ccwg
Internet-Draft
Intended status: Standards Track
Expires: 18 September 2026

J. Chung
viasat
M. Kachooei
WPI
F. Li
viasat
M. Claypool
WPI

17 March 2026

SEARCH -- a New Slow Start Algorithm for TCP and QUIC
draft-chung-ccwg-search-09

Abstract

TCP slow start is designed to ramp up to the network congestion point quickly, doubling the congestion window each round-trip time until the congestion point is reached, whereupon TCP exits the slow start phase. Unfortunately, the default Linux TCP slow start implementation -- TCP Cubic with HyStart [HYSTART] -- can cause premature exit from slow start, especially over wireless links, degrading link utilization. However, without HyStart, TCP exits slow start too late, causing unnecessary packet loss. To improve TCP slow start performance, this document proposes using the Slow start Exit At Right CHokepoint (SEARCH) algorithm [KCL24] where the TCP sender determines the congestion point based on acknowledged deliveries -- specifically, the sender computes the delivered bytes compared to the sent bytes, smoothed to account for link latency variation and normalized to accommodate link capacities, and initiates exits slow start if the delivered bytes are lower than expected. We implemented SEARCH in Linux, FreeBSD, and QUIC and evaluated it over WiFi, 4G/LTE, and low earth orbit (LEO) and geosynchronous (GEO) satellite links. Analysis of the results show that SEARCH reliably exits from slow start after the congestion point is reached but before inducing packet loss.

Status of This Memo

This Internet-Draft is submitted in full conformance with the provisions of BCP 78 and BCP 79.

Internet-Drafts are working documents of the Internet Engineering Task Force (IETF). Note that other groups may also distribute working documents as Internet-Drafts. The list of current Internet-Drafts is at https://datatracker.ietf.org/drafts/current/.

Copyright Notice

Table of Contents

1. Introduction

The TCP slow start mechanism starts sending data rates cautiously yet rapidly increases towards the congestion point, approximately doubling the congestion window (cwnd) each round-trip time (RTT). Unfortunately, default implementations of TCP slow start, such as TCP Cubic with HyStart [HYSTART] in Linux, often result in a premature exit from the slow start phase, or, if HyStart is disabled, excessive packet loss upon overshooting the congestion point. Exiting slow start too early curtails TCP's ability to capitalize on unused link capacity, a setback that is particularly pronounced in high bandwidth-delay product (BDP) networks (e.g., GEO satellites) where the time to grow the congestion window to the congestion point is substantial. Conversely, exiting slow start too late overshoots the link's capacity, inducing unnecessary congestion and packet loss, particularly problematic for links with large (bloated) bottleneck queues.

To determine the slow start exit point, we propose that the TCP sender monitors the acknowledged delivered bytes in an RTT and compares them to the bytes sent during the previous RTT. A large difference between the data sent earlier and the data currently delivered indicates that the network has reached the congestion point and that the slow start phase should exit. We call our approach the Slow start Exit At Right CHokepoint (SEARCH) algorithm. SEARCH is based on the observation that during slow start the congestion window typically increases by one maximum segment size (MSS) for each acknowledgment (ACK) received, causing the sender's transmission rate to grow rapidly. As long as the path is not capacity-limited, increases in sent data result in proportional increases in delivered data. However, once the sending rate surpasses the network congestion point, additional sent data no longer produces proportional forward progress, and delivered bytes begin to lag behind previously sent data. SEARCH detects this divergence and transitions to a draining phase that converges the congestion window toward the path capacity before exiting slow start. To accommodate links with a wide range of capacities, SEARCH normalizes the difference between delivered bytes and previously sent bytes relative to the sent-byte baseline. Since link latencies can vary over time independently of data rates (especially for wireless links), SEARCH smooths the measured sent and delivered rates over several RTTs.

This document describes the current version of the SEARCH algorithm, version 4. Active work on the SEARCH algorithm is continuing.

This document is organized as follows: Section 2 provides terminology and definitions relevant used throughout this document; Section 3 describes the SEARCH algorithm in detail; Section 4 provides

justification for algorithm settings; Section 5 describes the implementation status; Section 6 describes security considerations; Section 7 notes that there are no IANA considerations; Section 8 closes with acknowledgments; and Section 9 provides references.

2. Terminology and Definitions

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in RFC 2119, BCP 14 [RFC2119] and indicate requirement levels for compliant CoAP implementations.

In this document, the term "byte" is used in its now customary sense as a synonym for "octet".

