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 "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 algortihms 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

*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*.

even if the protocol makes mistakes in finding the max window, it finds the next max window near the previous max point first, thus staying at the previous saturation point longer. But the exponential function quickly catches up and its increment becomes very large if the losses do not occur (in which case, the saturation point has become much larger than the previous one). Because it stays longer near the previous saturation point than other variants, it can be sluggish to find the new saturation point if the saturation point has increased far beyond the last one. BIC-TCP, however, safely reacts fast to reduced capacity because packet losses occur before the previous max and it reduces the window by a multiplicative factor. This tradeoff is a design choice of BIC-TCP. It is known [31] that available bandwidth in the Internet change over a long time scale of several hours. Given that packet losses would occur very asynchronously and also proportionally to the bandwidth consumption of a flow under a highly statistically multiplexed environment, fast convergence is a natural consequence of the network environment – something the protocol does not have to force. Thus, although BIC-TCP may converge slowly under low statistical multiplexing where only a few flows are competing, its convergence speed is not an issue under typical Internet environments.

CUBIC [27] is the next version of BIC-TCP. It greatly simplifies the window adjustment algorithm of BIC-TCP by replacing the concave and convex window growth portions of BIC-TCP by a cubic function (which contains both concave and convex portions). In fact, any odd order polynomial function has this shape. The choice for a cubic function is incidental and out of convenience. The key feature of CUBIC is that its window growth depends only on the real time between two consecutive congestion events. One congestion event is the time when TCP undergoes fast recovery. We call this real time a *congestion epoch*. Thus, the window growth becomes independent of RTTs. This feature allows CUBIC flows competing in the same bottleneck to have approximately the same window size independent of their RTTs, achieving good RTT-fairness. Furthermore, when RTTs are short, since the window growth rate is fixed, its growth rate could be slower than TCP standards. Since TCP standards (e.g., TCP-SACK) work well under short RTTs, this feature enhances the TCP-friendliness of the protocol.

The implementation of CUBIC in Linux has gone through several upgrades. The most notable upgrade is the efficient implementation of cubic root calculation. Since it requires a floating point operation, implementing it in the kernel requires some integer approximation. Initially it used the bisection method and later changed to the Newton-Raphson method which reduces the computational cost almost by 10 times. Another change to CUBIC after inception is the removal of window clamping. Window clamping was introduced in BIC-TCP where window increments are clamped to a maximum increment and was inherited to CUBIC for the first version. This forces the window growth to be linear when the target mid-point is much larger than the current window size. The authors conclude that this feature is not needed after extensive testing due to the increased stability of CUBIC. CUBIC replaced BIC-TCP as the default TCP algorithm in 2006 after version 2.6.18. The changes and upgrades of CUBIC in Linux are documented in Table 1.

The remainder of this paper is organized as follows. Section 2 gives related work, Section 3 presents the details of CUBIC algorithms in Linux, Section 4 includes the evolution of CUBIC and its implementation in Linux, and Section 5 includes discussion related to fairness property of CUBIC. Section 6 presents the results of experimental evaluation and Section 7 gives conclusion.

2. RELATED WORK

Kelly proposed Scalable TCP (STCP) [25]. The design objective of STCP is to make the recovery time from loss events be constant regardless of the window size. This is why it is called "Scalable". Note that the recovery time of TCP-NewReno largely depends on the current window size.

HighSpeed TCP (HSTCP) [15] uses a generalized AIMD where the linear increase factor and multiplicative decrease factor are adjusted by a convex fucntion of the current congestion window size. When the congestion window is less than some cutoff value, HSTCP uses the same factors as TCP. Most of high-speed TCP variants support this form of TCP compatibility, which is based on the window size. When the window grows beyond the cutoff point, the convex function increases the increase factor and reduces the decrease factor proportionally to the window size.

HTCP [28], like CUBIC, uses the elapsed time (Δ) since the last congestion event for calculating the current congestion window size. The window growth function of HTCP is a quadratic function of Δ. HTCP is unique in that it adjusts the decrease factor by a function of RTTs which is engineered to estimate the queue size in the network path of the current flow. Thus, the decrease factor is adjusted to be proportional to the queue size.

TCP-Vegas [10] measures the difference (δ) between expected throughput and actual throughput based on round-trip delays. When δ is less than a low threshold α, TCP-Vegas believes the path is not congested and thus increases the sending rate. When δ is larger than a upper threshold β, which is a strong indication of congestion, TCP-Vegas reduces the sending rate. Otherwise, TCP-Vegas maintains the current sending rate. The expected throughput is calculated by dividing the current congestion window by the minimum RTT which typically contains the delay when the path is not congested. For each round trip time, TCP-Vegas computes the actual throughput by dividing the number of packets sent by the sampled RTT.

FAST [24] determines the current congestion window size based on both round-trip delays and packet losses over a path. FAST updates the sending rate at every other RTT with rate-pacing. The algorithm estimates the queuing delay of the path using RTTs and if the delay is well below a threshold, it increases the window aggressively and if it gets closer to the threshold, the algorithm slowly reduces the increasing rate. The opposite happens when the delay increases beyond the threshold: slowly decreases the window first and then aggressively decreases the window. For packet losses, FAST halves the congestion window and enters loss recovery just like TCP.

TCP-Westwood [14] estimates an end-to-end available band-

width by accounting the rate of returning ACKs. For packet losses, unlike TCP which "blindly" reduces the congestion window to the half, TCP-Westwood sets the slow start threshold to this estimate. This mechanism is effective especially over wireless links where frequent channel losses are misinterpreted as congestion losses and thus TCP reduces the congestion window unnecessarily.

TCP-Illinois [26] uses a queueing delay to determine an increase factor α and multiplicative decrease factor β instantaneously during the window increment phase. Precisely, TCP-Illinois sets a large α and small β when the average delay d is small, which is the indication that congestion is not imminent, and sets a small α and large β when d is large because of imminent congestion.

TCP-Hybla [13] scales the window increment rule to ensure fairness among the flows with different RTTs. TCP-Hybla behaves as TCP-NewReno when the RTT of a flow is less than a certain reference RTT (e.g., 20ms). Otherwise, TCP-Hybla increases the congestion window size more aggressively to compensate throughput drop due to RTT increase.

TCP-Veno [17] determines the congestion window size very similar to TCP-NewReno, but it uses the delay information of TCP-Vegas to differentiate non-congestion losses. When packet loss happens, if the queue size inferred by the delay increase is within a certain threshold, which is the strong indication of random loss, TCP-Veno reduces the congestion window by 20%, not by 50%.

