

TCP Maintenance Working Group  
Cheng  
Internet-Draft  
Cardwell  
Intended status: Experimental  
Inc  
Expires: April 21, 2016  
2015

Y.  
N.  
Google,  
October 19,

**RACK: a time-based fast loss detection algorithm for TCP  
draft-cheng-tcpm-rack-00**

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 replace the conventional DUPACK threshold approach and its variants, as well as other nonstandard approaches.

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 <http://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 April 21, 2016.

Copyright Notice

Copyright (c) 2015 IETF Trust and the persons identified as the document authors. All rights reserved.

This document is subject to [BCP 78](#) and the IETF Trust's Legal Provisions Relating to IETF Documents (<http://trustee.ietf.org/license-info>) in effect on the date of publication of this document. Please review these documents carefully, as they describe your rights and restrictions with respect to this document. Code Components extracted from this document must include Simplified BSD License text as described in [Section 4.e](#) of

Cheng & Cardwell  
1]

Expires April 21, 2016

[Page

the Trust Legal Provisions and are provided without warranty as described in the Simplified BSD License.

## **1. 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 some packet sent sufficiently later has been cumulatively or selectively acknowledged. It does this by recording packet transmission times and inferring losses using cumulative acknowledgments or selective acknowledgment (SACK) TCP options.

The main motivation for RACK is to replace both the standard and nonstandard loss detection algorithms [[RFC5681](#)][[RFC6675](#)][[RFC5827](#)][[RFC4653](#)][[FACK](#)][[THIN-STREAM](#)] to simplify TCP development.

Another motivation is to improve loss detection for modern traffic patterns and underlying network changes. First, the prevalence of interactive request-response traffic means TCP is often application-limited. Second, wide deployment of traffic policers results in frequent lost retransmissions and losses at the tail of transactions.

Third, mobile wireless and router load-balancing cause frequent occurrences of small degrees of reordering.

These three factors together make existing packet or sequence counting approaches inefficient. This is because mechanisms based purely on counting packets in sequence order can either detect loss quickly or accurately, but it is hard to achieve both, especially when the sender is application-limited and reordering is unpredictable. And under these conditions none of them can detect lost retransmission well.

## **2. 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 store 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. 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.

### **3. Definitions of variables**

A sender needs to store these new RACK variables:

"Packet.xmit\_time" is the time of the last transmission of a data packet, including any 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.xmit\_time" is the most recent Packet.xmit\_time among all the packets that were delivered (either cumulatively acknowledged or selectively acknowledged) on the connection.

"RACK.RTT" is the associated RTT measured when RACK.xmit\_time, above, was changed. It is the RTT of the most recently transmitted packet that has been delivered (either cumulatively acknowledged or selectively acknowledged) on the connection.

"RACK.reo\_wnd" is a reordering window for the connection, 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.min\_RTT" is the estimated minimum round-trip time (RTT) of the connection.

Note that the Packet.xmit\_time variable is per packet in flight. The RACK.xmit\_time, RACK.RTT, RACK.reo\_wnd, and RACK.min\_RTT variables are per connection.

Cheng & Cardwell  
3]

Expires April 21, 2016

[Page

## 4. Algorithm Details

### 4.1. Transmitting a data packet

Upon transmitting or retransmitting a packet, record the time in `Packet.xmit_time`.

### 4.2. Upon receiving an ACK

Step 1: Update `RACK.min_RTT`.

Use the RTT measurements obtained in [[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.reo_wnd`.

To handle the prevalent small degree of reordering, `RACK.reo_wnd` serves as an allowance for settling time before marking a packet lost. By default it is 1 millisecond. We RECOMMEND implementing the reordering detection in [[REORDER-DETECT](#)][RFC4737] to dynamically adjust the reordering window. When the sender detects packet reordering `RACK.reo_wnd` MAY be changed to `RACK.min_RTT/4`. We discuss more about the reordering window in the next section.

Step 3: Advance `RACK.xmit_time` and update `RACK.RTT`.

Given the information provided in an ACK, each packet cumulatively ACKed or SACKed is marked as delivered in the scoreboard. Among all the packets ACKed or SACKed so far in the connection, record the most recent `Packet.xmit_time` in `RACK.xmit_time` if it is ahead of `RACK.xmit_time`, unless the retransmission is considered as likely spurious by the following check. Ignore the packet if it has been retransmitted and either of two condition 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. While it is still possible the packet is spuriously retransmitted because of a recent RTT decrease, we believe that our experience suggests this is a reasonable heuristic.

