Computer Networks 55 (2011) 2092–2110

Contents lists available at ScienceDirect

Computer Networks

journal homepage: www.elsevier.com/locate/comnet

Taming the elephants: New TCP slow start ☆

Sangtae Ha [a,*], Injong Rhee [b]

[a] Department of Electrical Engineering, Princeton University, Princeton, NJ 08544, United States
[b] Department of Computer Science, North Carolina State University, Raleigh, NC 27606, United States

ARTICLE INFO

Article history:
Received 29 December 2009
Received in revised form 5 October 2010
Accepted 26 January 2011
Available online 27 February 2011
Responsible Editor: G. Morabito

Keywords:
Slow start
Congestion control
Linux slow start
SACK processing
High-speed TCP protocols

ABSTRACT

Standard slow start does not work well under large bandwidth-delay product (BDP) networks. We find two reasons for this problem in three popular existing operating systems: Linux, FreeBSD and Windows XP. The first reason is that heavy packet losses occur because of the exponential increase of the congestion window during standard slow start. Recovering from heavy packet losses puts extremely high loads on end systems, rendering the end systems completely unresponsive for a long period of time, and results in a long blackout period without transmission. This problem commonly occurs with all three operating systems. The second reason is that some proprietary protocol optimizations applied to slow start happen to slow down the loss recovery followed by slow start. Although improving the system bottleneck by optimizing data structures is valuable especially for addressing the processing overload with heavy packet losses, it is not effective for the prolonged loss recovery caused by proprietary optimizations. In addition, a large number of packet losses are not desirable since they waste network bandwidth and lead TCP into frequent timeouts and loss synchronization which results in under-utilization of the network. We propose a new slow start algorithm, called Hybrid Start (HyStart), that finds a "safe" exit point for slow start at which it can terminate and safely advance to the congestion avoidance phase without causing heavy packet loss. HyStart uses ACK trains and RTT delay samples to detect whether (1) the forward path is congested or (2) the current size of the congestion window has reached the available capacity of the forward path. HyStart is a plugin to TCP senders and requires no change on TCP receivers. We implement HyStart for TCP-NewReno and TCP-SACK in Linux and compare its performance with five different slow start schemes on the TCP receivers of the three different operating systems on the Internet and also in lab testbeds. Our results indicate that HyStart works consistently well under diverse network environments, including asymmetric links, wireless networks, and high and low BDP networks. Especially with different operating system receivers (Windows XP and FreeBSD), HyStart improves the start-up throughput of TCP significantly by more than 2 to 3 times and is the default slow start algorithm of CUBIC since Linux 2.6.29.

1. Introduction

Long distance networks spanning several continents are growing in importance. Many multi-national companies are now centralizing their data centers for economic reasons. Thus, the high performance of TCP in these networks is critical for the effective operation of their data centers. These networks typically have large bandwidth-delay products (BDPs) ranging from several thousands to several hundreds of thousands. This is not a typical environment for which most existing commercial TCP stacks are optimized. Often we find that when these stacks run in these environments, they fall into some "rare" states where TCP achieves extremely low performance. One of

☆ An earlier short version [1] of this paper was presented at the International Workshop on Protocols for Fast and Long Distance Networks in 2008.

* Corresponding author.
E-mail addresses: sangtaeh@princeton.edu, sangtae.ha@ieee.org (S. Ha), rhee@csc.ncsu.edu (I. Rhee).

1389-1286/$ - see front matter © 2011 Elsevier B.V. All rights reserved.
doi:10.1016/j.comnet.2011.01.014

the most commonly cited problems of TCP in these networks is its slow window growth. There have been many proposals to adopt more scalable window growth functions to fix the sluggish performance of TCP in these networks. However, there are many other functions of TCP that still require greater optimization. One of these functions is slow start.

Standard TCP doubles the congestion window (cwnd) for every round-trip time (RTT) during slow start. However, the exponential growth of cwnd results in burst packet losses. Since the cwnd overshoots the path capacity an amount as large as the entire BDP, in large BDP networks, this overshoot causes strong disruption in the networks. A large number of packet losses are undesirable since they waste network bandwidth. They may also cause timeouts and high loss synchronization among many competing flows, which result in low utilization of networks. Furthermore, long bursts of packet losses caused by this overshoot also add burden to the end systems for loss recovery, and this burden often translates into consecutive timeouts and long blackout periods.

The selective acknowledgement (SACK) option [2–4] relieves some of these problems. As SACK informs the sender the blocks of packets successfully received, the sender can be more intelligent about recovering from multiple packet losses. SACK is currently enabled in most commercial TCP stacks [5]. However, for a large BDP network where a large number of packets are in flight, the processing overhead of SACK information at the end points can be overwhelming. This is because each SACK block invokes a search into the large packet buffers of the sender for the acknowledged packets in the block, and every recovery of a lost packet causes the same search at the receiver end. During fast recovery, every packet reception generates a new SACK packet. Given that the size of the cwnd can be quite large (sometimes, greater than 100,000), the overhead of such a search can be overwhelming. This system overload is devastating: it can prevent the system from responding to other services and processes, and it can cause multiple timeouts (as even packet transmissions and receptions are delayed) and a long period of zero throughput. Many operating systems attempt to optimize SACK processing using better data structures for packet buffers or limiting the number of SACKs. But we find that despite these optimizations, multiple packet losses still cause a significant CPU load or a reduced number of SACKs slows down loss recovery significantly, resulting in a blackout period These problems occurred consistently in all three dominant operating systems, Linux, Windows XP and FreeBSD, during slow start runs in large BDP networks.

There are clearly two general approaches to fixing the problem. One is to further optimize the SACK processing such that even under many occurrences of loss bursts, the system does not become overloaded. The other is to fix slow start so that the occurrences of loss bursts are reduced. Both approaches are important. As many embedded systems with low system resources are becoming popular and since slow starts are not necessarily the only cause of long bursts, the first approach is important. The second approach is also important since a large number of packet losses caused by aggressive slow starts not only burden end-systems, but also result in network under-utilization by leading TCP into frequent timeouts and incurring loss synchronization among competing flows. Moreover, SACK processing implementations have considerably been and will be improved with later versions. This paper focuses primarily on the second approach.

In order to examine the extent of damage caused by the overshooting of cwnd during slow start, we closely examine the SACK processing overhead of Linux by profiling related components. We evaluate the slow start performance optimizations of current TCP stack implementations in Linux, Windows XP and FreeBSD and discuss their pitfalls. We then present a new slow start algorithm, called HyStart [1], that effectively prevents the overshooting of slow start while maintaining full utilization of the network. While keeping the exponential growth of slow-start, HyStart finds the "exit" point where it can safely finish the exponential growth before overshooting and advance to congestion avoidance. The overshooting prevention of HyStart greatly reduces the occurrences of loss bursts and avoids system overload during fast recovery. HyStart requires modification only at the sender side of TCP and can be incrementally deployed on the Internet. In this paper, we demonstrate its performance by implementing HyStart and various other proposed solutions on the latest Linux kernel and testing them with Linux, FreeBSD, and Windows XP receivers in both real production networks and in a laboratory testbed. We report superior performance of HyStart over the other solutions in terms of network and CPU utilization.

To the best of our knowledge, the contributions of the paper are as follows:

(1) *A detailed analysis of system bottlenecks.* Even though the literature [6] provided the detailed analysis of Linux SACK processing, we particularly focused on the SACK processing overhead with TCP slow start and investigated the effect of burst losses on different versions of Linux systems. Specifically, we look into real Linux SACK implementations and confirm that significant processing overhead occurs when TCP recovers from burst losses. Searching a large number of lost packets from the retransmit queue during loss recovery puts an extremely high load on end systems, resulting in a long blackout period. We also explain protocol misbehavior with burst losses on popular operating systems.

(2) *A practical, non-intrusive implementation of packet-train-based estimation of available bandwidth for TCP.* We acknowledge that PacedStart [7] is the first proposal that applies packet pair techniques to TCP slow start. Unfortunately, it requires a modification on the transmission path of packets and the support of high-resolution system clock, which is not a part of the TCP layer. This reduces the chance of deployment on real systems. On the other hand, HyStart is a small plugin which conforms to the underlying layers and requires no significant modifications. The novelty of the HyStart algorithm is that it measures an ACK train passively without incurring high overhead on the TCP sender side, and it incorporates

a delay change into the algorithm in order to prevent burst losses when the network is congested. HyStart is therefore more friendly to the networks.

(3) *Practical evaluations of HyStart and other slow start algorithms.* While many slow start algorithms have been proposed for more than two decades, their evaluation is either limited to simulations, or to a minimal number of comparisons [8–11]. This hindered real deployment of the proposed solutions and thus the standard slow start algorithm had no chance to be improved. By recognizing this challenge, this paper implements HyStart and a large number of algorithms in the latest Linux kernel, and shows their performance results under diverse experimental scenarios in both real production networks and in a lab testbed.

2. Motivations

In this section, we identify two main reasons for the poor slow-start performance of current commercial TCP stacks. One is the overwhelming processing load during slow start in large BDP networks. The other is the proprietary optimizations of slow start performed by developers of various operating systems that inadvertently cause extremely slow packet loss recovery after multiple packet losses.

2.1. Processing overload during slow start

We investigate how system overhead can affect TCP performance during slow start. System overload is frequently observed during slow start which is followed by multiple timeouts or long blackouts. The problem occurs consistently with three popular operating systems: Linux, Windows XP and FreeBSD. Table 1 shows the end-system processing overhead for different network scenarios. Two TCP-SACK flows join at different times and we measure the CPU load (utilization) of the Linux senders and receivers. The processing overhead is likely to become problematic when the bandwidth, the one-way delay, and the buffer size are greater than 200 Mbps, 120 ms, and 4194 packets respectively. Also we can see that the CPU load increases as BDP increases and the maximum CPU load of each experimental run consistently shows 100% in the network of 400 Mbps and 120 ms one-way delay.

Table 1
End-system processing overhead. For each bottleneck bandwidth, one-way delay, and buffer size, we measure CPU loads of end systems and report the maximum CPU load among them and the percentage of runs showing a full CPU load. We repeated each experiment 50 times. Socket buffer auto-tuning is enabled for both end systems (Linux 2.6.23.9). The upper limit of buffer size is two times of a BDP buffer size, so that a TCP sender can effectively double the cwnd during slow start. At the last round of slow start, the TCP sender maintains two times of BDP packets at a retransmit queue and overshoots (loses) half of these packets. Interestingly, the auto-tuning reduces the occurrence of full CPU load by throttling the maximum number of packets in flight. We measured 40 s after slow start.

Bandwidth	One-way delay	Buffer size (100% BDP)	Maximum CPU load	% of runs with full CPU load
100 Mbps	60 ms	1048 packets	36%	0%
100 Mbps	120 ms	2096 packets	48%	0%
200 Mbps	60 ms	2096 packets	36%	0%
200 Mbps	120 ms	4194 packets	100%	58%
400 Mbps	60 ms	4194 packets	100%	38%
400 Mbps	120 ms	8388 packets	100%	100%

In order to study the problem in more detail, from the 400 Mbps bandwidth experiment shown in the last row in Table 1, we measure the average time for which the CPU load remains 100% and the average time for which the throughput of TCP remains zero or very low. Table 2 shows the results. A CPU load of 100% happens on both TCP senders and receivers with minor difference. On average, full CPU load persists less than 2 s but the impact of this overload lasts more than 14 s.

As an illustration, Fig. 1 (a), (b) and (c) show the throughput observed at a router in a network of 400 Mbps bandwidth and 120 ms one-way delay. We turned off auto-tuning and explicitly set the socket buffer size to two times of BDP since we found that the auto-tuning in Linux occasionally delays the exponential growth of cwnd. The experimental setup of our testbed is detailed in Section 5. From Fig. 1, both flows have almost zero throughput for more than 30 s after a timeout. These blackouts follow after the saturation of CPU utilization, occasionally at both the senders and receivers. When the system reaches overload, it cannot react fast enough to perform loss recovery. This results in timeouts. Repeated losses of retransmitted packets during timeouts also cause back-to-back timeouts with exponential backoff of RTO (retransmission timeout) timers where the senders do not transmit any packets.

2.1.1. Processing overhead at TCP senders

We first examine the SACK processing overhead of TCP senders in Linux. Fig. 2 illustrates how a Linux TCP sender processes an ACK packet. The function *tcp_ack*() is called at the reception of ACK, and it checks whether the ACK number is within the left and right edges of a congestion window – *snd_una* and *snd_nxt*. Then *sacktag_write_queue*() is called to search for the acknowledged packets in the *retransmit queue* and estimate the number of packets in flight (*packets_in_flight*).

