TCP Maintenance Working Group
Internet-Draft
Intended status: Standards Track
Expires: September 10, 2020

Y. Cheng
N. Cardwell
N. Dukkipati
P. Jha
Google, Inc

March 9, 2020

# RACK: a time-based fast loss detection algorithm for TCP
## draft-ietf-tcpm-rack-08

Abstract

This document presents a new TCP loss detection algorithm called RACK ("Recent ACKnowledgment"). RACK uses the notion of time, instead of packet or sequence counts, to detect losses, for modern TCP implementations that can support per-packet timestamps and the selective acknowledgment (SACK) option. It is intended to be an alternative to the DUPACK threshold approach [RFC6675], as well as other nonstandard approaches such as FACK [FACK].

Status of This Memo



Copyright Notice

## 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 a new loss detection algorithm called RACK ("Recent ACKnowledgment"). RACK uses the notion of time instead of the conventional packet or sequence counting approaches for detecting losses. RACK deems a packet lost if it has not been delivered and some packet sent sufficiently later has been delivered. It does this by recording packet transmission times and inferring losses using cumulative acknowledgments or selective acknowledgment (SACK) TCP options.

In recent years we have been observing several increasingly common loss and reordering patterns in the Internet:

1. Slow recovery due to lost retransmissions. Traffic policers [POLICER16] and burst losses often cause retransmissions to be lost again. This severely increases latency because the lost retransmissions can only be recovered by retransmission timeouts (RTOs).

2. Tail drops. Structured request-response traffic turns more losses into tail drops. In such cases, TCP is application-limited, so it cannot send new data to probe losses and has to rely on retransmission timeouts (RTOs).

3. Reordering. Link-layer protocols (e.g., 802.11 block ACK), link bonding, or routers' internal load-balancing can deliver TCP packets out of order. The degree of such reordering is usually within the order of the path round trip time.

Despite TCP stacks (e.g. Linux) that implement many of the standard and proposed loss detection algorithms

[RFC4653][RFC5827][RFC5681][RFC6675][RFC7765][FACK][THIN-STREAM], we've found that together they do not perform well. The main reason is that many of them are based on the classic rule of counting duplicate acknowledgments [RFC5681]. They can either detect loss quickly or accurately, but not both, especially when the sender is application-limited or under reordering that is unpredictable. And under these conditions none of them can detect lost retransmissions well.

Also, these algorithms, including RFCs, rarely address the interactions with other algorithms. For example, FACK may consider a packet is lost while RFC6675 may not. Implementing N algorithms while dealing with N^2 interactions is a daunting task and error-prone.

The goal of RACK is to solve all the problems above by replacing many of the loss detection algorithms above with one more effective algorithm to handle loss and reordering.

## 3. Overview

The main idea behind RACK is that if a packet has been delivered out of order, then the packets sent chronologically before that were either lost or reordered. This concept is not fundamentally different from [RFC5681][RFC6675][FACK]. But the key innovation in RACK is to use a per-packet transmission timestamp and widely deployed SACK options to conduct time-based inferences instead of inferring losses with packet or sequence counting approaches.

Using a threshold for counting duplicate acknowledgments (i.e., DupThresh) alone is no longer reliable because of today's prevalent reordering patterns. A common type of reordering is that the last "runt" packet of a window's worth of packet bursts gets delivered first, then the rest arrive shortly after in order. To handle this effectively, a sender would need to constantly adjust the DupThresh to the burst size; but this would risk increasing the frequency of RTOs on real losses.

Today's prevalent lost retransmissions also cause problems with packet-counting approaches [RFC5681][RFC6675][FACK], since those approaches depend on reasoning in sequence number space. Retransmissions break the direct correspondence between ordering in sequence space and ordering in time. So when retransmissions are lost, sequence-based approaches are often unable to infer and quickly repair losses that can be deduced with time-based approaches.

Instead of counting packets, RACK uses the most recently delivered packet's transmission time to judge if some packets sent previous to

that time have "expired" by passing a certain reordering settling window. On each ACK, RACK marks any already-expired packets lost, and for any packets that have not yet expired it waits until the reordering window passes and then marks those lost as well. In either case, RACK can repair the loss without waiting for a (long) RTO. RACK can be applied to both fast recovery and timeout recovery, and can detect losses on both originally transmitted and retransmitted packets, making it a great all-weather loss detection mechanism.

## 4. Design Rationale for Reordering Tolerance

The reordering behavior of networks can evolve (over years) in response to the behavior of transport protocols and applications, as well as the needs of network designers and operators. From a network or link designer's viewpoint, parallelization (eg. link bonding) is the easiest way to get a network to go faster. Therefore their main constraint on speed is reordering, and there is pressure to relax that constraint. If RACK becomes widely deployed, the underlying networks may introduce more reordering for higher throughput. But this may result in excessive reordering that hurts end to end performance:

1. End host packet processing: extreme reordering on high-speed networks would incur high CPU cost by greatly reducing the effectiveness of aggregation mechanisms, such as large receive offload (LRO) and generic receive offload (GRO), and significantly increasing the number of ACKs.

2. Congestion control: TCP congestion control implicitly assumes the feedback from ACKs are from the same bottleneck. Therefore it cannot handle well scenarios where packets are traversing largely disjoint paths.

