A Swifter Start For TCP

Craig Partridge, D. Rockwell, Mark Allman, R. Krishnan, James Sterbenz

BBN Technologies

Abstract

While TCP is capable of adapting its data rate to almost any capacity, it has long been known that TCP takes a long time to achieve full data rates on paths with high capacity (especially links with long delays, e.g. satellite links). In this paper we present a new way for TCP to estimate available network capacity and swiftly scale its transmission rate at the start of a TCP connection. While the method does estimate capacity, as predicted by prior simulation [12], the limited studies reported in this paper suggest it performs slightly worse than regular TCP over a low-delay modest-bandwidth Internet path.

1. Introduction

When a Transmission Control Protocol (TCP) [16] connection is opened and data transmission starts, TCP uses an algorithm known as *slow start* to probe the network to determine the available capacity over the connection's path [9, 2]. The slow start algorithm probes the network by sending between 50% and 100% more data in each round-trip transfer across the network. A consequence of this growth pattern is that it takes TCP approximately $\log_{1.5} x$ to $\log_2 x$ round-trips to find the right data rate, where x is the available capacity and is computed as the round-trip time multiplied by the available bandwidth.

On a common terrestrial link (low-delay, modest-bandwidth) the time it takes TCP to correctly estimate and begin transmitting data at the available capacity is only a few round-trips. However, over links that are long (e.g. satellite links) or have high bandwidth (several hundred megabits per second), or both, it may take TCP several seconds to complete the first slow start. Indeed, for transactional applications such as HTTP, the TCP connection may have transferred all the data to be sent before completing the slow start, without ever achieving its potential transmission rate. Table 1 illustrates this point by showing the amount of data sent and the effective data rate over terrestrial links with 100 ms of delay (about the round-trip time across North America) at various rates during the slow start phase.[1]

TCP connections often share a link, and thus the question of how fast a single connection can scale up to fill an entire link (which is what Table 1 shows) is a somewhat artificial question. But the larger message from Table 1 is that TCP's ability to scale its data rate swiftly at startup is not as dynamic as we would like as we enter an era of long delay-bandwidth product paths.[2]

This work was funded under contract number NA53-99175 by NASA Glenn Research Center and NASA's Earth Science Technology Office.

[1] Amounts of data computed assume a 1024 byte MTU and 984 byte effective payload after TCP/IP headers, that the TCP implementations use delayed acknowledgements, starts with a four segment congestion window, and increase the window by one segment per acknowledgement (e.g., the window grows by a factor of 1.5 per round-trip). The window size is assumed to be arbitrarily large (note that this is somewhat inconsistent with the 984 byte effective payload as congestion windows larger than 64KB require TCP options that reduce the effective payload size). The number of bytes sent is the number at the end of the round-trip cycle in which slow-start ends (i.e., it will somewhat overestimate the total).

[2] We are entering an era of long delay-bandwidth paths because the Internet path delays are already close to the speed of light delay and the bandwidth of links continues to go up. As a result, over the coming years, the delay-bandwidth product of terrestrial links will come increasingly close to the delay-bandwidth products of satellite links today.

In this paper we seek to mitigate the startup problem (in particular we limit the start up delay to no more than ten round-trips in most cases) using a startup algorithm that combines two techniques: packet pair [11] (to estimate capacity) and packet pacing (to meter packets to establish initial self clocking).

2. Prior Work

The problem of getting TCP to start faster has been a pressing issue for over a decade. The challenge is that we know enough about network dynamics to say that allowing a TCP connection to begin sending at a data rate substantially in excess of the available capacity is usually harmful: it hurts the other TCP connections along the path and often leads to crippling loss for the new TCP, hindering startup even more than starting slowly [9].

So, the first problem is quickly and correctly estimating the capacity so that the TCP connection can send more data early and still does not send more data than its relevant share of the available capacity.

Simply estimating, however, the available capacity is not enough. TCP relies on the arrival of acknowledgements of its data to properly separate in time, or self-clock, its data transmissions [9]. If a TCP connection somehow correctly estimates its fair share of capacity but then sends all its data in one burst, the burst will often overwhelm router queues and lead to loss that, again, harms the connection's throughput. So we need some mechanism to pace out segments in a way that does not immediately overwhelm intermediate router queues.

This section surveys the work to date on both problems.

2.1. Slow Start

When a TCP connection is getting started, it has no estimate of the bottleneck rate nor does it know when to place its data on the network (the self-clocking mechanism noted above). So, TCP uses an algorithm called ''slow start'' to both find the bottleneck rate and establish self-clocking.

The basic idea behind slow start is that, during the slow start stage, the sending TCP transmits both the amount of data that is acknowledged plus an additional, maximum-sized, TCP segment on receipt of every acknowledgement. Depending on whether the receiving TCP acknowledges every data segment or every other data segment, this means that the sending TCP is transmitting at 1.5 to 2 times the data rate being acknowledged.

