Validation and Evaluation of HyStart++ for Linux

Maryam Ataei Kachooei[†], Clive Thompson[†], Zimraan Ahmad[†], Joshua Solomon[†],
Jae Chung[*], Feng Li[*], Benjamin Peters[*], Mark Claypool[†]
[†]Worcester Polytechnic Institute, Worcester, MA, USA
[*]Viasat Inc., Marlboro, MA, USA
[†]{mataeikachooei, csthompson, zaahmad, jasolomon, claypool}@wpi.edu [*]{jaewon.chung, feng.li, benjamin.peters}@viasat.com

***Abstract*—Linux TCP CUBIC defaults to using HyStart to exit slow start and transition to congestion avoidance, but HyStart can perform poorly over networks with delay variation by exiting slow start prematurely. HyStart++ was designed to improve HyStart by adding Conservative Slow Start, an intermediate phase between slow start and congestion avoidance that reduces the exponential growth until either confirming a transition to congestion avoidance or going back to slow start. Our work validates an implementation of HyStart++ for Linux, confirming adherence of the code base to the Request for Comments (RFC 9406) and demonstrating functionality through case studies under known conditions. Then, we evaluate HyStart++ over several network paths, focusing on networks with high delay variation, including satellite links. Our results show that HyStart++ improves upon HyStart by continuing to grow after HyStart would have exited slow start, leading to a higher congestion window and better early goodput in some cases. However, HyStart++'s reaction to delay variation restricts utilization, motivating further adjustments to HyStart++ and other slow start mechanisms to improve slow start for wireless access networks.**

***Index Terms*—slow start, HyStart++, delay variation**

I. Introduction

TCP slow start rapidly increases a sender's rate at the beginning of a connection or after an idle period, allowing the sender to discover available network capacity [1]. However, unchecked slow start growth can build queues and trigger packet loss before the sender reaches a stable operating point. Linux CUBIC therefore uses HyStart as an early slow start exit mechanism that attempts to leave slow start before loss occurs [2].

However, HyStart can perform poorly on paths with high delay variation [3]. Wireless and satellite access networks often experience variable round-trip times (RTTs) where increases in delay do not always indicate persistent congestion. As a result, HyStart may exit slow start too early, limiting the congestion window (`cwnd`), reducing early throughput, and increasing download completion time, especially for short and medium-sized transfers.

HyStart++ was proposed to reduce this shortcoming of HyStart by adding Conservative Slow Start (CSS), an intermediate phase between slow start and congestion avoidance [4]. Rather than making the first delay signal irreversible, HyStart++ continues increasing the congestion window but at a reduced growth rate before deciding whether to enter congestion avoidance or return to regular slow start.

This paper validates and evaluates a Linux implementation of HyStart++. We first compare the implementation against RFC 9406 by constructing flowcharts: one representing the RFC-specified HyStart++ logic and others representing the corresponding Linux implementation paths. Comparing these flowcharts allows us to validate whether or not the Linux implementation follows the intended RFC state transitions among slow start, CSS, and congestion avoidance. Our validation also includes instrumenting the code with log messages run during a representative trace illustrating the HyStart++ state variables and transitions.

Finally, we evaluate HyStart++ in a controlled testbed across a range of bottleneck rates, base RTTs, router queue capacities, and jitter levels, and over a real geosynchronous (GEO) satellite path. We compare traditional slow start, HyStart, and HyStart++ to understand how each approach behaves under both stable and variable-delay network conditions. In the controlled testbed, we examine completion time, retransmissions, slow start exit behavior, and queue occupancy. For the GEO satellite dataset, we also report completion time, showing how the algorithms behave for transfers of different sizes on an operational satellite path. The results show that HyStart++ generally improves performance compared with HyStart by avoiding some premature slow start exits and improving early data delivery in the controlled setting. However, HyStart++ can still be conservative under high delay variation, leading to lower utilization than traditional slow start in some cases.

The main contributions we make in this paper are:

- We validate a Linux implementation of HyStart++ against RFC 9406 using flowcharts derived from the RFC specification and the Linux code path.
- We provide evidence via a representative trace showing how HyStart++ transitions between standard slow start, CSS, and congestion avoidance as expected.
- We evaluate traditional slow start, HyStart, and HyStart++ across over 2400 runs per algorithm in a controlled testbed with varying bottleneck rates, RTTs, queue capacities, and jitter.
- We evaluate the three algorithms over a Viasat satellite path using about 160 runs per algorithm collected over a 24-hour window, and report download time behavior across object sizes.
- Finally, we show that HyStart++ reduces premature slow start exits compared with HyStart, but can still under-

utilize paths with high delay variation compared with traditional slow start.

The rest of the paper is organized as follows: Section II reviews related work; Section III presents the RFC 9406 and Linux implementation flows; Section IV validates the Linux implementation and identifies implementation-level differences; Section V presents the experimental evaluation; Section VI discusses the implications of the results and directions for improving slow start on variable-delay paths; and Section VII summarizes the conclusions and future work.

II. Related Work

Slow Start and Slow Start Exit: Early work in congestion control introduced slow start and congestion avoidance as core mechanisms for probing network capacity while reacting to congestion [1]. Later work attempted to make this startup phase less aggressive. For example, Limited Slow-Start reduced slow start growth once the congestion window became large [5], while Speeding Up Slow Start (SUSS) explored alternative startup behavior to reduce overshoot during the initial capacity-probing phase [6].

Because loss-based slow start exit can be too aggressive, prior work explored non-loss signals, including delay and bandwidth estimation signals. TCP Vegas used delay changes to infer queue buildup before loss occurred [7], while TCP Westwood+ estimated available bandwidth from the returning ACK stream [8]. These approaches showed the value of reacting before packet loss, but they were general congestion control mechanisms rather than mechanisms specifically designed to control slow start exit in a Linux implementation.

HyStart: HyStart was proposed to improve slow start exit by detecting congestion before packet loss [2] and is enabled by default in the Linux kernel. Instead of waiting for loss, HyStart monitors delay increases and ACK-train behavior to infer when the sender may be approaching the available capacity. However, delay-based slow start exit is difficult on paths where RTT varies for reasons other than persistent congestion. Wireless and satellite paths can experience delay variation due to scheduling, link-layer retransmissions, changing channel conditions, reverse-path effects, and queueing dynamics. In these environments, transient delay increases can be mistaken for congestion. Prior work has shown that HyStart can exit slow start prematurely on such paths, limiting congestion window growth and reducing early throughput [3]. This premature exit behavior is especially important for short and medium transfers, where slow start behavior can strongly affect completion time.

HyStart++: HyStart++ was specified in RFC 9406 to address some limitations of HyStart [4]. Its main change is the addition of Conservative Slow Start (CSS), an intermediate phase between regular slow start and congestion avoidance. When HyStart++ observes a delay signal, it does not immediately exit to congestion avoidance. Instead, it enters CSS, reduces the growth rate, and continues probing. If the delay signal persists, HyStart++ exits to congestion avoidance; otherwise, it can return to regular slow start. This design is intended to reduce premature exits caused by transient delay variation while still limiting excessive overshoot of the link capacity and packet loss. However, because HyStart++ depends on delay measurements, its behavior can still be affected by high RTT variation. In particular, the sender must distinguish between delay increases caused by real queue buildup and delay increases caused by path variability. This distinction is challenging on wireless and satellite links.

