Hybrid slow start for high-bandwidth and long-distance networks

Sangtae Ha
Institute of Electrical and Electronics Engineers
124 PUBLICATIONS 5,675 CITATIONS
SEE PROFILE

Injong Rhee
North Carolina State University
82 PUBLICATIONS 10,152 CITATIONS
SEE PROFILE

Hybrid Slow Start for High-Bandwidth and Long-Distance Networks

Sangtae Ha and Injong Rhee
Dept. of Computer Science
North Carolina State University
Email: {sha2, rhee}@ncsu.edu

Abstract— **Slow Start is a technique to probe for unknown and time-varying available bandwidth of a network path. A sender increases its congestion window by one for each ACK received (when ACKs are not delayed), which effectively doubles its congestion window when receiving ACKs for all the packets within a congestion window. Even if an exponential increase of congestion window during Slow Start grabs unused bandwidth quite well, a large number of packet losses within an RTT is inevitable because of its over-shooting. Furthermore, for fast and long distance networks, a large number of packet losses would result in unnecessarily long timeouts and create system performance bottlenecks related to handling the recovery of lost packets. We propose a new algorithm, called *Hybrid Slow Start* that maintains the existing Slow Start mechanism of TCP-NewReno but provides trust-worthy signals to Slow Start for safely switching to Congestion Avoidance without incurring an extremely large number of packet losses. Hybrid Slow Start uses two pieces of information - ACK train length and increase in packet delays. By measuring ACK train length, a TCP sender roughly infers the maximum number of packets in flight which is typically smaller than the Bandwidth Delay Product (BDP) of the path if taken into account cross traffic and routing delays along the path. Increase in delays for the first few packets in each RTT round during Slow Start strongly indicates the path is getting congested by other traffic. Hybrid Slow Start is easy to implement using a only very small set of TCP state variables available in standard TCP. We validate our claims by applying Hybrid Slow Start to CUBIC and testing it under a realistic mix of background traffic with TCP receivers implemented in FreeBSD, Windows and Linux.**

I. INTRODUCTION

Slow Start in TCP doubles the congestion window of the sender (*cwnd*) for each corresponding received ACKs in every RTT. This is a fast way to probe for the currently unknown available bandwidth of a network path. However, the exponential growth of *cwnd* during Slow Start results in a large number of packet losses within one round-trip time, especially in a large BDP network, and more seriously disturbs TCP self-clocking and therefore causes TCP timeouts. For a fast and long distance network, a huge number of packet losses frequently forces TCP to experience long black-outs (sometimes more than 100 seconds), either caused by very long timeouts or by system-related bottlenecks caused by operations such as freeing up a large number of packet buffers, e.g., SKBs in Linux, within a short period of time.

A recent report [1] indicates that most of residential lines (cable modems and DSLs) are over-provisioned with larger buffers (more than 100% BDP buffer size). With a large buffer size, TCP has more chances to experience back-to-back loss events within several consecutive RTTs. For instance, if the buffer of the bottleneck router is large enough to hold more than two or three times of BDP, TCP can seriously overshoot its *cwnd* well beyond its BDP.

In this paper, we address the overshooting problem of Slow Start. Briefly our solution works as follows. While keeping the exponential growth of Slow Start as it is still a fast way to probe for unknown available bandwidth, our algorithm helps Slow Start find the "exit" point where it can "safely" finish Slow Start and switch to Congestion Avoidance. Our proposed algorithm, called *Hybrid Slow Start*, uses two pieces of information - ACK train length and increase in packet delays. The ACK train length is measured by calculating the sum of inter-arrival times of all the closely spaced ACKs within an RTT round. The train length is strongly affected by the bottleneck bandwidth, routing delays and buffer sizes along the path, and is easily stretched out by congestion caused by cross traffic in the path, so by estimating the train length, we can reliably find a safe exit point of Slow Start. For every RTT round during Slow Start, the ACK train length is re-calculated and compared against the estimated smallest forward delay of the path. This information is particularly reliable when the path is not congested so that the estimated smallest forward delay implies the delay during no congestion.

