase FlightSize by Segment length
        Else:
            /* Retransmit the highest-sequence Segment sent */
            Segment = send buffer segment ending at SND.NXT
            Transmit Segment
            RACK_retransmit_data(Segment)
            TLP.end_seq = SND.NXT
```

7.4. Detecting losses using the ACK of the loss probe

   When there is packet loss in a flight ending with a loss probe, the
   feedback solicited by a loss probe will reveal one of two scenarios,
   depending on the pattern of losses.

7.4.1. General case: detecting packet losses using RACK

   If the loss probe and the ACK that acknowledges the probe are
   delivered successfully, RACK-TLP uses this ACK -- just as it would
   with any other ACK -- to detect if any segments sent prior to the
   probe were dropped. RACK would typically infer that any
   unacknowledged data segments sent before the loss probe were lost,
   since they were sent sufficiently far in the past (at least one PTO
   has elapsed, plus one round-trip for the loss probe to be ACKed).
   More specifically, RACK_detect_loss() (step 5) would mark those
   earlier segments as lost. Then the sender would trigger a fast
   recovery to recover those losses.

7.4.2. **Special case: detecting a single loss repaired by the loss probe**

   If the TLP retransmission repairs all the lost in-flight sequence
   ranges (i.e. only the last segment in the flight was lost), the ACK
   for the loss probe appears to be a regular cumulative ACK, which
   would not normally trigger the congestion control response to this
   packet loss event. The following TLP recovery detection mechanism
   examines ACKs to detect this special case to make congestion control
   respond properly [RFC5681].

   After a TLP retransmission, the sender checks for this special case
   of a single loss that is recovered by the loss probe itself. To
   accomplish this, the sender checks for a duplicate ACK or DSACK
   indicating that both the original segment and TLP retransmission
   arrived at the receiver, meaning there was no loss. If the TLP
   sender does not receive such an indication, then it MUST assume that
   either the original data segment, the TLP retransmission, or a
   corresponding ACK were lost, for congestion control purposes.

   If the TLP retransmission is spurious, a receiver that uses DSACK
   would return an ACK that covers TLP.end_seq with a DSACK option (Case
   1). If the receiver does not support DSACK, it would return a DUPACK
   without any SACK option (Case 2). If the sender receives an ACK
   matching either case, then the sender estimates that the receiver
   received both the original data segment and the TLP probe
   retransmission, and so the sender considers the TLP episode to be
   done, and records that fact by setting TLP.end_seq to None.

   Upon receiving an ACK that covers some sequence number after
   TLP.end_seq, the sender should have received any ACKs for the
   original segment and TLP probe retransmission segment. At that time,
   if the TLP.end_seq is still set, and thus indicates that the TLP
   probe retransmission remains unacknowledged, then the sender should
   presume that at least one of its data segments was lost. The sender
   then SHOULD invoke a congestion control response equivalent to a fast
   recovery.

   More precisely, on each ACK the sender executes the following:

```
TLP_process_ack(ACK):
    If TLP.end_seq is not None AND ACK's ack. number >= TLP.end_seq:
        If not TLP.is_retrans:
            TLP.end_seq = None    /* TLP of new data delivered */
        Else if ACK has a DSACK option matching TLP.end_seq:
            TLP.end_seq = None    /* Case 1, above */
        Else If ACK's ack. number > TLP.end_seq:
            TLP.end_seq = None    /* Repaired the single loss */
            (Invoke congestion control to react to
             the loss event the probe has repaired)
        Else If ACK is a DUPACK without any SACK option:
            TLP.end_seq = None     /* Case 2, above */
```

8. Managing RACK-TLP timers

The RACK reordering, the TLP PTO timer, the RTO and Zero Window Probe
(ZWP) timer [RFC793] are mutually exclusive and used in different
scenarios. When arming a RACK reordering timer or TLP PTO timer, the
sender SHOULD cancel any other pending timer(s). An implementation
is to have one timer with an additional state variable indicating the
type of the timer.

9. Discussion

9.1. Advantages and disadvantages

The biggest advantage of RACK-TLP is that every data segment, whether
it is an original data transmission or a retransmission, can be used
to detect losses of the segments sent chronologically prior to it.