Slow Start on Wireless and Satellite Paths: Slow start behavior is important on wireless and satellite access networks because these paths often combine large BDPs, high RTTs, and substantial delay variation. On such paths, exiting slow start too early can leave much of the available capacity unused, while exiting too late can create large queues and retransmissions. Prior work on the SEARCH algorithm studied slow start exit under delay-variable paths and showed that delivery-based behavior can help identify when a sender is approaching available capacity [9].

Relation to Prior Work: This paper does not propose a new slow start mechanism – instead, it validates and evaluates a Linux implementation of HyStart++ based on RFC 9406. We compare the RFC 9406 logic with the Linux HyStart++ implementation, using flowcharts to explain state transitions and an instrumented trace to verify behavior. We also evaluate traditional slow start, HyStart, and HyStart++ across controlled network conditions and a GEO satellite path.

III. HyStart++; RFC and Linux Implementation

This section validates the Linux implementation of HyStart++ by comparing the code control flow with the algorithm control flow specified in RFC 9406. We first present an RFC-derived flowchart that captures the intended HyStart++ state transitions, followed by a set of flowcharts extracted from the corresponding Linux implementation. In the RFC flowchart, green blocks represent standard slow start (SS), yellow blocks represent Conservative Slow Start (CSS), red blocks represent transitions from SS or CSS into congestion avoidance (CA), and gray blocks represent common supporting checks or variable updates shared by SS and CSS. The Linux flowcharts use the same color scheme to facilitate direct comparison. Each state or decision point is annotated with the corresponding RFC or Linux source code line number, allowing both flows to be directly aligned to their sources. We then summarize the resulting matches and differences.

A. RFC 9406 HyStart++ Flow

Figure 1 summarizes the HyStart++ behavior defined by RFC 9406 [4]. The algorithm begins in standard slow start, where the sender increases the congestion window aggressively (approximately $2\times$ each RTT) while monitoring delay samples for evidence that the path may be approaching congestion. If no delay increase is detected, the sender remains in slow start and continues aggressive congestion window growth.

When HyStart++ detects a delay increase larger than a fixed threshold, it does not immediately enter congestion avoidance.

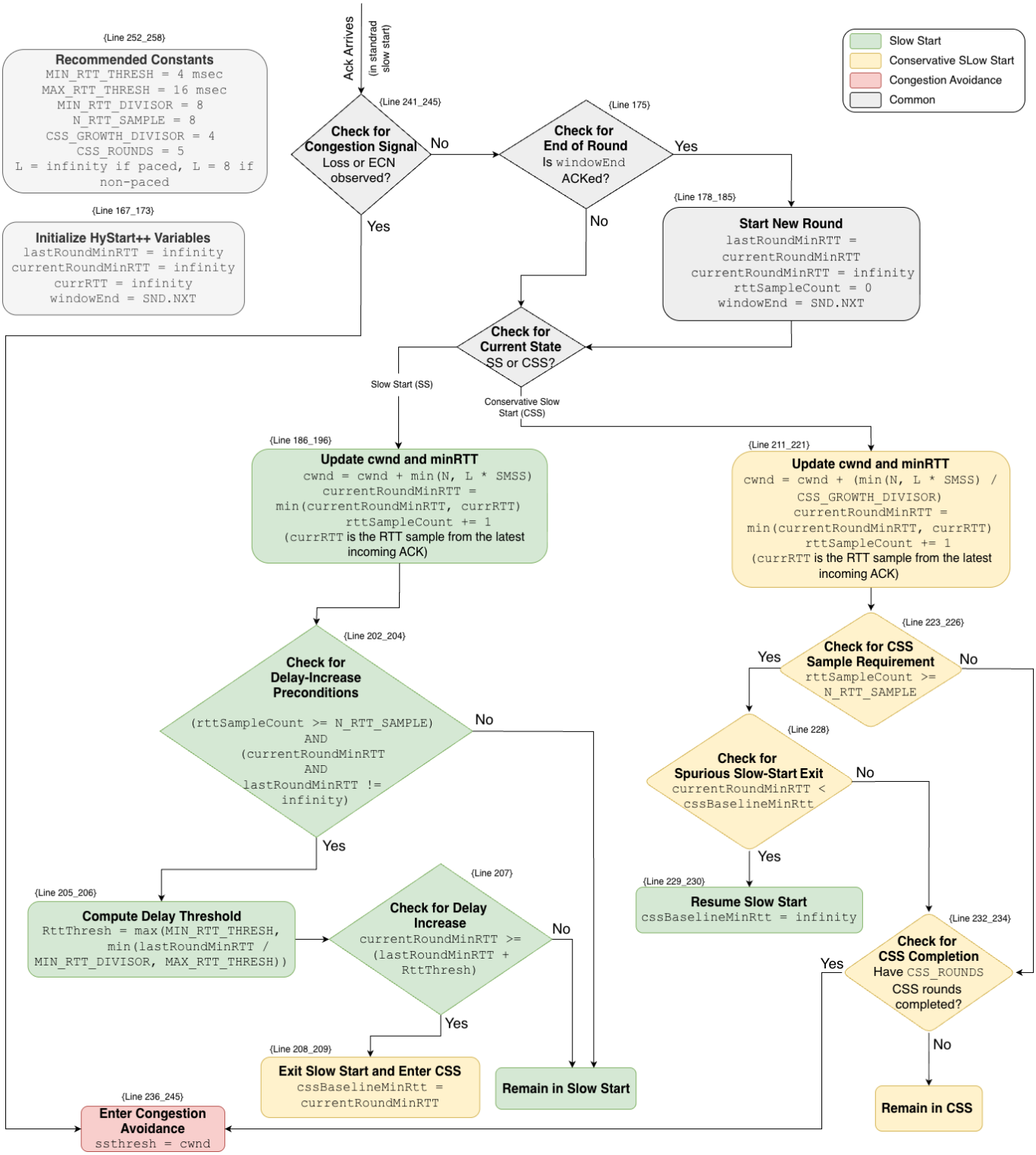

Fig. 1: Flowchart of the HyStart++ algorithm specified in RFC 9406. Green indicates standard slow start, yellow indicates Conservative Slow Start (CSS), red indicates transitions to congestion avoidance, and gray indicates supporting checks or updates.

Instead, it transitions into Conservative Slow Start (CSS), an intermediate state that reduces the congestion window growth rate (to approximately $1.25\times$ each RTT) while continuing to test whether the delay signal is persistent, inferring congestion. If the high delay signal continues for five RTTs during CSS, HyStart++ exits slow start and enters congestion avoidance. If the increased delay signal does not persist, HyStart++ returns to standard slow start. This additional CSS state is the key difference between HyStart++ and the earlier HyStart design: it gives the sender a chance to distinguish transient RTT variation from sustained queue buildup before leaving slow start.

B. Linux HyStart++ Implementation Flow

Figures 2–6 show the corresponding control flow extracted from the Linux implementation of HyStart++. The implementation is based on modified Linux CUBIC source code available in the HyStart++ GitHub repository [10]. As in the RFC-derived flowchart, the colors distinguish the main HyStart++ states and supporting logic: green indicates standard slow start, yellow indicates CSS, red indicates transitions into congestion avoidance, and gray indicates common checks or variable updates. The annotations beside the blocks identify the relevant Linux source code line numbers, allowing each state, condition, and update to be aligned with the implementation.

