A TCP CUBIC Implementation in ns-3

Brett Levasseur
MIT Lincoln Laboratory
244 Wood St
Lexington, MA
brettl@ll.mit.edu

Mark Claypool
Worcester Polytechnic Institute
100 Institute Rd
Worcester, MA
claypool@cs.wpi.edu

Robert Kinicki
Worcester Polytechnic Institute
100 Institute Rd
Worcester, MA
rek@cs.wpi.edu

ABSTRACT

To allow network researchers to simulate currently deployed networks, ns-3 needs to incorporate as many standard networking technologies as possible. One such technology is TCP CUBIC, the default TCP congestion control algorithm in the Linux kernel and one of the most widely deployed variants of TCP. Despite this prevalence, ns-3 does not natively currently support TCP CUBIC. This paper presents the design and implementation of CUBIC in ns-3 based on literature describing the CUBIC algorithm and examination of Linux kernel source. Verification and validation of our ns-3 implementation, both within the simulator and in comparison to Linux measurements, show the ns-3 CUBIC mimics the key features of Linux CUBIC and performs better than TCP NewReno, the default TCP congestion control algorithm in ns-3.

1. INTRODUCTION

TCP is the dominant transport protocol on the Internet, accounting for the vast majority of the number of flows and bytes transferred. To deal with network congestion, TCP uses congestion control algorithms to vary its transmission rate and manage on lost packets. There have been numerous variants proposed for TCP [6], but the current default used in the Linux kernel is CUBIC [5]. Along with its predecessor BIC [8], CUBIC and the Microsoft Windows Compound [7] have been the dominant three TCP algorithms for years. A 2011 study [9] of about 30,000 Web servers show about 25% use TCP CUBIC, 20% use TCP BIC and 15% to 25% use TCP Compound. An important step in the development of these congestion control algorithms, as well as for developing new TCP variants and other aspects of computer networking, is simulation.

Computer simulator provide valuable insight into potential implementations before extending the considerable effort that may be required to build a technology. Simulations also allow repeatability of measurements and can provide in-depth details on the inner workings of technologies that may not normally be available in actual measurements. The ns-3 network simulator [2], which succeeds the popular ns-2, is used by researchers around the world to simulate a variety of complex networking configurations and technologies. ns-3 includes implementations of some TCP variants, including TCP based on RFC 793, Tahoe, Reno, NewReno and Westwood. However, Yang et al. [9] found that these algorithms (not including no congestion control and Tahoe, which was not examined) accounted for only about 5% to 15% of the TCP variants in their sample. Unfortunately, ns-3 does not currently provide a native implementation of TCP CUBIC, TCP BIC or TCP Compound. To accurately characterize real world network behavior, ns-3 should provide implementations of the most widespread congestion control algorithms (e.g., TCP CUBIC) in use on the Internet today.

This paper presents the design and implementation of CUBIC in ns-3. In order to match the ns-3 TCP CUBIC with the most widely deployed Internet version, our implementation focuses on the CUBIC algorithm found in Linux. However, the original CUBIC algorithm introduced by Ha et al. [5] differs somewhat from the current Linux kernel implementation. To deal with these discrepancies, our implementation follows the basic details of the CUBIC algorithm as much as possible, using the Linux-specific details provided in the kernel code when appropriate. Verification of our ns-3 CUBIC implementation is provided via a demonstration of the functioning ns-3 code, and validation is provided by comparing performance for TCP CUBIC in Linux TCP CUBIC in ns-3 with our ns-3 implementation. A final comparison of TCP CUBIC to TCP New Reno, the default TCP for ns-3, is provided to highlight the differences in performance between these two TCP variants.

The rest of this paper is organized as follows: Section 2 reviews basic TCP congestion control concepts; Section 3 briefly discusses prior related work; Section 4 presents an overview of CUBIC; Section 5 describes the CUBIC implementation in Linux; Section 6 presents our ns-3 implementation of CUBIC; Section 7 details the verification and validation of our implementation; and Section 8 summarizes our conclusions and presents possible future work.

2. TCP BASICS

TCP provides a reliable, ordered delivery of packets between computers connected on the Internet. TCP senders add sequence numbers to packets and TCP receivers acknowledge packet receptions. Unacknowledged TCP packets are retransmitted based on information about successfully received sequence numbers and, in some cases, timeouts.

WNS3, May 07 2014, Atlanta, GA, USA.
Copyright ACM 978-1-4503-3003-9/14/05.

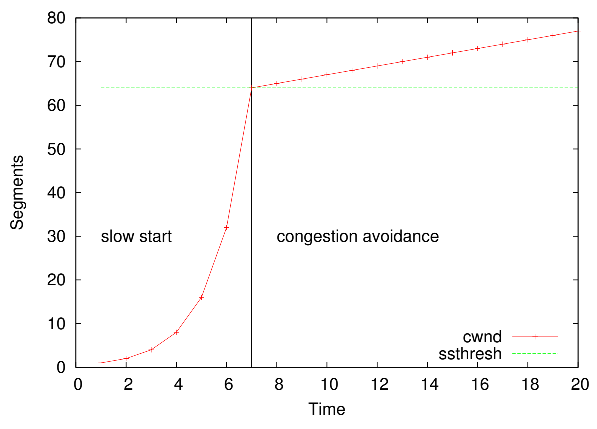


Figure 1: Simple example of congestion window growth in TCP.

The various TCP congestion control algorithms proposed over the last 25 years roughly aim to maximize throughput while avoiding congestion at Internet routers.

