TCP Cubic Is Ready to Take Flight

BY RICHARD SCHEFFENEGGER

With FreeBSD 13.0, numerous improvements were made to the cc_cubic loadable congestion control module. TCP Cubic was originally implemented by Lawrence Stewart during his time at Swinburne University of Technology, Center for Advanced Internet Architectures based on an early draft of what eventually became RFC8312. TCP Cubic has become the de-facto standard congestion control mechanism in use today.

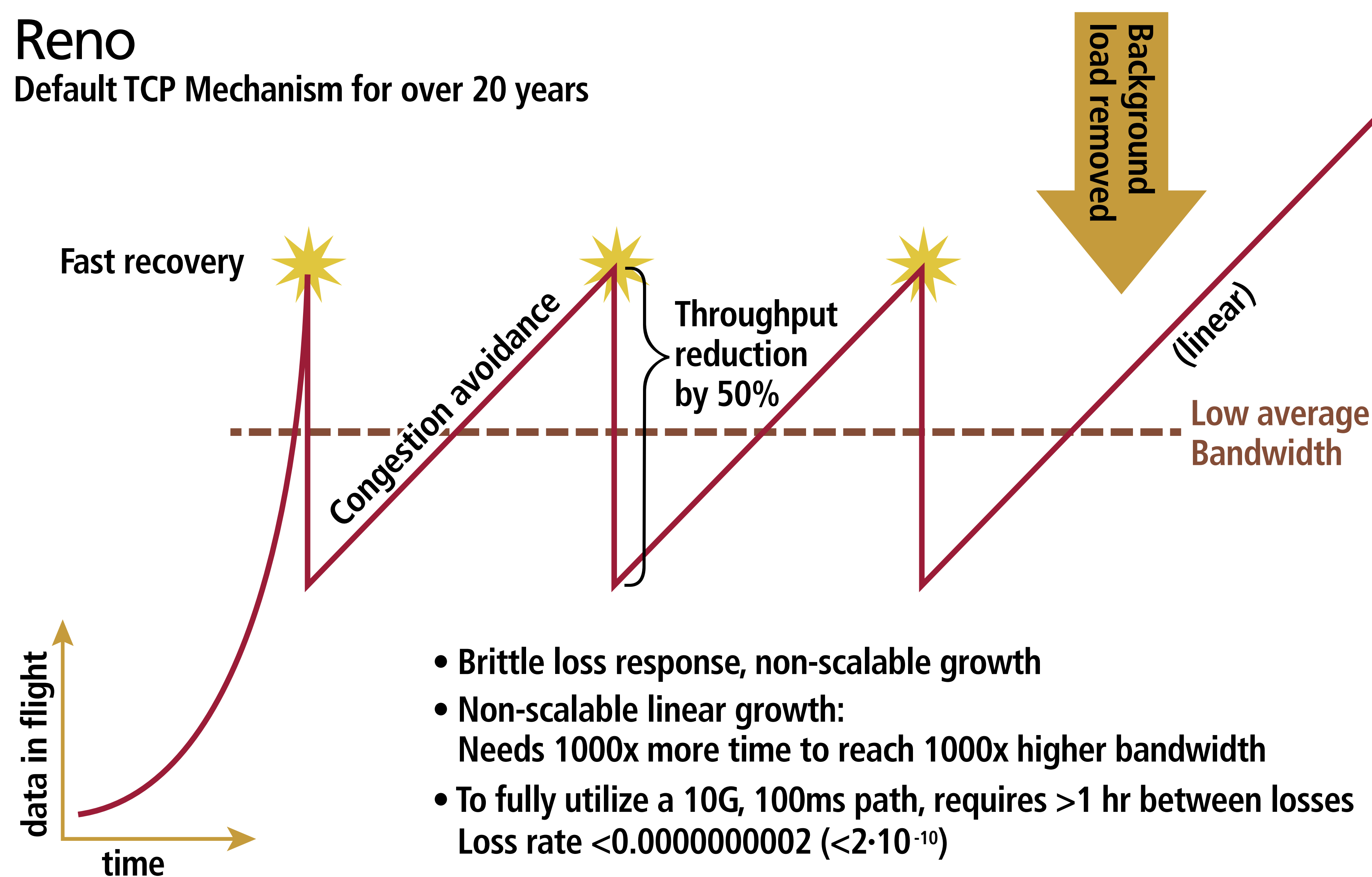
TCP Cubic

The default TCP congestion control in use by FreeBSD for the longest time is name NewReno—a variant of the Reno congestion control mechanism with improved loss recovery. The job of a congestion control algorithm is to detect and prevent an overload situation of the network where more data is injected than can be transported or delivered. NewReno used to be the gold standard in this space but does suffer a few restrictions.

