IMPROVING TCP PERFORMANCE IN THE MOBILE, HIGH SPEED, HETEROGENOUS AND EVOLVING INTERNET

WU XIUCHAO

NATIONAL UNIVERSITY OF SINGAPORE

2009

IMPROVING TCP PERFORMANCE IN THE MOBILE, HIGH SPEED, HETEROGENOUS AND EVOLVING INTERNET

WU XIUCHAO

B.E., USTC

M.Sc., NUS

A THESIS SUBMITTED

FOR THE DEGREE OF PH.D. IN COMPUTER SCIENCE

DEPARTMENT OF COMPUTER SCIENCE

NATIONAL UNIVERSITY OF SINGAPORE

2009

Acknowledgements

I want to express my deeply-felt thanks to my M.Sc. and Ph.D. supervisor, A/P Akkihebbal L. Ananda, for his inspiring ideas, valuable suggestions, and constant encouragement during all these years. Without him, the work would not have been possible. I am grateful to my Ph.D. co-supervisor, Dr. Mun Choon Chan, for his thoughtful and important advices throughout this work.

I wish to express my special thanks to Dr. Wei Tsang Ooi, Dr. Haifeng Yu, and Dr. Rajesh Krishna Balan, for their comments and suggestions on my thesis. I would also like to express my gratitude to all present and former members of Communication and Internet Research Lab, as well as my friends and classmates who helped me at different periods of my work. In particular, I would like to thank Mr. Chetan Ganjihal, Dang Thanh Son, Myo Myint, Soe Hla Win, Huynh Gia Huy, and Indradeep Biswas for the countless hours in coding together, as well as the interesting discussions. I would like thank Mr. Venkatesh S. Obanaik, Aurbind Sharma, and Feng Xiao for their patient help in locating and using lab resources. I would also express special thanks to Dr. Sridhar K.N. Rao, Mingze Zhang, Binbin Chen, Tao Shao, Zhiguo Ge, and Yong Xiao for their helps in many aspects of my work and my life. I would like to thank all my friends who supported me in the completion of my study.

Lastly, my special thanks go to my wife, my daughter, and all my family who always support me and encourage me in my life.

Contents

Abstract

As the de facto standard transport protocol, TCP has contributed to the enormous success of the Internet. TCP provides an attractive connection-oriented end-to-end transport service and ensures a reliable and in-order transfer of data. With TCP congestion control, TCP has also ensured good performance of applications and kept the stability of the exponentially increasing Internet. However, in recent years, many new types of networks with different characteristics have been deployed in the Internet. Within these new types of networks, the original assumptions of TCP congestion control, such as reliable links with low/medium bandwidth and stationary hosts, are frequently undermined, and it is very important to improve the performance of TCP in these networks. In this thesis, three very important problems are addressed to improve the performance of TCP in the context of the mobile, high speed, heterogeneous and evolving Internet.

Firstly, as we deploy many different kinds of wireless networks, the mobile Internet access through heterogeneous wireless networks will become more and more popular. Since TCP congestion control is designed for stationary hosts, TCP performs quite badly when users move around these heterogeneous wireless networks and handoff occurs frequently. This problem is investigated further in this thesis, and *TCP-HO*, a *sender+receiver centric* practical adaptation for handoff, is proposed to improve the performance of TCP in heterogeneous mobile environments through exploiting explicit cooperation between fixed servers and mobile hosts. TCP-HO has been implemented in FreeBSD. Experimental results indicate that in heterogeneous mobile environments, TCP-HO can improve TCP performance substantially without adversely affecting cross traffic, even while a mobile host has only a coarse estimation of new wireless link's bandwidth. Considering that more and more users are accessing the Internet through heterogeneous wireless networks and mobile host could have a coarse estimation of wireless link's bandwidth, it should be worthwhile to change both fixed server and mobile host for improving the performance of TCP.

Secondly, as bandwidth in the Internet continues to grow, there will be more and more long fat network pipes with abundant residual bandwidth. It is well known that TCP cannot work well on these network pipes and a new high speed congestion control (HSCC) algorithm is needed by bandwidth-greedy and elastic applications for efficient utilization of the abundant bandwidth. Considering that there are many different kinds of applications in the Internet, the tradeoff between efficiency and friendliness is investigated further in this thesis. In this thesis, *Sync-TCP*, a *sender-centric* delay-based HSCC algorithm, is also proposed to safely ramp up the throughput of bandwidth-greedy and elastic applications. Based on queue delay (a *noisy* and *delayed* network feedback), Sync-TCP is designed to drive the network to operate around the knee and to distribute residual bandwidth fairly among competing flows, even when the number of competing flows varies and their round trip propagation delays differ significantly. Sync-TCP has been implemented in NS-2 and FreeBSD. Extensive simulations and preliminary testbed evaluations show that Sync-TCP achieves its design goals and it performs better than existing HSCC approaches including Fast TCP, Compound TCP and Cubic-TCP, especially in the trade-off between throughput and friendliness.

Thirdly, these new types of networks not only bring challenges to TCP protocol, they also bring challenges on how to implement TCP. With their deployment, the Internet is becoming a highly heterogeneous inter-network and it will keep evolving continuously. Hence, TCP implementation of a host needs to run on different kinds of network pipes, and the classical TCP implementation, that uses the same congestion control mechanism for all, cannot always achieve good performance. In this thesis, *TCP KentRidge*, a new TCP implementation framework, is proposed for the heterogeneous and evolving Internet. This new framework is carefully designed so that new congestion control mechanisms can be added conveniently for new types of networks, and the host can intelligently apply the most appropriate congestion control mechanism to each connection based on its current environment. An initial prototype of TCP KentRidge has been implemented in FreeBSD.

At the end of this thesis, future works relating to Sync-TCP and TCP KentRidge are also discussed.

List of Tables

List of Figures

Chapter 1

Introduction

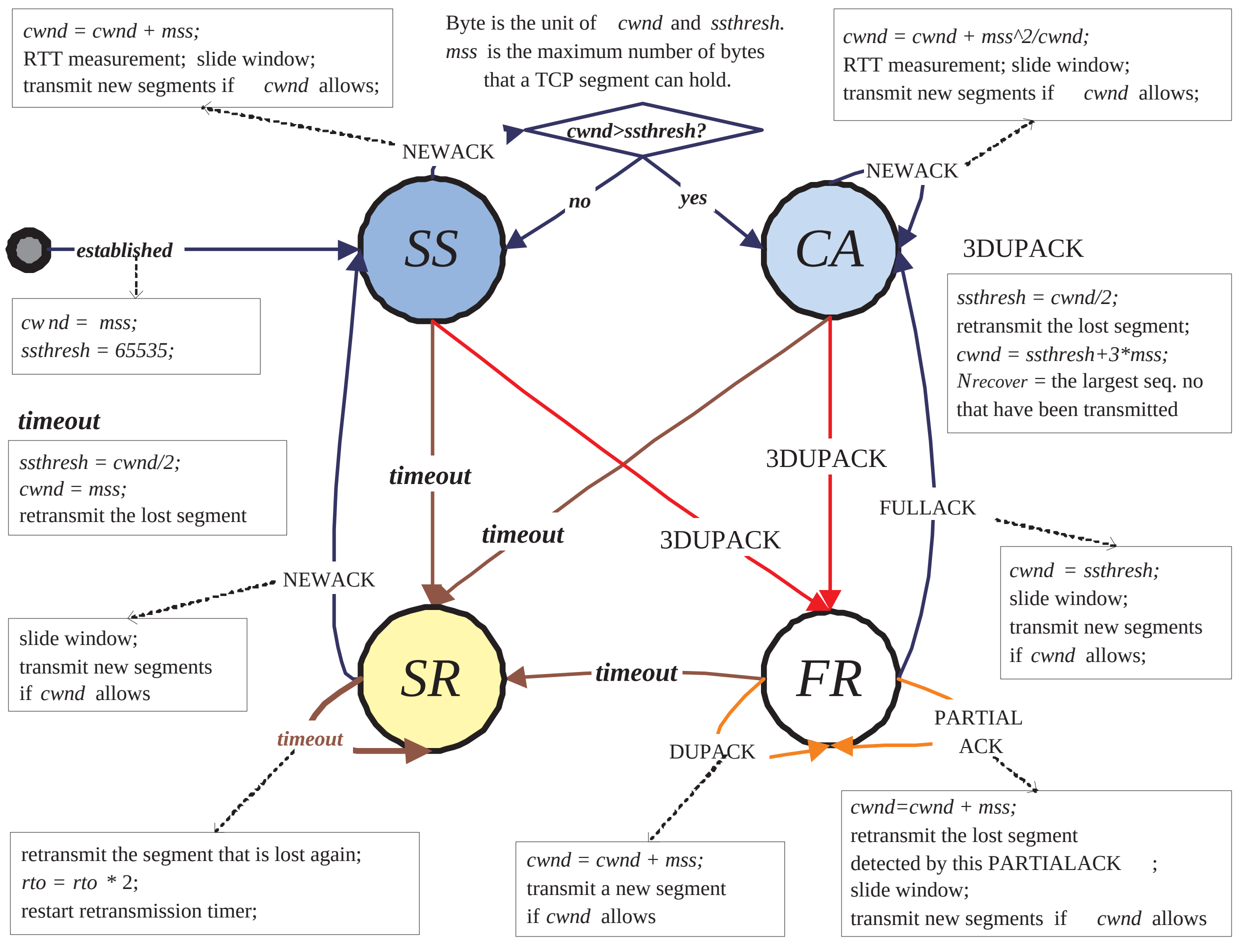
1.1 TCP Congestion Control Mechanism

Everyday innumerable business and personal activities are being carried out over the Internet, and as such the Internet has become an indispensable entity in our lives. The cornerstone of the Internet is the TCP/IP protocol suite. IP [111] is the glue that holds heterogeneous networks together and provides necessary functions to transfer packets over these networks. TCP [112] provides an attractive connection-oriented end-to-end service and ensures a reliable and in-order transfer of data. Due to its congestion control mechanism, TCP has also provided good performance to network applications and kept the stability of the exponentially increasing Internet.

TCP congestion control is responsible to probe network capacity, respond to network dynamics, and maintain network stability. Over these years, it has become one of the most active research areas. TCP congestion control was originally designed for highly reliable links with low/medium bandwidth and stationary hosts [70]. With better understanding of the Internet behavior, several new TCP versions (TCP Reno [17], TCP Newreno [53], and TCP SACK [100]) are designed and standardized to improve the performance of TCP. In the following subsection, more details of TCP Newreno, the latest and the most widely deployed version of TCP congestion control, will be presented.

1.1.1 TCP Newreno

For carrying out congestion control, TCP Newreno sender maintains two variables, *cwnd* and *ssthresh*. Along with the current round trip time, *cwnd* determines the current sending rate. As for *ssthresh*, it can be regarded as the coarse estimation of the fair share bandwidth delay product. Based on the state transition diagram shown in figure 1.1, the behaviors of TCP Newreno are described in the following paragraphs.



cwnd = cwnd + mss; RTT measurement; slide window; transmit new segments if cwnd allows;
Byte is the unit of cwnd and ssthresh. mss is the maximum number of bytes that a TCP segment can hold.
cwnd = cwnd + mss^2/cwnd; RTT measurement; slide window; transmit new segments if cwnd allows;
cwnd>ssthresh?
no
yes
NEWACK
NEWACK
established
SS
CA
3DUPACK
cwnd = mss; ssthresh = 65535;
ssthresh = cwnd/2; retransmit the lost segment; cwnd = ssthresh+3*mss; Nrecover = the largest seq. no that have been transmitted
timeout
ssthresh = cwnd/2; cwnd = mss; retransmit the lost segment
timeout
timeout
3DUPACK
3DUPACK
FULLACK
NEWACK
cwnd = ssthresh; slide window; transmit new segments if cwnd allows;
slide window; transmit new segments if cwnd allows
SR
timeout
FR
timeout
PARTIAL ACK
DUPACK
retransmit the segment that is lost again; rto = rto * 2; restart retransmission timer;
cwnd = cwnd + mss; transmit a new segment if cwnd allows
cwnd=cwnd + mss; retransmit the lost segment detected by this PARTIALACK ; slide window; transmit new segments if cwnd allows

Figure 1.1: State Transition Diagram of a TCP Newreno Sender

After a connection is established, *cwnd* is initialized to one segment, i.e., a packet that includes at most *mss* bytes, *ssthresh* is set to 65535 bytes, and the sender begins to increase *cwnd* for probing network capacity.

In order to probe the network capacity well and avoid frequent network congestion, two sub-states of capacity probing are used by TCP Newreno. If $cwnd < ssthresh$, TCP sender is in *SS* (slow start) state and *cwnd* is increased by one segment for each incoming NEWACK, that acknowledges some new data. Consequently, *cwnd* is doubled per RTT. Hence, *cwnd* is increased exponentially in *SS* state so that TCP sender can quickly reach its fair share of network capacity. If $cwnd \geq ssthresh$, TCP sender is in *CA* (congestion avoidance) state and *cwnd* is increased by one segment per RTT. Hence, *cwnd* is increased linearly in *CA* state so that TCP sender can keep probing network capacity without causing network congestion frequently.

When probing network capacity, the sender also carries out congestion detection. TCP Newreno regards segment loss as the only signal of network congestion. And two signals, *timeout* (the expiration of TCP retransmission timer) and 3DUPACK (the arrival of three consecutive duplicate ACK packets), are used for segment loss detection. Compared to *timeout*, 3DUPACK can detect congestion earlier when the segments, that follow the lost segment, still can arrive the receiver and trigger the duplicate ACK packets. But 3DUPACK may give a false congestion signal in networks with packet-reordering. *timeout* is the most reliable congestion signal, and *rto* (the *timeout* value of TCP retransmission timer) is updated based on the smoothed average and the variance of RTT samples. RTT is measured when TCP sender receives a NEWACK. If Timestamp option is used, TCP sender can get one RTT sample per NEWACK. Otherwise, only one RTT sample can be measured per window of data. When *rto* is calculated according to RTT samples, *rto* should be large enough to tolerate RTT variance. On the other hand, *rto* should not be too large since large *rto* will slow down TCP sender's response to severe network congestion.

When congestion is detected in *SS* or *CA* state, the lost segment is retransmitted and *ssthresh* is set to half of the current *cwnd*. If congestion is detected by *timeout*, *cwnd* is set to one segment, and the sender enters into *SR* (Slow Recovery), a kind of congestion recovery state. If congestion is detected by 3DUPACK, *cwnd* is set to $ssthresh + 3 * mss$ and the sender enters into another congestion recovery state, *FR* (fast recovery).

In *SR* state, if *timeout* is detected, it indicates that the retransmitted segment is lost again. In this case, the segment is retransmitted again and a binary exponential back-off algorithm is adopted by TCP retransmission timer. More specifically, *rto* is doubled to reduce the sending rate further. When a NEWACK is received in *SR* state, it indicates that the retransmitted segment is received correctly and the network has recovered from congestion. Hence, the sender will slide its sending window, discard acknowledged data, and enter into *SS* state.

In *FR* state, if *timeout* is detected, the sender will halve *ssthresh* again, set *cwnd* to one segment, and enter into the *SR* state. When a DUPACK is received in *FR* state, *cwnd* is increased by one segment so that TCP sender can keep filling the network pipe. As for NEWACK received in *FR* state, TCP Newreno further differentiates NEWACK into PARTIALACK, which acknowledges part of the segments sent out before the first lost segment is detected, and FULLACK, which acknowledges all segments sent out before the first lost segment is detected. TCP Newreno exits *FR* state only when FULLACK is received. When a PARTIALACK is received, the sender slides its sending window, discards acknowledged data, and increases *cwnd* by one segment. The sender also retransmits the lost segment just detected by this PARTIALACK, and if its sending window allows, new segments are transmitted too. Through this scheme, TCP Newreno can avoid to reduce *cwnd* more than once when multiple segments are dropped in one congestion event.

According to the above description, the core functions of TCP congestion control can be divided into *initialization*, *capacity probing*, *congestion detection*, *congestion recovery*, and *RTT measurement*. Table 1.1 summarizes the corresponding algorithms used by TCP Newreno. Since retransmission is tightly involved with congestion control, this function is also listed. In summary, TCP Newreno is a simple sender-centric, loss and window based congestion control algorithm designed for reliable links with low/medium bandwidth and

Components	**Algorithms**
initialization	$cwnd = mss$, $ssthresh = 65535$
capacity probing	exponential increase in SS: $cwnd = cwnd + mss$ per NEWACK linear increase in CA: $cwnd = cwnd + mss$ per RTT
congestion detection	*timeout* and 3DUPACK
congestion recovery	*timeout*: $ssthresh = cwnd/2, cwnd = mss$ 3DUPACK: $ssthresh = cwnd/2$; $cwnd = ssthresh + 3 * mss$ persistent loss: $rto = rto * 2$
RTT measurement	one sample per window of data, or one sample per NEWACK if Timestamp option is used. $\Delta = srtt - rtt, srtt = rtt/8 + srtt * 7/8$ $rttvar = rttvar * 3/4 + \|\Delta\|/4, rto = srtt + 4 * rttvar$
retransmission	retransmit the lost segment detected by *timeout* or 3DUPACK. PARTIALACK is also used to detect and retransmit the segments that are lost in the same congestion event

Table 1.1: Algorithms Used by TCP Newreno

stationary hosts. When segment loss is detected mainly through 3DUPACK, *cwnd* is regulated by linear/additive increase algorithm in *CA* state and multiplicative decrease (by half) algorithm when congestion is detected. This is why TCP congestion control is regarded as an AIMD (Additive Increase and Multiplicative Decrease) algorithm with fixed parameters [43]. TCP congestion control is also frequently characterized as AIMD(1, 0.5), i.e., *cwnd* is increased by one segment per RTT in *CA* state and it is decreased by half when segment loss is detected [146].

1.2 Problem Formulation

Over the years, TCP has become the dominant transport protocol of the Internet and its congestion control algorithm has also become the de facto standard. TCP has facilitated the development of various network applications, such as FTP, Telnet, and WWW, which are responsible for the enormous success of the Internet.

However, in recent years, many new types of networks with different characteristics have been deployed. Within these new types of networks, the original assumptions of TCP congestion control are frequently undermined, and TCP performs very badly. Considering the current wide deployment of TCP, TCP will continue to be the dominant transport protocol in the foreseeable future, and it is very valuable to improve the performance of TCP in these networks.

In this thesis, three major trends are noticed in the ever expanding Internet. Firstly, more and more users are accessing the Internet through various wireless networks, such as WCDMA [1], Wi-Fi [8], etc. Secondly, the bandwidth of the Internet is increasing very quickly and there will be more and more long fat network pipes. Hence, it is imperative that one should improve TCP performance in these two network scenarios. Thirdly, we also observe that the Internet is becoming more and more heterogeneous and it still keeps changing continuously. These changes bring many challenges to TCP implementation, and a new implementation framework, along with TCP adaptations proposed for different networks, is necessary to provide efficient service to users in the heterogeneous and evolving Internet.

This thesis focuses on the above three problems, and their details are discussed further in the following subsections.

1.2.1 Improving TCP Performance in Heterogeneous Mobile Environments

In recent years, many kinds of wireless networks, such as cellular network (WCDMA [1], etc.) and Wireless LAN (Wi-Fi [8], etc.), have been deployed and have become integral parts of the Internet. These wireless networks complement each other in terms of coverage, bandwidth, latency, etc. and form a heterogeneous mobile environment. These networks along with portable and affordable computing devices, such as laptops and PDAs that are installed with multiple different kinds of wireless interface cards, enable wide spread and affordable mobile Internet access. Figure 1.2 illustrates a typical scenario of mobile Internet access through heterogeneous wireless networks.

But TCP congestion control was designed for reliable links and stationery hosts. The

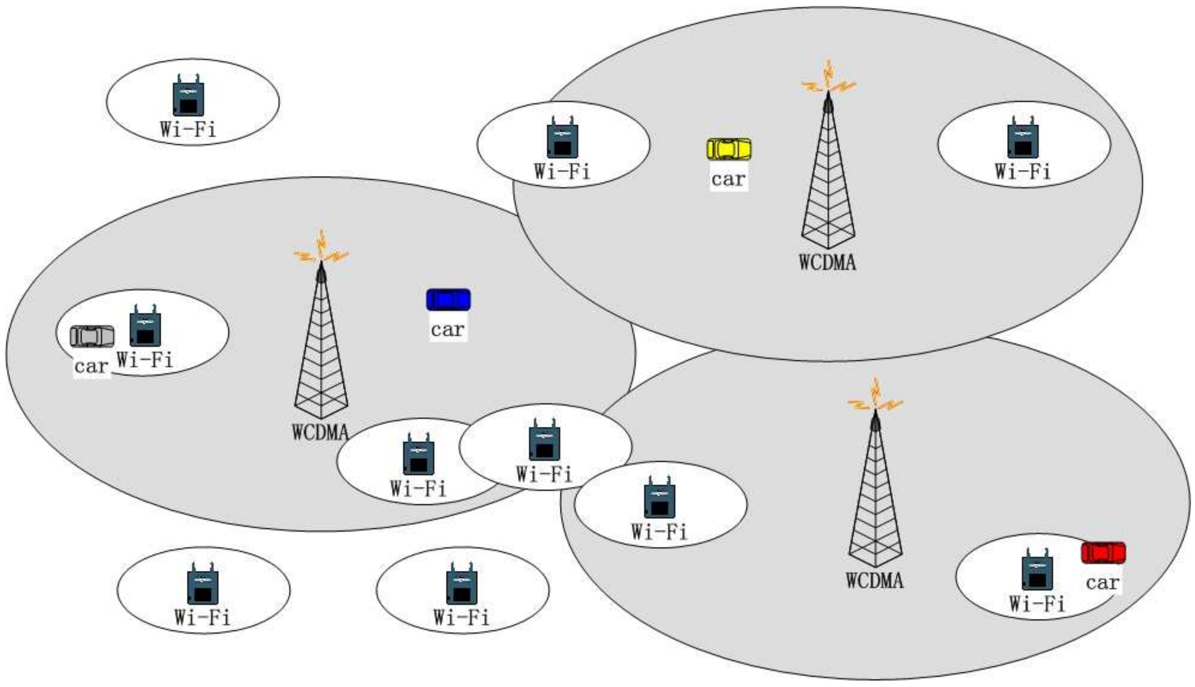


Figure 1.2: Mobile Internet Access through Heterogeneous Wireless Networks: Cars Run Around a Campus Covered by WCDMA Network and Wi-Fi Hot-spots

characteristics of wireless networks, such as lossy wireless link and user mobility, bring severe problems to TCP. Hence, it is very important to improve TCP performance in wireless networks. This area has been a research hot-spot for quite a long time and many solutions have been proposed so that TCP can work well on lossy wireless links. In this thesis, we focus on the challenges brought by user mobility in a heterogeneous mobile environment. Handoff will occur when a user moves around and switches among different wireless channels. Compared with common network dynamics, network path characteristics could vary much more significantly during handoff. This thesis tries to address the following questions.

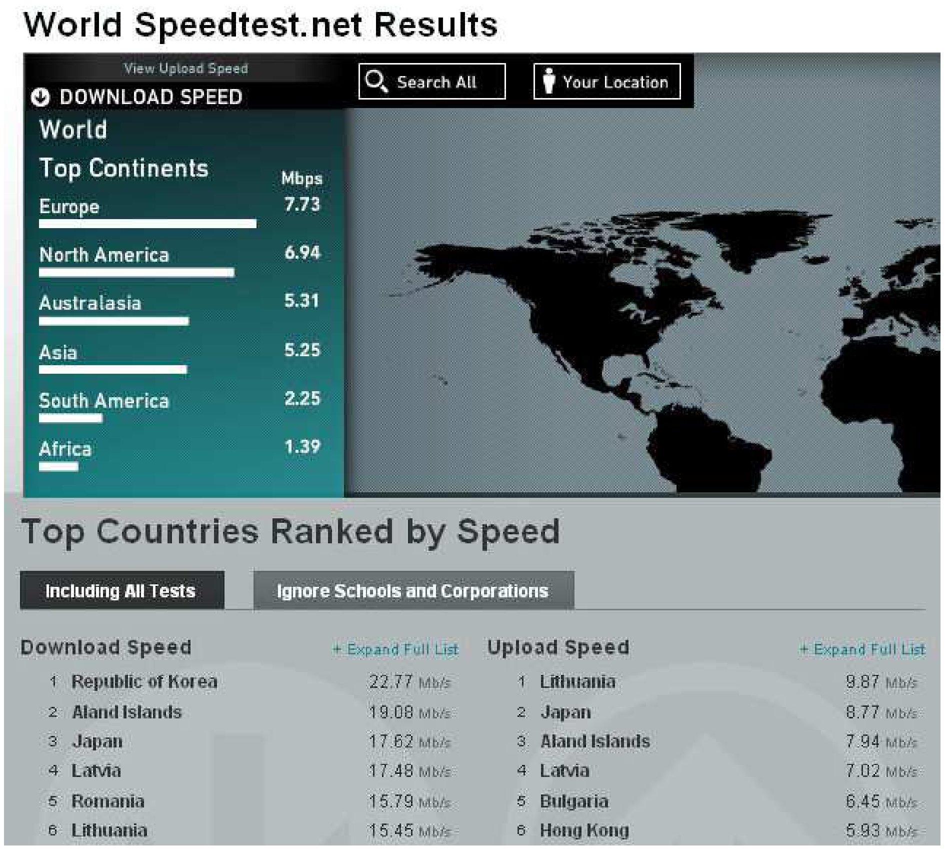
1. What kinds of handoff may occur in a heterogeneous mobile environment? What are the challenges faced by TCP during each kind of handoff?

2. How to design a mechanism so that it could systematically solve the challenges brought by all kinds of handoff? This mechanism should be deployable in the heterogeneous mobile environment, and it should not bring new problems. For example, TCP flows

supported by this mechanism should not hurt the cross traffic, and users cannot cheat through this mechanism to acquire (un-fairly) more bandwidth.

1.2.2 Improving TCP Performance on Long Fat Network Pipes

The bandwidth of the Internet has been increasing very quickly. For example, expected capacity of the forthcoming TPE (Trans-Pacific Express) is 5.12 Tera-bps and FTTx (Fiber To The Home, Building, etc.) has been widely deployed in many countries. Recent bandwidth measurement statistics (figure 1.3) show that the average download speeds vary from 7.73Mbps in Europe to 1.39Mbps in Africa. The average download speed for the top 6 countries already exceeds 15Mbps. In some countries, for example, Singapore, the goal is to provide up to 1Gbps broadband access by 2015.



World

Top Continents

	Mbps
Europe	7.73
North America	6.94
Australasia	5.31
Asia	5.25
South America	2.25
Africa	1.39

Top Countries Ranked by Speed

Download Speed			Upload Speed		
1	Republic of Korea	22.77 Mb/s	1	Lithuania	9.87 Mb/s
2	Aland Islands	19.08 Mb/s	2	Japan	8.77 Mb/s
3	Japan	17.62 Mb/s	3	Aland Islands	7.94 Mb/s
4	Latvia	17.48 Mb/s	4	Latvia	7.02 Mb/s
5	Romania	15.79 Mb/s	5	Bulgaria	6.45 Mb/s
6	Lithuania	15.45 Mb/s	6	Hong Kong	5.93 Mb/s

Figure 1.3: Bandwidth Measurement Statistics (http://www.speedtest.net/) on 2010-03-12

As the bandwidth of the Internet continues to grow, there will be more and more long fat network pipes with abundant residual bandwidth. However, it is well known that TCP cannot work well on long fat network pipe where the bandwidth-delay product (BDP) is large [51]. Due to the loss based AIMD(1, 0.5) algorithm, legacy TCP versions (TCP Reno, Newreno, SACK, etc.) cannot send out data fast enough.

In order to address this problem, many high speed congestion control (HSCC) algorithms, such as Highspeed TCP [51], Cubic-TCP [59], H-TCP [91], Fast TCP [75], TCP Illinois [94], and Compound TCP (CTCP) [125], have been proposed in recent years. Some of them have also been deployed in popular operating systems. For example, Compound TCP has been distributed with Windows Vista, and Linux has selected Cubic-TCP as its default congestion control mechanism.

For bandwidth-greedy and elastic applications, such as video/software distribution and P2P file sharing, it is now easy and irresistible to adopt a HSCC algorithm, which can efficiently utilize the abundant residual bandwidth and provide higher throughput to users on long fat network pipes of the Internet. On the other hand, bandwidth-greedy and elastic applications are not the only applications running in the Internet. Considering that a HSCC algorithm probes the network more aggressively for higher throughput, it should pay more attention on friendliness to cross traffic. In particular, its deployment should not hurt the applications using legacy TCP and the interactive applications, such as web surfing and media-streaming that are more important and profitable to network providers. More specifically, existence of HSCC-based bandwidth-greedy and elastic applications should not significantly increase packet loss rate, queue delay, and jitter experienced by these cross traffic applications. However, the existing proposals cannot satisfy this criteria [142].

As pointed out in [73], an end-to-end delay-based congestion control algorithm has the potential of driving the network to operate around the knee, at which network throughput is high, queue delay is short, and packet drop rate is minimum. Hence, a delay-based HSCC algorithm may enable bandwidth-greedy and elastic applications to efficiently utilize long fat network pipes while not hurting the cross traffic. In this thesis, we try to address the following questions encountered by such a delay-based HSCC algorithm.

1. How to learn network state correctly based on queue delay, a *noisy* and *delayed* network feedback? Estimation of RTPD (Round Trip Propagation Delay) is already a very challenging task since it is highly correlated with congestion control behaviors of distributed senders [49].

2. How to drive the network to operate around the knee and distribute bandwidth fairly among competing flows, irrespective of the number of competing flows and the values of their RTPD?

3. How to solve the challenges brought by re-routing, varying loads of cross traffic generated by different kinds of applications, etc.?

1.2.3 Re-engineering TCP Implementation for the Heterogeneous and Evolving Internet

The Internet has become a highly heterogeneous inter-network comprising networks with varying characteristics (bandwidth, delay, packet error rate, etc.), such as optical network, satellite network, Ethernet, ADSL, Wi-Fi, etc. The routers may also have different queue sizes and adopt different queue management schemes. Figure 1.4 illustrates the heterogeneity of the current Internet in part.

Within the highly heterogeneous Internet, a host needs to communicate with other hosts that spread across the globe. Hence, TCP implementation of a host needs to run on network pipes with different characteristics. On many of these network pipes, the assumptions of TCP congestion control are frequently violated and a connection with standard TCP congestion control mechanism can only utilize a few percent of the provisioned bandwidth [26][51].

In recent years, significant research work have been done to improve TCP performance on different kinds of network pipes, and many TCP adaptations have been proposed for these network pipes [16][22][45][46][60][69][104]. But, very few of them have been implemented

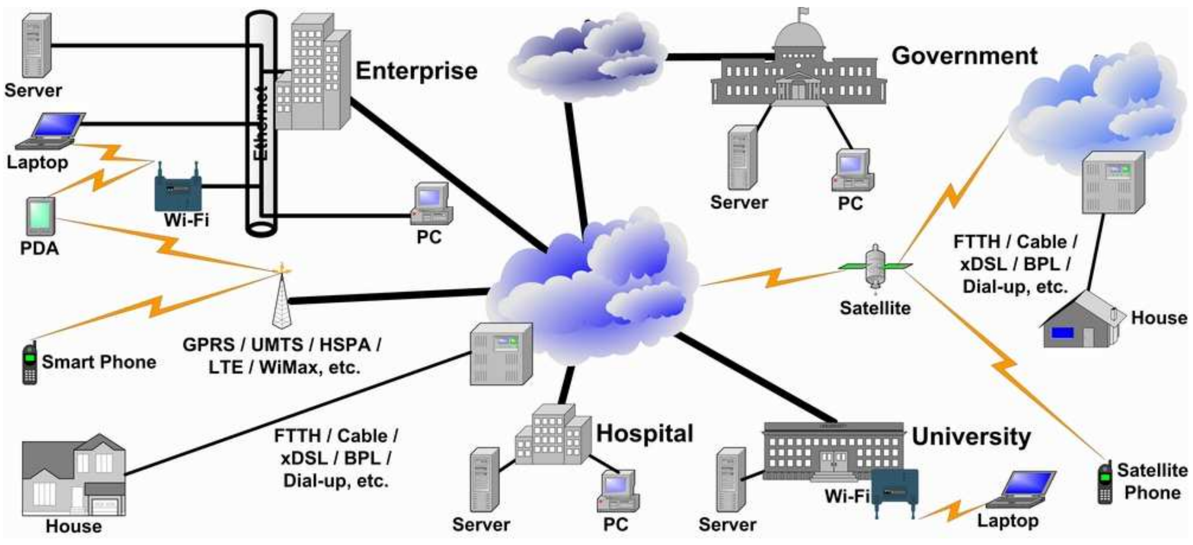


Figure 1.4: The Highly Heterogeneous Internet: an Example

in the popular operating systems. The classical TCP implementation still uses the same standardized congestion control mechanism for all active connections and cannot always achieve good performance in the heterogeneous Internet. An intelligent TCP, with which a host can automatically apply the most appropriate TCP adaptation on each connection according to the current end-to-end path characteristics, will be useful. However, there are a very few studies on how to implement such an ideal TCP.

In addition, many hosts with different resources, have been attached to the current Internet, different operating systems are installed on these hosts, and applications with various expectations are also running on these hosts. An intelligent TCP should also handle the heterogeneity in these aspects.

Furthermore, the Internet also keeps changing continuously. The topology and links' bandwidth change with the deployment and/or upgrade of network infrastructure. New networks technologies, such as Wi-Max [5], will be deployed soon, and new network applications will also be introduced. Hence, it is necessary to re-engineer TCP implementation for solving the above challenges systematically, and any redesign has to evolve with the changing Internet technologies. In this thesis, we try to address the following questions encountered by such an intelligent TCP implementation.

1. How to implement a large number of existing TCP adaptations without hurting the maintainability of TCP source codes? How to facilitate the implementation of new TCP adaptations that will be proposed in the future?

2. How to learn the environment of a TCP connection and select the most appropriate adaptation accordingly? How to enable a connection to change its behaviors according to the changes of the environment?

3. How to optimize the implementation for saving system resources?

1.3 Thesis Contributions

This thesis focuses on the above three problems and has made the following contributions.

1. TCP HandOff (TCP-HO): A practical TCP adaptation for mobile Internet access through heterogeneous wireless networks.

2. Synchronized TCP (Sync-TCP): A novel delay-based high speed congestion control algorithm for safely ramping up the throughput of bandwidth-greedy and elastic applications of the Internet.

3. TCP KentRidge: A new TCP implementation framework for providing efficient service to users in the context of the heterogeneous and evolving Internet.

1.3.1 TCP-HO

Based on the observation that congestion control is carried out by the server and handoff is only known by mobile host, TCP-HO resorts to explicit cooperation (between server and mobile host) to solve challenges brought by heterogeneous mobile environments and improve the performance of mobile host while not hurting the other traffics.

TCP-HO is designed based on the assumptions that a mobile host is able to detect the completion of handoff immediately and has a coarse estimation of new wireless link's bandwidth. When the completion of handoff is detected by mobile host, it notifies the server about the new wireless link's bandwidth through two DUPACK packets. After the server receives the notification, it starts to transmit immediately and starts to update its *ssthresh* based on bandwidth notification and RTT of the new path for a while. Compared to the existing approaches, the assumptions of TCP-HO are more reasonable in the real world. TCP-HO is also designed as a pure end-to-end proposal for facilitating the deployment. Furthermore, the server responds to bandwidth notification only after it just experienced *timeout* and it starts to transmit with $cwnd=mss$. Hence, TCP-HO can also thwart cheating users and avoid to hurt cross traffic.

Experimental results indicate that TCP-HO achieves its design goals. Considering that more and more users are accessing the Internet through heterogeneous wireless networks, it should be worthwhile to implement TCP-HO at both server and mobile host for improving the performance of TCP. *Hence, this thesis provides a promising solution (TCP-HO) for mobile Internet access through heterogeneous wireless networks.*

1.3.2 Sync-TCP

Considering that the Internet is now very important to the world, the deployment of any HSCC algorithm should not degrade user experience of the existing applications, especially the interactive ones, such as web-surfing and media-streaming. Sync-TCP is perhaps the first one that is designed to achieve this goal.

The key insight of Sync-TCP is that if competing flows could detect the same congestion signal through queue delay, these flows can then coordinate their congestion control behaviors for driving the network to operate around their desired point, the knee. Sync-TCP is carefully designed such that with high probability, competing flows can detect the same congestion signal through queue delay. In combination with synchronized congestion signal, Sync-TCP uses an adaptive queue-delay-based congestion window decrease rule and a RTT-independent congestion window increase rule. These rules are designed to drive the network to operate around the knee and to distribute the residual bandwidth fairly even when the number of competing flows varies and their RTPDs differ significantly.

Extensive simulations and preliminary testbed evaluations indicate that Sync-TCP achieves its design goals and performs better than the existing proposals. *Hence, this thesis also provides a promising solution (Sync-TCP) for safely ramping up the throughput of bandwidth-greedy and elastic applications that run on long fat network pipes of the Internet.*

1.3.3 TCP KentRidge

TCP KentRidge is the first proposal that systematically considers the challenges faced by a TCP implementation in the heterogeneous and evolving Internet.

TCP KentRidge is carefully designed so that a host could have the potential of automatically applying the most appropriate TCP adaptation to each connection based on the current environment. TCP KentRidge is also designed so that new TCP adaptations can be implemented in this framework and the necessary intelligence can be added easily. An initial prototype of TCP KentRidge has been implemented in FreeBSD using which a host could automatically and intelligently change its behaviors according to the environment.

1.4 Thesis Organization

Chapter 2 presents TCP HandOff (TCP-HO). This chapter first classifies handoff that may occur in heterogeneous mobile environments, and analyzes the challenges faced by TCP during each kind of handoff. The details of TCP-HO and experimental results are then presented. At the end of this chapter, wireless link bandwidth estimation issues are discussed, TCP-HO's sensitivity to bandwidth estimation error is analyzed, and its performance under achievable bandwidth estimation accuracy is evaluated too.

Chapter 3 presents Synchronized TCP (Sync-TCP). This chapter first introduces the existing delay-based congestion control algorithms and discusses the challenges that Sync-TCP must solve to achieve its design goals. The design of Sync-TCP is then presented in details. Parameter selection guidelines and deployment issues are also discussed. Finally, extensive simulations and preliminary testbed evaluations are presented in details.

Chapter 4 presents TCP KentRidge. The existing TCP adaptations proposed for different networks are first summarized. This chapter then discusses the challenges faced by TCP implementation within the heterogeneous and evolving Internet. State of the art TCP implementations are also introduced. After that, the design details of TCP KentRidge and an initial prototype implementation are presented.

Chapter 5 concludes this thesis with a summary and points out some future works.

Chapter 2

TCP-HO: A Practical Adaptation for Heterogeneous Mobile Environments

2.1 Introduction

In recent years, many kinds of wireless networks, such as cellular network (WCDMA [1], GPRS [6], etc.) and Wireless LAN (Wi-Fi [8][9][10][12], HiperLAN [7], etc.), have been deployed and have become integral parts of the Internet. These wireless networks complement each other in terms of coverage, bandwidth, latency, etc. and form a heterogeneous mobile environment. These networks along with portable and affordable computing devices, such as laptops, PDAs, and smart phones, enable the wide spread and affordable mobile Internet access. But the lossy wireless links and mobile hosts violate the assumptions of TCP, the most widely used transport protocol of the Internet. As a result, TCP performs very bad when users move around in these wireless networks.

TCP was designed for reliable links. When packets are transmitted on lossy wireless link, they are corrupted frequently. These corrupted packets are wrongly regarded as congestion signals and TCP sender reduces its sending rate unnecessarily. Hence, TCP cannot efficiently utilize the precious bandwidth of wireless link and provides very bad performance to applications. Many mechanisms, such as I-TCP [21], Snoop [24], TCP Veno [56], TCP Hack [117], and TCP ELN [25][83][140], had been proposed for enhancing TCP performance over lossy wireless links. Based on the above investigations, new wireless networks normally adopt FEC (Forward Error Correction) and/or ARQ (Automatic Retransmission reQuest) in link layer with the aim to hide lossy characteristic of wireless link. For example, RLC [11], the link layer protocol of GPRS/WCDMA, adopts link layer ARQ. Link layer ARQ is also used by the MAC layer of IEEE 802.11 [8]. Hence, the current wireless networks have already become reliable wireless networks. In reliable wireless networks, link layer ARQ may bring large bandwidth and delay variation. This problem has also been solved by regulating the sending rate of ACK packets or changing *awnd* (advertised window) of ACK packets at the base station [41][42].

The challenges brought by lossy wireless link had been well studied. But TCP was also designed for stationary hosts. When a user moves around in these heterogeneous wireless networks, TCP performance is very poor due to the challenges brought by handoff. This chapter will focus on how to enhance TCP performance when users move around in the heterogeneous mobile environments.

Within the heterogeneous mobile environments, many kinds of handoff may occur and they can be classified in many different ways. For example, in [121], handoff is classified into vertical handoff and horizontal handoff according to the techniques used by the wireless networks. In this chapter, handoff is classified from TCP point of view. More specifically, since the BDP (Bandwidth-Delay Product) of a network path affects TCP performance significantly, handoff is classified according to BDP of the old network path (before handoff) and that of the new network path (after handoff).

Handoff is first classified into HH (horizontal handoff) and VH (vertical handoff) based on whether BDP changes significantly during handoff. Within HH, BDPs of new and old paths do not vary appreciably. For VH, the difference in BDP is large. According to the value of BDP, HH is further divided into L-HH and H-HH. Within L-HH, BDPs of both paths are low. Within H-HH, BDPs of both paths are high. According to the direction of BDP change, VH is further divided into D-VH (Downward-VH) and U-VH (Upward-VH) [102]. During all kinds of handoff, there is normally a long disconnection time, which means a period of *zero* BDP. Figure 2.1 shows BDP fingerprints of the above four kinds of handoff.

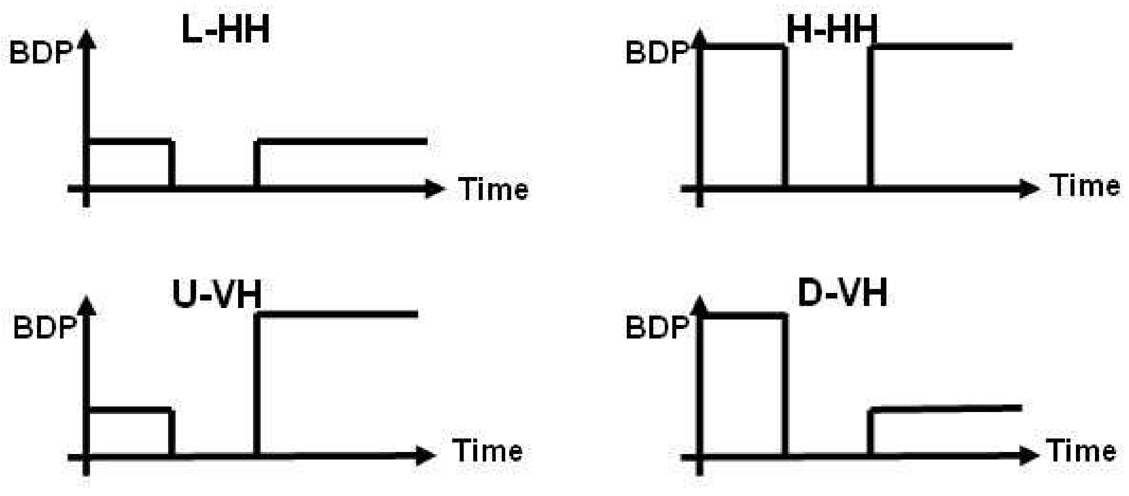


Figure 2.1: BDP Fingerprints of Different Kinds of Handoff

Although TCP reacts to the changes of network capacity through congestion control, abrupt BDP changes during handoff can still bring severe challenges. In this chapter, the potential benefits of bringing explicit cooperation between server and mobile host are studied. TCP HandOff (TCP-HO), a practical end-to-end mechanism based on explicit cooperation, is proposed for solving the challenges brought by all kinds of handoff with the aim to improve TCP performance in heterogeneous mobile environments.

The rest of this chapter is organized as follows. In section 2.2, the challenges faced by TCP during each kind of handoff are analyzed. Section 2.3 presents related work, and the details of TCP-HO are described in section 2.4. The testbed and experimental results are presented in section 2.5, and the issues related with wireless link bandwidth estimation are discussed in section 2.6. This chapter is briefly summarized in section 2.7.

2.2 TCP During Handoff

TCP, the most widely used transport protocol, uses congestion control to probe network capacity and keep network stability. According to the standard TCP congestion control mechanism described in subsection 1.1.1, TCP sender maintains two variables, *cwnd* and *ssthresh*. When *cwnd* is less than *ssthresh*, the sender is in *SS* state. For each NEWACK, *cwnd* is increased by one segment so that TCP sender can quickly probe network capacity. When *cwnd* is larger than *ssthresh*, the sender is in *CA* state. And for each round trip time, *cwnd* is increased by one segment so that the sender can still probe network capacity and avoid frequent network congestion. TCP regards segment loss as the only signal of network congestion. When congestion is detected, *ssthresh* is reduced to half of the current *cwnd*. *cwnd* is reduced to one segment if congestion is detected by *timeout*. If congestion is detected by 3DUPACK, *cwnd* is set to $ssthresh + 3 * mss$. When a lost segment is detected by retransmission timer and has to be retransmitted more than once, TCP retransmission timer adopts an exponential back-off algorithm with a maximal *rto* value (64 seconds).

With TCP congestion control and its ACK-based self-clock, TCP sender can react to slow changes of network capacity and achieve good throughput. However, during handoff, BDP changes abruptly and long disconnection destroys TCP's self-clock. In the following paragraphs, the problems faced by TCP during all kinds of handoff will be analyzed further.

Firstly, *segments are lost because that the sender does not know the occurrence of handoff.* The sender keeps sending data to the old base station although the old wireless link has broken. Unless seamless handoff is implemented and the old base station can forward these packets to the new base station, many segments are lost during handoff. For heterogeneous wireless data access networks, availability of seamless handoff is unlikely due to standardization issues, administration issues, etc. As a result, when handoff occurs, segments will be discarded at the base station of previous wireless link.

Secondly, *silly waiting occurs because that the sender does not know the completion of handoff.* After handoff is completed and the new wireless link is available, TCP sender does not know this fact and still sillily waits for *timeout*. Due to TCP retransmission timer's exponential back-off algorithm and long disconnection time of handoff, the sender may wait for quite a long time and precious bandwidth of new wireless link cannot be utilized.

Thirdly, *since the sender does not know the bandwidth difference between the new wireless link and the old wireless link, slow start or over-shooting may also occur.* After handoff completion, the sending rate is quite slow for a long time. During handoff, timeout occurs at TCP sender, *ssthresh* is reduced to half of the *cwnd*, and *cwnd* is reduced to one. TCP sender needs some time to efficiently utilize the bandwidth of new link. Especially when *ssthresh* is much less than BDP of new path, TCP sender will enter into *CA* state too early. This problem is worse in H-HH than in L-HH, and it is much worse in U-VH because the BDP of new path is much larger than *ssthresh*, a coarse estimation of old path's BDP.

After the completion of D-VH, TCP sender may also over-shoot the new wireless link. Within D-VH, the BDP of new path is much less than *ssthresh*, a coarse estimation of old path's BDP. The exponential *cwnd* increase in *SS* state may cause multiple segment loss, trigger timeout, and adversely affect TCP throughput.

Table 2.1 summarizes the problems faced by TCP during four kinds of handoff that may occur in the heterogeneous mobile environments.

	Segment Loss	*Silly Waiting*	*Slow Start*	*Over-shooting*
L-HH	yes	yes	a little	no
H-HH	yes	yes	severe	no
U-VH	yes	yes	very severe	no
D-VH	yes	yes	very little	yes

Table 2.1: Problems Brought by Different Kinds of Handoff

2.3 Related Work

The root of TCP's poor performance during handoff is that congestion control is carried out by the server and handoff is only known to mobile host. Based on this observation, the related works are classified according to where adaptations take place.

1. Network-Centric Approaches: I-TCP [21] and MTCP [145] split a connection between a server and a mobile host at the base station. Handoff is entirely handled by mobile host and base stations. TCP server continues to send segments to the old base station during a handoff. These segments will be buffered at the old base station and forwarded to the new base station after handoff is completed. In this kind of approaches, handoff is hidden from TCP servers. But within heterogeneous mobile environments, wireless networks may use different techniques and belong to different administrative domains. These proposals may not be practical again. In addition, base stations need to maintain large buffers within high-speed wireless networks.

   M-TCP [34] only works at the base station. Base station holds back the acknowledgement of the last byte. When it detects handoff occurrence, it sends back the last byte's acknowledgement with *zero window*, which will force TCP sender to enter into persist mode, thus the values of *cwnd* and *ssthresh* are frozen. When a new base station detects handoff completion, it immediately notifies TCP sender to resume transmission with the frozen *cwnd* and *ssthresh*. M-TCP is a good network-centric solution. But the overhead of base station is too large and increases with the speed of wireless networks. More severely, M-TCP, I-TCP, and MTCP cannot work with IPSEC [81].

2. Receiver-Centric Approaches: RCP [67], which moves congestion control to mobile host, is the most radical proposal of this category. In RCP, mobile host carries out congestion control and notifies the current sending rate to the server. The server just needs to transmit data according to the sending rate specified by mobile host. Since mobile host knows handoff and carries out congestion control, it can solve all problems

brought by handoff. However, mobile host has a strong motivation to cheat with the server for acquiring high throughput. It is too dangerous to let mobile users decide the sending rate of more powerful fixed servers, especially in today's Internet.

Freeze-TCP [57] and TCP ACK-Pacing [102] are two other receiver-centric proposals. They change mobile host's behaviors during handoff with the aim to drive TCP server's congestion control to probe network capacity well. Freeze-TCP and TCP ACK-Pacing are end-to-end proposals and only mobile hosts need to be changed. Hence, they can work with IPSEC and have good deployability. But in the heterogeneous mobile environments, Freeze-TCP and TCP ACK-Pacing also have their own problems.

Freeze-TCP shifts the tasks of M-TCP's base station to mobile host and avoids hold the acknowledgement of the last byte. Freeze-TCP assumes accurate prediction of the occurrence of handoff. *zero window* must be sent out just before the predicted handoff occurs so that *zero window* can arrive the server for freezing its state and old wireless link will not be under-utilized. But it is very hard to accurately predict the occurrence of handoff, especially in heterogeneous wireless networks. After handoff completion, DUPACK is sent by mobile host to trigger the server to begin to transmit with previous frozen *cwnd* and *ssthresh*. This mechanism may work well during L-HH. But it violates the congestion control principle that data sender should re-probe network capacity after a long idle time [61]. Considering the fast and faster wireless networks, it may generate large data burst and harm cross traffics. In addition, Freeze-TCP does not consider the challenges brought by D-VH and U-VH.

TCP ACK-Pacing has considered the challenges brought by U-VH and D-VH. It assumes that a mobile host knows RTT of the new path and the bandwidth of the new wireless link. Hence it knows the BDP of the new path when the wireless link is the bottleneck. But mobile host normally acts as TCP receiver and does not keep measuring RTT. The ACK-generation algorithm of the mobile host is changed in order to drive the server's *cwnd* converge to new path's BDP quickly. However, less ACKs after

D-VH may cause TCP sender to generate large data burst and break TCP's self clock. More ACKs after U-VH may consume precious uplink bandwidth of (asymmetric) wireless link. In addition, TCP ACK-Pacing cannot work at all if TCP Byte-Counting [18] is used by the server. Finally, when the server and network allow TCP ACK-Pacing, it may be exploited by misbehaving users to get unfairly high throughput (through generating more ACK packets).

Since the root of TCP's poor performance during handoff is that congestion control is carried out by the server and handoff is only known to mobile host, it should be promising to introduce explicit cooperation between server and mobile host. In the following section, TCP HandOff (TCP-HO), a practical end-to-end TCP enhancement based on explicit cooperation, is designed for heterogeneous mobile environments. Within TCP-HO, a mobile host reports handoff information to the server. And the server is responsible to well utilize wireless links based on mobile host's feedback. Hence, TCP-HO is a *sender+receiver approach* and both TCP end-points need to be changed. Considering that more and more users are accessing the Internet through heterogeneous wireless networks, there should be enough motivations to change both the server and mobile host for implementing TCP-HO.