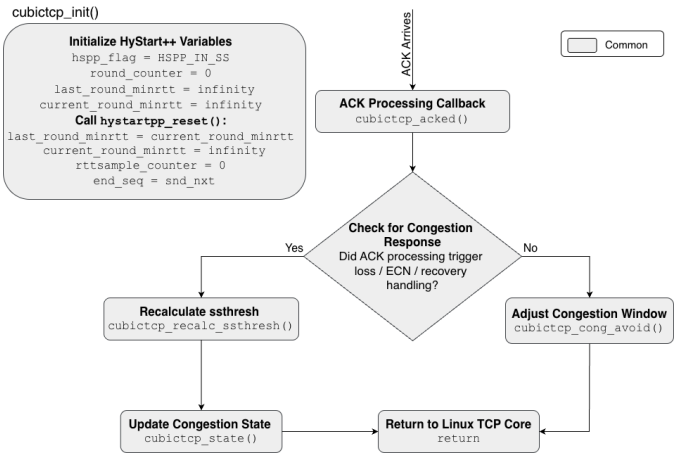

Fig. 2: Overview of the TCP callbacks used by the Linux HyStart++ implementation.

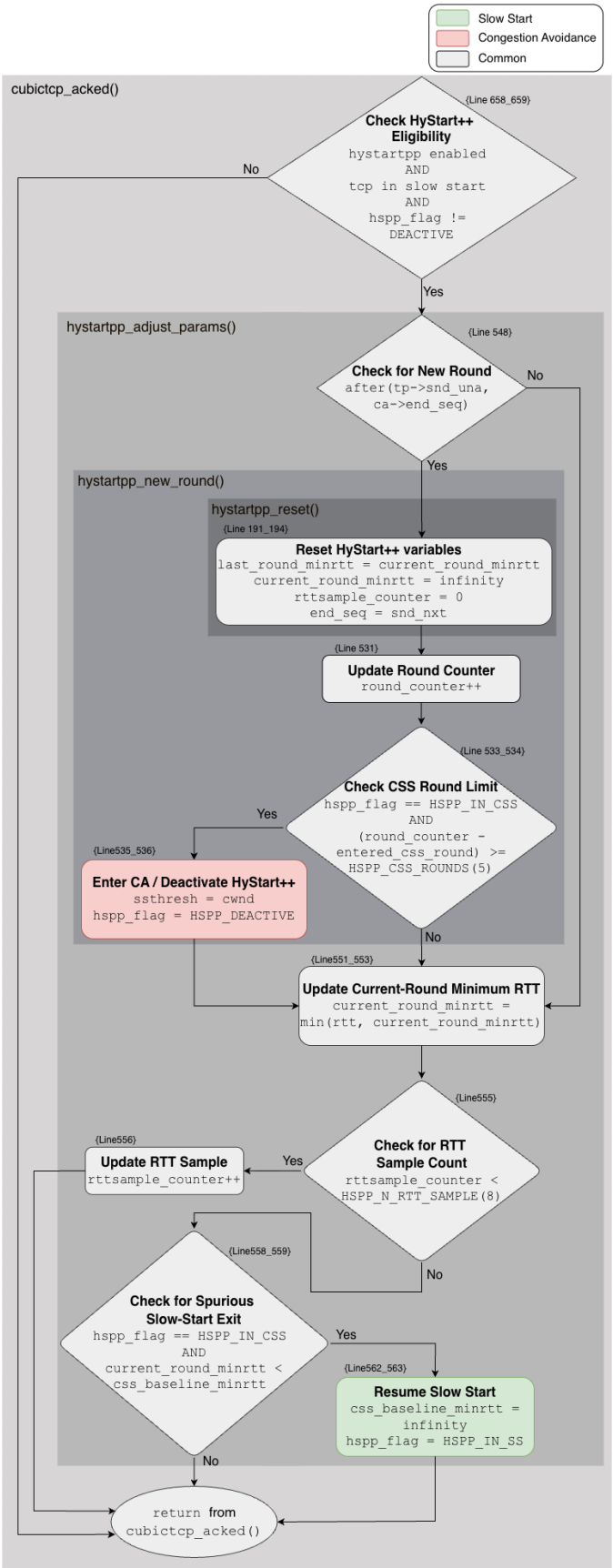

Fig. 3: RTT sampling, round tracking, and CSS rollback handling in `cubictcp_acked()`.

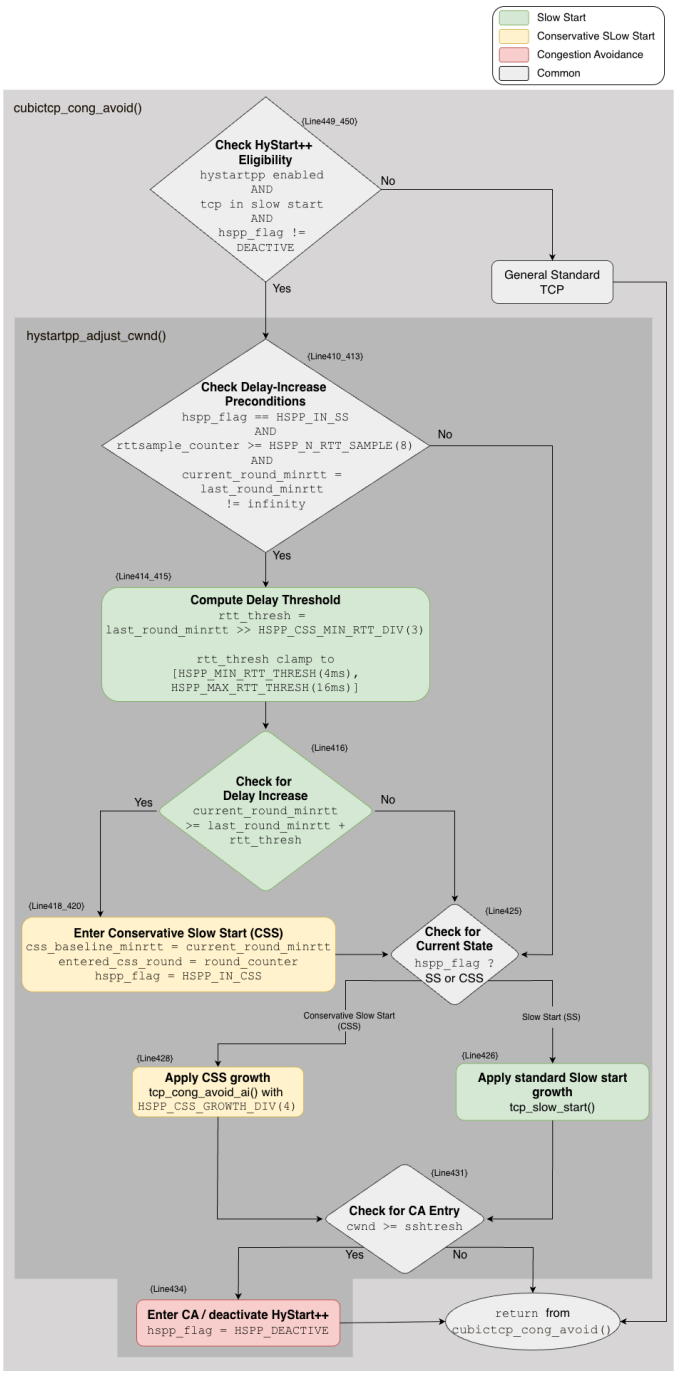

Fig. 4: Congestion window growth and transitions among standard slow start, CSS, and congestion avoidance in `cubictcp_cong_avoid()`.

