Internet Engineering Task Force (IETF)
Request for Comments: 9438
Obsoletes: 8312
Updates: 5681
Category: Standards Track
ISSN: 2070-1721

L. Xu
UNL
S. Ha
Colorado
I. Rhee
Bowery
V. Goel
Apple Inc.
L. Eggert, Ed.
NetApp

August 2023

CUBIC for Fast and Long-Distance Networks

Abstract

CUBIC is a standard TCP congestion control algorithm that uses a cubic function instead of a linear congestion window increase function to improve scalability and stability over fast and long-distance networks. CUBIC has been adopted as the default TCP congestion control algorithm by the Linux, Windows, and Apple stacks.

This document updates the specification of CUBIC to include algorithmic improvements based on these implementations and recent academic work. Based on the extensive deployment experience with CUBIC, this document also moves the specification to the Standards Track and obsoletes RFC 8312. This document also updates RFC 5681, to allow for CUBIC's occasionally more aggressive sending behavior.

Status of This Memo

This is an Internet Standards Track document.

This document is a product of the Internet Engineering Task Force (IETF). It represents the consensus of the IETF community. It has received public review and has been approved for publication by the Internet Engineering Steering Group (IESG). Further information on Internet Standards is available in Section 2 of RFC 7841.

Information about the current status of this document, any errata, and how to provide feedback on it may be obtained at https://www.rfc-editor.org/info/rfc9438.

Table of Contents

1. Introduction

CUBIC has been adopted as the default TCP congestion control algorithm in the Linux, Windows, and Apple stacks, and has been used and deployed globally. Extensive, decade-long deployment experience in vastly different Internet scenarios has convincingly demonstrated that CUBIC is safe for deployment on the global Internet and delivers substantial benefits over classical Reno congestion control [RFC5681]. It is therefore to be regarded as the currently most widely deployed standard for TCP congestion control. CUBIC can also be used for other transport protocols such as QUIC [RFC9000] and the Stream Control Transmission Protocol (SCTP) [RFC9260] as a default congestion controller.

The design of CUBIC was motivated by the well-documented problem classical Reno TCP has with low utilization over fast and long-distance networks [K03] [RFC3649]. This problem arises from a slow increase of the congestion window (cwnd) following a congestion event in a network with a large bandwidth-delay product (BDP). [HLRX07] indicates that this problem is frequently observed even in the range of congestion window sizes over several hundreds of packets. This problem is equally applicable to all Reno-style standards and their variants, including TCP-Reno [RFC5681], TCP-NewReno [RFC6582]

[RFC6675], SCTP [RFC9260], TCP Friendly Rate Control (TFRC) [RFC5348], and QUIC congestion control [RFC9002], which use the same linear increase function for window growth. All Reno-style standards and their variants are collectively referred to as "Reno" in this document.

CUBIC, originally proposed in [HRX08], is a modification to the congestion control algorithm of classical Reno to remedy this problem. Specifically, CUBIC uses a cubic function instead of the linear window increase function of Reno to improve scalability and stability under fast and long-distance networks.

This document updates the specification of CUBIC to include algorithmic improvements based on the Linux, Windows, and Apple implementations and recent academic work. Based on the extensive deployment experience with CUBIC, it also moves the specification to the Standards Track, obsoleting [RFC8312]. This requires an update to Section 3 of [RFC5681], which limits the aggressiveness of Reno TCP implementations. Since CUBIC is occasionally more aggressive than the algorithms defined in [RFC5681], this document updates the first paragraph of Section 3 of [RFC5681], replacing it with a normative reference to guideline (1) in Section 3 of [RFC5033], which allows for CUBIC's behavior as defined in this document.

Specifically, CUBIC may increase the congestion window more aggressively than Reno during the congestion avoidance phase. According to [RFC5681], during congestion avoidance, the sender must not increment cwnd by more than Sender Maximum Segment Size (SMSS) bytes once per round-trip time (RTT), whereas CUBIC may increase cwnd much more aggressively. Additionally, CUBIC recommends the HyStart++ algorithm [RFC9406] for slow start, which allows for cwnd increases of more than SMSS bytes for incoming acknowledgments during slow start, while this behavior is not allowed as part of the standard behavior prescribed by [RFC5681].

Binary Increase Congestion Control (BIC-TCP) [XHR04], a predecessor of CUBIC, was selected as the default TCP congestion control algorithm by Linux in the year 2005 and had been used for several years by the Internet community at large.

CUBIC uses a window increase function similar to BIC-TCP and is designed to be less aggressive and fairer to Reno in bandwidth usage than BIC-TCP while maintaining the strengths of BIC-TCP such as stability, window scalability, and RTT-fairness.

[RFC5033] documents the IETF's best current practices for specifying new congestion control algorithms, specifically those that differ from the general congestion control principles outlined in [RFC2914]. It describes what type of evaluation is expected by the IETF to understand the suitability of a new congestion control algorithm and the process of enabling a specification to be approved for widespread deployment in the global Internet.

There are areas in which CUBIC differs from the congestion control algorithms previously published in Standards Track RFCs; those changes are specified in this document. However, it is not obvious that these changes go beyond the general congestion control principles outlined in [RFC2914], so the process documented in [RFC5033] may not apply.

Also, the wide deployment of CUBIC on the Internet was driven by direct adoption in most of the popular operating systems and did not follow the practices documented in [RFC5033]. However, due to the resulting Internet-scale deployment experience over a long period of time, the IETF determined that CUBIC could be published as a Standards Track specification. This decision by the IETF does not

alter the general guidance provided in [RFC2914].

The following sections

1. briefly explain the design principles of CUBIC,

2. provide the exact specification of CUBIC, and

3. discuss the safety features of CUBIC, following the guidelines specified in [RFC5033].

2. Conventions

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "NOT RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in BCP 14 [RFC2119] [RFC8174] when, and only when, they appear in all capitals, as shown here.

3. Design Principles of CUBIC

CUBIC is designed according to the following design principles:

Principle 1: For better network utilization and stability, CUBIC uses both the concave and convex profiles of a cubic function to increase the congestion window size, instead of using just a convex function.

Principle 2: To be Reno-friendly, CUBIC is designed to behave like Reno in networks with short RTTs and small bandwidth where Reno performs well.

Principle 3: For RTT-fairness, CUBIC is designed to achieve linear bandwidth sharing among flows with different RTTs.

Principle 4: CUBIC appropriately sets its multiplicative window decrease factor in order to achieve a balance between scalability and convergence speed.

3.1. Principle 1 for the CUBIC Increase Function

For better network utilization and stability, CUBIC [HRX08] uses a cubic window increase function in terms of the elapsed time from the last congestion event. While most congestion control algorithms that provide alternatives to Reno increase the congestion window using convex functions, CUBIC uses both the concave and convex profiles of a cubic function for window growth.

After a window reduction in response to a congestion event detected by duplicate acknowledgments (ACKs), Explicit Congestion Notification-Echo (ECN-Echo (ECE)) ACKs [RFC3168], RACK-TLP for TCP [RFC8985], or QUIC loss detection [RFC9002], CUBIC remembers the congestion window size at which it received the congestion event and performs a multiplicative decrease of the congestion window. When CUBIC enters into congestion avoidance, it starts to increase the congestion window using the concave profile of the cubic function. The cubic function is set to have its plateau at the remembered congestion window size, so that the concave window increase continues until then. After that, the cubic function turns into a convex profile and the convex window increase begins.