The Linux TCP stack ensures that the variable, *packets_in_flight*, always matches the size of cwnd. After that, *tcp_clean_rtx_queue*() is called to remove and free the acknowledged packets from the retransmit queue, and the variable *packets_out* is reduced by the number of the freed packets. The function *tcp_fastretrans_alert*() executes fast retransmit and updates the scoreboard that keeps track of lost and acknowledged packets. For every ACK, *tcp_cong_avoid*() is called which then increases cwnd by entering slow start and congestion avoidance phases.

The retransmit queue is implemented using a linked list to hold all of the packets currently in flight to the receiver. Each SACKed packet is searched in this list sequentially. The size of the list is proportional to the number of packets in flight. Three functions marked (1), (2) and (3) in Fig. 2 are the most CPU-intensive as they need multiple traversals of the retransmit queue. Suppose that the TCP sender has sent W packets in the retransmit queue and receives $\frac{W}{2}$ delayed ACKs in one RTT round. Let us examine each of these functions below.

Table 2
More details on the experiment of 400 Mbps bandwidth and 120 ms delay network, shown in the last row in Table 1. We only measured 40 s right after slow start, and therefore the average time of zero TCP throughput is limited to 40 s. CPU load remains more than 100% for 1.64 s in our experiment and this results in more than 14 s black-out in TCP throughput. The average and standard deviation are presented.

End hosts	% of full CPU load	Average time of full CPU load	Average time of zero throughput
TCP sender	52%	1.83 (0.69)	16.95 (11.66)
TCP receiver	48%	1.44 (0.64)	12.53 (13.99)
Total	100%	1.64 (0.69)	14.82 (13.02)

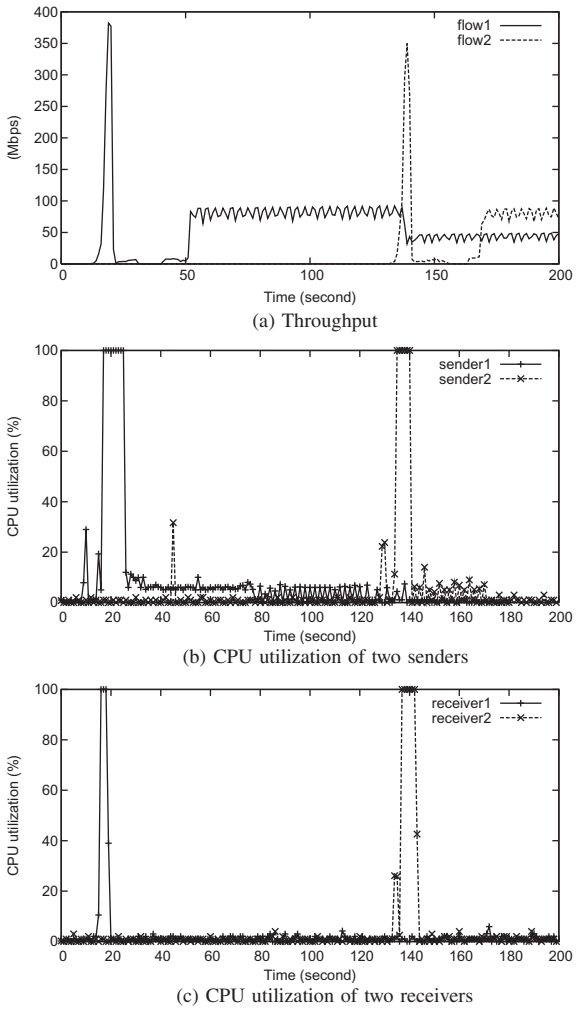


Fig. 1. The example of two black-out periods. The bandwidth, one-way delay, and buffer sizes are set to 400 Mbps, 120 ms and 100% BDP of a network.

- *sacktag_write_queue*(): SACK contains up to four SACK information blocks, each of which consists of a block of the sequence numbers of packets received out of sequence by the receiver. For every SACK, it searches in the retransmit queue for the packets whose sequence number is in between each SACK block and updates their states. In the worst case, one SACK block causes a traversal of the entire retransmit queue. Therefore, it requires $O(W^2)$ running time in one RTT round.
- *tcp_clean_rtx_queue*(): It frees the packets cumulatively acknowledged in each incoming ACK (i.e., packets in the left edge of cwnd). Each incoming ACK typically frees two packets in the retransmit queue due to delayed ACK. In the worst case, one cumulative ACK packet acknowledges W packets all at once. Therefore, $\frac{W}{2}$ ACK packets require $O(W)$ running time in one RTT round.
- *tcp_fastretrans_alert*(): Invoked at the reception of a duplicate ACK, it updates the variable *left_out* to account for the number of lost packets. It usually marks the first packet in the retransmit queue to be retransmitted first. In the worst case, a retransmission timeout happens and all packets in the retransmit queue are timed out. As a result, it needs as much as one traversal of the retransmit queue to mark all packets in the retransmit queue and therefore requires $O(W)$ running time.

We profile the CPU usage of the three functions using OProfile [12] in Linux. OProfile collects the number of standard CLK_UNHALTED counters for the functions in the kernel and presents their CPU utilization. We consider the same network scenario shown in Fig. 1 for profiling. Fig. 3 (a) shows the results of 100 runs and we plot them on a log scale. The function *sacktag_write_queue*() consumes between 10% and 100% of the CPU clocks. Also the overhead of *tcp_clean_rtx_queue*() is larger than *tcp_fastretrans_alert*() because of the cost of freeing memory, but it is significantly less than that of *sacktag_write_queue*().

To see the relationship between W (cwnd) and CPU utilization, we change the bottleneck buffer size from 1% to 200% of the path BDP while maintaining the same testing parameters. Fig. 3 (b) plots the CPU usage of the three functions on a log scale as W increases. The CPU usage of *tcp_clean_rtx_queue*() is quite independent of the buffer size while *tcp_fastretrans_alert*() shows a small increase of CPU time. We note that *sacktag_write_queue*() requires significantly more CPU time as W increases.

Note that maliciously manipulated SACK options can easily overload the above three functions on the TCP sender, and this opens up possibilities of Denial of Service Attacks (DoS) from a malicious TCP client [13].

2.1.2. Processing overhead at TCP receivers

In Linux TCP, *tcp_data_queue*() processes data in received packets. If packets are received in order, it copies the data in the packets to the user space. When packets arrive out of order, it puts the packets into the out-of-order queue. The packets in the out-of-order-queue can only be freed when the left edge of the receiver window advances. When receiving a retransmitted packet, the receiver searches the out-of-order queue to place the packet in

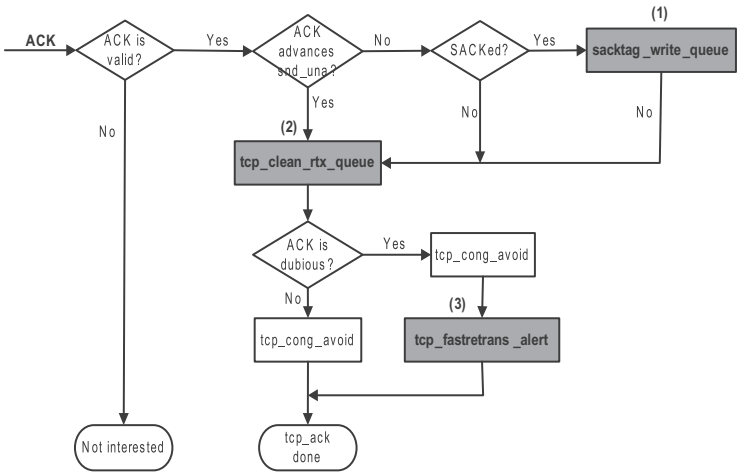
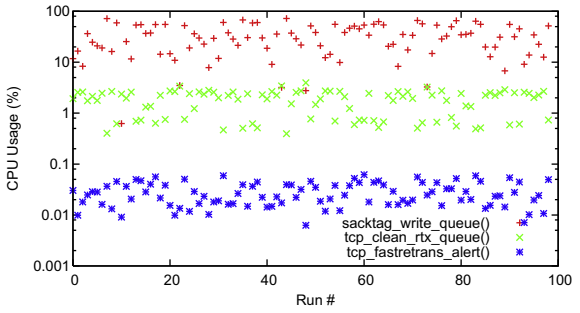
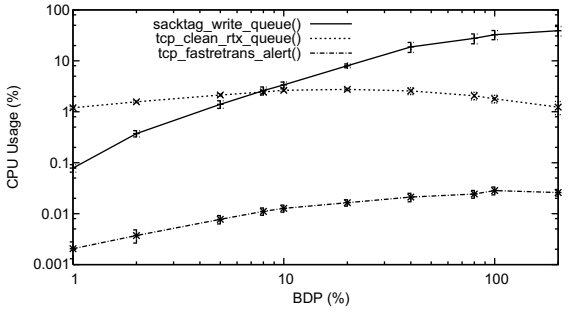


Fig. 2. Linux TCP ACK processing.



(a) SACK processing overhead



(b) BDP (W) vs. CPU utilization

Fig. 3. CPU utilization of the three functions in TCP SACK processing.

the right order. Linux TCP uses a linked list for the out-of-order queue. The increase of the right edge of the congestion window during fast recovery does not significantly affect the performance as it typically goes to the end of the queue. However, with a large number of retransmitted packets, the processing overload occurs when each retransmitted packet causes a traversal of the queue to place it in the right place.

We profile the receiver kernel and measure the system clocks of *tcp_data_queue*() of the second receiver shown in Fig. 1 (c). Our profile shows that it consumes more than 30% of CPU time for the entire run, and is a major contributor to the state of 100% CPU utilization at around the 140th second in Fig. 1 (c). Copying data to user space takes only 2% of CPU time.

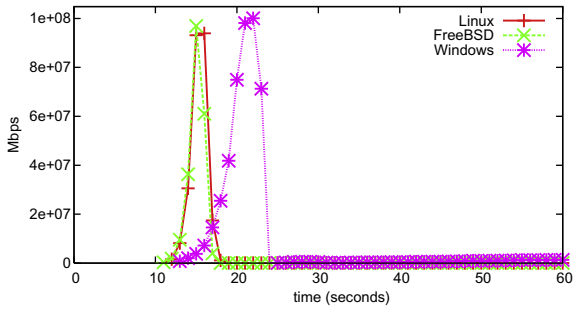
2.2. Protocol misbehavior during slow start

In this section, we examine protocol misbehaviors of slow-start implementations in various operating systems that are yet another cause of sluggish performance during TCP start-up. To alleviate the effect of system overload and focus only on protocol misbehaviors during long loss bursts, we run the following experiment. We scale down BDP by adjusting the bandwidth to 100 Mbps, one-way delay to 120 ms and the buffer size to 100% of BDP (2050 for 1 Kbyte packet) of the network. Finally, we use tcpdump to track protocol behaviors.

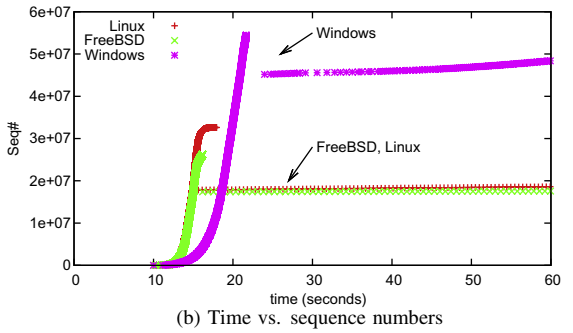
Fig. 4 shows the results of the experiment involving TCP-NewReno senders and receivers of various operating systems. All stacks show very poor performance – after the overshooting of cwnd during slow-start, all stacks enter fast retransmit and recovery, but their recovery speed is very slow. This happens because these stacks implement "*Slow-but-Steady variant of NewReno*" (SS-NewReno) [14] where the TCP sender resets the timeout timer whenever a partial acknowledgment indicating the left edge of the window arrives. This effectively prevents the sender from entering timeout during fast recovery. When almost every packet in a window is lost (other than some duplicate ACKs to trigger the initial fast recovery), each retransmission recovers only the left-edge of the current window. Thus it takes a long time for the sender to recover all of the lost packets.

This problem of a slow ramp-up after the use of slow start does not occur in TCP-SACK since the sender is better informed about lost packets beyond the left-edge and retransmits all of the lost packets almost immediately. Fig. 5 shows the performance of TCP-SACK for the three operating systems.[1] We find that all stacks now enter a

[1] For the FreeBSD experiment, we use a Linux receiver because the latest version of FreeBSD does not enable SACK properly although FreeBSD sender and receiver negotiate for the SACK option. We investigated this problem and found that FreeBSD failed in negotiating a window scaling factor properly even though the application used a large fixed memory.