2.4 TCP HandOff Mechanism

During handoff, the ideal solution is to let TCP sender stop transmitting one RTT before handoff occurrence and immediately begin to transmit (*cwnd*=BDP of new path) after handoff completion. By this way, TCP sender can efficiently utilize the old and new wireless links and avoid any segment loss. However, this is not a practical solution. Below is the design principles and mechanism details of TCP-HO.

2.4.1 Design Principles

1. *Assumptions made must be reasonable.* Since it is very hard to accurately predict handoff in heterogeneous wireless networks, mobile host of TCP-HO does not predict handoff and notify the server. Hence, TCP-HO cannot avoid segment loss.

2. *The mechanism should be deployable in the current Internet.* In order to avoid complicating network infrastructure and work with IPSEC, TCP-HO should be a purely end-to-end mechanism. In addition, it should be easy to implement TCP-HO in the existing TCP codes.

3. *The mechanism should not bring new security problems.* Especially, when TCP sender accepts the notification of new link's bandwidth, it must use this information scrupulously and discard it quickly.

4. *The mechanism should be friendly to cross traffic.* With the increase of the number of mobile users and the increase of wireless link's bandwidth, TCP-HO must consider its effects on network stability and cross traffic. After handoff completion, instead of sending with full speed, TCP-HO must probe network capacity as same as normal TCP. In order to accelerate capacity probing, *ssthresh* can be set according to BDP of new path.

2.4.2 Details of TCP-HO

Based on the observation that congestion control is carried out by the server and handoff is only known by mobile host, TCP-HO resorts to explicit cooperation (between server and mobile host) to solve challenges brought by heterogeneous mobile environments and improve the performance of mobile host while not hurting the other traffics.

There are only two assumptions in TCP-HO. Firstly, a mobile host can immediately know the completion of handoff. This knowledge can be acquired easily by some cross-layer implementations. Secondly, a mobile host has a *coarse* estimation of new wireless link's bandwidth. This assumption is reasonable since mobile host can estimate bandwidth based on the type of wireless interface and the corresponding bandwidth estimation mechanisms, such as [139][147]. These bandwidth estimation mechanisms will be discussed further in section 2.6 when the effects of bandwidth estimation error are studied. Based on previous studies [76][78], the knowledge of the bottleneck link's bandwidth is helpful for improving TCP performance. By exploiting this knowledge carefully, the performance improvement of TCP-HO should be better than that of *receiver-centric* approaches.

When the completion of handoff is detected by mobile host, it notifies the server about the new wireless link's bandwidth through two DUPACK packets. After the server receives the notification, it starts to transmit immediately and starts to update its *ssthresh* based on bandwidth notification and RTT of the new path for a while. The details of TCP-HO are described below.

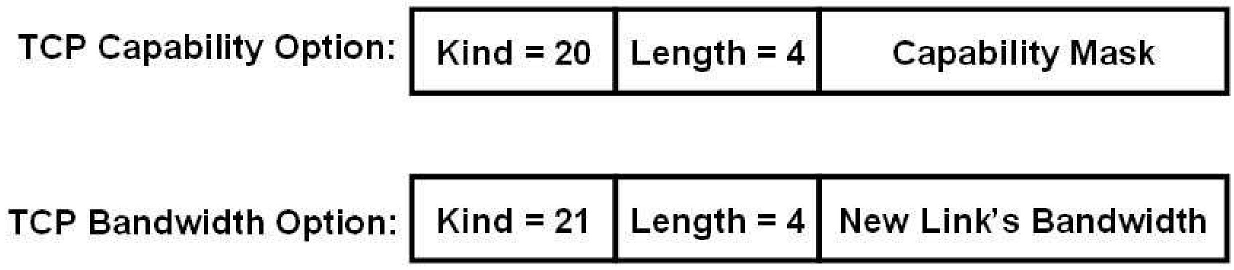


Figure 2.2: TCP Options for TCP-HO

TCP-HO extends TCP by including two new TCP options (see Figure 2.2). TCP Capability Option is an enabling option used in SYN segments. Each bit of its 16-bits capability mask can be used to negotiate one TCP capability. TCP-HO is negotiated through the last bit. TCP Capability Option is designed to save precious option space of a SYN segment. As for TCP Bandwidth Option, it is used by mobile host to notify the server about the completion of handoff and the bandwidth of new wireless link. The unit of bandwidth in TCP Bandwidth Option is Kbps.

Within TCP-HO, when a mobile host gets to know the completion of handoff, it immediately sends out *two* duplicate ACKs with TCP Bandwidth Option which carries bw_{new} (the bandwidth of new wireless link). Two duplicate ACKs make the notification more robust in lossy networks. If more than two duplicate ACKs are sent out, they may waste bandwidth, and even may trigger TCP sender's fast retransmit and fast recovery [70].

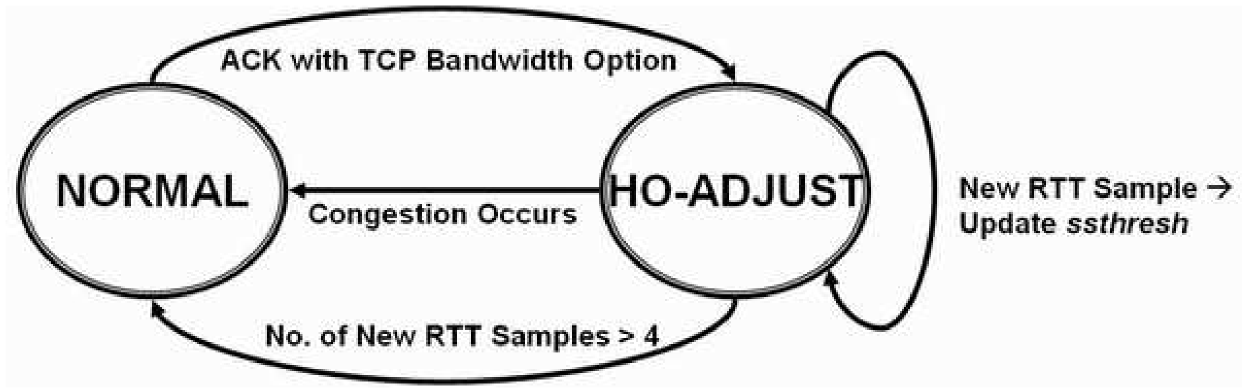


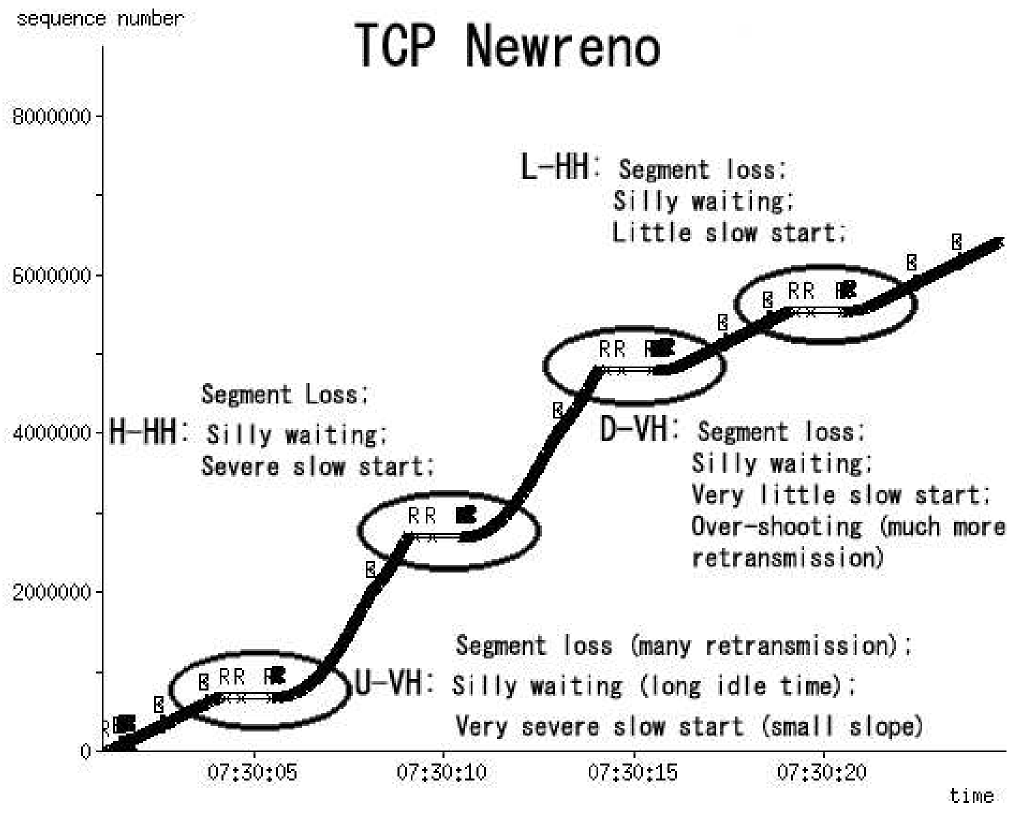
Figure 2.3: State Transition Diagram of TCP-HO Sender

When the server receives one ACK with TCP Bandwidth Option, it starts to transmit immediately, sets $ssthresh = bw_{new} * srtt_{old}$, and enters into HO-ADJUST state (see Figure 2.3). Here, the server only responds to TCP Bandwidth Options received just after a *timeout* event with the aim to avoid misbehaving users to exploit this option for acquiring higher throughput. Mobile host also has no motivation in cheating on bw_{new}. If mobile host sends out a smaller value, the sender needs more time to converge to the new wireless link's bandwidth and the throughput becomes lower. If mobile host sends out a much larger value, *timeout* tends to occur and the throughput may become even worse.

In HO-ADJUST state, for each new RTT sample, the server keeps updating *ssthresh* according to bw_{new} and $srtt_{new}$ (the smooth average of new RTT samples). If congestion is detected in HO-ADJUST state, the server returns to normal state and works as normal TCP. According to source codes of FreeBSD 5.4, four RTT samples can give a quite accurate estimation of average RTT. In addition, the bandwidth of wireless link may change with time and CPU is consumed in HO-ADJUST state for updating *ssthresh*. Hence, TCP-HO server also returns to normal state after four new RTT samples. Figure 2.3 shows the state transition diagram used by TCP-HO sender.

Figure 2.4 shows the time *vs.* sequence number graph of TCP Newreno [53] and TCP-HO during four kinds of handoff which occur in the same mobile scenario. For better comparison, TCP-HO is implemented in NS2 [3] and simulations are used to generated the two graphs. This figure shows that TCP Newreno does suffer the problems listed in Table 2.1. It also indicates that TCP-HO solves all problems except segment loss and achieves much higher throughput than TCP Newreno.

Within [127], a similar idea was presented and evaluated by simulation. But its network scenario is different. In particular, the authors assume that the mobile host is the data sender, who also knows everything about handoff.



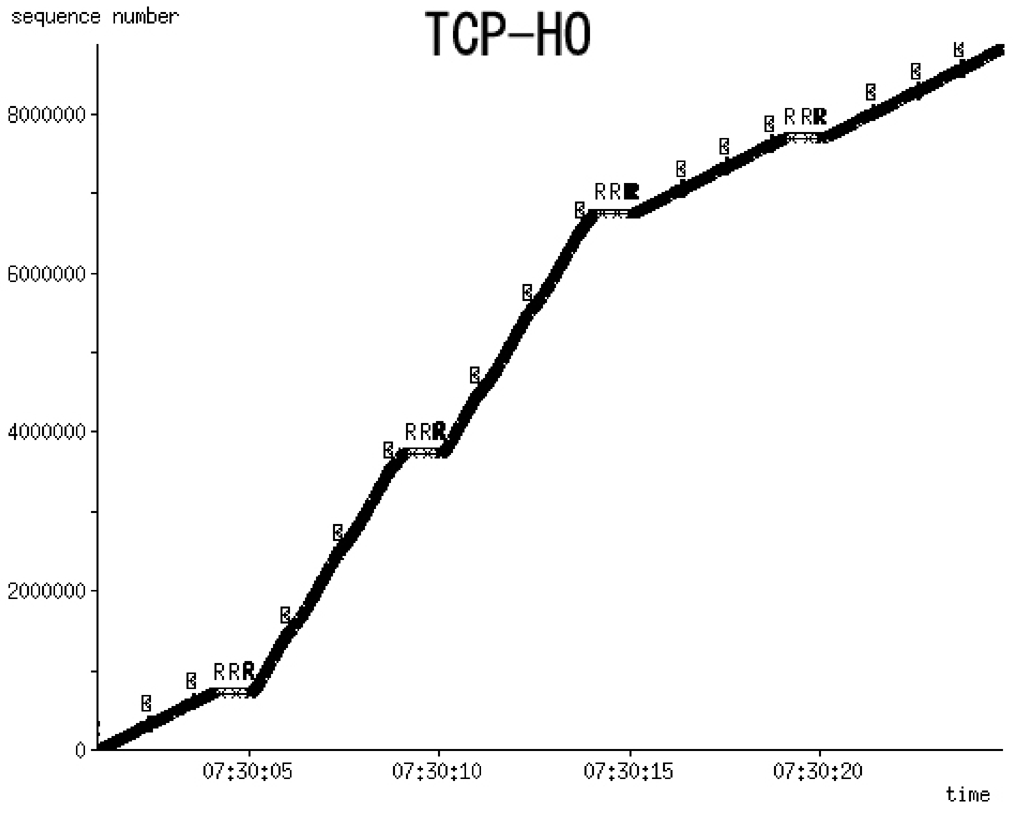


Figure 2.4: Time *vs.* Sequence Number Graph of TCP Newreno and TCP-HO under Four Kinds of Handoff Occurred in the Same Mobile Scenario

2.5 Performance Evaluation

With *immediate transmission* [37], TCP-HO can reduce the disconnection time of data flow. This metric is very important to interactive applications, such as Telnet and WWW. The improvement of TCP-HO can be analyzed as follows.

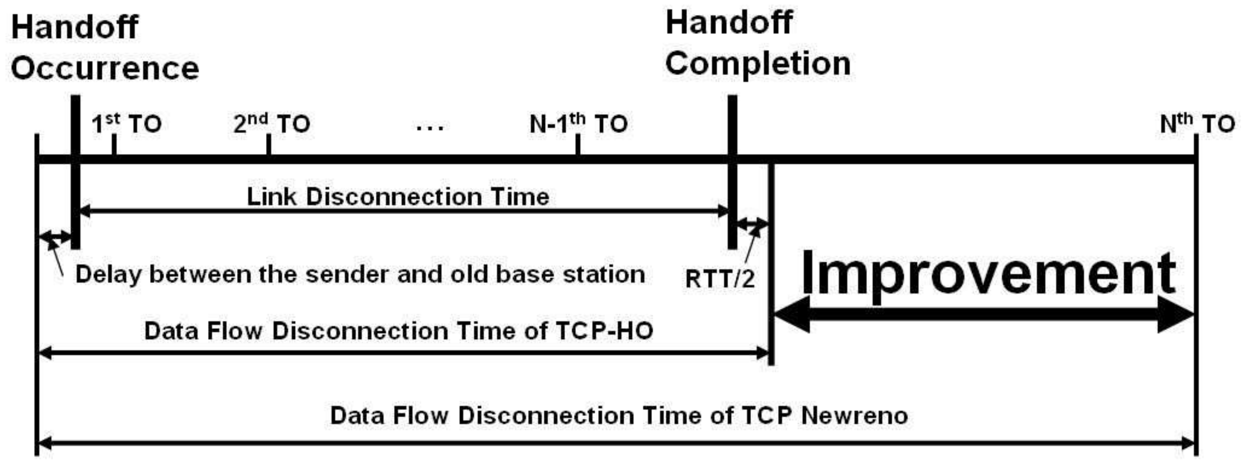

Handoff Occurrence
Handoff Completion
1st TO
2nd TO
...
N-1th TO
Nth TO
Link Disconnection Time
Delay between the sender and old base station
RTT/2
Improvement
Data Flow Disconnection Time of TCP-HO
Data Flow Disconnection Time of TCP Newreno

Figure 2.5: Improvement of TCP-HO on Data Flow Disconnection Time

Since TCP's retransmission timer adopts binary exponential backoff algorithm with a maximal *rto* value (64 seconds), when link disconnection time is so long that the *rto* reaches 64 seconds, the improvement is between 0 to 64 seconds. Otherwise, the improvement can be illustrated by Figure 2.5. If the time between handoff occurrence and the $N-1_{th}$ timeout is denoted as T, the time between handoff occurrence and the N_{th} timeout is about $2T$. We assume that handoff may be completed at any time between T and $2T$. Under this assumption, if link disconnection time during handoff is denoted as x, the improvement on data flow disconnection time can be approximately expressed as $\frac{2T-x}{x}$; and the average can be calculated by the following equation.

$$\begin{aligned}\frac{1}{T} * \int_{T}^{2T} \frac{2T-x}{x}dx &= \frac{[2T*(\ln(x)|_{T}^{2T}) - (x|_{T}^{2T})]}{T} = \frac{2T*[\ln(2T)-\ln(T)]-(2T-T)}{T} \\ &= \frac{2T*\ln(2)-T}{T} = 2\ln(2)-1 \approx 0.3863\end{aligned}$$

Hence the improvement on data flow disconnection time is about 0.3863 of the link disconnection time. Freeze TCP can achieve the same improvement in this metric.

This section focuses on evaluating and comparing these proposals in another important metric, the throughput of a long-lived TCP connection. For the purpose of performance evaluation, TCP-HO is implemented in FreeBSD 5.4, a controlled network environment is set up, and many experiments are carried out.

2.5.1 Testbed Setup

Figure 2.6 shows the topology of our network testbed. A Mobile Host is connected with Handoff Emulator by cross-cable and all other computers are connected by two virtual LANs of a 100Mbps 3COM Super Stack II Switch 3900.

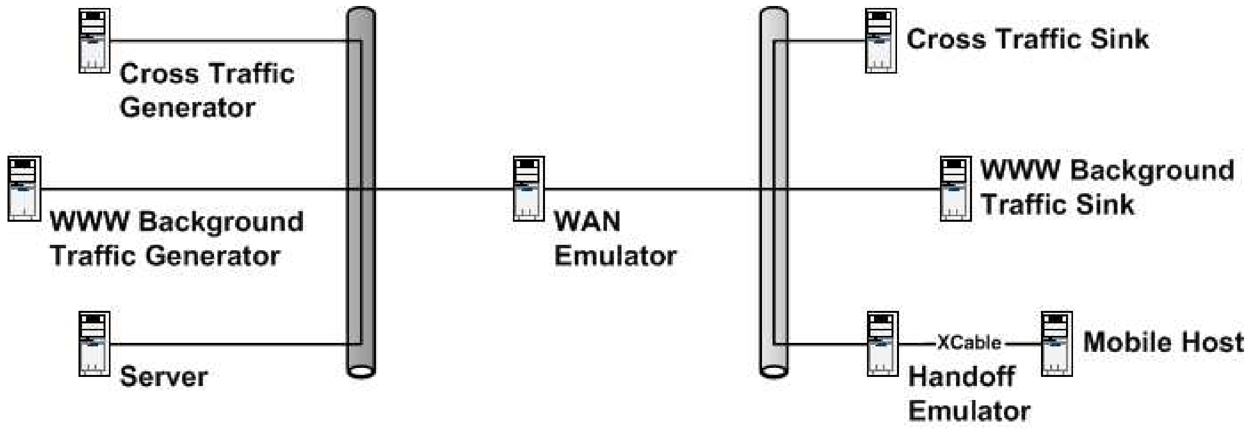


Figure 2.6: Testbed for TCP-HO Evaluation

FreeBSD 5.4 is installed on all of these computers. The flow between Server and Mobile Host is the flow to be investigated. A cross traffic flow is generated between Cross Traffic Generator and Cross Traffic Sink in order to investigate the friendliness of TCP-HO. Iperf [2] is used by Server and Cross Traffic Generator for generating long-lived data flows and measuring their throughput. WWW Background Traffic Generator and WWW Background Traffic Sink are responsible to generate some Internet-like background traffic. Apache is installed on WWW Background Traffic Generator and Surge [27] is used to emulate WWW traffics generated by ten users.

Dummynet [116] is used by WAN Emulator to emulate the delay and bandwidth of a wide area network. The bandwidth of WAN is set to 10Mbps. Considering the small number of connections in the following experiments, the queue at WAN Emulator is set to 20 packets. As for the delay of WAN, it is changed in different experiments in order to emulate that mobile host accesses servers that locate at different places.

In order to better emulate the bandwidth & delay of different wireless links, instead of Mobile Host, a separate computer (Handoff Emulator) is used and its kernel is re-built with finer timer resolution (1ms). Dummynet is used for emulating mobile scenarios and a special ICMP message is used by Handoff Emulator to notify Mobile Host about handoff completion and the bandwidth of new wireless link. This ICMP message emulates handoff completion detection and bandwidth estimation functions of Mobile Host.

2.5.2 Experimental Results

In order to evaluate TCP-HO, several mobile scenarios (handoff sequences), that may occur in different mobile environments of the real world (WLAN and WCDMA), are first generated. For each mobile scenario, the delay of WAN are set to different values (10ms, 20ms, 50ms, and 100ms) to create various network scenarios.

For each network scenario, the flow between Server and Mobile Host may use TCP Newreno, Freeze TCP, and TCP-HO. In order to analyze the effects of different parts of TCP-HO mechanism, two additional TCP variants, TCP-HO-Immediate and TCP-HO-Aggressive, are also investigated. Within TCP-HO-Immediate, the sender immediately begins to transmit after receiving notification. But the bandwidth of new wireless link is ignored. In TCP-HO-Aggressive, the sender sets both $cwnd$ and $ssthresh$ to $bw_{new} * srtt_{old}$. Hence, for each network scenario, five experiments will be carried out.

Within each experiment, the cross traffic flow uses TCP Newreno and runs simultaneously with the connection between Server and Mobile Host. WWW background traffics are also there. In order to get accurate results, each experiment is carried out for ten times. Their average and 95% confidence interval will be computed and presented. In the following parts, experiment results under three emulated mobile scenarios are presented and discussed.

WCDMA Scenario

WCDMA scenario emulates that a user drives a car (speed: 60km/h) in a rural area covered by WCDMA network (cell coverage: 2km) for half an hour. The average dwelling time is 50 seconds and the disconnection time is 3 seconds. Another assumption is that ten users share one 2Mbps WCDMA channel and the bandwidth of each user is about 200Kbps.

Figure 2.7 shows the throughput between Server and Mobile Host when different TCP mechanisms are used. This figure shows that Freeze TCP, TCP-HO and the variants of TCP-HO do improve TCP performance. Their throughput is 20%-60% higher than TCP Newreno. The difference among these mechanisms is very small. It is reasonable since L-HH dominates the WCDMA mobile scenario and *silly waiting* is the main problem. These results accord with Freeze-TCP's claim that *immediate transmission* dominates Freeze-TCP's performance enhancement. Figure 2.7 also shows that Freeze TCP performs better than TCP-HO sometimes. The reason is that, after the completion of handoff, the sending rate of Freeze TCP is higher than that of TCP-HO. As for TCP-HO-Aggressive, its throughput is lower than TCP-HO even though it is more aggressive than Freeze TCP. It may be too aggressive and hurts itself due to the large data burst after handoff completion. In addition, Figure 2.7 shows that the throughput does not respond to the change of WAN delay well. The reason is that RTT is dominated by the slow WCDMA link whose base station has a large buffer (20 packets).

Figure 2.8 shows the throughput of cross traffic flow under WCDMA mobile scenario. It indicates that under WCDMA scenario, all TCP mechanisms are friendly to cross traffic flow. Due to the low bandwidth of WCDMA link, the flow between Server and Mobile Host cannot significantly affect cross traffic flow, irrespective of TCP variant used by the flow.

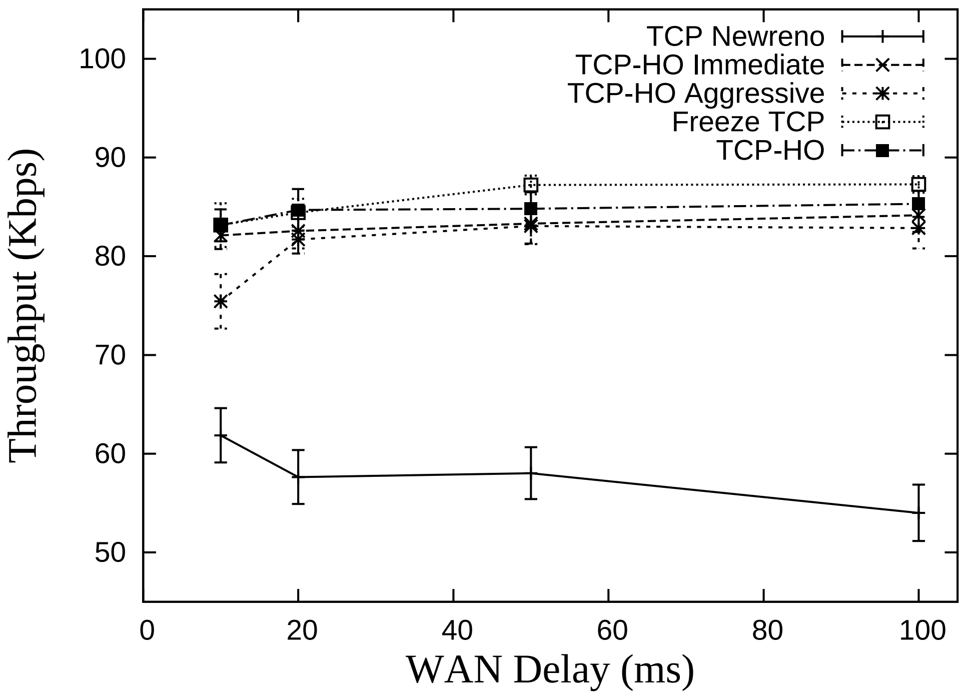


Figure 2.7: WCDMA Scenario: Average and 95% Confidence Interval of the Throughput Received by the Flow between Server and Mobile Host

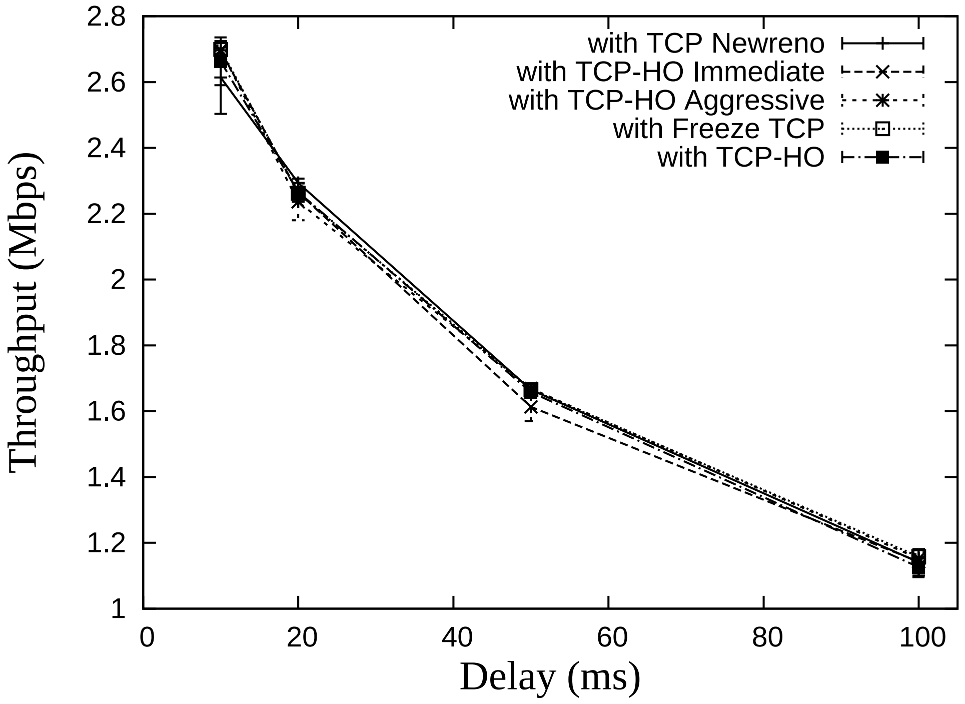


Figure 2.8: WCDMA Scenario: Average and 95% Confidence Interval of Cross Traffic Flow's Throughput

WLAN Scenario

WLAN scenario emulates that a user drives a car (speed: 30km/h) around a campus with some Wi-Fi hot-spots (cell coverage: 100m) for 15 minutes. The average dwelling time is 12 seconds and the disconnection time is 10 seconds (bad coverage). Per-cell user number is either large (9∼10) or very small (1∼2). These users share 7Mbps channel bandwidth (Effective Channel Bandwidth of IEEE 802.11b [10]). Available bandwidth for each user depends on the number of users per cell and the value can vary significantly .

Figure 2.9 shows the throughput between Server and Mobile Host in WLAN mobile scenario. It shows that TCP-HO achieves the highest throughput in all cases. Compared to TCP Newreno, the throughput is improved by almost 100%. It is reasonable since TCP-HO has been designed to solve the challenges brought by H-HH, U-VH and D-VH. Figure 2.9 also shows that TCP-HO-Immediate achieves trivial improvement and TCP-HO-Aggressive is even worse than Newreno. This is reasonable since H-HH, U-VH and D-VH dominate this mobile scenario. TCP-HO-Immediate does not solve severe "*slow start*" problem of H-HH and U-VH. In the case of TCP-HO-Aggressive, during each kind of handoff occurred in this high-speed wireless network, the large data burst (generated immediately after handoff completion) will cause multiple segment loss and trigger *timeout*. Consequently, the throughput of TCP-HO-Aggressive becomes very low.

Figure 2.9 also shows that Freeze TCP can acquire quite high throughput too. But Freeze TCP is too aggressive under WLAN mobile scenario. Figure 2.10 shows the throughput of cross traffic flow under WLAN mobile scenario. It indicates that under WLAN scenario, Freeze-TCP obviously hurts cross traffic flow in all cases. The reason may be that after handoff completion, Freeze TCP is too aggressive and the segments of cross traffic flow are dropped due to the large data burst of Freeze TCP. Furthermore, with the handoff occurrence prediction failure in the real world, Freeze-TCP's performance tends to be worse. As for TCP-HO-Aggressive, it may be too aggressive, harm itself too much, and cross traffic flow can acquire more bandwidth when the flow driven by TCP-HO-Aggressive is idle.

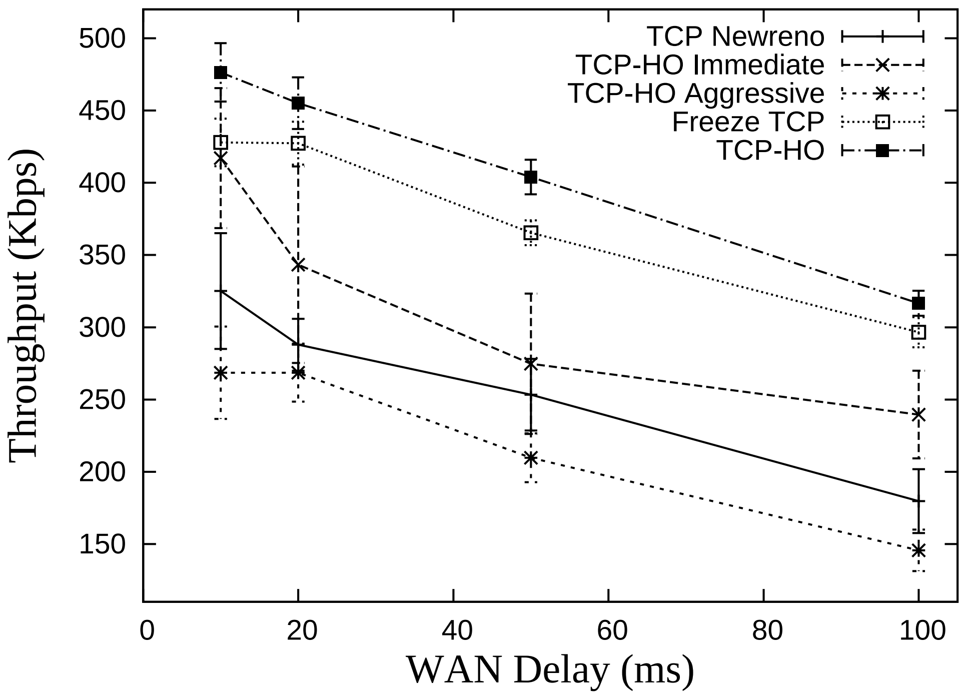


Figure 2.9: WLAN Scenario: Average and 95% Confidence Interval of the Throughput Received by the Flow between Server and Mobile Host

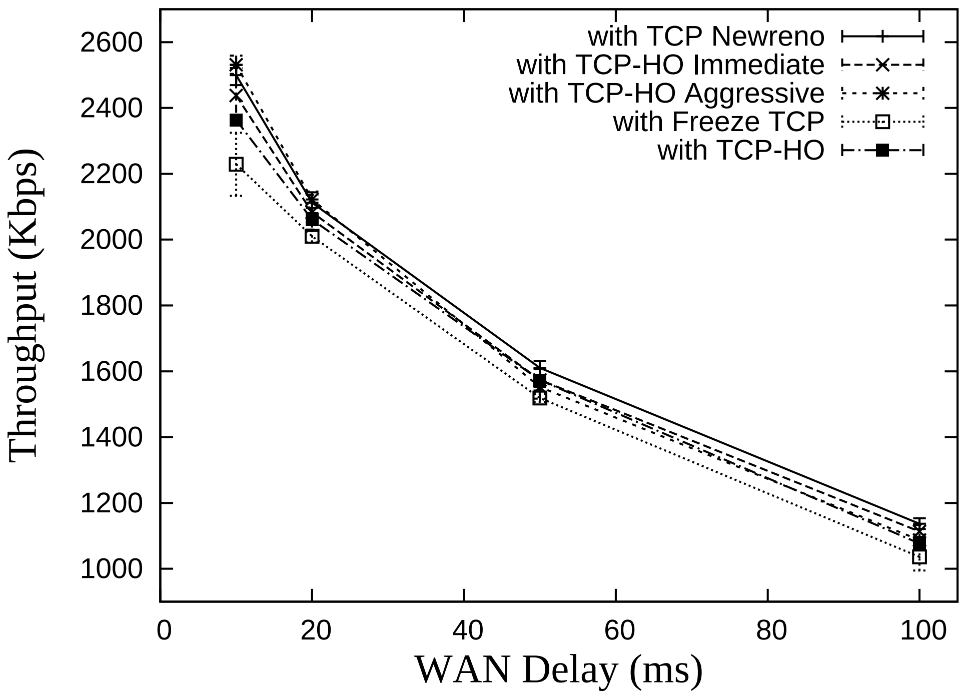


Figure 2.10: WLAN Scenario: Average and 95% Confidence Interval of Cross Traffic Flow's Throughput

WCDMA&WLAN Scenario

This scenario emulates that a user drives a car (speed: 45km/h) around a city with WCDMA coverage and sporadic Wi-Fi hot-spots for 20 minutes. If Wi-Fi hot-spot exists, the user always switches to Wi-Fi. Otherwise, WCDMA is used. The probability of handoff between two networks and the disconnection time follow the numbers in Table 2.2. These numbers are calculated based on the assumption that two Wi-Fi hot-spots exist in one WCDMA cell.

Handoff Probability / Disconnection Time	*WCDMA*	*WLAN*
WCDMA	0.33 / 3s	0.67 / 5s
WLAN	0.9 / 5s	0.1 / 10s

Table 2.2: Handoff Probability and Disconnection Time

The average dwelling time of WCDMA cell is 20 seconds and per-user bandwidth is 200Kbps. The average dwelling time of Wi-Fi is 8 seconds and per-cell user number follows the same distribution used by WLAN scenario.

Figure 2.11 shows the throughput between Server and Mobile Host in WCDMA&WLAN mobile scenario. It indicates that TCP-HO can achieve the highest throughput. Its throughput is about 70% higher than TCP Newreno. Freeze TCP and TCP-HO-Immediate are also quite good. TCP-HO-aggressive is only a little better than Newreno. Figure 2.11 looks like a mixture of the results of WCDMA scenario and WLAN scenario. It is reasonable since L-HH, H-HH, U-VH, and D-VH all occur in this scenario.

Figure 2.11 also indicates that TCP-HO does help WCDMA users to utilize Wi-Fi hot-spots. If Newreno is used, a user who uses WLAN and WCDMA interfaces alternately cannot acquire much higher throughput than a user who only uses the WCDMA interface.

Figure 2.12 shows the throughput of cross traffic flow under WCDMA&WLAN mobile scenario. It indicates that Freeze TCP is still too aggressive and harms cross traffic flow in some cases.

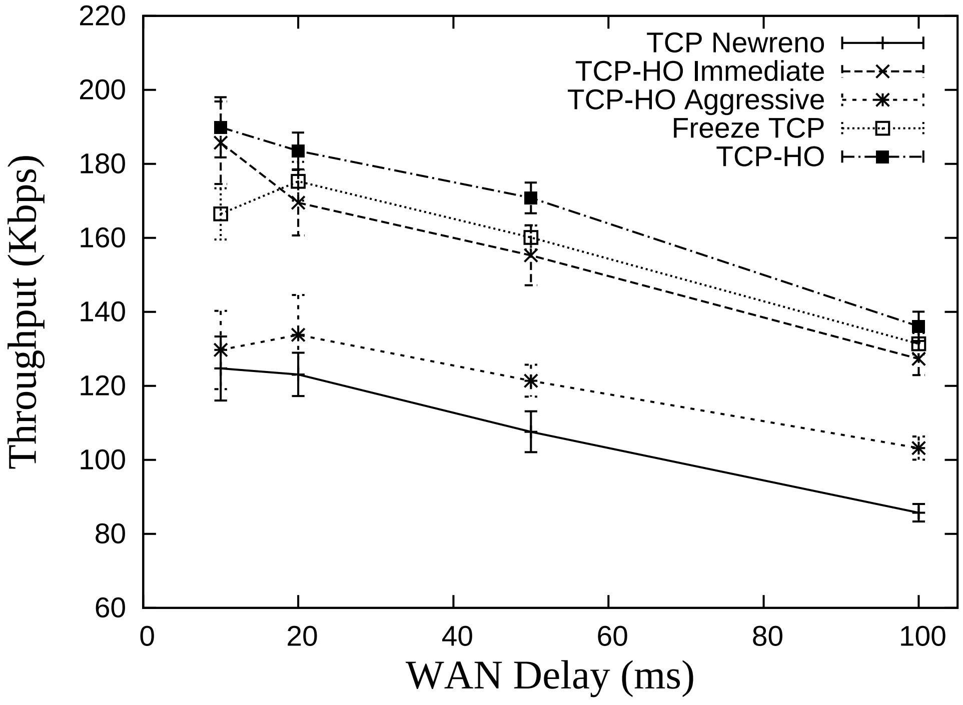


Figure 2.11: WCDMA&WLAN Scenario: Average and 95% Confidence Interval of the Throughput Received by the Flow between Server and Mobile Host

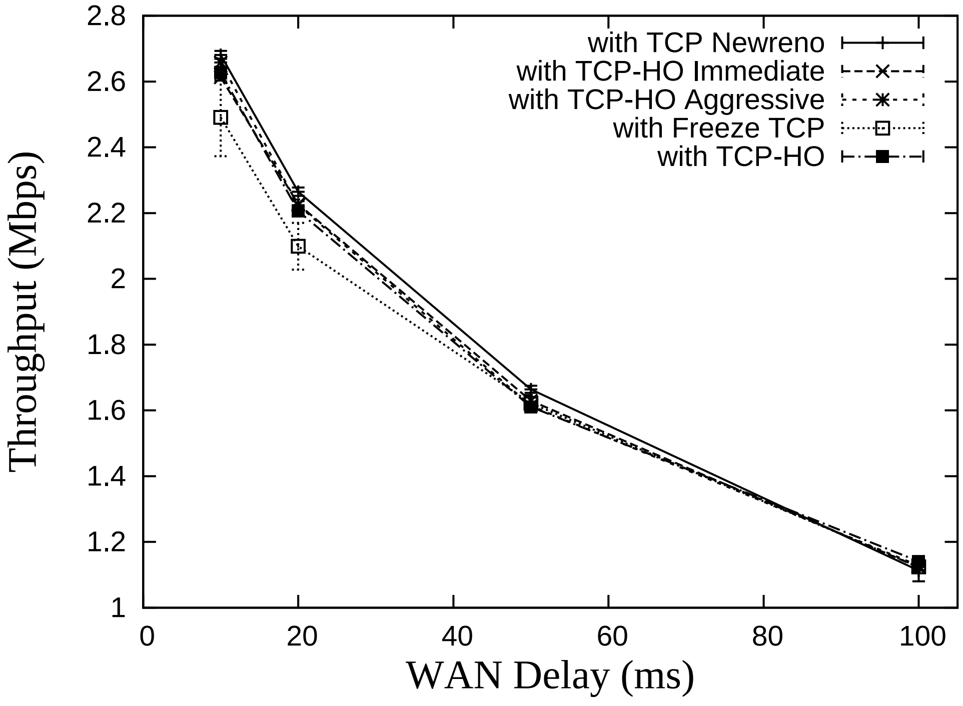


Figure 2.12: WCDMA&WLAN Scenario: Average and 95% Confidence Interval of Cross Traffic Flow's Throughput

According to the above results, TCP-HO is the best mechanism. In most of the above experiments, it can improve mobile host's throughput without adversely affecting cross traffic flow. Although Freeze TCP can improve mobile host's throughput quite well, it is too aggressive and even one Freeze-TCP flow can obviously harm cross traffic flow in some cases, especially under WLAN mobile scenario. Considering that more and more users will access the Internet through wireless network and the bandwidth of wireless network is increasing quickly [9][12], Freeze TCP is not a safe solution again. Under the handoff occurrence prediction failure in the real world, Freeze-TCP's performance tends to be even worse. With the considerations that more and more users are accessing the Internet through wireless networks and wireless link is normally the last link with the smallest bandwidth, it should be worthwhile to deploy TCP-HO by changing both TCP end-points.

2.6 TCP-HO and Wireless Link Bandwidth Estimation Mechanisms

Bandwidth estimation error can be regarded as the reason that TCP suffers *slow start* and *over-shooting* during handoff. The server regards the bandwidth probed on old wireless link as the bandwidth of the new wireless link. During handoff, especially during vertical handoff, compared with the server, mobile host normally has better opportunity to estimate the new wireless link's bandwidth timely and accurately. This fact is the main motivation of TCP-HO. In this section, we will discuss wireless bandwidth estimation at mobile host, analyze the effects of its bandwidth estimation error on TCP-HO, and present TCP-HO performance under the achievable bandwidth estimation accuracy.

2.6.1 Wireless Link Bandwidth Estimation Mechanisms

The estimation of a wireless link's bandwidth has been used by base station selection, routing protocol, and other cross-layer optimizations [89][90][128], and a lot of mechanisms have been proposed to estimate the bandwidth of a wireless link [47][88][89][90][128][135][139][147]. These mechanisms estimate the bandwidth based on the model of layer 2 protocol with the parameters of signal strength and contention state that are acquired through passively monitoring the wireless link or actively transmitting several probing frames.

In the real world, the type of wireless link can be learned based on which network interface card (NIC) is used. Hence, layer 2 protocol model can be determined. Signal strength and data rate used for coding have also been provided by tools from NIC manufacturers. If the wireless link is a dedicated channel or time slot, its available bandwidth can be estimated without any further support. If the wireless link is shared, such as IEEE 802.11-based WLAN, contention state could be acquired by modifying the firmware and drivers of NIC, and this method had been verified in [135]. Hence, these bandwidth estimation mechanisms should be implementable in the real systems. Existing simulation and field trial results show that mobile host with these mechanisms can get an accurate estimation of the bandwidth in a short time. It is reported that, even in the complex IEEE 802.11 network, bandwidth estimation relative error can be less than 10% [135].

In the case of TCP-HO, before sending out the first TCP segment through the new wireless link, mobile host need associate to the new wireless network, acquire IP address from it, and complete layer 3 handoff [36]. Quite a few packets will be exchanged in these procedures, and these procedures can take several seconds in some networks [35]. During this period, mobile host with these mechanisms could learn wireless link's quality and its contention state. Hence, a quite accurate bandwidth estimation is feasible for mobile host when TCP Bandwidth Option needs to be sent out.

2.6.2 Effects of Mobile Host's Bandwidth Estimation Error

In this part, TCP-HO performance under wireless link bandwidth estimation error is numerically studied. In the following analysis, $\hat{b}$ represents the real BDP after handoff, $\hat{x}$ represents the product of bandwidth estimated by mobile host and RTT estimated by the server, and

$\hat{c}$ is the *ssthresh* maintained by the server before handoff.

When b is small, bandwidth estimation accuracy has a much less significant effect on the performance of TCP-HO. Firstly, when $\hat{x}$ is smaller than the small $\hat{b}$, since TCP-HO still increases *cwnd* after it has been exponentially increased to $\hat{x}$, TCP-HO can efficiently utilize the new wireless link very soon. Secondly, even when $\hat{x}$ is larger than the small $\hat{b}$, queue at the new base station can accommodate the extra segments. Hence, in this analysis, only U-VH and H-HH, during which $\hat{b}$ is quite large, are considered.

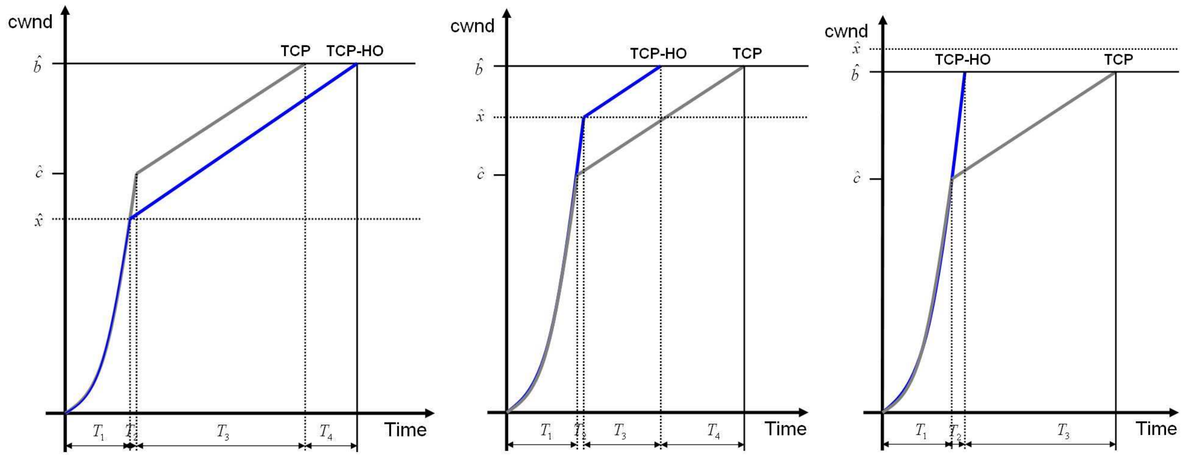
For simplicity, it is reasonable to assume that $\hat{x}$ follows a normal distribution (mean: $\hat{b}$, variance: σ^2) and σ will vary when analyzing TCP-HO sensitivity to bandwidth estimation error. $\hat{x}$ is also limited to be within $(0, 2\hat{b}]$. This constraint is made so that when $\hat{b} < \hat{x} < 2\hat{b}$, queue at the new base station can accommodate these extra segments. Hence, we can avoid to analyze the effects of segment loss. Considering the achievable bandwidth estimation accuracy, this range should be large enough.

With the above assumptions, TCP-HO performance in two cases, $\hat{b} > \hat{c}$ and $\hat{b} < \hat{c}$, are first modeled. Its sensitivity to bandwidth estimation error is also analyzed by fixing $\hat{b}$ and $\hat{c}$ according to several typical handoff scenarios and varying the value of σ.

The difference between N_{tcpho} and N_{tcp} are used for comparing the performance of TCP-HO and TCP and analyzing TCP-HO sensitivity to bandwidth estimation error. Here, N_{tcpho} and N_{tcp} are the number of segments, that are transmitted by TCP-HO and TCP before the new wireless link can be well utilized by both protocols. Since the purpose is to analyze the effects of bandwidth estimation error, the effects of TCP-HO's *immediate transmission* is ignored and it is assumed that TCP-HO and TCP begin to transmit at the same time. This will result similar performance for TCP-HO and TCP when the difference between $\hat{b}$ and $\hat{c}$ is small. Nevertheless, the analysis will show that TCP-HO performance will not degrade much even when there is substantial bandwidth estimation error.

Model for $\hat{b} > \hat{c}$

According to the approximate *cwnd vs.* time graphes shown in figure 2.13, without considering Delayed ACK [44], N_{tcp} and N_{tcpho} can be deduced in the following three scenarios.



(a) $\hat{x} < \hat{c} < \hat{b}$ (b) $\hat{c} < \hat{x} < \hat{b}$ (c) $\hat{c} < \hat{b} < \hat{x}$

Figure 2.13: *cwnd vs.* Time Graphs of TCP and TCP-HO When $\hat{b}$ is Larger Than $\hat{c}$

1. $\hat{x} < \hat{c} < \hat{b}$: As shown in figure 2.13(a), in the unit of RTT, $T_1 = log_2\hat{x}$, $T_2 = log_2\hat{c} - log_2\hat{x}$, $T_3 = \hat{b} - \hat{c}$, and $T_4 = (\hat{b} - \hat{x}) - T_2 - T_3 = \hat{c} - \hat{x} + log_2\hat{x} - log_2\hat{c}$. Hence,

$$N_{tcp} = A_{T_1+T_2} + A_{T_3} + A_{T_4} = \hat{c} + \frac{(\hat{b}^2 - \hat{c}^2)}{2} + \hat{b}(\hat{c} - \hat{x} + log_2\hat{x} - log_2\hat{c})$$

$$N_{tcpho} = A_{T_1} + A_{T_2+T_3+T_4} = \hat{x} + \frac{(\hat{b}^2 - \hat{x}^2)}{2}$$

$$\triangle_1(\hat{x}) = N_{tcpho} - N_{tcp} = -\frac{\hat{x}^2}{2} + (\hat{b}+1)\hat{x} - \hat{b}log_2\hat{x} + \frac{\hat{c}^2}{2} - (\hat{b}+1)\hat{c} + \hat{b}log_2\hat{c}$$

2. $\hat{c} < \hat{x} < \hat{b}$: As shown in figure 2.13(b), $T_1 = log_2\hat{c}$, $T_2 = log_2\hat{x} - log_2\hat{c}$, $T_3 = \hat{b} - \hat{x}$, and $T_4 = (\hat{b} - \hat{c}) - T_2 - T_3 = \hat{x} - \hat{c} + log_2\hat{c} - log_2\hat{x}$. Hence,

$$N_{tcp} = A_{T_1} + A_{T_2+T_3+T_4} = \hat{c} + \frac{(\hat{b}^2 - \hat{c}^2)}{2}$$

$$N_{tcpho} = A_{T_1+T_2} + A_{T_3} + A_{T_4} = \hat{x} + \frac{(\hat{b}^2 - \hat{x}^2)}{2} + \hat{b}(\hat{x} - \hat{c} + log_2\hat{c} - log_2\hat{x})$$

$$\triangle_2\ (\hat{x}) = N_{tcpho} - N_{tcp} = -\frac{\hat{x}^2}{2} + (\hat{b}+1)\hat{x} - \hat{b}log_2\hat{x} + \frac{\hat{c}^2}{2} - (\hat{b}+1)\hat{c} + \hat{b}log_2\hat{c}$$

3. $\hat{c} < \hat{b} < \hat{x}$: As shown in figure 2.13(c), $T_1 = log_2\hat{c}$, $T_2 = log_2\hat{b} - log_2\hat{c}$, and $T_3 = (\hat{b} - \hat{c}) - T_2 = \hat{b} - \hat{c} + log_2\hat{c} - log_2\hat{b}$. Hence,

$$N_{tcp} = A_{T_1} + A_{T_2+T_3} = \hat{c} + \frac{(\hat{b}^2 - \hat{c}^2)}{2}$$

$$N_{tcpho} = A_{T_1+T_2} + A_{T_3} = \hat{b} + \hat{b}(\hat{b} - \hat{c} + log_2\hat{c} - log_2\hat{b})$$

$$\triangle_3\ (\hat{x}) = N_{tcpho} - N_{tcp} = \frac{\hat{b}^2 + \hat{c}^2}{2} + \hat{b} - \hat{c} - \hat{b}\hat{c} + \hat{b}log_2\hat{c} - \hat{b}log_2\hat{b}$$

Finally, the expected performance improvement of TCP-HO can be calculated by the following equation. Here, $p(\hat{x}) = \frac{1}{\sigma\sqrt{2\pi}}exp(-\frac{(\hat{x}-\hat{b})^2}{2\sigma^2})$.

$$\triangle\ (\hat{b}, \hat{c}, \sigma) = \frac{\int_0^{\hat{c}} \triangle_1\ (\hat{x})p(\hat{x}) + \int_{\hat{c}}^{\hat{b}} \triangle_2\ (\hat{x})p(\hat{x}) + \int_{\hat{b}}^{2\hat{b}} \triangle_3\ (\hat{x})p(\hat{x})}{\int_0^{2\hat{b}} p(\hat{x})} \tag{2.1}$$

Model for $\hat{b} < \hat{c}$

Since only U-VH and H-HH are considered, for simplicity, it is assumed that $\hat{c} < 2\hat{b}$. Hence, we can assume that TCP will not cause segment loss. Consequently, the analysis can be simplified and the analyzed TCP-HO performance improvement is lower than its improvement in the real world. Following the methods used in the above case, the following results are deduced.

