Taming the Elephants: New TCP Slow Start *

Sangtae Ha
Department of Computer Science
North Carolina State University
Raleigh, NC 27695, USA
sha2@ncsu.edu

Injong Rhee
Department of Computer Science
North Carolina State University
Raleigh, NC 27695, USA
rhee@ncsu.edu

Abstract

Standard slow start does not work well under large bandwidth-delay product (BDP) networks. We find two causes of this problem in existing three popular operating systems, Linux, FreeBSD and Windows XP. The first cause is that because of the exponential increase of cwnd during standard slow start, heavy packet losses occur. Recovering from heavy packet losses puts extremely high load on end systems which renders the end systems completely unresponsive for a long time, resulting in a long blackout period of no transmission. This problem commonly occurs with the three operating systems. The second cause is that some of proprietary protocol optimizations applied for slow start by these operating systems to relieve the system load happen to slow down the loss recovery followed by slow start. To remedy this problem, we propose a new slow start algorithm, called Hybrid Start (HyStart) that finds a "safe" exit point of slow start at which slow start can finish and safely move to congestion avoidance without causing any heavy packet losses. HyStart uses ACK trains and RTT delay samples to detect whether (1) the forward path is congested or (2) the current size of congestion window has reached the available capacity of the forward path. HyStart is a plug-in to the TCP sender and does not require any change in TCP receivers. We implemented HyStart for TCP-NewReno and TCP-SACK in Linux and compare its performance with five different slow start schemes with the TCP receivers of the three different operating systems in the Internet and also in the lab testbeds. Our results indicate that HyStart works consistently well under diverse network environments including asymmetric links and high and low BDP networks. Especially with different operating system receivers (Windows XP and FreeBSD), HyStart improves the start-up throughput of TCP more than 2 to 3 times.

1. INTRODUCTION

Long distance networks spanning several continents are gaining importance. Many multi-national companies are now centralizing their data centers due to economical reasons. Thus, the high performance of TCP over these networks is critical for the effective operation of these data centers. These networks typically have large bandwidth and delay products (BDP) ranging from several 1,000's to 100,000's. This is not a typical environment where most existing commercial TCP stacks are optimized for. Often we find that when these stacks run in these environments, they fall in some "rare" states where TCP gets extremely low performance. One of 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 much more optimizations. One of these functions are slow start.

Standard TCP doubles congestion window (cwnd) in every round-trip time (RTT) during slow start. However, the exponential growth of cwnd results in burst packet losses. Since the cwnd overshoots beyond the path capacity as large as the whole BDP, in large BDP networks, this overshoot causes strong perturbation in the networks. This may cause high loss synchronization among many competing flows, resulting in low utilization of networks. Furthermore, long bursts of packet losses caused by the overshoot also add a lot of burden on the end systems for loss recovery and this burden often translates into consecutive timeouts and a long blackout period of no transmission.

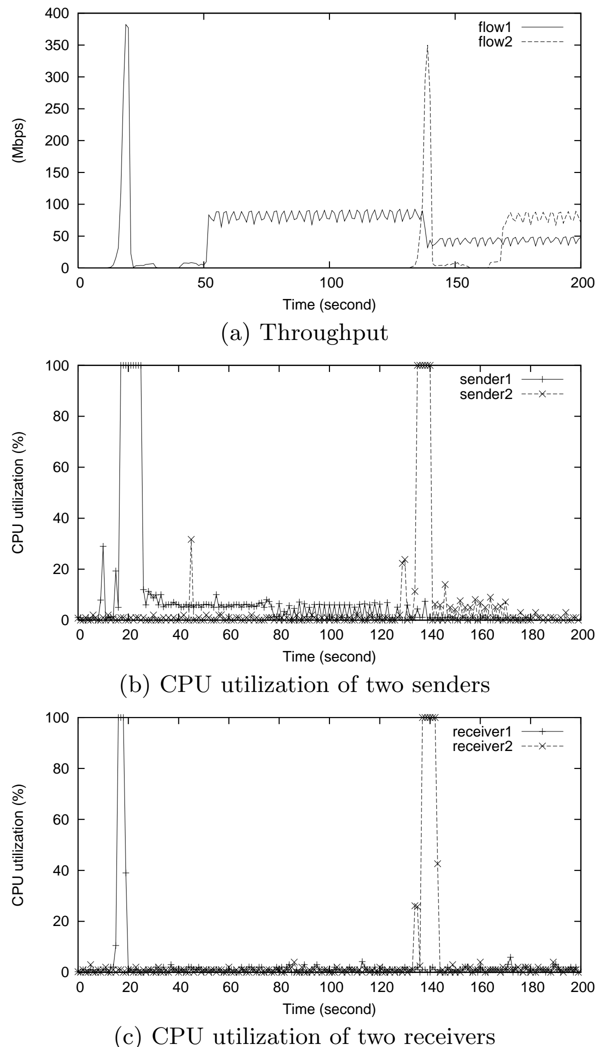
The selective acknowledgement (SACK) option [28, 19, 10] relives some of these burdens. As SACK informs to the sender the blocks of packets successfully received, the sender can be more intelligent about recovering from multiple packet losses. SACK is currently enabled in most commercial TCP stacks [29]. However, for a large BDP network where the number of packets are in flight, the processing overhead of SACK information at the end points can be quite overwhelming 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. During fast recovery, every packet reception generates a new SACK packet. Given that the size of cwnd can be quite large (sometimes, beyond 100,000), the overhead of search can be sometimes overwhelming. This system overload is quite 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 the 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 almost 100% CPU utilization or a reduced number of SACKs extremely slows down loss recovery result-

*An earlier short version [21] of this paper was presented at the International Workshop on Protocols for Fast and Long Distance Networks in 2008.

ing in a blackout period of no transmission over 100 seconds. The problems consistently occur in all three dominant operating systems, Linux, Windows XP and FreeBSD during more than 40% of 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 so that even under many occurrences of loss bursts, the system does not get 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 also slow starts are not necessarily the only causes of long bursts, the first approach is important. The second approach is also important since aggressive slow starts burden networks as well as end-systems. This paper primarily focuses on the second approach.

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 own pitfalls. We then present a new slow start algorithm, called Hybrid Slow Start (HyStart) [21] that effectively prevents the overshooting of slow start while maintaining the 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 move 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 in 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 both in 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.



(a) Throughput

(b) CPU utilization of two senders

(c) CPU utilization of two receivers

Figure 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.

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 the developers of various operating systems that inadvertently cause extremely slow packet loss recovery after multiple packet losses during slow start.