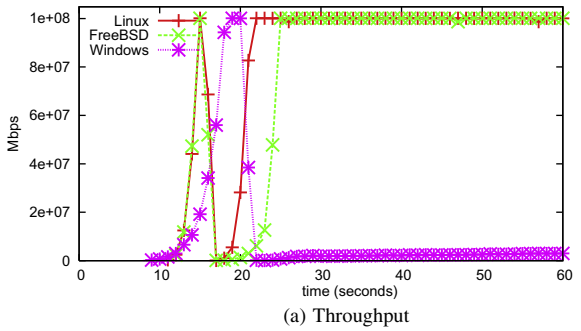


(a) Throughput

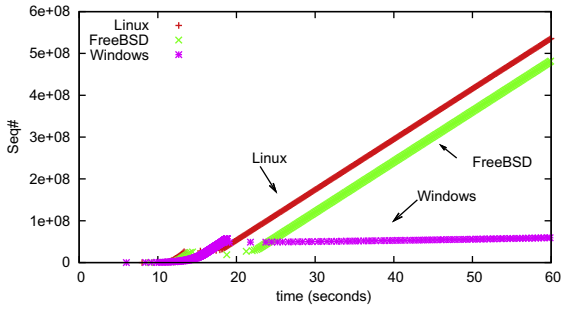


(b) Time vs. sequence numbers

Fig. 4. TCP-NewReno on the three systems.



(a) Throughput



(b) Time vs. sequence numbers

Fig. 5. TCP-SACK on the three systems.

timeout quickly. While FreeBSD and Linux recover relatively quickly, Windows XP shows a very slow recovery. This is because the number of SACK blocks sent by the Windows XP receiver is limited and after a specified threshold, the receiver simply reports only cumulative ACKs. This is an optimization added by the Windows developers for various purposes, for example, to reduce system overloads during fast recovery and to limit buffering at the receiver. This forces the sender to act like TCP-NewReno, showing similar behavior as in Fig. 4. We can also obtain the same performance result even if we use a Linux sender along with a Windows XP receiver.

3. Existing solutions

There are basically two approaches to fixing the start-up performance of TCP. One is a reactive approach. While letting burst losses occur, it devises better techniques to handle many multiple packet losses more efficiently and effectively. Most operating systems use this approach. This approach is important as it can be applied to any situation where burst losses occur (not only to slow-start). The second approach is to prevent burst losses by designing a better slow-start protocol. It is important since packet losses in general are not desirable as they waste network capacity, lead to timeouts, and can result in loss synchronization and reduced utilization. Moreover, it can prevent both network and system overload. In this section, we discuss existing techniques for the two approaches.

3.1. Linux: linked-list optimization

The Linux SACK processing overhead has been known in the community. Kelly's Scalable TCP provided a SACK fast-path patch that addresses the overhead of SACK processing [15]. Also, Li raised the issues related to SACK processing and showed that it still requires a greater optimization [16]. Even et al. highlighted issues with SACK processing in high BDP networks and analyzed the overhead in more detail by profiling the CPU utilization of different functions related to SACK processing [6]. Compared to their work, our work provides (1) the analysis focused on SACK processing overhead with TCP slow start, (2) the explanation of system bottlenecks not only on TCP senders but also on TCP receivers, (3) the comparison between different versions of Linux and (4) the explanation of protocol misbehavior with burst losses. Most recently, Järvinen et. al significantly improved the performance of SACK scoreboard related operations almost two times as compared to previous kernel releases [17]. McManus looked at the SACK implementation in Linux and its performance and reported the possibilities of Denial of Service Attacks (DoS) through simple manipulations of SACK information even under commodity network conditions [13].

Since Linux 2.6.15, SACK processing has undergone several improvements involving better caching and data structures to reduce the look-up time in the retransmit queue. Fig. 6 shows the improvement of Linux SACK processing over the evolution of the Linux kernel for the same experiment shown in Fig. 1. Linux 2.6.14 is the version that uses a linked list without any caching. The other three versions include incremental optimizations using caching.

We run 10 sets of the experiment shown in Fig. 1 with different kernel versions and measure the CPU utilization of *sacktag_write_queue*(). We can see the overhead reduces as the kernel version increases. However, the SACK

processing overhead has been reduced by only about 20% in the latest version.[2] Relieving the system bottlenecks by efficiently optimizing data structures is valuable but it still requires further improvement.

3.2. FreeBSD – limiting CWND

FreeBSD implements a method to limit the congestion window to the BDP of the network. It estimates the bandwidth by dividing the amount of packets it sent by the minimum RTT observed. This method is very similar to the method used in TCP-Vegas [9] and TCP-Westwood [18], and can prevent the overshooting experienced with slow start. The sender always sends at most the estimated bandwidth. This approach works well in some environments, but we observe that it can lead to frequent underestimation because of noise in the RTT measurement. Because of bursty packet transmissions of TCP especially during slow start, packets are frequently queued in the bottleneck buffer even if the current *average* sending rate is lower than the capacity. Thus, the proposed technique ends slow start, often prematurely, as the estimation technique is too simple and often underestimates the BDP. For instance, we find that TCP-SACK with this technique achieves 15.6% link utilization in the 100 Mbps and 120 ms one-way delay networks. We have additional performance results of TCP-Vegas in Section 5.2.

3.3. Windows – suppressing SACK options

When a TCP-SACK receiver on Windows XP has many SACK blocks to report, it suppresses the SACK options and sends only cumulative ACKs. The TCP-SACK receiver discards all the out-of-order packets received after reaching this limit. Sending only cumulative ACKs prevents the TCP sender from being overloaded. But the sender acts like TCP-NewReno and exhibits very slow recovery as shown in Fig. 4.

3.4. Existing slow start algorithms

Hoe [8] proposes to estimate the bottleneck bandwidth using a packet-pair measurement, and to use the estimated value to set the *ssthresh* of TCP-NewReno. The study by Dovrolis et al. [19], however, shows that this estimation is not robust enough and may need sophisticated filtering. It is also problematic because other cross traffic may hinder proper estimation, resulting in a frequent overestimation of the bottleneck link bandwidth.

Vegas [9] introduces a modified slow start mechanism which allows an exponential growth of cwnd at alternating RTTs and, in between, compares its current transmission rate with the expected rate to determine whether the path has room for rate increase. The modified slow start of Vegas is known to cause a premature termination of slow start because of an abrupt increase of RTT caused by temporal queue build-ups in the router during bursty TCP transmissions [11].

Limited slow start [10] is an experimental RFC. It is designed to prevent a large number of packet losses in one RTT by limiting the increment of a congestion window to *max_ssthresh/2* per RTT. But using the fixed number *max_ssthresh* does not scale well. For example, assume the upper bound of the capacity is 5000 packets and max_ssthresh is set to 100. It takes 21 s[3] before reaching a congestion window size of 5000 with an RTT of 200 ms. Quick-Start [20,21] determines its allowed sending rate very quickly, but it needs cooperation with the routers that support Quick-Start along the path.

Adaptive start [11] repeatedly resets its *ssthresh* to the value of the expected rate estimation (ERE) when ERE is greater than ssthresh. Therefore, with adaptive start, TCP repeats the exponential growth and linear growth of the window until a packet loss occurs. However, adaptive start can be slower than standard slow start, and it is not easily integrated with congestion control algorithms other than TCP-Westwood [22].

Paced start [7] probes an available bandwidth during slow start by precisely controlling the gap between the packets.

It, however, requires a precise control of packet transmissions and ACK receptions. This is very costly and the implementation is often not trivial for high BDP networks [23].

CapStart [24] switches between slow start and limited slow start by identifying whether the network path between two TCP end-points has a bottleneck. If the network path has a bottleneck (less than 90% of the capacity compared to the local interface at TCP sender), it runs the limited slow start algorithm. Otherwise, it runs the standard slow start algorithm. For capacity estimation, it uses a packet-pair approach while favoring a longer inter-packet gap for safe estimation.

Padmanabhan et al. [25] suggest the use of cached information from previous connections. Wang et al. [26] throttle the exponential rate of increase when the congestion window approaches *ssthresh*. However, in all cases, because available bandwidths keep changing and it is difficult pick an initial value of *ssthresh*, there is always a high possibility of wrong decisions.

4. Hybrid start (HyStart)

In this section we describe our slow start algorithm, called *HyStart* that reduces burst packet losses during slow start and hence achieves better throughput and a lower system load. The algorithm does not change the doubling of cwnd during slow start, but based on clues from ACK spacing and round-trip delays it heuristically finds safe exit points at which it can finish slow start and move to congestion avoidance before cwnd overshoots. When packet losses occur during slow start, HyStart behaves in the same way as the original slow start protocol. HyStart is a plugin to the sender

[2] Linux 2.6.26-rc4 was the latest version of Linux at the time of this experiment. Note that Järvinen's recent SACK improvement to Linux 2.6.30 reduced the CPU utilization to reasonable ranges, but we observed significantly long TCP timeouts with zero throughput in many cases. For fair comparison, we exclude the results with Linux 2.6.30.

[3] $\log(100) + (5000 - 100)/(100/2) = 105$ RTT rounds. When the RTT is 200 ms the total time is around 20 s.

side of TCP and is easy to implement as it uses only TCP state variables available in common TCP stacks.

4.1. Safe exit points

The main objective of slow start is to quickly ramp up cwnd as close as possible to the capacity of the forward path while maintaining TCP ACK clocking. The capacity of a path can be informally defined by the sum of unused available bandwidth on the forward path and the size of buffers at bottleneck routers. Typically, this capacity estimation is given by *ssthresh*. But when a flow starts, ssthresh is set to an arbitrarily large number. Thus, slow start may overshoot beyond the capacity of the forward path. Similar situations may occur when path conditions change after the timeouts so that the ssthresh set at the timeout event is an incorrect estimation of the network capacity. These factors motivate the need for a more intelligent slow start that finds a safe exit point when it can stop slow start and advance to congestion avoidance.

The safe exit point corresponds to the size of the cwnd before slow start finishes safely without incurring losses and without low network utilization. Let the unused available bandwidth of the forward path, the minimum forward path one-way delay and available buffer space of the forward path be B, D_{min} and S respectively. Then a safe exit point must be less than $C = B \times D_{min} + S$. If cwnd gets larger than C, then packet losses occur. Slow start cannot finish arbitrarily before this upper bound which will cause low network utilization, so need a lower bound. After slow start, congestion avoidance will increase cwnd if there is no loss. During steady state where the average capacity of a path does not change, standard TCP reduces cwnd by half ($C/2$) for fast recovery, after which congestion avoidance takes over to restore cwnd to half ($C/2$). This also happens during a timeout when ssthresh is set to ($C/2$). The congestion avoidance algorithm of standard TCP requires a BDP amount of buffer size to achieve full utilization under synchronized losses [27]. This is because when the packet is dropped and the sender pauses to acknowledge many outstanding packets after halving cwnd, the router needs the buffer to be large enough, so that it can keep the router fully utilized by making the router buffer never go empty [28]. By contrast, slow start only needs to fill the packets in the forward path ($B \times D_{min}$), so that it can immediately advance to the congestion avoidance period without much performance penalty. Therefore, assuming that the packet loss happens immediately followed by slow start, which is the worst case scenario, it is reasonable to set the lower bound for a safe exit point at $(B \times D_{min})/2$, which is identical to $C/2$ minus the half of the buffer space. In summary, a safe exit point is bounded between $B \times D_{min} \times \beta$ and C, where β is the multiplicative decrease factor of cwnd during fast recovery (i.e., $0 < \beta < 1$). Standard TCP sets β at 0.5 and other high speed variants use a value larger than 0.5.

4.2. Algorithm description

HyStart uses two sets of information – the time space between consecutively received ACK packets and round-trip delays then applies heuristic methods to look for an indication that the current value of cwnd is larger than the available bandwidth. HyStart uses a passive measurement and estimation technique – it does not inject a probe packet of its own. Below we describe the two methods. Both run independently and concurrently and slow start exits when either of them detects an exit point. Algorithm 1 shows its pseudocode. The Linux implementation is also available at [29].