Figure 2 summarizes how the Linux TCP stack invokes the TCP CUBIC callbacks used by the HyStart++ implementation. We include these functions because the HyStart++ implementation modifies or depends on each of them. In particular, `cubictcp_acked()` maintains the RTT and round information used to decide whether to enter or leave CSS (Figure 3), while `cubictcp_cong_avoid()` applies the corresponding congestion window growth behavior (Figure 4).

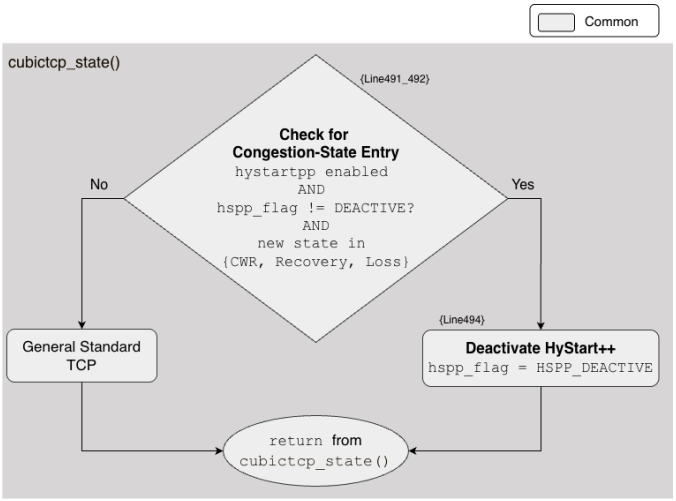

Fig. 5: HyStart++ deactivation during Linux TCP congestion state transitions in `cubictcp_state()`.

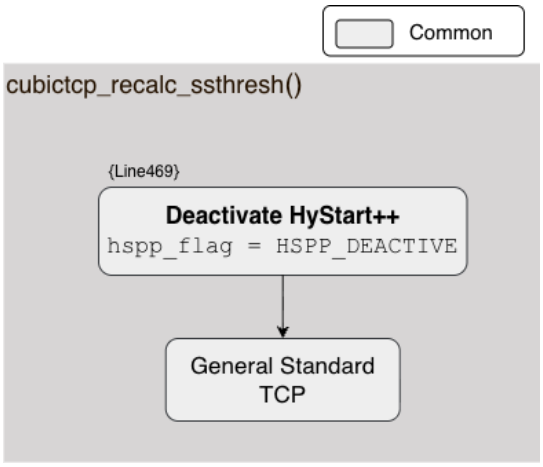

Fig. 6: HyStart++ deactivation and CUBIC slow start threshold recalculation in `cubictcp_recalc_ssthresh()`.

The `cubictcp_state()` path captures interactions with TCP congestion state changes such as loss recovery (Figure 5), while `cubictcp_recalc_ssthresh()` deactivates HyStart++ and applies the standard CUBIC slow start threshold calculation after a congestion event (Figure 6).

IV. Linux HyStart++ Validation

Tables I and II summarize the validation results. Table I maps each major RFC 9406 behavior to the corresponding state or decision in the RFC-derived flowchart and to the Linux functions that implement the behavior. Table II summarizes the validation outcome. A status of "Match" indicates behaviorally equivalent operation. "Match*" indicates that the main HyStart++ behavior is present, but the Linux implementation differs slightly from the RFC 9406 in timing, parameter handling, or interaction with generic Linux TCP processing.

The RFC combines congestion window growth and RTT-state maintenance within the same "Update cwnd and min-

TABLE I: Mapping of RFC 9406 HyStart++ behaviors and flowchart states to the corresponding Linux implementation.

RFC behavior	**RFC flowchart state**	**Linux implementation function**
Initialize minimum RTT values and round boundary	**Initialize HyStart++ Variables**	`cubictcp_init()`, `hystartpp_reset()`
Start a new round and reset per-round measurements	**Check for End of Round**; **Start New Round**	`cubictcp_acked()`, `hystartpp_adjust_params()`, `hystartpp_new_round()`, `hystartpp_reset()`
Update current-round RTT measurements	**Update cwnd and minRTT** (RTT-update portion)	`cubictcp_acked()`, `hystartpp_adjust_params()`
Detect delay increase and enter CSS	**Check for Delay-Increase Preconditions**; **Compute Delay Threshold**; **Check for Delay Increase**; **Remain in Slow Start**; **Exit Slow Start and Enter CSS**	`cubictcp_cong_avoid()`, `hystartpp_adjust_cwnd()`
Apply slow start and CSS growth	**Check for Current State**; **Update cwnd and minRTT** (cwnd-update portion)	`cubictcp_cong_avoid()`, `hystartpp_adjust_cwnd()`, `tcp_slow_start()`, `tcp_cong_avoid_ai()`
Return from CSS after a spurious slow start exit	**Check for CSS Sample Requirement**; **Check for Spurious Slow-Start Exit**; **Resume Slow Start**	`cubictcp_acked()`, `hystartpp_adjust_params()`
Complete CSS and enter congestion avoidance	**Check for CSS Completion**; **Remain in CSS**; **Enter Congestion Avoidance**	`cubictcp_acked()`, `hystartpp_adjust_params()`, `hystartpp_new_round()`
Handle loss or ECN during slow start or CSS	**Check for Congestion Signal**; **Enter Congestion Avoidance**	`cubictcp_state()`, `cubictcp_recalc_ssthresh()`
Restrict HyStart++ to the initial slow start	**Enter Congestion Avoidance**	`cubictcp_init()`, `hystartpp_new_round()`, `hystartpp_adjust_cwnd()`, `cubictcp_state()`, `cubictcp_recalc_ssthresh()`

TABLE II: Validation summary for the Linux HyStart++ implementation against RFC 9406.

RFC behavior	**Status**	**Comment**
Initialize minimum RTT values and round boundary	Match	–
Start new round and reset per-round measurements	Match*	The RFC starts a new round when the ACK reaches or passes the saved round boundary. Linux uses the strict `after()` function in comparing `snd_una` and therefore only starts a new round after passing the saved round boundary.
Update current-round RTT measurements	Match	–
Detect delay increase and enter CSS	Match	–
Apply slow start and CSS growth	Match*	The RFC applies the per-ACK limit `L` before increasing `cwnd`. Linux passes the full `acked` value to `tcp_slow_start()` or `tcp_cong_avoid_ai()`.
Return from CSS after a spurious slow start exit	Match*	The RFC evaluates the rollback condition once eight RTT samples have been collected. Linux evaluates the rollback condition only after nine samples have been collected because the check is in the `else` branch after the counter reaches eight.
Complete CSS and enter congestion avoidance	Match	–
Handle loss or ECN during slow start or CSS	Match*	RFC sets `ssthresh = cwnd` when leaving HyStart++ after a congestion signal. Linux deactivates HyStart++ but uses the standard CUBIC multiplicative-decrease calculation for setting `ssthresh`.
Restrict HyStart++ to the initial slow start	Match	–

RTT" state. Linux performs these operations through separate callback paths; therefore, this RFC state appears in two rows of Table I.

The following discussion groups the validation results into three categories: 1) matching behaviors in both the RFC and Linux implementation (Section IV-A), 2) matching behaviors but with slight timing and state-handling differences (Section IV-B), and 3) different behavior in processing congestion windows (Section IV-C).

