ABSTRACT

HA, SANGTAE. Improving TCP Congestion Control for High Bandwidth and Long Distance Networks. (Under the direction of Professor Injong Rhee).

Long distance networks spanning several continents are growing in importance, and many multi-national companies are now centralizing their data centers for economical reasons. While high performance of TCP in these networks is critical for the effective operation of their data centers, it is commonly reported that TCP substantially underutilizes network bandwidth in these environments. The performance problem of TCP in long distance networks can be largely attributed to three different components of TCP congestion control: (1) a slow increase of congestion window following a congestion event in congestion avoidance, (2) poor start-up throughput due to burst packet losses during slow start, and (3) high susceptibility to non-congestion related losses due to poor loss detection and recovery of standard TCP. While TCP congestion control was created to support an early, fledgling Internet, consisting of relatively low bandwidth and short distance networks, we show that TCP can embrace current technology trends of high-speed and long distance networks. Motivated by this aspiration, we explore solutions that address these challenges.

This dissertation proposes three practical solutions (one solution for each problem) for improving TCP performance in high bandwidth and long distance networks. These solutions are CUBIC, HyStart, and BLAST. They address orthogonal components of TCP congestion control, and thus they can be applied separately or in conjunction with each other. Specifically, CUBIC modifies the linear window growth function of existing TCP standards to be a cubic function in order to improve the scalability of TCP and also keeps the protocol TCP friendly. HyStart is a practical slow start algorithm that conforms to underlying layers and avoids immense system overload, which frequently results in end systems unresponsive for an extended period while recovering from burst packet losses; HyStart finds a "safe" exit point where slow start can finish and safely advance to the congestion avoidance phase without causing any heavy packet loss. BLAST makes loss-based TCPs more resilient in the face of non-congestion related losses by heuristically disambiguating non-congestion related losses with high accuracy. BLAST is integrated with the loss detection and recovery path in Linux and outperforms existing loss and delay-based TCPs by an order of magnitude in throughput.

The contributions of this dissertation are as follows. First, it departs from observing the performance issues regarding TCP of real end systems in high bandwidth and long distance networks. Specifically, we test TCP stacks on popular operating systems in our realistic experimental testbeds and recognize the functions of TCP that still require greater optimization. Second, we present three practical solutions that improve overall TCP performance in these environments. The proposed algorithms are designed with practical deployment in mind, and thus they are easy to understand and only require modification of the TCP sender. Finally, unlike most prior work, the proposed algorithms have been integrated into TCP stacks in Linux, and have undergone extensive testing in lab testbeds and also on the Internet. For example, CUBIC and HyStart have been integrated as a default congestion control algorithm in Linux, and BLAST will be used for providing loss resilience to Cisco's Wide Area Application Service (WAAS) WAN optimizers.

Improving TCP Congestion Control for High Bandwidth and Long Distance Networks

by
Sangtae Ha

A dissertation submitted to the Graduate Faculty of
North Carolina State University
in partial fullfillment of the
requirements for the Degree of
Doctor of Philosophy

Computer Science

Raleigh, North Carolina

2009

APPROVED BY:

Dr. Khaled Harfoush	Dr. Rudra Dutta
Dr. Injong Rhee Chair of Advisory Committee	Dr. George N. Rouskas

DEDICATION

To my wife Kyunghwa for her constant love and encouragement. To my children Harrie and Jihun, who taught me the joy of being a father.

BIOGRAPHY

Sangtae Ha was born in Busan, Korea. He received a B.E. in Computer Engineering, summa cum laude, from Kyung Hee University in 1999, M.S. in Computer and Communications Engineering, cum laude, from Pohang University of Science and Technology (POSTECH) in 2001. From 2004 to 2009, he pursued his Ph.D. degree in Computer Science from North Carolina State University, Raleigh, NC. Upon completion of his doctoral degree, he will be joining the Department of Electrical Engineering at Princeton University as a Postdoctoral Research Associate, working with Professor Mung Chiang. His research interests include congestion control, peer-to-peer networking, and wireless networks.

ACKNOWLEDGMENTS

I would like to express my deepest appreciation to my advisor, Dr. Injong Rhee, for giving me the opportunity to work with him over the last five years. I first met him 10 years ago when he visited Pohang University of Science and Technology (POSTECH) in Korea where I was an MS student. He sparkled my interest for further study while I took his Multimedia Networking course. After several years in industry, I joined the Department of Computer Science at North Carolina State University as a PhD student and began working with Dr. Rhee in 2004. I am deeply indebted to him for providing an opportunity to work on problems closely related to my interest and background. His insight and encouragement have been invaluable to me and have kept me motivated during my study.

I would also like to give many thanks to Dr. Lisong Xu for his advice and his early work in establishing a testing lab. I was very fortunate to have him as one of my research collaborators. Dr. Injong Rhee and Dr. Lisong Xu were undoubtedly the key mentors for my congestion control research.

I also would like to thank my dissertation committee, Dr. George N. Rouskas, Dr. Khaled Harfoush, and Dr. Rudra Dutta. I must also give special thanks to Dr. David Thuente, who is the Director of the Graduate Program. My Ph.D study would have been much more difficult without his dedication to graduate student growth and accomplishment. Also many thanks to Dr. Mladen A. Vouk, who is the head of the Computer Science Department and has also been a great teacher for the networking project course. I was fortunate to learn about how he could help students succeed in their projects and keep them motivated, while working as a teaching assistant for his course.

I would never have established my system-related career without the guidance of Dr. James Won-Ki Hong, who was my advisor during my MS studies. Working on several interesting projects related to network and embedded systems helped me to develop my experience in network systems. This also made me comfortable in implementing my research ideas on real networking systems.

Also, special thanks to Dr. Byungki Kim for his valuable consultation related to my research interests. I was also fortunate in having him as my first golf instructor in Australia, and he has been an active and effective mentor (he also officiated at Kyunghwa and my wedding).

I was very pleased to have interacted and collaborated with several researchers at Cisco Systems. Dr. Nandita Dukkipati, who is the inventor of RCP, was a great mentor when I worked at Cisco Systems as a Summer Intern. It was a special experience to work with one has a profound insight on the WAN optimization problems. Her insight has been an integral part of the BLAST idea presented in this dissertation. I also appreciate Dr. Vijay Subramanian's help on FEC. Special thanks to Dr. Flavio Bonomi for his warm welcome and support on this research.

I was also very fortunate to work with the WAAS (Wide Area Application Service) team at Cisco Systems. I appreciate Hamid Amirazizi, for his consideration and support in so many ways. Also thanks to my mentors in the WAAS team, Mani Ramasamy and Bharat Parekh, for providing interesting problems to solve.

Many thanks to my graduate student colleagues and friends for their critical thinking and their senses of humor: Hyungsuk, Seongik, Jeongki, Sungro, Kyuyong, Yaogong, Sankar, Kanishika, Puneet, Sanidhya, Min-Yeol, Young-Hee, Yonghee, Eunyoung, Young-June, and many other friends.

I am very grateful to John Sobrero, a member of Mensa, for his diligent proof-reading my dissertation.

Lastly, I would like to express my love and gratitude to my wife, Kyunghwa, for her constant love and encouragement throughout the last five years. Very special thanks to my parents and parents-in-law for being so supportive of my studies. Thanks to my two sisters and my brother-in-law.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

Chapter 1

Introduction

Long distance networks spanning several continents are growing in importance. Many multi-national companies are now centralizing their data centers for economical reasons. Thus, the high performance of Transmission Control Protocol (TCP) [87, 95] in these networks is critical for the effective operation of their data centers. These networks typically have large bandwidth and delay products (BDPs) ranging from several 1,000's to 100,000's. This is not a typical environment for which most existing commercial TCP stacks are optimized. Often we find that when these stacks run in such environments, they fall into some "rare" states where TCP obtains extremely low performance [42, 70, 68, 106].

One of most commonly cited problems of TCP in these networks is low utilization caused by its slow congestion window growth. TCP increases its congestion window by one for every round trip time (RTT) and reduces it by half at a loss event. For 10Gb/s and 100ms RTT networks, TCP requires over 83,333 RTTs (1.5 hours) to reach the BDP of the networks. For full utilization, the loss rate should be less than 1 loss event per $5 * 10^9$ packets which is less than the theoretical physical limit of today's networks. To counter the sluggish performance of TCP in these networks, many "high-speed" TCP variants, which adopt more scalable window growth functions (during congestion avoidance), are proposed. Among these protocols, BIC-TCP attracted many practitioners with its stability, and was accepted as the default TCP algorithm in Linux since 2004. BIC-TCP uses a binary search algorithm where the window grows to the midpoint between the last window size where TCP has a packet loss (Wmax) and the window size after its multiplicative drop (Wmin). While BIC-TCP has shown bandwidth utilization and stability, it still has room to improve

its implementation complexity, TCP-friendliness and RTT-fairness. TCP-friendliness is defined to be the fairness of a given TCP flow in sharing bandwidth with another TCP-NewReno or TCP-SACK flow over the same end-to-end path. RTT-fairness measures the ability of a protocol in sharing bandwidth fairly among flows of its own, but with different RTTs.

Another problem of TCP in these networks is poor start-up throughput due to burst packet losses at the initial stage of a connection. Precisely, this happens during slow start. Standard slow start doubles the congestion window of the sender for each corresponding acknowledgement (ACK) 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 a congestion window during slow start results in a large number of packet losses within one round-trip time, especially in a large BDP network. This can cause high loss synchronization among many competing flows, resulting in low utilization. Moreover, the long bursts of packet losses add a great deal of burden onto end systems for loss recovery and this burden often translates into consecutive timeouts and a long blackout period of no transmission. Even though there have been many proposals to fix the overshooting problem caused by slow start (e.g., Packet-Pair [63], Vegas' modified slow start [31], Limited slow start [43], Adaptive start [103], and Paced Start [64]), no proposals have been successfully used in real systems because they require either heavy modifications of TCP stacks or breaking layers other than TCP. Also, no proposals have systematically investigated "real" causes of poor startup throughput of slow start under large bandwidth delay product (BDP) networks.

Finally, TCP's throughput is very sensitive to packet losses, and so even low packet loss rates can cripple TCP's throughput in these large BDP networks. In steady-state, with a packet loss rate of p, TCP's throughput decreases as $\sqrt{p}$, placing a severe constraint on the achievable throughput in environments with natural losses [83, 46, 106]. To achieve reasonable performance with natural losses (e.g., wireless channel losses, physical losses, CRC checksum errors, etc.), making TCP respond only to congestion-related losses has grabbed the attention of many researchers. Biaz et al. [28] showed that the loss differentiation schemes using the gradients of either delay or throughput at the TCP sender are extremely poor congestion predictors whose accuracy is similar to that of a coin toss. By realizing this limitation, many proposals rely on the cooperation of networks such as

Explicit Congestion Notification (ECN), for disambiguating non-congestion related losses accurately. However ECN [88, 96, 44, 74] is not being widely used or deployed, and therefore this is not a viable solution for today's networks. For an incremental deployment, we need a viable solution with reasonable accuracy on the sender side. Applications requiring a high TCP throughput over lossy links such as Wide Area Network (WAN) optimizers and content delivery applications can also benefit from this work.

While the TCP congestion control algorithm was created to support an early, fledgling Internet more than two decades ago, consisting of relatively low bandwidth and short distance networks, we demonstrate that TCP remains suitable for current technology trends of high-speed and long distance networks if we augment the existing TCP congestion control algorithm to function in these new environments. This will help the Internet evolve seamlessly to embrace more challenging future network environments. By recognizing this aspiration, this dissertation proposes three practical solutions (one solution for each problem) for improving TCP performance in high bandwidth and long distance networks. These solutions are CUBIC [57], HyStart [55, 56], and BLAST [54].

CUBIC, the next version of BIC-TCP, is a congestion control protocol for TCP which is designed to fix the slow window growth of TCP and the current default TCP algorithm in Linux. It modifies the linear window growth function of existing TCP standards to be a cubic function in order to improve the scalability of TCP over fast and long distance networks. It also achieves more equitable bandwidth allocations among flows with different RTTs by making the window growth independent of RTT - thus, those flows grow their congestion window at the same rate. During steady state, CUBIC increases the window size aggressively when the window is far from the saturation point, and slowly when it is close to the saturation point. This feature allows CUBIC to be very scalable when the bandwidth and delay product of the network is large, and at the same time, highly stable and also fair to standard TCP flows. CUBIC improves on the weakness of its predecessor BIC-TCP as follows. CUBIC simplifies the congestion window control of BIC-TCP involving several window control phases (i.e., additive increase, binary search increase, and max probing), by introducing a cubic function for the window control in all these phases. This improves the tractability of a protocol and makes it easier to analyze the performance of the protocol. CUBIC enhances its TCP-friendliness by introducing a new "TCP mode", especially under short RTT or low speed networks, where BIC-TCP's growth function continues to be too

aggressive for TCP and thus steals more bandwidth from TCP. CUBIC improves RTT-fairness by making its window growth independent of RTTs. This makes two flows of different RTTs sharing the same bottleneck maintain the same window size (the same throughput) independent of their RTTs. Based on the feedback from users and researchers, the implementation of CUBIC in Linux has gone through several upgrades for improving scalability and stability. We present its design, implementation, performance and evolution as the default TCP algorithm in Linux.

HyStart is a new slow start algorithm for fast and long distance networks to remedy the poor startup throughput caused by a long burst of packet losses that often results in consecutive timeouts and a long blackout period. By looking at the implementation of Linux's TCP stack and systematically profiling the applicable functions in Linux TCP's stack, we show the "real" causes of poor startup for existing slow start and present HyStart for its solution. HyStart finds a "safe" exit point of slow start at which slow start can finish and safely advance to the congestion avoidance phase without causing heavy packet losses. It uses ACK trains and RTT delay samples to detect whether (1) the forward path is congested or (2) the current size of the congestion window has reached the available capacity of the forward path. It is a plug-in to the TCP sender and does not require any change in TCP receivers. We implemented HyStart for TCP-NewReno [47] and TCP-SACK [79] in Linux and compared its performance with five different slow start schemes on the TCP receivers of the three different operating systems on the Internet and also in lab testbeds. Our results indicate that HyStart works consistently well under diverse network environments, including asymmetric links and high and low BDP networks. Based on these encouraging results, HyStart has been integrated as a default slow start algorithm to CUBIC in Linux.

Finally, BLAST is an enhancement to TCP that makes loss-based TCPs more resilient in the face of non-congestion related losses. It distinguishes between congestion and non-congestion related losses, by computing the path queueing delay using round-trip time measurements. The two key points characterizing BLAST's heuristics are: (1) it uses the mean queuing delay which provides more robust information about the queue build-up in the bottleneck router, because it filters then averages the samples in each RTT round and thus reduces the effect of outlier values, and (2) it uses a "grey-zone" of queueing delays which contributes to detecting the trend in queueing delay changes. While the

idea of disambiguating losses is not new, the contribution of BLAST is twofold: (1) the precise technique of distinguishing congestion losses from non-congestion losses, and (2) integrating the technique with the mechanisms of TCP's congestion control to achieve high throughput in large BDP networks, even in the presence of non-congestion related losses. We implemented BLAST in Linux and evaluate its performance with other congestion control algorithms which use different congestion indicators, e.g., BIC-TCP (loss-based) [106], TCP-Vegas (delay-based) [31] and TCP-Westwood (rate-based) [36]. Our results show that BLAST achieves an order of magnitude higher throughput compared to existing TCPs with non-congestion related losses while performing as well as BIC-TCP with congestion related losses.

In this dissertation, we close the gap between theory and practice by implementing CUBIC, HyStart and BLAST in real Linux systems and testing them under realistic experimental settings. These solutions provide an answer to aforementioned three orthogonal problems in TCP congestion control and improve TCP performance in long distance networks. CUBIC modifies congestion avoidance, HyStart modifies slow start, and BLAST modifies loss detection and recovery of standard TCP. Being designed with practical deployment in mind, they only require modification on the TCP sender and their algorithm is easy to understand and implement. Furthermore, unlike most prior work, they have undergone extensive testing in lab testbeds and also on the Internet. For instance, CUBIC and HyStart have been integrated as a default congestion control algorithm in Linux, and BLAST will be used for providing loss resilience to Cisco's Wide Area Application Service (WAAS) WAN optimizers.

Our proposed solutions allow TCP to embrace current networking trends of high-speed and long distance networks seamlessly. TCP applications built upon the standard programming interface (e.g., socket interface) transparently benefit from this work. For instance, Internet users, who are geographically far apart and have heterogenous bandwidths, can exchange large content much faster while sharing the bandwidth resource fairly among other users, enjoy improved Web content loading times, and experience noteworthy improvements of their TCP application performance in high lossy networks.

1.1 How to Read This Dissertation

In this dissertation, we propose three practical solutions for improving TCP performance in high bandwidth and long distance networks. CUBIC modifies the congestion avoidance function, HyStart modifies the slow start function, and BLAST modifies loss detection and recovery of the standard TCP. Since these solutions address orthogonal components, they can be applied to TCP congestion control separately or in conjunction with each other.

In Chapter 2, we exhibit background information. First, we present a brief introduction of TCP congestion control (e.g., slow start, congestion avoidance, and fast retransmit/recovery) before presenting our work in great detail. Next, we derive the deterministic AIMD TCP throughput which serves as good background information for understanding CUBIC's steady-state throughput in Chapter 5.

Chapter 3 presents related work for improving TCP performance in high bandwidth and long distance networks. First, we survey some recently proposed TCP variants which modify the congestion avoidance phase of TCP congestion control. Second, we present existing slow start proposals which improve the start-up throughput of TCP. Finally, we describe the related work which improves the accuracy of loss detection and recovery of TCP.

Chapter 4 presents our detailed experimental network model used in this dissertation that captures complex and realistic characteristics of propagation delays and background traffic. We describe the setup of our laboratory network and discuss how we generate background traffic with realistic propagation delays in our experiment. We also explain our experimental procedure.

In Chapter 5, we describe the details of the Linux CUBIC algorithm and implementation. We present important updates to Linux CUBIC implementation since its first introduction into Linux. We also derive CUBIC's steady-state throughput with a deterministic model.

Chapter 6 shows the causes of poor start-up for existing slow start on three production operating systems (i.e., Linux, FreeBSD, and Windows) and describes many pitfalls and several enhancements. We also explain the detailed algorithm of our new TCP slow start, called HyStart and its extensive evaluation results on the Internet and also in

the lab testbed.

In Chapter 7, we present BLAST which makes loss-based TCPs more resilient in the face of non-congestion related losses. Specifically, we show BLAST heuristics used for disambiguating non-congestion related losses and their accuracy. We also present how BLAST is integrated with the loss detection and recovery path in Linux's TCP stack.

Finally, in Chapter 8, we conclude this dissertation and discuss our future work.

Chapter 2

Background

Congestion control is an important component of a transport protocol in a packet-switched shared network. The congestion control algorithm of the widely used transport protocol Transmission Control Protocol (TCP) [87, 21, 95] is responsible for detecting and reacting to overloads on the Internet and has been the key to the Internet's operational success. In this Chapter, we present a brief introduction of TCP congestion control before delving into great details of our enhancements. Section 2.1 includes general introduction of four main functions of TCP congestion control - slow start, congestion avoidance, fast retransmit and fast recovery. Section 2.2 derives the deterministic AIMD TCP throughput.

2.1 TCP Congestion Control

TCP plays a vital role for a reliable communication in today's Internet. TCP runs on the top of Internet Protocol (IP) [86, 95] and ensures a reliable delivery of data to an application. IP only provides a "best-effort" service, meaning that IP packets can be lost and delivered out of sequence. TCP addresses this limitation of IP and guarantees a reliable delivery of data by using a retransmission technique.

With the reliability of TCP, congestion control is one of main aspects of TCP. TCP requires achieving high performance while preventing "congestion collapse" [65, 21, 81, 41], where network performance drops by several orders of magnitude due to heavy packet losses. In order to avoid such a congestion collapse, TCP congestion control follows a "packet conservation" principle [65], which states that packets should not enter a network

until previously transmitted packets have been received and acknowledged or confirmed as having been lost. Since packets and their acknowledgements are in transit most of time, it is hard for a TCP sender to precisely count the number of packets currently in flight in the network. TCP makes a conservative assumption that all unacknowledged packets are still in the network, so that the number of packets outstanding in the network is always less than or equal to the estimated number of unacknowledged packets.

Precisely speaking, the sender controls its sending rate by using a variable, called "congestion window" (cwnd), which determines the number of packets[1] that the sender is allowed to send. The receiver also advertises to the sender the amount of data it willing to buffer for the connection. We call this "advertised window". By using these two controlling variables, the sender can transmit up to the minimum of the congestion window and the advertised window.

Figure 2.1 shows a typical saw-tooth behavior of TCP congestion control [79, 21, 47]. TCP congestion control has four main parts - slow start, congestion avoidance, fast retransmit and fast recovery. The slow start algorithm increases the congestion window exponentially to quickly find the equilibrium state of the network because TCP has no advance knowledge of the network. This usually happens during the initial stage of a connection and a restart of the connection after a long idle time. The congestion avoidance algorithm, on the other hand, controls the congestion window not too aggressively because the sender has already reached the equilibrium state of the network, and so drastic a change of the congestion window may break this equilibrium state. Upon receiving three duplicate ACKs, the fast retransmit algorithm responds to the loss event immediately by retransmitting what appears to be the missing packet without waiting for a retransmission timer to expire. After the fast retransmit algorithm, the fast recovery algorithm continues to work to maintain the same number of packets prior to entering into the fast recovery. The fast recovery algorithm terminates when the sender receives an ACK acknowledging new data.

2.1.1 TCP Slow Start

The slow start algorithm is used for probing unknown and time–varying available bandwidth of a network path. A sender increases its congestion window by one for each ACK

[1] Linux uses the number of packets and other systems use the amount of bytes for their congestion window.

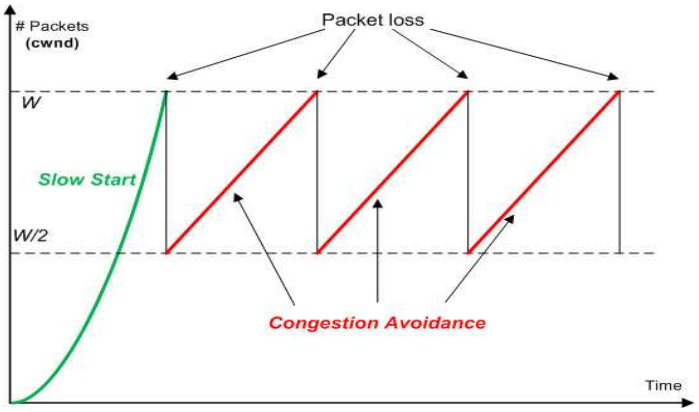


Figure 2.1: TCP's slow start and congestion avoidance.

received (when ACKs are not delayed), which effectively doubles its congestion window when receiving ACKs for all the packets in a congestion window. Eq. 2.1 shows the congestion window evolution during slow start upon receiving an ACK.

$$\text{ACK} \ : \ cwnd \longleftarrow cwnd + 1 \tag{2.1}$$

A receiver running different ACK schemes affects the ramp-up speed of slow start on the sender. Even though implementing a delayed ACK is mandatory for TCP end systems, some operating systems use a quick ACK for a specified period of time to improve their start-up performance. Windows and FreeBSD use a delayed ACK from the beginning of a connection, but Linux uses a quick ACK for the initial 16 packets to improve the start-up throughput for short-lived transfers such as Web traffic.

The sender exits slow start and enters into congestion avoidance when its cwnd becomes larger than a variable, called "slow start threshold" (ssthresh). The slow start threshold is a conservative measure of available bandwidth in the network. Thus, filling the network pipe very rapidly to the slow start threshold is a natural step and also important for performance.

2.1.2 TCP Congestion Avoidance

In the congestion avoidance phase, a TCP sender increases its cwnd by 1/cwnd for each incoming ACK[2]. This effectively makes the sender gradually increase its cwnd by 1 packet per every round-trip time (RTT). This makes sense because the sender has already reached the equilibrium state of the network. The TCP sender, on the other hand, reduces its cwnd by half upon detecting losses by receiving three duplicate ACKs. Compared to the slow start algorithm which runs when cwnd is less than ssthresh, the congestion avoidance algorithm governs when cwnd is greater than or equal to ssthresh. Eq. 2.2 shows the congestion window evolution during congestion avoidance upon receiving an ACK and upon detecting losses.

$$\begin{aligned} \text{ACK} &: cwnd \longleftarrow cwnd + \frac{1}{cwnd} \\ \text{Loss} &: cwnd \longleftarrow \frac{1}{2} \times cwnd \end{aligned} \tag{2.2}$$

A TCP sender uses packet loss as an indication of network congestion. When there is no network congestion (the sender receives cumulative ACKs), the sender increases its cwnd additively. When the sender detects network congestion by receiving three duplicate ACKs indicating packet loss, it drops cwnd by half of its current cwnd to reduce the congestion in the network. This process is called Additive Increase and Multiplicative Decrease (AIMD) [38, 21, 46]. Eq. 2.3 shows the general AIMD congestion avoidance algorithm. Note that the standard TCP is AIMD(1,1/2).

$$\begin{aligned} \text{ACK} &: cwnd \longleftarrow cwnd + \frac{\alpha}{cwnd} \\ \text{Loss} &: cwnd \longleftarrow (1 - \beta) \times cwnd \end{aligned} \tag{2.3}$$

TCP has been widely used as a ubiquitous transfer protocol on the Internet. However, as the Internet evolves to include networks with high bandwidth and long distance paths, the small additive increment of cwnd used in TCP congestion control was blamed for its poor performance on these networks. There have been many "advanced" TCP congestion

[2]TCP represents standard TCP which includes many TCP-Reno variants such as TCP-Reno [21], TCP-NewReno [47], and TCP-SACK [79].

control algorithms, adopting more scalable window growth functions for better performance in these networks. Most of these algorithms only modify the protocol behavior during the congestion avoidance phase. We explain these advanced TCP algorithms (we call them "TCP Variants") briefly in Section 3.1.

2.1.3 TCP Loss Recovery (Fast Retransmit and Fast Recovery)

To provide a reliable transmission between a sender and a receiver, TCP requires a mechanism to detect the loss of packets and retransmit those lost packets. TCP associates data bytes with a unique number, called a sequence number. By receiving the data sent from a sender, the receiver acknowledges the arrival of data to the sender by sending an ACK packet carrying the next in-order sequence number it expects to receive. We call this "cumulative" ACK. When the data packet is lost or arrives out-of-order at the receiver, the duplicate ACK packets carrying the same sequence number are delivered to the sender.

TCP maintains a retransmit timer and resets the timer whenever it receives a cumulative ACK. If the ACK packet doesn't arrive at the sender until the timer expires, TCP assumes unacknowledged packets are lost and retransmits those packets. Retransmitting lost packets after a retransmission timeout is very costly and inefficient because the retransmission timeout is longer than RTT and the sender needlessly waits for the timeout without transmitting any packets.

TCP fast retransmit [21, 47] reduces the time that a sender waits before retransmitting a lost packet. If a receiver receives packets out-of-order, it immediately sends a duplicate ACK to the sender. If a sender receives three duplicate ACKs, the sender assumes the packet indicated by duplicate ACKs is lost and immediately retransmits the lost packet to the receiver. The fast retransmit algorithm prevents a TCP sender from wasting its time while waiting for the timeout for retransmission.

After TCP fast retransmit sends the lost packet, TCP fast recovery [21, 47] makes TCP operate in congestion avoidance mode instead of slow start. This improves the performance significantly. Since the receipt of three duplicate ACKs not only means that a packet has been lost, but also indicates that there are still data flowing between two ends, maintaining the same rate while recovering from packet losses makes sense. Precisely speaking, upon receiving three duplicate ACKs, a TCP sender sets ssthresh to half of the current cwnd and resets cwnd to ssthresh plus three segments accounting for three

packets having left the network. For each duplicate ACK, the sender increases its cwnd by accounting for an additional segment that has left the network. The sender transmits packets if cwnd permits. When the sender receives a cumulative ACK that acknowledges new data, it sets cwnd to ssthresh and exits TCP fast recovery.

2.2 TCP Throughput Analysis

Padhye et al. [83] developed a stochastic model of TCP congestion control and presented an upper bound on TCP's sending rate T in bytes/sec, as a function of the packet size s, round-trip time R, steady-state loss event rate p, and TCP retransmit timeout t_{RTO}.

$$T = \frac{s}{R\sqrt{\frac{2p}{2}} + t_{RTO}\left(3\sqrt{\frac{3p}{8}}\right)p(1+32p^2)} \tag{2.4}$$

Floyd et al. [46] presented a deterministic AIMD model which is useful for identifying the role of the increase and decrease factors (α and β) in AIMD congestion control. In this section, we also derive a deterministic AIMD model with greater explanation. This analysis serve as good background information for understanding CUBIC's steady-state throughput shown in Section 5.3.1.

Figure 2.2 depicts AIMD protocol's congestion window in steady-state. The following is the notation we used for this analysis.

- W: The window size immediately before a loss event.
- β: The multiplicative decrease factor. After a loss event, the window size is reduced to $(1-\beta)W$.
- RTT: The RTT of a flow.
- p: Loss event rate.

The total number of packets sent in a loss epoch is

$$\frac{1}{p} = A = \left(W\beta\frac{W\beta}{\alpha}\right)\frac{1}{2} + \left(W(1-\beta)\frac{W\beta}{\alpha}\right) = \frac{W^2(2-\beta)\beta}{2\alpha} \tag{2.5}$$

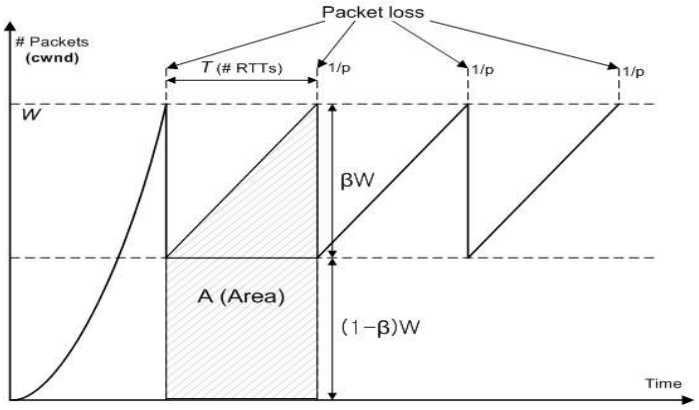


Figure 2.2: AIMD's congestion window in steady-state.

By solving Eq. 2.5 for W, we obtain the following:

$$W = \frac{\sqrt{2\alpha}}{\sqrt{\beta(2-\beta)p}} \tag{2.6}$$

The number of RTTs during one loss epoch is

$$T = \frac{W\beta}{\alpha} RTT \tag{2.7}$$

Thus, the average window size of AIMD protocol $\mathbb{E}\{W_{aimd}\}$ can be calculated by dividing the total number of packets sent by the number of RTTs for one loss epoch.

$$\mathbb{E}\{W_{aimd}\} = \frac{A}{T} = \frac{\frac{W^2(2-\beta)\beta}{2\alpha}}{\frac{W\beta}{\alpha} RTT} = \frac{\sqrt{2-\beta}\sqrt{\alpha}}{\sqrt{2\beta} RTT \sqrt{p}} \tag{2.8}$$

By substituting α and β with 1 and 1/2 respectively, we obtain standard TCP's throughput.

$$\mathbb{E}\{W_{tcp}\} = \frac{1.2}{RTT\sqrt{p}} \tag{2.9}$$

Chapter 3

Related Work

This chapter presents related work for improving TCP performance in high bandwidth and long distance networks. These proposals address different functions of TCP congestion control - congestion avoidance, slow start, and loss detection and recovery of TCP. Section 3.1 surveys some recently proposed TCP variants which modify the congestion avoidance phase of TCP congestion control. Section 3.2 presents existing slow start proposals which prevent burst losses during slow start and thus improve the start-up throughput of TCP. In Section 3.3, we describe the proposals which improve the accuracy of loss detection and recovery of TCP.

3.1 TCP Variants

3.1.1 Scalable TCP (SCTP)

Kelly proposed Scalable TCP (STCP) [70]. The design objective of STCP is to make the recovery time from loss events constant regardless of the window size. This is why it is called "Scalable". Note that the recovery time of TCP-NewReno largely depends on the current window size. For instance, suppose that the available bandwidth of the path is 10Gb/s, RTT of the path is 100ms, and the packet size is 1250 Bytes. TCP-NewReno detects a loss when the window size is around 100,000 packets and reduces the window to 50,000 packets. TCP-NewReno requires 50,000 RTTs (1.4 hour) to fully utilize the link again after a loss while STCP requires around 70 RTTs (7 seconds) for recovery from the same situation. Precisely, STCP increases a congestion window 1% per each incoming ACK,

and drops 12.5% from the current congestion window size. Therefore, α and β are 0.01 and 0.125 respectively.

$$\begin{aligned} \text{ACK} &: cwnd \longleftarrow cwnd + \alpha \\ \text{Loss} &: cwnd \longleftarrow (1 - \beta) \times cwnd \end{aligned} \tag{3.1}$$

Figure 3.1 presents the behavior of two STCP flows in the network with 100Mb/s bandwidth and 160ms RTT.

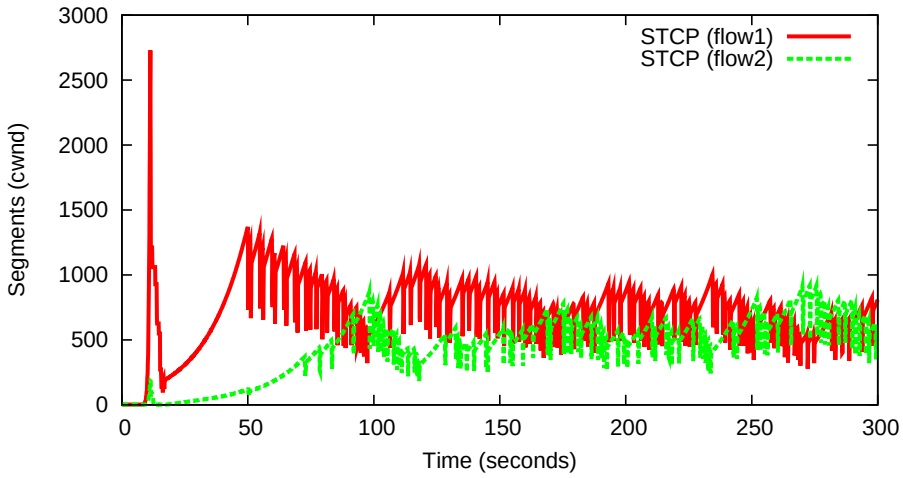


Figure 3.1: CWND evolutions of two STCP flows. Setup: link-rate is 100Mb/s, RTT is 160ms, BDP buffering is used, and no background traffic is introduced.

3.1.2 HighSpeed TCP (HSTCP)

HighSpeed TCP (HSTCP) [42] uses a generalized AIMD where the linear increase factor $f_\alpha(cwnd)$ and the multiplicative decrease factor $g_\beta(cwnd)$ are adjusted by a convex function of the current congestion window size. When the congestion window is less than a specified cutoff value, HSTCP uses the same factors as TCP. Most high-speed TCP variants support this form of TCP compatibility, based on the window size. When the window grows beyond the cutoff point, the convex function raises the increase factor and reduces the decrease factor proportional to the window size.

$$\begin{aligned} \text{ACK} &: cwnd \longleftarrow cwnd + \frac{f_\alpha(cwnd)}{cwnd} \\ \text{Loss} &: cwnd \longleftarrow g_\beta(cwnd) \times cwnd \quad \text{(Refer to [42, 76])} \end{aligned} \tag{3.2}$$

Figure 3.2 shows the behavior of two HSTCP flows in the network with 100Mb/s bandwidth and 160ms RTT.

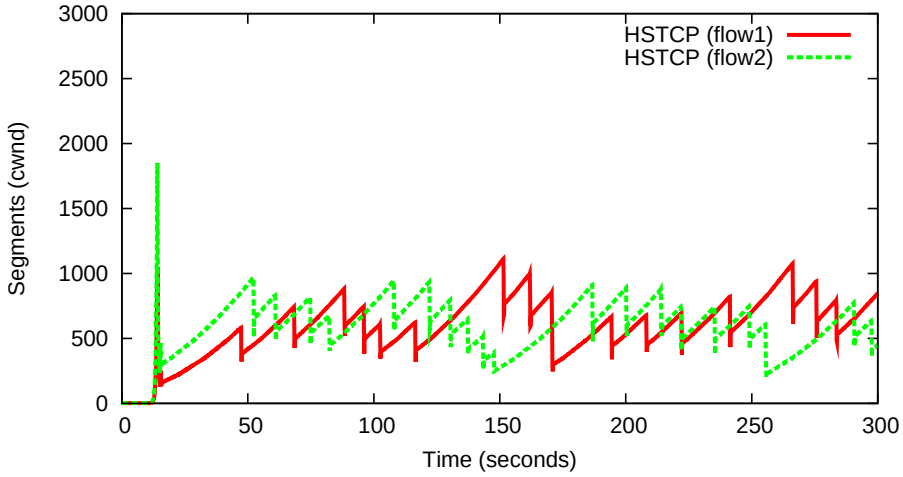


Figure 3.2: CWND evolutions of two HSTCP flows. Setup: link-rate is 100Mb/s, RTT is 160ms, BDP buffering is used, and no background traffic is introduced.

3.1.3 Hamilton TCP (HTCP)

HTCP [94], like CUBIC, uses elapsed time (Δ) since the last congestion event for calculating the current congestion window size. The window growth function of HTCP is a quadratic function of Δ, denoted as $\bar{f}_\alpha(\Delta)$. HTCP is unique in that it adjusts the decrease factor $g_\beta(B)$ by a function of RTTs which is engineered to estimate the queue size B in the network path of the current flow. Thus, the decrease factor is adjusted proportional to the queue size. T_{min} and T_{max} are the minimum and maximum RTTs observed by the current flow, and $B(K+1)$ is a measurement of the throughput during the last congestion epoch [76].

$$\begin{aligned} \text{ACK} &: cwnd \longleftarrow cwnd + \frac{2(1-\beta)f_\alpha(\Delta)}{cwnd} \\ \text{Loss} &: cwnd \longleftarrow g_\beta(B) \times cwnd \end{aligned}$$

where

$$f_\alpha(\Delta) = \begin{cases} 1 & \text{if } \Delta \leq \Delta_L \\ max(\bar{f}_\alpha T_{min}, 1) & \text{if } \Delta > \Delta_L \end{cases}$$

$$g_\beta(B) = \begin{cases} 0.5 & \text{if } \mid \frac{B(k+1)-B(k)}{B(k)} \mid > \Delta(B) \\ min(\frac{T_{min}}{T_{max}}, 0.8) & \text{if } \mid \frac{B(k+1)-B(k)}{B(k)} \mid \leq \Delta(B) \end{cases}$$

$$\bar{f}_\alpha(\Delta) = 1 + 10(\Delta - \Delta_L) + 0.25(\Delta - \Delta_L)^2 \qquad \text{(Refer to [94, 76])} \tag{3.3}$$

Figure 3.3 shows the behavior of two HTCP flows in the network with 100Mb/s bandwidth and 160ms RTT.