Increase in packet delays during Slow Start may indicate the possibility of the bottleneck router being congested. Suppose that there is only one flow in the bottleneck link. Because of bursty transmission of packets, the bottleneck router can be temporarily congested although the average transmission rate within an RTT can be much smaller than the bottleneck bandwidth. Thus, the bottleneck router can go through a repeated process of building and draining the router buffers during a single RTT. This makes the estimated round-trip delays of packets less trust-worthy signals for congestion. To avoid this problem, we use the round-trip delays of the first few packets of each packet train because those packets are unlikely subject to the estimation problems caused by TCP bursty transmission. Therefore, increase in delays for the first few packets in each RTT round strongly implies the presence of other traffic in the bottleneck links. This delay information can be useful because it works well even when the path is congested by other traffic. This signal may provide a false-positive indication of congestion building up before packet losses, but our experimental evaluation proves that it is a reliable way of finding the exit point of Slow Start.

Hybrid Slow Start sets *ssthresh* based on these two indicators and forces Slow Start to safely exit into Congestion Avoidance. The protocol is incorporated into the slow start mechanism of CUBIC which is the current default of Linux. We implemented Hybrid Slow Start on the latest Linux kernel and tested with Linux, FreeBSD, and Windows receivers in the testbed and find that the results are very promising.

The remainder of this paper is organized as follows. Section II gives related work, Section III presents our motivation and details of our detection mechanisms, Section IV presents the experimental results of our approach and Section V gives conclusions.

II. Related Work

Hoe [2] 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 literature [3], 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. With Hoe's modification, as multiple flows can get the same answer in the worst case, the estimation can overshoot *cwnd* to the value of $N * C$ where N is number of flows and C is the capacity of the link.

Vegas [4] 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-up in the router due to bursty TCP transmission [5].

Limited Slow Start [6] prevents 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 [7] determines its allowed sending rate very quickly, but it needs cooperation with the routers that approve Quick-Start along the path.

Adaptive Start [5] repeatedly resets its *ssthresh* to the value of the expected rate estimation (ERE) when the ERE is greater than the current *ssthresh*. Therefore, with Adaptive Start, TCP-NewReno repeats exponential growth and linear growth until it experiences a packet loss. However, Adaptive Start is slower than Slow Start, and it is not easily integrated with other congestion control algorithms than TCP-Westwood [8].

Most recently, Delay-based Slow Start [9] is proposed. It uses the fact the sending rate is doubled at each RTT round during Slow Start and the maximum queueing time can be estimated by $RTT_{max} - RTT_{min}$. When the queueing delay, denoted by $RTT - RTT_{min}$, is larger than a certain threshold (τ_0), the algorithm sets the current congestion window to the $max\_ssthresh$ of Limited Slow Start, so that the number of RTTs to reach a given window takes $O(RTT/\tau_0)$. Using the threshold (τ_0) works well during Congestion Avoidance because TCP already has RTT_{max} upon packet losses and hence infers the queueing time ($RTT_{max} - RTT_{min}$) without problems. However, during Slow Start, RTT_{max} keeps increasing until TCP experiences initial packet losses, so calculating a faithful drain time may be difficult.

Except for changing the algorithm itself, Padmanabhan et al. [10] use the cached information of previous connections. Wang et al. [11] throttle down the exponential increasing rate when the congestion window approaches to *ssthresh*. However, in all cases, as an available bandwidth keep changing, there is always a possibility of a wrong (i.e., false-negative) decision.

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

III. Hybrid Slow Start

In this section we describe Hybrid Slow Start that reduces packet losses during Slow Start substantially and hence achieves better throughput. We first show the performance issues of existing Slow Start to motivate the new protocol, and then explains the algorithm in detail.