This style of window adjustment (concave and then convex) improves algorithm stability while maintaining high network utilization [CEHRX09]. This is because the window size remains almost constant, forming a plateau around the remembered congestion window size of the last congestion event, where network utilization is deemed highest.

Under steady state, most window size samples of CUBIC are close to that remembered congestion window size, thus promoting high network utilization and stability.

Note that congestion control algorithms that only use convex functions to increase the congestion window size have their maximum increments around the remembered congestion window size of the last congestion event and thus introduce many packet bursts around the saturation point of the network, likely causing frequent global loss synchronizations.

3.2. Principle 2 for Reno-Friendliness

CUBIC promotes per-flow fairness to Reno. Note that Reno performs well over paths with small BDPs and only experiences problems when attempting to increase bandwidth utilization on paths with large BDPs.

A congestion control algorithm designed to be friendly to Reno on a per-flow basis must increase its congestion window less aggressively in small-BDP networks than in large-BDP networks.

The aggressiveness of CUBIC mainly depends on the maximum window size before a window reduction, which is smaller in small-BDP networks than in large-BDP networks. Thus, CUBIC increases its congestion window less aggressively in small-BDP networks than in large-BDP networks.

Furthermore, in cases when the cubic function of CUBIC would increase the congestion window less aggressively than Reno, CUBIC simply follows the window size of Reno to ensure that CUBIC achieves at least the same throughput as Reno in small-BDP networks. The region where CUBIC behaves like Reno is called the "Reno-friendly region".

3.3. Principle 3 for RTT-Fairness

Two CUBIC flows with different RTTs have a throughput ratio that is linearly proportional to the inverse of their RTT ratio, where the throughput of a flow is approximately the size of its congestion window divided by its RTT.

Specifically, CUBIC maintains a window increase rate that is independent of RTTs outside the Reno-friendly region, and thus flows with different RTTs have similar congestion window sizes under steady state when they operate outside the Reno-friendly region.

This notion of a linear throughput ratio is similar to that of Reno under an asynchronous loss model, where flows with different RTTs have the same packet loss rate but experience loss events at different times. However, under a synchronous loss model, where flows with different RTTs experience loss events at the same time but have different packet loss rates, the throughput ratio of Reno flows with different RTTs is quadratically proportional to the inverse of their RTT ratio [XHR04].

CUBIC always ensures a linear throughput ratio that is independent of the loss environment. This is an improvement over Reno. While there is no consensus on the optimal throughput ratio for different RTT flows, over wired Internet paths, use of a linear throughput ratio seems more reasonable than equal throughputs (i.e., the same throughput for flows with different RTTs) or a higher-order throughput ratio (e.g., a quadratic throughput ratio of Reno in synchronous loss environments).

3.4. Principle 4 for the CUBIC Decrease Factor

To achieve a balance between scalability and convergence speed, CUBIC sets the multiplicative window decrease factor to 0.7, whereas Reno uses 0.5.

While this improves the scalability of CUBIC, a side effect of this decision is slower convergence, especially under low statistical multiplexing. This design choice is following the observation that HighSpeed TCP (HSTCP) [RFC3649] and other approaches (e.g., [GV02]) made: the current Internet becomes more asynchronous with less frequent loss synchronizations under high statistical multiplexing.

In such environments, even strict Multiplicative-Increase Multiplicative-Decrease (MIMD) can converge. CUBIC flows with the same RTT always converge to the same throughput independently of statistical multiplexing, thus achieving intra-algorithm fairness. In environments with sufficient statistical multiplexing, the convergence speed of CUBIC is reasonable.

4. CUBIC Congestion Control

This section discusses how the congestion window is updated during the different stages of the CUBIC congestion controller.

4.1. Definitions

The unit of all window sizes in this document is segments of the SMSS, and the unit of all times is seconds. Implementations can use bytes to express window sizes, which would require factoring in the SMSS wherever necessary and replacing _segments_acked_ (Figure 4) with the number of acknowledged bytes.

4.1.1. Constants of Interest

* β __cubic_: CUBIC multiplicative decrease factor as described in Section 4.6.

* α __cubic_: CUBIC additive increase factor used in the Reno-friendly region as described in Section 4.3.

* _C_: Constant that determines the aggressiveness of CUBIC in competing with other congestion control algorithms in high-BDP networks. Please see Section 5 for more explanation on how it is set. The unit for _C_ is

$$\frac{\text{segment}}{\text{second}^3}$$

4.1.2. Variables of Interest

This section defines the variables required to implement CUBIC:

* _RTT_: Smoothed round-trip time in seconds, calculated as described in [RFC6298].

* _cwnd_: Current congestion window in segments.

* _ssthresh_: Current slow start threshold in segments.

* _cwnd_prior_: Size of _cwnd_ in segments at the time of setting _ssthresh_ most recently, either upon exiting the first slow start or just before _cwnd_ was reduced in the last congestion event.

* _W_max_: Size of _cwnd_ in segments just before _cwnd_ was reduced in the last congestion event when fast convergence is disabled

(same as _cwnd_prior_ on a congestion event). However, if fast convergence is enabled, _W_max_ may be further reduced based on the current saturation point.

* _K_: The time period in seconds it takes to increase the congestion window size at the beginning of the current congestion avoidance stage to _W_max_.

* _t_current_: Current time of the system in seconds.

* _t_epoch_: The time in seconds at which the current congestion avoidance stage started.

* _cwnd_epoch_: The _cwnd_ at the beginning of the current congestion avoidance stage, i.e., at time _t_epoch_.

* W_cubic(_t_): The congestion window in segments at time _t_ in seconds based on the cubic increase function, as described in Section 4.2.

* _target_: Target value of the congestion window in segments after the next RTT -- that is, W_cubic(_t_ + _RTT_), as described in Section 4.2.

* _W_est_: An estimate for the congestion window in segments in the Reno-friendly region -- that is, an estimate for the congestion window of Reno.

* _segments_acked_: Number of SMSS-sized segments acked when a "new ACK" is received, i.e., an ACK that cumulatively acknowledges the delivery of previously unacknowledged data. This number will be a decimal value when a new ACK acknowledges an amount of data that is not SMSS-sized. Specifically, it can be less than 1 when a new ACK acknowledges a segment smaller than the SMSS.

4.2. Window Increase Function

CUBIC maintains the ACK clocking of Reno by increasing the congestion window only at the reception of a new ACK. It does not make any changes to the TCP Fast Recovery and Fast Retransmit algorithms [RFC6582] [RFC6675].

During congestion avoidance, after a congestion event is detected as described in Section 3.1, CUBIC uses a window increase function different from Reno.

CUBIC uses the following window increase function:

$$W_{cubic}(t) = C * (t - K)^3 + W_{max}$$

Figure 1

where _t_ is the elapsed time in seconds from the beginning of the current congestion avoidance stage -- that is,

$$t = t_{current} - t_{epoch}$$

and where _t_epoch_ is the time at which the current congestion avoidance stage starts. _K_ is the time period that the above function takes to increase the congestion window size at the beginning of the current congestion avoidance stage to _W_max_ if there are no further congestion events. _K_ is calculated using the following equation:

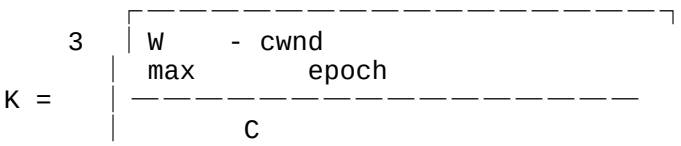


Figure 2

where _cwnd_epoch_ is the congestion window at the beginning of the current congestion avoidance stage.