2.1 Processing Overload during Slow Start

We investigate how the system overhead during slow start can affect TCP performance. System overload is frequently observed during slow start which are followed by multiple timeouts or long blackouts of no transmission. The problem occurs consistently with all three popular operating systems, Linux, Windows XP and FreeBSD. Figure 1 (a), (b) and (c) illustrates the throughput observed at a router when two TCP-SACK flows join at different times in a network of 400 Mbps bandwidth and 240 ms RTT. We also measure the CPU utilization of the Linux senders and receivers. We use the latest version of Linux, Linux 2.6.26-rc4 for this test. What is shown is typical and repeatable phenomena observed about 30 to 40% of times we run the experiment. (The results of repeated experiments are given in Section 5.) The experimental setup of our testbed is detailed in Section 5.1. From the figure, both flows have almost zero throughput for over 30 seconds after a timeout. These blackouts follow after the saturation of the CPU utilization at the senders and receivers. When the system reaches the overload, it cannot react fast enough to perform loss recovery, which 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 The processing overhead at TCP senders

We first examine the SACK processing overhead of the TCP sender in Linux. Figure 2 illustrates how the Linux TCP

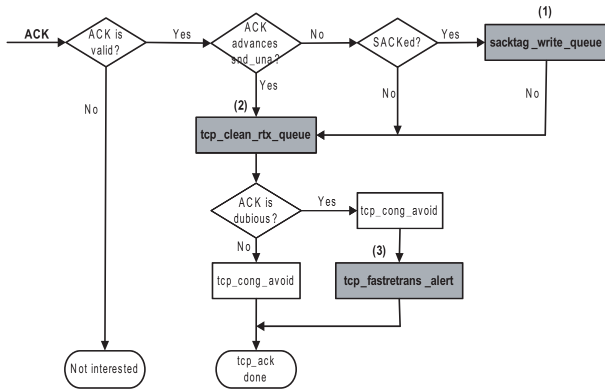


Figure 2: Linux TCP ACK processing

sender processes an ACK packet. $tcp_ack()$ called at the reception of ACK, check whether the ACK number is within the left and right edges of 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 controls ensures that $packets_in_flight$ always matches the size of the congestion window. After that, $tcp_clean_rtx_queue()$ is called to remove and free the acknowledged packets from the retransmit queue, and $packets_out$ is decremented by the number of the freed packets. $tcp_fastretrans_alert()$ executes fast retransmit and updates the scoreboard that keeps track of lost and acknowledged packets. For every ACK, $tcp_cong_avoid()$ is called which increases *cwnd* by entering slow start and congestion avoidance phases.

The retransmit queue is implemented using a linked list to hold all the packets currently in flight to the receiver. Each SACKed packet is searched for 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 Figure 2 are 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 and let us examine each of these functions below.

- $sacktag_write_queue()$: 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 within 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 a 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. Then. it needs at most one traversal of the retransmit queue to mark all the packets in the retransmit queue and therefore, requires $O(W)$ running time.

We profile the CPU usage of the three functions by using OProfile [5] in Linux. OProfile collects the number of standard CLK_UNHALTED counters for the functions in the kernel and presents their CPU utilization. We run the same testing shown in Figure 1 for the profiling. Figure 3 (a) shows the results of 100 runs and we plot them in a log scale. $sacktag_write_queue()$ consumes around 10% to 100% 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 $sacktag_write_queue()$.

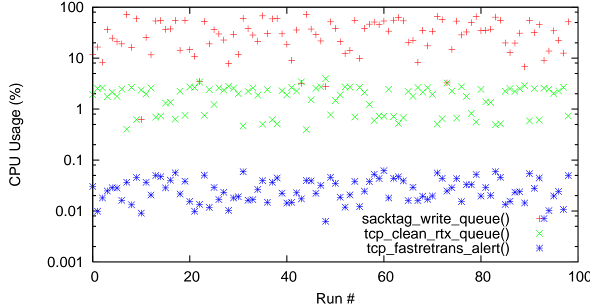
To see the relationship between W (*cwnd*) and CPU utilization, we change the bottleneck buffer size from 1% to 200% of the path BDP while maintaining the same testing parameters. Figure 3 (b) plots the CPU usage of the three functions as W increases in a log scale. The CPU usage of $tcp_clean_rtx_queue()$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

2.1.2 *The processing overhead at TCP receivers*

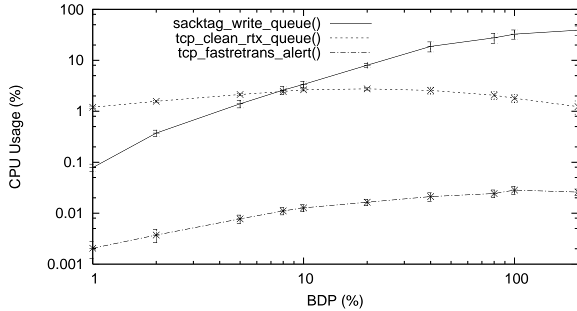
In Linux TCP, $tcp_data_queue()$ processes the data in received packets. If packets are received in order, it copies the data in the packets to the user space. When packets arrives out of order, it puts the packets in the out-of-order queue. The packets in the out-of-order-queue can only be freed when 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 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 affect the performance very much as it typically goes to the end of the queue. However, with the large number of retransmitted packets, the processing overload occurs when each retransmitted packet causes a traversal of the queue to place the packet into the right place.

We profile the receiver kernel and measure the system clocks of $tcp_data_queue()$ of the second receiver shown in Figure 1 (c). Our profile shows that $tcp_data_queue()$ consumes more than 30% of CPU time for the entire run which is a major contributor to the period of 100% CPU utilization around 140-th second in Figure 1 (c). The data copy to the user takes only 2% of CPU time.

2.2 Protocol Misbehavior during Slow Start

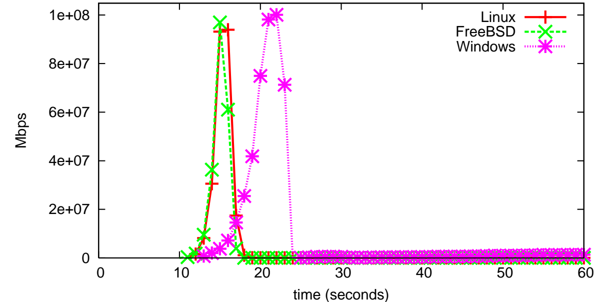


(a) SACK processing overhead

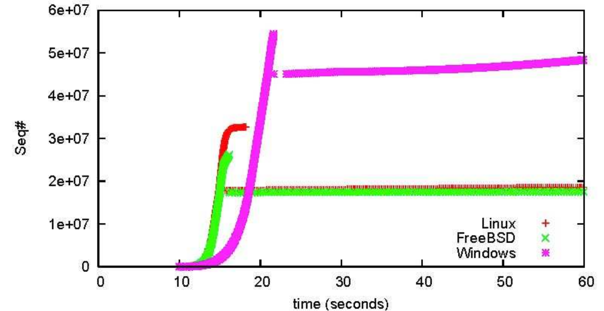


(b) BDP (W) vs. CPU utilization

Figure 3: CPU utilization of the three functions in TCP SACK processing



(a) Throughput



(b) Time vs. sequence numbers

Figure 4: TCP-NewReno on the three systems.