While many TCP congestion control algorithms have been proposed, all have several features in common. The TCP sender keeps a dynamic congestion control window (cwnd) that limits the number of outstanding transmitted packets before an acknowledgement (ACK) needs to arrive from the receiver. Figure 1 shows an example of TCP cwnd growth. The TCP connection begins in *slow start* mode with cwnd = 1 and doubles the size of cwnd every packet round trip time (RTT) (i.e., the time from when a packet is sent until an ACK is received back at the sender). TCP stays in slow start mode until cwnd reaches the slow start threshold (ssthresh), indicated by the horizontal green line in Figure 1. Upon reaching ssthresh, TCP switches to the *congestion avoidance* phase, where cwnd is slowed growing linearly to avoid network congestion. While the exact management of the cwnd growth depends on the specific congestion control algorithm used, Section 4 explains the specifics of the CUBIC's cwnd growth during congestion avoidance.

3. RELATED WORK

While there is no CUBIC model currently built inside of ns-3, CUBIC and other congestion control algorithms can still be used, albeit only on some platforms, by linking into the local operating systems network code to run simulations. There are two ns-3 components that support this: Direct Code Execution (DCE) and the Network Simulation Cradle (NSC).[1] DCE and NSC allow a simulation to use the precompiled, OS version of TCP CUBIC, as long as: 1) the host OS running ns-3 uses CUBIC, and 2) NSC and/or DCE are supported for that platform. NSC supports four real world stacks: FreeBSD, OpenBSD, lwIP and Linux. For Linux, NSC has been somewhat within DCE, and has not been ported to kernel versions newer than 2.6.26. DCE is only available for Linux.

However, in addition to the platform limitations, DCE and NSC are not the best choice for all TCP CUBIC simulation scenarios. While TCP CUBIC can be used through these setups, it is tied to the specific version of the congestion control algorithm used on a system. Since its initial introduction, CUBIC has gone through multiple updates to the algorithm meaning a simulation run on one computer will not produce the same results as a simulation run on another. Another issue is that the TCP implementation in the local operating system does not support the native ns-3 logging and trace capabilities. Without these functions, the TCP components of the simulation become a "black box" with unknown changes to cwnd and other TCP parameters over time. While not all simulations require these details, some definitely do.

To implement TCP CUBIC in ns-3, a reasonable first step is to examine the CUBIC implementation in ns-2, perhaps simply porting it into ns-3. Unfortunately, this task is far from a trivial cut and paste. ns-3 is a major redesign from ns-2 to make the structure more object-oriented, resulting in a different application interface (API) for the TCP congestion control. In ns-3, all congestion control algorithms extend from a base TCP class and need to implement specific methods. Any ns-2 code to be used in ns-3 first needs to be modified to fit into these inherited methods. In addition, the ns-2 CUBIC implementation expects state information to be passed into its methods, whereas in ns-3, the API does not provide this information to the congestion control algorithms. Other ns-2 to ns-3 porting issues impact more than just the congestion control algorithm. For example, CUBIC relies on TCP timestamps in the header of the packets to operate properly. While TCP timestamps may eventually be needed in ns-3, they are not currently available. Adding these feature into the TCP header in ns-3, which we attempted, has a ripple effect, impacting other code that relies on TCP in ns-3 and requires an extensive validation effort to ensure the additions do not break existing code.

4. CUBIC BASICS

New technologies have allowed network architects to increase network link capacities, accommodating more traffic and faster transmission rates. Unfortunately, legacy TCP algorithms like Reno and New Reno are not well suited for large capacity links. Since the congestion window in older TCP variants grows by one each RTT, it takes too long for a TCP flow to expand to meet the capacities available on some of today's Internet links. An example given by Ha et al. [5] is that traditional TCP, using 1250 byte packets on a 10 Gb/s link with a 100 ms RTT, requires 1.4 hours just to reach half the bandwidth delay product.

To more efficiently utilize the capacities of modern Internet links, a new algorithm was needed to grow the TCP congestion control window (`cwnd`) faster. This resulted in the *Binary Increase Congestion* (BIC) congestion control algorithm for TCP [8]. In TCP BIC, a binary search algorithm is used to grow cwnd from its current position to the mid-point of the cwnd position when the last loss occurred, called W_{max}. When cwnd is far away from W_{max}, cwnd grows quickly, but as cwnd approaches W_{max}, cwnd grows slowly. After passing W_{max}, cwnd continues to grow slowly for a short time before continuing a more rapid growth until the next loss event occurs. The pattern of cwnd growth when graphed over time creates a curved line with a concave region followed by convex region.

The BIC algorithm described by Ha et al. [5] for managing the concave and convex regions of cwnd growth is complicated. Since BIC depends on the RTT to change cwnd,

[1]http://www.nsnam.org/docs/models/html/tcp.html

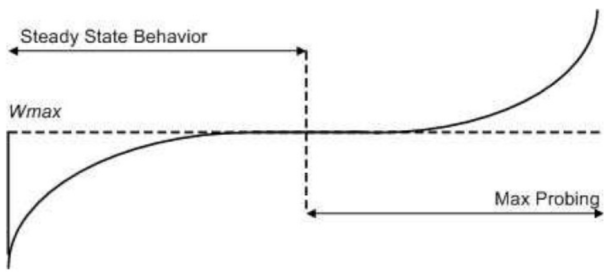


Figure 2: CUBIC growth function [5]

controlling growth is problematic when a network has a low RTT. Each time an ACK for a packet is received, BIC grows cwnd by only small increments to compensate for the short RTT. This strategy magnifies the inherent unfairness experienced by TCP flows with long RTTs.

