Updates On TCP in FreeBSD 14

BY RICHARD SCHEFFENEGGER

It's been about 3 ½ years since I last reported on the area of the FreeBSD project I focus on, namely, the TCP protocol implementation. For those who don't know, FreeBSD doesn't feature only one TCP stack, but multiple ones with development occurring dominantly in the RACK and base stack. Currently, the one used by default (base stack) is the stack long evolved and derived from BSD4.4. Also, since 2018, we have had a completely refactored stack ("RACK stack" – with the ***R***ecent ***ACK***nowledgement mechanism as its namesake), that provides many advanced capabilities that are lacking in the base stack. For example, the RACK stack provides high granularity pacing capabilities. That is, the stack can time the sending of packets and even out the consumption of network resources. In contrast, if an application presents the base stack with a sudden burst of data to transmit, there are instances where this data will be sent out in a large burst at near line rate (the speed of the interface, provided the CPU and internal busses are not the bottleneck). This happen most notably whenever there is a short application pause from the last application IO by a couple tens of milliseconds. (Further details on the RACK stack are beyond the scope of this article and can be read in the accompanying article by Michael Tuexen and Randall Stewart.).

Here, I want to highlight some of the new features that have been brought into the base stack – many of which are enabled by default, and some of which may need to be specifically activated. Each feature will be described with details that may help improve the networking experience.

Overall, there have been around 1033 commits since the release of FreeBSD 13.0 to the sys/netinet directory where all the transport protocols traditionally live. This gives an overview of selected changes to the base stack, where functionality was improved:

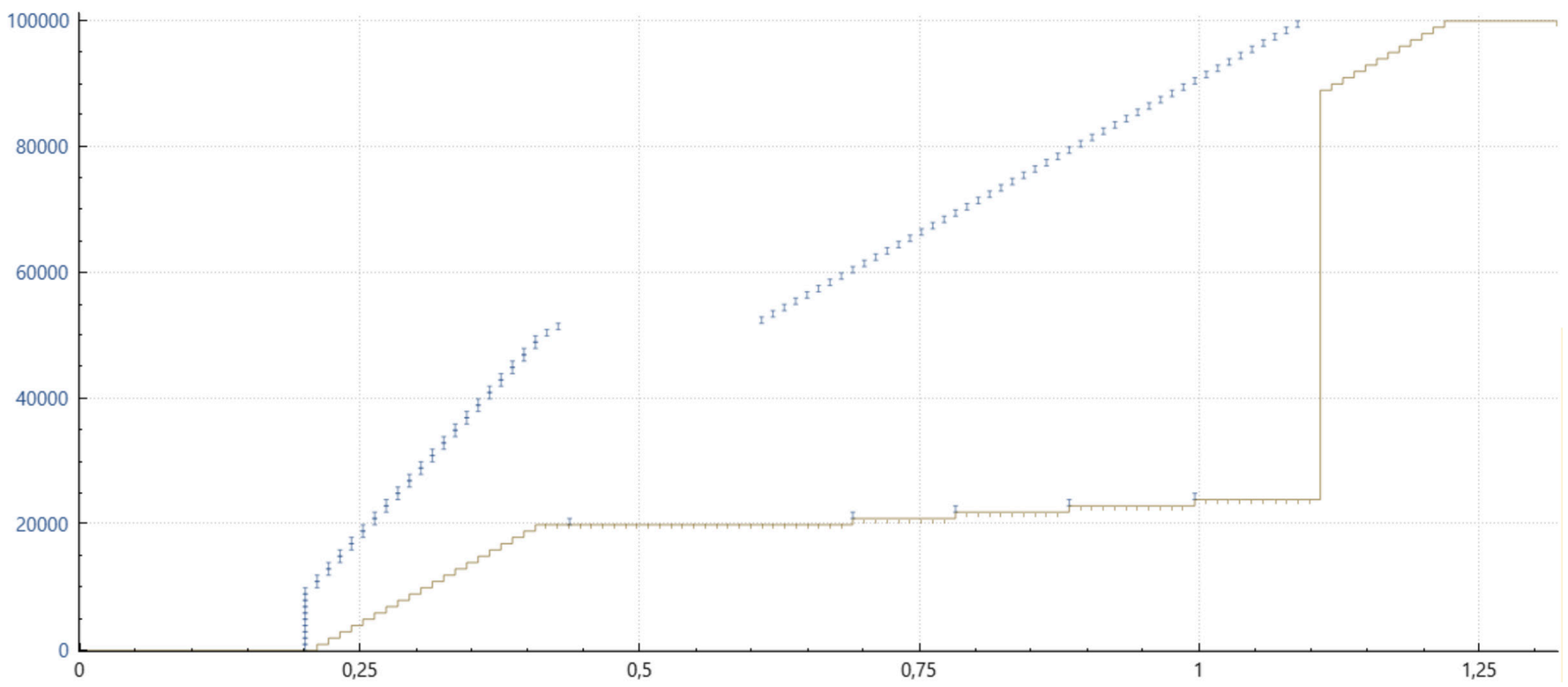
Proportional Rate Reduction

The first feature brought into the base stack is PRR – Proportional Rate Reduction (RFC6937). To understand PRR on a high level, let's first understand how SACK behavior functions during loss recovery. One issue with standard SACK loss recovery was that while the congestion window is adjusted on entering loss recovery (e.g., to 50% of the value at onset of congestion with NewReno or 70% with Cubic) after a single packet loss, the estimation of how many packet are still in flight will initially not allow any packets to be sent while ACKs return for the first half (NewReno) or initial 30% of the window. Once that limit has been reached, every incoming ACK will elicit a new packet, but this may well happen at an effective rate that overwhelms the congestion point in the network. The initial quiet period may serve to drain queues and allow for the subsequent, faster-than-ideal transmissions. Often that behavior leads to subsequent losses (perhaps even losses of retransmitted packets – more on that later).

To quickly adjust the effective sending rate – and also deal more appropriately when there are multiple losses of data packets or maybe even losses of ACK packets – PRR will calculate how much data should be sent out for every new, incoming ACK and sends out as many full-sized packets as appropriate at that that time. In the simple example with NewRe-

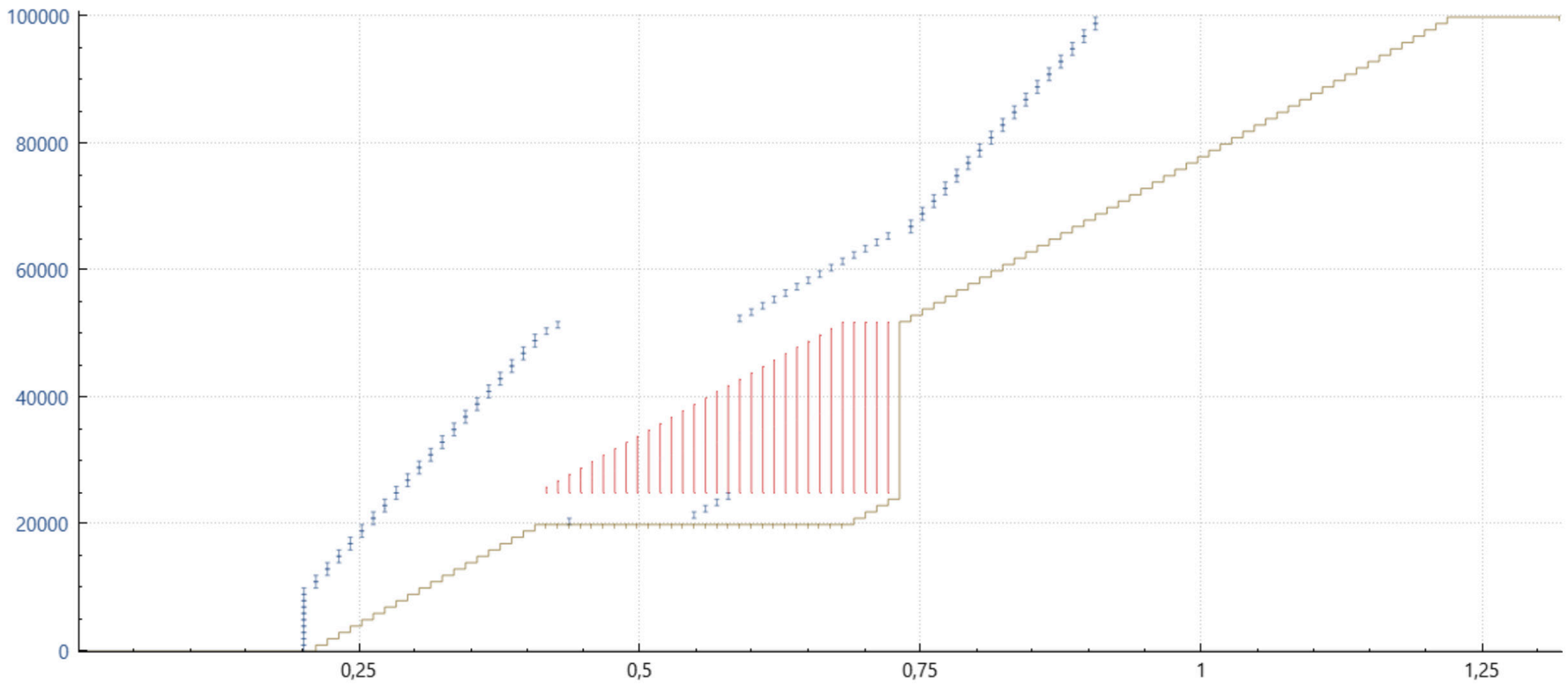
no with a reduction of the congestion window to half and just a single loss, this will cause one new packet to be sent for every two ACKs returned. Thus, the sending rate will adjust instantly to exactly half of what it used to be – stopping the congested device from being overloaded. In the presence of multiple data losses or ACKs discarded, PRR may inject even more than one packet when an ACK is finally received, making good on these missed sending opportunities. Overall, this behavior ensures that at the end of the window (RTT) when loss recovery happens, the effective congestion window is as close as possible to the expected congestion window, and that no transmit opportunities are missed, even under problematic scenarios like multiple packet losses or ACK losses.