If this ACK causes a change to RACK.xmit\_time then record the RTT implied by this ACK:  $\text{set RACK.RTT} = \text{now} - \text{RACK.xmit\_time}$ .

Cheng & Cardwell  
4]

Expires April 21, 2016

[Page



Exit here and omit step 3 if RACK.xmit\_time has not changed.

Step 3: Detect losses.

For each packet that has not been fully SACKed, if RACK.xmit\_time is after Packet.xmit\_time + 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, the packet was likely lost.

If a packet that was sent later has been delivered, but the reordering window has not passed, then it is not yet safe to deem the given packet lost. Using the basic algorithm above, the sender would

wait for the next ACK to further advance RACK.xmit\_time; 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, which is the minimum value of (Packet.xmit\_time + RACK.RTT + RACK.reo\_wnd + 1ms) across all unacknowledged packets.

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 the S/ACK of the RACK packet (RACK.sacked\_time) and now

So we mark a packet as lost if:

```
RACK.xmit_time > Packet.xmit_time
AND
(RACK.xmit_time - Packet.xmit_time) + (now - RACK.sacked_time) >
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_time + RACK.reo_wnd + (RACK.sacked_time -  
RACK.xmit_time)
```

Cheng & Cardwell  
5]

Expires April 21, 2016

[Page

Then  $(\text{RACK.sacked\_time} - \text{RACK.xmit\_time})$  is just the RTT of the packet we used to set  $\text{RACK.xmit\_time}$ , so this reduces to:

```
now > Packet.xmit_time + RACK.RTT + RACK.reo_wnd
```

The following pseudocode implements the algorithm above. When an ACK is received or the RACK timer expires, call `RACK_detect_loss()`:

```
RACK_detect_loss():  
  min_timeout = 0
```

```
  For each packet, Packet, in the scoreboard:
```

```
    If Packet is already SACKed, ACKed,  
    or marked lost and not yet retransmitted:  
      Skip to the next packet
```

```
    If Packet.xmit_time > RACK.xmit_time:  
      Skip to the next packet
```

```
    timeout = Packet.xmit_time + RACK.RTT + RACK.reo_wnd + 1
```

```
    If now >= timeout
```

```
      Mark Packet lost
```

```
    Else If (min_timeout == 0) or (timeout is before min_timeout):  
      min_timeout = timeout
```

```
  If min_timeout != 0
```

```
    Arm the RACK timer to call RACK_detect_loss() at the time  
min_timeout
```

## **5. Algorithm Analysis**

### **5.1. 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 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  $\text{RACK.reo\_wnd}$  (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](#)]

Cheng & Cardwell  
6]

Expires April 21, 2016

[Page

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 (as a tail drop) 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 due to application-limited behavior. Suppose the sender has detected reordering previously (e.g., by implementing the algorithm in [\[REORDER-DETECT\]](#)) 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 the 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.

## **5.2. 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) to track transmission times. In contrast, the conventional approach requires one variable to track number of duplicate ACK threshold.



### **5.3. Adjusting the reordering window**

RACK uses a reordering window of  $\text{min\_rtt} / 4$ . 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). Alternatively, RACK can use the smoothed RTT used in RTT estimation [[RFC6298](#)]. However, smoothed RTT

can be significantly inflated by orders of magnitude due to congestion and buffer-bloat, which would result in an overly conservative reordering window and slow loss detection.

Furthermore,

RACK uses a quarter of minimum RTT because Linux TCP uses 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.

One potential improvement is to further adapt the reordering window by measuring the degree of reordering in time, instead of packet distances. But that requires storing the delivery timestamp of each packet. Some scoreboard implementations currently merge SACKed packets together to support TSO (TCP Segmentation Offload) for faster

scoreboard indexing. Supporting per-packet delivery timestamps is difficult in such implementations. However, we acknowledge that the current metric can be improved by further research.

### **5.4. 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](#)] and nonstandard ones [[FACK](#)][[THIN-STREAM](#)]. While RACK can be a supplemental loss detection

on top of these algorithms, this is not necessary, because the 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.