1. $\hat{x} < \hat{b} < \hat{c}$: As shown in figure 2.14(a), in the unit of RTT, $T_1 = log_2\hat{x}$, $T_2 = log_2\hat{b} - log_2\hat{x}$, and $T_3 = (\hat{b} - \hat{x}) - T_2 = \hat{b} - \hat{x} + log_2\hat{x} - log_2\hat{b}$. Hence,

$$N_{tcp} = A_{T_1+T_2} + A_{T_3} = \hat{b} + \hat{b}(\hat{b} - \hat{x} + log_2\hat{x} - log_2\hat{b})$$

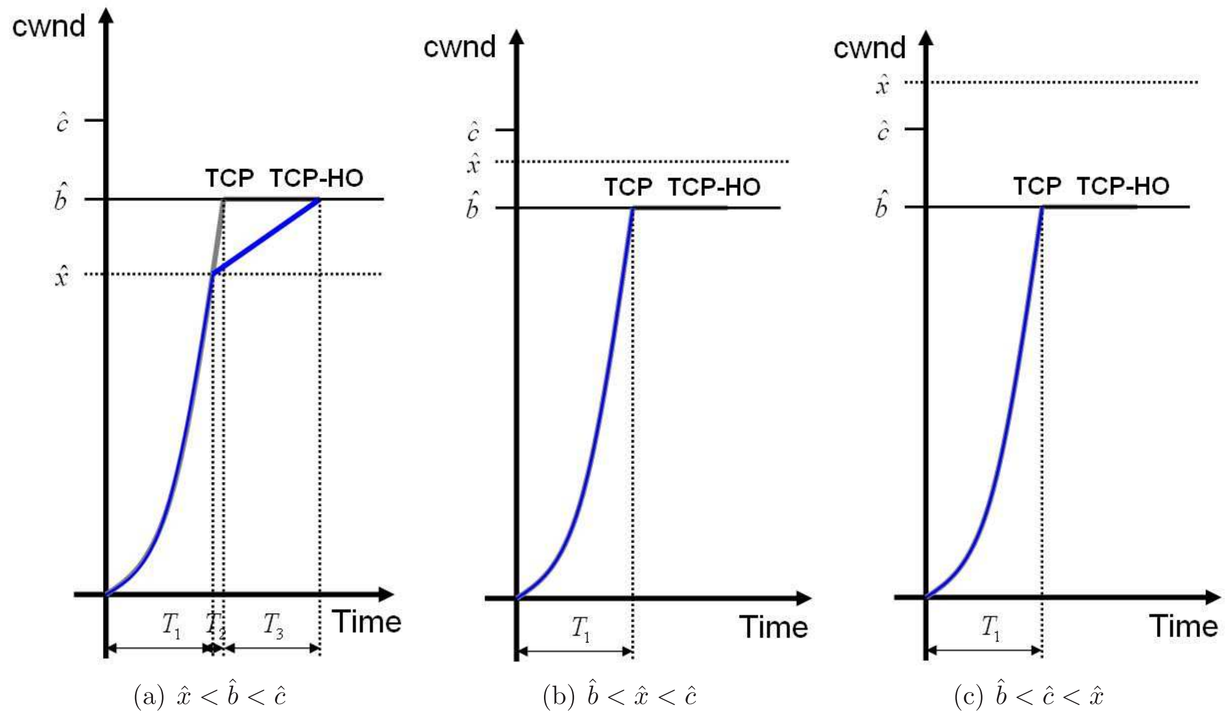


Figure 2.14: *cwnd vs.* Time Graphs of TCP and TCP-HO When $\hat{b}$ is Less Than $\hat{c}$

$$N_{tcpho} = A_{T_1} + A_{T_2+T_3} = \hat{x} + \frac{(\hat{b}^2 - \hat{x}^2)}{2}$$

$$\nabla_1(\hat{x}) = N_{tcpho} - N_{tcp} = -\frac{\hat{x}^2}{2} + (\hat{b}+1)\hat{x} - \hat{b}log_2\hat{x} - \frac{\hat{b}^2}{2} - \hat{b} + \hat{b}log_2\hat{b}$$

2. $\hat{b} < \hat{x} < \hat{c}$: As shown in figure 2.14(b), the curves of TCP and TCP-HO are totally overlapped. Hence, $\nabla_2(\hat{x}) = 0$.

3. $\hat{b} < \hat{c} < \hat{x}$: As shown in figure 2.14(c), $\nabla_3(\hat{x})$ also equals to 0.

Finally, the expected performance improvement of TCP-HO can be calculated by the following equation. Here, $p(\hat{x}) = \frac{1}{\sigma\sqrt{2\pi}}exp(-\frac{(\hat{x}-\hat{b})^2}{2\sigma^2})$.

$$\nabla(\hat{b}, \hat{c}, \sigma) = \frac{\int_0^{\hat{b}} \nabla_1(\hat{x})p(\hat{x})}{\int_0^{2\hat{b}} p(\hat{x})} \tag{2.2}$$

Bandwidth Estimation Error Sensitivity Analysis

In this part, bandwidth estimation error sensitivity analysis is carried out by fixing $\hat{b}$ and $\hat{c}$ according to several typical handoff scenarios and varying the value of σ from 5% to 50% of b. The following four handoff scenarios are used.

1. U-VH Scenario($\hat{b} = 117, \hat{c} = 8$): In this scenario, the effects of bandwidth estimation error are investigated when mobile host switches from a WCDMA link (bandwidth=200Kbps, RTT=500ms, and $\hat{c} \approx 8$ segments) to a WLAN channel (bandwidth=7Mbps, RTT=200ms, and $\hat{b} \approx 117$ segments). $\hat{b}$ and $\hat{c}$ in equation 2.1 are substituted with these values and $\triangle$ is calculated with different σ through MATLAB.

2. U-VH Scenario($\hat{b} = 58, \hat{c} = 8$): This is another U-VH scenario whose BDP difference is smaller than the above one.

3. H-HH Scenario($\hat{b} = 58, \hat{c} = 50$): In this scenario, the effects of bandwidth estimation error are investigated when mobile host switches between two different WLAN channels and the new channel has a little more available bandwidth. And the same method used by U-VH scenarios is adopted.

4. H-HH Scenario($\hat{b} = 58, \hat{c} = 66$): In this scenario, the effects of bandwidth estimation error are investigated when mobile host switches between two different WLAN channels and the new channel has a little less available bandwidth. $\hat{b}$ and $\hat{c}$ in equation 2.2 are substituted with these values and ∇ is calculated with different values of σ.

Figure 2.15 shows these results of sensitivity analysis. It indicates that TCP-HO performance during U-VH and H-HH is not sensitive to the accuracy of bandwidth estimation. During U-VH, due to the large difference in BDP, TCP-HO still can improve TCP performance a lot even when mobile host only has a coarse bandwidth estimation. During H-HH, the performance of TCP-HO is just a little worse than TCP when mobile host has a significant bandwidth estimation error.

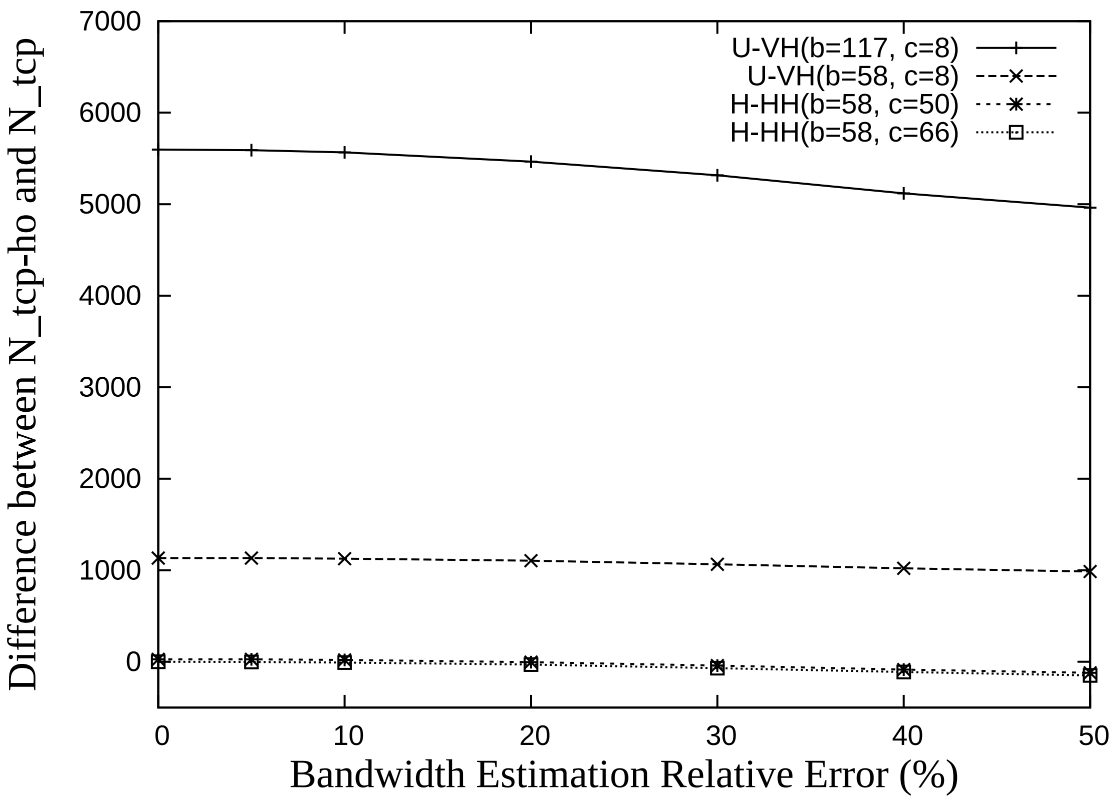
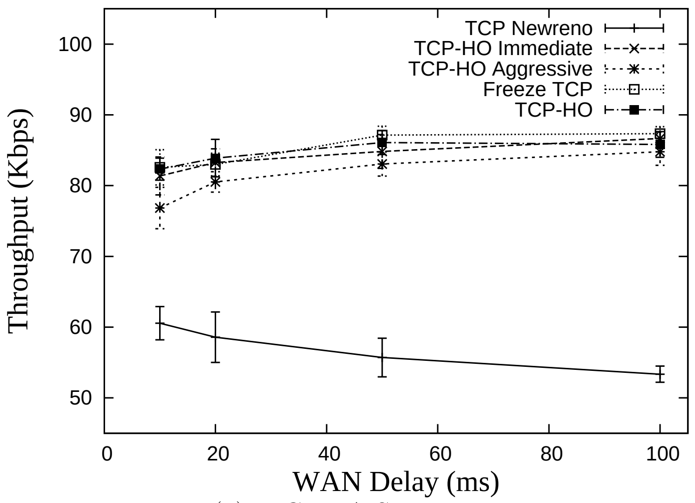


Figure 2.15: Bandwidth Estimation Error Sensitivity Analysis

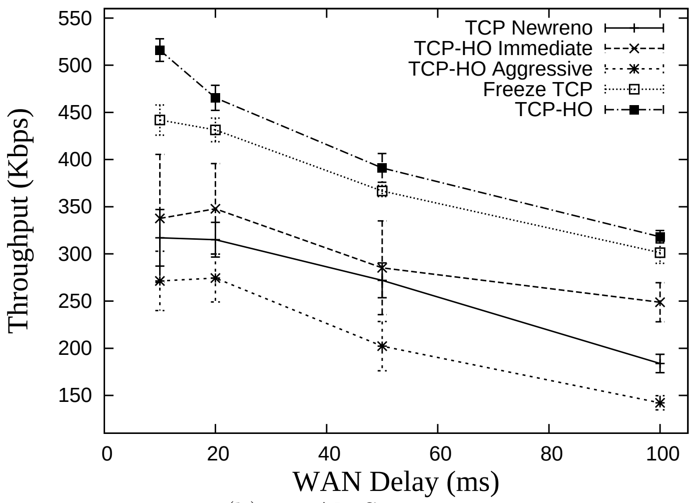
2.6.3 TCP-HO Performance under Achievable Bandwidth Estimation Accuracy

In the experiments of section 2.5, we assume that mobile host can estimate the bandwidth of new wireless link very accurately. In this subsection, TCP-HO is evaluated when mobile host only has an achievable accurate estimation of wireless link's bandwidth with the aim to evaluate the effects of bandwidth estimation error.

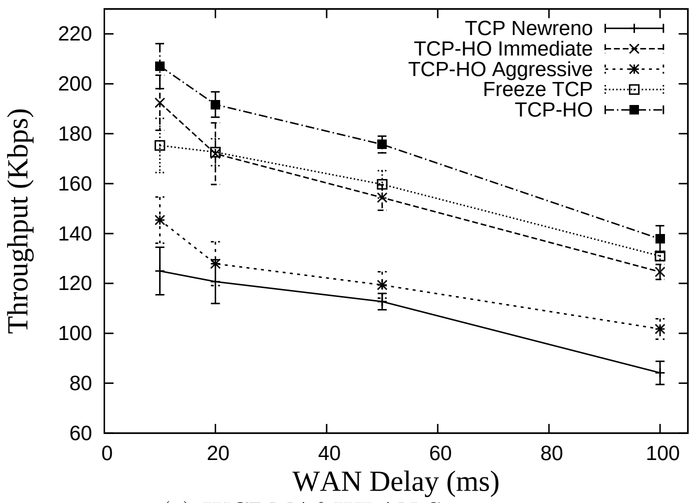
WCDMA, WLAN, and WCDMA&WLAN mobile scenarios are re-generated, and the seeds of random number generator are different with subsection 2.5.2. The bandwidth sent to Mobile Host is different with the bandwidth emulated by Handoff Emulator, and the simulated bandwidth estimation error is within 15% of the new wireless link's emulated bandwidth. The experiments of subsection 2.5.2 are re-run with these new mobile scenarios, and figures 2.16 and 2.17 show the new experiment results. These results are similar to the results in subsection 2.5.2. Compared to TCP Newreno, TCP-HO can improve the throughput of mobile host by 30%-100% and cross traffic is not adversely affected. They indicate that TCP-HO still can improve TCP performance substantially without adversely affecting cross traffic, even when mobile host only has a coarse estimation of wireless link's bandwidth.



(a) WCDMA Scenario

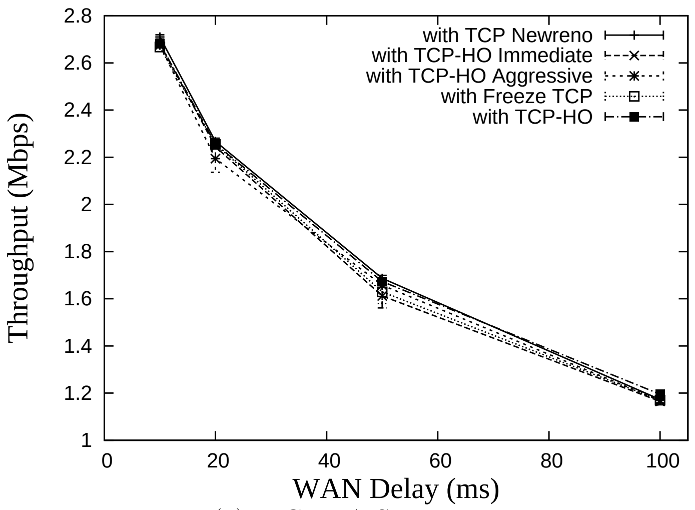


(b) WLAN Scenario

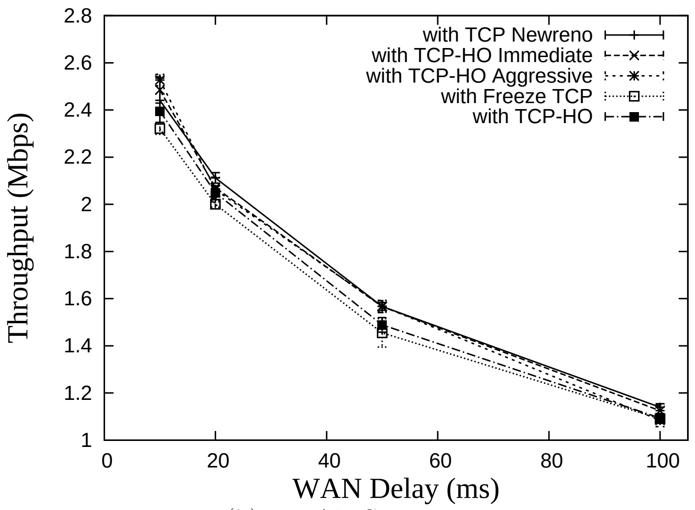


(c) WCDMA&WLAN Scenario

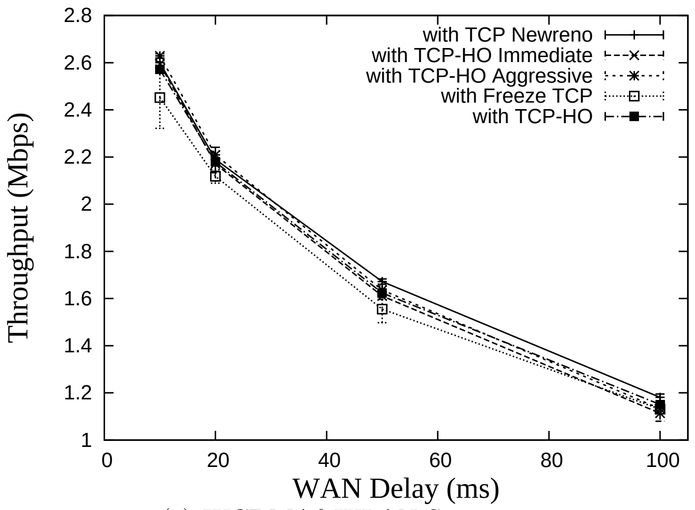
Figure 2.16: Average Throughput Received by the Flow between Server and Mobile Host with 15% Bandwidth Estimation Error



(a) WCDMA Scenario



(b) WLAN Scenario



(c) WCDMA&WLAN Scenario

Figure 2.17: Average Throughput of Cross Traffic Flow with 15% Bandwidth Estimation Error

2.7 Summary

In this chapter, TCP-HO, the first practical end-to-end TCP enhancement based on explicit cooperation, is proposed to solve the challenges brought by all kinds of handoff that may occur in heterogeneous mobile environments. Experimental results indicate that in heterogeneous mobile environments, TCP-HO improves TCP performance substantially without adversely affecting cross traffic. Considering that more and more users are accessing the Internet through heterogeneous wireless networks and mobile hosts could have a coarse estimation of wireless link's bandwidth, it should be worthwhile to implement TCP-HO at both server and mobile host for improving TCP performance.

Chapter 3

Sync-TCP: A New Approach to High Speed Congestion Control

3.1 Introduction

In recent years, bandwidth of the Internet continues to increase quickly. For example, design capacity of the forthcoming Trans-Pacific Express [4] is 5.12 Tera-bps and FTTx (Fiber To The Home, Building, Neighborhood, etc.) has been widely deployed in many countries. Recent bandwidth measurement statistics (http://www.speedtest.net/ at 2010-03-12) show that the average download speeds vary from 7.73Mbps in Europe to 1.39Mbps in Africa. The average download speed for the top 6 countries already exceeds 15Mbps. In some countries, for example, Singapore, the goal is to provide up to 1Gbps broadband access by 2015. Considering that networks are and likely to remain lightly-loaded [107], in the future high speed Internet, there will be more and more long fat network pipes with abundant residual bandwidth, which is very attractive to bandwidth-greedy and elastic applications, such as video and software distribution, data backup, peer-to-peer file sharing, etc.

However, TCP, the de-facto standard transport protocol of the Internet, adopts a window and loss based congestion control algorithm [70], and it is well known that standard

TCP congestion control variants (TCP Reno [17], TCP Newreno [53], TCP SACK [100], etc.) cannot work well on long fat network pipes whose bandwidth-delay product (BDP) is large [51][79]. When segment loss is detected mainly through 3DUPACK, these legacy TCP versions can be characterized as AIMD(1, 0.5) [146]. Due to this simple algorithm with fixed parameters, these legacy TCP versions cannot send data fast enough to utilize long fat network pipes efficiently. It may be better to explain this issue with the network performance model shown in figure 3.1, in which network load is the data that all senders pump into the network. The load, that legacy TCP flows can generate, is too low (far left away from the knee) to efficiently utilize the bandwidth of a long fat network pipe.

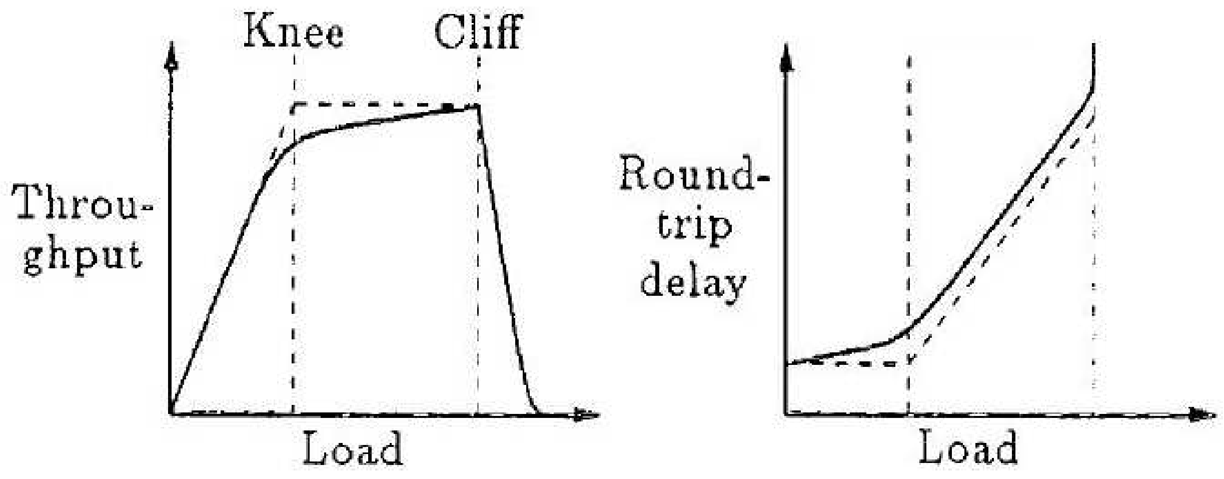


Figure 3.1: Network Performance Model as a Function of Network Load (from R. Jain)

In order to address this problem, many high speed congestion control (HSCC) algorithms, such as Highspeed TCP [51], Cubic-TCP [59], H-TCP [91], Fast TCP [75], TCP Illinois [94], Compound TCP [125], Yeah-TCP [20], TCP Fusion [77], and Delay-based AIMD [48], have been proposed in recent years. Some of these proposals have also been implemented in popular operating systems. For example, Compound TCP has been implemented in Windows, and Linux has selected Cubic-TCP as its default congestion control mechanism. For bandwidth-greedy and elastic applications, it is now easy and irresistible to adopt a HSCC algorithm, which can efficiently utilize the abundant residual bandwidth and provide higher throughput to end users on long fat network pipes.

On the other hand, bandwidth-greedy and elastic applications are not the only applications running in the Internet. Considering that a HSCC algorithm probes network resources more aggressively for higher throughput, it should pay more attention on friendliness to cross traffic, especially applications using legacy TCP versions and the interactive ones, such as web surfing and media-streaming that are more important to users' daily life and are more profitable to network providers. More specifically, existence of HSCC-based bandwidth-greedy and elastic applications should not significantly increase packet loss rate, queue delay, and jitter experienced by these cross traffic applications. As shown in figure 3.1, it is desirable if a HSCC algorithm could drive the network to operate around the knee, at which network throughput is high, queue delay is short, and packet drop rate is minimum. Such a HSCC algorithm could enable bandwidth-greedy and elastic applications to utilize long fat network pipes of the Internet efficiently without hurting the other applications.

But most of the existing HSCC algorithms are not designed to drive the network to operate around the knee [137][141], and a new HSCC algorithm is needed for safely ramping up the throughput of bandwidth-greedy and elastic applications on long fat network pipes of the Internet. The open problem is how to drive the network to operate around the knee and to distribute bandwidth fairly even when the number of competing flows varies and their round trip propagation delays (RTPD) differ significantly.

As pointed out in [73], an end-to-end delay-based congestion control algorithm has the potential of driving the network to operate around the knee. Although there are some controversies on regarding queue delay as a congestion signal and some measurement results in the Internet are also quite negative [29][99][113], it is another story on long fat network pipes of the Internet. On these network pipes, per-flow throughput can be high and the senders could have enough RTT samples to learn network state correctly through queue delay. Furthermore, there are abundant residual bandwidth for bandwidth-greedy and elastic applications and their behaviors can affect the point at which these network pipes will operate. Hence, a delay-based HSCC algorithm should have the potential of driving long fat network pipes to operate around the knee and to distribute bandwidth fairly among competing flows. Following this promising direction, Synchronized TCP (Sync-TCP) is designed and evaluated in this chapter.

The key insight of Sync-TCP is that if competing flows could detect the same congestion signal through queue delay, these flows can then coordinate their congestion control behaviors for driving the network to operate around their desired point, the knee. This is in contrast to the classic view on the legacy loss-based TCP versions in which congestion synchronization leads to bad performance since it is caused by traffic overload and competing flows simultaneously reduce their sending rate (blindly) by half [13].

Based on the above observation, Sync-TCP is carefully designed such that with high probability, competing flows can detect the same congestion signal through queue delay. In combination with *synchronized* congestion signal, Sync-TCP uses an adaptive queue-delay-based congestion window decrease rule, and a RTT-independent congestion window increase rule. These rules are designed to drive the network to operate around the knee and to distribute the residual bandwidth fairly even when the number of competing flows varies and their RTPDs differ significantly. Extensive simulations and preliminary testbed evaluations indicate that Sync-TCP does achieve its design goals.

This chapter is organized as follows. Section 3.2 firstly presents several influential delay-based congestion control algorithms. The challenges, that should be solved for driving the network to operate around the knee and for distributing bandwidth fairly among competing flows, are then discussed in section 3.3. After that, section 3.4 presents the details of Sync-TCP, its deployment issues, and parameter selection guidelines. Extensive simulations and preliminary testbed evaluations are presented in section 3.5 and 3.6. Finally, related work is discussed in section 3.7, and this chapter is summarized in section 3.8.

3.2 Background

Since the potential of delay-based congestion control was pointed out [73], many delay-based congestion control algorithms have been proposed. A sender driven by such an algorithm can be regarded as an intelligent agent that learns and controls network state based on queue delay, which must be deduced from RTT samples measured at the sender. In the rest of this section, several influential delay-based congestion control algorithms will be introduced based on how a sender samples RTT experienced by its packets, deduces queue delay from these RTT samples, determines network state according to queue delay, and adjusts its sending rate accordingly with the aim to drive the network to operate around its desired point.

3.2.1 TCP Vegas

TCP Vegas [33], a matured and influential delay-based congestion control algorithm, was proposed in 1994 and had attracted a lot of attention [65][95][103]. At the end of each round trip time, TCP Vegas measures *srtt*, the RTT experienced by the packet transmitted at the start of the current round trip time. TCP Vegas then calculates $\triangle$, the difference between the expected and the actual throughput, according to equation 3.1.

$$\begin{aligned} \triangle &= (thr_{expected} - thr_{actual}) * brtt \\ &= (\frac{cwnd}{brtt} - \frac{cwnd}{srtt}) * brtt = \frac{cwnd * (srtt - brtt)}{brtt * srtt} * brtt \\ &= thr_{actual} * (srtt - brtt) = thr_{actual} * qd \end{aligned} \tag{3.1}$$

Here, *brtt* is set to the minimum of all RTT samples. Approximately, *brtt* can be regarded as the estimation of RTPD. *qd* is the difference between *srtt* and *brtt*, and it can be regarded as the estimation of current queue delay's absolute value.

As shown in the following equation, the new calculated $\triangle$ is then compared with two constants, γ_1 and γ_2 ($\gamma_1 < \gamma_2$), and *cwnd* is adjusted based on these comparison results.

$$cwnd_{i+1} = \begin{cases} cwnd_i + 1 & \triangle < \gamma_1, \\ cwnd_i - 1 & \triangle > \gamma_2, \\ cwnd_i & otherwise. \end{cases}$$

It is obvious that TCP Vegas adjusts *cwnd* with the aim to maintain several packets (between γ_1 and γ_2) in the queue of the bottleneck link. Consequently, with the increase in the number of competing TCP Vegas flows, more and more packets are maintained in the queue of the bottleneck link, and the operation point of the bottleneck link slides from the knee to the cliff. Hence, TCP Vegas is not designed to drive the network to operate around the knee independent of the number of competing TCP Vegas flows.

In addition, as existing TCP Vegas flows have already maintained some packets in the queue of the bottleneck link, a new flow may not learn its RTPD correctly, and its *brtt* is very likely to be larger than its real RTPD. Hence, compared with existing flows, a new flow will maintain more packets in the queue and acquire more bandwidth. Furthermore, with the arrival and departure of competing flows, RTPD estimation error of active flows can be increased and they may drive the network to operate in a persistent-congested state [87].

Due to the above problems and the possibility of being starved by loss-based TCP Reno [103], TCP Vegas has not been widely deployed in the Internet. Since TCP Vegas increases *cwnd* by at most one segment per round trip time, TCP Vegas is not suitable for bandwidth-greedy and elastic applications on long fat network pipes of the Internet. Similarly, some delay-based congestion control algorithms, such as TCP-LP [85] and TCP Nice [130] that are proposed for low-priority background transfers, are not suitable too.

3.2.2 Delay-based HSCC Algorithms

Fast TCP [75]

Fast TCP is designed to utilize long fat network pipes efficiently, stably, and fairly. Fast TCP uses pacing [13] to smooth the packets to be transmitted and RTT is sampled on each acknowledgement (ACK). The smooth average of these RTT samples, *srtt*, is used to judge network state, and it is updated based on the exponentially weighted moving average (EWMA) algorithm shown in equation 3.2. Here, μ_k, the smooth factor, is adjusted based on the value of *cwnd* when rtt_k (the k^{th} RTT sample) is measured so that *srtt* can reflect queue dynamics of a time window, whose length is about one round trip time.

$$srtt_{k+1} = (1 - \mu_k) * srtt_k + \mu_k * rtt_k$$

At the end of every other round trip time, Fast TCP updates *cwnd* based on equation 3.2 with the aim to maintain γ packets in the queue of the bottleneck link. Here, η is a smooth factor used for updating *cwnd*. *brtt* is the minimum of all RTT samples.

$$cwnd_{i+1} = (1 - \eta) * cwnd_i + \eta * (\frac{brtt}{srtt} * cwnd_i + \gamma) \tag{3.2}$$

In Fast TCP, the value of γ also determines how quickly Fast TCP can increase or decrease its *cwnd* based on queue delay. In order to work well on long fat network pipes, this value has to be sufficiently large as well. In the prototype implementation of Fast TCP [75], γ is a large constant (about 100 segments). Hence, Fast TCP can be regarded as a high speed version of TCP Vegas and it also inherits the problems found in TCP Vegas. With a large γ, more packets remain in the network and these problems even become worse. A small number of Fast TCP flows will drive the network to operate far (right) away from the knee. Furthermore, due to the large γ, many packets could be dropped if buffer-overflow occurs before Fast TCP flows could detect congestion through queue delay. Consequently, cross traffic can be adversely affected.

To improve its scalability with network bandwidth, the authors of Fast TCP also propose that when *qd* (the difference between *srtt* and *brtt*) is very small, γ can be a function of *cwnd*. But this modification's effects on the metrics of convergence and fairness should be investigated further. Strange convergence behaviors have been observed in some experimental studies [93]. In this chapter, Fast TCP without this modification will be evaluated and compared with Sync-TCP.

In addition, even when the network is under-utilized, *srtt* still can be a little larger than *brtt* due to the variance of processing delay at each node of the network path (endpoints, routers, etc.) and the burstiness of cross traffic. According to equation 3.2, when *cwnd* is huge, this small difference may cause Fast TCP to estimate network state in wrong, decrease its sending rate unnecessarily, and result in network under-utilization.

Compound TCP (CTCP) [125]

Compound TCP runs two congestion control algorithms concurrently: the legacy TCP's AIMD algorithm and a delay-based HSCC algorithm. *win*, which determines the sending rate of CTCP, is the sum of *cwnd* and *dwnd*. Here, *cwnd* follows the legacy TCP's AIMD algorithm, and *dwnd* is adjusted based on its delay-based HSCC algorithm.

In its delay-based HSCC algorithm, CTCP measures one RTT sample per millisecond (ms). At the end of each round trip time, $\triangle$ is calculated according to equation 3.1, in which *cwnd* is substituted by *win*, *brtt* is still the minimum of all RTT samples, but *srtt* is the arithmetic average of RTT samples measured in the current round trip time. Based on $\triangle$ and a global constant (γ), *dwnd* is adjusted according to equation 3.3.

$$dwnd_{i+1} = \begin{cases} dwnd_i + (\alpha * win_i^k - 1)^+ & \triangle < \gamma, \\ (dwnd_i - \zeta * \triangle)^+ & \triangle \geq \gamma, \\ (win_i * (1 - \beta) - cwnd/2)^+ & loss. \end{cases} \tag{3.3}$$

Here, $(\cdot)^+$ is defined as $max(\cdot , 0)$ so that $dwnd$ will not be less than 0 and CTCP will not acquire less throughput than the legacy TCP. β is a constant and is now set to 1/2 so that CTCP will reduce sending rate by half when segment loss is detected.

α and k are also constants whose values are carefully selected so that when $\triangle$ is less than γ, CTCP increases win like Highspeed TCP [51]. When congestion is detected through queue delay, the reduction of $dwnd$ is related with $\triangle$ (a little larger than γ) and may become too small when per-flow throughput is high. Hence, when driven by the delay-based HSCC algorithm, CTCP acts like a MIAD (Multiplicative Increase and Additive Decrease) algorithm and competing flows converge slowly [43][92]. Considering that competing CTCP flows may act like the legacy TCP at different times, convergence behaviors of these flows can be very complex [136].

ζ is a constant that affects the desired operation point of CTCP. The number of packets, that each CTCP flow maintains in the queue of the bottleneck link, fluctuates between ϕ and γ ($\phi < \gamma$). The smaller ζ is, the larger ϕ is. However, even when $\zeta > 1$, CTCP still cannot ensure that the queue of the bottleneck link can be emptied. According to equation 3.1, flows with higher throughput will detect congestion before flows with lower throughput. Although this method can speed up convergence, competing flows will not detect congestion simultaneously. Hence, competing flows cannot reduce $dwnd$ simultaneously, and consequently the queue of the bottleneck link cannot be emptied and new flows cannot estimate their RTPDs correctly. Furthermore, the number of packets buffered in the queue of the bottleneck link also tends to increase with the increasing number of competing CTCP flows. In a word, CTCP will also suffer the problems found in TCP Vegas.

In CTCP, γ should not be too small. According to equation 3.1, when thr_{actual} is high on long fat network pipes and γ is small, the noise in RTT samples can be wrongly regarded as congestion signals. These spurious signals will result in unnecessary sending rate reduction and network under-utilization. This issue also indicates that CTCP-Tube [124] may not work well when its auto-tuned γ becomes too small [92].

With a large γ, a small number of CTCP flows may also cause buffer-overflow before congestion could be detected through queue delay. As the rate, that *dwnd* is increased, is related with *win* and can be very large on long fat network pipes, many packets could be dropped in a congestion event and cross traffic could be hurt.

Yeah-TCP [20]

At the end of each round trip time, Yeah-TCP still uses equation 3.1 to judge network state. Here, *srtt* is the minimum of RTT samples measured in this round trip time. Although RTT samples in one round trip time are lower bounded by the sum of RTPD and the current queue delay, different flows may observe different *srtt* since only one RTT sample is used, RTT samples are noisy, and network load may vary even within one round trip when a HSCC algorithm is used. Hence, Yeah-TCP flows are likely to suffer short-term unfairness.

Based on Δ, Yeah-TCP also adjusts *cwnd* with the aim to maintain γ (a constant) packets in the queue of the bottleneck link. Hence, Yeah-TCP is not designed to drive the network to operate around the knee, irrespective of the number of competing flows. Furthermore, Yeah-TCP tries to detect whether it is competing with loss-based flows. If there is no competing loss-based flows, Yeah-TCP is in *slow* mode and acts like Fast TCP. Otherwise, it is in *fast* mode and does not reduce *cwnd* when congestion is detected through queue delay.

In addition, when segment loss is detected, unlike TCP Vegas, Fast TCP, and CTCP that reduce sending rate by half, Yeah-TCP follows TCPW-RE [132], that reduces *cwnd* based on the value of queue delay. Hence, Yeah-TCP can work better in high speed wireless network, in which BDP is large and packet corruption rate is high.

TCP Fusion [77]

At the end of each round trip time, TCP Fusion also uses equation 3.1 to judge network state. However, it does not specify how to sample RTT and how to update *srtt* based on RTT samples. Furthermore, the thresholds (γ_1 and γ_2), that Δ is compared with, are set to

$cwnd * \frac{4ms}{srtt}$ and $cwnd * \frac{12ms}{srtt}$. According to equation 3.1, TCP Fusion fundamentally judges network state by comparing the absolute value of queue delay with 4ms and 12ms. Hence, TCP Fusion is not designed to maintain several packets in the queue of the bottleneck link for each flow. In fact, all competing TCP Fusion flows adjust their *cwnd* with the aim to keep the queue delay (at the bottleneck link) between 4ms and 12ms. That means TCP Fusion has the potential of being scalable with the number of competing TCP Fusion flows. However, since the existing flows still try to occupy some part of the queue of the bottleneck link, unfairness to old flow and persistent congestion found in TCP Vegas still may exist.

Like CTCP, to avoid being starved by the legacy TCP, TCP Fusion maintains $cwnd_{reno}$ by strictly following the legacy TCP's AIMD algorithm and its sending rate is determined by $cwnd_{reno}$ when $cwnd \leq cwnd_{reno}$. Hence, TCP Fusion is not designed for driving the network to operate around the knee too. In addition, when segment loss is detected, TCP Fusion acts like Yeah-TCP in the way of reducing *cwnd*.

Delay-based AIMD [48]

Delay-based AIMD is the first HSCC algorithm that is explicitly designed to drive the network to operate around the knee, irrespective of the number of competing flows. To perform good in the metrics of convergence and fairness, an adaptive AIMD algorithm is adopted by Delay-based AIMD.

For each ACK, RTT is sampled and *srtt* is a smoothed estimation of these samples. However, it does not specify how to smooth RTT samples. Network state is then judged based on *qd* (the difference between *srtt* and *brtt*) and τ_0 (a threshold). Here, *brtt* is still the minimum of all RTT samples.

If $qd < \tau_0$, Delay-based AIMD tries to increase *cwnd* by α segments per round trip time. As shown in equation 3.4, when each ACK is received, α is updated following the same convex and RTT-independent function used by H-TCP. Here, t is the absolute time elapsed since the last *cwnd* reduction, and T_0 is set to one second.

$$\alpha(t) = \begin{cases} 1 & t \leq T_0, \\ 1 + 10 * (t - T_0) + (\frac{t-T_0}{2})^2 & t > T_0. \end{cases} \tag{3.4}$$

If $qd \geq \tau_0$ and *cwnd* is larger than some threshold, *cwnd* is reduced immediately and a new congestion epoch begins. Here, congestion epoch is the period between two consecutive *cwnd* reductions. As for β (the multiplicative decrease factor), it is calculated according to equation 3.5.

$$\beta = \delta * \frac{brtt}{srtt}, \qquad \delta < 1 \tag{3.5}$$

The authors argue that $\frac{brtt}{srtt}$ is the value, with which the queue of the bottleneck link can be emptied if there is no RTPD estimation error. By multiplying with δ, which is smaller than one, *cwnd* is over-reduced and the correct RTPD can be finally observed even when the initial *brtt* is larger than the real RTPD [49].

However, depending on the kind and the load of cross traffic applications, β in equation 3.5 may not be small enough to empty the queue of the bottleneck link. This issue is discussed further in subsection 3.3.2. Furthermore, there is one implicit assumption that queue delay is simultaneously regarded as a congestion signal by almost all competing Delay-based AIMD flows. Otherwise, only a part of competing flows will reduce their sending rate, and the total reduction of network load may not be large enough to empty the queue of the bottleneck link. But Delay-based AIMD does not carefully consider how to carry out queue delay measurement for assuring this assumption. It even adjusts τ_0 based on the following equation for coexisting with flows driven by loss-based congestion control algorithms.

$$\tau_0 = (1 - \zeta) * \tau_0 + \zeta * (rtt_{max} - brtt)$$

Since there is no upper bound for rtt_{max} (the maximum of all RTT samples), τ_0 maintained by competing flows may be different. Hence, it is even harder for competing flows to simultaneously regard queue delay as a congestion signal.

In addition, *cwnd* is reduced immediately when congestion is detected through queue delay. Considering that queue delay is a *delayed* network feedback, the *cwnd* reduction based on queue delay at one round trip time ago may not be large enough, especially when Delay-based AIMD is used and α can be large. After *cwnd* is reduced, *cwnd* is also immediately increased for probing network resources. It is very likely that the queue of the bottleneck link will not be emptied and competing flows cannot observe their real RTPD. In a word, Delay-based AIMD may not be able to achieve its design goals.

3.3 Challenges and Key Observations

Based on the above section, most of the existing delay-based HSCC algorithms are not designed for driving the network to operate around the knee and Delay-based AIMD still has some problems to achieve its aims on long fat network pipes of the Internet. Hence, it should be worthwhile to further investigate the challenges, that Sync-TCP must solve, for driving these network pipes to operate around the knee and for distributing network resources fairly based on queue delay, especially when the number of competing flows varies and their RTPDs differ significantly.

In order to achieve the above purposes, Sync-TCP should first enable competing flows to learn network state correctly based on queue delay. First of all, competing flows should know which point is the knee. In another word, these flows should be able to learn their RTPD correctly. However, it is not an easy task on long fat network pipes of the Internet.

In the existing delay-based congestion control algorithms, *brtt*, the estimation of RTPD, is set to the minimum of all RTT samples. This method is reasonable as RTT samples are lower bounded by RTPD. However, if the distributed senders cannot cooperate with each other and some packets always remain in the queue of the bottleneck link, the new arrival flows will not be able to observe their correct RTPD. With the arrival and departure of flows, RTPD estimation error of active flows can be increased and the network will not operate around the knee. Hence, the correlation between congestion control mechanism and RTPD estimation must be considered carefully.

The key insight of Sync-TCP is that if competing flows could detect the same congestion signals through queue delay, these flows can then coordinate their congestion control behaviors for driving the network to operate around their desired point, the knee. More specifically, they can reduce *cwnd* based on the value of queue delay so that the queue of the bottleneck link can be emptied periodically and the network will not be under-utilized obviously. Consequently, the new flows can estimate their RTPD correctly and the bottleneck link can keep operating around the knee with the arrival and departure of flows. This is in contrast to the classic view on loss-based TCP, in which congestion synchronization leads to bad performance since segment loss is caused by traffic overload and competing flows blindly reduce their sending rate by half at the same time [13].

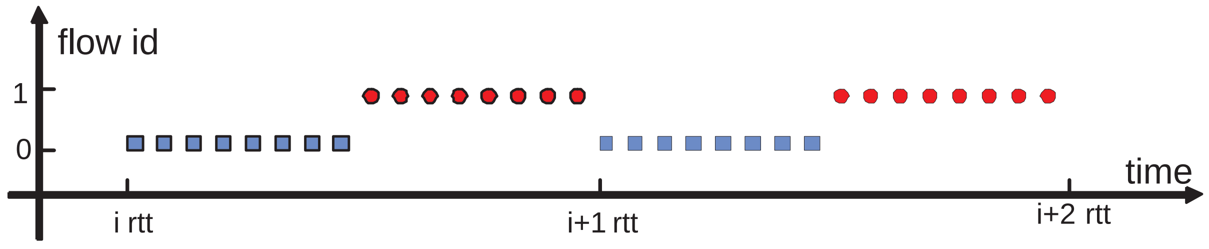
It is important to note while the term *synchronize* is used to describe how competing flows should detect a congestion signal through queue delay, all Sync-TCP needs is a consistent view. Hence, the measurements need not enable competing flows to detect congestion at the same time[1]. Instead, the *congested state* only need last for a sufficiently long period so that competing flows have high probability of detecting the same congestion state.

In the following subsections, we will discuss how to enable competing flows to simultaneously detect congestion signal through queue delay. According to simple analysis [43] and stochastic matrix model [119], AIMD-based flows can converge quickly and share network resources fairly in synchronized communication networks. Hence, Sync-TCP will adopt an adaptive AIMD algorithm. In the following subsections, we will also discuss how to set β (the multiplicative decrease factor) for emptying the queue of the bottleneck link and how to set α (the additive increase factor) for efficiency, fairness, etc.

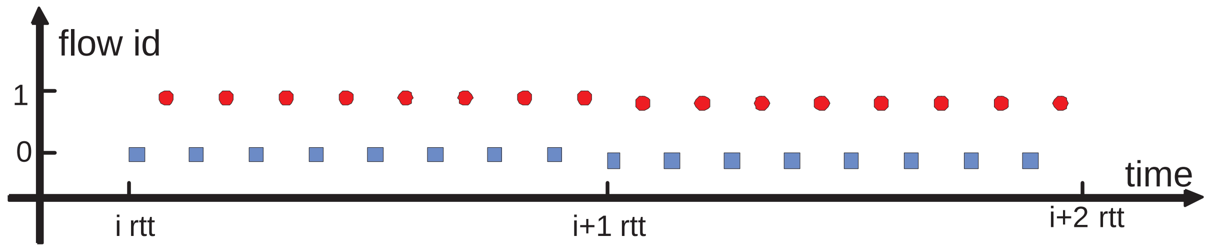
[1]In fact, queue delay is a delayed network feedback. It may be impossible for competing flows with different RTPDs to detect congestion through queue delay at the same time.

3.3.1 How to Simultaneously Detect Queue-Delay-Based Congestion Signals?

As an end-to-end congestion control mechanism, Sync-TCP is fundamentally a distributed algorithm. Even under the assumption that RTPD is correctly estimated by *brtt*, it is still not easy to enable competing flows to detect congestion simultaneously through queue delay, which must be deduced from the RTT samples measured at the sender. Although competing flows all pass through the same bottleneck link, they still may not be able to have the same view of queue dynamics at the bottleneck link.



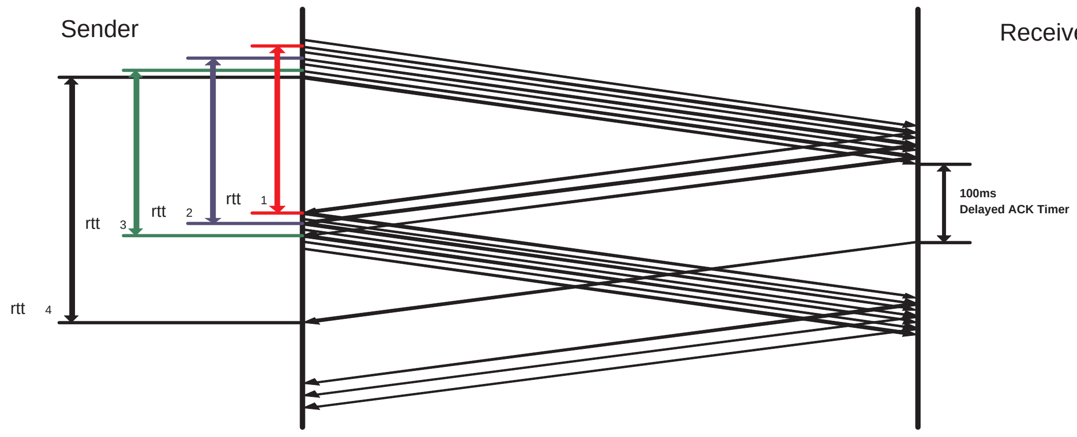
(a) TCP



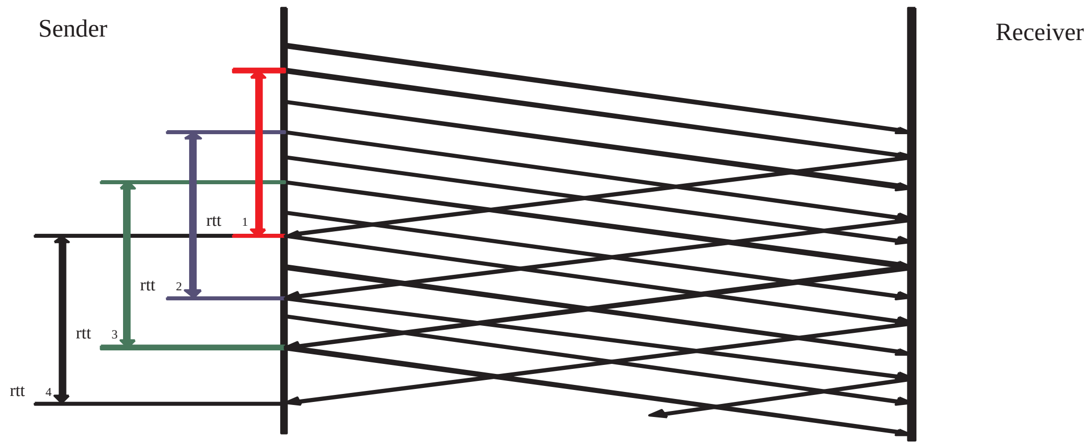
(b) Pacing

Figure 3.2: Packet Arrival Time of Two Competing Flows

Firstly, queue delay is an *elusive* network feedback. Queue dynamics of the bottleneck link can only be observed by a sender through RTT experienced by its sampled packets. Since TCP is a window-based congestion control, packets sent by a TCP flow tend to be clustered [123]. Figure 3.2(a) depicts the time, that the packets of two competing TCP flows with the same RTPD arrive at the bottleneck link. When a HSCC algorithm is used, queue length may be changed obviously during one round trip time. Hence, different flows, that pass through the same bottleneck link, may have different view of network state and acquire different throughput. In addition, as shown in 3.3(a), Delayed ACK [44] may also cause large noise in RTT samples.



(a) TCP



(b) Pacing

Figure 3.3: The Effects of Delayed ACK on RTT Measurement

As shown in figure 3.2(b) and 3.3(b), pacing [13] can distribute a window of segments evenly within one round trip time and the above problems can be solved. Curiously, among the existing delay-based HSCC algorithms, only Fast TCP explicitly adopts pacing to smooth large data burst generated by the sender in high speed networks.

One consideration is that pacing may consume too much system resources, especially CPU time spent for context switches that are triggered by the expiration events of pacing timer. Since computers are more and more powerful, pacing is very useful on long-fat network pipes, and there are some efficient pacing algorithms [84], the overhead should be low and worthwhile. According to our implementation in FreeBSD and third party's implementation in Linux [75], CPU processing overhead only increases slightly when the pacing granularity is several millisecond.

It has also been reported that pacing will synchronize congestion signals (segment loss) among TCP flows and achieve less overall throughput since TCP flows reduce their *cwnd* by half simultaneously [13]. However, the synchronization of queue-delay-based congestion signal is just what Sync-TCP needs. Sync-TCP can also reduce *cwnd* based on the value of queue delay so that the network will not be under-utilized obviously.

Secondly, due to cross traffic, scheduling granularity, and processing delay, RTT samples measured by a flow can be very noisy. Considering that per-flow throughput on long fat network pipes can be high, the sender could have enough RTT samples to learn network state correctly through queue delay. It can use *srtt*, the smoothed average of these RTT samples, to estimate the current queue delay and judge network state.

