Implementation of the SEARCH Slow Start Algorithm in the Linux Kernel

Maryam Ataei Kachooei[†], Joshua Chung[+], Amber Cronin[†], Benjamin Peters[*], Feng Li[*], Jae Won Chung[*], and Mark Claypool[†]

[†] Worcester Polytechnic Institute, Worcester, MA, USA
[+] Lexington Christian Academy, Lexington, MA, USA
[*] Viasat Inc., Marlboro, MA, USA

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 [4] – can cause premature exit from slow start, especially over wireless links, degrading link utilization. However, without HyStart, TCP exits too late, causing unnecessary packet loss. To improve TCP performance during slow start, we developed the Slow start Exit At Right CHokepoint (SEARCH) algorithm [8] where the congestion point is determined based on bytes delivered compared to the expected bytes delivered, smoothed to account for link latency variation and normalized to accommodate link capacities. In prior work, we implemented SEARCH and evaluated it over 4G LTE, low earth orbit (LEO), and geosynchronous (GEO) satellite links. In this paper, we implemented search as a Linux kernel v5.16 module, illustrate its performance over GEO satellite links by example, and evaluate SEARCH over Wi-Fi. Over all networks, SEARCH reliably exits from slow after the congestion point is reached but before inducing packet loss. Our Linux kernel module is open-source and available for general use and further evaluation.

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 [4] 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, the TCP sender can 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 are then the indicator that the 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.[1] 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 paper describes the current version of the SEARCH algorithm, version 2. Active work on the SEARCH algorithm is continuing. The paper is organized as follows: The Related Work section provides background and related work of this research. The SEARCH section describes the SEARCH algorithm in detail. The subsequent sections summarize our preliminary test results over production GEO satellite networks and WiFi environments. The final section presents our conclusions and potential future work.

Related Work

Jasim et al. [1] propose a technique for approximating the TCP congestion window thresholds for high latency connections by using a packet-pair technique to estimate bandwidth. The authors perform experiments using a variety of network configurations and find that their approach can improve the performance of TCP when latencies are high. Additionally, they compare their approach to other existing methods and find it outperforms them in terms of both efficiency and accuracy.

Ye et al. [11] propose a new algorithm, Personalized FAST TCP, which improves the performance of the FAST TCP con-

[1] Project page: `https://search-ss.wpi.edu/`

gestion control for personalized healthcare systems. The algorithm uses the number of remaining link buffers to judge exit times for slow start, designed to help the queuing delay ratio stays within the expected range and not change with variations in bandwidth or other parameters, leading to faster convergence of the system. They also propose a method for dynamically adjusting the gain parameters of the controller based on local information obtained from each connection source, allowing the system to adjust to changes in network operation states within the range of relevant protocol parameters to maintain stability. With this method, they find the FAST TCP system achieves a small queuing delay and quickly converges to the equilibrium point.

Gál, et al [3] introduce BIC (Binary Increase Congestion Control) and Hybla as solutions to improve TCP's performance in high-latency networks. Hybla modifies the slow start and congestion avoidance phases of New Reno to reduce the dependency on RTT, achieving RTT fairness but with amplified reactions in higher RTT flows. BIC aims to approximate the optimal congestion window size using a binary search approach, though it may exhibit RTT fairness issues comparable to Reno's. Unlike these loss-based algorithms, delay-based algorithms preemptively adjust to network conditions.

Kachooei et al [6] present the BEST algorithm, a bandwidth estimation technique based on packet-pair measurements for determining the slow start exit point. While showing promise, BEST encounters challenges in environments with high variability in estimated bandwidth and RTT, leading to underestimation or overestimation of the available bandwidth.

Li et al. [10] present Fast Bandwidth Estimation (FBE), a solution designed to optimize the slow start phase in high bandwidth networks like WiFi 6 and 5G. Recognizing the issue of link underutilization due to the slow ramp-up of congestion windows, FBE utilizes the initial ACK feedback to estimate link capacity without sending extra probe packets. By refining ACK intervals to reflect true link rates and dynamically adjusting the congestion window based on driver queue feedback, FBE attempts to address the inaccuracies in bandwidth estimation that are especially prevalent in variable wireless links. Comparative experiments show FBE outperforms traditional slow start methods, cutting down convergence time by more than half and improving short flow completion times by up to 44% against well-known algorithms such as CUBIC and BBR.

Kasoro et al. [9] propose ABCSS, a method that combines the strengths of Appropriate Byte Counting (ABC) and the Slow Start (SS) algorithm. This approach is designed to increase the congestion window more effectively than the traditional slow start, aiming to address the invariant congestion window during initial round trips that lead to TCP burst issues and potential buffer overflows.

Huang and Olson propose an IETF draft named Hystart++ [5] as an alternative to HyStart [4], and is currently implemented in the Microsoft Windows TCP Cubic stack. HyStart++ removes the ACK train method used by HyStart and only uses RTT sampling to decide when to exit slow start. HyStart++ adds a Limited Slow Start (LSS) phase after exiting slow start where the congestion window grows slower than during legacy slow start but faster than during congestion avoidance. Thus, instead of entering congestion avoidance directly, slow start exits to LSS and then exits LSS to congestion avoidance only when encountering packet loss. Thus, HyStart++ likely exits slow start prematurely for the same scenarios that HyStart does, but may not limit throughput growth as much and may reduce packet loss compared to legacy slow start.

While the above approaches are all alternatives, and in some sense improvements, to traditional slow start, unlike SEARCH they do not focus on the fundamental problem of avoiding an early exit to slow start before the link capacity is reached while still exiting before inducing unnecessary congestion.