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

Figure 4 shows the results of 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) [18] where the TCP sender resets the timeout timer whenever a partial acknowledgment acknowledging the left edge of the window arrives. This effectively prevents the sender from entering timeouts 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 the lost packets.

This problem of a slow ramp-up after slow start does not occur in TCP-SACK since the sender is more informed about lost packets beyond the left-edge and retransmits all the lost packets almost immediately. Figure 5 shows the performance of TCP-SACK for the three operating systems.* We find that all stacks now enter a 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 some threshold, the receiver simply reports only cumulative ACKs. This is an optimization added by the Windows developers for various purposes including reducing system overloads during fast recovery or limiting buffering at the receiver. This forces the sender to act like TCP-NewReno, showing the similar behavior as in Figure 4. We can also obtained the same performance result even if we use a Linux sender along with a Windows XP receiver.

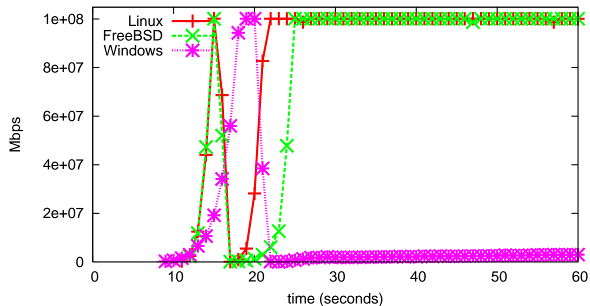
3. EXISTING SOLUTIONS

There are basically two approaches to fix 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 situations where burst losses occur (not necessarily to slow-start only). The second approach is to prevent burst losses by designing a better slow-start protocol. It is important for preventing both network and system overloads. In this section, we discuss existing techniques for the two approaches.

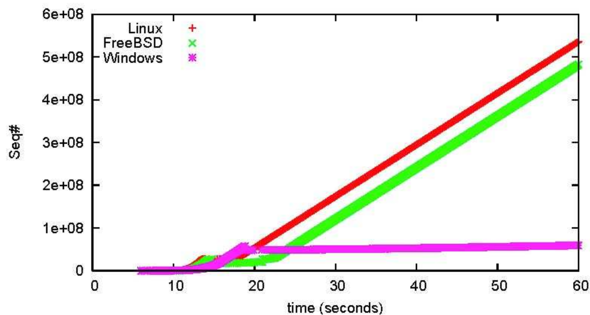
3.1 Linux: Linked-list optimization

Since Linux 2.6.15, SACK processing has undergone several improvements in all cases involving better caching and data structures to reduce the look-up time in the retransmit

*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. This is a bug in FreeBSD.



(a) Throughput



(b) Time vs. sequence numbers

Figure 5: TCP-SACK on the three systems.

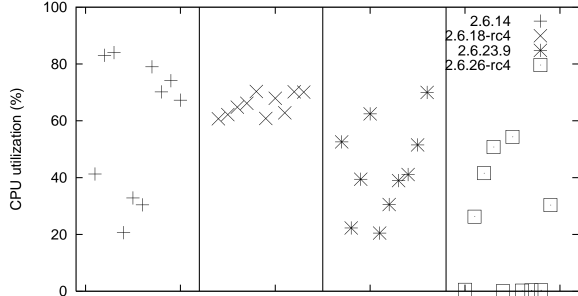


Figure 6: Improvement of SACK processing on Linux after slow-start in a large BDP network. The SACK processing have been improved over the evolution of Linux kernel.

queue. Figure 6 shows the improvement on Linux SACK processing over the evolution of the Linux kernel for the same experiment shown in Figure 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 Figure 1 with different kernel versions and measure the CPU utilization of *sacktag_write_queue*(). We can see the overhead reduces as the the kernel version increases. However, the SACK processing overhead has reduced only about 20% in the latest version. Improving the system bottlenecks by efficiently optimizing data structures is valuable and it still requires further improvement. As we have seen in Figure 1, the latest version of Linux still consumes almost 100% of CPU time during fast recovery after slow start in large BDP networks.

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 [11] and TCP-Westwood [14] and can prevent the overshooting of the 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 very frequent under-estimation because of noise in RTT estimation. Because of the bursty packet transmission 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 often prematurely ends slow start. as the estimation technique is too simple and often makes under-estimation of BDP. For instance, we find that TCP-SACK with this technique shows 15.6% link utilization, in the 100Mbps and 120ms one-way delay networks. We have more performance results of TCP-Vegas in the next section.

3.3 Windows - Suppressing SACK options

When the 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-ordered 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 Figure 4.

3.4 Existing Slow Start Algorithms

Hoe [23] proposes to estimate the bottleneck bandwidth using a packet-pair measurement, and to use the estimated value to set the *ssthresh* of TCP-NewReno. [15], however, indicates 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 over-estimation of the bottleneck link bandwidth.

Vegas [12] introduces a modified slow start mechanism which allows an exponential growth of *cwnd* at every other RTT and, in between, compares its current transmission rate with the expected rate to see whether the path has still some room to increase. The modified slow start of Vegas is known to incur 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 transmission [35].

Limited slow start [16] is an experimental RFC. It is designed to prevent a large number of packet losses in one RTT by limiting the increment of congestion window to *max_ssthresh*/2 per RTT. But using the fixed number of *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 seconds[†] before reaching the congestion window size of 5000 under RTT of 200ms. Quick-Start [17, 32] determines its allowed sending rate very quickly,

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

but it needs cooperation with the routers that approve Quick-Start along the path.

Adaptive start [35] 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. However, adaptive start can be slower than stardard slow start, and it is not easily integrated with other congestion control algorithms than TCP-Westwood [27].

Padmanabhan et al. [30] suggest to use the cached information of previous connections. Wang et al. [34] throttle down the exponential increasing rate when the congestion window approaches to *ssthresh*. However, in all cases, as an available bandwidth keep changing and also it is hard to pick an initial value of *ssthresh*, there is always a high possibility of wrong decisions.

4. HYBRID SLOW 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 lower system load. The algorithm does not change the doubling of cwnd during slow start, but based on hints 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 overshooting. When packet losses occur during slow start, HyStart behaves the same way as the original slow start protocol. HyStart is a plugin to the sender 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 ramp up fast cwnd as closely 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 in 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 even after timeouts when path conditions change after the timeouts so that the ssthresh set at the timeout event is a wrong estimation of the network capacity. These factors motivate the need for more intelligent slow start that finds a safe exit point when it can stop slow start and move to congestion avoidance.

The safe exit point corresponds to the size of cwnd before slow start finishes safely without incurring losses and 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 then causes low network utilization. We need a lower bound. After slow start, congestion avoidance will grow cwnd if there is no loss. Under 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 increase cwnd to another half ($C/2$). This also happens during a timeout where ssthresh. Therefore, it is reasonable to set the lower bound for a safe exit point to be $C/2$ minus the buffer space. As far as network utilization is concerned, when cwnd reaches beyond $B \times D_{min}$, it has 100% utilization. Thus, the buffer space does not influence the utilization. In summary, a safe exit point is bounded in 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 to be 0.5 and other high speed variants use a value larger than 0.5.