Algorithm 1 Hybrid Start (HyStart)

```
// When found > 0, it leaves slow start.
Initialization:
// We sample initial 8 ACKs every RTT round, so
   the lower bound of ssthresh is set to 16 by
   considering a delayed ACK.
low_ssthresh ← 16        nSampling ← 8
found ← 0
At the start of each RTT round:
begin
  if ! found and cwnd ⩽ ssthresh then
    // Save the start of an RTT round
    roundStart ← lastJiffies ← Jiffies
    lastRTT ← curRTT
    curRTT ← ∞
    // Reset the sampling count
    cnt ← 0
end
On each ACK:
begin
  RTT ← usecs_to_jiffies(RTT_us)
  dMin ← min(dMin, RTT)
  if ! found and cwnd ⩽ ssthresh then
    // ACK is closely spaced, and the train
    length reaches to T_forward?
    if Jiffies − lastJiffies ⩽ msecs_to_jiffies(2) then
      lastJiffies ← Jiffies
      if Jiffies − roundStart ⩾ dMin/2 rhen
        // First exit point
        found ← 1
    // Samples the delay from first few packets
    every round
    if cnt < nSampling then
      curRTT ← min(curRTT, RTT)
      cnt ← cnt + 1
    // Delay increase (η) should have some
    bounds
    η ← min(8, max(2, ⌈lastRTT/16⌉))
    // If the delay increase is over η
    if cnt ⩾ nSampling and curRTT ⩾ lastRTT + η then
      // Second exit point
      found ← 2
    if found and cwnd ⩾ low_ssthresh then
      ssthresh ← cwnd
end
Timeout:
begin
  // Reset the variables on timeouts
  dMin ← ∞    found ← 0
end
```

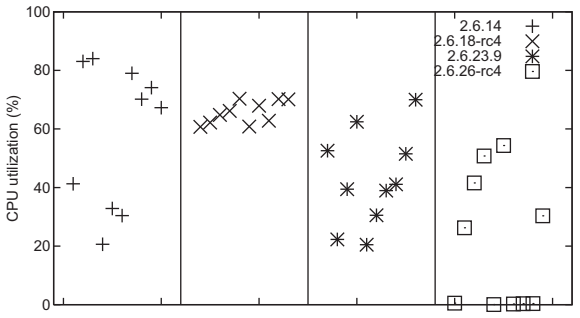


Fig. 6. Improvement of SACK processing on Linux after slow-start in a large BDP network. The SACK processing have been improved over the evolution of Linux kernel.

4.2.1. ACK train length

Estimating unused bandwidth in a path has been an active topic of research [30–32]. Many bandwidth estimation techniques, such as packet-pair [33] and packet train [19] use active probing to estimate bandwidth by sending probes back-to-back in a short period of time.

Dovrolis et al. [19] recently showed that the mean of packet train dispersion can be translated into Average Dispersion Rate (ADR), a ratio between the available bandwidth and the maximum capacity of the path. Formally, the sender sends N back-to-back probe packets of size L to the receiver. When $N > 2$, we call these probe packets a *packet train*. The packet train length can be written as $\Delta(N) = \sum_{k=1}^{N-1} \delta_k$, where N is the number of packets in the train and δ_k is the inter-arrival time between packets k and $k+1$. Using the packet train length, the receiver measures the bandwidth $b(N)$ as follows.

$$b(N) = \frac{(N-1)L}{\Delta(N)} \tag{1}$$

However, packet-pair or train techniques are not practically implementable in existing commercial TCP stacks. They require modifications to both TCP sender and receiver programs. This requirement hinders their incremental deployment. Even with such modifications, it is not practical to accurately measure the time spacing between two consecutively received packets in current operating systems because such a measurement requires a high-resolution system clock and real-time interrupt handling. Because of the extremely short time spacing between two consecutively received probes due to the high speed of the network, even slight system delays in getting the timestamps of probes may cause significant errors in the estimation. Under a high system load of packet transmission during slow start, avoiding system delays is not trivial.

To remedy these problems, we propose the following approach. For now, suppose that we can measure the unused available bandwidth B of the forward path and the minimum forward one-way delay D_{min}. The bandwidth and delay product of the path is $b(N)D_{min}$. Since $b(N) = \frac{(N-1)L}{\Delta(N)}$, a safe exit point occurs when cwnd (i.e., $(N-1)L$) becomes as close to the one-way forward path BDP as possible. Cwnd becomes equal to the BDP when $\Delta(N)$ is equal to D_{min}. Thus, by checking whether $\Delta(N)$ is larger than D_{min}, we can detect whether cwnd has reached the available capacity of the path. This supposition permits the following heuristics to measure the BDP of the network in real systems.

1. We are able to have data packets transmitted during slow start as packet probes for a packet train. During slow start, many packets are sent in bursts. We can use those packets which are transmitted within the same window as a packet train. This removes the need for active probes. To estimate $\Delta(N)$ from the sender side, we use the train of ACKs received in response to a packet train. Fig. 7 illustrates the difference between an ACK train and a packet train. Since ACKs take reverse paths, the time spacing between two consecutively received ACKs, λ_i, is always larger than δ_i. The total sum of λ_i, $\Lambda(N)$, is always larger than $\Delta(N)$. This permits conservative estimation of the available capacity of the path.
2. It is not practical to measure δ_i directly without a high resolution clock. But we do not need individual samples of δ_i or λ_i. Instead, our scheme requires the sum of inter-arrival times of packets in a train. We measure $\Lambda(N)$ by measuring the time period between the receptions of the first and last ACKs in an ACK train.
3. It is not feasible to measure D_{min} on the sender side. We approximate it by dividing the measured minimum RTT by two. This approximation is not accurate when the path is asymmetric. Below, we explain why this does not lead to under or over-estimation of the capacity.

Discussion. There are several issues with the above heuristics. First, if the reverse path is heavily congested, then $\Lambda(N)$ can be much larger than $\Delta(N)$. Since TCP ACKs are less than 50 bytes, it is not very common that ACKs are being delayed because of congestion. In this case, it allows slow start to exit before the forward path becomes saturated. Exiting slow start early in this case is reasonable because the reverse path is too congested to carry even the ACK traffic.

Second, the receiver may use a delayed ACK scheme where ACKs are transmitted only for alternate packets received. In some systems, ACKs are arbitrarily delayed, but ACKs are always generated immediately at the reception of a packet (however, not necessarily at the reception of every packet). The ACK delay does not affect the performance of HyStart greatly because $\Lambda(N)$ is computed by taking the time difference between the reception times of

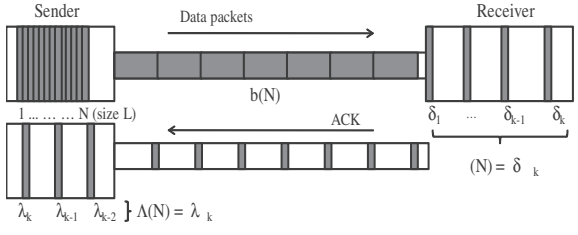


Fig. 7. ACK train measurement.

the first and last ACKs in a train. Since ACKs are triggered immediately after the reception of a packet, $\Lambda(N)$ may contain the delay for the last delayed ACK. Since an ACK train is typically large, this delay does not move the exit point beyond its safety bounds. To alleviate this problem, we filter out the ACK train length samples if the last ACK of a train is significantly delayed and mixed with the ACKs for the next train.

Third, path delay asymmetry may significantly thwart the correct estimation of safe exit points. Consider this situation. Suppose that a and b are the forward and reverse path one-way delays respectively. We estimate a by $(a+b)/2$. Suppose again that K is the BDP of the forward path and K' is our estimated BDP, and $\Lambda(N) = \Delta(N)$. Then if $a = b$, HyStart can precisely compute K. If $a \neq b$, then K'/K is $(a+b)/(2a)$. Considering the safe exit point bounds discussed in Section 4.1, our scheme satisfies the bounds if K'/K is larger than β, but less than $1 + S/K$. Since β is 0.5 in standard TCP, as long as b is larger than 0, it satisfies the lower bound. Thus, when the reverse path delay is much smaller than the forward path delay, under-utilization is very unlikely. For the upper bound, suppose that $S = \alpha K$. Then if K'/K is less than $1 + \alpha$, our scheme meets the upper bound, which means that b must be less than $a(2\alpha + 1)$. If $\alpha = 1$ (i.e., S is as large as the BDP), b can be as large as $3a$ to meet the upper bound. According to a recent measurement study of delay asymmetry on the Internet [34], the fraction of paths whose reverse path delays are 3 times longer than the forward path delays is less than 5%. Thus, for those fractions of Internet paths, HyStart behaves like standard slow start since it may overestimate the BDP and the overshooting of cwnd will trigger packet losses as in standard slow start.

4.2.2. Delay increase

When a path is not completely empty and one or more flows are competing for the same path, it may not be possible to measure the minimum RTT of the path. So the above ACK-train-based technique would be less effective. To handle this situation, we use the increase in round-trip delays as another metric to find the safe exit point. However, as TCP sends packets in bursts, it causes temporary queuing even if cwnd is less than the available BDP. Thus, measuring RTT using all packets may lead to erroneous exit points. In fact, this is a common problem of many delay-based slow start schemes [9]. We remedy this problem by using the RTT samples of a few ACKs at the beginning of each ACK train. Since packets arriving at the beginning of each train do not suffer from queuing caused by packet bursts, these samples return more accurate estimations of persistent queuing delays. Suppose that RTT_k is the average of RTTs of a few packets in the beginning of the kth train. We trigger an early exit when RTT_k is larger than $RTT_{k-1} + \eta$ where η is a fixed threshold. This scheme does not require the estimation of the minimum delay of a path and thus can be used for both congested and lightly loaded networks. Note that our delay-based technique can also be effective even when the network is asymmetric, especially when the reverse path delays are much longer than the forward path delays.

5. Experimental evaluation

5.1. Experimental setup

We set up a dumbbell topology composed of a set of Dell Intel Xeon 2.8 GHz servers with 1 gigabyte RAM as shown in Fig. 8 where two dummynet routers are located at the bottleneck between two end points. Two servers on each side are dedicated for sending and receiving long-lived TCP flows and Iperf [35] is used to measure the goodput of the TCP flows. The other two servers on each side are used to generate cross background traffic by using a modification of Surge [36] (a Web traffic generator) and Iperf for emulating long-lived TCP flows such as FTP connections.

The socket buffer size of background traffic machines is set by default to 64 KB while four TCP sender and receiver machines are configured to have a very large buffer so that the congestion control algorithms of TCP senders are affected only by the algorithm itself. Precisely, we enable the auto-tuning for TCP send and receive buffers and set the upper limit at two times of BDP of a path, so that slow start can fully evolve during the experiment. Also, Dummynet has been installed on two FreeBSD machines in the middle of the dumbbell and controls the bottleneck bandwidth, the round-trip time of the flows, and the buffer size in the bottleneck. Specifically, the first Dummynet router monitors the throughput of the bottleneck link and the second Dummynet router assigns per-flow delays to background traffic flows based on the empirical distribution from an Internet measurement study [37].

Linux 2.6.23.9, FreeBSD 7.0 and Windows XP are installed on the two TCP servers at each side. Two dummynet routers and four TCP servers (two TCP servers at each side) are tuned to generate and forward traffic close to 1 Gbps. In order to eliminate any overloading of the Dummynet routers, we set the bottleneck bandwidth below 400 Mbps. TCPProbe [38] and SIFTR [39] are used for actively tracking the TCP state variables in Linux and FreeBSD respectively. For Windows XP, we use Tcpdump. All performance measures are averaged from at least 10 runs. Results are reported with 95% confidence interval.

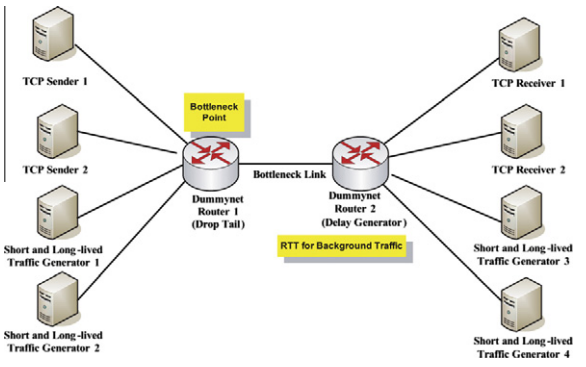


Fig. 8. Testbed.

5.2. Comparison with other slow starts

We compare the typical behaviors of various slow-start protocols including HyStart and those discussed in Section 2. All protocols are implemented over TCP-SACK in Linux 2.6.23.9. In the tests, the bottleneck bandwidth is set to 100 Mbps, the RTTs of two flows are set to 102 ms, and the buffer size is set to 100% BDP. We add no background traffic for this experiment.

Fig. 9 presents the trajectories of cwnd and ssthresh of six different slow-start protocols discussed in Section 5.2. The original standard slow start of TCP-SACK (SS) (a) shows a high burst of packets at around ten seconds and experiences a high rate of loss. Limited slow start (LSS) (b) reduces the burst losses observed in SS (a) by limiting the increment of the congestion window in one RTT. With HyStart (c), using the information of the ACK train length, the first flow finishes slow start at approximately packet 870 at which point, cwnd reaches the BDP of the path. The second flow uses a delay increase to gauge the congestion on the path caused by the first flow, and leaves slow-start early on. Adaptive Start (AStart) (d) calculates ERE upon receiving ACKs and sets ssthresh to ERE if ERE is larger than cwnd during slow-start. But ERE is consistently smaller than cwnd in this experiment. As a result, AStart shows the same overshooting behavior as SS. The packet-pair slow-start (PSS) (e) shows an inconsistent estimation of path capacity for each run because of the lack of high resolution clocks and real-time interrupt handling. The two flows estimate the capacity of 100 Mbps path to be 248 Mbps (2300 packets) and 308 Mbps (2800 packets) respectively. Also, the modified slow-start of Vegas (VStart)(f) terminates slow-start prematurely.