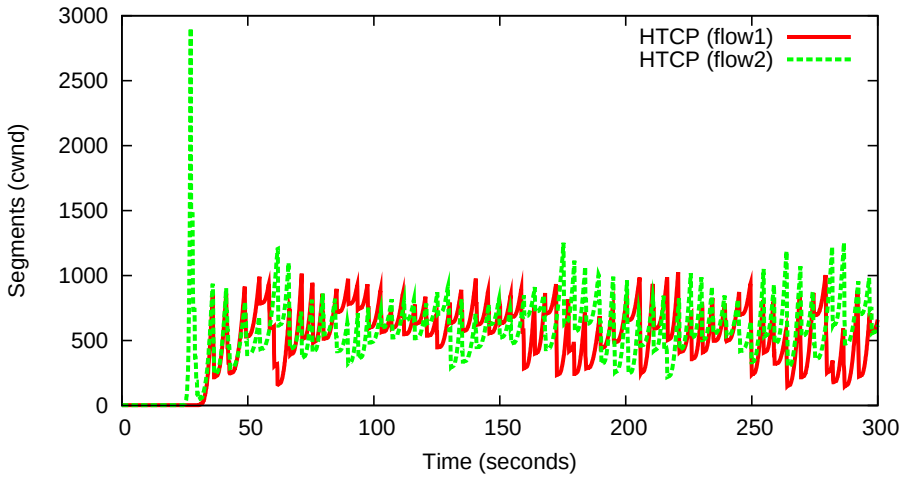


Figure 3.3: CWND evolutions of two HTCP flows. Setup: link-rate is 100Mb/s, RTT is 160ms, BDP buffering is used, and no background traffic is introduced.

3.1.4 TCP Vegas

TCP-Vegas [31] measures the difference (δ) between expected throughput R_E and actual throughput R_A, based on round-trip delays. R_E and R_A are updated once per RTT.

When δ is less than a low threshold α, TCP-Vegas believes the path is not congested and thus increases the transmission rate. When δ is larger than a upper threshold β, which is a strong indication of congestion, TCP-Vegas reduces the transmission rate. Otherwise, TCP-Vegas maintains the current transmission rate. The expected throughput is calculated by dividing the current congestion window by the minimum RTT which typically contains the delay when the path is not congested. For each round trip event, TCP-Vegas computes the actual throughput by dividing the number of packets sent by the sampled RTT. Figure 3.4 shows the behavior of two TCP-Vegas flows under the same settings above.

$$
\begin{aligned}
\text{ACK} \quad &: \quad cwnd \longleftarrow cwnd + \frac{f_\gamma(\delta)}{cwnd} \\
\text{Loss} \quad &: \quad cwnd \longleftarrow \frac{cwnd}{2} \\
R_E &= \frac{cwnd}{RTT_{min}}, \quad R_A = \frac{cwnd}{RTT} \\
\delta &= \max(R_E - R_A, 0) \\
f_\gamma(\delta) &= \begin{cases} 1 & \text{if } \delta \leq \alpha \\ -1 & \text{if } \delta > \beta \end{cases}
\end{aligned}
\tag{3.4}
$$

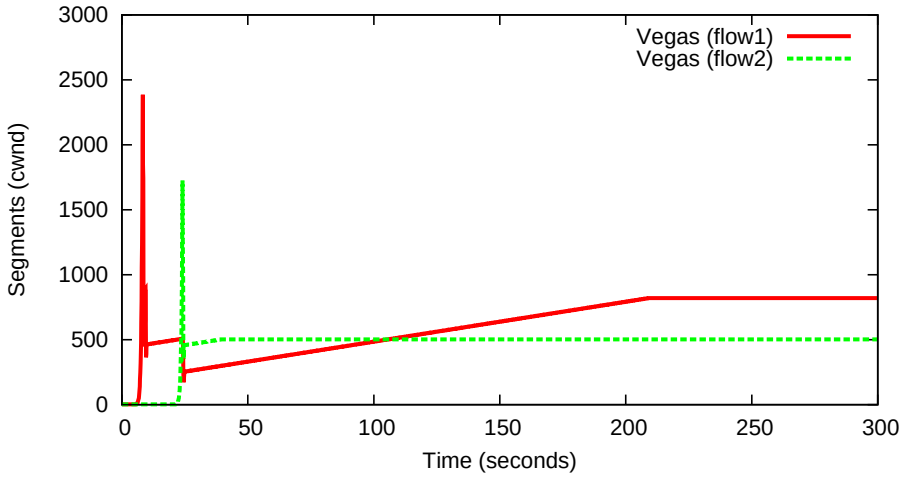


Figure 3.4: CWND evolutions of two TCP-Vegas flows. Setup: link-rate is 100Mb/s, RTT is 160ms, BDP buffering is used, and no background traffic is introduced.

3.1.5 FAST

FAST [68] determines the current congestion window size based on both round-trip delays and packet losses over a path. FAST updates the sending rate at every other RTT. The algorithm estimates the queuing delay of the path using RTTs and if the delay is well below a threshold, it increases the window aggressively and if it gets closer to the threshold, the algorithm slowly reduces the rate of increase. The opposite happens when the delay increases beyond the threshold: slowly decreasing the window at first and then aggressively decreasing the window. For packet losses, FAST halves the congestion window and enters loss recovery in the same manner as TCP. Figure 3.5 shows the trajectories of two FAST flows.

$$
\begin{aligned}
\text{ACK} \quad &: \quad cwnd \longleftarrow \min(2 \times cwnd, \\
&\qquad (1-\gamma)cwnd + \gamma(\frac{T_{min}}{\bar{T}} cwnd + f_\alpha(B, T_q))) \\
\text{Loss} \quad &: \quad cwnd \longleftarrow \frac{cwnd}{2} \\
&\qquad f_\alpha(B, T_q) = \begin{cases} \alpha \times cwnd & \text{if } T_q = 0 \\ \bar{F}_\alpha(B) & \text{otherwise} \end{cases}
\end{aligned}
\tag{3.5}
$$

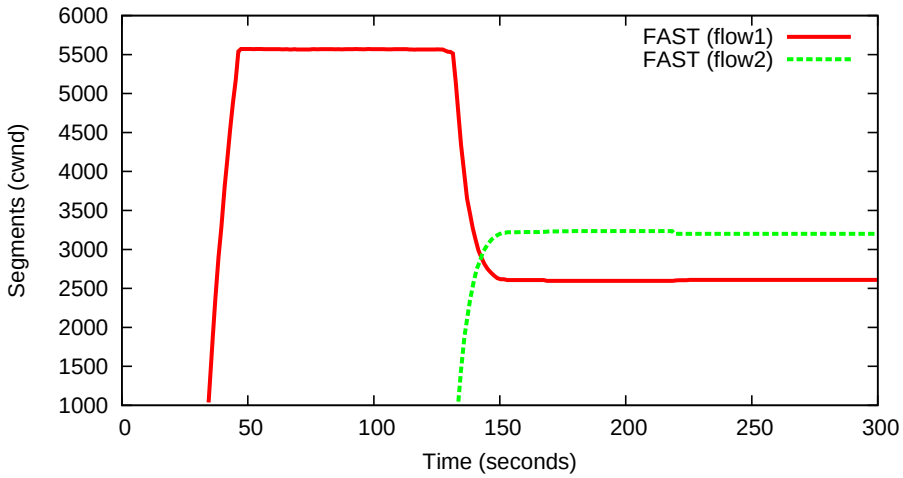


Figure 3.5: CWND evolutions of two FAST flows. Setup: link-rate is 400Mb/s, RTT is 160ms, BDP buffering is used, and no background traffic is introduced.

3.1.6 TCP Westwood

TCP-Westwood [36] estimates an end-to-end available bandwidth by accounting for the rate of returning ACKs. It obtains the estimate b_k by dividing the amount of data sent (d_k) by the interarrival time between t_k and t_{k-1} where t_{k-1} is the time the previous ACK was received (Eq. 3.6). After that, it uses the Tustin approximation to filter out high-frequency components (Eq. 3.7). For packet losses, TCP-Westwood sets a slow start threshold to this estimate, unlike TCP which "blindly" reduces the congestion window by half. This mechanism is effective over wireless links where frequent channel losses are misinterpreted as congestion losses.

$$\text{ACK} \quad : \quad b_k = \frac{d_k}{t_k - t_{k-1}} \tag{3.6}$$

$$\hat{b}_k = \frac{\frac{2\tau}{t_k - t_{k-1}} - 1}{\frac{2\tau}{t_k - t_{k-1}} + 1}\hat{b}_{k-1} + \frac{b_k + b_{k+1}}{\frac{2\tau}{t_k - t_{k-1}} + 1} \tag{3.7}$$

$$\text{Loss} \quad : \quad ssthresh \longleftarrow \frac{\hat{b}_k \times RTT_{min}}{s}$$

$$cwnd \longleftarrow ssthresh$$

$$\text{Timeout} \quad : \quad ssthresh \longleftarrow \frac{\hat{b}_k \times RTT_{min}}{s}$$

$$cwnd \longleftarrow 1$$

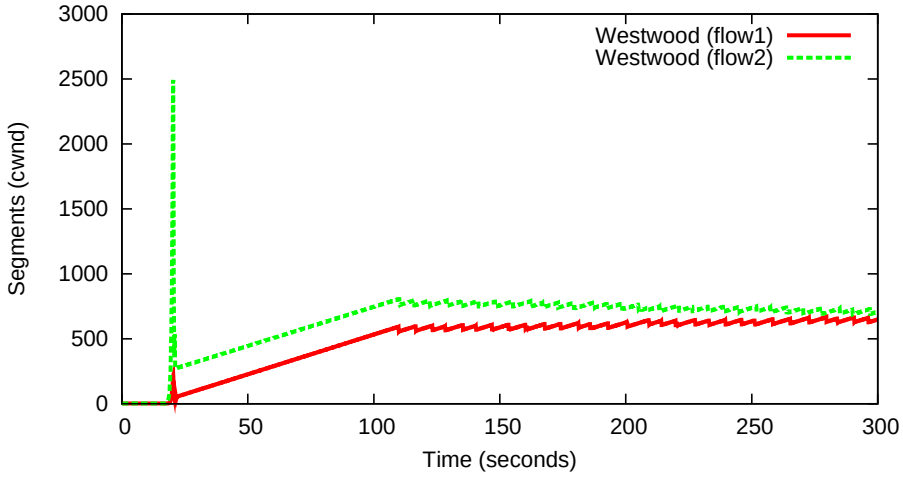


Figure 3.6: CWND evolutions of two TCP-Westwood flows. Setup: link-rate is 100Mb/s, RTT is 160ms, BDP buffering is used, and no background traffic is introduced.

3.1.7 TCP Illinois

TCP-Illinois [77] uses a queueing delay to determine an increase factor α and multiplicative decrease factor β instantaneously during the window increment phase. Precisely, TCP-Illinois sets a large α and small β when the average delay d is small, which is an indication that congestion is not imminent, and sets a small α and large β when d is large because this implies imminent congestion. The average queueing delay of the current RTT round is d_a and the min and max queueing delays so far are d_1 and d_m respectively, and thus $k_1 = \frac{(d_m - d_1)\alpha_{min}\alpha_{max}}{\alpha_{max} - \alpha_{min}}$, $k_2 = \frac{(d_m - d_1)\alpha_{min}}{\alpha_{max} - \alpha_{min}} - d_1$, $k_3 = \frac{\beta_{min} d_3 - \beta_{max} d_2}{d_3 - d_2}$, and $k_4 = \frac{\beta_{max} - \beta_{min}}{d_3 - d_2}$.

$$\begin{aligned} \text{ACK} &: cwnd \longleftarrow cwnd + \frac{\alpha}{cwnd} \\ \text{Loss} &: cwnd \longleftarrow (1 - \beta) \times cwnd \end{aligned} \tag{3.8}$$

$$\alpha = f_1(d_a) = \begin{cases} \alpha_{max} & \text{if } d_a \leq d_1 \\ \frac{k_1}{k_2 + d_a} & \text{otherwise.} \end{cases} \tag{3.9}$$

$$\beta = f_2(d_a) = \begin{cases} \beta_{min} & \text{if } d_a \leq d_2 \\ k_3 + k_4 d_a & \text{if } d_2 < d_a < d_3 \\ \beta_{max} & \text{otherwise.} \end{cases} \tag{3.10}$$

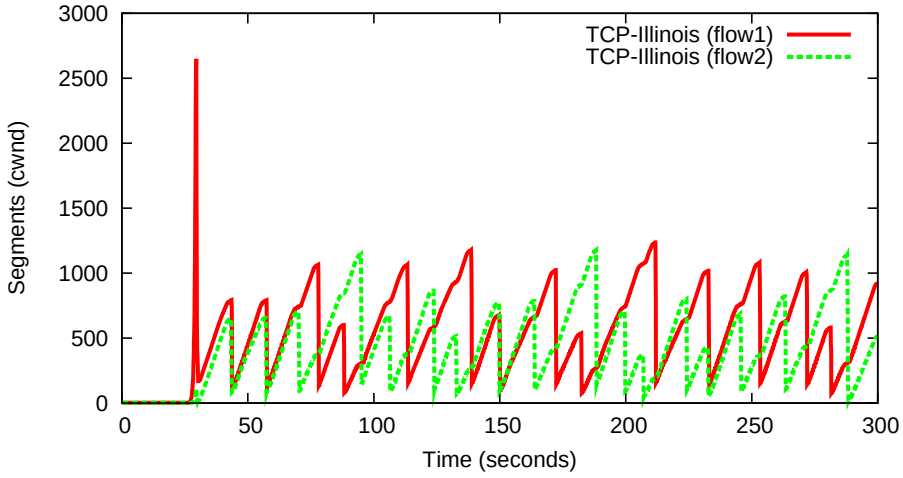


Figure 3.7: CWND evolutions of two TCP-Illinois flows. Setup: link-rate is 100Mb/s, RTT is 160ms, BDP buffering is used, and no background traffic is introduced.

3.1.8 Compound TCP (C-TCP)

Compound TCP [69] maintains two congestion windows - a regular congestion window (*cwnd*) and a delay congestion window (*dwnd*). It sets the congestion window (*wnd*) by summing these two congestion windows. The *cwnd* increases in the same way that TCP-NewReno increases its congestion window while the *dwnd* is determined by using a queueing delay. If the queueing delay is small, *dwnd* increases very rapidly in order to utilize the link. If the queueing delay is large (an indication that the path is getting congested), *dwnd* gradually decreases to compensate for the increase in *cwnd*. The queueing delay component of TCP-Vegas (δ) is used in the algorithm. Compound TCP is used by current Windows operating systems such as Windows Vista and Windows Server 2008.

$$\text{ACK and Loss} : wnd \longleftarrow cwnd + dwnd \tag{3.11}$$

$$cwnd \longleftarrow \begin{cases} cwnd + \frac{1}{wnd} & \text{upon receiving an ACK} \\ \frac{1}{2} \times cwnd & \text{upon detecting loss} \end{cases}$$

$$dwnd(t+1) \longleftarrow \begin{cases} dwnd(t) + (\alpha win(t)^k - 1)^+ & \text{if } \delta < \gamma \\ (dwnd(t) - \xi\delta)^+ & \text{if } \delta \geq \gamma \\ (win(t)(1-\beta) - \frac{cwnd}{2})^+ & \text{upon detecting loss} \end{cases}$$

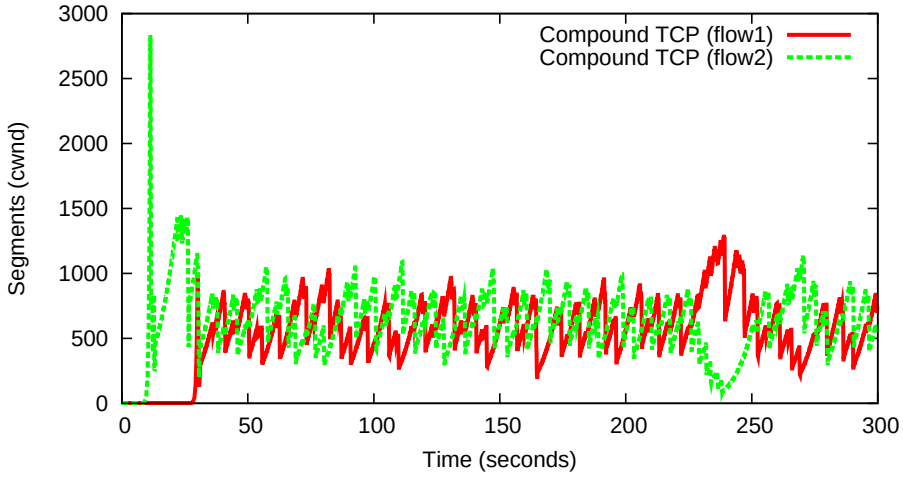


Figure 3.8: CWND evolutions of two Compound-TCP flows. Setup: link-rate is 100Mb/s, RTT is 160ms, BDP buffering is used, and no background traffic is introduced.

3.1.9 TCP Hybla

TCP-Hybla [34] scales the window increment rule to ensure fairness among the flows with different RTTs. It behaves in a similar manner to TCP-NewReno when the RTT of a flow is less than a certain reference RTT (e.g., 20ms). Otherwise, it increases the congestion window size more aggressively to compensate throughput drop due to RTT increase. TCP-Hybla introduces a normalized round-trip time, ρ, where RTT_0 is the RTT of the reference. Note that $t_{\gamma,0}$ is the time at which the congestion window reaches the value $\rho\gamma$. Eq. 3.13 shows the congestion window control of TCP-Hybla. Dividing Eq. 3.13 by RTT, we derive the transmission rate of TCP-Hybla as shown in Eq. 3.14. This confirms that TCP-Hybla is independent of the RTT.

$$\rho = \frac{RTT}{RTT_0} \tag{3.12}$$

$$W(t) = \begin{cases} \rho 2^{\frac{\rho t}{RTT}} & \text{if } 0 \leq t < t_{\gamma,0} \text{ SS} \\ \rho\left(\rho\frac{t-t_{\gamma,0}}{RTT} + \gamma\right) & \text{if } t \geq t_{\gamma,0} \quad \text{CA} \end{cases} \tag{3.13}$$

$$r(t) = \begin{cases} 2^{\frac{t}{RTT_0}} & \text{if } 0 \leq t < t_{\gamma,0} \text{ SS} \\ \frac{1}{RTT_0}\left(\frac{t-t_{\gamma,0}}{RTT_0} + \gamma\right) & \text{if } t \geq t_{\gamma,0} \quad \text{CA} \end{cases} \tag{3.14}$$

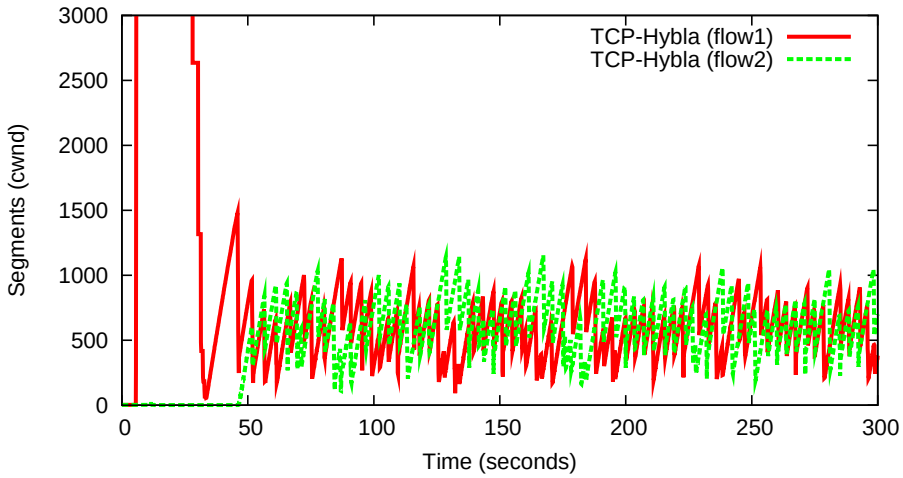


Figure 3.9: CWND evolutions of two TCP-Hybla flows. Setup: link-rate is 100Mb/s, RTT is 160ms, BDP buffering is used, and no background traffic is introduced.

3.1.10 YeAH TCP

YeAH-TCP [24] uses 1) an estimated number of packets enqueued by the flow Q and 2) the level of congestion L. Let RTT_{base} be the minimum RTT observed by the sender and RTT_{min} is the minimum RTT in the current RTT round. The queueing delay is $RTT_{queue} = RTT_{min} - RTT_{base}$. Using RTT_{queue}, we can estimate the number of packets enqueued by the flow Q as: $Q = RTT_{queue}\frac{cwnd}{RTT_{min}}$. By evaluating the ratio between the queueing delay and the minimum RTT of a flow, we can also obtain L as: $L = \frac{RTT_{queue}}{RTT_{base}}$. When Q and L are small, YeAH-TCP is in "fast" mode and runs STCP for rapid increase of the congestion window. When Q is large, YeAH-TCP drains out the queue by reducing the congestion window by Q and sets a slow start threshold at half of the current congestion window.

$$\begin{cases} \text{Fast Mode} : cwnd = cwnd + 0.01 & \text{if } (Q < Q_{max}) \text{ and } (L < \frac{1}{\varphi}) \\ \text{Slow Mode} : & \text{otherwise} \end{cases} \tag{3.15}$$

$$\text{Slow Mode} \begin{cases} cwnd = cwnd - Q,\ ssthresh = \frac{cwnd}{2} & \text{if } (Q > Q_{max}) \\ cwnd = cwnd + \frac{1}{cwnd}, & \text{otherwise} \end{cases} \tag{3.16}$$

$$\text{Loss} : cwnd = cwnd - \min\left(\max\left(\frac{cwnd}{8}, Q\right), \frac{cwnd}{2}\right) \tag{3.17}$$

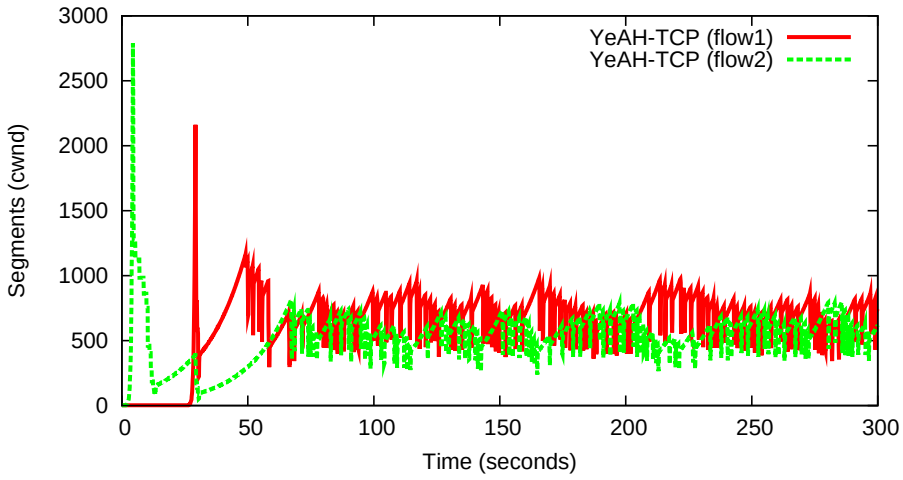


Figure 3.10: CWND evolutions of two YeAH-TCP flows. Setup: link-rate is 100Mb/s, RTT is 160ms, BDP buffering is used, and no background traffic is introduced.

3.1.11 TCP Veno

TCP-Veno [52] determines the current congestion window size in a manner similar to TCP-NewReno, but it uses the delay information of TCP-Vegas to differentiate non-congestion losses. When packet loss happens, if the queue size N implied by the delay increase is within a certain threshold ($\beta = 3$), a strong indication of random loss, TCP-Veno reduces the congestion window by 20%, not by the typical 50%. R_E and R_A are the expected throughput and the actual throughput of a flow, and R_E and R_A are updated once per RTT.

$$\begin{aligned}
&\text{ACK} : cwnd \longleftarrow cwnd + \frac{1}{cwnd} \\
&\text{Loss} : \begin{cases} cwnd \longleftarrow \frac{1}{2} \times cwnd & \text{if } N < \beta \\ cwnd \longleftarrow \frac{4}{5} \times cwnd & \text{otherwise} \end{cases} \\
&R_E = \frac{cwnd}{RTT_{min}}, \quad R_A = \frac{cwnd}{RTT} \\
&\Delta = \max(R_E - R_A, 0) \\
&N = \Delta \times RTT_{min}
\end{aligned} \tag{3.18}$$

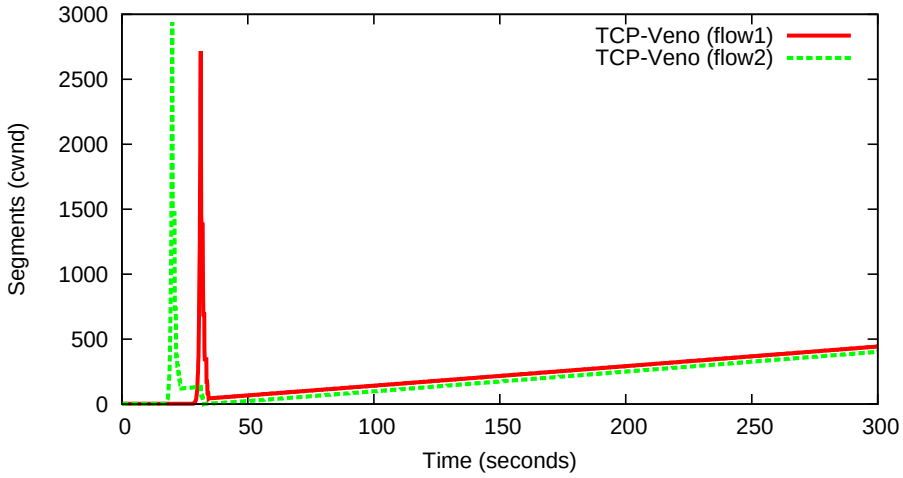
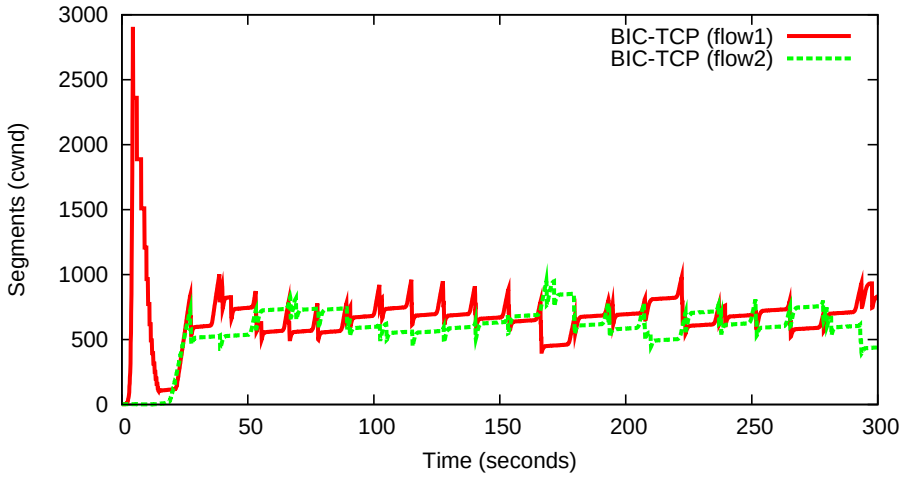


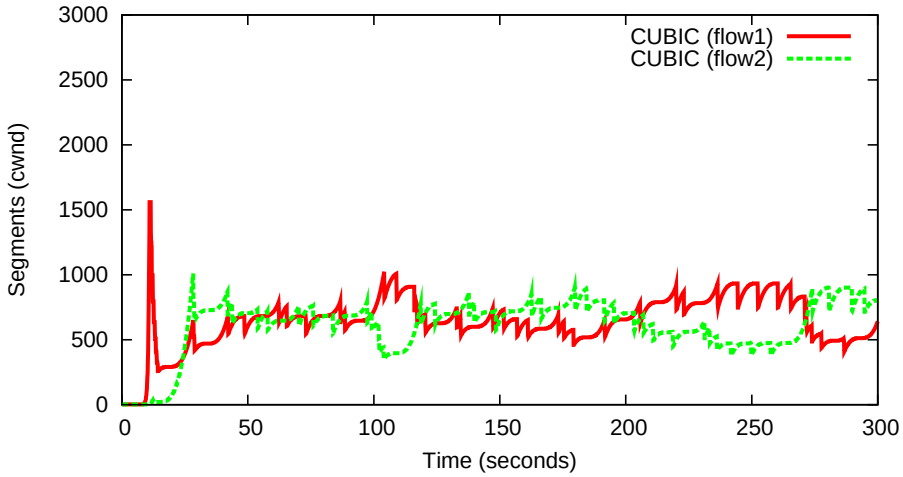
Figure 3.11: CWND evolutions of two TCP-Veno flows. Setup: link-rate is 100Mb/s, RTT is 160ms, BDP buffering is used, and no background traffic is introduced.

3.1.12 BIC and CUBIC

While the detailed algorithms of BIC [106] and CUBIC [57] are presented in Chapter 5, we present snapshots of BIC and CUBIC under the same settings with other protocols for comparison purposes. BIC and CUBIC are known to use concave-convex window profiles which maximize the bandwidth utilization while ensuring excellent stability of the protocols [33].



(a) CWND evolutions of two BIC flows.



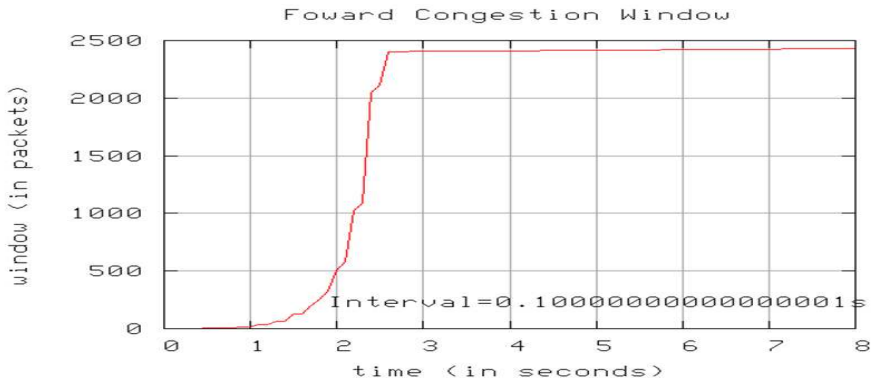
(b) CWND evolutions of two CUBIC flows.

Figure 3.12: CWND evolutions of two BIC flows (a) and two CUBIC flows (b) respectively. Setup: link-rate is 100Mb/s, RTT is 160ms, BDP buffering is used, and no background traffic is introduced.

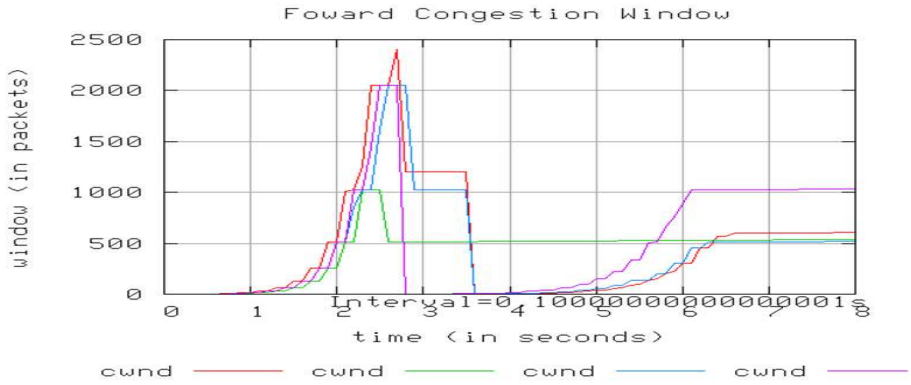
3.2 Slow Start Algorithms

3.2.1 Hoe's Packet-Pair based approach

Hoe [63] proposes to estimate the bottleneck bandwidth using a packet-pair measurement, and to use the estimated value to set the *ssthresh* of TCP-NewReno. Dovrolis et al. [39], however, indicate 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, 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. Figure 3.13 shows its accuracy of single flow and four flows in the network.



(a) One TCP-SACK flow

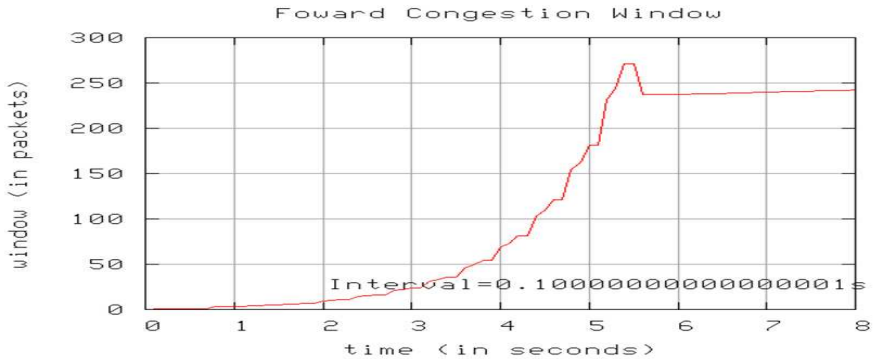


(b) Four TCP-SACK flows

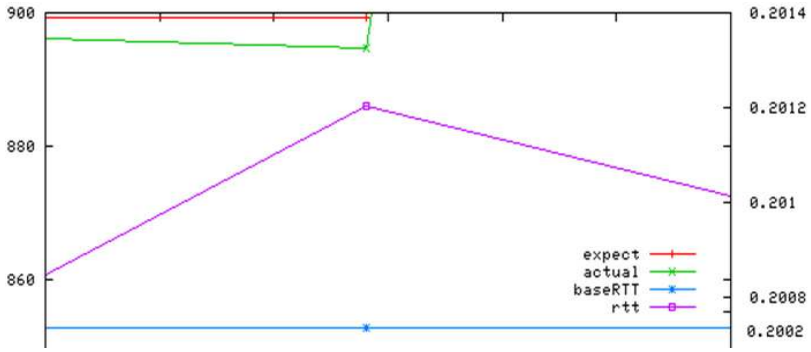
Figure 3.13: Hoe's Packet-Pair based slow start. NS2 simulation setup: link-rate is 100Mb/s, RTT is 200ms, BDP buffering is used, and no background traffic is introduced.

3.2.2 Vegas' modified slow start

Vegas [31] introduces a modified slow start mechanism which allows an exponential growth of *cwnd* at alternating RTTs and, in between, compares its current transmission rate with the expected rate to determine whether the path has more room to increase. The modified slow start of Vegas is known to cause a premature termination of slow start because of an abrupt increase of RTT caused by temporal queue build-ups in the router during bursty TCP transmissions [103]. Figure 3.14 shows the modified slow start of one TCP-Vegas flow. While the BDP of the network is around the 2,500 packets, as shown in (a), TCP-Vegas prematurely terminates the slow start around the 270 packets. This is because of the difference (δ) between the expected throughput and the actual throughput of TCP-Vegas becomes larger due to the temporal RTT increase, and thus terminates the slow start, which is also shown in (b).



(a) One TCP-Vegas flow



(b) Trajectories of TCP-Vegas variables

Figure 3.14: Vegas's modified slow start. NS2 simulation setup: link-rate is 100Mb/s, RTT is 200ms, BDP buffering is used, and no background traffic is introduced.

3.2.3 Limited slow start

Limited slow start [43] is an experimental RFC. It is designed to prevent a large number of packet losses in one RTT by limiting the increment of a congestion window to $max\_ssthresh/2$ per RTT. But using the fixed number 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[1] before reaching the congestion window size of 5000 with an RTT of 200ms. Figure 3.16 shows the snapshot of the limited slow start.

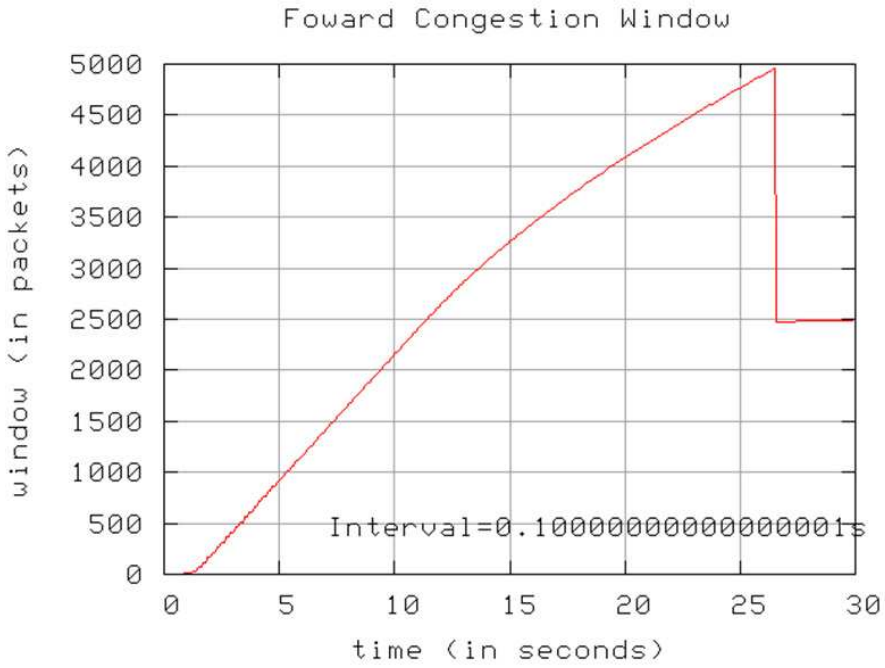


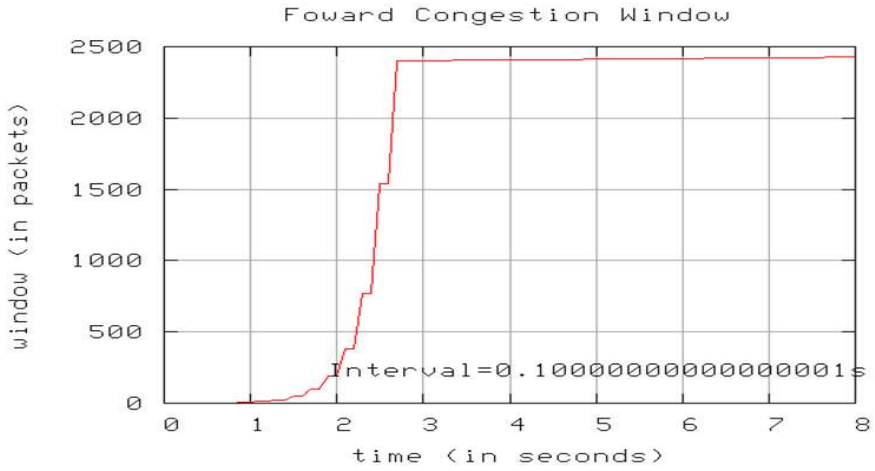
Figure 3.15: Limited slow start. NS2 simulation setup: link-rate is 100Mb/s, RTT is 200ms, BDP buffering is used, and no background traffic is introduced.