Our previous work on SEARCH first explored the algorithm (version 1) over GEO and LEO satellite networks [7]. Experiments used an instrumented kernel and ran the algorithm only after the fact via Python script, and found SEARCH reliably exits slow start before packet loss but after congestion. More extensive development (version 2) evaluates SEARCH over GEO, LEO, and 4G LTE networks [8], showing search improves performance compared to TCP with and without HyStart enabled, and our SEARCH implementation for QUIC [2] shows the approach effective beyond the TCP stack. This paper continues to contribute to SEARCH with a Linux kernel implementation and preliminary evaluation over WiFi.

SEARCH

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 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).

Algorithm 1 SEARCH 2.0 Algorithm for Linux

```
 Parameters
 1:  WINDOW_FACTOR = 3.5
 2:  W = 10
 3:  EXTRA_BINS = 15
 4:  NUM_BINS = W + EXTRA_BINS
 5:  THRESH = 0.35

// Set SEARCH variables for new TCP connection.
initialization( initial_rtt ):
 6:  window_size = initial_rtt × WINDOW_FACTOR
 7:  bin_duration = window_size / W
 8:  bin[NUM_BINS] = {}
 9:  curr_idx = -1
10:  prev_seq_num = 0
11:  bin_end = now + bin_duration

// Update for each ack that arrives.
ack_arrival( seq_num, rtt ):
12: if (now > bin_end) then  // Passed bin boundary?
13:       update_bins ()

          // Enough data for SEARCH?
14:       prev_idx = curr_idx - (rtt / bin_duration)
15:       if  (prev_idx ≥ W and
16:           (curr_idx - prev_idx) ≤ EXTRA_BINS) then

              // Run SEARCH check.
17:           curr_delv = sum_bins(curr_idx - W, curr_idx)
18:           f = (rtt % bin_duration) / bin_duration
19:           prev_delv = sum_bins(prev_idx - W, prev_idx, f)
20:           norm_diff = (2 · prev_delv − curr_delv) / (2 · prev_delv)
21:           if (norm_diff ≥ THRESH) then
22:               sthresh = cwnd
23:           end if

24:       end if // Enough data for SEARCH.
25: end if // Passed bin boundary.

// Update bins - more than one might have passed.
update_bins( seq_num ):
26:  passed_bins = (now − bin_end) / BIN_DURATION + 1
27:  bin_end += passed_bins × bin_duration
28: for i = curr_idx + 1 to curr_idx + passed_bins do
29:       bin[i % NUM_BINS] = 0
30: end for
31:  curr_idx += passed_bins
32:  bin[curr_idx % NUM_BINS] = seq_num - prev_seq_num
33:  prev_seq_num = seq_num

// Add up bins, interpolating fraction end bins (default is 0).
sum_bins( idx1, idx2, fraction = 0 ):
34:  sum = 0
35: for i = idx1 + 1 to idx2 - 1 do
36:       sum += bin[i  %  NUM_BINS]
37: end for
38:  sum += bin[idx1] × fraction
39:  sum += bin[idx2] × (1 - fraction)
40:  return sum
```

The SEARCH algorithm for Linux (that runs on the TCP sender only) is shown in Algorithm 1.[2]

The parameters in CAPS (Lines 1 - 5) are constants.

The variables in initialization() (Lines 6 - 11) are set to their intial values upon establishment of a TCP connection. The initial_rtt (Line 6) is obtained via the first round-trip time measured in the TCP connection.

The variable **now** on Lines 11, 12 and 26 is the current system time when the code is called.

The variable `seq_num` and `rtt` above Line 12 are obtained upon arrival of an acknowledgement from the receiver.

The variable **cwnd** on Line 22 is the current congestion window.

Lines 1 - 5 set the predefined parameters for the SEARCH algorithm. The window factor (`WINDOW_FACTOR`) is 3.5 , which is used to define the window size in Line 6. Ten bins (`W`) are used to approximate the delivered rates over the window size. There are an additional 15 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.

After initialization, SEARCH only acts when acknowledgments (ACKS) are received and even then, only when the current time (**now**) has passed the end of the latest bin boundary (stored in the variable `bin_end`).

In function `update_bins()` (Lines 26 to 33), Line 26 computes how many bins have passed. Under most TCP connections, the time (**now**) is in the successive bin, but in some cases (such as during an RTT spike or a TCP connection without data to send), more than one bin boundary may have been passed. In Lines 28-30, for each bin passed, the `bin[]` is set to 0. For the latest bin, the delivered bytes is updated by taking the latest sequence number (from the ACK) and subtracting the previously recorded sequence number in the last bin boundary (Line 32). In Line 33, the current sequence number is stored into `prev_seq_num`) for computing the delivered bytes the next time when a bin boundary is passed.

Once the bins are updated, Lines 14 - 16 check if enough bins have been filled to run SEARCH. This requires at least `W` (10) bins (i.e., on SEARCH window worth of delivered data), but also enough bins to shift back by an RTT to compute a window (10) bins one RTT ago, too.

If there is enough bin data to run SEARCH, Lines 17 and 19 compute the current and previously delivered bytes over a window (`W`) of bins, respectively. This sum is computed in the function `sum_bins()` (Lines 34 - 40). For previously delivered bytes, shifting by an RTT may mean the SEARCH window lands between bin boundaries, so the sum is interpolated by the fraction of each of the end bins.

[2]The code for SEARCH 2.0 kernel module is available at: `https://github.com/Project-Faster/tcp_ss_search/tree/main/src`

In the function `sum_bins()`, `idx1` and `idx2` are the indices into the `bin[]` array for the start and end of the bin summation and, as explained above, fraction is the proportion (from 0 to 1) of the end bins to use in the summation. In Lines 35 - 37, the summation loops through the `bin[]` array for the middle bins, modulo the number of bins allocated (`NUM_BINS`) and then adds the fractions of the end bins in Lines 38 and 39.

