TCP HyStart Performance over a Satellite Network

Benjamin Peters*, Pinhan Zhao*, Jae Chung†, Mark Claypool*
*Worcester Polytechnic Institute
Worcester, MA, USA
{btpeters, pzhao2, claypool}@wpi.edu
†Viasat
Marlboro, MA, USA
jaewon.chung@viasat.com

Abstract

TCP slow start is designed to begin at a conservative bitrate, but quickly ramp up to the available bandwidth. To avoid overshooting, TCP slow start has a HyStart mode (on by default in Linux) that may exit slow start before packets are lost. Unfortunately, HyStart may also exit slow start prematurely, making it take longer for TCP to reach intended bitrates. This is especially problematic for links with high bandwidth and high latency, such as an satellite Internet connection. This paper evaluates TCP HyStart performance over a commercial satellite Internet link, first evaluating how sender and receiver buffer size settings might limit throughput and subsequently assessing TCP with HyStart on versus TCP with HyStart off for both isolated and simultaneous flows. Analysis shows HyStart on (the default) significantly degrades TCP start-up performance, with average throughputs about half that of HyStart off during starting performance.

Keywords

TCP, slow start, start-up, HyStart, congestion control, round-trip time, packet loss, satellite Internet, buffer sizes, rmem, wmem, cwnd

Introduction

During slow start, TCP increases its congestion window by one for each acknowledgment received, effectively doubling its congestion window when there are no packet losses. Since packet losses are harmful to TCP flows during start-up, HyStart seeks to avoid losses by exiting slow start before it might overshoot the desired window threshold [10]. Unfortunately, exiting early can sometimes degrade performance if it exits before the available bitrate is reached since it will then take longer for the congestion window to reach the desired size. This can be particularly damaging for high-bandwidth, high-latency links since it takes a long time to reach the large congestion window required. In other words, slow start has a double-edged sword: without HyStart, the potential for low throughput caused by loss from a slow start overshoot, but with HyStart the potential for low throughput caused by prematurely exiting slow start.

Despite this, the original HyStart has been enabled by default for all TCP connections in Linux since about 2010. This means it is enabled by default for all high-bandwidth, high-latency Internet links even if it is not clear whether it helps or hurts the connection. Of particular interest are satellite Internet networks, an essential part of modern network infrastructures given their ubiquitous network connectivity for remote areas and especially in emergencies when traditional (i.e., wired) connections may not be available. The number of satellites in orbit is over 2100, a 67% increase from 2014 to 2019 [17], and recent research has improved satellite transmission capacities more than 20x, to a total planned capacity for geostationary satellites of over 5 Tb/s. While throughput gains for satellite Internet are promising, satellite latencies remain a challenge. The physics involved for round-trip time Internet communication between terrestrial hosts using a geostationary satellite accounts for about 550 milliseconds of latency at a minimum [5].

Despite their importance, there are few published studies measuring network performance over actual satellite networks [16], with most studies just using either simulation [2] or emulation with satellite parameters [18, 8]. To the best of our knowledge, there is no prior work assessing the performance of TCP HyStart over actual satellite networks.

This paper presents results from experiments that measure the performance of HyStart in a commercial satellite Internet network. We first assess the role in sender and receiver buffer sizes, including Linux buffer auto-tuning, then performance with HyStart on versus HyStart off. The experiments are done over the Internet, but designed to be as comparable across runs as possible by interlacing runs serially to minimize temporal differences.

Analysis of the results shows for a satellite Internet connection, TCP sender and receiver buffer settings limit performance, but when the maximum buffer settings are raised, Linux auto-tuning does not impede exponential window growth. TCP HyStart significantly limits start-up performance, with HyStart off getting about twice as much bandwidth as HyStart on during start-up, translating to faster Web page (and similarly small) downloads. The advantage HyStart off has over HyStart on carries over to when TCP flows are competing for the satellite link bandwidth.

The rest of this report is organized as follows: Related Work describes work in slow start and satellite Internet related to this paper, Methodology describes our testbed and experimental methodology, Results analyzes our experiment data, and Conclusions summarizes our conclusions and suggests possible future work.

Related Work

This section describes work related to our paper, including TCP slow start and TCP performance over satellite networks.

TCP Slow Start

Ha and Rhee [10] identify overshooting of the available bandwidth during slow start as a problem for performance. They propose *Hybrid Slow Start* (*HyStart*) that uses slow start for growth, but incorporates additional signals to exit slow start in addition to the traditional packet loss. These signals include ACK packet-trains and an increase in round-trip delays. They evaluate HyStart in Linux TCP Cubic and show it can increase link utilization for links from 10 to 400 Mb/s and latencies up to 200 ms.