3.2.4 Adaptive start

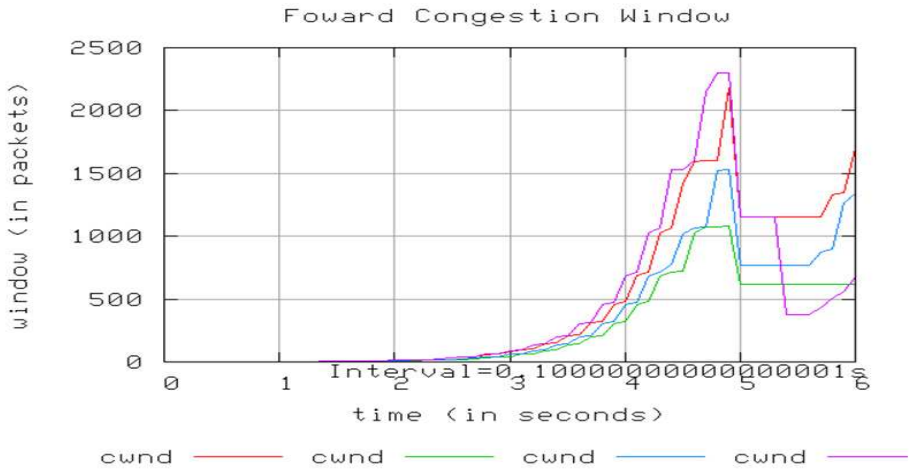
Adaptive start [103] repeatedly resets its *ssthresh* to the value of the expected rate estimation (ERE) when ERE is greater than *ssthresh*. Therefore, with adaptive start, TCP repeats the exponential growth and linear growth of the window until a packet loss

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

occurs. However, adaptive start can be slower than standard slow start, and it is not easily integrated with congestion control algorithms other than TCP-Westwood [78]. Figure 3.16 shows the accuracy of adaptive start with a single flow and with four flows in the network. With the four flows in the network, we can see that four flows have similar ERE values and overshoot the link capacity.



(a) One flow



(b) Four flows

Figure 3.16: Adaptive start. NS2 simulation setup: link-rate is 100Mb/s, RTT is 200ms, BDP buffering is used, and no background traffic is introduced.

3.2.5 Paced start

Paced start [64] probes an available bandwidth during slow start by precisely controlling the gap between the packets. By measuring the gap from incoming ACKs, it regulates the sending gap until it finds a trustworthy bandwidth estimation. Paced Start, however, requires a precise control of packet transmissions and ACK receptions. This is very costly and often impossible to achieve for high bandwidth and long distance networks.

3.2.6 Miscellaneous approaches

Quick-Start [45, 93] determines its allowed sending rate very quickly, but it needs cooperation with the routers that approve Quick-Start along the path. Padmanabhan et al. [84] suggest the use of cached information from previous connections. Wang et al. [102] throttle down the exponential rate of increase when the congestion window approaches *ssthresh*. However, in all cases, because the available bandwidths keep changing and it is difficult pick an initial value of *ssthresh*, there is always a high potential for wrong decisions.

3.3 Loss Differentiation Schemes

Achieving high throughput in the face of packet losses has been of interest ever since error-prone wireless and satellite links have been an integral part of the Internet. We are interested in end-to-end solutions (independent of network participation), and have classified existing literature into the following four main categories:

3.3.1 Distinguishing congestion losses from non-congestion ones

This approach enhances TCP with heuristics that aim to distinguish congestion losses from random ones. Some distinguish the losses based upon queuing delay estimates while others use the history of packet loss rates, and inter-arrival times of packets at the receiver. Samaraweera [92] proposes a delay-based non-congestion loss indicator, based on Jain's congestion predictor [67], that compares the measured RTT to a knee threshold delay. While this work comes the closest to our approach in BLAST, it has two problems in high bandwidth-delay product (BDP) networks: the technique relies on n (number of flows in a network) varying slowly, and also depends on obtaining a good estimate of the *total* number

of bytes in the network (from all other flows). This is a difficult endeavor. The accuracy of the detection mechanism hasn't been evaluated.

TCP Veno [52] disambiguates a loss by comparing its estimate of the number of packets in the bottleneck buffer to a threshold of 3 packets, and reduces its window by 20% on a non-congestive loss (as opposed to 50% otherwise). While Veno is suitable for low bandwidth lossy links, it does not scale in high BDP networks. Others such as ZigZag [37] and ZBS [37] are hybrids that switch between discriminators based on interarrivals of packets at the receiver, and relative one-way delay measurements. TCP-Casablanca [28] and TCP-Ifrane [28] aim to distinguish losses under light congestion conditions using a *biased* queue management scheme. Biaz et al. [28] further provide a more extensive survey of mechanisms that aim to disambiguate congestion losses from random ones.

TCPs relying on Explicit Congestion Notification (ECN) can correctly disambiguate non-congestion related losses, however with ECN not being widely used or deployed, this is not a viable solution for today's networks.

3.3.2 Approaches that do not rely on packet loss as an indication of congestion

This category does not use packet loss rather uses round-trip measurements (TCP Vegas [31], FAST TCP [68]) or estimates of available bandwidth (TCP Westwood [36], WTCP [89]) to modulate TCP's sending rate. Delay-based approaches take increasing RTT measurements as a cue for early congestion and decrease their sending rate without incurring large queuing delays or losses. FAST TCP attempts to maintain the bottleneck per-flow buffer occupancy to a threshold, α. The main problem is its inability to attain its fair-share when competing with loss-based schemes, or when the reverse path is congested.

A TCP Westwood sender estimates the connection rate using incoming ACKs. On a packet loss, it uses the estimate to select a slow-start threshold and a congestion window that are in line with the effective bandwidth being used at the time of the loss. Westwood's main problem is that it cannot accurately estimate the connection rate in the presence of losses (just when it needs it the most) and also in the presence of non-TCP cross traffic (common on the Internet), particularly on the reverse path [28]. When loss is high, the estimate gives a low representation of the available bandwidth, and the sender needlessly reduces its rate despite non-congestion losses.

3.3.3 A hybrid delay and loss based approach

Some proposals, such as H-TCP [94], TCP-Illinois [77], Compound-TCP [69], TCP-Africa [72] and Delay-based AIMD congestion control [75], use a hybrid approach by modulating their sending rate based on both loss and delay information. H-TCP uses delay information to determine the backoff factor on experiencing a loss, $\beta_i = \frac{RTT_{min,i}}{RTT_{max,i}}$. TCP Illinois uses an AIMD algorithm and dynamically adjusts the increase and decrease factors based on queuing delay measurements. Delay-based AIMD aims to decouple TCP's performance from the level of buffer provisioning in the networks and reduces its rate in response to both increased delay and losses. Compound TCP's congestion window is the sum of a Reno-style AIMD window, and a delay-based window that increases or decreases based on the estimated queue occupancy, or upon experiencing a loss.

The main problem is that these TCPs do not explicitly identify the cause of losses, but in fact reduce their transmission rate in response to every loss experienced.

3.3.4 Forward Error Correction (FEC)

While most prior work applied FEC to recover losses without retransmissions for streaming video, it has recently been used to speed up TCP transfers, especially in the presence of non-congestion losses. LT-TCP [101] recovers losses using an adaptive FEC scheme (tuned to the measured error rate) at the TCP layer. Maelstrom [25] is an edge appliance that uses FEC for aggregate traffic at the IP layer, to transparently mask packet losses from TCP. These two FEC schemes mask losses from the TCP layer and rely on ECN for congestion detection — unfortunately this is not an option in networks lacking wide-spread deployment of ECN.

Chapter 4

Experimental Methodology

Experimental methodology plays a vital role in performance evaluation of network protocols because experimental results obtained from unrealistic environments are not representative for real-world scenarios, and unrealistic testing environments can lead to incorrect conclusions about protocol behavior. This Chapter presents our detailed experimental network model that captures complex and realistic characteristics of propagation delays and background traffic on the Internet. Section 4.2 describes the setup of our laboratory network. Section 4.3 discusses how we emulate propagation delays in our experiment. We explain how background traffic is generated for our experiment in Section 4.4 and our experimental procedure in Section 4.5.

4.1 Why Background Traffic is Important

Internet measurement studies showed complex behaviors and characteristics of Internet traffic [19, 26, 51] that are necessary for realistic testing environments. Unfortunately, existing evaluation work [76, 32, 73, 105] did not capture and reproduce these behaviors in their testing environments; the evaluation was done with only a few flows, with little or no background traffic and no control over cross-traffic.

Since congestion control algorithms are sensitive to environmental variables such as background traffic and propagation delays, realistic performance evaluation of congestion control protocols entails creating realistic network environments similar to those of the Internet. The performance evaluation of congestion control protocols also must explore

systematically the parameter space and various protocol aspects in order to expose hidden behaviors, if any, of the protocols under evaluation.

There are several reasons why background traffic is important in protocol testing. First, since it is unlikely that TCP flows under evaluation are used exclusively in a production network, it is essential to investigate the impact of background traffic on these flows, and vice versa. The aggregate behavior of background traffic can induce a rich set of dynamics such as queue fluctuations, patterns of packet losses and fluctuations of the total link utilization at bottleneck links. This can have a significant impact on the performance of high-speed flows.[1] Second, network environments without any randomness in packet arrivals and delays are highly susceptible to the phase effect [107, 49], a commonly observed simulation artifact caused by extreme synchronization among flows. A good mix of background traffic with diverse arrival patterns and delays reduces the likelihood of synchronization. Third, the core of the Internet allows a high degree of statistical multiplexing. Hence, several protocols assume that their flows always run under these types of environment. For instance, the designers of HSTCP [42] and STCP [70] rely on statistical multiplexing for faster convergence (therefore criticizing these protocols for slow or no convergence in testing environments without background traffic, thus no statistical multiplexing, is unfair). Therefore, performance evaluation of network protocols with no or little background traffic does not fully investigate the protocol behaviors that are likely to be observed when these new protocols are deployed on the Internet.

4.2 Testbed Setup

We have constructed a laboratory network as shown in Figure 4.1 to emulate a bottleneck link between two subnetworks. We use this dumbbell setup because it is impossible to create a network topology of realistic scale; even if we use parking lots or multiple bottlenecks, they still fall short of realism. Instead we focus on diverse traffic patterns that background traffic may have when arriving to a core router on the Internet, as these background flows must have gone through many hops and congested routers in the Internet. This allows us to create more realistic environments for testing without actually

[1] We note that randomness at a bottleneck link could be induced by other mechanisms such as the RED algorithm as well. However, RED is not currently widely used in the Internet [40].

creating a topology spanning the entire Internet where network flows might travel through before reaching the bottleneck link.

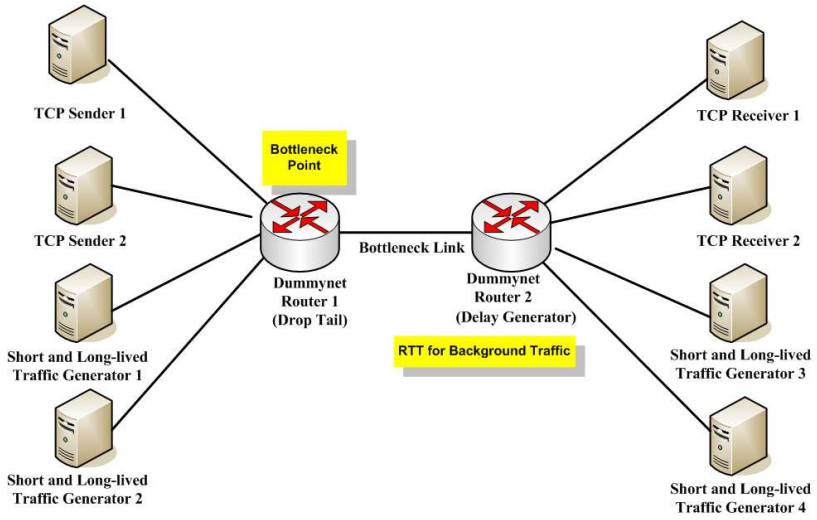


Figure 4.1: Experimental network setup.

At each edge of our testbed are four identical Intel Xeon 2.8GHz Linux machines. Two machines on each side are used for flows of high-speed protocols (high-speed flows) and run Iperf [7] to emulate high-performance applications that have high-bandwidth demand. These machines are tuned so that high-speed protocols can saturate up to 1Gb/s bandwidth without overloading the machines. These machines can also load different operating systems such as Linux, FreeBSD, and Windows. In addition, these four machines are configured to have a very large receive buffer so that transmission rates of high-speed flows are only limited by the congestion control algorithms on the sender side. The other four machines in the network (two machines on each side) run Linux 2.4/2.6 and are dedicated to generate time-varying background traffic by using a modification of Surge [26], a Web traffic generator and Iperf for emulating long-lived TCP flows such as FTP connections. The details of background traffic generation are discussed in Section 4.4.

The core of the network consists of two FreeBSD 5.2.1 machines that are used as routers and run a modified version of Dummynet [90] software[2]. While these two PC-

[2]The reason for using FreeBSD rather than Linux for the PC-based routers is because Dummynet software is only available for FreeBSD.

based routers are tuned to forward traffic at a rate of speed close to 1Gb/s, to eliminate any overloading of the routers, we set the bottleneck bandwidth below 400Mb/s. The first router emulates the bottleneck of the network where high-speed flows and background TCP flows compete for bandwidth. The Dummynet software on the first router is configured to control the bottleneck bandwidth, round-trip times of the flows, the buffer size, and Active Queue Management (AQM) schemes such as DropTail and Random Early Detection (RED) [48]. The second router is used to assign per-flow delays to background traffic flows based on the empirical distribution from an Internet measurement study [19]. The emulation of propagation delays is explained in section 4.3.

While various rules of thumb recommend that the buffer size be proportional to the bandwidth delay product (BDP), we also test protocols with smaller router buffer sizes than BDP. The small buffer size is a likely trend in high-speed routers. This trend is in line with recent research results showing that the buffer size of a bottleneck link with a high degree of multiplexing of TCP connections can be much less than the BDP of a network path [23, 27].

A monitoring program runs on two routers in the middle and collects statistics about the number of forwarded and dropped packets at each router interface. These statistics are used for calculating the throughput and packet loss rate of the router. A modified version of Iperf runs on two high-speed machines on each side to measure the goodput of TCP flows under evaluation, TCP congestion control variables such as cwnd, ssthresh, and RTTs during each experimental run. Additionally, TCPProbe [15] and SIFTR [97] are installed in Linux and FreeBSD kernels respectively and track TCP stack variables in real time. For Windows XP, we use Tcpdump [14]. All the machines in our testbed also gather their system load statistics such as CPU load, memory usage, interrupt latency, and etc.

If our evaluation requires comparison with other protocols, we use the implementation of the protocols that are made available in Linux by their designers. Also it is known that the implementation of Selective Acknowledgement (SACK) in earlier versions of Linux is inefficient for high-speed protocols [76]. Thus, we applied our own fix for those old Linux kernels. Our acknowledgement handling improvement is equally effective for all high-speed protocols under testing.

4.3 Model for Propagation Delays

An extension of Dummynet installed on the second router assigns per-flow delays to background flows in order to emulate the effects of wide-area networks. The Dummynet software is clocked at a 1-millisecond interval - a reasonably fine granularity to have minimal effects on congestion control algorithms. This configuration allows us to apply a different network delay to a different network flow. Each delay sample is randomly selected from the distribution obtained from an Internet measurement study [19]. The distribution is similar to an exponential distribution. Although our network setup has a simple dumbbell topology, the emulation of propagation delays allows us to obtain experimental results as if our experiments were performed on actual wide-area network where different flows with different propagation delays are passing through a router. The round-trip time experienced by a TCP connection (high-speed or background flows) is the sum of the delay induced by Dummynet and any additional queuing delays incurred on the router machines.

4.4 Models for Background Traffic

Two types of flows are used to emulate background traffic: long-lived and short-lived. Long-lived flows are generated by using Iperf. These flows are used to emulate regular long-lived TCP flows such as FTP connections. The amount of background traffic contributed by these flows is controlled by the number of Iperf connections (and TCP's adaptive behavior as well). These flows begin at random times staggered around the beginning of an experiment in order to reduce synchronization and to persist until the end of an experiment.

Short-lived flows are used to emulate web sessions and are generated by a modification of Surge [26], a web traffic generator. The amount of data to be transferred in each web session (i.e., flow size) is randomly sampled from the distribution of file sizes that consists of a log-normal body and a Pareto tail [104, 26, 51]. By controlling the ratio between the body and tail, and also the minimum file size in the Pareto distribution, we can adjust the variability of flow sizes. The inter-arrival time between two successive short-lived flows follows an exponential distribution [104, 51] and is used to control the amount of traffic generated by short-lived flows. For example, the amount of background traffic

contributed by short-lived flows can be increased by reducing the average inter-arrival time between successive web sessions.

The file size distributions of background traffic are categorized into five different distributions (Types I through V). Their details are shown in Table 4.1. These traffic types are used to examine the impact of varying flow size distributions and the amount of background traffic. All short-lived TCP flows have their maximum window size set to 64 KB, which is a typical initial setting of popular operating systems such as Windows, Linux, and FreeBSD. Figure 4.2 shows the cumulative distribution of throughput rates of our background traffic measured at every second from runs containing only the short-lived flows. It shows that Type II has the longest tail showing the largest variance among all types. Type III has slightly smaller variance, but larger than Type I. Type IV has the smallest variance.

We control the amount of background traffic by controlling the average inter-arrival time of two successive flows (we call this "intensity"). By reducing the intensity of flow arrivals, we can increase the amount of background traffic, and vice versa. For different bandwidth experiments, we make background traffic contribute to approximately 15% of the bandwidth. For instance, with the intensity of 0.2 shown in Table 4.1, the average throughput of background traffic (Types I through IV) is approximately 70Mb/s (15%) with the 400Mb/s bandwidth link. The throughput of Type V depends on the aggressiveness of competing high-speed TCP variant flows, and in our experiments, we find the long-lived flows take from 40 to 200Mb/s (10% - 50%) in addition to the web traffic generated by Type II. Further, the same type of background traffic is injected into the reverse direction of the bottleneck link to achieve the effects of ACK compression and to reduce the phase effect [49, 107].

4.5 Experimental Procedure

We tune the kernel of the routers with a number of monitoring programs to collect network data. These monitoring programs sample the bottleneck queue and record the number of forwarded and dropped packets. These data are processed to compute performance metrics such as throughput, fairness, packet losses and queue fluctuations. In addition, we also record the goodput reported by Iperf and TCP stack variables on the

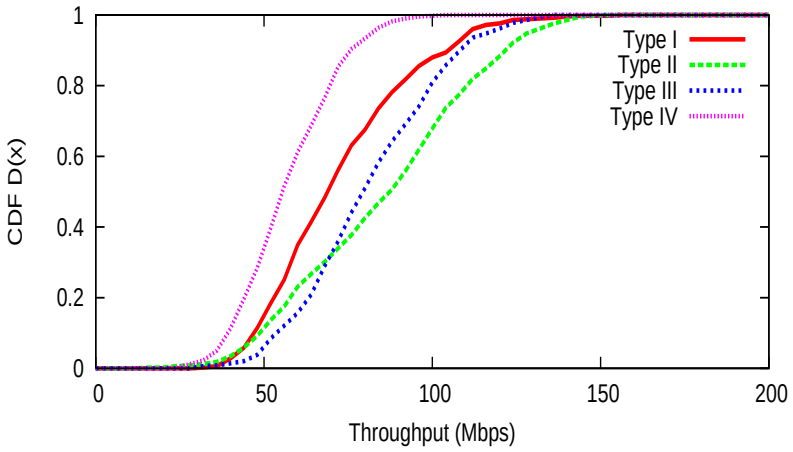


Figure 4.2: The cumulative distribution of throughput rates measured at every second for different types of background traffic. Only short-lived flows are measured. Type II shows the largest variance while Type IV shows the smallest variance.

senders for both high-speed and long-lived background flows.

The detailed procedure for running experiments is as follows. First, two high-speed TCP machines at each side are loaded with specified operating systems. These machines are initialized and configured with their parameters (e.g., a large receive buffer, tuning parameters for network cards, and CPU affinity settings). Meanwhile, two Dummynet router machines are configured with their parameters (e.g., the bottleneck bandwidth, round-trip times for flows, and the buffer size of the router). Second, the monitoring applications on each machine are run and begin gathering their statistics. Background traffic is generated in the forward and reverse directions of the dumbbell, when necessary. Finally, high-speed TCP flows are initiated and the goodput of these flows is measured in the application level. During the lifetime of high-speed TCP flows, TCP stack variables (e.g., cwnd, ssthresh, RTTs and other TCP related variables) are monitored by either actively probing the stack or using Tcpdump.

Unless otherwise mentioned, the long-lived and short-lived background traffic starts at time 0. Each experiment is repeated at least 10 times and the running time of each experiment is 600 to 1200 seconds. Results are reported with 95% confidence interval.

4.6 Summary

Creating a realistic network testing environment is crucial in the evaluation of network protocols. The results from unrealistic network environments may lead to incorrect conclusions about protocol behaviors under evaluation. Recognizing this challenge, we show our realistic testbed setup used for experiments conducted in this dissertation. Specifically, we present details of testbed setups including testbed topology, modeling of round-trip times, and modeling of background traffic which has five different file size distributions.

Table 4.1: File size distributions used in experiments

Traffic type	Description
Type I	This traffic is generated from the default parameters used in Surge [26], a web traffic generator. It uses the body with a log-normal distribution and the tail with a Pareto distribution. It consists of 93% body and 7% tail (i.e., 93% flow arrivals follow the log-normal distribution), the shape parameter of the Pareto distribution is 1.1, and the mean and standard deviation of the log-normal distribution are 9.357 and 1.328 respectively, and the minimum file size for the Pareto distribution is 133 KB. The arrival time of flows follows an exponential distribution with intensity 0.2.
Type II	Type II adds more rate variance to Type I. It consists of web traffic with 70% body and 30% tail. The minimum file size of the Pareto distribution is 1Mbyte. The arrival time of flows follows an exponential distribution with intensity 0.6. The remaining parameter values are the same as those of Type I.
Type III	The variance in the flow size distribution of this traffic type is in between that of Types I and II. It has 10% of all the arriving flows are Peer-to-peer traffic (P2P). The P2P traffic is generated with a Pareto distribution whose shape parameter is 1.1 and the minimum file size is 3Mbytes. The remaining 90% flows are Type I traffic. The inter-arrival time follows an exponential distribution with intensity 0.2. These values are chosen to generate the same amount of traffic as Types I.
Type IV	This traffic type removes a Pareto distribution from the web traffic to further reduce the variance of flow sizes from Type I. All flow sizes follow a log-normal distribution with mean 13.357 and standard deviation 1.318. These values are chosen to generate the same amount of traffic as Types I.
Type V	This traffic is created to increase the amount of traffic without changing the variance of the Type II flow size distribution. We add 12 long-lived flows to the Type II traffic. The RTTs of the flows are varied randomly. The amount of traffic is different from the other types and the actual amount of throughput that this traffic has in a test run depends on the aggressiveness of high-speed flows competing with this traffic because of the long-lived TCP flows.

Chapter 5

CUBIC: A New TCP-Friendly High-Speed TCP Variant

As the Internet evolves to include many very high speed and long distance network paths, the performance of TCP was challenged. These networks are characterized by large bandwidth and delay products (BDPs) which represent the total number of packets needed in flight while keeping the bandwidth fully utilized, in other words, the size of the congestion window. In standard TCPs such as TCP-Reno, TCP-NewReno and TCP-SACK, TCP grows its congestion window by one window per round trip time (RTT). This makes the data transport speed of TCP[1] used in all major operating systems including Windows and Linux rather sluggish, to say the least, severely under-utilizing the networks especially if the duration of the length of flow is much shorter than the time it takes TCP to grow its windows to the full size of the BDP of a path. For instance, if the bandwidth of a network path is 10Gb/s and the RTT is 100ms, with packets of 1250 bytes, the BDP of the path is around 100,000 packets. For TCP to grow its window from the mid-point of the BDP, for example 50,000, it takes about 50,000 RTTs which amounts to 5000 seconds (1.4 hours). If a flow finishes before that time, it severely under-utilizes the path.

To counter this under-utilization problem of TCP, many "high-speed" TCP variants are proposed (e.g., FAST [68], HSTCP [42], STCP [70], HTCP [94], SQRT [59], Westwood [36], CTCP [69], TCP-Hybla [34], YeAH-TCP [24], and BIC-TCP [106]). Recognizing this problem with TCP, the Linux community responded quickly to implement

[1]For brevity, we also denote *Standard TCP* as *TCP*.

a majority of these protocols in Linux and ship them as part of its operating system. After a series of third-party testing and performance validation [32, 61], in 2004, since version 2.6.8, Linux selected BIC-TCP as the default TCP algorithm and the other TCP variants as optional.

What makes BIC-TCP stand out from other TCP algorithms is its stability. It uses a binary search algorithm where the window grows to the mid-point between the last window size (i.e., max) where TCP has a packet loss and the last window size (i.e., min) it has no loss for one RTT period. This "search" into the mid-point intuitively makes sense because the capacity of the current path must be somewhere between the two min and max window sizes if the network conditions do not quickly change from the last congestion signal (the last packet loss). After the window grows to the mid-point, if the network does not have packet losses, then it means that the network can handle more traffic and thus BIC-TCP sets the mid-point the new min and performs another "binary-search" with the min and max windows. This has the effect of growing the window rapidly when the current window size is far from the available capacity of the path, and furthermore, if it is close to the available capacity (where we had the previous loss), it slowly reduces its window increment. It has the smallest window increment at the saturation point and its overshoot beyond the saturation point where losses occur is very small. The whole window growth function is simply a logarithmic concave function. This concave function keeps the congestion window much longer at the saturation point (or equilibrium) than convex or linear functions where they have the largest window increment at the saturation point and thus have the largest overshoot at the time packet losses occur. These features make BIC-TCP very stable and at the same time highly scalable.

BIC-TCP trades speed in reacting to changes in available bandwidth (i.e., convergence speed) for stability. If the available capacity has increased since the last packet losses, the window can grow beyond the max without having a loss. At that time, BIC-TCP increases the window exponentially. Note that an exponential function (a convex function) grows very slowly at the beginning (slower than a linear function). This feature adds to the stability of the protocol because even if the protocol makes mistakes in finding the max window, it first finds the next max window near the previous max point, thus staying at the previous saturation point longer. But the exponential function quickly catches up and its increment becomes very large if the losses do not occur (in which case, the saturation point

has become much larger than the previous one). Because it stays longer near the previous saturation point than other variants, finding a new saturation point can be sluggish if the saturation point has increased far beyond the last one. BIC-TCP, however, reacts quickly and safely to reduced capacity because packet losses occur before the previous max and thus reduces the window by a multiplicative factor. This tradeoff is a design choice of BIC-TCP. It is known that available bandwidth on the Internet changes over a long time scale of several hours [108]. Given that packet losses would occur asynchronously and also proportionally to the bandwidth consumption of a flow under a highly statistically multiplexed environment. Therefore rapid convergence is a natural consequence of the network environment – something the protocol does not have to force. Thus, although BIC-TCP may converge slowly under low statistical multiplexing where only a few flows are competing, its convergence speed is not an issue under typical Internet environments.

CUBIC [57] is the next version of BIC-TCP. It greatly simplifies the window adjustment algorithm of BIC-TCP by replacing the concave and convex window growth portions of BIC-TCP by a cubic function (containing both concave and convex portions). In fact, any odd order polynomial function has this shape. The choice for a cubic function is incidental and convenient. The key feature of CUBIC is that its window growth depends only on the real time between two consecutive congestion events and thus, the window growth is independent of RTTs. Note that one congestion event is the time at which TCP undergoes fast recovery. This feature allows CUBIC flows competing in the same bottleneck to have approximately the same window size independent of their RTTs, achieving good RTT-fairness. Furthermore, when RTTs are short, since the window growth rate is fixed, its growth rate could be slower than TCP standards. Since TCP standards (e.g., TCP-SACK) work well under short RTTs, this feature, in turn, enhances the TCP-friendliness of the protocol.

The implementation of CUBIC in Linux has undergone several upgrades. The most notable upgrade is the efficient implementation of cubic root calculation. Since it requires a floating point operation, implementing it in the kernel requires some integer approximation. Initially it used the bisection method and later changed to the Newton-Raphson method which reduces the computational cost by a factor of nearly 10. Another change to CUBIC after inception is the removal of window clamping. Window clamping was introduced in BIC-TCP where window increments are clamped to a maximum increment

and was inherited by CUBIC for the first version. This forces the window growth to be linear when the target mid-point is much larger than the current window size. The authors conclude that this feature is not needed after extensive testing due to the increased stability of CUBIC. The latest change to CUBIC is adopting a new slow start, called HyStart. The standard slow start of TCP is known for its poor start-up throughput in large BDP networks. HyStart prevents long burst losses by finding a "safe" exit point during slow start and thus improves the start-up throughput of CUBIC substantially in those environments. The details of HyStart are explained in Chapter 6. CUBIC replaced BIC-TCP as the default TCP algorithm of Linux in 2006 after version 2.6.18. The changes and upgrades of CUBIC in Linux are documented in Table 5.1.

The remainder of this Chapter is organized as follows. Section 5.1 presents the details of CUBIC algorithms in Linux, Section 5.2 describes the evolution of CUBIC and its implementation in Linux, and Section 5.3 includes discussion related to the fairness property of CUBIC. Section 5.4 presents the results of experimental evaluation and Section 5.5 presents the conclusion.

5.1 CUBIC Congestion Control

5.1.1 BIC-TCP

In this section, we give some details on BIC-TCP which is a predecessor of CUBIC. The main feature of BIC-TCP is its unique window growth function as discussed in the introduction. Figure 5.1 (a) shows the growth function of BIC-TCP. When a packet loss event occurs, BIC-TCP reduces its window by a multiplicative factor β. The window size just before the reduction is set to the maximum W_{max} and the window size just after the reduction is set to the minimum W_{min}. Then, BIC-TCP performs a binary search using these two parameters - by jumping to the "midpoint" between W_{max} and W_{min}. Since packet losses have occurred at W_{max}, the window size that the network can currently accommodate without loss must be somewhere between these two numbers.

However, jumping to the midpoint could be too much of an increase within one RTT, so if the distance between the midpoint and the current minimum is larger than a fixed constant, called S_{max}, BIC-TCP increments *cwnd* by S_{max} (linear increase). If BIC-TCP does not experience packet losses at the updated window size, that window size becomes

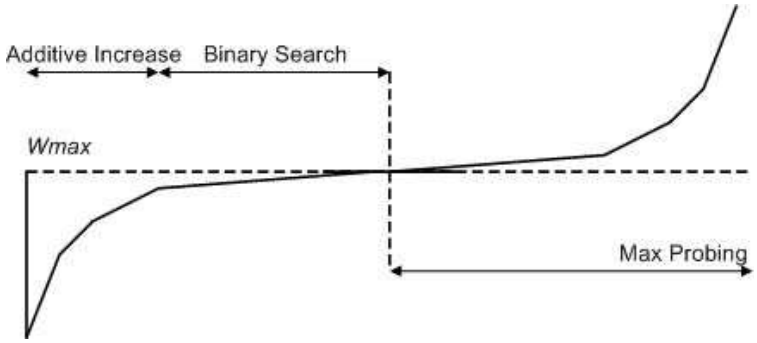
the new minimum. This process continues until the window increment is less than some small constant called S_{min} at which point, the window is set to the current maximum. So the growth function after a window reduction will most likely be a linear function followed by a logarithmic one (marked as "additive increase" and "binary search" respectively in Figure 5.1 (a).)

If the window grows past the maximum, the equilibrium window size must be larger than the current maximum and a new maximum must be found. BIC-TCP enters a new phase called "max probing". Max probing uses a window growth function exactly symmetric to those used in additive increase and binary search (which is logarithmic; its reciprocal will be exponential) and again in additive increase. Figure 5.1 (a) shows the growth function during max probing. During max probing, the window grows slowly initially to find the new maximum nearby, and after some time of slow growth, if it does not find the new maximum (i.e., packet losses), then it guesses the new maximum is further away so it switches to a faster increase by switching to an additive increase where the window size is increased by a large fixed increment. The benefit of BIC-TCP comes from the slow increase around W_{max} and the linear increase during additive increase and max probing.

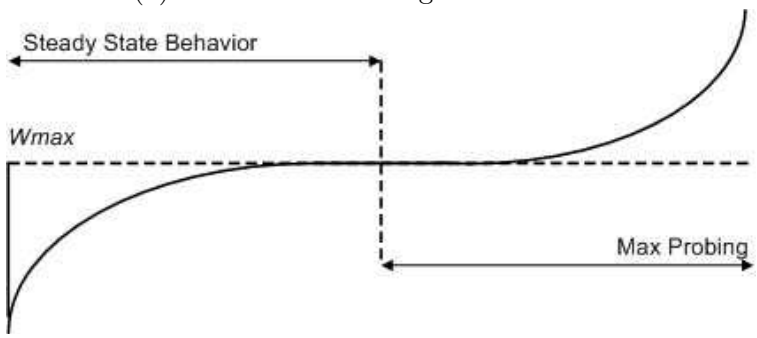
5.1.2 CUBIC window growth function

BIC-TCP achieves good scalability in high speed networks, fairness among competing flows of its own and stability with low window oscillations. However, BIC-TCP's growth function can still be too aggressive for TCP, especially under short RTT or low speed networks. Furthermore, the several different phases (binary search increase, max probing, S_{max} and S_{min}) of window control add complexity in implementing the protocol and analyzing its performance. We have been searching for a new window growth function that simplifies the window control and enhances its TCP friendliness while retaining the strengths of BIC-TCP (especially, stability and scalability).

We introduce a new high-speed TCP variant, called CUBIC. As the name of the protocol suggests, the window growth function of CUBIC is a cubic function whose shape is very similar to the growth function of BIC-TCP. CUBIC uses a cubic function of the elapsed time from the last congestion event. Most alternative algorithms to Standard TCP use a convex increase function, where after a loss event the window increment continuously increases. CUBIC, however, uses both the concave and convex profiles of a cubic function



(a) BIC-TCP window growth function.



(b) CUBIC window growth function.

Figure 5.1: Window growth functions of BIC-TCP and CUBIC.

for window increase. Figure 5.1 (b) shows the growth function of CUBIC.

The details of CUBIC are as follows. After a window reduction following a loss event, it registers W_{max} as the window size where the loss event occurred. Then it decreases the congestion window by a factor of β where β is a window decrease constant and advance to the regular fast retransmit and recovery of TCP. When CUBIC enters into congestion avoidance from fast recovery, it begins to increase the size of window using the concave profile of the cubic function. The cubic function is set to have its plateau at W_{max} so the concave growth continues until the window size becomes W_{max}. After that, the cubic function turns into a convex profile and the convex window growth begins. This style of window adjustment (concave and then convex) improves protocol and network stability while maintaining high network utilization [33]. This is because the window size remains almost constant, forming a plateau around W_{max} where network utilization is deemed

highest. Under steady state, most window size samples of CUBIC are close to W_{max}, thus promoting high network utilization and protocol stability. Note that protocols with convex growth functions tend to have the largest window increment around the saturation point, thus introducing a large burst of packet losses.

The window growth function of CUBIC uses the following function:

$$W(t) = C(t - K)^3 + W_{max} \tag{5.1}$$

where C is a CUBIC parameter, t is the elapsed time from the last window reduction, and K is the time period that the above function requires to increase W to W_{max} when there are no further loss events and is calculated by using the following equation:

$$K = \sqrt[3]{\frac{W_{max}\beta}{C}} \tag{5.2}$$

Upon receiving an ACK during congestion avoidance, CUBIC computes the window growth rate during the next RTT period using Eq. 5.1. It sets $W(t + RTT)$ as the candidate target value of congestion window. Suppose that the current window size is `cwnd`. Depending on the value of `cwnd`, CUBIC runs in three different modes. First, if `cwnd` is less than the window size that (standard) TCP would reach at time t after the last loss event, then CUBIC is in the *TCP mode* (we describe below how to determine this window size of standard TCP in term of time t). Otherwise, if `cwnd` is less than W_{max}, then CUBIC is in the concave region, and if `cwnd` is larger than W_{max}, CUBIC is in the convex region. Figure 5.2 and 5.3 show the pseudo-code of the window adjustment algorithm of CUBIC implemented in Linux.