This enables RACK-TLP to use fast recovery in cases with application-
limited flights of data, lost retransmissions, or data segment
reordering events. Consider the following examples:

1. Packet drops at the end of an application data flight: Consider a
    sender that transmits an application-limited flight of three data
    segments (P1, P2, P3), and P1 and P3 are lost. Suppose the
    transmission of each segment is at least RACK.reo_wnd after the
    transmission of the previous segment. RACK will mark P1 as lost
    when the SACK of P2 is received, and this will trigger the
    retransmission of P1 as R1. When R1 is cumulatively
    acknowledged, RACK will mark P3 as lost and the sender will
    retransmit P3 as R3. This example illustrates how RACK is able
    to repair certain drops at the tail of a transaction without an
    RTO recovery. Notice that neither the conventional duplicate ACK
    threshold [RFC5681], nor [RFC6675], nor the Forward
    Acknowledgment [FACK] algorithm can detect such losses, because
    of the required segment or sequence count.

2. Lost retransmission: Consider a flight of three data segments
    (P1, P2, P3) that are sent; P1 and P2 are dropped. Suppose the
    transmission of each segment is at least RACK.reo_wnd after the
    transmission of the previous segment. When P3 is SACKed, RACK
    will mark P1 and P2 lost and they will be retransmitted as R1 and
    R2. Suppose R1 is lost again but R2 is SACKed; RACK will mark R1
    lost and trigger retransmission again. Again, neither the
    conventional three duplicate ACK threshold approach, nor
    [RFC6675], nor the Forward Acknowledgment [FACK] algorithm can
    detect such losses. And such a lost retransmission can happen
    when TCP is being rate-limited, particularly by token bucket
    policers with large bucket depth and low rate limit; in such
    cases retransmissions are often lost repeatedly because standard
    congestion control requires multiple round trips to reduce the
    rate below the policed rate.

3. Packet reordering: Consider a simple reordering event where a
    flight of segments are sent as (P1, P2, P3). P1 and P2 carry a
    full payload of MSS octets, but P3 has only a 1-octet payload.
    Suppose the sender has detected reordering previously and thus
    RACK.reo_wnd is min_RTT/4. Now P3 is reordered and delivered
    first, before P1 and P2. As long as P1 and P2 are delivered
    within min_RTT/4, RACK will not consider P1 and P2 lost. But if
    P1 and P2 are delivered outside the reordering window, then RACK
    will still spuriously mark P1 and P2 lost.

The examples above show that RACK-TLP is particularly useful when the
sender is limited by the application, which can happen with
interactive or request/response traffic. Similarly, RACK still works
when the sender is limited by the receive window, which can happen
with applications that use the receive window to throttle the sender.

RACK-TLP works more efficiently with TCP Segmentation Offload (TSO)
compared to DUPACK-counting. RACK always marks the entire TSO
aggregate lost because the segments in the same TSO aggregate have
the same transmission timestamp. By contrast, the algorithms based
on sequence counting (e.g., [RFC6675] [RFC5681]) may mark only a
subset of segments in the TSO aggregate lost, forcing the stack to
perform expensive fragmentation of the TSO aggregate, or to
selectively tag individual segments lost in the scoreboard.

The main drawback of RACK-TLP is the additional states required
compared to DUPACK-counting. RACK requires the sender to record the
transmission time of each segment sent at a clock granularity that is
finer than 1/4 of the minimum RTT of the connection. TCP
implementations that record this already for RTT estimation do not
require any new per-packet state. But implementations that are not
yet recording segment transmission times will need to add per-packet

internal state (expected to be either 4 or 8 octets per segment or
TSO aggregate) to track transmission times. In contrast, [RFC6675]
loss detection approach does not require any per-packet state beyond
the SACK scoreboard; this is particularly useful on ultra-low RTT
networks where the RTT may be less than the sender TCP clock
granularity (e.g. inside data-centers). Another disadvantage is the
reordering timer may expire prematurely (like any other
retransmission timer) to cause higher spurious retransmission
especially if DSACK is not supported.

9.2. Relationships with other loss recovery algorithms

The primary motivation of RACK-TLP is to provide a general
alternative to some of the standard loss recovery algorithms
[RFC5681] [RFC6675] [RFC5827] [RFC4653]. [RFC5827] [RFC4653]
dynamically adjusts the duplicate ACK threshold based on the current
or previous flight sizes. RACK-TLP takes a different approach by
using a time-based reordering window. RACK-TLP can be seen as an
extended Early Retransmit [RFC5827] without a FlightSize limit but
with an additional reordering window. [FACK] considers an original
segment to be lost when its sequence range is sufficiently far below
the highest SACKed sequence. In some sense RACK-TLP can be seen as a
generalized form of FACK that operates in time space instead of
sequence space, enabling it to better handle reordering, application-
limited traffic, and lost retransmissions.

RACK-TLP is compatible with the standard RTO [RFC6298], RTO-restart
[RFC7765], F-RTO [RFC5682] and Eifel algorithms [RFC3522]. This is
because RACK-TLP only detects loss by using ACK events. It neither
changes the RTO timer calculation nor detects spurious RTO. RACK-TLP
slightly changes the retransmission behavior of [RFC6298] by
preceding the RTO with TLP and reducing potential spurious
retransmission after RTO.