Another feature of slow start is that because the sending TCP transmits its data segments back-to-back, it is transmitting packets in a pattern similar to packet pair. Because data packets get spaced by the bottleneck link, the spacing of their acknowledgements roughly approximates the data rate of the bottleneck link. If the TCP simply transmitted one segment for every acknowledgement, it would be correctly pacing (without, unfortunately, growing the window). However the sending TCP performing slow start is injecting data at 1.5 to 2 times the data rate of the bottleneck link. A consequence of sending at twice the bottleneck rate is that the router at the bottleneck link develops a queue.[3] This phenomenon is illustrated in Figure 1. In response to two acknowledgements (bottom) spaced according to the bottleneck rate, a sender sends four segments (shown at left). At the bottleneck, two of these segments get placed on the wire at the bottleneck rate, while two are forced to sit in the queue.

[3] For simplicity, through this discussion we will talk about a single bottleneck link and a single bottleneck router. In fact, there may be a series of relative bottlenecks, where the next link has less capacity than its predecessor, and a series of bottleneck routers. In this case, the analysis still holds, but the extra segments are buffered at multiple points in the network.

Table 1 - Slow Start Round-Trips and Data Sent for various data rates					
	1.5Mb/s	45Mb/s	155Mb/s	1Gb/s	10Gb/s
Round-Trips	5	14	17	21	27
Bytes	53,136	2,414,736	8,179,008	41,465,760	472,490,232
Data Rate	850Kb/s	14Mb/s	38Mb/s	158Mb/s	1,400Mb/s

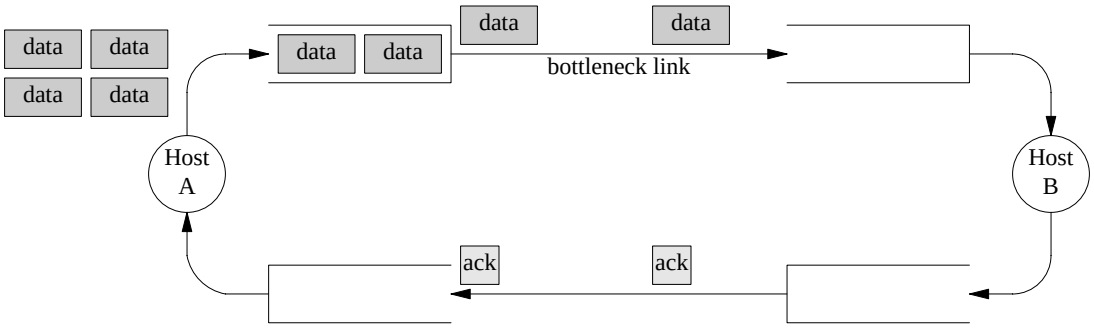

Figure 1: Slow Start Sends at Double Capacity

It is useful to examine the bottleneck queue over the course of the TCP connection's round-trip time. In slow start, over the first part of the round-trip time, the sending TCP is sending in excess of the bottleneck rate and the queue is filling with the excess TCP segments. At some point the sending TCP stops sending segments, as it has now received all the acknowledgements from the prior round-trip's window of data. At this point the excess segments begin to drain from the queue, until they have all been forwarded. The queue then remains empty (of this connection's segments) until the next round-trip cycle. See [12] for extensive discussion of this behavior.

After some number of round-trip cycles, the burst of TCP traffic at the start of the round-trip will exceed the queuing space at the bottleneck router and the router will drop one or more segments. TCP interprets this segment loss as an indication it is overdriving the bottleneck link, and TCP shifts into its steady-state sending mode (called congestion avoidance).

In general, this process works well and causes TCP to quickly find the bottleneck data rate and establish self-clocking. Slow start, however, typically fails if the bottleneck is a long delay, high bandwidth, link such as a satellite channel [15, 12]. The reason is that slow start requires the bottleneck router to have buffering equal to the product of the delay and the bandwidth. If the link has both high delay and high bandwidth, the bottleneck router typically has too little buffering. The result is that the sending TCP will overwhelm the router queue long before it reaches the bottleneck data rate, and mistakenly set its target data rate for congestion avoidance too low. Over time, this data rate gets corrected, as TCP repeatedly probes the network and discovers additional capacity is actually available.

2.2. Estimating Capacity

So if slow start sometimes does not find the correct bottleneck bandwidth at the start of the connection, what alternative ways to estimate a bottleneck bandwidth are there? To date there's only one known approach: packet pair [11].