CUBIC reduces this complexity by using a cubic function, an odd order polynomial which naturally handles concave and convex growth. Being independent of RTT, CUBIC relies instead on the time between consecutive congestion events to adjust cwnd. Hence, CUBIC maintains the same concave and convex patterns found in BIC, shown in Figure 2. The horizontal axis represents time and the vertical axis represents cwnd. The left side of the figure shows the concave region where the rate of cwnd growth slows as it approaches the position where it last lost packets, midway along the line. After flat growth for some time around this mid-point, the right side shows the convex region where cwnd grows more rapidly as TCP CUBIC probes for a new loss region.

Like BIC, CUBIC keeps track of W_{max} when a loss event occurs. To keep track of time CUBIC uses the variables *epoch_start*, K and t. *epoch_start* is the time when the first new TCP acknowledgment arrives after the loss event. At this point, cwnd is at its lowest in the curve and CUBIC needs to start growing cwnd to W_{max}. The variable K is the time when cwnd should reach W_{max}. The variable t keeps track of how long it has been since *epoch_start*. With these three variables, CUBIC tracks where the cwnd growth should be in comparison to W_{max}. When just starting cwnd growth, t is small as little time has passed since *epoch_start*. This causes large updates to cwnd. As t grows larger it starts to approach the value K and the increases to cwnd become smaller. After t grows larger than K, CUBIC changes from convex to concave where the updates to cwnd start small but increase the further t grows from K.

5. CUBIC IN LINUX

This section looks at the specific algorithms used in Linux to implement CUBIC. As Linux only increments cwnd in segment size units, an algorithm that adjusts cwnd by amounts other than one segment cannot be used. Instead, Linux CUBIC adjusts when a cwnd increment occurs. Namely, if CUBIC would normally grow cwnd by less than the segment size, Linux instead increases the amount of time before it updates cwnd. This section describes the Linux implementation of CUBIC based on version 3.11 of the Linux kernel [1].

5.1 Linux Constants and Variables

The Linux implementation of CUBIC has the same constants and variables used in the original CUBIC paper [5], but with some notable differences. For example, when CUBIC was implemented in Linux adjustments had to be made for tracking time. CUBIC uses TCP timestamps, represented in the Linux kernel as *jiffies*. (Informally, a *jiffy* is an unspecified, but short, amount of time.) The actual amount of time in 1 jiffy ranges from 1 ms to 10 ms, depending upon the computer's system clock – not consistent across platforms. Other differences in hardware architectures mean that the Linux implementation of CUBIC needs additional parameters and different values for constants than those presented in the original CUBIC paper [5]. The following is a list of the main constants and variables used in Linux and in our ns-3 implementation of CUBIC.

bic_scale – Constant scaling factor used to set the Linux version of CUBIC's C and *cube_rtt_scale*.

BICTCP_BETA_SCALE – Constant scaling factor used with the CUBIC constant β .

C – Constant used for comparing time and congestion window size in the CUBIC equations.

cube_rtt_scale – Scaling factor used in place of the CUBIC constant C when calculating recommended CUBIC update, but not when calculating the value of K.

BICTCP_HZ – Constant used to convert units of time.

epoch_start – Time of last loss event – from this point, time is tracked to find values of t and K.

cnt – Count set by CUBIC to indicate when next cwnd growth can occur.

cwnd_cnt – Global variable incremented on every ACK received. Once *cwnd_cnt* == *cnt*, cwnd is incremented and *cwnd_cnt* is reset.

β – Constant used as a multiplicative factor for decreasing cwnd during loss events.

t – Elapsed time since last loss event.

W_max – cwnd position at last loss event.

5.2 CUBIC Update

In Linux, the file `tcp_cubic.c` contains the CUBIC class. Upon instantiation, constants and starting variables are set.

Linux calls the *bictcp_update* method when an acknowledgment is received. Even though the method is called *bictcp_update*, it still refers to a CUBIC update. This method first checks to see if K has been set. In CUBIC, the growth of cwnd is controlled by a rate determined by the time since the last packet loss. K is used to represent when cwnd should return to the point it was at just before the last loss event. K is set based on Equation 1.

$$K = \sqrt[3]{C \times (W_{max} - cwnd)} \qquad (1)$$

The cube root `tcp_cubic.c` is calculated by method *cubic_root*. Code comments indicate the cubic root is found using a lookup table and an iteration of the Newton-Raphson root finding method,with an average error of about 0.195%. Along with calculating K during a loss event the time of the event is recorded to *epoch_start* using TCP timestamps.

After K is calculated, a value for t is calculated using Equation 2. As with *epoch_start*, time is calculated using TCP timestamps, which in this equation is represented by *tcp_time_stamp*. The value of t takes the current TCP timestamp plus the smallest RTT and subtracts *epoch_start*. In this equation, minRTT is the minimum RTT seen and a bit shift operation is performed three positions to the right ($>>$). The variable t is then converted so it can be compared against K. This is comprised by the bit shift to the right ($<<$) by BICTCP_HZ and then the final division by HZ.

$$t = ((tcp\_time\_stamp + (minRTT >> 3) - epoch\_start) << BICTCP\_HZ)/HZ \quad (2)$$

Next CUBIC calculates *W(t)* using either Equation 3 or Equation 4 to insure that the expression t - K is positive. Note, in these equations $>>$ indicates a bit shift to the right which scales the units of time so the result can be used with W_{max}.