A. Motivation

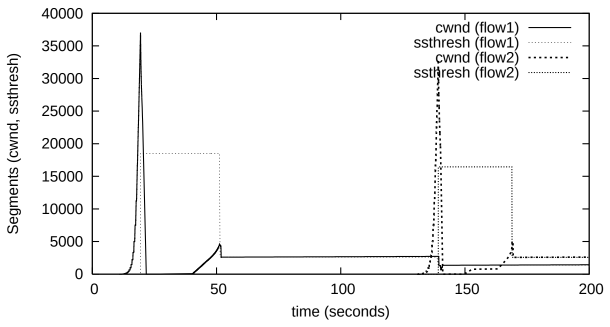
During Slow Start, TCP doubles its congestion window per RTT. This results in a large increase of the congestion window in one RTT, followed by a large number of packet losses. Figure 1 shows the results of Slow Start over an emulated trans-oceanic link. For this experiment, we set the bandwidth of the link to 400Mbps, one-way delay to 180ms, and the buffer size to 100% BDP (12333 packets) of the link[†]. As shown in Figure 1 (b), about one BDP worth of packets (12333 packets) are lost in one RTT because of reckless ramp-up of Slow Start. Figure 1 (a) presents that the back-to-back losses due to this serious overshoot leaves $ssthresh$ at a very small value, resulting in very poor overall utilization.

While there are a couple of experimental RFCs [6] [7] to address this overshooting of *cwnd*, still most of TCP stacks use the original Slow Start mechanism which sets the initial value of *ssthresh* to *infinity* for probing an available bandwidth regardless of path capacity until packet losses occur. We can completely re-design Slow Start but a small plugin into the current algorithm gives a better chance of being implemented on many systems. In the following section, we propose a plugin to the existing Slow Start, called Hybrid Slow Start, which is not CPU-intensive due to its low algorithmic complexity and is easy to implement as it uses only TCP state variables available in common TCP stacks.

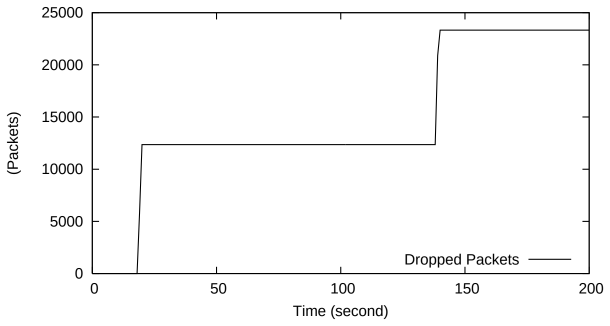
B. Algorithm

Hybrid Slow Start uses two indicators which decide when TCP exits from Slow Start and switches to Congestion Avoidance. Algorithm 1 shows the pseudo code.

[†]In order to emulate the bandwidth close to typical 1G trans-oceanic path, we double the delay while fixing the bandwidth to 400Mbps, so that it has the same BDP as the trans-oceanic path of 800Mbps and 90ms one-way delay.



(a) $cwnd$ and $ssthresh$



(b) Number of dropped packets

Fig. 1. The BDP of this experiment is around 18Mbytes (12333 packets). As the buffer size is set to 100% BDP of the path, the maximum amount of packets the path holds is around two times the BDP, 36Mbytes (24666 packets). As shown in (a), $cwnd$ has been increased to around 37000 with Slow Start, hence excess 12334 packets (37000 - 24666) are dropped in the router - See (b). A large number of drops forces TCP to experience unnecessary timeouts and prolonged recovery. As shown in (a), the first flow took more than 20 seconds for recovery.

1) ACK train length: we are inspired by the fact that bandwidth measurement techniques, whether based on packet-pair [12] or packet train [3], are only applicable when packets are sent in a burst, meaning back-to-back packet transmissions in a short period of time. Slow Start is a natural way, without injecting any additional probe packets, to send packets in burst. The burstiness, due to the doubling of $cwnd$ in every RTT during Slow Start, makes bandwidth estimation using these techniques more reliable without any overhead.

Research on bandwidth estimation has been active since the introduction of the packet-pair [12] bandwidth measurement technique. Further, measuring the dispersion of long packet trains is also introduced to measure the available bandwidth of a path. However, Dovrolis et al. [3] recently showed that the mean of packet train dispersion is related to another bandwidth metric, referred to as Average Dispersion Rate (ADR). They proved that ADR is in between an available bandwidth and a capacity of the path. Their promising results directly match with the indicator, referred to "ACK train length", which we found useful in setting the exit point of Slow Start.