9.3. Interaction with congestion control

RACK-TLP intentionally decouples loss detection from congestion
control. RACK-TLP only detects losses; it does not modify the
congestion control algorithm [RFC5681] [RFC6937]. A segment marked
lost by RACK-TLP MUST NOT be retransmitted until congestion control
deems this appropriate.

The only exception -- the only way in which RACK-TLP modulates the
congestion control algorithm -- is that one outstanding loss probe
can be sent even if the congestion window is fully used. However,
this temporary over-commit is accounted for and credited in the in-
flight data tracked for congestion control, so that congestion
control will erase the over-commit upon the next ACK.

If packet losses happen after the reordering window has been
increased by DSACK, RACK-TLP may take longer to detect losses than
the pure DUPACK-counting approach. In this case TCP may continue to
increase the congestion window upon receiving ACKs during this time,
making the sender more aggressive.

The following simple example compares how RACK-TLP and non-RACK-TLP
loss detection interacts with congestion control: suppose a sender
has a congestion window (cwnd) of 20 segments on a SACK-enabled
connection. It sends 10 data segments and all of them are lost.

Without RACK-TLP, the sender would time out, reset cwnd to 1, and
retransmit the first segment. It would take four round trips (1 + 2
+ 4 + 3 = 10) to retransmit all the 10 lost segments using slow
start. The recovery latency would be RTO + 4*RTT, with an ending
cwnd of 4 segments due to congestion window validation.

With RACK-TLP, a sender would send the TLP after 2*RTT and get a
DUPACK, enabling RACK to detect the losses and trigger fast recovery.
If the sender implements Proportional Rate Reduction [RFC6937] it
would slow start to retransmit the remaining 9 lost segments since
the number of segments in flight (0) is lower than the slow start
threshold (10). The slow start would again take four round trips (1
+ 2 + 4 + 3 = 10) to retransmit all the lost segments. The recovery
latency would be 2*RTT + 4*RTT, with an ending cwnd set to the slow
start threshold of 10 segments.

The difference in recovery latency (RTO + 4*RTT vs 6*RTT) can be
significant if the RTT is much smaller than the minimum RTO (1 second
in [RFC6298]) or if the RTT is large. The former case can happen in
local area networks, data-center networks, or content distribution
networks with deep deployments. The latter case can happen in
developing regions with highly congested and/or high-latency
networks.

9.4. TLP recovery detection with delayed ACKs

Delayed or stretched ACKs complicate the detection of repairs done by
TLP, since with such ACKs the sender takes longer time to receive
fewer ACKs than would normally be expected. To mitigate this
complication, before sending a TLP loss probe retransmission, the
sender should attempt to wait long enough that the receiver has sent
any delayed ACKs that it is withholding. The sender algorithm
described above features such a delay, in the form of
TLP.max_ack_delay. Furthermore, if the receiver supports DSACK then
in the case of a delayed ACK the sender's TLP recovery detection
mechanism (see above) can use the DSACK information to infer that the
original and TLP retransmission both arrived at the receiver.

If there is ACK loss or a delayed ACK without a DSACK, then this
algorithm is conservative, because the sender will reduce the
congestion window when in fact there was no packet loss. In practice
this is acceptable, and potentially even desirable: if there is
reverse path congestion then reducing the congestion window can be
prudent.

9.5. RACK for other transport protocols

RACK can be implemented in other transport protocols (e.g.,
[QUICLR]). The [Sprout] loss detection algorithm was also
independently designed to use a 10ms reordering window to improve its
loss detection.

10. Security Considerations

RACK-TLP algorithm behavior is based on information conveyed in SACK
options, so it has security considerations similar to those described
in the Security Considerations section of [RFC6675].

Additionally, RACK-TLP has a lower risk profile than [RFC6675]
because it is not vulnerable to ACK-splitting attacks [SCWA99]: for
an MSS-size segment sent, the receiver or the attacker might send MSS
ACKs that SACK or acknowledge one additional byte per ACK. This
would not fool RACK. In such a scenario, RACK.xmit_ts would not
advance, because all the sequence ranges within the segment were
transmitted at the same time, and thus carry the same transmission
timestamp. In other words, SACKing only one byte of a segment or
SACKing the segment in entirety have the same effect with RACK.

11. IANA Considerations

This document makes no request of IANA.

Note to RFC Editor: this section may be removed on publication as an
RFC.