Once bin sums are tallied, Line 20 calculates the difference between the expected delivered bytes (`2 * prev_delv`) and the current delivered bytes (`curr_delv`), normalized by dividing by the expected delivered bytes. In Line 21, this normalized difference (`norm_diff`) is compared to the threshold (`THRESH`). If `norm_diff` is larger than `THRESH`, that means the current delivered bytes is lower than expected (i.e., the delivered bytes did not double over the previous RTT) and slow start exits. Slow start exit is acheived by setting the slow start threshold (`ssthresh`) to the congestion window in Line 22.

Parameter Selection

This section provides justification and some sensitivity analysis for key SEARCH algorithm constants. Details can be found in our previous paper [8].

Window Factor (`WINDOW_FACTOR`) 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.

The SEARCH window size needs to 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 RTT 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 LEO links is not as pronounced as they are in GEO links, the analysis yields a similar window size. The FFT of LEO RTTs 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 about 3.33 RTTs, similar to that for GEO. 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. The FFT of LTE RTTs has a dominant peak at 6 Hz, with a period of about 0.17 seconds. With the minimum RTT of the LTE network of 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 it should smooth out the variance for this type of link as well.

For WiFi, the FFT of RTTs has a dominant peak at 67 Hz, meaning the periodic cycle occurs about every 0.015 seconds. Given the minimum initial RTT for our WiFi connections is about 4 ms, this yields a window factor is 3.75. This supports using a window factor of 3.5 as a balance across the factors measured so far – 3.33 for GEO and LEO, 2.8 for LTE, and 3.75 for WiFi.

Threshold (`THRESH`) The threshold (`THRESH`) determines when the difference between the bytes delivered currently and the bytes delivered during the previous RTT is large enough to exit the slow start phase. A small threshold is desirable to exit slow start close to the 'at capacity' point (i.e., without overshooting too much), 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 (1 is used as an example, but typical congestion windows start at 10 or more), 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 delivered 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 about 2 RTTs after reaching capacity.

Number of Bins (`NUM_BINS`) Dividing the delivered byte window into bins reduces the sender'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 sender 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 previously conducted aimed 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 observed the impact on SEARCH's performance over GEO, LEO and 4G LTE downloads. For all three link types, a bin size 10 of provides nearly identical performance as SEARCH running with more bins, while 10 also minimizes early exits from slow start while having an "at chokepoint" percentage that is close to the maximum.

Previous Performance Evaluation

Evaluation of hundreds of downloads of TCP with 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 [8]. 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 congestion point and exits slow start before packet loss occurs [2]. 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.

Evaluation over GEO Satellite by Example

This section primarily shows and example of our Linux implementation of SEARCH, describes our measurement testbed, and comparing TCP Cubic with and without HyStart to TCP Cubic with SEARCH over a GEO satellite network.

Measurement Setup

We create a measurement testbed to evaluate SEARCH where the client utilizes a Viasat GEO satellite as the "last mile" of connectivity, downloading from an Internet server, which mirrors common satellite user scenarios.

Figure 1 depicts our experiments setup. The client is a PC with an Intel i7-5820K CPU @ 3.30GHz and 32GB RAM running with Ubuntu 20.04 with 5.4.0 kernel. The client connected with a Viasat-2 small beam (beam# 738 11338) through a Viasat terminal. The server is an 32GB AWS EC2 instance running Ubuntu 22.04 with 5.13.12 kernel, customized to support the SEARCH module and experiments.

The Viasat gateway performs per-client queue management, where the queue for each client can grow up to 36 MBytes. Queue lengths are controlled at the gateway by

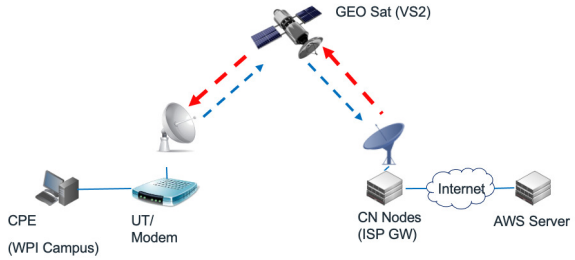


Figure 1: GEO satellite measurement testbed.

Active Queue Management that randomly drops 25% of incoming packets when the queue is over half of the limit (i.e., 18 MBytes). The GEO performance-enhancing proxy (PEP or VWA) is deliberately deactivated to simulate conditions where encryption or other constraints preclude PEP usage.

Both the client and server are instrumented to run *tcpdump* for post-experiment analysis. The experiments use TCP Cubic with HyStart enabled (the Linux default), TCP Cubic with HyStart disabled, and TCP Cubic with SEARCH.

Results

All examples are collected during local "peak" busy hours.

TCP Cubic with HyStart Enabled Figure 2(a-d) illustrates the behavior of TCP CUBIC with HyStart over a GEO satellite link during peak busy hours. Figure 2(a) shows the congestion window size[3] versus time. Because HyStart is designed for wired links with low RTT variance, TCP exits from slow start prematurely around the 5th second and has less than exponential growth until the Cubic function in congestion avoidance kicks in. The RTT has oscillation normal for a GEO link over time (Figure 2(b)) with low retransmissions computed every 10 ms (Figure 2(d)) but also low throughput and goodput (Figure 2(c)) until about 30 seconds into the run.