3. Loss recovery: Having an excessively large reordering window to accommodate widely different latencies from different paths would increase the latency of loss recovery.

An end-to-end transport protocol cannot tell immediately whether a hole is reordering or loss. It can only distinguish between the two in hindsight if the hole in the sequence space gets filled later without a retransmission. How long the sender waits for such potential reordering events to settle is determined by the current reordering window.

Given these considerations, a core design philosophy of RACK is to adapt to the measured duration of reordering events, within

reasonable and specific bounds. To accomplish this RACK places the following mandates on the reordering window:

1. The initial RACK reordering window SHOULD be set to a small fraction of the round-trip time.

2. If no reordering has been observed, then RACK SHOULD honor the classic 3-DUPACK rule for initiating fast recovery. One simple way to implement this is to temporarily override the reorder window to 0.

3. The RACK reordering window SHOULD leverage Duplicate Selective Acknowledgement (DSACK) information [RFC3708] to adaptively estimate the duration of reordering events.

4. The RACK reordering window MUST be bounded and this bound SHOULD be one round trip.

As a flow starts, either condition 1 or condition 2 or both would trigger RACK to start the recovery process quickly. The low initial reordering window and use of the 3-DUPACK rule are key to achieving low-latency loss recovery for short flows by risking spurious retransmissions to recover losses quickly. This rationale is that spurious retransmissions for short flows are not expected to produce excessive network traffic.

For long flows the design tolerates reordering within a round trip. This handles reordering caused by path divergence in small time scales (reordering within the round-trip time of the shortest path), which should tolerate much of the reordering from link bonding, multipath routing, or link-layer out-of-order delivery. It also relaxes ordering constraints to allow sending flights of TCP packets on different paths dynamically for better load-balancing (e.g. flowlets).

However, the fact that the initial RACK reordering window is low, and the RACK reordering window's adaptive growth is bounded, means that there will continue to be a cost to reordering and a limit to RACK's adaptation to reordering. This maintains a disincentive for network designers and operators to introduce needless or excessive reordering, particularly because they have to allow for low round trip time paths. This means RACK will not encourage networks to perform inconsiderate fine-grained packet-spraying over highly disjoint paths with very different characteristics. There are good alternative solutions, such as MPTCP, for such networks.

To conclude, the RACK algorithm aims to adapt to small degrees of reordering, quickly recover most losses within one to two round

trips, and avoid costly retransmission timeouts (RTOs). 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 longer-running flows.

## 5. Requirements

The reader is expected to be familiar with the definitions given in the TCP congestion control [RFC5681] and selective acknowledgment [RFC2018] RFCs. Familiarity with the conservative SACK-based recovery for TCP [RFC6675] is not expected but helps.

RACK has three requirements:

1. The connection MUST use selective acknowledgment (SACK) options [RFC2018].

2. For each packet sent, the sender MUST store its most recent transmission time with (at least) millisecond granularity. For round-trip times lower than a millisecond (e.g., intra-datacenter communications) microsecond granularity would significantly help the detection latency but is not required.

3. For each packet sent, the sender MUST remember whether the packet has been retransmitted or not.

We assume that requirement 1 implies the sender keeps a SACK scoreboard, which is a data structure to store selective acknowledgment information on a per-connection basis ([RFC6675] section 3). For the ease of explaining the algorithm, we use a pseudo-scoreboard that manages the data in sequence number ranges. But the specifics of the data structure are left to the implementor.

RACK does not need any change on the receiver.

## 6. Definitions

The reader is expected to be familiar with the definitions given in [RFC793], including SND.UNA, SND.NXT, SEG.ACK, and SEG.SEQ.

### 6.1. Definitions of variables

A sender implementing RACK needs to store these new RACK variables:

"Packet.xmit_ts" is the time of the last transmission of a data packet, including retransmissions, if any. The sender needs to record the transmission time for each packet sent and not yet

acknowledged. The time MUST be stored at millisecond granularity or finer.

"RACK.packet". Among all the packets that have been either selectively or cumulatively acknowledged, RACK.packet is the one that was sent most recently including retransmissions.

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

"RACK.end_seq" is the ending TCP sequence number of RACK.packet.

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

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

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

"RACK.dupthresh" is a constant specifying the number of duplicate acknowledgments, or selectively acknowledged segments, that can (under certain conditions) trigger fast recovery, similar to [RFC6675]. As in [RFC5681] and [RFC6675], this threshold is defined to be 3.

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

"RACK.ack_ts" is the time when all the sequences in RACK.packet were selectively or cumulatively acknowledged.

"RACK.reo_wnd_incr" 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.dsack" indicates if a DSACK option has been received since last RACK.reo_wnd change

"RACK.pkts_sacked" returns the total number of packets selectively acknowledged in the SACK scoreboard.

"RACK.reord" indicates the connection has detected packet reordering event(s)

"RACK.fack" is the highest selectively or cumulatively acknowledged sequence

Note that the Packet.xmit_ts variable is per packet in flight. The RACK.xmit_ts, RACK.end_seq, RACK.rtt, RACK.reo_wnd, and RACK.min_RTT variables are kept in the per-connection TCP control block. RACK.packet and RACK.ack_ts are used as local variables in the algorithm.