Ha and Rhee [11] continue their work with evaluation over a broader set of systems (Linux, FreeBSD and Windows XP) and network conditions (wireless and asymmetric networks), with a comparison to other approaches. They show HyStart outperforms by 2x to 3x standard slow start, particularly for end hosts running Microsoft Windows (XP) and large bandwidth-delay product networks.

Zong et al. [20] propose an enhanced TCP mechanism that increases the amount of data transferred in the slow start phase of TCP. Their approach uses the link latency compared to a traditional (e.g., "wired") network as a multiplier for window growth, with some checks to differentiate wireless packet loss from congestion packet loss. Evaluation shows some promise to their approach, but implementation is only via simulation.

Li et al. [14] look to overcome the high round-trip times in satellite networks with a modification to slow start. Their slow start increase is exponential, as in traditional slow start, but becomes logarithmic as the desired thresholds is reached, similar to the TCP Cubic algorithm [12]. Simulation results show improvements to throughput over TCP Reno.

TCP over Satellite Networks

Obata et al. [16] evaluate TCP performance over actual (not emulated, as is typical) satellite networks. They compare a satellite-oriented TCP congestion control algorithm (STAR) with NewReno and Hybla. Experiments with the Wideband InterNetworking Engineering test and Demonstration Satellite (WINDS) network show throughputs around 26 Mb/s and round-trip times around 860 milliseconds. Both TCP STAR and TCP Hybla have better throughputs over the satellite link than TCP NewReno.

Wang et al. [19] provide preliminary performance evaluation of QUIC with BBR on an emulated a satellite network (capacities 1 Mb/s and 10 Mb/s, RTTs 200, 400 and 1000 ms, and packet loss rates up to 20%). Their results confirm QUIC with BBR has throughput improvements compared with TCP Cubic for their emulated satellite network.

Utsumi et al. [18] develop an analytic model for TCP Hybla for steady state throughput and round-trip time over satellite links. They verify the accuracy of their model with simulated and emulated satellite links (capacity 8 Mb/s, RTT 550 ms, and packet loss rates up to 2%). Their analysis shows substantial improvements to throughput over that of TCP Reno for loss rates above 0.0001%

Our Work

Our work extends the above work by using a commercial satellite Internet network, not simulation, considering TCP buffer settings and measuring the impact of HyStart on versus HyStart off. To the best of our knowledge, HyStart, in particular, has not been evaluated on actual satellite networks.

Methodology

To evaluate the impact of TCP settings over satellite links, we setup a testbed, serially bulk-download data with different TCP settings, and analyze the results.

Testbed

We setup a Viasat satellite Internet link so as to represent a client with a "last mile" satellite connection. Our servers are configured to allow for repeated tests and comparative performance by consecutive serial runs with all conditions the same, except for the change in TCP settings.

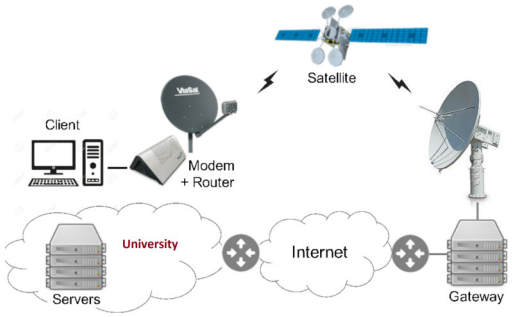


Figure 1: Satellite measurement testbed.

Our testbed is depicted in Figure 1. The client is a Linux PC with an Intel i7-1065G7 CPU @ 1.30GHz and 32 GB RAM. There are two servers, each with an Intel Ken E312xx CPU @ 2.5 GHz and 32 GB RAM. The servers and client all run Ubuntu 18.04.4 LTS, Linux kernel version 4.15.0.

The servers connect to our University LAN via Gb/s Ethernet. The campus network is connected to the Internet via several 10 Gb/s links, all throttled to 1 Gb/s. Wireshark captures all packet header data on each server and the client. The client connects to a Viasat satellite terminal (with a modem and router) via a Gb/s Ethernet connection. The client's downstream Viasat service plan provides a peak data rate of 144 Mb/s.

The terminal communicates through a Ka-band outdoor antenna (RF amplifier, up/down converter, reflector and feed) through the Viasat 2 satellite[1] to the larger Ka-band gateway antenna. The terminal supports adaptive coding and modulation using 16-APK, 8 PSK, and QPSK (forward) at 10 to 52 MSym/s and 8PSK, QPSK and BPSK (return) at 0.625 to 20 MSym/s.