> The job of a congestion control algorithm is to detect and prevent an overload situation of the network.

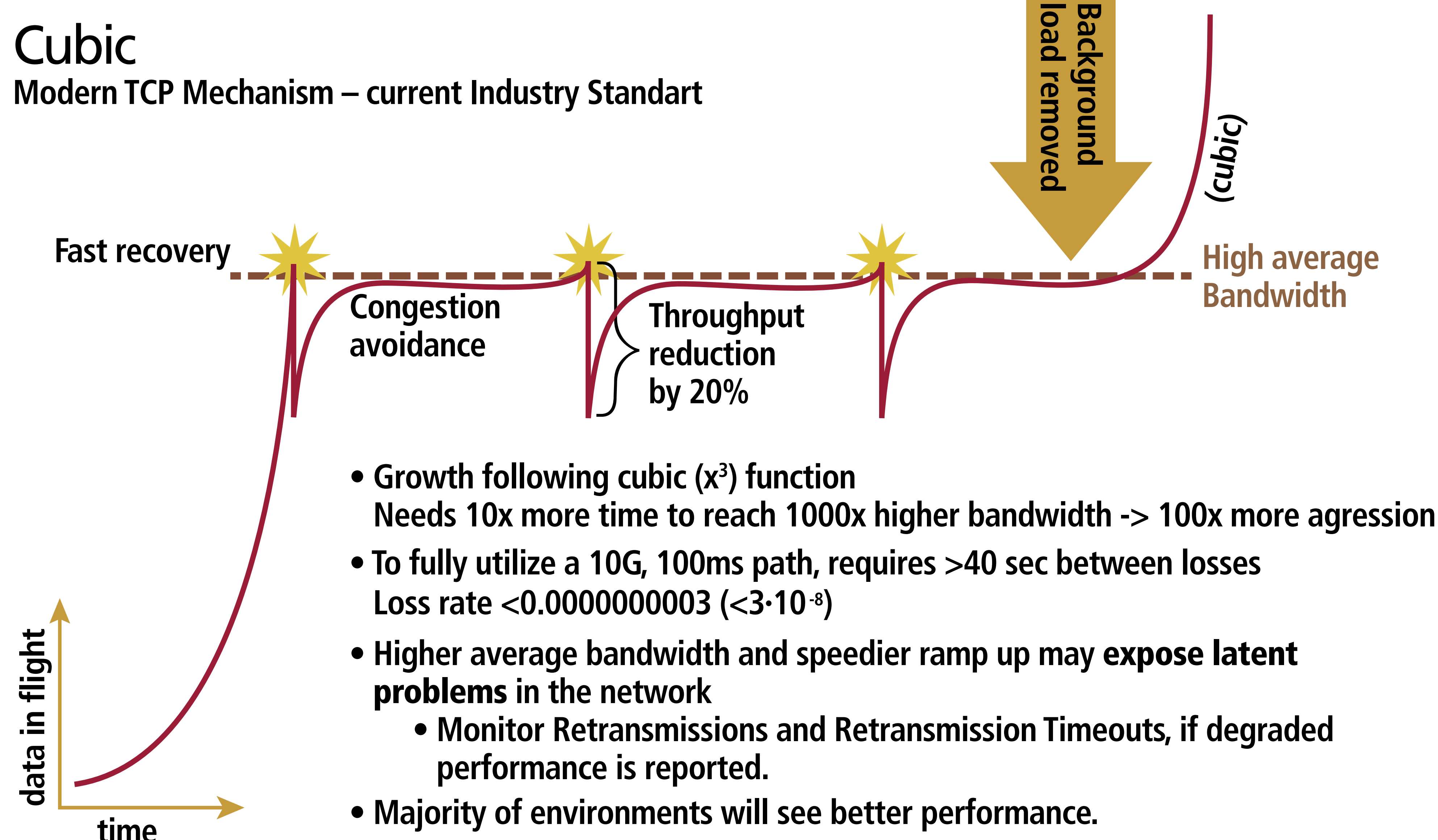
While Van Jacobson has shown that any AIMD (additive increase, multiplicative decrease) scheme exhibits a stable operation for controlling the traffic, with modern high-speed links, the time it takes NewReno to ramp up the effective transmission speed is lackluster. If an overload situation is detected--typically using an explicit signal like a packet loss or specific bits in the TCP/IP headers—NewReno will reduce the effective transmission speed—and I use this term loosely, to not get bogged down on details like available data to transmit, congestion window and burst behavior, and timing when the application is ready to send more data—to 50% of the speed at the time the overload occurred. With a sufficient amount of data to transmit, provided by the local application at a sufficiently high speed, NewReno will then ramp up the transmission speed by roughly 1 full-sized packet every round-trip time (RTT).