$$W(t) = ((cube\_rtt\_scale \times (K - t)^3) >> (10 + 3 \times BICTCP\_HZ)) - W_{max} \quad (3)$$

$$W(t) = ((cube\_rtt\_scale \times (t - K)^3) >> (10 + 3 \times BICTCP\_HZ)) + W_{max} \quad (4)$$

The next step is to determine *cnt*, which is a count of acknowledgments that need to be received before CUBIC will increment cwnd. *cnt* is calculated by either Equation 5 or Equation 6, depending on if cwnd is less than *W(t + RTT)* or not. Further modification is made to the value of *cnt* depending on other Linux settings, but these are omitted for brevity.

$$cnt = \frac{cwnd}{W(t + RTT) - cwnd} \quad (5)$$

$$cnt = 100 \times cwnd \quad (6)$$

Since the growth of cwnd could be slower for CUBIC than other TCP congestion control mechanisms, CUBIC uses a check referred to as TCP-friendliness. In TCP-friendliness, a different value for *cnt* is calculated and the larger of the two *cnt* values is used.

Finally, the value of *cnt* is compared to *cwnd_cnt* and handled as mentioned in Section 5.1.

5.3 CUBIC Packet Loss

The method *bictcp_recalc_ssthresh* in the Linux implementation of CUBIC handles lowering cwnd and resetting ssthresh. *epoch_start* is reset to 0 since the congestion window is being lowered. If CUBIC is in fast convergence, Equation 7 is used to calculate W_{max}.

$$W_{max} = \frac{cwnd \times (BICTCP\_BETA\_SCALE + \beta)}{2 \times BICTCP\_BETA\_SCALE} \quad (7)$$

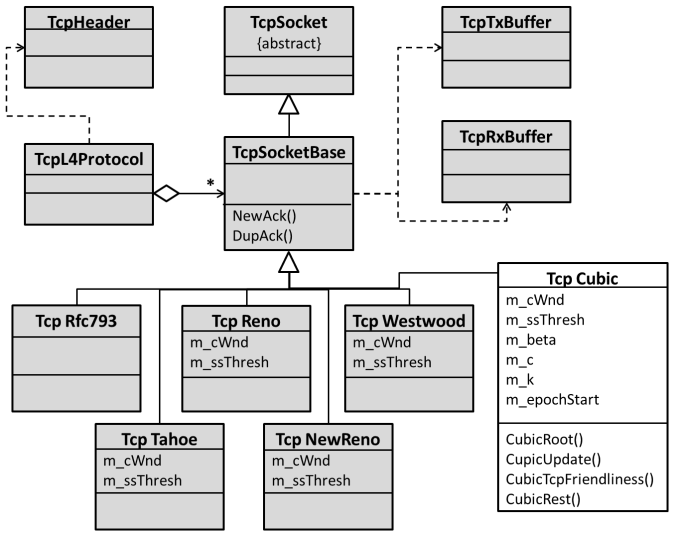


Figure 3: TCP class diagram in ns-3. Grey classes currently in ns-3 and white TcpCubic class presented in this paper. Diagram modified from [4].

However, when fast convergence is not in use, W_{max} is set to cwnd. The reduction of cwnd and ssthresh uses Equation 8 (note the max() function ensures cwnd and ssthresh are at least 2):

$$cwnd = ssthresh = max(\frac{cwnd \times \beta}{BICTCP\_BETA\_SCALE}, 2) \quad (8)$$

6. NS-3 IMPLEMENTATION

Researchers and educators studying modern computer network performance are increasingly using ns-3, an open source, discrete-event simulator. ns-3 has focused on improving the architecture and existing software modules of the widely used ns-2 simulator. Many (but not all) of the existing ns-2 modules have been ported into ns-3. While ns-3 includes support for new lower layer technologies (e.g., 4G LTE), not all of the TCP variants found in ns-2 have been ported or implemented in ns-3. The TCP variants currently supported by ns-3 exist as part of the Internet module which handles the IP layer, wired and wireless routing and the TCP and UDP transport layer protocols.

The class diagram for TCP in ns-3, which includes the CUBIC implementation presented in this paper, is shown in Figure 3. As of version 3.18, ns-3 supports the TCP congestion control algorithms of Tahoe, Reno, New Reno, Westwood and Westwood+.

6.1 TCP Cubic

Our ns-3 CUBIC implementation creates the TcpCubic class from TcpSocktBase using two files, `tcp-cubic.h` and `tcp-cubic.cc`. In TcpCubic, the methods `NewAck()` and `DupAck()` are inherited from TcpSocketBase (among others). `NewAck()` is called for each new TCP acknowledgment received and `DupAck()` is called whenever a duplicate acknowledgment is detected. `DupAck()` records the number of duplicate acknowledgments received to properly handle fast retransmit or for reducing cwnd. In addition to the inherited methods, TcpCubic has added:

CubicRoot() – Mimics the cubic_root Linux method and provides its own mechanism for finding the cubic root of a number.

CubicUpdate() – Contains most of the CUBIC related code from the Linux bictcp_update method.

CubicTcpFriendliness() – Contains code related to handling the TCP-friendly region.

CubicReset() – Resets CUBIC variables during timeouts.

6.1.1 NewAck()

`NewAck()` gets called for each new acknowledgment received. First, the current cwnd is compared to ssthresh. If (cwnd <= ssthresh) then TCP is in slow start and one segment size is added to cwnd. Otherwise, the method `CubicUpdate()` is called to get the value of *cnt*. As described in Section 5.2, cnt is compared to cwnd_cnt to determine if cwnd is incremented or not. `NewAck()` is then called by TcpSocketBase to return execution to the rest of the ns-3 TCP implementation.