## 7. Algorithm Details

### 7.1. Transmitting a data packet

Upon transmitting a new packet or retransmitting an old packet, record the time in Packet.xmit_ts. RACK does not care if the retransmission is triggered by an ACK, new application data, an RTO, or any other means.

### 7.2. 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 can track a simple global minimum of all RTT measurements from the connection, or a windowed min-filtered value of recent RTT measurements. This document does not specify an exact approach.

Step 2: Update RACK stats

Given the information provided in an ACK, each packet cumulatively ACKed or SACKed is marked as delivered in the scoreboard. Among all the packets newly ACKed or SACKed in the connection, record the most recent Packet.xmit_ts in RACK.xmit_ts if it is ahead of RACK.xmit_ts. Sometimes the timestamps of RACK.Packet and Packet could carry the same transmit timestamps due to clock granularity or segmentation offloading (i.e. the two packets were handed to the NIC as a single unit). In that case the sequence numbers of RACK.end_seq and Packet.end_seq are compared to break the tie.

Since an ACK can also acknowledge retransmitted data packets, and retransmissions can be spurious, the sender must 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 packet as invalid if any of its TCP sequences have 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 packet.

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

If the ACK is not ignored as invalid, update the RACK.rtt to be the RTT sample calculated using this ACK, and continue. If this ACK or SACK was for the most recently sent packet, then record the RACK.xmit_ts timestamp and RACK.end_seq sequence implied by this ACK. Otherwise exit here and omit the following steps.

Notice that the second condition above is a heuristic. This heuristic would fail to update RACK stats if a data packet is spuriously retransmitted because of a recent minimum RTT decrease (e.g. path change). For example, in cases with a TCP connection without TCP timestamps, and where the first M packets in a flight of data packets travel an old (longer) original path, and the remaining N packets in that flight travel a new (shorter) path and arrive out of order and elicit SACKs, then those SACKs for the N packets can initiate a spurious retransmission of the first M packets. In such scenarios, the sender would not be able to update its RACK.min_rtt using the (ambiguous) RTT samples from retransmissions, so during recovery all RTT samples may be less than RACK.min_rtt, and thus meet the second condition. In such cases RACK may not detect losses from ACK events and the recovery would then resort to the (slower) TLP or RTO timer-based recovery. However, such events should be rare and the connection would pick up the new minimum RTT when the recovery ends, so the sender can avoid repeated similar failures.

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 Packet newly acknowledged cumulatively or selectively:
        rtt = Now() - Packet.xmit_ts
        If Packet.retransmitted is TRUE:
            If ACK.ts_option.echo_reply < Packet.xmit_ts:
               Return
            If rtt < RACK.min_rtt:
               Return

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

Step 3: Detect packet reordering

To detect reordering, the sender looks for original data packets being delivered out of order in sequence space. The sender tracks the highest sequence selectively or cumulatively acknowledged in the RACK.fack variable. The name fack stands for the most forward ACK originated from the [FACK] draft. If the ACK selectively or cumulatively acknowledges an unacknowledged and also never retransmitted sequence below RACK.fack, then the corresponding packet has been reordered and RACK.reord is set to TRUE.

The heuristic above only detects reordering if the re-ordered packet has not yet been retransmitted. This is a major drawback because if RACK has a low reordering window and the network is reordering packets, RACK may falsely retransmit frequently. Consequently RACK may fail to detect reordering to increase the reordering window, because the reordered packets were already (falsely) retransmitted.

DSACK [RFC3708] can help mitigate this issue. The false retransmission would solicit DSACK option in the ACK. Therefore if the ACK has a DSACK option covering some sequence that were both acknowledged and retransmitted, this implies the original packet was reordered but RACK retransmitted the packet too quickly and should set RACK.reord to TRUE.

```
RACK_detect_reordering():
    For each Packet newly acknowledged cumulatively or selectively:
        If Packet.end_seq > RACK.fack:
            RACK.fack = Packet.end_seq
        Else if Packet.end_seq < RACK.fack AND
                Packet.retransmitted is FALSE:
            RACK.reord = TRUE

    For each Packet covered by the DSACK option:
        If Packet.retransmitted is TRUE:
            RACK.reord = TRUE
```

Step 4: Update RACK reordering window

To handle the prevalent small degree of reordering, RACK.reo_wnd serves as an allowance for settling time before marking a packet lost. This section documents a detailed algorithm following the design rationale section. RACK starts initially with a conservative window of min_RTT/4. If no reordering has been observed, RACK uses RACK.reo_wnd of 0 during loss recovery, in order to retransmit quickly, or when the number of DUPACKs exceeds the classic DUPACK threshold. The subtle difference between this approach and the conventional one [RFC5681][RFC6675] is discussed later in the section "RACK and TLP Discussion".

Further, RACK MAY use DSACK [RFC3708] to adapt the reordering window, to higher degrees of reordering, if DSACK is supported. Receiving an ACK with a DSACK indicates a spurious retransmission, which in turn suggests that the RACK reordering window, RACK.reo_wnd, is likely too small. The sender MAY increase the RACK.reo_wnd window linearly for every round trip in which the sender receives a DSACK, 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. The inflated RACK.reo_wnd would persist for 16 loss recoveries and after which it resets to its starting value, min_RTT / 4.