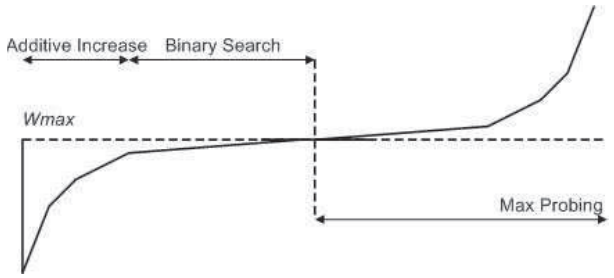
3. CUBIC CONGESTION CONTROL

3.1 BIC-TCP

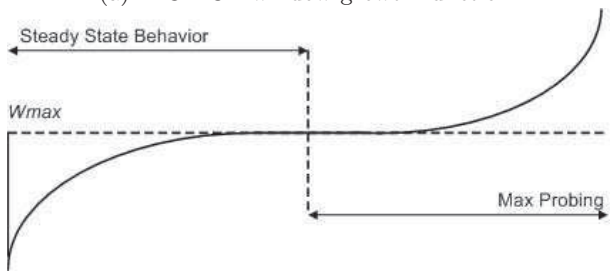
In this section, we give some details on BIC-TCP which is a predecessor of CUBIC. The main feature of BIC-TCP is its unique window growth function as discussed in the introduction. Figure 1 shows the growth function of BIC-TCP. When it gets a packet loss event, BIC-TCP reduces its window by a multiplicative factor β. The window size just before the reduction is set to the maximum W_{max} and the window size just after the reduction is set to the minimum W_{min}. Then, BIC-TCP performs a binary search using these two parameters - by jumping to the "midpoint" between W_{max} and W_{min}. Since packet losses have occurred at W_{max}, the window size that the network can currently handle without loss must be somewhere between these two numbers.

However, jumping to the midpoint could be too much increase within one RTT, so if the distance between the midpoint and the current minimum is larger than a fixed constant, called S_{max}, BIC-TCP increments *cwnd* by S_{max} (linear increase). If BIC-TCP does not get packet losses at the updated window size, that window size becomes the new minimum. This process continues until the window increment is less than some small constant called S_{min} at which point, the window is set to the current maximum. So the growth function after a window reduction will be most likely to be a linear one followed by a logarithmic one (marked as "additive increase" and "binary search" respectively in Figure 1 (a).)

If the window grows past the maximum, the equilibrium window size must be larger than the current maximum and a new maximum must be found. BIC-TCP enters a new phase called "max probing". Max probing uses a window growth function exactly symmetric to those used in additive increase and binary search (which is logarithmic; its reciprocal will be exponential) and then additive increase. Figure 1 (a) shows the growth function during max probing. During max probing, the window grows slowly initially to find the new maximum nearby, and after some time of slow growth, if it does not find the new maximum (i.e., packet losses), then it guesses the new maximum is further away so it switches to a faster increase by switching to additive increase where the window size is incremented by a large fixed increment. The good performance of BIC-TCP comes from the slow increase around W_{max} and linear increase during additive increase and max probing.



(a) BIC-TCP window growth function.



(b) CUBIC window growth function.

Figure 1: Window growth functions of BIC-TCP and CUBIC.

3.2 CUBIC window growth function

BIC-TCP achieves good scalability in high speed networks, fairness among competing flows of its own and stability with low window oscillations. However, BIC-TCP's growth function can still be too aggressive for TCP, especially under short RTT or low speed networks. Furthermore, the several different phases (binary search increase, max probing, S_{max} and S_{min}) of window control add complexity in implementing the protocol and analyzing its performance. We have been searching for a new window growth function that while retaining strengths of BIC-TCP (especially, its stability and scalability), simplifies the window control and enhances its TCP friendliness.

We introduce a new high-speed TCP variant: CUBIC. As the name of the protocol represents, the window growth function of CUBIC is a cubic function whose shape is very similar to the growth function of BIC-TCP. CUBIC uses a cubic function of the elapsed time from the last congestion event. While most alternative algorithms to Standard TCP uses a convex increase function where after a loss event, the

window increment is always increasing, CUBIC uses both the concave and convex profiles of a cubic function for window increase. Figure 1 (b) shows the growth function of CUBIC.

The details of CUBIC are as follows. After a window reduction following a loss event, it registers W_{max} to be the window size where the loss event occurred and performs a multiplicative decrease of congestion window by a factor of β where β is a window decrease constant and the regular fast recovery and retransmit of TCP. After it enters into congestion avoidance from fast recovery, it starts to increase the window using the concave profile of the cubic function. The cubic function is set to have its plateau at W_{max} so the concave growth continues until the window size becomes W_{max}. After that, the cubic function turns into a convex profile and the convex window growth begins. This style of window adjustment (concave and then convex) improves protocol and network stability while maintaining high network utilization [12]. This is because the window size remains almost constant, forming a plateau around W_{max} where network utilization is deemed highest and under steady state, most window size samples of CUBIC are close to W_{max}, thus promoting high network utilization and protocol stability. Note that protocols with convex growth functions tend to have the largest window increment around the saturation point, introducing a large burst of packet losses.

The window growth function of CUBIC uses the following function:

$$W(t) = C(t-K)^3 + W_{max} \quad (1)$$

where C is a CUBIC parameter, t is the elapsed time from the last window reduction, and K is the time period that the above function takes to increase W to W_{max} when there is no further loss event and is calculated by using the following equation:

$$K = \sqrt[3]{\frac{W_{max}\beta}{C}} \quad (2)$$

Upon receiving an ACK during congestion avoidance, CUBIC computes the window growth rate during the next RTT period using Eq. (1). It sets $W(t+RTT)$ as the candidate target value of congestion window. Suppose that the current window size is `cwnd`. Depending on the value of `cwnd`, CUBIC runs in three different modes. First, if `cwnd` is less than the window size that (standard) TCP would reach at time t after the last loss event, then CUBIC is in the *TCP mode* (we describe below how to determine this window size of standard TCP in term of time t). Otherwise, if `cwnd` is less than W_{max}, then CUBIC is in the concave region, and if `cwnd` is larger than W_{max}, CUBIC is in the convex region. Algorithm 1 shows the pseudo-code of the window adjustment algorithm of CUBIC implemented in Linux.

3.3 TCP-friendly region

When receiving an ACK in congestion avoidance, we first check whether the protocol is in the TCP region or not. This is done as follows. We can analyze the window size of TCP in terms of the elapsed time t. Using a simple analysis in [16], we can find the average window size of additive increase and multiplicative decrease (AIMD) with an additive factor

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

α and a multiplicative factor β to be the following function:

$$\frac{1}{RTT}\sqrt{\frac{\alpha}{2}\frac{2-\beta}{\beta}\frac{1}{p}} \quad (3)$$

By the same analysis, the average window size of TCP with $\alpha = 1$ and $\beta = 0.5$ is $\frac{1}{RTT}\sqrt{\frac{3}{2}\frac{1}{p}}$. Thus, for Eq. 3 to be the same as that of TCP, α must be equal to $\frac{3\beta}{2-\beta}$. If TCP increases its window by α per RTT, we can get the window size of TCP in terms of the elapsed time t as follows:

$$W_{tcp(t)} = W_{max}(1-\beta) + 3\frac{\beta}{2-\beta}\frac{t}{RTT} \quad (4)$$

If `cwnd` is less than $W_{tcp(t)}$, then the protocol is in the TCP mode and `cwnd` is set to $W_{tcp(t)}$ at each reception of ACK. The `cubic_tcp_friendliness()` in Algorithm 1 describes this behavior.