Formally, the sender S sends N back-to-back probe packets of size L to the receiver R. When $N > 2$, we call these probe packets a packet train. By following the notations shown in Dovrolis et al. [3], 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, R measures the bandwidth $b(N)$

Algorithm 1: Hybrid Slow Start

```
Initialization:
$low\_ssthresh \longleftarrow 16$        $nSampling \longleftarrow 8$
At the start of each RTT round:
begin
    if $!found$ and $cwnd \leq ssthresh$ then
        // Save the start of an RTT round
        $roundStart \longleftarrow lastJiffies \longleftarrow Jiffies$
        $lastRTT \longleftarrow curRTT$
        $curRTT \longleftarrow \infty$
        $samplingCnt \longleftarrow nSampling$
end
On each ACK:
begin
    $RTT \longleftarrow usecs\_to\_jiffies(RTT_{us})$
    $dMin \longleftarrow min(dMin, RTT)$
    if $!found$ and $cwnd \leq ssthresh$ then
        // ACK is closely spaced, and the
         train length reaches to $T_{forward}$?
        if $Jiffies - lastJiffies \leq msecs\_to\_jiffies(2)$
        then
            $lastJiffies \longleftarrow Jiffies$
            if $Jiffies - roundStart \geq dMin/2$ then
                $found \longleftarrow 1$
        // Samples the delay
        if $samplingCnt$ then
            $curRTT \longleftarrow min(curRTT, RTT)$
            $samplingCnt \longleftarrow samplingCnt - 1$
        $\eta \longleftarrow max(2, \lceil lastRTT/16 \rceil)$
        // If the delay increase is over $\eta$
        if $!samplingCnt$ and $curRTT \geq lastRTT + \eta$
        then
            $found \longleftarrow 2$
        if $found$ and $cwnd \geq low\_ssthresh$ then
            $ssthresh \longleftarrow cwnd$
end
Timeout:
begin
    $dMin \longleftarrow \infty$        $found \longleftarrow 0$
end
```

by

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

Unlike the general packet train measurement where R measures the dispersion of the packet train, the ACK train measurement allows S to calculate the dispersion roughly by accumulating the inter-arrival times of corresponding ACKs. For simplicity, suppose the receiver doesn't delay an ACK. Then the ACK train length can be written as $\Lambda(N)$ = $\sum_{k=1}^{N-1} \lambda_k$, where λ_k is the inter-arrival time between an ACK k and $k+1$. Because $\Lambda(N) \geq \Delta(N)$, this inequality ensures the "safer" estimation of the available bandwidth than that of the original packet-train estimation. In order to estimate $\Delta(N)$ in Equation (1), S measures the ACK train length, denoted as $\Lambda(N)$, by

$$\Lambda(N) = \frac{(N-1)L}{b(N)} \tag{2}$$

Therefore, when $\Lambda(N)$ is the same as the minimum forward delay of the link, $b(N)\Lambda(N)$ is simply the BDP of the forward link which holds $(N-1)L$ packets in flight. Without knowing the exact number of packets in flight, denoted as $(N-1)L$, we can use $\Lambda(N)$ to infer whether the packets in flight is approaching the BDP of a path. One more advantage of this approach is that a TCP sender doesn't need a high-resolution clock and fine-grained timestamps because the TCP sender needs only a rough estimation of ACK train.

2) Delay increase: when the path is not completely empty and one or more greedy TCP flows exist, a new joining flow would detect an increase in packet delays during Slow Start. Precisely, a TCP sender measures an increase in packet delays of first few packets in each RTT round during Slow Start. When the increase in delays is over a certain threshold, it strongly implies other flows are building up the bottleneck queues. In fact, using an increase in delays reduces the chances of disturbing existing flows in the path and thus doesn't incur a large packet losses, resulting in less synchronization losses in the link.

Formally, let RTT_k be the minimum RTT during a kth RTT round and η be a threshold. The delay increase is detected when RTT_k satisfies the following:

$$RTT_k \geq RTT_{k-1} + \eta \tag{3}$$

Note that this scheme does not require the estimation of the minimum delay of a path and thus can be used for both congested and uncongested cases.

