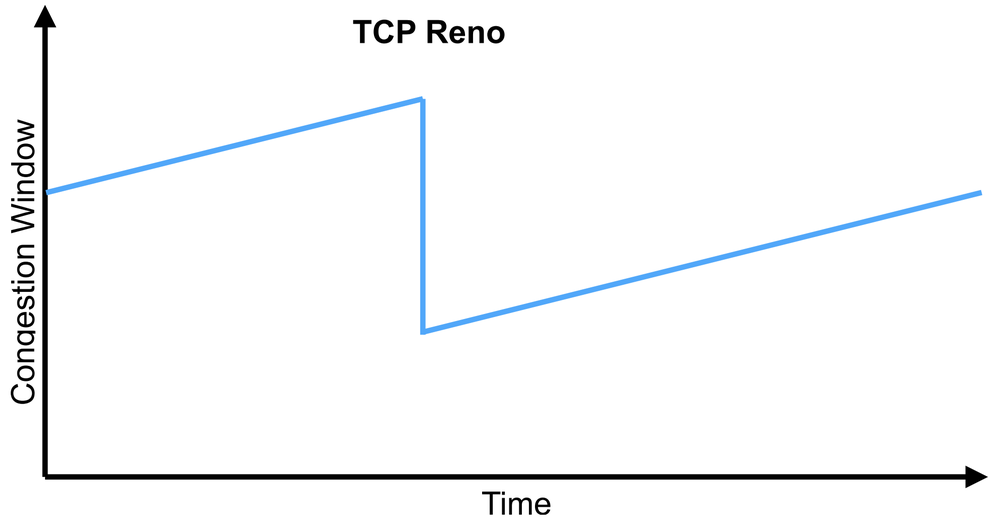
TCP Cubic

- Improving TCP performance on fast, long-distance, networks

Colin Perkins | https://csperkins.org/ | Copyright © 2020 University of Glasgow

TCP Performance on Fast Long-distance Networks

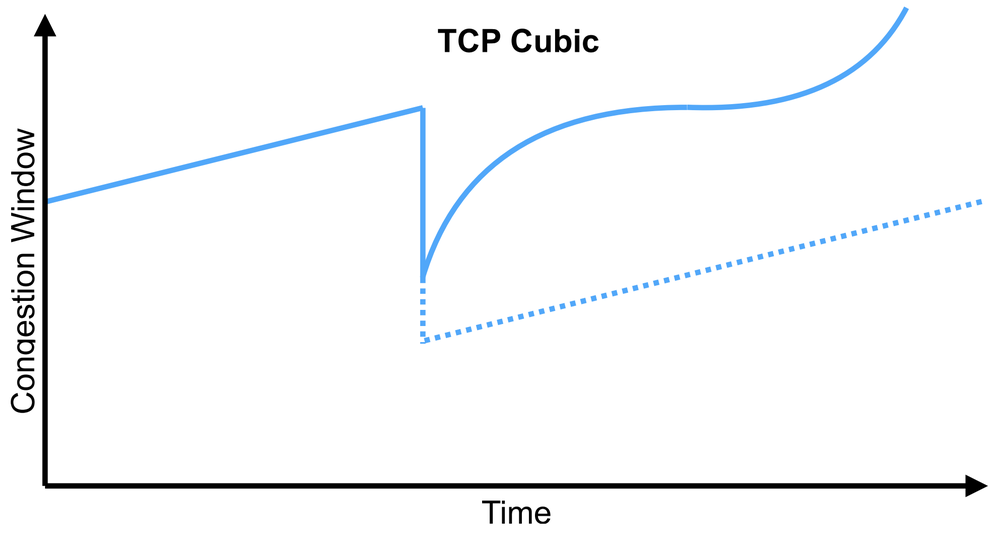
- TCP Reno can perform poorly on **fast long-distance networks**
  - e.g., multi-gigabit per second inter-continental links
  - Path with 10Gbps bandwidth and 100ms RTT requires congestion window ~100,000 packets
  - In congestion avoidance, one packet lost and detected via triple duplicate ACK halves the window, then increase by 1 packet per RTT → 50,000 RTTs to recover sending rate
    - Approximately 1.4 hours with 100ms RTT!