The Viasat gateway performs per-client queue management, where the queue can grow up to 36 MBytes, allowing a maximum queuing delay of about 2 seconds at the peak data

[1] https://en.wikipedia.org/wiki/ViaSat-2

rate. Queue lengths are controlled at the gateway by Active Queue Management (AQM) that randomly drops 25% of incoming packets when the queue is over a half of the limit (i.e., 18 MBytes).

The performance enhancing proxy (PEP) that Viasat deploys by default is disabled for all experiments in order to assess congestion control performance independent of the PEP implementation, and to represent cases where a PEP could not be used (e.g., for encrypted flows).

Our previous work [6] assessed baseline performance for the link (i.e., without any added traffic) and shows the vast majority (99%) of round-trip times are from 560 and 625 milliseconds (median 597 ms, mean 597.5 ms, standard deviation 16.9 ms), and average packet loss rates are about 0.05%, with most of these (77%) single-packet losses.

Downloads

We compare the performance for bulk-downloads, which allows us to assess start-up as well as steady state performance. The servers are configured to use `iperf`[2] (v3.3.1) with the default Linux TCP congestion control algorithm (Cubic). The client initiates a connection to one server via `iperf`, downloading 1 GByte, then proceeds to the next test condition. After cycling through each setting, the client pauses for 1 minute. The process repeats either 5 or 10 times. We analyze results from a weekday before 6pm (local time) in May 2021.

Given the measured round-trip times and the peak data rate, the bandwidth-delay product (BDP) of our satellite link is approximately $140Mb/sec \times 0.6sec = 10.5MBytes$. When indicated, the default TCP buffer settings are changed on the server and/or client – e.g., setting `tcp_mem`, `tcp_wmem` (wmem), `tcp_rmem` (rmem) or `tcp_moderate_rcvbuf` – before starting `iperf`.

Results

This section analyzes performance for TCP buffer settings and then TCP HyStart.

Buffer Settings

For high-bandwidth, high-latency networks, TCP buffer sizes on either the sender or receiver can limit throughput.

As of Linux kernel v2.4 (in 2001), the kernel includes code for TCP buffer sizing. Buffer limits are controlled by kernel variables `net.ipv4.tcp_rmem` (rmem) and `net.ipv4.tcp_wmem` (wmem). The rmem and wmem settings have 3 values that specify the minimum, starting and maximum buffer sizes (in bytes). The rmem defaults are (4096, 131072, 6291456) and the wmem defaults are (4096, 16384, 4194304). For flows that do not explicitly set the TCP buffer sizes, the kernel attempts to grow the window size to match the available bandwidth up to the maximum window. The number of bytes a TCP congestion window can have in flight is the minimum receiver window (`rwind`), congestion window (`cwnd`), and the sender buffer size (wmem). The receiver window can use up to 1/2 the rmem setting, with the remaining space is reserved for overhead.

[2]https://software.es.net/iperf/

As of Linux kernel v2.6 (in 2006), in order to overcome buffer limitations without having to pre-allocate large amounts of memory or manually tune buffer sizes for each network path, Linux includes an auto-tuning feature. Auto-tuning can be disabled system-wide at the receiver,[3] but there is no such setting for auto-tuning at the sender. Auto-tuning code from `net/ipv4/tcp_input.c` in our kernel (v4.15) shows the receiver buffer size grows by either 3x or 4x, depending upon the rate the buffer is increasing. With auto-tuning, however, performance may be still limited by the maximum buffer sizes for high-bandwidth, high-latency networks.

Keeping the wmem default settings, we evaluate the effects of manually changing rmem compared to auto-tuning the receiver window. Figure 2 depicts the results. The text in **bold** at the top highlights the settings that have been changed from the default settings. From the left, the first graphs are for the rmem default settings, the second are for the starting value doubled, the third are for the maximum doubled, and the right is for minimum, starting and maximum all set to 60 Mbytes. For each set of graphs, the x axis is the time, in seconds, since the download started. The charts are aligned temporally and stacked, and from the top down are throughput, round-trip time, `rwnd`, `cwnd` and retransmissions. Each line is the mean across 5 runs, computed once per second, with the shading showing 95% confidence intervals around the mean. The blue lines are with receiver auto-tuning on and the orange lines are with receiver auto-tuning off.

From the graphs, the throughput for receiver auto-tuning is similar in all cases – effectively, `rwnd` is capped at 1/2 the default max, or about 3 MBytes, which limits throughput to about 30 Mb/s for stacks 1 and 2. For stacks 3 and 4, `rwnd` is no longer the limit, growing larger than 3 Mbytes, but throughput is still limited by wmem (which we explore next). Without receiver auto-tuning, throughput is severely limited by the default 131 KBytes in the first stack of graphs, is doubled for the second, does not benefit from an increased max for the third, and finally works about as well auto-tuning when the rmem settings are set large (60 MBytes) in the fourth stack of graphs.