Nevertheless RACK is still an experimental algorithm. Since the oldest loss detection algorithm, the 3 duplicate ACK threshold [[RFC5681](#)], has been standardized and widely deployed, we RECOMMEND



TCP implementations use both RACK and the algorithm specified in [Section 3.2 in \[RFC5681\]](#) for compatibility.

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

Furthermore, RACK naturally works well with Tail Loss Probe [[TLP](#)] because a tail loss probe solicit seither 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 can be modified to retransmit the first unacknowledged packet, which can improve application latency.

### **[5.5.](#) 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](#)]. However, RACK may detect losses earlier or later than the conventional duplicate ACK threshold approach does. A packet marked lost by RACK SHOULD NOT be retransmitted until congestion control deems this appropriate (e.g. using [[RFC6937](#)]).

RACK is applicable for both fast recovery and recovery after a retransmission timeout (RTO) in [[RFC5681](#)]. The distinction between fast recovery or RTO recovery is not necessary 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.

### **[6.](#) 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\_time 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.



## 7. IANA Considerations

This document makes no request of IANA.

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

## 8. Acknowledgments

The authors wish to thank Matt Mathis for his insights in FACK that inspired this work. Nandita Dukkupati, Eric Dumazet, Van Jacobson, Ian Swett, and Jana Iyengar contributed to the algorithm and the implementations in TCP and QUIC. We thank the authors of [RFC3517](#) for a great document on an excellent loss recovery algorithm that this draft is trying to improve upon.

## 9. References

### 9.1. Normative References

- [RFC793] Postel, J., "Transmission Control Protocol", September 1981.
- [RFC2018] Mathis, M. and J. Mahdavi, "TCP Selective Acknowledgment Options", [RFC 2018](#), October 1996.
- [RFC6937] Mathis, M., Dukkupati, N., and Y. Cheng, "Proportional Rate Reduction for TCP", May 2013.
- [RFC4737] Morton, A., Ciavattone, L., Ramachandran, G., Shalunov, S., and J. Perser, "Packet Reordering Metrics", [RFC 4737](#), November 2006.
- [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.
- [RFC6298] Paxson, V., Allman, M., Chu, J., and M. Sargent, "Computing TCP's Retransmission Timer", [RFC 6298](#), June 2011.
- [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.



- [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.
- [RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", [RFC 2119](#), March 1997.
- [RFC5681] Allman, M., Paxson, V., and E. Blanton, "TCP Congestion Control", [RFC 5681](#), September 2009.
- [RFC2883] Floyd, S., Mahdavi, J., Mathis, M., and M. Podolsky, "An Extension to the Selective Acknowledgement (SACK) Option for TCP", [RFC 2883](#), July 2000.
- [RFC7323] Borman, D., Braden, B., Jacobson, V., and R. Scheffenegger, "TCP Extensions for High Performance", September 2014.

## **9.2. 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.
- [TLP] Dukkkipati, N., Cardwell, N., Cheng, Y., and M. Mathis, "Tail Loss Probe (TLP): An Algorithm for Fast Recovery of Tail Drops", [draft-dukkkipati-tcpm-tcp-loss-probe-01](#) (work in progress), August 2013.
- [RTO-RESTART]  
"TCP  
(work  
in progress), June 2015.  
Hurtig, P., Brunstrom, A., Petlund, A., and M. Welzl,  
and SCTP RTO Restart", [draft-ietf-tcpm-rtorestart-08](#)  
in progress), June 2015.
- [REORDER-DETECT]  
Zimmermann, A., Schulte, L., Wolff, C., and A. Hannemann,  
"Detection and Quantification of Packet Reordering with  
TCP", [draft-zimmermann-tcpm-reordering-detection-02](#) (work  
in progress), November 2014.
- [THIN-STREAM]  
Petlund, A., Evensen, K., Griwodz, C., and P. Halvorsen,  
"TCP enhancements for interactive thin-stream  
applications", NOSSDAV , 2008.



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

Authors' Addresses

Yuchung Cheng  
Google, Inc  
1600 Amphitheater Parkway  
Mountain View, California 93117  
USA

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

Neal Cardwell  
Google, Inc  
76 Ninth Avenue  
New York, NY 10011  
USA

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

