A Simple Refinement of Slow-start of TCP Congestion Control

Haining Wang† Hongjie Xin‡ Douglas S. Reeves‡ Kang G. Shin†

†Department of EECS
The University of Michigan
Ann Arbor, MI 48109
{*hxw,kgshin*}@*eecs.umich.edu*

‡Department of Computer Science
North Carolina State University
Raleigh, NC 27695
{*hxin,reeves*}@*eos.ncsu.edu*

Abstract

This paper presents a new variant of Slow-start, called *Smooth-start*, which provides a smooth transition between the exponential growth and linear growth phases of TCP congestion window. Slow-start is known to make an abrupt transition between Slow-start and Congestion Avoidance, and hence, often causes multiple packet losses from a window of data and retransmission timeouts, which, in turn, reduce effective throughput and result in globe synchronization. Smooth-start solves this problem by approaching the slow-start threshold more gradually. Our extensive simulation results show that Smooth-start can significantly reduce both packet losses and traffic burstiness, thus improving the performance of TCP congestion control at the start of a TCP connection or after a retransmission timeout. Furthermore, it is very simple to implement Smooth-start and requires TCP modifications at the sending side only.

Keywords: Congestion control, TCP, Slow-start, performance evaluation

1 Introduction

The wide use of the TCP/IP protocol suite and the explosive growth of the Internet have made TCP congestion control crucial to the performance of the Internet. Many popular Internet application protocols, such as HTTP, ftp, and telnet, are implemented with TCP. Since these protocols generate a large percentage of traffic on the Internet, the TCP congestion control algorithm should be optimized by adapting itself to the common behavior of these protocols.

The latest Internet traffic measurements [25] indicate that many TCP flows are short-lived. The characteristics of the short transfers are that a relatively small number of data packets are delivered and a TCP connection is usually terminated before it reaches steady state. Moreover, the popularity of the Internet has increased the complexity and size of networks, which makes it take longer to probe appropriate control parameters of a TCP connection. Thus, the start-up period of a TCP connection can greatly influence the performance of short-lived TCP connections, and the satisfaction of users' requirements.

One of the problems in the current implementations of TCP congestion control, which degrades the TCP performance, lies in the Slow-start algorithm. TCP Slow-start is initiated both at the start of a TCP connection or after a retransmission timeout. The objective of Slow-start is to enable a TCP sender to discover the available network bandwidth by gradually increasing the amount of data injected into the network from an initial window size of one segment, which prevents the TCP sender from congesting the network with an large burst of data. Unfortunately, the approach that Slow-start uses to probe the network bandwidth is counter-intuitive. It dribbles data out at the beginning, and ends with a drastic leap to reach the slow-start threshold. If we fill a pipe or a container with water, the approach we take is opposite to the way Slow-start does.

Due to the approach that Slow-start uses to probe the network bandwidth, Slow-start suffers from the following two problems. First, since the Slow-start algorithm begins with sending one segment, it takes many round-trip times to reach the optimal operating point, resulting in poor utilization of the available bandwidth for short transfers which are small compared to the bandwidth-delay product of the path. Second, since a TCP sender has no knowledge about the capacity of the available resources on the networks and uses default parameters at the beginning of transmission, the exponential growth of the congestion window often leads the sender to send too many packets too quickly, thus causing a severe buffer overflow at the bottleneck link. The severe buffer overflow results in multiple packet losses from a window of data, thereby making the TCP senders lose their self-clocking. The subsequent retransmission timeouts cause a global synchronization [23, 27]. The global synchronization lowers the aggregate throughput and makes the network traffic load oscillate, and hence, the TCP performance degrades substantially.

To resolve the first problem, a larger initial window and TCP Fast-Start have been proposed [2, 3, 19] to speed up the transfer rate at the very beginning of Slow-start, which greatly benefits short TCP transfers. To efficiently recover multiple packet losses within a window of data, New-Reno, SACK and FACK [10, 17, 16] have been proposed to recover from bursty packet losses without losing self-clocking. Furthermore, to remove the second problem of Slow-start, estimated initial *ssthresh* and safe *ssthresh* [10, 26] have been proposed to replace the default setting of *ssthresh*. However, the abrupt transition of congestion window between exponential growth and linear growth phases that causes highly bursty traffic and frequent buffer overflows, remains unaddressed.

Although the estimated initial *ssthresh* and safe *ssthresh* can make significant improvements over an arbitrary default *ssthresh*, several obstacles exist that make the accurate and timely estimation of the congestion control parameters very challenging: (1) routes are often asymmetric, and routing asymmetry can lead to bandwidth and latency asymmetry; (2) with a varying number of connections, the remaining network capacity along the path also varies over time; (3) out-of-order packet delivery, which occurs frequently; (4) routes can also change dynamically; (5) multi-channel bottleneck links, which violate the assumption that there is a single end-to-end forwarding path; and (6) ACK compression, which makes the pacing between ACKs not reflect the bottleneck delay experienced by the data packets. Therefore, it is very difficult to set up and maintain an appropriate *ssthresh*. The improvement of estimated *ssthresh* is thus limited.

In this paper, we propose a modification of Slow-start, called *Smooth-start*, which complements the proposals for selecting an appropriate initial *ssthresh*. The objective of Smooth-start is to make a smooth transition between the exponential growth and linear growth phases of the congestion window by changing the way that the TCP sender uses to reach the slow-start threshold. A new threshold *smsthresh* is introduced, which is set to $ssthresh/2^t$, where t is a non-negative integer. As the window size becomes larger than *smsthresh*, we slightly reduce the acceleration rate of the congestion window. We call the period where the window size is in-between *smsthresh* and *ssthresh* the *smooth-start period*. During this period, the window size still increases exponentially, but at a reduced rate. Therefore, the aggressiveness of the congestion window's growth during the smooth-start period is reduced, thus reducing packet losses and dampening traffic burstiness. Smooth-start brings several benefits, even without the *ssthresh* estimator or when the estimated *ssthresh* is inaccurate or out of date.

- Smooth-start delays congestion because the window size approaches *ssthresh* more slowly. In the meantime, many packets can be transmitted. This delayed congestion benefits short-lived connections because they often finish transmission even before the congestion occurs.

- Smooth-start reduces the chance for buffer overflow at the intermediate routers. If intermediate routers run short of buffers, the slower output of packets may allow sufficient time for the routers to clear buffers.

- The reduced chance for buffer overflow also reduces the chance for multiple packet losses within the same window. Isolated packet losses can be efficiently handled by Fast Recovery [12, 24], avoiding retransmission timeouts.

- Smooth-start produces less bursty traffic than Slow-start, which reduces the fluctuation of the offered load on the networks.

Moreover, the implementation of Smooth-start is very simple and its overhead is very small. Only the sending side requires modifications, thus facilitating incremental deployment in today's Internet. Also, due to the inherent conservativeness of Smooth-start, no unfairness is introduced by Smooth-start. Since Smooth-start is triggered at the beginning of a connection or after a retransmission timeout, it benefits not only short-lived TCP transfers but also bulk TCP transfers if a congestion occurs.

The drawback of Smooth-start is that it takes longer for the transmission rate to reach the optimal equilibrium operating point. However, the number of extra round-trip times introduced by Smooth-start is small and deployment of a larger initial window or TCP Fast-Start, which increases the growth rate of the congestion window at the first round-trip time of Slow-start, compensates the slower growth rate of the congestion window in the smooth-start period.

The remainder of this paper is organized as follows. Related work is described in Section 2. Slow-start is briefly introduced in Section 3. Section 4 presents a detailed description of the Smooth-start algorithm and the possible probe strategies. Section 5 shows the simulation results with respect to the constant-load experiment and the changed-load experiment. Finally, Section 6 concludes the paper.