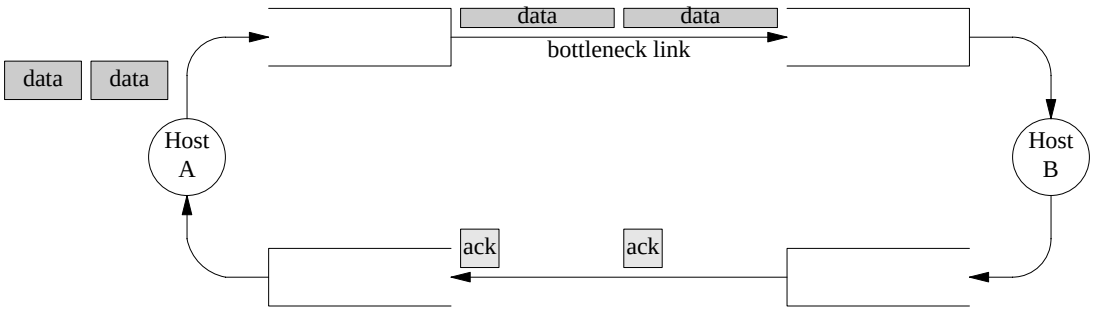

Figure 2: Packet Pair Estimation

The idea behind packet pair is illustrated in Figure 2. A host (in this case, Host A) sends two back-to-back data packets. At the bottleneck link in the path, these packets get separated in time. As the packets are received, they are acknowledged. If the acknowledgements are sent promptly, the spacing of the acknowledgements reflects the spacing of the data packets. Thus, by measuring the time between the arrivals of the acknowledgements (and knowing the size of the data packets being acknowledged), the sender can estimate the available capacity.

TCP already implicitly uses acknowledgement spacing to estimate capacity. In steady state, TCP only injects new data when an acknowledgement is received. So the transmissions of new data are implicitly sent at the rate of the bottleneck link.

2.3. Problems with Packet Pair

Unfortunately, a number of problems can creep into packet pair bandwidth estimation in real networks. For instance, a phenomenon called ‘‘ACK compression’’ [17, 14] causes acknowledgments to bunch up in the network path between the data receiver and the data sender. When these compressed ACKs are used for packet pair, an overestimate of the bandwidth is obtained because the sender does not see the full ‘‘spreading effect’’ of the data packets. More generally [3], shows that the ACK stream also imposes jitter in the other direction. That is, the ACKs are further spread out on the path to the data sender, yielding an underestimate of the bandwidth. Another challenge is the delayed ACK timer which can add time to the generation of an ACK (which appears to increase the spreading of the packets, but is really a host-introduced event).

Unfortunately the problems of packet pair estimation cannot wholly be explained by fluctuations in the ACK stream. Reference [3] shows that changes in the spacing of the data packets after the bottleneck occur. So, even the receiver is not always able to accurately estimate the bottleneck bandwidth.

We take some precautions against inaccurate bandwidth estimates in the algorithm we use for capacity estimation (discussed further in section 3).

2.4. Pacing

A steady state TCP uses its acknowledgements to clock out new data at the bottleneck rate (a mechanism often called self-clocking). And a TCP currently uses slow start to establish ack clocking at the start of the connection. But if we have used packet pair to quickly estimate the available capacity at the start of the connection, how does the TCP properly space its transmissions?

One solution to this problem is to have the sending TCP pace out its segments so that they don’t come bunched at the start of the round-trip cycle. Broadly, pacing seeks to space the transmission of segments roughly evenly over the entire round-trip time. (With some emphasis on roughly − issues of clock granularity often make precise spacing difficult). Beyond eliminating the bunching of packets in slow start [12], pacing has also been used experimentally to avoid ack compression [7], and to improve web performance [7].

Unfortunately, the most recent study of pacing [1] also shows that TCPs that use pacing

for the entire connection do not perform as well as regular, non-paced TCPs. While the exact reasons are not clear, the authors of the study believe that pacing can suffer synchronization effects, where multiple flows pacing through a bottleneck end up synchronizing their behavior. Also [1] notes that by pacing out the packets a sending TCP apparently increases its chance of suffering congestion loss, vis-a-vis regular TCPs. Part of the benefit of TCP's self-clocking is that the acknowledgements reflect when space will be available in queues in the path, and pacing loses this information.

The overall message of the work on pacing is that pacing can help, but is not a complete replacement for TCP's self-clocking mechanism.

2.5. Increasing the Initial Window Size

One other piece of work is of interest in this context. The Internet Engineering Task Force is considering revising the TCP specification to recommend sending up to four TCP segments in the first round-trip, rather than just one [4]. The idea is to open the window faster. Another consequence is that by sending a volley of four segments, a TCP can do a packet pair analysis at the end of the first round-trip.

3. A Swifter Startup Algorithm

The idea we explore in this paper is to combine packet pair, to estimate *initial* capacity only, with pacing to spread out the initial burst until self-clocking is established. The idea is to take the best of both packet pair and pacing, while limiting our use of each algorithm to avoid their limitations.

3.1. The Basic Idea

In this approach, a sending TCP begins with the four-segment burst of packets to start the connection. However, when the acknowledgements of the segments are received, the sending TCP performs the packet pair algorithm and estimates the available bandwidth in the path.[4] The sending TCP then computes the estimated delay×bandwidth product by multiplying the estimated available bandwidth by the measured round-trip time. A fraction of this estimate (in our implementation, a configurable value γ between 1 and 1/8th) is then used as the congestion window size for the next round-trip time. More formally, the computation is:

$$BW = \frac{(t_{ack2} - t_{ack1})}{SegSize}$$

$$Capacity = BW \times (t_{ack1} - t_{seg1})$$

where t_{ackX} is the time the acknowledgement of segment X is received and t_{segX} is the time when segment X was sent.