ACK: a TCP acknowledgement.

bins: aggregates of bytes measured over small time windows used to track transmission history. SEARCH maintains separate bins for acknowledged delivered bytes (acked bins) and for sent bytes (sent bins).

congestion window (cwnd): A TCP state variable that limits the amount of data a TCP can send. At any given time, a TCP flow MUST NOT send data with a sequence number higher than the sum of the highest acknowledged sequence number and the minimum of the cwnd and receiver window.

norm: the normalized difference between delivered bytes and the previously sent bytes used as the expected progress baseline.

round-trip time (RTT): the round-trip time for a segment sent until the acknowledgement is received.

THRESH: the norm value above which SEARCH considers the congestion point to be reached and transitions to the drain phase.

drain phase: the SEARCH phase entered after the congestion point is detected, where the congestion window converges toward a target derived from delivered bytes.

3. SEARCH Algorithm

The concept that during the slow start phase the amount of data successfully delivered should increase rapidly until the congestion point is reached is core to the SEARCH algorithm. In SEARCH, the sender compares the delivered bytes with the data sent one RTT earlier. When the delivered bytes closely match the previously sent bytes, the link is not yet capacity-limited, whereas when the delivered bytes fall significantly below the previously sent bytes, the link capacity has been reached and SEARCH transitions to a drain phase, during which the congestion window converges toward a target based on an estimate of the congestion window at the paths' available capacity. The drain phase reduces excess in-flight data accumulated during slow start and allows the sender to converge toward the path capacity before exiting slow start.

One challenge in monitoring sent and delivered data across multiple RTTs is latency variability for some links. Variable latency in the absence of congestion - common in some wireless links - can cause RTTs to differ over time even when the network is not yet at the congestion point. This variability complicates comparing delivered bytes with sent bytes during an earlier RTT. A temporary decrease in latency can make the delivered bytes appear lower relative to previously sent bytes, falsely suggesting that the congestion point has been reached when it has not.

To counteract link latency variability, SEARCH tracks sent and delivered data over several RTTs in a sliding window providing a more stable basis for comparison. Since tracking individual segment sent and delivery times is prohibitive in terms of memory use, the data within the sliding window is aggregated over bins representing small, fixed time periods. The window then slides over bin-by-bin, rather than sliding every acknowledgement (ACK), reducing both the computational load (since SEARCH only triggers at the bin boundary) and the memory requirements (since sent and delivered byte totals are kept for a bin-sized time interval instead of for each segment).

3.1. Algorithm Overview

An overview of the SEARCH algorithm (that runs on the TCP server only) is provided below.

In lines 1-3, upon initialization of a TCP connection, the SEARCH window (W) is set based on the initial round trip time (init_rtt), and the sent and delivery histories are cleared (hist_sent[] and hist_devl[], respectively)

The main part of the SEARCH algorithm runs in lines 4-20 when an ACK is received. Line 4 does bookkeeping, updating the sent and delivery history based on the current time and the TCP sequence number (sequence_num).

Line 5 checks whether SEARCH is currently in the drain phase. If SEARCH is not in the drain phase, line 6 computes the number of bytes delivered in the current SEARCH window, using the delivered byte history to tally the delivered bytes from the now (the current time) to the now minus W. Line 7 computes the number of bytes sent for the previous RTT i.e,. from now minus an RTT to now minus an RTT minus W.

Line 8 computes the difference between the previous sent bytes and the current delivered bytes. Line 9 normalizes this difference since it is the relative deviation between delivered and expected sent bytes that indicates the congestion point has been reached. Line 10 compares this normalized difference (norm_diff) to the SEARCH threshold (THRESH), and if this threshold has been surpassed, line 11 estimates the target congestion window (target_cwnd) and line 12 transitions SEARCH to the drain phase.

If SEARCH is already in the drain phase, line 14 adjusts the congestion window toward target_cwnd. Lines 15-18 check whether the congestion window has reached target_cwnd; if so, SEARCH sets ssthresh to cwnd and exits slow start.

SEARCH ALGORITHM OVERVIEW

```
upon TCP connection:

1:  W = 3.5 * init_rtt  // SEARCH window size
2:  hist_sent[] = {}    // Array holding sent history
3:  hist_delv[] = {}    // Array holding delivery history

on ACK arrived (sequence_num, rtt):

    // Update history.
4:  update_hist(hist_delv, hist_sent, sequence_num)

    // If not in drain phase, evaluate the SEARCH signal
5:  if (not in_drain_phase) then
      // Compute delivered and sent windows.
6:    curr_delv = compute_delv(hist_delv, now - W, now)
7:    prev_sent = compute_sent(hist_sent, now - W - rtt, now - rtt)

      // Compare sent to delivered, normalized
8:    diff = prev_sent - curr_delv
9:    norm_diff = diff / prev_sent
10:   if (norm_diff >= THRESH) then
11:     target_cwnd = estimate_capacity()
12:     enter_drain_phase()
13:   end if

    // If in drain phase, converge toward target_cwnd
14: else
15:   update_cwnd_toward(target_cwnd)
16:   if (cwnd <= target_cwnd) then
17:     ssthresh = cwnd
18:     exit_slow_start()
19:   end if
20: end if
```

3.2. The Complete Algorithm

The complete SEARCH algorithm (that runs on the TCP server only) is shown below.

