Improving TCP Startup Performance using Active Measurements: Algorithm and Evaluation

Ningning Hu
Computer Science Department
School of Computer Science
Carnegie Mellon University, PA, USA
hnn@cs.cmu.edu

Peter Steenkiste
Computer Science Department and
Department of Electrical and Computer Engineering
Carnegie Mellon University, PA, USA
prs@cs.cmu.edu

Abstract

TCP Slow Start exponentially increases the congestion window size to detect the proper congestion window for a network path. This often results in significant packet loss, while breaking off Slow Start using a limited slow start threshold may lead to an overly conservative congestion window size. This problem is especially severe in high speed networks. In this paper we present a new TCP startup algorithm, called Paced Start, that incorporates an available bandwidth probing technique into the TCP startup algorithm. Paced Start is based on the observation that when we view the TCP startup sequence as a sequence of packet trains, the difference between the data packet spacing and the acknowledgement spacing can yield valuable information about the available bandwidth. Slow Start ignores this information, while Paced Start uses it to quickly estimate the proper congestion window for the path. For most flows, Paced Start transitions into congestion avoidance mode faster than Slow Start, has a significantly lower packet loss rate, and avoids the timeout that is often associated with Slow Start. This paper describes the Paced Start algorithm and uses simulation and real system experiments to characterize its properties.

1. Introduction

At the start of a new TCP connection, the sender does not know the proper congestion window for the path. Slow Start exponentially increases the window size to quickly identify the right value. It ends either when the congestion window reaches a threshold *ssthresh*, at which point TCP converts to a linear increase of the congestion window, or when packet loss occurs. The performance of Slow Start is unfortunately very sensitive to the initial value of *ssthresh*. If *ssthresh* is too low, TCP may need a very long time to reach the proper window size, while a high *ssthresh* can cause significant packet losses, resulting in a timeout that can greatly hurt the flow's performance. Moreover, traffic during Slow Start can be very bursty and can far exceed the available bandwidth of the network path. That may put a heavy load on router queues, causing packet losses for other flows. While these problems are not new, steady increases in the bandwidth delay products of network paths are exacerbating these effects. Many other researchers have observed these problems [8][16] [13][19][28][22].

In this paper we present a new TCP startup algorithm, called Paced Start (*PaSt*), that builds on recent results in the area of available bandwidth estimation [17][20] to automatically determine a good value for the congestion window size, without relying on a statically configured initial *ssthresh* value. Several projects [17] [24][22] have observed that it is possible to gain information about the available bandwidth of a network path by observing how competing traffic on the bottleneck link affects the inter-packet gap of packet trains. Paced Start is based on the simple observation that the TCP startup packet sequence is in fact a sequence of packet trains. By monitoring the difference between the spacing of the outgoing data packets and the spacing of the incoming acknowledgements (which reflects the data packet spacing at the destination) we can obtain valuable information about the available bandwidth on the network path. Unfortunately, Slow Start ignores this information and as a result it has to rely on crude means (e.g. a static *ssthresh* or packet loss) to estimate an initial congestion window.

By using the available bandwidth information, Paced Start can quickly estimate the proper congestion window. Since it does not depend on a statically configured *ssthresh*, which tends to have the wrong value for many paths, Paced Start avoids the problems associated with a static initial *ssthresh* value. For most flows, Paced Start transitions into congestion avoidance mode faster than Slow Start, has a significantly lower packet loss rate, and avoids the timeout that is often associated with Slow Start. We present an extensive set of simulation results and real system measurements that compare Paced Start with the traditional Slow Start algorithm, as it is used in TCP Reno and Sack, on many different types of paths (different roundtrip time, capacity, available bandwidth, etc).

This paper is organized as follows. In the next section, we provide some background on TCP Slow Start and available bandwidth probing techniques. In Section 3, we present the Paced Start algorithm and briefly describe its implementation. Next, we describe our evaluation metrics and then present simulation results (Section 5) and real system measurements (Section 6) that compare the performance of Paced Start with that of the traditional Slow Start. We end with a discussion on related work and conclusions.

2. TCP Slow Start and Active Measurements

In this section, we briefly review the TCP Slow Start algorithm and available bandwidth measurement techniques.

2.1. TCP Slow Start

The TCP Slow Start algorithm serves two purposes:

- **Determine the initial congestion window size**: TCP congestion control is based on a congestion window that limits how many unacknowledged packets can be outstanding in the network. The ideal size of the congestion window corresponds to the *bandwidth-delay product*, i.e., the product of the roundtrip time and the sustainable bandwidth of the path. For a new connection, this value is not known, and Slow Start is used to detect it.
- **Bootstrap the self-clocking behavior**: TCP is a self-clocking protocol, i.e., it uses ACK packets as a clock to strobe new packets into the network [18]. When there are no segments in transit, which is the case for a new connection or after a timeout, there are no ACKs to serve as strobes. Slow Start is then used to gradually increase the amount of data in transit.

To achieve these two goals, Slow Start first sends a small number (2-4) of packets. It then rapidly increases the transmission rate while trying to keep the spacing between packets as even as possible. Principally, this is done by injecting two packets in the network for each ACK that is received, i.e. the congestion window is doubled every roundtrip time. This exponential increase can end in one of two ways. First, when the congestion window reaches a predetermined threshold (*ssthresh*), TCP transfers into congestion avoidance mode [18], i.e. the congestion window is managed using the AIMD (additive increase, multiplicative decrease) algorithm [14]. Alternatively, TCP can experience packet loss, which often results in a timeout. In this case, TCP sets *ssthresh* to half of the congestion window used at the time of the packet loss, executes another Slow Start to resume self-clocking, and then transfers into congestion avoidance mode.

Given the high cost of TCP timeouts, the first termination option is clearly preferable. However, for a new connection the sender does not have any good value for *ssthresh*, so TCP implementations typically use a fixed value. Unfortunately, this does not work well since this arbitrary value is wrong for most connections. When *ssthresh* is too small, TCP Slow Start will stop prematurely, and the following linear increase may need a long time to reach the appropriate congestion window size. This hurts user throughput. When *ssthresh* is too large, the instantaneous congestion window can be much too large for the path, which can cause significant packet loss. This hurts the network, since routers will not only drop packets of the new flow but also of other flows. This problem becomes more severe as the bandwidth-delay product increases.

In this paper, we propose a method to automatically obtain a good initial congestion window value by leveraging recent results in available bandwidth estimation. Note that our focus is on replacing the initial Slow Start of a new connection; the traditional Slow Start can continue to be used to bootstrap the self-clocking behavior after a timeout. Before we discuss our technique in detail, we briefly review related work on available bandwidth estimation.

2.2. Active Bandwidth Probing

Recently, a number of tools have been developed to estimate the available bandwidth of a network path, where the available bandwidth is roughly defined as the bandwidth that a new, well-behaved (i.e. congestion-controlled) flow can achieve. These tools include PTR [17] and Pathload [20]. While these techniques are quite different, they are based on the same observation: the available bandwidth can be estimated by measuring how the competing traffic on the bottleneck link changes the inter-packet gap of a sequence of packet train probes.

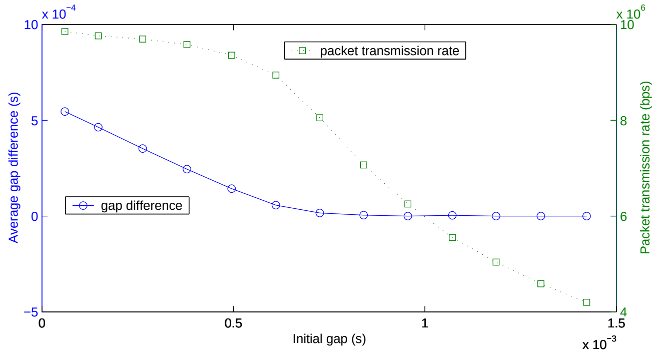
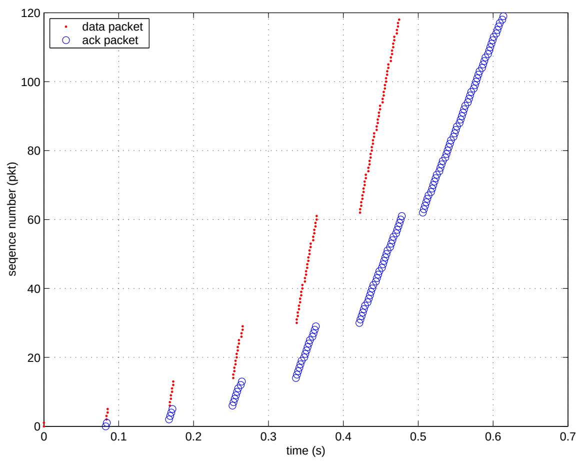