Hopefully, a few graphs can explain this niche detail better. Below, we have time-sequence graphs which can be obtained from wireshark or the combination of tcptrace and associated xplot. The small blueish vertical bars indicate when a particular packet covering the data sequence was sent – as on the left axis. The greenish more horizontal line below indicates which data was received contiguous by the receiver. Red vertical lines signify any discontinuous range of data that made it to the receiver.



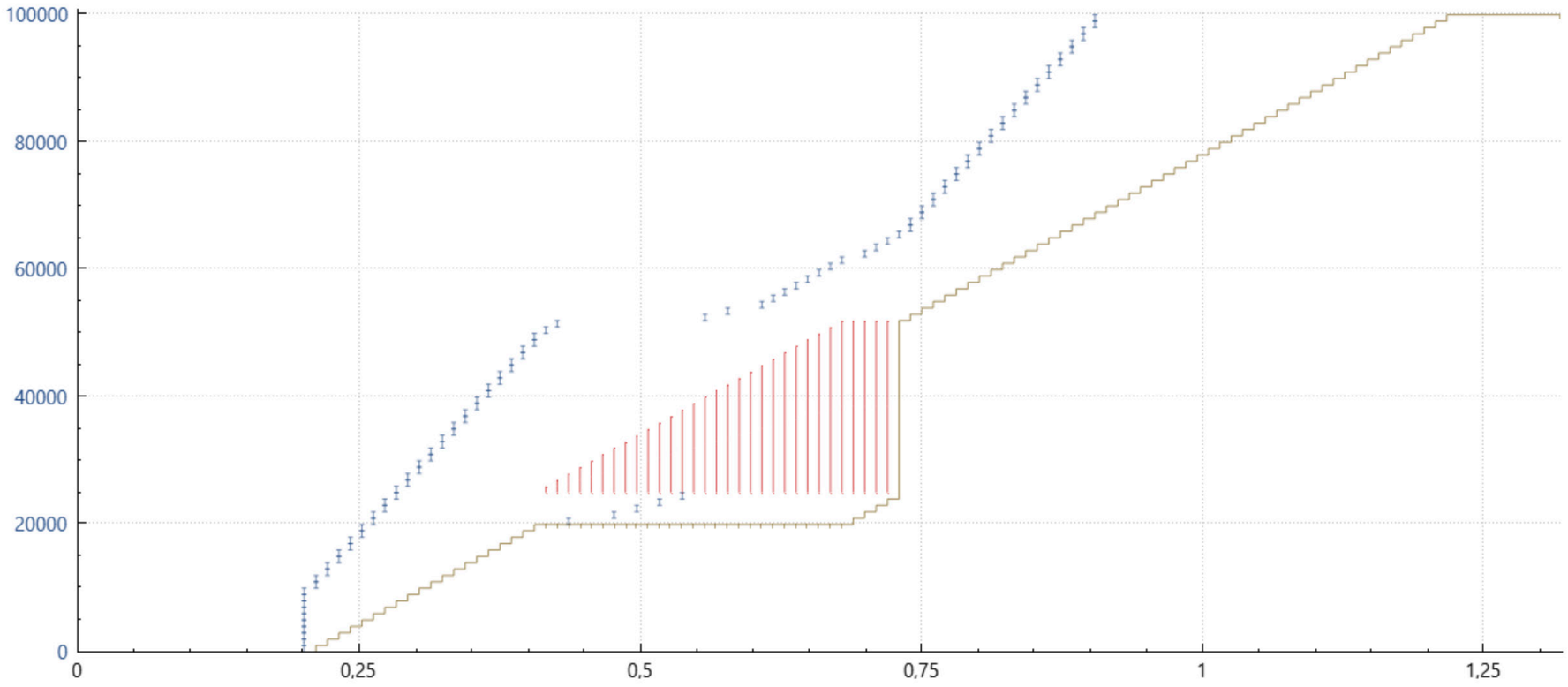
Cubic Without SACK or PRR, Classic NewReno Loss Recovery