Initialization:

$tcp\_friendliness \longleftarrow 1$

$fast\_convergence \longleftarrow 1$

$C \longleftarrow 0.4,\ \beta \longleftarrow 0.2$

$cubic\_reset()$

On each ACK:

begin

if $dMin$ **then** $dMin \longleftarrow min(dMin, RTT)$

else $dMin \longleftarrow RTT$

if $cwnd \leq ssthresh$ **then** $cwnd \longleftarrow cwnd + 1$

else

$cnt \longleftarrow cubic\_update()$

if $cwnd\_cnt > cnt$ **then**

$cwnd \longleftarrow cwnd + 1$

$cwnd\_cnt \longleftarrow 0$

else $cwnd\_cnt \longleftarrow cwnd\_cnt + 1$

end

Packet loss:

begin

$epoch\_start \longleftarrow 0$

if $cwnd < W_{last\_max}$**and** $fast\_convergence$ **then**

$W_{last\_max} \longleftarrow cwnd * \frac{(2-\beta)}{2}$ ..... **(5.1.7)**

else $W_{last\_max} \longleftarrow cwnd$

$ssthresh \longleftarrow cwnd \longleftarrow cwnd * (1 - \beta)$ ..... **(5.1.6)**

end

Timeout:

begin

$cubic\_reset()$

end

Figure 5.2: Linux CUBIC algorithm v2.2.

cubic_update(): ..... **(5.1.2)**

begin

- $ack\_cnt \longleftarrow ack\_cnt + 1$
- **if** $epoch\_start \leq 0$ **then**
  - $epoch\_start \longleftarrow tcp\_time\_stamp$
  - **if** $cwnd < W_{last\_max}$ **then**
    - $K \longleftarrow \sqrt[3]{\frac{W_{last\_max} - cwnd}{C}}$
    - $origin\_point \longleftarrow W_{last\_max}$
  - **else**
    - $K \longleftarrow 0,\ origin\_point \longleftarrow cwnd$
  - $ack\_cnt \longleftarrow 1$
  - $W_{tcp} \longleftarrow cwnd$
- $t \longleftarrow tcp\_time\_stamp + dMin - epoch\_start$
- $target \longleftarrow origin\_point + C(t - K)^3$
- **if** $target > cwnd$ **then** $cnt \longleftarrow \frac{cwnd}{target - cwnd}$ ..... **(5.1.4, 5.1.5)**
- **else** $cnt \longleftarrow 100 * cwnd$
- **if** $tcp\_friendliness$ **then** $cubic\_tcp\_friendliness()$

end

cubic_tcp_friendliness(): ..... **(5.1.3)**

begin

- $W_{tcp} \longleftarrow W_{tcp} + \frac{3\beta}{2-\beta} * \frac{ack\_cnt}{cwnd}$
- $ack\_cnt \longleftarrow 0$
- **if** $W_{tcp} > cwnd$ **then**
  - $max\_cnt \longleftarrow \frac{cwnd}{W_{tcp} - cwnd}$
  - **if** $cnt > max\_cnt$ **then** $cnt \longleftarrow max\_cnt$

end

cubic_reset():

begin

- $W_{last\_max} \longleftarrow 0,\ epoch\_start \longleftarrow 0,\ origin\_point \longleftarrow 0$
- $dMin \longleftarrow 0,\ W_{tcp} \longleftarrow 0,\ K \longleftarrow 0,\ ack\_cnt \longleftarrow 0$

end

Figure 5.3: Linux CUBIC algorithm v2.2. (Continued)

5.1.3 TCP-friendly region

When receiving an ACK in congestion avoidance, we first check to determine whether the protocol is in the TCP region or not. This is done as follows. We can analyze the window size of TCP in terms of the elapsed time t. Using a simple analysis in [46], we can find the average window size of additive increase and multiplicative decrease (AIMD) with an additive factor α and a multiplicative factor β.

$$\frac{1}{RTT}\sqrt{\frac{\alpha}{2}\frac{2-\beta}{\beta}\frac{1}{p}} \tag{5.3}$$

By the same analysis, the average window size of TCP with $\alpha = 1$ and $\beta = 0.5$ is $\frac{1}{RTT}\sqrt{\frac{3}{2}\frac{1}{p}}$. Thus, for Eq. 5.3 to be the same as that of TCP, α must be equal to $\frac{3\beta}{2-\beta}$. If TCP increases its window by α per RTT, we derive the window size of TCP in terms of the elapsed time t as follows:

$$W_{tcp(t)} = W_{max}(1-\beta) + 3\frac{\beta}{2-\beta}\frac{t}{RTT} \tag{5.4}$$

If `cwnd` is less than $W_{tcp(t)}$, then the protocol is in the TCP mode and `cwnd` is set to $W_{tcp(t)}$ at each reception of ACK. The `cubic_tcp_friendliness()` in Figure 5.3 describes this behavior.

5.1.4 Concave region

When receiving an ACK in congestion avoidance, if the protocol is not in the TCP mode and `cwnd` is less than W_{max}, then the protocol is in the concave region. In this region, *cwnd* is incremented by $\frac{W(t+RTT)-\texttt{cwnd}}{\texttt{cwnd}}$, which is shown at (5.1.4) in Figure 5.3.

5.1.5 Convex region

When the window size of CUBIC is larger than W_{max}, it passes the plateau of the cubic function after which CUBIC follows the convex profile of the cubic function. Since `cwnd` is larger than the previous saturation point W_{max}, it indicates that the network conditions might have changed since the last loss event, possibly implying more available bandwidth after some flow departures. Since the Internet is highly asynchronous, fluctuations in available bandwidth always exist. The convex profile ensures that the window increases very slowly at the beginning and gradually increases its growth rate. We also call

this phase the maximum probing phase since CUBIC is searching for a new W_{max}. Since we do not modify the window increase function for the convex region only, the window growth function for both regions remains unchanged. To be exact, if the protocol is the convex region outside the TCP mode, `cwnd` is incremented by $\frac{W(t+RTT)-\texttt{cwnd}}{\texttt{cwnd}}$, which is shown at (5.1.5) in Figure 5.3.

5.1.6 Multiplicative decrease

When a packet loss occurs, CUBIC reduces its window size by a factor of β. We set β to 0.2. A side effect of setting β to a smaller value than 0.5 is slower convergence. We believe that while a more adaptive setting of β could result in faster convergence, it would make the analysis of the protocol much more difficult and would also affect the stability of the protocol. This adaptive adjustment of β is a future research issue. The pseudo code for this operation is shown at (5.1.6) in Figure 5.2.

5.1.7 Fast Convergence

To improve the convergence speed of CUBIC, we add a heuristic to the protocol. When a new flow joins the network, existing flows on the network need to give up their bandwidth shares to allow the new flow some room for growth. To increase this release of bandwidth by existing flows, we add the following mechanism called *fast convergence*.

With fast convergence, when a loss event occurs, before a window reduction of the congestion window, the protocol remembers the last value of W_{max} before it updates W_{max} for the current loss event. Let us call the last value of W_{max} W_{last_max}. At a loss event, if the current value of W_{max} is less than the last value of it, W_{last_max}, this indicates that the saturation point experienced by this flow is diminishing because of the change in available bandwidth. Then we allow this flow to release more bandwidth by reducing $W_{\max}$ further. This action effectively lengthens the time for this flow to increase its window because the reduced $W_{\max}$ forces the flow to reach a plateau earlier. This allows more time for the new flow to catch up to its window size. The pseudo code for this operation is shown at (5.1.7) in Figure 5.2.

5.2 CUBIC in Linux Kernel

Since the first release of CUBIC to the Linux community in 2006, CUBIC has undergone several upgrades. This section documents those changes.

5.2.1 Evolution of CUBIC in Linux

Table 5.1 summarizes important updates [3] on the implementation of CUBIC in Linux since its first introduction in Linux 2.6.13. Most updates on CUBIC are focussed on performance and implementation efficiency improvements. One of notable optimizations is the improvement of cubic root calculation. The implementation of CUBIC requires solving Eq. 5.2, a cubic root calculation. The initial implementation of CUBIC [53] in Linux uses the bisection method. But the Linux developer community worked together to replace it with the Newton-Raphson method which improves the running time by more than 10 times on average (1032 clocks vs. 79 clocks) and reduces the variance in running times. CUBIC also underwent several algorithmic changes to have its current form and to enhance its scalability, fairness and convergence speed.

5.2.2 Pluggable Congestion Module

More inclusions of TCP variants to the Linux kernel have substantially increased the complexity of the TCP code in the kernel. Even though a new TCP algorithm comes with a patch for the kernel, this process requires frequent kernel recompilations and threatens the stability of the TCP code. To eliminate the need of kernel recompilation and to help experiment with a new TCP algorithm in Linux, Stephen Hemminger introduces a new architecture [62, 18], called *pluggable congestion module*, in Linux 2.6.13. It is dynamically loadable and allows switching between different congestion control algorithm modules on the fly without recompilation. Figure 5.4 shows the interface to this module, named *tcp_congestion_ops*. Each method in *tcp_congestion_ops* is a hook in the TCP code that provides access to the Linux TCP stack. A new congestion control algorithm requires the definition of *cong_avoid* and *ssthresh*, but the other methods are optional.

The *init* and *release* functions are called for the initialization and termination of a given TCP algorithm. The *ssthresh* function sets the slow start threshold and it is called when the given TCP detects a loss. The lower bound on the congestion window is the

Table 5.1: CUBIC version history

Version	Kernel	Updates
2.0-pre	2.6.13	The authors release the first CUBIC implementation in Linux to the Linux community [53].
2.0	2.6.15	CUBIC is officially included in the Linux kernel.
	2.6.18	CUBIC replaces BIC-TCP as the default TCP protocol in Linux kernel.
	2.6.19	The original implementation of CUBIC has a scaling bug. It took about a month to fix this bug since CUBIC replaced BIC-TCP.
	2.6.21	Its original implementation by the authors is optimized by Linux developers for better performance [60, 99]. In particular, the cubic root calculation in CUBIC, originally implemented in the bisection method, is now replaced by a Newton-Raphson method with table lookups for small values. This results in a performance improvement of more than 10 times in cubic root calculation. On average, the bisection method costs 1,032 clocks while the improved version costs only 79 clocks.
2.1	2.6.22	The original implementation of CUBIC clamped the maximum window increment to 32 packets per RTT. This feature is inherited from BIC-TCP (S_{max}). An extensive lab testing confirmed that CUBIC can safely remove this window clamping in the concave region. This enhances the scalability of CUBIC over very large BDP network paths. This is incorporated in CUBIC 2.1 (Linux 2.6.22).
	2.6.22-rc4	CUBIC improves slow start for fast start-up by removing *initial_ssthresh*.
	2.6.23	The use of received timestamp option value from RTT calculation is removed to prevent possible malicious receiver attacks that report wrong timestamps to reduce RTTs for more throughput.
2.2	2.6.25-rc3	Window clamping during the convex growth phase is also removed. This feature allows CUBIC to improve its convergence speed while maintaining its fairness and TCP friendliness.
2.3	2.6.29	CUBIC adopts a new TCP slow start, called HyStart which improves start-up throughput of TCP substantially over high BDP paths.

```
struct tcp_congestion_ops {
..
  void (*init)(struct sock *sk);
  void (*release)(struct sock *sk);
  u32 (*ssthresh)(struct sock *sk);
  u32 (*min_cwnd)(const struct sock *sk);
  void (*cong_avoid)(struct sock *sk, u32 ack,
                  u32 in_flight);
  void (*set_state)(struct sock *sk, u8 new_state);
  void (*cwnd_event)(struct sock *sk,
                   enum tcp_ca_event ev);
  u32  (*undo_cwnd)(struct sock *sk);
  void (*pkts_acked)(struct sock *sk, u32 num_acked,
                   s32 rtt_us);
  void (*get_info)(struct sock *sk, u32 ext,
                  struct sk_buff *skb);
  char  name[TCP_CA_NAME_MAX];
..
};
```

Figure 5.4: *tcp_congestion_ops* structure

slow start threshold, but when congestion control needs to override this lower bound, the *min_cwnd* function can be used. The *cong_avoid* function is called whenever an ACK arrives and the congestion window (*cwnd*) is adjusted. For instance, in standard TCP New-Reno, when an ACK arrives, *cong_avoid* increments *cwnd* by one, if the current *cwnd* is less than the slow start threshold (during slow start). Otherwise, *cong_avoid* increments *cwnd* by $\frac{1}{cwnd}$ (during congestion avoidance). The *set_state* function is called when the congestion control state of TCP is changed among Normal, Loss Recovery, Loss Recovery after Timeout, Reordering, and Congestion Window Reduction. The *cwnd_event* function is called when the events defined in *tcp_ca_event* occur. When an algorithm is required to handle one of the events, we can create a hook to *cwnd_event* which is called when the corresponding event occurs. The *undo_cwnd* function handles false detection of loss or timeout. When TCP recognizes the change to *cwnd* is wrong, it falls back to the original *cwnd* using the *undo_cwnd* function. The *pkts_acked* function is a hook for counting ACKs; many protocols

(e.g., BIC-TCP, CUBIC, and H-TCP) use this hook to obtain RTT information. The *get_info* function is a hook for providing congestion control information to the user space.

CUBIC has been implemented as one of pluggable congestion control modules. The following are the hooks that CUBIC use for its implementation [8].

1. *bictcp_init*: initializes private variables used for CUBIC algorithm. If *initial_ssthresh* is not 0, then set the slow start threshold to this value. If *initial_ssthresh* is properly set by users when there is no history information about the end-to-end path, it can improve the start-up behavior of CUBIC significantly.

2. *bictcp_recalc_ssthresh*: If the fast convergence mode is turned on and the current *cwnd* is smaller than *last_max*, set *last_max* to $cwnd * (1 - \frac{\beta}{2})$. Otherwise, set *last_max* to $cwnd * (1 - \beta)$. The slow start threshold is always set to $cwnd * (1 - \beta)$ because TCP needs to back off due to congestion.

3. *bictcp_cong_avoid*: increases *cwnd* by computing the difference between the current *cwnd* value and its expected value in the next RTT round which is obtained by cubic root calculation.

4. *bictcp_set_state*: resets all variables when a timeout happens.

5. *bictcp_undo_cwnd*: returns the maximum between the current *cwnd* value and the *last_max* (which is the congestion window before the drop).

6. *bictcp_acked*: maintains the minimum delay observed so far. The minimum delay is reset when a timeout happens.

5.3 Discussion

We discuss the fairness of CUBIC to the standard TCP. Section 5.3.1 derives steady-state throughput of CUBIC with a deterministic loss model. Section 5.3.2 shows the fairness of CUBIC compared to the standard TCP in terms of their steady-state throughput. Note that we derive the steady-state throughput of the standard TCP in Section 2.2. Section 5.3.3 shows some representative snapshots of CUBIC in different network environments.

5.3.1 CUBIC's steady-state throughput

Before delving into the analysis, we define the variables used for CUBIC analysis in Table 5.2. With a deterministic loss model where the number of packets between two successive loss events is proportional to $1/p$, CUBIC always operates with the concave window profile which greatly simplifies the performance analysis of CUBIC. Figure 5.5 shows the evolution of CUBIC's congestion window in steady-state. The interval between two consecutive loss events in the steady state is K, and thus the number of RTTs between two consecutive loss events is $\frac{K}{T}$.

Table 5.2: Variables used for CUBIC throughput analysis.

Variable	Definition
W	The window size right before a loss event.
β	Multiplicative decrease factor. After a loss event, the window size reduces to $(1-\beta)W$.
T	The RTT of a flow.
C	CUBIC scaling factor.
K	The interval between two consecutive loss events in the steady-state.
p	Loss event rate.

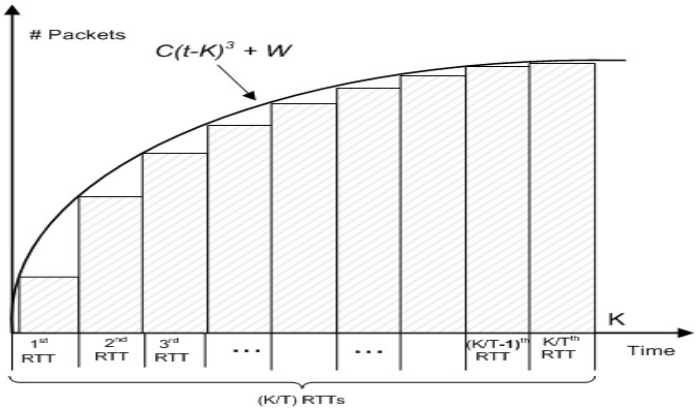


Figure 5.5: CUBIC's congestion window in steady-state.

We recall that CUBIC window growth has the following function:

$$W(t) = C(t-K)^3 + W \tag{5.5}$$

By solving Eq. 5.5 for K, we obtain the following:

$$K = \left(\frac{W\beta}{C}\right)^{\frac{1}{3}} \tag{5.6}$$

In the i^{th} RTT after a loss event, the window size of CUBIC is $C(iT-K)^3 + W$. Then the number of packets sent in a loss epoch is

$$\begin{aligned}
\frac{1}{p} &= \sum_{i=0}^{\frac{K}{T}} (C(iT-K)^3 + W) && (5.7)\\
&= C\sum_{i=0}^{\frac{K}{T}} (K-iT)^3 + W\frac{K}{T} \quad = \quad -C\sum_{i=0}^{\frac{K}{T}} (iT)^3 + W\frac{K}{T}\\
&= -CT^3\sum_{i=0}^{\frac{K}{T}} i^3 + W\frac{K}{T} \quad = \quad -CT^3\frac{(\frac{K}{T})^2(\frac{K}{T}+1)^2}{4} + W\frac{K}{T}\\
&\approx -CT^3\frac{(\frac{K}{T})^2(\frac{K}{T})^2}{4} + W\frac{K}{T}\\
&= -\frac{CK^4}{4T} + W\frac{K}{T} && (5.8)
\end{aligned}$$

By substituting Eq. 5.8 with K in Eq. 5.6, we obtain

$$\begin{aligned}
\frac{1}{p} &= \frac{W}{T}\left(\frac{W\beta}{C}\right)^{\frac{1}{3}} - \frac{C}{4T}\left(\frac{W\beta}{C}\right)^{\frac{4}{3}}\\
&= \frac{W^{\frac{4}{3}}\beta^{\frac{1}{3}}}{C^{\frac{1}{3}}T}\left(\frac{4-\beta}{4}\right) && (5.9)
\end{aligned}$$

By solving Eq. 5.9 for W, we obtain

$$W = \left(\frac{C}{\beta}\right)^{\frac{1}{4}}\left(\frac{4T}{p(4-\beta)}\right)^{\frac{3}{4}} \tag{5.10}$$

The average window size $\mathbb{E}\{W_{cubic}\}$ is calculated by dividing the number of packets being sent by the number of RTTs for one loss epoch.

$$\mathbb{E}\{W_{cubic}\} = \frac{-\frac{CK^4}{4T} + W\frac{K}{T}}{\frac{K}{T}} = \left(\frac{4-\beta}{4}\right)W$$
$$= \sqrt[4]{\frac{C(4-\beta)}{4\beta}(\frac{RTT}{p})^3} \quad (5.11)$$

To ensure fairness to Standard TCP based on our argument in the introduction, we set C to 0.4. We find that this value of C allows the size of the TCP friendly region to be large enough to encompass most of the environments where Standard TCP performs well while preserving the scalability of the window growth function. With β set to 0.2, the Eq. 5.11 is reduced to the following function:

$$\mathbb{E}\{W_{cubic}\} = 1.17\sqrt[4]{(\frac{RTT}{p})^3} \quad (5.12)$$

Eq. 5.12 is used to argue the fairness of CUBIC to Standard TCP and its safety for deployment below.

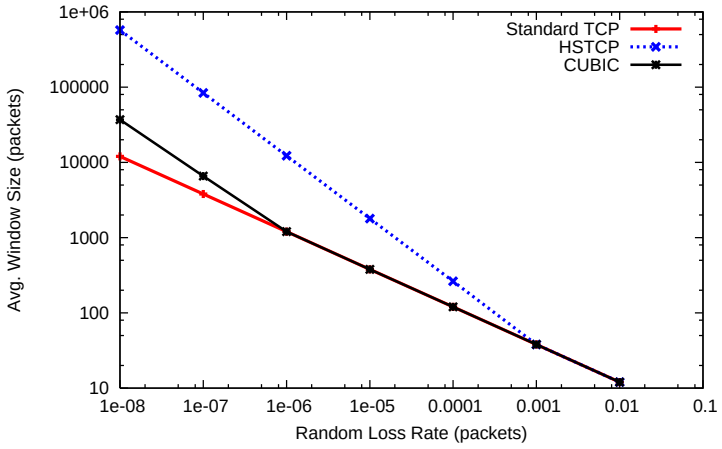
5.3.2 Fairness to standard TCP

In environments where standard TCP is able to make reasonable use of the available bandwidth, CUBIC does not significantly change this state. Standard TCP performs well in the following two types of networks:

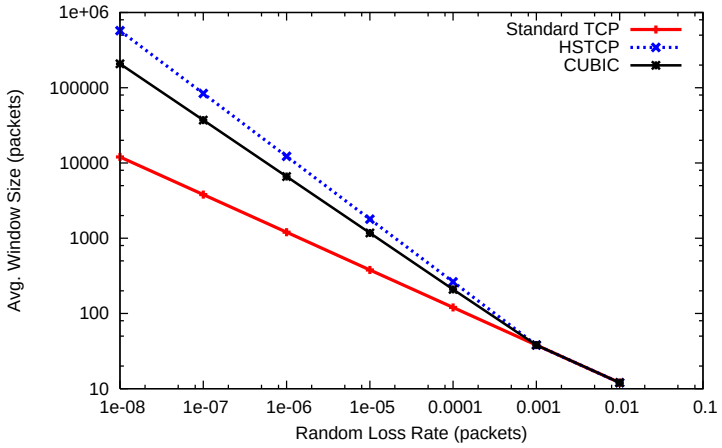
1. Networks with a small bandwidth-delay product (BDP).

2. Networks with a short RTT, but not necessarily a small BDP.

CUBIC is designed to behave very similarly to standard TCP in the above two examples. Figure 5.6 shows the response function (average window size) of standard TCP, HSTCP, and CUBIC. The average window size of standard TCP and HSTCP is shown in reference [42]. The average window size of CUBIC is calculated by using Eq. 5.12 and the CUBIC TCP-friendly equation in Eq. 5.4. Figure 5.6 shows that CUBIC is more friendly to TCP than HSTCP, especially in networks with a short RTT where TCP performs reasonably well. For example, in a network with RTT = 10ms and p = 10^{-6}, TCP has an average window of 1200 packets. If the packet size is 1500 bytes, then TCP can achieve an average



(a) Networks with 10ms RTT.



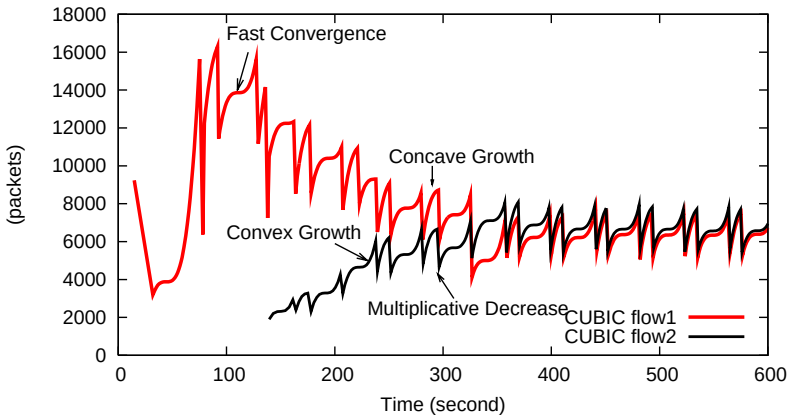
(b) Networks with 100ms RTT.

Figure 5.6: Response function of standard TCP, HSTCP, and CUBIC in networks with 10ms (a) and 100ms (b) RTTs respectively.

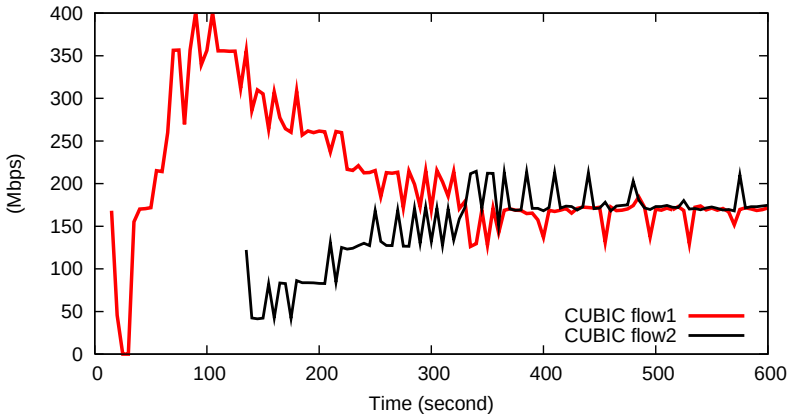
rate of 1.44Gb/s. In this case, CUBIC achieves exactly the same rate as Standard TCP, whereas HSTCP is about ten times more aggressive than Standard TCP.

5.3.3 CUBIC in action

Figure 5.7 shows the window curve of CUBIC over the running time. This graph is obtained by running a testbed experiment on a dumbbell network configuration with type V background traffic in both directions. The bottleneck capacity is 400Mb/s and the RTT is set to 240ms. Drop tail routers are used. There are two CUBIC flows, and they have the



(a) CUBIC window curves.

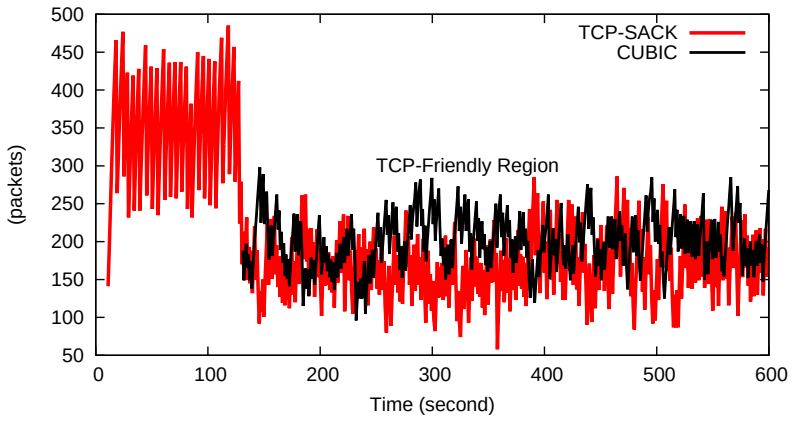


(b) Throughput of two CUBIC flows.

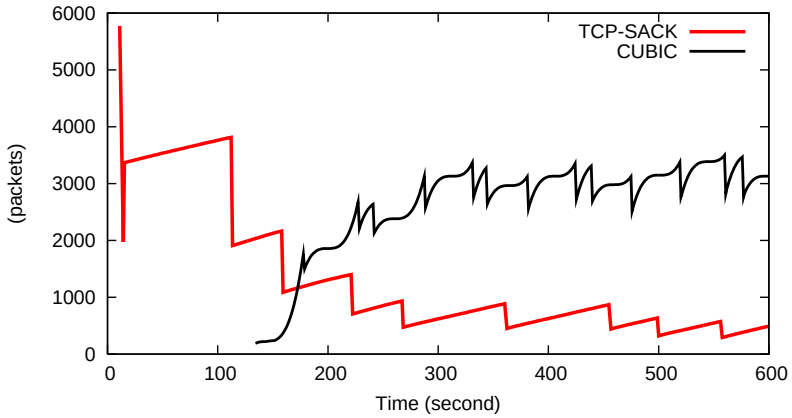
Figure 5.7: Two CUBIC flows with 246ms RTT.

same RTT and bottleneck. Note that the curves have plateaus around W_{max} which is the window size at the time of the last packet loss event. We observe that the two flows use all phases of CUBIC functions over the running time and both flows converge to a fair share within 200 seconds.

Figure 5.8 shows the friendliness of CUBIC with respect to TCP-SACK. In this experiment, we run one CUBIC flow with one TCP-SACK flow over a short-RTT network path (8ms) and a long-RTT network path (82ms), respectively. Background traffic (Type ll) is introduced in forward and backward direction of the dumbbell. Under the short-RTT (8ms) network where even TCP-SACK can use the full bandwidth of the path, CUBIC



(a) RTT 8ms.



(b) RTT 82ms.

Figure 5.8: One CUBIC flow and one TCP-SACK flow. Bandwidth is set to 400Mb/s.

operates in the TCP-friendly mode. Figure 5.8 (a) confirms that one CUBIC flow runs in the TCP-friendly mode and shares the bandwidth fairly with the other TCP-SACK flow by maintaining the congestion window of CUBIC similar to that of TCP-SACK. Under the long-RTT (82ms) network where Standard TCP has the under-utilization problem, CUBIC uses a cubic function to be scalable for this environment. Figure 5.8 (b) confirms that the CUBIC flow runs a cubic window growth function in contrast to the case with the short-RTT network where CUBIC is indistinguishable from TCP-SACK.

Figure 5.9 shows the experiment with four TCP-SACK flows and four CUBIC flows. For this experiment, we set the bandwidth to 400Mb/s, RTT to 40ms, and buffer

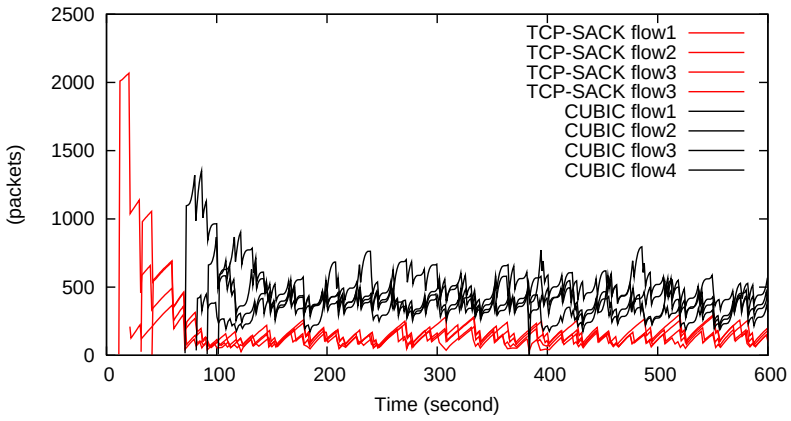


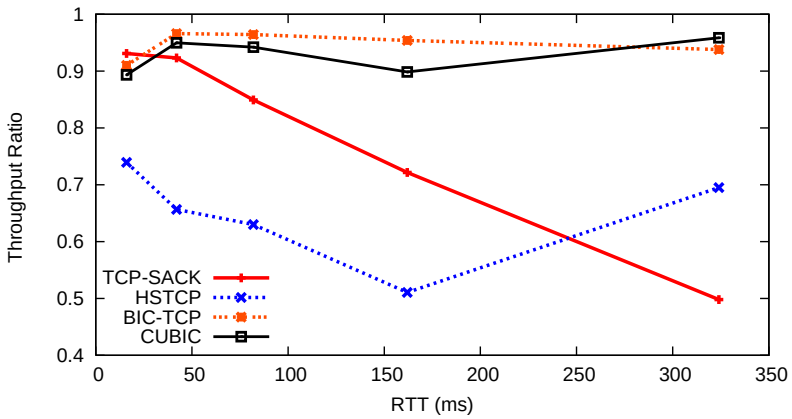
Figure 5.9: Four TCP-SACK flows and four CUBIC flows over 40ms RTT

size to 100% BDP of a flow. No background traffic is used in this experiment. Since many flows share the bottleneck, the possibility for phase effects is very low. We observe that four flows of CUBIC converge to a fair share within a short period of time. Their *cwnd* curves are very smooth and do not cause much disturbance to competing TCP flows. In this experiment, the total network utilization is around 95%: the four CUBIC flows take about 72% of the total bandwidth, the four TCP flows take 23%.

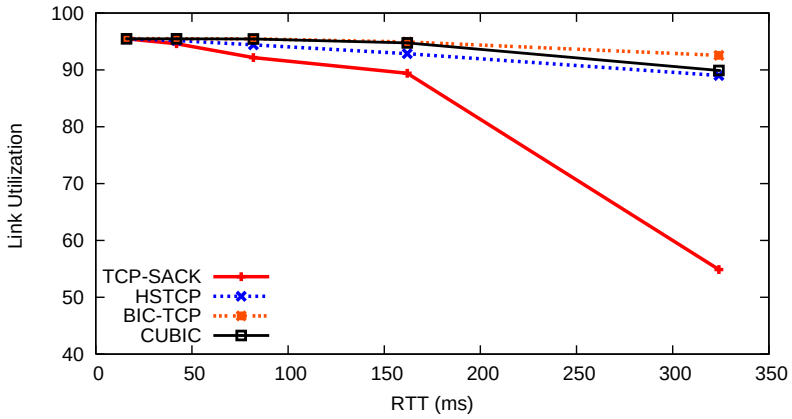
5.4 Experimental Evaluation

5.4.1 Experimental Setup

We use a dumbbell topology as shown in Figure 4.1 and use experimental methodology presented in Chapter 4. For this experiment, the maximum bandwidth of the bottleneck router is set to 400Mb/s. The bottleneck buffer size is set to 100% BDP unless otherwise explicitly specified. We use the Type II background traffic shown in Table 4.1 because medium size flows tend to fully execute slow start and increase the variability. We use the drop-tail router at the bottleneck.



(a) Intra-Protocol Fairness.

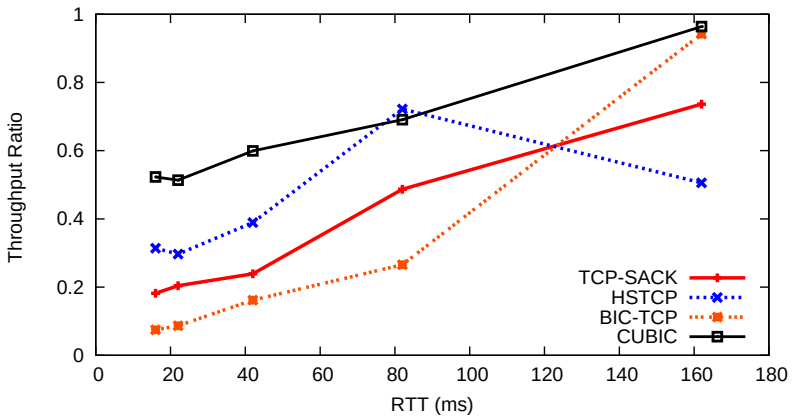


(b) Link Utilization.

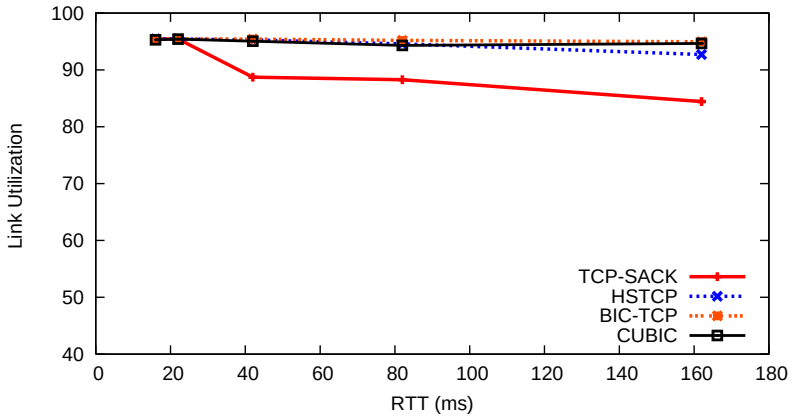
Figure 5.10: Intra-protocol fairness (a) and link utilization (b). The bottleneck bandwidth is set to 400Mb/s and 2Mbytes bottleneck buffer is used. RTT varies between 16ms and 324ms and two flows have the same RTT.

5.4.2 Intra-Protocol Fairness

We measure the intra-protocol fairness between two flows of a protocol with the same RTT. We use a throughput ratio between these two flows for representing the intra-protocol fairness. This metric represents a proportion of bandwidth shares between two flows of the same protocol. For this experiment, we vary RTTs between 16ms and 324ms and test CUBIC, BIC-TCP, HSTCP, and TCP-SACK protocols. Figure 5.10 (a) and 5.10 (b) show the intra-protocol fairness and link utilization for the tested protocols. CUBIC and BIC-TCP show a higher fairness index than TCP-SACK and HSTCP, representing



(a) Inter-RTT Fairness.



(b) Link Utilization.

Figure 5.11: Inter-RTT fairness. The bottleneck bandwidth is set to 400Mb/s and 2Mbytes buffer is used. One flow has a fixed RTT of 162ms and the other flow varies its RTT from 16ms to 162ms.

better fair sharing between the flows. With 16ms RTT, TCP-SACK shows the best fairness index indicating that Standard TCP works fairly well under small RTT networks. CUBIC, BIC-TCP, and HSTCP utilize the link regardless of RTTs while TCP-SACK suffers under-utilization with larger RTTs.

5.4.3 Inter-RTT Fairness

We measure the fairness in sharing the bottleneck bandwidth between two competing flows that have different RTTs. For this experiment, we fix the RTT of one

flow to 162ms and vary RTT of the other flow between 16ms and 162ms. This setting gives us the RTT ratio up to 10. We test CUBIC, BIC-TCP, HSTCP, and TCP-SACK protocols. Figure 5.11 (a) shows that TCP-SACK achieves RTT fairness linearly proportional to the inverse of the RTT ratio, which means that the short RTT flow has proportionally more bandwidth shares than the longer RTT flow. Even though there is no commonly accepted notion of RTT-fairness, we think the proportional fairness of TCP-SACK is desirable because long RTT flows tend to use more resources than short RTT flows. But some high-speed protocols are designed to provide an equal bandwidth sharing among the flows with different RTTs (e.g., H-TCP and FAST). Based on this notion of RTT fairness, if the RTT fairness of a protocol has a similar slope to TCP-SACK, we can say the protocol is "acceptable". Figure 5.11 (a) confirms that CUBIC has a similar slope with TCP-SACK but with a higher fairness ratio indicating a better share of resources (bandwidth) while HSTCP fails in achieving a similar slope. Even though BIC-TCP shows a similar slope to TCP-SACK, it shows the lowest fairness ratios among tested protocols. This is what CUBIC improves over BIC-TCP for RTT-fairness. We also observe that even though two HSTCP flows fully utilize the link regardless of their RTT ratio (See Figure 5.11 (b)), the slow convergence of HSTCP flows hinders even two flows of the same RTT from reaching a fair share within a reasonable amount of time.

5.4.4 Impact on standard TCP traffic

As many new high-speed TCP protocols modify the window growth function of TCP in a more scalable fashion, these new protocols tend to affect the performance of Standard TCP flows which share the same bottleneck link along the path. Since being fair to Standard TCP is critical to the safety of the protocol, we need to make sure that the window growth function of a new protocol does not unfairly affect the Standard TCP flows.