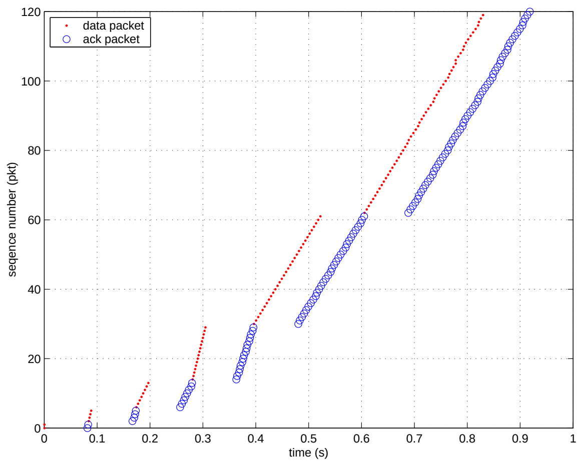
Figure 1. PTR (Packet Transmission Rate) method

Let us look at the PTR (Packet Transmission Rate) method in more detail since it forms the basis of our TCP startup algorithm. To provide some intuition, we use an experiment in which we send a sequence of packet trains with different inter-packet gaps over a 10 Mbps link in a controlled testbed. The trains compete with an *Iperf* [4] competing traffic flow of 3.6 Mbps. Figure 1 shows the difference between the average source and destination gaps (i.e. the inter-packet gaps measured on the sending and receiving hosts) and the average packet train rate as a function of the source gap. For small source gaps, we are flooding the network, and since packets of the competing traffic are queued between the packets of the probing train, the inter-packet gap increases. The gap difference becomes smaller as the source gap increases, since the lower packet train rate results in less queueing delay. When the packet train rate drops below the residual bandwidth (6.4Mbps), there is no congestion and packets in the train will usually not experience any queueing delay, so the source and destination gap become equal. The point where the gap difference first becomes 0 is called the *turning point*. Intuitively, the rate of the packet train at the turning point is a good estimate for the available bandwidth on the network path since the train is using the residual bandwidth without causing congestion. For this example, the estimate is 6.8 Mbps, which is very close to the available bandwidth of 6.4 Mbps.

The idea behind the PTR method is that the source machine sends a sequence of packet trains, starting with a very small inter-packet gap. It then gradually increases this gap until the inter-packet gap at the destination is the same as that at the source. At the turning point, the rate of the packet train is used as an estimate of the available bandwidth. An important property of the PTR method is that it is fast and efficient. On most network paths, it can get a good estimate of the available bandwidth in about 6 roundtrip times using 16-packet trains [17].



(a) Slow Start (Sack)



(b) Paced Start

Figure 2. Sequence plot for Slow Start (Sack) and Paced Start. These are from an *ns2* simulation of a path with a roundtrip time of 80ms, a bottleneck link of 5Mbps, and an available bandwidth of 3Mbps. Delayed ACKs are disabled, so there are twice times as many outgoing data packets as incoming ACKs.

3. Paced Start

This section describes how we integrate an active measurement technique into the TCP startup phase to form the Paced Start algorithm. We motivate our design, describe our algorithm in detail, and review our implementations.

3.1. Design

The idea behind Paced Start is to apply the available bandwidth estimation algorithm described above to the packet sequence used by Slow Start. Similar to PTR, the goal is to get a reasonable estimate for the available bandwidth without flooding the path. This bandwidth estimate can be used to select an initial congestion window. In this context, the "available bandwidth" is defined as the throughput that can be achieved by a new TCP flow in its stable state, since that is the value reflected in the congestion window. The available bandwidth is determined by many factors including hard-to-characterize features such as the relative aggressiveness of the flows sharing the bottleneck link. Fortunately, the TCP startup period only needs to obtain a good approximation, because, in order to be useful, it is sufficient that the initial value of the congestion window is within a factor of two of the "true" congestion window, so that TCP can start the congestion avoidance phase efficiently.

Figure 2(a) shows an example of a sequence number plot for Slow Start. We have disabled delayed ACKs during Slow Start as is done by default in some common TCP implementations, e.g. Linux; the results are similar when delayed ACKs are enabled. The graph clearly shows that Slow Start already sends a sequence of packet trains. This sequence has the property that there is one packet train per round trip time, and consecutive trains grow longer (by a factor of two) and become slower (due to the clocking). We decided to keep these general properties in Paced Start, since they keep the network load within reasonable bounds. Early trains may have a very high instantaneous rate, but they are short; later trains are longer but they have a lower rate. Using the same general packet sequence as Slow Start also has the benefit that it becomes easier to engineer Paced Start so it can coexist gracefully with Slow Start. It is not too aggressive or too "relaxed", which might result in dramatic unfairness.

The two main differences between Slow Start and Paced Start are (1) how a packet train is sent and (2) how we transition into congestion avoidance mode.

The self-clocking nature of Slow Start means that packet transmission is triggered by the arrival of ACK packets. Specifically, during Slow Start, for every ACK it receives, the sender increases the congestion window by one and sends out two packets (three packets if delayed ACKs are enabled). The resulting packet train is quite bursty and the inter-packet gaps are not regular because the incoming ACKs may not be evenly spaced. This makes it difficult to obtain accurate available bandwidth estimates. To address this problem, Paced Start does not use self-clocking during startup, but instead directly controls the gap between the packets in a train so that it can set the gap to a specific value and make the gaps even across the train. As we discuss in more detail below, the gap value for a train is adjusted based on the average gap between the ACKs for the previous train (we use it as an approximation for the inter-packet gaps at the destination). To do that, we do not transmit the next train until *all* the ACKs for the previous train have been received.

Note that this means that Paced Start is less aggressive than Slow Start. First, in Slow Start, the length of a packet train (in seconds) is roughly equal to the length of the previous ACK train. In contrast, the length of the packet train in Paced Start is based on the sender's estimate on how the available bandwidth of the path compares with the rate of the previous packet train. As a result, Paced Start trains are usually more stretched out than the corresponding Slow Start trains. Moreover, the spacing between the Paced Start trains is larger than that between the Slow Start trains. In Figure 2(b), this corresponds to a reduced slope for the trains and an increased delay between trains, respectively. Since Slow Start is widely considered to be very aggressive, making it less aggressive is probably a good thing.

Another important design issue for Paced Start is how to transition into congestion avoidance mode. Slow Start waits for packet loss or until it reaches the statically configured *ssthresh*. In contrast, Paced Start iteratively calculates an estimate for the congestion window of the path and then uses that estimate to transition into congestion avoidance mode. This typically takes three or four probing phases (RTTs), as is discussed in Section 3.2.3. If packet loss occurs during that period, Paced Start transitions into congestion avoidance mode in exactly the same way as Slow Start does.

3.2. Algorithm

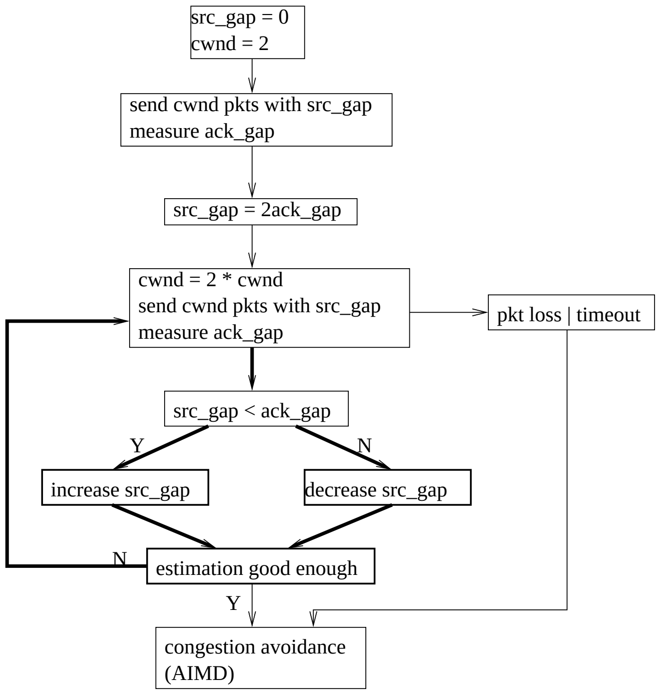


Figure 3. The Paced Start (PaSt) algorithm

The Paced Start algorithm is shown in the diagram in Figure 3. It starts with an initial probing using a packet pair to get an estimate of the path capacity B; this provides an upper bound for the available bandwidth. It then enters the main loop, which is highlighted using bold arrows: the sender sends a packet train, waits for all the ACKs, and compares the average ACK gap with the average source gap. If the ACK gap is larger than the source gap, it means the sending rate is larger than the available bandwidth and we increase the source gap to reduce the rate; otherwise, we decrease the source gap to speed up. In the remainder of this section, we describe in detail how we adjust the gap value and how we terminate Paced Start. Table 1 lists the notations we use.