IV. Experimental Evaluation

A. Comparison with other Slow Start proposals

We compare the performance of Hybrid Slow Start with that of other Slow Start protocols discussed in Section II. All protocols are implemented over TCP-SACK in Linux 2.6.23.9. We set up a simple dumbbell topology in Linux/FreeBSD based dummynet emulation testbed used in [13]. 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 2 presents the trajectories of $cwnd$ and $ssthresh$, of six different Slow Start protocols discussed in Section II. The original slow start of TCP-SACK (a) shows a high burst of packets around ten seconds and experiences a high rate of losses. Limited Slow Start (b) reduces the burst losses observed in Slow Start (a) by limiting the increment of congestion window in one RTT. With Hybrid Slow Start (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 (d) calculates ERE upon receiving ACKs and sets $ssthresh$ to the ERE value if the ERE value is larger than $cwnd$ during slow start. But ERE is consistently smaller than $cwnd$ in our testing. As a result, Adaptive Start shows the same behavior with Slow Start. Packet-pair slow start (e) shows an inconsistent estimation of path capacity for each run. For instance, (e) shows that 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 (f) terminates slow start prematurely which is consistent to the discussion in Section II.

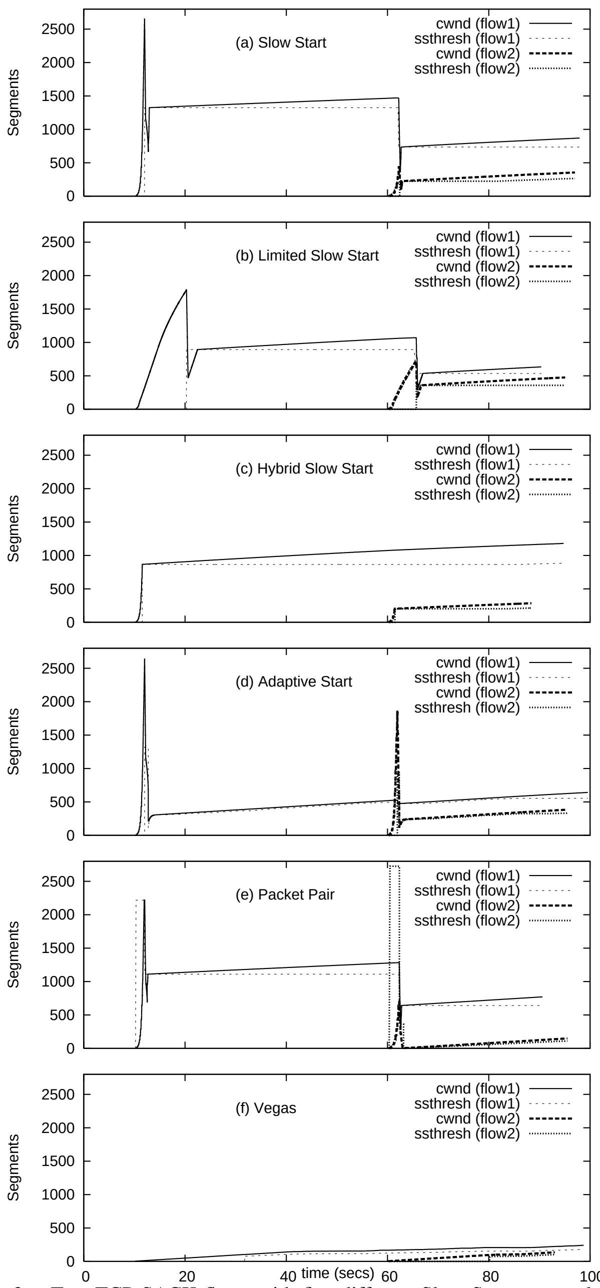


Fig. 2. Two TCP-SACK flows with five different Slow Start proposals. For this experiment, we fix the bandwidth to 100Mbps, RTT to 106ms, and the buffer size to 100% BDP of a flow. The BDP for this experiment is around 883 packets. Therefore, when $cwnd$ is between 883 and 1766 packets, the link is fully utilized.