6.1.2 CubicUpdate()

We implemented `CubicUpdate()` to be similar to the Linux method bictcp_update, described in Section 5.2. In Linux, t is calculated using TCP timestamps, but ns-3 does not implement TCP timestamps. Instead we implemented a standin for TCP timestamps as described in Section 6.2. More minor changes from the Linux implementation includes further dividing some code into separate methods for readability.

6.1.3 CubicTcpFriendliness()

The code in `CubicTcpFriendliness()` is the same as the code used in the Linux implementation. The details are omitted for brevity, but basically if the growth suggested by TCP is greater than that suggested by CUBIC, then the TCP friendliness growth is used for cnt.

6.1.4 DupAck()

`DupAck()` is called for every duplicate acknowledgment received, with parameters passed into `DupAck()` indicating the number of duplicates. If three duplicate acknowledgments have been received and if TCP is not in fast recovery, our Linux implementation of CUBIC lowers cwnd and ssthresh (Equation 8) and sets W_{max}.

If three duplicate acknowledgments have not been received, He et al. [5] do not indicate CUBIC's response. Our implementation assumes that CUBIC would act in a manner similar to that of New Reno which includes increasing cwnd by one segment while in fast recovery.

6.2 TCP Timestamps

Ideally TCP timestamps should be fully implemented and included in the TCP header. However, this would have required more coding and changes to the base TCP classes that would impact all other TCP related code. We chose instead to create a local method for finding timestamps to get a consistent count of time to track when loss events happen and when cwnd should grow to W_{max}. In ns-2, timestamps are set in the class `tcp-linux.cc` and are generated by looking at the clock from the ns-2 scheduler, which returns time in seconds, multiplies this by the constant JIFFY_RATIO (in the example simulations, this was 1000) and then truncated for the result. For our implementation, we followed this same pattern by getting the simulator clock in seconds, multiplying by 1000 and then truncating. Our timestamps are intended to represent the same consistent gap, but small errors may exist due to not having a full implementation of timestamps. Future work would take the code for generating the timestamps and move it so that the timestamps are generated by the sender and added to the TCP header.

6.3 Issues

Challenges to overcome during the ns-3 CUBIC implementation can be categorized into unavailable implementations and unit conversions. Unavailable implementations refers to Linux code that was not available for use in ns-3. For example, the Linux cubic_root algorithm relies on the method `fls64()` to find the last set bit of a 64-bit number. This method is defined in the Linux kernel and is not part of the normal C/C++ libraries – instead an implementation of this method was copied from the ns-2 TCP package.

Other Linux code and constants could not be definitively found. For example, code from [1] has multiple implementations for some methods in different files, with constants having differing definitions. Some of these differences stem from differences in CPU architectures, where processor clock cycles require specific values for the scaling and conversion constants used with the variable t.

CUBIC uses timestamps to track time between packet losses and growth to W_{max}. As mentioned in Section 6.2, added code keeps consistent track of time but this code needs to be incorporated into the base classes of TCP.

7. VALIDATION

This section verifies and validates the TCP CUBIC implementation in ns-3. First, Section 7.1 compares the performance of our implementation to that of the CUBIC implementation in Linux 3.0.0-26-generic on the Ubuntu 11.10 operating system through the use of NSC. Section 7.2 compares our implementation to the CUBIC implementation in ns-2.34. Finally, Section 7.3 examines how our CUBIC implementation compares to the default congestion control algorithm in ns-3, NewReno.

7.1 NSC Comparison

Our first validation step is to compare our CUBIC implementation against a CUBIC implementation used by the underlining operating system through the use of NSC. As mentioned in Section 3, the detailed logging capabilities and parameter adjustments available in ns-3 are not available when using the operating systems' TCP stack. However, we can compare the overall performance of both simulations. The ns-3 module Flow Monitor, which tracks the bytes/packets sent and received as well as dropped packets, can be used to compare data rates between an operating system CUBIC and our ns-3 CUBIC.

Our simulation had one node transmitting to a receiver over a 5 Mbps link with a delay of 40 ms and an error rate of 0.000001. The error rate generated a small number of loss events (in this simulation, exactly 10) so we could see

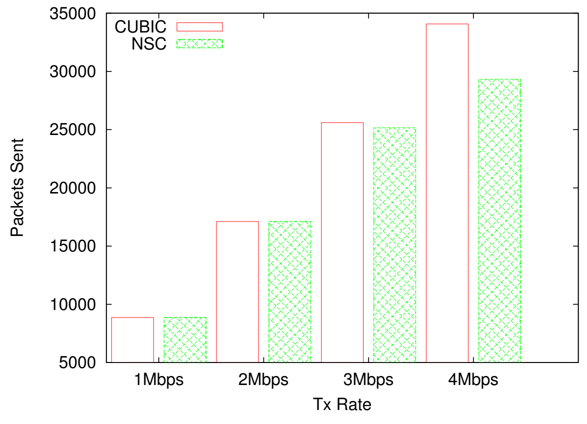


Figure 4: Comparison of our CUBIC model and the operating system's CUBIC accessed by NSC.

the differences in TCP response between loss and recovery clearly. The application, from the ns-3 documentation, sent packets of a specific size at a specific data rate. The data rate was set to 1 Mbps, in order to keep the transmissions lower than the capacity of the pipe. While packet size can be controlled in ns-3 with the maximum TCP segment size, there is no direct control over this in the operating system. To determine the maximum segment size and bytes sent per packet, we first ran the application using NSC, recorded the number of bytes and packets transmitted by the Flow Monitor, and computed the maximum packet sized used. The underline operating system used a maximum segment size of 1410 bytes with headers totaling 1450 bytes. Hence, the ns-3 TCP CUBIC simulation also used a maximum segment size of 1410 bytes.