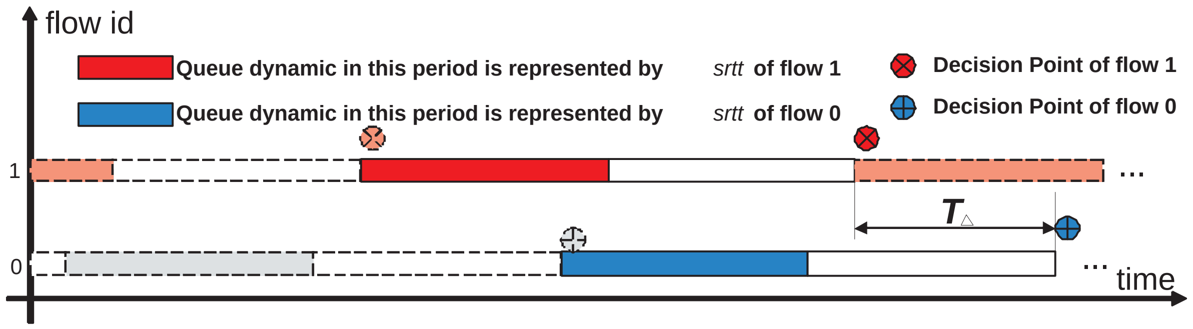

flow id
Queue dynamic in this period is represented by srtt of flow 1
Queue dynamic in this period is represented by srtt of flow 0
Decision Point of flow 1
Decision Point of flow 0
1
0
T△
...
time

Figure 3.4: Queue Delay Measurement and Detection of Two Competing Fast TCP Flows with the Same RTPD

Thirdly, many existing delay-based HSCC algorithms determine network state at the end of each or each other round trip time based on *srtt*, which is normally the average of RTT samples measured in the current round trip time. Figure 3.4 illustrates the problem when two Fast TCP flows with the same RTPD pass through the same bottleneck link and compete with each other. It is obvious that these competing flows try to detect congestion at different times based on queue delay values, which reflect queue dynamics of the bottleneck link during different time windows. On long fat network pipes, RTPD is normally quite long and T_Δ (the interval between the nearest points that competing flows determine network state) may also be quite large. After a HSCC algorithm is deployed, queue length of the bottleneck link may change significantly even within T_Δ. Hence, competing flows may end up having different views of the congestion level. When competing flows have different RTPD, the situation would likely be worse. To solve these issues, *srtt* should reflect queue dynamics of a fixed-length time window and the senders should judge network state more frequently.

Fourthly, many existing delay-based HSCC algorithms judge network state through comparing Δ, which is calculated according to equation 3.1, with some constants. With this method, flows with higher throughput will detect congestion earlier than flows with lower throughput. Although this method can speed up convergence, competing flows will not detect congestion simultaneously. Hence, Sync-TCP should detect congestion by comparing the absolute value of queue delay with a constant.

Finally, the existing delay-based HSCC algorithms reduce *cwnd* immediately when congestion is detected through queue delay. Since RTT samples are noisy, when one flow detects congestion through queue delay, queue delay observed by other competing flows may still be less than the threshold. If this flow reduces *cwnd* immediately, other flows may miss this congestion signal. To solve this problem, Sync-TCP should freeze its *cwnd* and delay the reduction of *cwnd* so that network load can keep to be high for a while and competing flows could also observe this congestion signal.

3.3.2 How to Reduce *cwnd* for Emptying the Queue of the Bottleneck Link?

When a congestion signal is detected by all competing flows simultaneously through queue delay, *cwnd* should be reduced based on the value of queue delay so that the queue of the bottleneck link can be emptied and the network will not be under-utilized. But there are still several issues to be considered, and it is even impossible to achieve these goals under some scenarios of the Internet.

Firstly, the existing delay-based HSCC algorithms reduce *cwnd* immediately after congestion is detected through queue delay. However, queue delay is a *delayed* network feedback. The real queue delay can be much larger than the current queue delay observed by the sender, especially when a HSCC algorithm is adopted. For reducing *cwnd* based on the real queue delay caused by the current *cwnd*, the sender should freeze *cwnd* for at least one RTT before reducing *cwnd*. This is another reason for delaying the reduction of *cwnd*.

Secondly, after *cwnd* is reduced, the existing delay-based HSCC algorithms begin to increase *cwnd* immediately. However, the sender should wait for a while so that the bottleneck link can empty packets previously buffered in the queue and competing flows can learn their RTPD correctly.

Thirdly, there are different kinds of cross traffic applications and they may consume different amount of bandwidth. This fact makes it even harder to decide how to reduce *cwnd*. The following analysis indicate that *cwnd should be reduced based on not only the value of queue delay, but also the kind and the load of cross traffic applications.*

For simplicity, we assume that all flows have the same RTPD and the following symbols are defined for analyzing this issue further based on Little's Law in queueing theory.

1. C: the bandwidth of the bottleneck link.
2. W_s: the number of packets that are pumped into the network pipe by all competing Sync-TCP flows before *cwnd* is reduced. Here, $W_s = \sum_{i=0}^{N-1} cwnd_i$ and N is the number of competing Sync-TCP flows.
3. W_c^-: the number of packets that are pumped into the network pipe by cross traffic applications before competing Sync-TCP flows reduce their *cwnd*.
4. W_c^+: the number of packets that are pumped into the network pipe by cross traffic applications after competing Sync-TCP flows have reduced their *cwnd*.

5. $srtt_{reduce}$: the value of *srtt* when *cwnd* is reduced. Since all competing Sync-TCP flows have the same RTPD, we can assume that these flows observe the same $srtt_{reduce}$.

6. β: the multiplicative decrease factor used by competing Sync-TCP flows. Since all competing flows have the same RTPD and observe the same $srtt_{reduce}$, they should use the same β.

Before competing Sync-TCP flows reduce their *cwnd*, the bottleneck link is fully utilized since its queue is not empty. Hence, $W_s + W_c^- = C * srtt_{reduce}$. To make sure that the queue of the bottleneck link can be emptied, β should satisfy the following condition.

$$W_s * \beta + W_c^+ <= C * brtt \Longleftrightarrow \beta <= \frac{C * brtt - W_c^+}{C * srtt_{reduce} - W_c^-}$$

In the case that $W_c^+ > C * brtt$, the queue of the bottleneck link cannot be emptied even when all competing Sync-TCP flows totally stop to transmit. This scenario may exist when cross traffic applications themselves can fully utilize the network and they do not reduce sending rate when queue delay is increased. In this scenario, bandwidth-greedy and elastic applications had better switch back to the legacy TCP for acquiring their fair share of bandwidth.

In the case that $W_c^+ < C * brtt$, it becomes possible to empty the queue of the bottleneck link. However, even without considering the dynamics of cross traffic flows, the kind and the load of cross traffic still bring many challenges to the way of calculating β.

If cross traffic applications are rate-based or transactional, such as VoIP and web surfing, based on Law of Large Numbers, we can assumed that R_c, the data rate generated by all cross traffic applications, is a constant. Hence,

$$W_c^+ = R_c * brtt, \text{ and } W_c^- = R_c * srtt_{reduce}$$

$$\beta <= \frac{C * brtt - R_c * brtt}{C * srtt_{reduce} - R_c * srtt_{reduce}} = \frac{brtt}{srtt_{reduce}}$$

This value has been adopted by TCPW-RE [132], TCP Fusion, Delay-based AIMD, etc. However, if cross traffic applications are long-lived flows driven by window-based flow control, such as the legacy FTP applications whose throughput is constrained by socket buffer, queue delay can affect their sending rate. With Law of Large Numbers, it is more reasonable to assume that the number of packets pumped into the network pipe by these cross traffic applications, is a constant.

$$W_c^- = W_c^+ = N_c * B = C * x$$

Here, N_c is the number of cross FTP flows and B is the socket buffer size of the legacy FTP clients. For simplicity, x is used to specify the load of legacy FTP applications. Hence, β should satisfy the following condition.

$$\beta <= \frac{C * brtt - C * x}{C * srtt_{reduce} - C * x} = \frac{brtt - x}{srtt_{reduce} - x} \tag{3.6}$$

Obviously, the value of β depends on not only queue delay, but also the kind and the load of cross traffic applications.

Considering that on long fat network pipes of the Internet, there are many kinds of cross traffic applications and they may acquire different amount bandwidth of the bottleneck link, we should pay more attention on how to calculate β based on queue delay.

3.3.3 How to Increase *cwnd* for Efficiency and Fairness?

When designing the rule used to increase *cwnd*, many metrics (efficiency, convergence, fairness, etc.) must be well balanced. In Sync-TCP, α, the additive increase factor, should be designed based on the following considerations with the aim to utilize network resources efficiently and share network resources fairly independent of their RTPD values.

- In order to work well on long fat network pipes, α should increase with the elapse of time. The reason is that if congestion is not detected after an extended period of time, there are either very few competing flows or there are substantial excess bandwidth, and *cwnd* should be increased more quickly.

- In order to distribute bandwidth fairly among competing flows independent of their RTPD values, α should be calculated by a RTT-independent function. Furthermore, instead of increasing *cwnd* by α segments per round trip time, it should be increased by α segments per *fixed-length* period so that all competing flows will increase *cwnd* by the same amount of segments in the same period.

3.4 The Design of Sync-TCP

Based on observations in previous section, Sync-TCP is carefully designed such that with high probability, competing flows can detect the same congestion signals through queue delay. In combination with *synchronized* congestion signals, Sync-TCP uses an adaptive queue-delay-based congestion window decrease rule and a RTT-independent congestion window increase rule. These rules are designed to drive the network to operate around the knee and to distribute the residual bandwidth fairly even when the number of competing flows varies and their RTPDs differ significantly. In this section, Sync-TCP is first briefly introduced. The design details of Sync-TCP, its deployment issues, and parameter selection guidelines are then presented in the following sub-sections.

3.4.1 Overview of Sync-TCP

Sync-TCP can be summarized through the state transition diagram shown in figure 3.5. In all sub-states of Sync-TCP (*Probing*, *Waiting*, and *Emptying*), RTT is sampled periodically. After each RTT sample is measured, the state variables related with queue delay measurement are updated. Only in the *Probing* sub-state, *cwnd* is increased and queue-delay-based congestion detection is carried out.

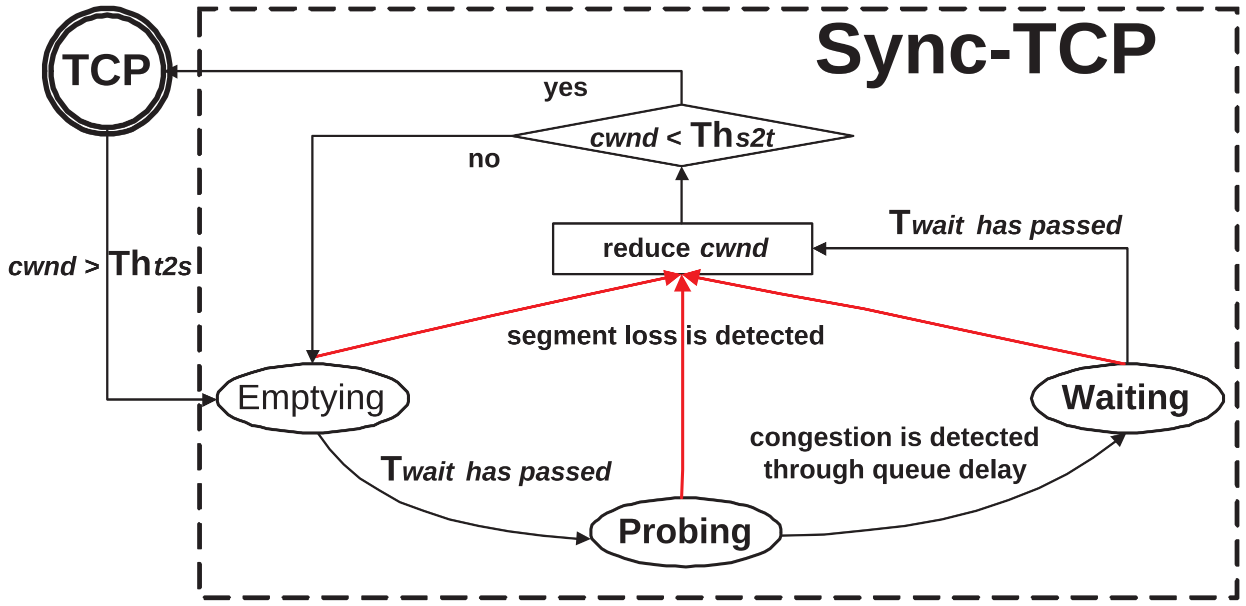


Figure 3.5: State Transition Diagram of Sync-TCP

1. When congestion is detected through queue delay, the sender leaves *Probing* sub-state and enters into *Waiting* sub-state. In *Waiting* sub-state, *cwnd* is frozen and queue-delay-based congestion detection is not carried out. The sender will stay in *Waiting* sub-state for a short period (T_{wait}) so that network load can be high for a while and competing flows will detect the same congestion signal.

2. When Sync-TCP leaves the *Waiting* sub-state, *cwnd* is reduced. The sender will then enter into *Emptying* sub-state, in which the just reduced *cwnd* is also frozen and queue-delay-based congestion detection is not carried out too. The sender will also stay in *Emptying* sub-state for T_{wait} so that the bottleneck link could have some time to empty packets previously buffered in its queue.

3. When Sync-TCP leaves the *Emptying* sub-state, the sender enters into *Probing* sub-state and starts to increase *cwnd* for probing network capacity again. When probing network capacity, queue-delay-based congestion detection is also carried out.

As shown in figure 3.5, when segment loss is detected, Sync-TCP will reduce its sending rate immediately for safety. Figure 3.5 also illustrates that a flow will switch between Sync-

TCP and TCP. At the beginning, TCP is used as default since the sender does not know the size of its network pipe. When its throughput is higher than a threshold ($cwnd > Th_{t2s}$), the sender will switch to Sync-TCP for efficiently utilizing the abundant network bandwidth. When its throughput becomes lower than another threshold ($cwnd < Th_{s2t}$), the sender will also switch back to TCP. The reason is that Sync-TCP is a delay-based congestion control algorithm. It should be used only when there are enough RTT samples to learn network state correctly. Switching back to TCP can also avoid that a Sync-TCP flow is totally starved when rerouting occurs, when the flow passes through multiple congested links, or when some loss-based flows coexist.

Since there are some adverse effects when working with TCP [13], pacing is activated only when Sync-TCP is used. Hence, pacing needs to be switched on/off when the sender switches between Sync-TCP and TCP. When TCP is used, pacing is deactivated and Delayed ACK may generate large noise in RTT samples. To avoid detecting spurious congestion signal (queue delay) immediately after switching to Sync-TCP, when $cwnd > Th_{t2s}$, the sender will activate pacing and enter into *Emptying* sub-state, in which Sync-TCP does not carry out congestion detection through queue delay. When Sync-TCP leaves *Emptying* sub-state after T_{wait} and enters into *Probing* sub-state, pacing should have already distributed packets evenly within a round trip time and Delayed ACK is not a threat any more.

3.4.2 Queue Delay Measurement and Congestion Detection

RTT Sampling

In order to save system resources (CPU, Memory, etc.) of a sender in high speed networks, instead of sampling on each ACK, Sync-TCP only measure RTT once per T_{sample}. Here, T_{sample} should be much smaller than round trip time of a long fat network pipe. In Sync-TCP, pacing is also adopted so that a sender can sample queue delay evenly within one round trip time. Sync-TCP does not require T_{pace}, the granularity of pacing timer, to be very small. It is enough if T_{pace} is less than T_{sample}.

srtt, *brtt*, and $brtt_{epoch}$ Updating

Considering that RTT samples are noisy and the importance of a RTT sample decreases with the elapse of time, *srtt*, that is used for calculating queue delay, should be an exponentially smoothed average of all measured RTT samples (without the boundary of round trip time). Hence, after measuring a RTT sample (rtt_i), *srtt* is updated based on equation 3.7, which is essentially an EWMA algorithm.

$$srtt_{i+1} = (1 - \frac{T_{sample}}{T_{win}}) * srtt_i + \frac{T_{sample}}{T_{win}} * rtt_i \tag{3.7}$$

In order to ensure competing flows, whose RTPD may be different, to have a consistent view of network state, it is necessary that the values of their *srtt* reflect queue dynamics of a *fixed-length time window*. Hence, T_{win} should be a global constant, and the typical value could be the average of round trip times experienced by all Sync-TCP flows on long fat network pipes of the Internet. Except *srtt*, *brtt* and $brtt_{epoch}$ should also be updated based on the current RTT sample (rtt_i).

$$brtt = min(brtt, rtt_i); \quad brtt_{epoch} = min(brtt_{epoch}, rtt_i)$$

In Sync-TCP, *brtt* is also the minimum of all RTT samples. Hence, *brtt* is immediately set to rtt_i if it is less than the current *brtt*. As for $brtt_{epoch}$, it is the minimum of RTT samples observed in the current congestion epoch. In Sync-TCP, congestion epoch is defined as the period between two consecutive *cwnd* reductions. When *cwnd* is reduced, $brtt_{epoch}$ will be used to judge whether previous *cwnd* reduction is large enough to empty the queue of the bottleneck link, and β will be adjusted correspondingly.

Congestion Detection

After *srtt* is updated with the current RTT sample, Sync-TCP calculates *qd* (the difference between *srtt* and *brtt*) and compares it with Th_{qd}, a global constant. If $qd > Th_{qd}$, a congestion signal is detected by this Sync-TCP flow.

The value of Th_{qd} reflects the amount of buffering desired at the bottleneck link (independent of the number of competing flows) and determines queue delay & jitter that cross traffic applications will experience. Section 3.4.7 will discuss how to set the value of Th_{qd}.

Since Sync-TCP flow determines network state per RTT sample, when pacing is adopted, $T_{\triangle}$ (the interval between the nearest points that determines the network state of competing flows) should not be much larger than T_{sample}, which is much smaller than RTPD of a flow on long fat network pipe. Hence, competing flows should be able to detect congestion at almost the same time and queue delay observed by them should be close to each other.

3.4.3 Delayed *cwnd* Decrease/Increase

According to discussions in section 3.3, after a congestion signal is detected through queue delay, instead of reducing *cwnd* immediately, *cwnd* is frozen for T_{wait} before it is reduced. In this *Waiting* period, RTT is sampled, *srtt* & *brtt* & $brtt_{epoch}$ are updated, but queue delay based congestion detection is not carried out and *cwnd* is not changed.

This delayed *cwnd* reduction is introduced for two purposes. Firstly, queue delay is a *delayed* network feedback. With this delayed *cwnd* reduction, end hosts can reduce *cwnd* based on the real queue delay caused by the *frozen cwnd*. Secondly, by keeping network load constant and high for a period (T_{wait}), it is unlikely that competing flows will miss this congestion signal and observe different queue delay. In general, for synchronizing congestion signal, T_{wait} should be larger than $\kappa * T_{win}$. Here, κ is a small integer. For reducing *cwnd* based on the correct queue delay, T_{wait} should be larger than $RTT_{max} + \kappa * T_{win}$. Here, RTT_{max} is the longest RTT experienced by all Sync-TCP flows. Hence, T_{wait} should be set according to $RTT_{max} + \kappa * T_{win}$.

After *cwnd* is reduced, $brtt_{epoch}$ is set to a huge value for tracking the minimal RTT sample that will be observed in the following congestion epoch. Instead of increasing *cwnd* immediately, Sync-TCP will also freeze the just reduced *cwnd* for T_{wait} so that the bottleneck link could have some time to empty packets previously buffered in its queue and competing flows could estimate their RTPD more accurately. Due to the following adaptive queue-delay-based *cwnd* decrease rule adopted by Sync-TCP, this *Emptying* period will not lead to much lower link utilization ratio as there are still sufficient packets in the network. In this *Emptying* period, the behaviors of Sync-TCP are identical to the behaviors in the above *Waiting* period.

According to discussions in section 3.3, *cwnd* should be adjusted based on an adaptive AIMD algorithm. The following subsections will present the details.

3.4.4 RTT-Independent *cwnd* Increase Rule

In Sync-TCP, a flow increases *cwnd* by α segment per T_{win}, and α is calculated based on equation 3.8. Here, t is the clock-time elapsed since Sync-TCP began to increase *cwnd*, in the unit of second.

$$\alpha = \max((1 + t + \frac{t^4}{32}) * \frac{Th_{qd} - qd}{Th_{qd}}, 1) \tag{3.8}$$

Hence, Sync-TCP and H-TCP adopt the same general form of how α should be increased.

$$\alpha = a_0 + a_1 * t + a_i * t^i, (i \geq 2).$$

α is first increased slowly (through the constant and linear term t) for safety. With the elapse of time, α increases very faster depending on the values of a_i and i. In H-TCP [91], a_0, a_1, i and a_i are set to 1, 10, 2, and $\frac{1}{4}$ respectively. Sync-TCP adopts different values so that *cwnd* can be increased much slower at the beginning for friendliness and the acceleration speed can be much higher for efficiency. By setting a_0, a_1, i and a_i to 1, 1, 4, and $\frac{1}{32}$ respectively, the higher order term dominates over the linear term t when t increases beyond 2.4s. Note that the exact coefficients and power of these terms are not crucial for the correctness but do affect the aggressiveness of *cwnd* increase and the speed of convergence. These values are currently selected based on simulation results. In addition, α will not be less than one segment so that when probing network resources, Sync-TCP will not be slower than the legacy TCP.

Considering that queue delay is a *delayed* network feedback, $\frac{Th_{qd}-qd}{Th_{qd}}$ is used here to slow down the increase of α as qd approaches Th_{qd} such that the maximal queue delay will not be much larger than Th_{qd}. It may be better if α has an upper bound, such as $\frac{5ms}{srtt} * cwnd * \frac{T_{win}}{srtt}$, which could make sure that queue delay will not be increased by more than 5ms per round trip time. But the convergence among competing flows may be slowed down. For results presented in section 3.5 and section 3.6, α does not have such an upper bound.

In order to save CPU, α is updated only once per T_{win} (based on qd at that moment). As for *cwnd*, it is adjusted when a RTT sample is measured (per T_{sample}) so that *cwnd* is evenly increased by α segments per T_{win}.

3.4.5 Adaptive Queue-delay-based *cwnd* Decrease Rule

With highly *synchronized* congestion signals and the delayed *cwnd* decrease mechanism, competing Sync-TCP flows could reduce *cwnd* simultaneously based on the real queue delay. Sync-TCP calculates, β, the multiplicative decrease factor, based on equation 3.9. Here, $srtt_{reduce}$ is the value of $srtt$ when $cwnd$ is reduced and qd_{reduce} is the difference between $srtt_{reduce}$ and $brtt$.

$$\beta = 1 - \frac{\lambda * qd_{reduce}}{srtt_{reduce}}, 0.125 \leq \beta \leq 0.95 \tag{3.9}$$

Based on discussions in subsection 3.3.2, to ensure that the queue of the bottleneck link is emptied and the network is not under-utilized, β should adopt different values according to queue delay and the kind & the load of cross traffic. Hence, λ should be an adaptive value that is always larger than 1. Considering that it is very hard for endpoints to estimate the kind & the load of cross traffic, λ is adjusted based on the following simple rule.

In Sync-TCP, the default value of λ is set to a small constant (1.25). When *cwnd* is reduced, Sync-TCP first calculates the difference between $brtt_{epoch}$ and $brtt$, and the result is used to judge whether the current λ is appropriate.

If the difference is larger than $Th_{emptied}$, it indicates that β calculated with the current λ cannot empty the queue of the bottleneck link. Hence, λ is increased by one. This large increase step is used to ensure that the queue of the bottleneck link can be emptied in a short time. If the difference is less than $Th_{emptied}$, it indicates that the current λ is large enough and λ might need to be decreased. Considering that we cannot deduce how much the current λ is larger than its optimal value, λ is reset to 1.25.

Due to the noise in RTT samples, $brtt_{epoch}$ still can be larger than $brtt$ even when the queue of the bottleneck link is empty. Since timer and scheduling granularity of the current operating systems used by end hosts is normally 1ms, $Th_{emptied}$ is set to 2ms in this work. Although the above adaptive rule performs quite well in simulations and testbed evaluations, the adaptive rule and its parameters might need to be tuned according to the real world.

Since $brtt_{epoch}$ is used to judge whether the queue is emptied and λ is reset to a common value when the queue is emptied, it is unlikely that λ of competing flows will converge to different values.

Considering that queue delay is a highly synchronized congestion signal and the additive increase rule used by Sync-TCP is RTT-independent, all competing Sync-TCP flows will increase their *cwnd* by the same number of segments (N_{incr}) within a congestion epoch. In a converged state, for each flow, N_{incr} equals to N_{decr}, the number of segments that *cwnd* is reduced due to a queue-delay-based congestion signal. Hence, all competing flows should have the same N_{decr}.

$$N_{decr} = (1-\beta) * cwnd = \frac{\lambda * qd_{reduce}}{srtt_{reduce}} * cwnd = \lambda * qd_{reduce} * thr^{-}$$

According to the above equation, thr^{-} (the throughput acquired before *cwnd* is reduced) should equal to each other since Sync-TCP is carefully designed so that competing flows could observe the same queue delay and use the same λ. In summary, *Sync-TCP is designed to distribute bandwidth fairly among competing flows independent of their RTPD values.*

3.4.6 Deployment Issues

Handling Segment Loss

When Sync-TCP (a delay-based HSCC algorithm) is used, segments are lost mainly due to transmission error. Hence, segment loss normally is not a signal of network congestion. When segment loss is detected, Sync-TCP still reduces *cwnd* for safety. In order to avoid under-utilizing the lossy and fast links, β (the multiplicative decrease factor) is also calculated based on equation 3.9.

On the other hand, segment loss may also be a signal of severe network congestion. For example, when many flows arrive simultaneously or the queue of the bottleneck link is too small, segments can be dropped due to buffer overflow. Hence, when segment loss is detected, the upper bound of β is set to 0.875 and *cwnd* is reduced immediately.

Switching between TCP and Sync-TCP

In general, when a flow begins to transmit, it does not know the type of its network pipe. Hence, the legacy TCP (e.g. Reno) should be used first and Sync-TCP is activated only when the flow is sure that it is on a long fat network pipe. In another word, Sync-TCP is activated only when *cwnd* is large enough. Furthermore, due to the following considerations, when its throughput is too low, Sync-TCP flow should also switch back to TCP.

Firstly, per-flow throughput should be high enough so that the senders have enough RTT samples to measure queue delay correctly. Secondly, for all of the following possible scenarios in which the throughput is low, Sync-TCP has justified reasons to switch back to TCP.

- Taking into account how Sync-TCP detects congestion and handles segment loss, it should be obvious that when the network utilization ratio is high because of the increased legacy TCP applications and interactive applications, Sync-TCP may be starved and per-flow throughput becomes low. In this scenario, letting flows of bandwidth-greedy and elastic applications act as legacy TCP is fairer and provides more incentives to adopt Sync-TCP.

- Since Sync-TCP detects congestion by comparing queue delay with a constant, a Sync-TCP flow which passes through multiple congested links (MCL), can be starved by Sync-TCP flows which only pass through one of these congested links. We argue that it is reasonable since this flow consumes more network resources for transmitting the same amount of data. MCL unfairness of Sync-TCP may motivate large content providers to deploy more mirrors and reduce the load of core networks. In addition, when a flow does pass through MCL and the throughput becomes too low, it should switch back to TCP to avoid being totally starved.

- When there are huge number of competing Sync-TCP flows, per-flow throughput will become low. Considering that queue delay is a *delayed* network feedback and *srtt* is a smoothed average of RTT samples, even though *cwnd* is frozen when queue delay is regarded as a congestion signal by the sender, qd_{max}, the maximal queue delay at the bottleneck link should be larger than Th_{qd}.

  This fact is helpful for synchronizing competing Sync-TCP flows. However, when there are huge number of competing flows, the difference between qd_{max} and Th_{qd} can be quite large. For controlling the jitter suffered by cross traffic and potential segment loss, when per-flow throughput is low, it is better to switch back to TCP (with the cost of increased average queue delay). Based on Little's Law in queueing theory, this issue is analyzed further in the following paragraphs.

For simplicity, we assume that Sync-TCP flows have identical RTPD, which is close to T_{win}. According to equation 3.8, when qd approaches Th_{qd} and there are many competing flows, α should equal to one segment. Since *cwnd* is frozen in *waiting* period, qd_{max} should satisfy the following equation.

$$C * rtt_{detect} + N * mss = C * (brtt + qd_{max})$$

Here, C is the bottleneck link's bandwidth, N is the number of competing Sync-TCP flows, mss is the maximum segment size, and rtt_{detect} is the value of the RTT sample which triggers Sync-TCP to regard queue delay as a congestion signal. Considering that $srtt$ is used to calculate qd, rtt_{detect} should be a little larger than $brtt + Th_{qd}$. Approximately,

$$\frac{C}{N} = \frac{mss}{brtt + qd_{max} - rtt_{detect}} \approx \frac{mss}{qd_{max} - Th_{qd}}$$

Since bandwidth-greedy and elastic applications may not acquire all bandwidth, the throughput of each Sync-TCP flow should be less than $\frac{C}{N}$. Consequently, we can decide qd_{max} based on QoS (Quality of Service) requirements of the expected cross traffic applications (VoIP, etc.) and calculate the threshold of per-flow throughput according to equation 3.10.

$$Th_{thr} = \frac{mss}{qd_{max} - Th_{qd}} \tag{3.10}$$

When per-flow throughput is lower than Th_{thr}, Sync-TCP should switch back to the legacy TCP.

Handling Rerouting

In Sync-TCP, *brtt* is also set to the minimum of all RTT samples. However, if RTPD is increased due to the change of network path, RTPD cannot be correctly estimated and λ will always be increased when *cwnd* is reduced. Although the long fat link of core network is not likely to be changed during the life of a Sync-TCP flow, other links in regional networks still can be changed due to mobility, load balance, etc. Hence, this issue must be considered.

In the case that there are some competing Sync-TCP flows, this flow will receive low and lower throughput. Hence, it will switch back to TCP. When switching to TCP, Sync-TCP will set *brtt* to a huge value so that this flow can correctly learn the increased RTPD and take back its fair share of bandwidth.

In the case that there is no other competing Sync-TCP flows or all flows experience rerouting simultaneously, this flow may not switch back to TCP since β has a lower bound. Consequently, Sync-TCP is still used, its sending rate will keep fluctuating a lot, and the bottleneck link can be slightly under-utilized. Hence, when *cwnd* is reduced and λ is adjusted, if λ is huge (> 20) and $brtt_{epoch}$ is still larger than *brtt*, Sync-TCP will set *brtt* to $brtt_{epoch}$ and λ is reset to 1.25.

Misbehaving Users

Fundamentally, Sync-TCP is a delay-based high speed congestion control algorithm. When users access servers driven by Sync-TCP, they have a very strong motivation to conceal queue delay from the server and get higher throughput. For example, when the real RTPD is 100ms and there is no queue delay, the receiver puts off its ACK packets for 20ms and the sender's RTPD estimation becomes 120ms. When queue is built up in network, the receiver can shorten the period, that ACK packets are delayed, according to the value of current queue delay. Hence, the sender cannot detect congestion through queue delay and this misbehaving user acquires (unfairly) higher throughput.

To perform the above attack without being caught up, the misbehaving user should be able to estimate queue delay in the network accurately. When time stamp option is used for RTT measurement, queue delay can be estimated based on the change of one way delay exposed by time stamp option. Hence, time stamp option is not used by Sync-TCP. To sample RTT experienced by a packet, the sender will record the sequence number and the sending time, and RTT is calculated when the corresponding acknowledgement is received. Hence, Sync-TCP increases the difficulty of this kind of attack through consuming more system resources of TCP sender. This is why we claim that RTT measurement is expensive and only one sample is measured per T_{sample}.

3.4.7 Parameter Selection Guidelines

Although many parameters are used by Sync-TCP, we only need to determine the values of the following global parameters. As for other parameters, they can be deduced based on these global parameters and/or host-specific configurations.

T_{win}

T_{win} is an important global variable of Sync-TCP. As shown in equation 3.7, when the sender updates *srtt*, T_{win} is used so that competing flows could have a consistent view of network state independent of their RTPD values. T_{win} must be large enough so that enough samples are considered, the noise in RTT samples can be filtered out, and *srtt* can correctly reflect queue dynamics in the last T_{win} period. On the other hand, if T_{win} is too large, response to changes in queue dynamics will be affected.

The value of T_{win} is determined in the following way. First, as T_{sample} determines the amount of processing and memory overhead required, we assume that a value of 10ms can be supported by most endpoints without imposing excessive load. Next, a sample size of at least 10 within a window of T_{win} is assumed to be needed for filtering out noise. Hence, the value of T_{win} is set to 100ms, so that there are at least 10 samples, with at least one sample every 10ms.

Since the value of T_{sample} is set to 10ms, T_{Δ}, the interval between the nearest points that competing flows determine network state, will not be larger than T_{sample} and will be much smaller than round trip time of a long fat network pipe. Hence, competing Sync-TCP flows will determine network state much more frequently. Consequently, they will detect congestion signal at almost the same time. In addition, with the current value of T_{sample}, T_{pace} can be as large as several milliseconds and CPU overhead of pacing will not be high.

T_{wait}

This global variable is used for synchronizing congestion signal and reducing *cwnd* based on the real queue delay caused by the frozen *cwnd*. In order for Sync-TCP to function correctly, T_{wait} must be long enough so that all Sync-TCP endpoints detect this synchronization signal.

For this to be the case with high probability, T_{wait} should be larger than $RTT_{max}+\kappa*T_{win}$. Here, RTT_{max} is the longest RTT experienced by all Sync-TCP flows and κ is a small number (not less than 1). This is to ensure all endpoints will detect the congestion signal with sufficient confidence. Taking into the maximum possible network path in a terrestrial network and the light speed in wire, T_{wait} is currently set to 500ms.

Th_{qd}

As the threshold used for detecting congestion through queue delay, the value of Th_{qd} reflects the amount of buffering desired at the bottleneck link (independent of the number of competing flows). Considering that *srtt* is a smoothed average of RTT samples and queue delay is a *delayed* network feedback, the largest queue delay at the bottleneck link should be larger than Th_{qd}. If qd_{max} is the largest queue delay that cross traffic applications can tolerate, Th_{qd} should be a value that is less than qd_{max}.

Considering that average one-way jitter experienced by VoIP should be targeted at less than 30ms[2] and VoIP packets may pass through multiple congested links, qd_{max} is set to

[2]http://www.ciscopress.com/articles/article.asp?p=357102

20ms when deciding parameters of Sync-TCP. With $qd_{max} = 20ms$, Th_{qd} is now set to 12ms. The current Th_{qd} should be large enough to avoid regarding noise in RTT samples as a congestion signal. Since Sync-TCP reduces $cwnd$ based on qd_{reduce}, which is normally larger than Th_{qd}, the current Th_{qd} should also be large enough so that competing Sync-TCP flows can converge quickly.

In order to prevent a flow from switching too frequently between TCP and Sync-TCP, two thresholds in the unit of segment, Th_{t2s} and Th_{s2t} (Th_{t2s}>Th_{s2t}), are adopted. When $cwnd$ is larger than Th_{t2s}, the sender switches from TCP to Sync-TCP and it also switches from Sync-TCP to TCP when $cwnd$ is less than Th_{s2t}. For each flow, it can calculate Th_{s2t} based on Th_{thr} in equation 3.10 and its own $brtt$. Currently, Th_{s2t} in the unit of segment is set according to equation 3.11. As for Th_{t2s}, it is set to $2 * Th_{s2t}$.

$$Th_{s2t} = (\frac{mss}{qd_{max} - Th_{qd}}) * \frac{brtt}{mss} = \frac{brtt}{8ms} \tag{3.11}$$

Hence, when Sync-TCP is adopted, each flow should be able to transmit at least one segment per 8ms. Considering that T_{sample} can be as large as 10ms, when Sync-TCP is used, the sender should have enough packets to sample queue delay correctly. Under the assumption that $mss = 1500$ bytes, when Sync-TCP is used, per-flow throughput will not be less than 1.5Mbps. This value should be high enough to motivate the deployment of Sync-TCP.

Please note that there are some assumptions in equation 3.10. Since some flows may have larger RTPD values, the load of cross traffic may fluctuate, etc., the queue size of the bottleneck link should be over-provisioned for avoiding congestive segment loss in more cases. More specifically, although Th_{s2t} is deduced with $qd_{max} = 20ms$, the queue size of the bottleneck link should be larger than $C * 20ms$.

Among the three global variables, Th_{qd} must be followed by all senders. As for T_{win} and T_{wait}, Sync-TCP still can work if the senders adopt slightly different values, but the fairness among their flows will be affected. Table 3.1 lists the parameters used by Sync-TCP. It also gives their values used in the evaluations of Sync-TCP.

	Scope	Value	Usages
T_{win}	global	100ms	the time window used for updating *srtt*. *cwnd* is also increased by α per T_{win}
T_{wait}	global	500ms	the period that Sync-TCP stays in *Waiting* and *Emptying* states
T_{sample}	host	10ms	the interval between two consecutive RTT samples
T_{pace}	host	5ms	the granularity of pacing timer
Th_{qd}	global	12ms	the threshold used to detect congestion through queue delay
Th_{s2t}	host	$\frac{brtt}{8ms}$	the threshold used to decide whether to switch to Sync-TCP
Th_{t2s}	host	$\frac{brtt}{4ms}$	the threshold used to decide whether to switch back to TCP

Table 3.1: Parameters of Sync-TCP

In the following two sections, we will present evaluation results of Sync-TCP. Sync-TCP has been evaluated through both simulation and testbed experiments. In the simulation, we can experiment with more high speed TCP variants, more variations in traffic load, and in particular, a much larger number of HSCC and cross traffic flows. On the other hand, we have a more realistic environment in the testbed evaluations, but the availability of high speed TCP variants and the scalability of the traffic load are much lower.

3.5 Simulation Results

Sync-TCP has been implemented in the framework of NS-2 TCP-Linux [133] so that we can compare it with HSCC algorithms that have been implemented in Linux, such as Cubic-TCP [59]. The leaky-bucket-based TCP pacing algorithm proposed in [84] is implemented for Sync-TCP and Fast TCP. Following the initial proposals, CTCP and Fast TCP are implemented in NS-2 TCP-Linux, and their parameters are set to default values. In this section, Sync-TCP is evaluated and compared with Cubic-TCP, CTCP, and Fast TCP. The legacy TCP implemented in Linux is also compared under the assumption that socket buffer is set to a large value and window scale option is enabled for supporting high speed data transmission. Although Delay-based AIMD is more similar to Sync-TCP, many details of Delay-based AIMD are not available and it is not evaluated here.

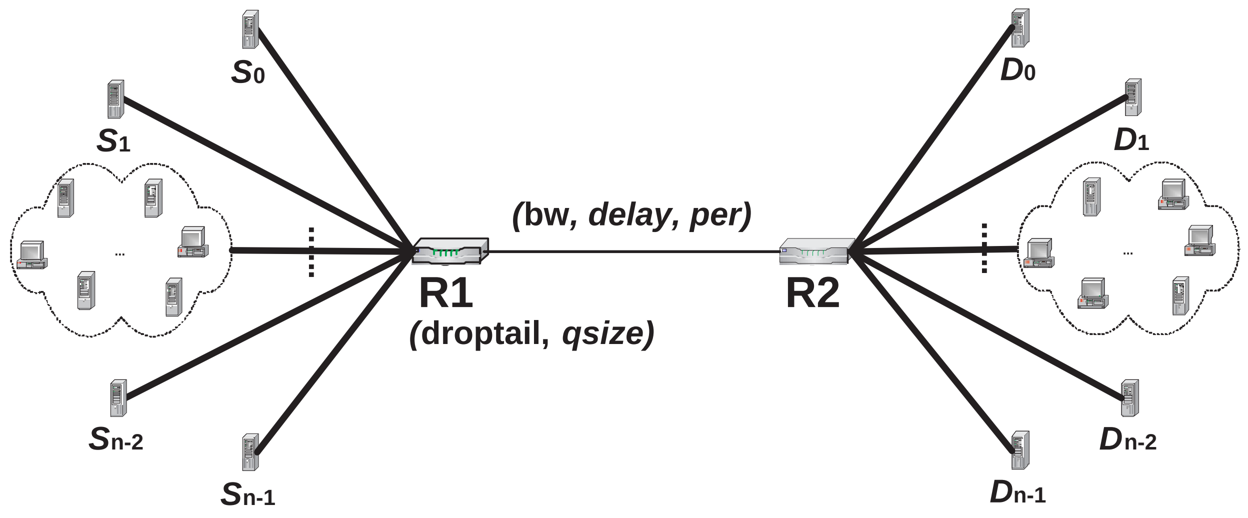


Figure 3.6: Dumbbell Network Topology

Unspecified, the dumbbell network configuration shown in figure 3.6 is used in this section. The link between $R1$ and $R2$ is a simulated transoceanic optical fibre link whose bandwidth is 1Gbps and DropTail is used by the routers. The simulated side links, that connect S_i to $R1$ or connect D_i to $R2$, are all highly reliable and fast enough so that $R1 \leftrightarrow R2$ is the only bottleneck. In all experiments, the flows from S_i to D_i are the flows to be investigated. For each experiment, five simulations are carried out, and in each simulation, one of the five congestion control algorithms to be investigated is used by these flows. With the results of these simulations, the five congestion control algorithms are compared.

The two clouds are responsible for generating background traffic. The links, that connect nodes in clouds to the two routers, are also highly reliable and fast enough. Their propagation delay follow an uniform distribution whose range is [5ms,25ms]. According to the connection-based web traffic model [39], the two clouds generate 3200 (800) HTTP sessions per second in the forward (reverse) path. For collecting user experience of the cross traffic applications, there are also several VoIP (Voice over IP) flows and several legacy FTP flows which are driven by the legacy TCP and are configured with small socket buffer (64KB).

These background traffic consumes about 300Mbps in the forward path and 100Mbps in the reverse path. Unspecified, this background traffic will be generated in all experiments presented in this section and Appendix A.

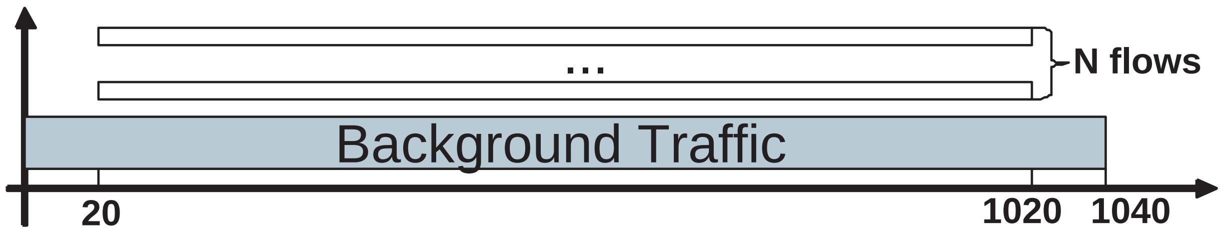


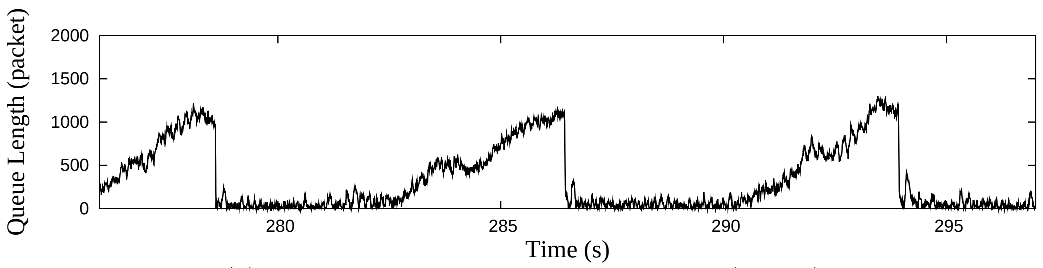
Figure 3.7: Block Scenario: the Arrival and Departure Sequence of Flows

Without specifying explicitly, flow arrival and departure sequence shown in block scenario (figure 3.7) is being used. In the following subsections, the setup of each experiment is described, simulation results are presented, and Sync-TCP is compared with other proposals.

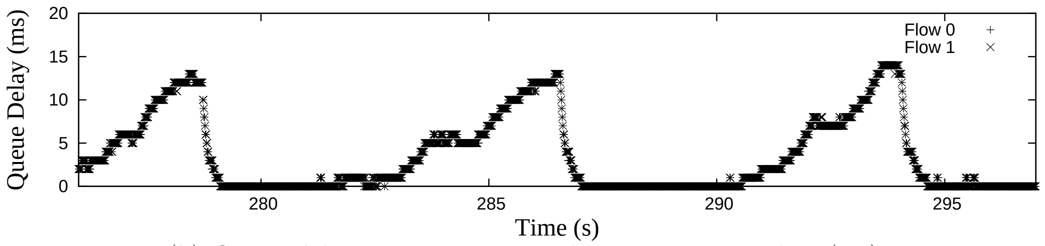
3.5.1 Synchronization of Congestion Signals

The first experiment is to demonstrate that competing Sync-TCP flows can have a consistent view of network state and detect congestion simultaneously through queue delay. Dumbbell topology and block scenario are used, and the bottleneck link is configured as propagation delay or $delay$=50ms, packet error rate or per=10^{-8}, and queue buffer size or $qsize$=0.5BDP. N, the number of competing flows to be investigated, is set to 2, and propagation delay of the side links are all set to 5ms.

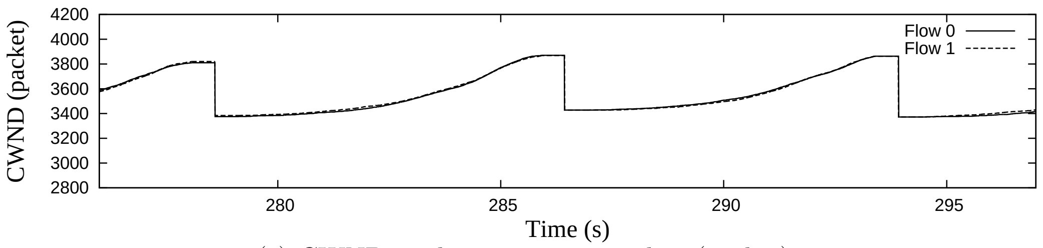
Figure 3.8 shows the queue dynamics of the bottleneck link and the behaviors of competing Sync-TCP flows. It indicates that competing Sync-TCP flows can observe their correct RTPD (queue can be emptied periodically), have a consistent view of network state (qd values observed by them are close to each other), and detect congestion and reduce $cwnd$ simultaneously. Within the 1000 seconds simulation of this experiment, 122 congestion signals are detected by Sync-TCP flows through queue delay and no single Sync-TCP flow misses anyone of these signals. As for Fast TCP and CTCP, figure 3.9 shows that competing flows



(a) Queue dynamics at the bottleneck link (packet)

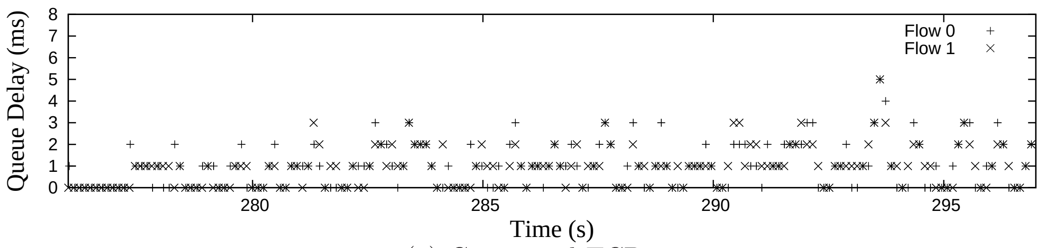


(b) Queue delay measurement at the competing senders (ms)

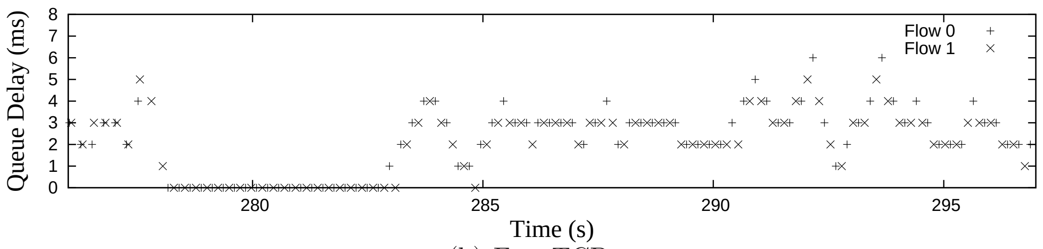


(c) CWND at the competing senders (packet)

Figure 3.8: Queue Delay Measurement and *cwnd* Evolution Behaviors of Competing Sync-TCP Flows Which Coexist with a lot of Cross Traffic



(a) Compound TCP



(b) Fast TCP

Figure 3.9: Decision Points and Queue Delay Observed by Competing Flows

judge network state at different times based on different *qd* values.

For the rest of the experiments presented below, the total simulation time is about 32,000 seconds. During these simulations, 5853 congestion signals are detected by Sync-TCP flows through queue delay and 5651 of these signals are detected by all competing Sync-TCP flows. *Note that in some simulations, the number of competing Sync-TCP flows can be very large (40, 64, 256, etc.) and their RTPD values can differ significantly.* 202 of these congestion signals are not detected by all competing Sync-TCP flows mainly because some flows have just experienced segment loss and are in the *Emptying* sub-state, in which queue-delay-based congestion detection is not carried out.

3.5.2 Scalability of Sync-TCP

In this subsection, dumbbell topology and block scenario are used. Propagation delay of the side links are all set to 5ms so that all competing flows have the same RTPD. As for $delay, per, qsize$ of the bottleneck link and N (the number of competing flows), different values are adopted. For each group of experiments, one parameter is varying and the other three parameters are fixed for investigating these congestion control algorithms' scalability with the varying parameter. In order to evaluate and compare these congestion control algorithms, the metrics to be used are listed below. Considering that Fast TCP and CTCP implementations in NS2 adopt the simple *slow start* algorithm of TCP, many segments may be dropped at the end of the first *slow start* phase. In order to avoid the bias on Fast TCP and CTCP in the metric of packet loss rate, we only present simulation results in steady state. More specifically, only the results during [100:1000] are plotted in this subsection.

1. Network Oriented Metrics: Three metrics, Link Utilization Ratio, Delay and Jitter, and Packet Loss Rate, are used to judge whether a congestion control algorithm can drive the network to operate around the knee. Link Utilization Ratio is the average utilization ratio of the bottleneck link. Delay and Jitter are the average queue delay and jitter experienced by packets at the bottleneck link. And Packet Loss Rate is the

probability that a packet is lost at the bottleneck link.

2. User Oriented Metrics: Fairness is measured by Jain's fairness index [72] of the average throughput acquired by competing flows. It is used to investigate whether flows can share network resources fairly in long term.

   Jain's fairness index of the average throughput acquired by competing flows in 1 second interval is also calculated, and their average during [100,1000] is used as the Short-term Fairness index. Short-term Fairness index is used to investigate the interaction among competing flows in small time scale.

   In addition, the stability or smooth of a flow's throughput is also evaluated. Following [75], S_i, the stability index of flow i, is the standard deviation normalized by the average throughput. Here, $\bar{x}_i$ is the average throughput of flow i during the interval [100,1000]. x_i is sampled per second during this interval. Hence, m equals to 900. The smaller the stability index, the less oscillation a flow experiences.

$$S_i = \frac{1}{\bar{x}_i}\sqrt{\frac{1}{m-1}\sum_{k=1}^{m}(x_i(k) - \bar{x}_i)^2}$$

   The stability metric shown in the following plots is the average over N flows.

Scalability with Flow Number

In this group of experiments, the bottleneck link is configured as *delay*=50ms, *per*=10^{-6}, and *qsize*=0.5BDP. N is then set to 1, 4, 16, 64, and 256 with the aim to investigate whether Sync-TCP can drive the network to operate around the knee independent of the number of competing flows.