2 Related Work

Van Jacobson [11] proposed Slow-start and Congestion Avoidance algorithms for TCP congestion control. The current Implementations of TCP congestion control were later augmented with Fast Retransmit and Fast Recovery algorithms [12, 24].

A larger initial window [2] has been proposed to enhance the TCP performance, which raises the initial window size from 1 MSS to 4Kbytes. Evaluation of the larger initial window [3] shows that it decreases the transfer time of short-lived TCP over dialup channels and the Internet, and also the larger initial window does not significantly increase the number of retransmitted packets. To speed up Web transfers, TCP Fast-start [19] reuses cached network parameters in the recent past to shorten the startup of Slow-start, which greatly reduces transfer latency for short bursts. Also, by assigning a higher drop priority to Fast-start packets and augmenting the TCP loss recovery, TCP Fast-Start tries to avoid performance degradation when the cached information is stale.

An appropriate initial value of *ssthresh*, which dictates when to switch from Slow-start phase to Congestion Avoidance phase, is important to the performance of a TCP connection. Recognizing this importance, new approaches [10, 26] have been proposed to replace the default setting of *ssthresh* with an estimated or safe value of *ssthresh*. In [10], the method of estimation is similar to the *packet-pair* technique proposed by Keshav [13]. In [26], a new variant of SPAND (Shared PAssive Network Discovery [22]) has been presented to extract the current network condition, and based on the extraction to derive optimal initial TCP parameters.

A modified Slow-start algorithm is introduced in TCP Vegas [6], which limits the exponential growth of Slow-start to every other round-trip time, not every round-trip time. However, the problem is that multiple packet losses from the same window may happen during the exponential growth phase in the modified Slow-start. The key part of TCP Vegas [6] is the modified congestion avoidance algorithm, which allows more efficient network bandwidth utilization and higher network throughput. Instead of constant linear growth of the window during the congestion avoidance phase, TCP Vegas adopts three different strategies depending on the round-trip delay: increment, decrement and unchanged. Using a live emulation, Ahn *et al.* [1] evaluated the performance of TCP Vegas and confirmed the claims about TCP Vegas made in [6].

A recent study [20] shows that packet loss rate on the Internet has doubled within a year and burst loss of packets is common. To solve the problem of multiple packet losses from the same window, Selective Acknowledgment (SACK) [17] has been proposed. Using selective acknowledgments, when non-contiguous data is received by a receiver, duplicate ACKs bearing SACK options inform the sender about the packets that have been correctly received. Multiple packet losses per window are recovered in a round-trip time. Fall and Floyd [7] show the benefits of SACKs by simulation.

Through trace analyses, it is found that over 85% of retransmission timeouts are due to small congestion windows that prevent Fast Retransmit and Fast Recovery from being triggered [4, 14]. Based on the packet conservation rule [11], an enhancement of Fast Recovery is made to resolve the small congestion window problem. Also an integrated approach to congestion control and loss recovery is proposed in [4] to address the problem of multiple concurrent connections from a single host. Lin and Kung [14] also presented a loss-sensitive window reduction mechanism.

3 Slow-Start

In the original TCP specification [21], the window used by the sender, denoted as *wnd*, is equal to the receiver advertised window *rwnd* regardless of the load in the network. However, in the TCP congestion control schemes initiated by Van Jacobson [11], the TCP window size *wnd* is set to the minimum of the congestion window and the receiver advertised window. The congestion window *cwnd* is adjusted dynamically in response to network congestion.

$$wnd \;=\; min(cwnd, rwnd)$$

The TCP congestion control algorithm runs in two phases: Slow-start and Congestion Avoidance. When *cwnd* is smaller than *ssthresh*, the algorithm is in the Slow-start phase. Every received acknowledgment increments *cwnd* by 1. During this phase, *cwnd* essentially increases exponentially at every round-trip time.

After *cwnd* reaching *ssthresh*, the congestion control algorithm is in the congestion avoidance phase. Every received acknowledgment increments *cwnd* by $1/cwnd$. During this phase, *cwnd* essentially increases linearly at every round-trip time.

The window growth rates of a TCP connection in the Slow-start and Congestion Avoidance phases are described as follows, where W represents the congestion window size and RTT represents the Round-Trip Time.

- Slow-start phase: $\frac{dW}{dt} = \frac{W}{RTT}$ (exponentially increase)
- Congestion Avoidance phase: $\frac{dW}{dt} = \frac{1}{RTT}$ (linearly increase)

If the TCP receiver acknowledges each received data packet and no congestion occurs during the Slow-start phase, the amount of time required for *cwnd* to reach *ssthresh* is as follows:

$$Slow-start\ Time \;=\; RTT \log_2 ssthresh$$

The fundamental problem in the Slow-start algorithm is that when the congestion window size closely approaches to the equilibrium of the connection, the increment of the congestion window size is too large, leading to packet loss. At the beginning of the transmission, since the network is empty, the exponential increase of congestion window size is reasonable and also necessary to fill the empty pipeline quickly. However, as *cwnd* gets close to the equilibrium of the connection (unfortunately the sender does not know this until packet loss occurs), the scheme of *one ACK causing one increment of congestion window size* becomes too aggressive. It makes the congestion window size much larger than the actual bandwidth-delay product at the next round-trip time, often causing multiple losses of packets from a window of data. Figure 1 illustrates the dynamics of the congestion window size if no congestion occurs. Figure 2 illustrates this fact in another viewpoint.

In Slow-start algorithm, when the congestion window size is equal to half of *ssthresh*, it will reach the value of *ssthresh* at the next round-trip time. So, intuitively it is reasonable to slow down the rate of increase of the congestion window size at this point, letting the sender spend a little more time to reach *ssthresh* while approaching the actual equilibrium of the connection.

4 Smooth-Start

The proposed variant of Slow-start does not make a more accurate estimation or dynamically adjust the estimation along with the changes. Rather,the variant proposes a more graceful rampup of congestion window size in the Slow-start phase, so that packet loss is significantly reduced.

4.1 Smooth-start: A New Variant of Slow-start

As Figure 2 shown, the problem with Slow-start is that *cwnd* takes i RTTs to reach $\frac{ssthresh}{2^{n-i}}$ from 1, but only 1 RTT to reach $\frac{ssthresh}{2^{n-(i+1)}}$ from $\frac{ssthresh}{2^{n-i}}$. This acceleration at the end can cause congestion, and packet loss. Here, we assume that *cwnd* needs n RTTs to reach *ssthresh* from 1.

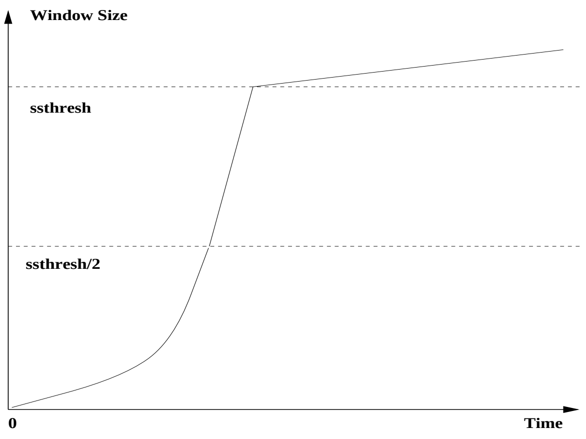


Figure 1: Illustration of the Slow-start Algorithm: The congestion window size grows exponentially in the Slow-start phase (up to ssthresh), and then grows linearly in the Congestion Avoidance phase.