In this experiment, we measure how much these high-speed protocols steal bandwidth from competing TCP-SACK flows. By following the scenarios shown in the recent evaluation proposal [22], we first measure the throughput shares of four TCP-SACK flows when they are competing with the other four TCP-SACK flows. After that, we replace the four flows with a new protocol. We measure the share of TCP-SACK flows and the other four TCP variant flows at each run and report only the accumulated average of their bandwidth shares. We test CUBIC, BIC-TCP, HSTCP, and TCP-SACK.

Figure 5.12 (a) shows the relationship between RTT and the throughput share between new protocol flows and TCP-SACK flows. We fix the bottleneck bandwidth to 400Mb/s and vary RTT between 10ms and 160ms. Clearly, TCP-SACK flows do not fully utilize the bottleneck bandwidth as RTT increases due to its slow window growth function. With 400Mb/s and 160ms RTT, 8 TCP-SACK flows achieve around 80% of the link bandwidth, but the underutilization will be very serious for larger BDP paths and with small number of flows. CUBIC, BIC-TCP, and HSTCP fully utilize the link, thanks to their scalable window growth functions. All the tested high-speed protocols grab more bandwidth share from TCP-SACK as RTT increases. Also we confirm that CUBIC gives more room to TCP-SACK than BIC-TCP and HSTCP for whole range of tested RTTs while achieving full utilization of the path. It is one of the design objectives of CUBIC that it operates like BIC-TCP and be nice to other flows in the network. As RTT increases, CUBIC, BIC-TCP and HSTCP appropriate more bandwidth from TCP-SACK. Some of the amount of bandwidth they steal is from that which TCP-SACK doesn't utilize.

Figure 5.12 (b) and 5.12 (c) show the performance results regarding the TCP friendliness over short-RTT networks (10ms RTT) and long-RTT networks (100ms RTT), respectively. According to [42], under high loss rate regions (small-RTT networks) where TCP is behaving well, the protocol must behave like TCP, and under low loss rate regions (large-RTT networks) where TCP has a low utilization problem, it can use more bandwidth than TCP. As shown in Figure 5.12 (b), with 10ms RTT, we can see that TCP-SACK still uses the full bandwidth. In this region, based upon the above arguments, all high-speed protocols need to be friendly to TCP-SACK. Interestingly, CUBIC behaves more TCP-friendly even compared to TCP-SACK for certain bandwidths. Rather than stealing the bandwidth from TCP-SACK flows, CUBIC flows employ a window growth function that is comparable to TCP-SACK, so that competing TCP-SACK flows have the same chance with CUBIC flows for grabbing the bandwidth shares. However, BIC-TCP and HSTCP show a tendency to operate in a scalable mode (being more aggressive) as the link speed increases. Even though the graph doesn't show the results corresponding to the link speed beyond 400Mb/s, it is obvious that a scalable mode of BIC-TCP and HSTCP will deprive most of bandwidth share of TCP-SACK. Most high-speed TCP protocols including BIC-TCP and HSTCP achieve TCP friendliness by having some form of "TCP modes" during which they behave in the same way as TCP. BIC-TCP and HSTCP enter their TCP mode

when the window size is less than 14 and 38 packets, respectively. Therefore, even with 1ms RTT, if BDP is larger than 38 packets, HSTCP will operate in a scalable mode. This is the limitation when the protocol uses a fixed cutoff for detecting a TCP-friendly region. CUBIC defines a TCP-friendly region in real-time; therefore, CUBIC doesn't have this scalability problem.

Figure 5.12 (c) shows the results with 100ms RTT. All four protocols show reasonable friendliness to TCP. As the bandwidth gets larger than 10Mb/s, the throughput ratio drops quite rapidly. As CUBIC, like BIC-TCP and HSTCP, regards this operating region as an unfriendly TCP region and behaves as if it were scalable to this environment. CUBIC and BIC-TCP show a similar aggressiveness (slightly more aggressive[2] than HSTCP) especially for the bandwidth less than 100Mb/s. Through extensive testing [13], we confirm that this doesn't highly impact the performance of TCP-SACK.

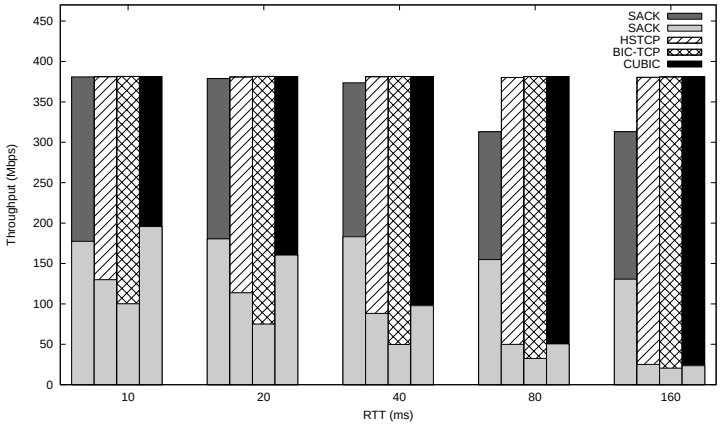
5.5 Conclusion

We propose a new TCP variant, called CUBIC, for fast and long distance networks. CUBIC is an enhanced version of BIC-TCP. It simplifies the BIC-TCP window control and improves its TCP-friendliness and RTT-fairness. CUBIC uses a cubic increase function in terms of the elapsed time since the last loss event. In order to provide fairness to Standard TCP, CUBIC also behaves like Standard TCP when the cubic window growth function is slower than Standard TCP. Furthermore, the real-time nature of the protocol keeps the window growth rate independent of RTT, which keeps the protocol TCP friendly under both short and long RTT paths. We show the details of Linux CUBIC algorithm and implementation. Through extensive testing, we confirm that CUBIC tackles the shortcomings of BIC-TCP and achieves fairly good intra-protocol fairness, RTT-fairness and TCP-friendliness.

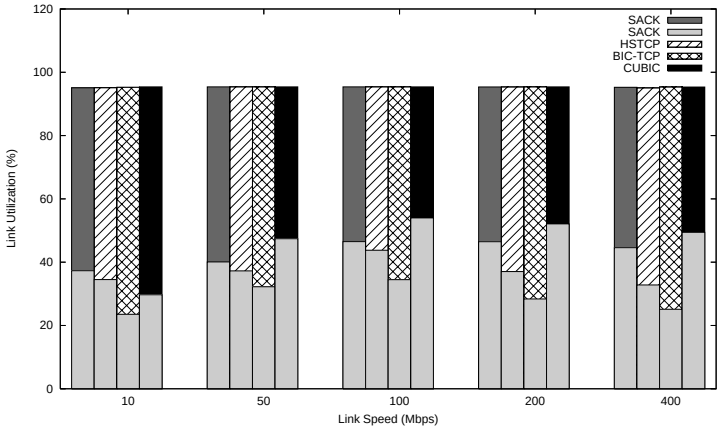
Since the acceptance of CUBIC as the default TCP congestion control algorithm in Linux, CUBIC has undergone several upgrades - improving the efficiency of cubic root calculation, removing the clamp in both convex and concave window growth regions, and

[2]We used the recent update of CUBIC (v2.2) which improved the scalability and convergence speed of the protocol, which doesn't clamp the increment in both convex and concave regions. A slight increase of aggressiveness is the trade-off between scalability and TCP-friendliness. Our extensive testing confirms that CUBIC has better scalability and convergence speed with this small change (trade-off) while obtaining reasonable TCP-friendliness.

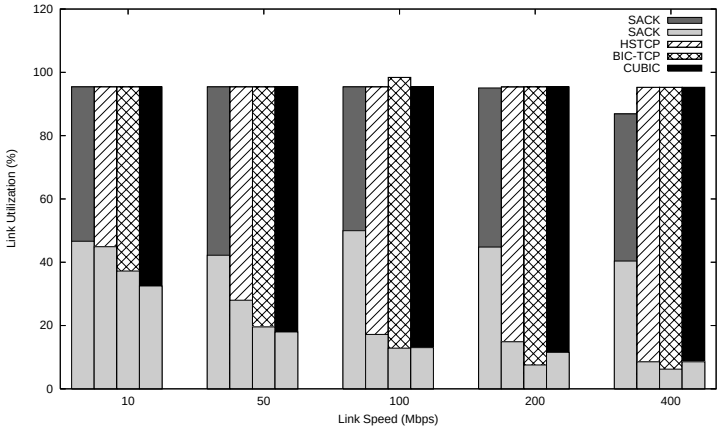
the adoption of a new slow start algorithm. Future work would involve providing some level of loss tolerance to CUBIC, so that CUBIC can also be used for error-prone networks such as lossy WAN links and wireless networks.



(a) Bandwidth 400Mb/s with varying RTT



(b) RTT 10ms with varying bandwidth



(c) RTT 100ms with varying bandwidth

Figure 5.12: Impact on standard TCP traffic.

Chapter 6

Taming the Elephants: New TCP Slow Start

Standard TCP doubles congestion window (cwnd) for every round-trip time (RTT) during slow start. However, the exponential growth of cwnd results in burst packet losses. Since the cwnd overshoots beyond the path capacity as large as the whole BDP, in large BDP networks, this overshoot causes strong disruption 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 this overshoot also add a great deal of burden to the end systems for loss recovery and this burden often translates into consecutive timeouts and a long blackout periods without transmission.

The selective acknowledgement (SACK) option [79, 50, 30] relieves some of these burdens. As SACK informs the sender about 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 [80]. However, for a large BDP network where a large number of packets are in flight, the processing overhead of SACK information at the end points can be quite overwhelming. This is because each SACK block invokes a search into the large packet buffers of the sender for the acknowledged packets in the block, and every recovery of a lost packet causes the same search at the receiver end. During fast recovery, every packet reception generates a new SACK packet. Given that the size of cwnd can be quite large (sometimes, beyond 100,000), the overhead of such a search can be sometimes overwhelming. This system overload is devastating: it can prevent

the system from responding to other services and processes, and it can cause multiple timeouts (as even packet transmissions and receptions are delayed) and a long period of zero throughput. Many operating systems attempt to optimize 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 overload or a reduced number of SACKs extremely slows down loss recovery resulting in a blackout period of no transmission greater than 100 seconds. These 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 since slow starts are not necessarily the only cause 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. We primarily focus 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 pitfalls. We then present a new slow start algorithm, called Hybrid Slow Start (HyStart) [55] 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 on the Internet. We demonstrate its performance by implementing HyStart and various other proposed solutions on the latest Linux kernel and testing them with Linux, FreeBSD, and Windows XP receivers in both real production networks and in a laboratory testbed. We report superior performance of HyStart over the other solutions in terms of network and CPU utilization.

6.1 Motivations

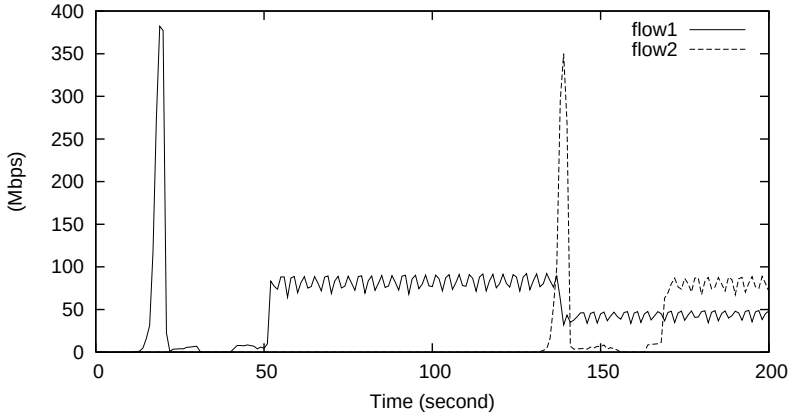
In this section, we identify two main reasons for the poor slow-start performance of current commercial TCP stacks. One is the overwhelming processing load during slow start in large BDP networks. The other is the proprietary optimizations of slow start performed by the developers of various operating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

6.1.1 Processing overload during Slow Start

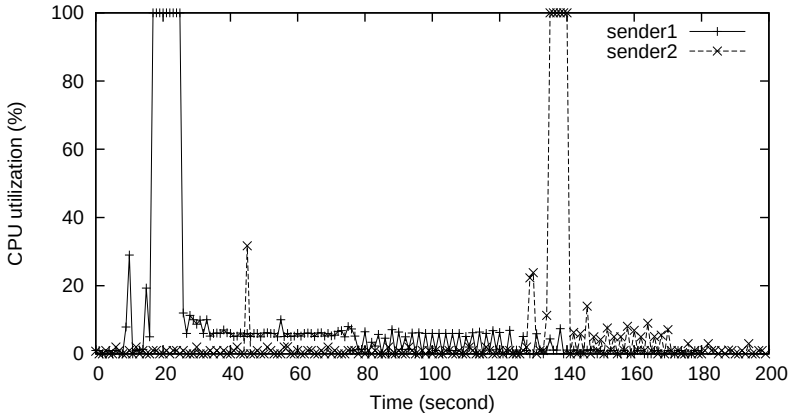
We investigate how the system overhead can affect TCP performance during slow start. System overload is frequently observed during slow start which is followed by multiple timeouts or long blackouts. The problem occurs consistently with all three popular operating systems, Linux, Windows XP and FreeBSD. Figure 6.1 (a), (b) and (c) illustrates the throughput observed at a router when two TCP-SACK flows join at different times in a network of 400Mb/s bandwidth and 240ms 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 a typical and repeatable phenomenon observed approximately 30 to 40% of the times we run the experiment. (The results of repeated experiments are given in Section 6.4.) The experimental setup of our testbed is detailed in Chapter 4. From Figure 6.1, both flows have almost zero throughput for more than 30 seconds after a timeout. These blackouts follow after the saturation of CPU utilization, occasionally at both senders and receivers. When the system reaches overload, it cannot react fast enough to perform loss recovery. This results in timeouts. Repeated losses of retransmitted packets during timeouts also cause back-to-back timeouts with exponential backoff of RTO (retransmission timeout) timers where the senders do not transmit any packets.

Processing overhead at TCP senders

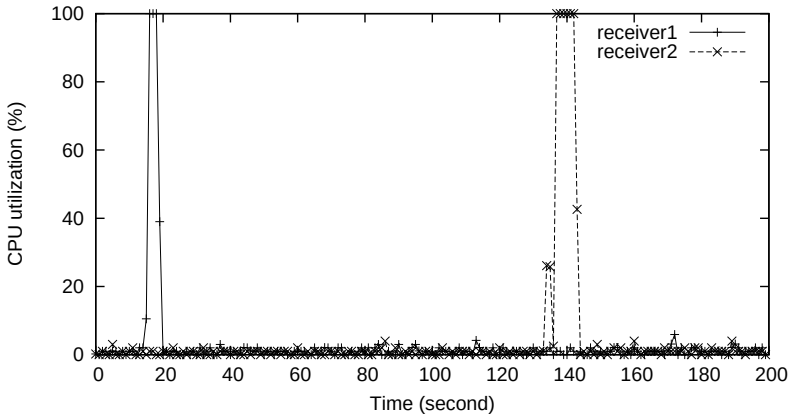
We first examine the SACK processing overhead of the TCP sender in Linux. Figure 6.2 illustrates how the Linux TCP sender processes an ACK packet. The *tcp_ack*() is called at the reception of ACK, the function checks whether the ACK number is within the left and right edges of a congestion window - *snd_una* and *snd_nxt*. Then



(a) Throughput



(b) CPU utilization of two senders



(c) CPU utilization of two receivers

Figure 6.1: The example of two black-out periods. The bandwidth, one-way delay, and buffer sizes are set to 400Mb/s, 120ms and 100% BDP of a network.

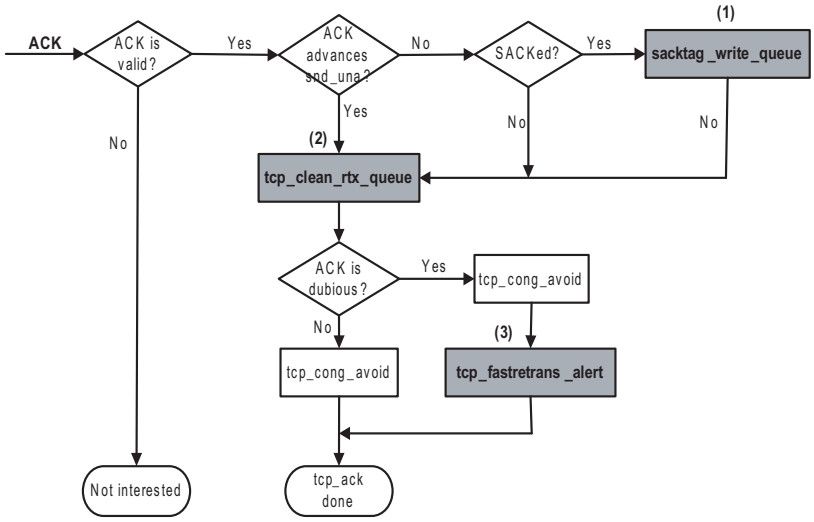


Figure 6.2: Linux TCP ACK processing

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 ensuring that the variable, *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 the variable *packets_out* is reduced by the number of the freed packets. The *tcp_fastretrans_alert*() executes fast retransmit and updates the scoreboard that keeps track of lost and acknowledged packets. For every ACK, *tcp_cong_avoid*() is called which then increases *cwnd* by entering slow start and congestion avoidance phases.

The retransmit queue is implemented using a linked list to hold all of the packets currently in flight to the receiver. Each SACKed packet is searched 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 6.2 are the most CPU-intensive as they need multiple traversals of the retransmit queue. Suppose that the TCP sender has sent W packets in the retransmit queue and receives $\frac{W}{2}$ delayed ACKs in one RTT round. Let us examine each of these functions below.

- *sacktag_write_queue*(): SACK contains up to four SACK information blocks, each of

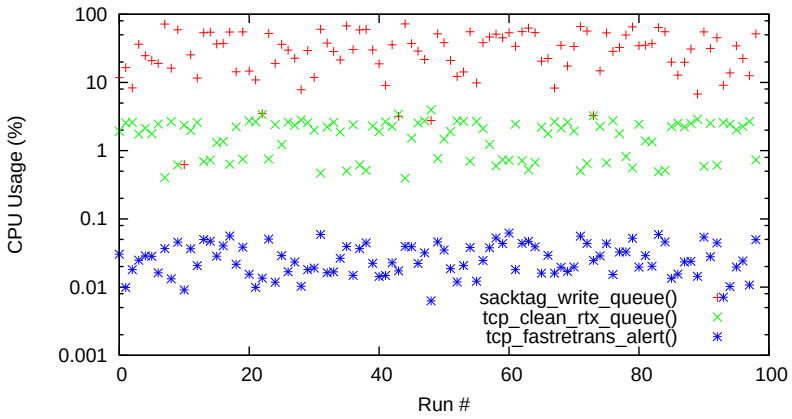
which consists of a block of the sequence numbers of packets received out of sequence by the receiver. For every SACK, it searches in the retransmit queue for the packets whose sequence number is in between each SACK block and updates their states. In the worst case, one SACK block causes a traversal 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.

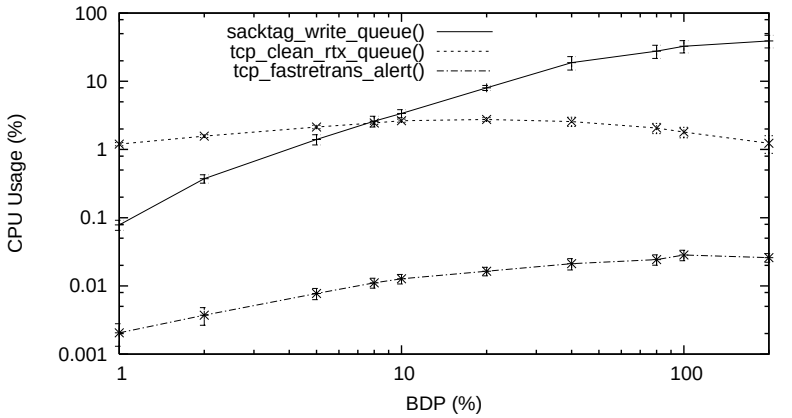
- *tcp_fastretrans_alert*(): Invoked at the reception of a duplicate ACK, it updates a variable $left\_out$ to account for the number of lost packets. It usually marks the first packet in the retransmit queue to be retransmitted first. In the worst case, a retransmission timeout happens and all packets in the retransmit queue are timed out. Then. it needs as much as 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 [10] 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 6.1 for profiling. Figure 6.3 (a) shows the results of 100 runs and we plot them on a log scale. The *sacktag_write_queue*() consumes between 10% and 100% of the CPU clocks. Also the overhead of *tcp_clean_rtx_queue*() is larger than *tcp_fastretrans_alert*() because of the cost of freeing memory, but it is significantly less than that of *sacktag_write_queue*().

To see the relationship between W (*cwnd*) and CPU utilization, we change the bottleneck buffer size from 1% to 200% of the path BDP while maintaining the same testing parameters. Figure 6.3 (b) plots the CPU usage of the three functions as W increases on a log scale. The CPU usage of *tcp_clean_rtx_queue*() is quite independent of the buffer size while *tcp_fastretrans_alert*() shows a small increase of CPU time. We note that *sacktag_write_queue*() requires significantly more CPU time as W increases.



(a) SACK processing overhead



(b) BDP (W) vs. CPU utilization

Figure 6.3: CPU utilization of the three functions in TCP SACK processing

Processing overhead at TCP receivers

In Linux TCP, *tcp_data_queue*() processes data in received packets. If packets are received in order, it copies the data in the packets to the user space. When packets arrive out of order, it puts the packets into the out-of-order queue. The packets in the out-of-order-queue can only be freed when the left edge of the receiver window advances. When receiving a retransmitted packet, the receiver searches the out-of-order queue to place the packet in the right order. Linux TCP uses a linked list for the out-of-order queue. The increase of the right edge of the congestion window during fast recovery does not significantly affect the performance as it typically goes to the end of the queue. However, with a large number

of retransmitted packets, the processing overload occurs when each retransmitted packet causes a traversal of the queue to place the packet in the right place.

We profile the receiver kernel and measure the system clocks of *tcp_data_queue*() of the second receiver shown in Figure 6.1 (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 a state of 100% CPU utilization at around the 140-th second in Figure 6.1 (c). The data copy to the user takes only 2% of CPU time.

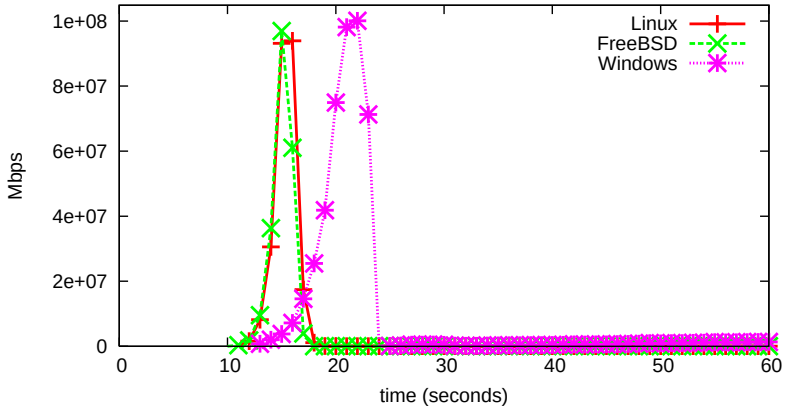
6.1.2 Protocol Misbehavior during Slow Start

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

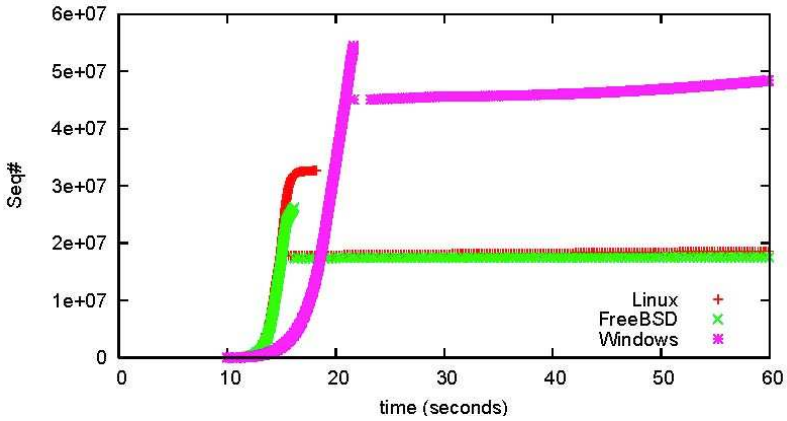
Figure 6.4 shows the results of the experiment involving TCP-NewReno senders and receivers of various operating systems. All stacks show very poor performance – after the overshooting of cwnd during slow-start, all stacks enter fast retransmit and recovery, but their recovery speed is very slow. This happens because these stacks implement "*Slow-but-Steady variant of NewReno*" (SS-NewReno) [47] where the TCP sender resets the timeout timer whenever a partial acknowledgment indicating the left edge of the window arrives. This effectively prevents the sender from entering 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 of the lost packets.

This problem of a slow ramp-up after the use of slow start does not occur in TCP-SACK since the sender is better informed about lost packets beyond the left-edge and retransmits all of the lost packets almost immediately. Figure 6.5 shows the performance of TCP-SACK for the three operating systems.[1] We find that all stacks now enter a timeout

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



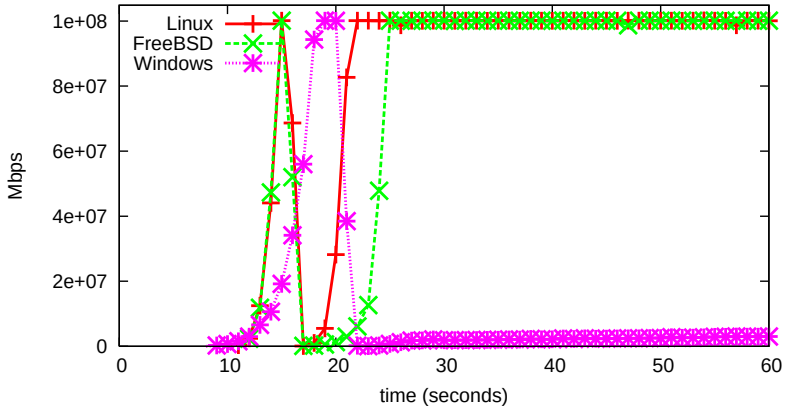
(a) Throughput



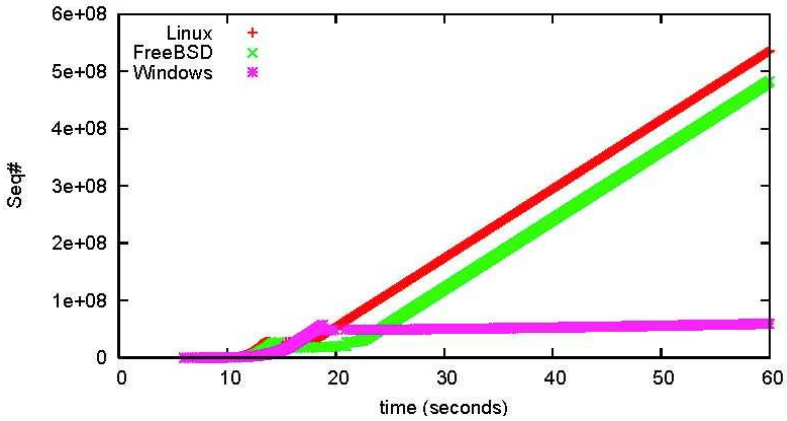
(b) Time vs. sequence numbers

Figure 6.4: TCP-NewReno on the three systems.

quickly. While FreeBSD and Linux recover relatively quickly, Windows XP shows a very slow recovery. This is because the number of SACK blocks sent by the Windows XP receiver is limited and after a specified threshold, the receiver simply reports only cumulative ACKs. This is an optimization added by the Windows developers for various purposes including reducing system overloads during fast recovery or limiting buffering at the receiver. This forces the sender to act like TCP-NewReno, showing similar behavior as in Figure 6.4. We can also obtain the same performance result even if we use a Linux sender along with a Windows XP receiver.



(a) Throughput



(b) Time vs. sequence numbers

Figure 6.5: TCP-SACK on the three systems.

6.2 Existing solutions

There are basically two approaches to fixing the start-up performance of TCP. One is a reactive approach. While letting burst losses occur, it devises better techniques to handle many multiple packet losses more efficiently and effectively. Most operating systems use this approach. This approach is important as it can be applied to any situation where burst losses occur (not 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. Section 3.2 describes the earlier work related to the second

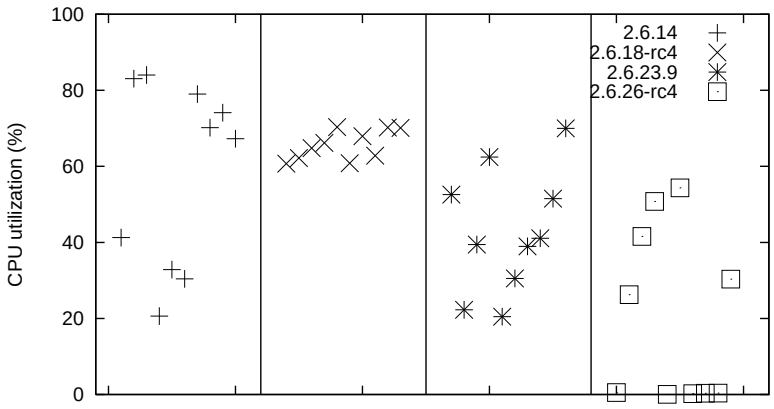


Figure 6.6: Improvement of SACK processing on Linux after slow-start in a large BDP network. The SACK processing have been improved over the evolution of Linux kernel.

approach. In this section, we discuss existing techniques for the first approach.

6.2.1 Linux: Linked-list optimization

Since Linux 2.6.15, SACK processing has undergone several improvements, involving better caching and data structures in order to reduce the look-up time in the retransmit queue. Figure 6.6 shows the improvement of Linux SACK processing over the evolution of the Linux kernel for the same experiment shown in Figure 6.1.

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 6.1 with different kernel versions and measure the CPU utilization of *sacktag_write_queue*(). We can see the overhead reduces as the kernel version increases. However, the SACK processing overhead has been reduced by only about 20% in the latest version. We recognize that improving the system bottlenecks by efficiently optimizing data structures is valuable but it still requires further improvement. As we have seen in Figure 6.1, the latest version of Linux still consumes almost 100% of CPU time during fast recovery after slow start in large BDP networks.

6.2.2 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 [31] and TCP-Westwood [36] and can prevent the overshooting experienced with 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 frequent under-estimation because of noise in the RTT measurement. 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 ends slow start, often prematurely, as the estimation technique is too simple and often underestimates the BDP. For instance, we find that TCP-SACK with this technique shows 15.6% link utilization, in the 100Mb/s and 120ms one-way delay networks. We have additional performance results of TCP-Vegas in the next section.

6.2.3 Windows - Suppressing SACK options

When a TCP-SACK receiver on Windows XP has many SACK blocks to report, it suppresses the SACK options and sends only cumulative ACKs. The TCP-SACK receiver discards all the out-of-order packets received after reaching this limit. Sending only cumulative ACKs prevents the TCP sender from being overloaded. But the sender acts like TCP-NewReno and exhibits very slow recovery as shown in Figure 6.4.

6.3 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 a lower system load. The algorithm does not change the doubling of cwnd during slow start, but based on clues from ACK spacing and round-trip delays, it heuristically finds safe exit points at which it can finish slow start and move to congestion avoidance before cwnd overshooting. When packet losses occur during slow start, HyStart behaves in the same way as the original slow start protocol. HyStart is a plugin to the sender side of TCP and is easy to implement because it uses only TCP state variables available in common TCP stacks.

6.3.1 Safe Exit Points

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

The safe exit point corresponds to the size of the cwnd before slow start finishes safely without incurring losses and without low network utilization. Let the unused available bandwidth of the forward path, the minimum forward path one-way delay and available buffer space of the forward path be B, D_{min} and S respectively. Then a safe exit point must be less than $C = B \times D_{min} + S$. If cwnd gets larger than C, then packet losses occur. Slow start cannot finish arbitrarily before this upper bound which will cause low network utilization. We need a lower bound. After slow start, congestion avoidance will grow cwnd if there is no loss. During steady state where the average capacity of a path does not change, standard TCP reduces cwnd by half $(C/2)$ for fast recovery, after which congestion avoidance takes over to restore cwnd to half $(C/2)$. This also happens during a timeout where ssthresh is set to $(C/2)$. Therefore, it is reasonable to set the lower bound for a safe exit point at $C/2$ 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 between $B \times D_{min} \times \beta$ and C where β is the multiplicative decrease factor of cwnd during fast recovery (i.e., $0 < \beta < 1$). Standard TCP sets β at 0.5 and other high speed variants use a value larger than 0.5.

6.3.2 Algorithm Description

HyStart uses two sets of information - the time space between consecutively received ACK packets and round-trip delays then applies heuristic methods to look for an

indication that the current value of cwnd is larger than the available bandwidth. HyStart uses a passive measurement and estimation technique – it does not inject a probe packet of its own. Below we describe the two methods. Both run independently and concurrently and slow start exits when either of them detects an exit point. Figure 6.7 shows its pseudo code.

ACK train length

Estimating unused bandwidth in a path has been an active topic of research [35, 66, 20]. Many bandwidth estimation techniques, such as packet-pair [71] and packet train [39] use active probing to estimate bandwidth by sending probes back-to-back in a short period of time.

Dovrolis et al. [39] recently showed that the mean of packet train dispersion can be translated into Average Dispersion Rate (ADR), a rate 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.

$$b(N) = \frac{(N-1)L}{\Delta(N)} \tag{6.1}$$

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

To remedy these problems, we propose the following approach. For now, suppose

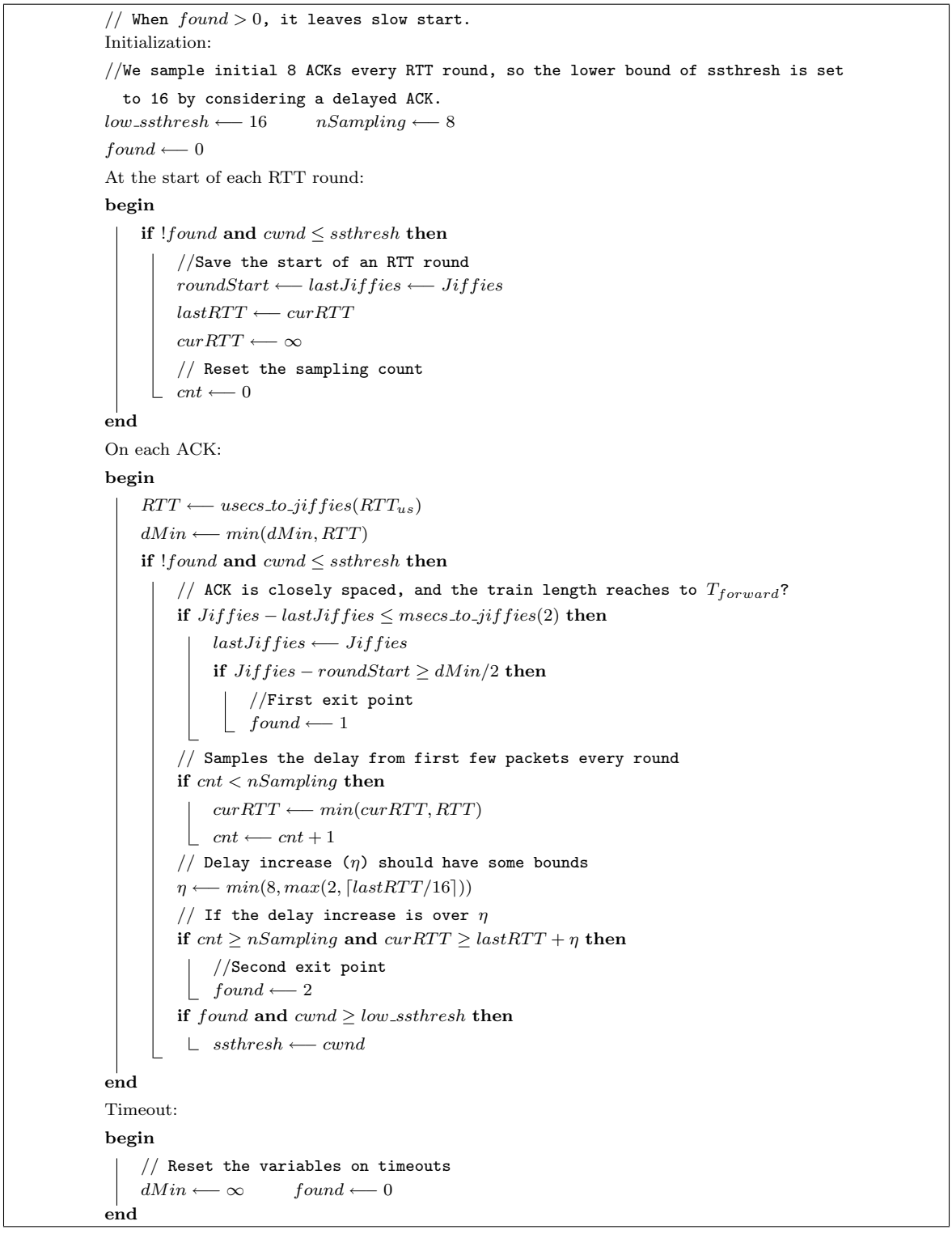


Figure 6.7: Hybrid Slow Start algorithm (HyStart)

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

1. We are able to have data packets transmitted during slow start as packet probes for a packet train. During slow start, many packets are sent in burst. We can use those packets which are transmitted within the same window as a packet train. This removes the need for active probes. To estimate $\Delta(N)$ from the sender side, we use the train of ACKs received in response to a packet train. Figure 6.8 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} on the sender side. We approximate it by dividing the measured minimum RTT by two. This approximation is not accurate when the path is asymmetric. Below, we explain why this does not lead to under or over-estimation of the capacity.

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

Second, the receiver may use a delayed ACK scheme where ACKs are transmitted only for alternate packets received. In some systems, ACKs are arbitrarily delayed, but ACKs are always generated immediately at the reception of a packet (however, not necessarily at the reception of every packet). The ACK delay does not affect the performance of HyStart greatly because $\Lambda(N)$ is computed by taking the time difference between the reception times of the first and last ACKs in a train. Since ACKs are triggered immediately after the reception of a packet, $\Lambda(N)$ may contain the delay for the last delayed ACK. Since an ACK train is typically large, this delay does not move the exit point beyond its safety bounds. To alleviate this problem, we filter out the ACK train length samples if the last ACK of a train is significantly delayed and mixed with the ACKs for the next train.