TCP Cubic with HyStart Disabled Figures 2(e-h) shows an example of TCP Cubic with HyStart disabled over a GEO satellite link during peak busy hours. In Figure 2(e) shows, the "bytes in flight" (or congestion window size) exponentially increases during start up after disabling HyStart – the "bytes in flight" reach 70 MB in less than 15 seconds. Meanwhile, the goodput and throughput reach more than 100 Mbps in about the same time period (Figure 2(g)). However, such large amount of data (70 MBytes) in flight causes the Viasat queue to fill, with RTTs spiking at over 5 seconds (Figure 2(f)) before causing considerable packet loss (Figure 2(g)). Since all ACKs are duplicated ACKs between the 16th and 20th seconds, the sender could not send new data (the flat curve after slow start in Figure 2(e)) but has to do "fast retransmission" to re-transmitted packets (the spikes in Figure 2(h)). Note, because of the shared nature of satellite links, the large queue buildup would impact other users traffic of the same class that share the beam.

[3] In this report, the congestion window size is interchangeable with "bytes in flight".

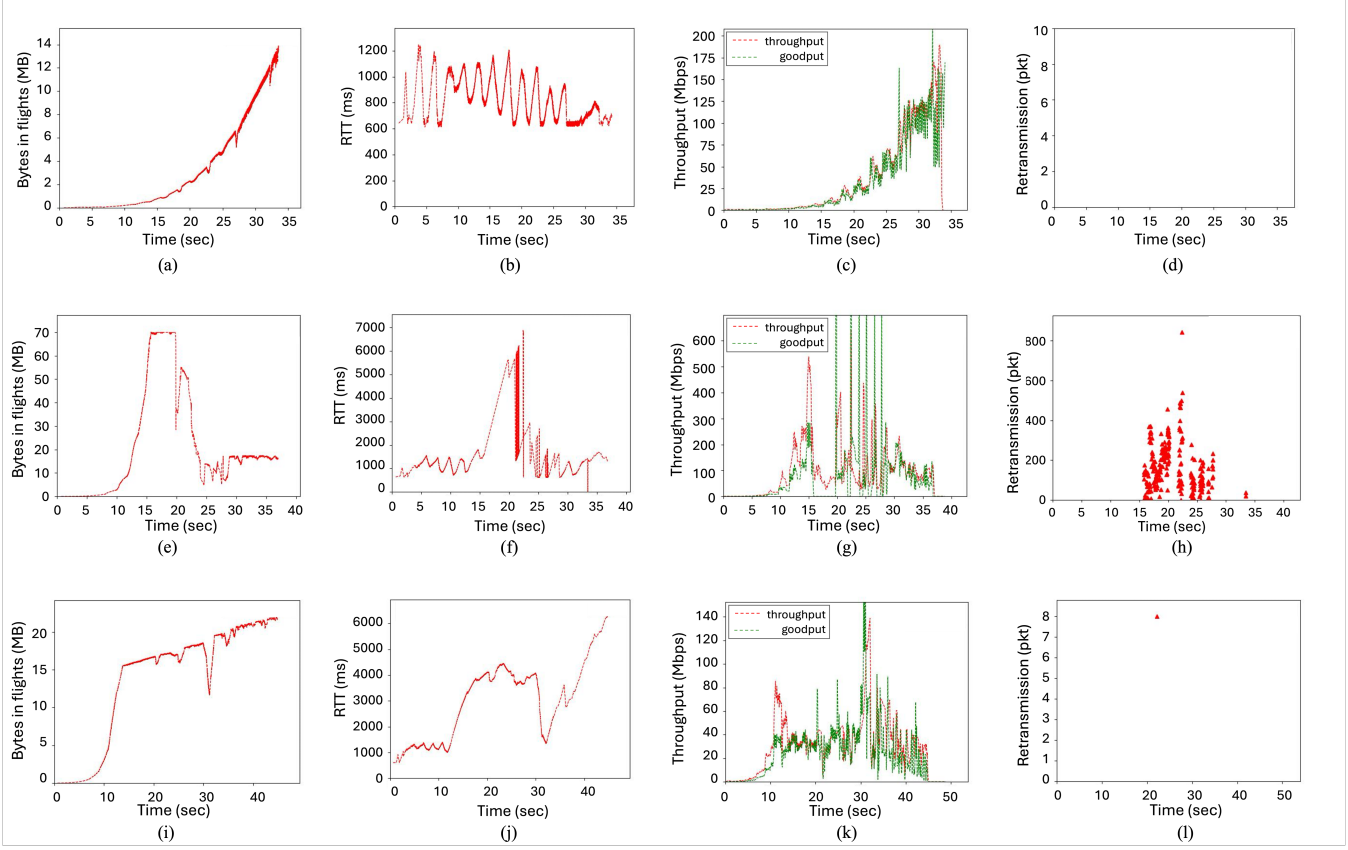


Figure 2: TCP performance over a GEO link during peak busy hours. (a-d): TCP Cubic w/HyStart enabled, (e-h): TCP Cubic w/HyStart disabled, (i-l): TCP Cubic w/SEARCH.

TCP Cubic with SEARCH Figures 2(i-l) shows an example of TCP Cubic with SEARCH over a GEO satellite link during peak busy hours. Like TCP Cubic with HyStart disabled, TCP Search's congestion window (CWND) grows fast and increases exponentially (Figure 2(i)). Unlike TCP Cubic with HyStart disabled, the CWND growth stops being exponential around the 12th second because SEARCH detects that the link reaches its congestion point and exits slow start. Since TCP exits slow start near capacity, the RTT still increases during CWND growth in congestion avoidance, albeit less dramatically than the RTT growth during slow start with HyStart disabled. TCP Cubic with SEARCH has far fewer retransmitted packets (Figure 2(l)) than TCP Cubic with HyStart disabled.