The reason for taking a fraction of the estimated delay×bandwidth product is to protect against an over-estimate from the packet pair algorithm. Note that because TCP effectively increases the window by about a factor of two each RTT during slow start, if the packet pair estimate is correct TCP will be sending at the correct rate within three to five round-trip times. (Recall that on a typical high delay×bandwidth path, it could take TCP fifty or more round-trips to reach the full data rate). At the same time if the packet pair estimate is too large, by choosing a fraction, we have reduced the chance that TCP will not overdrive the link.

[4] Actually the computation is a bit more complex. In general, because most TCP implementations seek to send one acknowledgement for every two segments, when we compute the packet-pair value, we should divide the size of two segments by the inter-ack time. However, because an ACK may be triggered by one segment (or due to a delayed-ack timer expiring between receptions of segments at the receiver), we compute the available bandwidth by dividing the size of one segment. So we would expect to underestimate the available bandwidth by at least a factor of two. For details on how delayed acknowledgements may be implemented to reduce acknowledgements see [5, 2].

Table 2 - Swift Start Round-Trips and Data Sent for various data rates					
	1.5Mb/s	45Mb/s	155Mb/s	1Gb/s	10Gb/s
Round-Trips $\gamma = 2$	4	4	4	4	4
Bytes	47,232	1,337,256	4,603,152	29,690,232	296,877,720
Data Rate	945Kb/s	27Mb/s	92Mb/s	594Mb/s	5,938Mb/s
Round-Trips $\gamma = 8$	5	7	7	7	7
Bytes	53,136	1,469,112	5,040,048	32,467,080	324,722,952
Data Rate	850Kb/s	17Mb/s	58Mb/s	371Mb/s	3,711Mb/s

After calculating the available capacity we employ pacing. Because the new window size is potentially large (hundreds of segments), we do not want to inject them as a single burst. Such a burst would likely overwhelm intermediate queues and disrupt other traffic. Rather we would like to space the burst and try to get TCP to begin self-clocking. So we pace out the segments over roughly one round-trip time. We continue pacing (but growing the window normally, at 1.5 to 2 times per round-trip) until the first slow-start ends, after which the sending TCP behaves according to the standard TCP algorithms.[5]

3.2. Comments on the Idea

The goal of this scheme is to make the best possible use of packet pair and pacing to allow a TCP to rapidly open its window on a link with a large delay×bandwidth product, while fundamentally keeping TCP working much as we know (and understand) it today.

The scheme makes effective use of packet pair by estimating the available capacity, but avoids much of the imprecision of the packet pair estimate by taking a fraction of the estimate as the starting point of the slow start process. Similarly, the scheme uses pacing to space out the initial burst, but then allows the more effective TCP self-clocking to take over.

[5] An alternative would be to do swift start for only the first round-trip after doing the packet pair estimation.

3.3. Theoretical Improvement

Before discussing experience, we can look at how this scheme would work in theory. Table 2 (like Table 1) looks at the start up effects over a 100ms network path and shows how long it would take (in theory) to increase the transmission rate to full rate using this swifter starting mechanism, assuming packet pair gave us the correct estimate of the available bandwidth.[6] We report all results for two values of γ, namely $\gamma = 2$ and $\gamma = 8$. ($\gamma = 1$ isn't interesting in theory, though it is often useful as a test case, as it says that all connections are running at full rate after the second RTT).

Comparing Table 2 with Table 1 highlights several useful points. First, the startup time remains long. While the startup time is now bounded to no more than seven round trips (compared with over 20 round trips for 1 Gb/s and 10 Gb/s in Table 1), there is still a notable startup transient. The amount of data sent during the startup phase also remains large: typically about 60% of what would be sent under standard slow start. However, the average data rate during the startup is higher, around half the link data rate which suggests we may make reasonably good use of the path.

[6] There's one minor difference between Table 1 and 2. In Table 1, we always report the number of round-trips as being the round-trip in which the slow start ended. In Table 2, for $\gamma = 8$, it turns out that slow start ends a few packets into the eighth round-trip. That is, slow start continues for the first 2% of packets sent in the eighth round-trip. To avoid warping the results, we chose to treat slow start as ending at the very end of round-trip seven.