Figure 3.10 shows link utilization ratio, packet loss rate, and queue delay & jitter of the bottleneck link. It indicates that Sync-TCP can drive the network to operate around the knee independent of the number of competing flows. Hence, the cross traffic applications will

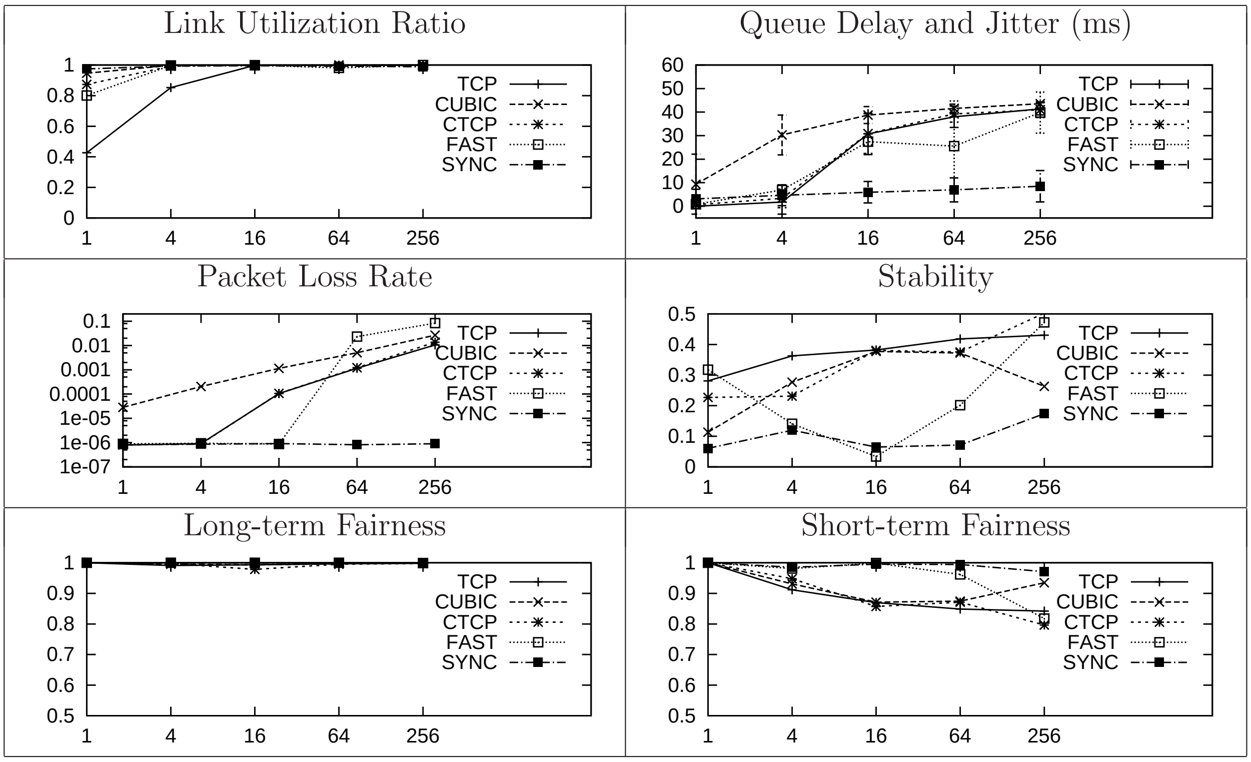


Figure 3.10: Scalability with Flow Number

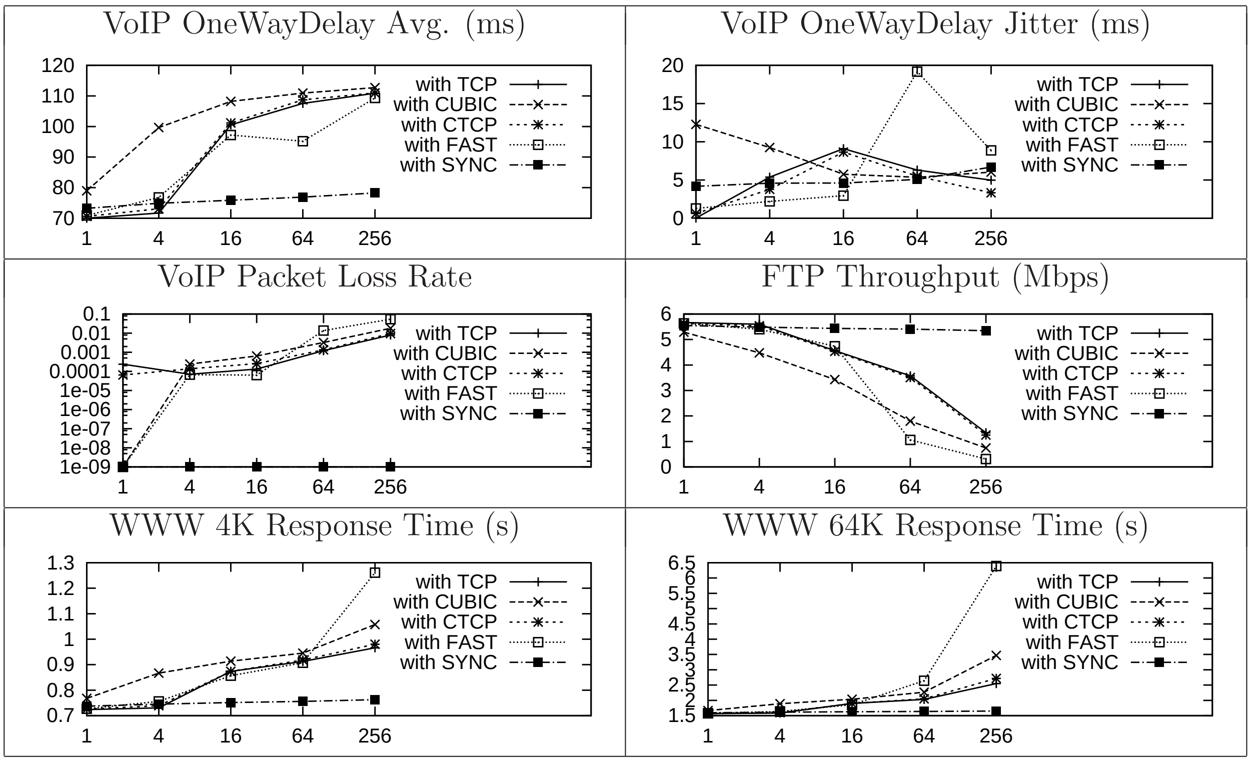


Figure 3.11: Scalability with Flow Number: User Experience of Cross Traffic Applications

not be hurt. This argument is supported by figure 3.11, which shows user experience of the VoIP, web surfing, and legacy FTP traffic. With 64 flows running other HSCC algorithms, the delay for web and VoIP traffic is almost 25% longer compare to Sync-TCP and the throughput for legacy FTP traffic is 30% lower.

Figure 3.10 also shows the fairness and stability indexes of competing flows. It indicates that Sync-TCP performs the best in all of these metrics, especially short-term fairness and stability. Hence, Sync-TCP can utilize the bottleneck link fairly and stably.

These congestion control algorithms have also been evaluated and compared when there are 1024 competing flows. In this case, per-flow throughput becomes very low and Sync-TCP will not be enabled. Simulation results do indicate that the performance is similar to TCP and is not worse than other HSCC algorithms. If Sync-TCP is always enabled, Sync-TCP can keep the friendliness to cross traffic even when $N = 1024$.

Scalability of Sync-TCP with respect to propagation delay, queue size, and packet loss rate have also been evaluated, and the simulation results are attached in Appendix A.1. Sync-TCP performs very well in all cases except when queue size is very small, such that the maximum queue delay is much less than Th_{qd}. In this case, Sync-TCP cannot detect congestion through queue delay, and $\frac{Th_{qd}-qd}{Th_{qd}}$ cannot effectively reduce α when buffer overflow approaches. It may be worthwhile to let a flow switch between Sync-TCP and Cubic TCP based on whether it can observe queue delay that is larger than Th_{qd}.

3.5.3 RTT Fairness

Dumbbell topology and block scenario are also used here. N is set to 2. As for the bottleneck link, $delay$=20ms, per=10^{-6}, and $qsize$=0.5BDP. Propagation delay of side links are set to different values in different experiments so that RTPD of flow 0 is always 60ms and RTPD of flow 1 is 60, 120, 240, 360, or 480ms.

Figure 3.12 shows the throughput ratio ($\frac{Thr_0}{Thr_1}$) against the RTPD ratio ($\frac{RTPD_1}{RTPD_0}$). It

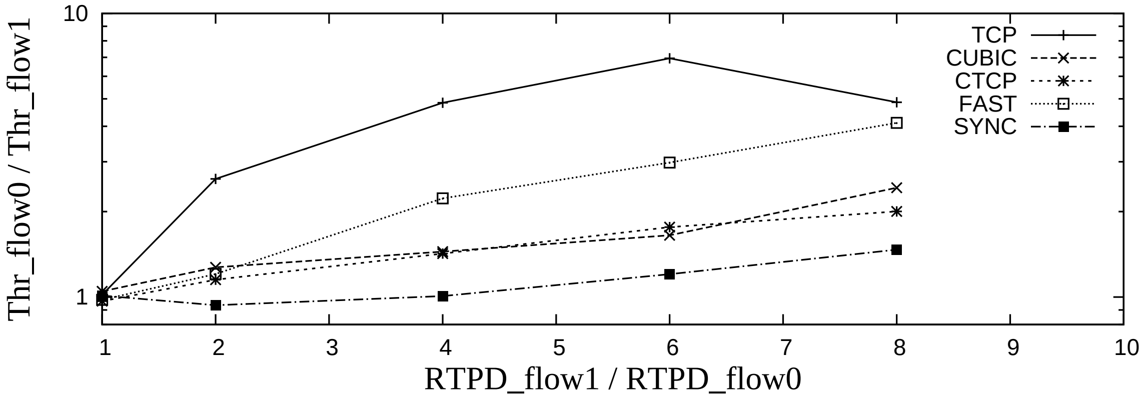


Figure 3.12: RTT Fairness ($per = 10^{-6}$)

indicates that Sync-TCP performs the best in the metric of RTT fairness. Irrespective of the competing flows' RTPD, Sync-TCP can distribute bandwidth fairly among them.

Curiously, although Fast TCP is designed to be RTT-independent, when Fast TCP is used, flow 0 still acquires higher throughput. The reason is that when some packets are corrupted and the sending rate of flow 0 and/or flow 1 is reduced, flow 0 can increase its *cwnd* with a higher frequency (due to its shorter RTPD). As shown in figure 3.13, when *per* is very small (10^{-8}), Fast TCP performs quit good in the metric of RTT fairness. Hence, this conjecture is confirmed.

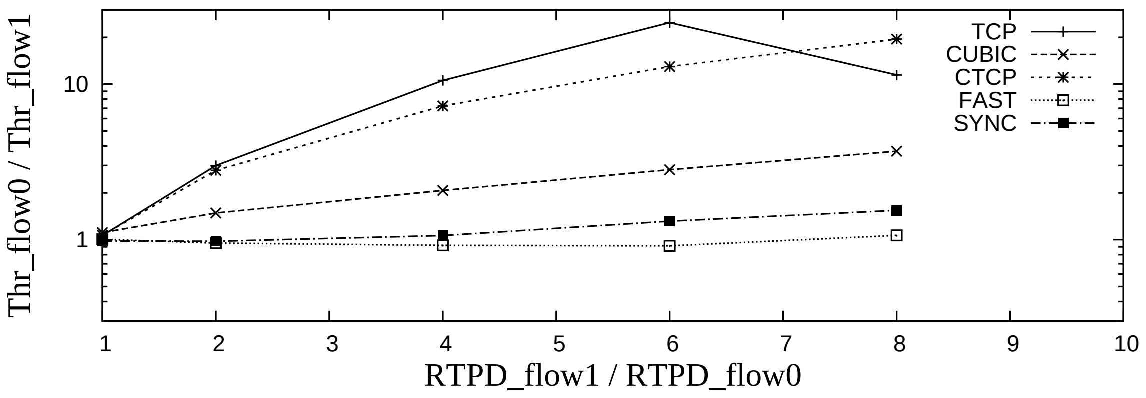


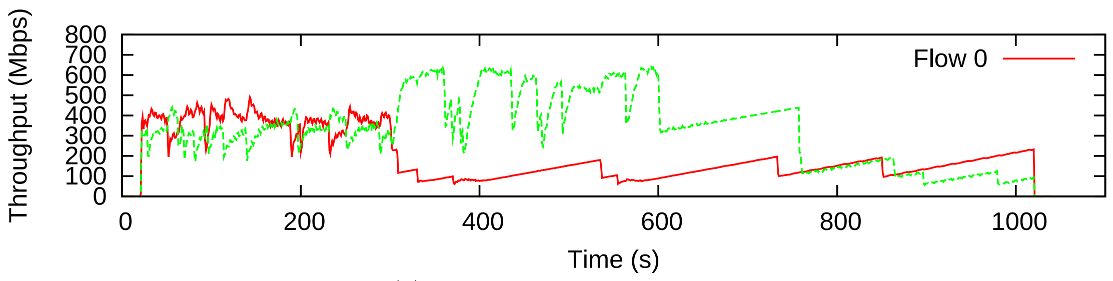
Figure 3.13: RTT Fairness ($per = 10^{-8}$)

However, when *per* is very small (10^{-8}), in the metric of RTT fairness, CTCP does not performs any better than TCP. The reason is that, *slow start* of TCP is used by CTCP. When the first congestive segment loss occurs, the sending rate of flow 1 is still very small (due to its long round trip time). After that, the two CTCP flows are driven by its delay-based HSCC algorithm, which acts like a MIAD algorithm. Consequently, the two flows converge very slowly and flow 1 keeps receiving much less throughput. With better algorithms in *slow start* phase, the situation may be better.

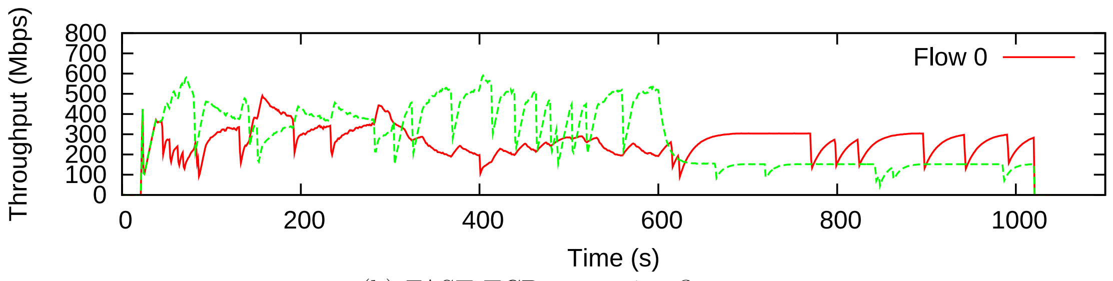
3.5.4 Rerouting Issue

In this group of experiments, dumbbell topology and block scenario are used. N is set to 2. The parameters of the bottleneck link are *delay*=50ms, *per*=10^{-6}, and *qsize*=0.5BDP. Propagation delay of side links are all set to 5ms at the beginning. At the 300^{th} second, propagation delay of the link between D_0 and $R2$ is changed to 7ms. This is used to simulate that flow 0 experiences rerouting and its RTPD is increased by 4ms. At the 600^{th} second, propagation delay of the link between D_1 and $R2$ is changed to 9ms. This is used to simulate that flow 1 experiences rerouting and its RTPD is increased by 8ms.

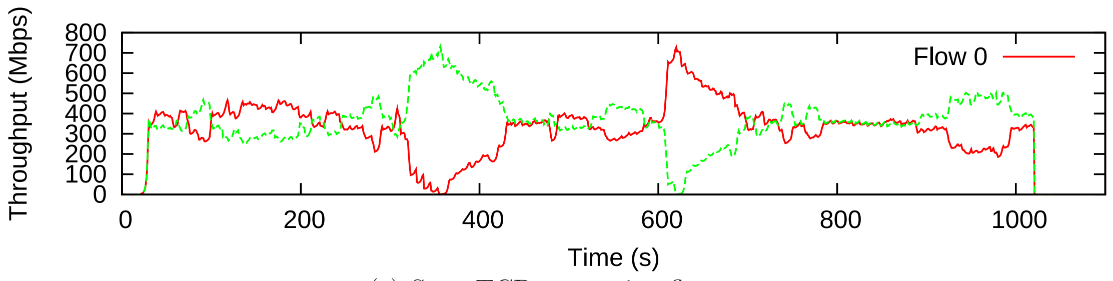
Figure 3.14 shows throughput trajectories of the two flows that are driven by CTCP, Fast TCP or Sync-TCP in which *brtt* is used. It indicates that only Sync-TCP can drive the two flows to converge to the fair and efficient point again. However, Sync-TCP flow, that experiences rerouting, cannot learn the increased RTPD correctly until it switches back to TCP. Hence, for a short period, its throughput will be very low. It may be worthwhile to investigate this issue further and improve Sync-TCP performance when rerouting occurs frequently in high speed wireless networks.



(a) CTCP competing flows



(b) FAST TCP competing flows

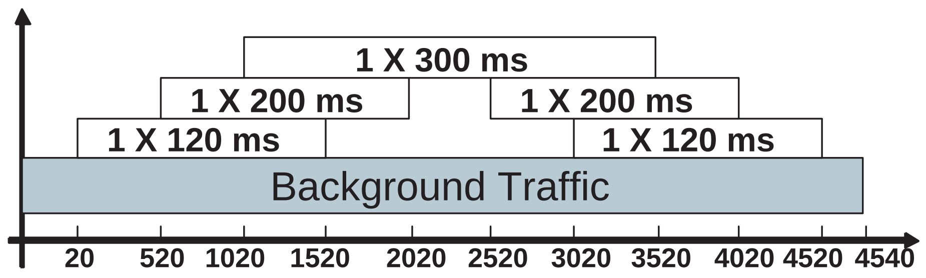


(c) Sync-TCP competing flows

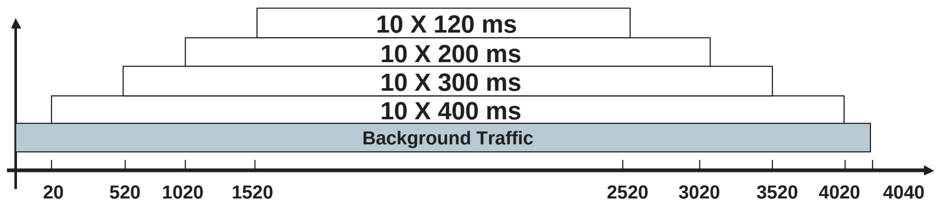
Figure 3.14: Throughput Trajectories of Flows Which Experience Rerouting

3.5.5 Dynamic Scenarios

In the following experiments, dumbbell topology is used, and the bottleneck link is configured as *delay*=50ms, *per*=10^{-6}, and *qsize*=0.5BDP. Flow arrival and departure sequences shown in figure 3.15 are used for evaluating these proposals as new flows arrived and old flows left. Two dynamic scenarios, door (figure 3.15(a)) and tower (figure 3.15(b)), are used for this purpose. There are 5 flows in door scenario, and different values are adopted by propagation delay of side links so that different flows may have different RTPDs. Tower scenario has 40 flows, and flows that belong to different groups also have different RTPDs. The results under door and tower scenarios are plotted in figure 3.16-3.18 and figure 3.19-3.21, respectively.



(a) Door



(b) Tower

Figure 3.15: Flows Arrival and Departure Sequence of Dynamic Scenarios

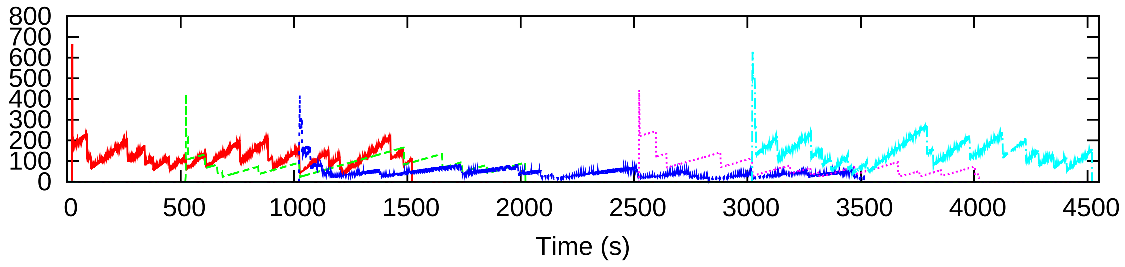
Figure 3.16 and figure 3.19 first show throughput trajectories of these competing flows under door and tower scenarios. They indicate that with the arrival and departure of flows, only Sync-TCP can drive active flows to converge to the new bandwidth allocation equilibrium rapidly and stably. In the equilibrium, Sync-TCP flows also receive the same throughput independent of the values of their RTPD.

Figure 3.17 and figure 3.20 show utilization ratio of the bottleneck link under door and tower scenarios, respectively. Queue dynamics of the bottleneck link under door and tower scenarios are also plotted in figure 3.18 and figure 3.21, respectively. It is obvious that Sync-TCP is the only one that can drive the network to operate around the knee in both door and tower scenarios. Hence, Sync-TCP could utilize the bottleneck link efficiently and maintain short queue length independent of the number of competing flows, the values of their RTPD, and the arrival and departure of these flows.

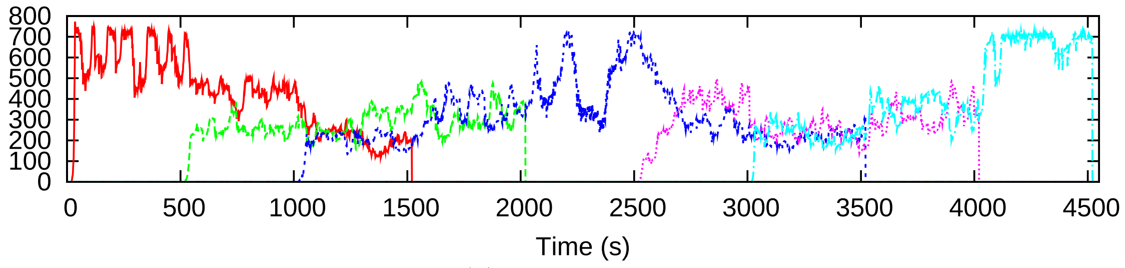
The above experiments with door and tower scenarios have also been repeated when the total load of cross traffic applications is varying. Simulation results are attached in Appendix A.2. These results indicate that Sync-TCP can maintain the above merits even when the load of cross traffic is varying.

Apart from the above experiments, many experiments, which use different parameter values $(bw, per, delay, qsize, N)$, different flow arrival and departure sequences, and different cross traffic loads, have also been carried out. They are not presented here due to the similarity of these results. But their results all support the argument that *Sync-TCP can drive long fat network pipes to operate around the knee and distribute the residual bandwidth fairly among competing flows, irrespective of the number of competing flows and the values of their RTPD.*

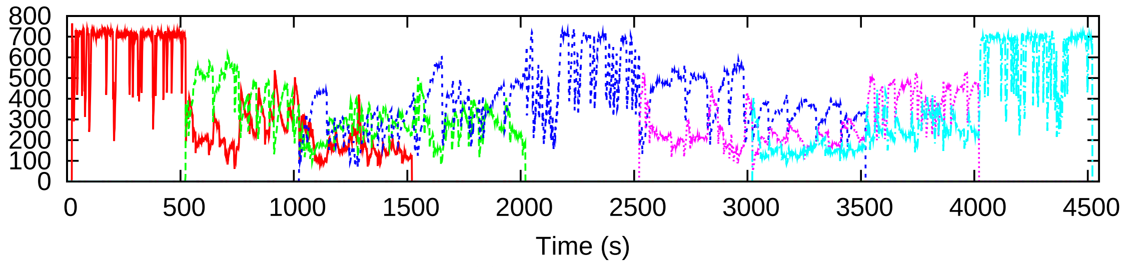
We have also studied MCL unfairness of Sync-TCP and its coexistence issues with the legacy TCP. These simulation results are attached in Appendix A.3 and A.4. In Appendix A.5, several simulation results are also attached to illustrate the necessary of letting λ react to network environment.



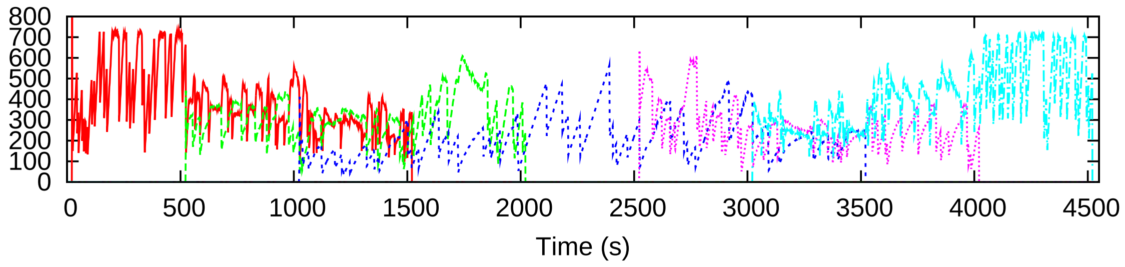
(a) TCP



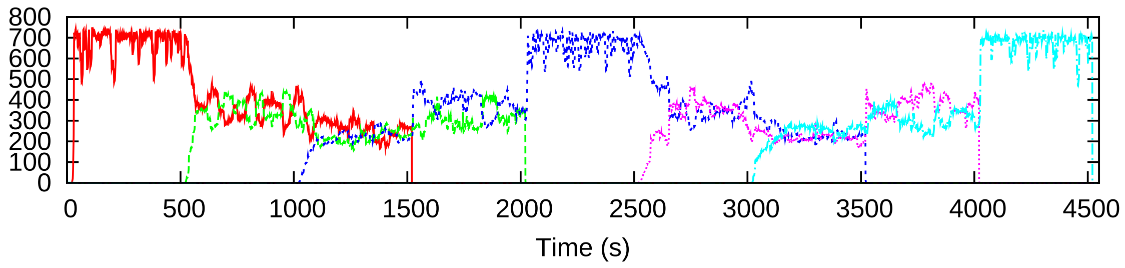
(b) CUBIC



(c) CTCP

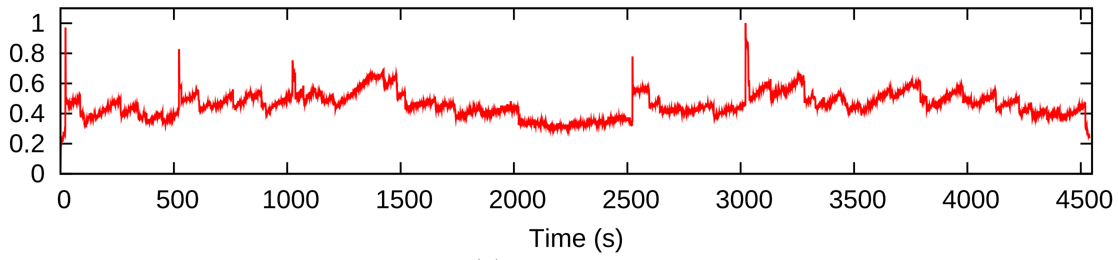


(d) FAST

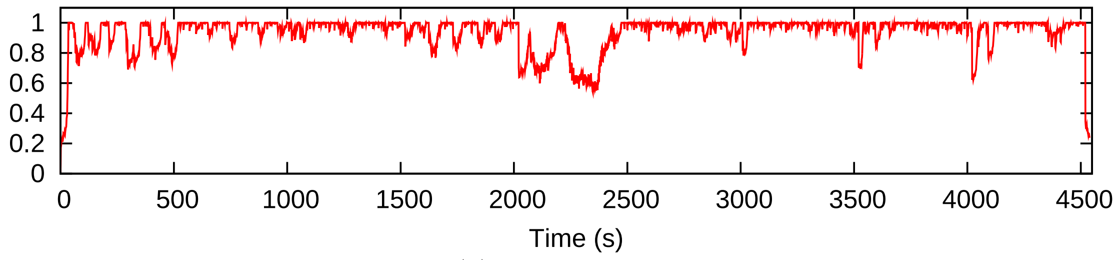


(e) SYNC

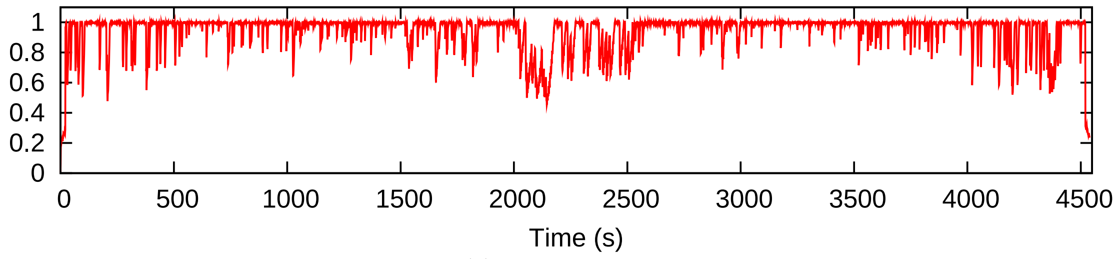
Figure 3.16: Door Scenario: Throughput Trajectories of All Competing Flows (Mbps)



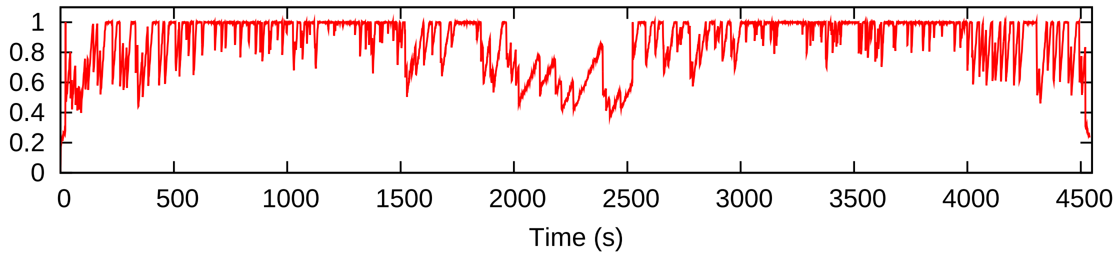
(a) TCP



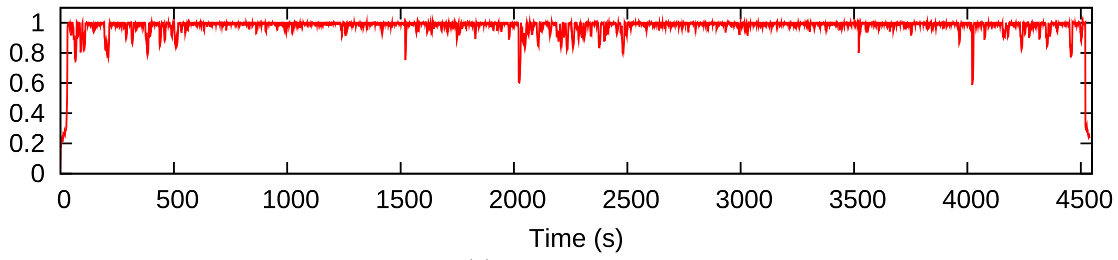
(b) CUBIC



(c) CTCP

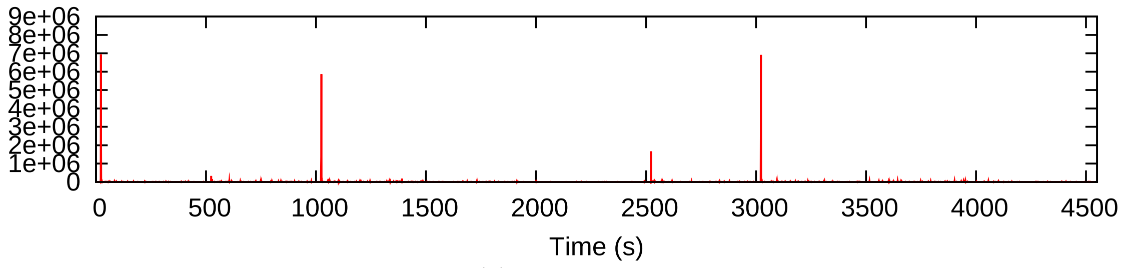


(d) FAST

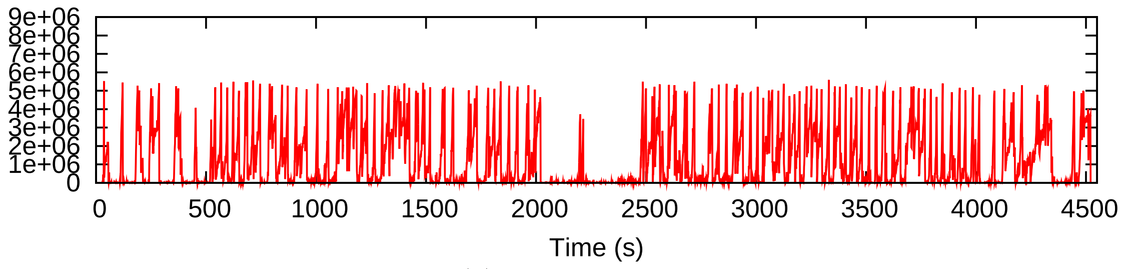


(e) SYNC

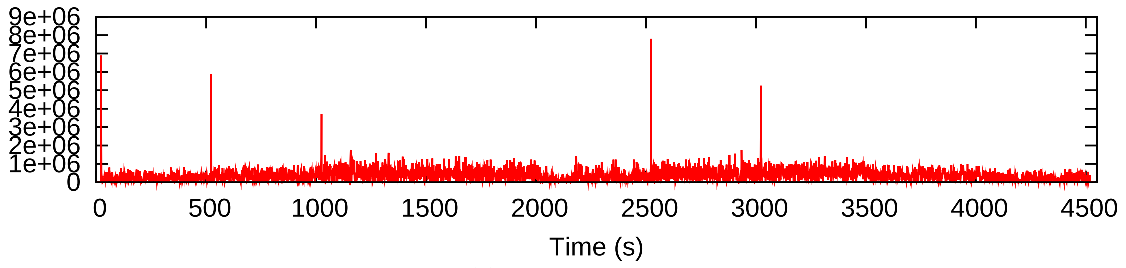
Figure 3.17: Door Scenario: Utilization Ratio of the Bottleneck Link



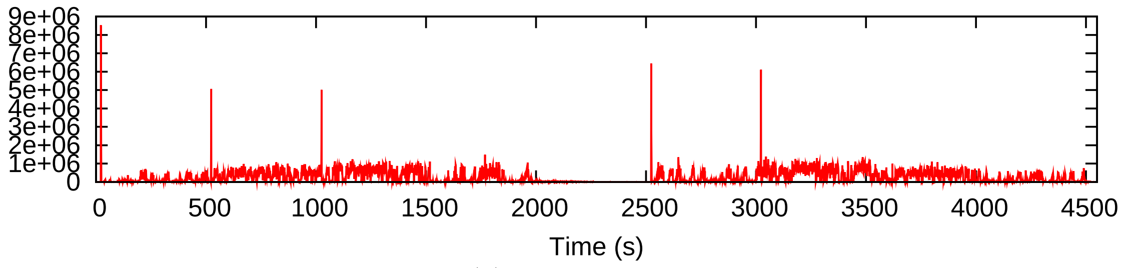
(a) TCP



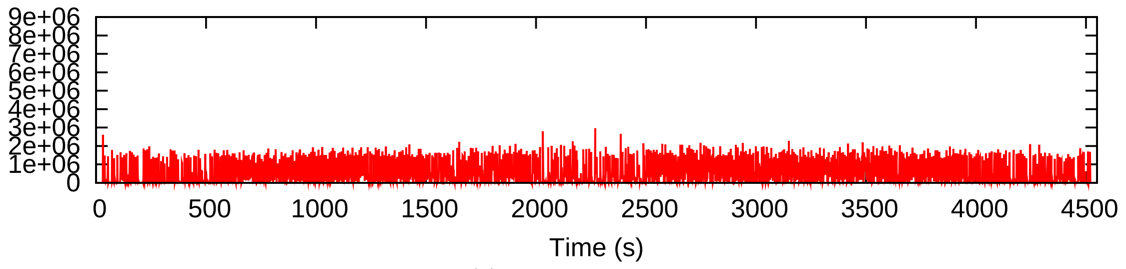
(b) CUBIC



(c) CTCP

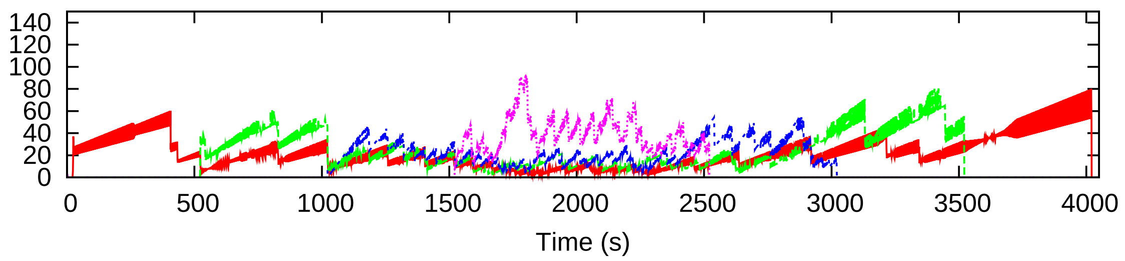


(d) FAST

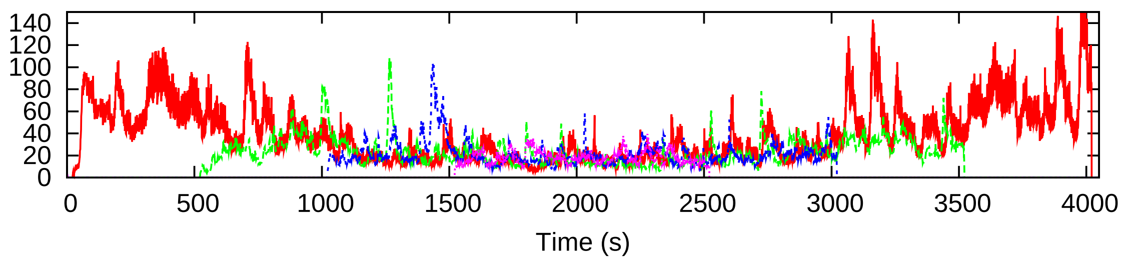


(e) SYNC

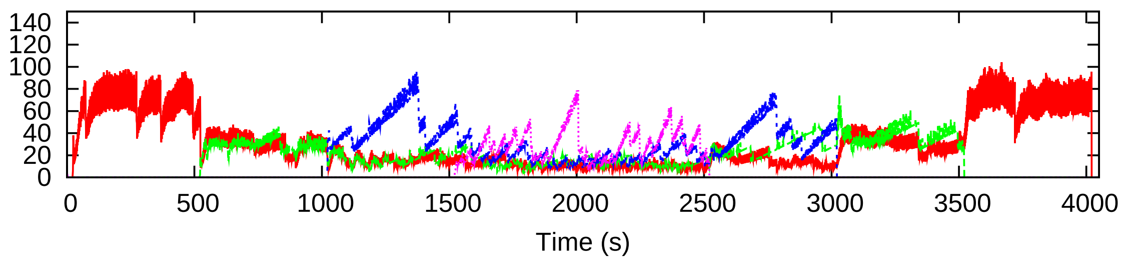
Figure 3.18: Door Scenario: Queue Dynamics at the Bottleneck Link (byte)



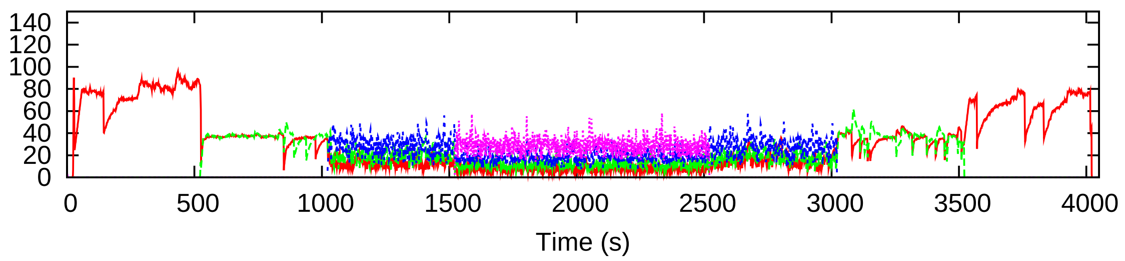
(a) TCP



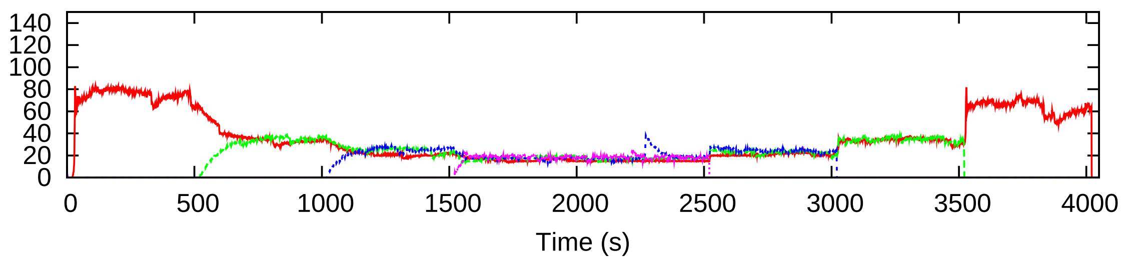
(b) CUBIC



(c) CTCP

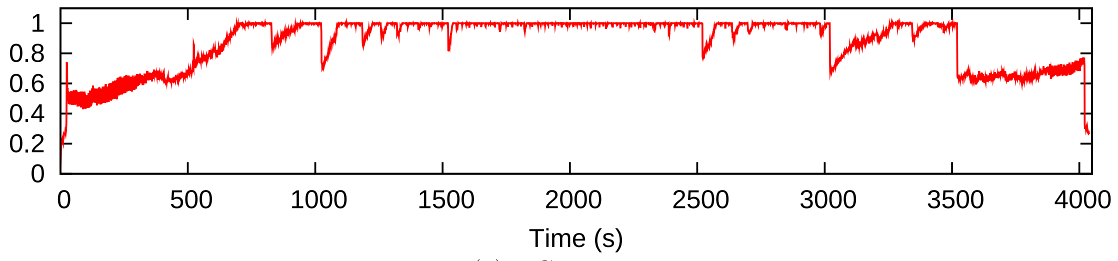


(d) FAST

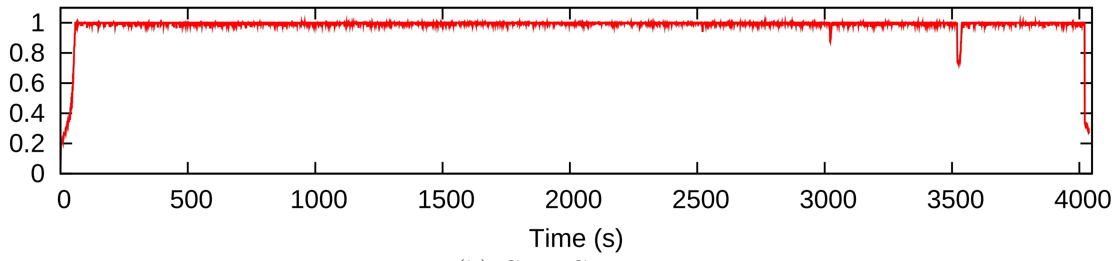


(e) SYNC

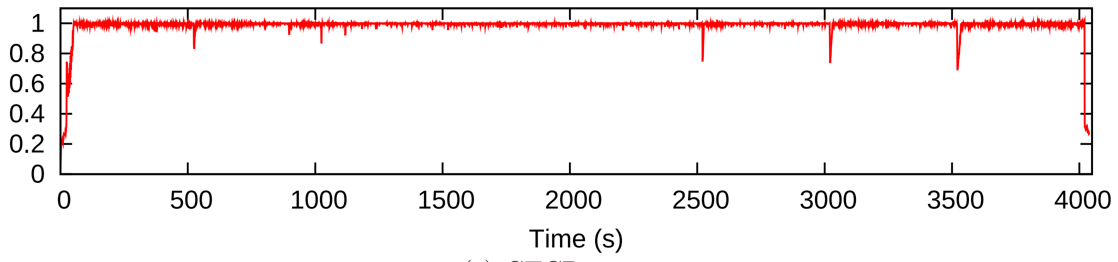
Figure 3.19: Tower Scenario: Throughput Trajectories of Flows 0, 10, 20, 30 (Mbps)



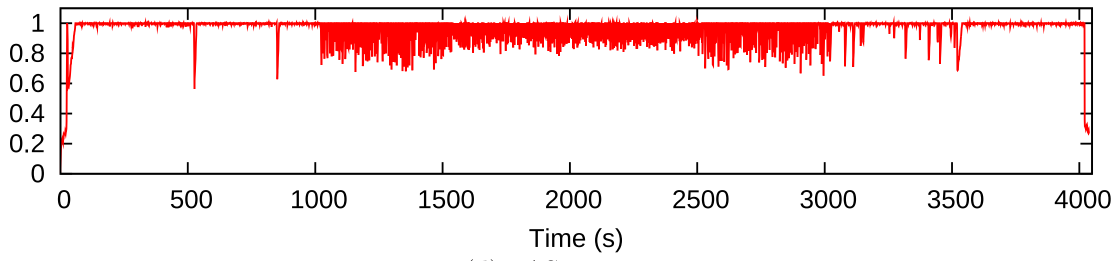
(a) TCP



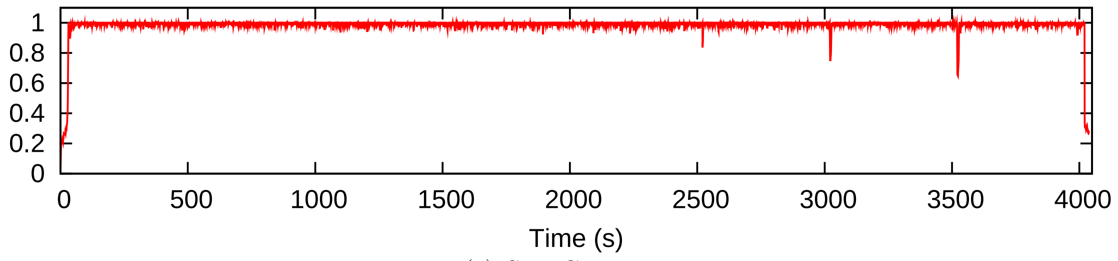
(b) CUBIC



(c) CTCP

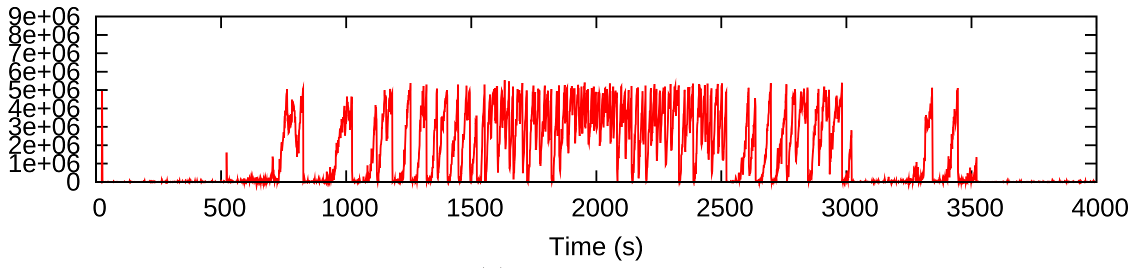


(d) FAST

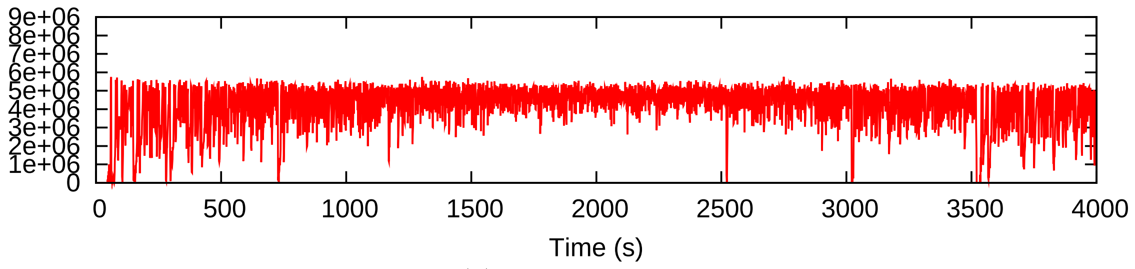


(e) SYNC

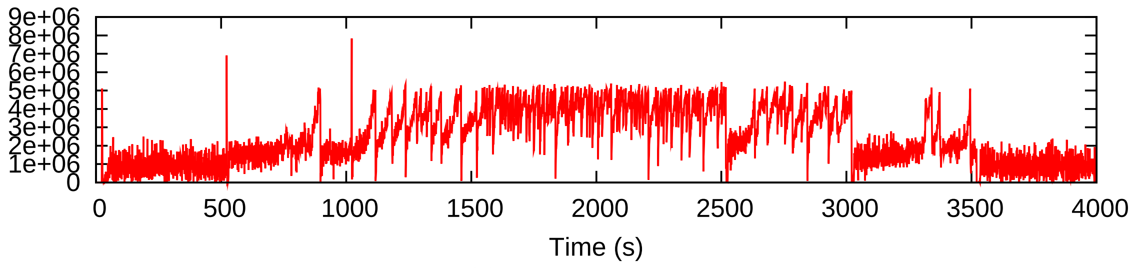
Figure 3.20: Tower Scenario: Utilization Ratio of the Bottleneck Link



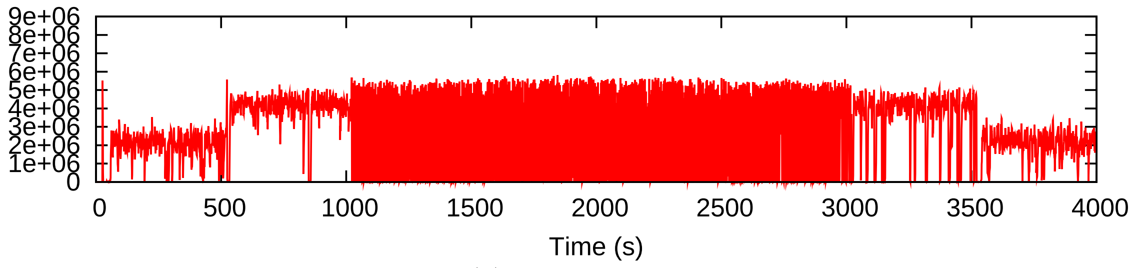
(a) TCP



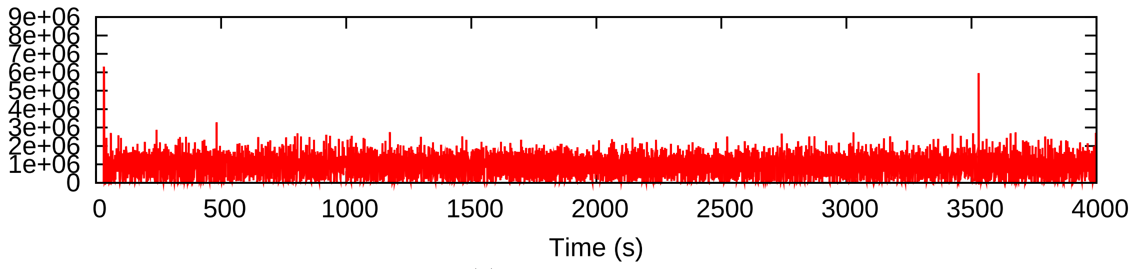
(b) CUBIC



(c) CTCP



(d) FAST



(e) SYNC

Figure 3.21: Tower Scenario: Queue Dynamics at the Bottleneck Link (byte)

3.6 Testbed Evaluation Results

In order to evaluate Sync-TCP in more realistic environments, Sync-TCP has been implemented in FreeBSD 7.1, and a high speed network testbed has also been set up. In this section, we first introduce configurations of the testbed. Experiment results are then presented and analyzed.

3.6.1 High Speed Network Testbed

Figure 3.22 illustrates the topology and configurations of our high speed network testbed. A HP Procurve 2900 switch, which is configured with two VLANs (Virtual LAN), is used to connect all computers. Three high end Dell PowerEdge T300 servers (SENDER, EMULATOR, and SINK), which are equipped with Myricom 10Gbps Ethernet cards, are connected to this switch's four 10Gbps ports.