3.4 Concave region

When receiving an ACK in congestion avoidance, if the protocol is not in the TCP mode and `cwnd` is less than W_{max}, then the protocol is in the concave region. In this region, *cwnd* is incremented by $\frac{W(t+RTT)-cwnd}{cwnd}$, which is shown at (3.4) in Algorithm 1.

3.5 Convex region

When the window size of CUBIC is larger than W_{max}, it passes the plateau of the cubic function after which CUBIC follows the convex profile of the cubic function. Since `cwnd` is larger than the previous saturation point W_{max}, this indicates that the network conditions might have been perturbed since the last loss event, possibly implying more available bandwidth after some flow departures. Since the Internet is highly asynchronous, fluctuations in available bandwidth always exist. The convex profile ensures that the window increases very slowly at the beginning and gradually increases its growth rate. We also call this phase as the maximum probing phase since CUBIC is searching for a new W_{max}. As we do not modify the window increase function only for the convex region, the window growth function for both regions remains unchanged. To be exact, if the protocol is the convex region outside the TCP mode, `cwnd` is incremented by $\frac{W(t+RTT)-\texttt{cwnd}}{\texttt{cwnd}}$, which is shown at (3.5) in Algorithm 1.

3.6 Multiplicative decrease

When a packet loss occurs, CUBIC reduces its window size by a factor of β. We set β to 0.2. A side effect of setting β to a smaller value than 0.5 is slower convergence. We believe that while a more adaptive setting of β could result in faster convergence, it will make the analysis of the protocol much harder and also affects the stability of the protocol. This adaptive adjustment of β is a future research issue.

3.7 Fast Convergence

To improve the convergence speed of CUBIC, we add a heuristic in the protocol. When a new flow joins the network, existing flows in the network need to give up their bandwidth shares to allow the new flow some room for growth. To increase this release of bandwidth by existing flows, we add the following mechanism called *fast convergence*.

With fast convergence, when a loss event occurs, before a window reduction of the congestion window, the protocol remembers the last value of W_{max} before it updates W_{max} for the current loss event. Let us call the last value of W_{max} to be W_{last_max}. At a loss event, if the current value of W_{max} is less than the last value of it, W_{last_max}, this indicates that the saturation point experienced by this flow is getting reduced because of the change in available bandwidth. Then we allow this flow to release more bandwidth by reducing $W_{\max}$ further. This action effectively lengthens the time for this flow to increase its window because the reduced $W_{\max}$ forces the flow to have the plateau earlier. This allows more time for the new flow to catch up its window size. The pseudo code for this operation is shown at (3.7) in Algorithm 1.

4. CUBIC IN LINUX KERNEL

Since the first release of CUBIC to the Linux community in 2006, CUBIC has gone through several upgrades. This section documents those changes.

4.1 Evolution of CUBIC in Linux

Table 1 summarizes important updates [1] on the implementation of CUBIC in Linux since its first introduction in Linux 2.6.13. The most updates on CUBIC are focussed on performance and implementation efficiency improvements. One of notable optimizations is the improvement on cubic root calculation. The implementation of CUBIC requires solving Eq. 2, a cubic root calculation. The initial implementation of CUBIC [18] in Linux uses the bisection method. But the Linux developer community worked together to replace it with the Newton-Rhaphson method which improves the running time by more than 10 times on average (1032 clocks vs. 79 clocks) and reduces the variance in running times. CUBIC also went through several algorithmic changes to have its current form to enhance its scalability, fairness and convergence speed.

4.2 Pluggable Congestion Module

More inclusions of TCP variants to the Linux kernel has substantially increased the complexity of the TCP code in the kernel. Even though a new TCP algorithm comes with a patch for the kernel, this process requires frequent kernel recompilations and exacerbates the stability of the TCP code. To eliminate the need of kernel recompilation and help experimenting with a new TCP algorithm with Linux, Stephen Hemminger introduces a new architecture [23, 6], called *pluggable congestion module*, in Linux 2.6.13. It is dynamically loadable and allows switching between different congestion control algorithm modules on the fly without recompilation. Figure 2 shows the interface to this module, named *tcp_congestion_ops*. Each method in *tcp_congestion_ops* is a hook in the TCP code that provides access to the TCP code. A new congestion control algorithm requires to define *cong_avoid* and *ssthresh*, but the other methods are optional.

The *init* and *release* functions are called for the initialization and termination of a given TCP algorithm. *ssthresh* is the slow start threshold which is called when the given TCP detects a loss. The lower bound on congestion window is the slow start threshold, but when congestion control needs to override this lower bound, *min_cwnd* can be used for that

```
struct tcp_congestion_ops {
..
  void (*init)(struct sock *sk);
  void (*release)(struct sock *sk);
  u32 (*ssthresh)(struct sock *sk);
  u32 (*min_cwnd)(const struct sock *sk);
  void (*cong_avoid)(struct sock *sk, u32 ack,
                  u32 in_flight);
  void (*set_state)(struct sock *sk, u8 new_state);
  void (*cwnd_event)(struct sock *sk,
                   enum tcp_ca_event ev);
  u32  (*undo_cwnd)(struct sock *sk);
  void (*pkts_acked)(struct sock *sk, u32 num_acked,
                   s32 rtt_us);
  void (*get_info)(struct sock *sk, u32 ext,
                  struct sk_buff *skb);
  char  name[TCP_CA_NAME_MAX];
..
};
```

Figure 2: *tcp_congestion_ops* **structure**

purpose. *cong_avoid* is called whenever an ACK arrives and the congestion window (*cwnd*) is adjusted. For instance, in standard TCP New-Reno, when an ACK arrives, *cong_avoid* increments *cwnd* by one if the current *cwnd* is less than *ssthresh* (during slow start). Otherwise, *cong_avoid* increments *cwnd* by $\frac{1}{cwnd}$ (during congestion avoidance). *set_state* is called when the congestion control state of TCP is changed among Normal, Loss Recovery, Loss Recovery after Timeout, Reordering, and Congestion Window Reduction. *cwnd_event* is called when the events defined in *tcp_ca_event* occur. When an algorithm requires to handle one of the events, it can create a hook to *cwnd_event* which is called when the corresponding event occurs. *undo_cwnd* handles false detection of loss or timeout. When TCP realizes the change to *cwnd* is wrong, it falls back to the original *cwnd* using *undo_cwnd*. *pkts_acked* is a hook for counting ACKs; many protocols (e.g., BIC-TCP, CUBIC, and H-TCP) also use this hook to get RTT information. *get_info* is a hook for providing congestion control information to the user space.