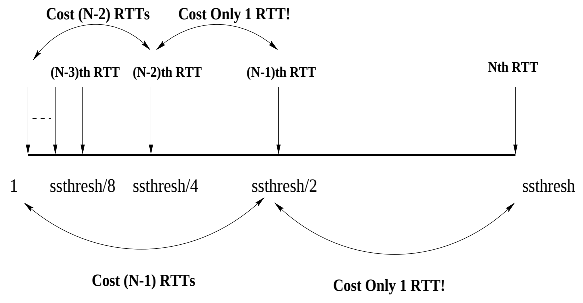


Figure 2: Illustration of the Slow-start Algorithm in Another Viewpoint

The key idea of Smooth-start is to add necessary lag points between $\frac{ssthresh}{2^{n-i}}$ and $\frac{ssthresh}{2^{n-(i+1)}}$ to dampen the effect of exponentially increase. In Smooth-start, there are two sub-phases divided by a chosen separator *smsthresh* (in Figure 3, *ssthresh/2* is chosen as a separator). In the first sub-phase, smooth start behaves the same way as slow start. Thus we call this period the *Slow-start* period or *pouring sub-phase.* In the second sub-phase, called the *Smooth-start* period or *probing sub-phase*, *cwnd* is incremented only upon the receipt of multiple ACKs (i.e., two or more). So, at every round-trip time *cwnd* increases by a factor of 1.5 or less. The number of round-trip times needed for *cwnd* to reach from $\frac{ssthresh}{2^{n-i}}$ to $\frac{ssthresh}{2^{n-(i+1)}}$ in the probing sub phase depends on our choice of the increasing gradient of the congestion window size. Two types of increasing gradient are introduced: *coarse-grained* and *fine-grained.*

4.1.1 Coarse-grained Probe

In the case of the coarse-grained probe, one *lag point* is inserted between $\frac{ssthresh}{2^{n-i}}$ and $\frac{ssthresh}{2^{n-(i+1)}}$, where a lag point represents an additional round-trip interval. In the first round-trip time after congestion window size reaching $\frac{ssthresh}{2^{n-i}}$, two ACKs trigger one increment of *cwnd.* In the second round-trip time, three ACKs trigger one increment of *cwnd.* Then two RTTs are needed to increase *cwnd* from $\frac{ssthresh}{2^{n-i}}$ to $\frac{ssthresh}{2^{n-(i+1)}}$. The detailed calculation is shown below.
At the end of the first RTT in the Smooth-start sub-phase,

$$CWND = \frac{ssthresh}{2^{n-i}} + \frac{1}{2} \times \frac{ssthresh}{2^{n-i}} = \frac{3 \cdot ssthresh}{2^{n-i+1}}$$

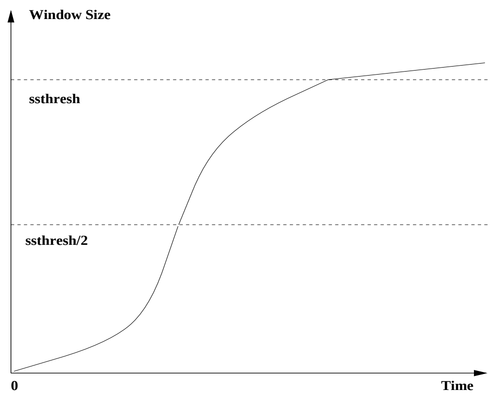


Figure 3: Illustration of the Smooth-start Algorithm: The congestion window size grows exponentially in the Slow-start sub-phase (up to *ssthresh*/2), at a slower exponential growth rate in the probing sub-phase, and then linearly in the congestion avoidance phase.

At the end of the second RTT,

$$CWND = \frac{3 \cdot ssthresh}{2^{n-i+1}} + \frac{1}{3} \times \frac{3 \cdot ssthresh}{2^{n-i+1}} = \frac{ssthresh}{2^{n-(i+1)}}$$

Therefore, two RTTs are required to increase *cwnd* from $\frac{ssthresh}{2^{n-i}}$ to $\frac{ssthresh}{2^{n-(i+1)}}$.

4.1.2 Fine-grained Probe

In the case of the fine-grained probe, two lag points are inserted between $\frac{ssthresh}{2^{n-i}}$ and $\frac{ssthresh}{2^{n-(i+1)}}$. In the first RTT after *cwnd* reaching $\frac{ssthresh}{2^{n-i}}$, three ACKs trigger one increment of *cwnd*. In the second RTT, four ACKs trigger one increment of the congestion window size, and in the third RTT, five ACKs trigger one increment of the congestion window size. The detailed calculation is as follows: At the end of the first RTT,

$$CWND = \frac{ssthresh}{2^{n-i}} + \frac{1}{3} \times \frac{ssthresh}{2^{n-i}} = \frac{4 \cdot ssthresh}{3 \times 2^{n-i}}$$

At the end of the second RTT,

$$CWND = \frac{4 \cdot ssthresh}{3 \times 2^{n-i}} + \frac{1}{4} \times \frac{4 \cdot ssthresh}{3 \times 2^{n-i}} = \frac{5 \cdot ssthresh}{3 \times 2^{n-i}}$$

At the end of the third RTT,

$$CWND = \frac{5 \cdot ssthresh}{3 \times 2^{n-i}} + \frac{1}{5} \times \frac{5 \cdot ssthresh}{3 \times 2^{n-i}} = \frac{ssthresh}{2^{n-(i+1)}}$$

Therefore, totally three RTTs are required to increase the congestion window size from $\frac{ssthresh}{2^{n-i}}$ to $\frac{ssthresh}{2^{n-(i+1)}}$.

In general, if we start with k ACKs triggering one increment of the congestion window size at the first RTT, and $(k+i)$ ACKs triggering one increment of *cwnd* at the following i-th RTT, we need k RTTs to increase *cwnd* from $\frac{ssthresh}{2^{n-i}}$ to $\frac{ssthresh}{2^{n-(i+1)}}$. We call the number k as *grain number*. The number of lag points inserted between $\frac{ssthresh}{2^{n-i}}$ and $\frac{ssthresh}{2^{n-(i+1)}}$ is $k-1$. As an analogy, we consider the probing as controlling the "accelerator pedal", and determining how quickly you "ease off" as you approach the desired "target velocity". In the case of coarse-grained or fine-grained probe that is studied in the rest of the paper, the probe number is set to 2 or 3 respectively.

4.2 Possible Separator

The general form of the separator is $ssthresh/2^d$, which separates the *slow-start* period from *smooth-start* period. The number d is called *depth number*. The two candidates in our consideration are: $ssthresh/2$ and $ssthresh/4$, in which the depth number d is set to 1 and 2. We do not consider $d \geq 3$ because in these cases the separator is too small. In the current implementations of TCP Slow-start, the default value of *ssthresh* is set to 64Kbytes. If we choose $d = 3$, *smsthresh* becomes 8Kbytes. Considering the proposed 4Kbytes initial window size, the slow-start period will only last for 1 RTT. In many cases except for heavy congestion, this would degrade the performance of a TCP connection since it would take too long to get to the optimal operating point.