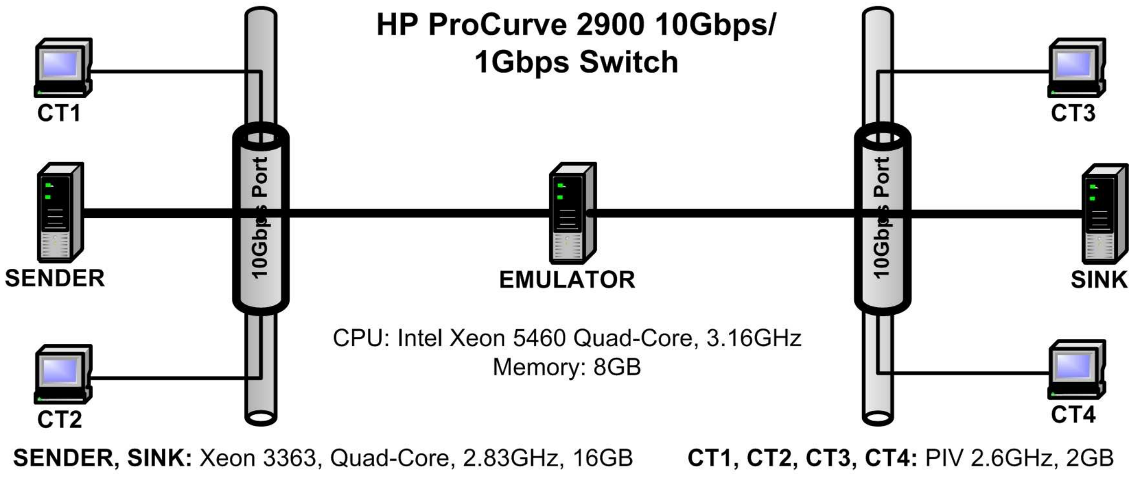

HP ProCurve 2900 10Gbps/ 1Gbps Switch
CT1
CT3
SENDER
10Gbps Port
EMULATOR
10Gbps Port
SINK
CPU: Intel Xeon 5460 Quad-Core, 3.16GHz
Memory: 8GB
CT2
CT4
SENDER, SINK: Xeon 3363, Quad-Core, 2.83GHz, 16GB
CT1, CT2, CT3, CT4: PIV 2.6GHz, 2GB

Figure 3.22: High Speed Network Testbed

SENDER is installed with FreeBSD 7.1, Fedora 9, and Windows Vista so that we can compare CTCP on Windows Vista, Cubic-TCP on Linux, and Sync-TCP on FreeBSD. Iperf [2] is used by SENDER and SINK to generate long-lived flows and collect statistics. In order to emulate the networks to be studied, EMULATOR is installed with FreeBSD and Dummynet [116], and the FreeBSD kernel is also rebuilt with a higher scheduling frequency for emulating a high speed network accurately. DropTail is emulated in all experiments. As for SINK, FreeBSD and Dummynet are installed. The Dummynet on SINK is used to emulate flows with different RTPDs.

Four lower end computers are connected to 1Gbps ports. They use D-ITG [32] to collect user experience of VoIP. Iperf is also used to emulate legacy FTP flows with a small socket buffer (64KB). These computers are also used to generate large amount of bursty traffic for emulating web-like traffic.

When setting up the following experiments, we make sure that there is enough memory for all flows and that the CPU is not the bottleneck. Hence, the number of flows per machine does not exceed 30, and the emulated bandwidth is not higher than 1Gbps even though the link rate is 10Gbps.

3.6.2 Synchronization of Congestion Signal

To verify whether Sync-TCP flows can detect congestion correctly in a testbed setting, three Sync-TCP flows with different RTTs (90ms, 100ms, and 110ms, respectively) are established between SENDER and SINK. EMULATOR emulates a bottleneck link whose *bw*=500Mbps, *delay*=25ms, *per*=10^{-6}, and *qsize*=0.5BDP. Web-like background traffic consumes about 40Mbps, and there are also a VoIP flow and a legacy FTP flow.

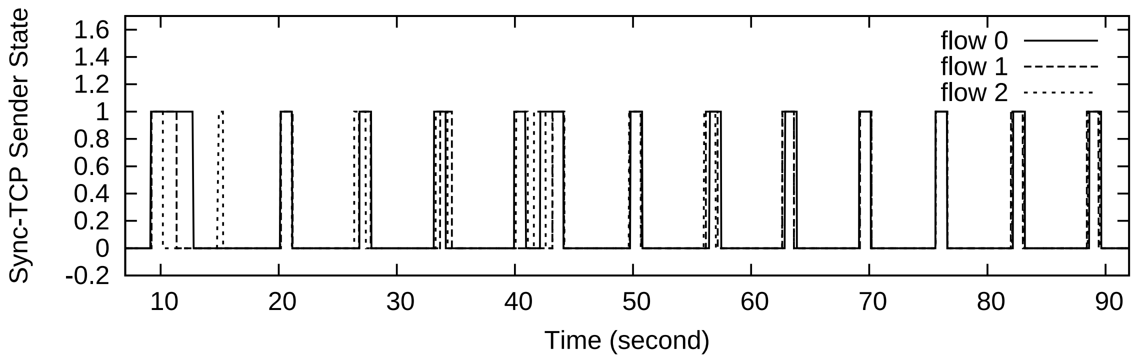
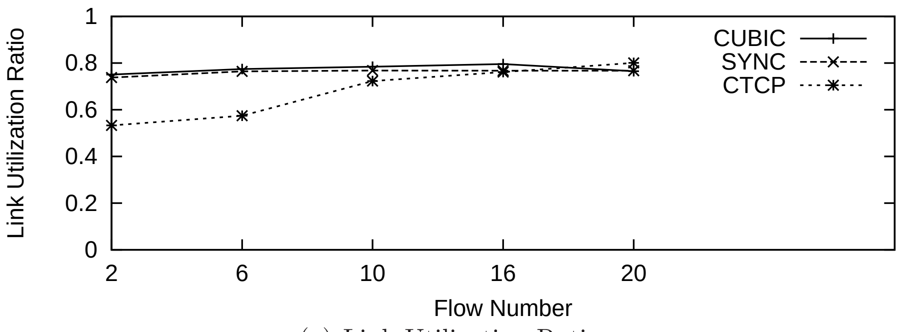


Figure 3.23: Synchronization of Congestion Detection through Queue Delay

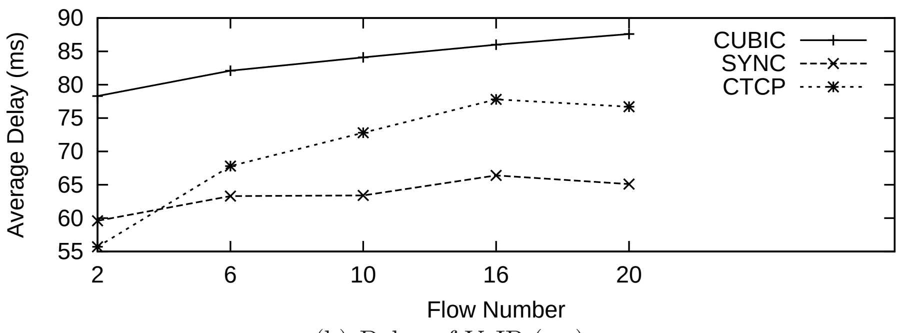
The length of this experiment is about 90 seconds. Figure 3.23 plots the changes of the three competing Sync-TCP flows' state with the time. "0" is used to represent *Probing* state, "1" is used to represent *Waiting* and *Emptying* states, and one pulse represents that a flow detects one congestion signal. Figure 3.23 indicates that except at the 15th and 42th seconds, all flows detect the same congestion signals. When background traffic is bursty and packet corruption is emulated (per=10^{-6}), a small amount of congestion misses are expected. According to log data at EMULATOR, the queue of the bottleneck link is indeed emptied periodically, allowing accurate queue delay measurements.

3.6.3 Flow Number Scalability

In this set of experiments, the configuration is bw=500Mbps, $delay$=25ms, per=10^{-6}, and $qsize$=0.5BDP. Web-like background traffic consumes about 100Mbps, and there are one legacy FTP flow and one VoIP flow. N, the number of competing HSCC flows, is set to 2, 6, 10, 16, and 20.



(a) Link Utilization Ratio



(b) Delay of VoIP (ms)

Figure 3.24: Scalability with Flow Number (Testbed Evaluation)

The duration of each experiment is 600 seconds. Figure 3.24 plots utilization ratio of the bottleneck link and user experience of VoIP. Link utilization ratio observed is lower than 1 due to Iperf measurement, header overhead, and Dummynet inaccuracy when emulating high speed networks. Based on these plots, we find that when N is small (2 or 6), CTCP cannot efficiently utilize the bottleneck link. A possible reason is that when per-flow throughput is high, CTCP is more likely to regard the noise in RTT samples as congestion signal. When N is large, CTCP can utilize the bottleneck link efficiently, but VoIP packets also experience longer delay. Cubic-TCP can efficiently utilize the bottleneck link in all cases. However, VoIP packets also always experience long delay, and the performance degrades with the increase of N. As for Sync-TCP, it can efficiently utilize the bottleneck link and keep the friendliness to cross traffic independent of the value of N.

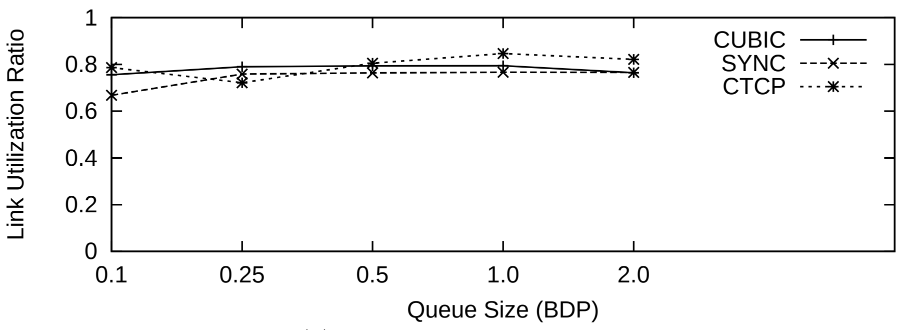
While the results are not shown here, we observed that when Cubic-TCP is used, throughput of legacy FTP drops by about 50%, compared to the case where either Sync-TCP or CTCP is used.

3.6.4 Effects of Buffer Sizes

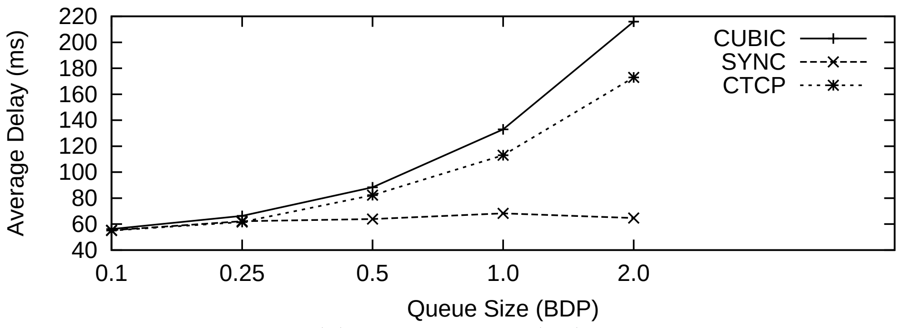
In this set of experiments, we vary the amount of buffer available on the router. The bottleneck link is emulated as bw=1Gbps, $delay$=25ms, per=10^{-6}, and $qsize$ varies from 0.1BDP to 2.0BDP. 30 flows, which are driven by CTCP, Cubic-TCP, or Sync-TCP, are established between SENDER and SINK. As for background traffic, web-like traffic consumes about 200Mbps and there are also a VoIP flow and 30 legacy FTP flows. Each experiment runs for 600 seconds.

Figure 3.25(a) plots the utilization ratio of the bottleneck link. It indicates that Sync-TCP will cause slightly lower utilization ratio when the queue size is smaller than the value expected by Sync-TCP ($C*$12ms). Please note that this value is independent of flow number.

Figure 3.25(b) plots the average delay experienced by VoIP packets, when queue size varies. It indicates that when queue size is large, the deployment of CTCP and Cubic-TCP



(a) Link Utilization Ratio



(b) Delay of VoIP (ms)

Figure 3.25: Effects of Different Buffer Sizes (Testbed Evaluation)

can severely degrade user experience of VoIP.

3.6.5 Dynamic Scenario

In this set of experiments, we evaluate whether Sync-TCP can work well with changes in HSCC flows and cross traffic. The bottleneck link is emulated as bw=1Gbps, $delay$=25ms, per=10^{-6}, and $qsize$=0.5BDP. Figure 3.26 shows the timing of traffic generation for these experiments.

Results from D-ITG and Iperf show that all three TCP variants can efficiently utilized the bottleneck link. Table 3.2 shows the VoIP user experience (delay and packet loss rate) when different HSCC TCP variants are adopted. It can be seen that Cubic-TCP has the highest delay and loss rate, follow by CTCP and finally Sync-TCP. Another way to interpret the delay measurement is to translate this delay into normalized average buffer occupancy. For example, when Sync-TCP is used, normalized average buffer occupancy is $\frac{61ms-RTPD}{MaximumQueuingDelay}$ =

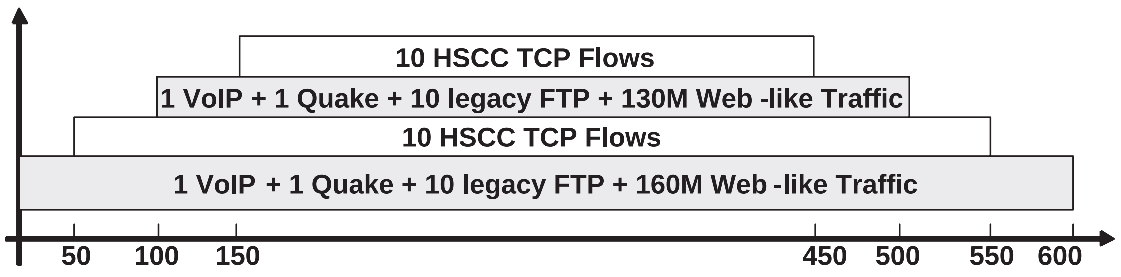

10 HSCC TCP Flows
1 VoIP + 1 Quake + 10 legacy FTP + 130M Web -like Traffic
10 HSCC TCP Flows
1 VoIP + 1 Quake + 10 legacy FTP + 160M Web -like Traffic
50
100
150
450
500
550
600

Figure 3.26: Flow Arrive and Leave Sequence of Dynamic Scenario (Testbed Evaluation)

44% since RTPD = 50ms and Maximum Queueing Delay is 25ms (queue size is 0.5BDP). This value can be more than 100% since there is imprecision in the scheduling interval and there can be buffering elsewhere. Interpreted this way, the impact of Cubic-TCP and CTCP on VoIP traffic can be very significant if large buffer is used.

VoIP Packets	Cubic-TCP	CTCP	Sync-TCP
Average Delay (ms)	86	75	61
Normalized Average Buffer Occupancy	144%	100%	44%
Packet Loss Rate	0.0075	0.0061	0.0031

Table 3.2: VoIP User Experience (Dynamic Scenario of Testbed Evaluation)

Figure 3.27 shows that Sync-TCP flows starting at different times can converge to the equilibrium point quickly, about 10s in this case. As for Cubic-TCP and CTCP, their results are much worse. In many cases, these HSCC flows do not converge to a steady state within the duration of the experiment.

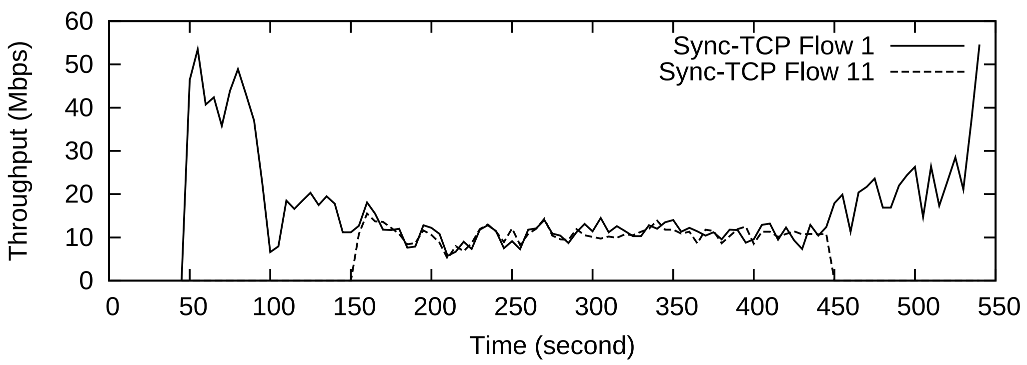


Figure 3.27: Throughput Trajectory of Two Sync-TCP Flows (Mbps)

3.6.6 Summary of Testbed Evaluations

For each of the above described experiments, we calculate the tuple consisting of (1) the ratio of wasted bandwidth (1 - link utilization ratio) and (2) the normalized average buffer occupancy. These tuples are plotted in Figure 3.28, each point corresponding to a testbed experiment. For a HSCC algorithm, the ideal performance is that bandwidth is not wasted and the buffering occupancy is minimum. Hence, the ideal performance is indicated by the position (0,0). Points closer to this location have better performance since they achieve good tradeoff between efficiency and friendliness.

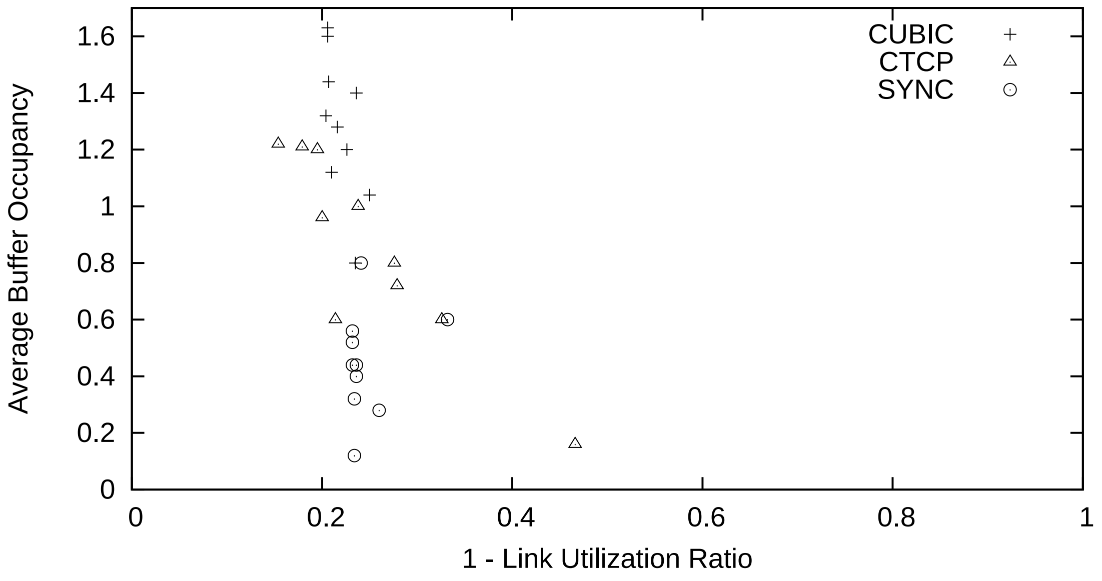


Figure 3.28: Tradeoff Between Efficiency and Friendliness

Figure 3.28 shows that Cubic-TCP always achieves very good efficiency. However, the cross traffic also always experience long queue delay and possible high packet loss rate. Hence, Cubic-TCP achieves high efficiency at the cost of hurting cross traffic (web surfing, VoIP, etc.). As for CTCP, there is quite a lot of variation in efficiency. When CTCP performs good in efficiency, buffer occupancy also becomes high.

Figure 3.28 also shows that Sync-TCP achieves the best tradeoff between efficiency and friendliness. Its performance is more reliable in terms of provide good link utilization while making sure that average delay is also low.

This section presents preliminary testbed evaluations of Sync-TCP. As for implementation details and additional experimental results, please refer to [129].

3.7 Related Work

Scalable TCP [80], a pure MIMD (Multiplicative Increase and Multiplicative Decrease) congestion control algorithm, was first proposed for long fat network pipes with large BDP. Since then, a lot of end-to-end HSCC algorithms have been proposed for distributing bandwidth more fairly and being more friendly to cross traffic. Highspeed TCP [51] adopts well-defined functions for its α and β with the aim to converge quickly and avoid large burst of segment loss. Bic-TCP [144] adopts a concave *cwnd* growth function for improving RTT fairness and reducing the number of packets dropped in one congestion event, and hence packet loss rate of cross traffic will be reduced. Cubic-TCP [59], an offspring of Bic-TCP, improves RTT fairness further by using a RTT-independent concave *cwnd* growth function. For reducing packet loss rate suffered by cross traffic, queue delay is exploited by TCP Illinois [94] whose α decreases with the increase of queue delay. This technique is also adopted by Sync-TCP for avoiding the case that queue delay is much larger than Th_{qd}. H-TCP [91] adopts AIMD for improving convergence and fairness. For improving RTT fairness, its additive increase factor, α, is designed as a function of the clock time. These techniques of H-TCP are followed by Sync-TCP. Sync-TCP also increases *cwnd* by α segments per T_{win} for improving RTT fairness further.

All the above proposals decrease *cwnd* only when segment loss is detected. Hence, the network needs to operate at the cliff sometimes and the cross traffic will unavoidably experience long queue delay. Based on this consideration, several delay-based HSCC algorithms are also proposed. As discussed in section 3.2.2, Fast TCP cannot drive the network to operate around the knee when there are many competing flows. In the metrics of friendliness, CTCP may be worse as it needs to act as loss-based TCP for ensuring that its throughput is not less than the legacy TCP. In this aspect, Yeah-TCP [20] and TCP Fusion [77] act like CTCP. Hence, they are not designed to drive the network to operate around the knee too. Although Delay-based AIMD [48] tries to drive the network to operate around the knee independent of the number of competing flows, it does not carefully consider how to empty the queue of the bottleneck link so that RTPD can be estimated correctly. Rerouting issue and the effects of cross traffic are not discussed too.

Except the above end-to-end HSCC algorithms, some mechanisms [68][79], that need supports from intermediate routers, have also been proposed for long fat network pipes.

3.8 Summary and Future Work

In this chapter, Sync-TCP, a new delay-based high speed congestion control algorithm, is proposed for safely ramping up the throughput of bandwidth-greedy and elastic applications on long fat network pipes of the Internet. Sync-TCP is designed to drive these network pipes to operate around the knee and distribute the residual bandwidth fairly among competing flows even when the number of competing flows varies and their RTPDs differ significantly. Extensive simulations and preliminary testbed evaluations indicate that Sync-TCP does enable bandwidth-greedy and elastic applications to utilize long fat network pipes efficiently and fairly without hurting the cross traffic applications, especially the interactive ones.

In the next step, more testbed evaluations and live experiments in the Internet will be carried out for evaluating Sync-TCP thoroughly under more realistic environments. The values of system parameters will also be investigated for tuning the performance of Sync-TCP. More information about Sync-TCP can be found at `http://cir.nus.edu.sg/synctcp/`.

Chapter 4

TCP KentRidge: A New TCP Framework for the Heterogeneous and Evolving Internet

4.1 Introduction

With the deployment of different communication technologies, the Internet has become a highly heterogeneous inter-network, and it is still evolving continuously. Such an inter-network not only brings many challenges to TCP protocol, it also affects how TCP should be implemented, especially the congestion control mechanism.

Over these years, it has been a hot research topic to improve TCP performance in the heterogeneous Internet and a large number of TCP adaptations have been proposed for different kinds of networks, such as LFN (Long Fat Network) [69], LTN (Long Thin Network) [104], Slow Link [45], Lossy Link [46], Asymmetric Link [22], Satellite Link [16], MANET (Mobile Ad hoc Network) [60], etc. However, very few of them have been implemented in the popular operating systems. The classical TCP implementation still uses the same standardized congestion control mechanism for all connections and cannot achieve good performance under many scenarios of the heterogeneous Internet. In this chapter, we will focus on how to implement TCP so that it can work well in the heterogeneous Internet and can easily evolve with the changing Internet. More specifically, a new TCP implementation framework will be designed so that a large number of TCP adaptations can be easily implemented and a connection could have the potential of learning its current environment and automatically selecting the most appropriate TCP adaptation to be used.

This chapter is organized as follow. In section 4.2, we first summarize the existing TCP adaptations, that have been proposed for different kinds of network pipes and may be utilized by TCP KentRidge. The challenges faced by a TCP implementation in the heterogeneous and evolving Internet are then discussed in section 4.3. The state of the art TCP implementations are also discussed in section 4.4. After that, section 4.5 presents the details of TCP KentRidge, which is designed to provide modular congestion control, per-connection congestion control configurability, and the capability of automatically and intelligently changing TCP adaptation used by a connection according to its current environment. TCP KentRidge implementation status in FreeBSD 7 is also introduced in section 4.6. Finally, section 4.7 summarizes the work that have been done and points out the necessary future works.

4.2 TCP in the Heterogeneous Internet

When segment loss is detected mainly through 3DUPACK, the standardized TCP congestion control versions (Reno, SACK, and Newreno) are very simple and can be characterized as AIMD(1, 0.5). Except these variants, many other congestion control mechanisms have also been proposed for TCP.

TCP Westwood [40][131] tries to measure available bandwidth by observing the returning rate of ACK packets. When congestion is detected, TCP Westwood sets *ssthresh* based on the measured bandwidth and the smallest observed RTT sample. In-path capacity estimation has also been adopted by PCP [19] and RAPID [82], and they directly adjust the sending rate based on the estimated bandwidth.

Apart from segment loss, other congestion signals have also been utilized by some proposals. Based on the network performance model as a function of network load [73], many delay-based congestion control algorithms have been proposed and they have been discussed in chapter 3. Not only end-to-end congestion control mechanisms based on Droptail queue, some congestion control mechanisms, such as TCP ECN [114] that is designed based on active queue disciplines (RED [54], etc.), have been proposed too.

Nowadays, many different links with different characteristics (bandwidth, delay, packet error rate, etc.) are attached to the Internet and these links pose different challenges to TCP. Hence, a large number of TCP adaptations have been proposed for some specific kind of network, and these proposals are summarized in the following subsections. Before that, the metrics of a network path and the ways that they affect TCP performance are first discussed in the following list.

- Available Bandwidth: the smallest available bandwidth among all links of a specific network path. It is also the highest throughput that can be achieved by the sender. Available bandwidth normally keeps changing due to cross-traffic and other reasons. The ultimate goal of TCP congestion control is to keep the sending rate as close to the current available bandwidth as possible.

- RTT (Round Trip Time) and Jitter (RTT Variance): RTT is the sum of propagation delay of all links, queue & process delay of all routers and the two endpoints. RTT determines the speed of *capacity probing*. RTT samples may also signal network congestion through queue delay. Jitter is the variance of RTT samples. RTT may change due to many reasons, such as changes of queue delay, re-route, link layer retransmission, etc. Jitter and the smooth average of RTT samples determine the value of *rto*, which is used by TCP retransmission timer. Hence, they affect the speed that the TCP sender responds to network congestion. In addition, large jitter may also cause spurious *timeout*.

- PLR (Packet Loss Rate): the probability that a packet is dropped at any router (congestion) or corrupted on any link (transmission error) of a network path. PLR affects TCP performance since TCP regards segment loss as the signal of network congestion.

  According to TCP throughput model [101][108], RTT, PLR, *rto*, segment size, and the number of segments acknowledged by each ACK determine the throughput of a TCP flow. Hence, the networks, that have extreme values of RTT, Jitter, and PLR, should be investigated.

- BDP (Bandwidth Delay Product): $AvailableBandwidth * RTT$. BDP is the optimal value of *cwnd* and networks with extreme BDP values should be investigated.

- Packet Reordering: the phenomenon that the receiving order of packets is different with the sending order. Packet Reordering is not a rare event in the Internet. Different segments may use different paths of IP networks, some routers may reorder packets for optimization, and some networks' link layer protocols may not support in-order data transmission. The reordering of ACK packets interrupts TCP's self-clock mechanism and causes large data burst at TCP sender. When TCP segments is reordered slightly, TCP receiver sends a NEWACK after one or two DUPACK. Thus, data transmission also becomes a little more bursty. When the reordering of TCP segments is larger than three, *spurious fast retransmission* is triggered and *cwnd* is reduced unnecessarily.

- Path Asymmetry: the large difference of network path characteristics between the two directions of a network path. The commonest asymmetry is bandwidth asymmetry of access link, such as ADSL and GPRS. In this case, the downlink cannot be efficiently utilized because the uplink does not have enough bandwidth to transmit all ACKs generated by the receiver [23].

Many networks may have extreme values for several of these metrics. For example, GPRS has low available bandwidth, long RTT, large jitter, high PLR, and asymmetric bandwidth.

Due to the large number of combinations, the following paragraphs will summarize the existing solutions according to the extreme values of each metric.

TCP Adaptations for Large BDP

BDP of long fat networks, such as optical and satellite networks, can be very large. TCP faces several problems in these networks since TCP congestion control was originally designed for links with low/medium bandwidth.

Firstly, TCP sender cannot efficiently utilize the abundant bandwidth due to its AIMD algorithm with fixed parameters. Many HSCC algorithms have been proposed for solve this issue and these proposals have been discussed in chapter 3.

Secondly, with large BDP, *ssthresh* will also be very large. The exponential increase at the end of *SS* phase will cause the loss of many segments. When *cwnd* is larger than some threshold, Limited Slow-Start [52] slows down the exponential increase algorithm used in *SS* state.

TCP Adaptations for Small BDP

In the current Internet, there are still some networks with small BDP, such as dial-up and GPRS. Due to the large initial value of *ssthresh*, multiple segments will be dropped at the end of the first *slow start* phase. This problem is called *slow start overshoot*. In [40][66][140], the authors propose to set *ssthresh* according to the BDP that may be measured by the sender or reported from the receiver. When BDP is small and a segment is lost, there may not be enough segments to trigger the receiver to send three consecutive DUPACK packets. Limited Transmit [14] lets TCP sender send out a new segment when the first or the second DUPACK packet is received. Consequently, TCP receiver will transmit more ACK packets and the sender is more likely to detect segment loss through 3DUPACK.

TCP Adaptations for Estimable Available Bandwidth

Available bandwidth may be fixed or estimable, especially when the access link (dial-up and ADSL, etc.) is slow and becomes the bottleneck link. With the fixed or estimable available bandwidth, TCP sender can just set its *cwnd* according to available bandwidth and RTT. In [140], the authors propose to set the maximum value of *cwnd* according to the last link's available bandwidth and RTT.

The estimable available bandwidth may also change frequently. For example, available bandwidth of a Wi-Fi node may change due to link quality, contention, and handoff. But the available bandwidth still can be estimated by some mechanisms [88][139]. Many solutions have been proposed for this kind of networks and they have been discussed in chapter 2.

TCP Adaptations for Long RTT

In *CA* state, *cwnd* is increased by one segment per RTT. This is unfair to connections with long RTT. In [50], the authors suggest to change TCP's AIMD algorithm so that all senders increase their sending rate with the same speed, irrespective of their RTT values. Based on this observation, some algorithms have been designed in [38][64].

TCP Adaptations for Large Jitter

When RTT changes abruptly, *spurious timeouts* may occur [58]. To solve this problem, Conservative RTO [58] and Delay Injection [126] propose to generate large *rto* value for avoiding *spurious timeout*. But these methods will slow down TCP response to the real network congestion. DSACK [55] and Eifel [97] try to detect *spurious timeout* and undo the unnecessary *cwnd* reduction. In [41][42], the authors try to hide large jitter and bandwidth variance of 3G networks by regulating the sending rate of ACK packets or changing *awnd* (advertised window) of ACK packets at the base station.

TCP Adaptations for Packet Reordering

When the reordering of TCP segments is larger than three, *spurious fast retransmission* will occur. In this case, DSACK [55] and Eifel algorithm [97] can also be used to detect *spurious fast retransmission* and undo the unnecessary *cwnd* reduction. In [30][98], the authors propose to adjust (blindly or according to network environment) the threshold used by the sender to detect congestion through consecutive DUPACK packets. In addition, when the ACK packets are reordered, they will cause large data burst and traffic shaping at TCP sender should be helpful.

TCP Adaptations for High Packet Loss Rate

When ACK packets are lost, data burst may occur and traffic shaping should be used at the sender. Furthermore, the loss of ACK slows down *capacity probing* since TCP sender increases *cwnd* according to the number of received ACKs. To solve this issue, TCP Byte Counting [18], which increases *cwnd* according to the number of bytes acknowledged by an incoming ACK, has been proposed.

When many segments are corrupted over some lossy links, TCP sender will reduce its *cwnd* unnecessarily. Small *mss* could be used to reduce the packet loss rate due to transmission error. Segment corruption can also be hidden from TCP through PEP (Performance Enhancement Proxy) proposals [21][24][31] and/or link layer proposals, such as Link Layer Automatic Retransmission reQuest and Forward Error Correction [8][11].

There are many mechanisms that try to differentiate corruption and congestion according to the correlation between the increase of RTT and congestive segment loss [28][56][109]. Furthermore, explicit transport error notification is also adopted by TCP HACK [117] and some other mechanisms [25][83][140]. However, when the server runs these mechanisms, the clients can acquire more bandwidth through cheating [86].

TCP Adaptations for Bandwidth Asymmetry

If the bandwidth difference between two directions is very large, bandwidth of the downlink cannot be efficiently utilized because the uplink cannot transmit ACK packets generated by the receiver [23]. Large *mss* can be used to reduce the number of ACK packets. ACK Congestion Control have also been proposed to solve this problem [23].

In summary, many TCP adaptations have been proposed for different kinds of networks. Through classifying them based on the type of network pipe that they are proposed for, we can draw some guidelines on how to select the most appropriate TCP adaptation for a connection based on its environment. In the following section, we will discuss how TCP should be implemented in the heterogeneous and evolving Internet.

4.3 TCP Implementation in the Heterogeneous and Evolving Internet

As the deployment of various communication technologies, the Internet has become a highly heterogeneous inter-network composed of networks with varying characteristics (bandwidth, delay, etc.), such as optical network, satellite network, ADSL, Broadband over Power Line, Wi-Fi, WCDMA, etc. In addition, many hosts with different resources (CPU, memory, power supply, etc.), such as Mainframe and smart phone, have been attached to the Internet. Different operating systems (Windows, Linux, etc.) with their own TCP implementations are installed on these hosts, and applications (FTP, WWW, etc.) with different expectations are also running on these hosts. Figure 1.4 in chapter 1 illustrated the heterogeneity of the Internet in part. In the following paragraphs, we will investigate how the heterogeneous and evolving Internet affects the way of implementing TCP.

1. Firstly, a host needs to communicate with other hosts that spread across the globe.

For example, a server needs to handle clients that come from different places through various access networks. A client also needs to access many servers located at different places. With the availability of Peer to Peer applications, clients need to communicate with each other too. Consequently, within the heterogeneous Internet, different connections driven by the same TCP implementation of a host need to run on network pipes with different characteristics. Even for the same connection, its network pipe may also have varying characteristics at different times due to mobility. Especially, with multiple network interface cards and mobile IP [36], a TCP connection may run on different kinds of network pipes during its life time. In summary, TCP implementation of a host needs to face challenges brought by different network path characteristics.

In addition, routers in the Internet may also have different queue sizes and adopt different queue management schemes, such as DropTail and Active Queue Management with ECN (explicit congestion notification) [114]. In another word, they may provide different kinds of congestion signals, such as segment loss, queue delay, and ECN bit. Hence, TCP implementation of a host needs to handle different network feedback provided by these routers.

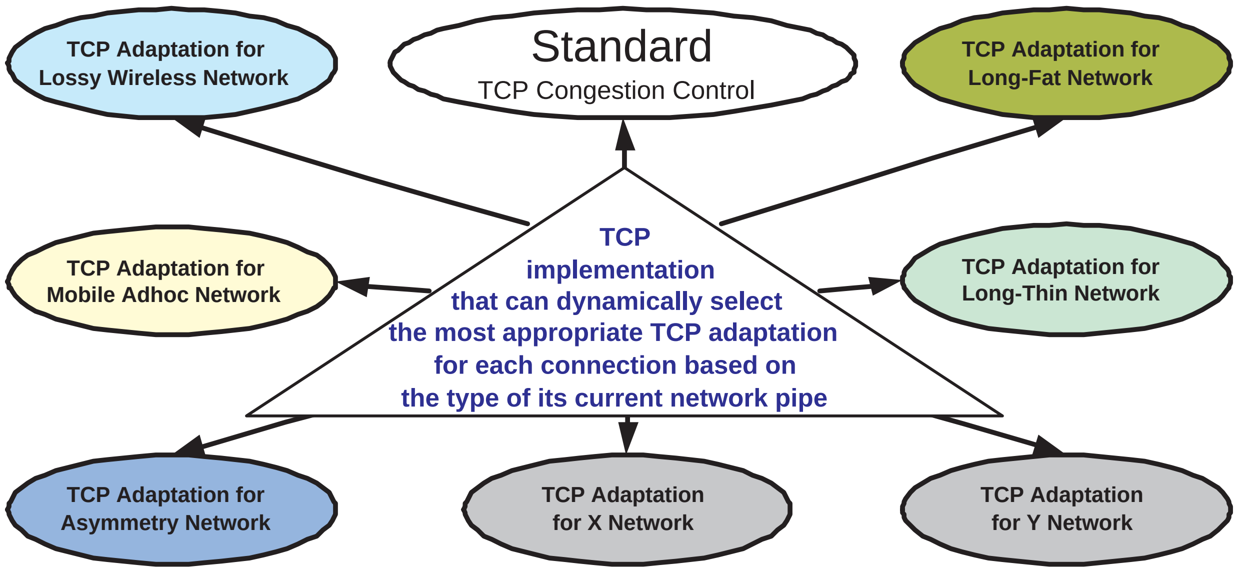

TCP Adaptation for Lossy Wireless Network
Standard
TCP Congestion Control
TCP Adaptation for Long-Fat Network
TCP Adaptation for Mobile Adhoc Network
TCP implementation that can dynamically select the most appropriate TCP adaptation for each connection based on the type of its current network pipe
TCP Adaptation for Long-Thin Network
TCP Adaptation for Asymmetry Network
TCP Adaptation for X Network
TCP Adaptation for Y Network

Figure 4.1: An Ideal TCP Implementation for a Highly Heterogeneous Internetwork

Based on the above observations, it is unlikely that a TCP implementation with only one congestion control mechanism could always work well in the Internet. In order to support diverse environments, it is now worthwhile to bring more TCP adaptations and more intelligence into TCP implementation. More specifically, TCP implementation should keep learning current characteristics of the network pipe used by each connection and select the most appropriate TCP adaptation accordingly. With such an ideal TCP implementation (shown in figure 4.1), a user could surf the heterogeneous Internet with more fulfilling experience.

2. Secondly, hosts of the Internet are installed with different operating systems, whose TCP codes may have different capabilities. Hence, TCP implementation of a host needs to talk with different TCP versions. In addition, TCP implementation of a host normally serves applications with different expectations (short delay, high throughput, smoothness, etc.). Furthermore, a TCP implementation may run on computing devices with different constraints (CPU, memory, battery, etc.). These heterogeneity should also be systematically handled by an ideal TCP implementation.

3. Finally, but very importantly, the Internet is still changing continuously in the aspects of network infrastructure, communication technologies, network applications, etc. An ideal TCP implementation should be modularized for better maintainability and extensibility. More specifically, it should be carefully designed so that new TCP adaptations and their corresponding intelligence can be added with ease.

4.4 State of the Art TCP Implementations

As a part of network protocol stack, TCP is normally implemented in the kernel of operating systems for efficiency and security. In this section, TCP implementations in three prominent operating systems (FreeBSD, Linux, and Windows) are described and discussed, particularly with focus on their ways of implementing congestion control.

4.4.1 FreeBSD

Source codes of many Unix-based operating systems, including FreeBSD, originate from BSD Unix. Hence, BSD's way of implementing TCP is also inherited by these operating systems.

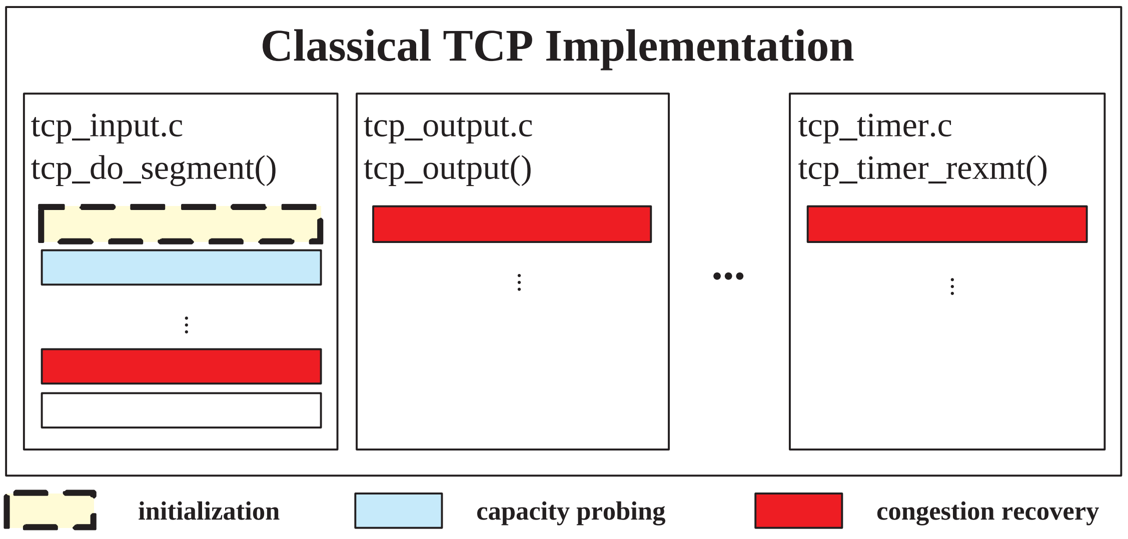


Figure 4.2: Classical TCP Implementation of BSD-like Unix Operating Systems

In FreeBSD 7.1, congestion control is highly interleaved with other TCP codes which are maze-like complex. The source codes related with congestion control are spread across several files and tens of functions. As illustrated in figure 4.2, FreeBSD 7.1 TCP codes, that reduce *cwnd*, appear in *tcp_timer_rexmt()*, *tcp_do_segment()*, and *tcp_output()*, etc. With such an implementation, it is inconvenient and difficult to change the existing codes. With the addition of congestion control related TCP adaptations and other features, the source codes tend to become more complex and unwieldy.

Except for TCP Reno [17], several RFCs in standards track, such as TCP Newreno [53], TCP SACK [100], Large Initial Window [15], and Limited Transmit [14], have been implemented in FreeBSD 7.1. These mechanisms can be enabled/disabled, but through global variables. Hence, the same set of standardized mechanisms will be used by all active connections of a host installed with FreeBSD 7.1.

4.4.2 Linux

In Linux, congestion control is now implemented in a framework, that is similar to the *backplane-slot* framework proposed in ADAPTIVE [118]. More specifically, congestion control has been abstracted and modularized for maintainability, extensibility, and configurability. Core functions of congestion control are implemented separately and are called through function pointers at corresponding places (*slots*) of the remaining TCP codes (*backplane*).

In Linux 2.6.26, more than ten TCP adaptations have been implemented according to this framework and Cubic-TCP is now the default congestion control mechanism. In the per-connection TCP control block, a new variable is defined to specify the TCP adaptation used by a connection. A new socket option has also been provided for facilitating per-connection congestion control configuration.

Although Linux has modularized congestion control, provided per-connection configuration, and implemented many TCP adaptations, it is still not good enough for the Internet. Firstly, the *slots* in Linux focus on how to change *cwnd* & *ssthresh*. They are not general enough to implement TCP adaptations that use some new TCP options [117][143]. Secondly, Linux does not consider how to learn the environment of a connection and how to dynamically select the most appropriate TCP adaptation for this connection. Considering that application developers and end users may not know congestion control very well, it is too risky to empower them to determine the TCP adaptations used in the Internet.

4.4.3 Windows

Except standard TCP congestion control, Compound TCP [125] has also been implemented in Windows (Vista and Windows Server 2008), and the user can switch between standard TCP and Compound TCP [125] by changing a global variable. Since source codes are not available, it is impossible to know how congestion control is implemented in Windows. Anyway, the same kind of congestion control, standard TCP congestion control or Compound TCP, will be used by all active connections of a host installed with Windows.

In summary, none of these implementations can solve all challenges brought by the Internet, which are discussed in section 4.3. It is now necessary and worthwhile to reengineer TCP implementation for handling these challenges systematically.

4.5 Design of TCP KentRidge

In this section, we present the details of our proposal, TCP KentRidge. TCP KentRidge is a new TCP framework designed to provide modular congestion control, per-connection congestion control configurability, and the potential to automatically and intelligently change TCP adaptation used by a connection according to its environment. This section first introduces the architecture of TCP KentRidge, and the details of its two important components (DC-TCP and *Network Pipe Classification*) are then presented.

4.5.1 The Architecture

Figure 4.3 depicts the architecture of TCP KentRidge comprising of four components, *Knowledge Base*, DC-TCP, *Network Pipe Classification*, and *Intelligent Agent*.

Knowledge Base includes the knowledge of all TCP adaptations supported by TCP KentRidge. For each TCP adaptation, it holds the kind of network pipe that this adaptation is proposed for, the capabilities that this adaptation depends on, and the advantages & shortcomings of this adaptation. For example, the network pipe, that Sync-TCP [142] is proposed for, is long fat network with large queue; the advantages of Sync-TCP are in-protocol fairness and friendliness to cross traffic; and its shortcomings are MCL unfairness and the starvation by loss-based TCP adaptations. The survey of the existing TCP adaptations presented in section 4.2 should be helpful in building *Knowledge Base*.

DC-TCP (Dynamically Configurable TCP Framework) is a framework that supports per-connection configuration and enables the host to dynamically change the TCP adaptation used by a connection. Following a *backplane-slots* framework, congestion control in DC-

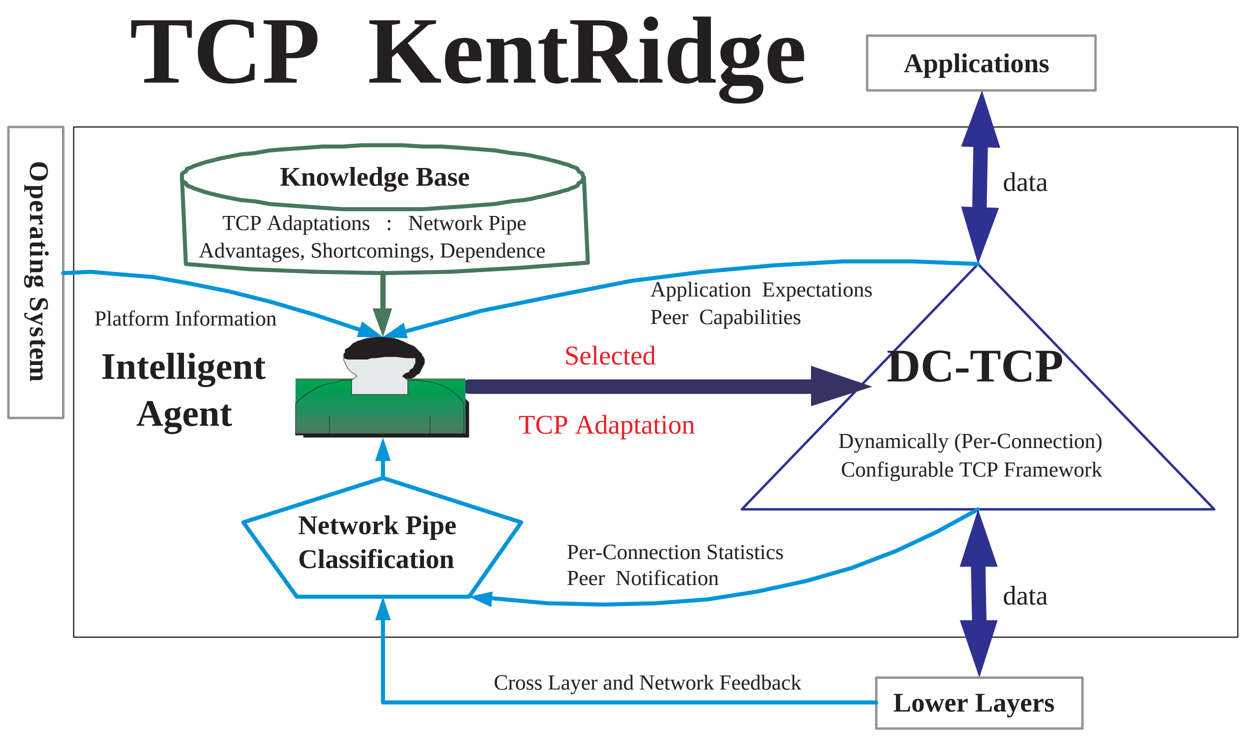

TCP KentRidge
Applications
Operating System
Knowledge Base
TCP Adaptations : Network Pipe
Advantages, Shortcomings, Dependence
data
Platform Information
Application Expectations
Peer Capabilities
Intelligent Agent
Selected
DC-TCP
TCP Adaptation
Dynamically (Per-Connection) Configurable TCP Framework
Network Pipe Classification
Per-Connection Statistics
Peer Notification
data
Cross Layer and Network Feedback
Lower Layers

Figure 4.3: TCP KentRidge: the Architecture

TCP is modularized and abstracted to improve maintainability & extensibility and support dynamically switching among TCP adaptations. In addition, DC-TCP collects the environmental variables of a connection, such as application's expectations and peer's capabilities. Statistics of the network pipe, such as packet loss rate and the achieved throughput, are also maintained through passive observing TCP data flow. Furthermore, DC-TCP handles the explicit notification from the peer and triggers *Network Pipe Classification* if needed.

Network Pipe Classification is responsible to determine the type of the current network pipe used by a connection. It makes the decision mainly based on network path characteristics passively observed by DC-TCP. In some trustworthy environments, explicit notification from the peer, the network, or lower layers, may also be used. If it finds out that the type of the current network pipe has changed, *Intelligent Agent* is triggered immediately.

Intelligent Agent chooses TCP adaptation for a connection based on its environment. The environment variables include the type of the current network pipe, peer's capabilities, application's expectations, and platform information. Based on these variables, *Intelligent Agent* queries *Knowledge Base* to get the most appropriate TCP adaptation and instructs DC-TCP to use the selected adaptation when processing the future events of this connection.

4.5.2 DC-TCP: The Workhorse

DC-TCP is the workhorse of TCP KentRidge. It fulfils TCP functions and collects information for *Network Pipe Classification* and *Intelligent Agent*.

Similar to the popular operating systems, DC-TCP follows the event-driven model. In order to improve maintainability, all events that are related with congestion control are first determined. For each related event, a *slice* is designed, and all congestion control related codes triggered by this event are put into the procedure written for this *slice*. At the proper locations (*slots*) of the remaining TCP codes (*backplane*), the procedures written for these *slices* are called. Hence, congestion control related codes are separated from other TCP codes and the maintainability is improved.

The main task of these *slices* is to fulfil congestion control. Congestion control has been a hot research topic in recent years, and many TCP adaptations have been proposed for different kinds of networks. These adaptations are different mainly in core functions of congestion control, such as *capacity probing* and *congestion recovery*. In order to facilitate the implementation of these adaptations and enable a host to dynamically and intelligently switch among them, core functions of congestion control are further *modularized* and *abstracted*. More specifically, an *element* is designed for each core function and the codes, that fulfil the function, will be organized into a procedure written for this *element*. For example, all codes, that reduce sending rate, will be organized into *element_cr*. When *slices* are processing their corresponding events, they will call *elements* through *function pointers* at proper locations (*sub-slots*) to fulfil congestion control. Hence, a *slice* can be regarded as a *sub-backplane* with *sub-slots* for calling *elements*. Figure 4.4 illustrates such a two level *backplane-slots* framework used by DC-TCP.