Upon receiving a new ACK during congestion avoidance, CUBIC computes the _target_ congestion window size after the next _RTT_ using Figure 1 as follows, where _RTT_ is the smoothed round-trip time. The lower and upper bounds below ensure that CUBIC's congestion window increase rate is non-decreasing and is less than the increase rate of slow start [SXEZ19].

```
             cwnd            if  W     (t + RTT) < cwnd
                                  cubic
             1.5 * cwnd      if  W     (t + RTT) > 1.5 * cwnd
   target =                       cubic
             W     (t + RTT) otherwise
              cubic
```

The elapsed time _t_ in Figure 1 MUST NOT include periods during which _cwnd_ has not been updated due to application-limited behavior (see Section 5.8).

Depending on the value of the current congestion window size _cwnd_, CUBIC runs in three different regions:

1. The Reno-friendly region, which ensures that CUBIC achieves at least the same throughput as Reno.

2. The concave region, if CUBIC is not in the Reno-friendly region and _cwnd_ is less than _W_max_.

3. The convex region, if CUBIC is not in the Reno-friendly region and _cwnd_ is greater than _W_max_.

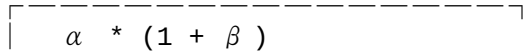
To summarize, CUBIC computes both W_cubic(_t_) and _W_est_ (see Section 4.3) on receiving a new ACK in congestion avoidance and chooses the larger of the two values.

The next sections describe the exact actions taken by CUBIC in each region.

4.3. Reno-Friendly Region

Reno performs well in certain types of networks -- for example, under short RTTs and small bandwidths (or small BDPs). In these networks, CUBIC remains in the Reno-friendly region to achieve at least the same throughput as Reno.

The Reno-friendly region is designed according to the analysis discussed in [FHP00], which studies the performance of an AIMD algorithm with an additive factor of α (segments per _RTT_) and a multiplicative factor of β, denoted by AIMD(α, β). _p_ is the packet loss rate. Specifically, the average congestion window size of AIMD(α, β) can be calculated using Figure 3.



$$\text{AVG\_AIMD}(\alpha, \beta) = \sqrt{\frac{\alpha * (1 + \beta)}{2 * (1 - \beta) * p}}$$

Figure 3

By the same analysis, to achieve an average window size similar to Reno that uses AIMD(1, 0.5), α must be equal to

$$3 * \frac{1 - \beta}{1 + \beta}$$

Thus, CUBIC uses Figure 4 to estimate the window size _W_est_ in the Reno-friendly region with

$$\alpha_{cubic} = 3 * \frac{1 - \beta_{cubic}}{1 + \beta_{cubic}}$$

which achieves approximately the same average window size as Reno in many cases. The model used to calculate α __cubic_ is not absolutely precise, but analysis and simulation as discussed in [AIMD-friendliness], as well as over a decade of experience with CUBIC in the public Internet, show that this approach produces acceptable levels of rate fairness between CUBIC and Reno flows. Also, no significant drawbacks of the model have been reported. However, continued detailed analysis of this approach would be beneficial. When receiving a new ACK in congestion avoidance (where _cwnd_ could be greater than or less than _W_max_), CUBIC checks whether W_cubic(_t_) is less than _W_est_. If so, CUBIC is in the Reno-friendly region and _cwnd_ SHOULD be set to _W_est_ at each reception of a new ACK.

_W_est_ is set equal to _cwnd_epoch_ at the start of the congestion avoidance stage. After that, on every new ACK, _W_est_ is updated using Figure 4. Note that this equation uses _segments_acked_ and _cwnd_ is measured in segments. An implementation that measures _cwnd_ in bytes should adjust the equation accordingly using the number of acknowledged bytes and the SMSS. Also note that this equation works for connections with enabled or disabled delayed ACKs [RFC5681], as _segments_acked_ will be different based on the segments actually acknowledged by a new ACK.

$$W_{est} = W_{est} + \alpha_{cubic} * \frac{segments\_acked}{cwnd}$$

Figure 4

Once _W_est_ has grown to reach the _cwnd_ at the time of most recently setting _ssthresh_ -- that is, _W_est_ >= _cwnd_prior_ -- the sender SHOULD set α __cubic_ to 1 to ensure that it can achieve the same congestion window increment rate as Reno, which uses AIMD(1, 0.5).

The next two sections assume that CUBIC is not in the Reno-friendly region and uses the window increase function described in Section 4.2. Although _cwnd_ is incremented in the same way for both concave and convex regions, they are discussed separately to analyze and understand the difference between the two regions.

4.4. Concave Region

When receiving a new ACK in congestion avoidance, if CUBIC is not in

the Reno-friendly region and _cwnd_ is less than _W_max_, then CUBIC is in the concave region. In this region, _cwnd_ MUST be incremented by

$$\frac{target - cwnd}{cwnd}$$

for each received new ACK, where _target_ is calculated as described in Section 4.2.

4.5. Convex Region

When receiving a new ACK in congestion avoidance, if CUBIC is not in the Reno-friendly region and _cwnd_ is larger than or equal to _W_max_, then CUBIC is in the convex region.

The convex region indicates that the network conditions might have changed since the last congestion event, possibly implying more available bandwidth after some flow departures. Since the Internet is highly asynchronous, some amount of perturbation is always possible without causing a major change in available bandwidth.

Unless the cwnd is overridden by the AIMD window increase, CUBIC will behave cautiously when operating in this region. The convex profile aims to increase the window very slowly at the beginning when _cwnd_ is around _W_max_ and then gradually increases its rate of increase. This region is also called the "maximum probing phase", since CUBIC is searching for a new _W_max_. In this region, _cwnd_ MUST be incremented by

$$\frac{target - cwnd}{cwnd}$$

for each received new ACK, where _target_ is calculated as described in Section 4.2.

4.6. Multiplicative Decrease

When a congestion event is detected by the mechanisms described in Section 3.1, CUBIC updates _W_max_ and reduces _cwnd_ and _ssthresh_ immediately, as described below. In the case of packet loss, the sender MUST reduce _cwnd_ and _ssthresh_ immediately upon entering loss recovery, similar to [RFC5681] (and [RFC6675]). Note that other mechanisms, such as Proportional Rate Reduction [RFC6937], can be used to reduce the sending rate during loss recovery more gradually. The parameter β __cubic_ SHOULD be set to 0.7, which is different from the multiplicative decrease factor used in [RFC5681] (and [RFC6675]) during fast recovery.

In Figure 5, _flight_size_ is the amount of outstanding (unacknowledged) data in the network, as defined in [RFC5681]. Note that a rate-limited application with idle periods or periods when unable to send at the full rate permitted by _cwnd_ could easily encounter notable variations in the volume of data sent from one RTT to another, resulting in _flight_size_ that is significantly less than _cwnd_ when there is a congestion event. The congestion response would therefore decrease _cwnd_ to a much lower value than necessary. To avoid such suboptimal performance, the mechanisms described in [RFC7661] can be used. [RFC7661] describes how to manage and use _cwnd_ and _ssthresh_ during a rate-limited interval, and how to update _cwnd_ and _ssthresh_ after congestion has been detected. The mechanisms defined in [RFC7661] are safe to use even when _cwnd_ is greater than the receive window, because they validate _cwnd_ based on the amount of data acknowledged by the network in an

RTT, which implicitly accounts for the allowed receive window.

