RACK: a time-based fast loss recovery

draft-ietf-tcpm-rack-02

Yuchung Cheng
Neal Cardwell
Nandita Dukkipati

Google

IETF98: Chicago, March 2017
What's RACK (Recent ACK)?

Key Idea: time-based loss inferences (not packet or sequence counting)

- If a packet is delivered out of order, then packets sent chronologically before it are either lost or reordered.
- Wait RTT/4 before retransmitting in case the unacked packet is just delayed. RTT/4 is empirically determined (more later on making it adaptive).
- Conceptually RACK arms a (virtual) timer on every packet sent. The timers are updated by the latest RTT measurement.
Tail Loss Probe (TLP)

- **Problem**
  - Tail drops are common on request/response traffic
  - Tail drops lead to timeouts, which are often 10x longer than fast recovery
  - 70% of losses on Google.com recovered via timeouts

- **Goal:**
  - Reduce tail latency of request/response transactions

- **Approach**
  - Convert RTOs to fast recovery
  - Solicit a DUPACK by retransmitting the last packet in 2 SRTTs
  - Requires RACK to trigger fast recovery

After 2 SRTTs... send TLP to get SACK to start RACK recovery of a tail loss
What’s new since last IETF in Nov.

1. Fully implemented in Linux 4.10
   a. On by default
   b. Reduced number of loss recovery heuristics from 9 to 4:
      RACK, TLP, F-RTO, DupThresh (RFC6675), FACK, Early Retransmit (RFC5827), Thin-Dupack
      (RFC4653), Forward Retransmit
   c. Deployed in Google TCP

2. -02 draft
   a. New experiments on reordering window length and removing DupThresh
   b. New text on interacting with congestion control
Exp: RACK+TLP vs DupACK threshold

4-way experiment at one Google DC in Europe for a week in 2017. ~1.5B flows sampled

- RFC6675-only retransmit rate is 1.3%
- RACK + TLP reduces recovery latency by 24% and may replace DupThresh approach

<table>
<thead>
<tr>
<th>Diffs compared to RFC6675 (DupThresh)</th>
<th>RACK</th>
<th>RACK + TLP</th>
<th>RACK + TLP + RFC6675 (Linux)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Time in loss recovery</td>
<td>-0.5%</td>
<td>-24.0%</td>
<td>-24.1%</td>
</tr>
<tr>
<td>%RTO reduced</td>
<td>-5.4%</td>
<td>-25.8%</td>
<td>-23.7%</td>
</tr>
<tr>
<td>Retrans. Rate (including TLP)</td>
<td>1.3%</td>
<td>1.5%</td>
<td>1.5%</td>
</tr>
</tbody>
</table>
How RACK interacts with congestion control

RACK influences congestion control indirectly

- Congestion control (Reno/RFC5681)
  - On fast recovery cwnd is reduced to ssthresh
  - On RTO cwnd resets to 1
- By reducing RTOs, RACK + TLP speeds up fast recovery and avoids cwnd resets
  - Rationale: cwnd should only reset if the entire flight is lost and the ack clock is also lost

Example: Reno C.C. w/ cwnd=20. Send 10 packets, which are all dropped.

<table>
<thead>
<tr>
<th>Events</th>
<th>Recovery Time</th>
<th>cwnd upon recovery ends</th>
</tr>
</thead>
<tbody>
<tr>
<td>Standard</td>
<td>RTO then slow start</td>
<td>RTO + 4*RTT</td>
</tr>
<tr>
<td>RACK + TLP</td>
<td>Fast recovery</td>
<td>2<em>RTT + 4</em>RTT</td>
</tr>
</tbody>
</table>
Next: smarter reordering window

Current window is \( \max(1\text{ms}, \min_{\text{RTT}}/4) \)

- Too low: high spurious retransmit if reordering exceeds the window
- Too high: 1ms is too high inside a data-center (RTT < 100us)
  - But <1ms timer has high cost
- WIP: adaptive reordering window
  - Measure reordering degree in time
    - \( \text{reor\_deg} = (\text{last\textunderscore out\textunderscore of\textunderscore order\textunderscore delivery\textunderscore time} - \text{last\textunderscore inorder\textunderscore delivery\textunderscore time}) \)
    - \( \text{reo\_wnd} = K \times \text{reordering\_degree} \)
  - Reduce K if recovery finished w/o signs of reordering
  - Increase K if DSACKs or timestamps indicate reo\_wnd was too small
  - Stress test on low-latency (<100us) and high-reordering (multi-path) environments
BBR/RACK on emulated 1Gbps, 53ms RTT and random packet delay jitter [0, 10ms]

RACK may cause excessive spurious retransmits if reordering > RACK.reo_wnd
Next: one loss recovery (RACK)

Linux still uses both RACK and RFC6675 (DupThresh)

- Runs both algorithms on each ACK
- Recovery starts when either algorithm marks a packet lost

Goal: RACK + TLP as the omnipotent recovery

- a/b experiment disabling RFC6675 on Google
- Experiment w/ DupThresh-triggered start to fast recovery
  - reo_wnd = 0 if not in recovery and #DupAcks >= DupThresh
  - Progressively phase out DupThresh approach
Conclusion

Vision: making TCP resilient and efficient to reordering and loss with one algorithm

- Better load-balancing (e.g. multi-paths, flowlets)
- Faster forwarding (e.g. parallel forwarding, wireless link layer optimization)
- Simpler transport with time-based approach

RACK is now the key loss recovery in Linux

Work-in-progress

1. Optimize reordering window for high reordering and/or low RTT
2. Pure time-based recovery by completely retiring DupThresh approach