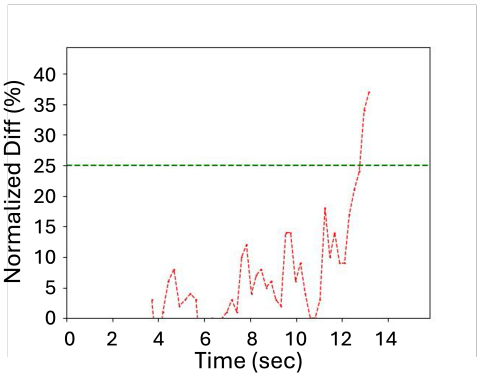
We instrumented the TCP search module to report the kernel-level socket statistics. Figure 3(b) shows the congestion window (`cwnd`), slow start threshold (`ssthresh`) and corresponding TCP congestion states for a TCP Cubic download with SEARCH. Different from Figure 2(i) in which "bytes in flight" is inferred via the pcap captured by `wireshark`, `cwnd` shown in Figure 3(a) is from the `struct tcp_sock`, although curves approximately match. Figure 3(a) also shows the slow start threshold (`ssthresh`) reported and set by SEARCH (Line 22 around 12 seconds.

The green dots in Figure 3(a) show the TCP states (`CA State`) of the sever socket. Note, around 22 seconds, TCP enters a transient "Disorder" state, because the sender receives duplicated ACKs caused by out-of-order delivery but without a negative impact on the `cwnd` growth- such transient out-of-order deliveries are quickly recovered from and not as harmful as packet loss by a congested router.

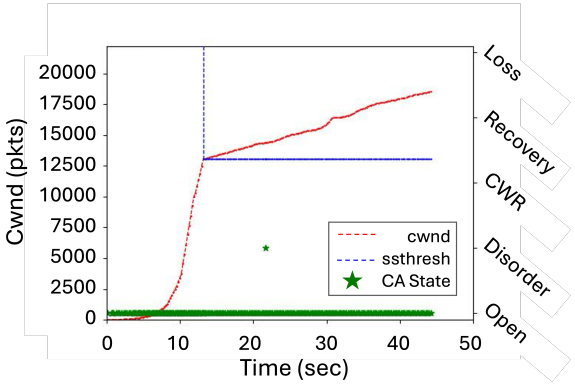
Figure 3(b) plots the normalized difference (as a percent) computed by the module (Line 20 in Algorithm 1) with the horizontal line showing the threshold (set to 25% or a THRESH of 0.25 although the SEARCH default is 0.35). TCP with SEARCH exits slow start when the threshold is surpassed (Line 20 in Algorithm 1).

Comparative Download Performance This section compares download performance over time for SEARCH versus TCP with HyStart enabled and disabled.

Figure 4(a) shows the time to download a given size file with TCP Cubic with SEARCH, TCP Cubic with HyStart, and TCP CUBIC without HyStart. The x-axis is the size of a file and the y-axis is the time to transfer. The lines are mean values with the shading showing 95% confidence intervals around the mean. From the figure, it takes significantly longer (about 2x over this range) for Cubic with HyStart enabled to download the same size of file compared to TCP Cubic without HyStart or TCP Cubic with SEARCH. TCP Cubic with

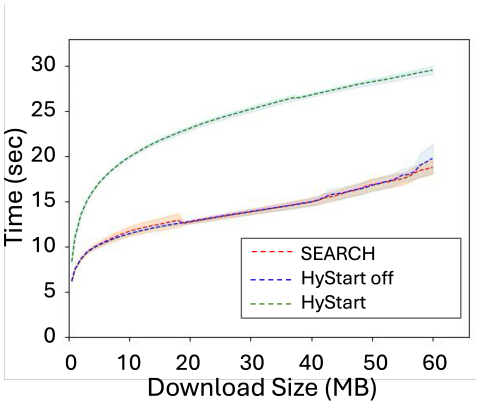


(a) Normalized Diff

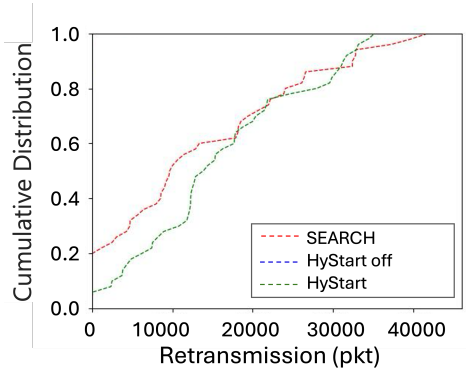


(b) Bytes in Flight

Figure 3: TCP Cubic Search Example



(a) Time vs. file size downloaded



(b) Retransmission distributions

Figure 4: Comparative TCP Download Performance

SEARCH and TCP Cubic with HyStart disabled have similar performance – e.g., both deliver a 60 MB file in around 17 seconds.

Figure 4(b) shows the cumulative distribution functions (CDFs) of retransmitted packets for a 7 seconds window after the first packet loss. Because TCP Cubic with HyStart exits its slow start prematurely and underutilizes the link capacity, there is almost no packet loss (curve not visible). For TCP Cubic without HyStart and TCP with SEARCH, generally, SEARCH has fewer retranmitted packets when loss happens - nearly 20% of the SEARCH flows have no loss compared to only 5% for HyStart disabled flows and the SEARCH flows have a median of about 9600 retransmitted packets compared to about 13800 for HyStart disabled.

Evaluation over WiFi

This section evaluates the performance of our Linux implementation of SEARCH over WiFi, comparing TCP Cubic with and without HyStart and TCP BBR (version 1.0) to TCP Cubic with SEARCH over 24 hours of downloads in two different WiFi conditions.

