ccwg
Internet-Draft
Intended status: Standards Track
Expires: 16 March 2026

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

12 September 2025

SEARCH -- a New Slow Start Algorithm for TCP and QUIC
draft-chung-ccwg-search-07

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 expected delivered bytes, smoothed to account for link latency variation and normalized to accommodate link capacities, and exits slow start if the delivered bytes are lower than expected. We implemented SEARCH as a Linux kernel v5.16 module and evaluated it over WiFi, 4G/LTE, and low earth orbit (LEO) and geosynchronous (GEO) satellite links. Analysis of the results show that the 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/.

Internet-Drafts are draft documents valid for a maximum of six months and may be updated, replaced, or obsoleted by other documents at any time. It is inappropriate to use Internet-Drafts as reference material or to cite them other than as "work in progress."

This Internet-Draft will expire on 16 March 2026.

Copyright Notice

Copyright (c) 2025 IETF Trust and the persons identified as the document authors. All rights reserved.

This document is subject to BCP 78 and the IETF Trust's Legal Provisions Relating to IETF Documents (https://trustee.ietf.org/license-info) in effect on the date of publication of this document. Please review these documents carefully, as they describe your rights and restrictions with respect to this document. Code Components extracted from this document must include Revised BSD License text as described in Section 4.e of the Trust Legal Provisions and are provided without warranty as described in the Revised BSD License.

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 monitor the acknowledged delivered bytes in an RTT and compare that to what is expected based on the bytes acknowledged as delivered during the previous RTT. Large differences between delivered bytes and expected delivered bytes is then the indicator that slow start has reached the network congestion point and the slow start phase should exit. We call our approach the Slow start Exit At Right CHokepoint (SEARCH) algorithm. SEARCH is based on the principle that during slow start, the congestion window expands by one maximum segment size (MSS) for each acknowledgment (ACK) received, prompting the transmission of two segments and effectively doubling the sending rate each RTT. However, when the network surpasses the congestion point, the delivery rate does not double as expected, signaling that the slow start phase should exit. Specifically, the current delivered bytes should be twice the delivered bytes one RTT ago. To accommodate links with a wide range in capacities, SEARCH normalizes the difference based on the current delivery rate and since link latencies can vary over time independently of data rates (especially for wireless links), SEARCH smooths the measured delivery rates over several RTTs.

This document describes the current version of the SEARCH algorithm, version 3. 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 Defitions

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: the aggregate (total) of acknowledged delivery bytes over a small time window.

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 in current delivered bytes and previously delivered bytes.

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

THRESH: the norm value above which SEARCH will consider the congestion point to be reached and the slow start phase exits.

3. SEARCH Algorithm

The concept that during the slow start phase, the delivered bytes should double each RTT until the congestion point is reached is core to the SEARCH algorithm. In SEARCH, when the bytes delivered one RTT prior is half the bytes delivered currently, the bitrate is not yet at capacity, whereas when the bytes delivered prior are more than half the bytes delivered currently, the link capacity has been reached and TCP exits slow start.

One challenge in monitoring 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 one RTT prior to those delivered currently in that a lowered latency can make it seem like the total bytes delivered currently is too low compared to the total delivered one RTT ago, making it seem like the link is at the congestion point when it is not.

To counteract link latency variability, SEARCH tracks delivered data over several RTTs in a sliding window providing a more stable basis for comparison. Since tracking individual segment 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 delivered byte totals are kept for a bin-sized time interval instead of for each segment).

3.1. Algorithm Overivew

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

In lines 1-2, upon initialization of a TCP connetion, the SEARCH window (W) is set based on the initial round trip time (init_rtt), and the delivery history (hist) is cleared.

The main part of the SEARCH algorithm runs in lines 3-10 when an ACK is received. Line 3 does bookkeeping, updating the delivery history (hist) based on the current time and the TCP ACK sequency number (sequence_num).

Line 4 computes the number of bytes delivered in the current SEARCH window, using hist to tally the delivered bytes from the current time (now) to now minus W. Line 5 does the same computation, but for the delivered bytes for the previous rtt - i.e,. from now minus an rtt to now minus an rtt minus W.

Line 6 computes the difference between the current delivered bytes and the previous delivered bytes, doubling the latter value since under normal slow start congestion window growth the delivered bytes will approximately double each round-trip time. Line 7 normalizes this difference since it is the relative delivered bytes that matter - i.e., it is the delivered bytes no longer doubling - independently of the actual byte amounts - that indicates the chokepoint has been

reaced. Line 8 compares this normalized difference (norm_diff) to the SEARCH threshold (THRESH), and if this threshold has been surpassed then slow start exits.