Table 1. Notations

B	network path capacity
g_B	$g_B = packet\_len/B$
$\alpha \cdot B$	available bandwidth, $0 \leq \alpha \leq 1$
g_s	source packet gap
g_d	destination packet gap

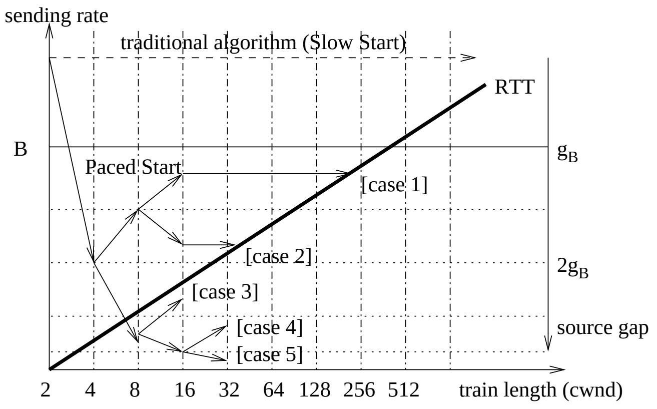


Figure 4. Behavior of different startup scenarios.

3.2.1. Gap Adjustment

Figure 4 provides some intuition for how we adjust the Paced Start gap value. The bold line shows, for a path with a specific RTT (roundtrip time), the relationship between the congestion window (x-axis) and the packet train sending rate ($1/source\_gap$). The goal of the TCP startup algorithm is to find the point ($cwnd$, $sending\_rate$) on this line that corresponds to the correct congestion window and sending rate of an ideal, stable, well-paced TCP flow. Since the "target" window and rate are related ($cwnd = RTT * sending\_rate$), we need to find only one coordinate.

The traditional Slow Start algorithm searches for the congestion window by moving along the X-axis ($cwnd$) without explicitly considering the Y-axis ($sending\_rate$). In contrast, Paced Start samples the 2-D space in a more systematic fashion, allowing it in many cases to identify the target more quickly. In Figure 4, the area below the B line includes the possible values of the available bandwidth. The solid arrows show how Paced Start explores this 2-D space; each arrow represents a probing cycle. Similar to Slow Start, Paced Start explores along the X-axis by doubling the packet train length every roundtrip time. Simultaneously, it does a binary search of the Y-axis, using information about the change in gap value to decide whether it should increase or decrease the rate. Paced Start can often find a good approximation for the available bandwidth after a small number of cycles (3 or 4 in our simulations), at which point it "jumps" to the target point, as shown in case 1 and case 2.

The binary search proceeds as follows. We first send two back-to-back packets; the gap at the destination will be the value g_B. In the next cycle, we set the source gap to $2 * g_B$, starting the binary search by testing the rate of $B/2$. Further adjustments of the gap are made as follows:

1) If $g_s < g_d$, we are exploring a point where the Packet Transmission Rate (PTR) is higher than the available bandwidth, so we need to reduce the PTR. In a typical binary search algorithm, this would be done by taking the middle point between the previous PTR and the current lower bound on PTR. In Paced Start, we can speed up the convergence by using $2 * g_d$ instead of $2 * g_s$. That allows us to use the most recent probing results, which are obtained from longer packet train and generally have lower measurement error.

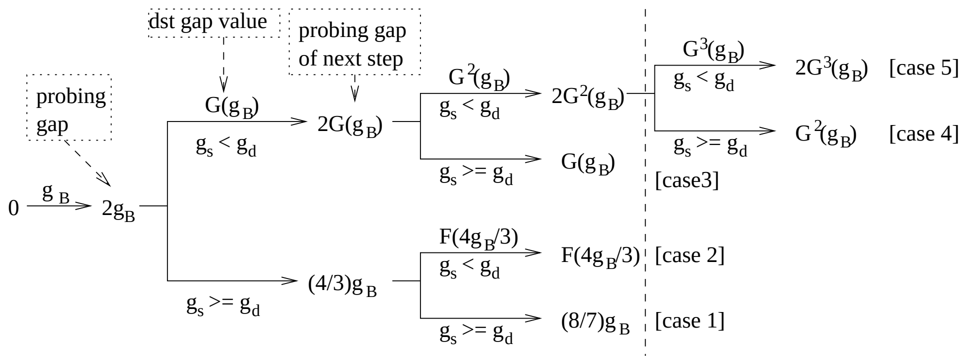


Figure 5. Paced Start gap adjustment decision tree. The formula pointed by an arrow is a possible probing gap g_s; the formula above an arrow is a g_d when $g_s < g_d$, no formula above an arrow means $g_s \geq g_d$.

2) If $g_s \geq g_d$, the PTR is lower than the available rate and we have to reduce the packet gap. The new gap is selected so the PTR of the next train is equal to the middle point between the previous PTR and the current upper bound on PTR.

3.2.2. Algorithm Termination

The purpose of the startup algorithm is to identify the "target" point, as discussed above. This can be done by either identifying the target congestion window or the target rate, depending on whether we reach the target along the x or y axis in Figure 4. This translates into two termination cases for Paced Start:

- **Identifying the target rate**: This happens when the difference between source and destination gap values shows that the PTR is a good estimate of the available bandwidth. As we discuss below, this typically takes 3 or 4 iterations. In this case, we set the congestion window size as $cwnd = RTT/g$, where g is the gap value determined by Paced Start. Then we send a packet train using $cwnd$ packets with packet gap g. That fills the transmission pipe, after which we can switch to congestion avoidance mode.
- **Identifying the target congestion window**: When we observe packet loss in the train, either through a timeout or duplicate ACKs, we assume we have exceeded the transmission capacity of the path, as in traditional TCP Slow Start. In this case, we transition into congestion avoidance mode. If there was a timeout, we use Slow Start to refill the transmission pipe, after setting *ssthresh* to half of the last train length. Otherwise we rely on fast recovery.

How Paced Start terminates depends on many factors, including available bandwidth, RTT, router queue buffer size, and cross traffic properties. From our experience, Paced Start terminates by successfully detecting the available bandwidth about 80% of the time, and in the remaining 20% cases, it exits either with a timeout or fast retransmit after packet loss.

3.2.3. Gap Estimation Accuracy

An important question is how many iterations it takes to obtain an available bandwidth estimate that is "close enough" for TCP, i.e. within a factor of two. This means that we need to characterize the accuracy of the available bandwidth estimate obtained by Paced Start.

Figure 5 shows the gap values that are used during the binary search assuming perfect conditions; we use the notation of Table 1. The conditions under which a branch is taken are shown below the arrows while the values above the arrows are the destination gaps; the values at the end of the arrows indicate the source gap for the next step. We rely on a result from [17] to estimate the destination gap: if $g_s < g_d$, then the relationship between the source and destination gap is given by

$$g_d = g_B + (1 - \alpha)g_s$$

We use $F(g) = g_B + (1 - \alpha)g$ to denote this relationship. We also use another function, $G(g) = F(2g)$.

This model allows us to calculate how we narrow down the range of possible values for α as we traverse the tree. For example, when during the second iteration we probe with a source gap of $2g_B$, we are testing the point $\alpha = 0.5$. If $g_s < g_d$, we need to test a smaller α by increasing the gap value to $2G(g_B)$ (based on $g_d = G(g_B)$); otherwise, we want to test a larger value ($\alpha = 0.75$, following a binary search) by reducing the gap value to $(4/3)g_B$.