The core of the algorithm overview presented above is preserved in the complete algorithm below. But in order to made the code practical, the sent and delivery history information from the TCP ACKs is binned, aggregating sent and delivered byte information over a small time period. Maintaining the bins is done via a circular array, with checks to make sure the array has enough data for the SEARCH computations (i.e., bins over time period W for the current delivery window, and bins over time period W for the sent window for the previous round-trip time). In addition, the total memory footprint used by the bins is managed via bit shifting, decreasing the byte values stored when they get too large.

The parameters in CAPS (lines 1-9) are constants, with the INITIAL_RTT (on line 1) obtained via the first round-trip time measured in the TCP connection.

The variables in Initialization (lines 10-17) are set once, upon establishment of a TCP connection.

The variable *now* on lines 19 and 42 is the current system time when the code is called.

The variable sequence_num and rtt in the ACK_arrived() function (above line 18) are obtained upon arrival of an acknowledgement from the receiver.

The variable *cwnd* on lines 36, 37 and 88 is the current congestion window.

Lines 1-9 set the predefined parameters for the SEARCH algorithm. The window size (WINDOW_SIZE) is 3.5 times the initial RTT. The delivered and sent bytes over a window are approximated using 10 (W) bins, with an additional 15 (EXTRA_BINS) bins used to maintain sent history so that sent data from approximately one RTT earlier can be compared to the current delivered data. The bin duration (BIN_DURATION) is the window size divided by the number of bins. The threshold (THRESH) defaults to 0.26, and is the upper bound of the permissible difference between the previously sent bytes and the current delivered bytes (normalized) above which detect congestion point. The maximum value for each bin (MAX_BIN_VALUE) that must be less than largest TCP sequence number. The DRAIN_RATE controls how many acknowledged segments must be observed before the congestion window is incremented during the drain phase.

Lines 10-17 do one-time initialization of search variables when a TCP connection is established. With a bin boundary (bin_end) of 0 initially, the first ack that arrives is placed in the first bin.

Once a TCP flow starts, SEARCH only acts when acknowledgements (ACKs) are received and even then, only for the first ACK that arrives after the end of the latest bin boundary (stored in the variable bin_end). This check happens on line 19 and, if the bin boundary is passed, the bin statistics are updated in the call to update_bins() on line 20.

In update_bins() (lines 42-69), in most TCP connections, the time (*now*) will be in the successive bin, but in some cases (such as an RTT spike or a TCP connection without data to send), more than one bin boundary may have been passed. Line 42 computes how many bins have been passed.

If more than one bin has been passed, any such "skipped" bins are updated with the most recently updated bin. These skipped bins are updated via the loop in lines 43-46. Line 47 updates the current bin index (curr_idx) based on the number of bins that have been passed (again, typically this will be 1) and line 48 updates the next bin boundary (bin_end) based on the number of passed bins (passed_bins) and the bin duration (BIN_DURATION).

In lines 49-67, the memory used by each bin can be reduced (e.g., a u16) to be less than the memory used for a TCP sequence number (e.g., a u32). To handle this, when updating the bin value, on lines 49 and 50 the sequence number is first scaled by the scale factor (initially set to 0). On line 51, the larger of the two scaled values is used to determine whether either value exceeds the maximum value a bin can hold (MAX_BIN_VALUE, set to the largest u16 by default). If the maximum scaled value is too large, then lines 52-57 shift (scale) the value until it fits. In lines 58-63, all previously stored delivered and sent bin values that had only been scaled by the previous scale factor are re-scaled by the additional amount (shift_amount), and the total scaling factor (scale_factor) is updated in line 64. Lastly, in lines 65 and 66, the current delivered and sent values are shifted by the same additional amount, and on lines 68 and 69 the most recent bin values are stored.

Once the bins are updated, lines 21-23 check if enough bins have been filled to run SEARCH. This requires more than W (10) bins, but also enough to shift back by an RTT to compute a window (10) of sent bins there.