The following pseudocode implements the above algorithm. Note that extensions that require additional TCP features (e.g. DSACK) would work if the feature functions simply return false.

```
RACK_update_reo_wnd():
    RACK.min_RTT = TCP_min_RTT()
    If DSACK option is present:
        RACK.dsack = true

    If SND.UNA < RACK.rtt_seq:
        RACK.dsack = false  /* React to DSACK once per round trip */

    If RACK.dsack:
        RACK.reo_wnd_incr += 1
        RACK.dsack = false
        RACK.rtt_seq = SND.NXT
        RACK.reo_wnd_persist = 16 /* Keep window for 16 recoveries */
    Else if exiting loss recovery:
        RACK.reo_wnd_persist -= 1
        If RACK.reo_wnd_persist <= 0:
            RACK.reo_wnd_incr = 1


    If RACK.reord is FALSE:
        If in loss recovery:  /* If in fast or timeout recovery */
            RACK.reo_wnd = 0
            Return
        Else if RACK.pkts_sacked >= RACK.dupthresh:
            RACK.reo_wnd = 0
            return
    RACK.reo_wnd = RACK.min_RTT / 4 * RACK.reo_wnd_incr
    RACK.reo_wnd = min(RACK.reo_wnd, SRTT)
```

Step 5: Detect losses.

For each packet that has not been SACKed, if RACK.xmit_ts is after Packet.xmit_ts + RACK.reo_wnd, then mark the packet (or its corresponding sequence range) lost in the scoreboard.  The rationale is that if another packet that was sent later has been delivered, and the reordering window or "reordering settling time" has already passed, then the packet was likely lost.

If another packet that was sent later has been delivered, but the reordering window has not passed, then it is not yet safe to deem the unacked packet lost.  Using the basic algorithm above, the sender would wait for the next ACK to further advance RACK.xmit_ts; but this risks a timeout (RTO) if no more ACKs come back (e.g, due to losses or application limit).  For timely loss detection, the sender MAY install a "reordering settling" timer set to fire at the earliest moment at which it is safe to conclude that some packet is lost.  The earliest moment is the time it takes to expire the reordering window of the earliest unacked packet in flight.

This timer expiration value can be derived as follows. As a starting point, we consider that the reordering window has passed if the RACK.packet was sent sufficiently after the packet in question, or a sufficient time has elapsed since the RACK.packet was S/ACKed, or some combination of the two. More precisely, RACK marks a packet as lost if the reordering window for a packet has elapsed through the sum of:

1. delta in transmit time between a packet and the RACK.packet

2. delta in time between RACK.ack_ts and now

So we mark a packet as lost if:

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

If we solve this second condition for "now", the moment at which we can declare a packet lost, then we get:

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

Then (RACK.ack_ts - RACK.xmit_ts) is the RTT of the packet the sender used to set RACK.xmit_ts: the round trip time of the most recently (re)transmitted packet that's been delivered. To be more robust to reordering, RACK uses a more conservative RTT value to decide if an unacknowledged packet should be considered lost, RACK.rtt: the round trip time of the most recently delivered packet on the connection that was not marked invalid as a possible spurious retransmission.

When packets are delivered in order, the most recently (re)transmitted packet that's been delivered is also the most recently delivered, hence RACK.rtt == RACK.ack_ts - RACK.xmit_ts. But if packets were reordered, then the packet delivered most recently was sent before the most recently (re)transmitted packet. 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 packet lost when:

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

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

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

The following pseudocode implements the algorithm above. When an ACK is received or the RACK timer expires, call RACK_detect_loss(). The algorithm includes an additional optimization to break timestamp ties by using the TCP sequence space. The optimization is particularly useful to detect losses in a timely manner with TCP Segmentation Offload, where multiple packets in one TCP Segmentation Offload (TSO) blob have identical timestamps.

```
RACK_detect_loss():
    timeout = 0

    For each packet, Packet, not acknowledged yet:
        If Packet.lost is TRUE AND Packet.retransmitted is FALSE:
            Continue /* Packet lost but not yet retransmitted */

        If RACK_sent_after(RACK.xmit_ts, RACK.end_seq,
                           Packet.xmit_ts, Packet.end_seq):
            remaining = Packet.xmit_ts + RACK.rtt +
                        RACK.reo_wnd - Now()
            If remaining <= 0:
                Packet.lost = TRUE
            Else:
                timeout = max(remaining, timeout)

    If timeout != 0
        Arm a timer to call RACK_detect_loss() after timeout
```

Implementation optimization: looping through packets in the SACK scoreboard above could be very costly on large-BDP networks since the inflight could be very large. If the implementation can organize the scoreboard data structures to have packets sorted by the last (re)transmission time, then the loop can start on the least recently sent packet and abort on the first packet sent after RACK.time_ts. This can be implemented by using a seperate doubly-linked list sorted in time order. The implementation inserts the packet at the tail of the list when it is (re)transmitted, and removes a packet from the list when it is delivered or marked lost. We RECOMMEND such an optimization because it enables implementations to support high-BDP networks. This optimization is implemented in Linux and sees orders of magnitude improvement in CPU usage on high-speed WAN networks.