Noting that throughputs are limited even for large rmem settings, we next investigate increasing sender-side limits on the buffer by setting wmem to 60 Mbytes for the minimum, default and maximum. Figure 3 depicts the results. The graphs are as for Figure 2, but here the wmem settings have all been increased.

From the figure, the first two stacks of graphs appear similar to those in Figure 2 because performance is receiver-window limited. However, when the maximum is doubled in the third stack, the TCP flows can take advantage and the throughput increases. But it is not until the fourth stack, when both the sender (wmem) and receiver (rmem) limits have been overcome that throughputs reach the expected rates of about 120 Mb/s for our satellite link.

However, previous work mentions receiver auto-tuning in Linux occasionally delayed the exponential growth of the TCP congestion window [11]. To assess this, Figure 4 zooms

[3]`sudo sysctl net.ipv4.tcp_moderate_rcvbuf=0`

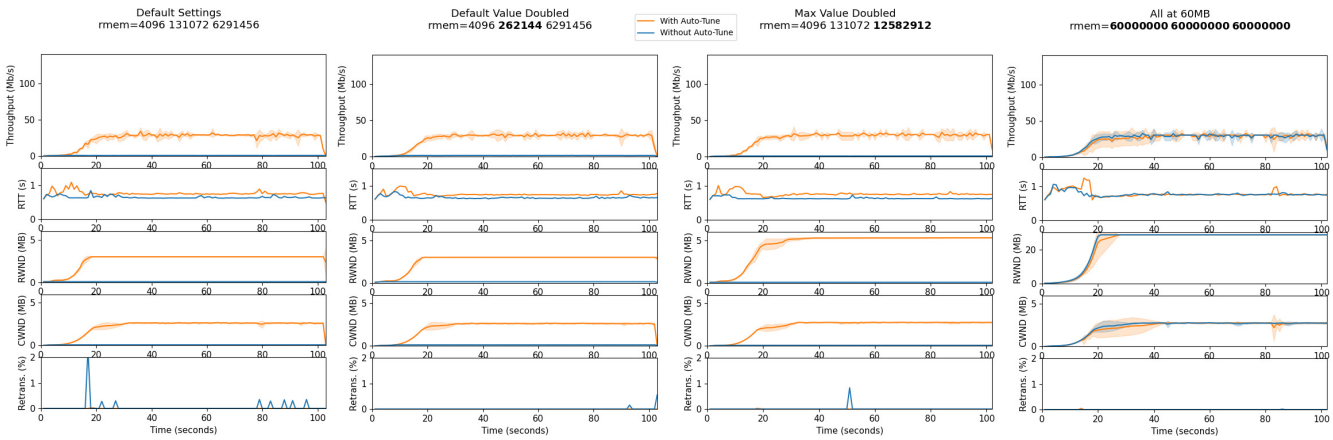


Figure 2: wmem default. rmem: default, 2x starting, 2x max, all 60 MB.

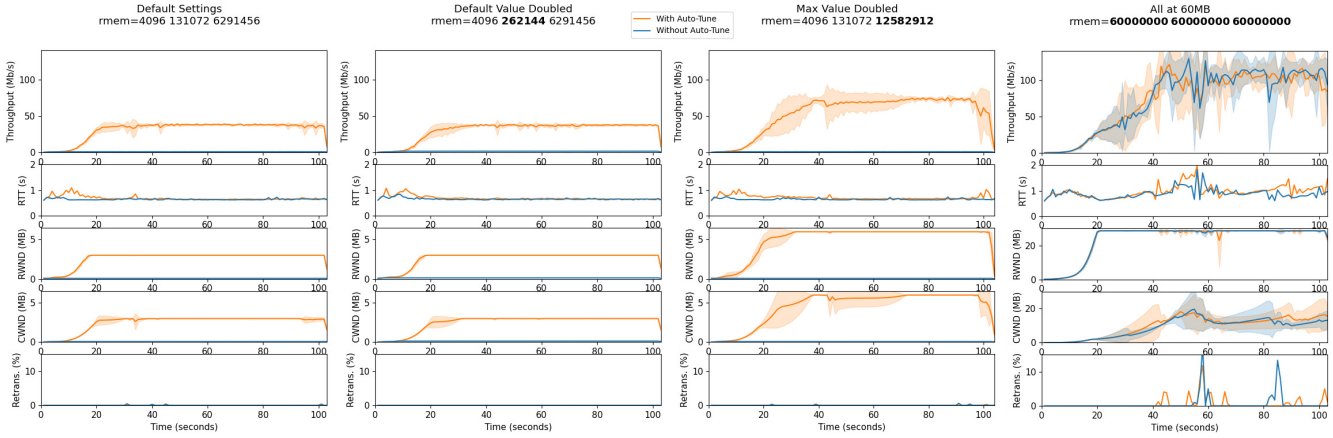


Figure 3: wmem 60 MB. rmem default, 2x starting, 2x max, all 60 MB.