Measurement Setup

Figure 5 depicts our Wi-Fi testbed on the WPI campus. The campus, spanning 95 acres in an urban setting, has wireless coverage across virtually every building. The wireless infrastructure employs a controller drop-off design, routing all user traffic through one of six campus controllers via encrypted tunnels. Each Access Point (AP) is individually wired to the network, avoiding a mesh network setup.

To manage network traffic, users connecting through guest or open authentication methods face a bitrate cap to prevent them from negatively impacting critical academic or research activities. The network employs typical QoS measures, prioritizing voice, video, and control traffic based on traffic type classification.

For security, the network incorporates distributed denial-of-service protection, limiting packets per second for various protocols and blocking sessions that appear malicious. There is considerable, legitimate on campus use that competes with our experiments, and the urban environment of the campus can also introduce competing wireless signals and potential signal degradation.

Performance measurements were conducted using a fixed, dedicated on-campus server and a mobile laptop client for bulk TCP downloads, with different TCP configurations controlled at the server. The server, a PC with an Intel i5-8500

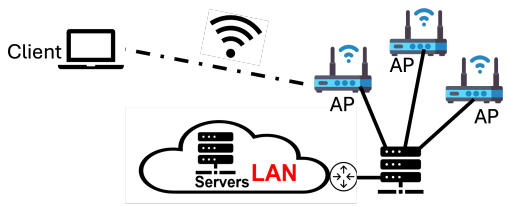


Figure 5: WiFi measurement testbed.

CPU @ 3GHz and 8 GB RAM, runs Mint-20.3 with Linux kernel version 5.10.79. The client is an Intel Core Ultra 7 155U×14 CPU and 16 GB of RAM running Ubuntu version 22.04 and Linux kernel version 6.8.0.

During our experiments, we evaluated the performance of TCP Cubic with the SEARCH algorithm, compiled as a module and loaded into the server kernel, and compared it with TCP configurations including TCP Cubic with HyStart enabled, TCP Cubic with HyStart disabled, and TCP BBR (version 1.0). These comparisons were conducted using the iperf3 tool for multiple download sessions.

The client initiated the download sessions with the server sequentially using one of the following configurations: 1) HyStart enabled, 2) HyStart disabled, 3) BBR, or 4) SEARCH. Each download session lasted 3 seconds, followed by a 7-second pause. After completing one download with each configuration, the client paused for about 6 minutes to allow system stabilization before repeating the process. This cycle was repeated 200 times during 24 hours, resulting in a total of 800 downloads per session. Each session was conducted at two different Received Signal Strength Indicator (RSSI) locations: strong and weak.

For Wi-Fi connections, the RSSI measures the strength of the received radio signal from the access point, serving as a crucial indicator of wireless connection quality. Higher RSSI values generally indicate stronger signal reception and better network performance, whereas lower RSSI values can signal weaker reception and potential connectivity issues. To find two different locations with suitably-different signal strenghts, we assessed Wi-Fi performance across various locations within our campus Wi-Fi environment. Figure 6 illustrates the distribution of RSSI values across different locations on our campus map. Each pin represents a site where we took an RSSI measurement as reported by the laptop's Wi-Fi driver and then did a 3 second TCP download with iperf3. Locations with strong signal strength are marked in green, while those with weak signal strength are marked n red. This visual representation of RSSI values enables us to identify areas with strong and weak signal reception, providing insights into the wireless coverage and performance across the campus.

We analyzed the relationship between RSSI and download rates to better understand how RSSI levels correspond to network performance. Figure 7 shows the results, where the x-axis is the RSSI measured by the client and the y-axis is the throughput for a 3-second TCP download using the default Linux TCP settings. There is a general correlation between higher RSSI values and increased throughput. However, at

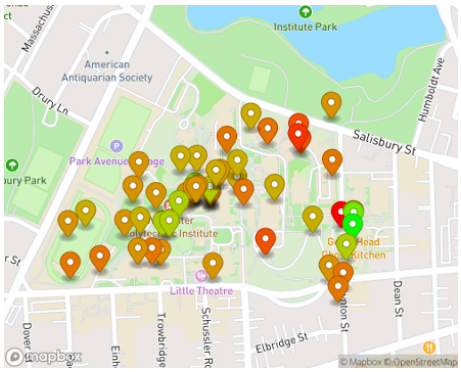


Figure 6: RSSI on WPI campus.

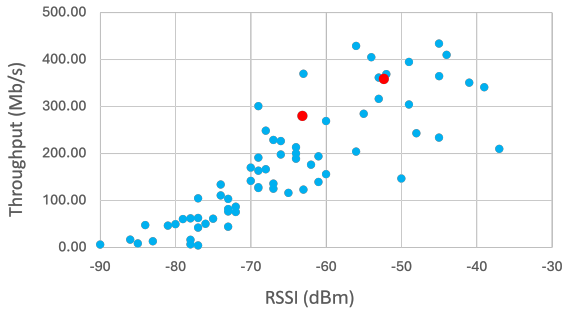


Figure 7: Relationship between throughput and Wi-Fi RSSI across WPI

higher RSSI values the throughput shows significant variability likely due to the access point being shared with devices that have lower RSSI levels.

We identified two key locations on our campus for a closer examination, marked with red points in Figure 7: one with strong signal strength (-53dBm) and one with weak signal strength (-64dBm). These two locations provide a basis for evaluating the performance of TCP with the SEARCH algorithm in comparison to TCP with HyStart enabled, HyStart disabled, and BBR (version 1.0).

Figure 8 displays the CDF of median throughput values aggregated from 200 cases per location over 24 hours, comparing selected strong and weak RSSI locations. The x-axis denotes throughput in Mbps, while the y-axis shows the cumulative distribution. From the CDFs, the median throughput for strong RSSI locations is generally higher than for weak RSSI locations. However, the strong RSSI downloads exhibit more variability in median throughput compared to the weak location, with overlap in throughput for the lowest 5% of the distribution.