### 7.3. Tail Loss Probe: fast recovery for tail losses

This section describes a supplemental algorithm, Tail Loss Probe (TLP), which leverages RACK to further reduce RTO recoveries. TLP triggers fast recovery to quickly repair tail losses that can otherwise be recovered only via RTOs. After an original data transmission, TLP sends a probe data segment within one to two RTTs.

The probe data segment can either be new, previously unsent data, or a retransmission of previously sent data just below SND.NXT. In either case the goal is to elicit more feedback from the receiver, in the form of an ACK (potentially with SACK blocks), to allow RACK to trigger fast recovery instead of an RTO.

An RTO occurs when the first unacknowledged sequence number is not acknowledged after a conservative period of time has elapsed [RFC6298]. Common causes of RTOs include:

1. The entire flight of data is lost.

2. Tail losses of data segments at the end of an application transaction.

3. Tail losses of ACKs at the end of an application transaction.

4. Lost retransmits, which can halt fast recovery based on [RFC6675] if the ACK stream completely dries up. For example, consider a window of three data packets (P1, P2, P3) that are sent; P1 and P2 are dropped. On receipt of a SACK for P3, RACK marks P1 and P2 as lost and retransmits them as R1 and R2. Suppose R1 and R2 are lost as well, so there are no more returning ACKs to detect R1 and R2 as lost. Recovery stalls.

5. An unexpectedly long round-trip time (RTT). This can cause ACKs to arrive after the RTO timer expires. The F-RTO algorithm [RFC5682] is designed to detect such spurious retransmission timeouts and at least partially undo the consequences of such events, but F-RTO cannot be used in many situations.

### 7.4. Tail Loss Probe: An Example

Following is an example of TLP. All events listed are at a TCP sender.

1. Sender transmits segments 1-10: 1, 2, 3, ..., 8, 9, 10. There is no more new data to transmit. A TLP is scheduled to be sent 2 RTTs after the transmission of the 10th segment.

2. Sender receives acknowledgements (ACKs) for segments 1-5; segments 6-10 are lost and no ACKs are received. The sender reschedules its TLP at a time relative to the last received ACK, which is the ACK for segment 5 in this case. The sender sets the time for the TLP using the calculation described in step (2) of the algorithm.

3. When the TLP timer fires, sender retransmits segment 10.

4. After an RTT, a SACK for packet 10 arrives. The ACK also carries SACK holes for segments 6, 7, 8 and 9. This triggers RACK-based loss recovery.

5. The connection enters fast recovery and retransmits the remaining lost segments.

## 7.5. Tail Loss Probe Algorithm Details

We define the terminology used in specifying the TLP algorithm:

FlightSize: amount of outstanding data in the network, as defined in [RFC5681].

RTO: The transport's retransmission timeout (RTO) is based on measured round-trip times (RTT) between the sender and receiver, as specified in [RFC6298] for TCP. PTO: Probe timeout (PTO) is a timer event indicating that an ACK is overdue and the sender should try to transmit a TLP. Its value is constrained to be smaller than or equal to an RTO.

SRTT: smoothed round-trip time, computed as specified in [RFC6298]. TLPRxtOut: a boolean indicating whether there is an unacknowledged TLP retransmission.

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

WCDelAckT: maximum delayed ACK timer value.

The TLP algorithm has three phases, which we discuss in turn.

### 7.5.1. Phase 1: Scheduling a loss probe

Step 1: Check conditions for scheduling a PTO.

A sender should check to see if it should schedule a PTO in the following situations:

1. After transmitting new data that was not itself a TLP probe

2. Upon receiving an ACK that cumulatively acknowledges data

A sender should schedule a PTO only if all of the following conditions are met:

1. The connection supports SACK [RFC2018]

2. The connection has no SACKed sequences in the SACK scoreboard

3. The connection is not in loss recovery

If a PTO can be scheduled according to these conditions, the sender should schedule a PTO. If there was a previously scheduled PTO or RTO pending, then that pending PTO or RTO should first be cancelled, and then the new PTO should be scheduled.

If a PTO cannot be scheduled according to these conditions, then the sender MUST arm the RTO timer if there is unacknowledged data in flight.

Step 2: Select the duration of the PTO.

A sender SHOULD use the following logic to select the duration of a PTO:

```
TLP_timeout():
    If SRTT is available:
        PTO = 2 * SRTT
        If FlightSize = 1:
           PTO += WCDelAckT
    Else:
        PTO = 1 sec

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

Aiming for a PTO value of 2*SRTT allows a sender to wait long enough to know that an ACK is overdue. Under normal circumstances, i.e. no losses, an ACK typically arrives in one SRTT. But choosing PTO to be exactly an SRTT is likely to generate spurious probes given that network delay variance and even end-system timings can easily push an ACK to be above an SRTT. We chose PTO to be the next integral multiple of SRTT.

WCDelAckT stands for worst case delayed ACK timer. When FlightSize is 1, PTO is inflated by WCDelAckT time to compensate for a potential long delayed ACK timer at the receiver. The RECOMMENDED value for WCDelAckT is 200ms.

Finally, if the time at which an RTO would fire (here denoted "TCP_RTO_expire") is sooner than the computed time for the PTO, then a probe is scheduled to be sent at that earlier time.

### 7.5.2. Phase 2: Sending a loss probe

When the PTO fires, transmit a probe data segment:

```
TLP_send_probe():
    If an unsent segment exists AND
       the receive window allows new data to be sent:
           Transmit the lowest-sequence unsent segment of up to SMSS
           Increment FlightSize by the size of the newly-sent segment

    Else if TLPRxtOut is not set:
           Retransmit the highest-sequence segment sent so far
           TLPRxtOut = true
           TLPHighRxt = SND.NXT
    The cwnd remains unchanged
```

When the loss probe is a retransmission, the sender uses the highest-sequence segment sent so far. This is in order 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 any required RACK reordering settling timer to then 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 this using TLP recovery detection (see below).

Note that whether or not a probe was sent in TLP_send_probe(), the sender MUST arm the RTO timer, not the PTO timer, at the end of TLP_send_probe() if FlightSize is not zero. This ensures that the sender does not send repeated, back-to-back TLP probes. Checking TLPRxtOut prior to sending the loss probe is also critical to avoid TLP loops if an application writes periodically at an interval less than PTO.

### 7.5.3. Phase 3: ACK processing

On each incoming ACK, the sender should check the conditions in Step 1 of Phase 1 to see if it should schedule (or reschedule) the loss probe timer.

### 7.6. TLP recovery detection

If the only loss in an outstanding window of data was the last segment, then a TLP loss probe retransmission of that data segment might repair the loss. TLP recovery detection examines ACKs to detect when the probe might have repaired a loss, and thus allows congestion control to properly reduce the congestion window (cwnd) [RFC5681].

Consider a TLP retransmission episode where a sender retransmits a tail packet in a flight. The TLP retransmission episode ends when the sender receives an ACK with a SEG.ACK above the SND.NXT at the time the episode started (i.e. TLPHighRxt). During the TLP retransmission episode the sender checks for a duplicate ACK or D-SACK indicating that both the original segment and TLP retransmission arrived at the receiver, meaning there was no loss that needed repairing. If the TLP sender does not receive such an indication before the end of the TLP retransmission episode, then it MUST estimate that either the original data segment or the TLP retransmission were lost, and congestion control MUST react appropriately to that loss as it would any other loss.

Since a significant fraction of the hosts that support SACK do not support duplicate selective acknowledgments (D-SACKs) [RFC2883] the TLP algorithm for detecting such lost segments relies only on basic SACK support [RFC2018].

#### 7.6.1. Initializing and resetting state

When a connection is created, or suffers a retransmission timeout, or enters fast recovery, it executes the following:

```
TLPRxtOut = false
```

#### 7.6.2. Recording loss probe states

Senders MUST only send a TLP loss probe retransmission if TLPRxtOut is false. This ensures that at any given time a connection has at most one outstanding TLP retransmission. This allows the sender to use the algorithm described in this section to estimate whether any data segments were lost.

Note that this condition only restricts TLP loss probes that are retransmissions. There may be an arbitrary number of outstanding unacknowledged TLP loss probes that consist of new, previously-unsent data, since the retransmission timeout and fast recovery algorithms are sufficient to detect losses of such probe segments.

Upon sending a TLP probe that is a retransmission, the sender sets TLPRxtOut to true and TLPHighRxt to SND.NXT.

### 7.6.3. Detecting recoveries accomplished by loss probes

Step 1: Track ACKs indicating receipt of original and retransmitted segments

A sender considers both the original segment and TLP probe retransmission segment as acknowledged if either 1 or 2 are true:

1. This is a duplicate acknowledgment (as defined in [RFC5681], section 2), and all of the following conditions are met:

    1. TLPRxtOut is true

    2. SEG.ACK == TLPHighRxt

    3. SEG.ACK == SND.UNA

    4. the segment contains no SACK blocks for sequence ranges above TLPHighRxt

    5. the segment contains no data

    6. the segment is not a window update

2. This is an ACK acknowledging a sequence number at or above TLPHighRxt and it contains a D-SACK; i.e. all of the following conditions are met:

    1. TLPRxtOut is true

    2. SEG.ACK >= TLPHighRxt

    3. the ACK contains a D-SACK block

If either of the conditions is met, 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 TLPRxtOut to false.

Step 2: Mark the end of a TLP retransmission episode and detect losses

If the sender receives a cumulative ACK for data beyond the TLP loss probe retransmission then, in the absence of reordering on the return path of ACKs, it should have received any ACKs for the original

segment and TLP probe retransmission segment. At that time, if the TLPRxtOut flag is still true 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, so it SHOULD invoke a congestion control response equivalent to fast recovery.

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