After testing, results show for a 100 second simulation, NSC CUBIC transmits 8864 packets with 12,849,224 total bytes. Our CUBIC transmits 8865 packets with 12,848,610 total bytes, just one more packet than the operating systems' CUBIC implementation and 614 fewer bytes. Both simulations lost 10 packets due to the error rate. The receiving statistics are similar to the transmission, with NSC CUBIC receiving 8851 packets and 12,829,724 bytes and our CUBIC receiving 8852 packets and 12,829,760 bytes.

Further tests were run with higher transmission rates, depicted in Figure 4. The x-axis is the transmission rate and the y-axis is the number of packets sent. The opaque green bar shows NSC CUBIC while the clear red bar shows our ns-3 CUBIC. At higher transmission rates, ns-3 CUBIC transmits more data than NSC CUBIC. Unfortunately, since there are no logging mechanisms with NSC to understand what the TCP CUBIC protocol there is doing, we have no way to better understand the differences in CUBIC behaviors. A possible difference is in the use of TCP timestamps. When setting up NSC, we specified that the CUBIC implementation should not use several TCP features not supported by ns-3, including TCP timestamps. However, our ns-3 CUBIC uses our own method of calculating time and these seemingly small differences may results in greater differences as the simulation time runs longer or the transmission rates increase.

Table 1: ns-2/ns-3 high level comparison.

	ns-2	ns-3
Packets Sent	7,159	6,979
Packets Received	7,146	6,972
Packets Lost	13	7
Bytes Sent	7,515,940	7,255,160
Bytes Received	7,502,290	7,247,880

Table 2: ns-2/ns-3 low level comparison.

	ns-2	ns-3
K	3,640 - 6,575	4,116 - 5,344
t	678 - 17,891	38 - 13,079
cnt	1 - 4,748	4 - 13,300

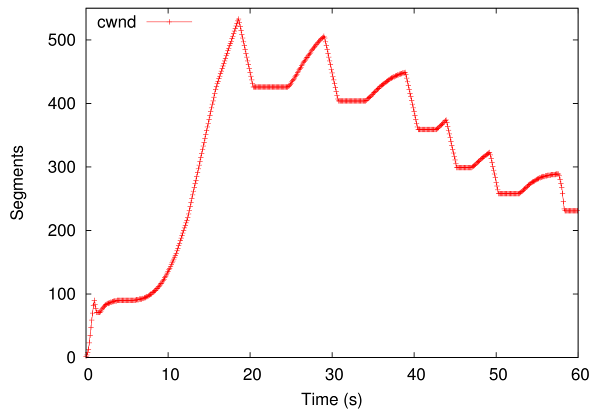
7.2 NS-2 Comparison

We created simulations that closely mirror one another to compare the general performance for ns-2 CUBIC and our ns-3 CUBIC. Both simulations had a network with two nodes (sender and receiver) and a link with a 1 Mbps capacity, a 40ms delay and an error rate. The error rate was chosen so that for both simulators, there were a few packet drops during the simulation, allowing us to analyze the CUBIC congestion window curves. Both ns-2 and ns-3 used a maximum TCP segment size of 1000 bytes. For applications, ns-2 used an FTP example while ns-3 used the bulk sending application. Graphs of cwnd for these simulations are shown in Figure 5.

Similar to the test with NSC CUBIC when limiting the maximum link bandwidth to 1 Mbps, our ns-3 CUBIC model performs similarly to the ns-2 implementation, as shown in Table 1. ns-2 sent 180 more packets than ns-3. Some of this difference is likely due to the retransmissions for the 6 extra packets lost in the ns-2 simulation compared to ns-3. ns-2 sent 260,780 more bytes than ns-3. Part of this comes from the extra packets sent, and examining the trace shows ns-2 sending an additional 10 bytes per TCP data packet, a result of the additional header data on the packets used in ns-2.

Next, we analyze how ns-3 CUBIC calculates the CUBIC variables K, t and cnt. Since the two simulation scenarios are not identical there is no direct one to one comparison. Instead, the goal is to see if under similar inputs ns-3 CUBIC produces results similar to ns-2 CUBIC. The range of values seen for these variables in ns-2 and ns-3 are shown in Table 2. For K, both ns-2 and ns-3 are returning 4 digit numbers. From Equation 1, our ns-3 CUBIC uses a copied version of the cubed root formula and the same constant C where the only variables in this equation are W_{max} and cwnd. Since the difference in the ns-2 and ns-3 simulation cwnds are not off by more than a few hundred segments, our K values do not diverge beyond 4 digit numbers.