in on the first 15 seconds of downloads with 60 MB settings for wmem and rmem. From this stack, both receiver auto-tuning on and receiver auto-tuning off show exponential growth in the congestion window, without a visual difference between the two of them.

Recommendation Linux has auto-tuning (`tcp_moderate_rcvbuf`) enabled by default. Given we did not observe performance degradations due to auto-tuning during slow start and did observe significant benefits to growing sender and receiver windows up to the maximum, we recommend auto-tuning *remain enabled* by default.

For the TCP receiver buffer (rmem) the defaults are (4096, 131072, 6291456) and for TCP sender buffer (wmem) the defaults are (4096, 16384, 4194304). Since we observed the buffer maximums limiting throughput for our high-bandwidth, high-latency satellite network, we recommend the maximum for both to be *increased* to 2621440 – i.e., rmem to (4096, 131072, **2621440**) and wmem to (4096, 16384, **2621440**).

HyStart

Using 60 Mbyte settings for minimum, default, and maximum for both rmem and wmem, we next turn our attention to the performance of HyStart.

Figure 5 depicts the results comparing HyStart on versus HyStart off, done one flow at a time (serially, interleaving HyStart on/off runs). The graphs are as for Figure 2, with 10 runs for each condition (on/off). From the stacked graphs, HyStart off ramps up throughput more quickly than HyStart on, getting to steady state throughput in about 15 seconds versus 35 seconds, on average. Correspondingly, HyStart off finishes the 1 GB download 25 seconds earlier than HyStart on (100 seconds versus 125 seconds total). These gains are without observable differences in the retransmission rates.

Since HyStart primarily affects slow start, we compare the start-up behavior for HyStart on versus HyStart off by analyzing the first 30 seconds of each trace, approximately long enough to download 50 MBytes on our satellite link. This is indicative of algorithm performance for some short-lived flows.

The average Web page size for the top 1000 sites was

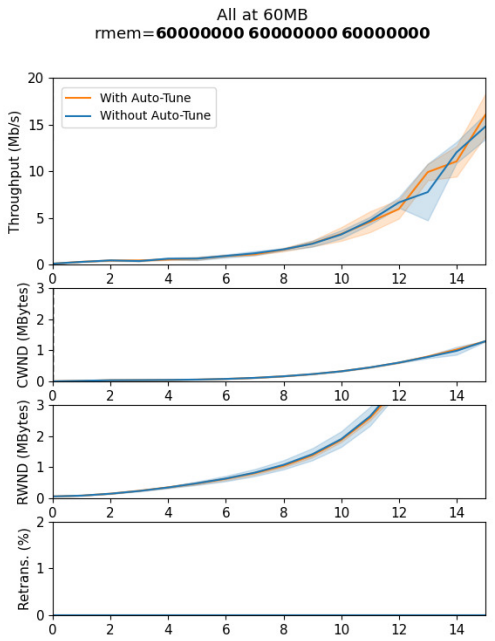


Figure 4: 60 Mbyte wmem and rmem settings.

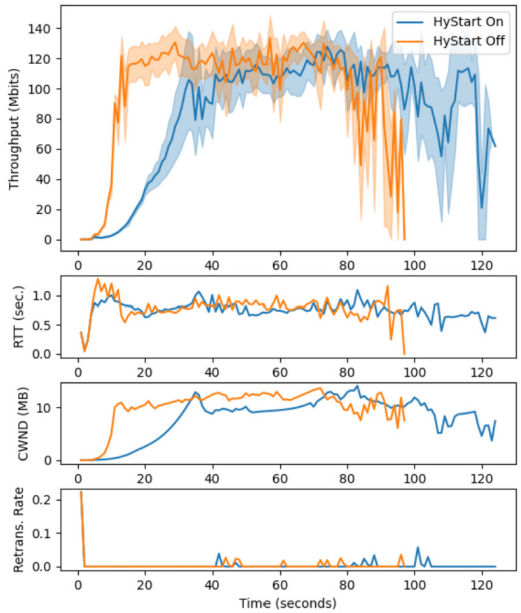


Figure 5: HyStart on/off overall.

around 2 MBytes as of 2018 [7], including HTML payloads and all linked resources (e.g., CSS files and images). The Web page size distribution's 95th percentile was about 6 MBytes and the maximum was about 29 MBytes. Today's average total Web page size is probably about 5 MBytes [9], dominated by images and video.