5.3. Impact of delayed ACK schemes

We evaluate HyStart under various ACK schemes including (a) a quick ACK, (b) a quick ACK initially and a delayed ACK later on, and (c) a delayed ACK. A quick ACK sends an ACK for each received packet. For delayed ACKs implemented in Linux, an ACK is sent for every two data packets received. When an ACK is not delayed, the spacing between consecutive ACKs is small and consistent. This makes a packet-train measurement effective. A delayed ACK may disturb the estimation of the ACK train length.

In this experiment, we introduce background traffic as it may disturb the behavior of the algorithm by varying available link bandwidths. The bottleneck bandwidth is set to 400 Mbps and the buffer size is set to 100% BDP. Two TCP-SACK flows with the same one-way delay of 80 ms start within the first 10 s of testing. The background traffic we generate includes medium-sized flows [40] whose average throughput is around 50 Mbps. The background traffic is introduced in both forward and reverse directions of the path.

Fig. 10 tracks the cwnd and RTT of two flows for three different ACK schemes while running HyStart. Fig. 10 (a) confirms that no delayed ACKs are found across two

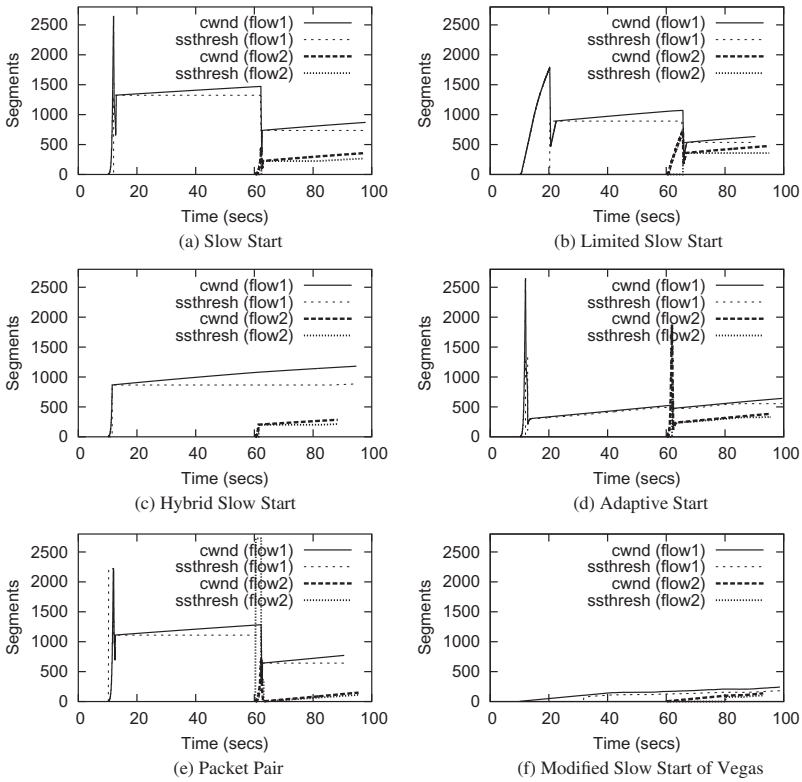


Fig. 9. Two TCP-SACK flows with five different Slow Start proposals. For this experiment, we fix the bandwidth to 100 Mbps, RTT to 106 ms, and the buffer size to 100% BDP of a flow. The BDP for this experiment is around 883 packets. Therefore, when *cwnd* is between 883 and 1766 packets, the link is fully utilized.

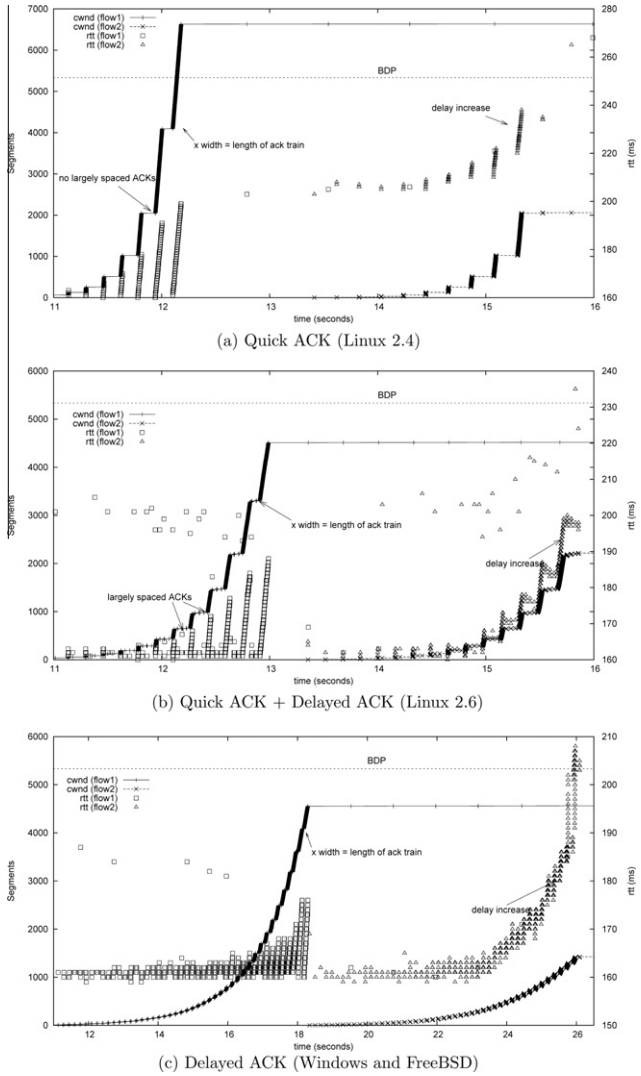


Fig. 10. Two TCP-SACK flows with HyStart, by varying the ACK schemes on the receivers.

consecutive RTT rounds. Hence, we know that it correctly calculates the ACK train length. The exit point is the round in which cwnd crosses over the BDP of the forward path. Linux 2.6, however, deliberately sends a quick ACK for up to an initial 16 segments to quickly ramp up the rate during slow-start. Even if Linux employs both quick and delayed ACKs, the ACK train is mostly composed of the closely spaced ACKs sent in a burst due to the initial quick ACKs. Fig. 10 (b) shows that a small number of largely spaced ACKs are found in between two chunks of ACK trains. Our delayed ACK filtering described in Section 4.2 filters out any significantly delayed ACK packets in a train. This allows HyStart to correctly estimate the BDP of the network. HyStart completes the slow start phase only one round before cwnd reaches the BDP. FreeBSD and Windows XP send delayed ACKs from the beginning of a connection and consequently, ACKs are spread over an entire RTT round. Even in this scenario, our filtering scheme works fairly well. Fig. 10 (c) shows that HyStart finishes slow start only one round before cwnd reaches the BDP.

5.4. Integration with high-speed protocols

In this section, we show the effectiveness of HyStart on high-speed TCP variants. Most high-speed variants use their own congestion avoidance algorithms while keeping the existing slow-start. As the algorithm of HyStart requires only an inter-arrival time of ACKs and RTT samples, we can easily integrate it into any protocols. We implemented the HyStart as an exported function within the Linux kernel so that it can be called from any protocol.

Figs. 11 and 12 present the results of two CUBIC [41] flows with and without HyStart, respectively in the same experimental setup as in Fig. 1. We plot the trajectory of cwnd and ssthresh and the throughput measured in the bottleneck router. In the experiment of CUBIC with SS, the first flow shows the initial timeout of 20 s because it overshoots to up to 20,000 packets, which is twice the BDP of the network. When the second flow joins the network, the path is already fully utilized. But the second flow perturbs the link utilization with its exponential probing

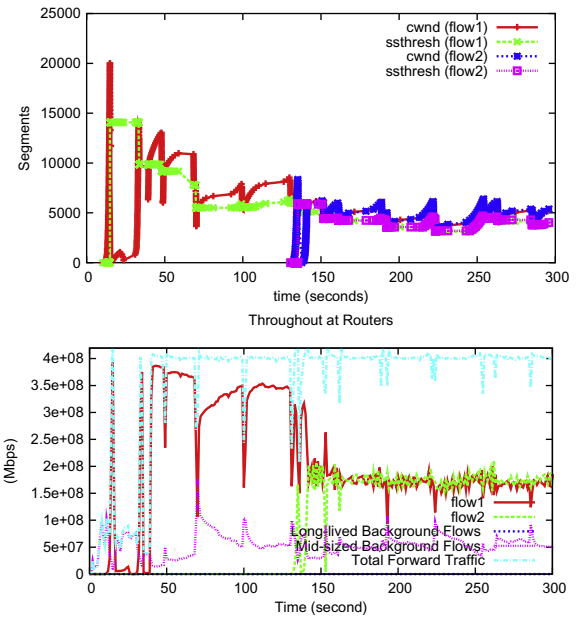


Fig. 11. CUBIC with standard slow start. The first flow experiences heavy packet losses around the first 10 s and multiple timeouts over a 20 s period.

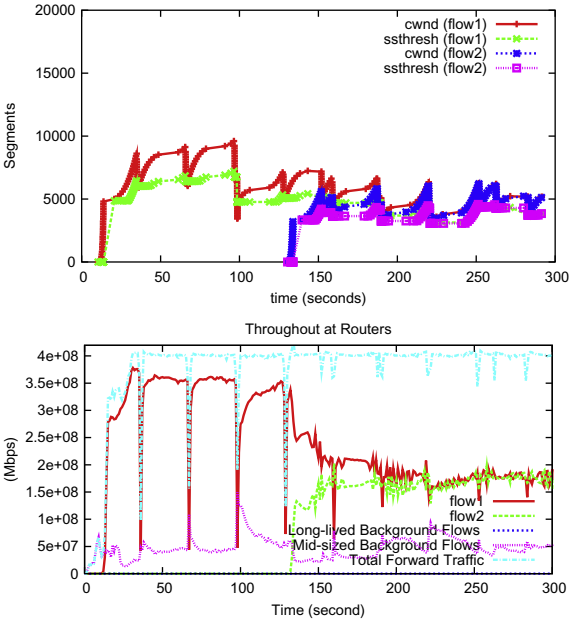


Fig. 12. CUBIC with HyStart. HyStart exits from slow start before packet losses occur.

and this leads to synchronized packet losses for both CUBIC flows and background traffic.

With HyStart, however, two CUBIC flows do not incur packet losses. The first flow detects an exit point slightly before full link utilization is reached. The second flow, which joins at the 130th second, detects the congestion of the path using the delay increase and exits to the congestion avoidance phase of CUBIC.

5.5. Testing with other OS receivers

In this experiment, we evaluate the performance of three representative slow-start algorithms, HyStart, SS and LSS, by varying the receiver side operating systems. We run two TCP flows. The sender machines are fixed to Linux 2.6.23.9. We set the bottleneck bandwidth to 400 Mbps and RTT to 100 ms. We add about 50 Mbps background traffic. Note that with FreeBSD and Windows XP receivers, the Linux sender behaves like TCP-NewReno as explained in Section 2.2. Fig. 13 shows that HyStart works much better than SS, regardless of the operating systems of the receivers. LSS works relatively as well as HyStart in this setup. Lower performance of SS under Windows XP and FreeBSD is observed because these operating systems force the sender to behave like TCP-NewReno and they are unable to handle high packet losses caused by the overshooting of cwnd in SS. However, HyStart finishes slow start before this overshooting happens. Thus, even if the sender behaves like TCP-NewReno, since there are no heavy packet losses, it shows a reasonably good performance.