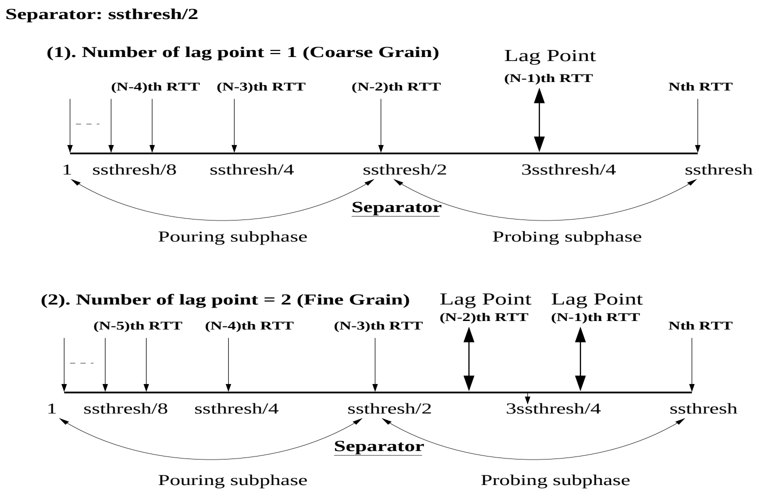


Figure 4: Separator is ssthresh/2

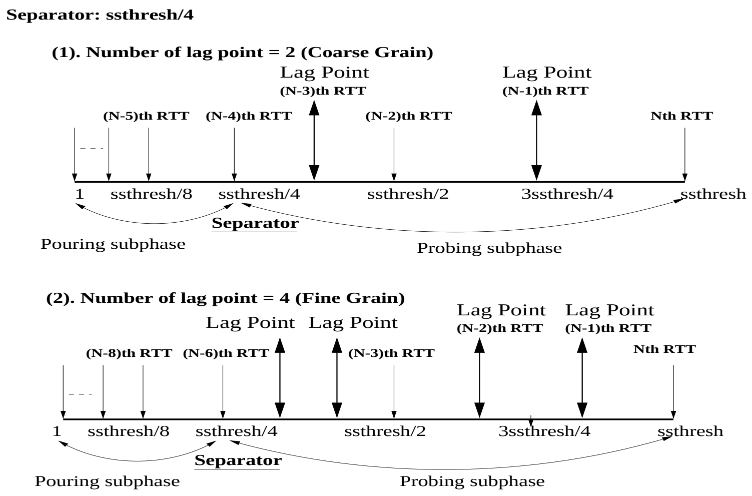


Figure 5: Separator is ssthresh/4

The rule for inserting lag points is illustrated in Figures 4 and 5 in the cases of the separator being ssthresh/2 and ssthresh/4, respectively. Each figure also contains two graphs representing the rules for the coarse-grained and fine-grained probing strategies respectively.

4.3 Overhead of the Smooth-start

The grain number k and the depth number d are two important parameters introduced by the Smooth-start algorithm. Both of them are non-negative integers. The conservativeness of Smooth-start is determined by theses two. The grain number k controls the probe gradient, and the depth number d controls the probe depth. As the increment of k or d, the conservativeness of Smooth-start increases.

The overhead of the Smooth-start depends on the probe strategy being taken, which is determined by the grain number k and the depth number d. In general, if the separator is set to $\frac{ssthresh}{2^t}$ and the grain number is taken as k, then the total number of extra RTTs introduced by Smooth-start $ExtraRTTs$ is:

$$ExtraRTTs = d \times (k-1)$$

Note if k is set to 1 and d is set to 0, then Smooth-start is converted to Slow-start. To some extent, Slow-start can be viewed as a special case of Smooth-start. In this paper, the grain number k is limited to $[2, 3]$ and the depth number d is limited to the range of $[1, 2]$.

Since only two choices of separator number and two kinds of probe number are considered in this paper, the total number of the studied probe strategies is 4. The number in Table 1 indicates how many lag points are inserted for each case. The number of lag points corresponds to the number of extra RTTs introduced by Smooth-start for *cwnd* to reach the *ssthresh*.

Separator	Coarse-grained Probe	Fine-grained Probe
ssthresh/2	1	2
ssthresh/4	2	4

Table 1: Number of Lag Points Inserted By the Probing Strategies

5 Simulations for Smooth-start

5.1 Simulation Setup

We tested the Smooth-start algorithm by implementing it in the LBNL network simulator *ns* [18]. The simulation network topology is shown in Figure 6, where S_i represents a sending host and K_i represents a receiving host. $R1$ and $R2$ represent two finite-buffer gateways. The links are labeled with their bandwidth and one-way propagation delay. Different traffic connections (from S_i to K_i) share a common bottleneck of 1.5 Mbps. In our simulation experiments each data packet size is 1024 bytes. To simplify the presentation of the paper, we assume that all window sizes are measured in units of fixed size packets, instead of bytes.

At the constant-load experiment, drop-tail gateways with FIFO service are used. However, at the changed-load experiment, RED gateways [9, 15] replace drop-tail gateways. The buffer size at each gateway R_i is set to 25 packets in each experiment. The type of data traffic in our simulation is FTP. The receiver sends an ACK for every data packet received.

Although we have modified different versions of TCP like Tahoe, Reno, New-Reno and SACK to have the Smooth-start, only the simulation results of TCP Reno are shown in this paper. The reason is that the key difference among various TCP versions lies in their individual packet loss recovery mechanism. All TCP versions employ Slow-start at the start of a TCP connection or after a retransmission timeout, thereby they have similar behaviors during the Slow-start phase. Another reason is that TCP Reno is built on UNIX BSD4.3, which is the most widely used TCP version.

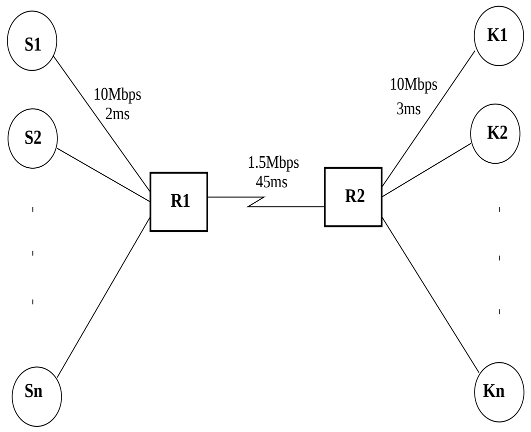


Figure 6: Simulation Topology Used for Smooth-Start Experiments

In the later sections that show the simulation results, *Coarse* stands for the coarse-grained probe strategy, and *Fine* stands for the fine-grained probe strategy. The number 2 or 4 indicates the chosen separator is *ssthresh/2* or *ssthresh/4* respectively.

5.2 Constant-Load Experiment with Drop-tail Gateways

In the constant-load experiment, the cross-traffic is constant and the TCP connection we measured has the fixed share of 1.0Mbps and 8 buffer unit. Therefore, the bandwidth-delay product of each TCP connection is:

$$BWP = 1.0Mbps \times 2(45ms + 5ms) = 100000\ bits = 125000\ bytes$$

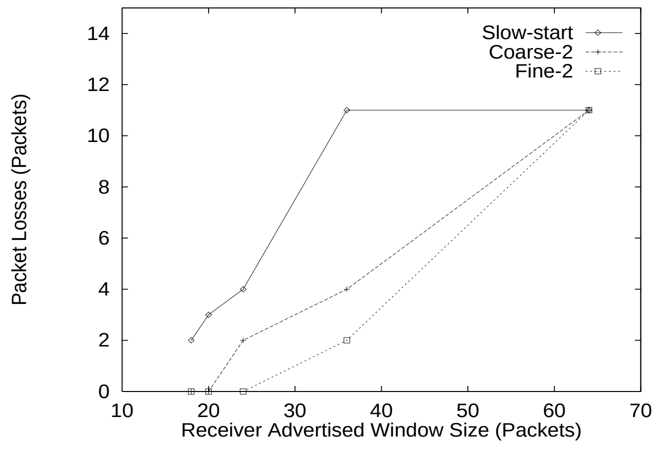
Since each data packet size is set to 1024 bytes, the actual bandwidth-delay product is approximately 12 packets. The total number of packets that can be in flight is 20, plus the buffer that is used by the TCP connection[1]. The file size transferred by this TCP connection is 60 Kbytes.