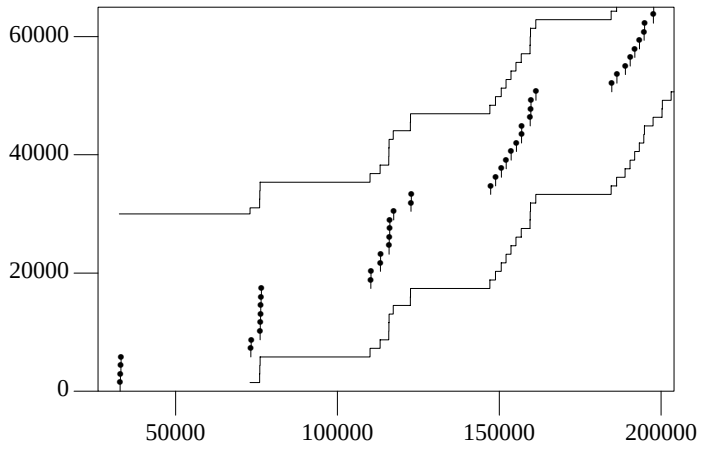

Figure 3: Slow Start Transition

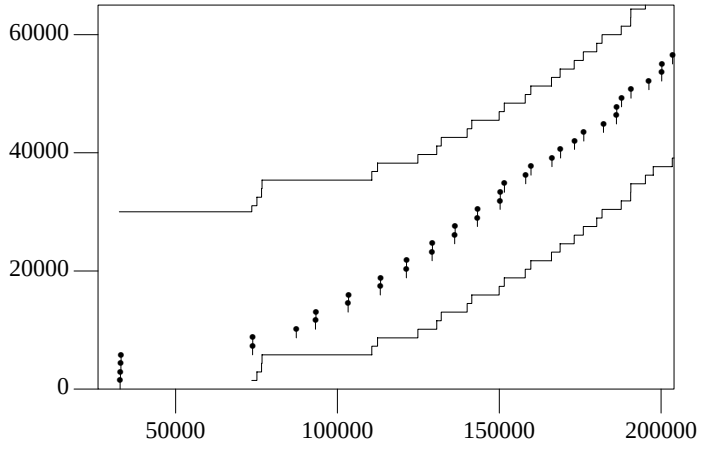

Figure 4: Swift Start Transition

4. Testing an Implementation

We have implemented swift start in FreeBSD (version 4.1) and tested its behavior over the Internet. In this section, we report on some of those experiences.

4.1. Experimental Design

All tests were run over a path between a system using the swift start algorithms located at BBN Technologies in Boston, MA, and a machine running NetBSD (version 1.3.3) at NASA Glenn Research Center in Cleveland, OH. The path had 19 hops, a minimum round-trip of just under 32 ms, and a bottleneck bandwidth of 3.5 Mbps. The resulting window size is approximately 17KB.[7] The segment size was 1472 bytes (1512 bytes including headers). The receiver offerred a window size of 32KB.

Note that this test environment is relatively difficult for swift start. The number of hops is large, so there are plenty of opportunities for ack spacing to be distorted. The bandwidth is fairly modest, so small errors in capacity estimation may cause packet pair to sharply overestimate capacity and overdrive the link. Finally, there's a

[7] Statistics about the path were determined using `traceroute` and `pathchar` as discussed below.

considerable amount of buffering in the path, which reduces the chance of loss when slow start sends bursts. This path is clearly not the high delay-bandwidth, low buffering path for which swift start was designed.

For each set of tests, we initiated a long TCP data transfer (10 MB) to see the startup behavior and then observe TCP transition into steady state behavior. A range of values were used for γ, however, for two reasons the implementation was unable to pace more than five to seven segments in the first round-trip, so only a few values of γ are of interest (see discussion below). Before and after each transfer, `traceroute` [10] or `pathchar` [8] was run to confirm that the path did not change in any significant way between runs and to detect major changes during a run. The path had little loss. Also, a stock TCP transfer was run every fourth connection to ensure we always had a stock TCP connection taken close in time to any swift start measurement, again as protection against the path behavior varying over time. In fact, the path did not change appreciably throughout the runs and we believe that all the test runs presented here are comparable. Tests at each γ were run multiple times. Characteristic plots (i.e. the results of single runs) are presented here.

4.2. Initial Example

Before making performance comparisons, it is useful to step through an example to illustrate the different behaviors of slow start and swift start.

Figure 3 illustrates a standard slow start.[8] The sending implementation uses a four segment initial congestion window, so we see an initial transmission of four segments. The receiver acknowledges each segment as received, so as each acknowledgement comes back, we see two segments transmitted. The result is the typical slow-start pattern of (vertical) bursts separated by approximately one round-trip time each.

Figure 4 illustrates a typical swift start (in this case with $\gamma = 1$). Like slow-start, the connection starts with a four-segment burst. When the first acknowledgement comes back, the sender duly transmits two segments. When the second acknowledgement comes back, the sender does the packet pair computation and paces out the segments.

The estimated capacity should be about 12 segments. However, the pacing implementation places some additional limits on what can be sent. In particular, it has limits on clock granularity and it sends no more than two segments per transmission (to avoid bursts). As a result, with 32ms round-trip, pacing is limited to around five to seven segments in the first round-trip.

The most interesting feature of Figure 4 is that the segments are indeed evenly spaced over the round-trips. The swift-start TCP is well on its way to self-clocking at a time that the slow-start TCP is still being bursty.

4.3. Slow Start, Pacing and Swift Start

Having looked at slow start and swift start behavior at the beginning of a TCP connection, we can now move to look at their effects over the entire life of a connection.