SEARCH ALGORITHM OVERVIEW

```
upon TCP connection:
1:  W = 3.5 * init_rtt  // SEARCH window size
2:  hist[] = {}  // Array holding delivery history

on ACK arrived (sequence_num, rtt):

    // Update delivery history.
3:  update_hist(hist, sequence_num)

    // Compute current and previous RTT deliveries.
4:  curr_delv = compute_delv(hist, now - W, now)
5:  prev_delv = compute_delv(hist, now - W - rtt, now - rtt)

    // Check if rate has not doubled.
6:  diff = curr_delv - 2 * prev_delv
7:  norm_diff = diff / (2 * prev_delv)
8:  if (norm_diff >= THRESH) then
9:     exit_slow_start()
10: 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 delivery history information from the TCP ACKs is binned, aggregating 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 delivery window for the previous round-trip time). In addition, the total memory footprint used by the bins is managed via bit shifting, decreasing the delivered byte values stored when they get too large.

The parameters in CAPS (lines 1-7) 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 9-12) are set once, upon establishment of a TCP connection.

The variable *now* on lines 13 and 27 is the current system time when the code is called.

The variable sequence_num and rtt in the ACK_arrived() function (above line 13) are obtained upon arrival of an acknowledgement from the receiver.

The variable *cwnd* on lines 54 and 55 is the current congestion window.

Lines 1-8 set the predefined parameters for the SEARCH algorithm. The window size (WINDOW_SIZE) is 3.5 times the initial RTT. The delivered bytes over a window is approximated using 10 (W) bins, with an additional 15 additional bins (EXTRA_BINS) bins (for a total of 25 (NUM_BINS)) to allow comparison of the current delivered bytes to the previously delivered bytes one RTT earlier. The bin duration (BIN_DURATION) is the window size divided by the number of bins. The threshold (THRESH) is set to 0.35, and is the upper bound of the permissible difference between the previously delivered bytes and the current delivered bytes (normalized) above which slow start exits. The maxium value for each (MAX_BIN_VALUE) can be less than largest TCP sequence number.

Lines 9-12 do one-time initialization of search variables when a TCP connection is established. By setting the bound boundary (bin_end) to 0 intially, that means 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 13 and, if the bin boundary is passed, the bin statistics are updated in the call to update_bins() on line 14.

In update_bins() (lines 27-48), 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 27 computes how many bins have been passed.

On line 28, if more than one RTT of bins has been missed (see the "Missed Bins" section for an explanation), then the SEARCH parameters are reset via reset_search(). The reset_search() function (lines 56-62) re-initializes the bin index (curr_index), the scale factor (scale_factor) and the bin end time (bin_end). On line 59, if there

has been a whole SEARCH window (W) of bins missed, the bin duration (BIN_DURATION) is reset, too, based on the current RTT (lines 60 and 61).

Otherwise, 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 31-33. Line 34 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 35 updates the next bin boundary (bin_end) based on the number of passed bins (passed_bins) and the bin duration (BIN_DURATION)

In lines 36-47, the memory used by each bin can be reduece (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 line 36 the sequence number is first scaled by the scale factor (initially set to 0). If the scaled value is larger than the maximum value the bin can hold (MAX_BIN_VALUE, set to the largest u16 by default), then lines 39-42 shift (scale) the value (bin_value) until it fits. In lines 43-45, all the previous 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 (scale_factor) updated in line 46.

Lastly, on line 48, the latest bin is updated to the most recent scaled bin value - i.e., the latest ACKed TCP sequence number (sequence_num), scaled/shifted by shift_amount.

Once the bins are updated, lines 15-17 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 bins there.

If there are enough bins to run SEARCH, lines 18 and 20 compute the current and previously delivered bytes over a window (W) of bins. This delivered bytes over the window is computed in the function compute_delv(). For previously delivered bytes, shifting by an RTT may land between bin boundaries, so the computation is interpolated by the fraction on either side, computed on line 18.

Lines 49-51 compute the delivered bytes over the bins, first by taking the "upper" delivered window (which is 0, if fraction is 0), then adding the "lower" delivered window, and finally returning this sum.

Once computed, the difference between expected delivered bytes (2 x prev_delv) and the current delivered bytes (curr_delv) is normalized (line 21) and then compared to the threshold (THRESH) in line 21. If this difference is larger than THRESH, slow start exits, which is handled by the function exit_slow_start().

The exit_slow_start() function is on lines 52-55. When exiting slow start at the chokepoint, SEARCH is delayed in its prediction by almost exactly two RTTs. SEARCH can compute exactly how many extra bytes have been added to the congestion window (cwnd) beyond this point. The cwnd is reduced by this amount, before setting ssthresh to the cwnd (line 55). This exits slow start.

SEARCH 3.1 ALGORITHM

```
Parameters:
1: WINDOW_SIZE = INITIAL_RTT x 3.5
2: W = 10
3: EXTRA_BINS = 15
4: NUM_BINS = W + EXTRA_BINS
5: BIN_DURATION = WINDOW_SIZE / W
6: THRESH = 0.35
7: MAX_BIN_VALUE = 0xFFFF  // 16-bit
8: MISSED_BIN_LIMIT = alpha x (INITIAL_RTT / BIN_DURATION)

Initialization():
9: bin[NUM_BINS] = {}
10: curr_idx = -1
11: bin_end = 0
12: scale_factor = 0

ACK_arrived(sequence_num, rtt):
    // Check if passed bin boundary.
13: if (*now* > bin_end) then
14:   update_bins()

      // Check if enough data for SEARCH.
15:   prev_idx = curr_idx - (rtt / BIN_DURATION)
16:   if (prev_idx > W) and
17:      (curr_idx - prev_idx) < EXTRA_BINS then

        // Run SEARCH check.
18:     curr_delv = compute_delv(curr_idx - W, curr_idx)
19:     frac = (rtt mod BIN_DURATION) / BIN_DURATION
20:     prev_delv = compute_delv(prev_idx - W, prev_idx, frac)
21:     norm_diff = (2 x prev_delv - curr_delv) / (2 x prev_delv)
22:     if (norm_diff >= THRESH) then
23:       exit_slow_start()
```

```
24:     end if

25:   end if // Enough data for SEARCH.