We vary the receiver advertised window size to be 18, 20, 24, 36 and 64. Because TCP sets *ssthresh* to be the minimum of the default value and the receiver advertised window size, varying the receiver advertised window size effectively varies the ratio of *ssthresh* over the actual pipe size. This categorizes the simulation into five different cases: (1) *ssthresh* is smaller than the actual pipe size; (2) *ssthresh* is equal to the actual pipe size; (3) *ssthresh* is larger than the actual pipe size; (4) *ssthresh* is much larger than the actual pipe size; (5) *ssthresh* is the default setting, which is more than three times larger than the actual pipe size. The case where TCP window size is much smaller than the actual pipe, so that no congestion occurs, is discussed in later section 5.5.

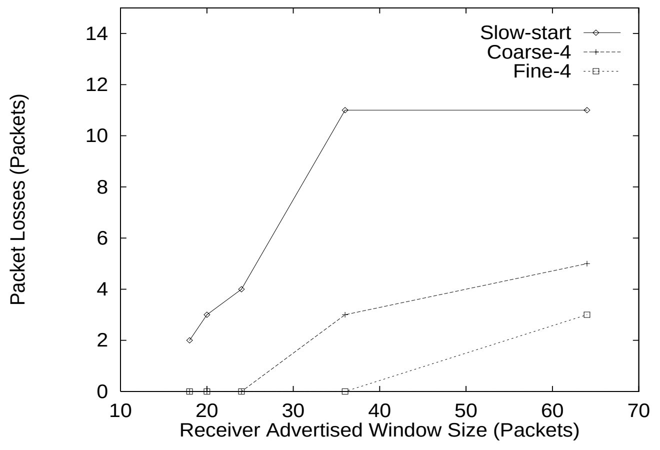
5.2.1 Simulation Results

Besides the number of packet losses, we also employ the effective throughput to demonstrate the superiority of Smooth-start to Slow-start. Effective throughput is a commonly-used metric of end-to-end protocol performance, which measures the actual amount of "good" data transmitted over the network, divided by the total elapsed time for the transfer of the data. The "good" data does not include retransmissions, lost segments, or duplicate data.

[1]It indicates the pipe size of the TCP connection is 20.



(a)



(b)

Figure 7: Packet Losses (a) Separator = ssthresh/2; (b) Separator = ssthresh/4;

Figure 7 presents the packet losses of a TCP connection using Slow-start and Smooth-start. Figure 7 (a) shows the reduced packet losses when the separator is *ssthresh/2*, while (b) shows the reduced packet losses when the separator is *ssthresh/4*.

Clearly, Smooth-start significantly reduces packet losses in all cases when the separator is *ssthresh/4* and in most cases when the separator is *ssthresh/2*. It is not surprising to see that if the separator of Smooth-start is chosen as *ssthresh/2* and the *ssthresh* is set to default value[2], the number of packet losses in Smooth-start is the same as that in Slow-start, because the separator *ssthresh/2* is larger than the actual pipe size. Thus, before *cwnd* reaches *ssthresh/2* to begin the smooth-start period, multiple packets have already been dropped due to the buffer overflow in the slow-start period. This is the main reason why we advance the probing phase by choosing the separator as *ssthresh/4*. It also shows why an inaccurate initial *ssthresh* estimator is still more helpful than the default setting.

To clearly illustrate the dynamics of a TCP connection, in the following figures, the simulation results are presented using the standard technique of TCP sequence number plots. The graphs from the simulations were generated by tracing packets entering and departing router R1. For each figure, the x-axis represents the packet arrival or departure time in seconds. The y-axis represents the packet number. Packets are numbered starting with packet 0. Each packet arrival and departure is indicated by a square on the figure. Desirable TCP behavior is reflected by steep upward-slope lines with a constant slope, illustrating steady throughput at the full bottleneck capacity. Figure 8 shows the dynamics of the TCP connection when the *rwnd* is 18, 20 and 24, respectively. Figure 9 shows the TCP dynamics when the *rwnd* is 36 and 64.

These figures show that Smooth-start has fewer packet drops, fewer timeouts, and consistently higher throughput than Slow-start. Because TCP Reno has poor capability of recovering multiple packet losses, the transfer latency is also greatly reduced. However, reduction of the transfer latency depends on the loss recovery mechanism of different TCP versions. For TCP New-Reno, SACK and FACK, the reduction of transfer latency is not as significant as that of TCP Reno.

[2]It is three times larger than the actual pipe size.

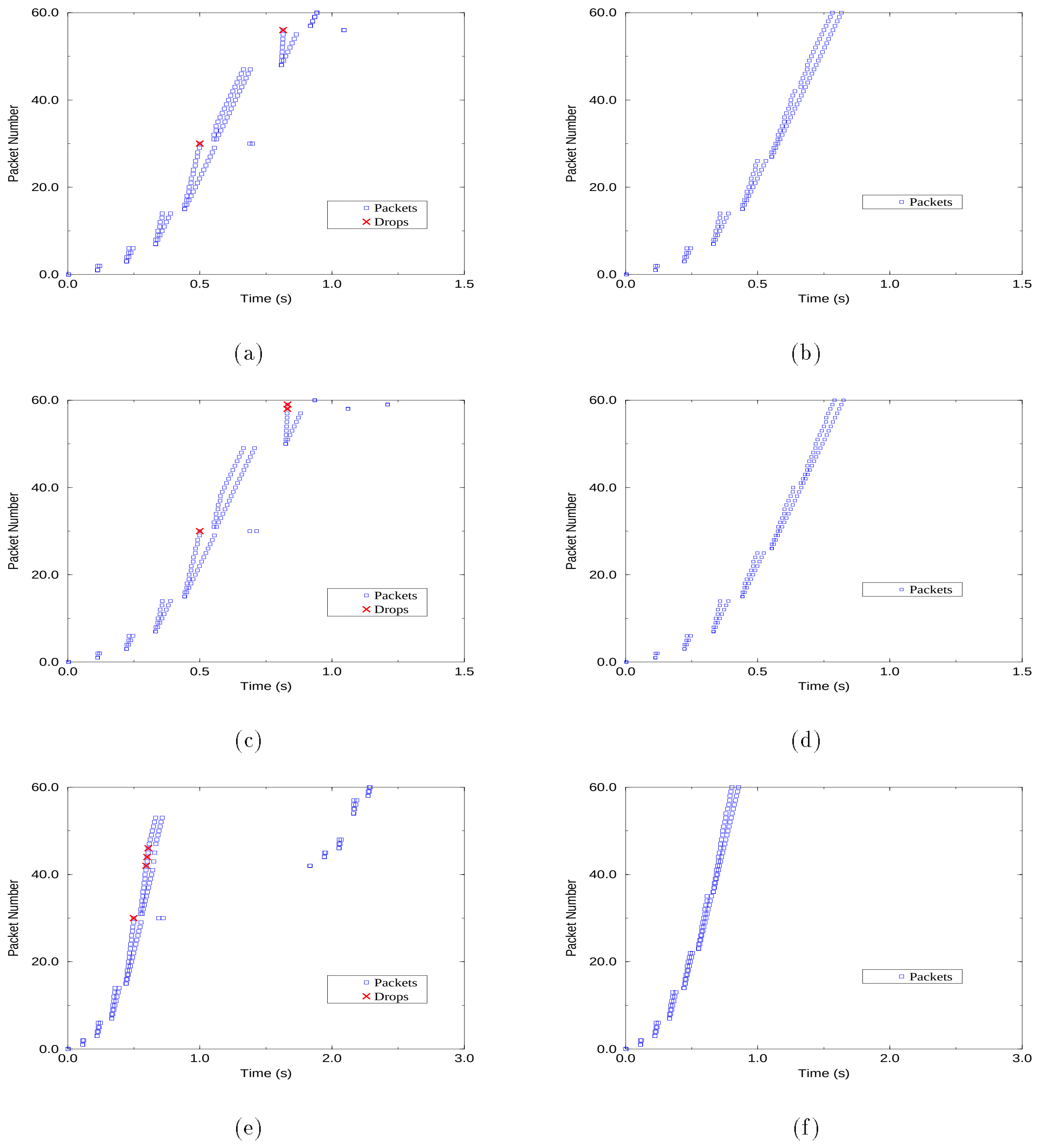
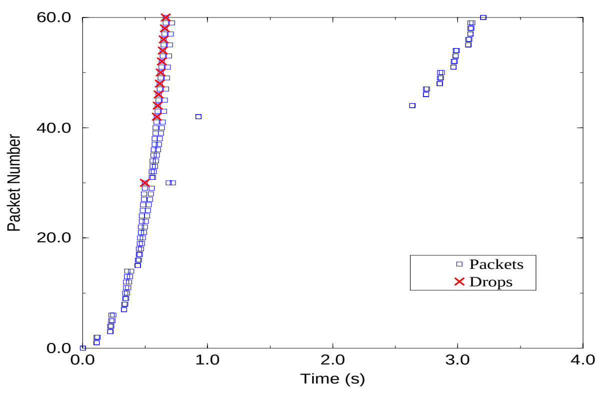
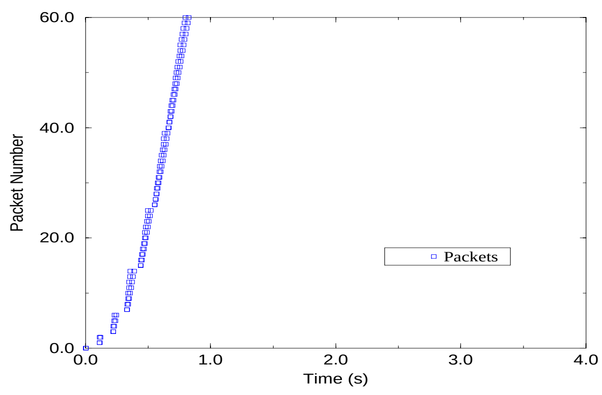


