Adventures in TCP/IP

# Improving TCP's Responses to Reordering

BY RANDALL STEWART

This column will be slightly different from the others in this series. It describes a set of problems that arose with the use of TCP and provides a walk-through of how the problems were troubleshooted, illustrating information covered in previous columns. It will also refer to some TCP mechanisms that were covered in previous columns in this series. It will conclude by tying it all together and showing not only how the problem was solved, but also how the RACK stack became more resilient to packet reordering due to a driver bug that was found and fixed.

## The Problem

So, sometime last year, Drew Gallatin approached me with a TCP problem he was having. It started with him attempting to download a set of packages to be updated in his new location with the FreeBSD package tool (`pkg`). And things were miserably slow. He expected Mbps speeds but instead got extremely slow downloads. So, he started testing with another FreeBSD site he has access to. This led him to try changing to the RACK stack instead, since it had performed better for him in the past. And sure enough, as he told me, he was getting three times the performance with the RACK stack as with the FreeBSD default stack. But three times 10 kbps is only 30 kbps, and he was expecting at least two orders of magnitude better performance than that. He passed this problem to me, asking for help as to why TCP in both the RACK stack and the FreeBSD stack was performing so badly on his 'high-speed' Internet.

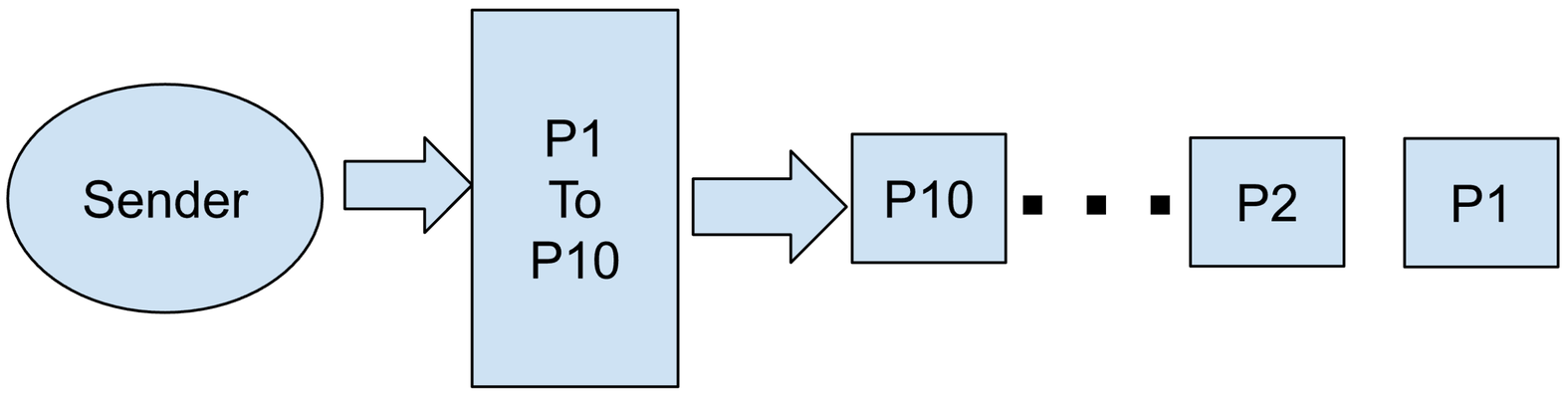
He expected Mbps speeds but instead got extremely slow downloads.