Table 2. Paced Start exiting gap values

case	α	g_{PaSt}	$\frac{1/\alpha}{g_{PaSt}}$
1	(0.75,1)	$8g_B/7$	(0.9, 1.2)
2	(0.5,0.75)	$F((4/3)g_B) = (7/3 - 4\alpha/3)g_B$	(1.0, 1.2)
3	(0.19,0.5)	$G(g_B) = (3 - 2\alpha)g_B$	(1.0, 2.0)
4	(0.08,0.19)	$G^2(g_B) = (1+2(1-\alpha)(3-2\alpha))g_B$	(1.0, 2.0)
5	(0,0.08)	$2G^3(g_B) = 2(1+2(1-\alpha)(1+(1-\alpha)2(3-2\alpha)))g_B$	(0.5, ∞)

Table 2 shows the ranges of α for the 5 exiting cases shown in Figure 5. It also lists the ratio between $1/\alpha$ and g_{PaSt}. This corresponds to the ratio between the real sending rate and the available bandwidth, i.e. it tells us how much we are overshooting the path available bandwidth when we switch to congestion avoidance mode. Intuitively, Figure 6 plots the difference between $1/\alpha$ and g_{PaSt} for a network path with a bottleneck link capacity of 100 Mbps.

From Table 2 and Figure 6, we can see that if α is high (e.g. cases 1, 2, and 3), we can quickly zoom in on an estimate that is within a factor of two. We still require at least 3 iterations because we want to make sure we have long

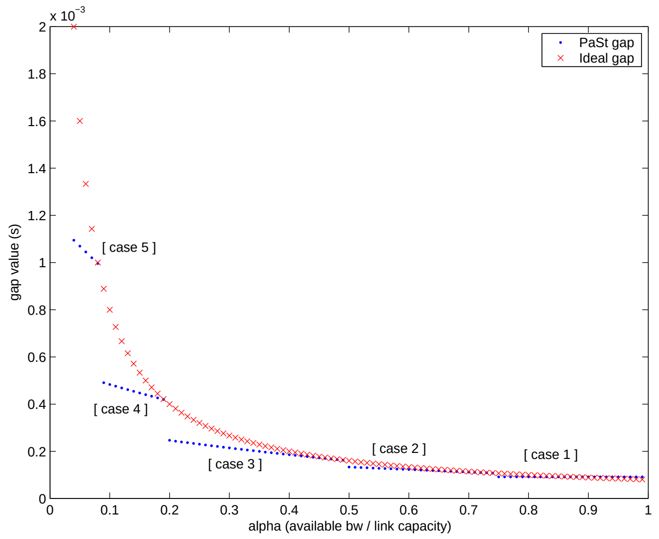


Figure 6. Difference between PaSt gap value and ideal gap value.

enough trains so the available bandwidth estimate is accurate enough. This is the case where Paced Start is likely to perform best relative to Slow Start: Paced Start can converge quickly while Slow Start will need many iterations before it observes packet loss.

For smaller α, more iterations are typically needed, and it becomes more likely that the search process will first "bump" into the target congestion window instead of the target rate. This means that Paced Start does not offer much of a benefit since its behavior is similar to that of Slow Start — upon packet loss it transitions into congestion avoidance mode in exactly the same way. In other words, Paced Start's performance is not any worse than that of Slow Start.

3.3. Implementation

We have implemented Paced Start in *ns2*[5], as a user-level library, and in the Linux kernel. For the implementation in *ns2*, we replaced the the Slow Start algorithm with the Paced Start algorithm in TCP SACK. This implementation allows us to evaluate both end-to-end performance and the impact on network internals, e.g. router queues, which is very hard, if not impossible, to analyze using real world experiments.

The user-level library implementation of Paced Start is for the purpose of experimenting over the Internet without kernel modifications and root privileges. This implementation is based on the *ns2* TCP code. We keep the packet nature of *ns2* TCP and omit the byte level implementation details. The user-level library embeds TCP packets (using *ns2* format for simplicity) in UDP packets. The porting consisted of extracting the TCP code from *ns2* and organizing it as send and receive threads. For timeouts and packet pacing we use the regular system clock.

The Linux kernel implementation allows us to study the issues associated with packet pacing inside the kernel, e.g. timer granularity and overhead. We can also use popular applications such as Apache [1] to understand PaSt's improvement on TCP performance. The main challenge in the kernel implementation is the need for fine-grained timer to pace out the packets. This problem has been studied by Aron et.al. [10], whose work shows that the soft timer technique allows the system timer to achieve 10us granularity in BSD without significant system overhead. With this timer granularity PaSt should be effective on network paths with capacities as high as 1Gbps. Our in-kernel implementation runs on Linux 2.4.18, which implements the TCP SACK option. We use the Linux high resolution timer patch [3] to pace the packets. We refer to the original and the modified Linux 2.4.18 kernel as the Sack kernel and PaSt kernel respectively.

In all three implementations, we measure the destination gap not at the destination, but on the sender based on the incoming stream of ACK packets. This approach has the advantage that Paced Start only requires changes to the sender. This simplifies deployment significantly, and also frees us from having to extend the TCP protocol with a mechanism to send gap information from the destination to the sender. One drawback of using ACKs to measure the destination gap is that the average gap value for the ACK train may not be identical to the average destination gap. This could reduce the accuracy of PTR.

Finally, delayed ACKs also adds some complexity since the delayed timer can affect the flow of ACKs. Our solution is to use the ACK sequence number to determine whether an ACK packet acknowledges two packets or one packet. If it acknowledges two packets, it can be used because it is sent out immediately after receiving the second packet. Otherwise, it is generally triggered by the delayed ACK timer[1]. In this case, the inter-ACK gap bears no relationship to the destination inter-packet gap, so we ignore this gap when estimating the destination gap. By doing this simple check, we can easily identify the ACK packets that can be used by PaSt.

4. Evaluation Methodology and Metrics

We use both simulation and system measurements to evaluate Paced Start. The focus of the simulation-based evaluation is on comparing two variants of TCP Sack, one with traditional Slow Start (called *Sack*) and the other with Paced Start (called *PaSt*). It includes two parts:

- **Perspective of a single user:** we look in detail at the behavior of a single PaSt flow. We show how PaSt avoids packet loss and timeouts during startup, thus improving performance.
- **Perspective of a large network:** we study PaSt's performance in a network where flows with different lengths arrive and leave randomly. We show that, by reducing the amount of packet loss, PaSt can make the network more effective in carrying data traffic.

Our real system evaluation also includes two sets of measurements. First, we present results for our user-level PaSt implementation running over the Internet. Second, we use in-kernel PaSt implementation to collect performance results for web traffic using an Apache server; these results were collected on Emulab [2].

[1]The real implementation is more complicated than described here due to the QUICKACK option in Linux 2.4.18 TCP implementation.

Table 3. Evaluation Metrics

throughput	throughput for the whole connection
loss	loss rate for the whole connection
su-throughput	throughput for the startup period
su-loss	loss rate for the startup period
su-time	time length of the startup period

Table 3 includes the metrics that we use in our evaluation. Our two primary performance metrics are the *throughput* and the *loss rate* (fraction of packets lost) of a TCP session. They capture user performance and the stress placed on the network, respectively. Since our focus is the TCP startup algorithm, we will report these two metrics not just for the entire TCP session, but also for the *startup time*, which is defined as the time between when the first packet is sent and when TCP transitions into congestion avoidance mode, i.e. starts the linear increase of the AIMD algorithm. Our final metric is the duration of the startup time (*su-time*).

A tricky configuration for the analysis in this paper is the choice of the initial *ssthresh* value. TCP implementations either use a fixed initial *ssthresh*, or cache the ssthresh used by the previous flow to the same destination. We choose to use the first method, because the cached value does not necessarily reflect the actual network conditions [6], and we cannot always expect to have cached value when a transmission starts. For the first method, no single fixed value serves all the network paths that we are going to study, since they have different bandwidth-delay products. For this reason, unless stated explicitly (e.g. in Figure 13), we set the initial *ssthresh* to a very high value (e.g. 20000 packets) such that Slow Start always ramps up until it saturates the path.