If there are enough bins to run SEARCH, lines 24-26 compute the current delivered bytes and the sent bytes from one RTT earlier. The delivered bytes over the window is computed in the function compute_delv() and the sent bytes over the window is computed in the function compute_sent(). For sent bytes, shifting by an RTT may land between bin boundaries, so the computation is interpolated by the fraction on either side, computed on line 25.

Lines 70-72 compute the delivered bytes over the delivered bins by taking the difference between the cumulative delivered values at the two bin boundaries and returning this difference. Lines 73-77 compute the sent bytes over the sent bins in a similar fashion, but since shifting the sent window by approximately one RTT may land between bin boundaries, first the "upper" sent window is computed (which is 0, if fraction is 0), then adding the "lower" sent window, and finally returning this sum.

Once computed, the difference between the previously sent bytes (prev_sent) and the current delivered bytes (curr_delv) is normalized (line 27) and then compared to the threshold (THRESH) in line 28. If this difference exceeds THRESH, SEARCH detects that the congestion point has been reached. SEARCH then estimates a target congestion window (target_cwnd) in estimate_target_cwnd() and enters the drain phase in line 29 and 30.

In estimate_target_cwnd() (lines 78-82), line 78 computes the number of bins crossed for the past RTT (rtt_bins), and line 79 sets the corresponding index (cong_idx). Line 80 computes the target congestion window (target_cwnd) based on the delivered bytes over the last RTT (the current bin index, curr_idx, minus cong_idx). Line 81 makes sure this window is not smaller than the initial congestion window (INIT_CWND). Line 82 returns this value.

During the drain phase (lines 34-41), the congestion window gradually converges toward the target congestion window using the function update_cwnd_toward(). In this function (lines 83-88), the current in-flight data is first measured (line 83). The number of segments acknowledged by the current ACK is accumulated in drain_acks (line 84). Once a sufficient number of segments have been acknowledged (controlled by DRAIN_RATE), the variable determines how many congestion window increments are permitted (adds, line 85), and the remaining acknowledged segments are retained for future updates (line 86). A new congestion window value is then computed based on the current in-flight data and the allowed increments (line 87). Finally, the congestion window is updated to the larger of this value or the target congestion window (line 88), ensuring that the congestion window does not fall below the estimated capacity.

Once the congestion window reaches the target value, slow start exits by setting ssthresh to cwnd and resetting the SEARCH state using reset_search() (lines 36-40). The reset_search() function (lines 89-94) resets the SEARCH state variables. The current bin index (curr_idx) and scaling factor (scale_factor) are cleared, the bin boundary (bin_end) is reset, and the drain phase state variables (target_cwnd, in_drain, and drain_acks) are reinitialized. This ensures that SEARCH restarts its measurements using fresh history when the algorithm is reset.