4.2 Algorithm Description

HyStart uses two pieces of information - the time space between consecutively received ACK packets and round-trip delays and 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 techniqu – it does not inject a probe packet of its own. Below we describe the two methods. Both run independently at the same time and slow start exits when any of them detects an exit point. Algorithm 1 shows its pseudo code.

4.2.1 ACK train length

Estimating unused bandwidth in a path has been an active topic of research [13, 24, 8]. Many bandwidth estimation techniques, such as packet-pair [25] and packet train [15] uses active probing to estimate bandwidth by sending probes back-to-back in a short period of time.

Dovrolis et al. [15] recently showed that the mean of packet train dispersion can be translated into Average Dispersion Rate (ADR), a rate in between 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 a train and δ_k is the inter-arrival time between packet k and $k+1$. Using the packet train length, the receiver measures the bandwidth $b(N)$ as follows.

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

However, packet-pair or train techniques are not practically implementable in existing commercial TCP stacks. First, they require modifications in 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 the current operating systems because it requires a high-resolution system clock and real-time interrupt handling. Because of 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 pose significant errors in the estimation. Under high system load of packet transmission during slow start, avoiding the system delays is not

Algorithm 1: Hybrid Slow 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 then
                //First exit point
                found <-- 1
        // Samples the delay from first few packets every round
        if cnt < nSampling then
            curRTT <-- min(curRTT, RTT)
            samplingCnt <-- samplingCnt + 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
```

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 is 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 intuition permits the following heuristics to measure the BDP of the network in real systems.

1. We can data packets transmitted during slow start as packet probes for a packet train. During slow start, many packets are sent in burst. We can use those packets transmitted within the same window as a packet train. This removes the need for active probes. To estimate $\Delta(N)$ at the sender, we use the train of ACKs received in response to a packet train. Figure 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 in 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} at the sender. 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 saturates. 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 every other packets received. In some systems, ACKs are arbitrarily delayed, but ACKs are always generated right at a reception of a packet (not necessarily a reception of every packet, though). The ACK delay does not affect the performance of HyStart much because $\Lambda(N)$ is computed by taking the time difference between the reception times of the first and last ACKs in a train. Since ACKs are triggered right after a 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 thwarts the correct estimation of safe exit points. Analyze 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 is K and K' is our estimated BDP, and $\Lambda(N) = \Delta(N)$. Then if $a = b$, then 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, underutilization is very unlikely. For the upper bound, suppose

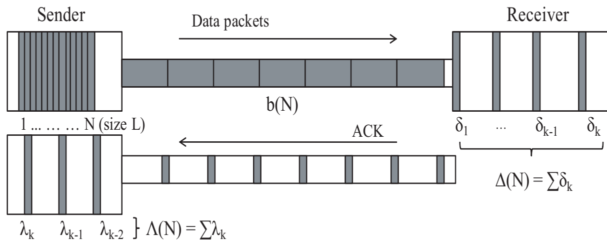


Figure 7: ACK train measurement

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 in the Internet [31], the fraction of paths whose reverse path delays are 3 times larger 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 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 increase in round-trip delays as another metric to find the safe exit point. However, as TCP sends packets in burst, it causes temporary queuing even if cwnd is less than 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 [11]. We remedy this problem by using the RTT samples of a few ACKs at the beginning of each ACK trains. Since packets arrive 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 RTTs of a few packets in the beginning of the k-th 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 larger 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 2.8GHz servers as shown in Figure 8 where two dummynet routers are located at the bottleneck between two end points. Two servers at each side are dedicated for sending and receiving TCP flows and Iperf [3] is used to measure the goodput of the TCP flows. The other two servers at each side are dedicated to generate a cross background traffic by using a modification of Surge [9], a Web traffic generator and Iperf for emulating long-lived TCP flows such as FTP connections.