5. Simulation Results

5.1. Perspective of a Single User

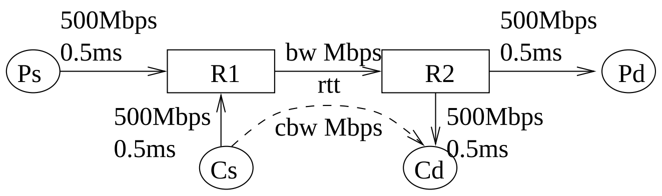


Figure 7. Network topology for simulation

The network topology used in this set of simulations is shown in Figure 7. Simulations differ in the competing traffic throughput (cbw), bottleneck link capacity (bw), round trip time (rtt), and router queue buffer size ($queue$, not shown). The default configuration is $cbw = 40Mbps$, $bw = 100Mbps$, $rtt = 80ms$, and $queue = 1000pkt$ (1000Byte/pkt). This queue size corresponds to the bandwidth-delay product of the Ps-Pd path. Unless otherwise stated, simulations use these default values. The results in this section use CBR traffic as the background traffic, but we see similar result for TCP background traffic. We use TCP background traffic in Section 5.2 and 5.3.

5.1.1. Effect of Competing Traffic Load

Figure 8 shows how the throughput and the loss rate change as a function of the competing traffic load (*cbw*). The first graph shows the throughput for Sack and PaSt,

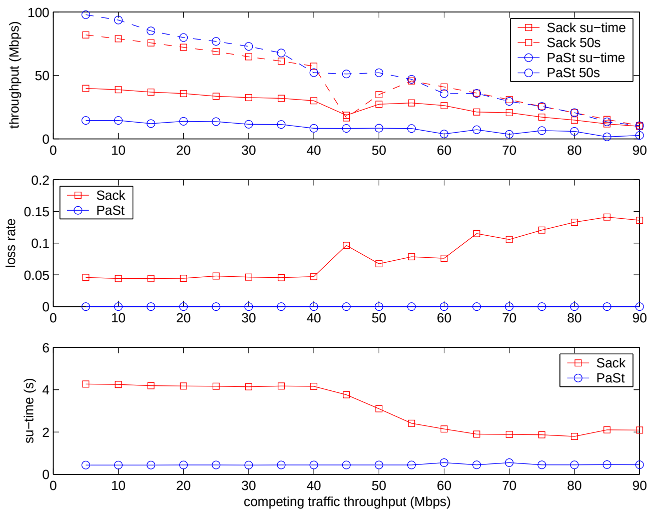


Figure 8. Impact of competing traffic load

both during startup (*su-time*) and for the entire run (50 seconds). During startup, PaSt has lower throughput than Sack, because PaSt is less aggressive, as we described in the previous section. The second graph confirms this: PaSt avoids placing a heavy load on the router and rarely causes packet loss during startup. Across the entire run, the throughput of Sack and PaSt are very similar[2]. The reason is that the sustained throughput is determined by the congestion avoidance algorithm, which is the same for both algorithms.

The third graph plots the *su-time*. We see that PaSt needs much less time to enter the congestion avoidance stage, and the startup time is not sensitive to the available bandwidth. The high *su-time* values for Sack are mainly due to the fast recovery or timeout caused by packet loss, and the value decreases as the competing traffic load increases, because the lower available bandwidth forces Sack to lose packet earlier.

We also studied how the performance of PaSt is affected by the bottleneck link capacity and the roundtrip time. The conclusions are similar to those of Figure 8 and we do not include the results here due to space limitation.

5.1.2. Comparing with NewReno and Vegas

TCP NewReno [16] [15] and TCP Vegas [13] are two well known TCP variants that may improve TCP startup performance. In this section, we use simple simulations to highlight how TCP NewReno, Vegas, and PaSt differ. The simulations are done using the default configuration of Figure 7. We again focus on the performance for different background traffic loads.

The simulation results are shown in Figure 9. NewReno's estimation for *ssthresh* eliminates packet loss when the competing traffic load is less than 60Mbps, but for higher loads we start to see significant packet loss during startup. This is because NewReno's estimation is based on a pair of SYN packets. A packet pair actually estimates the bottleneck link capacity, not the available bandwidth. When the competing traffic load is low, this might work, but as the competing

[2]The dip of Sack throughput when cbw is 45Mbps is because Sack experiences more than one timeout during the 50s simulation.

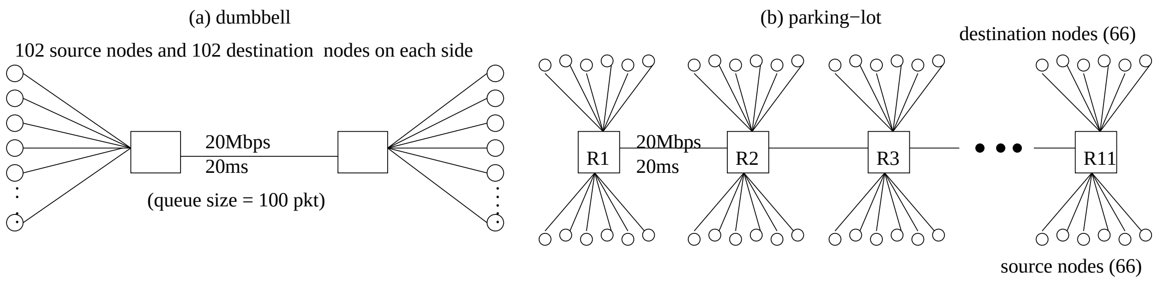


Figure 10. Dumbbell topology and parking-lot topology

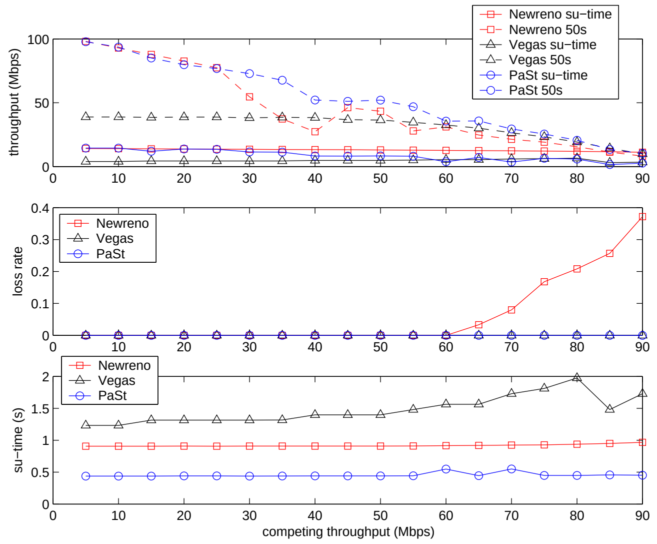


Figure 9. Comparison between PaSt and NewReno/Vegas

traffic increases, NewReno tends to overestimate the available bandwidth and thus the initial congestion window. This can easily result in significant packet loss.

Vegas can sometimes eliminate packet loss during Slow Start by monitoring changes in RTT. As shown in Figure 9, both Vegas and PaSt have 0 packet loss for this set of simulations. However, Vegas has the longer startup time than PaSt and NewReno. The reason is that Vegas only allows exponential growth every other RTT; in between, the congestion window stays fixed so a valid comparison of the expected and actual rates can be made. [13] points out that the benefits of Vegas' improvements to Slow Start are very inconsistent across different configurations. For example, Vegas sometimes overestimate the available bandwidth.

5.2. Perspective of A Large Network

We now simulate a set of scenarios in which a large number of flows coexist and interfere with each other. We chose two network topologies: a dumbbell topology and a parking-lot topology (Figure 10). In the dumbbell topology many flows interfere with each other, while the parking-lot topology provides us with a scenario where flows have different roundtrip times.

5.2.1. Dumbbell Topology

In the dumbbell network, we use 204 nodes (102 source nodes and 102 destination nodes) on *each* side. The edge link capacity is 50Mbps, and the bottleneck link capacity is 20Mbps, with a 20ms latency. The router queue buffer is configured as the bandwidth delay product, i.e., 100 packets (1000 bytes/pkt). The source and destination node for each flow are randomly (uniformly) chosen out of the 408 nodes, except that we make sure that the source and destination are on different sides of the bottleneck link. The inter-arrival time of the flows is exponentially distributed with a mean of 0.05 seconds (corresponds to mean flow arrival rate of 20 flows/second). The flow size is randomly selected according to an exponential distribution with a mean of 50 packets. After starting the simulation, we wait for the network to become stable, that is, the number of flows changes in a fixed limited range. Then we monitor the performance of the next 2000 flows. In the stable state, about 7 flows are active at any time.