Note that only a single data packet can be recovered within one window (or round-trip time) and the long stretch of horizontal green line indicates the latency induced before the receiving application gets to process additional data.



Cubic with SACK, but no PRR

As this example shows, SACK dramatically improves the situation, since all the lost packets can (typically) be retransmitted within one RTT. However, take note of the pause and resumption of sending on each ACK later. This behavior drives data at an effective rate that caused some packets to be dropped into the network. Often, this causes one or more of the retransmissions to arrive too quickly and the network drops the retransmission. The only recourse then is to wait for a retransmission timeout (RTO).



Cubic with SACK (6675) and PRR

The improvement with PRR depicted here is subtle. Where previously no data was sent for half a window, and then at the old, likely too high rate for the second half, PRR injects packets approximately every other received ACK until the new sending rate has been reached, and then on nearly each subsequent incoming ACK. This serves to reduce the effective sending rate of the retransmissions and making it less likely that these will get discarded by the network. Fewer RTOs and improved latency are the consequences.

The graph shown here is not entirely correct but attempts to convey the aspect of PRR "dithering" packets sent appropriately over the received ACKs to send them out – in this case, on average, 0.7 packets for every ACK including those which may have been discarded by the network.

The final update in this space was that PRR now automatically switches to a less conservative mode unless there are additional losses in loss recovery. This effectively improves the transmission speed during loss recovery, similar to what would happen during normal operation in the congestion avoidance phase. PRR works best (naturally) in conjunction with SACK, but also when only non-SACK duplicate ACKs are available. Even with nothing but ECN feedback, PRR improves the transmission timings.

SACK Handling

In recent years, the adherence of the base stack to SACK loss recovery as specified in RFC6675 has been improved. But while parts of the estimation on how much data is still outstanding in the network were improved, other aspects of RFC6675 were missing.