The goal of the SEARCH algorithm is to exit slow start once the congestion window has reached the congestion point but before inducing packet loss. In networks with high RTT, the one-way delay latency recorded on the client is useful for estimating when the congestion point is reached [8]. Unfortunately, this technique is not effective for networks with low RTT, such as Wi-Fi. Instead, we use the median throughput

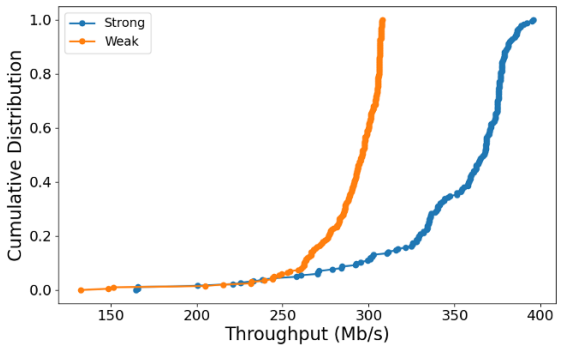


Figure 8: Distribution of median throughputs for strong and weak RSSI locations.

after slow start as the bitrate at the congestion point.

Figure 9 shows the performance for the 200 downloads at the strong RSSI location. This figure compares the performance of the SEARCH algorithm (green), HyStart on (red), HyStart off (purple), and BBR (blue). Figure 9(a) has the time required to download bytes from 0 to 4 MB for each configuration. The x-axis is the download size in Megabytes (MB) and the y-axis is the download time in seconds. Each point represents the average download time for 200 cases, measured at 100 KB intervals from 0 to 4 MB. The shaded areas denote the standard error around the mean. The horizontal dashed lines indicate the average slow start exit times for each configuration, matching their respective colors. As shown, HyStart enabled has the earliest exit time from slow start, resulting in the longest download times. For the other configurations – SEARCH, BBR, and HyStart off – the download times are close together, with HyStart off the lowest. followed closely by SEARCH and then BBR. Figure 9(b) shows the corresponding slow start exit times for each configuration as mean values bound by standard error bars. HyStart on exits slow start the earliest at approximately 0.02 seconds HyStart off delays slow exit times to about 0.2 seconds. BBR exits start up at about 0.1 seconds, between the two HyStart configurations. TCP with SEARCH exits slow start at roughly 0.04 seconds, earlier than BBR but later than HyStart on. Figure 9(c) shows the corresponding retransmissions (mean values with standard error bars). HyStart off shows the highest number of retransmissions, approximately 2600 packets, whereas the other configurations have substantially fewer retransmissions, all below 300 packets. HyStart on has the fewest retransmissions, followed by SEARCH and then BBR.

Based on these results, enabling HyStart in TCP results in the lowest packet loss and subsequent retransmissions but requires the most time to download. Disabling HyStart reduces download times but significantly increases packet loss, as evidenced by the higher number of retransmissions. TCP with SEARCH strikes a favorable balance, achieving lower download times while maintaining low packet loss. TCP BBR performs better than HyStart enabled in terms of download time and better than HyStart disabled in terms of packet loss, but exhibits higher download times and packet loss compared to SEARCH.

Figure 10 shows the same results from 200 download cases conducted at the weak RSSI location. In general, downloads at the weak RSSI location take longer than at the strong RSSI location, but comparatively, the four experimental conditions are similar. Similar to the strong RSSI location, SEARCH has the best performance of the four by balancing download time and packet loss compared to the other configurations.

Conclusions

In this study, we evaluated the SEARCH algorithm implemented as a congestion control module in the Linux kernel. We illustrated SEARCH performance compared to TCP Cubic with and without HyStart over a GEO satetellite network. The examples demonstrate that TCP with SEARCH effectively exits slow start after reaching the congestion point, reducing packet loss and improving throughput compared to the other methods. We also did extensive evaluation over a Wi-Fi campus network at two locations with different signal strengths – strong and weak – comparing TCP Cubic with SEARCH, TCP Cubic with and without HyStart, and TCP BBR (version 1.0). Results of 200 downloads at each location over a 24 hour period show SEARCH consistently has lower download times than HyStart enabled, and comparable download times to HyStart disabled and BBR, but lower packet loss than either.

Current ongoing development of SEARCH algorithm is focused on refining the slow start exit strategy to more precisely match the congestion condition, taking into account the delay inherent in SEARCH upon detecting the chokepoint. Additionally, our intent is to upstream SEARCH into the Linux mainstream kernel, as well as integrate it into open-source QUIC libraries at the user level. Furthermore, we plan to evaluate the performance of TCP SEARCH implemented in the Linux kernel over other network environments, such as cellular networks and LEO satellite links.