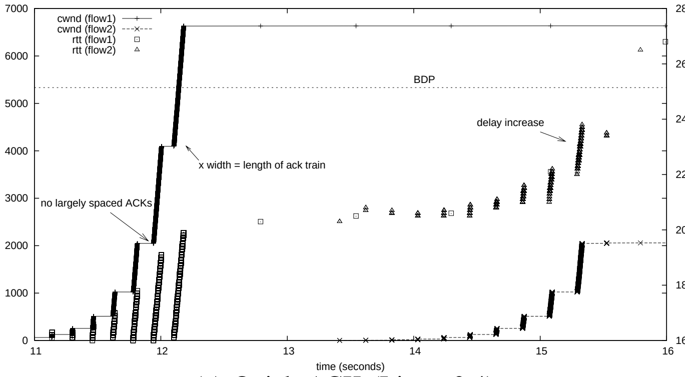
B. Impact of different ACK schemes on Hybrid Slow Start

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 [14] whose average throughput is around 50Mbps. The background traffic is introduced in both forward and reverse directions of the path.

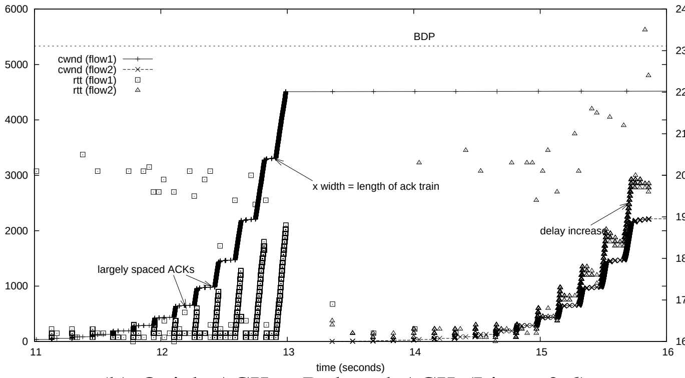
Figure 3 presents $cwnd$ and RTT of two flows, for three different ACK schemes. The ACK schemes we tested include (a) a quick ACK, (b) a quick ACK initially and a delayed ACK later on, and (c) a delayed ACK. When an ACK is not delayed, the spacing between consecutive ACKs are small and consistent. This makes a packet-train measurement so effective that a sender needs to calculate the time difference between the start of RTT round and the time when an ACK arrives. Figure 3 (a) confirms that no delayed ACKs are found in between two consecutive RTT rounds. Hence, no difficulty exists in calculating the ACK train length.

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 which have been sent in burst due to the initial quick ACKs. Figure 3 (b) shows that a small number of largely spaced ACKs are found in between two chunks of ACK trains. Therefore, the sender does not consider these largely spaced ACKs when calculating the ACK train length.

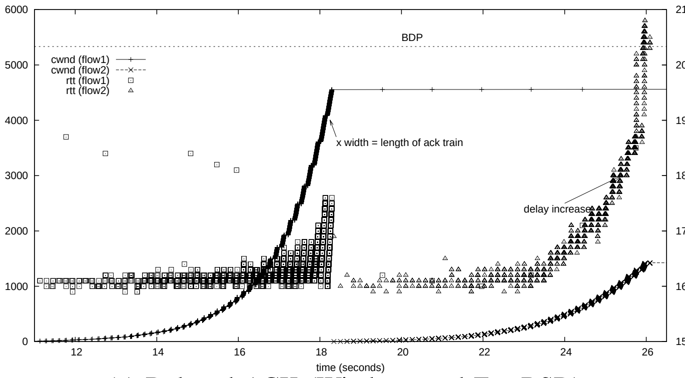
FreeBSD and Windows send delayed ACKs from the beginning of a connection and consequently, ACKs spread over an entire RTT round. In this case, the sender obtains a meaningful ACK train length only when ACKs fill the gap between their neighbor ACKs and finally makes them closely spaced. As the algorithm calculates the ACK train length only over closely spaced ACKs, it handles all the three ACK schemes [§]. Also, we can see the delay indicator of Hybrid Slow Start successfully detecting the congestion of the path for all three ACK schemes.



(a) Quick ACK (Linux 2.4)



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



(c) Delayed ACK (Windows and FreeBSD)

Fig. 3. Two TCP-SACK flows with Hybrid Slow Start, by varying the ACK schemes at receivers. For this experiment, we fix the bandwidth to 400Mbps, RTT to 160ms, and the buffer size to 100% BDP of a flow (5333 packets).