CUBIC has been implemented as one of pluggable congestion control modules. The followings are the hooks that CUBIC use for its implementation [3].

1. *bictcp_init*: initializes private variables used for CUBIC algorithm. If *initial_ssthresh* is not 0, then set *ssthresh* to this value. If *initial_ssthresh* is properly set by users when there is no history information about the end-to-end path, it can improve the start-up behavior of CUBIC significantly.

2. *bictcp_recalc_ssthresh*: If the fast convergence mode is turned on and the current *cwnd* is smaller than *last_max*, set *last_max* to $cwnd * (1 - \frac{\beta}{2})$. Otherwise, set *last_max* to $cwnd * (1 - \beta)$. *ssthresh* is always set to $cwnd * (1 - \beta)$ because TCP needs to back off for congestion.

3. *bictcp_cong_avoid*: increases *cwnd* by computing the difference between the current *cwnd* value and its expected value of the next RTT round which is obtained by cubic root calculation.

4. *bictcp_set_state*: resets all the variables when a timeout happens.

5. *bictcp_undo_cwnd*: returns the maximum between the current *cwnd* value and the *last_max* (which is the congestion window before the drop).

6. *bictcp_acked*: maintains the minimum delay observed so far. The minimum delay is reset when a timeout happens.

5. DISCUSSION

With a deterministic loss model where the number of packets between two successive loss events is always $\frac{1}{p}$, CUBIC always operates with the concave window profile which greatly simplifies the performance analysis of CUBIC. The average window size of CUBIC can be obtained by the following function:

$$\mathbb{E}\{W_{cubic}\} = \sqrt[4]{\frac{C(4-\beta)}{4\beta}(\frac{RTT}{p})^3} \quad (5)$$

To ensure fairness to Standard TCP based on our argument in the introduction, we set C to 0.4. We find that this value of C allows the size of the TCP friendly region to be large enough to encompass most of the environments where Standard TCP performs well while preserving the scalability of the window growth function. With β set to 0.2, the above formula is reduced to the following function:

$$\mathbb{E}\{W_{cubic}\} = 1.17\sqrt[4]{(\frac{RTT}{p})^3} \quad (6)$$

(6) is used to argue the fairness of CUBIC to Standard TCP and its safety for deployment below.

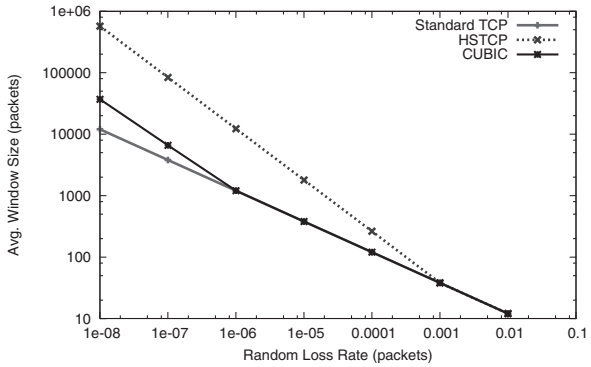
5.1 Fairness to standard TCP

In environments where standard TCP is able to make reasonable use of the available bandwidth, CUBIC does not significantly change this state.

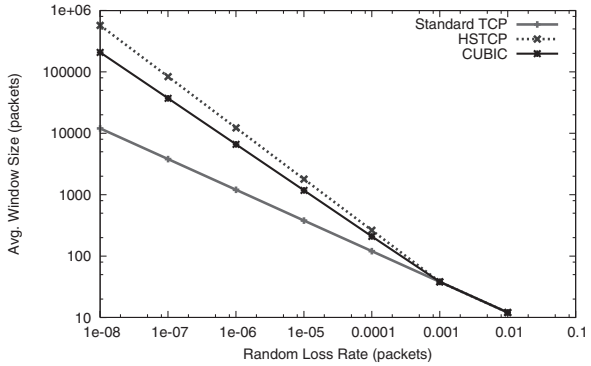
Standard TCP performs well in the following two types of networks:

1. networks with a small bandwidth-delay product (BDP).
2. networks with a short RTT, but not necessarily a small BDP.

CUBIC is designed to behave very similarly to standard TCP in the above two types of networks. Figure 3 shows the response function (average window size) of standard TCP, HSTCP, and CUBIC. The average window size of standard TCP and HSTCP is from [15]. The average window size of CUBIC is calculated by using (6) and CUBIC TCP-friendly equation in (4). Figure 3 shows that CUBIC is more friendly to TCP than HSTCP, especially in networks with a short RTT where TCP performs reasonably well. For example, in a network with RTT = 10ms and p = 10^{-6}, TCP has an average window of 1200 packets. If the packet size is 1500 bytes, then TCP can achieve an average rate of 1.44 Gbps. In this case, CUBIC achieves exactly the same rate as Standard TCP, whereas HSTCP is about ten times more aggressive than Standard TCP.



(a) Networks with 10ms RTT.



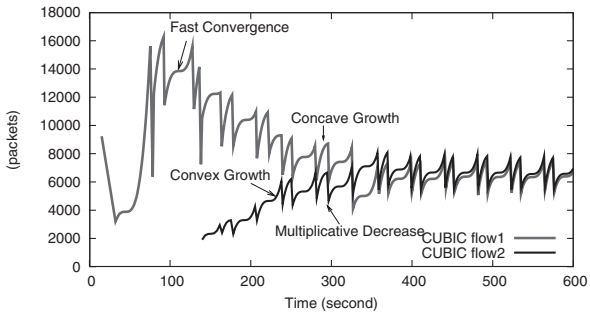
(b) Networks with 100ms RTT.

Figure 3: Response function of standard TCP, HSTCP, and CUBIC in networks with 10ms (a) and 100ms (b) RTTs respectively.

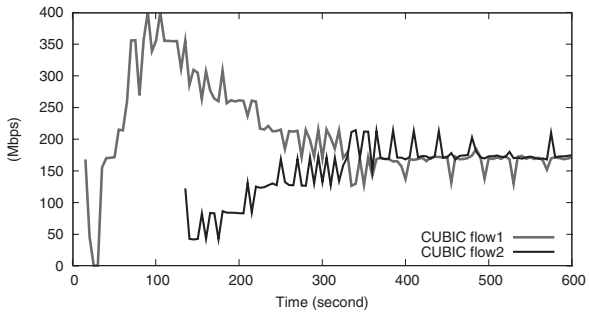
5.2 CUBIC in action

Figure 4 shows the window curve of CUBIC over the running time. This graph is obtained by running testbed experiment on a dumbbell network configuration with significant background traffic in both directions. The bottleneck capacity is 400Mbps and the RTT is set to 240ms. Drop tail routers are used. There are two CUBIC flows, and they have the same RTT and bottleneck. Note that the curves have plateaus around W_{max} which is the window size at the time of the last packet loss event. We observe that two flows use all phases of CUBIC functions over the running time and two flows converges to a fair share within 200 seconds.