Some implementations of CUBIC currently use _cwnd_ instead of _flight_size_ when calculating a new _ssthresh_. Implementations that use _cwnd_ MUST use other measures to prevent _cwnd_ from growing when the volume of bytes in flight is smaller than _cwnd_. This also effectively prevents _cwnd_ from growing beyond the receive window. Such measures are important for preventing a CUBIC sender from using an arbitrarily high cwnd _value_ when calculating new values for _ssthresh_ and _cwnd_ when congestion is detected. This might not be as robust as the mechanisms described in [RFC7661].

A QUIC sender that uses a _cwnd_ _value_ to calculate new values for _cwnd_ and _ssthresh_ after detecting a congestion event is REQUIRED to apply similar mechanisms [RFC9002].

```
 ssthresh =  flight_size * β        new  ssthresh
                            cubic
 cwnd      = cwnd                   save  cwnd
     prior
             max(ssthresh, 2)    reduction on loss, cwnd >= 2 SMSS
 cwnd =      max(ssthresh, 1)    reduction on ECE, cwnd >= 1 SMSS

 ssthresh =  max(ssthresh, 2)     ssthresh >= 2 SMSS
```

Figure 5

A side effect of setting β __cubic_ to a value bigger than 0.5 is that packet loss can happen for more than one RTT in certain cases, but it can work efficiently in other cases -- for example, when HyStart++ [RFC9406] is used along with CUBIC or when the sending rate is limited by the application. While a more adaptive setting of β __cubic_ could help limit packet loss to a single round, it would require detailed analyses and large-scale evaluations to validate such algorithms.

Note that CUBIC MUST continue to reduce _cwnd_ in response to congestion events detected by ECN-Echo ACKs until it reaches a value of 1 SMSS. If congestion events indicated by ECN-Echo ACKs persist, a sender with a _cwnd_ of 1 SMSS MUST reduce its sending rate even further. This can be achieved by using a retransmission timer with exponential backoff, as described in [RFC3168].

4.7. Fast Convergence

To improve convergence speed, CUBIC uses a heuristic. When a new flow joins the network, existing flows need to give up some of their bandwidth to allow the new flow some room for growth if the existing flows have been using all the network bandwidth. To speed up this bandwidth release by existing flows, the following fast convergence mechanism SHOULD be implemented.

With fast convergence, when a congestion event occurs, _W_max_ is updated as follows, before the window reduction described in Section 4.6.

$$W_{max} = \begin{cases} cwnd * \frac{1+\beta_{cubic}}{2} & \text{if } cwnd < W_{max} \text{ and fast convergence enabled, further reduce } W_{max} \\ cwnd & \text{otherwise, remember cwnd before reduction} \end{cases}$$

During a congestion event, if the current _cwnd_ is less than

_W_max_, this indicates that the saturation point experienced by this flow is getting reduced because of a change in available bandwidth. This flow can then release more bandwidth by reducing _W_max_ further. This action effectively lengthens the time for this flow to increase its congestion window, because the reduced _W_max_ forces the flow to plateau earlier. This allows more time for the new flow to catch up to its congestion window size.

Fast convergence is designed for network environments with multiple CUBIC flows. In network environments with only a single CUBIC flow and without any other traffic, fast convergence SHOULD be disabled.

4.8. Timeout

In the case of a timeout, CUBIC follows Reno to reduce _cwnd_ [RFC5681] but sets _ssthresh_ using β __cubic_ (same as in Section 4.6) in a way that is different from Reno TCP [RFC5681].

During the first congestion avoidance stage after a timeout, CUBIC increases its congestion window size using Figure 1, where _t_ is the elapsed time since the beginning of the current congestion avoidance stage, _K_ is set to 0, and _W_max_ is set to the congestion window size at the beginning of the current congestion avoidance stage. In addition, for the Reno-friendly region, _W_est_ SHOULD be set to the congestion window size at the beginning of the current congestion avoidance stage.

4.9. Spurious Congestion Events

In cases where CUBIC reduces its congestion window in response to having detected packet loss via duplicate ACKs or timeouts, it is possible that the missing ACK could arrive after the congestion window reduction and a corresponding packet retransmission. For example, packet reordering could trigger this behavior. A high degree of packet reordering could cause multiple congestion window reduction events, where spurious losses are incorrectly interpreted as congestion signals, thus degrading CUBIC's performance significantly.

For TCP, there are two types of spurious events: spurious timeouts and spurious fast retransmits. In the case of QUIC, there are no spurious timeouts, as the loss is only detected after receiving an ACK.

4.9.1. Spurious Timeouts

An implementation MAY detect spurious timeouts based on the mechanisms described in Forward RTO-Recovery [RFC5682]. Experimental alternatives include the Eifel detection algorithm [RFC3522]. When a spurious timeout is detected, a TCP implementation MAY follow the response algorithm described in [RFC4015] to restore the congestion control state and adapt the retransmission timer to avoid further spurious timeouts.

4.9.2. Spurious Fast Retransmits

Upon receiving an ACK, a TCP implementation MAY detect spurious fast retransmits either using TCP Timestamps or via D-SACK [RFC2883]. As noted above, experimental alternatives include the Eifel detection algorithm [RFC3522], which uses TCP Timestamps; and DSACK-based detection [RFC3708], which uses DSACK information. A QUIC implementation can easily determine a spurious fast retransmit if a QUIC packet is acknowledged after it has been marked as lost and the original data has been retransmitted with a new QUIC packet.

This section specifies a simple response algorithm when a spurious

fast retransmit is detected by acknowledgments. Implementations would need to carefully evaluate the impact of using this algorithm in different environments that may experience a sudden change in available capacity (e.g., due to variable radio capacity, a routing change, or a mobility event).

When packet loss is detected via acknowledgments, a CUBIC implementation MAY save the current value of the following variables before the congestion window is reduced.

```
undo_cwnd =      cwnd
undo_cwnd      = cwnd
          prior       prior
undo_ssthresh =  ssthresh
undo_W    =      W
      max         max
undo_K =         K
undo_t      =    t
      epoch       epoch
undo_W    =      W
      est         est
```

Once the previously declared packet loss is confirmed to be spurious, CUBIC MAY restore the original values of the above-mentioned variables as follows if the current _cwnd_ is lower than _cwnd_prior_. Restoring the original values ensures that CUBIC's performance is similar to what it would be without spurious losses.

```
cwnd =      undo_cwnd
cwnd      = undo_cwnd
    prior            prior
ssthresh =  undo_ssthresh
W    =      undo_W
 max              max      if cwnd < cwnd
K =         undo_K                        prior
t      =    undo_t
 epoch            epoch
W    =      undo_W
 est              est
```

In rare cases, when the detection happens long after a spurious fast retransmit event and the current _cwnd_ is already higher than _cwnd_prior_, CUBIC SHOULD continue to use the current and the most recent values of these variables.