Many TCP flows stream video content and the amount streamed depends upon the video encoding. However, assuming videos are downloaded completely, about 90% of YouTube videos are less than 30 MBytes [4].

Figure 6 depicts the time on the y-axis (in seconds) to download an object for the given size on the x-axis (in MBytes). The object size increment is 1 MByte. Each point is the mean time required to download an object of the indicated size, shown with a 95% confidence interval.

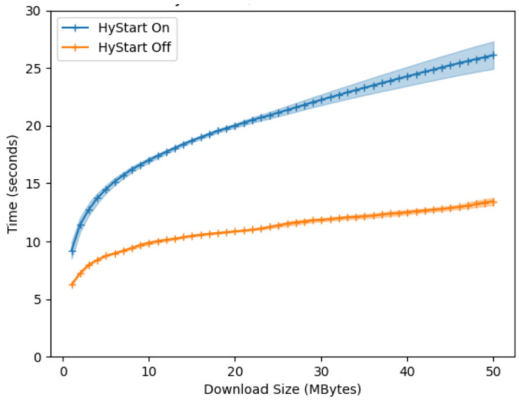


Figure 6: HyStart on/off start-up

From the graph, HyStart off has a significant benefit to download times. For the smallest objects (1 MByte), HyStart on takes about 50% longer (6 versus 9 seconds). For an average Web page download (5 MBytes), HyStart on takes about 2x longer (8 versus 16 seconds), a ratio similar for 90% of all videos and the largest Web pages (30 MBytes) (11 versus 22 seconds).

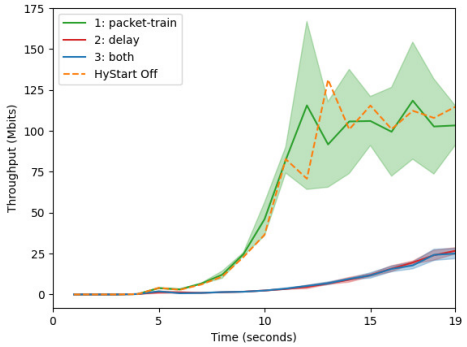


Figure 7: HyStart options.

HyStart includes two different mechanisms to detect when the TCP slow start state should be exited:

1. *packet-train:* When the ACK inter-arrival time goes over a threshold, it exits slow start early.
2. *delay*: when the round-trip time increases over a threshold, it exits slow start early.

By default, both delay and packet train are on by default. We explore the effect these mechanisms have individually and in tandem.

Figure 7 depicts the throughput results comparing the HyStart options (again, run serially, one flow at a time), with 5 runs per option. The x-axis is the time since the download started and the y-axis is the throughput. Each option (1, 2 and both) is shown with a mean bounded by a 95% confidence interval. For reference, the average HyStart off mean performance is shown with a dashed line. From the graph, the packet-train only option 1 performs similarly to HyStart off, suggesting the packet-train method does not cause an early exit. Conversely, the delay only option 2 performs similarly to HyStart with both option 1 and option 2, suggesting the delay mechanism is what causes HyStart to exit slow start. Note the proposed HyStart++ algorithm [1] does not include the packet-train option, only the round-trip time delay option.

Lastly, we analyze the effects that HyStart has for TCP flows run simultaneously (i.e., the flows are *competing* for the link bandwidth, unlike the previous cases where they were run serially). Figure 8 shows two sets of runs: in Figure 8(a), two default TCP flows are run, so both have HyStart on, and in Figure 8(b), one TCP flow has HyStart on and the other TCP flow has HyStart off. Both graphs show the means for 5 runs bounded by 95% confidence intervals.

Computing Jain's fairness index [13] shows two HyStart on flows (Figure 8(a)) has a fairness of 1.0 for the first 15 seconds, and 1.0 at steady state (time 55-115 seconds), while one HyStart on flow versus one HyStart off flow (Figure 8(b)) has a fairness of 0.56 for the first 15 seconds, and 1.0 at steady state.

From the graphs and Jain's fairness, two TCP flows with HyStart on (the Linux default) get approximately the same bandwidth in start-up and steady state. Conversely, when one of the TCP flows has HyStart off, it get significantly more throughput during start-up, which translates to more throughput for the first 40 seconds of the transfer, with corresponding unfairness.