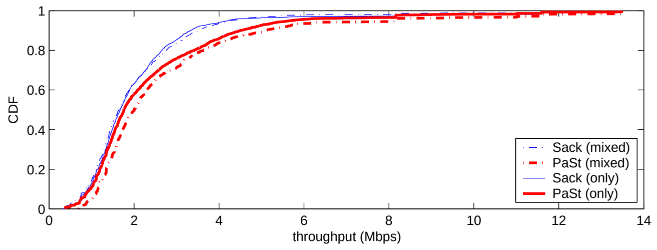


Figure 11. Dumbbell simulation with a mean flow size of 50 packets.

Figure 11 shows the cumulative throughput distribution for three simulations: *Sack only*, *PaSt only*, and a *mixed* scenario in which the flow type (Sack/PaSt) is randomly chosen with equal probability (0.5). This figure shows that PaSt flows achieve somewhat higher throughput than Sack flows. It is interesting to note that the performance of Sack (mixed) and that of Sack (only) are very similar; this suggests that in this scenario, PaSt has little impact on the performance of Sack. Also, PaSt (mixed) and PaSt (only) achieve similar performance, although PaSt (mixed) has a small advantage. Because of the short flow size, there is almost no packet loss in this case.[3]

We repeated the same set of simulations with a mean flow size of 200 packets and a mean flow arrival rate of 10 flows/second. The results are shown in Figure 12. The throughput results (top graph) are very similar to those for

[3]When we increase the flow arrival rate, e.g. to 100 flows/second, the system no longer reaches a stable state before the size of the ns2 trace file exceeds the system limit (2GBytes).

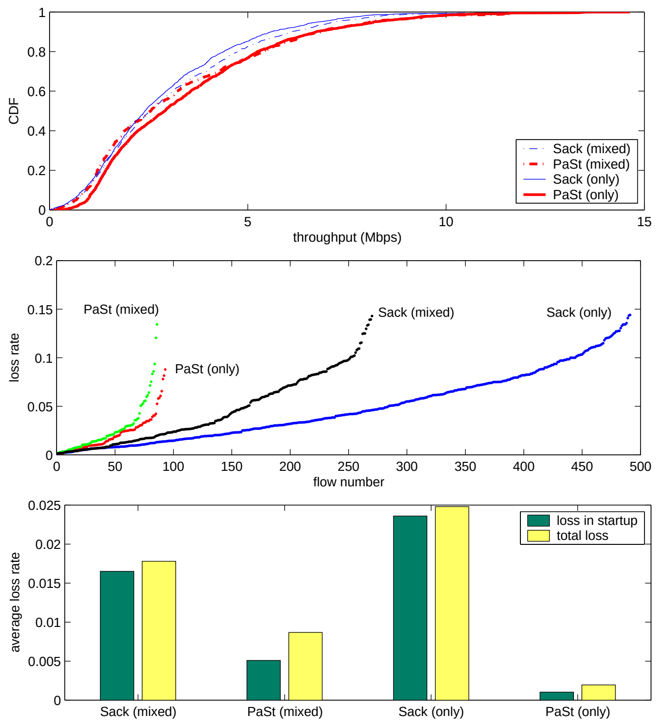


Figure 12. Results for the dumbbell topology and a mean flow size of 200 packets. In the middle graph, the flows are ordered according to their loss rate. In the bottom graph, "loss in startup" denotes the ratio between the number of packet lost during startup and the number of total packets sent; "total loss" denotes the ratio between the number of packet lost and the number of total packets sent.

the shorter flows (Figure 11). PaSt (only) systematically outperforms Sack (only). The "mixed" and "only" results are fairly similar, although we see that the presence of PaSt flows has a good influence on Sack flows, while the presence of Sack flows has a negative influence on PaSt flows.

The second figure plots the loss rate for all the flows that lose packets. The Paced Start algorithm clearly improves the loss rate significantly. In both the "mixed" and the "only" scenario, significantly fewer PaSt flows than Sack flows experience packet loss (note that for the "mixed" cases, there are only half as many flows). Also, for the mixed scenario, the presence of PaSt flows significantly reduces packet loss by reducing the number of lossy flows (PaSt(mixed) + Sack(mixed) $<$ Sack(only)).

To better understand the details of PaSt's impact on packet loss, we traced packet loss during startup. The bottom graph in Figure 12 shows the loss rate for the different simulations, separating losses during startup from the total loss. The graph confirms that PaSt has a lower loss rate than Sack — the difference is especially large for losses during startup. Comparing PaSt (mixed) and PaSt (only) also shows that the presence of Sack flows increases packet loss both during startup and during congestion avoidance, suggesting that the aggressiveness of Slow Start has a negative influence on network performance.

Finally, in our results so far, Sack flows have used an initial

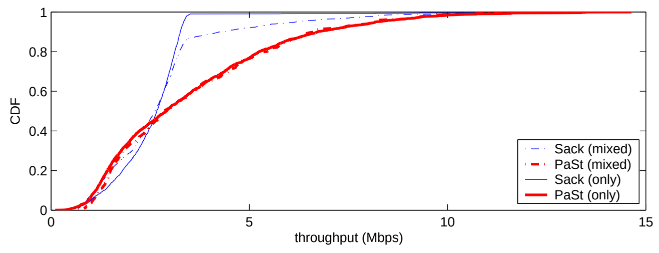


Figure 13. Dumbbell simulation with a mean flow size of 200 packets and an initial ssthresh of 20 packets.

ssthresh that is very large, so it would not cut off Slow Start. Figure 13 shows the results for the same configuration as used in Figure 12, but with the initial *ssthresh* set to 20, which is the default value in ns2. We see that the Sack throughput pretty much levels off around 4Mbps. This is not surprising. With a roundtrip time of 40ms, a congestion window of 20 packets results in a throughput 4Mbps (500 packets/second). In other words, with this particular *ssthresh* setting, only very long flows are able to really benefit from congestion windows larger than the initial *ssthresh*.

5.2.2. Parking-Lot Topology

The parking-lot topology (Figure 10) has 10 "backbone" links connected by 11 routers. Each router supports 6 source nodes and 6 destination nodes through a 50Mbps link. *Each* backbone link has 20Mbps capacity, 20ms latency, and is equipped with 1000 packets router queue buffer (corresponding to the longest path). The traffic load and methodology for collecting results is similar to that used for the dumbbell topology. Flows randomly pick a source and destination, making sure each flow traverses at least one "backbone" link. The mean flow size is 200 packets, and the mean flow rate arrival rate is 10 flows/second. In the stable state, about 15 flows are active at any time. We again collected data for 2000 flows.

The simulation results are shown in Figure 14. The results are very similar to those for the dumbbell topology (Figure 12). PaSt improves network performance in both the "mixed" and "only" scenario. It has both higher throughput and lower packet loss rate, thus alleviating the stress on the network caused by the TCP startup algorithm. Since the flows in this scenario have different roundtrip times, these results confirm the benefits of Paced Start demonstrated in the previous section.

5.3. Throughput Analysis

In the previous section, we presented aggregate throughput and loss rate results. We now analyze more carefully how PaSt compares with Sack for different types of flows. First, we study how flow length affects the relative performance of Paced Start and Slow Start.

5.3.1. Sensitivity of PaSt to Flow Length

The impact that TCP startup has on flow throughput is influenced strongly by the flow length. Very short flows never get out of the startup phase, so their throughput is

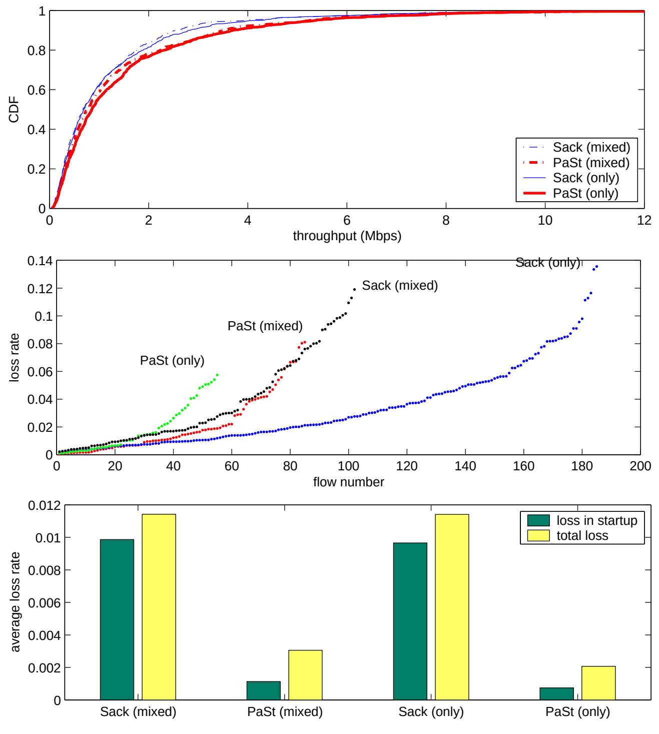


Figure 14. Results for the parking-lot topology. Refer to Figure 12 for the definition of "loss in startup" and "total loss".

completely determined by the startup algorithm. For long flows, the startup algorithm has negligible impact on the total throughput since most of the data is transmitted in congestion avoidance mode. For the intermediate-length flows, the impact of the startup algorithm depends on how long the flow spends in congestion avoidance mode.