Third, path delay asymmetry may significantly thwart the correct estimation of safe exit points. Consider this situation. Suppose that a and b are the forward and reverse path one-way delays respectively. We estimate a by $(a+b)/2$. Suppose again that K is the BDP of the forward path and K' is our estimated BDP, and $\Lambda(N) = \Delta(N)$. Then if $a = b$, 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 6.3.1, our scheme satisfies the bounds if K'/K is larger than β, but less than $1 + S/K$. Since β is 0.5 in standard TCP, as long as b is larger than 0, it satisfies the lower bound. Thus, when the reverse path delay is much smaller than the forward path delay, under-utilization is very unlikely. For the upper bound, suppose that $S = \alpha K$. Then if K'/K is less than $1 + \alpha$, our scheme meets the upper bound which means that b must be less than $a(2\alpha + 1)$. If $\alpha = 1$ (i.e., S is as large as the BDP), b can be as large as $3a$ to meet the upper bound. According to a recent measurement study of delay asymmetry on the Internet [85], the fraction of paths whose reverse path delays are 3 times longer than the forward path delays is less than 5%. Thus, for those fractions of Internet paths, HyStart behaves like standard slow start since it may overestimate the BDP and the overshooting of cwnd will trigger packet losses as in standard slow start.

Delay increase

When a path is not completely empty and one or more flows are competing for the same path, it may not be possible to measure the minimum RTT of the path. So the above ACK train based technique would be less effective. To handle this situation, we use increase in round-trip delays as another metric to find the safe exit point. However, as TCP

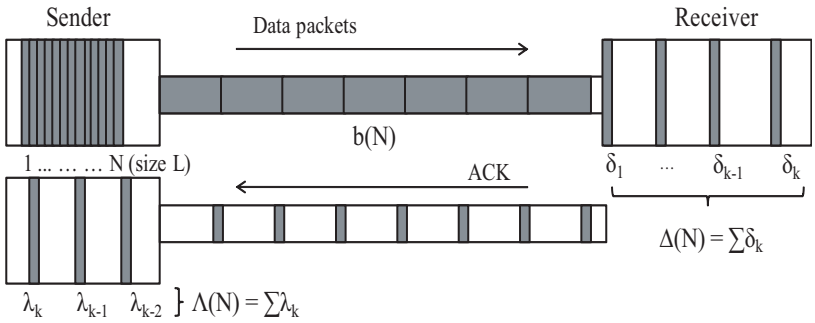


Figure 6.8: ACK train measurement

sends packets in bursts, 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 [31]. We remedy this problem by using the RTT samples of a few ACKs at the beginning of each ACK train. Since packets arriving at the beginning of each train do not suffer from queuing caused by packet bursts, these samples return more accurate estimations of persistent queuing delays. Suppose that RTT_k is the average of RTTs of a few packets in the beginning of the 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 longer than the forward path delays.

6.4 Experimental Evaluation

6.4.1 Experimental Setup

We use a dumbbell topology as shown in Figure 4.1 and use experimental methodology presented in Chapter 4. For this experiment, we set the bandwidth of the bottleneck router between 100Mb/s and 400Mb/s, depending on the conditions. The bottleneck buffer size is set to 100% BDP unless otherwise specified. If we use background traffic in the experiment, we use Type II background traffic as shown in Table 4.1 because

medium size flows tend to fully execute slow start and increase the variability. We use a drop-tail router at the bottleneck.

6.4.2 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 100Mb/s, 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 6.9 presents the trajectories of cwnd and ssthresh of six different slow-start protocols discussed in Section 6.4.2. The original standard slow start of TCP-SACK (SS) (a) shows a high burst of packets at around ten seconds and experiences a high rate of loss. Limited slow start (LSS) (b) reduces the burst losses observed in SS (a) by limiting the increment of congestion window in one RTT. With HyStart (c), using the information of the ACK train length, the first flow finishes slow start at approximately packet 870 at which point, cwnd reaches the BDP of the path. The second flow uses a delay increase to gauge the congestion on the path caused by the first flow, and leaves slow-start early on. Adaptive Start (AStart) (d) calculates ERE upon receiving ACKs and sets ssthresh to ERE if ERE is larger than cwnd during slow-start. But ERE is consistently smaller than cwnd in this experiment. As a result, AStart shows the same overshooting behavior as SS. The packet-pair slow-start (PSS) (e) shows an inconsistent estimation of path capacity for each run because of the lack of high resolution clocks and real-time interrupt handling. The two flows estimate the capacity of 100Mb/s path to be 248Mb/s (2300 packets) and 308Mb/s (2800 packets), respectively. Also, the modified slow-start of Vegas (VStart)(f) terminates slow-start prematurely.

6.4.3 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 send an ACK for every two data packets received. When an ACK is not delayed, the spacing between

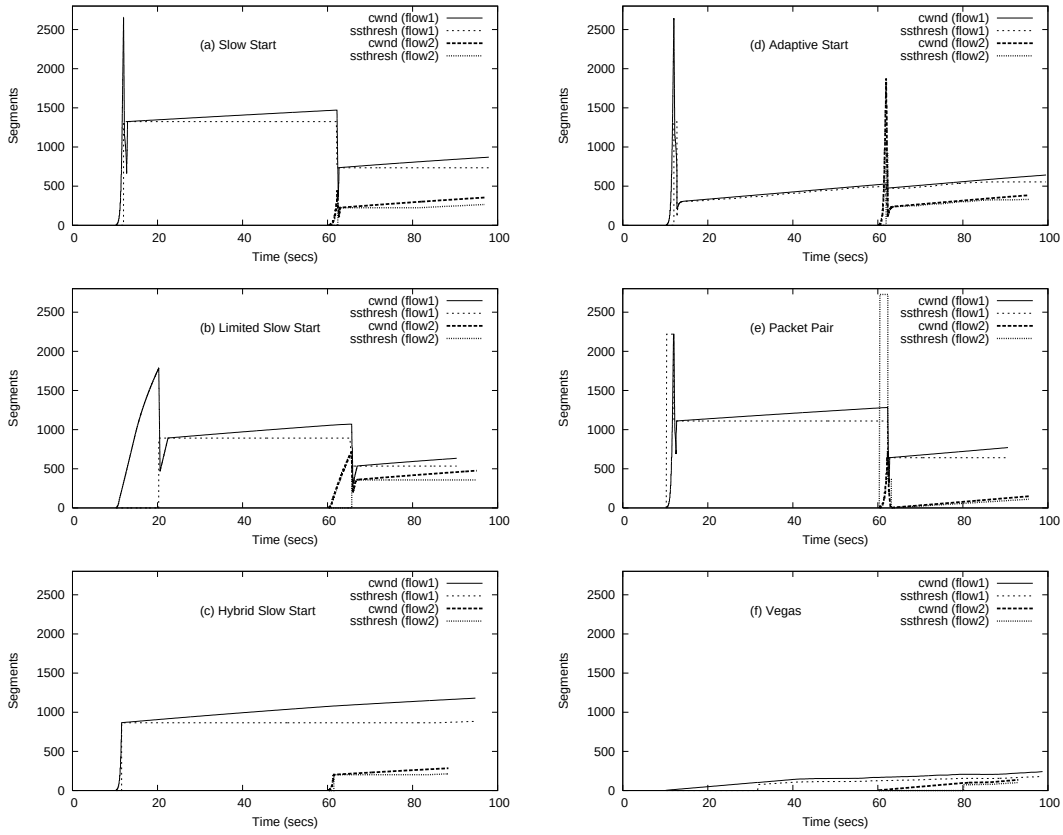
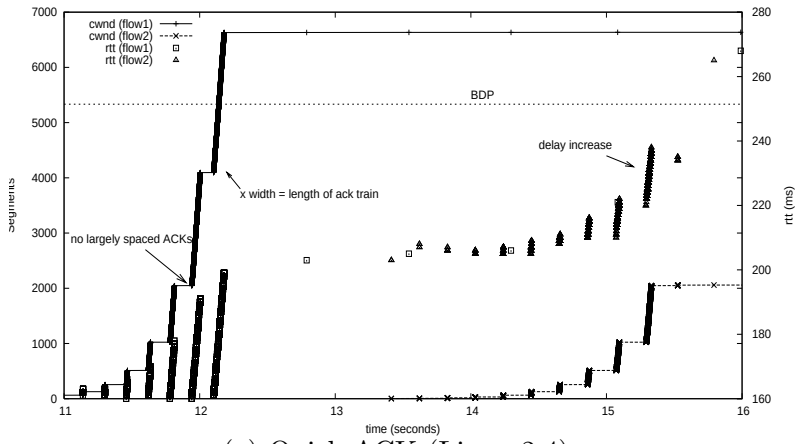


Figure 6.9: Two TCP-SACK flows with five different Slow Start proposals. The BDP for this experiment is around 883 packets. Therefore, when cwnd is between 883 and 1766 packets, the link is fully utilized.

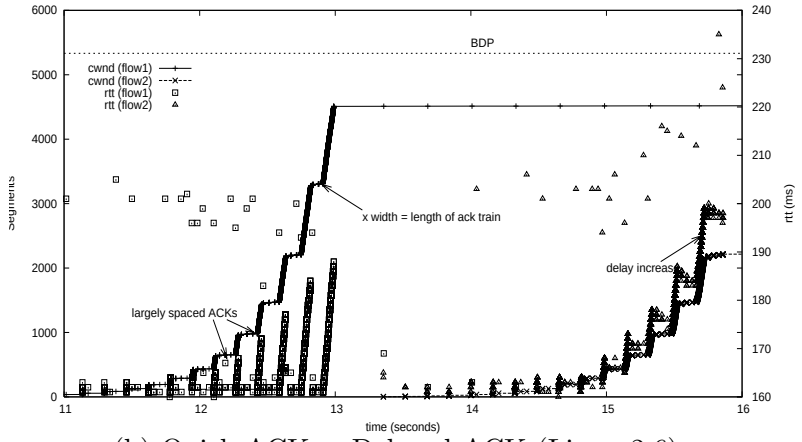
consecutive ACKs is small and consistent. This makes a packet-train measurement effective. A delayed ACK may disturb the estimation of the ACK train length.

In this experiment, we introduce background traffic as the background traffic may disturb the behavior of the algorithm by varying available link bandwidths. The bottleneck bandwidth is set to 400Mb/s and the buffer size is set to 100% BDP. Two TCP-SACK flows with the same one way delay of 80ms start within the first 10 seconds of the testing. We use the Type II background traffic that is introduced in both forward and reverse directions of the path.

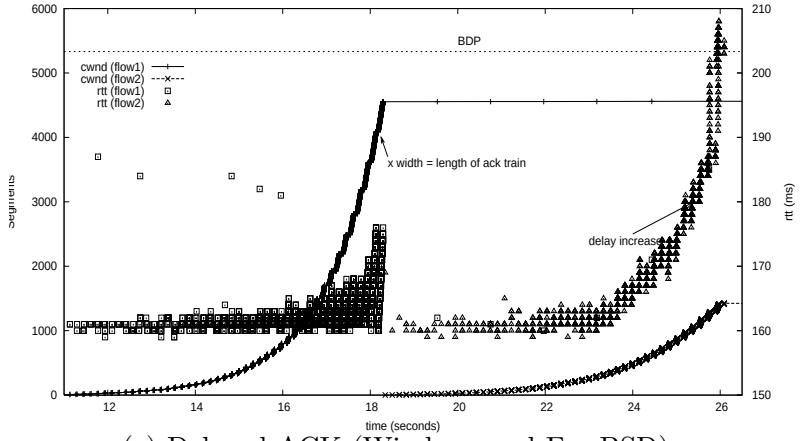
Figure 6.10 tracks the cwnd and RTT of two flows for three different ACK schemes while running HyStart. Figure 6.10 (a) confirms that no delayed ACKs are found in between



(a) Quick ACK (Linux 2.4)



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



(c) Delayed ACK (Windows and FreeBSD)

Figure 6.10: Two TCP-SACK flows with HyStart, by varying the ACK schemes on the receivers.

two consecutive RTT rounds. Hence, we know it correctly calculates the ACK train length. The exit point is the round in which cwnd crosses over the BDP of the forward path. Linux, however, deliberately sends a quick ACK for up to an initial 16 segments to quickly ramp up the rate during slow-start. Even if Linux employs both quick and delayed ACKs, the ACK train is mostly composed of the closely spaced ACKs sent in a burst due to the initial quick ACKs. Figure 6.10 (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 6.3.2 filters out any significantly delayed ACK packets in a train. This allows HyStart to correctly estimate the BDP of the network. HyStart completes the slow start phase only one round before cwnd reaches the BDP. FreeBSD and Windows XP send delayed ACKs from the beginning of a connection and consequently, ACKs are spread over an entire RTT round. Even in this circumstance, our filtering scheme works fairly well. Figure 6.10 (c) shows that HyStart finishes slow start only one round before cwnd reaches the BDP.

6.4.4 Integration with high-speed protocols

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

Figures 6.11 and 6.12 present the results of two CUBIC [57] flows with and without HyStart, respectively in the same experimental setup as in Figure 6.1. We plot the trajectory of cwnd and ssthresh and the throughput measured in the bottleneck router. In the experiment of CUBIC with SS, the first flow shows the initial timeout of 20 seconds because it overshoots up to 20,000 packets which are twice the BDP of the network. When the second flow joins the network, the path is already fully utilized. But the second flow perturbs the link utilization with its exponential probing 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 the 130th second, detects the congestion of the path using the delay increase and

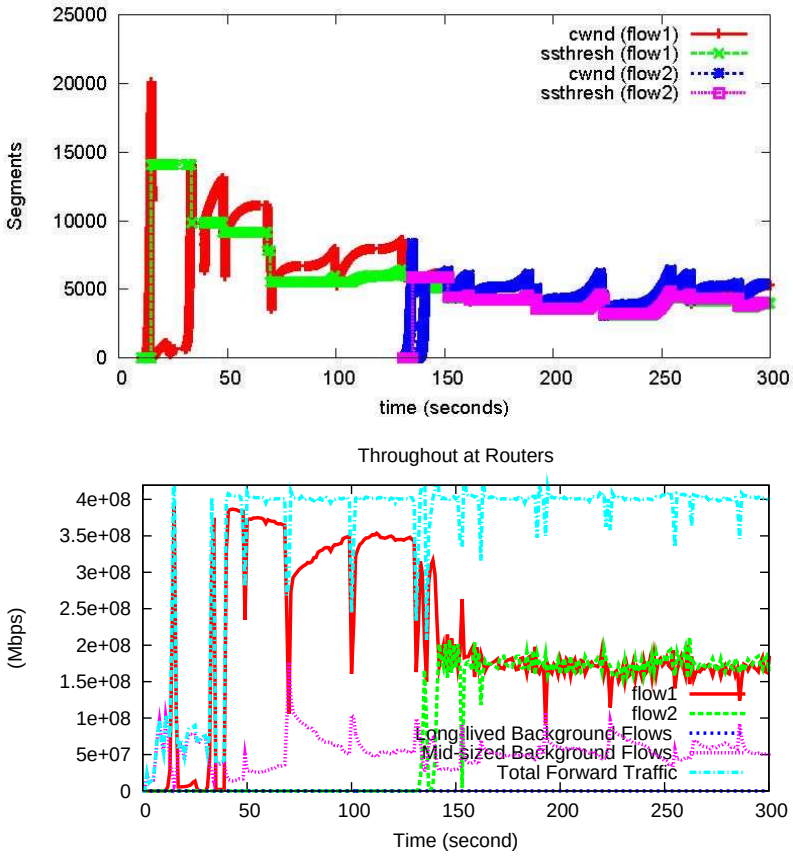


Figure 6.11: CUBIC with standard slow start. The first flow experiences heavy packet losses around the first 10 seconds and multiple timeouts over a 20 second period.

exits to the congestion avoidance phase of CUBIC.

6.4.5 Testing with other OS Receivers

In this experiment, we evaluate the performance of three representative slow-start algorithms, HyStart, SS and LSS, by varying the receiver side operating systems. We run two TCP flows. The sender machines are fixed to Linux 2.6.23.9. We set the bottleneck bandwidth to 400Mb/s and RTT to 100ms. We add Type II background traffic to the bottleneck. Note that with FreeBSD and Windows XP receivers, the Linux sender behaves like TCP-NewReno as explained in Section 6.1.2. Figure 6.13 shows that HyStart works much better than SS, regardless of the operating systems of the receivers. LSS works

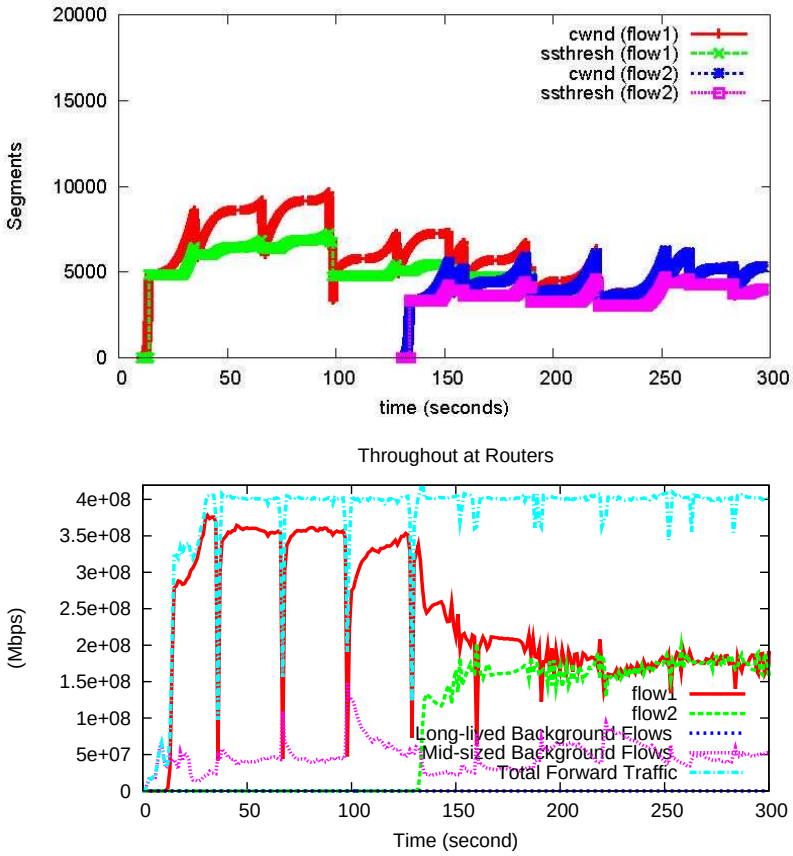


Figure 6.12: CUBIC with HyStart. HyStart exits from slow start before packet losses occur.

relatively as well as HyStart in this setup. Lower performance of SS under Windows XP and FreeBSD results because these operating systems force the sender to behave like TCP-NewReno and they are unable to handle high packet losses caused by the overshooting of cwnd in SS. However, HyStart finishes slow start before this overshooting happens. Thus, even if the sender behaves like TCP-NewReno, since there are no heavy packet losses, it shows a reasonably good performance.

6.4.6 More diverse experimental settings

In this experiment, we compare the performance between all slow-start proposals discussed in Section 6.4.2 and HyStart under more diverse experimental settings. To measure the start-up throughput, we use two flows starting at the 10th and 40th seconds,

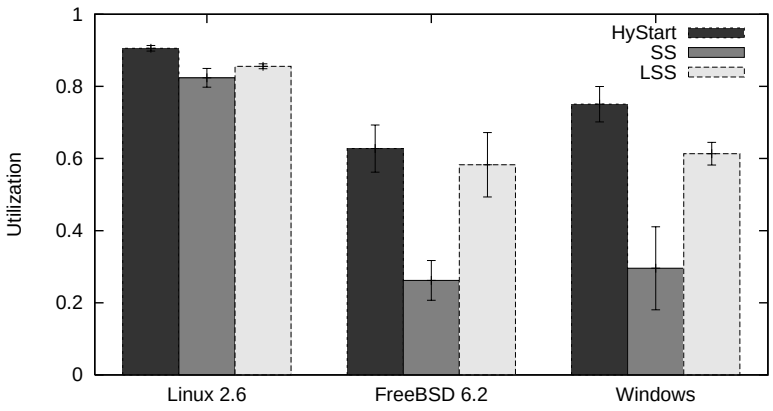
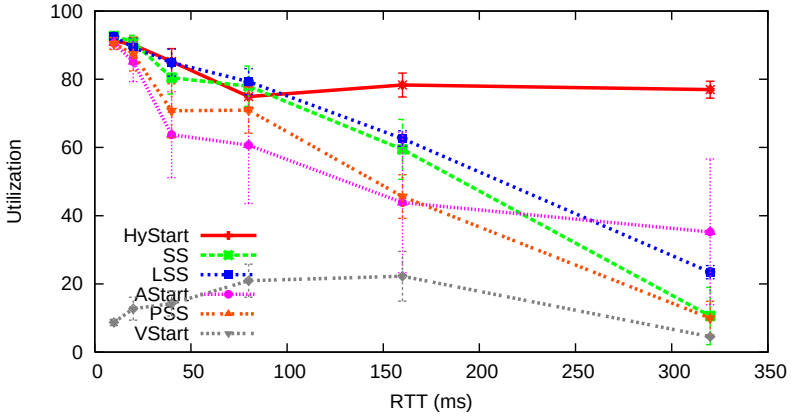


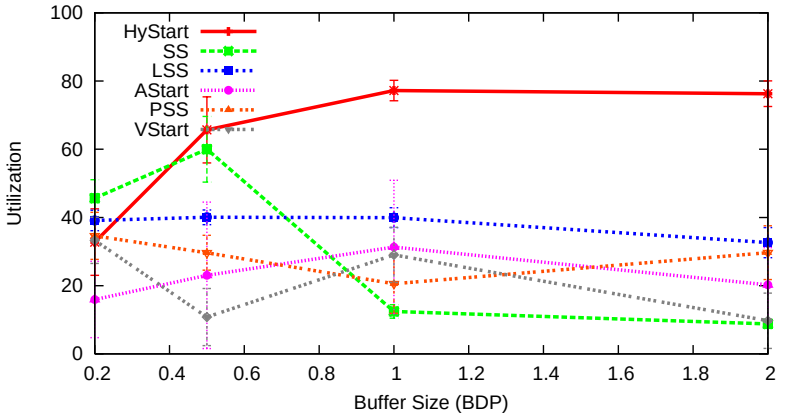
Figure 6.13: Two CUBIC flows with three different slow-start algorithms (HyStart, SS and LSS), by changing the OS of receivers.

respectively and the utilization is measured between the 10th and 70th second. We vary bandwidth from 10Mb/s to 400Mb/s, RTT from 10ms to 160ms, and the buffer sizes from 10% to 200% BDP. We add Type II background traffic to the bottleneck. 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 400Mb/s and RTT to 240ms. We use TCP-SACK for both sender and receiver. Figure 6.14 shows the performance results. HyStart exhibits consistently good network utilization independent of network bandwidth, buffer space and RTTs except in the case of very small buffer spaces (where no protocols work well). Especially under large BDP networks, HyStart outperforms the other schemes better than 3-5 times. The other schemes suffer high performance losses as the BDP increases.

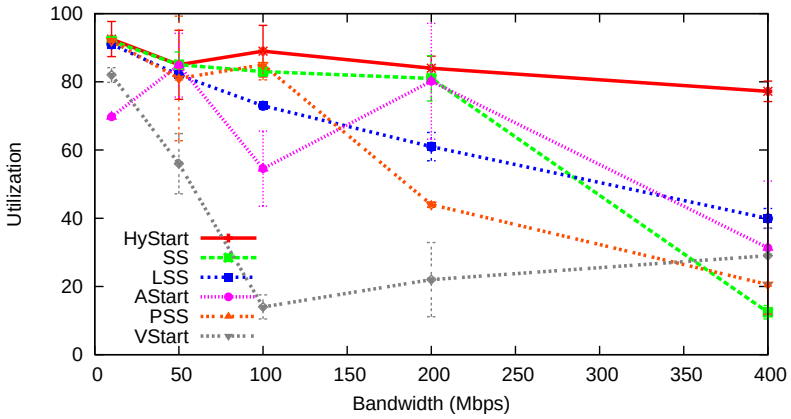
Figure 6.15 measures the CPU utilization of the sender when running various slow start protocols in Figure 6.14 (a). The results from other runs are similar. We find that the CPU utilization under SS and AStart is extremely high under medium and high BDP networks. LSS also shows relatively high CPU utilization. HyStart, PSS and VStart show very low CPU utilization because they terminate slow start earlier than packet losses occur. While PSS and VStart do so prematurely causing low network utilization, HyStart maintains good network utilization.



(a) RTT vs. Utilization



(b) Buffer size vs. Utilization



(c) Bandwidth vs. Utilization

Figure 6.14: Two TCP-SACK flows with different slow-start algorithms by varying their RTTs (a), buffer sizes (b), and bandwidth (c).

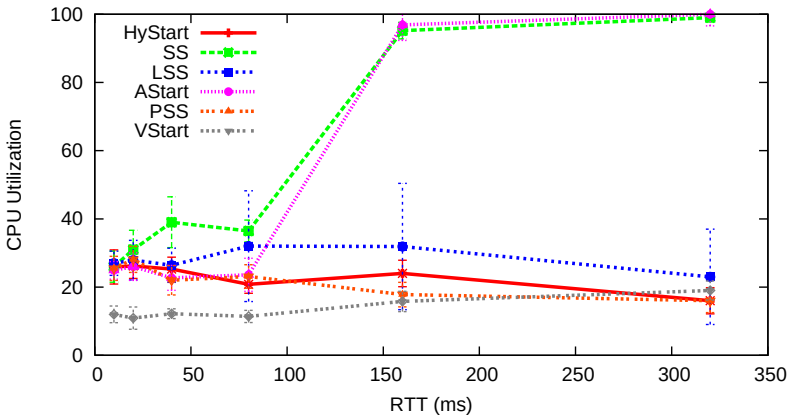
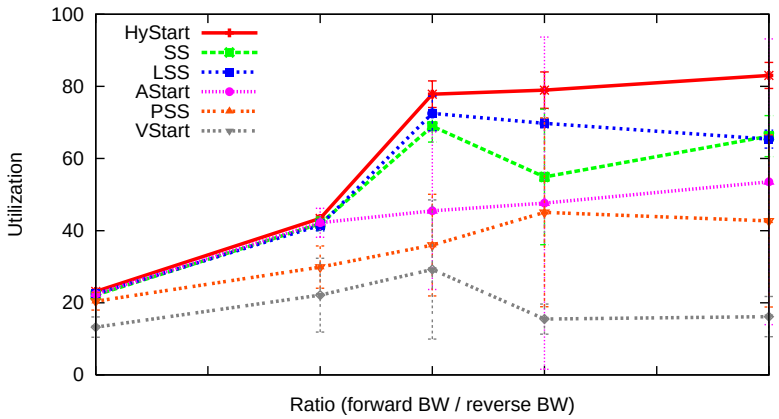


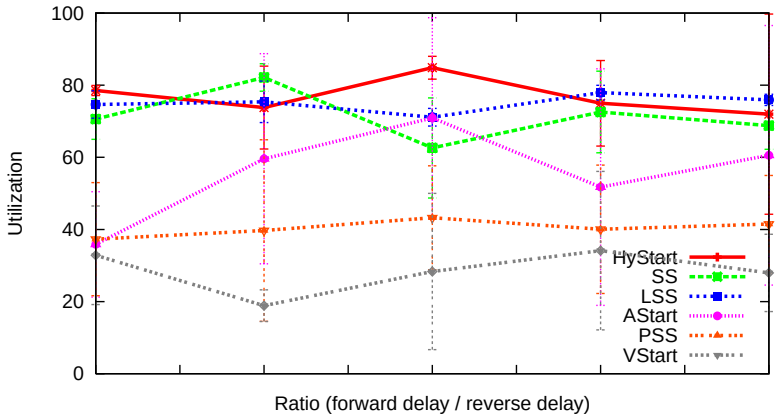
Figure 6.15: CPU utilization of the Linux sender with various slow start schemes for the run in Figure 6.14 (a).)

6.4.7 Testing over asymmetric links

Recent Internet measurement [85] reports that when there is an asymmetry in delays in the Internet, more than 80% of the paths show less than a 20ms delay difference between forward and reverse paths. The performance of HyStart may be affected by such asymmetry in delays and bandwidth. In this experiment, we test all slow-start proposals under various asymmetric network environments by changing the bandwidth and delays in forward and reverse directions. We use two TCP-SACK flows starting at the 10th and 40th seconds, respectively and measure the utilization until the 70th second. For bandwidth asymmetry testing, we fix RTT to 120ms, and vary the bandwidth ratio between forward and reverse directions from 1/4 to 4 by using the bandwidth from 100Mb/s to 400Mb/s. We also introduce Type V 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 400Mb/s, and vary the ratio between the forward delay and reverse delay from 1/3 to 3, and set the sum of the delay in both directions at 120ms. Figure 6.16 (a) and (b) show the results respectively. HyStart achieves a good start-up throughput even in the networks with high asymmetry in bandwidth and delay.



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



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

Figure 6.16: Two TCP-SACK flows with different slow-start algorithms, under asymmetric delays (a) and bandwidth (b).

6.5 Internet2 Experiment

We show the results of all the slow start proposals with TCP-SACK in three production networks such as Internet2 [6], National LambdaRail (NLR) [9] and GEANT [2].

6.5.1 Experimental setups

Figure 6.17 shows the Internet 2 testbed. Internet2 and NLR paths from North Carolina to Chicago (25ms RTT) and from Chicago to Japan (225ms RTT) have a 1Gb/s connection, and the GEANT path from North Carolina to Germany (107ms RTT) has a

100Mb/s connection. GEANT testing involves servers inside the campus networks, so some of the traffic load is expected. We run two TCP flows with different slow-start algorithms over the paths to Chicago, Germany and Japan from North Carolina. The first flow starts at the 10th second and the second flow starts at the 30th second and the utilization is measured until the 50th second. We run the experiment in three different periods of each day (5 AM to 9 AM, 1 PM to 5 PM, and 9 PM to 1 AM all EDT). Linux is installed on all of the machines in the testbed.

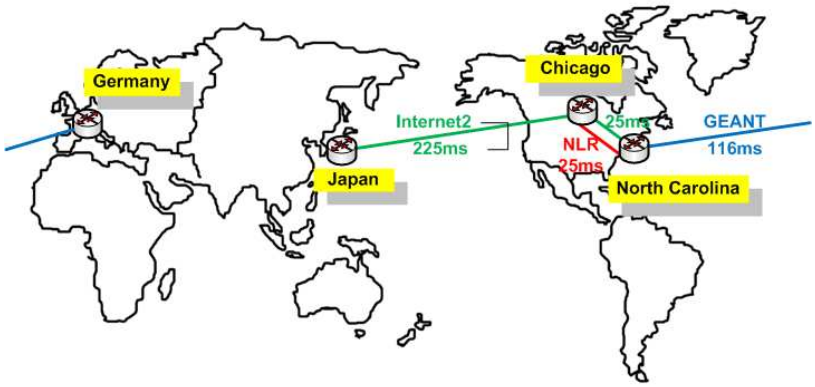
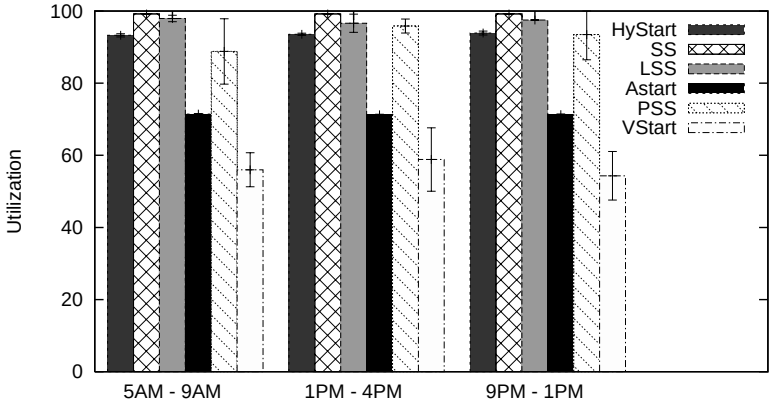


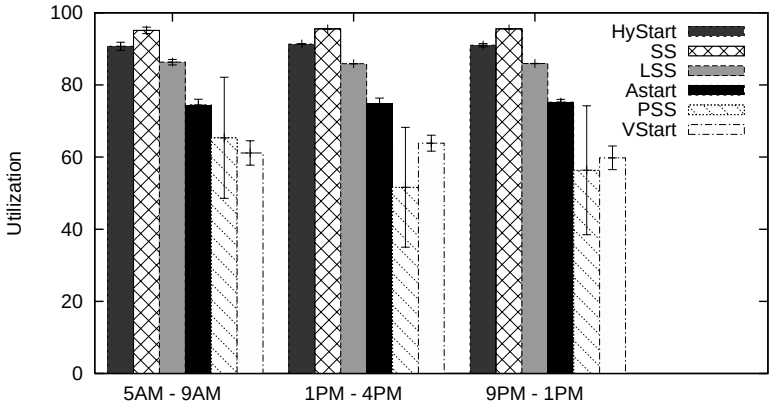
Figure 6.17: Research testbed (Internet2, NLR and GEANT)

6.5.2 Internet2 results

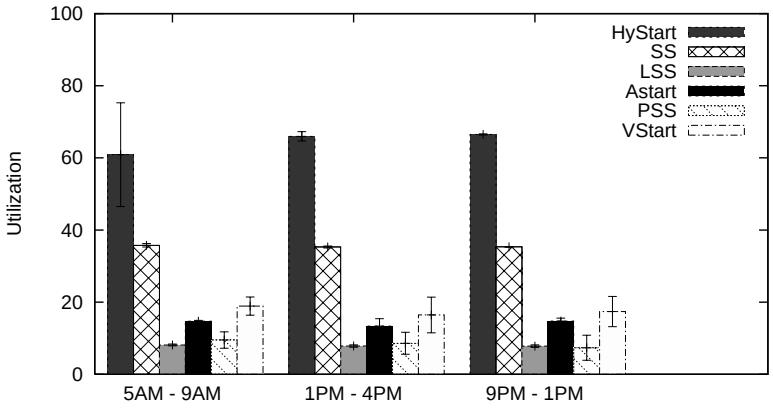
Figure 6.18 shows the results of various slow-start algorithms with a TCP-SACK sender and receiver. SS, LSS and HyStart achieves good network utilization in relatively small BDP networks (both Chicago and Germany paths). HyStart shows exceptionally good throughput for the high-BDP path between North Carolina and Japan, which has 250ms RTT and 1Gb/s link speed while LSS shows the worst performance due to its sluggish increase of cwnd in the same link. PSS typically underestimates the bandwidth so that its performance is not predictable. VStart also terminates the slow-start period prematurely and shows low utilization even in small BDP networks. In all tests, HyStart delivers consistent start-up throughput.



(a) Chicago (1Gb/s and 25ms RTT)



(b) Germany (100Mb/s and 107ms RTT)



(c) Japan (1Gb/s and 250ms RTT)

Figure 6.18: The network utilization of two TCP-SACK flows with different slow-start algorithms over the three Internet paths.

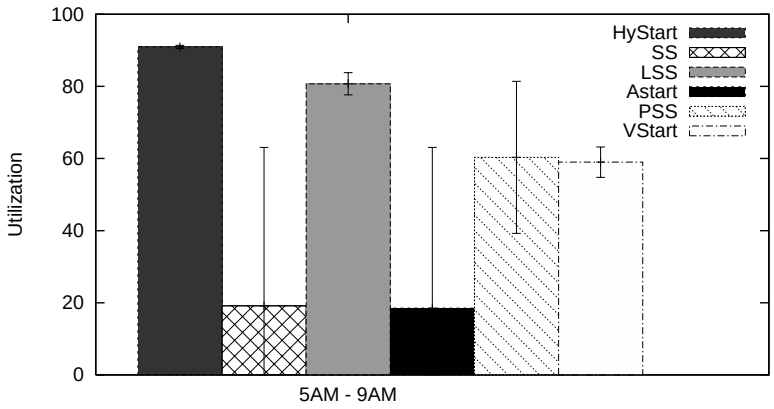


Figure 6.19: The network utilization with various slow start protocols when the sender is TCP-NewReno. The run is over a path between Germany and North Carolina.)

To further encourage 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 emulates the behavior when using a Windows XP receiver which forces the sender to behave like TCP-NewReno in medium and large size BDP networks. Figure 6.19 shows the result. SS shows extremely low utilization because of slow recovery after heavy packet losses. However, LSS and HyStart deliver outstanding utilization because HyStart exits slow start before packet losses occur and LSS reduces the cwnd growth rates during slow start thus, preventing heavy packet losses. This result indicates that a protocol like HyStart or LSS must be used to achieve an improved performance in medium size BDP networks when the receiver is Windows XP or end-systems use TCP-NewReno.

6.6 Conclusion

In this Chapter, 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 frequently in long blackouts with no transmission. This problem often occurs with TCP-SACK and TCP-NewReno which are the most popular versions of TCP used on 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 do not lead to heavy packet losses or low network utilization, preventing heavy system overload or low performance during the start-up of TCP. HyStart uses the concept of packet trains and RTT delay increases to find safe exit points. To the best of our knowledge, our work is the first that shows a practical implementation of packet 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 is maximized when the receiver-side end systems are Windows, as HyStart can greatly outperform standard slow start in this setup (even in medium-size BDP networks). This is very likely since a large number of Internet servers are Linux and many end-system users are Windows users. The performance of HyStart under large BDP networks is unsurpassed by any existing slow start protocols independent of the receiver operating systems.

Chapter 7

BLAST: A Practical Loss tolerant TCP for WAN Optimizers

WAN optimizers such as Cisco's Wide Area Application Acceleration Services (WAAS) [17] and Riverbed's Steelhead appliance [11] improve users' experience by optimizing remote access to applications over WAN links. Apart from reducing response times by accelerating client-server applications (e.g., Web, Email, and SQL) between enterprise branch office clients and datacenter servers, WAN optimizers also reduce the usage of enterprise link bandwidth and offload datacenter servers. WAN optimization is typically a two-box solution, one of which is located at the branch office to intercept and terminate application client requests (Edge box), and another one within the datacenter (Core box). The Edge box communicates with the Core box to optimize the client-server traffic over the WAN. The Core box establishes a connection and interacts with the datacenter server. Besides the application-specific optimizations, WAN optimizers use a combination of the following three techniques: *Data redundancy elimination* identifies repeating data patterns across all incoming traffic and replaces them with very small signatures; *Compression* reduces the size of transmitted data by encoding a byte sequence with a shorter code; and *TCP optimizations* include improved congestion control and socket buffer tuning.