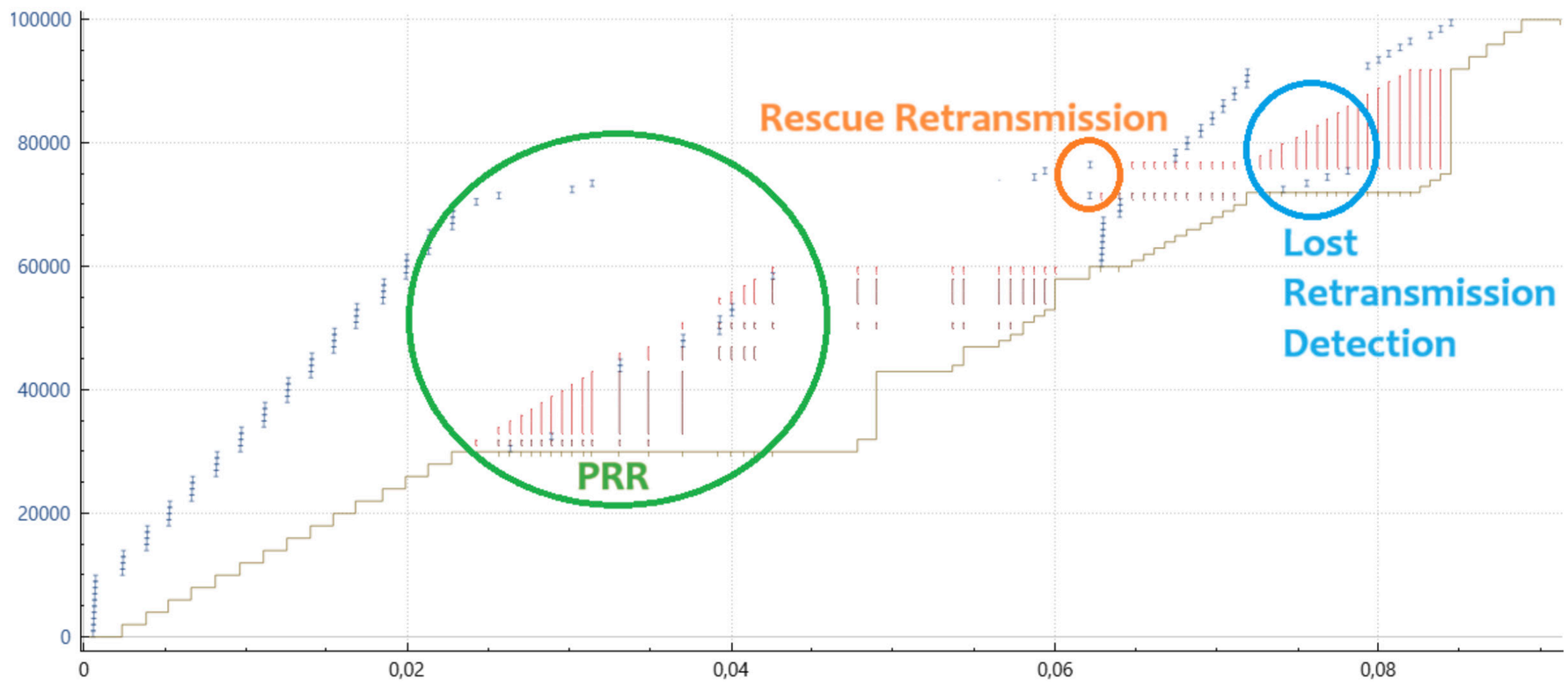
Improvements in this space now include the use of so-called rescue retransmissions – a precursor of the Tail-Loss Probe, which is implemented in the RACK stack. In short, when the final few packets of a transfer are lost in addition to earlier packet losses, the stack can detect the problem and will retransmit the ultimate packet to perform a timely loss recovery.

And, by implementing additional accounting when processing any incoming SACK block,

the stack keeps better track of whether a particular packet ought to have left the network either by being received or having very likely been dropped.

A final enhancement was to track whether retransmissions may also have been dropped by the network, but unlike RACK, which uses the time domain, the base stack looks at the sequence domain. While this lost retransmission detection is not specified in the RFC series, it's an extremely valuable addition to reduce the flow completion time / IO service response time for any request-response (e.g., RPC) protocol using the TCP stack. Tracking and recovering from lost retransmissions does not yet happen by default. In FreeBSD 14, this can be activated with net.inet.tcp.do_lrd – but with FreeBSD 15, this will move to net.inet.tcp.sack.lrd and be enabled by default.

Overall, these changes make the base stack more resilient during frequently encountered pathological issues around congestion in the IP network.



Finally, the base stack (and the RACK stack) creates DSACK (RFC2883) responses when receiving spurious duplicate data packets. While receiving such DSACK information doesn't influence the stack behavior, providing this to a remote sender may permit that sender to better adjust to the specific network path behavior – e.g., Linux could increase the dupthresh or detect spurious retransmissions because of a spike in the path round-trip time (RTT).

Logging and Debugging

Over the decades, the base stack accumulated several different mechanisms to be debugged on a live system. One of the least known tools, trpt, and its support was removed in FreeBSD 14. Still, numerous other options exist (dtrace, siftr, bblog, …).

BlackBox Logging was introduced with the RACK stack and extended to cover more and more of the base stack as well. Tools are being prepared to extract internal state changes from a running system as well as extract them from core dumps – along with the packet trace itself. (See https://github.com/Netflix/tcplog_dumper and https://github.com/Netflix/read_bbrlog)

Cubic

As described in my previous article, TCP Cubic is the de facto standard congestion control algorithm in use virtually everywhere. Recently, Cubic was also made the default for FreeBSD – regardless of which TCP stack is being used.