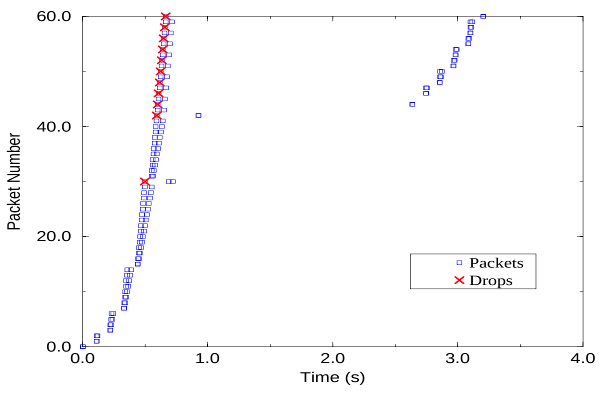
Figure 8: Simulation Results for TCP Reno with different *rwnd* and probe strategies: (a) Slow-start, rwnd = 18; (b) Coarse-2, rwnd = 18; (c) Slow-start, rwnd = 20; (d) Fine-2, rwnd = 20; (e) Slow-start, rwnd = 24; (f) Coarse-4, rwnd = 24.



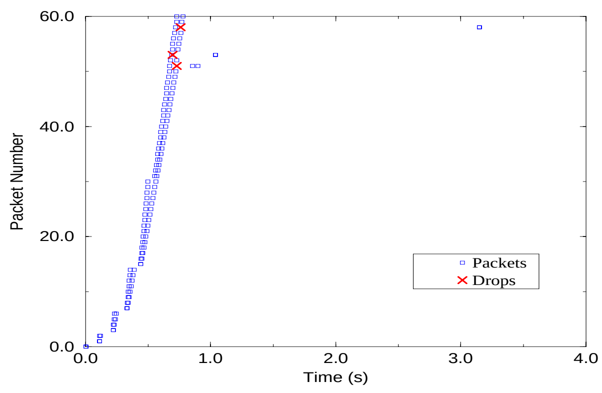
(a)



(b)

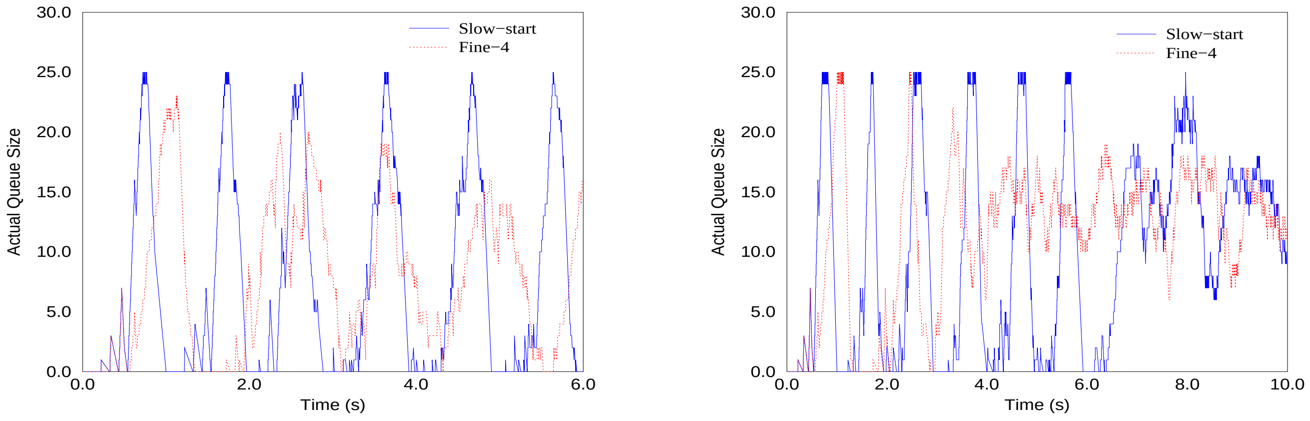


(c)



(d)

Figure 9: Simulation Results for TCP Reno with Fine-grained and separator is ssthresh/4: (a) Slow-start, rwnd = 36; (b) Fine-4, rwnd = 36; (c) Slow-start, rwnd = 64; (d) Fine-4, rwnd = 64.



(a) (b)

Figure 10: Dynamics of Actual Queue at the RED Gateway: (a) Without ECN; (b) With ECN.

5.2.2 Appropriate Probe Strategies and Inherent Advantages of Smooth-start

Based on the simulation results, *Fine-4* is the probe strategy that achieves much less packet losses and very good effective throughput improvements with various TCP receiver advertised window sizes.

Note when the initial *ssthresh* is closely set to the actual pipe size[3], Smooth-start makes significant TCP performance improvements over Slow-start. Even an accurate and timely *ssthresh* estimation cannot remove the whole performance degradation caused by Slow-start. The TCP performance degradation due to the inappropriate approach that Slow-start uses in reaching *ssthresh* can only be rectified by employing Smooth-start. An accurate estimation of initial *ssthresh* can significantly improve TCP performance, but is not a panacea to remedy all TCP start-up problems.