5.6. Impact of RED on start-up performance

Unlike the Internet routers in the backbone, access links in the last mile show considerable deployment of advanced Active Queue Management (AQM) algorithms such as random early detection (RED) [42] and over-provision of buffers for absorbing temporal traffic bursts and for traffic shaping [43]. In this experiment, we show the effect of RED on HyStart. Since typical residential access networks provide less than 10 Mbps bandwidth links, we set the bandwidth to 10 Mbps for this experiment. We integrate HyStart on TCP-SACK and measure their start-up performance with the router running Drop-tail and RED. Fig. 14 shows the results. For short to medium file sizes (10 Kbytes to 1 Mbyte), HyStart shows shorter file completion times than SS with Drop-tail, but they have no difference with RED. This is because when RED randomly drops packets during a slow start period, a TCP-SACK flow will immediately run its congestion avoidance algorithm (TCP-SACK), which is not modified by HyStart. For large file sizes (10 and 100 Mbytes), HyStart significantly reduces file completion times by more than one second when compared to SS with Drop-tail. By contrast, RED makes SS close to the completion of HyStart by dropping packets before it

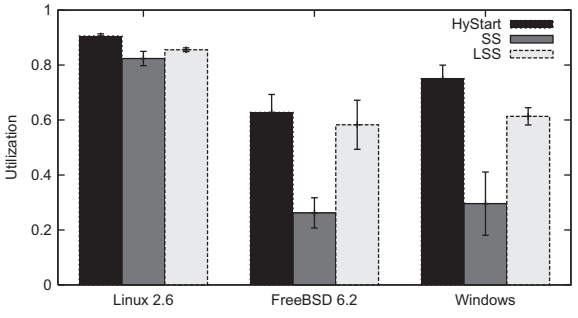
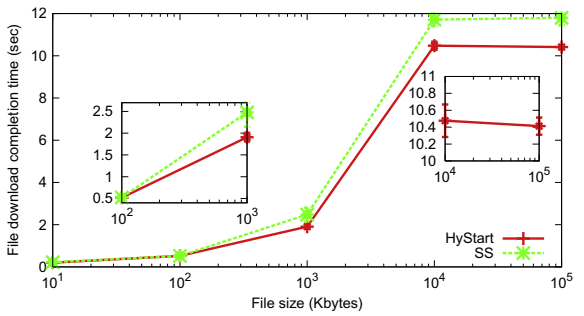
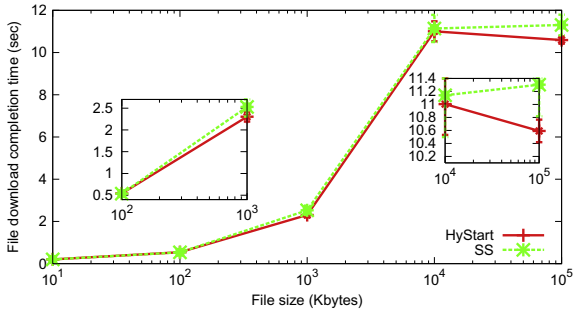


Fig. 13. Two CUBIC flows with three different slow-start algorithms (HyStart, SS and LSS), by changing the OS of receivers.



(a) Drop-tail router



(b) RED router

Fig. 14. Impact of RED on HyStart. The bandwidth and RTT are set to 10 Mbps and 120 ms respectively. The buffer size for the Drop-tail router is set to 100 packets. The file sizes we tested are from 10 Kbytes to 10 Mbytes. The recommended parameters for RED in [44] are used ($W_q = 0.002, max_p = 0.1, min\text{th} = 20, max\text{th} = 80$), so that packets are linearly dropped with probability when the number of queued packets in the router are between 20 and 80 packets. (For interpretation of the references to colour in this figure legend, the reader is referred to the web version of this article.)

overshoots the capacity of the link, which is what HyStart achieves during slow start.

5.7. Testing for short to medium transfers

The slow start algorithm plays an important role in making short transfers finish quickly. In this experiment, we compare the file completion time between SS and HyStart. To prevent CPU overload and measure the performance of the slow start algorithm itself, we only vary the bandwidth from 10 Mbps to 100 Mbps while fixing RTT to 120 ms. The file sizes we tested are between 10 Kbytes and 10 Mbytes. We use TCP-SACK for both the sender and the receiver. Table 3 shows the file completion time when we run SS and HyStart. For short-transfer file sizes (10 Kbytes and 100 Kbytes), SS and HyStart spend the same amount of time for downloading, regardless of the bandwidth. For mid-transfer file sizes (1 Mbyte and 10 Mbytes), HyStart shows a shorter completion time as the bandwidth increases except for the case transferring a 10 Mbyte file in a 10 Mbps bandwidth network.

5.8. Impact on convergence

Since HyStart makes flows exit slow start earlier than with the standard slow start, it can slow down the convergence rate for fairness. In this experiment, we compare the convergence speed of two TCP-SACK flows with HyStart and SS respectively. We run two flows at the 10th and 40th seconds respectively and measure the time until the instantaneous throughput of two flows are within 80%. The duration of each run is 600 s. We fix the bandwidth to 100 Mbps and vary RTT from 60 ms to 240 ms. The buffer size is set to 100% BDP. To see the worst case scenario, we do not introduce any background traffic. Fig. 15 shows the results. SS shows a slightly better convergence time than HyStart for 60 ms RTT while HyStart shows a better convergence time for 120 ms RTT. For 240 ms RTT, both algorithms show very poor convergence. While we expect the poor convergence speed of flows using HyStart, HyStart delivers the convergence speed similar to or better than SS. The results show that congestion avoidance algorithms play an important role in convergence. Also, we observe that asynchronous packet drops as a result of bandwidth competition among flows help their convergence.

Table 3
Comparison between HyStart and standard slow start (SS) for short to medium file size transfers. We ran the experiment 25 times and present the average file completion time and its standard deviation. For short transfer sizes such as 10 Kbytes and 100 Kbytes, HyStart and SS show the same completion time, regardless of the bandwidth. Even though HyStart shows a slightly longer completion time for transferring 10 Mbyte file in a 10 Mbps bandwidth path, it shows faster download as the bandwidth increases.

Algorithm	Bandwidth			File size (bytes)
	10 Mbps	50 Mbps	100 Mbps	
HyStart	0.4 (0.0)	0.4 (0.0)	0.4 (0.0)	10 K
	0.8 (0.0)	0.8 (0.0)	0.8 (0.0)	100 K
	1.85 (0.25)	1.5 (0.01)	1.5 (0.0)	1 M
	9.92 (1.03)	3.51 (0.17)	2.6 (0.0)	10 M
SS	0.4 (0.0)	0.4 (0.0)	0.4 (0.0)	10 K
	0.8 (0.0)	0.8 (0.0)	0.8 (0.0)	100 K
	2.04 (0.06)	1.5 (0.0)	1.5 (0.0)	1 M
	9.62 (0.04)	3.81 (0.59)	3.01 (0.88)	10 M

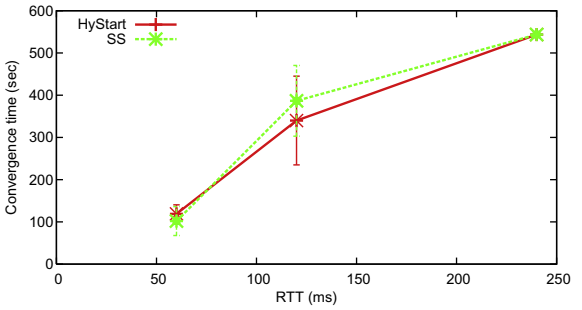


Fig. 15. Convergence speed between HyStart and SS. Two TCP-SACK flows start at different times in a 100 Mbps bandwidth network. Each experiment lasts for 600 s and 60, 120 and 240 ms RTTs are tested. We record the elapsed time when the throughput of the second flow reaches 80% of the throughput of the first flow. HyStart has the convergence speed similar to or better than that of SS.

5.9. Fairness to standard TCP flows

In this experiment, we check whether the performance of standard TCP flows suffer from TCP flows running HyStart. This is particularly important for incremental deployment of HyStart in the Internet. We first measure the throughput share between two TCP-SACK flows using standard SS. Then we replace the second flow's slow start algorithm with HyStart and measure the throughput share between the two flows. We set the bandwidth to 100 Mbps and vary RTT from 60 and 120 ms RTT. Fig. 16 shows the results. The throughput share of the second TCP flow running HyStart grabs less bandwidth from the first TCP flow running SS. This indicates that standard TCP flows running SS do not suffer throughput degradation from TCP flows running HyStart.

5.10. Testing over asymmetric links

Recent Internet measurement [34] reports that when there is an asymmetry in delays in the Internet, more than 80% of the paths show less than a 20 ms delay difference between forward and reverse paths. The performance of HyStart may be affected by such asymmetry in delays and bandwidth. In this experiment, we test all slow-start proposals under various asymmetric network environments by changing the bandwidth and delays in forward and reverse directions. We use two TCP-SACK flows starting at the 10th and 40th seconds, respectively and measure the utilization until the 70th second. For bandwidth asymmetry testing, we fix RTT to 120 ms, and vary the bandwidth ratio between forward and reverse directions from 1/4 to 4 by using the bandwidth from 100 Mbps to 400 Mbps. We also introduce background traffic in both forward and reverse directions of the bottleneck. The amount of background traffic is around 15% of the minimum bandwidth between the two. For delay asymmetry testing, we fix the bandwidth in both forward and reverse directions to 400 Mbps, and vary the ratio between the forward delay and reverse delay from 1/3 to 3, and set the sum of the delays in both directions to 120 ms. We introduce background traffic comparable to 50 Mbps in both forward and reverse directions. Fig. 17 (a) and (b) show the results respectively. HyStart achieves a good start-up throughput even in the networks with high asymmetry in bandwidth and delay.

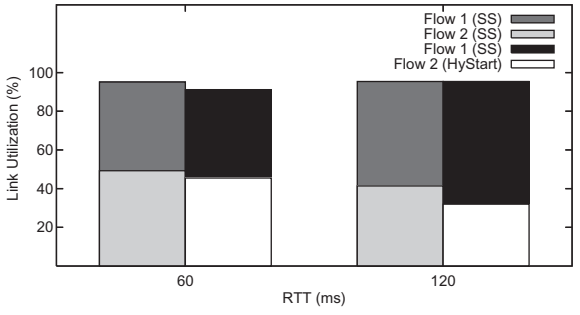


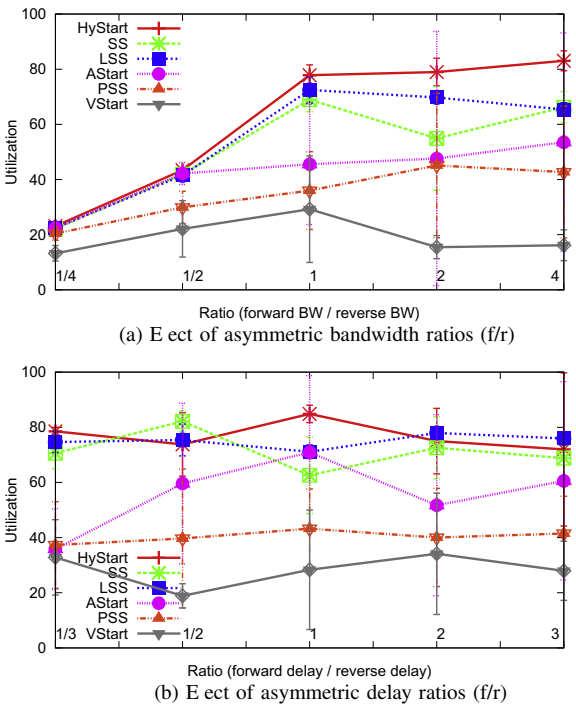
Fig. 16. Fairness of HyStart on standard TCP flows. We first measure the throughput share between the two TCP flows running SS. We then replace the second flow's slow start algorithm with HyStart and measure the throughput share. We report the average of more than 25 experiments. The bandwidth is fixed at 100 Mbps and 60 and 120 ms RTTs are tested. The TCP flow running HyStart grabs less bandwidth compared to the TCP flow running SS.

5.11. More diverse experimental settings

In this experiment, we compare the performance between all slow-start proposals discussed in Section 5.2 and HyStart under more diverse experimental settings. To measure the start-up throughput, we use two TCP-SACK flows starting at the 10th and 40th seconds respectively and the utilization is measured between the 10th and 70th second. We vary the bandwidth from 10 Mbps to 400 Mbps, RTT from 10 ms to 160 ms, and the buffer sizes from 10% to 200% BDP. For RTT and bandwidth experiments, we fix the buffer size to 100% BDP of a flow. For buffer size experiments, we fix the bandwidth to 400 Mbps and RTT to 240 ms. Fig. 18 shows the performance results.

HyStart exhibits consistently good network utilization independent of network bandwidth, buffer space and RTTs except in the case of very small buffer spaces (where no protocols work well). Note that recent research [28] suggests that buffers in the backbone routers can be significant smaller. With very smaller buffer sizes (e.g., 0.2 BDP), the transition from SS to HyStart can less compelling since HyStart shows no additional benefits. However, especially under large BDP networks, HyStart outperforms the other schemes by 3–5 times. The other schemes suffer high performance losses as the BDP increases.