Next, a comparison of the variable t from Equation 2 shows other differences. Here, the constants *BICTCP_HZ* and *HZ* are the same in both ns-2 and ns-3. The custom implementation of TCP timestamps differs, which is likely to cause differences in behavior. In our ns-3 CUBIC, t ranges from 2 to 5 digits while in ns-2, the values for t range from 3 to 5 digits, although other ns-2 simulations showed that t could be 2 digits, too. The ACK counts (cnt) required to increase cwnd range from 1 to 4 digits in ns-2 and 1 to 5



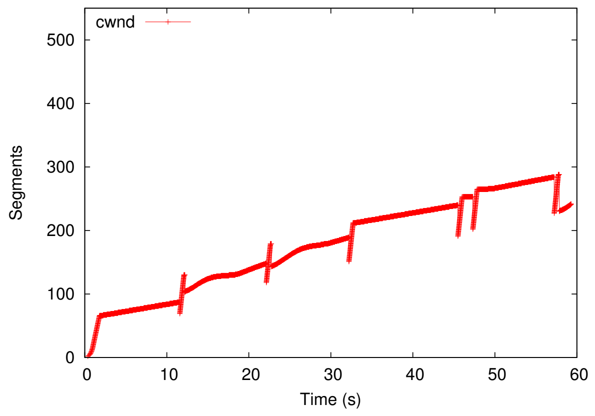


Figure 5: ns-2 (left) and ns-3 (right) simulations using CUBIC.

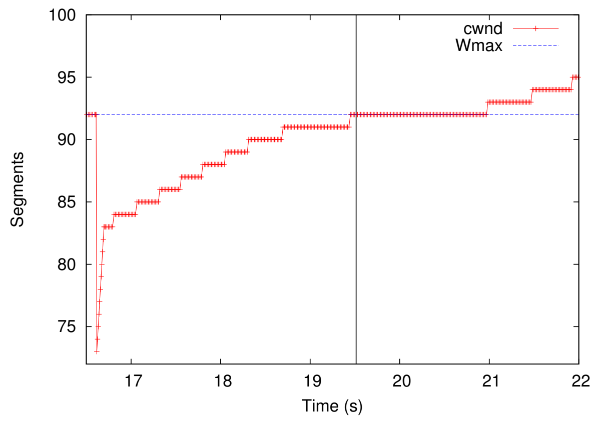


Figure 6: ns-3 CUBIC simulation showing when cwnd growth changes from convex to concave, depicted by the solid black vertical line.

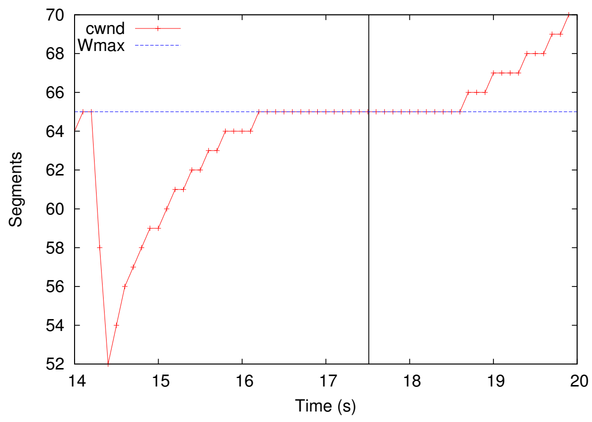


Figure 7: ns-2 CUBIC simulation showing when cwnd growth changes from convex to concave, depicted by the solid black vertical line.

digits in ns-3.

7.2.1 Packet Loss

In order to compare specific parts of our ns-3 CUBIC to ns-2 CUBIC, we edited the ns-2 `tcp-cubic.c` file and add logging statements.

During a loss event, CUBIC decreases cwnd as shown in Equation 8 and sets ssthresh to cwnd. This equation has two constants, $\beta$and BICTCP_BETA_SCALE. In both ns-2 and ns-3, the constants are set to the same value, 819 and 1024 respectively. The only variable is the size of cwnd before the loss event. During the ns-2 simulation, there was one loss event when cwnd was equal to 90 and after running through Equation 8 it became 71. The ns-3 simulation has the same equation and in one instance there is a cwnd of size 92, and after Equation 8 became 73. The standard handling of loss events appears the same for both the ns-2 and ns-3 implementations.

7.2.2 CUBIC Update

An key aspect of CUBIC is cwnd growth being convex up to the point where it expects packet loss and then flipping to concave when cwnd grows past W_{max} with no loss. An example of this for our CUBIC implementation (a different simulation from Figure 5) is shown in Figure 6. In this graph, the x-axis is time and the y-axis is the number of TCP segments. The blue line shows the value of W_{max}, the red line indicates the value of cwnd and the black line marks when CUBIC changes from concave to convex. This figure shows that in CUBIC cwnd reaches W_{max} about the same time the CUBIC variable t reaches the variable K, as expected.

For comparison, Figure 7 is an example of ns-2 CUBIC cwnd growth (a different simulation from Figure 5). The axes and trendlines are the same as in Figure 6. In the ns-2 simulation, t does not pass K until a short time after cwnd reaches W_{max}. Possible discrepancies for this are the custom TCP timestamps created for our ns-3 CUBIC model.

7.3 NewReno Comparison

TCP CUBIC's method of using W_{max} as a target to rapidly increase cwnd is intended to provide a better congestion avoidance mechanism in many circumstances when compared against the older TCP NewReno method. This section compares simulations using both the standard NewReno (currently the ns-3 default) and our ns-3 CUBIC, as seen in

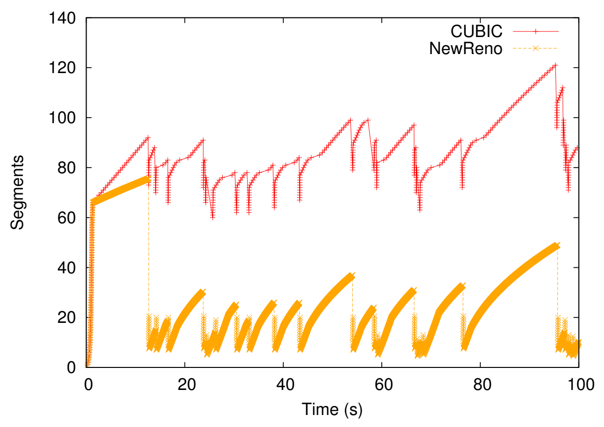


Figure 8: ns-3 Simulation comparing CUBIC and NewReno.