5.3 Changing-Load Experiment with RED Gateways

The setting of the changing-load experiment is similar to previous one, except for the fact that RED gateways replace Drop-tail gateways and the traffic load is periodically changed. Twelve TCP connections share the common bottleneck of 1.5 Mbps. The first connection starts at time 0.5. After that, every 0.5 seconds, we start a new TCP connection. The probe strategy of Smooth-start is *Fine-4* and the receiver advertised window size is 64 packets.

The simulation results are shown in two parts. The first part is about the queueing behaviors at the RED gateway, which indicates the degree of burstiness of the incoming TCP traffic. The second part of the simulation results show the traffic dynamics of one TCP connection which starts at 2.0s and sends 1 Mbytes of data.

The dynamics of the actual queue at the RED gateway are plotted in Figure 10. Clearly, Smooth-start greatly reduces the fluctuation of actual queue size and the frequency of the buffer overflow. By replacing Slow-start with Smooth-start, the burstiness of the incoming traffic is reduced, and hence, the pressure upon the buffer of the gateway is reduced.

Figure 11 shows the the traffic dynamics of the TCP connection, which starts at 2.0s. As expected, Smooth-start has consistently higher throughput, less packet losses, and less transfer

[3]In the case of 18, 20 and 24.

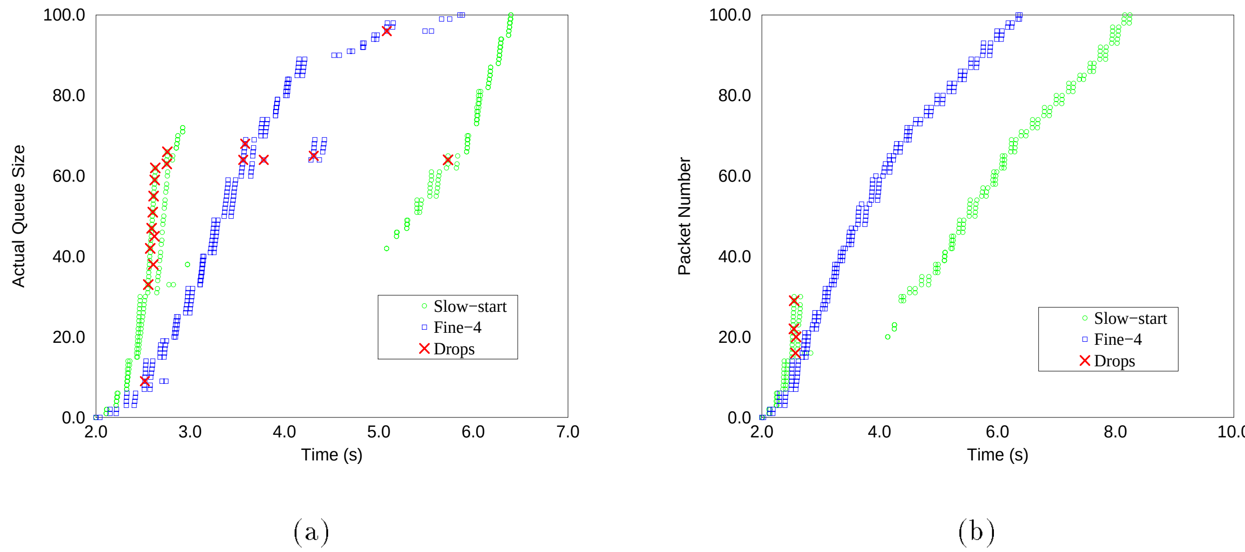


Figure 11: Illustration of the dynamics of a TCP connection (a) Without ECN; (b) With ECN.

latencies than Slow-start.

5.4 Integrating with *ssthresh* Estimator

As Smooth-start does not require an *ssthresh* estimator [10], the integration of Smooth-start and *ssthresh* estimator yields a better performance. The key point here is that the estimated *ssthresh* is no worse than, and is frequently better than, an arbitrary default. The simulation results are plotted in Figures 12 and 13. Note that the y-axis shows the packet number modulo 60.

In the same scenario where six packets are dropped, by employing Smooth-start, the number of packet losses is reduced to 2. By integrating Smooth-start with an *ssthresh* estimator, only one packet is dropped.

5.5 Fairness and Side-Effect of Smooth-start

The Smooth-start algorithm is employed at the sender side. It is less aggressive than Slow-start, getting to the equilibrium of a TCP connection in a more conservative way. The goal of Smooth-start is to achieve less packet losses and improve effective throughput of a TCP connection. Ensuring fairness among competing connections is beyond the capability of the Smooth-start algorithm. However, since Smooth-start is less aggressive than Slow-start, no more unfairness will be caused by Smooth-start. On the contrary, the flow becomes less bursty and the less bursty flow is more easily controlled during the start-up period.

If no congestion occurs, the performance of Smooth-start is worse than that of Slow-start. However, since the transmission rate still increases exponentially during the smooth-start period, the degradation of TCP performance is not significant. Moreover, the proposed larger initial window size and TCP Fast-Start at the first round-trip time of slow-start period compensates the slower growth rate of the congestion window in the smooth-start period. According to the evaluation report from [3], the total savings provided by the proposed initial window size is up to 3 RTTs if the receiver acknowledges every received packet. By integrating Smooth-start with larger initial

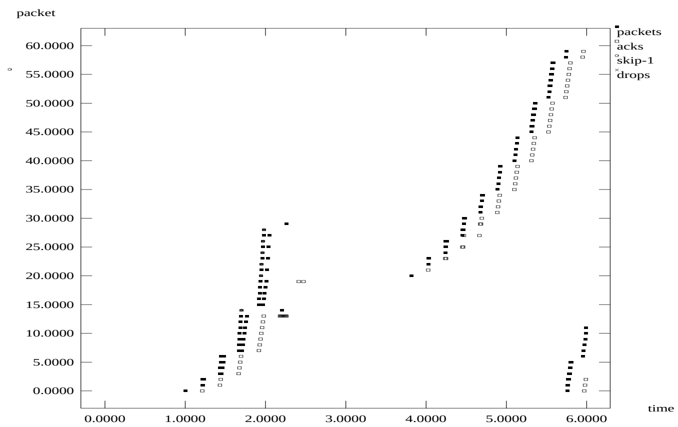


Figure 12: Simulation Result for TCP Reno with Slow-start.