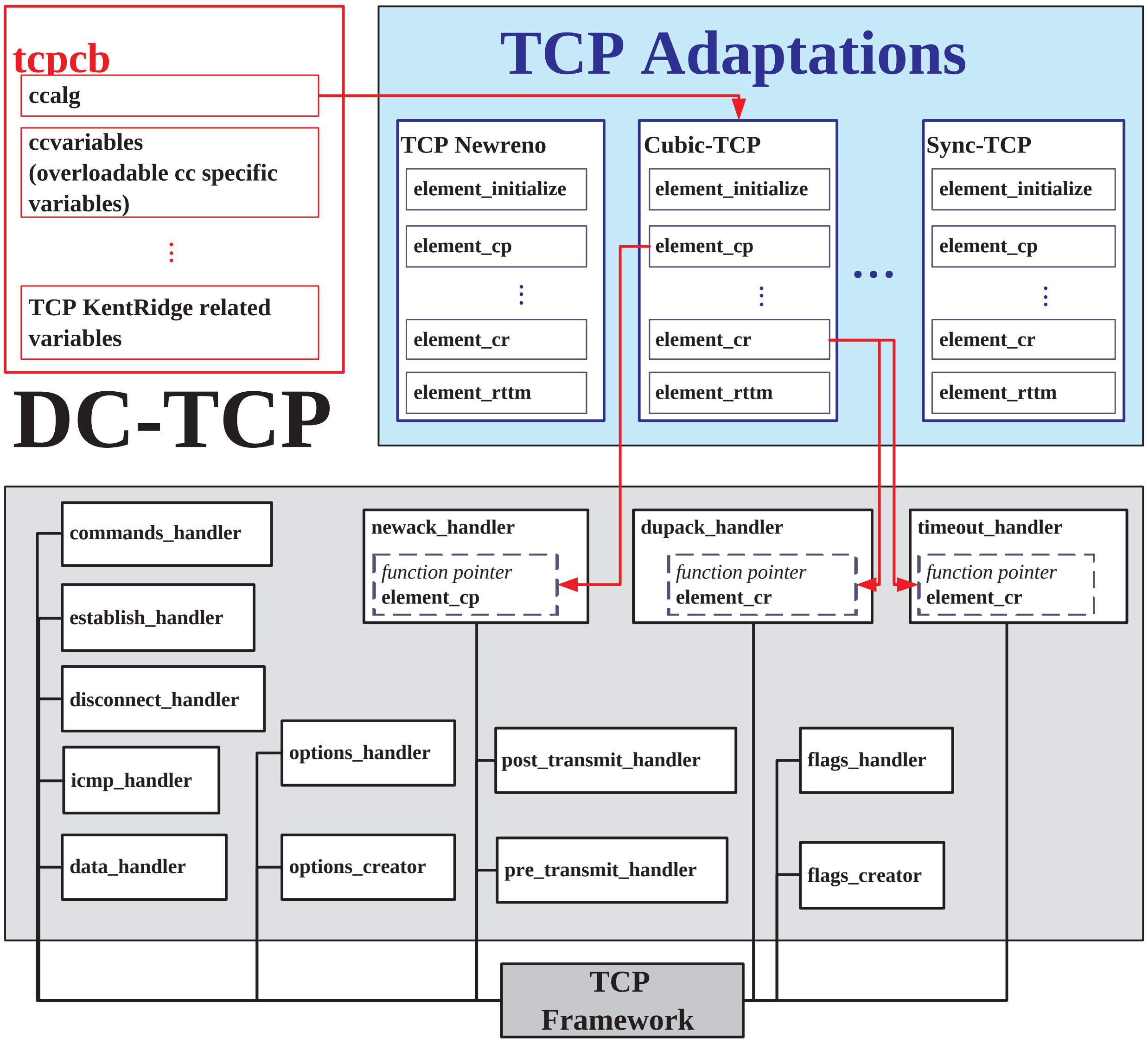


Figure 4.4: Design Pattern of DC-TCP: the two level backplane-slots framework

With this framework, a TCP adaptation can be implemented by instantiating a few *elements*. Hence, *the modularity, maintainability, and extensibility are improved, and it is now much easier to implement the existing adaptations and new adaptations that may be proposed in the future.*

A new structure, *tcp_ccalg*, is defined to hold the addresses of procedures written for these *elements*. For each supported TCP adaptation, there will be a corresponding *tcp_ccalg*. In the per-connection TCP control block, there is a pointer (*ccalg*) that points to the *tcp_ccalg* structure of the TCP adaptation used by this connection. Through changing *ccalg*, *a host can dynamically change the adaptation used by a connection.*

Considering that TCP KentRidge will support many TCP adaptations, it is impossible to insert the variables used by all of these adaptations into the per-connection TCP control block. Hence, *ccvariables* is declared in TCP control block for reserving some space. *ccvariables* is used to hold state variables required by a TCP adaptation, and it will be overloaded when the adaption used by a connection is changed.

Except congestion control, the *slices* of DC-TCP also carry out some other tasks of TCP KentRidge. For example, they collect application's expectations and peer's capabilities, which are needed by *Intelligent Agent*. These *slices* also update statistics of the network pipe through passively observing TCP data flow. These slices are also responsible to handle explicit notification from the peer and call *Network Pipe Classification* if needed.

In the following paragraphs, all *slices* and *elements* of DC-TCP are described in details.

Elements for Core Functions of Congestion Control

Since core functions of congestion control are to adjust sending rate based on network state, *element_initialize* is designed for initializing the sending rate, *element_cp* is designed for increasing the sending rate, and *element_cr* is designed for reducing the sending rate. In addition, it is a very important task of TCP congestion control to set *rto* appropriately and *element_rttm* is also designed for this purpose.

A core function of congestion control may be triggered by different events. For example, the sending rate will be reduced when 3DUPACK is received, the retransmission timer expires, or ECN-bit is received. Hence, the codes of a procedure (that is written for an *element*) are normally organized according to the *slices*, in which the *element* is called.

1. *element_initialize*: After a connection is established or the sender begins to transmit after a long idle period, this *element* is called to initialize *cwnd*, *ssthresh*, and other variables related with congestion control.

2. *element_cp*: This *element* is responsible to increase *cwnd* for probing network capacity. It is called when NEWACK is received, and it may also be called when some explicit notification comes from the network, the peer, or other layers.

3. *element_cr*: This *element* is responsible to reduce *cwnd* so that the network can recover from congestion. *ssthresh* and other related variables may also be adjusted in this *element*. Within a TCP adaptation, sending rate may be reduced in different ways when congestion is detected through different signals (*timeout*, 3DUPACK, etc.). Some explicit notification, that comes from the network, the peer, and other layers of the host, may also trigger this *element*.

4. *element_rttm*: After a RTT sample is measured, this element will be called to update *srtt* (smoothed RTT), *rttvar* (variance of RTT samples), and *rto*. *brtt* (the minimal RTT sample) may also be tracked for calculating queue delay. If Timestamp option is used, for each NEWACK, this *element* will be called. Otherwise, this *element* will be called once per window of data.

Slices for Congestion Control Related Events

In DC-TCP, congestion control related events include the establishment/disconnection of a connection, the expiration of retransmission timer, the arrival of a TCP segment, the arrival of an ICMP message, and system calls. Since DC-TCP processes different parts of a TCP segment sequentially, several *slices* are designed for different parts of a TCP segment, such as flags, options, acknowledgement number, sequence number, etc. Two *slices* are designed for acknowledgement number since DUPACK and NEWACK are treated very differently.

1. *commands_handler*:

   This *slice* is called when an application specifies its expectations through TCP socket options. It will store these information in TCP control block.

2. *establish_handler*:

   This *slice* is called after a connection is established. Its main function is to initialize *cwnd*, *ssthresh*, and other related variables. Hence, it has a *sub-slot* for calling *element_initialize*. In this *slice*, DC-TCP may also deduce application's expectations based on the port used by a connection.

3. *disconnect_handler*:

   This *slice* is called after a connection is disconnected. Some TCP adaptations may cache network information learned by this connection for the future usage.

4. *flags_creator* and *flags_handler*:

   flags_creator is called when TCP sets flags of an outgoing segment, and *flags_handler* is called when TCP flags of an incoming segment are processed. Considering that all TCP flag bits had been allocated, the two *slices* are designed solely for implementing TCP ECN [114]. Hence, *element_cr* will be called in *flags_handler*.

5. *options_creator* and *options_handler*:

   options_creator is called when TCP constructs options of an outgoing segment, and *options_handler* is called when TCP options of an incoming segment are processed. The two *slices* are designed to implement TCP adaptations that utilize TCP options for explicit cooperation, etc.

During *three-way-handshake*, the two *slices* are also responsible to negotiate the capabilities of the peer. For example, TCP SACK is negotiated through TCP SACK Permit option. Peer's capabilities will be stored in TCP control block.

During data transmission, *options_handler* may call different *elements* for different purposes based on the option to be processed. For example, when Timestamp option is processed, *element_rttm* will be called.

6. *newack_handler*:

   This *slice* is called when TCP processes a NEWACK. If the sender is in *SS* or *CA* state, *element_cp* will be called for probing network capacity. When the sender is in *FR* state, this *slice* is responsible to differentiate FULLACK and PARTIALACK, and act accordingly. When Timestamp option is not supported by the peer, this *slice* is also responsible to call *element_rttm* for measuring RTT, updating *rto*, etc.

7. *dupack_handler*:

   This *slice* is called when a DUPACK is received. This *slice* is responsible to detect congestion through 3DUPACK if the sender is in *SS* or *CA* states. When congestion is detected, this *slice* will retransmit the lost segment and call *element_cr*. When the sender is in *FR* state, this *slice* needs to treat DUPACK more carefully for maintaining an appropriate number of segments in the network pipe.

8. *data_handler*:

   This *slice* is called when the sequence number of an incoming segment is processed. It judges whether this packet is an in-order segment and decides whether an ACK packet should be sent back.

9. *pre_transmit_handler*:

   This *slice* is called when the sender tries to send out a segment. It will call *element_initialize* if the sender has stayed idly for a long period. This *slice* also decides the data to be transmitted or retransmitted.

10. *post_transmit_handler*:

    This *slice* is called after the outgoing segment is passed to IP layer. When Timestamp option is not used, some variables need to be updated for RTT measurement.

11. *timeout_handler*:

    This *slice* is called when retransmission timer expires. *element_cr* is called to reduce *cwnd* and update *ssthresh*. For the first expiration, some state variables may be saved to implement Eifel algorithm that undoes *cwnd* reduction triggered by spurious *timeout* [96]. For the subsequent expirations due to persistent segment loss, this *slice* will double *rto* to reduce sending rate further.

12. *icmp_handler*:

    This *slice* is called when an ICMP message is processed. ICMP message may come from routers of the Internet. Other layers of the same host may also notify TCP some useful information through ICMP message. For example, physical layer can report wireless link quality, with which TCP may deduce packet corruption rate and decide how to react to segment loss. This *slice* may call *element_cp* or *element_cr* to adjust sending rate based on the content of ICMP message.

In addition, the procedures written for the above *slices* should also collect statistics through observing TCP data flow. These statistics will be used by *Network Pipe Classification* for passively learning the type of the network pipe.

4.5.3 Network Pipe Classification

Network Pipe Classification is responsible for learning the most dynamic part of a connection's environment. More specifically, its task is to determine the type of the current network pipe used by this connection, based on network path characteristics. As discussed in section

4.2, the network path characteristics, that could affect TCP performance, include Available Bandwidth, RTT, Jitter, Re-ordering, PLR, and Asymmetry.

Many mechanisms have been proposed to estimate these characteristics of a network path. Based on whether probing packets are needed, these mechanisms can be divided into two categories: Intrusive and Passive. Intrusive mechanisms send packets in some special sequences and deduce network path characteristics through analyzing the experience of these probing packets [71][74][115]. Passive mechanisms deduce network path characteristics by analyzing the experience of the packets, that belong to the existing data flow. In the context of TCP KentRidge, since there is already a data flow, it mainly depends on the following per-connection statistics, that are passively maintained by DC-TCP.

1. Available Bandwidth: The current sending rate, $cwnd/srtt$, can be regarded as a coarse estimation of the current available bandwidth. Some in-path capacity estimation mechanisms [40][19][82][110] can also be used for estimating available bandwidth more accurately.

2. RTT and Jitter: For setting rto properly, RTT and Jitter has been maintained by TCP sender. Based on some EWMA (Exponentially Weighted Moving Average) filters, the sender maintains $srtt$ (a smoothed average of RTT samples) and $rttvar$ (a smoothed average of variance). $brtt$, the smallest RTT sample that can be regarded as the estimation of RTPD, is a very valuable metric and should also be tracked.

3. Packet Reordering: Packet reordering can be measured by calculating the number of consecutive DUPACK packets, in the case that spurious fast retransmission is detected. Hence, DSACK [55] or Eifel [97] should be used for carrying out spurious fast retransmission detection.

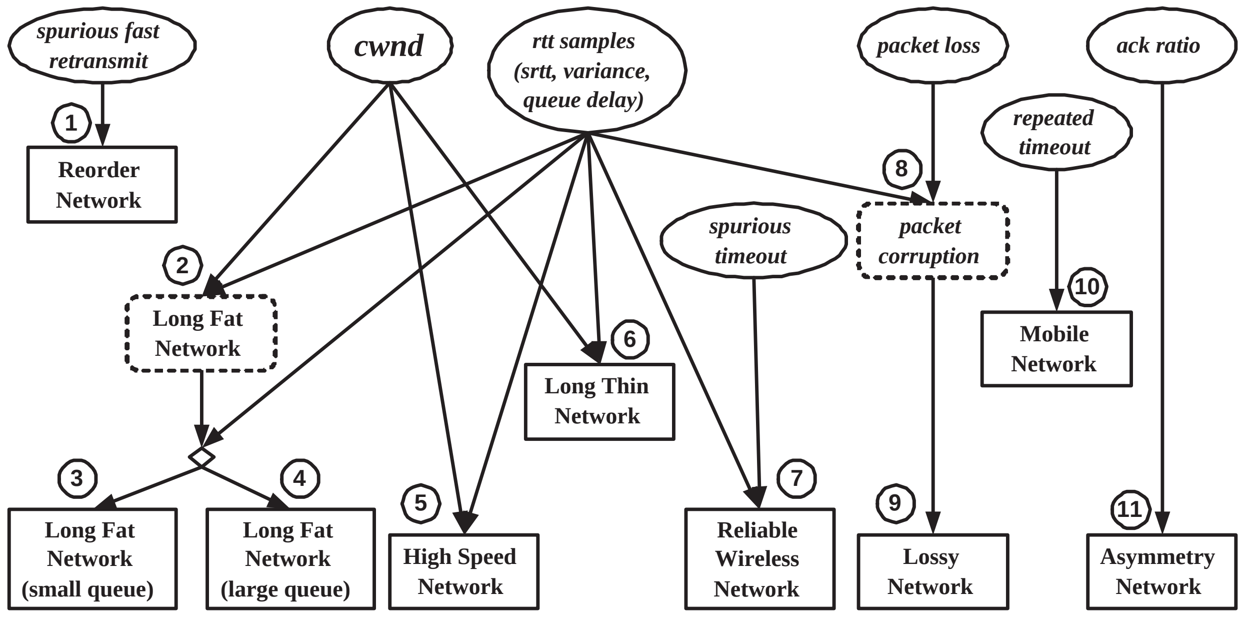
4. PLR: For reliable data transmission and congestion control, TCP sender has already carried out segment loss detection. Hence, the interval between two consecutive loss events can be measured, and PLR can be calculated based on the smoothed average

of loss intervals. Packet loss pattern, sporadic or bursty, may also be estimated by measuring the number of *fast retransmit* (caused by sporadic packet loss) and *timeout* (caused by bursty packet loss). In addition, the frequency of persistent *timeout* should also be maintained. It can be used to judge whether the network is frequently disconnected. Furthermore, TCP packet may be lost due to network congestion or transmission error. In some scenarios, queue delay can be used to differentiate the loss reason, and packet corruption rate can be calculated.

5. Asymmetry: If a TCP connection has bi-directional data flows, end-points may cooperate to find the asymmetry of the network path. If data flow of the application is uni-direction, the sender can count the number of transmitted data segments and the number of received ACK packets. The ratio between the two numbers can be used to judge whether the connection pass through some asymmetric networks.

According to the statistics passively maintained by DC-TCP, *Network Pipe Classification* can periodically determine the type of the current network pipe and trigger *Intelligent Agent* if needed. Hence, a new timer is necessary so that a connection can keep changing its behaviors with the change of its environment. Figure 4.5 shows the current model and rules used by *Network Pipe Classification*. The thresholds of each metric should be selected very carefully. For example, watermark violation proposed in [122] should be used to avoid that a sender keeps changing its TCP adaptation too frequently.

Normally, the routers can know the network much better than TCP sender. Lower layers also has more information about the first link used by TCP sender. Furthermore, in the current Internet, the access link normally dominates the characteristics of a network path, especially when wireless link or slow link is used by TCP receiver. In this case, TCP receiver has better opportunity to estimate the last link's characteristics timely and accurately [139][147]. Based on the above observations, instead of only utilizing the statistics passively maintained by DC-TCP, explicit notification, that comes from the peer, the network, or other layers of the same host, is also used by *Network Pipe Classification* in some



1. frequently spurious fast retransmit : Reorder Network
2. large cwnd & long rtt : Long Fat Network
3. small observed queue delay & Long Fat Network : Long Fat Network with small queue
4. large observed queue delay & Long Fat Network : Long Fat Network with large queue
5. large cwnd & short rtt : High Speed Network
6. small cwnd & long rtt : Long Thin Network
7. large jitter & frequent timeout : Reliable Wireless Network
8. packet loss & small queue delay : packet corruption
9. frequently packet corruption : Lossy Network
10. frequently repeated timeout : Mobile Network
11. large ratio between the number of transmitted segments and received ack : Asymmetry Network

Figure 4.5: Network Pipe Classification

trustworthy environments.

When TCP KentRidge receives some explicit notification from the peer (through TCP option), the network (through ICMP message), or other layers (through ICMP message), it normally indicates that some important events happen. Hence, *Network Pipe Classification* will immediately judge the type of the current network pipe, and trigger *Intelligent Agent* if the type of network pipe has changed.

4.6 Implementation Status of TCP KentRidge

Although Linux TCP is more similar to TCP KentRidge, FreeBSD 7.1 is selected as the platform for implementing TCP KentRidge since its source codes are stabler and have been well documented by [134][138]. To keep the stability of kernel source codes and facilitate the development, TCP KentRidge is implemented as a Loadable Kernel Module. TCP KentRidge Console, a Graphical User Interface tool, is also provided to researchers for investigating congestion control with TCP KentRidge.

4.6.1 The Loadable Kernel Module

TCP source codes of FreeBSD 7.1 are read through, and *slots* (the locations of DC-TCP's *slices*) are first located. Congestion control related codes are then extracted out and reorganized into procedures written for *slices* of DC-TCP.

At these *slots*, the procedures written for *slices* are called through function pointers so that these procedures can be put into the loadable kernel module. During the implementation of TCP KentRidge, these procedures written for *slices* need to be changed frequently for supporting new features and network path characteristics estimation, such as TCP Pacing [13] and TCP Probing [110]. Putting these procedures in a loadable kernel module can speed up the development. For example, the kernel needs to be rebuilt only when some new variables are inserted into *tcpcb*, the per-connection TCP control block. The much more fallible logic programming can be carried out through rebuilding the codes and reloading the module into the kernel.

In the kernel, a new global variable, *tcp_itcp_loaded* is defined to hold whether the module has been loaded. A new global structure, *tcp_intelligence*, is also defined to support the loadable kernel module. After the module is loaded, *tcp_intelligence* will hold addresses of the procedures that instantiate the *slices* of DC-TCP. List 4.1 shows how the remaining TCP codes call a procedure written for a *slice* at the corresponding *slot*.

Listing 4.1: Call Procedure Written for *slice*: *newack_handler*

```
if(tcp_itcp_loaded && (tcp_intelligence.newack_h != NULL)) {
        tcp_intelligence.newack_h(tcpcb, ... );
} else {          ...  }
```

tcp_intelligence also holds the addresses of two procedures (implemented in the module) that return the default TCP adaptation and all supported TCP adaptations of TCP KentRidge. Finally, *tcp_intelligence* holds the address of the procedure that instantiates *Network Pipe Classification*. This procedure is called by a new timer so that a connection can react to the network periodically.

DC-TCP

These procedures written for *slices* of DC-TCP are refined. A new variable, *ccstate*, is inserted into *tcpcb* for explicitly holding congestion control state of a TCP sender, and the codes of these *slices* are changed accordingly for improving maintainability.

Many other variables are inserted into *tcpcb* for holding statistics of the network pipe used by a connection. And the codes that maintain these variables, are inserted into these procedures written for *slices*.

More importantly, within the procedures written for *slices*, *sub-slots* (the locations of DC-TCP's *elements*) are located. At these *sub-slots*, the procedures written for *elements* are also called through *function pointers* so that core functions of congestion control can be abstracted. Figure 4.6 shows the locations of the *slots* for all *slices* and the locations of the *sub-slots* for all *elements*.

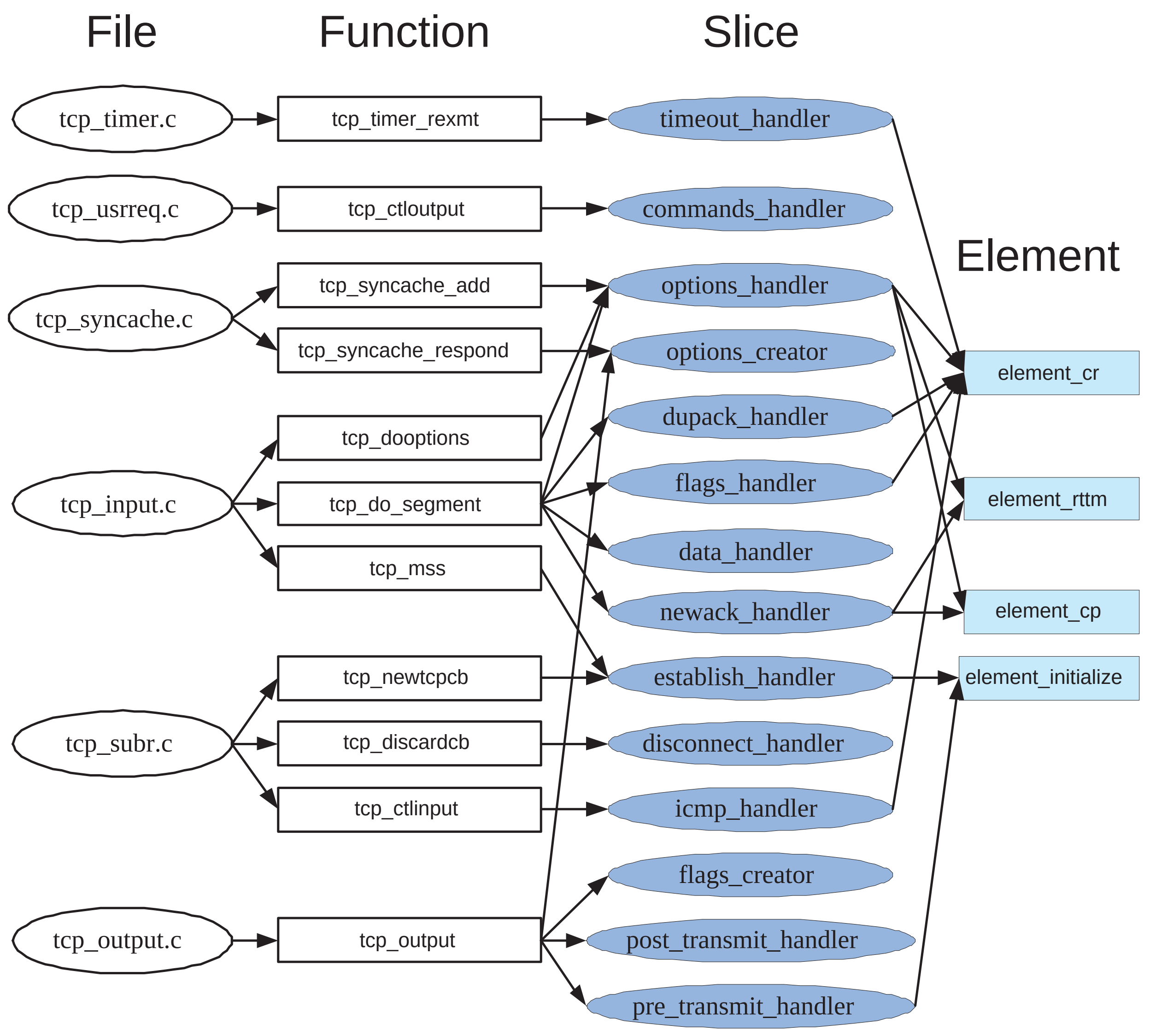


Figure 4.6: Locations of Slices and Elements

With these *sub-slots* designed for *elements*, a TCP adaptation can be implemented by instantiating these *elements*. A new structure, *tcp_ccalg*, is defined for holding the addresses of procedures written for *elements*. For each supported TCP adaptation, there is a corresponding *tcp_ccalg* structure in the loadable kernel module. Currently, except the standard TCP congestion control, Sync-TCP and Cubic-TCP have also been implemented in TCP

KentRidge. The codes of all supported TCP adaptations are also put into the same loadable kernel module.

In *tcpcb*, there is a pointer (*ccalg*) that points to the *tcp_ccalg* structure of the TCP adaptation used by this connection. Through changing this pointer, TCP KentRidge can change TCP adaptation used by a connection. List 4.2 shows how a *slice* calls a procedure written for an *element* at the corresponding *sub-slot*.

Listing 4.2: Call Procedure Written for an *element*: *element_cp*

```
if(tcpcb->ccalg!=NULL && (tcpcb->ccalg->element_cp!=NULL)) {
        tcpcb->ccalg->element_cp(tcpcb, ... );
} else {          ...  }
```

In addition, *ccvariables* is also defined in *tcpcb* to reserve 64 bytes. *ccvariables* is used to hold variables needed by a TCP adaptation, and it will be overloaded when the adaption used by a connection is changed.

Knowledge Base*, *Intelligent Agent*, and *Network Pipe Classification

In this loadable kernel module, *Knowledge Base* is instantiated as a table. For each supported TCP adaptation, there is a corresponding row, which holds the kind of network pipe that the TCP adaptation is proposed for, the capabilities that the adaptation depends on, and the advantages & shortcomings of the adaptation.

As for *Intelligent Agent*, it is instantiated as a procedure which looks up *Knowledge Base* based on the current environment of a connection with the aim to select the most appropriate TCP adaptation for this connection. It will instruct DC-TCP to use the selected TCP adaptation through changing *ccalg* of this connection's *tcpcb*.

Network Pipe Classification is also instantiated as a procedure. After this procedure determines the kind of network pipe used by a connection and finds out that the network has changed, it will call *Intelligent Agent* to re-select TCP adaptation for this connection. *Network Pipe Classification* will be called when some explicit notification comes from the peer, the network, or other layers. In addition, a new timer is added into *tcpcb* so that a connection can trigger *Network Pipe Classification* periodically. Hence, a connection will be able to keep changing its behaviors according to the changes of its environment.

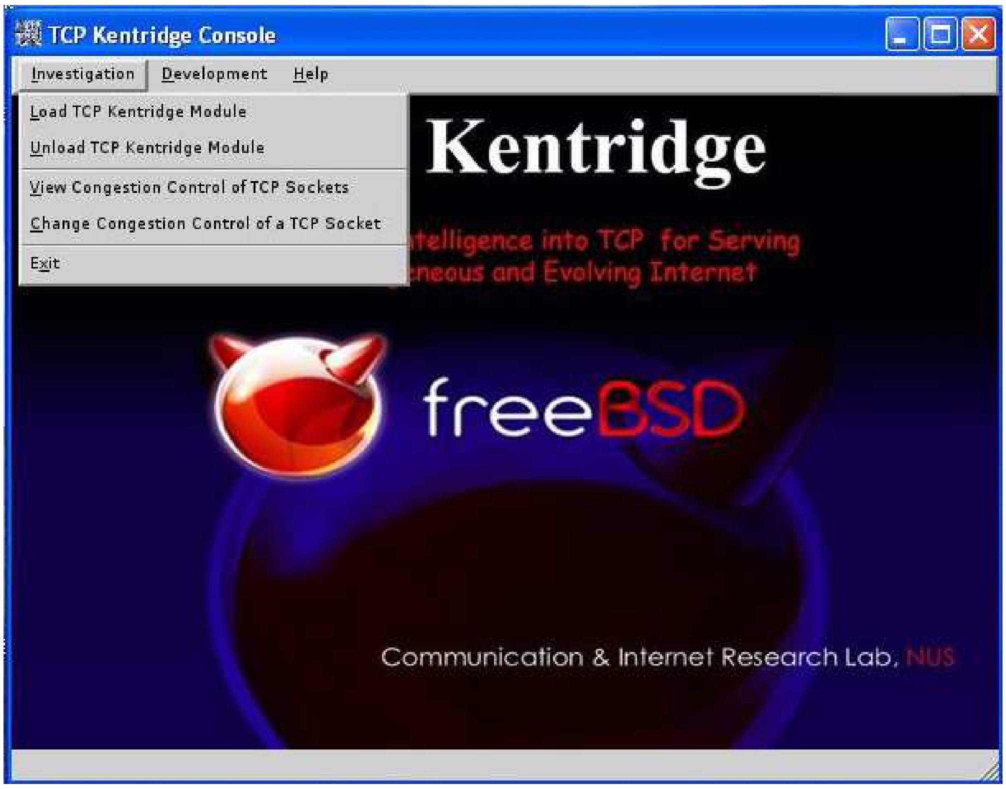
With this loadable kernel module, we can significantly shorten the time used for compiling the modified codes. On a PC with normal configuration (CPU: PIV 1.6GHz, Memory: 512MB), it needs more than half an hour to compile the kernel with the modified TCP codes. With this loadable kernel module, we can compile the module with the changed codes and load it into the kernel within a few minutes. With the two level *backplane-slots* framework of DC-TCP, new TCP adaptations can also be implemented easily. Sync-TCP and Cubic-TCP have been implemented in DC-TCP within three days.

4.6.2 TCP KentRidge Console

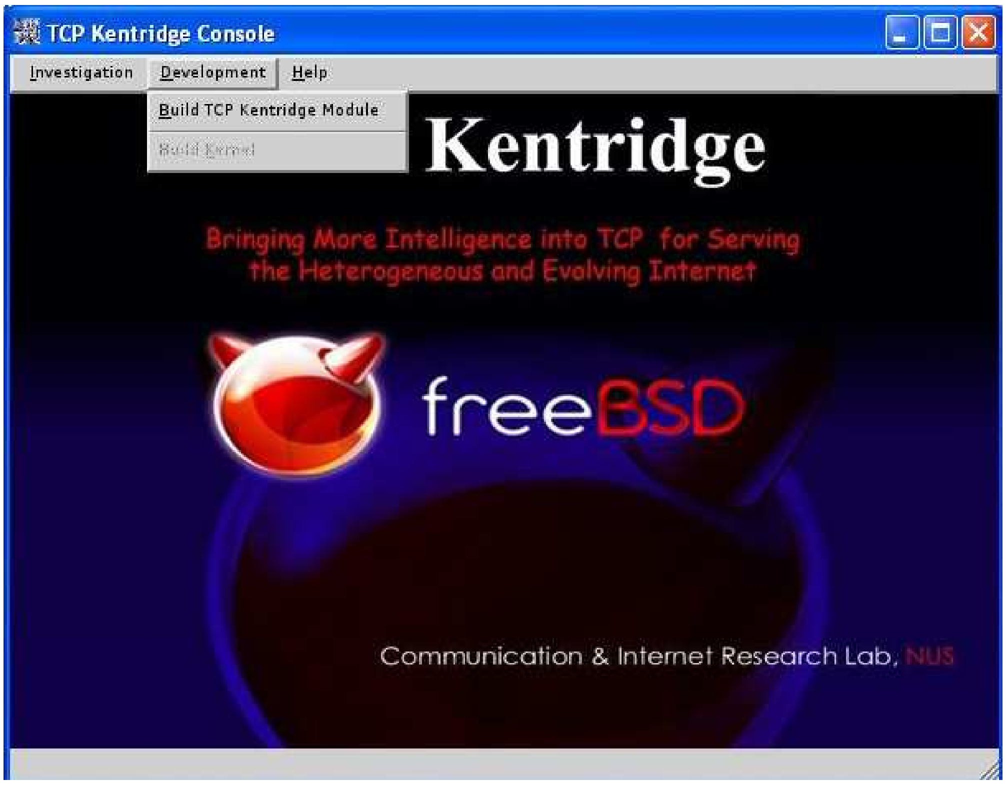
To facilitate researchers, TCP KentRidge Console, a GUI tool, has also been provided. Figure 4.7 shows menu items of TCP KentRidge Console. Through this tool, users can carry out the following tasks.

1. Rebuild the kernel when it is necessary.
2. Rebuild the loadable kernel module of TCP KentRidge after its codes are changed.
3. Load the module into the kernel.
4. Unload the model from the kernel.
5. View TCP adaptations used by active TCP sockets.
6. Change TCP adaptation used by a TCP socket.

Figure 4.8(a) shows the interface designed for displaying TCP adaptations used by active TCP sockets, and figure 4.8(b) illustrates how to change the adaptation used by a TCP

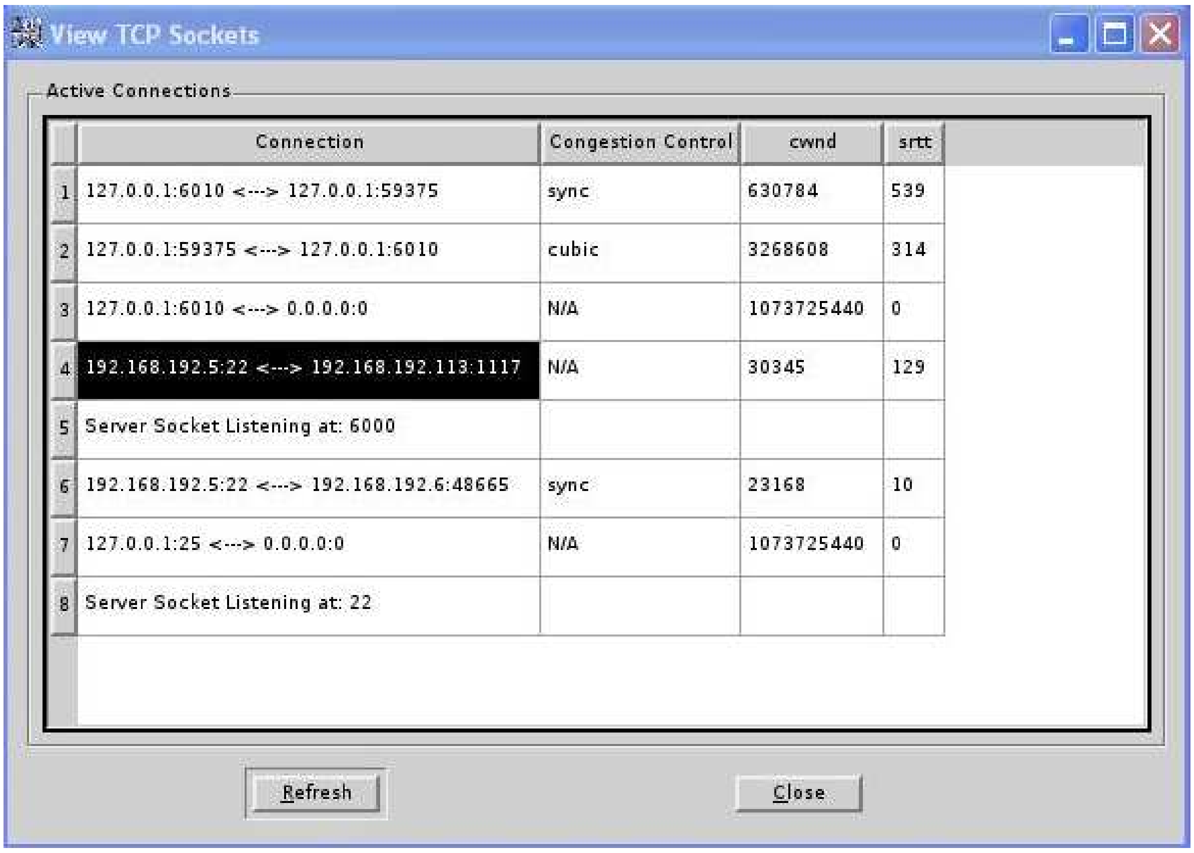

TCP Kentridge Console
Investigation
Development
Help
Load TCP Kentridge Module
Unload TCP Kentridge Module
View Congestion Control of TCP Sockets
Change Congestion Control of a TCP Socket
Exit
Kentridge
freeBSD
Communication & Internet Research Lab, NUS

(a) Investigation


TCP Kentridge Console
Investigation
Development
Help
Build TCP Kentridge Module
Build Kernel
Kentridge
Bringing More Intelligence into TCP for Serving the Heterogeneous and Evolving Internet
freeBSD
Communication & Internet Research Lab, NUS

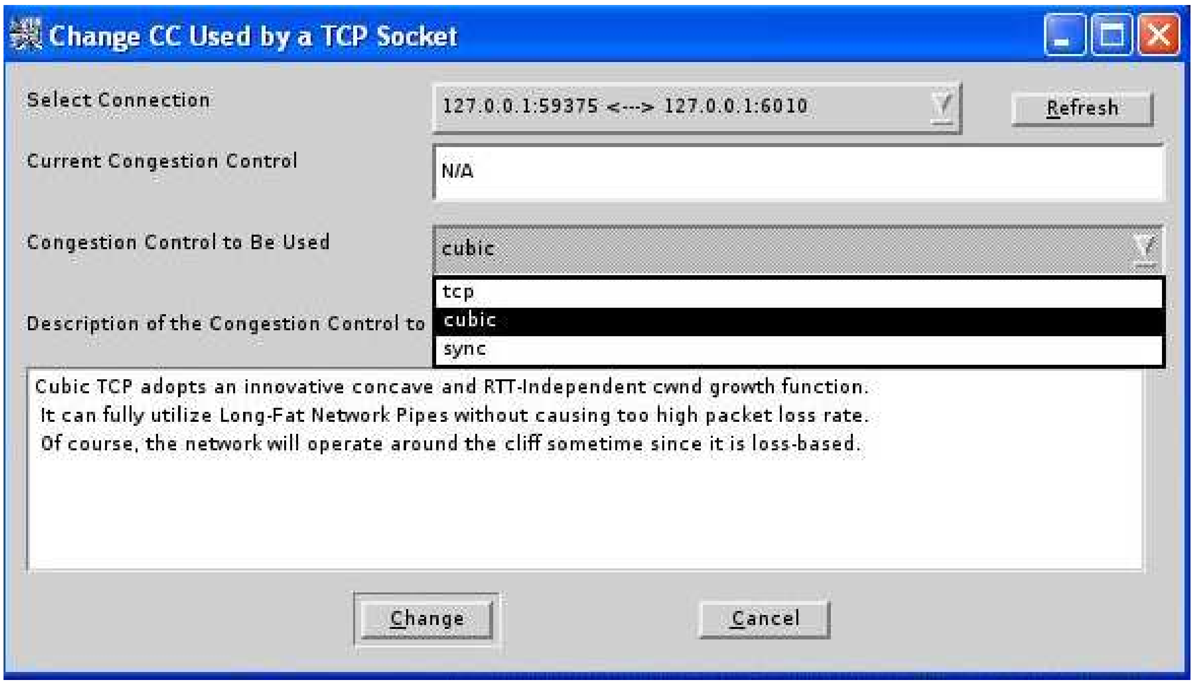
(b) Development

Figure 4.7: TCP KentRidge Console: Menu Items



	Connection	Congestion Control	cwnd	srtt
1	127.0.0.1:6010 <---> 127.0.0.1:59375	sync	630784	539
2	127.0.0.1:59375 <---> 127.0.0.1:6010	cubic	3268608	314
3	127.0.0.1:6010 <---> 0.0.0.0:0	N/A	1073725440	0
4	192.168.192.5:22 <---> 192.168.192.113:1117	N/A	30345	129
5	Server Socket Listening at: 6000			
6	192.168.192.5:22 <---> 192.168.192.6:48665	sync	23168	10
7	127.0.0.1:25 <---> 0.0.0.0:0	N/A	1073725440	0
8	Server Socket Listening at: 22			

(a) View Active Sockets



Select Connection: 127.0.0.1:59375 <---> 127.0.0.1:6010

Current Congestion Control: N/A

Congestion Control to Be Used: cubic

Description of the Congestion Control to

Cubic TCP adopts an innovative concave and RTT-Independent cwnd growth function.
It can fully utilize Long-Fat Network Pipes without causing too high packet loss rate.
Of course, the network will operate around the cliff sometime since it is loss-based.

(b) Change TCP Adaptation of a TCP Socket

Figure 4.8: View and Change TCP Adaptation Used by TCP Socket

socket. In the case that X Window is not available, two commands can be used to view TCP adaptations used by active TCP sockets (*tcpccview*) and change the adaptation used by a TCP socket (*tcpccchange*). When executing *tcpccchange*, four tuples of this socket (*source IP*, *source port*, *destination IP*, *destination port*) and the name of the adaptation to be used by this socket must be specified.

4.7 Summary and Future Work

In this chapter, the existing TCP adaptations proposed for different networks are first summarized. The necessity of re-engineering TCP implementation for the heterogeneous and evolving Internet is then discussed, and the design of TCP KentRidge is presented in detail. An initial prototype of TCP KentRidge has also been implemented in FreeBSD 7.1. The details of TCP KentRidge implementation on FreeBSD 5.4 can be found in [106][120]. More information about TCP KentRidge can be found at `http://cir.nus.edu.sg/tcpkentridge`.

However, there are still a lot of work to be done before TCP KentRidge can be a full-fledged solution. For example, more TCP adaptations should be implemented, *Knowledge Base* should be expanded accordingly, and the algorithms used by *Intelligent Agent* and *Network Pipe Classification* should also be refined. The necessary future works will be discussed further in chapter 5.

Chapter 5

Conclusion and Future Work

Improving the performance of TCP in the Internet has been a hot research topic for quite a long time. As TCP is the de-facto standard transport protocol of the Internet, it is very valuable to improve its performance. Furthermore, the Internet is changing continuously, and challenges faced by TCP seem to be endless.

In this thesis, we focus on three very important recent trends of the Internet, namely mobile Internet access, high speed Internet, and the heterogeneous and evolving Internet. For the first two trends, we propose the corresponding TCP enhancements to improve the performance of TCP. As for the last trend, we consider the challenges faced by TCP implementation in the heterogeneous and evolving Internet, and propose a new TCP implementation framework, with which a host could have the potential of changing its behaviors according to the environment automatically and intelligently. In this chapter, the detailed research results and contributions of this thesis are summarized, and the potential future work is discussed.

5.1 Research Summary

To improve TCP performance in a heterogeneous mobile environment, TCP HandOff (TCP-HO), a practical end-to-end mechanism, is proposed in this thesis (chapter 2). TCP-HO is designed after analyzing the kind of handoff that may occur in a heterogeneous mobile environment and the challenges that TCP faces during each kind of handoff. We assume that a mobile host is able to detect the completion of handoff immediately and has a coarse estimation of new wireless link's bandwidth. TCP-HO is then proposed to improve the performance of mobile host through exploiting the explicit cooperation between server and mobile host. The design of TCP-HO also considers how to thwart cheating users and how to avoid to hurt cross traffic. Experimental results indicate that in a heterogeneous mobile environment, TCP-HO can improve TCP performance significantly without adversely affecting cross traffic, even when mobile host has only a coarse estimation of new wireless link's bandwidth. Considering that more and more users are accessing the Internet through heterogeneous wireless networks and mobile host could have a coarse estimation of wireless link's bandwidth, it should be worthwhile to implement TCP-HO at both server and mobile host for improving the performance of TCP. *Hence, this thesis provides a promising solution (TCP-HO) for mobile Internet access through heterogeneous wireless networks.*

Synchronized TCP (Sync-TCP), a new delay-based high speed congestion control algorithm, is also proposed in this thesis (chapter 3) for safely ramping up the throughput of bandwidth-greedy and elastic applications that run on long fat network pipes (*with large queue*) of the Internet. The existing delay-based congestion control algorithms are first analyzed and the challenges, that Sync-TCP must solve to achieve its design goals, are discussed. The key insight of Sync-TCP is that if competing flows could detect the same congestion signal through queue delay, these flows can coordinate their behaviors and drive the network to operate around their desired point, the *knee*. Based on this observation, Sync-TCP is carefully designed to convey highly *synchronized* congestion signals to competing flows, even though queue delay is a *noisy* and *delayed* network feedback. An adaptive queue delay based congestion window decrease rule and a RTT-independent congestion window increase rule are adopted for driving the network to operate around the knee and for distributing the residual bandwidth fairly among competing flows, even when the number of competing flows varies and their round trip propagation delays differ significantly. Extensive simulations and preliminary testbed evaluations indicate that Sync-TCP does achieve its design goals. *Hence, this thesis also provides a promising solution (Sync-TCP) for safely ramping up the throughput of bandwidth-greedy and elastic applications that run on long fat network pipes of the Internet.*

In this thesis (chapter 4), TCP KentRidge, a new TCP implementation framework, is proposed for the heterogeneous and evolving Internet. We first summarizes the existing TCP adaptations proposed for different networks. The challenges faced by a TCP implementation in the Internet are then discussed. After that, TCP KentRidge is carefully designed so that a host could have the potential of automatically applying the most appropriate TCP adaptation to each connection according to its current environment. TCP KentRidge is also carefully designed so that new TCP adaptations can be implemented in this framework and the necessary intelligence can be added easily. An initial prototype of TCP KentRidge has been implemented in FreeBSD. *Hence, this thesis has made a solid step towards the intelligent TCP, with which a host could automatically and intelligently change its behaviors according to the environment.*

5.2 Future Work

Several extensions to the research work presented in this thesis are possible. Firstly, although preliminary testbed evaluations have been carried out, it is worthwhile to further evaluate and tune Sync-TCP on a high speed network testbed and in the live Internet. These results can be used to convince the community and advertise Sync-TCP as a standard way of ramping up the throughput of bandwidth-greedy and elastic applications. Secondly, we have implemented only an initial prototype of TCP KentRidge. Many algorithms, that instantiate the intelligence, still need to be designed. Hence, there are still a lot of works to be done before it could become a full-fledged TCP implementation. Below is a list of these necessary works and open problems.

1. Firstly, the two level *backplane-slots* framework of DC-TCP should be re-scrutinized. Its maintainability and extensibility should also be evaluated quantitively. For example, we can compare the time spent to implement a TCP adaptation in DC-TCP with the time spent to implement the same TCP adaptation in the kernel.

2. Secondly, the existing TCP adaptations should be investigated thoroughly for achieving consensuses on which TCP adaptation should be used on each kind of network pipe. We have to decide the TCP adaptations to be supported by TCP KentRidge, implement these adaptations in the framework of DC-TCP, and construct *Knowledge Base* accordingly. Furthermore, with TCP KentRidge, different TCP adaptations will be simultaneously used in the Internet. Coexistence issues among these supported adaptations should also be studied.

3. Thirdly, *Network Pipe Classification* needs some algorithms to ensure that senders can learn their environment correctly. Thresholds used by the model illustrated in figure 4.5 should also be carefully selected so that these algorithms could make sure that the distributed senders on the same network pipe can converge to the correct type. This task is very challenging since TCP learns the environment intrusive. For example, the sender learns the bandwidth through adjusting its sending rate and observing network congestion state. However, at the same time, its behaviors also change the phenomenons observed by other senders.

4. Fourthly, *Intelligent Agent* also needs to be refined. In the current Internet, many networks can be classified to multiple types. For example, GPRS network has small available bandwidth, long RTT, large jitter, high PLR, and asymmetric bandwidth. The algorithms used by *Intelligent Agent* should be well designed so that the sender can select the most appropriate TCP adaptation for these networks.

In addition, assuming that application's expectations and peer's capabilities will not change during the life time of a connection, the *Intelligent Agent* may first filter out TCP adaptations that cannot be applied to a connection after the connection is established. The kind of filtering is executed only once, and it may speed up *Intelligent Agent* a lot in the remaining life time of this connection.

5. Finally, compared with the existing TCP implementations, TCP KentRidge, which is designed to be more intelligent, will be more complex. Hence, TCP KentRidge will unavoidably consume more resources (CPU, memory, etc.). When implementing TCP KentRidge, more attention should be paid to optimizations so that TCP KentRidge will not become a bottleneck.

Bibliography

[1] 3gpp. Available online at http://www.3gpp.org.

[2] Iperf. http://dast.nlanr.net/Projects/Iperf/.

[3] Ns2 network simulator. http://www.isi.edu/nsnam/ns/.

[4] Trans-pacific express. Available online at http://www.networkworld.com/news/2006/121806-verizon-business.html.

[5] Wimax. Available online at http://www.wimaxforum.org.

[6] General packet radio service (gprs) service description, ver. 7.1.0. European Standard 301 344, GSM 03.60, August 1999.

[7] High performance radio local area network, type 2, requirements and architectures for wireless broadband access. TR 101 031 V2.2.1, January 1999.

[8] Wireless lan medium access control (mac) and physical layer (phy) specifications. IEEE P802.11, 1999.

[9] Wireless lan medium access control (mac) and physical layer (phy) specifications—high-speed physical layer in the 5 ghz band. IEEE P802.11, 1999.

[10] Wireless lan medium access control (mac) and physical layer (phy) specifications—higher-speed physical layer extension in the 2.4ghz band. IEEE P802.11, 1999.

[11] Rlc protocol specification. 3G TS 25.322, V3.2.0, March 2000.

[12] Wireless lan medium access control (mac) and physical layer (phy) specifications—amendment 4: Further higher data rate extension in the 2.4 ghz band. IEEE P802.11, 2003.

[13] Amit Aggarwal, Stefan Savage, and Thomas Anderson. Understanding the performance of tcp pacing. In *INFOCOM*, 2000.

[14] M. Allman, H. Balakrishnan, and S. Floyd. Enhancing tcp's loss recovery using limited transmit. RFC 3042, January 2001.

[15] M. Allman, S. Floyd, and C. Partridge. Increasing tcp's initial window. RFC 3390, 2002.

[16] M. Allman, D. Glover, and L. Sanchez. Enhancing tcp over satellite channels using standard mechanisms. RFC 2488, 1999.

[17] M. Allman, V. Paxson, and W. Stevens. Tcp congestion control. RFC 2581, April 1999.

[18] Mark Allman. Tcp congestion control with appropriate byte counting (abc). RFC 3465, February 2003.

[19] Thomas Anderson, Andrew Collins, Arvind Krishnamurthy, and John Zahorjan. Pcp: Efficient endpoint congestion control. In *NSDI*, 2006.

[20] Andrea Baiocchi, Angelo P. Castellani, and Francesco Vacirca. Yeah-tcp: Yet another highspeed tcp. In *PFLDnet Workshop*, 2007.

[21] A. Bakre and B.R. Badrinath. I-tcp: indirect tcp for mobile hosts. In *ICDCS*, 1995.

[22] H. Balakrishnan, V. N. Padmanabhan, G. Fairhurst, and M. Sooriyababdara. Tcp performance implications of network path asymetry. RFC 3449, 2002.

[23] H. Balakrishnan, V. N. Padmanabhan, and R. H. Katz. The effects of asymmetry on tcp performance. In *MOBICOM*, 1997.

[24] H. Balakrishnan, S. Seshan, E. Amir, and R.H. Katz. Improving tcp/ip performance over wireless networks. In *MOBICOM*, 1995.

[25] Hari Balakrishnan and Randy H. Katz. Explicit loss notification and wireless web performance. In *Globecom*, 1998.

[26] Hari Balakrishnan, Venkata N. Padmanabhan, Srinivasan Seshan, and Randy H. Katz. A comparison of mechanisms for improving tcp performance over wireless links. *IEEE/ACM Transactions on Networking*, 5, 1997.

[27] Paul Barford and Mark Crovella. Generating representative web workloads for network and server performance evaluation. In *ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, 1998.

[28] S. Biaz and N. Vaidya. Sender-based heuristics for distinguishing congestion losses from wireless transmission losses. Technical report, TAMU, June 1998.

[29] Saad Biaz and Nitin H. Vaidya. Is the round-trip time correlated with the number of packets in flight? In *IMC*, 2003.

[30] Ethan Blanton and Mark Allman. On making tcp more robust to packet reordering. *Computer Communication Review*, 32, January 2002.

[31] J. Border, M. Kojo, J. Griner, G. Montenegro, and Z. Shelby. Performance enhancing proxies intended to mitigate link-related degradations. RFC 3135, June 2001.

[32] Alessio Botta, Alberto Dainotti, and Antonio Pescap. Multi-protocol and multi-platform traffic generation and measurement. In *INFOCOM (DEMO)*, 2007.

[33] Lawrence S. Brakmo, Sean W. O'Malley, and Larry L. Peterson. Tcp vegas: New techniques for congestion detection and avoidance. In *SIGCOMM*, 1994.

[34] Kevin Brown and Suresh Singh. M-tcp: Tcp for mobile cellular networks. *Computer Communication Review*, 27, October 1997.

[35] V. Bychkovsky, B. Hull, A. Miu, H. Balakrishnan, and S. Madden. A measurement study of vehicular internet access using in situ wifi networks. In *MOBICOM*, 2006.

[36] Ed. C. Perkins. Ip mobility support for ipv4. RFC 3344, August 2002.

[37] Ramon Caceres and Liviu Iftode. Improving the performance of reliable transport protocols in mobile computing environments. *IEEE Journal on Selected Areas in Communications*, 13, June 1995.

[38] Carlo Caini and Rosario Firrincieli. Tcp hybla: a tcp enhancement for heterogeneous networks. *International Journal of Satellite Communications and Networking*, 22, 2004.

[39] Jin Cao, William S. Cleveland, Yuan Gao, Kevin Jeffay, F. Donelson Smith, and Michele Weigle. Stochastic models for generating synthetic http source traffic. In *INFOCOM*, 2004.