C. Testing under more diverse experimental settings

In this experiment, we compare the performance among three representative Slow Start algorithms, Hybrid Slow Start, Slow Start, and Limited Slow Start, under more diverse experimental settings. We use two CUBIC flows starting at the 10th and 40th seconds respectively, and the utilization measured until the 70th second. In the experiment, the bottleneck bandwidth is varied from 10Mbps to 400Mbps, RTT from 20ms to 160ms, and the buffer size from 10% to 200% BDP.

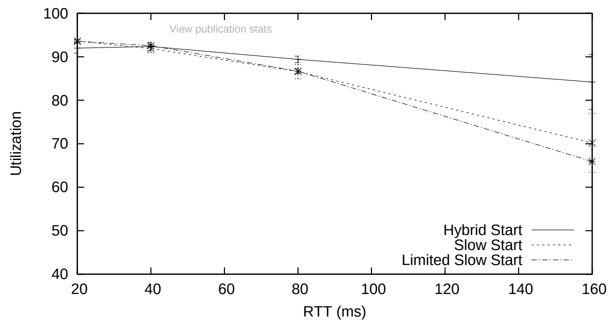
Figure 4 shows that Hybrid Slow Start performs consistently better for all three parameters. Figures 4 (a) and (c) show that when the BDP increases, Hybrid Slow Start successfully prevents a large number of packet losses relatively well while Slow Start and Limited Slow Start fail to do so. Also, Figure 4 (b) shows that Hybrid Slow Start works with the over-provisioned buffer size.

D. Testing with Linux, FreeBSD, and Windows Receivers

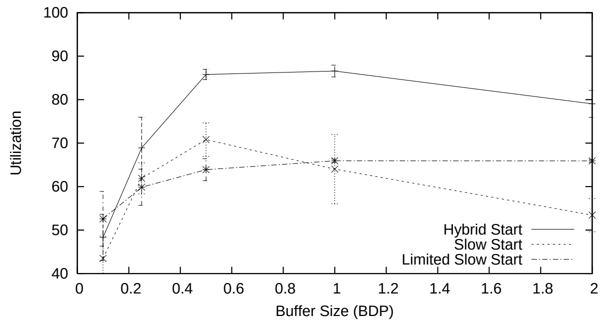
In this experiment, we evaluate the performance of Slow Start algorithms by varying the operating system of receiver machines. Figure 5 shows that Linux receivers obtain more throughput than Windows and FreeBSD receivers, thanks to the quick ACK of Linux during an initial stage of the connection. The throughput of Slow Start with FreeBSD and

[‡]Linux sends a quick ACK for initial $min(rcv\_wnd/2, 16)$ packets

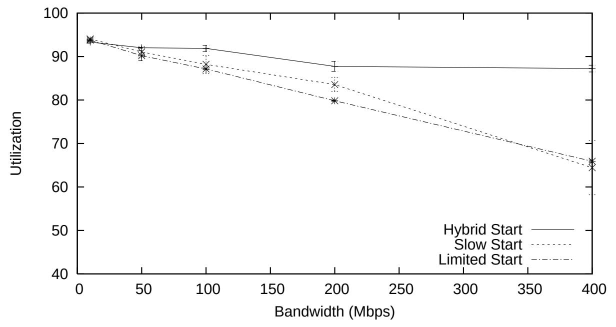
[§]Although the quick ACK scheme estimates 10% more than the BDP of the path, the bottleneck buffers most likely compensate for this overestimation



(a) RTT vs. Utilization



(b) Buffer size vs. Utilization



(c) Bandwidth vs. Utilization

Fig. 4. Two CUBIC flows with three different Slow Start algorithms, by varying their RTTs (a), buffer sizes (b), and bandwidth (c). Two flows start at the 10th and 40th seconds respectively, and utilization is measured until the 70th second. For RTT and bandwidth experiments, we fix the buffer size to 100% BDP of a flow. For buffer size experiments, we fix RTT to 160ms.