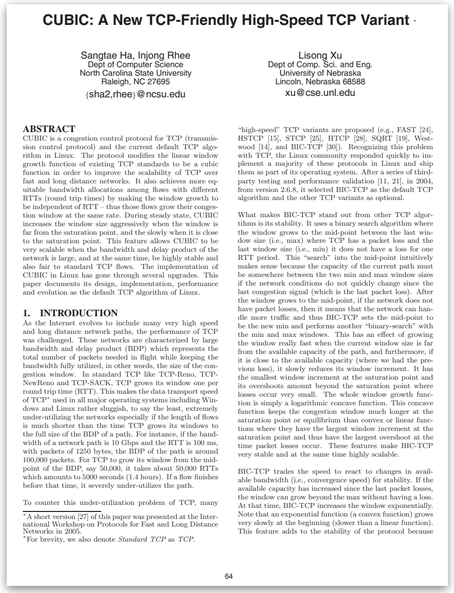


Colin Perkins | https://csperkins.org/ | Copyright © 2020 University of Glasgow

TCP Cubic

- **TCP Cubic** changes the congestion control algorithm
  - During congestion avoidance, increases congestion window faster than TCP Reno on fast long-distance networks
  - Rapidly increases congestion window after packet loss
  - Slows rate of increase as window approaches the largest successfully achieved window





CUBIC: A New TCP-Friendly High-Speed TCP Variant

Sangtae Ha, Injong Rhee
Dept of Computer Science
North Carolina State University
Raleigh, NC 27695
{sha2,rhee}@ncsu.edu

Lisong Xu
Dept of Comp. Sci. and Eng.
University of Nebraska
Lincoln, Nebraska 68588
xu@cse.unl.edu

ABSTRACT

CUBIC is a congestion control protocol for TCP (transmission control protocol) and the current default TCP algorithm in Linux. The protocol modifies the linear window growth function of existing TCP standards to be a cubic function in order to improve the scalability of TCP over fast and long distance networks. It also achieves more equitable bandwidth allocations among flows with different RTTs (round trip times) by making the window growth to be independent of RTT – thus those flows grow their congestion window at the same rate. During steady state, CUBIC increases the window size aggressively when the window is far from the saturation point, and the slowly when it is close to the saturation point. This feature allows CUBIC to be very scalable when the bandwidth and delay product of the network is large, and at the same time, be highly stable and also fair to standard TCP flows. The implementation of CUBIC in Linux has gone through several upgrades. This paper documents its design, implementation, performance and evolution as the default TCP algorithm of Linux.

1. INTRODUCTION

As the Internet evolves to include many very high speed and long distance network paths, the performance of TCP was challenged. These networks are characterized by large bandwidth and delay product (BDP) which represents the total number of packets needed in flight while keeping the bandwidth fully utilized, in other words, the size of the congestion window. In standard TCP like TCP-Reno, TCP-NewReno and TCP-SACK, TCP grows its window one per round trip time (RTT). This makes the data transport speed of TCP* used in all major operating systems including Windows and Linux rather sluggish, to say the least, extremely under-utilizing the networks especially if the length of flows is much shorter than the time TCP grows its windows to the full size of the BDP of a path. For instance, if the bandwidth of a network path is 10 Gbps and the RTT is 100 ms, with packets of 1250 bytes, the BDP of the path is around 100,000 packets. For TCP to grow its window from the mid-point of the BDP, say 50,000, it takes about 50,000 RTTs which amounts to 5000 seconds (1.4 hours). If a flow finishes before that time, it severely under-utilizes the path.

To counter this under-utilization problem of TCP, many

*A short version [27] of this paper was presented at the International Workshop on Protocols for Fast and Long Distance Networks in 2005.
†For brevity, we also denote *Standard TCP* as *TCP*.

"high-speed" TCP variants are proposed (e.g., FAST [24], HSTCP [15], STCP [25], HTCP [28], SQRT [19], Westwood [14], and BIC-TCP [30]). Recognizing this problem with TCP, the Linux community responded quickly to implement a majority of these protocols in Linux and ship them as part of its operating system. After a series of third-party testing and performance validation [11, 21], in 2004, from version 2.6.8, it selected BIC-TCP as the default TCP algorithm and the other TCP variants as optional.