SEARCH 4.0 ALGORITHM

```
Parameters:
1: WINDOW_SIZE = INITIAL_RTT x 3.5
2: W = 10
3: EXTRA_BINS = 15
4: NUM_ACKED_BINS = W + 1
5: NUM_SENT_BINS = W + EXTRA_BINS
6: BIN_DURATION = WINDOW_SIZE / W
7: THRESH = 0.26
8: MAX_BIN_VALUE = 0xFFFF  // 16-bit
9: DRAIN_RATE = 3

Initialization():
10: acked_bin[NUM_ACKED_BINS] = {}
11: sent_bin[NUM_SENT_BINS] = {}
12: curr_idx = -1
13: bin_end = 0
14: scale_factor = 0
15: in_drain = false
16: target_cwnd = 0
17: drain_acks = 0

ACK_arrived(sequence_num, rtt):
    // If not in drain phase, run SEARCH detection.
18: if (in_drain == false) then
      // Check if passed bin boundary.
19:   if (*now* > bin_end) then
20:     update_bins()

        // Check if enough data for SEARCH.
21:     prev_idx = curr_idx - (rtt / BIN_DURATION)
22:     if (prev_idx > W) and
23:        (curr_idx - prev_idx) < EXTRA_BINS then

          // Run SEARCH check.
24:       curr_delv = compute_delv(curr_idx - W, curr_idx)
25:       frac = (rtt mod BIN_DURATION) / BIN_DURATION
```

```
26:       prev_sent = compute_sent(prev_idx - W, prev_idx, frac)
27:       norm_diff = (prev_sent - curr_delv) / prev_sent
28:       if (norm_diff >= THRESH) then
29:         target_cwnd = estimate_target_cwnd()
30:         in_drain = true
31:       end if
32:     end if // Enough data for SEARCH.
33:   end if // Passed bin boundary.

    // If in drain phase, converge toward target_cwnd.
34: else
35:   update_cwnd_toward(target_cwnd)
36:   if (cwnd <= target_cwnd) then
37:     ssthresh = cwnd
38:     exit_slow_start()
39:     reset_search()
40:   end if
41: end if // Each ACK.



// Update bin statistics.
// Handle cases where more than one bin boundary passed.
// Scale bins (shift) if larger than max bin size.
update_bins():
42: passed_bins = (*now* - bin_end) / BIN_DURATION + 1

  // For remaining skipped, propagate prev bin value.
43: for i = curr_idx+1 to (curr_idx + passed_bins - 1)
44:   acked_bin[i mod NUM_ACKED_BINS] = acked_bin[curr_idx]
45:   sent_bin[i mod NUM_SENT_BINS] = sent_bin[curr_idx]
46: end for

47: curr_idx += passed_bins
48: bin_end += passed_bins x BIN_DURATION

    // Scale bins (shift) if too large.
49: acked_value = delivered_bytes >> scale_factor
50: sent_value = sent_bytes >> scale_factor
51: max_value = max(acked_value, sent_value)

52: if (max_value > MAX_BIN_VALUE) then
53:   shift_amount = 0
54:   while (max_value > MAX_BIN_VALUE)
55:     shift_amount += 1
56:     max_value >>= 1
57:   end while
58:   for i = 0 to NUM_ACKED_BINS
```

```
  59:     acked_bin[i] >>= shift_amount
  60:   end for
  61:   for i = 0 to NUM_SENT_BINS
  62:     sent_bin[i] >>= shift_amount
  63:   end for
  64:   scale_factor += shift_amount
  65:   acked_value >>= shift_amount
  66:   sent_value >>= shift_amount
  67: end if

  68: acked_bin[curr_idx mod NUM_ACKED_BINS] = acked_value
  69: sent_bin[curr_idx mod NUM_SENT_BINS] = sent_value

  // Compute delivered bytes over bins, interpolating a fraction of each
  // bin on the ends (default is 0).
  compute_delv(idx1, idx2):
  70: delv = acked_bin[idx2 mod NUM_ACKED_BINS] -
  71:        acked_bin[idx1 mod NUM_ACKED_BINS]
  72: return delv

  compute_sent(idx1, idx2, frac = 0):
  73: sent = (sent_bin[idx2+1 mod NUM_SENT_BINS] -
  74:         sent_bin[idx1+1 mod NUM_SENT_BINS]) x frac
  75: sent += (sent_bin[idx2 mod NUM_SENT_BINS] -
  76:          sent_bin[idx1 mod NUM_SENT_BINS]) x (1-frac)
  77: return sent

  estimate_target_cwnd():
  78: rtt_bins = ceil(INITIAL_RTT / BIN_DURATION)
  79: cong_idx = curr_idx - rtt_bins
  80: target_cwnd = compute_delv(cong_idx, curr_idx) << scale_factor
  81: target_cwnd = max(target_cwnd, INIT_CWND)
  82: return target_cwnd

  update_cwnd_toward(target_cwnd):
  83: inflight = bytes_in_flight()
  84: drain_acks += bytes_this_ack / MSS
  85: adds = drain_acks / DRAIN_RATE
  86: drain_acks = drain_acks mod DRAIN_RATE
  87: new_cwnd = inflight + adds * MSS
  88: cwnd = max(new_cwnd, target_cwnd)

  // Reset SEARCH parameters.
  reset_search():
  89: curr_idx = -1
  90: scale_factor = 0
  91: bin_end = 0
  92: target_cwnd = 0
```

```
  93: in_drain = false
  94: drain_acks = 0
```

4. SEARCH Parameters

4.1. Window Size (WINDOW_SIZE)

The SEARCH window smooths over RTT fluctuations in a connection that are unrelated to congestion. The window size must be large enough to encapsulate meaningful link variation, yet small in order to allow SEARCH to respond near when slow start reaches link capacity. In order to determine an appropriate window size, we analyzed RTT variation over time for GEO, LEO, and 4G LTE links for TCP during slow start. See [KCL24] for details.

The SEARCH window size should be large enough to capture the observed periodic oscillations in the RTT values. In order to determine the oscillation period, we use a Fast Fourier Transform (FFT) to convert measured RTT values from the time domain to the frequency domain. For GEO satellites, the primary peak is at 0.5 Hz, meaning there is a large, periodic cycle that occurs about every 2 seconds. Given the minimum RTT for a GEO connection of about 600 ms, this means the cycle occurs about every 3.33 RTTs. Thus, a window size of about 3.5 times the minimum RTT should smooth out the latency variation for this type of link.