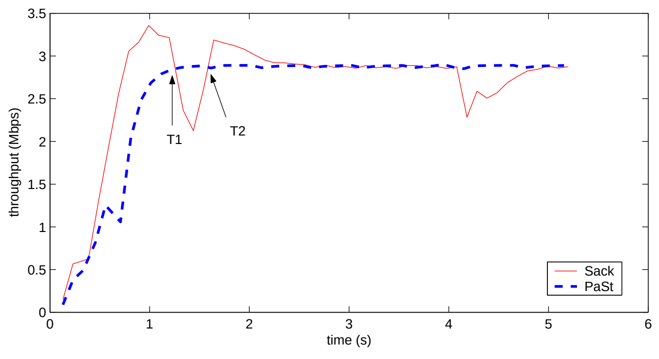


Figure 15. Comparison of instantaneous Sack and PaSt flow throughput. The data is a moving average over a 0.1 second interval.

To better understand the relationship between flow size and flow throughput, Figure 15 compares the instantaneous throughput of a Sack and PaSt flow. This is for a fairly typical run using the default configuration of Figure 7. We observe that there are three periods. The first period (up to T1) corresponds to the exponential increase of Slow Start. In this period, Sack achieves higher throughput because it is more aggressive. In the second period (T1-T2), packet loss often causes Sack to have a timeout or to go through fast recovery, while PaSt can maintain its rate. In this phase, PaSt has better performance. Finally, during the congestion avoidance phase period (after T2), the two flows behave the same way.

While this analysis is somewhat of an oversimplification (e.g. PaSt cannot always avoid packet loss, Sack does not always experience it, ...), it does help to clarify the difference in throughput between Slow Start and Paced Start. For short-lived flows, Sack is likely to get better throughput than PaSt. For very long-lived flows, both algorithms should have similar throughput, although the lower loss rate associated with PaSt might translate into slightly higher throughput for PaSt. For intermediate flows, PaSt can achieve higher throughput than Sack, once it has overcome its handicap.

In this context, flow size is not defined in absolute terms (e.g. MBytes) but it is based on when (if ever) the flow enters congestion avoidance mode. This is determined by parameters such as path capacity and RTT. For a low capacity path, the bandwidth-delay product is often small, and a flow can finish the startup phase and enter congestion avoidance mode after sending relatively few packets. For high capacity paths, however, a flow will have to send many more packets before it qualifies as an intermediate or long flow.

5.3.2. Flow Length Analysis

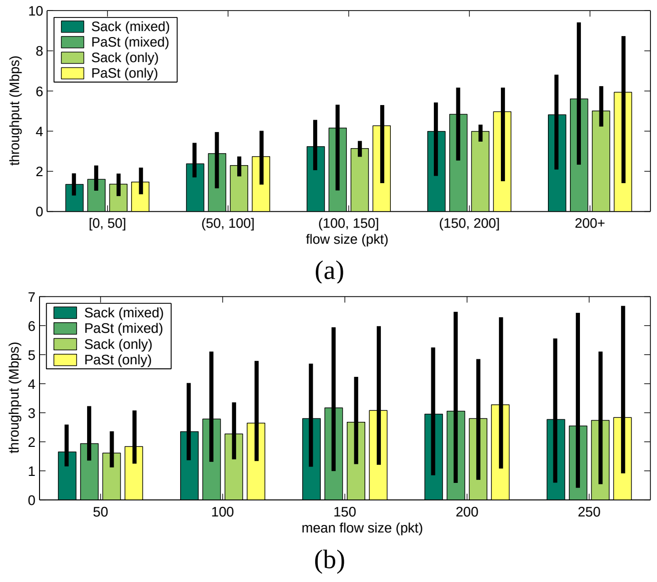


Figure 16. Average throughput for Sack and PaSt as a function of flow size.

We grouped the flows in Figure 11 based on their flow length, and plot the median flow throughput and the [5%, 95%] interval in Figure 16(a). When flow size is less than 50 packets, there is little difference in terms of throughput. But when the flow size increases to 100, 150, 200, and 200+ packets, there are significant improvement from PaSt, both in the "mixed" and "only" scenario. Note that the higher bins contain relatively few flows. Still, the fact that PaSt outperforms Sack even for long flows (200+) suggests that

Table 4. Comparison between Sack and PaSt for an Internet path

	Sack			PaSt		
	Min	Median	Max	Min	Median	Max
su-time(s)	0.2824	0.4062	2.3155	0.1037	0.1607	1.1578
su-throughput(Mbps)	3.05	21.63	35.80	1.38	6.77	15.28
su-loss	0.0081	0.0575	0.3392	0.0040	0.0055	0.0192
throughput(Mbps)	15.64	22.68	40.19	14.22	22.07	27.39
loss	0.0016	0.0090	0.1593	0.0001	0.0011	0.0071

the lower packet loss has a fairly substantial impact, and we need longer flows to recover from it.

Figure 16(b) studies this issue from another aspect. We repeat the dumbbell simulation with different mean flow sizes, ranging from 50 to 250 packet. The arrival rate is again 10 flows/second. For each simulation, we only report results for the flows in the [25%, 75%] interval of the flow length distribution; this means that in each case, the flows have very similar lengths. The results in Figure 16(b) confirm our analysis of the previous section: PaSt achieves the best throughput improvement for intermediate flows, i.e. flows with 100-150 packets in this case, and has a smaller impact on long flows.

6. System Measurements

We report real system measurement results obtained with our user-level and in-kernel Paced Start implementation.

6.1. Internet Experiment

The Internet experiments use our user-level PaSt implementation. We transmit data using Sack and PaSt from CMU to MIT. This Internet path has a bottleneck link capacity of 100Mbps, an RTT of 23 msec, and an available bandwidth of about 50Mbps (measured using PTR [17]). We sent Sack and PaSt flows in an interleaved fashion, separated by 5 seconds of idle time. Each flow transmits 10000 packets (1000 bytes/packet). The same measurement is repeated 20 times. Since the Internet traffic keeps changing, the measurement environments for Sack and PaSt are not exactly the same, so we cannot compare them side by side as we did in the simulations. Instead, we list the maximum, median and minimum value for each metric, which provides at least a coarse grain comparison.

The results are listed in Table 4. The conclusions from Internet measurements are very similar to those from the simulation results. During the startup period, PaSt converges faster than Sack, has a smaller packet loss rate, and a lower throughput. The median throughputs over the entire data transfer are very similar for both algorithms. Note that the loss rate of PaSt for the entire session is still less than Sack, suggesting that 10000 packets is not enough to hide the effect of the startup algorithm for this path.

6.2. In-Kernel Experiment

We ran a number of experiments using our in-kernel PaSt implementation on Emulab [2]. We report an experiment that uses two Apache servers and two web requests generators — Surge [12] — to evaluate what impact Paced Start has on web traffic. The testbed is a dumbbell network, with one Apache server and one Surge client host on each side of the bottleneck link. The bottleneck link has a 20Mbps capacity, 20ms latency, and its router buffers are 66 packets (1500 bytes/packet). The two Surge clients are started at the same time, so there is web traffic flowing in both directions over the bottleneck link. The web pages are generated following a zipf distribution, with a minimum size of 2K bytes. The web requests generation uses Surge's default configuration. We ran each experiment for 500 seconds, generating around 2100 requests. We focus on the performance of one Apache server, for which we use both the Sack and PaSt kernel. The other three hosts, i.e., the two Surge clients and the other Apache server, always use the Sack kernel.