What makes BIC-TCP stand out from other TCP algorithms is its stability. It uses a binary search algorithm where the window grows to the mid-point between the last window size (i.e., max) where TCP has a packet loss and the last window size (i.e., min) it does not have a loss for one RTT period. This "search" into the mid-point intuitively makes sense because the capacity of the current path must be somewhere between the two min and max window sizes if the network conditions do not quickly change since the last congestion signal (which is the last packet loss). After the window grows to the mid-point, if the network does not have packet losses, then it means that the network can handle more traffic and thus BIC-TCP sets the mid-point to be the new min and performs another "binary-search" with the min and max windows. This has an effect of growing the window really fast when the current window size is far from the available capacity of the path, and furthermore, if it is close to the available capacity (where we had the previous loss), it slowly reduces its window increment. It has the smallest window increment at the saturation point and its overshoots amount beyond the saturation point where losses occur very small. The whole window growth function is simply a logarithmic concave function. This concave function keeps the congestion window much longer at the saturation point or equilibrium than convex or linear functions where they have the largest window increment at the saturation point and thus have the largest overshoot at the time packet losses occur. These features make BIC-TCP very stable and at the same time highly scalable.

BIC-TCP trades the speed to react to changes in available bandwidth (i.e., convergence speed) for stability. If the available capacity has increased since the last packet losses, the window can grow beyond the max without having a loss. At that time, BIC-TCP increases the window exponentially. Note that an exponential function (a convex function) grows very slowly at the beginning (slower than a linear function). This feature adds to the stability of the protocol because

64

S. Ha, I. Rhee, and L. Xu, "CUBIC: A New TCP-Friendly High-Speed TCP Variant", ACM SIGOPS Operating System Review, Vol. 54, Num. 5, pages 64-74, July 2008
https://dx.doi.org/10.1145/1400097.1400105

I. Rhee, L. Xu, S. Ha, A. Zimmermann, L. Eggert, R. Scheffenegger, "CUBIC for Fast Long-Distance Networks", IETF, February 2018, RFC 8312.
https://datatracker.ietf.org/doc/rfc8312/

CC BY ND

TCP Cubic Congestion Control

- **On packet loss** during congestion avoidance, TCP Cubic reduces congestion window: $W_i = W_{i-1} \times 0.7$
  - TCP Reno uses $W_i = W_{i-1} \times 0.5$
  - TCP Cubic is more aggressive
- **After packet loss** during congestion avoidance, TCP Cubic increases the congestion window as:
  $W_{cubic} = C\,(t - K)^3 + W_{max}$
  - W_{max} is the maximum window size reached before the loss
  - t is the time since the packet loss
  - K is the time it will take to increase the window back to W_{max}, assuming no further packet losses
  - C = 0.4 is a constant that controls fairness to TCP Reno
- **Many** additional details included to ensure fairness with TCP Reno on slower, shorter-RTT, networks

Algorithm 1: Linux CUBIC algorithm (v2.2)