An initial comment is in order, before looking at details. Only about a third (15 of 44) data transfers we conducted experienced loss. That is, there was often enough buffering in the path to buffer the full TCP window size (32KB) even though the window size exceeded the available capacity. In no case was the congestion loss more than one segment -- so TCP fast retransmit generally recovered after one segment loss. In one case (illustrated in Figure 5) two segment losses occurred close enough together that TCP was forced to wait for a timeout.

[8] The format of the figures shown here was developed by Jacobson [9]. The vertical lollipop shapes each represent a packet transmission (the length of the vertical line represents the number of bytes sent). The y-axis is the sequence space in bytes. The x-axis is time (in ms). The solid line along the bottom is the left edge of the TCP window and represents the value of the most recently acknowledged byte of data. The solid line along the top is the right edge of the TCP window and represents the value of the highest byte (in the sequence space) that currently may be sent.

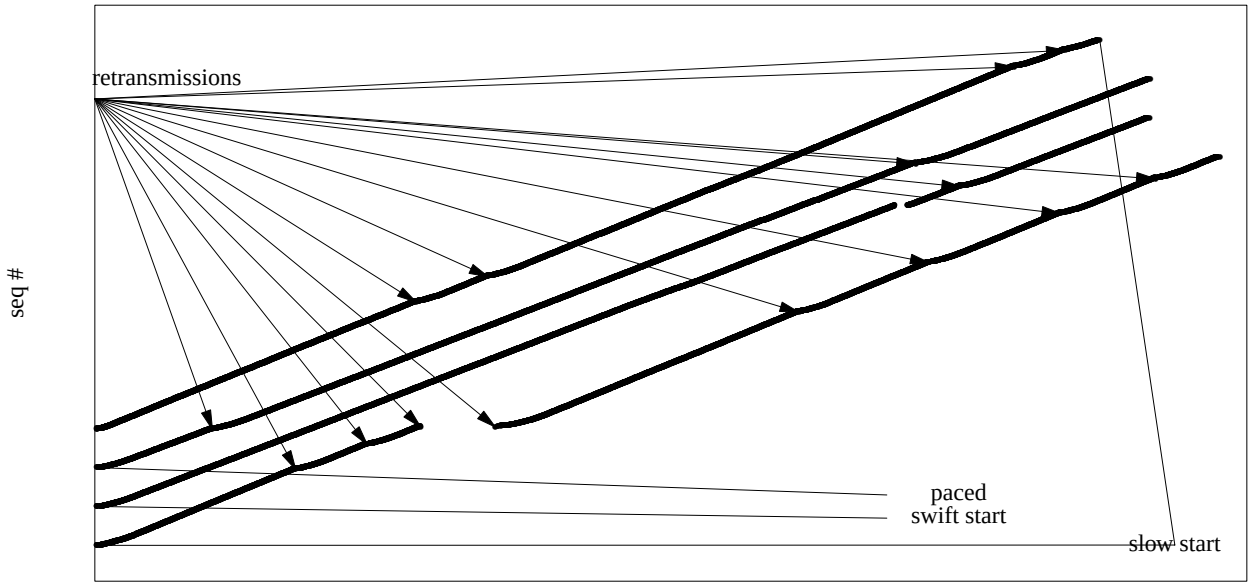

Figure 5: Swift-Start, Pacing and Slow-Start

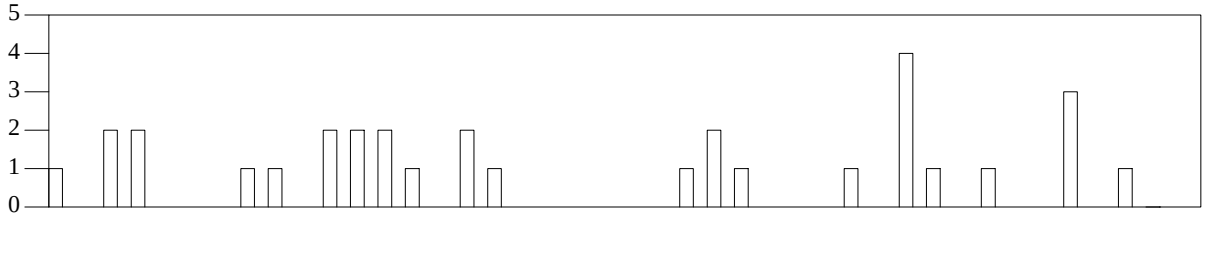

Figure 6: Sequence Space vs. Losses

Another interesting point is that almost all the losses were suffered by standard TCP doing slow start and slow start was far more likely to cause losses in the first third of the connection's lifetime. That is, as observed by [12, 15], the burstiness of slow start was more likely to overrun router buffers. Paced TCPs tend to experience loss about two-thirds of the way into the connection's lifetime (when slow start also tends to suffer losses).