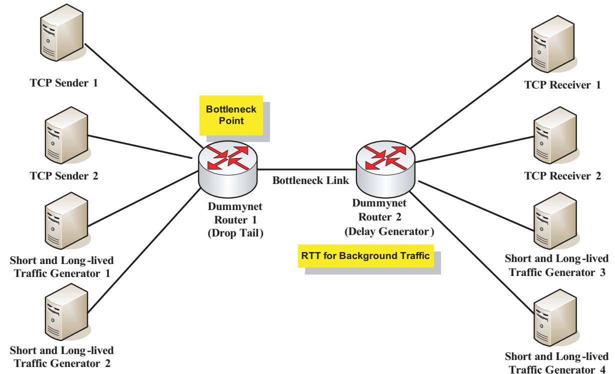


Figure 8: Testbed

The socket buffer size of background traffic machines is fixed to default 64KB while four TCP sender and receiver machines are configured to have a very large buffer so that the congestion control algorithm of TCP senders are only affected by the algorithm itself. Dummynet has been installed on two FreeBSD machines in the middle of the dumbbell and controls the bottleneck bandwidth, 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 [7].

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 1Gbps. To eliminate any overloading of the Dummynet routers, we set the bottleneck bandwidth below 400 Mbps. TCPProbe [6] and SIFTR [33] are used for actively tracking the TCP state variables in Linux and FreeBSD, respectively. For Windows XP, we use *tcpdump*. All performance numbers are averaged from 10 to 30 runs. Results are reported with 95% confidence interval.

5.2 Comparison with other Slow Starts

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

Figure 9 presents the trajectories of cwnd and ssthresh of six different slow-start protocols discussed in Section 3. The original standard slow start of TCP-SACK (SS) (a) shows a high burst of packets around ten seconds and experiences a high rate of losses. Limited slow start (LSS) (b) reduces the burst losses observed in SS (a) by limiting the increment of congestion window in one RTT. With HyStart (c), using the information of the ACK train length, the first flow finishes slow start around packet 870 at which point, cwnd reaches the BDP of the path. The second flow detects the congestion of the path using a delay increase caused by the first flow, and leaves slow-start early on. Adaptive Start

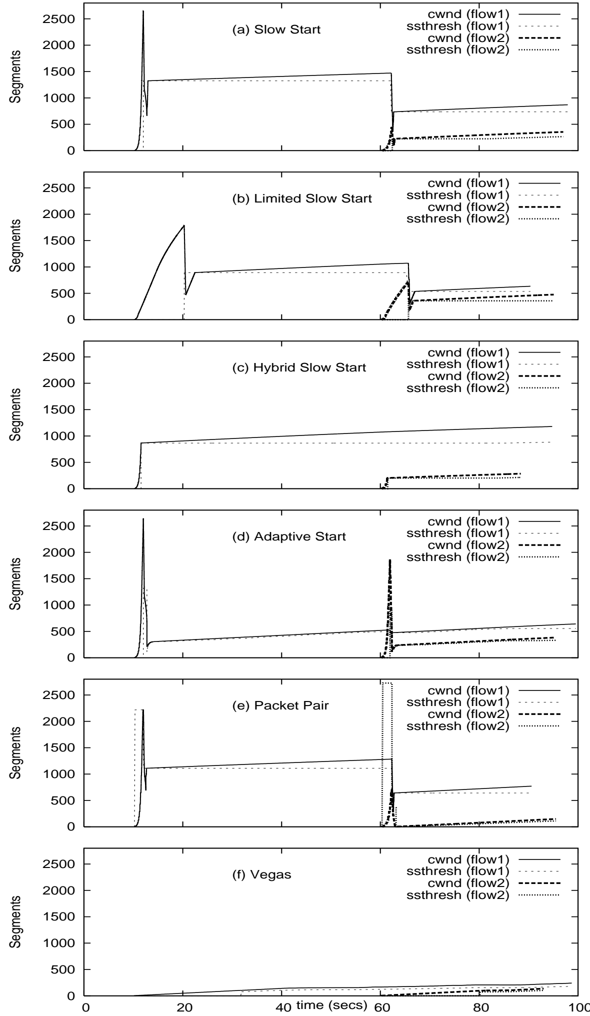
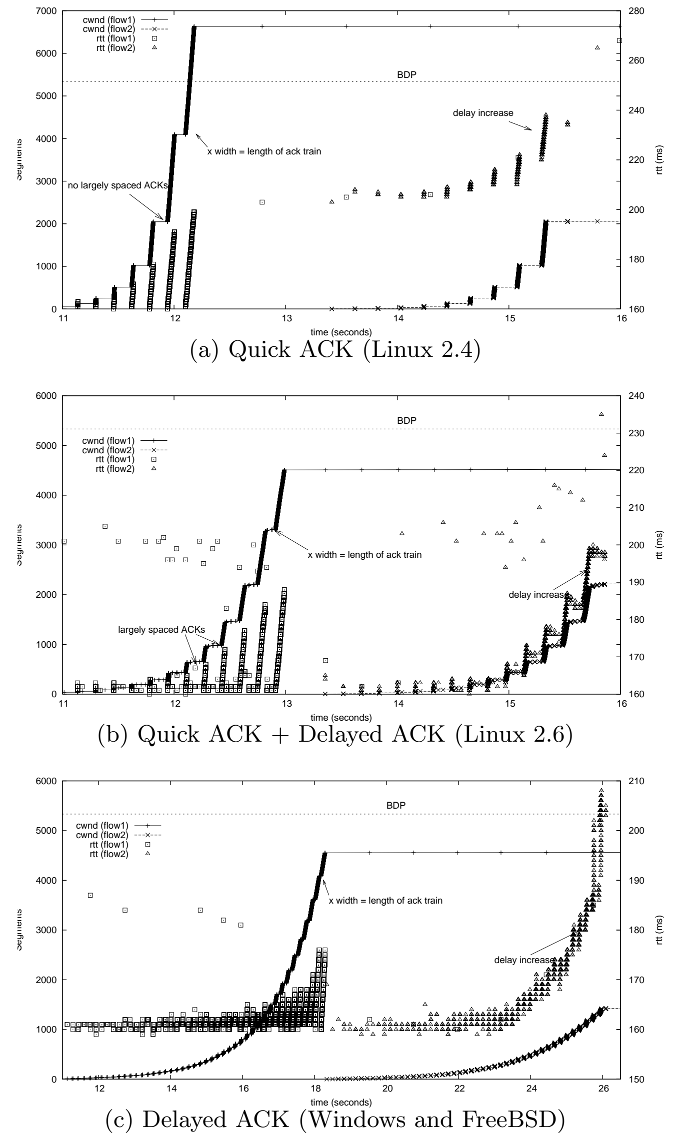


Figure 9: Two TCP-SACK flows with five different Slow Start proposals. The BDP for this experiment is around 883 packets. Therefore, when cwnd is between 883 and 1766 packets, the link is fully utilized.



(a) Quick ACK (Linux 2.4)

(b) Quick ACK + Delayed ACK (Linux 2.6)

(c) Delayed ACK (Windows and FreeBSD)

Figure 10: Two TCP-SACK flows with HyStart, by varying the ACK schemes at receivers.

(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 lack of high resolution clocks and real-time interrupt handling. The two flows estimates the capacity of 100Mbps path to be 248Mbps (2300 packets) and 308Mbps (2800 packets), respectively. Also, the modified slow-start of Vegas (VStart)(f) terminates slow-start too 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. Delayed ACKs implemented in Linux sends ACK for every two data packets received. When an ACK is not delayed, the spacing between consecutive ACKs are small and consistent. This makes a packet-train measurement effective. Delayed ACK may disturb the estimation of the ACK train length.

In this experiment, we introduce background traffic as the background traffic may disturb the behavior of the algorithm by varying available link bandwidth. The bottleneck bandwidth is set to 400Mbps 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 seconds of the testing. The background traffic we generated includes mid-sized flows [20] whose average throughput is around 50Mbps. The background traffic is introduced in both forward and reverse directions of the path.

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

two consecutive RTT rounds. Hence, it correctly calculates the ACK train length. The exit point is the round at which cwnd crosses over the BDP of the forward path. Linux, however, deliberately sends a quick ACK for up to 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 burst due to the initial quick ACKs. Figure 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 lack ACK of a train. This allows HyStart to correctly estimate the BDP of the network. HyStart finishes slow start 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 under this situation, our filtering scheme works fairly well. Figure 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 protocols. Most of 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 functions inside the Linux kernel so that it can be called from any protocols.