Figure 5 shows the friendliness of CUBIC with respect to TCP-SACK. In this experiment, we run one CUBIC flow with one TCP-SACK flow over a short-RTT network path (8ms) and a long-RTT network path (82ms), respectively. Under the short-RTT (8ms) network where even TCP-SACK can use the full bandwidth of the path, CUBIC operates in the TCP-friendly mode. Figure 5 (a) confirms that one CUBIC flow runs in the TCP-friendly mode and shares the bandwidth fair with the other TCP-SACK flow by maintaining the congestion window of CUBIC similar with that of TCP-SACK. Under the long-RTT (82ms) network where Standard TCP has the under-utilization problem, CUBIC uses a cubic function to be scalable for this environment. Figure 5 (b) confirms that the CUBIC flow runs a cubic window growth function unlike the case with the short-RTT network where CUBIC is indistinguishable with TCP-SACK.



(a) CUBIC window curves.



(b) Throughput of two CUBIC flows.

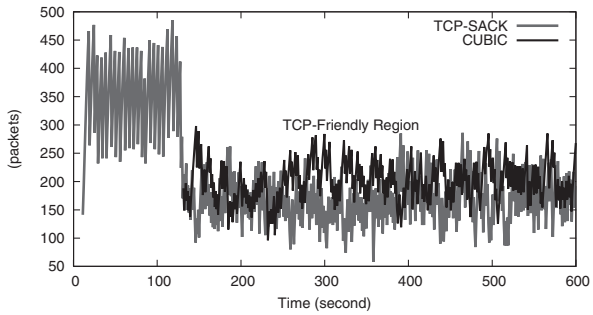
Figure 4: Two CUBIC flows with 246ms RTT.

Figure 6 shows the experiment with four TCP-SACK flows and four CUBIC flows. For this experiment, we set the bandwidth to 400Mbps, RTT to 40ms, and buffer size to 100% BDP of a flow. We observe that four flows of CUBIC converge to a fair share nicely within a short period of time. Their *cwnd* curves are very smooth and do not cause much disturbance to competing TCP flows. In this experiment, the total network utilization is around 95%: the four CUBIC flows take about 72% of the total bandwidth, the four TCP flows take 23%.

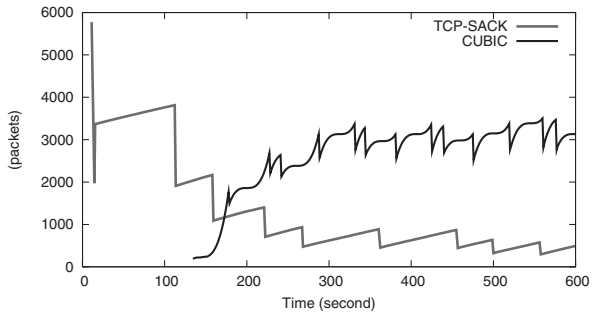
6. EXPERIMENTAL EVALUATION

6.1 Experimental Setup

We construct a dumbbell topology shown in Figure 7 where two *dummynet* routers are located at the bottleneck between two end points. Each end points consists of a set of Dell Linux servers dedicated to high-speed TCP variant flows and background traffic. Background traffic is generated by using a modification of a web-traffic generator, called *Surge* [9] and Iperf [2]. We modified *Surge* to generate a wider range of flow sizes in order to increase variability in cross traffic because medium size flows tend to fully execute the slow start and increase the variability in available bandwidth. The RTT of each background traffic is randomly selected from an exponential distribution found in [7]. The socket buffer size of background traffic machines is fixed to default 64KB while high-speed TCP machines are configured to have a very large buffer so that the transmission rates of high-speed flows are only limited by the congestion control algorithm. Two dummynet routers and four high-speed TCP machines are tuned to generate or forward traffic close to 1Gbps. The details of system tuning for both Linux and



(a) RTT 8ms.



(b) RTT 82ms.

Figure 5: One CUBIC flow and one TCP-SACK flow. Bandwidth is set to 400Mbps.

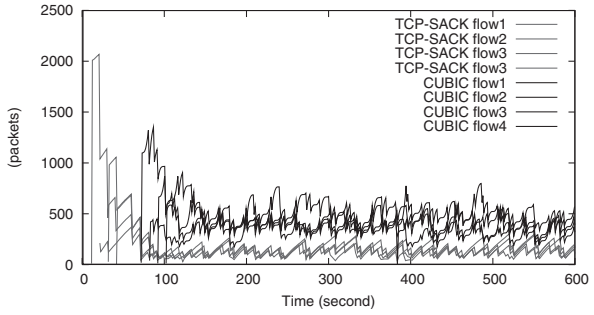


Figure 6: Four TCP-SACK flows and four CUBIC flows over 40ms RTT

FreeBSD systems are shown in [5]. Note that Netem [22] in Linux provides the same functionality with the dummynet software in FreeBSD. For this experiment, the maximum bandwidth of the bottleneck router is set to 400Mbps. The bottleneck buffer size is set to 100% BDP if it is not explicitly specified. The amount of background traffic comparable to around 15% of the bottleneck bandwidth is pushed into forward and backward directions of the dumbbell. We use the drop-tail router at the bottleneck.

6.2 Intra-Protocol Fairness

We measure the intra-protocol fairness between two flows of a protocol with the same RTT. We use a throughput ratio between these two flows for representing the intra-protocol fairness. This metric represents a degree of bandwidth shares between two flows of the same protocol. For this experiment, we vary RTTs between 16ms and 324ms and test CUBIC, BIC-TCP, HSTCP, and TCP-SACK protocols. Figure 8 (a) and 8 (b) show the intra-protocol fairness and link utilization for the tested protocols. CUBIC and BIC-TCP show higher fairness index than TCP-SACK and HSTCP, representing better fair sharing between the flows. With 16ms RTT, TCP-SACK shows the best fairness index indicating that Standard TCP works fairly well under small RTT networks. CUBIC, BIC-TCP, and HSTCP utilize the link regardless of RTTs while TCP-SACK suffers under-utilization with larger RTTs.