Figure 8. The x-axis is time and the y-axis is number of TCP segments, with the yellow 'x' trendline depicting NewReno and the red '+' trendline depicting CUBIC. When the simulations start, both NewReno and CUBIC behave similarly in slow start where they both double cwnd every RTT. However, once the two algorithms pass the initial sshtresh value and enter congestion avoidance, differences emerge. CUBIC starts to probe for the first W_{max} value while NewReno uses Equation 9 to increment cwnd as specified in RFC 2581 [3]. Note, SMSS is the sender's maximum segment size.

$$cwnd+ = SMSS \times SMSS/cwnd \tag{9}$$

At the first loss event, CUBIC sets the value of W_{max} and reduces cwnd as described in Section 7.2.1. On the other hand, NewReno uses Equation 10, as given in RFC 2581 [3], to set ssthresh and modify cwnd to be 3 segments larger than ssthresh. Consequently, CUBIC does not lower cwnd as much as does NewReno during loss events.

After exiting fast recovery, CUBIC displays its normal concave/convex pattern (i.e., concave approaching W_{max}, then convex when passing). CUBIC updates cwnd less than NewReno. However, once passing W_{max} CUBIC starts to increment cwnd much more rapidly than NewReno. The multiple loss events seen in Figure 8 keep cwnd low in NewReno while the CUBIC cwnd stays much higher. Hence, TCP CUBIC makes use of more of the channel capacity than does TCP NewReno.

$$ssthresh = max(BytesInFlight/2, 2 \times SMSS) \tag{10}$$

8. CONCLUSIONS

The current ns-3 TCP implementations lack the newer TCP congestion control algorithms used by most computers today [9]. For realistic network simulations that more closely model Internet behavior, additional implementations of TCP congestion control algorithms are needed in ns-3.

This paper introduces a TCP CUBIC implementation for ns-3,[2] based on the original CUBIC algorithm [5] while using many of the implementation details found in Linux [1]. Evaluation of our ns-3 CUBIC verifies the implementation, and comparisons with Linux measurements provide validation. In particular, the ns-3 implementation produces a CUBIC-like cwnd growth pattern which is critical for modern TCP implementations to effectivley utilize available network capacities. The comparison of the simulated performance with TCP NewReno implies a significant bitrate advantage gained by employing TCP CUBIC.

Existing issues with our ns-3 CUBIC include lack of support for TCP timestamps in the base TCP implementation of ns-3, making it unable to exactly match CUBIC implementation in the Linux kernel or ns-2. Future work includes implementing TCP timestamps in ns-3 and running more validation tests involving multiple flows and different network conditions, such as low and high congestion as well as a wide range of capacities and RTTs, that stress TCP functionality. Additionally, native implementations of other TCP protocols, such as TCP BIC and TCP Compound, with corresponding verification and validation, could be undertaken.

[2]Available for download at http://perform.wpi.edu/downloads/#cubic

9. ACKNOWLEDGMENTS

This work is sponsored by the Department of the Air Force under Air Force Contract FA8721-05-C-0002. Opinions, interpretations, conclusions and recommendations are those of the author and not necessarily endorsed by the United States Government.

10. REFERENCES

[1] CUBIC Implementation in Linux Kernel Version 3.11. `http://lxr.free-electrons.com/source/net/ipv4/tcp_cubic.c?v=3.11` [Accessed 06/19/2014].

[2] ns-3 network simulator. `http://www.nsnam.org/` [Acessed 06/19/2014].

[3] M. Allman, V. Paxson, and W. Stevens. TCP Congestion Control, April 1999. RFC2581.

[4] S. Gangadhar, T. A. N. Nguyen, G. Umapathi, and J. P. G. Sterbenz. TCP Westwood(+) Protocol Implementation in ns-3. In *Proceedings of ICST Conference on Simulation Tools and Techniques*, SimuTools, pages 167–175, ICST, Brussels, Belgium, 2013.

[5] S. Ha, I. Rhee, and L. Xu. CUBIC: A New TCP-Friendly High-Speed TCP Variant. *SIGOPS Oper. Syst. Rev.*, 42(5):64–74, July 2008.

[6] L. C. Kho, X. Defago, A. Lim, and Y. Tan. A taxonomy of congestion control techniques for tcp in wired and wireless networks. In *Proceedings of Wireless Technology and Applications (ISWTA)*, Sept 2013.

[7] K. Tan, J. Song, Q. Zhang, and M. Sridharan. A Compound TCP Approach for High-Speed and Long Distance Networks. In *Proceedings of INFOCOM*, 2006.

[8] L. Xu, K. Harfoush, and I. Rhee. Binary Increase Congestion Control (BIC) for Fast Long-Distance Networks. In *Proceedings of INFOCOM*, volume 4, pages 2514–2524 vol.4, 2004.

[9] P. Yang, J. Shao, W. Luo, L. Xu, J. Deogun, and Y. Lu. TCP Congestion Avoidance Algorithm Identification. *IEEE/ACM Transactions on Networking*, PP(99):1–1, 2013.