But running these numbers using modern networking technology, e.g. 10G links across the country with a latency of 100ms, it may take a singular NewReno session up to (5 Gbps / (1500 * 8)) * 0.1 sec RTT ~= 10 hours to ramp back up to utilize all the available bandwidth—provided no other packet drops (as indication of congestion) happen.



While TCP—when sending unlimited amounts of data—is designed to probe and eventually exceed the maximum bandwidth of the network, slow ramp-up is detrimental to this goal.

TCP Cubic addresses these limitations with two major changes. The first one is to reduce the speed only to 70% (80% in early drafts) of the transmission speed at time of overload. The second is to ramp up afterwards using a cube function which is scaled in such a way as to linger around the previous limit for a good time, but ramping up to that limit quickly--and if the available bandwidth of the network is no longer as restricted, to ramp up faster and faster, effectively matching the exponential bandwidth growth during TCP slow start.



While all these foundations were implemented—including a fast integer approximation for calculating the cube-root--some of the parameters did change between the cubic draft of 2007 and ultimate RFC8312. Thus, some work was necessary to being the existing code in-line with the RFC.

In the meantime, most other major OSs adopted Cubic as their default congestion mechanism, as in a direct competition between NewReno and Cubic, a flow using NewReno will get less share of the bandwidth available. Fortunately, Cubic was designed in a way to not fully starve out other congestion control mechanisms.

The existing code also assumed some implicit limits in the cubic code, which do not always hold with general purpose traffic patterns. A number of edge cases were not fully addressed. For example, nowadays, application-limited sessions are the norm. This is when TCP basically runs out of data to send, and all the state engines driven by processing more data have a discontinuation in time. As Cubic uses wall clock time rather than the passing of data over the session…<== rather than D23655 (cubic and slot start interaction) <== slow start…this has created some undesirable effects. (Author—is this change correct or did we misunderstand?)

While starting to run Cubic as a general-purpose congestion control on FreeBSD, the following issues were addressed without any claim of this being a complete list. Some general issues with the TCP base stack also showed up and were fixed while working on Cubic.

D26181 (editorial nit)
D26060 (adjust cwnd continuously, not only once per window – leading to massive traffic bursts)
D25976 (treat ECN like packet loss for Cubic)
D25746 (properly time the start of a cubic epoch with slowstart)
D25133 (cubic and RTO interaction)
D25065 (cubic and application limited)
D24657 (editorial)
D23655 (cubic and slot start interaction)
D23353 (cubic and ECN)
D19118 (deal with overflows during cubic math)
D18982 (prepare for good cubic math)
D18954 (cubic and After-Idle)

Overall, the foundation of cubic that has been available since FreeBSD 8.0 has been a solid foundation of the basic functionality and algorithms. A lack of production deployment left a number of corner cases and boundary conditions—e.g. for very long running TCP sessions--unchecked.

With the above improvements done, exercising the TCP Cubic variant in FreeBSD 13.0 should allow for slightly better throughput, especially across the public internet with high latency sessions. Nevertheless, additional exposure to peculiar traffic patterns may still show some shortcomings, even though the code is now in a more robust state to deal with most scenarios.

Not only was Lawrence Stewart very helpful in this improvement effort, but much of the heavy lifting was performed by Cheng Cui, especially doing regression and unit testing as well as finding all these edge cases and providing code improvements. There have also been many productive discussions on the bi-weekly FreeBSD Transport group calls.

RICHARD SCHEFFENEGGER is Consulting Solution Architect at NetApp.