```
if (TLPRxtOut and SEG.ACK >= TLPHighRxt) {
    TLPRxtOut = false
    EnterRecovery()
    ExitRecovery()
}
```

## 8. RACK and TLP Discussion

### 8.1. Advantages

The biggest advantage of RACK is that every data packet, whether it is an original data transmission or a retransmission, can be used to detect losses of the packets sent chronologically prior to it.

Example: TAIL DROP. Consider a sender that transmits a window of three data packets (P1, P2, P3), and P1 and P3 are lost. Suppose the transmission of each packet is at least RACK.reo_wnd (1 millisecond by default) after the transmission of the previous packet. 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 any timer. 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 packet or sequence count.

Example: LOST RETRANSMIT. Consider a window of three data packets (P1, P2, P3) that are sent; P1 and P2 are dropped. Suppose the transmission of each packet is at least RACK.reo_wnd (1 millisecond by default) after the transmission of the previous packet. 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 for 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 is very common when TCP is being rate-limited, particularly by token bucket policers with large bucket depth and low rate limit. Retransmissions are often

lost repeatedly because standard congestion control requires multiple round trips to reduce the rate below the policed rate.

Example: SMALL DEGREE OF REORDERING. Consider a common reordering event: a window of packets 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 falsely mark P1 and P2 lost. We discuss how to reduce false positives in the end of this section.

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

For some implementations (e.g., Linux), RACK works quite efficiently with TCP Segmentation Offload (TSO). RACK always marks the entire TSO blob lost because the packets in the same TSO blob have the same transmission timestamp. By contrast, the algorithms based on sequence counting (e.g., [RFC6675][RFC5681]) may mark only a subset of packets in the TSO blob lost, forcing the stack to perform expensive fragmentation of the TSO blob, or to selectively tag individual packets lost in the scoreboard.

### 8.2. Disadvantages

RACK requires the sender to record the transmission time of each packet sent at a clock granularity of one millisecond or finer. TCP implementations that record this already for RTT estimation do not require any new per-packet state. But implementations that are not yet recording packet transmission times will need to add per-packet internal state (commonly either 4 or 8 octets per packet or TSO blob) to track transmission times. In contrast, the conventional [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 is far less than the sender TCP clock granularity (e.g. inside data-centers).

RACK can easily and optionally support the conventional approach in [RFC6675][RFC5681] by resetting the reordering window to zero when the threshold is met. Note that this approach differs slightly from [RFC6675] which considers a packet lost when at least DupThresh higher-sequence packets are SACKed. RACK's approach considers a

packet lost when at least one higher sequence packet is SACKed and the total number of SACKed packets is at least DupThresh. For example, suppose a connection sends 10 packets, and packets 3, 5, 7 are SACKed. [RFC6675] considers packets 1 and 2 lost. RACK considers packets 1, 2, 4, 6 lost.

### 8.3. Adjusting the reordering window

When the sender detects packet reordering, RACK uses a reordering window of min_rtt / 4. It uses the minimum RTT to accommodate reordering introduced by packets traversing slightly different paths (e.g., router-based parallelism schemes) or out-of-order deliveries in the lower link layer (e.g., wireless links using link-layer retransmission). RACK uses a quarter of minimum RTT 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 shows that this seems to work reasonably well. We have evaluated using the smoothed RTT (SRTT from [RFC6298] RTT estimation) or the most recently measured RTT (RACK.rtt) using an experiment similar to that in the Performance Evaluation section. They do not make any significant difference in terms of total recovery latency.

### 8.4. Relationships with other loss recovery algorithms

The primary motivation of RACK is to ultimately provide a simple and general replacement for some of the standard loss recovery algorithms [RFC5681][RFC6675][RFC5827][RFC4653], as well as some nonstandard ones [FACK][THIN-STREAM]. While RACK can be a supplemental loss detection mechanism on top of these algorithms, this is not necessary, because RACK implicitly subsumes most of them.

[RFC5827][RFC4653][THIN-STREAM] dynamically adjusts the duplicate ACK threshold based on the current or previous flight sizes. RACK takes a different approach, by using only one ACK event and a reordering window. RACK can be seen as an extended Early Retransmit [RFC5827] without a FlightSize limit but with an additional reordering window. [FACK] considers an original packet to be lost when its sequence range is sufficiently far below the highest SACKed sequence. In some sense RACK 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.

Since the 3 duplicate ACK threshold for triggering fast recovery [RFC5681] has been widely deployed and usually works well in the absence of reordering, RACK uses this signal to trigger fast recovery if a connection has not observed reordering.

RACK is compatible with and does not interfere with the standard RTO [RFC6298], RTO-restart [RFC7765], F-RTO [RFC5682] and Eifel algorithms [RFC3522]. This is because RACK only detects loss by using ACK events. It neither changes the RTO timer calculation nor detects spurious timeouts.

Furthermore, RACK naturally works well with Tail Loss Probe [TLP] because a tail loss probe solicits either an ACK or SACK, which can be used by RACK to detect more losses. RACK can be used to relax TLP's requirement for using FACK and retransmitting the the highest-sequenced packet, because RACK is agnostic to packet sequence numbers, and uses transmission time instead. Thus TLP could be modified to retransmit the first unacknowledged packet, which could improve application latency.

### 8.5. Interaction with congestion control

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

RACK is applicable for both fast recovery and recovery after a retransmission timeout (RTO) in [RFC5681]. RACK applies equally to fast recovery and RTO recovery because RACK is purely based on the transmission time order of packets. When a packet retransmitted by RTO is acknowledged, RACK will mark any unacked packet sent sufficiently prior to the RTO as lost, because at least one RTT has elapsed since these packets were sent.