Figures 5 and 6 illustrate these points. Figure 5 plots the entire connection lifetime for four TCP connections: the two slow-start connections that suffered the most loss, one of the swift-start connections ($\gamma = 1$) that suffered the most loss and, for comparison, a TCP that paces but does not do capacity estimation (i.e., just follows the slow start algorithm of doubling the congestion window every round-trip time). The points that retransmissions occur are marked.[9]

Figure 6 plots (across all tests) the number of losses that occur at particular spots in the sequence space.[10] Note that while Figures 5 and 6 have different x-axes (Figure 5 is time, Figure 6 is sequence number), the two are closely related and so the loss pattern in Figure 6 is reflected in the retransmissions in Figure 5.

[9] There's a point on the swift-start graph (about 70% through the connection) where a gap suggests a loss. The raw data, however, shows that no retransmission occurred − there was simply an anomalous pause of nearly two round-trip times in receiving the acknowledgement.

[10] Losses are binned into 250K bytes intervals.

Another observation from Figure 5 is that even though it suffered three more losses, one of the two slow-start connections finished the data transfer before either the paced or swift-start TCP. Indeed, despite suffering eight losses and a two round-trip pause, the slowest slow-start connection only finished about two round-trips after the other TCPs. Those results are indicative of a broader trend we observed. In almost all cases, a slow start TCP completes the transfer before a swift start TCP − typically taking between 2% and 10% less time. Figure 7 illustrates this point by graphing two connections, one slow start, one swift start, neither of which sufferred loss. (Note the differing slopes of the two connections).

5. Discussion

The goal of the project that funded this report was to implement a swift start TCP, based on the promising simulation results of [12]. This work was not intended to be a thorough study of how swift start works in the real Internet. Such a test would require more extensive testing over a more diverse range of paths (in particular, high-delay bandwidth paths and more heavily congested paths). With those points having been said, however, the limited tests of the previous section raise several interesting questions.

First, there is every reason to believe that swift start does indeed get the benefit of pacing: namely router queues are not overloaded during the initial phase of the TCP connection. Whereas around two-thirds of slow start TCP connections experience a loss (signalling an overloaded router queue) in the first third of their connection lifetimes, less than a quarter of the swift start TCP connections suffer a loss. The overall loss rates of the two types of connections are also different (12 losses in total over 22 connections for swift start vs. 14 losses in 12 connections for slow start).

Swift start, however, consistently underperforms slow start over a connection lifetime. This result is contrary to what we would expect from our prior work [12], but is consistent with the work of Aggarwal *et al* [1]. However, Aggarwal *et al.* were unable to explain why pacing did not perform as well.

Before seeking insight, we should recognize that there are probably bugs in our implementation. The implementation is fairly new and probably has some misfeatures. However, the implementation is doing pacing, and even though it caps the data that can be sent in the first paced round-trip too low, it sends as much data as slow start during that time. An implementation problem is, therefore, an unlikely explanation for the performance gap.

A more convincing explanation is clock granularity. The pacing implementation is limited in how it schedules bursts by the one ms granularity of the clock it uses to schedule paced bursts. So a segment that should be transmitted at 0.5 ms may be transmitted at 1 ms. Effectively the implementation is very slightly increasing the round-trip time during the swift start phase. It is not clear, however, why this increase should continue to affect the connection after the swift start phase is over.

Overall the testing experience is yet another testament to the remarkable versatility and tenacity of slow start. It is not easy to do better than slow start over a general range of Internet conditions.

6. Future Work

Having an implementation of swift start enables a wide range of experiments over the Internet. In this section we sketch some of the interesting problems.

First and foremost, it would valuable to test swift start over the long delay-bandwidth paths for which it is was designed. Does it achieve the higher transfer rates that simulation predicts? And if it does not, why not?

It would also be valuable to obtain more traces over a variety of paths to understand why swift start performed less well than slow start on the path we used for testing. Our tests revealed some interesting questions. For instance, given that swift start suffered fewer loss events on average than slow start and that loss is a signal to a connection to slow down, why is it that slow start completes its data transfers first? How serious are the clock granularity issues, and how do they affect performance?

Finally, there are questions of how swift start interacts with other TCP mechanisms. For instance, we hypothesize that in cases where multiple segments are discarded, pacing will spread out the losses over the entire window. We contrast this loss pattern with slow start where losses tend to come in contiguous segments. How will this spreading affect TCP's recovery from loss?

7. Code Availability

The code used for these tests is available at `http://www.ir.bbn.com`.

8. Acknowledgements

We would like to gratefully acknowledge help from Steve Polit and Will Ivansic.