WAN optimizers are expected to achieve efficient throughput for a wide range of networks, independent of the long round-trip times (RTT) and packet-losses that characterize unreliable media such as satellite, wireless, and long-haul international links. A timely example is the need to maximize throughput of bulk TCP transfers across data-

centers (backup, replication etc.) over WAN links with typical SLAs of 0.1% non-congestion losses, such as those provided by Sprint [12]. Although most terrestrial WAN links have very low packet loss rates most of the time (e.g. Sprint's network in North America had 0% packet losses for six months from Feb. – Jul. 2008 [12]), measurements of the Internet at large have shown that packet loss can be persistently high, especially on some enterprise and international links [12, 4, 5]. While, the predominant view is that most losses are caused when router queues overflow, there is mounting evidence that conditions other than congestion cause a significant portion of the losses. In fact, a simple experiment of sending UDP probes on TeraGrid [16] (Supercomputer Network) revealed non-zero packet losses even under zero congestion [25]. Non-congestion losses have also been observed on well run long-haul networks such as Abilene/Internet2 and National LambdaRail [25]. Hacker *et al.* in [58] generated traffic on the Abilene network over 2 days and measured a loss rate of 0.01% while confirming the absence of congestive loss. Non-congestion losses can happen for several reasons: transient (sub-rtt level) congestion, degraded fiber, malfunctioning and/or mis-configured hardware, switching contention, or a low-power receiver [25]. Stone *et al.* reported in [98] that between 1 packet in 1,100 and 1 packet in 32,000 fails the TCP checksum and is dropped, even when the data-link CRC checksum passes. A thorough investigation of this failure in the study revealed at least 40 different sources of error. Nitzan *et al.* in [82] found ATM cell drops due to CRC errors limited TCP performance over OC-12 links.

So, what are the typical packet loss rates on a WAN link? Global Crossing reported average loss-rates between 0.01% and 0.03% on four of its six long-haul links for the month of Dec. 2007 [25]. For the same month, Qwest reported loss rates of 0.01% and 0.02% on its trans-pacific link [25]. Sprint reported loss rates of 0.09% on a link from Puerto Rico to the U.S. in Oct. 2008 [12]. SLAC reported minimum loss rates of 1.5% on links in Africa for Jan. 2009 [4]. There are practically no data exclusively reporting the non-congestion loss rates, and for the purpose of this work we will work with a loss rate of up to 1%. Packet loss rates on the order of 50% or higher have been measured in certain kinds of wireless networks, such as the Roofnet Mesh network [29], and we believe such high losses warrant a solution beyond TCP's congestion control.

Packet losses can cripple TCP's performance, especially when they occur on paths which also have a long round-trip time. Figure 7.1 shows an example, and it is easy to see that a loss rate of 1% reduces TCP's throughput to less than 5% on a 45Mb/s link

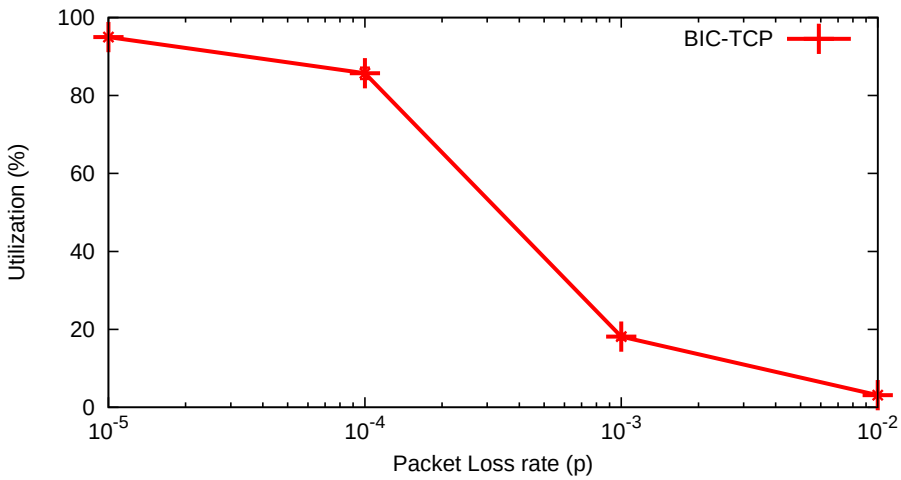


Figure 7.1: This plot shows an experiment with BIC-TCP as non-congestion loss rates increase from 10^{-5} to 10^{-2}. The throughput decreases to less than 5% of the link-rate for a loss rate of 10^{-2}. Setup: 1 flow, 45Mb/s, 100ms RTT, bandwidth-delay buffering, BIC-TCP in Linux.

with 100ms RTT. Standard loss-based TCPs treat every packet loss as an indication of congestion and reduce their window size in response. In steady-state, with a packet loss-rate of p, the average congestion window for a loss-based TCP (such as BIC-TCP), decreases as $1/p^d; 1/2 < d < 1$ [106], placing a severe constraint on the achievable throughput in environments with natural losses.

It is imperative that WAN optimizers be designed to achieve high throughput even in the presence of natural packet losses. Riverbed's WAN optimizer uses MX-TCP (Max-Speed TCP) [11] to fill allocated bandwidth without backing off under any packet losses. MX-TCP relies on a pre-configured knowledge of the number of flows and available bandwidth, which is impractical in today's networks.

Our goal in this work is to design a practical highspeed TCP that is tolerant to non-congestion losses even for large bandwidth-delay product networks.[1] This work makes three contributions. First, we evaluate the performance of existing loss and delay-based TCPs in a regime of large round-trip times and high packet loss rates (up to 1%). We find

[1]The motivation for this work is drawn from our experience with WAN optimizers where it is important even for a single TCP connection to scale to high BDP networks in the presence of losses. While such scaling is less stringent for typical Internet users, it can nevertheless be useful in settings beyond WAN optimization.

that none of the existing TCPs are able to successfully maintain link-rate throughput in the face of non-congestion related losses. Somewhat surprisingly, this is true of even those TCPs explicitly designed for lossy wireless networks.

Second, we describe the design and evaluation of BLAST (BIC-TCP with Loss Tolerance). BLAST is a simple heuristic based on average queueing delay measurements to successfully differentiate congestion losses from non-congestion related ones. Our evaluation shows that BLAST improves TCP's throughput by a significant portion in the presence of natural losses.

Our third contribution is to integrate the BLAST heuristic with TCP's mechanisms in the Linux stack. While simple in principle, in practice it was more complex because under high losses a TCP sender is in most part in the loss recovery phase, where congestion control works differently (and far less understood) than its normal operation. While we have implemented BLAST using BIC-TCP as the base, BLAST can however be used in conjunction with any loss or delay based TCPs.

As a comparison of some of the described approaches in Section 3.3, Figure 7.2 shows an experiment with Linux TCPs and BLAST under non-congestion losses. Existing TCPs achieve less than one tenth of the link-rate for a loss rate of 1%. Our main conclusion is that the existing approaches do not offer solutions that scale to high BDP networks in the face of non-congestion packet losses. BLAST aims to bridge this gap.

The rest of the paper is organized as follows: we describe the design of the BLAST algorithm in Section 7.1, its integration with Linux TCP in Section 7.2, and its performance evaluation in Section 7.3. Finally, we conclude in Section 7.4.

7.1 The design of BLAST Algorithm

BLAST achieves its goal of resiliency to non-congestion losses by combining the best aspects of loss and delay based TCPs. Loss-based TCPs reduce their sending rates only on detecting a packet loss. On the other hand, delay-based TCPs decrease their rate on inferring (through RTT measurements) a growing bottleneck queue size. Even though delay-based TCPs can achieve high utilization without incurring large queues and losses, they are often fragile because of their reliance on accurate RTT measurements to decide the window increment and decrement. Further, because delay is an early indicator of congestion,

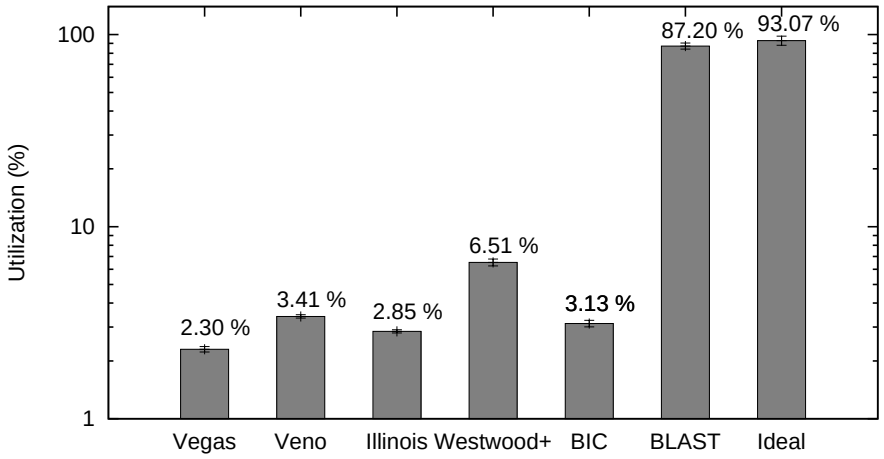


Figure 7.2: Experiment showing throughputs of different Linux TCP flavors: TCP-Veno (discriminates congestion versus non-congestion), TCP-Illinois (loss + delay based), TCP-Vegas (delay based), TCP Westwood+ (rate based), BIC-TCP (loss based). Also shown for comparison is BLAST's throughput as well as that of *ideal* congestion control which fixes TCP's *cwnd* to the bandwidth-delay product. Setup: 1 flow, 45Mb/s, 100ms RTT, loss-rate: 1%, BDP buffering.

delay-based schemes fail to attain their fair share of bandwidth when competing with loss-based schemes. Loss-based TCPs are more robust as they do not rely on RTT measurements to evolve their congestion windows. However, they equate *every* loss with congestion and blindly reduce their sending rate. BLAST overcomes this shortcoming by distinguishing the losses related to congestion from those that are not.

7.1.1 Description of BLAST algorithm

Fundamentally, BLAST is a loss-based TCP in the sense that it decreases its rate only upon detection a loss. BLAST is built upon BIC-TCP [106]. Like many other TCP flavors, BIC reduces its window multiplicatively on a packet loss, but has a unique growth window. BIC performs a binary search between two parameters, *Wmax* (the window size just before the multiplicative decrease) and *Wmin* (the window size just after the decrease), by jumping to the midpoint. Since packet losses have occurred at *Wmax*, the window size that the network can currently handle without loss must be somewhere between these two numbers. However, unlike BIC which treats every loss as congestion, BLAST heuristically

distinguishes a loss as unrelated to congestion if its estimated queuing delay is lower than a threshold. After a non-congestion loss, BLAST retransmits the lost packet and continues evolving its congestion window (*cwnd*) as if there were no loss. It is better than pure delay-based TCPs because it relies only on average queuing delays as binary indicators whether a particular loss is congestion related or not. We describe the details of BLAST in the rest of this section.

Let W_k^i denote the congestion window size, R_k^i the RTT measurement, and D_k^i the queueing delay estimated through the i^{th} ACK in k^{th} RTT round. D_k^i, can be written as Eq. 7.1:

$$D_k^i = R_k^i - \min_{j \le k, l \le i} R_j^l, \quad j \ge 1,\ l \ge 1 \tag{7.1}$$

As *cwnd* increases within a congestion epoch (and correspondingly the number of delay samples), the mean queuing delay in the most recent RTT round reflects the latest information of queue build-up at the bottleneck. BLAST uses the estimated mean queuing delay , $\overline{D}_k$ (Eq. 7.2), to identify a congestion related loss.

$$\overline{D}_k = \overline{R}_k - \min_{j \le k, l \le i} R_j^l, \quad j \ge 1,\ l \ge 1 \tag{7.2}$$

where $\overline{R}_k$ is the mean RTT in k^{th} round. If the number of ACKs in the current round is insufficient to get a reliable estimate of the queuing delay (e.g., if a loss happens at a very early stage of the current round k), BLAST relies on the previous round's estimate $\overline{D}_{k-1}$. BLAST's heuristic of classifying losses is shown below in Eq. 7.3, and also illustrated in Figure 7.3. When $\overline{D}_k$ is greater than threshold, δ_H, the loss is inferred as congestion related, and similarly if $\overline{D}_k$ is smaller than a threshold, δ_L, the loss is treated as non-congestion related. In the *grey* zone region between δ_L and δ_H, a loss is treated as congestion related if the queuing delay estimate in the current round has increased from the prior round. The grey zone is effectively used to detect monotonically increasing queuing delays, a signature of congestion. In summary:

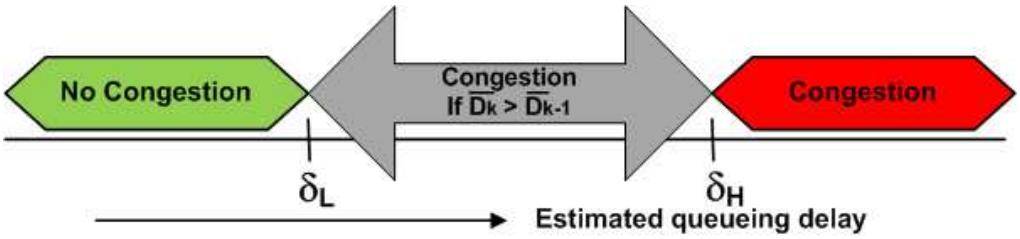


Figure 7.3: The above diagram illustrates BLAST's heuristic of disambiguating the nature of a loss. δ_L and δ_H are lower and upper thresholds determined from the estimate of maximum queuing delay.

$$is\_congestion = \begin{cases} 1; & \text{if } \overline{D}_k \geq \delta_H \text{ or} \\ & (\delta_L < \overline{D}_k < \delta_H, \text{ and } \overline{D}_k \geq \overline{D}_{k-1}) \\ 0; & \text{if } \overline{D}_k \leq \delta_L \text{ or} \\ & (\delta_L < \overline{D}_k < \delta_H, \text{ and } \overline{D}_k < \overline{D}_{k-1}) \end{cases} \tag{7.3}$$

Eq. 7.4 shows the congestion window evolution on a packet loss. α and β are the AIMD parameters which BIC-TCP uses in controlling its congestion window.

$$\begin{aligned} cwnd &= cwnd + \frac{\alpha}{cwnd}, \quad is\_congestion = 0 \\ &= \beta \times cwnd, \qquad is\_congestion = 1 \end{aligned} \tag{7.4}$$

The thresholds, δ_L and δ_H, are fixed percentages (γ_L and γ_H respectively), of the maximum queuing delay seen for this connection, as shown below:

$$\begin{aligned} \delta_L &\longleftarrow \gamma_L \max_{j,l} D_j^l, \quad j \geq 1,\ l \geq 1 \\ \delta_H &\longleftarrow \gamma_H \max_{j,l} D_j^l, \quad j \geq 1,\ l \geq 1 \end{aligned} \tag{7.5}$$

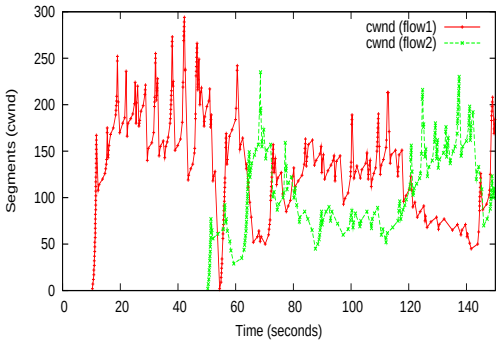
Each BLAST flow measures its own view of maximum bottleneck queue delay by calculating the difference between minimum and maximum of RTTs seen so far. It infers congestion by comparing the average queuing delay over one RTT round, $\overline{D}_k$, with its view of the bottleneck's maximum queuing delay. BLAST sets $(\gamma_L, \gamma_H) = (25\%, 35\%)$ by

default. We show in section 7.3 that this simple heuristic can successfully distinguish a large percentage of losses unrelated to congestion. Before delving further into BLAST's RTT computations, let's look at an example of how BLAST works.

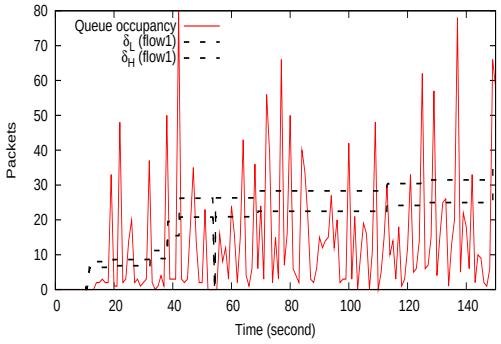
7.1.2 An illustrative example

Figure 7.4 shows BLAST in action on a link with 2 flows and 0.5% non-congestion related losses. Flow 1, starting at time 10s, increases its *cwnd* until it observes a self-induced queuing delay, $\overline{D}_k$, greater than δ_H (or on detecting monotonically increasing delays in the grey zone between δ_L and δ_H). When flow 2 joins the link at time 50s and contributes to congestion by increasing the router queue, flow 1 reduces its *cwnd* by detecting the congestion signatures on some of these drops. Flow 2 experiences relatively fewer reductions because of its small starting cwnd, and continues increasing its rate to claim its fair share (Figure 7.4 (a)).

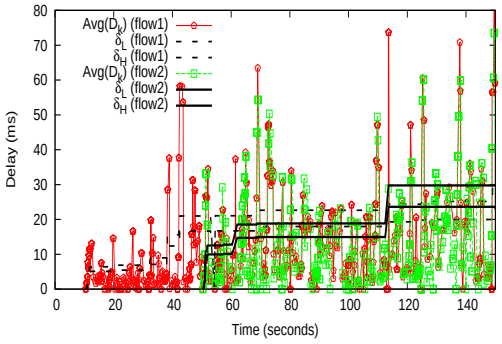
We also note that $\overline{D}_k$ matches well with the queue occupancy at the router, and the δ_L, δ_H of the two BLAST flows settle down to 25% and 35% of the maximum queue occupancy allowing the flows to effectively detect the onset of congestion (Figure 7.4 (b)). We note that as BLAST reduces *cwnd* far before the bottleneck queue overflows, the router queue occupancy is small and there are no congestion drops (Figure 7.4 (c)). The grey zone is effective in detecting the onset of congestion using the increasing trend of delay (Figure 7.4 (d)).



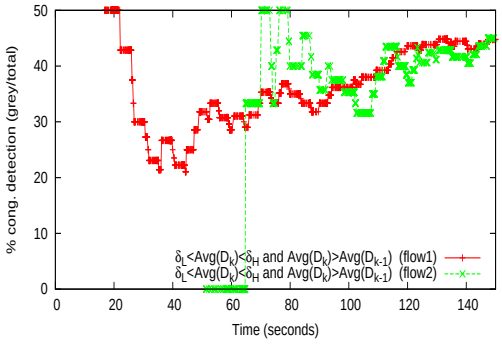
(a) Evolution of CWND (link utilization: 94%)



(c) Queue occupancy at the router



(b) Evolution of BLAST variables



(d) Congestion detection using grey zone

Figure 7.4: This plot shows how BLAST can distinguish between non-congestion and congestion losses. Two flows share a 15Mb/s link with 160ms RTT, BDP buffering (200 packets) and 0.5% non-congestion related losses. Flows 1 and 2 reduce *cwnd* upon losses only if the queueing delay is larger than a certain threshold ($\overline{D}_k > \delta_H$), or if the queueing delay shows an increasing trend in the grey zone ($\overline{D}_k \geq \overline{D}_{k-1}$ where $\delta_L < \overline{D}_k \leq \delta_H$). (a) and (b) show BLAST's *cwnd* evolution and its variables used to detect congestion. (c) shows a small queue occupancy at the router because BLAST's heuristics detect the onset of congestion early on. Note that δ_L and δ_H in (c) are calculated by multiplying the bandwidth (15 Mb/s) and δ_L, δ_H in (b). (d) shows that the grey zone significantly contributes to disambiguating the congestion-related losses.

7.1.3 Obtaining reliable estimates of RTT

A BLAST flow requires three kinds of RTT estimates to compute the average queuing delay $\overline{D}_k$, and its parameters, δ_L and δ_H: the minimum RTT ($minRTT$) and maximum RTT ($maxRTT$) which it uses to compute the maximum queueing delay along its path (Eq. 7.1), and an average RTT estimate within a round which is used to calculate the average queuing delay (Eq. 7.2).

On receiving the i^{th} ACK where $i > 2$, BLAST updates the following RTT estimates:

$$minRTT_i = \min(minRTT_{i-1}, RTT_i) \tag{7.6}$$

$$avgRTT_i = \frac{\sum_{i-n+1}^{i} RTT_j}{n} \tag{7.7}$$

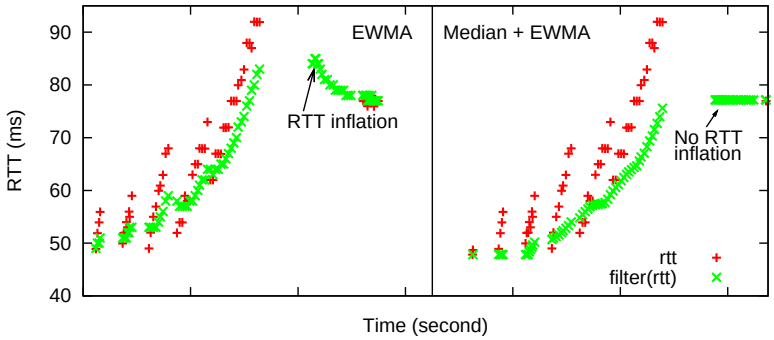
$$mRTT_i = Median(RTT_{i-2}, RTT_{i-1}, RTT_i) \tag{7.8}$$

$$sRTT_i = \frac{7}{8} * sRTT_{i-1} + \frac{1}{8} * mRTT_i \tag{7.9}$$

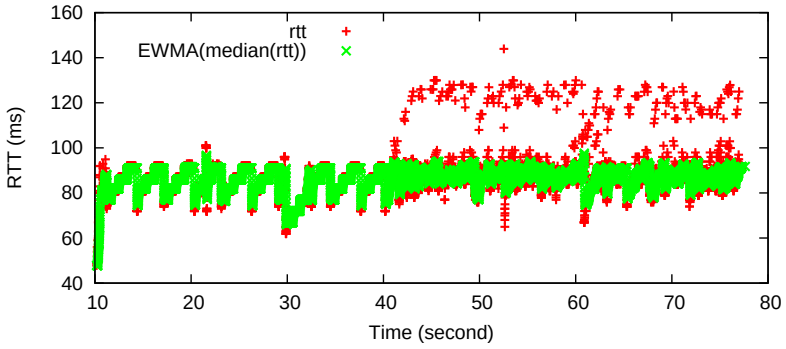
$$maxRTT_i = \max(maxRTT_{i-1}, sRTT_i) \tag{7.10}$$

where n denotes the number of RTT samples in the current congestion epoch. Estimating $minRTT$ and $avgRTT$ is simple, as shown in Eq. 7.6 and Eq. 7.7. The raw RTT measurements are used to calculate $avgRTT$, in order to track the most recent changes in the bottleneck queue, as opposed to using smoothed estimates which can mask the relevant recent changes. BLAST calculates the average queueing delay for each round, $\overline{D}_k$, by subtracting $minRTT$ from the $avgRTT$.

Computing the $maxRTT$ is trickier as it could be inflated for reasons other than the path queuing delay, e.g., delayed ACK timeouts and additional buffering delays at both sender and receiver sides. An inflated $maxRTT$ also inflates δ_L, δ_H for congestion detection and makes BLAST more aggressive in the face of congestive losses. If $maxRTT$ is simply computed as the maximum of all RTT samples, a single spurious large RTT sample can inflate the threshold delay and make BLAST unnecessarily aggressive. BLAST applies two filters to the raw RTT measurements before using them to compute $maxRTT_i$: Median filter (Eq. 7.8) and Exponential Weighted Moving Average (EWMA) (Eq. 7.9). We find these filters effective in damping large RTT outliers. To reflect any changes in a connection's RTT if its route changes in the network, we update $minRTT$ and $maxRTT$ over a window of T_w (in the order of a minute).



(a) EWMA filtering vs. Median+EWMA filtering



(b) EWMA+Median filtering over the lifetime of a TCP flow

Figure 7.5: This plot shows an example of how BLAST filters spurious RTT measurements to arrive at an estimate of $maxRTT$. Top plots show the effect of just EWMA filtering (left plot) and that combined with Median filtering (right plot). The high RTT values resulting from delayed ACK timeouts were successfully filtered out. The bottom plot shows the filtered and raw RTT measurements over the lifetime of a TCP flow. A new flow joins the network at time 40s and results in noisy raw RTT values, but filtering avoids an inflation of $maxRTT$. For this experiment, we set the bandwidth to 5Mb/s, RTT to 50ms, and buffer size to 100% BDP of a flow.

Figure 7.5 shows an example of BLAST's RTT estimates. Without any filtering techniques, $maxRTT$ would over-estimate the queuing delay, primarily due to the two clusters of inflated RTT measurements by delayed ACK timeouts (Figure 7.5(a)). EWMA alone filters the $maxRTT$ to 85ms, and eventually smooths out the temporal inflation of RTT estimates. But since BLAST takes the maximum of all RTT estimates, even a temporal RTT inflation still affects the $maxRTT$ estimate. On the other hand, using the Median filter before filtering through EWMA (right plot of Figure 7.5(a)), BLAST obtains a more conservative estimate of $maxRTT$.

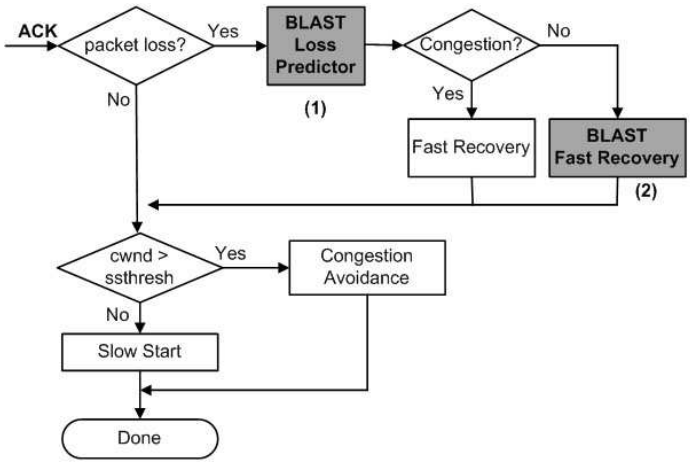


Figure 7.6: BLAST TCP congestion control processing

7.1.4 Limitations and scope

An underlying assumption in BLAST is that the most significant variations in network delay are a result of queuing in the network buffers. While this is true most of the time in wired networks, certain kinds of wireless networks can have large delay fluctuations not necessarily associated with congestion. In these networks, BLAST will conservatively treat losses in the period of fluctuation as congestion related ones, and behave no worse than BIC or other loss based congestion control.

7.2 Making BLAST work in Linux TCP

In this section we describe how we integrated BLAST with TCP's mechanisms in Linux.

7.2.1 Implementation and practical considerations

We have implemented BLAST in Linux 2.6.23.9, using BIC-TCP as the base. BLAST adds two main additional components to the existing path of BIC-TCP, as illustrated in the block diagram of Figure 7.6. On detecting a loss, the *BLAST Loss Predictor* infers the cause of packet loss. If the cause is congestion related, BLAST traverses

Initialization:

$\gamma_L \leftarrow 0.25, \quad \gamma_H \leftarrow 0.35, \quad cntRTT \leftarrow 0, \quad sumRTT \leftarrow 0$

$blast\_reset(), \quad prev \leftarrow 0, \quad cur \leftarrow 0, \quad nxt \leftarrow 0$

On each ACK:

begin

$dMin \leftarrow min(dMin, RTT)$

$mRTT \longleftarrow median\_filter(RTT)$

$sRTT \longleftarrow \frac{7}{8}sRTT + \frac{1}{8}mRTT$

$dMax \leftarrow \max(dMax, sRTT)$

$sumRTT \leftarrow sumRTT + RTT$

$cntRTT \leftarrow cntRTT + 1$

end

At the start of each RTT round:

begin

$prev\_delay \longleftarrow avg\_delay()$

$cntRTT \longleftarrow 0, \quad sumRTT \longleftarrow 0$

end

Packet loss:

begin

$ssthresh \longleftarrow blast\_recalc\_ssthresh()$

end

Timeout:

begin

$blast\_reset()$

end

$median\_filter(RTT)$:

begin

$prev \leftarrow cur, \quad cur \leftarrow nxt, \quad nxt \leftarrow RTT$

$cur \leftarrow max(min(prev,\ nxt),\ cur)$

$cur \leftarrow min(max(prev,\ nxt),\ cur)$

return cur

end

Figure 7.7: BLAST algorithm.

```
Fast Recovery for each ACK:
begin
    // Maintain pipe number of packets during Fast Recovery
    pipe ⟵ ssthresh
    is_cong ⟵ blast_loss_predictor()
    if !is_cong then pipe ⟵ pipe + bictcp_update()
    // When TCP is mostly in F.R due to large cwnds and high
      loss rates, BLAST regulates the cwnd by using valid
      ACKs received during each F.R round. This is crucial
      for achieving high performance in high-BDP networks.
    if snd_una ≥ high_seq then
        pipe ⟵ blast_recalc_ssthresh()
        prev_delay ⟵ avg_delay()
end
blast_recalc_ssthresh():
begin
    is_cong ⟵ blast_loss_predictor()
    if is_cong then ssthresh ← bictcp_recalc_ssthresh()
    else ssthresh ← cwnd
    return ssthresh
end
blast_loss_predictor():
begin
    // Identify the congestion-related losses using D̄
    if avg_delay() ≤ low_threshold() then is_cong ← 0
    else if avg_delay() > high_threshold() then is_cong ← 1
    else
        // Is the delay increasing in the grey zone ?
        if avg_delay() > prev_delay then is_cong ← 1
        else is_cong ← 0
    return is_cong
end
```

Figure 7.8: BLAST algorithm. (Continued)

avg_delay():
begin
  if $cntRTT > 0$ **and** $sumRTT > 0$ **then return** $(\frac{sumRTT}{cntRTT} - dMin)$
  else return 0
end
low_threshold():
begin
  return $\gamma_L * (dMax \ - \ dMin)$
end
high_threshold():
begin
  return $\gamma_H * (dMax \ - \ dMin)$
end
blast_reset():
begin
  bictcp_reset()
  $prev\_delay \leftarrow 0, \ \ dMin \leftarrow \infty, \ \ dMax \leftarrow 0$
end

Figure 7.9: BLAST algorithm. (Continued)

the normal path of Fast Recovery. Otherwise, *BLAST Fast Recovery* continues the operation as if there were no loss, by increasing the congestion window by the same amount that BIC-TCP increases during Congestion Avoidance.

Figures 7.7, 7.8 and 7.9 detail the pseudo code of BLAST algorithms in Linux. The min, max, smoothed, and average RTT estimates are updated on receipt of every ack carrying a valid timestamp. Aside from RTT estimation, there are two other important parts (shown in boxes). First, *blast_recalc_ssthresh*() uses the BLAST heuristic (*blast_loss_predictor*()) to disambiguate the loss and reduce *cwnd* only on a congestion related loss. Note that *last_max_cwnd* is BIC-TCP's variable $Wmax$. Similarly *bictcp_update*() and *bictcp_recalc_ssthresh*() are the BIC-TCP's window increase and drop functions.

The second important part is how BLAST changes the standard Fast Recovery path. Just like in standard TCP, BLAST enters the Fast Recovery phase on detecting a packet loss through three duplicate acknowledgments. The standard Fast Recovery tries to maintain a certain number of packets in flight (*pipe*) over the entire period of the recovery. Usually, *pipe* is set to $\beta \times cwnd$ $(0.5 < \beta < 1)$ for most of AIMD algorithms, including BIC-TCP. Because standard Fast Recovery is designed for a quick recovery of the lost packets without incurring further packet losses or timeouts, no congestion control is integrated apart from conservatively adhering to the same rate as before. This makes sense when TCP has no way to infer congestion when in the recovery phase. However, the problem is: in high bandwidth-delay product networks, even a small non-congestion loss rate forces a TCP sender to be in the Fast Recovery phase for most of its lifetime and thus, constrains its throughput severely. BLAST overcomes this crucial bottleneck. It changes the Fast Recovery congestion control by increasing *cwnd* on every ACK in the absence of congestion, and reducing *cwnd* when its heuristic detects congestion. BLAST uses BIC-TCP's binary search to increase *cwnd* and multiplicatively decrease *cwnd* during Fast Recovery. Note that BLAST's *Wmax* is only set on a congestion related window drop, so *Wmax* can be thought of as the maximum window into which BLAST's congestion window can safely evolve. Congestion control during fast recovery is one of the unique and crucial features in BLAST that allows its throughput to scale in high BDP networks under non-congestive losses. The standard path of Fast Recovery in Linux TCP limits *cwnd* at several points to the number of packets currently in flight (e.g., rate halving and undo mechanism). BLAST

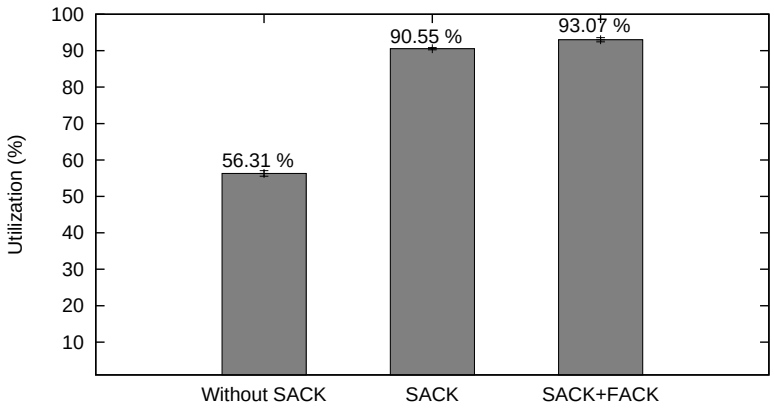


Figure 7.10: The effect of SACK and FACK (Forward Acknowledgment) on performance. We use an ideal congestion control which fixes the congestion window to the bandwidth-delay product. The setup: single flow, 45Mb/s 100ms RTT, BDP buffering and loss rate is 1%.

modifies this path to avoid limiting *cwnd* if a loss is classified to be non-congestion related.

BLAST does not change the timeout behavior of standard TCP.

Considering that the Linux timer granularity is 1ms, BLAST is turned on only for those connections whose threshold queueing delay is at least 2ms. This means if the maximum queueing delay of a connection is less than 8ms, BLAST falls back to vanilla BIC-TCP.

Finally, we note that BLAST's heuristics can be easily used in conjunction with any loss or delay based TCPs, and are not limited to our implementation with BIC-TCP.

7.2.2 Non congestion control issues in BLAST

Apart from congestion control, Selective Acknowledgment (SACK) mechanisms play a key role in TCP throughput. Figure 7.10 shows an example where even under an *ideal* congestion control mechanism (which fixes *cwnd* to the BDP value for a single flow) TCP is not able to achieve line-rate throughput without SACK. SACK helps a TCP sender learn about multiple lost TCP segments per RTT, allowing the sender to retransmit only the missing segments.

7.3 Evaluation

In this section we describe how BLAST performs over a wide range of network and traffic conditions, using a mix of testbed experiments (with a real implementation in Linux 2.6.23.9), as well as with simulations for more complex scenarios (using the TCP evaluation tool in ns2 [1]). Our evaluation is inspired by the work in [22] where Andrew et al., have compiled an extensive list of scenarios for objectively evaluating TCPs. We first describe our experimental setup below, followed by an evaluation of the accuracy of BLAST heuristics, and then its performance in different network scenarios.

7.3.1 Experimental testbed

We use a dumbbell topology as shown in Figure 4.1 and use experimental methodology presented in Chapter 4. If we use background traffic in the experiment, we use Type II background traffic shown in Table 4.1. We evaluate BLAST by varying a link rate, RTT, bottleneck buffer size and link loss-rate. We use a drop-tail router at the bottleneck.

7.3.2 BLAST's accuracy in distinguishing non-congestion losses

We quantify in simple terms the accuracy of BLAST in disambiguating losses. If a higher proportion of losses are classified as more non-congestion related than they actually are, a TCP sender would be overly aggressive in the face of congestion. And so, ideally, we want the ratio of non-congestion losses (NCL) to the total losses detected by TCP senders (ncl_{blast}) to be close to but not greater than the ratio of NCL to total losses at the bottleneck queue (ncl_{queue}).

Figure 7.11 shows ncl_{blast} and ncl_{queue} by increasing non-congestion loss rates when 5 flows share a bottleneck link. When loss rates are low, ncl_{blast} tracks ncl_{queue} closely, and underestimates ncl_{queue} as loss rates increase. It does not come as a surprise that BLAST underestimates the non-congestion loss. When the queuing delay increases at the onset of congestion, BLAST treats a non-congestion loss as congestion related *before* packets overflow from buffers. One can think of this behavior as similar to Random Early Drop (RED) or other AQM schemes which drop packets proactively as queues rise, well before buffers are overrun. As we will see in the rest of the evaluations below, BLAST continues to achieve link-rate throughput, in contrast to other schemes.

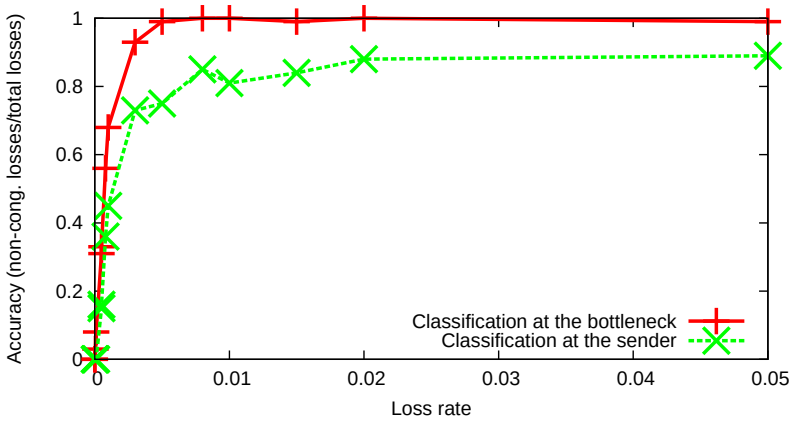


Figure 7.11: This plot shows how accurately BLAST discriminates non-congestion losses from congestion losses. There are 5 flows sharing a 45Mb/s link with 80ms RTT and BDP buffering. The non-congestion loss rate is varied from 0 to 5%. We find ncl_{blast} tracks ncl_{queue} closely for smaller losses and underestimates it for higher losses because of its early congestion detection.