RACK may detect losses faster or slower than the conventional duplicate ACK threshold approach does. RACK can detect losses faster by not requiring three DUPACKs, so congestion control may reduce the congestion window earlier. When the network path has both reordering and losses, RACK detects losses slower by waiting for the reordering window to expire. TCP may continue to increase the congestion window upon receiving ACKs during this time, making the sender more aggressive. Certain congestion control algorithms can benefit from accounting for this increase in the congestion window during the reordering window.

#### 8.5.1. Example: interactions with congestion control

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

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

With RACK, a sender would send the TLP after 2*RTT and get a DUPACK. If the sender implements Proportional Rate Reduction [RFC6937] it would slow start to retransmit the remaining 9 lost packets since the number of packets 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). The recovery latency would be 2*RTT + 4*RTT, with an ending cwnd set to the slow start threshold of 10 packets.

In both cases, the sender after the recovery would be in congestion avoidance. 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 is common in local area networks, data-center networks, or content distribution networks with deep deployments. The latter case is more common in developing regions with highly congested and/or high-latency networks.

### 8.6. TLP recovery detection with delayed ACKs

Delayed ACKs complicate the detection of repairs done by TLP, since with a delayed ACK the sender receives one fewer ACK 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 WCDelAckT. Furthermore, if the receiver supports duplicate selective acknowledgments (D-SACKs) [RFC2883] then in the case of a delayed ACK the sender's TLP recovery detection algorithm (see above) can use the D-SACK 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 D-SACK, then this algorithm is conservative, because the sender will reduce cwnd 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 cwnd can be prudent.

### 8.7. RACK for other transport protocols

RACK can be implemented in other transport protocols. The algorithm can be simplified by skipping step 3 if the protocol can support a unique transmission or packet identifier (e.g. TCP timestamp options

[RFC7323]). For example, the QUIC protocol implements RACK [QUIC-LR]. The [Sprout] loss detection algorithm was also independently designed to use a 10ms reordering window to improve its loss detection.

## 9. Experiments and Performance Evaluations

RACK and TLP have been deployed at Google, for both connections to users in the Internet and internally. We conducted a performance evaluation experiment for RACK and TLP on a small set of Google Web servers in Western Europe that serve mostly European and some African countries. The experiment lasted three days in March 2017. The servers were divided evenly into four groups of roughly 5.3 million flows each:

Group 1 (control): RACK off, TLP off, RFC 6675 on

Group 2: RACK on, TLP off, RFC 6675 on

Group 3: RACK on, TLP on, RFC 6675 on

Group 4: RACK on, TLP on, RFC 6675 off

All groups used Linux with CUBIC congestion control, an initial congestion window of 10 packets, and the fq/pacing qdisc. In terms of specific recovery features, all groups enabled RFC5682 (F-RTO) but disabled FACK because it is not an IETF RFC. FACK was excluded because the goal of this setup is to compare RACK and TLP to RFC-based loss recoveries. Since TLP depends on either FACK or RACK, we could not run another group that enables TLP only (with both RACK and FACK disabled). Group 4 is to test whether RACK plus TLP can completely replace the DupThresh-based [RFC6675].

The servers sit behind a load balancer that distributes the connections evenly across the four groups.

Each group handles a similar number of connections and sends and receives similar amounts of data. We compare total time spent in loss recovery across groups. The recovery time is measured from when the recovery and retransmission starts, until the remote host has acknowledged the highest sequence (SND.NXT) at the time the recovery started. Therefore the recovery includes both fast recoveries and timeout recoveries.

Our data shows that Group 2 recovery latency is only 0.3% lower than the Group 1 recovery latency. But Group 3 recovery latency is 25% lower than Group 1 due to a 40% reduction in RTO-triggered recoveries! Therefore it is important to implement both TLP and RACK

for performance. Group 4's total recovery latency is 0.02% lower than Group 3's, indicating that RACK plus TLP can successfully replace RFC6675 as a standalone recovery mechanism.

We want to emphasize that the current experiment is limited in terms of network coverage. The connectivity in Western Europe is fairly good, therefore loss recovery is not a major performance bottleneck. We plan to expand our experiments to regions with worse connectivity, in particular on networks with strong traffic policing.

## 10. Security Considerations

RACK does not change the risk profile for TCP.

An interesting scenario is ACK-splitting attacks [SCWA99]: for an MSS-size packet sent, the receiver or the attacker might send MSS ACKs that SACK or acknowledge one additional byte per ACK. This would not fool RACK. RACK.xmit_ts would not advance because all the sequences of the packet are transmitted at the same time (carry the same transmission timestamp). In other words, SACKing only one byte of a packet or SACKing the packet in entirety have the same effect on 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, and Ilpo Jarvinen 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.

[RFC4737] Morton, A., Ciavattone, L., Ramachandran, G., Shalunov, S., and J. Perser, "Packet Reordering Metrics", RFC 4737, November 2006.

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

[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.

[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.

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

[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.

### 13.2. Informative References

[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.

[QUIC-LR] Iyengar, J. and I. Swett, "QUIC Loss Recovery And Congestion Control", draft-ietf-quic-recovery-latest (work in progress), March 2020.

[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.

[THIN-STREAM]
Petlund, A., Evensen, K., Griwodz, C., and P. Halvorsen, "TCP enhancements for interactive thin-stream applications", NOSSDAV , 2008.

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