4.10. Slow Start

When _cwnd_ is no more than _ssthresh_, CUBIC MUST employ a slow start algorithm. In general, CUBIC SHOULD use the HyStart++ slow start algorithm [RFC9406] or MAY use the Reno TCP slow start algorithm [RFC5681] in the rare cases when HyStart++ is not suitable. Experimental alternatives include hybrid slow start [HR11], a predecessor to HyStart++ that some CUBIC implementations have used as the default for the last decade, and limited slow start [RFC3742]. Whichever startup algorithm is used, work might be needed to ensure that the end of slow start and the first multiplicative decrease of congestion avoidance work well together.

When CUBIC uses HyStart++ [RFC9406], it may exit the first slow start without incurring any packet loss and thus _W_max_ is undefined. In this special case, CUBIC sets _cwnd_prior = cwnd_ and switches to congestion avoidance. It then increases its congestion window size using Figure 1, where _t_ is the elapsed time since the beginning of the current congestion avoidance stage, _K_ is set to 0, and _W_max_ is set to the congestion window size at the beginning of the current congestion avoidance stage.

5. Discussion

This section further discusses the safety features of CUBIC, following the guidelines specified in [RFC5033].

With a deterministic loss model where the number of packets between two successive packet losses is always _1/p_, CUBIC always operates with the concave window profile, which greatly simplifies the performance analysis of CUBIC. The average window size of CUBIC (see Appendix B) can be obtained via the following function:

$$AVG\_W_{cubic} = \frac{\sqrt[4]{\frac{C * (3 + \beta_{cubic})}{4 * (1 - \beta_{cubic})}}}{\sqrt[4]{\frac{RTT^3}{p^3}}}$$

Figure 6

With β __cubic_ set to 0.7, the above formula reduces to

$$AVG\_W_{cubic} = \frac{\sqrt[4]{\frac{C * 3.7}{1.2}}}{\sqrt[4]{\frac{RTT^3}{p^3}}}$$

Figure 7

The following subsection will determine the value of _C_ using Figure 7.

5.1. Fairness to Reno

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

Reno performs well in the following two types of networks:

1. networks with a small bandwidth-delay product (BDP)

2. networks with short RTTs, but not necessarily a small BDP

CUBIC is designed to behave very similarly to Reno in the above two types of networks. The following two tables show the average window sizes of Reno TCP, HSTCP, and CUBIC TCP. The average window sizes of Reno TCP and HSTCP are from [RFC3649]. The average window size of CUBIC is calculated using Figure 7 and the CUBIC Reno-friendly region for three different values of _C_.

Loss Rate P	Reno	HSTCP	CUBIC (C=0.04)	CUBIC (C=0.4)	CUBIC (C=4)
1.0e-02	12	12	12	12	12
1.0e-03	38	38	38	38	59
1.0e-04	120	263	120	187	333
1.0e-05	379	1795	593	1054	1874

1.0e-06	1200	12280	3332	5926	10538
1.0e-07	3795	83981	18740	33325	59261
1.0e-08	12000	574356	105383	187400	333250

Table 1: Reno TCP, HSTCP, and CUBIC with RTT = 0.1 Seconds

Table 1 describes the response function of Reno TCP, HSTCP, and CUBIC in networks with _RTT_ = 0.1 seconds. The average window size is in SMSS-sized segments.

Loss Rate P	Reno	HSTCP	CUBIC (C=0.04)	CUBIC (C=0.4)	CUBIC (C=4)
1.0e-02	12	12	12	12	12
1.0e-03	38	38	38	38	38
1.0e-04	120	263	120	120	120
1.0e-05	379	1795	379	379	379
1.0e-06	1200	12280	1200	1200	1874
1.0e-07	3795	83981	3795	5926	10538
1.0e-08	12000	574356	18740	33325	59261

Table 2: Reno TCP, HSTCP, and CUBIC with RTT = 0.01 Seconds

Table 2 describes the response function of Reno TCP, HSTCP, and CUBIC in networks with _RTT_ = 0.01 seconds. The average window size is in SMSS-sized segments.

Both tables show that CUBIC with any of these three _C_ values is more friendly to Reno TCP than HSTCP, especially in networks with a short _RTT_ where Reno TCP performs reasonably well. For example, in a network with _RTT_ = 0.01 seconds and p=10^-6, Reno TCP has an average window of 1200 packets. If the packet size is 1500 bytes, then Reno TCP can achieve an average rate of 1.44 Gbps. In this case, CUBIC with _C_=0.04 or _C_=0.4 achieves exactly the same rate as Reno TCP, whereas HSTCP is about ten times more aggressive than Reno TCP.

_C_ determines the aggressiveness of CUBIC in competing with other congestion control algorithms for bandwidth. CUBIC is more friendly to Reno TCP if the value of _C_ is lower. However, it is NOT RECOMMENDED to set _C_ to a very low value like 0.04, since CUBIC with a low _C_ cannot efficiently use the bandwidth in fast and long-distance networks. Based on these observations and extensive deployment experience, _C_=0.4 seems to provide a good balance between Reno-friendliness and aggressiveness of window increase. Therefore, _C_ SHOULD be set to 0.4. With _C_ set to 0.4, Figure 7 is reduced to

$$AVG\_W_{cubic} = 1.054 * \frac{\sqrt[4]{RTT^3}}{\sqrt[4]{p^3}}$$

Figure 8

Figure 8 is then used in the next subsection to show the scalability of CUBIC.

5.2. Using Spare Capacity

CUBIC uses a more aggressive window increase function than Reno for fast and long-distance networks.

Table 3 shows that to achieve the 10 Gbps rate, Reno TCP requires a packet loss rate of 2.0e-10, while CUBIC TCP requires a packet loss rate of 2.9e-8.

Throughput (Mbps)	Average W	Reno P	HSTCP P	CUBIC P
1	8.3	2.0e-2	2.0e-2	2.0e-2
10	83.3	2.0e-4	3.9e-4	2.9e-4
100	833.3	2.0e-6	2.5e-5	1.4e-5
1000	8333.3	2.0e-8	1.5e-6	6.3e-7
10000	83333.3	2.0e-10	1.0e-7	2.9e-8

Table 3: Required Packet Loss Rate for Reno TCP, HSTCP, and CUBIC to Achieve a Certain Throughput

Table 3 describes the required packet loss rate for Reno TCP, HSTCP, and CUBIC to achieve a certain throughput, with 1500-byte packets and an _RTT_ of 0.1 seconds.

The test results provided in [HLRX07] indicate that, in typical cases with a degree of background traffic, CUBIC uses the spare bandwidth left unused by existing Reno TCP flows in the same bottleneck link without taking away much bandwidth from the existing flows.

5.3. Difficult Environments

CUBIC is designed to remedy the poor performance of Reno in fast and long-distance networks.

5.4. Investigating a Range of Environments

CUBIC has been extensively studied using simulations, testbed emulations, Internet experiments, and Internet measurements, covering a wide range of network environments [HLRX07] [H16] [CEHRX09] [HR11] [BSCLU13] [LBEWK16]. They have convincingly demonstrated that CUBIC delivers substantial benefits over classical Reno congestion control [RFC5681].

Same as Reno, CUBIC is a loss-based congestion control algorithm. Because CUBIC is designed to be more aggressive (due to a faster window increase function and bigger multiplicative decrease factor) than Reno in fast and long-distance networks, it can fill large drop-tail buffers more quickly than Reno and increases the risk of a standing queue [RFC8511]. In this case, proper queue sizing and management [RFC7567] could be used to mitigate the risk to some extent and reduce the packet queuing delay. Also, in large-BDP networks after a congestion event, CUBIC, due to its cubic window increase function, recovers quickly to the highest link utilization

point. This means that link utilization is less sensitive to an active queue management (AQM) target that is lower than the amplitude of the whole sawtooth.