While the RTT periodicity for the LEO link is not as pronounced as in the GEO link, the FFT still has a dominant peak at 10 Hz, so a period of about 0.1 seconds. With LEO’s minimum RTT of about 30 ms, the period is also about 3.33 RTTs. Thus, a window size of about 3.5 times the minimum RTT should smooth out the latency variation for this type of link, too.

Similarly to the LEO link, the LTE network does not have a strong RTT periodicity. It has a dominant peak at 6 Hz, with a period of about 0.17 seconds. With the minimum RTT of the LTE network about 60 ms, this means a window size of about 2.8 times the minimum RTT is needed. A SEARCH default of 3.5 times the minimum RTT exceeds this, so should smooth out the variance for this type of link as well.

4.2. Threshold (THRESH)

The threshold determines when the difference between the bytes delivered currently and the bytes sent during a previous RTT window is great enough for SEARCH to detect that the congestion point has been reached and to transition to the drain phase. A small threshold is desirable to exit slow start close to the ‘at capacity’ point, but the threshold must be large enough not to trigger an exit from slow

start prematurely due to noise in the measurements.

During slow start, the congestion window doubles each RTT. In ideal conditions and with an initial cwnd of 1, this results in a sequence of delivered bytes that follows a doubling pattern (1, 2, 4, 8, 16, ...). Once the link capacity is reached, the delivered bytes each RTT cannot increase despite cwnd growth.

For example, consider a window that is 4x the size of the RTT. After 5 RTTs, the current delivered window comprises 2, 4, 8, 16, while the previous delivered window is 1, 2, 4, 8. The current delivered bytes is 30, exactly double the bytes delivered in the previous window. Thus, SEARCH would compute the normalized difference as zero.

Once the cwnd ramps up to meet full link capacity, the delivered bytes plateau. Continuing the example, if the link capacity is reached when cwnd is 16, the delivered bytes growth would be 1, 2, 4, 8, 16, 16. The current delivered window is 4+8+16+16 = 44, while the previously delivered window is 2+4+8+16 = 30. Here, the normalized difference between sent bytes (2x the previously delivered) window and the current window is about (60-44)/60 = 0.27. After 5 more RTTs, the previous delivered and current delivered bytes would both be 16 + 16 + 16 + 16 = 64 and the normalized difference would be (128 - 64) / 64 = 0.5.

Thus, the norm values typically range from 0 (before the congestion point) to 0.5 (well after the congestion point) with values between 0 and 0.5 when the congestion point has been reached but not surpassed by the full window.

To generalize this relationship, the theoretical underpinnings of this behavior can be quantified by integrating the area under the congestion window curve for a closed-form equation for both the current delivered bytes (curr_delv) and the previously sent bytes (prev_sent), the normalized difference can be computed based on the RTT round relative to the "at capacity" round. While SEARCH seeks to detect the "at capacity" point as soon as possible after reaching it, it must also avoid premature exit in the case of noise on the link. The 0.26 threshold value chosen does this and can be detected with 1 RTTs of reaching capacity.

4.3. Number of Bins (NUM_BINS)

Dividing the sent and delivered byte histories into bins reduces the server's memory load by aggregating data into manageable segments instead of tracking each packet. SEARCH maintains separate bins for acknowledged (delivered) bytes and sent bytes. The delivered bins cover the most recent window of size W, while the sent bins include additional bins so that the sent window from approximately one RTT earlier can be reconstructed for comparison with the current delivered window. However, more bins provide more fidelity to the actual sent and delivered byte totals and allow SEARCH to make decisions (i.e., compute if it has reached the congestion point) more often, but require more memory for each flow. The sensitivity analysis conducted here aims to identify the impact of the number of bins used by SEARCH and the ability to detect the congestion point in a timely fashion.

Using a window size of 3.5x the initial RTT and a threshold of 0.26, we varied the number of bins from 5 to 40 and observe the impact on SEARCH's performance over GEO, LEO and 4G LTE downloads. For all three links, a bin size of provides nearly identical performance as SEARCH running with more bins, while 10 minimizes early exits while having an at chokepoint percentage that is close to the maximum.

4.4. Drain Phase Parameter (DRAIN_RATE)

During the drain phase, SEARCH gradually converges the congestion window toward the estimated capacity. Rather than adjusting the congestion window abruptly, SEARCH increases the window in controlled increments based on the number of acknowledged segments.