A. Matching Behaviors

The initialization behavior in the Linux implementation matches that in the RFC – both the RFC and the Linux implementation initialize the previous- and current-round minimum RTT values and set the initial round boundary to the current next-send sequence number, represented as `SND.NXT` in the RFC and `snd_nxt` in Linux.

The current-round RTT measurement behavior also matches – both update the current-round minimum RTT using each eligible RTT sample. The RFC increments the sample count for every sample, whereas the Linux implementation saturates the counter at eight. Because subsequent HyStart++ checks only verify that at least 8 samples have been collected, this implementation difference does not affect the algorithm's behavior.

Delay-increase detection in the Linux implementation likewise matches the RFC – both evaluate the condition after eight RTT samples have been collected, compute the threshold as one-eighth of the previous-round minimum RTT, clamp it to 4–16 ms, and enter CSS when the current-round minimum RTT reaches or exceeds the previous-round minimum RTT plus that threshold.

Exiting the CSS state also matches in the Linux implementation and the RFC – both limit time in CSS to five rounds, count a partial CSS-entry round toward that limit, set `ssthresh = cwnd`, and enter congestion avoidance after five rounds in CSS. After deactivating HyStart++, the Linux implementation continues updating the current-round minimum RTT and sample counter each ack. These updates are unnecessary because HyStart++ has already been deactivated, but they do not affect the state transition or subsequent congestion control behavior.

Finally, the Linux implementation follows the RFC recommendation to restrict HyStart++ to the initial slow start episode; HyStart++ is activated during connection initialization and is not reactivated after the connection leaves the initial slow start & CSS state.

B. Timing and State-handling Differences

The round-transition behavior is marked Match* in Table II. Both the Linux implementation and RFC transfer the current-round minimum RTT to the previous-round value, reset the current-round minimum RTT and RTT sample counter, and update the round boundary. However, the RFC starts a new round when the ACK reaches or passes the saved boundary, whereas the Linux implementation uses the strict `after()` comparison function. Consequently, Linux does not begin the new round when `snd_una` is exactly equal to the saved boundary and instead waits until an ACK advances beyond it.

The CSS rollback behavior is also marked Match*. Both implementations resume standard slow start when the current-round minimum RTT falls below the CSS baseline, but the RFC evaluates this condition once eight samples have been obtained, whereas the Linux implementation first evaluates the current-round minimum RTT while processing the ninth eligible RTT sample.

C. Congestion Window Growth Differences

Slow start and CSS growth are marked Match* in Table II. Both the Linux implementation and RFC apply normal growth during slow start and reduce the growth rate by a divisor of four during CSS. In the RFC, N denotes the number of newly acknowledged bytes carried by the current ACK, while SMSS is measured in bytes. The RFC limits the data credited by one ACK to $\min(N, L \times \mathrm{SMSS})$, thereby limiting the increase to at most L SMSS-sized segments per ACK. In the Linux implementation, the `acked` argument represents the number of newly acknowledged packets. Linux passes this value directly to `tcp_slow_start()` or `tcp_cong_avoid_ai()` without explicitly applying the RFC limit L.

Loss and ECN handling are also marked Match* – both leave the HyStart++ slow start or CSS states after a congestion signal. However, the RFC specifies setting `ssthresh = cwnd`, whereas Linux uses the standard CUBIC multiplicative-decrease calculation (0.7) when setting `ssthresh`.

Overall, the Linux implementation follows the main RFC states, albeit with a few implementation-level differences: round-boundary detection, CSS rollback timing, handling of byte limits, and slow start threshold (`ssthresh`) updates after loss or ECN.

V. Evaluation

We evaluate the Linux implementations of traditional slow start, HyStart, and HyStart++ in both controlled and operational network environments. Our evaluation has three objectives. First, we use a representative HyStart++ trace to illustrate how Conservative Slow Start (CSS) changes slow start behavior compared with HyStart. Second, we compare the three algorithms over a broad range of controlled network conditions. Third, we evaluate their behavior over an operational geosynchronous satellite connection whose capacity, delay, and congestion conditions vary over time.

A. Experimental Methodology

1) Controlled Testbed: The testbed mimics a typical scenario in which a laptop downloads from a server where the "last mile" is the bottleneck. We use a PC as the TCP sender and a laptop as the TCP receiver, both connected through a router that applies rate limiting, queueing, delay, and jitter.

The sender PC is a virtual machine running Linux kernel 6.14.0, configured with four virtual CPUs based on Intel Xeon

E5-2630L v4 processors at 1.80 GHz and 8 GB of RAM. The receiver PC is a mobile laptop equipped with an Intel Core Ultra 7 155U CPU and 15 GB of RAM, running Ubuntu 24.04.2 LTS with Linux kernel version 6.17.0. The router is a Raspberry Pi running Linux 5.10.63-v7l+ on a quad-core ARM Cortex-A72 processor. The router applies the bottleneck rate, base delay, queue capacity limit, and delay jitter using Linux traffic-control.

The sender runs Linux with the TCP stack configured to run one of: HyStart, HyStart++, or traditional slow start, each implemented based on the default Linux TCP CUBIC congestion control module. The receiver runs an unmodified Linux TCP stack.

The controlled testbed varies four network parameters: bottleneck rate, base RTT, queue size, and jitter. We consider bottleneck rates of 50, 100, and 150 Mb/s; base RTTs of 25, 100, and 400 ms; queue capacities of 1, 2, and 4 BDP; and jitter magnitudes of one-quarter RTT, one-half RTT, and one RTT.

For each configuration, we perform 30 `iperf3` downloads with each congestion control algorithm. This produces 2,430 runs per algorithm and 7,290 controlled runs overall. Each controlled run lasts 10 s. All three algorithms (HyStart, HyStart++, and traditional) are evaluated under the same set of network configurations.

Each experiment uses one continuous (10 s) bulk-download transfer. Completion times for downloads of different sizes are determined based on when the cumulative acknowledged data reaches each size. Thus, a single run provides completion-time measurements for multiple object sizes. In addition to object completion time, we examine two aspects of the slow start tradeoff. First, we measure the number of retransmitted packets observed when 5 MB has been acknowledged. Second, we examine the amount of data downloaded and the estimated queue occupancy (normalized by the BDP) when slow start exits to congestion avoidance.

2) Geosynchronous Satellite Testbed: We also evaluate the three algorithms over a geosynchronous satellite connection. The sending side consists of three separate virtual servers configured as for the controlled testbed: one server runs traditional slow start, one runs HyStart, and one runs HyStart++. Using a dedicated virtual server for each algorithm allows temporarily striping iterations across servers without having to reconfigure a single server between runs.

The receiving client is a PC equipped with an Intel i7-5820K processor operating at 3.30 GHz and 32 GB of RAM. It runs Ubuntu 20.04 with Linux kernel 5.4.0. The client connects to a Viasat satellite terminal through a gigabit Ethernet link. The terminal is associated with the Viasat-2 satellite and supports a peak downlink rate of 144 Mb/s.

The Viasat gateway maintains a per-client queue that can grow to approximately 36 MB. Active Queue Management is used to limit excessive queue growth. Once the queue exceeds approximately 18 MB, the gateway randomly drops 25% of incoming packets. To emulate encrypted or otherwise constrained scenarios in which performance-enhancing proxies cannot be used, the performance-enhancing proxy is intentionally disabled.