Figures 11 and 12 present the results of two CUBIC [22] flows with and without HyStart, respectively in the same experiment setup as in Figure 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 for 20 seconds because it overshoots up to 20,000 packets which is more than two times of 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 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 a bit before the full utilization of the link. The second flow, which joins at 130th second, detects the congestion of the path using delay increase and exits to congestion avoidance.

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 system. 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 3. Figure 13 shows that HyStart works much better than SS independent of the operating systems of the receivers. LSS works relatively as well as HyStart under this setup. Lower performance of SS under Windows XP and FreeBSD is because these operating systems force the sender to behave like TCP-NewReno and they cannot 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.

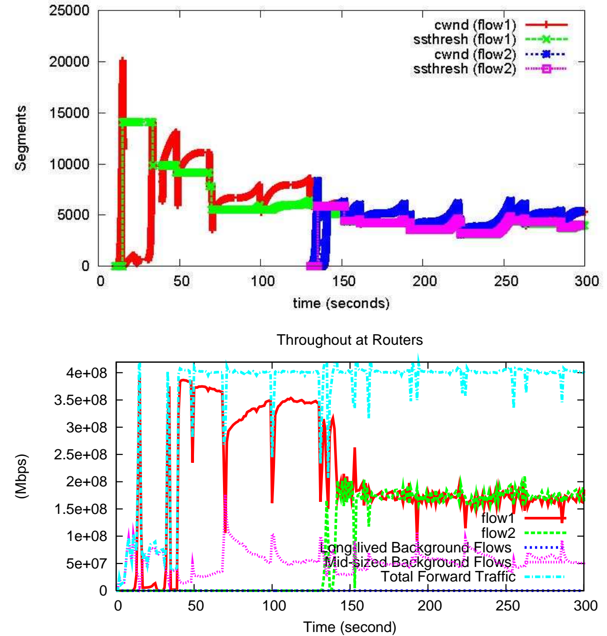


Figure 11: CUBIC with standard slow start. The first flow experiences heavy packet losses around the first 10 seconds and multiple timeouts over a 20 second period.

5.6 More diverse experimental settings

In this experiment, we compare the performance among 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 flows starting at the 10th and 40th seconds, respectively and the utilization is measured between the 10th and 70th second. We vary 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. We use TCP-SACK for both sender and receiver. Figure 14 show the performance results. HyStart shows consistently good network utilization independent of network bandwidth, buffer space and RTTs except with very small buffer spaces (where no protocols work well). Especially under large BDP networks, HyStart outperforms the other schemes more than 3-5 times. The other schemes suffer high performance losses as the BDP increases.

Figure 15 measures the CPU utilization of the sender when running various slow start protocols in Figure 14 (a). The results from other runs are similar. We find that the CPU utilization under SS and AStart is extremely high under

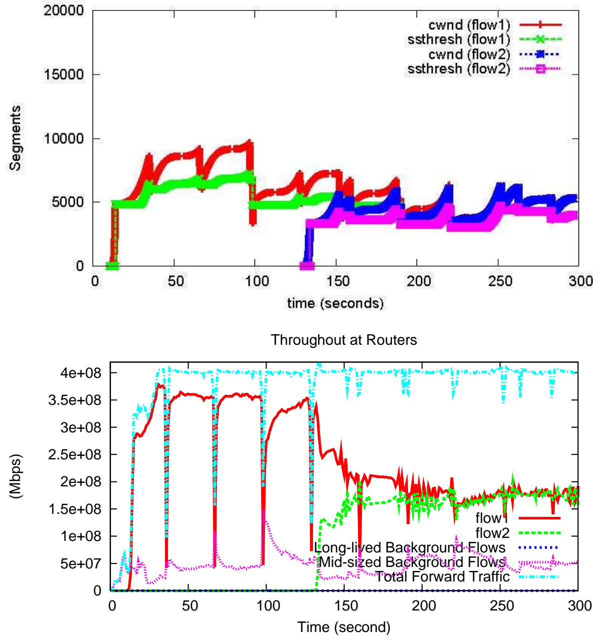


Figure 12: CUBIC with HyStart. HyStart exits from slow start before packet losses occur.

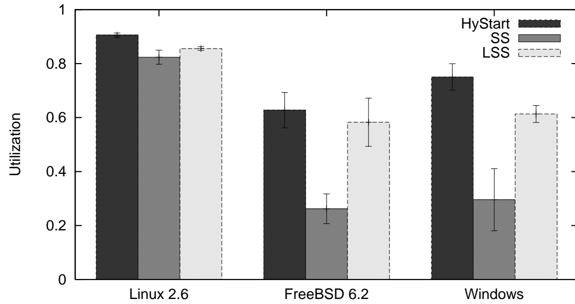
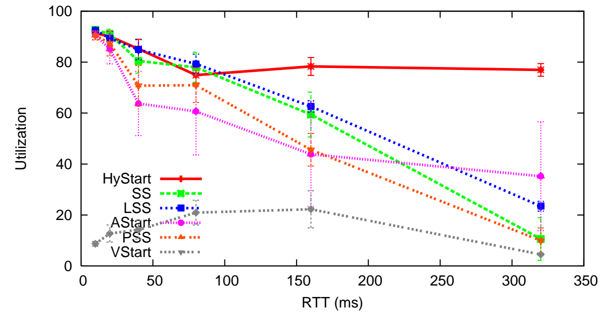
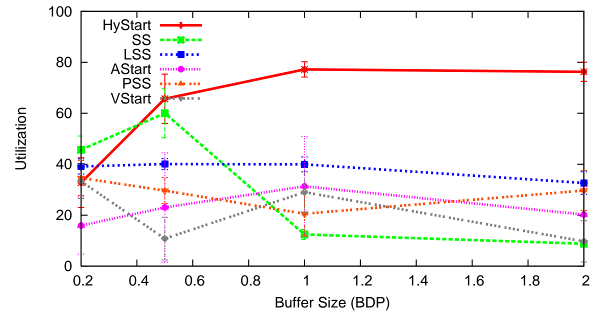


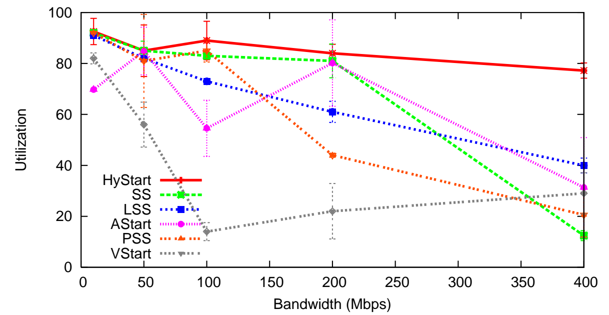
Figure 13: Two CUBIC flows with three different slow-start algorithms (HyStart, SS and LSS), by changing the OS of receivers.



(a) RTT vs. Utilization



(b) Buffer size vs. Utilization



(c) Bandwidth vs. Utilization

Figure 14: Two TCP-SACK flows with different slow-start algorithms by varying their RTTs (a), buffer sizes (b), and bandwidth (c).

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 much before packet losses occur. While PSS and VStart do so too early causing low network utilization, HyStart maintains a good network utilization.

5.7 Testing over asymmetric links

Recent Internet measurement [31] reports that when there is an asymmetry in delays in the Internet, more than 80% of the paths shows less than 20 ms delay difference between forward and reverse paths. The performance of HyStart may get affected by the 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 direction 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 both in forward and reverse direction to 400 Mbps, and vary the ratio between the forward delay and reverse delay from 1/3 to 3, and make the sum of the delay in both directions, RTT, to 120 ms. We introduce the background traffic comparable to 50 Mbps in both foward and reverse directions. Figure 16 (a) and (b) show the results respectively. HyStart obtains a good startup throughput even under high asymmetry in bandwidth and delay.

6. INTERNET2 EXPERIMENT

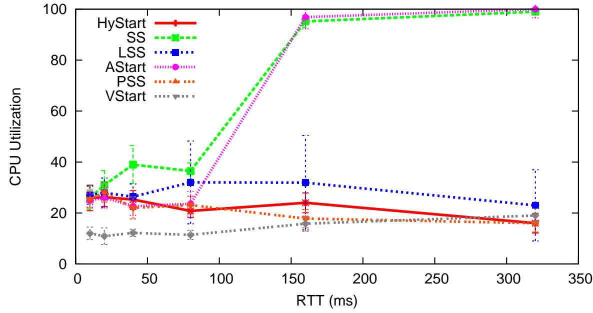


Figure 15: CPU utilization of the Linux sender with various slow start schemes for the run in Figure 14(a).)

We show the results of all the slow start proposals with TCP-SACK in three production networks such as Internet2 [2], National LamdaRail (NLR) [4] and GEANT [1].

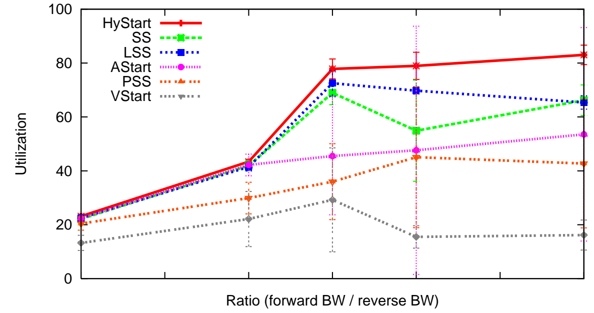
6.1 Experimental setups