26: 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():
27: passed_bins = (*now* - bin_end) / BIN_DURATION + 1

    // Missed too many bins, or app limit rate --> reset SEARCH
28: if (passed_bins > MISSED_BIN_LIMIT || is_app_limited()) then
29:   return reset_search()
30: end if

    // For remaining skipped, propagate prev bin value.
31: for i = curr_idx+1 to (curr_idx + passed_bins)
32:   bin[i mod NUM_BINS] = bin[curr_idx]
33: end for
34: curr_idx += passed_bins
35: bin_end += passed_bins x BIN_DURATION

    // Scale bins (shift) if too large.
36: bin_value = sequence_num >> scale_factor
37: if (bin_value > MAX_BIN_VALUE) then
38:   shift_amount = 0
39:   while (bin_value > MAX_BIN_VALUE)
40:     shift_amount += 1
41:     bin_value >>= 1
42:   end while
43:   for i = 0 to NUM_BINS
44:     bin[i] >>= shift_amount
45:   end for
46:   scale_factor += shift_amount
47: end if

48: bin[curr_idx mod NUM_BINS] = bin_value

// Compute delivered bytes over bins, interpolating a fraction of each
// bin on the ends (default is 0).
compute_delv(idx1, idx2, frac = 0):
49: delv = (bin[idx2+1 mod NUM_BINS] - bin[idx1+1 mod NUM_BINS]) x frac
50: delv += (bin[idx2 mod NUM_BINS] - bin[idx1 mod NUM_BINS]) x (1-frac)
51: return delv

// Exit slow start by setting cwnd and ssthresh.
```

```
exit_slow_start():
52: cong_idx = curr_idx - 2 x INITIAL_RTT / BIN_DURATION
53: overshoot = compute_delv(cong_idx, curr_idx)
54: cwnd -= overshoot
55: ssthresh = cwnd

// Reset SEARCH parameters.
reset_search():
56: curr_idx = -1
57: scale_factor = 0
58: bin_end = *now*
59: if passed_bins > W then
60:   WINDOW_SIZE = rtt x 3.5
61:   BIN_DURATION = WINDOW_SIZE / W
62: end
```

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 delivered during the previous RTT is great enough to exit the slow start 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 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 delivered bytes (prev_delv), 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 0.35 threshold value chosen does this and can be detected with 2 RTTs of reaching capacity.

4.3. Number of Bins (NUM_BINS)

Dividing the delivered byte window into bins reduces the server’s memory load by aggregating data into manageable segments instead of tracking each packet. This approach simplifies data handling and minimizes the frequency of window updates, enhancing server efficiency. However, more bins provide more fidelity to actual delivered byte totals and allow SEARCH to make decisions (i.e., compute if it should exit slow start) 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 exit slow start in a timely fashion.

Using a window size of 3.5x the initial RTT and a threshold of 0.35, 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. Missed Bins (MISSED_BIN_LIMIT)

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 won’t get confirmation of the expected doubling of the delivered bytes each RTT even though this lack of doubling is probably not due to congestion on the forward link. So, in this case, SEARCH does not exit slow start. However, the delivered byte data in the bins that is used as the congestion signal (i.e., the lack of doubling) is no longer clean and so the SEARCH variables are reset (in reset_search()). By default, SEARCH does this if more than about two RTTs of bins are missed (Line 8 sets the MISSED_BIN_LIMIT), representing not receiving ACKs for a full round-trip time multiplied by an alpha factor - the current recommendation is for an alpha of 2, but this can be tuned as needed.

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 delivered 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 7), 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 39-42, but in practice, there is typically only one shift or at most two.

4.6. App Limited (is_app_limited())

If the sending rate for TCP flow is limited by the application and not by the congestion window, then the delivery rate will not double each RTT during slow start. SEARCH (and any congestion-point detection algorithm) will not be able to tell when the chokepoint has been reached. Line 28 detects if the flow is application-limited via is_app_limited() that is intended to be a query to the TCP stack. When it is, SEARCH resets its parameters.

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] and [CKC24].

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

[CKC24] Cronin, A., Kachooei, M., Chung, J., Li, F., Peters, B., and M. Claypool, "Improving QUIC Slow Start Behavior in Wireless Networks with SEARCH", Proceedings of the IEEE International Symposium on Local and Metropolitan Area Networks (LANMAN), Boston, MA, USA , 2024, <https://web.cs.wpi.edu/~claypool/papers/quic-search-lanman-24/paper.pdf>.

[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