12. Acknowledgments

The authors thank Matt Mathis for his insights in FACK and Michael
Welzl for his per-packet timer idea that inspired this work. Eric
Dumazet, Randy Stewart, Van Jacobson, Ian Swett, Rick Jones, Jana
Iyengar, Hiren Panchasara, Praveen Balasubramanian, Yoshifumi
Nishida, Bob Briscoe, Felix Weinrank, Michael Tuexen, Martin Duke,
Ilpo Jarvinen, Theresa Enghardt, Mirja Kuehlewind, Gorry Fairhurst,
and Yi Huang contributed to the draft or the implementations in
Linux, FreeBSD, Windows, and QUIC.

13. References

13.1. Normative References

[RFC2018] Mathis, M. and J. Mahdavi, "TCP Selective Acknowledgment
           Options", RFC 2018, October 1996.

[RFC2119] Bradner, S., "Key words for use in RFCs to Indicate
           Requirement Levels", RFC 2119, March 1997.

[RFC2883] Floyd, S., Mahdavi, J., Mathis, M., and M. Podolsky, "An
           Extension to the Selective Acknowledgement (SACK) Option
           for TCP", RFC 2883, July 2000.

[RFC3042] Allman, M., Balakrishnan, H., and S. Floyd, "Enhancing
           TCP's Loss Recovery Using Limited Transmit", January 2001.

[RFC5681] Allman, M., Paxson, V., and E. Blanton, "TCP Congestion
           Control", RFC 5681, September 2009.

[RFC6298] Paxson, V., Allman, M., Chu, J., and M. Sargent,
           "Computing TCP's Retransmission Timer", RFC 6298, June
           2011.

[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, August 2012.

[RFC7323] Borman, D., Braden, B., Jacobson, V., and R.
           Scheffenegger, "TCP Extensions for High Performance",
           September 2014.

[RFC793]   Postel, J., "Transmission Control Protocol", September
           1981.

[RFC8174] Leiba, B., "Ambiguity of Uppercase vs Lowercase in RFC
           2119 Key Words", May 2017.

13.2. Informative References

[DMCG11]   Dukkipati, N., Matthis, M., Cheng, Y., and M. Ghobadi,
           "Proportional Rate Reduction for TCP", ACM SIGCOMM
           Conference on Internet Measurement , 2011.

   [FACK]   Mathis, M. and M. Jamshid, "Forward acknowledgement:
               refining TCP congestion control", ACM SIGCOMM Computer
               Communication Review, Volume 26, Issue 4, Oct. 1996. ,
               1996.

   [POLICER16]
               Flach, T., Papageorge, P., Terzis, A., Pedrosa, L., Cheng,
               Y., Karim, T., Katz-Bassett, E., and R. Govindan, "An
               Analysis of Traffic Policing in the Web", ACM SIGCOMM ,
               2016.

   [QUICLR]   Iyengar, J. and I. Swett, "QUIC Loss Detection and
               Congestion Control", draft-ietf-quic-recovery (work in
               progress), Octobor 2020.

   [RFC3522]   Ludwig, R. and M. Meyer, "The Eifel Detection Algorithm
               for TCP", April 2003.

   [RFC4653]   Bhandarkar, S., Reddy, A., Allman, M., and E. Blanton,
               "Improving the Robustness of TCP to Non-Congestion
               Events", August 2006.

   [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,
               September 2009.

   [RFC5827]   Allman, M., Ayesta, U., Wang, L., Blanton, J., and P.
               Hurtig, "Early Retransmit for TCP and Stream Control
               Transmission Protocol (SCTP)", RFC 5827, April 2010.

   [RFC6937]   Mathis, M., Dukkipati, N., and Y. Cheng, "Proportional
               Rate Reduction for TCP", May 2013.

   [RFC7765]   Hurtig, P., Brunstrom, A., Petlund, A., and M. Welzl, "TCP
               and SCTP RTO Restart", February 2016.

   [SCWA99]   Savage, S., Cardwell, N., Wetherall, D., and T. Anderson,
               "TCP Congestion Control With a Misbehaving Receiver", ACM
               Computer Communication Review, 29(5) , 1999.

   [Sprout]   Winstein, K., Sivaraman, A., and H. Balakrishnan,
               "Stochastic Forecasts Achieve High Throughput and Low
               Delay over Cellular Networks", USENIX Symposium on
               Networked Systems Design and Implementation (NSDI) , 2013.

Authors' Addresses

    Yuchung Cheng
    Google, Inc

    Email: ycheng@google.com

    Neal Cardwell
    Google, Inc

    Email: ncardwell@google.com

    Nandita Dukkipati
    Google, Inc

    Email: nanditad@google.com

    Priyaranjan Jha
    Google, Inc

    Email: priyarjha@google.com