The parameter DRAIN_RATE determines how many acknowledged segments must be observed before an increment to the congestion window is applied. This mechanism prevents abrupt congestion window changes while allowing the sender to smoothly converge toward the estimated capacity. A default value of 3 provides a balance between responsiveness and stability.

4.5. MAX_BIN_VALUE

Based on our analysis (see [KCC25] for details), storing the incoming values in a u32 is not needed - since SEARCH compares previously sent bytes to currently delivered bytes, the _relative_ amounts are all SEARCH really needs. This means fewer bytes - u16 or even u8 - can be used for each bin without sacrificing SEARCH accuracy. In fact, the approach presented -- bit-shifting on demand, only when values get too large - is tunable to different environments by adjusting the MAX_BIN_VALUE constant (line 8), doing so based on the memory needs

and possibly the link capacity of the server. TCP servers that handle only a few connections but are on a high-capacity link may choose to use large bins - u32 or even larger if the kernel uses larger values for TCP - since per-flow memory overhead is not an issue but fidelity to the acked bytes could be. Conversely, TCP servers on resource-constrained devices may use small bins - u8 or even smaller - if the per-memory overhead is critical and the network capacity is not large.

When bit-shifting is required - i.e., the incoming value is too large to fit into the bin - there is some CPU overhead in the shift itself and in the shift for each previously-stored bin. There could be multiple shifts required (i.e., the shifting is done in a loop in lines 58-63, but in practice, there is typically only one shift or at most two.

4.6. Handling Missed Bins (Optional)

For most TCP connections, each bin covers about 1/2 an RTT of time. Thus, most bins have multiple ACKs that arrive before the bin boundary passes. However, in some cases, when an ACK arrives it may be after more than one bin boundary in time. This could be because of intermittent network congestion, delayed end host scheduling, or end hosts without data to send. In such cases, the sender may not observe the expected growth of delivered bytes relative to previously sent bytes, even though this lack of growth is probably not due to congestion on the forward link. As a result, the byte data stored in the bins that is used as the congestion signal may no longer accurately reflect the network state.

An implementation MAY choose to reset the SEARCH state if more than about some RTTs worth of bins are missed. One possible implementation is shown below. In this approach, if the number of skipped bins exceeds a limit derived from the RTT and bin duration, the SEARCH variables are reset using reset_search().

```
MISSED_BIN_LIMIT = alpha x (INITIAL_RTT / BIN_DURATION)
if (passed_bins > MISSED_BIN_LIMIT) then
   reset_search()
end if
```

E.g, a value for alpha is 2, representing approximately two RTTs worth of bins; implementations may tune this value depending on their environment.

4.7. Estimating Sent Bytes from Delivered Bytes (Optional)

Implementations MAY approximate the expected sent bytes based on the observation that during the slow start phase, the delivered bytes approximately double each RTT until the congestion point is reached. In this case, the expected sent bytes can be estimated by doubling the delivered bytes measured approximately one RTT earlier. Implementations that can efficiently maintain a sent-bytes history SHOULD instead use the sent bytes from an earlier RTT. This approach remains applicable even when slow start does not strictly follow a doubling pattern (e.g., due to pacing, ACK thinning, or stack-specific slow-start behaviors). Implementations that cannot efficiently maintain a sent-bytes history MAY use the delivered-byte approximation, which reduces per-flow state but relies on the doubling assumption.

In term of using the delivered-byte approximation, if the sending rate for a TCP flow is limited by the application and not by the congestion window, then the delivery rate may not reflect the actual available network capacity during slow start. In such cases, the delivered bytes may not grow proportionally relative to previously sent bytes, even though the path may not yet be congested. As a result, implementations that rely on the delivered-byte approximation MAY use is_app_limited() as a query to the TCP stack and reset SEARCH state when the flow becomes application-limited.

5. Deployment and Performance Evaluations

Evaluation of hundreds of downloads of SEARCH across GEO, LEO, and 4G LTE network links compared to TCP with HyStart and TCP without HyStart shows SEARCH almost always exits after capacity has been reached but before packet loss has occurred. This results in capacity limits being reached quickly while avoiding inefficiencies caused by lost packets.

Evaluation of a SEARCH implementation in an open source QUIC library (QUICly) over an emulated GEO satellite link validates the implementation, illustrating how SEARCH detects the chokepoint and exits slow start before packet loss occurs. Evaluation over a commercial GEO satellite link shows SEARCH can provide a median improvement of up to 3 seconds (14%) compared to the baseline by limiting cwnd growth when capacity is reached and delaying any packet loss due to congestion.

Details can be found at [KCL24].