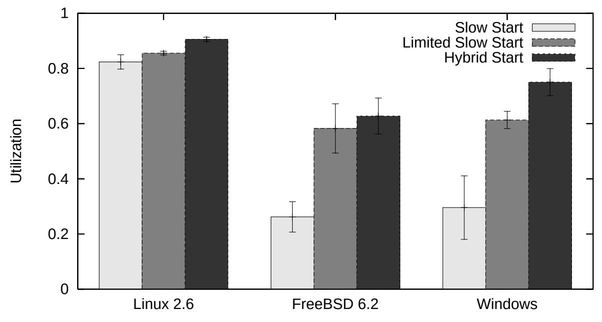


Fig. 5. Two CUBIC flows with three different Slow Start algorithms, by changing the OS of receivers. The sender machines are fixed to Linux 2.6. The experimental parameters used for this testing are identical to the experiment shown in Figure 4, except that we fix RTT to 100ms for this testing.

Windows is significantly worse than that of Linux. We observe that FreeBSD and Windows receivers incur long blackouts¶ (more than 20 seconds) after Slow Start. However, with Hybrid Slow Start or with Limited Slow Start, FreeBSD and Windows improve the throughput noticeably, indicating that large burst losses introduced by Slow Start exacerbate their performance quite a bit. Overall, we can see that Hybrid Slow Start works well with all three types of receivers.

V. Conclusion

In this paper, we present Hybrid Slow Start, which determines the termination of Slow Start using two indicators of path characteristics. We show that using ACK train and delay information can significantly improve the reliability of this decision. The proposed Hybrid Slow Start is a small plugin to an existing Slow Start and can be easily integrated with existing TCP congestion control algorithms. We integrated this plugin into an existing CUBIC protocol and validate its operation over a Linux/FreeBSD based *dummynet* testbed. This is still an on-going work and needs more testing over real production networks. Our future work includes refinements for handling asymmetric link and congestion on the backward path.

¶We need a further investigation to see whether it is because of buggy SACK implementation.

VI. Acknowledgments

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

References

[1] M. Dischinger, A. Haeberlen, K. P. Gummadi, and S. Saroiu, “Characterizing residential broadband networks,” in *IMC’07*, 2007, pp. 43–56.
[2] J. C. Hoe, “Improving the start-up behavior of a congestion control sheme for TCP,” in *ACM SIGCOMM*, 1996.
[3] C. Dovrolis, P. Ramanathan, and D. Moore, “Packet-dispersion techniques and a capacity-estimation methodology,” *IEEE/ACM Transactions on Networking*, vol. 12, no. 6, pp. 963–977, 2004.
[4] L. S. Brakmo and L. L. Peterson, “TCP vegas: End to end congestion avoidance on a global internet,” *IEEE JSAC*, vol. 13, no. 8, pp. 1465–1480, 1995.
[5] R. Wang, G. Pau, K. Yamada, M. Sanadidi, and M. Gerla, “TCP Startup Performance in Large Bandiwdth Delay Networks,” in *Proceedings of IEEE INFOCOM*, Hong Kong, 2004.
[6] S. Floyd, “Limited Slow-Start for tcp with large congestion windows,” *RFC 3742*, March 2004.
[7] S. Floyd, M. Allman, A. Jain, and P. Sarolahti, “Quick Start for tcp and ip,” *RFC 4782*, January 2007.
[8] S. Mascolo, C. Casetti, M. Gerla, M. Y. Sanadidi, and R. Wang, “TCP westwood: Bandwidth estimation for enhanced transport over wireless links,” in *Mobile Computing and Networking*, 2001, pp. 287–297.
[9] D. Leith, R. Shorten, G. McCullagh, J. Heffner, L. Dunn, and F. Baker, “Delay-based AIMD congestion control,” in *PFLDNet*, February 2007.
[10] V. Padmanabhan and R. Katz, “Tcp fast start: a technique for speeding up web transfers,” 1998.
[11] H. Wang, H. Xin, D. Kang, and G. Shin, “A simple refinement of slow start of tcp congestion control,” 2000.
[12] S. Keshav, “A control-theoretic approach to flow control,” *Proceedings of the conference on Communications architecture & protocols*, 1993.
[13] S. Ha, L. Le, I. Rhee, and L. Xu, “Impact of background traffic on performance of high-speed tcp variant protocols,” *Computer Networks*, vol. 51, no. 7, pp. 1748–1762, 2007.
[14] P. Barford and M. Crovella, “Generating representative web workloads for network and server performance evaluation,” in *Measurement and Modeling of Computer Systems*, 1998, pp. 151–160.