Figure 17 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 1Gbps connection, and the GEANT path from North Carolina to Germany (107 ms RTT) has a 100 Mbps connection. Especially, the GEANT testing involves the servers inside the campus networks, so some of 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 50th second. We run the experiment in three different periods of each day (EDT 5 AM to 9 AM, 1 PM to 5 PM, and 9 PM to 1 AM). Linux is installed on all the machines in testbed.

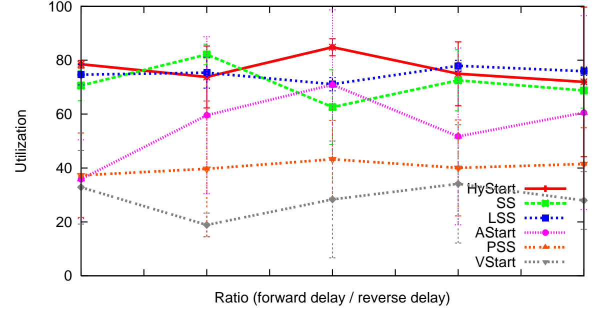
6.2 Internet2 results

Figure 18 shows the results of various slow-start algorithms with a TCP-SACK sender and receiver. SS, LSS and HyStart achieves good network utilization in relatively small BDP networks (both Chicago and Germany paths). HyStart shows the exceptionally good throughput for the high-BDP path between North Carolina and Japan, which has 250 ms RTT and 1Gbps link speed while LSS shows the worst performance due to its sluggish increase of cwnd in the same link. PSS mostly reports the under-estimated bandwidth so that its performance is not guaranteed. VStart also terminates the slow-start period prematurely and shows low utilization even under small BDP networks. In all testings, HyStart delivers the consistent start-up throughput.

To further motivate the use of HyStart, we run a TCP-NewReno sender at the North Carolina site over the Germany path. Since we cannot modify the receiver site servers, we change the sender side only. This set-up is emulating the behavior when using Windows XP receiver which forces the sender to behave like TCP-NewReno under medium and large size BDP networks. Figrue 19 shows the result. SS shows extremely low utilization because of slow recovery after heavy packet losses. However, LSS and HyStart gives



(a) Effect of asymmetric bandwidth ratios (f/r)



(b) Effect of asymmetric delay ratios (f/r)

Figure 16: Two TCP-SACK flows with different slow-start algorithms, under asymmetric delays (a) and bandwidth (b).

pretty good utilization as HyStart exits slow start before packet losses and LSS reduces the cwnd growth rates during slow start thus, preventing heavy packet losses. This result indicates that a protocol like HyStart or LSS must be used to gain a good performance under medium size BDP networks when the receiver is Windows XP or end-systems use TCP-NewReno.

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 frequent long blackouts with no transmission. This problem frequently happen 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 to 1000 ranges). Our new slow start protocol, HyStart, fixes this problem by detecting safe exit points of slow start that does not lead to heavy packet losses or low network utilization, preventively avoiding heavy system overload or low performance during the start-up of TCP. HyStart uses the concept of packet trains and RTT delay increase to find the safe exit points. To the best of our knowledge, our work is the first one that shows a practical implementation of packet pair and 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 gets maximized

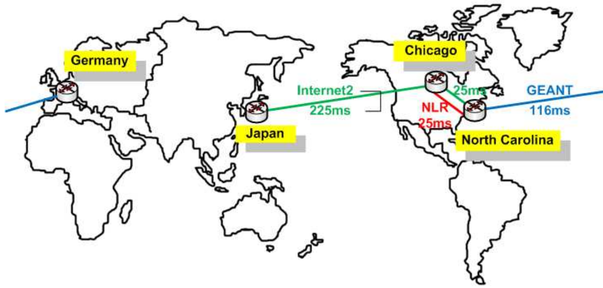


Figure 17: Research testbed (Internet2, NLR and GEANT)

when the receiver-side end systems is Windows as HyStart can greatly outperform standard slow start in this setup (even in medium-size BDP networks). This is very likely usage patterns as around 40% of the 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.

8. ACKNOWLEDGMENTS

This work is financially supported in part by a generous gift from Cisco Systems.

9. REFERENCES

[1] GEANT - pan-european research and education network. *http://www.geant.net/*.

[2] Internet2. *http://www.internet2.edu/*.

[3] IPerf - the TCP/UDP bandwidth measurement tool. *http://dast.nlanr.net/projects/Iperf/*.

[4] National LamdaRail. *http://www.nlr.net/*.

[5] Oprofile - a system profiler for linux. *http://oprofile.sourceforge.net/news/*.

[6] TCPProbe - observe the TCP flow with kprobes. *http://lxr.linux.no/linux+v2.6.26.5/net/ipv4/tcp_probe.c*.

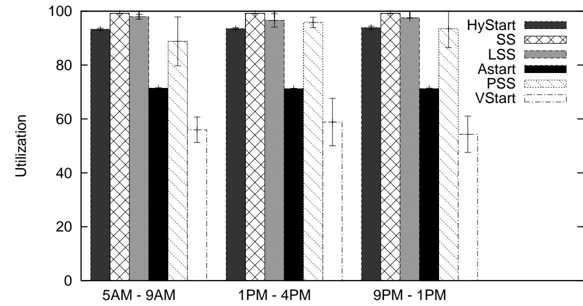
[7] Aikat, J., Kaur, J., Smith, F., and Jeffay, K. Variability in TCP round-trip times. In *Proceedings of the ACM SIGCOMM Internet Measurement Conference* (Miami, FL, October 2003).

[8] Allman, M., and Paxson, V. On estimating end-to-end network path properties. In *SIGCOMM '99: Proceedings of the conference on Applications, technologies, architectures, and protocols for computer communication* (New York, NY, USA, 1999), ACM, pp. 263–274.

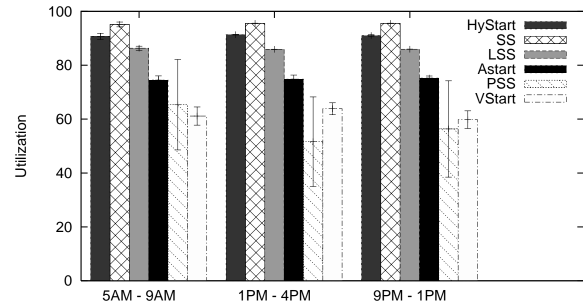
[9] Barford, P., and Crovella, M. Generating representative web workloads for network and server performance evaluation. In *Measurement and Modeling of Computer Systems* (1998), pp. 151–160.