Acknowledgments

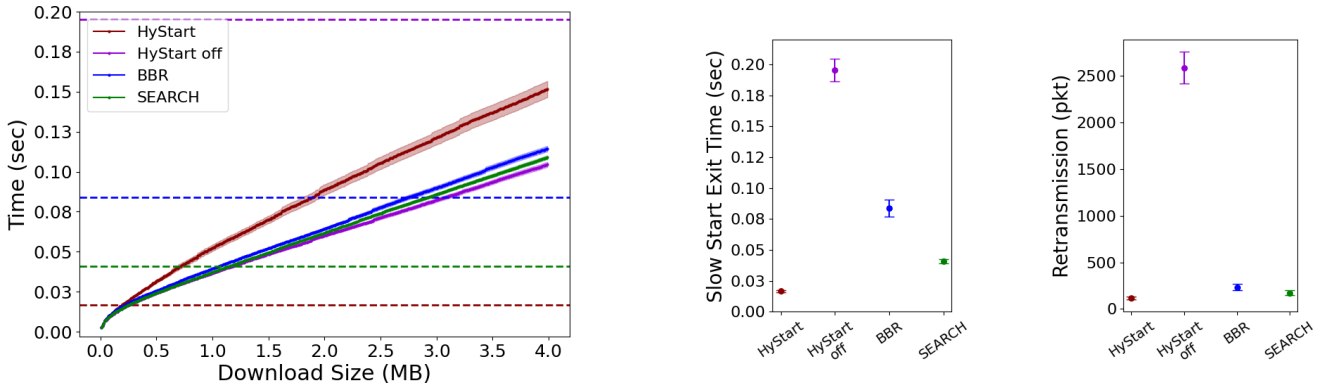
Thanks to Connor Tam, Jose Manuel Perez Jimenez and Katy Stuparu for running the experiments for the Wi-Fi heatmap and doing preliminary experimental design.

References

[1] Abed, G. A., and Jasim, A. M. 2022. An Effective Practice to Approximating TCP Congestion Window Threshold in High Latency Connections. *Al-Iraqia Journal for Scientific Engineering Research* 1.

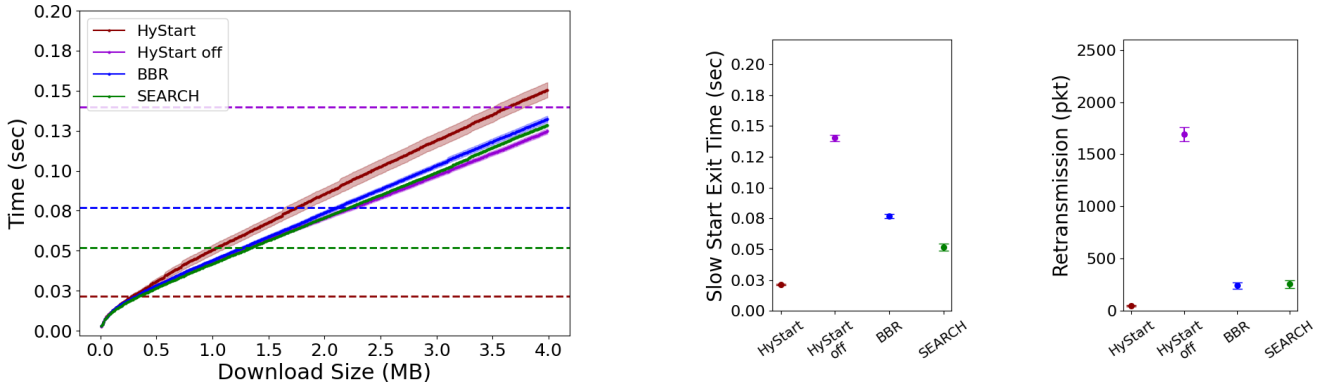
[2] Cronin, A.; Kachooei, M. A.; Chung, J. W.; Li, F.; Peters, B.; and Claypool, M. 2024. Improving QUIC Slow Start Behavior in Wireless Networks with SEARCH. In *Proceedings of the IEEE Local and Metropolitan Area Conference (LANMAN)*.

[3] Gál, Z.; Kocsis, G.; Tajti, T.; and Tornai, R. 2021. Performance Evaluation of Massively Parallel and High Speed Connectionless vs. Connection Oriented Communication Sessions. *Advanced Engineering Software* 157(C).



(a) Time vs. bytes downloaded (b) Average slow start exit time (c) Average retransmissions

Figure 9: Strong RSSI.



(a) Time vs. bytes downloaded (b) Average slow start exit time (c) Average retransmissions

Figure 10: Weak RSSI.

[4] Ha, S., and Rhee, I. 2011. Taming the Elephants: New TCP Slow Start. *Computer Networks* 55(9):2092–2110.

[5] Huang, Y., and Olson, M. 2023. HyStart++: Modified Slow Start for TCP. RFC 9406, RFC Editor.

[6] Kachooei, M. A.; Zhao, P.; Li, F.; Chung, J.; and Claypool, M. 2022. Fixing TCP Slow Start for Slow Fat Links. In *Proceedings of the 0x16 Netdev Conference*.

[7] Kachooei, M. A.; Chung, J.; Li, F.; Peters, B.; and Claypool, M. 2023. SEARCH: Robust TCP Slow Start Performance over Satellite Networks. In *Proceedings of the IEEE 48th Conference on Local Computer Networks (LCN)*, 1–4.

[8] Kachooei, M. A.; Chung, J.; Cronin, A.; Chung, J.; Li, F.; Peters, B.; and Claypool, M. 2024. Improving TCP Slow Start Performance in Wireless Networks with SEARCH. In *Proceedings of the IEEE World of Wireless, Mobile and Multimedia Networks (WoWMoM)*.

[9] Kasoro, N.; Kasereka, S.; Alpha, G.; and Kyamakya, K. 2021. ABCSS: A Novel Approach for Increasing the TCP Congestion Window in a Network. *Procedia Computer Science* 191(C).

[10] Li, L.; Chen, Y.; and Li, Z. 2023. Small Chunks can Talk: Fast Bandwidth Estimation without Filling up the Bottleneck Link. In *IEEE/ACM 31st International Symposium on Quality of Service (IWQoS)*.

[11] Ye, J.; Huang, B.; Chen, X.; and Sangaiah, A. K. 2021. An Improved Algorithm to Enhance the Performance of FAST TCP Congestion Control for Personalized Healthcare Systems. *Wireless Communication and Mobile Computing* 2021.