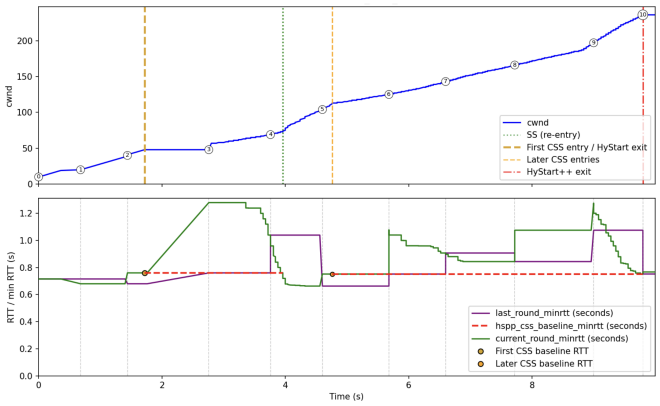

Fig. 7: Representative HyStart++ run. The upper panel shows the congestion window and the transitions between slow start, CSS, and congestion avoidance. The lower panel shows the previous-round minimum RTT, current-round minimum RTT, and CSS baseline RTT.

The experiments are scheduled sequentially in a round-robin order: HyStart, HyStart++, and traditional. This sequence is repeated throughout the measurement period. Consequently, all three algorithms are sampled across the same range of time-of-day conditions rather than evaluating one algorithm entirely before the next. Only one transfer is run at a time, preventing the three experimental flows from competing with one another.

For each algorithm, we collect 161 runs distributed over approximately 24 hours, resulting in 483 runs overall. Each run lasts 45 s and, as in the controlled testbed experiments, each run is a single, long transfer, with download times for a given object size determined by when the cumulative acknowledged data reaches that size.

B. Representative HyStart++ Behavior

Figure 7 presents a representative HyStart++ run illustrating key variables used by the algorithm. The upper panel shows the congestion window evolution, while the lower panel shows the minimum RTT measurements used by HyStart++: green is the current minimum RTT, purple is the minimum RTT from the previous round, and red is baseline RTT when in CSS. Numbered markers in the upper panel indicate successive rounds, shown as dashed gray lines in the bottom panel. Dashed vertical lines show entry into CSS (orange), re-entry into standard slow start (green), and the final transition to congestion avoidance (red).

The trace illustrates the HyStart++ behaviors which can be contrasted with HyStart. At the first sustained increase in RTT just after round 2, HyStart would immediately exit slow start to congestion avoidance. HyStart++ instead enters CSS and continues increasing the congestion window at a reduced rate. This allows the sender to test whether the observed RTT increase represents persistent congestion or only transient delay variation. In the example, the first RTT increase after

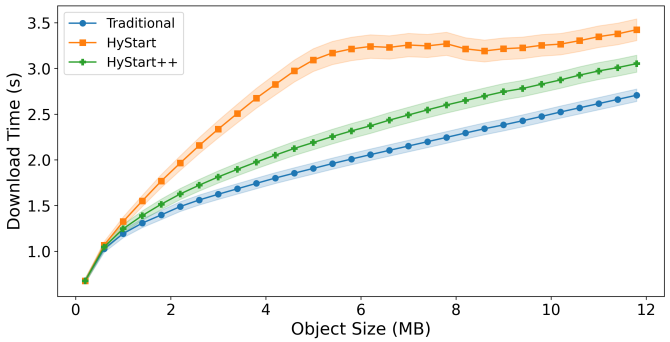

Fig. 8: Download time as a function of object size over the controlled dataset. Each curve aggregates the 2,430 runs for one algorithm. Shaded regions show 95% confidence intervals.

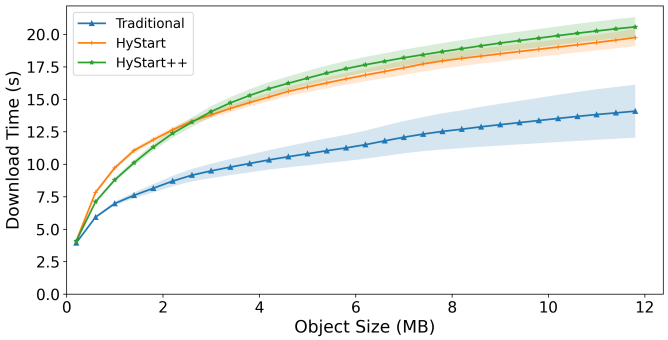

Fig. 9: Download time as a function of object size over the geosynchronous satellite network. Each curve summarizes mean values across 161 runs scheduled over 24 hours, with shaded regions depicting 95% confidence intervals.

round 2 does not immediately exit to congestion avoidance but instead enters CSS, continuing congestion window growth but at a reduced rate (down from $2\times$ to $1.25\times$). While in CSS, HyStart++ measures the minimum observed RTT, and just after round 4, the minimum RTT (green line) drops below the RTT baseline (red line) and HyStart++ transitions back to traditional slow start growth. Another RTT increase just after round 5 triggers another transition back to CSS. This time, the RTT (green line) never drops below the minimum RTT baseline (red line) and so, after 5 rounds, HyStart++ exits CSS to enter congestion avoidance.

C. Object Completion Time

Figure 8 shows download time as a function of object size over the controlled dataset. Each point is the mean value for that size taken over all experimental conditions. The solid line connects the mean completion times and the shaded regions show 95% confidence intervals.

The three algorithms are relatively close in download time for the smallest objects because those transfers complete before their slow start behaviors have diverged substantially. The differences become more visible as the object size increases.

Traditional slow start provides the shortest average completion time across the evaluated object sizes. Because traditional slow start does not use an RTT-based early-exit mechanism, it continues its exponential growth until a loss or another congestion event forces it to reduce its window. This aggressive behavior allows it to reach larger sending rates (and the link capacity) quickly, but it also creates substantially more queuing and retransmission (see subsequent analysis).

HyStart has the largest completion times. Its delay-based detector can interpret a temporary RTT increase as congestion and leave slow start before the available capacity has been fully utilized. After this transition, the congestion window grows more slowly in congestion avoidance, increasing the completion time of medium and large transfers.

HyStart++ generally lies between traditional and HyStart. The CSS state allows it to continue probing after the first delay signal rather than immediately committing to congestion avoidance. Consequently, it retains more of the performance benefits of traditional slow start than HyStart while remaining less aggressive than traditional.

Figure 9 shows the corresponding object-size completion curves for the geosynchronous satellite dataset.

As for the controlled testbed, traditional slow start provides the shortest mean download times. The gap between traditional and HyStart and HyStart++ increases with object size. This indicates over the satellite network, continuing aggressive slow start growth provides a considerable completion time advantage.

HyStart and HyStart++ have similar completion times for the smallest object sizes. For these transfers, completion occurs before the algorithms spend substantial time in their different post-detection states. As the object size increases, both algorithms become slower than traditional slow start. HyStart++ is initially a bit faster in performance to HyStart, but for transfers above about 3 MB, HyStart++ is a bit slower than HyStart.

The satellite result highlights an important limitation of fixed RTT-based slow start heuristics. The satellite path exhibits large and time-varying delay, so an RTT increase does not necessarily indicate persistent congestion. HyStart responds to this variability by exiting too early and while HyStart++ avoids making the first RTT increase the signal to transition to congestion avoidance, the reduced congestion window growth rate in CSS reduces performance for 5 rounds before then entering congestion avoidance. In both cases, HyStart and HyStart++ exit to congestion avoidance before the link capacity is reached.