Similar to Reno, the performance of CUBIC as a loss-based congestion control algorithm suffers in networks where packet loss is not a good indication of bandwidth utilization, such as wireless or mobile networks [LIU16].

5.5. Protection against Congestion Collapse

With regard to the potential of causing congestion collapse, CUBIC behaves like Reno, since CUBIC modifies only the window adjustment algorithm of Reno. Thus, it does not modify the ACK clocking and timeout behaviors of Reno.

CUBIC also satisfies the "full backoff" requirement as described in [RFC5033]. After reducing the sending rate to one packet per RTT in response to congestion events detected by ECN-Echo ACKs, CUBIC then exponentially increases the transmission timer for each packet retransmission while congestion persists.

5.6. Fairness within the Alternative Congestion Control Algorithm

CUBIC ensures convergence of competing CUBIC flows with the same RTT in the same bottleneck links to an equal throughput. When competing flows have different RTT values, their throughput ratio is linearly proportional to the inverse of their RTT ratios. This is true and is independent of the level of statistical multiplexing on the link. The convergence time depends on the network environments (e.g., bandwidth, RTT) and the level of statistical multiplexing, as mentioned in Section 3.4.

5.7. Performance with Misbehaving Nodes and Outside Attackers

CUBIC does not introduce new entities or signals, so its vulnerability to misbehaving nodes or attackers is unchanged from Reno.

5.8. Behavior for Application-Limited Flows

A flow is application limited if it is currently sending less than what is allowed by the congestion window. This can happen if the flow is limited by either the sender application or the receiver application (via the receiver’s advertised window) and thus sends less data than what is allowed by the sender’s congestion window.

CUBIC does not increase its congestion window if a flow is application limited. Per Section 4.2, it is required that _t_ in Figure 1 not include application-limited periods, such as idle periods; otherwise, W_cubic(_t_) might be very high after restarting from these periods.

5.9. Responses to Sudden or Transient Events

If there is a sudden increase in capacity, e.g., due to variable radio capacity, a routing change, or a mobility event, CUBIC is designed to utilize the newly available capacity more quickly than Reno.

On the other hand, if there is a sudden decrease in capacity, CUBIC reduces more slowly than Reno. This remains true regardless of whether CUBIC is in Reno-friendly mode and regardless of whether fast convergence is enabled.

5.10. Incremental Deployment

CUBIC requires only changes to congestion control at the sender, and it does not require any changes at receivers. That is, a CUBIC sender works correctly with Reno receivers. In addition, CUBIC does not require any changes to routers and does not require any assistance from routers.

6. Security Considerations

CUBIC makes no changes to the underlying security of a transport protocol and inherits the general security concerns described in [RFC5681]. Specifically, changing the window computation on the sender may allow an attacker, through dropping or injecting ACKs (as described in [RFC5681]), to either force the CUBIC implementation to reduce its bandwidth or convince it that there is no congestion when congestion does exist, and to use the CUBIC implementation as an attack vector against other hosts. These attacks are not new to CUBIC and are inherently part of any transport protocol like TCP.

7. IANA Considerations

This document does not require any IANA actions.

8. References

8.1. Normative References

[RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, DOI 10.17487/RFC2119, March 1997, <https://www.rfc-editor.org/info/rfc2119>.

[RFC2883] Floyd, S., Mahdavi, J., Mathis, M., and M. Podolsky, "An Extension to the Selective Acknowledgement (SACK) Option for TCP", RFC 2883, DOI 10.17487/RFC2883, July 2000, <https://www.rfc-editor.org/info/rfc2883>.

[RFC2914] Floyd, S., "Congestion Control Principles", BCP 41, RFC 2914, DOI 10.17487/RFC2914, September 2000, <https://www.rfc-editor.org/info/rfc2914>.

[RFC3168] Ramakrishnan, K., Floyd, S., and D. Black, "The Addition of Explicit Congestion Notification (ECN) to IP", RFC 3168, DOI 10.17487/RFC3168, September 2001, <https://www.rfc-editor.org/info/rfc3168>.

[RFC4015] Ludwig, R. and A. Gurtov, "The Eifel Response Algorithm for TCP", RFC 4015, DOI 10.17487/RFC4015, February 2005, <https://www.rfc-editor.org/info/rfc4015>.

[RFC5033] Floyd, S. and M. Allman, "Specifying New Congestion Control Algorithms", BCP 133, RFC 5033, DOI 10.17487/RFC5033, August 2007, <https://www.rfc-editor.org/info/rfc5033>.

[RFC5348] Floyd, S., Handley, M., Padhye, J., and J. Widmer, "TCP Friendly Rate Control (TFRC): Protocol Specification", RFC 5348, DOI 10.17487/RFC5348, September 2008, <https://www.rfc-editor.org/info/rfc5348>.

[RFC5681] Allman, M., Paxson, V., and E. Blanton, "TCP Congestion Control", RFC 5681, DOI 10.17487/RFC5681, September 2009, <https://www.rfc-editor.org/info/rfc5681>.

[RFC5682] Sarolahti, P., Kojo, M., Yamamoto, K., and M. Hata, "Forward RTO-Recovery (F-RTO): An Algorithm for Detecting

Spurious Retransmission Timeouts with TCP", RFC 5682, DOI 10.17487/RFC5682, September 2009, <https://www.rfc-editor.org/info/rfc5682>.

[RFC6298] Paxson, V., Allman, M., Chu, J., and M. Sargent, "Computing TCP's Retransmission Timer", RFC 6298, DOI 10.17487/RFC6298, June 2011, <https://www.rfc-editor.org/info/rfc6298>.

[RFC6582] Henderson, T., Floyd, S., Gurtov, A., and Y. Nishida, "The NewReno Modification to TCP's Fast Recovery Algorithm", RFC 6582, DOI 10.17487/RFC6582, April 2012, <https://www.rfc-editor.org/info/rfc6582>.

[RFC6675] Blanton, E., Allman, M., Wang, L., Jarvinen, I., Kojo, M., and Y. Nishida, "A Conservative Loss Recovery Algorithm Based on Selective Acknowledgment (SACK) for TCP", RFC 6675, DOI 10.17487/RFC6675, August 2012, <https://www.rfc-editor.org/info/rfc6675>.

[RFC7567] Baker, F., Ed. and G. Fairhurst, Ed., "IETF Recommendations Regarding Active Queue Management", BCP 197, RFC 7567, DOI 10.17487/RFC7567, July 2015, <https://www.rfc-editor.org/info/rfc7567>.

[RFC8174] Leiba, B., "Ambiguity of Uppercase vs Lowercase in RFC 2119 Key Words", BCP 14, RFC 8174, DOI 10.17487/RFC8174, May 2017, <https://www.rfc-editor.org/info/rfc8174>.

[RFC8985] Cheng, Y., Cardwell, N., Dukkipati, N., and P. Jha, "The RACK-TLP Loss Detection Algorithm for TCP", RFC 8985, DOI 10.17487/RFC8985, February 2021, <https://www.rfc-editor.org/info/rfc8985>.