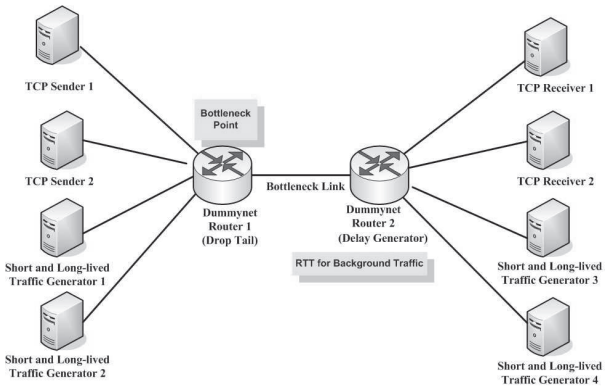


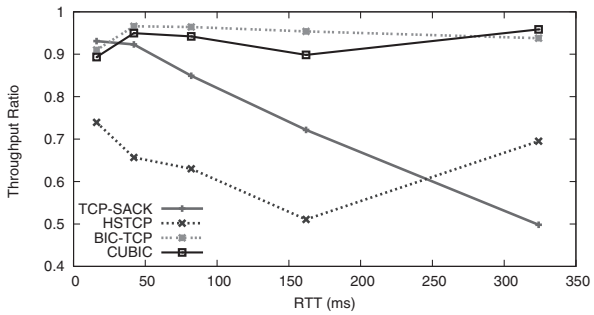
Figure 7: Testbed

6.3 Inter-RTT Fairness

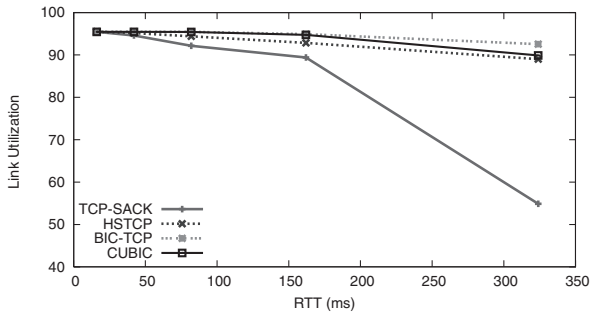
We measure the fairness in sharing the bottleneck bandwidth between two competing flows that have different RTTs. For this experiment, we fix RTT of one flow to 162ms and vary RTT of the other flow between 16ms and 164ms. This setting gives us the RTT ratio up to 10. We test CUBIC, BIC-TCP, HSTCP, and TCP-SACK protocols. Figure 9 (a) shows that TCP-SACK achieves RTT fairness linearly proportional to the inverse of the RTT ratio, which means that the short RTT flow has proportionally more bandwidth shares than the longer RTT flow. Even though there is no commonly accepted notion of RTT-fairness, we think the proportional fairness of TCP-SACK is desirable because long RTT flows tend to use more resources along the longer path than short RTT flows. But some of high-speed protocols are desiged to provide an equal bandwidth sharing among the flows with different RTTs (e.g., H-TCP and FAST). Based on this notion of RTT fairness, if the RTT fairness of a protocol has a similar slope with TCP-SACK, we can say the protocol is "acceptable". Figure 9 (a) confirms that CUBIC has a similar slope with TCP-SACK but with a higher fairness ratio indicating better share of resources (bandwidth) while HSTCP fails in achieving a similar slope. Even though BIC-TCP shows the similar slope with TCP-SACK, it shows the lowest fairness ratios among tested protocols. This is what CUBIC improves over BIC-TCP for RTT-fairness. We also observe that even though two HSTCP flows fully utilize the link regardless of their RTT ratio (See Figure 9 (b)), the slow convergence of HSTCP flows hinders even two flows of the same RTT from reaching to a fair share within a reasonable amount of time.

6.4 Impact on standard TCP traffic

As many new high-speed TCP protocols modify the window growth function of TCP in a more scalable fashion,

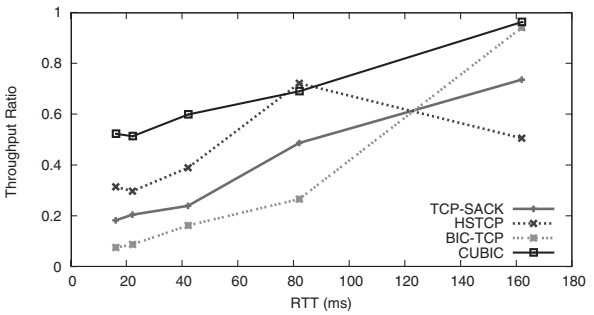


(a) Intra-Protocol Fairness.

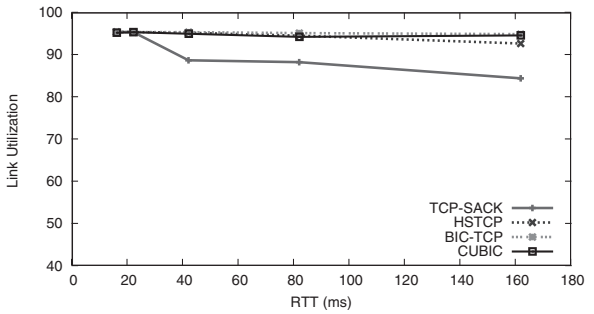


(b) Link Utilization.

Figure 8: Intra-protocol fairness (a) and link utilization (b). The bottleneck bandwidth is set to 400Mbps and 2Mbyte bottleneck buffer is used. RTT is varied between 16ms and 324ms and two flows have the same RTT.



(a) Inter-RTT Fairness.



(b) Link Utilization.

Figure 9: Inter-RTT fairness. The bottleneck bandwidth is set to 400Mbps and 2Mbyte buffer is used. One flow has a fixed RTT of 162ms and the other flow varies its RTT from 16ms to 162ms.

these new protocols tend to affect the performance of Standard TCP flows which share the same bottleneck link along the path. As being fair to Standard TCP is critical to the safety of the protocol, we need to make sure that the window growth function of a new protocol does not unfairly affect the Standard TCP flows.

In this experiment, we measure how much these high-speed protocols steal the bandwidth from competing TCP-SACK flows. By following the scenarios shown in the recent evaluation proposal [8], we first measure the throughput shares of four TCP-SACK flows when they are competing with the other four TCP-SACK flows. After that, we replace four flows with a new protocol. We measure the share of TCP-SACK flows and the other four TCP variant flows at each run and report only the accumulated average of their bandwidth shares. We test CUBIC, BIC-TCP, HSTCP, and TCP-SACK.

Figure 10 (a) shows the relationship between RTT and the throughput share between a new protocol flows and TCP-SACK flows. We fix the bottleneck bandwidth to 400Mbps and vary RTT between 10ms and 160ms. Clearly, TCP-SACK flows do not fully utilize the bottleneck bandwidth as RTT increases due to its slow window growth function. With 400Mbps and 160ms RTT, 8 TCP-SACK flows achieve around 80% of the link bandwidth, but the underutilization will be very serious for larger BDP path and with small number of flows. CUBIC, BIC-TCP, and HSTCP fully utilize the link, thanks to their scalable window growth functions. All the tested high-speed protocols grab more bandwidth share from TCP-SACK as RTT increases. Also we confirm that CUBIC gives more room to TCP-SACK than BIC-TCP and HSTCP for whole range of tested RTTs while achieving full utilization of the path. This is one of the design objective of CUBIC that it operates like BIC-TCP and be nice to other flows in the network. As RTT increases, CUBIC, BIC-TCP and HSTCP steal more bandwidth from TCP-SACK. Some amount of bandwidth they steal is from the amount of bandwidth that TCP-SACK doesn't utilize.