D. Performance Tradeoffs

Figure 10 compares the mean time required to download 5 MB with the mean number of retransmitted packets observed by that point in the controlled dataset. Along both axes, lower is better, so the best performance is in the bottom left corner.

The results illustrate the performance–retransmission tradeoff. Traditional slow start completes the 5 MB transfer the fastest, but it also produces the largest number of retransmissions. Its aggressive growth fills the available path quickly, but

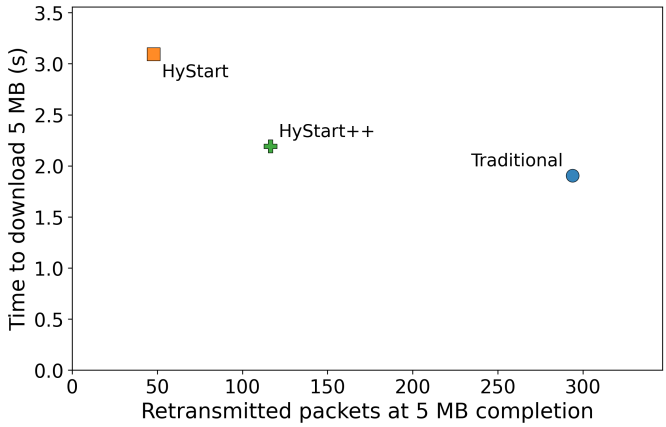

Fig. 10: Mean time to download 5 MB versus retransmitted packets at 5 MB completion over the controlled dataset. Lower values are preferable on both axes.

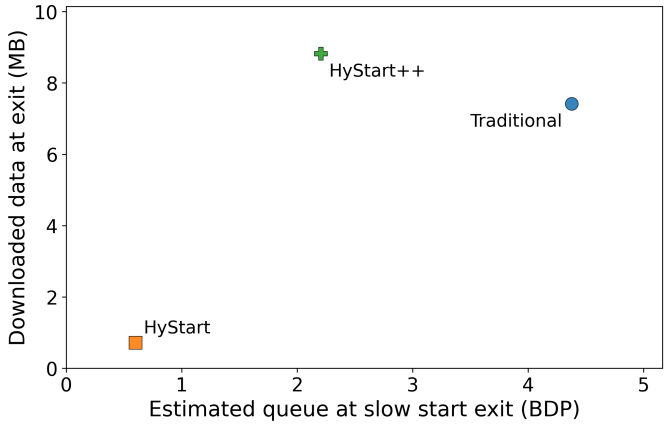

Fig. 11: Downloaded data and estimated queue occupancy when slow start exits. Queue occupancy is normalized by the BDP of each controlled configuration.

frequently overshoots the available link and queue capacity, causing packet loss.

HyStart produces substantially fewer retransmissions, but its early exit from slow start results in the longest completion time. HyStart++ occupies an intermediate operating point. It completes 5 MB faster than HyStart while generating considerably fewer retransmissions than traditional slow start. This result is consistent with the intended role of CSS: rather than choosing between immediate exit and unrestricted exponential growth, HyStart++ uses a temporary reduced-growth phase.

Figure 11 shows a second tradeoff: the relationship between the estimated normalized queue occupancy at slow start exit and the amount of data downloaded before exit. We compute queue occupancy as the excess in-flight data above the BDP, normalized by the BDP: $(\text{inflight} - \text{BDP})/\text{BDP}$. For the x-axis, lower is better while for the y-axis higher is better, so the best performance is in the upper left of the graph.

From the graph, HyStart exits with the smallest estimated queue occupancy, but it also exits after downloading the least data. This indicates that its low queue occupancy is obtained partly by terminating slow start conservatively, often before the sender has fully used the available path.

TABLE III: Classification of slow start exits over the controlled dataset. Percentages are computed over the runs reported for each algorithm.

Algorithm	Early (%)	Timely (%)	Late (%)
Traditional	0.0	0.0	100.0
HyStart	64.4	32.3	3.3
HyStart++	10.8	15.3	73.9

Traditional slow start reaches a much larger downloaded amount before its loss-driven transition, but it does so with the largest estimated queue occupancy. Its exit point therefore reflects substantial overshoot beyond the path BDP.

HyStart++ again provides an intermediate tradeoff. It permits substantially more download progress than HyStart before leaving slow start while maintaining a smaller estimated queue than traditional slow start. In the aggregate result, HyStart++ downloads more data before exit while using roughly half of the normalized queue occupancy observed for traditional slow start. This behavior shows that CSS can reduce premature exits without requiring the sender to continue unrestricted slow start growth until loss.

Taken together, the controlled results show that the three algorithms occupy different operating points. Traditional slow start favors completion time at the expense of loss and queueing, HyStart favors low retransmissions and low queue occupancy at the expense of high completion times, and HyStart++ attempts to balance these objectives.

E. Aggregate Metrics

Tables III and IV summarize the controlled testbed slow start exit behavior and aggregate performance of the three algorithms.

Table III classifies each slow start exit as early, timely, or late. An early exit occurs when slow start exits before the congestion window reaches the BDP. A timely exit occurs when the congestion window reaches the BDP but before packet loss. A late exit occurs when there is packet loss before the algorithm exits slow start. Traditional exits late in all runs because it continues slow start growth until a congestion event occurs. In contrast, HyStart exits early in approximately 65% of runs and exits within the timely region in about 32% of runs. HyStart++ substantially reduces the early-exit rate to about 11%; however, about 74% of its exits are classified as late. Thus, CSS prevents many premature exits, but often allows slow start to continue beyond the desired exit region.

Table IV reinforces the tradeoff observed in the controlled experiments. Traditional slow start reaches the BDP quickly and continues for several RTTs beyond that point before exiting, which explains its short completion times but also its high loss and queue occupancy. HyStart exits well before reaching the BDP on average, shown by its negative time-after-BDP

TABLE IV: Average controlled testbed metrics by algorithm. Values are reported as means with standard deviation in parentheses. Time after BDP is computed as $(t_{exit} - t_{BDP})/RTT$, where negative values indicate exit before reaching the BDP.

Algorithm	Time to BDP (s)	Slow start Exit (s)	Time after BDP (RTTs)
Traditional	1.51 (1.68)	2.12 (2.02)	5.0 (3.01)
HyStart	1.79 (2.37)	0.91 (0.84)	-13.95 (22.50)
HyStart++	1.56 (2.02)	2.52 (2.49)	5.67 (7.46)

value, indicating conservative early termination of slow start. HyStart++ reaches the BDP almost as quickly as traditional and remains in slow start for several RTTs afterward, showing that CSS substantially reduces premature exits compared with HyStart. However, this later exit also explains why HyStart++ can incur more queueing than HyStart.

F. Summary of Findings

The controlled and satellite evaluations lead to three main observations. First, traditional slow start achieves the shortest completion times, but this performance comes with substantially more retransmissions and larger queue occupancies. Second, HyStart greatly reduces these congestion costs but often exits slow start before fully utilizing the available path. Third, HyStart++ reduces the impact of the first delay signal by introducing the Conservative Slow Start (CSS) state and allowing re-entry into standard slow start. In the controlled dataset, this produces a useful intermediate point between the aggressiveness of traditional and the conservativeness of HyStart.