[10] Blanton, E., Allman, M., Fall, K., and Wang, L. A Conservative Selective Acknowledgment (SACK)-based Loss Recovery Algorithm for TCP. RFC 3517 (Proposed Standard), Apr. 2003.

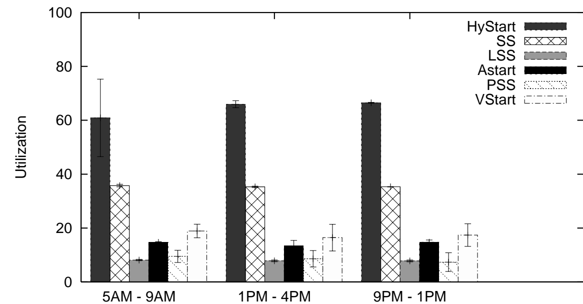
[11] Brakmo, L., and Peterson, L. TCP vegas: End to end congestion avoidance on a global internet. *IEEE Journal of Selected Areas in Communications* (October 1995).



(a) Chicago (1 Gbps and 25 ms RTT)



(b) Germany (100 Mbps and 107 ms RTT)



(c) Japan (1 Gbps and 250 ms RTT)

Figure 18: The network utilization of two TCP-SACK flows with different slow-start algorithms over the three Internet paths.

[12] Brakmo, L. S., and Peterson, L. L. TCP vegas: End to end congestion avoidance on a global internet. *IEEE JSAC 13*, 8 (1995), 1465–1480.

[13] Carter, R. L., and Crovella, M. E. Measuring bottleneck link speed in packet-switched networks. *Perform. Eval. 27-28* (1996), 297–318.

[14] Casetti, C., Gerla, M., Mascolo, S., Sanadidi, M. Y., and Wang, R. TCP Westwood: Bandwidth estimation for enhanced transport over wireless links. In *Proceedings of ACM Mobicom* (Rome, Italy, July 2001).

[15] Dovrolis, C., Ramanathan, P., and Moore, D. Packet-dispersion techniques and a capacity-estimation methodology. *IEEE/ACM Transactions on Networking 12*, 6 (2004), 963–977.

[16] Floyd, S. Limited Slow-Start for TCP with Large Congestion Windows. RFC 3742 (Experimental), Mar. 2004.

[17] Floyd, S., Allman, M., Jain, A., and Sarolahti, P. Quick-Start for TCP and IP. RFC 4782 (Experimental), Jan. 2007.

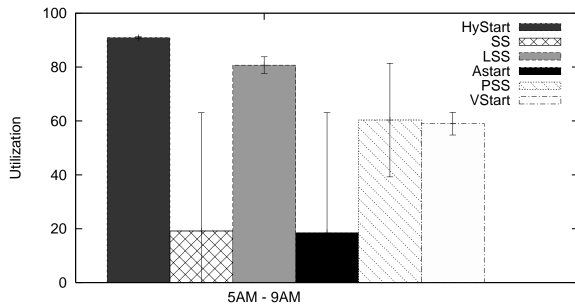


Figure 19: The network utilization with various slow start protocols when the sender is TCP-NewReno. The run is over a path between Germany and North Carolina.)

[18] Floyd, S., Henderson, T., and Gurtov, A. The NewReno Modification to TCP's Fast Recovery Algorithm. RFC 3782 (Proposed Standard), Apr. 2004.

[19] Floyd, S., Mahdavi, J., Mathis, M., and Podolsky, M. An Extension to the Selective Acknowledgement (SACK) Option for TCP. RFC 2883 (Proposed Standard), July 2000.

[20] Ha, S., Le, L., Rhee, I., and Xu, L. Impact of background traffic on performance of high-speed TCP variant protocols. *Computer Networks 51*, 7 (2007), 1748–1762.

[21] Ha, S., and Rhee, I. Hybrid slow start for high-bandwidth and long-distance networks. In *Proceedings of the fourth PFLDNet Workshop* (Manchester, UK, March 2008).

[22] Ha, S., Rhee, I., and Xu, L. Cubic: A new tcp-friendly high-speed tcp variant. *SIGOPS Oper. Syst. Rev. 42*, 5 (2008), 64–74.

[23] Hoe, J. C. Improving the start-up behavior of a congestion control sheme for TCP. In *ACM SIGCOMM* (1996).

[24] Jain, M., and Dovrolis, C. End-to-end available bandwidth: measurement methodology, dynamics, and relation with tcp throughput. *IEEE/ACM Trans. Netw. 11*, 4 (2003), 537–549.

[25] Keshav, S. A control-theoretic approach to flow control. *Proceedings of the conference on Communications architecture & protocols* (1993).

[26] Leith, D., Shorten, R., McCullagh, G., Heffner, J., Dunn, L., and Baker, F. Delay-based AIMD congestion control. In *PFLDNet* (February 2007).

[27] Mascolo, S., Casetti, C., Gerla, M., Sanadidi, M. Y., and Wang, R. TCP westwood: Bandwidth estimation for enhanced transport over wireless links. In *Mobile Computing and Networking* (2001), pp. 287–297.

[28] Mathis, M., Mahdavi, J., Floyd, S., and Romanow, A. TCP Selective Acknowledgment Options. RFC 2018 (Proposed Standard), Oct. 1996.

[29] Medina, A., Allman, M., and Floyd, S. Measuring the evolution of transport protocols in the internet. *SIGCOMM Comput. Commun. Rev. 35*, 2 (2005), 37–52.

[30] Padmanabhan, V., and Katz, R. Tcp fast start: a technique for speeding up web transfers.

[31] Pathak, A., Pucha, H., Zhang, Y., Hu, Y. C., and Mao, Z. M. A measurement study of internet delay asymmetry. In *Proceedings of Passive and Active Measurement Conference (PAM)* (Cleveland, Ohio, April 2008), pp. 37–52.

[32] Scharf, M., Hauger, S., and K"ogel, J. Quick-start TCP: From theory to practice. In *Proceedings of the third PFLDNet Workshop* (UK, March 2008).

[33] Stewart, L., Armitage, G., and Healy, J. Characterising the behavior and performance of siftr v1.1.0. *Technical Report 070824A* (August 2007).

[34] Wang, H., Xin, H., Kang, D., and Shin, G. A simple refinement of slow start of TCP congestion control, 2000.

[35] Wang, R., Pau, G., Yamada, K., Sanadidi, M., and Gerla, M. TCP Startup Performance in Large Bandiwdth Delay Networks. In *Proceedings of IEEE INFOCOM* (Hong Kong, 2004).