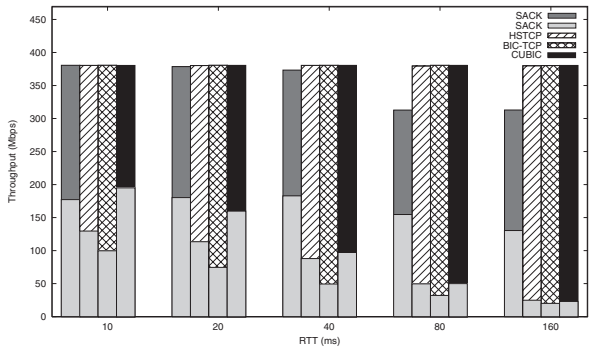
Figure 10 (b) and 10 (c) show the performance results regarding the TCP friendliness over short-RTT networks (10ms RTT) and long-RTT networks (100ms RTT), respectively. According to [15], under high loss rate regions (small-RTT networks) where TCP is well-behaving, the protocol must behave like TCP, and under low loss rate regions (large-RTT networks) where TCP has a low utilization problem, it can use more bandwidth than TCP. As shown in Figure 10 (b), with 10ms RTT, we can see that TCP-SACK still uses the full bandwidth. In this region, all high-speed protocols need to be friendly to TCP-SACK by following the arguments above. Interestingly, CUBIC behaves more TCP-friendly even comparing to TCP-SACK for certain bandwidths. Rather than stealing the bandwidth from TCP-SACK flows, CUBIC flows employs a window growth function that is comparable to TCP-SACK, so that competing TCP-SACK flows have the same chance with CUBIC flows for grabbing the bandwidth shares. However, BIC-TCP and HSTCP show a tendency to operate in a scalable mode (being more aggressive) as the link speed increases. Even

though the graph doesn't show the results corresponding to the link speed beyond 400Mbps, it is obvious that a scalable mode of BIC-TCP and HSTCP will deprive most of bandwidth share of TCP-SACK. As most high-speed TCP protocols including BIC-TCP and HSTCP achieve TCP friendliness by having some form of "TCP modes" during which they behave in the same way as TCP. BIC-TCP and HSTCP enter their TCP mode when the window size is less than 14 and 38 packets, respectively. Therefore, even with 1ms RTT, if BDP is larger than 38 packets, HSTCP will operate in a scalable mode. This is the limitation when the protocol uses a fixed cutoff for detecting a TCP-friendly region. CUBIC defines a TCP-friendly region in real-time; therefore, CUBIC doesn't have this scalability problem.

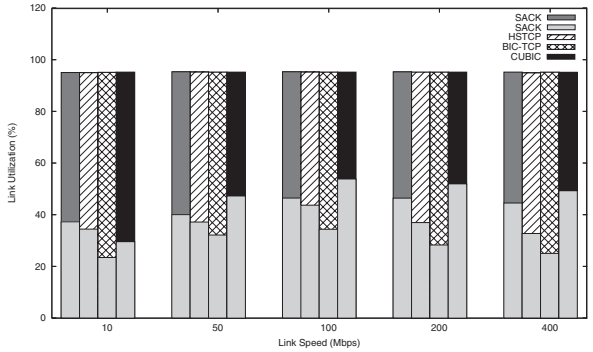
Figure 10 (c) shows the results with 100ms RTT. All four protocols show reasonable friendliness to TCP. As the bandwidth gets larger than 10Mbps, the throughput ratio drops quite rapidly. As CUBIC, like BIC-TCP and HSTCP, regards this operating region is out of TCP-friendly region and behaves to be scalable to this environment. CUBIC and BIC-TCP show a similar aggressiveness which is slightly more aggressive[†] than HSTCP especially for the bandwidth less than 100Mbps. Through an extensive testing [4], we confirm that this doesn't highly impact on the performance of TCP-SACK.

7. CONCLUSION

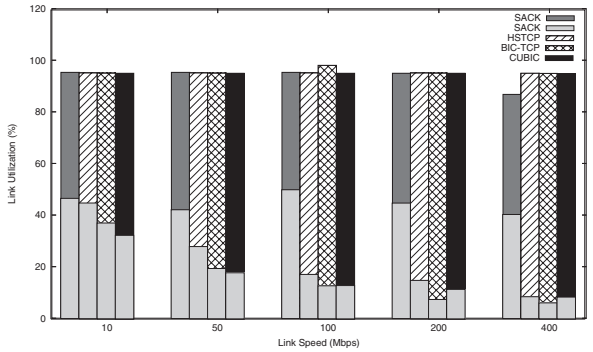
We propose a new TCP variant, called CUBIC, for fast and long distance networks. CUBIC is an enhanced version of BIC-TCP. It simplifies the BIC-TCP window control and improves its TCP-friendliness and RTT-fairness. CUBIC uses a cubic increase function in terms of the elapsed time since the last loss event. In order to provide fairness to Standard TCP, CUBIC also behaves like Standard TCP when the cubic window growth function is slower than Standard TCP. Furthermore, the real-time nature of the protocol keeps the window growth rate independent of RTT, which keeps the protocol TCP friendly under both short and long RTT paths. We show the details of Linux CUBIC algorithm and implementation. Through extensive testing, we confirm that CUBIC tackles the shortcomings of BIC-TCP and achieves fairly good Intra-protocol fairness, RTT-fairness and TCP-friendliness.



(a) Bandwidth 400Mbps with varying RTT



(b) RTT 10ms with varying bandwidth



(c) RTT 100ms with varying bandwidth

Figure 10: Impact on standard TCP traffic.

8. REFERENCES

[1] Git logs for CUBIC updates. *http://git.kernel.org/?p=linux/kernel/git/davem/net-2.6.git;a=history;f=net/ipv4/tcp_cubic.c;h=eb5b9854c8c7330791ada69b8c9e8695f7a73f3d;hb=HEAD*.

[2] Iperf. *http://sourceforge.net/projects/iperf* .

[3] Linux CUBIC source navigation. *http://lxr.linux.no/linux/net/ipv4/tcp_cubic.c*.

[4] TCP Testing Wiki. *http://netsrv.csc.ncsu.edu/wiki/index.php/TCP_Testing*.