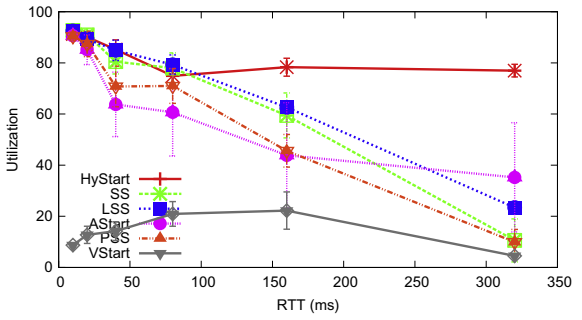
Fig. 19 measures the CPU utilization of the sender when running various slow start protocols in Fig. 18 (a). The



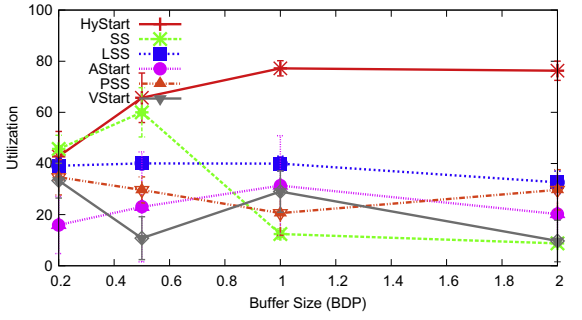
(a) E ect of asymmetric bandwidth ratios (f/r)

(b) E ect of asymmetric delay ratios (f/r)

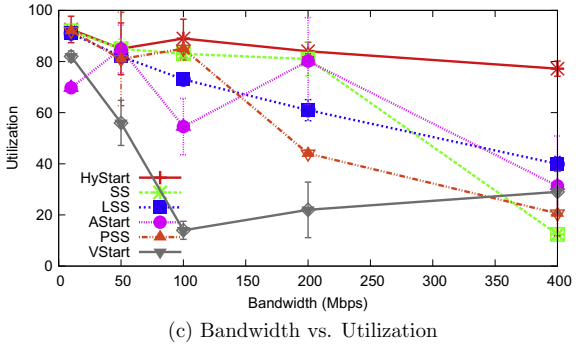
Fig. 17. Two TCP-SACK flows with different slow-start algorithms, under asymmetric delays (a) and bandwidth (b).



(a) RTT vs. Utilization



(b) Buffer size vs. Utilization



(c) Bandwidth vs. Utilization

Fig. 18. Two TCP-SACK flows with different slow-start algorithms by varying their RTTs (a), buffer sizes (b), and bandwidth (c).

results from other runs are similar. We find that the CPU utilization under SS and AStart is extremely high under medium and high BDP networks. LSS also shows relatively high CPU utilization. HyStart, PSS and VStart show very low CPU utilization because they terminate slow start before packet losses occur. While PSS and VStart do so prematurely causing low network utilization, HyStart maintains good network utilization.

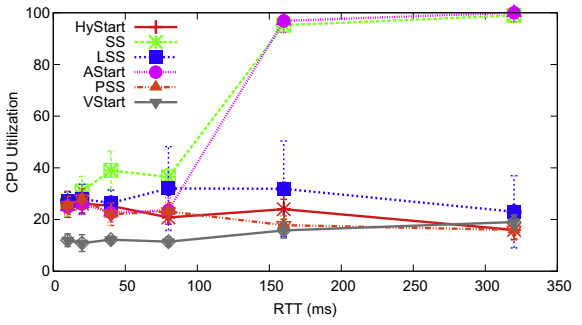


Fig. 19. CPU utilization of the Linux sender with various slow start schemes for the run in Fig. 18(a).

6. Internet experiment

We show the results of all the slow start proposals with TCP-SACK in three high-speed production networks such as Internet2 [45], National LambdaRail (NLR) [46] and GEANT [47]. We also show the performance of HyStart and SS in typical access networks such as cable networks and in wireless networks.

6.1. Internet2 experimental setups

Fig. 20 shows the Internet 2 testbed. Internet2 and NLR paths from North Carolina to Chicago (25 ms RTT) and from Chicago to Japan (225 ms RTT) have a 1 Gbps connection, and the GEANT path from North Carolina to Germany (107 ms RTT) has a 100 Mbps connection. GEANT testing involves servers inside the campus networks, so some of the traffic load is expected. We run two TCP flows with different slow-start algorithms over the paths to Chicago, Germany and Japan from North Carolina. The first flow starts at the 10th second and the second flow starts at the 30th second and the utilization is measured until the 50th second. We run the experiment in three different periods of each day (5 AM to 9 AM, 1 PM to 5 PM, and 9 PM to 1 AM all EDT). Linux is installed on all of the machines in the testbed.

6.2. Internet2 results

Fig. 21 shows the results of various slow-start algorithms with a TCP-SACK sender and receiver. SS, LSS and HyStart achieve good network utilization in relatively small BDP networks (both Chicago and Germany paths). HyStart shows exceptionally good throughput for the high-BDP path between North Carolina and Japan, which has 250 ms RTT and 1 Gbps link speed while LSS shows the worst performance due to its sluggish increase of cwnd in the same link. PSS typically underestimates the bandwidth so that its performance is not predictable. VStart also terminates the slow-start period prematurely and shows low utilization even in small BDP networks. In all tests, HyStart delivers consistent start-up throughput.

To further encourage the use of HyStart, we run a TCP-NewReno sender at the North Carolina site over the

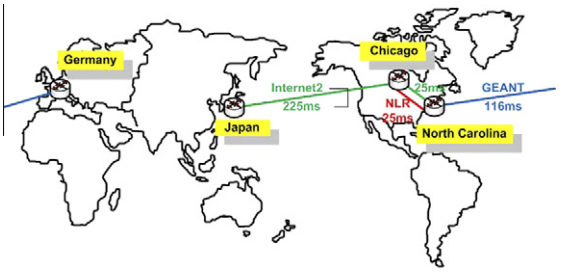


Fig. 20. Research testbed (Internet2, NLR and GEANT).

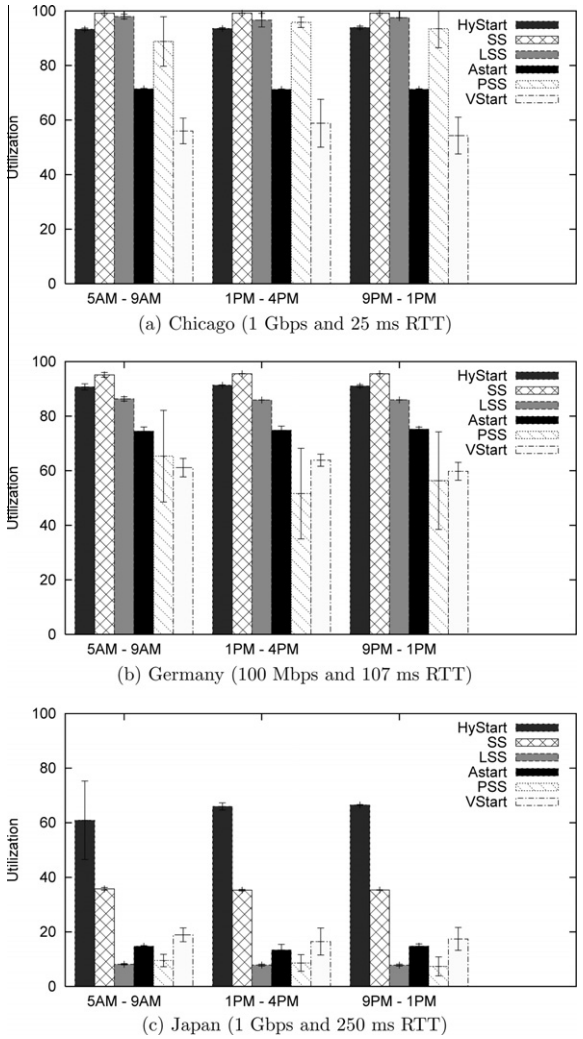


Fig. 21. The network utilization of two TCP-SACK flows with different slow-start algorithms over the three high-speed Internet paths.

Germany path. Since we cannot modify the receiver site servers, we change the sender side only. This set-up emulates the behavior when using a Windows XP receiver which forces the sender to behave like TCP-NewReno in medium and large size BDP networks. Fig. 22 shows the result. Using SS results in extremely low utilization because of slow recovery after heavy packet losses. However, LSS and HyStart deliver outstanding utilization because HyStart exits slow start before packet losses occur and LSS reduces the cwnd growth rates during slow start thus, prevents heavy packet losses. This result indicates that a protocol like HyStart or LSS must be used to achieve an improved performance in medium size BDP networks when the receiver is Windows XP or end-systems use TCP-NewReno.

6.3. Cable networks

Cable networks typically provide asymmetric bandwidths between their downlink bandwidth and upload

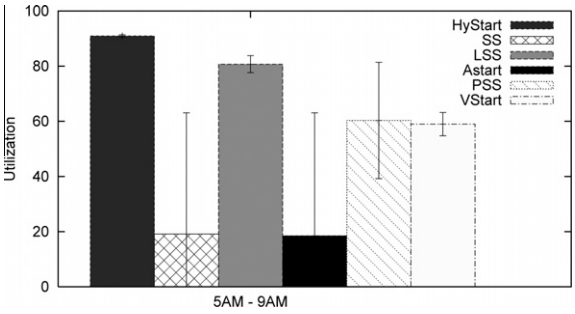


Fig. 22. The network utilization with various slow start protocols when the sender is TCP-NewReno. The run is over a path between Germany and North Carolina).

bandwidth. While they allow more than 40 Mbps download bandwidth, the bandwidth for individual users is much smaller, e.g., from 128 Kbit to 10 Mbps, since the bandwidth is determined by the details of the user plan, which is usually charged based on maximum download capacity of each user. From the most common cable modem standard, DOCSIS [48], a cable modem termination system (CMTS) merge multiple streams from cable modem (CM) into a single stream which results in heavy fluctuations in latency when other users in the same cable network compete for uplink bandwidth [43]. Moreover, cable networks deploy traffic shaping to restrict users from consuming more than their limit, which is not typical in the Internet.

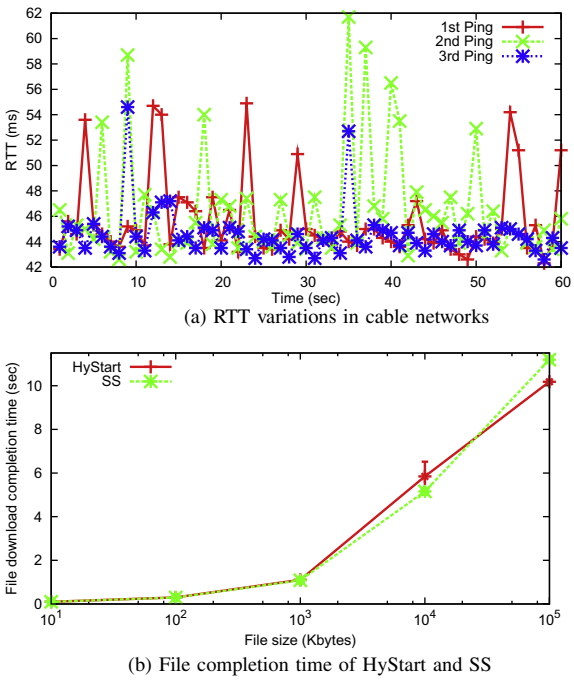
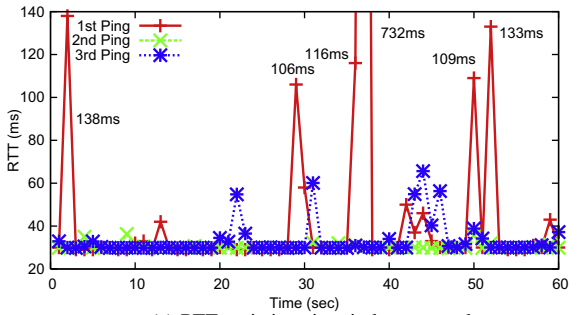


Fig. 23. The file completion times of HyStart and SS in cable networks. The maximum download capacity and upload capacity are 10 Mbps and 384 Kbps respectively. HyStart shows same or shorter file completion times than SS except for the file size of 10 Mbytes where it takes a slightly longer time for completing the file download due to high fluctuations of RTTs.