References

1. Aggarwal, A., S. Savage, and T. Anderson, "Understanding the Performance of TCP Pacing," *Proc. 2000 IEEE INFOCOM Conference,* Tel-Aviv, Israel (March 2000).
2. Allman, M., V. Paxson, and W. Stevens, "TCP Congestion Control; RFC-2581," *Internet Requests for Comments,* 2581 (April 1999).
3. Allman, M., and V. Paxson, "On Estimating End-to-End Network Path Properties," *Proc. ACM SIGCOMM '99* (September 1999).
4. Allman, M., S. Floyd, and C. Partridge, "Increasing TCP's Initial Window; RFC-2414," *Internet Requests for Comments,* 2414 (September 1998).
5. Braden, R.T., "Requirements for Internet Hosts -- Communication Layers; RFC-1122," *Internet Requests for Comments,* 1122 (October 1989).
6. Aron, M., and P. Druschel, *TCP: Improving Startup Dynamics by Adaptive Timers and Congestion Control (Rice TR98-318),* Rice University Computer Science (1998).
7. Balakrishnan, H., V. N. Padmanabhan, and R. H. Katz, "The Effects of Asymmetry on TCP Performance," *ACM Mobile Networks and Applications (MONET),* 4, 3, pp. 219-241 (1999).
8. Downey, A.B., "Using pathchar to estimate Internet link characteristics," *Proc. ACM SIGCOMM '99* (August 1999).
9. Jacobson, V., "Congestion Avoidance and Control," *Proc. ACM SIGCOMM '88,* pp. 314-329, Stanford, CA (August 1988).
10. Jacobson, V., *Presentation at MSRI* (April 1997).
11. Keshav, S., "A Control-Theoretic Approach to Flow Control," *Proc. ACM SIGCOMM '91,* pp. 3-16, Zurich, Switzerland (August 1991).
12. Kulik, J., R. Coulter, D. Rockwell, and C. Partridge, "Paced TCP for High Delay-Bandwidth Networks," *IEEE Workshop on Satellite Based Information Systems,* Rio de Janeiro (December 1999).
13. Lai, K., and M. Baker, "Nettimer: A tool for Measuring Bottleneck Link Bandwidth," *Proc. 3rd USENIX Symposium on Internet Technologies and Systems* (March 2001).
14. Mogul, J.C., "Observing TCP Dynamics in Real Networks," *Proc. ACM SIGCOMM '92,* pp. 305-317, Baltimore, MD (August 1992).
15. Partridge, C., and T. Shepard, "TCP/IP Performance Over Satellite Links," *IEEE Network Magazine,* 11, 5, pp. 44-49 (September 1997).
16. Postel, J., "Transmission Control Protocol; RFC 793," *Internet Requests for Comments,* 793 (September 1981).
17. Zhang, L., S. Shenker, and D.D. Clark, "Observations on the Dynamics of a Congestion Control Algorithm: The Effects of Two-Way Traffic," *Proc. ACM SIGCOMM '91,* pp. 133-148, Zurich, Switzerland (August 1991).

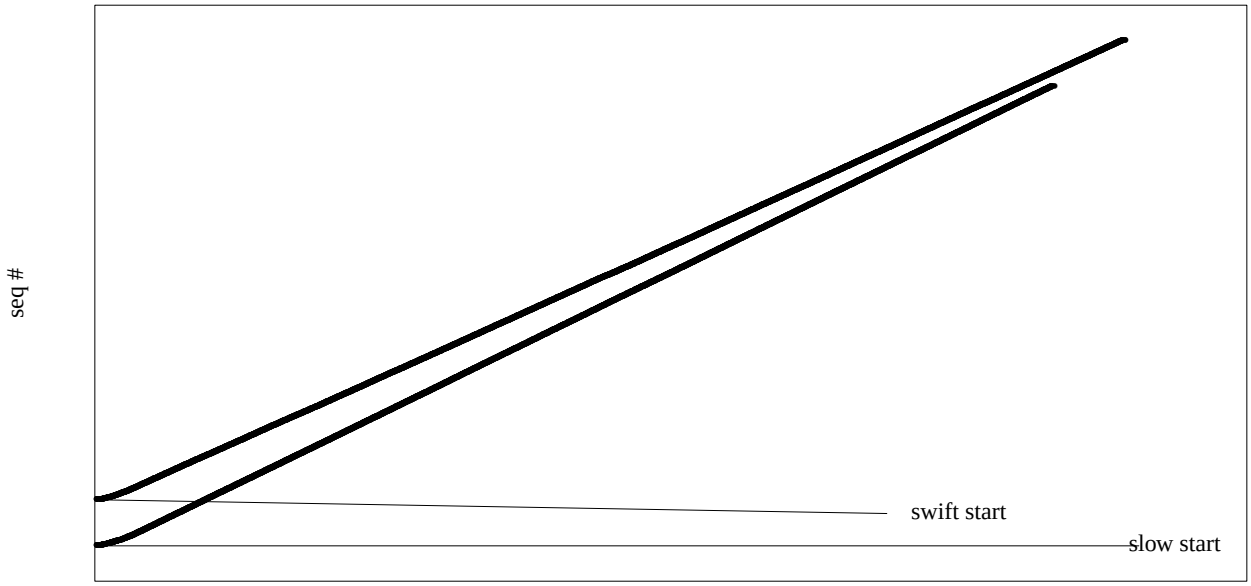

Figure 7: Swift Start and Slow Start without Loss