## Identifying the Problem

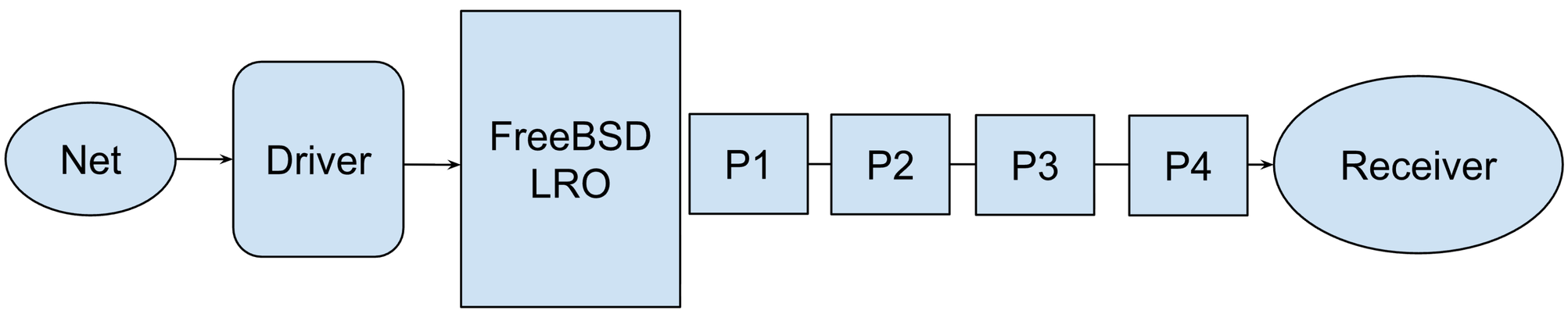
As discussed in the column on Black Box Logging [1], Black Box Logging (BBlog) provides detailed debugging, especially for the RACK stack. The very first thing to do was to enable BBlog on both the sender and receiver. Please also refer to that column for detailed descriptions of how to enable BBlog.

For Drew's situation, I ended up providing him with a couple of custom client-server programs that would send a file and discard it but also enable BBlogs. Drew then ran the BBlog

collector (`tcplog_dumper`) and the specific programs to send a comparably sized file, similar to what he was trying to download to his new location. Once he had the BBlogs, he allowed me to read and interpret them. I won't bore you with the copious text that the BBlog entails, but they quickly painted a picture of the sender and receiver that immediately told what was happening at the TCP layer. Basically, the sender would begin with its initial window of roughly 10 MSS (14,600 bytes) and send a burst to the internal driver TSO mechanism:




The data packets would cross the Internet and arrive at the TCP receiver as follows:




The receiver was sending SACKs and ACKs, indicating that the packets were processed by the receiver in the order P4, P4, P2, and finally P1.

Now, from the BBlogs, I could not determine where the reordering was coming from — you are only seeing TCP's perspective after all. What is happening below the FreeBSD LRO layer (i.e., the driver and network) is not visible in the BBlogs (note that some LRO logs are available in BBlogs). It is also not uncommon to see reordering in a network, as shown by Bennett, Partridge, and Shectman [2]. So, I asked Drew to investigate why his network was experiencing massive reordering — and it was quite massive.

Now, from the BBlogs, I could not determine where the reordering was coming from — you are only seeing TCP's perspective after all.

It appeared that every group of 4-5 packets would arrive in the reverse order in which they were sent. So, you would see your burst of 10 initial packets translated into an arrival pattern of P4, P3, P2, P1, followed by P8, P7, P6, P5, followed by P10 and P9. This behavior will cause the FreeBSD TCP stack to go into recovery constantly since it sees three duplicate ACKs/SACKs, which start retransmitting just as the ACK for all the packets arrives. Once you enter fast recovery, as discussed in [3], you exit the recovery after all in-flight data is acknowledged and end up with a congestion window cut in half and the `ssthresh` point (the point at which we flip from slow start to congestion avoidance) set to that same value.

This meant the FreeBSD TCP stack could never really get out of recovery for very long, which easily explained the horrible performance Drew was seeing. Once recovery ended, it would drop back into congestion avoidance. Anytime the congestion window allowed a

four-packet burst, the stack would once again see three duplicate ACKs/SACKs and reenter recovery.

### The Root Cause of the Issue

Drew, a FreeBSD developer (who works with drivers for fun), found after a little investigation that it was actually an error in the driver interface to LRO that caused the reordering. Basically, the hardware had multi-queue disabled, so that the hardware stopped calculating the RSS hash because it was not needed. The driver, however, was still marking the RSS hash as valid. The FreeBSD LRO code uses the RSS hash to sort the received packets. This mis-marking thus caused LRO to sort the packets incorrectly. Once his driver was patched to send packets in the correct order to LRO, both RACK and the FreeBSD stack's performance improved massively, on the order he was expecting. It was great helping Drew find this issue using BBlogs, but the whole incident raised more questions for me from a TCP perspective. And it may even spark a question in your mind, too, if you are unfamiliar with the RACK stack.