Recommendation Since we observed HyStart's premature exit of TCP slow start over our high-bandwidth, high-latency satellite network can significantly limit throughput, we recommend some changes to the HyStart code. The `HYSTART_DELAY_MAX` parameter provides an upper bound on the deviation from the minimum observed delay that triggers HyStart's exit of slow start. However, this parameter is currently fixed (`16U<<3`) regardless of the underlying network's parameters. To address this shortcoming, we recommend one of two possible changes: 1) Remove the maximum limit, allowing the HyStart threshold trigger to be only determined by the relative deviation from the base latency (currently, 1/8th of the base latency), or 2) adjust the HyStart threshold trigger to be dynamic, based on the mean and standard deviation of the observed network latency. Further tests over a wide range of network conditions (i.e., not just our satellite network) are needed to determine which approach is appropriate.

Conclusion

HyStart is designed to improve TCP performance by avoiding loss caused by a slow start overshoot, but HyStart itself can degrade performance if slow start undershoots. This undershoot can be especially problematic for networks with high bandwidth and high delay, such as satellite Internet networks. Satellite Internet connections are important for providing reliable, ubiquitous connectivity with high bandwidths, but pose challenges to TCP due to their high latencies.

This paper presents results from experiments on a production satellite Internet network, comparing the effects of TCP with HyStart on versus TCP with HyStart off. Also considered are the effects of TCP buffer settings at the sender and receiver, both manual settings and auto-tuned, and the impact these settings have in limiting TCP performance.

Analysis of the results shows Linux defaults for buffer settings are not sufficient to achieve more than a fourth of the available bandwidth (about 30 Mb/s out of 120 Mb/s) on the satellite link, owing to buffer limits that are reached before the link is saturated. Buffer auto-tuning is effective at adjusting the sender and receiver buffers up to their maximums, but the default maximums need to be increased to saturate the satellite link. Auto-tuning itself does not appear to limit exponential window growth during slow start.

Over our satellite link, TCP HyStart degrades performance by exiting slow start too early, making it take longer for flows to reach the available bandwidth. Start-up downloads are about 2x faster with HyStart off than with HyStart on, downloading Web-page sized objects in 7 seconds versus 14 seconds. These effects persist when TCP flows compete over the satellite link, with TCP flows that have HyStart off getting about 5x more bandwidth during starting conditions than TCP flows with HyStart on.

The insights on buffer sizes and HyStart should be useful for improving TCP performance over high-bandwidth, high-latency networks such as for satellite Internet.

There are several areas we are keen to pursue as future work. There are other settings to TCP, such as the initial congestion window, fast convergence, the β parameter for window factor decrease, and HyStart setting, that may have a significant impact on performance. For flows competing for a satellite Internet link bottleneck there, are other TCP congestion controls available in Linux that are of interest, including but not limited to bandwidth estimation-based BBR [15] and satellite-optimized Hybla [3].

References

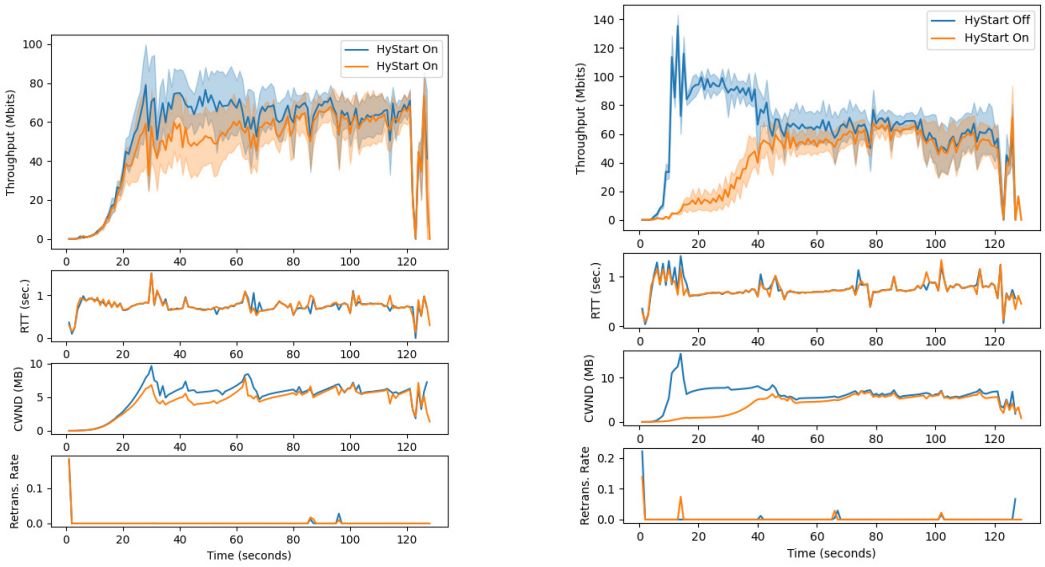
[1] Balasubramanian, P.; Huang, Y.; and Olson, M. 2020. HyStart++: Modified Slow Start for TCP. *IETF Draft draft-balasubramanian-tcpm-hystartplusplus-03*.