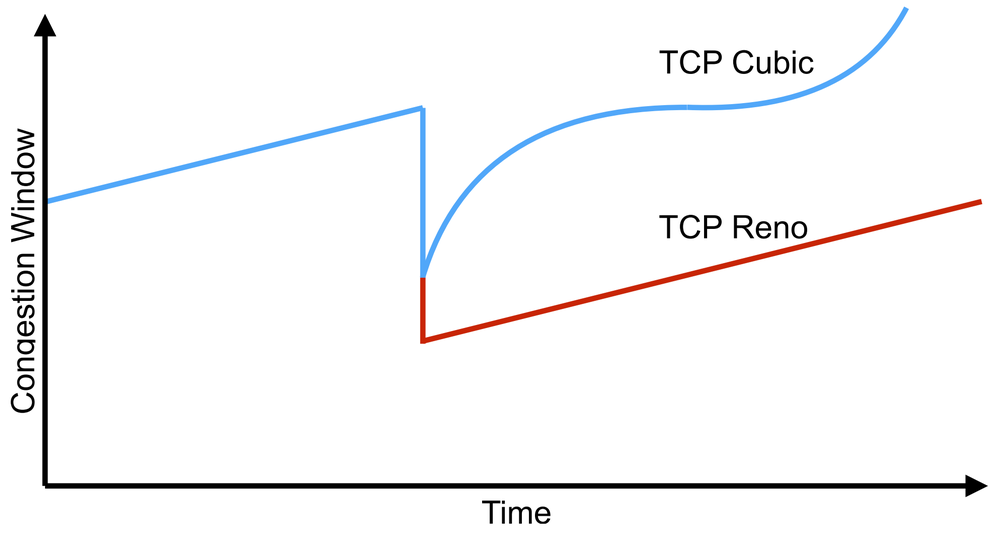
Initialization:
  $tcp\_friendliness \longleftarrow 1,\ \beta \longleftarrow 0.2$
  $fast\_convergence \longleftarrow 1,\ C \longleftarrow 0.4$
  $cubic\_reset()$
On each ACK:
begin
  if $dMin$ **then** $dMin \longleftarrow min(dMin, RTT)$
  else $dMin \longleftarrow RTT$
  if $cwnd \leq ssthresh$ **then** $cwnd \longleftarrow cwnd + 1$
  else
    $cnt \longleftarrow cubic\_update()$
    if $cwnd\_cnt > cnt$ **then**
      $cwnd \longleftarrow cwnd + 1, cwnd\_cnt \longleftarrow 0$
    else $cwnd\_cnt \longleftarrow cwnd\_cnt + 1$
end
Packet loss:
begin
  $epoch\_start \longleftarrow 0$
  if $cwnd < W_{last\_max}$ **and** $fast\_convergence$ **then**
    $W_{last\_max} \longleftarrow cwnd * \frac{(2-\beta)}{2}$ ..... **(3.7)**
  else $W_{last\_max} \longleftarrow cwnd$
  $ssthresh \longleftarrow cwnd \longleftarrow cwnd * (1 - \beta)$ ..... **(3.6)**
end
Timeout:
begin
  $cubic\_reset()$
end
$cubic\_update()$: ..... **(3.2)**
begin
  $ack\_cnt \longleftarrow ack\_cnt + 1$
  if $epoch\_start \leq 0$ **then**
    $epoch\_start \longleftarrow tcp\_time\_stamp$
    if $cwnd < W_{last\_max}$ **then**
      $K \longleftarrow \sqrt[3]{\frac{W_{last\_max} - cwnd}{C}}$
      $origin\_point \longleftarrow W_{last\_max}$
    else
      $K \longleftarrow 0$
      $origin\_point \longleftarrow cwnd$
    $ack\_cnt \longleftarrow 1$
    $W_{tcp} \longleftarrow cwnd$
  $t \longleftarrow tcp\_time\_stamp + dMin - epoch\_start$
  $target \longleftarrow origin\_point + C(t - K)^3$
  if $target > cwnd$ **then** $cnt \longleftarrow \frac{cwnd}{target - cwnd}$ .. **(3.4,3.5)**
  else $cnt \longleftarrow 100 * cwnd$
  if $tcp\_friendliness$ **then** $cubic\_tcp\_friendliness()$
end
$cubic\_tcp\_friendliness()$: ..... **(3.3)**
begin
  $W_{tcp} \longleftarrow W_{tcp} + \frac{3\beta}{2-\beta} * \frac{ack\_cnt}{cwnd}$
  $ack\_cnt \longleftarrow 0$
  if $W_{tcp} > cwnd$ **then**
    $max\_cnt \longleftarrow \frac{cwnd}{W_{tcp} - cwnd}$
    if $cnt > max\_cnt$ **then** $cnt \longleftarrow max\_cnt$
end
$cubic\_reset()$:
begin
  $W_{last\_max} \longleftarrow 0,\ epoch\_start \longleftarrow 0,\ origin\_point \longleftarrow 0$
  $dMin \longleftarrow 0,\ W_{tcp} \longleftarrow 0,\ K \longleftarrow 0,\ ack\_cnt \longleftarrow 0$
end

S. Ha, I. Rhee, and L. Xu, "CUBIC: A New TCP-Friendly High-Speed TCP Variant", ACM SIGOPS Operating System Review, Vol. 54, Num. 5, pages 64-74, July 2008 https://dx.doi.org/10.1145/1400097.1400105

Colin Perkins | https://csperkins.org/ | Copyright © 2020 University of Glasgow

TCP Cubic vs Reno

- TCP Cubic is default in most modern operating systems
  - Much more complex than TCP Reno – core response is relatively straight-forward
  - Much complexity ensures fairness with TCP Reno in its typical operating regime; improves performance for networks with longer RTT and higher bandwidth
- Both algorithms use packet loss as congestion signal and eventually fill router buffers
  - Trade latency for throughput



Colin Perkins | https://csperkins.org/ | Copyright © 2020 University of Glasgow

TCP Cubic

- Improving TCP performance on fast, long-distance, networks

Colin Perkins | https://csperkins.org/ | Copyright © 2020 University of Glasgow