[RFC9002] Iyengar, J., Ed. and I. Swett, Ed., "QUIC Loss Detection and Congestion Control", RFC 9002, DOI 10.17487/RFC9002, May 2021, <https://www.rfc-editor.org/info/rfc9002>.

[RFC9406] Balasubramanian, P., Huang, Y., and M. Olson, "HyStart++: Modified Slow Start for TCP", RFC 9406, DOI 10.17487/RFC9406, May 2023, <https://www.rfc-editor.org/info/rfc9406>.

8.2. Informative References

[AIMD-friendliness] Briscoe, B. and O. Albisser, "Friendliness between AIMD Algorithms", DOI 10.48550/arXiv.2305.10581, May 2023, <https://arxiv.org/abs/2305.10581>.

[BSCLU13] Belhareth, S., Sassatelli, L., Collange, D., Lopez-Pacheco, D., and G. Urvoy-Keller, "Understanding TCP cubic performance in the cloud: A mean-field approach", 2013 IEEE 2nd International Conference on Cloud Networking (CloudNet), DOI 10.1109/cloudnet.2013.6710576, November 2013, <https://doi.org/10.1109/cloudnet.2013.6710576>.

[CEHRX09] Cai, H., Eun, D., Ha, S., Rhee, I., and L. Xu, "Stochastic convex ordering for multiplicative decrease internet congestion control", Computer Networks, vol. 53, no. 3, pp. 365-381, DOI 10.1016/j.comnet.2008.10.012, February 2009, <https://doi.org/10.1016/j.comnet.2008.10.012>.

[FHP00] Floyd, S., Handley, M., and J. Padhye, "A Comparison of Equation-Based and AIMD Congestion Control", May 2000, <https://www.icir.org/tfrc/aimd.pdf>.

[GV02] Gorinsky, S. and H. Vin, "Extended Analysis of Binary Adjustment Algorithms", Technical Report TR2002-39, Department of Computer Sciences, The University of Texas at Austin, August 2002, <https://citeseerx.ist.psu.edu/document?repid=rep1&type=pdf&doi=1828bdcef118b02d3996b8e00b8aaa92b50abb0f>.

[H16] Ha, S., "Deployment, Testbed, and Simulation Results for CUBIC", Wayback Machine archive, 3 November 2016, <https://web.archive.org/web/20161118125842/http://netsrv.csc.ncsu.edu/wiki/index.php/TCP_Testing>.

[HLRX07] Ha, S., Le, L., Rhee, I., and L. Xu, "Impact of background traffic on performance of high-speed TCP variant protocols", Computer Networks, vol. 51, no. 7, pp. 1748-1762, DOI 10.1016/j.comnet.2006.11.005, May 2007, <https://doi.org/10.1016/j.comnet.2006.11.005>.

[HR11] Ha, S. and I. Rhee, "Taming the elephants: New TCP slow start", Computer Networks, vol. 55, no. 9, pp. 2092-2110, DOI 10.1016/j.comnet.2011.01.014, June 2011, <https://doi.org/10.1016/j.comnet.2011.01.014>.

[HRX08] Ha, S., Rhee, I., and L. Xu, "CUBIC: a new TCP-friendly high-speed TCP variant", ACM SIGOPS Operating Systems Review, vol. 42, no. 5, pp. 64-74, DOI 10.1145/1400097.1400105, July 2008, <https://doi.org/10.1145/1400097.1400105>.

[K03] Kelly, T., "Scalable TCP: improving performance in highspeed wide area networks", ACM SIGCOMM Computer Communication Review, vol. 33, no. 2, pp. 83-91, DOI 10.1145/956981.956989, April 2003, <https://doi.org/10.1145/956981.956989>.

[LBEWK16] Lukaseder, T., Bradatsch, L., Erb, B., Van Der Heijden, R., and F. Kargl, "A Comparison of TCP Congestion Control Algorithms in 10G Networks", 2016 IEEE 41st Conference on Local Computer Networks (LCN), DOI 10.1109/lcn.2016.121, November 2016, <https://doi.org/10.1109/lcn.2016.121>.

[LIU16] Liu, K. and J. Lee, "On Improving TCP Performance over Mobile Data Networks", IEEE Transactions on Mobile Computing, vol. 15, no. 10, pp. 2522-2536, DOI 10.1109/tmc.2015.2500227, October 2016, <https://doi.org/10.1109/tmc.2015.2500227>.

[RFC3522] Ludwig, R. and M. Meyer, "The Eifel Detection Algorithm for TCP", RFC 3522, DOI 10.17487/RFC3522, April 2003, <https://www.rfc-editor.org/info/rfc3522>.

[RFC3649] Floyd, S., "HighSpeed TCP for Large Congestion Windows", RFC 3649, DOI 10.17487/RFC3649, December 2003, <https://www.rfc-editor.org/info/rfc3649>.

[RFC3708] Blanton, E. and M. Allman, "Using TCP Duplicate Selective Acknowledgement (DSACKs) and Stream Control Transmission Protocol (SCTP) Duplicate Transmission Sequence Numbers (TSNs) to Detect Spurious Retransmissions", RFC 3708, DOI 10.17487/RFC3708, February 2004, <https://www.rfc-editor.org/info/rfc3708>.

[RFC3742] Floyd, S., "Limited Slow-Start for TCP with Large Congestion Windows", RFC 3742, DOI 10.17487/RFC3742, March 2004, <https://www.rfc-editor.org/info/rfc3742>.

[RFC6937] Mathis, M., Dukkipati, N., and Y. Cheng, "Proportional Rate Reduction for TCP", RFC 6937, DOI 10.17487/RFC6937, May 2013, <https://www.rfc-editor.org/info/rfc6937>.

[RFC7661] Fairhurst, G., Sathiaseelan, A., and R. Secchi, "Updating TCP to Support Rate-Limited Traffic", RFC 7661, DOI 10.17487/RFC7661, October 2015, <https://www.rfc-editor.org/info/rfc7661>.

[RFC8312] Rhee, I., Xu, L., Ha, S., Zimmermann, A., Eggert, L., and R. Scheffenegger, "CUBIC for Fast Long-Distance Networks", RFC 8312, DOI 10.17487/RFC8312, February 2018, <https://www.rfc-editor.org/info/rfc8312>.

[RFC8511] Khademi, N., Welzl, M., Armitage, G., and G. Fairhurst, "TCP Alternative Backoff with ECN (ABE)", RFC 8511, DOI 10.17487/RFC8511, December 2018, <https://www.rfc-editor.org/info/rfc8511>.

[RFC9000] Iyengar, J., Ed. and M. Thomson, Ed., "QUIC: A UDP-Based Multiplexed and Secure Transport", RFC 9000, DOI 10.17487/RFC9000, May 2021, <https://www.rfc-editor.org/info/rfc9000>.

[RFC9260] Stewart, R., Txen, M., and K. Nielsen, "Stream Control Transmission Protocol", RFC 9260, DOI 10.17487/RFC9260, June 2022, <https://www.rfc-editor.org/info/rfc9260>.

[SXEZ19] Sun, W., Xu, L., Elbaum, S., and D. Zhao, "Model-Agnostic and Efficient Exploration of Numerical Congestion Control State Space of Real-World TCP Implementations", IEEE/ACM Transactions on Networking, vol. 29, no. 5, pp. 1990-2004, DOI 10.1109/tnet.2021.3078161, October 2021, <https://doi.org/10.1109/tnet.2021.3078161>.