[†]We used the latest update of CUBIC (v2.2) which improved the scalability and convergence speed of the protocol, which doesn't clamp the increment in both convex and cocave regions. A slight increase of aggressivenss is the trade-off between scalability and TCP-friendlines. Our extensive testing confirms that CUBIC has better scalability and convergence speed with this small change (trade-off) while obtaining reasonable TCP-friendliness.

[5] Testing setup for Linux and FreeBSD. *http://netsrv.csc.ncsu.edu/wiki/index.php/Testing_Setup_of_kernel_2.6.23.9*.

[6] Pluggable congestion avoidance modules. *http://lwn.net/Articles/128681/* (2005).

[7] Aikat, J., Kaur, J., Smith, F., and Jeffay, K. Variability in TCP round-trip times. In *Proceedings of the ACM SIGCOMM Internet Measurement Conference* (Miami, FL, October 2003).

[8] Andrew, L., Marcondes, C., Floyd, S., Dunn, L., Guillier, R., Gang, W., Eggert, L., Ha, S., and Rhee, I. Towards a common TCP evaluation suite. In *Proceedings of the fourth PFLDNet Workshop* (UK, March 2008).

[9] Barford, P., and Crovella, M. Generating representative web workloads for network and server performance evaluation. In *Measurement and Modeling of Computer Systems* (1998), pp. 151–160.

[10] Brakmo, L., and Peterson, L. TCP vegas: End to end congestion avoidance on a global internet. *IEEE Journal of*

Selected Areas in Communications (October 1995).

[11] Bullot, H., Cottrell, R. L., and Hughes-Jones, R. Evaluation of advanced TCP stacks on fast long-distance production networks. In *Proceedings of the third PFLDNet Workshop* (Illinois, February 2004).

[12] Cai, H., Eun, D., Ha, S., Rhee, I., and Xu, L. Stochastic ordering for internet congestion control and its applications. In *Proceedings of IEEE INFOCOM* (Anchorage, Alaska, May 2007).

[13] Caini, C., and Firrincieli, R. TCP hybla: a TCP enhancement for heterogeneous networks. *International Journal of Satellite Communication and Networking 22*, 5 (September 2004), 547–566.

[14] Casetti, C., Gerla, M., Mascolo, S., Sanadidi, M. Y., and Wang, R. TCP Westwood: Bandwidth estimation for enhanced transport over wireless links. In *Proceedings of ACM Mobicom* (Rome, Italy, July 2001).

[15] Floyd, S. HighSpeed TCP for Large Congestion Windows. RFC 3649 (Experimental), Dec. 2003.

[16] Floyd, S., Handley, M., and Padhye, J. A Comparison of Equation-Based and AIMD Congestion Control, May 2000.

[17] Fu, C. P., and Liew, S. C. TCP Veno: TCP Enhancement for Transmission Over Wireless Access Networks. *IEEE Journal of Selected Areas in Communications* (Feb 2003).

[18] Ha, S. Cubic v2.0-pre patch. *http://netsrv.csc.ncsu.edu/twiki/pub/Main/BIC/cubic-kernel-2.6.13.patch*.

[19] Hatano, T., Fukuhara, M., Shigeno, H., and Okada, K. TCP-friendly SQRT TCP for High Speed Networks. In *Proceedings of APSITT* (November 2003), pp. 455–460.

[20] Hemminger, S. Cubic root benchmark code. *http://lkml.org/lkml/2007/3/13/331*.

[21] Hemminger, S. Linux TCP Performance Improvements. *Linux World 2004* (2004).

[22] Hemminger, S. Network Emulation with NetEm. *Linux Conf Au* (2005).

[23] Hemminger, S. TCP infrastructure split out. *http://lwn.net/Articles/128626/* (2005).

[24] Jin, C., Wei, D. X., and Low, S. H. FAST TCP: motivation, architecture, algorithms, performance. In *Proceedings of IEEE INFOCOM* (Hong Kong, March 2004).

[25] Kelly, T. Scalable TCP: Improving performance in highspeed wide area networks. *ACM SIGCOMM Computer Communication Review 33*, 2 (April 2003), 83–91.

[26] Liu, S., Basar, T., and Srikant, R. TCP-Illinois: A loss and delay-based congestion control algorithm for high-speed networks. In *Proceedings of VALUETOOLS* (Pisa, Italy, October 2006).

[27] Rhee, I., and Xu, L. CUBIC: A new TCP-friendly high-speed TCP variant. In *Proceedings of the third PFLDNet Workshop* (France, February 2005).

[28] Shorten, R. N., and Leith, D. J. H-TCP: TCP for high-speed and long-distance networks. In *Proceedings of the Second PFLDNet Workshop* (Argonne, Illinois, February 2004).

[29] Tarreau, W. Cubic optimization. *http://git.kernel.org/?p=linux/kernel/git/davem/net-2.6.git;a=commit;h=7e58886b45bc4a309aeaa8178ef89ff767daaf7f*.

[30] Xu, L., Harfoush, K., and Rhee, I. Binary increase congestion control for fast long-distance networks. In *Proceedings of IEEE INFOCOM* (Hong Kong, March 2004).

[31] Zhang, Y., Duffield, N., Paxson, V., and Shenker, S. On the constancy of Internet path properties. In *Proceedings of ACM SIGCOMM Internet Measurement Workshop* (November 2001).

Version	Kernel	Updates
2.0-pre	2.6.13	The authors releases the first CUBIC implementation in Linux to the Linux community [18].
2.0	2.6.15	CUBIC is officially included in the Linux kernel.
	2.6.18	CUBIC replaces BIC-TCP as the default TCP protocol in Linux kernel.
	2.6.19	The original implementation of CUBIC has a scaling bug. It has taken about a month to fix this bug since CUBIC replaced BIC-TCP.
	2.6.21	Its original implementation by the authors are optimized by the Linux developer for better performance [20, 29]. In particular, the cubic root calculation in CUBIC, originally implemented in the bisection method, is now replaced by a Newton-Raphson method with table loopups for small values. This results in more than 10 times performance improvement in the cubic root calculation. On average, the bisection method costs 1032 clocks while the improved version costs only 79 clocks.
2.1	2.6.22	The original implementation of CUBIC clamped the maximum window increment to 32 packets per RTT. This feature is inherited from BIC-TCP (S_{max}). An extensive lab testing confirmed that CUBIC can safely remove this window clamping in the concave region. This enhances the scalability of CUBIC over very large BDP network paths. This is incorporated in CUBIC 2.1 (Linux 2.6.22).
	2.6.22-rc4	CUBIC improves slow start for fast start-up by removing *initial_ssthresh*.
	2.6.23	The use of received timestamp option value from RTT calculation is removed for preventing possible malicious receiver attacks that reports wrong timestamps to reduce RTTs for more throughput.
2.2	2.6.25-rc3	The window clamping during the convex growth phase is also removed. This feature allows CUBIC to improve its convergence speed while maintaining its fairness and TCP friendliness.

Table 1: CUBIC version history