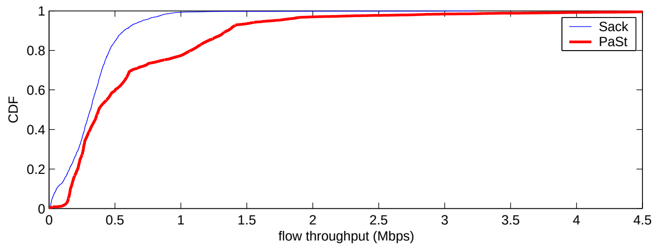


Figure 17. Emulab experiment comparing the Sack and PaSt kernels using the Apache and Surge applications

We use the log files of Surge to calculate the throughput distribution for downloading web pages. Figure 17 compares the throughput distributions for the Apache server using the Sack and PaSt kernels. The throughput improvement from PaSt is very clear. The two kernels also suffered very different packet loss rates: 94186 packets were dropped with Sack, but only 1168 packet were dropped using PaSt. These results are similar to our simulation results.

7. Related Work

Using active measurements to improve TCP startup is not a new idea. Hoe [16] first proposed to use the packet pair algorithm to get an estimate for the available bandwidth, which is then used to set the initial ssthresh value. Aron et.al. [9] improved on Hoe's algorithm by using multiple packet pairs (of 4 packets) to iteratively improve the estimate of ssthresh as Slow Start progresses. The Swift-start [24] algorithm uses a similar idea. However, there is ample evidence [17][20] that simply using packet pairs cannot accurately estimate available bandwidth. For this reason, Paced Start uses packet trains with carefully controlled inter-packet gaps to estimate the available bandwidth and ssthresh.

A number of groups have proposed to use packet pacing to reduce the burstiness of TCP flows. For example, Paced TCP

[21] uses a leaky-bucket to pace the TCP packets; rate-based pacing [26] uses packet pacing to improve TCP performance for sessions involving multiple short transfers (e.g. many HTTP sessions).

TCP Vegas [13] also proposed a modified Slow Start algorithm. It compares the “expected rate” with the “actual rate” and when the actual rate falls below the expected rate by the equivalent of one router buffer, it transitions into congestion avoidance mode. The evaluation shows however that this algorithm offers little benefit over Slow Start.

There have been many other proposals to improve the TCP startup performance. Some proposals can also be used by PaSt, e.g. increasing the initial ssthresh value [7][8], while others will only be effective under some circumstances, e.g. sharing the connection history information between different flows [25][11]. Smooth-Start [27] proposes to split the slow start algorithm into a “slow” and “fast” phase that adjust the congestion window in different ways. While Smooth-Start can reduce packet loss, it does not address the question of how the *ssthresh* and the threshold that separates the phases should be selected. Finally, some researchers have proposed to use explicit network feedback to set the congestion window, e.g. Quick-Start [19], but this requires router support.

Researchers have also studied how to quickly restart transmission on a TCP connection that has gone idle, which is a common problem in web servers. Both TCP fast start [23] and rate-based pacing [26] belong in this category. The idea is to make use of history information of the TCP connection. In contrast, we focus on the initial Slow Start sequence of a new TCP connection.

8. Conclusion

In this paper we present a new TCP startup algorithm, called Paced Start (*PaSt*). It builds on recent results in the area of available bandwidth estimation to automatically determine a good value for *ssthresh*. Paced Start is based on the simple observation that the TCP startup packet sequence is in fact a sequence of packet trains. By monitoring the difference between the spacing of the outgoing data packets and the spacing of the incoming acknowledgements, Paced Start can quickly estimate the proper congestion window for the path. In this paper, we describe the analytical properties of Paced Start, and present an extensive set of simulation results and real system measurements that compare Paced Start with the traditional Slow Start algorithm. We show that, for most flows, Paced Start transitions into congestion avoidance mode faster than Slow Start, has a significantly lower packet loss rate, and avoids the timeout that is often associated with Slow Start. In terms of deployment, Paced Start can coexist with the current TCP implementations that use Slow Start.

Acknowledgement

This research was sponsored in part by the Defense Advanced Research Project Agency and monitored by AFRL/IFGA, Rome NY 13441-4505, under contract F30602-99-1-0518.

We would like to thank Srinivasan Seshan, Jun Gao, An-Cheng Huang, and Aman Shaikh for their useful comments and suggestions. We also thank the Emulab team for the use of Emulab and for their help in setting up the experiments.

References

[1] The Apache software foundation. http://www.apache.org/.
[2] Emulab. http://www.emulab.net.
[3] High res posix timer. http://sourceforge.net/projects/high-res-timers/.
[4] Iperf. http://dast.nlanr.net/Projects/Iperf/.
[5] ns2. http://www.isi.edu/nsnam/ns.
[6] TCP auto-tuning zoo. http://www.csm.ornl.gov/∼dunigan/netperf/auto.html.
[7] M. Allman, S. Floyd, and C. Partridge. Increasing TCP's initial window. *RFC 2414*, September 1998.
[8] M. Allman, C. Hayes, and S. Ostermann. An evaluation of TCP with larger initial windows. *ACM Computer Communication Review*, 28(3), July 1998.
[9] M. Aron and P. Druschel. TCP: Improving startup dynamics by adaptive timers and congestion control. *Technical Report (TR98-318), Dept. of Computer Science, Rice University*, 1998.
[10] Mohit Aron and Peter Druschel. Soft timers: effi cient microsecond software timer support for network processing. In *ACM Transactions on Computer Systems (TOCS)*, August 2000.
[11] Hari Balakrishnan, Hariharan Rahul, and Srinivasan Seshan. An integrated congestion management architecture for internet hosts. In *Proc. ACM SIGCOMM '99*, Cambridge, MA, September 1999.
[12] Paul Barford. *Modeling, Measurement and Performance of World Wide Web Transactions*. PhD thesis, December 2000.
[13] Lawrence S. Brakmo and Larry L. Peterson. TCP Vegas: End to end congestion avoidance on a global internet. *IEEE Journal on Selected Areas in Communications*, 13(8):1465–1480, 1995.
[14] D.M. Chiu and R. Jain. Analysis of increase and decrease algorithms for congestion avoidance in computer networks. In *Journal of Computer Networks and ISDN*, volume 17, July 1989.
[15] S. Floyd and T. Henderson. The NewReno modifi cation to TCP's fast recovery algorithm. *RFC 2582*, April 1999.
[16] J. C. Hoe. Improving the start-up behavior of a congestion control sheme for TCP. In *Proc. ACM SIGCOMM 96*, volume 26,4, pages 270–280, New York, 26–30 1996. ACM Press.
[17] Ningning Hu and Peter Steenkiste. Evaluation and characterization of available bandwidth probing techniques. *IEEE JSAC Special Issue in Internet and WWW Measurement, Mapping, and Modeling*, 21(6), August 2003.
[18] Van Jacobson. Congestion avoidance and control. In *Proc. ACM SIGCOMM 88*, August 1988.
[19] A. Jain and S. Floyd. Quick-start for TCP and IP. *draft-amit-quick-start-00.txt*, June 2002.
[20] Manish Jain and Constantinos Dovrolis. End-to-end available bandwidth: Measurement methodology, dynamics, and relation with TCP throughput. In *SIGCOMM 2002*, Pittsburgh, PA, August 2002.
[21] J. Kulik, R. Coulter, D. Rockwell, and C. Partridge. Paced TCP for high delay-bandwidth networks. In *Proceedings of IEEE Globecom'99*, December 1999.
[22] Saverio Mascolo, Claudio Casetti, Mario Gerla, M. Y. Sanadidi, and Ren Wang. TCP Westwood: bandwidth estimation for enhanced transport over wireless links. In *Mobile Computing and Networking*, pages 287–297, 2001.
[23] V. Padmanabhan and R. Katz. TCP fast start: A technique for speeding up web transfers. In *Globecom 1998 Internet Mini-Conference.*, Sydney, Australia, November 1998.
[24] Craig Partridge, Dennis Rockwell, Mark Allman, Rajesh Krishnan, and James Sterbenz. A swifter start for tcp. Technical Report BBN-TR-8339, BBN Technologies, mar 2002.
[25] J. Touch. TCP control block interdependence. *RFC 2140*, April 1997.
[26] Vikram Visweswaraiah and John Heidemann. Improving restart of idle TCP connections. Technical Report 97-661, University of Southern California, November 1997.
[27] Haining Wang and Carey L. Williamson. A new TCP congestion control scheme: Smooth-start and dynamic recovery. In *Proceedings of IEEE MASCOTS'98*, Montreal, Canada, 1998.
[28] Z. Wang and J. Crowcroft. A new congestion control scheme: Slow start and search (Tri-S). *ACM Computer Communication Review*, 21(1):32–43, 1991.