## Further Questions Generated by the Bug

### Why did the RACK Stack Perform Better than the FreeBSD Stack?

For those of you unfamiliar with RACK, you may wonder why RACK performed three times better than the FreeBSD stack (even though that was only 30 kbps, it was still quite a difference). This is because, as discussed in [4], the RACK stack has protections against reordering. It does this by using a timer in combination with loss reports. When a SACK arrives telling that data is lost, it will wait to retransmit if not enough time has elapsed. The timer is usually the latest RTT plus a small extra delay, but as RACK sees reordering in the network through DSACKs (a DSACK is a SACK indicating that a TCP segment was received more than once by the receiver), it will expand this timer up to two times the last RTT seen. This meant that after the initial recovery, where the extra delay was small, the arrival of DSACKs increased the RACK delay timer enough that it would not constantly go into recovery but would instead wait long enough for all acknowledgments to flow back.

You may wonder why RACK performed three times better than the FreeBSD stack.

### Why was the RACK Stack Still So Slow?

For me, this was the fundamental question posed by this bug. RACK has this wonderful mechanism to protect from reordering, so why, after its initial recovery, did RACK not quickly speed up to reach Drew's anticipated megabits per second? With a bit of digging into the BBlog, I ultimately discovered the answer.

This question goes back to the fundamentals of the way congestion control works. Once you exit the initial slow start with a loss/recovery incident, your `ssthresh` is set to 5 packets (the initial 10 cut in half). This then means that RACK is forever in congestion avoidance. And this places a significant drag on TCP. When in congestion avoidance, you increase your congestion window by one MSS every time a full congestion window of data is acknowledged. So, you send and have acknowledged five packets and now raise your congestion window to six packets. Now you send and get acknowledged six packets, and then raise it to seven. This

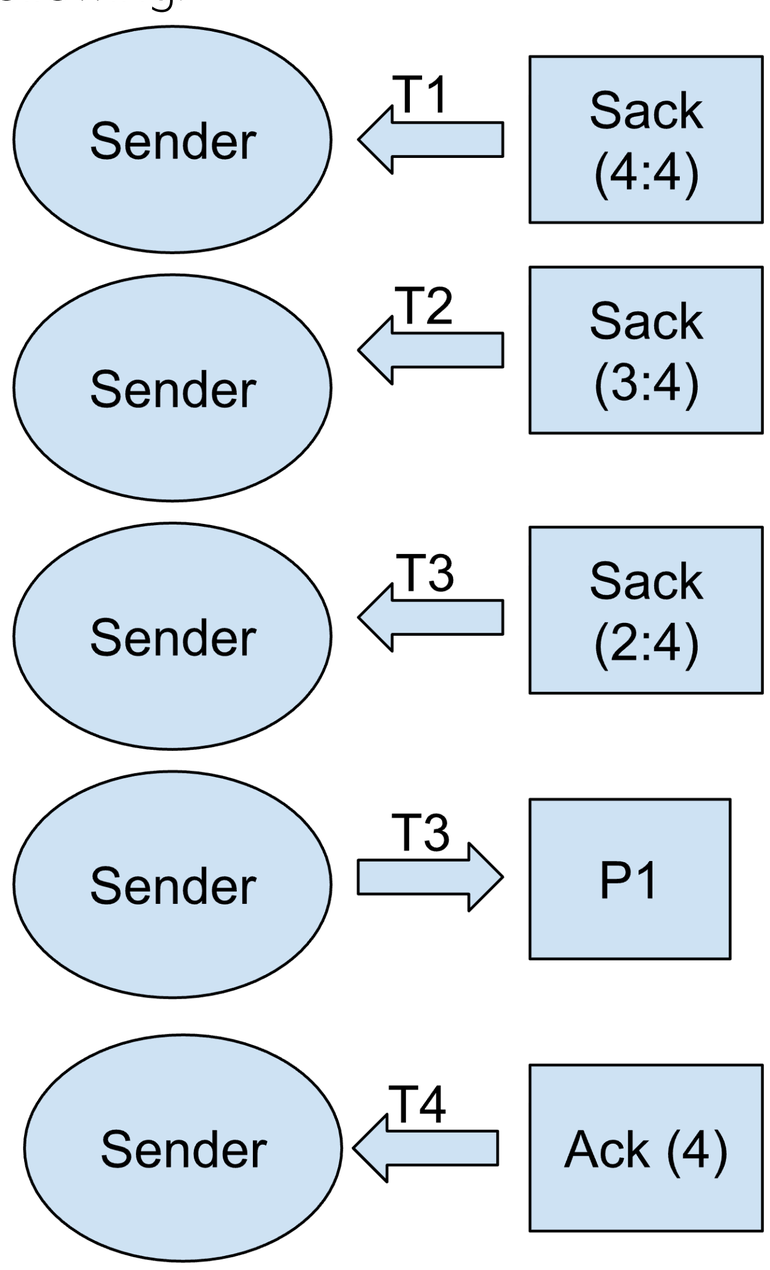
continues until some other error occurs or the connection data is all sent, and the connection completes.