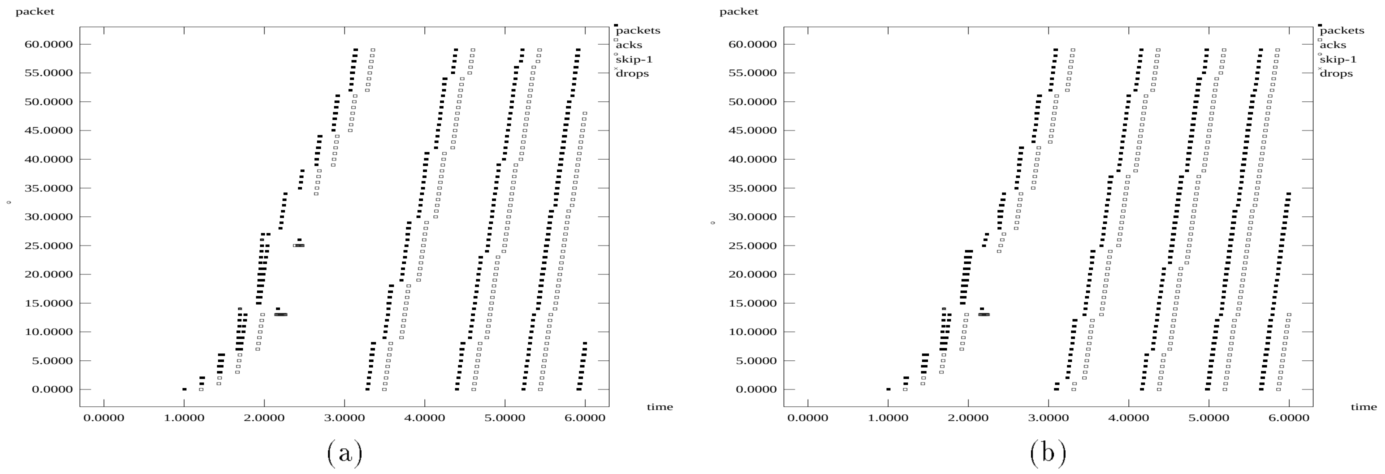


Figure 13: Simulation Results for TCP Reno (a) Smooth-start alone; (b) Smooth-start with *ssthresh* Estimator;

window, the amount of time required for *cwnd* to reach ssthresh is computed as follows:

$$Smooth-start\ Time \;=\; RTT\log_2 ssthresh \;+\; d\times(k-1)RTT \;-\; 3RTT$$

Therefore, in the case of probe strategy being *Fine-4*: Smooth-start takes only one more RTT to reach *ssthresh*; but for the rest of probe strategy considered in this paper: Smooth-start takes less RTTs to reach *ssthresh* if larger initial window is employed.

6 Conclusion

This paper proposes and evaluates a new variant of Slow-start called Smooth-start, to improve the TCP start-up performance. Smooth-start smoothes the transition between Slow-start and Congestion Avoidance. Two important parameters, grain number k and depth number d, are introduced by Smooth-start, which determine the conservativeness of Smooth-start. Several possible probe strategies of Smooth-start are evaluated through simulation. The inherent superiority of Smooth-start over Slow-start is shown in the simulation results. With the Smooth-start algorithm, the chance of having multiple packet losses from the same window of data is significantly reduced and effective throughput is greatly improved. Smooth-start can also be easily implemented and deployed since it requires only the sender side to be modified.

References

[1] J. Ahn, P. Danzig, Z. Liu, and L. Yan, "Evaluation of TCP Vegas: Emulation and Experiment", *Proceedings of ACM SIGCOMM'95*, Cambridge, MA, pp. 185-195, August 1995.

[2] M. Allman, S. Floyd, and C. Partridge, "Increasing TCP's Initial Window", *INTERNET DRAFT*, draft-floyd-incr-init-win-03.txt, April 1998.

[3] M. Allman, C. Hayes, and S. Ostermann, "An Evaluation of TCP with Larger Initial Windows", *ACM Computer Communication Review*, Vol. 28 No. 3, pp 41-52, July, 1998.

[4] H. Balakrishnan, V. Padmanabhan, S. Seshan, M. Stemm, R. Katz, "TCP Behavior of a Busy Internet Server: Analysis and Improvements", *Proceedings of IEEE INFOCOM'98.*

[5] B. Braden et al, "Recommendations on Queue Management and Congestion Avoidance in the Internet", *INTERNET DRAFT*, draft-irtf-e2e-queue-mgt-00.txt, March 1997.

[6] L. Brakmo, S. O'Malley, and L. Peterson, "TCP Vegas: New Technique for Congestion Detection and Avoidance", *Proceedings of ACM SIGCOMM'94*, London, UK, pp. 24-35, August 1994.

[7] K. Fall and S. Floyd, "Simulation-based Comparisons of Tahoe, Reno, and SACK TCP", *ACM Computer Communication Review*, Vol. 26, No. 3, pp. 5-21, July 1996.

[8] S. Floyd, and V. Jacobson, "On Traffic Phase Effects in Packet-Switched Gateways", *Internetworking: Research and Experience*, Vol. 3, No. 3, pp. 115-156, September 1992.

[9] S. Floyd and V. Jacobson, "Random Early Detection gateways for Congestion Avoidance", *IEEE/ACM Transactions on Networking*, Vol. 1, No. 4, pp. 397-413, August 1993.

[10] J. Hoe, "Improving the Start-up Behavior of a Congestion Control Scheme for TCP", *Proceeding of ACM SIGCOMM'96*, Stanford, CA, pp. 270-280, August 1996.

[11] V. Jacobson, "Congestion Avoidance and Control", *Proceedings of ACM SIGCOMM'88*, Stanford, CA, pp. 314-329, August 1988.

[12] V. Jacobson, "Berkeley TCP evolution from 4.3-tahoe to 4.3-reno", *Proceedings of the Eighteenth Internet Engineering Task Force*, pp. 365, 1990.

[13] S. Keshav, "A Control-Theoretic Approach to Flow Control", *Proceedings of ACM SIGCOMM'91*, Zurich, Switzerland, pp. 3-15, September 1991.

[14] D. Lin and H. T. Kung, "TCP Fast Recovery Strategies: Analysis and Improvements", *Proceedings of IEEE INFOCOM'98.*

[15] D. Lin and R. Morris, "Dynamics of Random Early Detection" *Proceedings of ACM SIGCOMM'97*, Cannes, France, pp. 127-137, September 1997.

[16] M. Mathis and J. Mahdavi, "Forward Acknowledgment (FACK): Refining TCP Congestion Control", *Proceedings of ACM SIGCOMM'96*, Stanford, CA, pp. 281-291, August 1996.

[17] M. Mathis, J. Mahdavi, S. Floyd, and A. Romanow, "TCP Selective Acknowledgment Option", Internet Draft, work in progress, May 1996.

[18] S. McCanne and S. Floyd, ns-LBNL Network Simulator. Obtain via: http://www-nrg.ee.lbl.gov/ns/.

[19] V. N. Padmanabhan and R. H. Katz, "TCP Fast Start: A Technique for Speeding Up Web Transfers", *Proceedings of IEEE GLOBECOM'98*, Sydney, Australia, November 1998.

[20] V. Paxson, "End-to-End Internet Packet Dynamics", *Proceedings of ACM SIGCOMM'97*, Cannes, France, pp. 139-152, September 1997.

[21] J. Postel, Transmission control protocol, Request for Comments 793, DDN Network Information Center, SRI International, September 1981.

[22] S. Seshan, M. Stemm, and R. H. Katz, "SPAND: Shared Passive Network Performance Discovery", *Proceedings of USITS'97*, Monterey, CA, December 1997.

[23] S. Shenker, L. Zhang, and D. D. Clark, "Some Observations on the Dynamics of a Congestion Control Algorithm", *ACM Computer Communication Review*, Vol. 20, No. 4, pp. 30-39, October 1990.

[24] W. R. Stevens, *TCP/IP Illustrated*, volume 1. Addison-Wesley Publishing Company, 1994.

[25] K. Thompson, G. J. Miller, and R. Wilder, "Wide-Area Internet Traffic Patterns and Characteristics", *IEEE Network*, Vol. 11, No. 6, pp. 10-23, November/December 1997.

[26] Yin Zhang, Lili Qiu, and S. Keshav, "Optimizing TCP Start-up Performance", Cornell CS Technical Report, February 1999.

[27] L. Zhang, S. Shenker, and D. D. Clark, "Observations on the Dynamics of a Congestion Control Algorithm: The Effects of Two Way Traffic", *Proceedings of ACM SIGCOMM'91*, Zurich, Switzerland, pp. 133-148, September 1991.