One notable extension here is the addition of HyStart++. When a TCP session starts up, the congestion control mechanism quickly ramps up bandwidth during a phase called slow start. Traditionally, the slow start phase ends when the first indication of congestionpacket loss, or possibly an explicit congestion notification (ECN) feedback – is received. With HyStart++, which is implemented as part of the Cubic module and always enabled, the RTT is monitored. When the RTT starts to rise – possibly because network queues start forming – a less aggressive phase (conservative slow start) is entered and the RTT is still monitored because any timing-based signal is notoriously hard to obtain reliably. If it turns out that the RTT reduces again while in this conservative slow start phase, regular slow start is resumed. If not, the less aggressive sending pace in CSS limits the so-called overshoot – which is how much data may need to be recovered due to inevitable losses.

Accurate Explicit Congestion Notification

As alluded to earlier, ECN is a mechanism to avoid packet losses as the sole signal to indicate congestion events. Over the last decade, there has been a large effort in the Internet Engineering Task Force (IETF) to improve this signaling. While, originally, ECN was viewed as an "identical" signal to packet loss, a more frequent signal with different semantics was found to work better to maintain shallow (fast) queues across a large range of bandwidths. The full architecture is named Low Latency, Low Loss, Scalable (L4S). While not all the pieces in FreeBSD are currently ready to implement a proper "TCP Prague" implementation, many individual features – such as the DCTCP congestion control module and, relevant here, Accurate ECN (AccECN) – are now part of the stack in FreeBSD 14.

ECN is a mechanism to avoid packet losses as the sole signal to indicate congestion events.

While in classic ECN, only a single congestion experience mark can be signaled per RTT. This necessitates a heavy-handed management by the congestion control module. In fact, CE marks are viewed as equal to packet loss indications when adjusting the TCP bandwidth while operating in RFC3168 mode. In contrast, with AccECN, an arbitrary number of explicit congestion marks can be signaled back to the data sender by the receiver. This enables a more modulated and fine-grained signal to be extracted from the network. This becomes relevant in environments where DCTCP – with the modified, much more aggressive marking thresholds by the intermediate switches – is to be used. It is also one of the key ingredients of the Low Latency, Low Loss, Scalable (L4S) architecture – also known as TCP Prague.

Authentication and Security

Recently, the RACK stack gained the capability to fully handle MD5 authentication of TCP packets. This is an improvement that allows the use of BGP with the RACK stack – another step in making the RACK stack fully featured and useable in any generic circumstance.

For a long time, there has been a tight coupling between two of the features in RFC7323

(RFC1323) – Window Scaling and Timestamp options. In this space, we now allow either of these to be enabled independently of the other while the default still permits both to be active. This can now be achieved by setting net.inet.tcp.rfc1323 not only to on (1) or off (0), but also 2 (only window scale) and 3 (timestamps only). Furthermore, in accordance with RFC7323, it is now possible to further secure TCP sessions by requiring proper use of TCP timestamps under all circumstances. This is achieved by setting net.inet.tcp.tolerate_missing_ts to 0.

What's Next?

While the improvements of various aspects of TCP features are well into the diminishing returns phase, there are still a couple of further enhancements under discussion.

For example, an erratum to RFC2018 (Selective Acknowledgments) now allows information to be retained during a Retransmission Timeout (RTO), unlike previously. The main motivation at the time of the original standard was allowing for "reneging" by the receiver. Unless explicitly acknowledged, subsequent data could still be discarded, e.g., because of memory pressure. In practice, such reneging hardly ever happens, but retransmission timeouts during a SACK loss recovery phase do occur quite frequently. Retaining this information allows more efficient retransmissions even after an RTO. The challenge is that the base stack has implicit tight couplings with other aspects of what should happen after a retransmission timeout (such as slow starting from a very small congestion window). Also, the impact of this change after an RTO needs to be evaluated – driving some additional capabilities into the dummynet path emulator to model loss in more controllable ways.

While the improvements of various aspects of TCP features are well into the diminishing returns phase, there are still a couple of further enhancements under discussion.

RICHARD SCHEFFENEGGER has been a FreeBSD committer since April 2020 and is interested in improving the features and functionality of the TCP stack, mainly focusing on the slow path (loss recovery, congestion control handling), and actively developing enhancements such as Accurate ECN with the IETF.