So, in looking at his round-trip time (about 45 ms or so), that means that every 45 ms you would grow the congestion window by one packet. That sounds fine until you realize that to fully utilize a 1,000 Mbps network, your congestion window at 45 ms RTT needs to be open to about 3,850–3,900 packets. So, you would need around 380 round trips for RACK to start using his network fully, which is about 170–180 seconds. If he had a single super-large transfer, this would not be too noticeable (in the grand scheme of things), but each time a transfer was completed, it would start a new TCP connection, once again hitting the same issues. This made his 'fast' network horribly slow in his perception.

So, this raised a fundamental question in my mind. How can we change the RACK stack to better recover from this situation?

**What Improvements Are Now Integrated Into RACK Stack?**

So, what I realized was a way for RACK to 're-enter' slow start (note that slow start is a misnomer in my opinion, since it's exponential growth, doubling the congestion window every RTT) so that it could open its window much faster. So, when looking at the initial situation, RACK would see the following:




The key here is that time mark T3 (in comparison to the transmit time) is just enough time longer than the RTT to tell RACK that yes, it's time to retransmit the packet. This is because RACK had not, to this point, seen any duplicate acknowledgments, and the initial send to T3 was longer than an RTT plus the small initial delta RACK uses. But even more importantly, T4 — the actual arrival of the acknowledgment that moves the cumulative point to P4 — happens in much less time than an RTT. Basically, the time between T4 and T3 was always less than half of the RTT. This provided an insight into how RACK could be adjusted to handle this scenario.

RACK now tracks how much data, in bytes, has been retransmitted. Any time a SACK or ACK arrives that has had only one retransmission, and the acknowledgment (SACK or ACK) is less than half of the SRTT from when the retransmission was sent, it will reduce that count. If the count reduction is from an ACK (where the cumulative acknowledgment point is moving forward) and the count has fallen to zero, this then means that we have hit a situation where the actual entry of recovery is false, and we should restore the previous `ssthresh` and congestion window and resume slow start.

Adding the above small change to RACK changes its poor performance to match closely what one would see with no reordering in the picture and provides the RACK stack with even better protection from reordering.


1. R. Stewart, M. Tüxen, "Adventures in TCP/IP: TCP Black Box Logging", in *The FreeBSD Journal May/June, 2024*.
2. C. R. Bennett, C. Partridge and N. Shectman, "Packet reordering is not pathological network behavior," in *IEEE/ACM Transactions on Networking*, vol. 7, no. 6, pp. 789-798, Dec. 1999.
3. R. Stewart, "Dynamic Goodput Pacing: A New Approach to Packet Pacing", in *The FreeBSD Journal November/December, 2024*.
4. R. Stewart, M. Tüxen, "RACK and Alternate TCP Stacks for FreeBSD", in *The FreeBSD Journal January/February, 2024*.



**RANDALL STEWART** (rrs@freebsd.org) has been an operating system developer for over 40 years and a FreeBSD developer since 2006. He specializes in Transports including TCP and SCTP but has also been known to poke into other areas of the operating system. He is currently an independent consultant.