6. Implementation Status

This section records the status of known implementations of the algorithm defined by this specification at the time of posting of this Internet-Draft, and is based on a proposal described in [RFC7942]. The description of implementations in this section is intended to assist the IETF in its decision processes in progressing drafts to RFCs. Please note that the listing of any individual implementation here does not imply endorsement by the IETF. Furthermore, no effort has been spent to verify the information presented here that was supplied by IETF contributors. This is not intended as, and must not be construed to be, a catalog of available implementations or their features. Readers are advised to note that other implementations may exist.

According to [RFC7942], "this will allow reviewers and working groups to assign due consideration to documents that have the benefit of running code, which may serve as evidence of valuable experimentation and feedback that have made the implemented protocols more mature. It is up to the individual working groups to use this information as they see fit".

As of the time of writing, the following implementations of SEARCH have been publicly released:

Linux TCP

Source code URL:

https://github.com/Project-Faster/tcp_ss_search.git (https://github.com/Project-Faster/tcp_ss_search.git)

Source: WPI Maturity: production License: GPL? Contact: claypool@cs.wpi.edu Last updated: May 2024

QUIC

Source code URLs:

https://github.com/Project-Faster/quicly/tree/generic-slowstart (https://github.com/Project-Faster/quicly/tree/generic-slowstart) https://github.com/AmberCronin/quicly (https://github.com/AmberCronin/quicly) https://github.com/AmberCronin/qperf (https://github.com/AmberCronin/qperf)

Source: WPI Maturity: production License: BSD-style Contact: claypool@cs.wpi.edu Last updated: May 2024

7. Security Considerations

This proposal makes no changes to the underlying security of transport protocols or congestion control algorithms. SEARCH shares the same security considerations as the existing standard congestion control algorithm [RFC5681].

8. IANA Considerations

This document has no IANA actions. Here we are using that phrase, suggested by [RFC5226], because SEARCH does not modify or extend the wire format of any network protocol, nor does it add new dependencies on assigned numbers. SEARCH involves only a change to the slow start part of the congestion control algorithm of a transport sender, and does not involve changes in the network, the receiver, or any network protocol.

Note to RFC Editor: this section may be removed on publication as an RFC.

9. Acknowledgements

Much of the content of this draft is the result of discussions with the Congestion Control Research Group (CCRG) at WPI https://web.cs.wpi.edu/~claypool/ccrg (https://web.cs.wpi.edu/~claypool/ccrg). In addition, feedback and discussions of early versions of SEARCH with the technical group at Viasat has been invaluable.

10. References

11. References

11.1. 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>.

[RFC5226] Narten, T. and H. Alvestrand, "Guidelines for Writing an IANA Considerations Section in RFCs", RFC 5226, DOI 10.17487/RFC5226, May 2008, <https://www.rfc-editor.org/info/rfc5226>.

[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>.

[RFC7942] Sheffer, Y. and A. Farrel, "Improving Awareness of Running Code: The Implementation Status Section", BCP 205, RFC 7942, DOI 10.17487/RFC7942, July 2016, <https://www.rfc-editor.org/info/rfc7942>.

11.2. Informative References

[HYSTART] 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 , 2008, <https://doi.org/10.1016/j.comnet.2011.01.014>.

[KCC25] Kachooei, M., Chung, J., Cronin, A., Li, F., Peters, B., and M. Claypool, "Reducing Per-flow Memory Use in TCP SEARCH", The IEEE World of Wireless, Mobile and Multimedia Networks conference (WoWMoM), Fort Worth, TX, USA , 2025.

[KCL24] Kachooei, M., Chung, J., Li, F., Peters, B., Chung, J., and M. Claypool, "Improving TCP Slow Start Performance in Wireless Networks with SEARCH", The IEEE World of Wireless, Mobile and Multimedia Networks conference (WoWMoM), Perth, Australia , 2024.

Appendix A. Historical Note

Authors' Addresses

Jae Won Chung
Viasat Inc
300 Nickerson Rd,
Marlborough, MA, 1002
United States of America
Email: jaewon.chung@viasat.com

Maryam Ataei Kachooei
Worcester Polytechnic Institute
100 Institute Rd
Worcester, MA, 01609
United States of America
Email: mataeikachooei@wpi.edu

Feng Li
Viasat Inc
300 Nickerson Rd,
Marlborough, MA, 1002
United States of America

Email: feng.li@viasat.com

Mark Claypool
Worcester Polytechnic Institute
100 Institute Rd
Worcester, MA, 01609
United States of America
Email: claypool@cs.wpi.edu