However, the satellite results show that CSS does not eliminate the fundamental difficulty of using RTT increases as a congestion signal on a high-delay, high-variability path. Repeated or persistent RTT variation can keep HyStart++ in reduced-growth operation and limit its completion-time benefit. These results motivate further investigation of slow start exit mechanisms that distinguish actual capacity saturation from transient delay variation.

VI. Discussion

Our results show that HyStart++ improves over HyStart in controlled settings by reducing the impact of premature delay-based slow start exits. The CSS phase allows the sender to continue probing after an initial delay increase, which preserves more of slow start's capacity discovery behavior. However, the results also show that HyStart++ does not eliminate the difficulty of using an RTT increase as a congestion signal on links with high inherent delay variability, such as satellite and many wireless paths. On these paths, delay increases are not always caused by persistent queue buildup. Scheduling effects, link-layer retransmissions, reverse-path variation, and cross traffic can all increase RTT without implying that the forward path has reached its available capacity. When these transient signals repeatedly trigger CSS, HyStart++ may reduce its growth rate even though additional capacity remains available. At the same time, the controlled exit classification suggests that HyStart++ can shift the dominant failure mode from premature exit to delayed exit: its early-exit rate is substantially lower than HyStart's, but many HyStart++ exits occur after the timely region. Further adjustment should therefore address both false delay signals and overly long probing after persistent congestion becomes evident.

One direction to improve performance could be to tune HyStart++ parameters for paths with high RTT variability. For example, the delay threshold could be adapted based on the observed variation in recent RTT samples so that isolated delay spikes are less likely to trigger CSS. The CSS growth divisor and CSS duration could also be adapted – a highly variable path may benefit from a less restrictive CSS response when delivery continues to improve, while a path showing persistent queue buildup may require a shorter CSS period or a faster transition to congestion avoidance. Such tuning would preserve the HyStart++ structure while making its response less dependent on fixed thresholds.

A more fundamental direction change is to use delivery behavior as an additional slow start exit signal. During slow start, each congestion window increase should produce a corresponding increase in delivered bytes while the sender is still below the available bandwidth. If the congestion window continues to grow but delivered bytes increase only marginally over successive RTT-scale observation windows, the path is likely at its available bandwidth, and additional growth is more likely to build queues than improve throughput. This type of signal is attractive for wireless and satellite networks because transient RTT increases would not necessarily force a slow start exit if delivered bytes continue to increase. Delivery behavior could be incorporated in several ways. A conservative design could use delivered-byte growth to confirm delay-based signals, entering CSS or exiting slow start only when an RTT increase is accompanied by weak delivery growth. A more aggressive design could use delivery behavior as the primary slow start exit signal and retain RTT, loss, and ECN as safety signals.

Overall, the results suggest that HyStart++ is a useful improvement over HyStart, but further improvement for wireless and satellite paths likely requires reducing the dependence on delay as the only signal to exit slow start. Adaptive HyStart++ tuning provides an incremental path, while delivery-based behavior provides a broader direction because it measures whether additional congestion window growth is producing useful delivered data. We leave the design and evaluation of such delivery-aware slow start mechanisms to future work.

VII. Conclusion

This paper validates and evaluates a Linux implementation of HyStart++. The validation compared the Linux implementation against RFC 9406 and showed that the main HyStart++ state machine is preserved, including the transition from standard slow start to Conservative Slow Start (CSS) and then to congestion avoidance. The comparison also identified several implementation-level differences, including round-boundary

handling, CSS rollback timing, handling of the RFC byte limit, and the `ssthresh` update after congestion signals.

The evaluation showed that traditional slow start, HyStart, and HyStart++ occupy different operating points. Traditional slow start achieves the shortest completion times, but does so with more retransmissions and larger queue occupancy. HyStart is more conservative and reduces these congestion costs but often exits slow start before fully using the available path. HyStart++ provides an intermediate behavior in the controlled testbed: CSS reduces premature exits relative to HyStart and allows additional capacity probing, but it can also remain in slow start beyond the timely exit region.

On the Viasat satellite path, the results show that networks with high delay variation remain challenging for RTT-based slow start mechanisms. HyStart++ can avoid making the first delay signal irreversible, but repeated or persistent RTT variation can still reduce its growth rate and limit completion-time performance. Thus, HyStart++ improves the slow start exit decision in some settings, but delay alone remains an ambiguous congestion signal on wireless and satellite paths.

Future work could consider adapting HyStart++ parameters, including CSS duration, CSS growth rate, and delay thresholds, to work well across a broader range of wireless, satellite, and high-BDP network conditions. Other future work could investigate slow start mechanisms that replace or augment delay-based detection with delivery-based behavior. In particular, tracking whether additional congestion window growth produces proportional increases in delivered data may help distinguish persistent capacity saturation from transient RTT variation. This direction is especially relevant for wireless, cellular, and satellite networks, where RTT can vary for reasons unrelated to persistent queue buildup.

REFERENCES

[1] V. Jacobson, "Congestion avoidance and control," *ACM SIGCOMM computer communication review*, vol. 18, no. 4, pp. 314–329, 1988.

[2] S. Ha and I. Rhee, "Taming the elephants: New TCP slow start," *Computer Networks*, vol. 55, no. 9, pp. 2092–2110, 2011.

[3] B. Peters, P. Zhao, J. W. Chung, and M. Claypool, "TCP HyStart performance over a satellite network," in *Proceedings of the 0x15 NetDev Conference*, 2021.

[4] P. Balasubramanian, Y. Huang, Q. Kuang, N. Cardwell, and Y. Cheng, "RFC 9406: HyStart++: Modified slow start for TCP," Internet Engineering Task Force, Internet-Draft draft-ietf-tcpm-hystartplusplus-03, July 2021. [Online]. Available: https://datatracker.ietf.org/doc/html/rfc9406

[5] S. Floyd, "Limited slow-start for TCP with large congestion windows," Internet Engineering Task Force, RFC 3742, March 2004. [Online]. Available: https://www.rfc-editor.org/info/rfc3742

[6] M. Arghavani, H. Zhang, D. Eyers, and A. Arghavani, "SUSS: Improving TCP performance by speeding up slow-start," in *Proceedings of the ACM SIGCOMM 2024 Conference*, 2024, pp. 151–165.

[7] L. S. Brakmo, S. W. O'malley, and L. L. Peterson, "TCP Vegas: New techniques for congestion detection and avoidance," in *Proceedings of the conference on Communications architectures, protocols and applications*, 1994, pp. 24–35.

[8] L. A. Grieco and S. Mascolo, "Performance evaluation and comparison of Westwood+, New Reno, and Vegas TCP congestion control," *ACM SIGCOMM Computer Communication Review*, vol. 34, no. 2, pp. 25–38, 2004.

[9] M. A. Kachooei, J. Chung, F. Li, B. Peters, J. Chung, and M. Claypool, "Improving TCP slow start performance in wireless networks with SEARCH," in *IEEE 25th International Symposium on a World of Wireless, Mobile and Multimedia Networks (WoWMoM)*. IEEE, 2024, pp. 279–288.

[10] Arghavani, Mahdi, "HyStartPP: Modified Linux CUBIC implementation," GitHub repository, commit c0a089e, Feb. 2025, file: implementation/modified_tcp_cubic.c. [Online]. Available: https://github.com/SUSSdeveloper/HyStartPP/blob/c0a089e/implementation/modified_tcp_cubic.c