[XHR04] Xu, L., Harfoush, K., and I. Rhee, "Binary increase congestion control (BIC) for fast long-distance networks", IEEE INFOCOM 2004, DOI 10.1109/infcom.2004.1354672, March 2004, <https://doi.org/10.1109/infcom.2004.1354672>.

Appendix A. Evolution of CUBIC since the Original Paper

CUBIC has gone through a few changes since the initial release [HRX08] of its algorithm and implementation. This appendix highlights the differences between the original paper and [RFC8312].

* The original paper [HRX08] includes the pseudocode of CUBIC implementation using Linux's pluggable congestion control framework, which excludes system-specific optimizations. The simplified pseudocode might be a good starting point for learning about CUBIC.

* [HRX08] also includes experimental results showing its performance and fairness.

* The definition of the β __cubic_ constant was changed in [RFC8312]. For example, β __cubic_ in the original paper was referred to as the window decrease constant, while [RFC8312] changed it to "CUBIC multiplicative decrease factor". With this change, the current congestion window size after a congestion event as listed in [RFC8312] was β __cubic_ * _W_max_, while it was (1-β __cubic_) * _W_max_ in the original paper.

* Its pseudocode used _W_(last_max)_, while [RFC8312] used _W_max_.

* Its AIMD-friendly window was _W_tcp_, while [RFC8312] used _W_est_.

Appendix B. Proof of the Average CUBIC Window Size

This appendix contains a proof for the average CUBIC window size _AVG_W_cubic_ in Figure 6.

We find _AVG_W_cubic_ under a deterministic loss model, where the number of packets between two successive packet losses is 1/_p_. With this model, CUBIC always operates with the concave window profile and the time period between two successive packet losses is _K_.

The average window size _AVG_W_cubic_ is defined as follows, where the numerator 1/_p_ is the total number of packets between two successive packet losses and the denominator _K_/_RTT_ is the total number of RTTs between two successive packet losses.

$$AVG\_W_{cubic} = \frac{\frac{1}{p}}{\frac{K}{RTT}}$$

Figure 9

Below, we find _K_ as a function of CUBIC parameters β __cubic_ and _C_, and network parameters _p_ and _RTT_. According to the definition of _K_ in Figure 2, we have

$$K = \sqrt[3]{\frac{W_{max} - W_{max} * \beta_{cubic}}{C}}$$

Figure 10

The total number of packets between two successive packet losses can also be obtained as follows, using the window increase function in Figure 1. Specifically, the window size in the first RTT (i.e., _n_=1, or equivalently, _t_=0) is _C_(-_K_)^3+_W_max_ and the window size in the last RTT (i.e., _n_=_K_/_RTT_, or equivalently, _t_=_K_-_RTT_) is _C_(-_RTT_)^3+_W_max_.

$$\frac{1}{p} = \sum_{n=1}^{\frac{K}{RTT}} C((n-1) * RTT-K)^3 + W_{max}$$

$$= \sum_{n=1}^{\frac{K}{RTT}} C * RTT^3 (-n)^3 + W_{max}$$

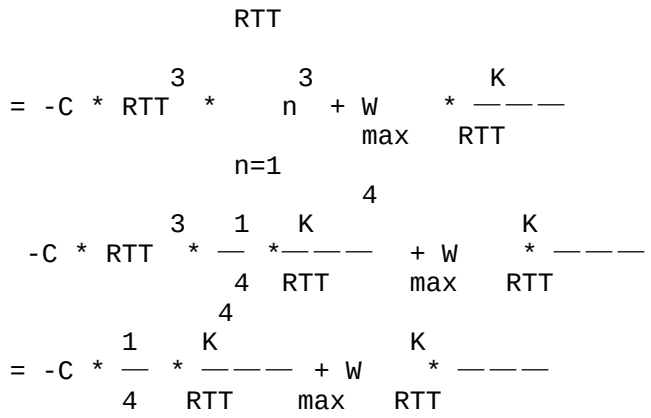


Figure 11

After solving the equations in Figures 10 and 11 for _K_ and _W_max_, we have

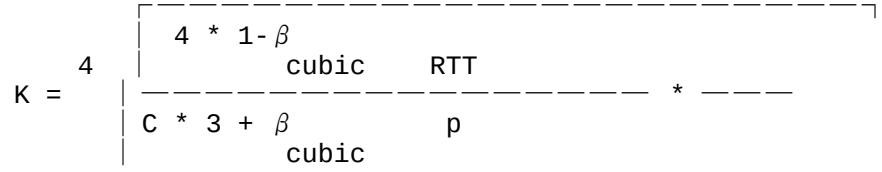
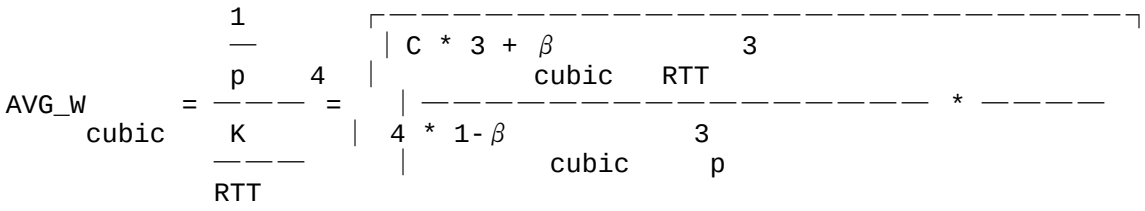


Figure 12

The average CUBIC window size _AVG_W_cubic_ can be obtained by substituting _K_ from Figure 12 in Figure 9.



Acknowledgments

Richard Scheffenegger and Alexander Zimmermann originally coauthored [RFC8312].

These individuals suggested improvements to this document:

* Bob Briscoe
* Christian Huitema
* Gorry Fairhurst
* Jonathan Morton
* Juhamatti Kuusisaari
* Junho Choi
* Markku Kojo
* Martin Duke
* Martin Thomson
* Matt Mathis
* Matt Olson
* Michael Welzl
* Mirja Khlewind
* Mohit P. Tahiliani
* Neal Cardwell
* Praveen Balasubramanian
* Randall Stewart
* Richard Scheffenegger
* Rod Grimes

* Spencer Dawkins
* Tom Henderson
* Tom Petch
* Wesley Rosenblum
* Yoav Nir
* Yoshifumi Nishida
* Yuchung Cheng

Authors' Addresses

Lisong Xu
University of Nebraska-Lincoln
Department of Computer Science and Engineering
Lincoln, NE 68588-0115
United States of America
Email: xu@unl.edu
URI: https://cse.unl.edu/~xu/

Sangtae Ha
University of Colorado at Boulder
Department of Computer Science
Boulder, CO 80309-0430
United States of America
Email: sangtae.ha@colorado.edu
URI: https://netstech.org/sangtaeha/

Injong Rhee
Bowery Farming
151 W 26th Street, 12th Floor
New York, NY 10001
United States of America
Email: injongrhee@gmail.com

Vidhi Goel
Apple Inc.
One Apple Park Way
Cupertino, CA 95014
United States of America
Email: vidhi_goel@apple.com

Lars Eggert (editor)
NetApp
Stenbergintie 12 B
FI-02700 Kauniainen
Finland
Email: lars@eggert.org
URI: https://eggert.org/