7.3.3 Basic scenarios

In this section, we evaluate how BLAST scales with network parameters such as link-rate, RTT, bottleneck buffer size, and loss-rates. In each of the experiments, the topology is as shown in Figure 4.1. We compare BLAST with BIC-TCP (loss-based), TCP-Vegas (delay-based) and TCP-Westwood (rate-based) which use different congestion indicators. In the interest of space, we show only the metric of bottleneck link utilization. We use a single TCP flow, as this is the worst case scenario for the bottleneck throughput metric. In subsequent sections, we evaluate the fairness properties of BLAST when there are multiple flows.

As the loss rate increases

Figure 7.12 varies the loss rate from 10^{-5} to 10^{-2}, while keeping all else fixed. BLAST maintains its high throughput while the other three protocols sharply drop their throughput as the loss rate increases. While the rate-based TCP-Westwood performs slightly better than BIC-TCP and TCP-Vegas, BLAST shows the best immunity to the higher loss rates (10^{-3} and 10^{-2}).

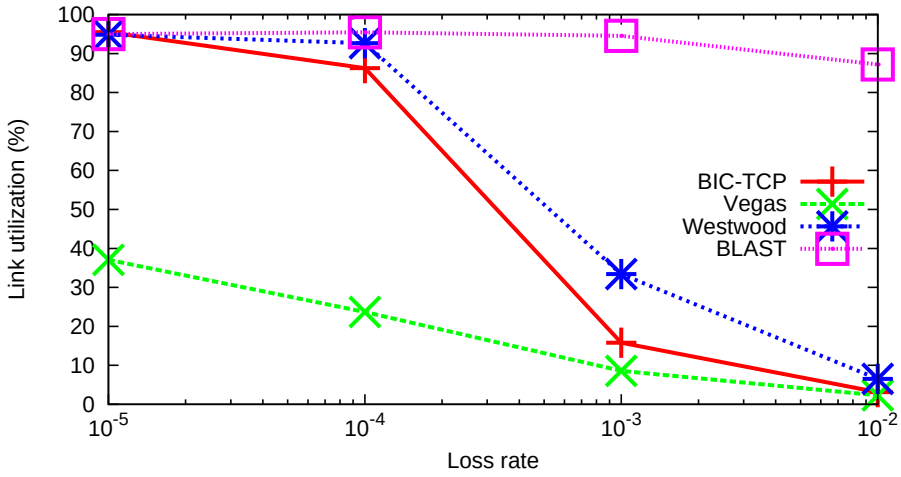


Figure 7.12: This plot shows the throughput for BLAST and the other three protocols for varying the loss rate from 10^{-5} to 10^{-2}. Setup: bandwidth is 45Mb/s, RTT is 100ms, BDP buffering is used, and no background traffic is introduced. The throughput of single TCP flow is measured.

As the link-rate increases

Figure 7.13 varies link rates from 1.5Mb/s to 100Mb/s, for a loss-rate of 1%, while keeping all else fixed. BLAST maintains its high throughput as the link-rate increases. For the higher link-rates (such as 45Mb/s, 75Mb/s and 100Mb/s), TCP is almost always in the Fast Recovery mode when the loss is 1%, and BLAST's congestion control in the recovery mode is crucial in allowing it to scale to a high BDP. The throughput of BIC-TCP, TCP-Vegas and TCP-Westwood plummets to 10% even for the smaller link-rates, and goes much lower as the link-rate increases.

As the round-trip delay increases

Figure 7.14 varies RTT from 10ms to 320ms, for a loss-rate of 1%, while keeping all else fixed. While the throughput of BIC-TCP, TCP-Vegas and TCP-Westwood is around 10% or lower, BLAST is able to realize a throughput that is sometimes an order of magnitude higher. Just as in the case of increasing link-rates, BLAST maintains a high throughput as the BDP increases. BLAST's Fast Recovery mechanisms once again play an important role for higher RTTs.

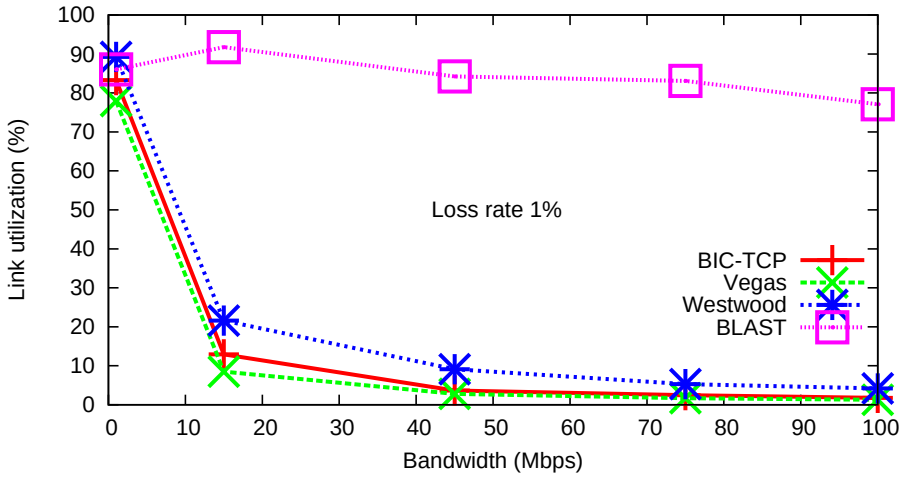


Figure 7.13: This plot shows the throughput for BLAST and the other protocols for varying link-rates from 1Mb/s to 100Mb/s. Setup: RTT is 80ms, the loss rate is 1%, BDP buffering is used, and no background traffic is introduced. The throughput of single TCP flow is measured.

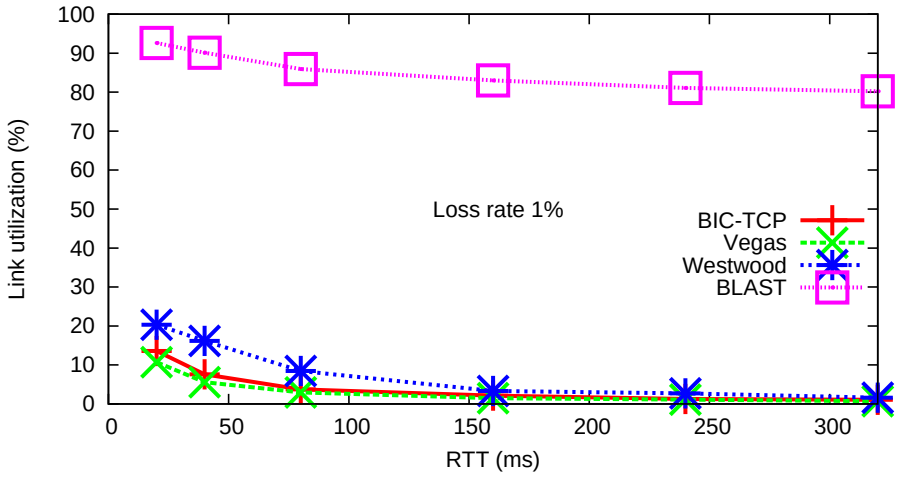


Figure 7.14: This plot shows the throughput for BLAST and the other protocols for increasing RTTs from 10ms to 320ms. Setup: bandwidth is 45Mb/s, the loss rate is 1%, and BDP buffering is used. The throughput of single TCP flow is measured. To introduce some variation, short-lived Web traffic (4Mb/s) is generated in both forward and reverse directions of the dumbbell.

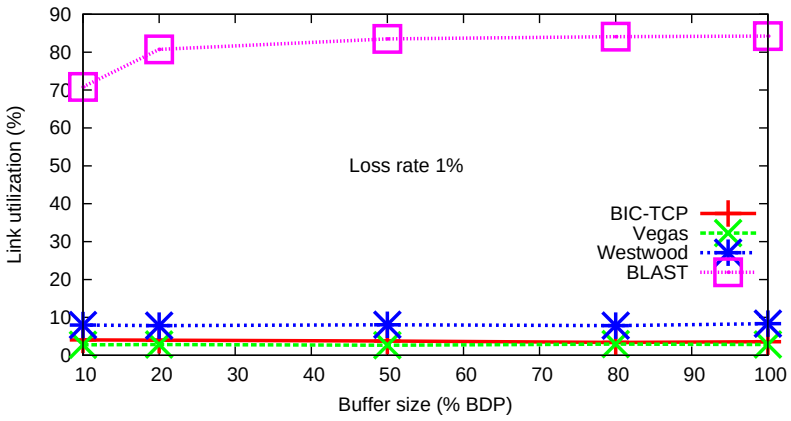


Figure 7.15: This plot shows the throughput for BLAST and the other protocols for varying bottleneck buffer-sizes from 10% to 100% BDP. Setup: bandwidth is 45Mb/s, RTT is 80ms, the loss rate is 1%, and no background traffic is introduced. The throughput of single TCP flow is measured.

As the buffer-size decreases

Figure 7.15 shows that BLAST is able to maintain its high throughput in the presence of non-congestion losses even when buffers are as small as a tenth of BDP. This is because BLAST obtains a fairly accurate estimate of the maximum bottleneck queuing delay independent of the buffer size and is able to disambiguate a large percentage of non-congestion losses.

7.3.4 Heterogeneous round-trip delays

Figure 7.16 shows the throughput share of two flows with heterogeneous round-trip delays. We vary the RTT of the flow 1 from 10ms to 100ms while keeping the RTT of flow 2 fixed at 100ms. We test their bandwidth sharing property for two loss rates (0% and 1%). In this experiment, we report only BIC-TCP and BLAST because TCP-Vegas and TCP-Westwood have trends similar to that of BIC-TCP. When the non-congestion loss is 0%, the two BLAST flows share the link in a very similar fashion as two BIC-TCP flows. At a loss-rate of 1%, the two BIC-TCP flows achieve very low overall throughput while BLAST maintains a high utilization for all RTT pairs, without much impact on its fairness properties.

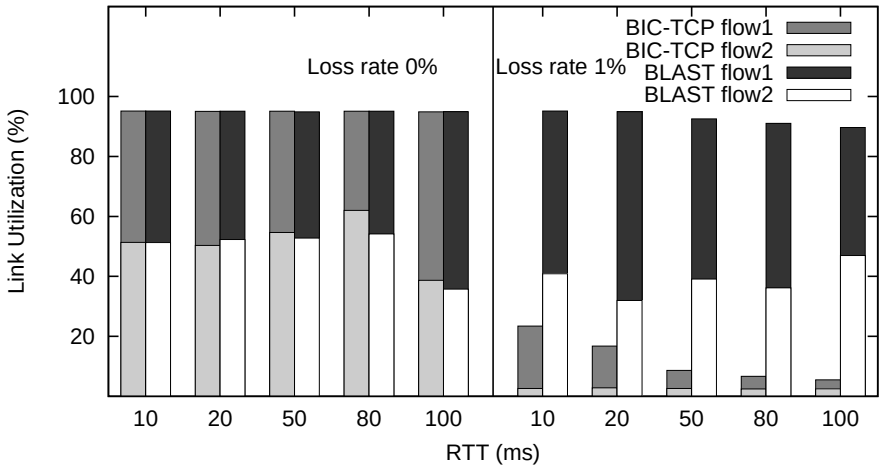


Figure 7.16: This plot shows the throughput share of two flows with heterogenous round-trip delays. Setup: The bandwidth and buffer size are fixed at 45 Mb/s and 100% BDP respectively. We vary the RTT of flow 1 from 10ms to 100ms while fixing the RTT of flow 2 at 100ms. Left plot shows the throughput share for 0% non-congestion loss and the right plot for 1%.

7.3.5 Multiple bottleneck links

Figure 7.17 shows BLAST's performance in a network with five bottleneck queues arranged in a parking lot topology, with each bottleneck also carrying cross traffic [22]. BLAST's throughput is three times higher than BIC-TCP. We note that BIC-TCP's throughput is higher than the previous examples because of the short RTT cross-traffic which is able to achieve a higher rate.

7.3.6 Coexistence with Standard TCP

In real networks, BLAST must coexist with other TCP flavors, most prominently TCP-Sack. Figure 7.18 shows that under high non-congestion losses, BLAST only grabs the unused bandwidth of the link and achieves good throughput. Without random losses, BIC-TCP and BLAST are identical in fully utilizing the link while sharing it with standard TCP.

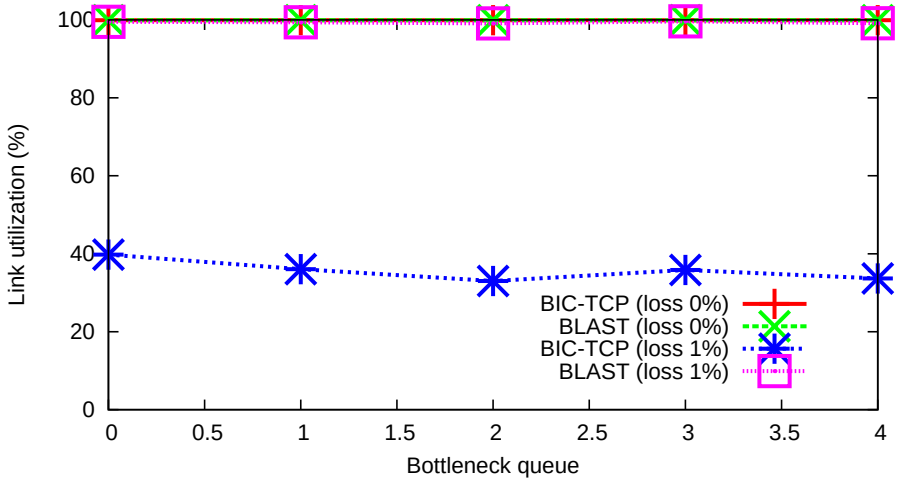


Figure 7.17: This plot shows BLAST and BIC-TCP throughputs when there are multiple bottlenecks. Setup: parking lot topology with five bottlenecks of each 45Mb/s, total RTT is 80ms, loss-rates = 0% and 1% on *each* link. There are 5 FTP flows traversing all the links and a cross traffic of 5 flows in each bottleneck. The set-up is adapted from NS2 TCP eval. Tool[1]

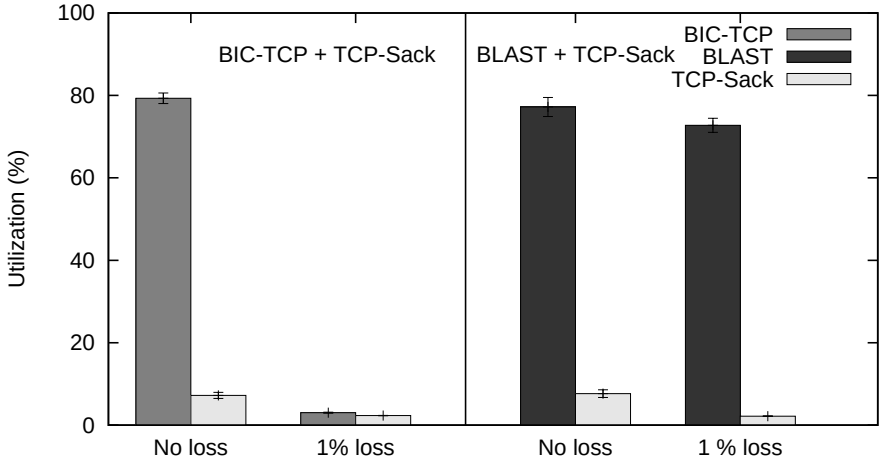


Figure 7.18: This plot shows the impact of a regular TCP-Sack flow when it competes with a BLAST (right plot) and a BIC-TCP flow (left plot). Setup: link-rate is 45Mb/s, RTT is 100ms, the loss rates are 0% and 1%. BDP buffering is used. Short-lived Web traffic (4Mb/s) is introduced in both forward and reverse directions of the dumbbell. With 0% loss, BIC-TCP and BLAST behave in the same fashion; they are able to fill the bandwidth unused by TCP-Sack. With 1% loss, BLAST continues to fill the unused link bandwidth, while BIC-TCP is unable to do so.

7.4 Conclusion

Our main conclusion is that BLAST improves TCP's throughput significantly in networks that have non-congestion related losses. It opens up the possibility for flows to react to losses in a more informed way, as opposed to blindly reacting to every single loss. BLAST's three main features are: 1) it achieves higher throughput compared to existing TCPs (often by an order of magnitude higher) by effectively disambiguating a large percentage of non-congestion losses, 2) BLAST's mechanisms scale to high bandwidth-delay product networks, and 3) its heuristics are simple, robust, easy to understand and implement. Going forward, the focus of our ongoing work will be to enhance BLAST in order to scale with higher loss rates, as well as to scale to networks with a high delay variability.

Chapter 8

Conclusion and Future Work

TCP is responsible for detecting and reacting to overloads on the Internet and has been the key to the Internet's operational success in the last few decades. However, as link capacity grows and new Internet applications with high-bandwidth demand emerge, TCP's performance is unsatisfactory, especially in high-speed and long-distance networks. This dissertation explores practical solutions that address these challenges. We conclude by discussing the proposed solutions and remaining challenges.

8.1 Conclusion

In this dissertation, we identify three different components (i.e., congestion avoidance, slow start, and loss detection and recovery) in TCP congestion control that still require greater optimization in order to achieve high performance in high bandwidth and long distance networks. We propose three practical solutions - CUBIC, HyStart, and BLAST. CUBIC modifies the congestion avoidance algorithm, HyStart modifies the slow start algorithm, and BLAST modifies the loss detection and recovery path of the standard TCP. They are independent solutions which can be implemented separately or in conjunction with each other, and thus improve TCP performance significantly in high bandwidth and long distance networks. Our conclusions are as follows:

- CUBIC, the next version of BIC-TCP, modifies the linear window growth function of existing TCP standards to be a cubic function, and this greatly simplifies the window adjustment algorithm (binary search) of BIC-TCP. Specifically, it computes

the window size in the next round by using the elapsed time since the last loss. This makes the cubic window growth independent of RTT and ensures an equitable bandwidth share among flows with different RTTs. A cubic function containing both concave and convex portions makes the protocol very scalable when the bandwidth and delay product of the network is large, and at the same time, highly stable and also fair to standard TCP flows. Incorporating time and cubic function into the protocol allows CUBIC flows competing in the same bottleneck to have approximately the same window size independent of their RTTs, achieving good RTT-fairness. Our extensive testing in lab testbeds and also on the Internet confirm that CUBIC achieves fairly good fairness (Intra-protocol fairness, RTT-fairness, and TCP-friendliness) while also achieving good performance.

- HyStart is a new TCP slow start algorithm that improves the start-up throughput by preventing long burst packet losses during slow start. It uses ACK trains and RTT delay samples to detect whether (1) the forward path is congested or (2) the current size of the congestion window has reached the available capacity of the forward path. Upon detecting (1) or (2), HyStart terminates slow start and advances to the congestion avoidance phase without heavy packet losses. Long burst packet losses frequently makes end systems unresponsive for an extended period without transmission while recovering from burst packet losses. Unfortunately, no previous work thoroughly investigates the effect of burst losses on end systems during slow start. Devising better techniques to handle many packet losses efficiently is important as it can be applied to any situation where burst losses occur. We realize that a preventive solution is also significantly more valuable. This is because some proprietary optimizations applied to slow start by common operating systems in order to relieve the system load happen to slow down loss recovery, and improving the system bottleneck by efficiently optimizing data structures still requires further improvement. We report superior performance of HyStart in lab testbeds and also on the Internet.

- BLAST is a practically designed loss tolerant TCP for WAN optimizers which require achieving high throughput for a wide range of networks including the networks with long round trip times and packet losses unrelated to congestion that typically characterize unreliable media such as satellite, wireless, and long haul international

links. Unfortunately, even low packet loss rates cripple standard TCPs' throughput, e.g., a packet loss rate of 1% constrains TCP's throughput to less than 5% of the link-rate on a 45Mb/s link with a 100ms round-trip time. The underutilization is worsening as bandwidth and delay increase. By recognizing this limitation, we introduces two heuristics - (1) mean queueing delay by averaging RTT samples in each RTT and (2) "grey" zone for detecting the trend in queueing delay changes. We integrate BLAST heuristics into the loss detection and loss recovery path of Linux and evaluate their effectiveness. We confirm that BLAST achieves an order of magnitude higher throughput than existing loss-based, delay-based, and rate-based protocols.

- The proposed algorithms are easy to understand and require modification only on the sender. We integrate these algorithms into Linux and verify their effectiveness.

8.2 Future Work

While we present our proposed solutions for improving TCP performance in high bandwidth and long distance networks, we identify the following remaining challenges.

- There are many high-speed TCP variants available and they are used on the Internet. For example, Compound TCP is used by recent Windows operating systems such as Windows Vista and Windows Server 2008 while CUBIC has been used as a default congestion control algorithm in Linux since 2006. We are interested in obtaining TCP census on the Internet. We plan to examine the methodology identifying the congestion control algorithm of TCP flows actively and passively.

- BLAST filters RTT samples to obtain a more accurate estimate of the bottleneck buffer size. Specifically, BLAST uses both median and EWMA filters to remove frequent outliers in RTT measurements. These two filters require negligible CPU computation. However, we need to explore more robust filters (e.g., Chauvenet's criterion [100] and Peirce's criterion [91]) for better accuracy in RTT measurements even though it may require more CPU computation. This improvement can be applied to any protocols that use RTT measurements as a part of their congestion control algorithms.

Bibliography

[1] An NS2 TCP Evaluation Tool. http://labs.nec.com.cn/tcpeval.htm.

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

[3] Git logs for CUBIC updates. http://git.kernel.org/?p=linux/kernel/git/davem/net-2.6.git;a=history;f=net/ipv4/tcp_cubic.c;h=eb5b9854c8c7330791ada69b8c9e8695f7a73f3d;hb=HEAD.

[4] Internet End-to-End Performance Monitoring. http://www-iepm.slac.stanford.edu/.

[5] Internet Traffic Report. http://www.internettrafficreport.com/.

[6] Internet2. http://www.internet2.edu/.

[7] IPerf. http://dast.nlanr.net/projects/Iperf/.

[8] Linux CUBIC source navigation. http://lxr.linux.no/linux/net/ipv4/tcp_cubic.c.

[9] National LamdaRail. http://www.nlr.net/.

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

[11] Riverbed. http://www.riverbed.com/.

[12] Sprint IP Network Performance. https://www.sprint.net/sla_performance.php.

[13] TCP Testing Wiki. http://netsrv.csc.ncsu.edu/wiki/index.php/TCP_Testing.

[14] Tcpdump and libpcap. http://www.tcpdump.org.

[15] TCPProbe. http://lxr.linux.no/linux+v2.6.26.5/net/ipv4/tcp_probe.c.

[16] TeraGrid. http://www.teragrid.org/.

[17] WAN Optimization – Cisco Systems. http://www.cisco.com/web/go/waas.

[18] Pluggable congestion avoidance modules. http://lwn.net/Articles/128681/, 2005.

[19] J. Aikat, J. Kaur, F. Smith, and K. Jeffay. Variability in TCP Round-trip Times. In *Proceedings of the ACM SIGCOMM Internet Measurement Conference*, Miami, FL, Oct. 2003.

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

[21] M. Allman, V. Paxson, and W. Stevens. TCP Congestion Control. RFC 2581 (Proposed Standard), Apr. 1999. Updated by RFC 3390.

[22] L. Andrew, C. Marcondes, S. Floyd, L. Dunn, R. Guillier, W. Gang, L. Eggert, S. Ha, and I. Rhee. Towards a Common TCP Evaluation Suite. In *Proceedings of the sixth PFLDnet Workshop*, Manchester, UK, Mar. 2008.

[23] G. Appenzeller, I. Keslassy, and N. McKeown. Sizing Router Buffers. In *Proceedings of ACM SIGCOMM*, Portland, OR, 2004.

[24] A. Baiocchi, A. P. Castellani, and F. Vacirca. YeAH: Yet Another Highspeed TCP. In *Proceedings of the fifth PFLDNet Workshop*, California, Feb. 2007.

[25] M. Balakrishnan, T. Marian, K. Birman, H. Weatherspoon, and E. Vollset. Maelstrom: transparent error correction for lambda networks. In *NSDI'08: Proceedings of the 5th USENIX Symposium on Networked Systems Design and Implementation*, pages 263–278, Berkeley, CA, USA, 2008. USENIX Association.

[26] P. Barford and M. Crovella. Generating Representative Web Workloads for Network and Server Performance Evaluation. In *Measurement and Modeling of Computer Systems*, pages 151–160, 1998.

[27] D. Barman, G. Smaragdakis, and I. Matta. The Effect of Router Buffer Size on HighSpeed TCP Performance. In *Proceedings of IEEE GLOBECOM*, 2004.

[28] S. Biaz and N. H. Vaidya. "De-randomizing" congestion losses to improve TCP performance over wired-wireless networks. *IEEE/ACM Transactions on Networking*, 13(3):596–608, 2005.

[29] S. Biswas and R. Morris. Opportunistic routing in multi-hop wireless networks. *ACM SIGCOMM Computer Communications Review*, 34(1):69–74, 2004.

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

[31] L. S. Brakmo and L. L. Peterson. TCP Vegas: End to End Congestion Avoidance on a Global Internet. *IEEE Journal on Selected Areas in Communications*, 13(8):1465–1480, 1995.

[32] H. Bullot, R. L. Cottrell, and R. Hughes-Jones. Evaluation of Advanced TCP Stacks on Fast Long-Distance Production Networks. In *Proceedings of the second PFLDNet Workshop*, Illinois, Feb. 2004.

[33] H. Cai, D. Eun, S. Ha, I. Rhee, and L. Xu. Stochastic Ordering for Internet Congestion Control and its Applications. In *Proceedings of IEEE INFOCOM*, Anchorage, Alaska, May 2007.

[34] C. Caini and R. Firrincieli. TCP Hybla: a TCP Enhancement for Heterogeneous Networks. *International Journal of Satellite Communication and Networking*, 22(5):547–566, Sept. 2004.

[35] R. L. Carter and M. E. Crovella. Measuring bottleneck link speed in packet-switched networks. *Performance Evaluation*, 27-28:297–318, 1996.

[36] C. Casetti, M. Gerla, S. Mascolo, M. Y. Sanadidi, and R. Wang. TCP Westwood: Bandwidth Estimation for Enhanced Transport over Wireless Links. In *Proceedings of ACM Mobicom*, Rome, Italy, July 2001.

[37] S. Cen, P. C. Cosman, and G. M. Voelker. End-to-end differentiation of congestion and wireless losses. *IEEE/ACM Transactions on Networking*, 11(5):703–717, 2003.

[38] D. Chiu and R. Jain. Analysis of the Increase/Decrease Algorithms for Congestion Avoidance in Computer Networks. *Journal of Computer Networks and ISDN*, 17(1):1–14, June 1989.

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

[40] end2end interest. Is RED dead? Email to end2end-interest mailing list, Oct. 2005.

[41] S. Floyd. Congestion Control Principles. RFC 2914 (Best Current Practice), Sept. 2000.

[42] S. Floyd. HighSpeed TCP for Large Congestion Windows. RFC 3649 (Experimental), Dec. 2003.

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

[44] S. Floyd. Specifying Alternate Semantics for the Explicit Congestion Notification (ECN) Field. RFC 4774 (Best Current Practice), Nov. 2006.

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

[46] S. Floyd, M. Handley, and J. Padhye. A Comparison of Equation-Based and AIMD Congestion Control, May 2000.

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

[48] S. Floyd and V. Jacobson. Random Early Detection gateways for Congestion Avoidance. *IEEE/ACM Transactions on Networking*, 1(4):397–413, Aug. 1993.

[49] S. Floyd and E. Kohler. Internet Research Needs Better Models. In *ACM HotNets*, Oct. 2002.

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

[51] S. Floyd and V. Paxson. Difficulties in Simulating the Internet. *IEEE/ACM Transactions on Networking*, 9(4), Aug. 2001.

[52] C. P. Fu and S. C. Liew. TCP Veno: TCP Enhancement for Transmission Over Wireless Access Networks. *IEEE Journal on Selected Areas in Communications*, Feb. 2003.

[53] S. Ha. Cubic v2.0-pre patch. http://netsrv.csc.ncsu.edu/twiki/pub/Main/BIC/cubic-kernel-2.6.13.patch.

[54] S. Ha, N. Dukkipati, V. Subramanian, F. Bonomi, and I. Rhee. BLAST: A practical loss tolerant TCP for WAN optimizers. Technical report, Computer Science Department, North Carolina State University, 2009.

[55] S. Ha and I. Rhee. Hybrid Slow Start for High-Bandwidth and Long-Distance Networks. In *Proceedings of the sixth PFLDNet Workshop*, Manchester, UK, Mar. 2008.

[56] S. Ha and I. Rhee. Taming the Elephants: New TCP Slow Start. Technical report, Computer Science Department, North Carolina State University, 2008.

[57] S. Ha, I. Rhee, and L. Xu. CUBIC: A New TCP-friendly High-Speed TCP Variant. *ACM SIGOPS Operating Systems Review*, 42(5):64–74, 2008.

[58] T. J. Hacker, B. D. Noble, and B. D. Athey. The effects of systemic packet loss on aggregate TCP flows. In *Supercomputing '02: Proceedings of the 2002 ACM/IEEE conference on Supercomputing*, pages 1–15, Los Alamitos, CA, USA, 2002. IEEE Computer Society Press.

[59] T. Hatano, M. Fukuhara, H. Shigeno, and K. Okada. TCP-friendly SQRT TCP for High Speed Networks. In *Proceedings of APSITT*, pages 455–460, Nov. 2003.

[60] S. Hemminger. Cubic root benchmark code. http://lkml.org/lkml/2007/3/13/331.

[61] S. Hemminger. Linux TCP Performance Improvements. Linux World 2004, 2004.

[62] S. Hemminger. TCP infrastructure split out. http://lwn.net/Articles/128626/, 2005.

[63] J. C. Hoe. Improving the Start-up Behavior of a Congestion Control Scheme for TCP. In *Proceedings of ACM SIGCOMM*, 1996.

[64] N. Hu and P. Steenkiste. Improving TCP Performance using Active Measurements: Algorithm and Evaluation. In *Proceedings of the 11th International Conference on Network Protocols (ICNP'03)*, Atlanta, GA, Nov. 2003.

[65] V. Jacobson. Congestion Avoidance and Control. *ACM Computer Communications Review*, 8(4):314–329, Aug. 1988.

[66] M. Jain and C. Dovrolis. End-to-end available bandwidth: measurement methodology, dynamics, and relation with TCP throughput. *IEEE/ACM Transactions on Networking*, 11(4):537–549, 2003.

[67] R. Jain. A Delay-Based Approach for Congestion Avoidance in Interconnected Heterogeneous Computer Networks. *ACM SIGCOMM Computer Communications Review*, 19:56–71, 1989.

[68] C. Jin, D. X. Wei, and S. H. Low. FAST TCP: motivation, architecture, algorithms, performance. In *Proceedings of IEEE INFOCOM*, Hong Kong, Mar. 2004.

[69] Q. Z. M. S. K. Tan, J. Song. Compound TCP: A Scalable and TCP-friendly Congestion Control for High-speed Networks. In *Proceedings of the fourth PFLDnet Workshop*, Japan, Feb. 2006.

[70] T. Kelly. Scalable TCP: Improving performance in highspeed wide area networks. *ACM SIGCOMM Computer Communication Review*, 33(2):83–91, Apr. 2003.

[71] S. Keshav. A Control-Theoretic Approach to Flow Control. *Proceedings of the conference on Communications architecture and protocols*, pages 3–15, 1993.

[72] R. King, R. Riedi, and R. Baraniuk. Evaluating and Improving TCP-Africa: an Adaptive and Fair Rapid Increase Rule for Scalable TCP. In *Proceedings of the third PFLD Workshop*, France, Feb. 2005.

[73] K. Kumazoe, K. Kouyama, Y. Hori, M. Tsuru, and Y. Oie. Transport Protocol for Fast Long-Distance Networks : Evaluation of Their Penetration and Robustness on JGNII. In *Proceedings of the third PFLDNet Workshop*, France, Feb. 2005.

[74] S. S. Kunniyur and R. Srikant. End-To-End Congestion Control: Utility Functions, Random Losses and ECN Marks. In *Proceedings of IEEE INFOCOM*, Tel-Aviv, Israel, Mar. 2000.

[75] D. Leith, J. Heffner, R. Shorten, and G. McCullagh. Delay-based AIMD congestion control. In *Proceedings of the fifth PFLDnet Workshop*, Marina Del Rey, CA, Feb. 2007.

[76] Y.-T. Li, D. Leith, and R. N. Shorten. Experimental Evaluation of TCP Protocols for High-Speed Networks. *IEEE/ACM Transactions on Networking*, 15:1109–1122, Oct. 2007.

[77] S. Liu, T. Basar, and R. Srikant. TCP-Illinois: A Loss and Delay-Based Congestion Control Algorithm for High-Speed Networks. In *Proceedings of VALUETOOLS*, Pisa, Italy, Oct. 2006.

[78] 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*, pages 287–297, 2001.

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

[80] A. Medina, M. Allman, and S. Floyd. Measuring the evolution of transport protocols in the internet. *ACM SIGCOMM Computer Communications Review*, 35(2):37–52, 2005.

[81] J. Nagle. Congestion control in IP/TCP internetworks. RFC 896, Jan. 1984.

[82] R. L. Nitzan and B. L. Tierney. Experiences with TCP/IP over an ATM OC12 WAN. Technical report, Lawrence Berkeley National Laboratory, 1999.

[83] J. Padhye, V. Firoiu, D. Towsley, and J. Kurose. Modeling TCP Throughput: a Simple Model and its Empirical Validation. In *Proceedings of ACM SIGCOMM*, 1998.

[84] V. Padmanabhan and R. Katz. TCP fast start: a technique for speeding up web transfers. 1998.

[85] A. Pathak, H. Pucha, Y. Zhang, Y. C. Hu, and Z. M. Mao. A Measurement Study of Internet Delay Asymmetry. In *Proceedings of Passive and Active Measurement Conference (PAM)*, pages 37–52, Cleveland, Ohio, Apr. 2008.

[86] J. Postel. Internet Protocol. RFC 791 (Standard), Sept. 1981. Updated by RFC 1349.

[87] J. Postel. Transmission Control Protocol. RFC 793 (Standard), Sept. 1981. Updated by RFC 3168.

[88] K. Ramakrishnan, S. Floyd, and D. Black. The Addition of Explicit Congestion Notification (ECN) to IP. RFC 3168 (Proposed Standard), Sept. 2001.

[89] K. Ratnam and I. Matta. WTCP: an efficient mechanism for improving wireless access to TCP services. *International Journal of Communication Systems*, 16, Feb. 2003.

[90] L. Rizzo. Dummynet: a simple approach to the evaluation of network protocols. Jan. 1997.

[91] S. Ross. Peirce's criterion for the elimination of suspect experimental data. In *Journal of Engineering Technology*, volume 20, Fall 2003.

[92] N. Samaraweera. Non-congestion packet loss detection for TCP error recovery using wireless links. In *IEE Proceedings - Communications*.

[93] M. Scharf, S. Hauger, and J. K"ogel. Quick-Start TCP: From Theory to Practice. In *Proceedings of the sixth PFLDNet Workshop*, UK, Mar. 2008.

[94] R. N. Shorten and D. J. Leith. H-TCP: TCP for high-speed and long-distance networks. In *Proceedings of the Second PFLDNet Workshop*, Argonne, Illinois, Feb. 2004.

[95] T. Socolofsky and C. Kale. TCP/IP tutorial. RFC 1180 (Informational), Jan. 1991.

[96] N. Spring, D. Wetherall, and D. Ely. Robust Explicit Congestion Notification (ECN) Signaling with Nonces. RFC 3540 (Experimental), June 2003.

[97] L. Stewart, G. Armitage, and J. Healy. Characterising the Behavior and Performance of SIFTR v1.1.0. Technical report, CAIA, Swinburne University of Technology, Aug. 2007.

[98] J. Stone and C. Partridge. When the CRC and TCP checksum disagree. In *SIGCOMM '00: Proceedings of the conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*, pages 309–319, New York, NY, USA, 2000. ACM.

[99] W. Tarreau. Cubic optimization. http://git.kernel.org/?p=linux/kernel/git/davem/net-2.6.git;a=commit;h=7e58886b45bc4a309aeaa8178ef89ff767daaf7f.

[100] J. R. Taylor. An Introduction to Error Analysis. 2nd edition. In *California: University Science Books*, pages 166–168, Sausalito, 1997.

[101] O. Tickoo, V. Subramanian, S. Kalyanaraman, and K. K. Ramakrishnan. LT-TCP: End-to-End Framework to Improve TCP Performance over Networks with Lossy Channels. In *IEEE International Workshop on Quality of Service (IWQoS)*, Marina Del Rey, CA, June 2005.

[102] H. Wang, H. Xin, D. Kang, and G. Shin. A Simple Refinement of Slow Start of TCP Congestion Control, 2000.

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

[104] D. Wei, P. Cao, and S. Low. Time for a TCP Benchmark Suite? Technical report, Caltech, Aug. 2005.

[105] D. X. Wei, C. Jin, S. H. Low, and S. Hegde. FAST TCP: motivation, architecture, algorithms, performance. 14:1246–1259, Dec. 2006.

[106] L. Xu, K. Harfoush, and I. Rhee. Binary Increase Congestion Control for Fast Long-Distance Networks. In *Proceedings of IEEE INFOCOM*, Hong Kong, Mar. 2004.

[107] L. Zhang, S. Shenker, and D. Clark. Observations on the Dynamics of a Congestion Control Algorithm: The Effects of Two-Way Traffic. In *Proceedings of ACM SIGCOMM*, 1991.

[108] Y. Zhang, N. Duffield, V. Paxson, and S. Shenker. On the constancy of Internet path properties. In *Proceedings of ACM SIGCOMM Internet Measurement Workshop*, Nov. 2001.