[2] Barakat, C.; Chaher, N.; Dabbous, W.; and Altman, E. 1999. Improving TCP/IP over Geostationary Satellite Links. In *Proceedings of GLOBECOM*.

[3] Caini, C., and Firrincieli, R. 2004. TCP Hybla: a TCP Enhancement for Heterogeneous Networks. *International Journal of Satellite Communications and Networking* 22(5):547–566.

[4] Che, X.; Ip, B.; and Lin, L. 2015. A Survey of Current YouTube Video Characteristics. *IEEE Multimedia* 22(2).

[5] Cisco. 2015. *Interface and Hardware Component Configuration Guide, Cisco IOS Release 15M&T*. Cisco Systems, Inc. Chapter: Rate Based Satellite Control Protocol.



(a) HyStart On versus HyStart On

(b) HyStart On versus HyStart Off

Figure 8: Simultaneous flows

[6] Claypool, S.; Chung, J.; and Claypool, M. 2021. Comparison of TCP Congestion Control Performance over a Satellite Network. In *Proceedings of the Passive and Active Measurement Conference (PAM)*.

[7] Data and Analysis. 2018. Webpages Are Getting Larger Every Year, and Here's Why it Matters. Solar Winds Pingdom. Online at: `https://tinyurl.com/y4pjrvhl`.

[8] Dong, M.; Li, Q.; Zarchy, D.; Godfrey, P. B.; and Schapira, M. 2015. PCC: Re-architecting Congestion Control for Consistent High Performance. In *Proceedings of the 12th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*.

[9] Everts, T. 2017. The Average Web Page is 3 MB. How Much Should We Care? Speed Matters Blog. Online at: `https://speedcurve.com/blog/web-performance-page-bloat/`.

[10] Ha, S., and Rhee, I. 2008. Hybrid Slow Start for High-Bandwidth and Long-Distance Networks. In *International Workshop on Protocols for Fast Long-Distance Networks*.

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

[12] Ha, S.; Rhee, I.; and Xu, L. 2008. CUBIC: A New TCP-Friendly High-Speed TCP Variant. *ACM SIGOPS Operating Systems Review* 42(5).

[13] Jain, R. 1991. *The Art of Computer Systems Performance Analysis: Techniques for Experimental Design, Measurement, Simulation, and Modeling*. John Wiley and Sons, Inc.

[14] Li, N.; Tu, Y.; and Deng, Z. 2019. Satellite network oriented tcp slow start algorithm. In *IEEE International Conference on Communication Technology (ICCT)*.

[15] N. Cardwell and Y. Cheng and C. S. Gunn and S. H. Yeganeh and Van Jacobson. 2017. BBR: Congestion-based Congestion Control. *Communications of the ACM* 60(2):58–66.

[16] Obata, H.; Tamehiro, K.; and Ishida, K. 2011. Experimental Evaluation of TCP-STAR for Satellite Internet over WINDS. In *Proceedings of the International Symposium on Autonomous Decentralized Systems*.

[17] Satellite Industries Association. 2020. Introduction to the Satellite Industry. Online presentation: `https://tinyurl.com/y5m7z77e`.

[18] Utsumi, S.; Muhammad, S.; Zabir, S.; Usuki, Y.; Takeda, S.; Shiratori, N.; Katod, Y.; and Kimb, J. 2018. A New Analytical Model of TCP Hybla for Satellite IP Networks. *Journal of Network and Computer Applications* 124.

[19] Wang, Y.; Zhao, K.; Li, W.; Fraire, J.; Sun, Z.; and Fang, Y. 2018. Performance Evaluation of QUIC with BBR in Satellite Internet. In *Proceedings of the 6th IEEE International Conference on Wireless for Space and Extreme Environments (WiSEE)*.

[20] Zong, L.; Bai, Y.; Zhao, C.; Luo, G.; Zhang, Z.; and Ma, H. 2020. On Enhancing TCP to Deal with High Latency and Transmission Errors in Geostationary Satellite Network for 5G-IoT. *Hindawi Security and Communication Networks* 2020(6693094).