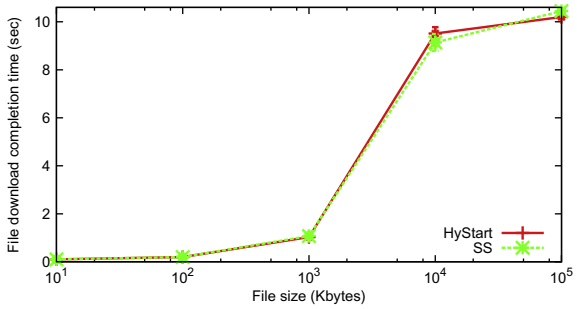
Since HyStart relies on packet-train and delay measurements, it can be detrimental to asymmetric bandwidths and heavy fluctuations of latency in cable networks. As a safety check, we examine how HyStart operates in real cable networks. To quantify latency variations, we measure instantaneous RTT by running ping from the client in the cable networks to the server in the Internet. We measured the RTTs over 60 s and repeated three times in a row. Fig. 23 (a) shows RTT variations measured at the time of this experiment. As reported by [43], we can see heavy fluctuations in RTT measurements, caused by other users' traffic sharing the same network. Fig. 23 (b) shows the download completion times of a single TCP-SACK flow running HyStart and SS respectively. We test file sizes from 10 Kbytes to 100 Mbytes. For short-to-medium file sizes (10 Kbytes to 1 Mbyte), HyStart and SS show no difference. For the file size of 10 Mbytes (10^4 Kbytes), HyStart shows slightly longer completion times due to high fluctuations of RTTs in cable networks. While this is not desirable, the damage on HyStart caused by high variations of RTTs is not significant and it shows shorter completion times as the file size increases. More sophisticated filters would improve the accuracy of the HyStart algorithm for this environment and this is left as our future work.

6.4. Wireless networks

As wireless networks become ubiquitous, many users connect to the Internet through heterogenous wireless technologies such as WiFi, femto-cell, 3G and LTE. Among these technologies, we evaluate HyStart in WiFi networks by downloading a set of files by using a single TCP-SACK flow. The client in the campus network connects to the Internet through a wireless access point (AP) and the server is located in the Internet. The RTT between the client and the server is around 30 ms. Other users in the campus also use the same wireless AP for their Internet access, and thus introduce wireless interference to the client.



(a) RTT variations in wireless networks



(b) File completion time of HyStart and SS

Fig. 24. The file completion times of HyStart and SS in wireless networks. HyStart shows comparable performance to SS while it shows a slightly longer completion times for the 10 Mbytes file size. However, the file download time of HyStart becomes shorter as the file size increases.

Fig. 24 (a) shows instantaneous RTTs observed by the client at the time of experiment. We can see that the RTT measurements during the first ping show several high peaks, due to the interference generated by other users in the same wireless network. We test file sizes from 10 Kbytes to 100 Mbytes by running HyStart and SS as a slow start algorithm for the flow. We plot the average and the confidence interval by accumulating more than 100 runs for each experiment. Fig. 24 (b) shows the results. Similar to the experiments in cable networks shown in Fig. 23, HyStart shows slightly longer completion times for the 10 Mbyte file size. This is due to highly varying RTTs, but the damage is not significant.

7. Conclusion

In this paper, we investigate the causes of long blackouts after slow-start by evaluating the current TCP stack implementations in Linux, FreeBSD and Windows XP. We realize that the overshooting of slow-start causes system bottlenecks and/or extreme slow loss recovery during fast recovery, thus resulting in long blackouts with no transmission. This problem often occurs with TCP-SACK and TCP-NewReno, which are the most popular versions of TCP used in the Internet. Especially with TCP-NewReno, the problem happens even in medium-size BDP networks (around 100–1000 ranges). Our new slow start protocol, HyStart, fixes this problem by detecting safe exit points of slow start that do not lead to heavy packet losses or low network utilization, preventing heavy system overload or low performance during the start-up of TCP. HyStart uses the concept of packet trains and RTT delay increases to find safe exit points. To the best of our knowledge, our work is the first that shows a practical implementation of packet-train-based estimation of available bandwidth for TCP. The performance of Windows XP with standard slow start is extremely poor as it uses SACK suppression and forces the sender to behave like TCP-NewReno. The utility of our work is maximized when the receiver-side end systems are Windows, as HyStart can greatly outperform standard slow start in this setup (even in medium-size BDP networks). This is very likely since a large number of Internet servers are Linux and many end-system users are Windows users. The performance of HyStart under large BDP networks is unsurpassed by any existing slow start protocols independent of the receiver operating systems.

Acknowledgments

This work is financially supported in part by a generous gift from Cisco Systems. We thank the anonymous reviewers for their detailed feedback and help in improving the paper.

References

[1] S. Ha, I. Rhee, Hybrid slow start for high-bandwidth and long-distance networks, in: Proceedings of the Sixth PFLDNet Workshop, Manchester, UK, 2008.
[2] M. Mathis, J. Mahdavi, S. Floyd, A. Romanow, TCP Selective Acknowledgment Options, RFC 2018, RFC 2018 (Proposed Standard).
[3] S. Floyd, J. Mahdavi, M. Mathis, M. Podolsky, An Extension to the Selective Acknowledgement (SACK) Option for TCP, RFC 2883.
[4] E. Blanton, M. Allman, K. Fall, L. Wang, A Conservative Selective Acknowledgment (SACK)-based Loss Recovery Algorithm for TCP, RFC 3517 (Proposed Standard).
[5] A. Medina, M. Allman, S. Floyd, Measuring the evolution of transport protocols in the internet, SIGCOMM Comput. Commun. Rev. 35 (2) (2005) 37–52.
[6] B. Even, D. Leigh, An Experimental Investigation of TCP Peformance in High Bandwidth-Delay Product Paths, MSc Thesis, National University of Ireland Maynooth.
[7] N. Hu, P. Steenkiste, Improving TCP performance using active measurements: Algorithm and evaluation, in: Proceedings of the 11th International Conference on Network Protocols (ICNP'03), Atlanta, GA, 2003.
[8] J.C. Hoe, Improving the start-up behavior of a congestion control sheme for TCP, in: ACM SIGCOMM, 1996.
[9] L. Brakmo, L. Peterson, TCP vegas: end to end congestion avoidance on a global internet, IEEE J. Selected Areas Commun (1995).
[10] S. Floyd, Limited Slow-Start for TCP with Large Congestion Windows, RFC 3742 (Experimental).
[11] R. Wang, G. Pau, K. Yamada, M. Sanadidi, M. Gerla, TCP Startup Performance in Large Bandiwdth Delay Networks, in: Proceedings of IEEE INFOCOM, Hong Kong, 2004.
[12] Oprofile, <http://oprofile.sourceforge.net/news/>.
[13] P. McManus, Performance tradeoffs of TCP selective acknowledgment – does the SACK optimization also present a denial-of-service opportunity?, IBM developersWorks.
[14] S. Floyd, T. Henderson, A. Gurtov, The NewReno Modification to TCP+s Fast Recovery Algorithm, RFC 3782 (Proposed Standard).
[15] T. Kelly, Scalable TCP: improving performance in highspeed wide area networks, ACM SIGCOMM Comput. Commun. Rev. 33 (2) (2003) 83–91.
[16] Y. Li, Experience with loss-based congestion controlled tcp stacks, in: First International GRID Networking Workshop (GNEW 2004), CERN, Geneva, Switzerland, 2004.
[17] L. Jrvinen, M. Kojo, Improving processing performance of linux tcp sack implementation, in: Proceedings of the Seventh PFLDNet Workshop, Tokyo, Japan, 2009.
[18] C. Casetti, M. Gerla, S. Mascolo, M.Y. Sanadidi, R. Wang, TCP Westwood: bandwidth estimation for enhanced transport over wireless links, in: Proceedings of ACM Mobicom, Rome, Italy, 2001.
[19] C. Dovrolis, P. Ramanathan, D. Moore, Packet-dispersion techniques and a capacity-estimation methodology, IEEE/ACM Trans. Networking 12 (6) (2004) 963–977.
[20] S. Floyd, M. Allman, A. Jain, P. Sarolahti, Quick-Start for TCP and IP, RFC 4782 (Experimental).
[21] M. Scharf, S. Hauger, J. K"ogel, Quick-start TCP: from theory to practice, in: Proceedings of the Third PFLDNet Workshop, UK, 2008.
[22] S. Mascolo, C. Casetti, M. Gerla, M.Y. Sanadidi, R. Wang, TCP westwood: bandwidth estimation for enhanced transport over wireless links, in: Mobile Computing and Networking, 2001, pp. 287–297.
[23] L. Konda, J. Kaur, RAPID: shrinking the congestion-control timescale, in: Proceedings of IEEE INFOCOM, Rio de Janeiro, Brazil, 2009.
[24] D. Cavendish, K. Kumazoe, M. Tsuru, Y. Oie, M. Gerla, CapStart: An adaptive tcp slow start for high speed networks, in: Proceedings of the Seventh PFLDNet Workshop, Tokyo, Japan, 2009.
[25] V. Padmanabhan, R. Katz, TCP Fast Start: A Technique for Speeding Up Web Transfers (1998).
[26] H. Wang, H. Xin, D. Kang, G. Shin, A Simple Refinement of Slow Start of TCP Congestion Control (2000).
[27] C. Villamizar, C. Song, High performance tcp in ansnet, ACM Comput. Commun. Rev. 24 (5) (1994) 45–60.
[28] G. Appenzeller, I. Keslassy, N. McKeown, sizing router buffers, in: Proceedings of ACM SIGCOMM, Portland, OR, 2004.
[29] HyStart implementation, <http://lxr.linux.no/linux+v2.6.29/net/ipv4/tcp_cubic.c>.
[30] R.L. Carter, M.E. Crovella, Measuring bottleneck link speed in packet-switched networks, Perform. Eval. 27-28 (1996) 297–318. doi:http://dx.doi.org/10.1016/0166-5316(96)00036-3.
[31] M. Jain, C. Dovrolis, End-to-end available bandwidth: measurement methodology, dynamics, and relation with tcp throughput, IEEE/ACM Trans. Netw. 11 (4) (2003) 537–549. doi:http://dx.doi.org/10.1109/TNET.2003.815304.
[32] M. Allman, V. Paxson, On estimating end-to-end network path properties, in: SIGCOMM '99: Proceedings of the Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication, ACM, New York, NY, USA, 1999, pp. 263–274. doi:<http://doi.acm.org/10.1145/316188.316230>.
[33] S. Keshav, A control-theoretic approach to flow control, in: Proceedings of the Conference on Communications Architecture & Protocols.
[34] A. Pathak, H. Pucha, Y. Zhang, Y.C. Hu, Z.M. Mao, A measurement study of internet delay asymmetry, in: Proceedings of Passive and Active Measurement Conference (PAM), Cleveland, Ohio, 2008, pp. 37–52.
[35] IPerf, <http://dast.nlanr.net/projects/Iperf/>.
[36] P. Barford, M. Crovella, Generating representative web workloads for network and server performance evaluation, in: Measurement and Modeling of Computer Systems, 1998, pp. 151–160.
[37] J. Aikat, J. Kaur, F. Smith, K. Jeffay, Variability in TCP round-trip times, in: Proceedings of the ACM SIGCOMM Internet Measurement Conference, Miami, FL, 2003.
[38] TCPProbe, <http://lxr.linux.no/linux+v2.6.26.5/net/ipv4/tcp_probe. c>.
[39] L. Stewart, G. Armitage, J. Healy, Characterising the behavior and performance of siftr v1.1.0, Technical Report 070824A.
[40] S. Ha, L. Le, I. Rhee, L. Xu, Impact of background traffic on performance of high-speed TCP variant protocols, Comput. Netw. 51 (7) (2007) 1748–1762.
[41] S. Ha, I. Rhee, L. Xu, CUBIC: a new TCP-friendly high-speed TCP variant, SIGOPS Oper. Syst. Rev. 42 (5) (2008) 64–74. doi:http://doi.acm.org/10.1145/1400097.1400105.
[42] S. Floyd, V. Jacobson, Random early detection gateways for congestion avoidance, IEEE/ACM Trans. Netw. 1 (4) (1993) 397–413.
[43] M. Dischinger, A. Haeberlen, K.P. Gummadi, S. Saroiu, Characterizing residential broadband networks, in: IMC'07, 2007, pp. 43–56.
[44] S. Floyd, R. Gummadi, S. Shenker, Adaptive RED: An Algorithm for Increasing the Robustness of RED's Active Queue Management, <http://www.icir.org/floyd/red.html>.
[45] Internet2, <http://www.internet2.edu/>.
[46] National LamdaRail, <http://www.nlr.net/>.
[47] GEANT, <http://www.geant.net/>.
[48] CableLabs, DOCSIS Specification - DOCSIS 1.1 interface, <http://www.cablelabs.com/cablemodem/specifications/specifications11.html>.

Sangtae Ha is currently with Princeton University, where he is an Associate Research Scholar in the Department of Electrical Engineering, leading the establishment of the Princeton EDGE Lab, as an Associate Director. He received his Ph.D. in Computer Science from North Carolina State University and has been an active contributor to the Linux kernel. His research interests include congestion control, peer-to-peer networking, wireless networks, network economics, and energy-aware information technology.

Injong Rhee (M'89) received his Ph.D. from the University of North Carolina at Chapel Hill. He is a Professor of Computer Science at North Carolina State University. His areas of research interests include computer networks, congestion control, wireless ad hoc networks and sensor networks.