[40] Claudio Casetti, Mario Gerla, Saverio Mascolo, M.Yahya Sanadidi, and Ren Wang. Tcpwestwood: End-to-end bandwidth estimation for enhanced transport over wireless links. *Wireless Networks*, 8:467–479, 2002.

[41] Mun Choon Chan and Ram Ramjee. Tcp/ip performance over 3g wireless links with rate and delay variation. In *MOBICOM*, 2002.

[42] Mun Choon Chan and Ram Ramjee. Improving tcp/ip perfornance over third generation wireless networks. In *INFOCOM*, 2004.

[43] Dah Ming Chiu and R. Jain. Analysis of the increase and decrease algorithms for congestion avoidance in computer networks. *Computer Networks and ISDN Systems archive*, 7, June 1989.

[44] David D. Clark. Window and acknowledgement stratedgy in tcp. RFC 813, July 1982.

[45] S. Dawkins, G. Montenegro, M. Kojo, and V. Magret. End-to-end performance implications of slow links. RFC 3150, 2001.

[46] S. Dawkins, G. Montenegro, M. Kojo, V. Magret, and N. Vaidya. End-to-end performance implications of links with errors. RFC 3155, 2001.

[47] M. Deziel and L. Lamont. Implementation of an ieee 802.11 link available bandwidth algorithm to allow cross-layering. In *IEEE WiMobapos*, 2005.

[48] D.Leith, R.Shorten, G.McCullagh, J.Heffner, L.Dunn, and F.Baker. Delay-based aimd congestion control. In *PFLDnet Workshop*, 2007.

[49] D.Leith, R.Shorten, G.Mccullagh, L.Dunn, and F.Baker. Making available base-rtt for use in congestion control applications. *IEEE Comm. Letters*, 12(6):429–431, June 2008.

[50] S. Floyd. Connections with multiple congested gateways in packet-switched networks part 1: One way traffic. *Computer Communication Review*, 21, 1991.

[51] S. Floyd. Highspeed tcp for large congestion windows. RFC 3649, December 2003.

[52] S. Floyd. Limited slow-start for tcp with large congestion windows. RFC 3742, March 2004.

[53] S. Floyd, T. Henderson, and A. Gurtov. The newreno modification to tcp's fast recovery algorithm. RFC 3782, April 2004.

[54] Sally Floyd and Van Jacobson. Random early detection gateways for congestion avoidance. *IEEE/ACM Transactions on Networking*, 1(4):397–413, 1993.

[55] Sally Floyd, Jamshid Mahdavi, Matt Mathis, and Matt Podolsky. An extension to the selective acknowledgment (sack) option for tcp. RFC 2883, July 2000.

[56] Cheng Peng Fu and Soung C. Liew. Tcp veno: Tcp enhancement for transmission over wireless access networks. *IEEE Journal on Selected Areas in Communications*, 21, 2003.

[57] Tom Goff, James Moronski, D. S. Phatak, and Vipul Gupta. Freeze-tcp: A true end-to-end tcp enhancement mechanism for mobile environments. In *INFOCOM*, 2000.

[58] Andrei Gurtov and Reiner Ludwig. Responding to spurious timeouts in tcp. In *INFOCOM*, 2003.

[59] Sangtae Ha, Injong Rhee, and Lisong Xu. Cubic: a new tcp-friendly high-speed tcp variant. *Operating Systems Review*, 42:64–74, July 2008.

[60] Ahmad Al Hanbali, Eitan Altman, and Philippe Nain. A survey of tcp over mobile ad hoc networks. Technical report, INRIA, France, May 2004.

[61] M. Handley, J. Padhye, and S. Floyd. Tcp congestion window validation. RFC 2861, June 2000.

[62] Garrett Hardin. The tragedy of the commons. *Science*, 162(3859):1243–1248, December 1968.

[63] G. Hasegawa, K. Kurata, and M. Murata. Analysis and improvement of fairness between tcp reno and vegas for deployment of tcp vegas to the internet. In *ICNP*, 2000.

[64] T.R. Henderson, E. Sahouria, S. McCanne, and R.H. Katz. On improving the fairness of tcp congestion avoidance. In *GLOBECOM*, 1998.

[65] U. Hengartner, J. Bolliger, and Th. Gross. Tcp vegas revisited. In *INFOCOM*, 2000.

[66] J. C. Hoe. Improving the startup behavior of a congestion control scheme for tcp. In *SIGCOMM*, 1996.

[67] H. Hsieh, K. Kim, Y. Zhu, and R. Sivakumar. A receiver-centric transport protocol for mobile hosts with heterogeneous wireless interfaces. In *MOBICOM*, 2003.

[68] Xiaomeng Huang, Chuang Lin, and Fengyuan Ren. A novel high speed transport protocol based on explicit virtual load feedback. *Computer Networks*, 51:1800–1814, 2007.

[69] V. Jacobson, R. Braden, and D. Borman. Tcp extensions for high performance. RFC 1323, May 1992.

[70] Van Jacobson. Congestion avoidance and control. In *SIGCOMM*, 1988.

[71] 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, August 2003.

[72] R. Jain, Dah-Ming W. Chiu, and William R. Hawe. A quantitative measure of fairness and discrimination for resource allocation in shared computer systems. Technical report, DEC, 1984.

[73] Raj Jain. A delay-based approach for congestion avoidance in interconnected heterogeneous computer networks. *Computer Communication Review*, 19:56–71, October 1989.

[74] Wenyu Jiang. Detecting and measuring asymmetric links in an ip network. Technical Report CUCS009 -99, Columbia University, 1999.

[75] Cheng Jin, David X. Wei, and Steven H. Low. Fast tcp: motivation, architecture, algorithms, performance. In *INFOCOM*, 2004.

[76] Lampros Kalampoukas, Anujan Varma, and K. K. Ramakrishnan. Explicit window adaptation: A method to enhance tcp performance. *IEEE/ACM Transactions on Networking*, 10:338–350, June 2002.

[77] Kazumi Kaneko, Tomoki Fujikama, Zhou Su, and Jiro Katto. Tcp-fusion: A hybrid congestion control algorithm for high-speed networks. In *PFLDnet Workshop*, 2007.

[78] Aditya Karnik and Anurag Kumar. Performance of tcp congestion control with explicit rate feedback. *IEEE/ACM Transactions on Networking*, 13:108–120, February 2005.

[79] D. Katabi, M. Handley, and C. Rohrs. Internet congestion control for future high bandwidth-delay product environments. In *SIGCOMM*, 2002.

[80] Tom Kelly. Scalable tcp: improving performance in highspeed wide area networks. *Computer Communication Review*, 33:83–91, 2003.

[81] S. Kent and R. Atkinson. Security architecture for the internet protocol. RFC 2401, November 1998.

[82] Vishnu Konda and Jasleen Kaur. Rapid: Shrinking the congestion-control timescale. In *INFOCOM*, 2009.

[83] Rajesh Krishnan, James P. G. Sterbenz, Wesley M. Eddy, Craig Partridge, and Mark Allman. Explicit transport error notification (eten) for error-prone wireless and satellite networks. *Computer Networks*, 46, October 2004.

[84] Joanna Kulik, Robert Coulter, Dennis Rockwell, and Craig Partridge. A simulation study of paced tcp. Technical report, NASA, January 2000.

[85] Aleksandar Kuzmanovic and Edward W. Knightly. Tcp-lp: A distributed algorithm for low priority data transfer. In *INFOCOM*, 2003.

[86] Aleksandar Kuzmanovic and Edward W.Knightly. A performance vs. trust perspective in the design of end-point congestion control protocols. In *ICNP*, 2004.

[87] R. La, J. Walrand, and V. Anantharam. Issues in tcp vegas. Technical Report, ERL, UC Berkeley, 2000.

[88] K. Lakshminarayanan, V.N. Padmanabhan, and J. Padhye. Bandwidth estimation in broadband access networks. In *Internet Measurement Conference*, 2004.

[89] H.K. Lee, V. Hall, K.H. Yum, K.I. Kim, and E.J. Kim. Bandwidth estimation in wireless lans for multimedia streaming. In *ICME*, 2006.

[90] S. Lee, S. Banerjee, and B. Bhattacharjee. The case for a multi-hop wireless local area network. In *INFOCOM*, 2004.

[91] D.J. Leith, R.N. Shorten, and Y. Lee. H-tcp: A framework for congestion control in high-speed and long-distance networks. In *PFLDnet Workshop*, 2005.

[92] Douglas J. Leith, Lachlan L. H. Andrew, Tom Quetchenbach, Robert N. Shorten, and Kfir Lavi. Experimental evaluation of delay/loss-based tcp congestion control algorithms. In *PFLDnet Workshop*, 2008.

[93] Yee-Ting Li, Douglas Leigh, and Robert N. Shorten. Experimental evaluation of tcp protocols for high-speed networks. *IEEE/ACM Transactions on Networking*, 15, October 2007.

[94] Shao Liu, Tamer Basar, and R. Srikant. Tcp-illinois: A loss and delay-based congestion control algorithm for high-speed networks. In *ValueTools*, 2006.

[95] Steven H. Low, Larry L. Peterson, and Limin Wang. Understanding tcp vegas: a duality model. *Journal of the ACM*, 49(2):207–235, March 2002.

[96] R. Ludwig and A. Gurtov. The eifel response algorithm for tcp. RFC 4015, February 2005.

[97] Reiner Ludwig and Randy H. Katz. The eifel algorithm: making tcp robust against spurious retransmissions. *Computer Communication Review*, 30, January 2000.

[98] Changming Ma and Ka-Cheong Leung. Improving tcp reordering robustness in multipath networks. In *LCN*, 2004.

[99] Jim Martin, Arne Nilsson, and Injong Rhee. Delay-based congestion avoidance for tcp. *IEEE/ACM Transactions on Networking*, 11(3):356–369, June 2003.

[100] M. Mathis, J. Mahdavi, S. Floyd, and A. Romanow. Tcp selective acknowledgment options. RFC 2018, 1996.

[101] M. Mathis, J. Semske, J. Mahdavi, and T. Ott. The macroscopic behavior of the tcp congestion avoidance algorithm. *Computer Communication Review*, 27, July 1997.

[102] Yosuke Matsushita, Takahiro Matsuda, and Miki Yamamoto. Tcp congestion control with ack-pacing for vertical handoff. In *WCNC*, 2005.

[103] Jeonghoon Mo, Richard J. La, Venkat Anantharam, and Jean Walrand. Analysis and comparison of tcp reno and vegas. In *INFOCOM*, 1999.

[104] G. Montenegro, S. Dawkins, M. Kojo, V. Magret, and N. Vaidya. Long thin networks. RFC 2757, 2000.

[105] Kashif Munir, Michael Welzl, and Dragana Damjanovic. Linux beats windows! or the worrying evolution of tcp in common operating systems. In *PFLDnet Workshop*, 2007.

[106] Myo Myint. A reconfigurable transport service for converged networks. Honours Year Project Report, National University of Singapore, 2006.

[107] Andrew Odlyzko. Data networks are lightly utilized, and will stay that way. *Review of Network Economics*, 2(3):210–237, September 2003.

[108] J. Padhye, V. Firoiu, D. Towsley, and J. Kurose. Modeling tcp throughput: A simple model and its empirical validation. In *SIGCOMM*, 1998.

[109] C. Parsa and J.J. Garcia-Luna-Aceves. Differentiating congestion vs. random loss: a method for improvingtcp performance over wireless links. In *WCNC*, 2000.

[110] Anders Persson, Cesar A.C. Marcondes, Ling-Jyh Chen, Li Lao, M.Y. Sanadidi, and Mario Gerla. Tcp probe: A tcp with built-in path capacity estimation. In *IEEE Global Internet Symposium in conjunction with INFOCOM*, 2005.

[111] J. Postel. Internet protocol. RFC 791, September 1981.

[112] Jon Postel. Transmission control protocol - darpa internet program protocol specification. RFC 793, September 1981.

[113] Ravi S. Prasad, Manish Jain, and Constantinos Dovrolis. On the effectiveness of delay-based congestion avoidance. In *PFLDnet Workshop*, 2004.

[114] K. Ramakrishnan, S. Floyd, and D. Black. The addition of explicit congestion notification (ecn) to ip. RFC 3168, September 2001.

[115] Vinay J. Ribeiro, Rudolf H. Riedi, Jiri Navratil, Les Cottrell, and Richard G. Baraniuk. pathchirp: Efficient available bandwidth estimation for network paths. In *Proceedings Workshop on Passive and Active Measurement*, 2003.

[116] Luigi Rizzo. Dummynet: a simple approach to the evaluation of network protocols. *Computer Communication Review*, 27(1):31–41, 1997.

[117] R.K.Balan, B.P.Lee, K.R.R.Kumar, L.Jacob, W.K.G.Seah, and A.L.Ananda. Tcp hack: a mechanism to improve performance over lossy links. *Computer Networks*, 39:347–361, 2002.

[118] D. Schmidt, D. Box, and T. Suda. Adaptive: A dynamically assembled protocol transformation, integration, and evaluation environment. *Concurrency: Practice and Experience*, 5, June 1993.

[119] Robert Shorten, Fabian Wirth, and Douglas Leith. A positive systems model of tcp-like congestion control: asymptotic results. *IEEE/ACM Transactions on Networking*, 14, June 2006.

[120] Hla Win Soe. Implementing congestion control framework for reactive tcp project. Honours Year Project Report, National University of Singapore, 2007.

[121] Mark Stemm and Randy H. Katz. Vertical handoffs in wireless overlay networks. *Mobile Notworks and Applications*, 3, 1998.

[122] Pradeep Sudame and B.R. Badrinath. On providing support for protocol adaptation in mobile wireless networks. *Mobile Networks and Applications*, 6:43–55, January 2001.

[123] Yutaka Sugawara, Takeshi Yoshino, Mary Inaba, and Kei Hiraki. Fine tune for parallel tcp streams on long fat-pipe network using hardware engine. In *PFLDnet Workshop*, 2008.

[124] Kun Tan, Jingmin Song, and Murari Sridharan. Ctcp-tube: Improving tcp friendliness over low buffered network links. In *PFLDnet Workshop*, 2008.

[125] Kun Tan, Jingmin Song, Qian Zhang, and Murari Sridharan. A compound tcp approach for high-speed and long distance networks. In *INFOCOM*, 2006.

[126] Klein T.E., Leung K.K., Parkinson R., and Samuel L.G. Avoiding spurious tcp timeouts in wireless networks by delay injection. In *GLOBECOM*, 2004.

[127] K. Tsukamoto, Y. Fukuda, Y. Hori, and Y. Oie. New flow control schemes of tcp for multimodal mobile hosts. In *VTC-Spring*, 2003.

[128] S. Vasudevany, K. Papagiannakiz, C. Diotz, J. Kurosey, and D. Towsley. Facilitating access point selection in ieee 802.11wireless networks. In *Internet Measurement Conference*, 2005.

[129] Chetan Ganjihal Veerabhadrappa. Experimental evaluation of synctcp and other high-speed congestion control algorithms. Master Thesis, National University of Singapore, 2009.

[130] Arun Venkataramani, Ravi Kokku, and Mike Dahlin. Tcp nice: A mechanism for background transfers. In *OSDI*, 2002.

[131] Ren Wang, M.Yahya Sanadidi, and Mario Gerla. Tcp with sender-side intelligence to handle dynamic, large, leaky pipes. *IEEE Journal on Selected Areas in Communications*, 23:235–248, 2005.

[132] Ren Wang, Massimo Valla, M.Y. Sanadidi, Bryan Ng, and Mario Gerla. Efficiency/friendliness tradeoffs in tcp westwood. In *IEEE Symposium on Computers and Communications*, 2002.

[133] David X. Wei and Pei Cao. Ns-2 tcp-linux: An ns-2 tcp implementation with congestion control algorithms from linux. In *ACM ValueTools - Workshop of NS-2*, 2006.

[134] G. Wright and W. Richard Stevens. *TCP/IP Illustrated, Volume 2: The Implementation.* Addison-Wesley, 1995.

[135] G. Wu and T. Chiueh. Passive and accurate traffic load estimation for infrastructure-mode wireless lan. In *ACM MSWIM*, 2007.

[136] Xiuchao Wu. A simulation study of compound tcp. Technical Report, School of Computing, National University of Singapore, 2007.

[137] Xiuchao Wu. Effects of applying high speed congestion control algorithms in the internet. Technical Report, School of Computing, National University of Singapore, 2008.

[138] Xiuchao Wu. Overview of freebsd 7 tcp implementation. Technical Report, School of Computing, National University of Singapore, 2009.

[139] Xiuchao Wu and A. L. Ananda. Link characteristics estimation for ieee 802.11 dcf based wlan. In *LCN*, 2004.

[140] Xiuchao Wu, Indradeep Biswas, Mun Choon Chan, and A. L. Ananda. Utilizing characteristics of last link to improve tcp performance. In *IPCCC*, 2005.

[141] Xiuchao Wu, Mun Choon Chan, and A. L. Ananda. Effects of applying high-speed congestion control algorithms in satellite network. In *ICC*, 2008.

[142] Xiuchao Wu, Mun Choon Chan, A. L. Ananda, and Chetan Ganjihal. Sync-tcp: A new approach to high speed congestion control. In *ICNP*, 2009.

[143] Xiuchao Wu, Mun Choon Chan, and A.L. Ananda. Improving tcp performance in heterogeneous mobile environments by exploiting the explicit cooperation between server and mobile host. *Computer Networks*, 52, November 2008.

[144] Lisong Xu, Khaled Harfoush, and Injong Rhee. Binary increase congestion control for fast long distance networks. In *INFOCOM*, 2004.

[145] Raj Yavatkar and Namrata Bhagawat. Improving end-to-end performance of tcp over mobile intemetworks. In *IEEE Worhhop on Mobile Computing Systems and Applications*, 1994.

[146] Y.R.Yang and S.S.Lam. General aimd congestion control. In *ICNP*, 2000.

[147] Jian Zhang, Liang Cheng, and Ivan Marsic. Models for non-intrusive estimation of wireless link bandwidth. In *Personal Wireless Communication Conference*, September 2003.

[148] Y. Zhang, N. Duffield, Vern Paxson, and S. Shenker. On the constancy of internet path properties. In *Internet Measurement Workshop*, 2001.

Appendix A

Additional Simulation Results of Sync-TCP

A.1 Scalability of Sync-TCP

In the following simulations, dumbbell topology (figure 3.6) and block scenario (figure 3.7) are always used. Web-like cross traffics described at the beginning of section 3.5 are also generated. The number of competing flows, N, is fixed to 2. Propagation delay of the side links are all set to 5ms so that all competing flows have the same RTPD. Through varying propagation delay, queue size, or packet loss rate of the bottleneck link, scalability of Sync-TCP is investigated further.

A.1.1 Scalability with Propagation Delay

In this group of experiments, the bottleneck link is configured as per=10^{-6} and $qsize$=0.5BDP. The varying parameter is *delay*, and it is set to 25ms, 50ms, 100ms, and 200ms. Figure A.1 shows the results which indicate that Sync-TCP performs the best in almost all metrics, independent of the value of round trip time.

Interestingly, when *delay* is large, Fast TCP performs quite bad, especially in the metric of link utilization ratio. The possible reason is that with $per = 10^{-6}$, there are still some corrupted segments. When segment loss (due to corruption) is detected, Fast TCP always reduces *cwnd* by half and *cwnd* is then increased by at most γ packets per round trip time. With the increase of *delay*, BDP of the network pipe will be increased, and Fast TCP flows cannot take back network bandwidth quickly enough. Hence, link utilization ratio of Fast TCP becomes lower when *delay* is large. As shown in figure A.2, when *per* is very small (10^{-8}), Fast TCP can efficiently utilize the bottleneck link, irrespective of the value of *delay*. Hence, the above conjecture is confirmed.

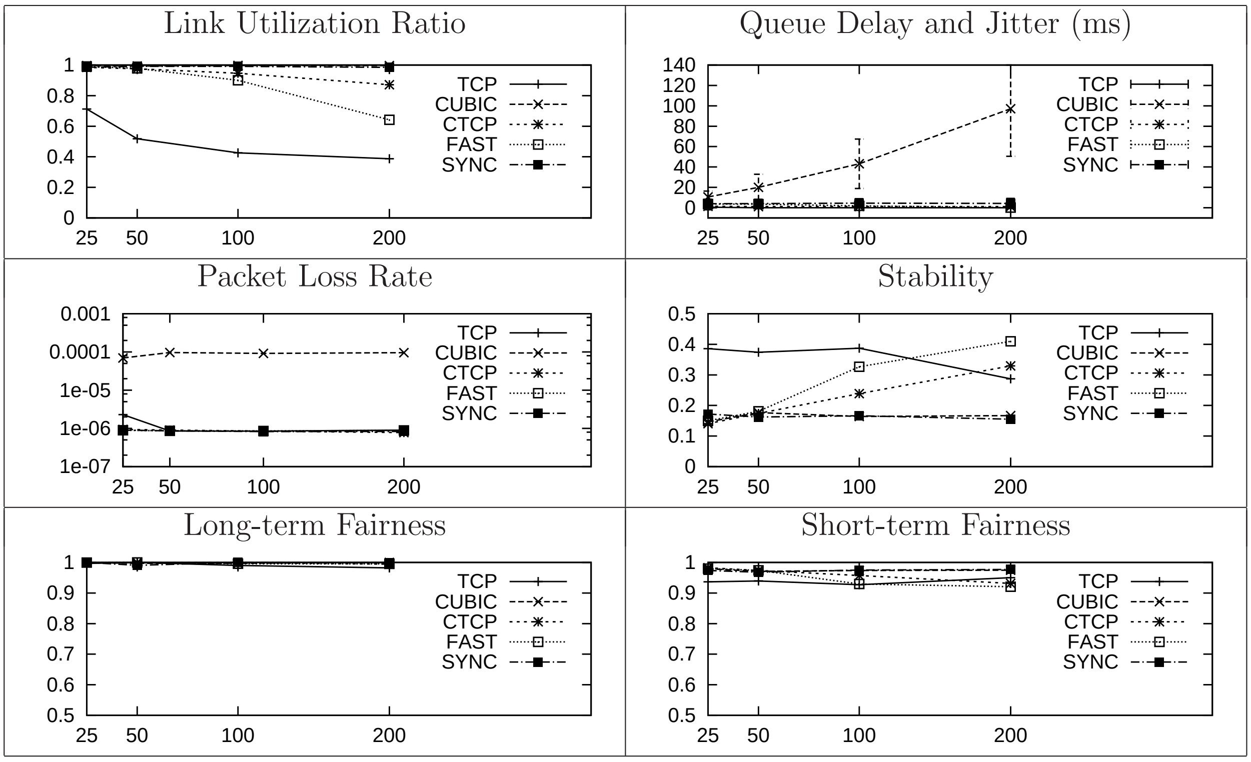


Figure A.1: Scalability with Propagation Delay ($per=10^{-6}$)

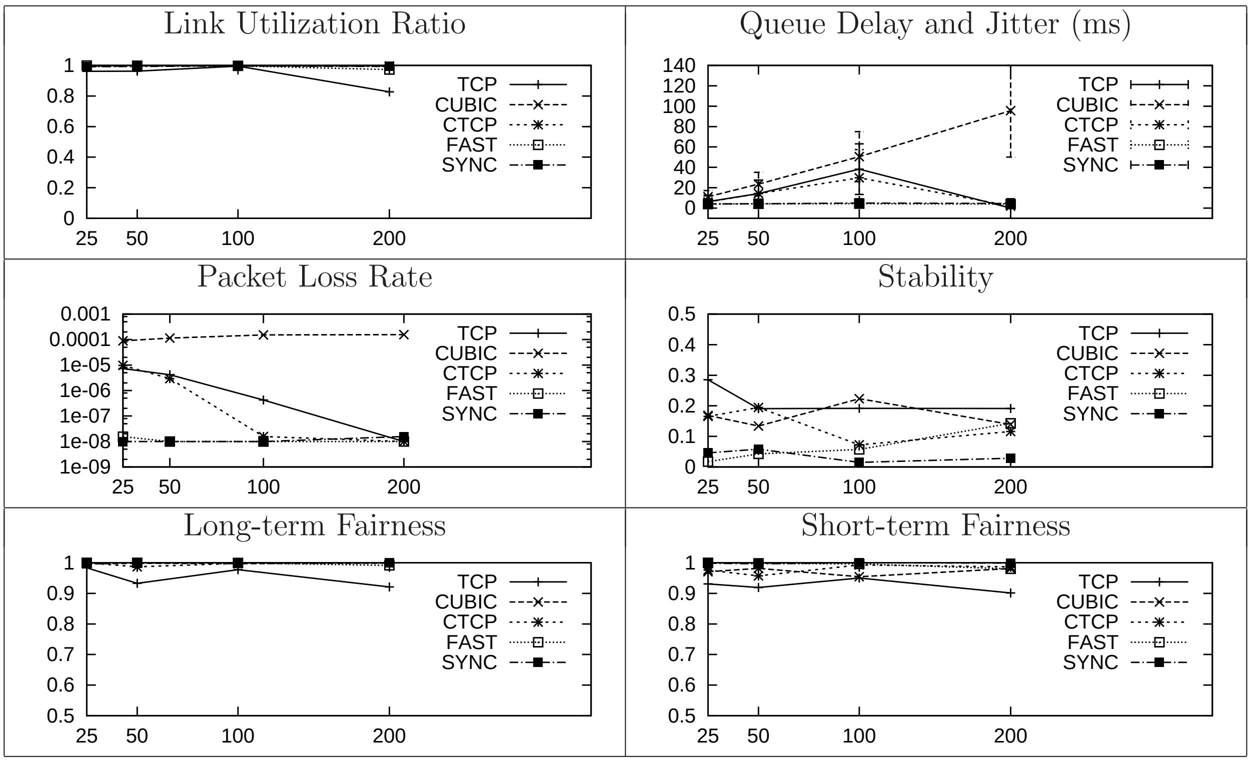


Figure A.2: Scalability with Propagation Delay ($per=10^{-8}$)

A.1.2 Scalability with Queue Size

In this group of experiments, the bottleneck link is configured as $delay$=50ms and per=10^{-6}. $qsize$ is the varying parameter, and it is set to 0.02, 0.05, 0.1, 0.2, 0.5, and 1.0 BDP of the bottleneck link.

Figure A.3 indicates that Sync-TCP still can perform quite well even when queue is small and queue delay cannot be larger than Th_{qd}. It is not worse than other algorithms except in the metric of packet loss rate. The reason of higher packet loss rate is that congestion cannot be detected through queue delay, $\frac{Th_{qd}-qd}{Th_{qd}}$ cannot effectively reduce α when buffer overflow approaches, and many segments are dropped in each congestion event.

Considering that queue delay and jitter cannot be large when queue size is small, it may be better to use Cubic-TCP since it can efficiently utilize the bottleneck link and maintain a low packet loss rate. Hence, a flow may switch between Sync-TCP and Cubic-TCP based on whether it can observe queue delay that is larger than Th_{qd}.

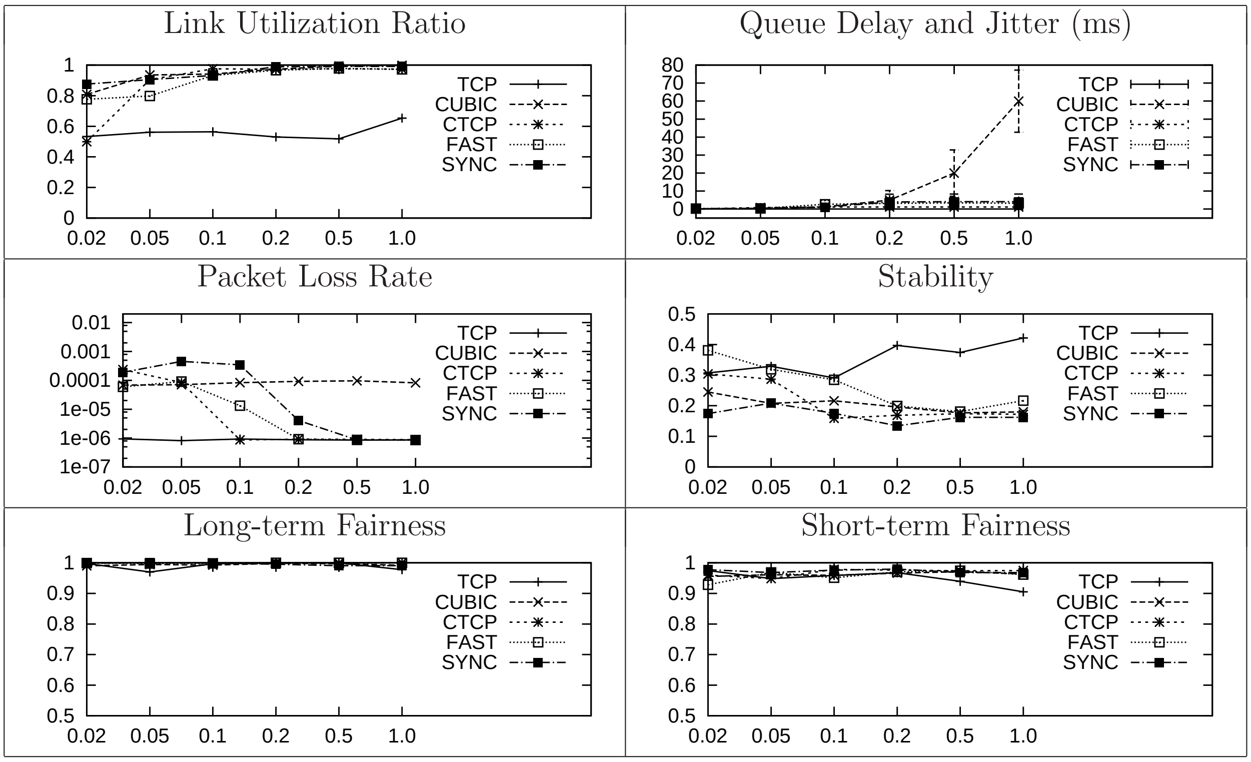


Figure A.3: Scalability with Queue Size

A.1.3 Scalability with Packet Loss Rate

In this group of experiments, the bottleneck link is configured as *delay*=50ms and *qsize*=0.5BDP. *per* is the varying parameter, and it is set to 10^{-8}, 10^{-7}, 10^{-6}, 10^{-5}, 10^{-4}, and 10^{-3}.

Figure A.4 indicates that Sync-TCP can perform well with the increase of *per*. When *per* is quite high, although Sync-TCP flows may not be able to detect congestion simultaneously through queue delay, they still can utilize the bottleneck link quite efficiently. In other metrics, its performance is not obviously worse than other HSCC algorithms.

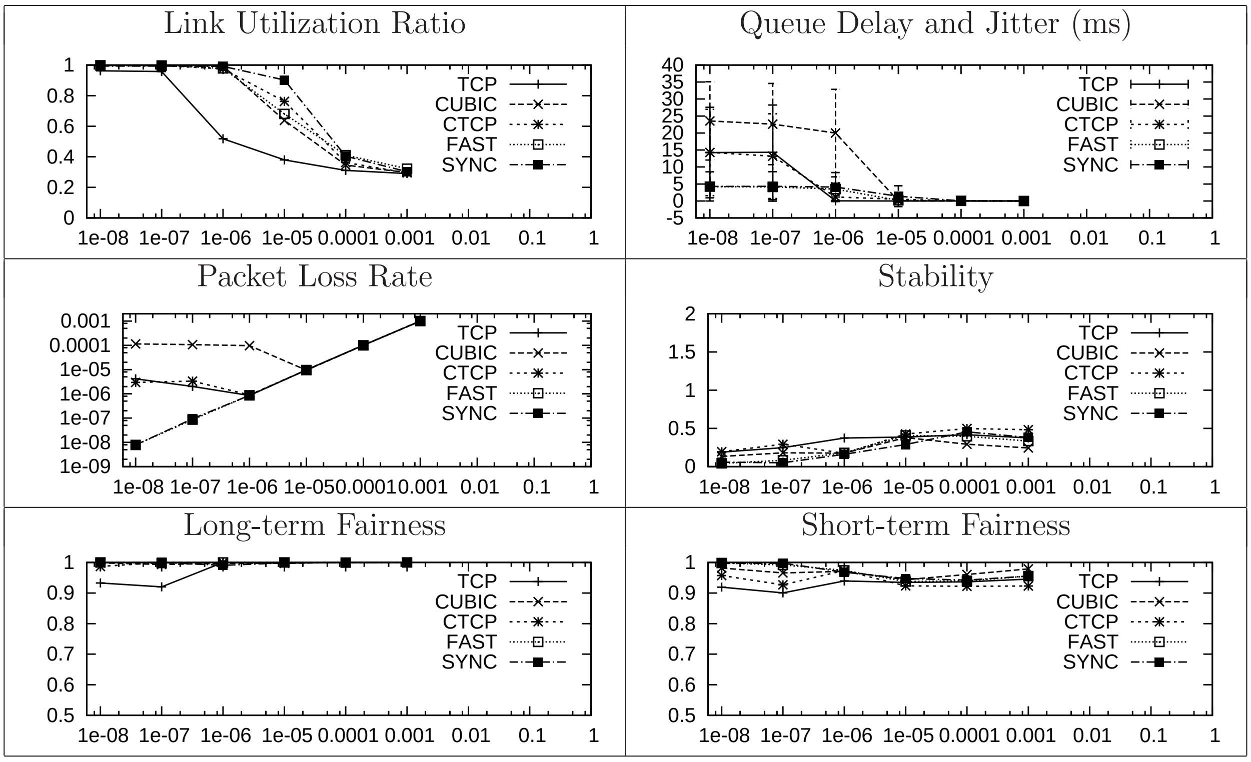
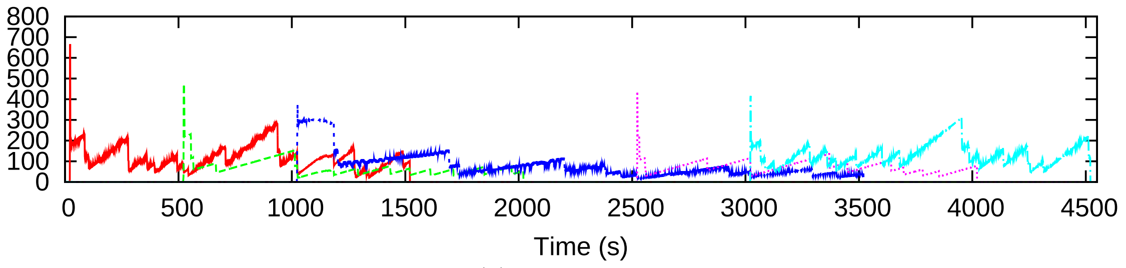


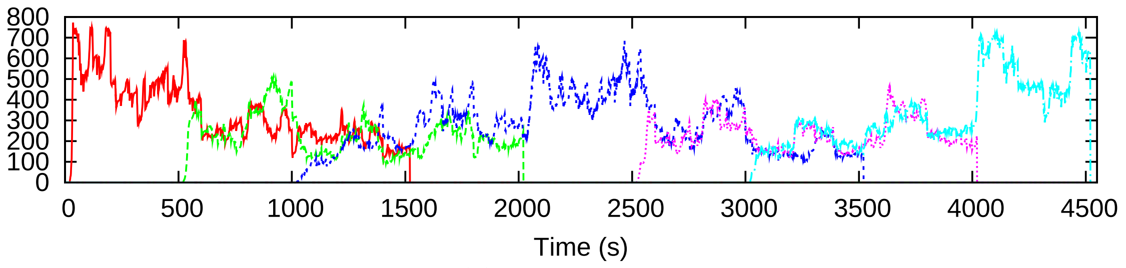
Figure A.4: Scalability with Packet Loss Rate

A.2 Door and Tower Scenarios with Varying Background Traffic

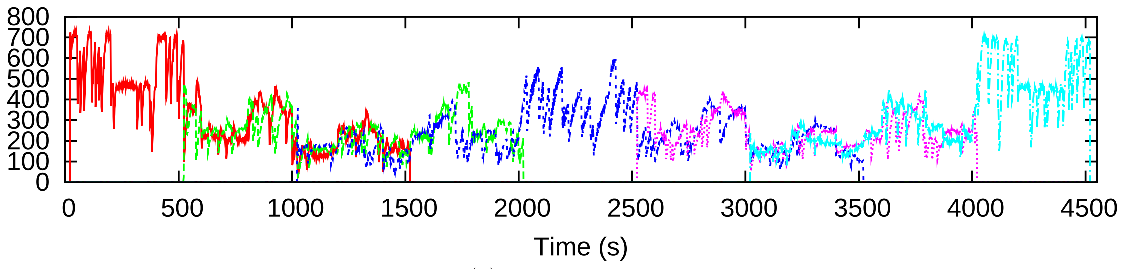
In order to evaluate Sync-TCP in more dynamically environments, experiments in subsection 3.5.5 are repeated with varying background traffic. Except the background traffic generated by web surfing (described at the beginning of section 3.5), a 240Mbps CBR (Constant Bit Rate) UDP flow is turned on/off alternately. More specifically, the length of each active/deactive period is 200 seconds, which is close to the temporal constancy of available bandwidth in the Internet [148]. The results shown in figure A.5-A.7 and figure A.8-A.10 indicate that, Sync-TCP can maintain its merits even when the load of cross traffic is also varying significantly.



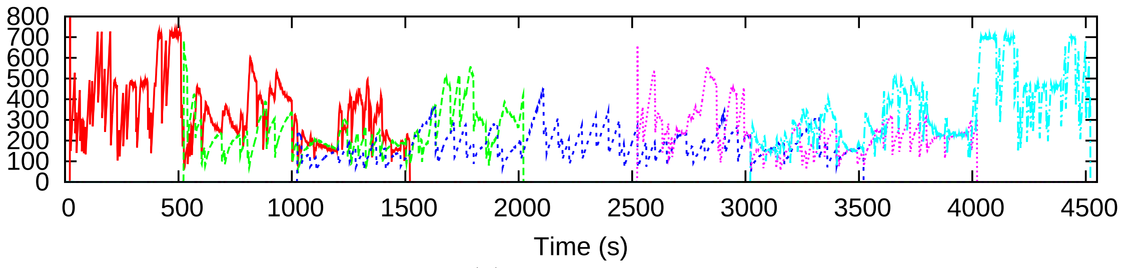
(a) TCP



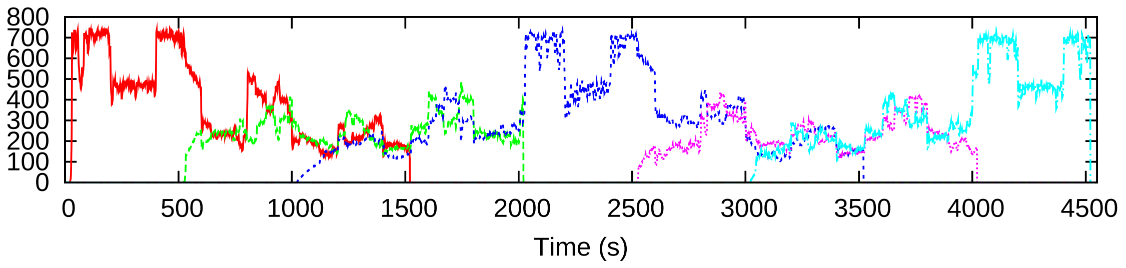
(b) CUBIC



(c) CTCP

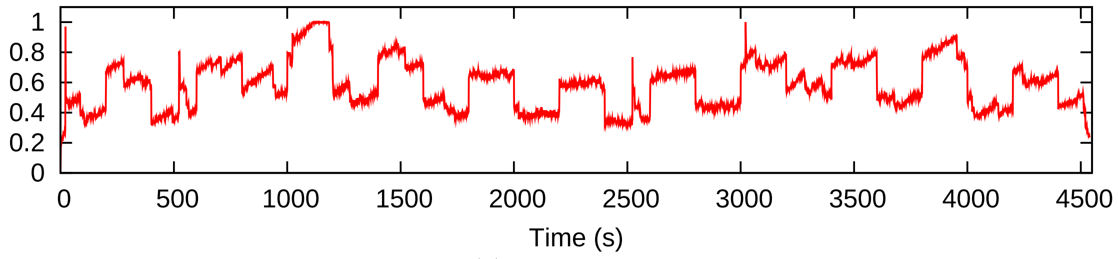


(d) FAST

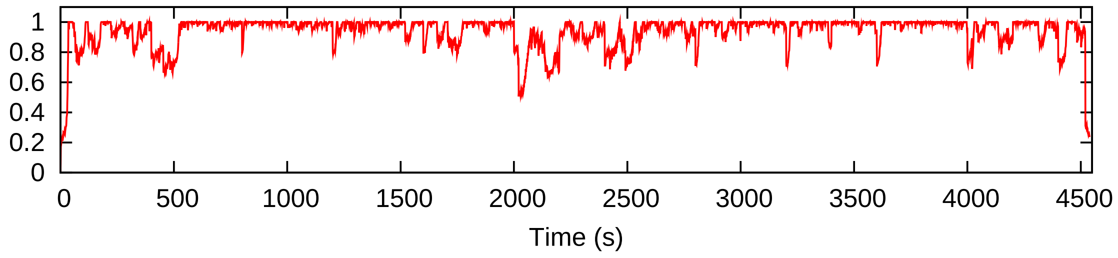


(e) SYNC

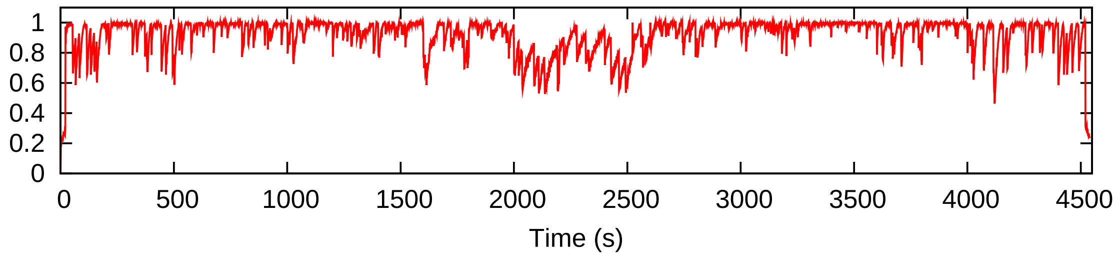
Figure A.5: Door Scenario with Varying Background Traffic: Throughput Trajectories of All Competing Flows (Mbps)



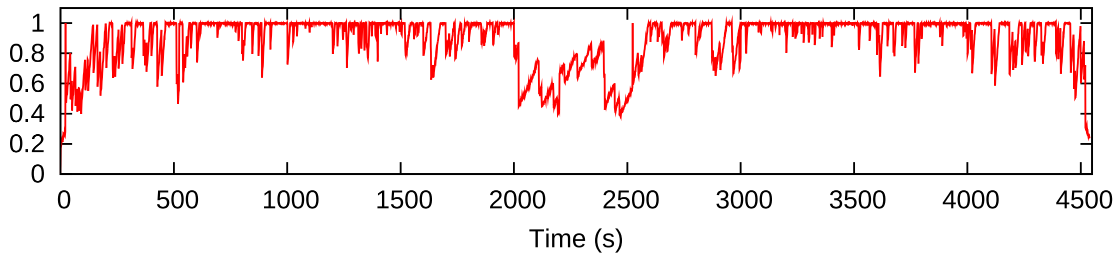
(a) TCP



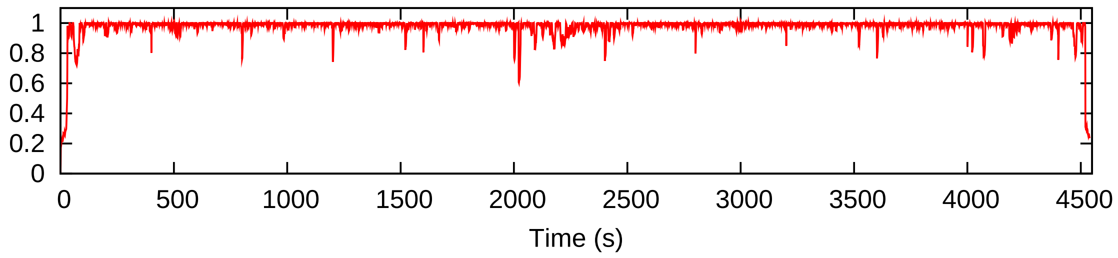
(b) CUBIC



(c) CTCP

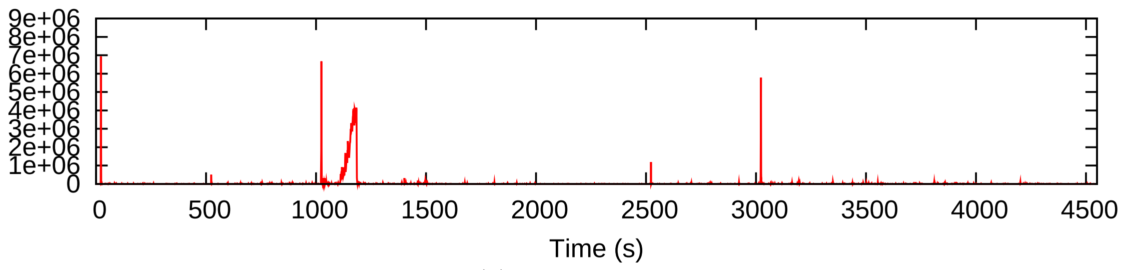


(d) FAST

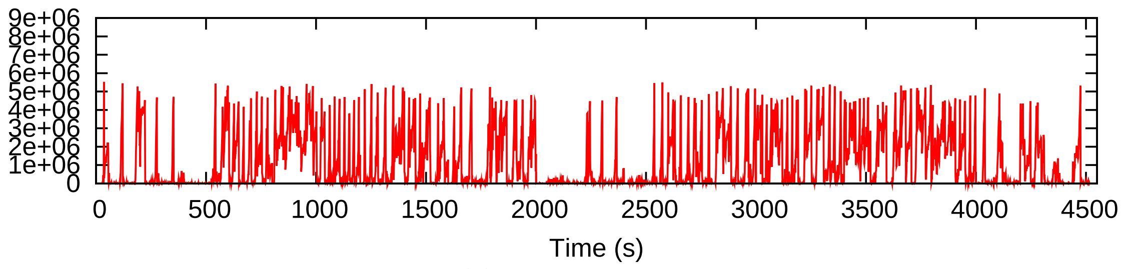


(e) SYNC

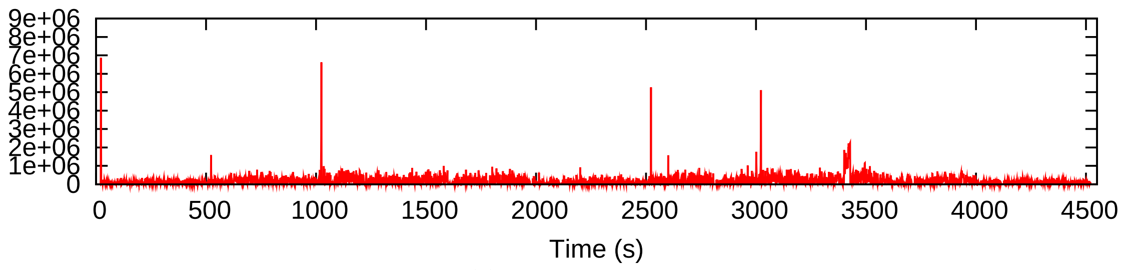
Figure A.6: Door Scenario with Varying Background Traffic: Utilization Ratio of the Bottleneck Link



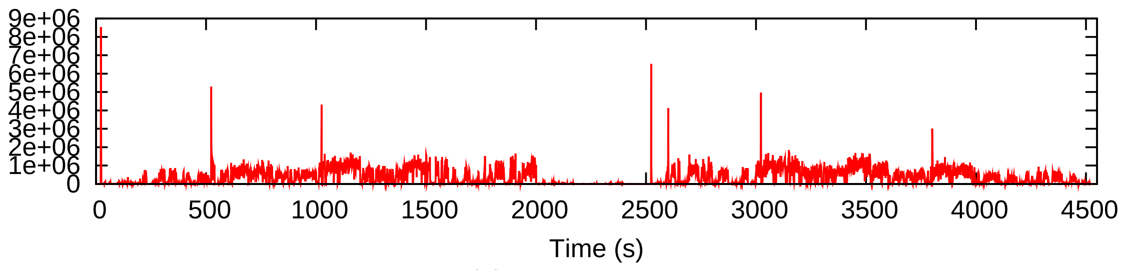
(a) TCP



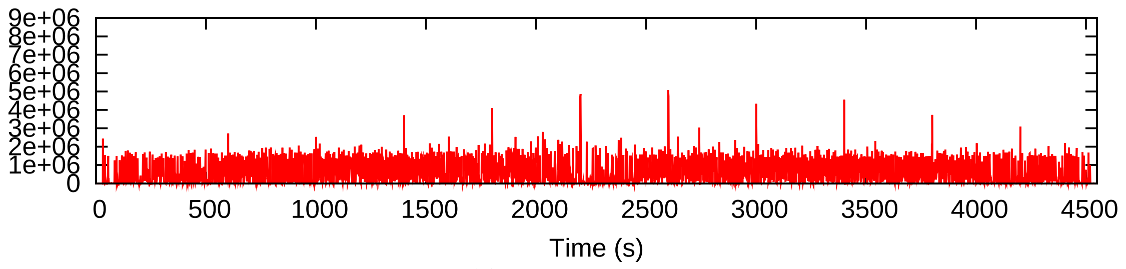
(b) CUBIC



(c) CTCP

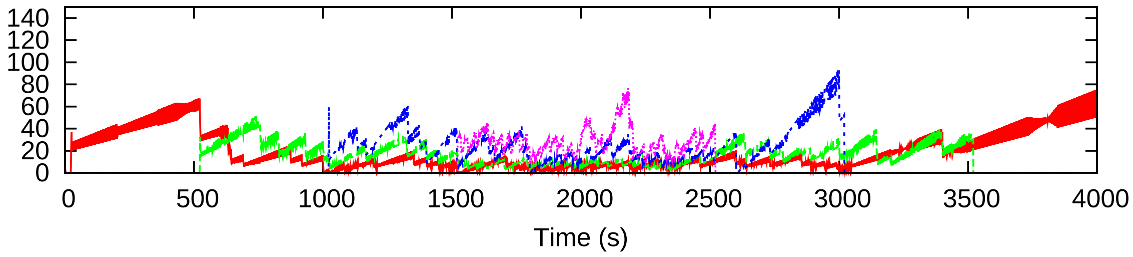


(d) FAST

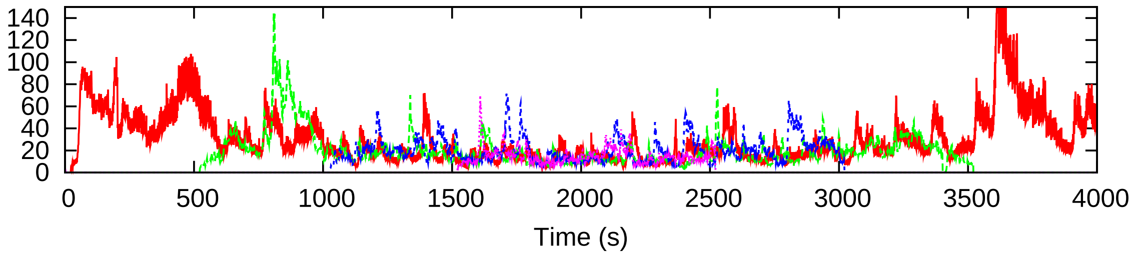


(e) SYNC

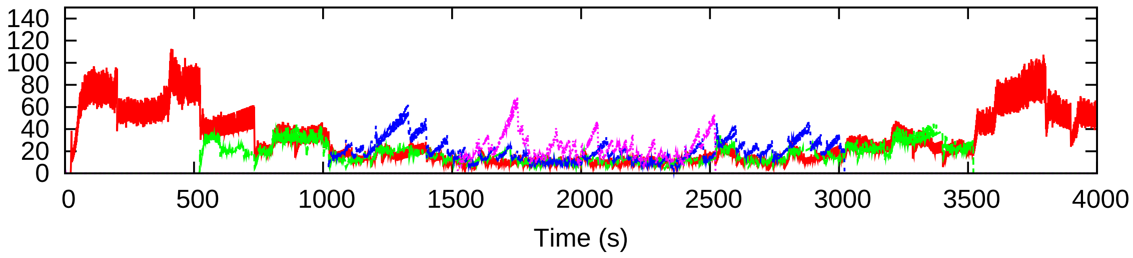
Figure A.7: Door Scenario with Varying Background Traffic: Queue Dynamics at the Bottleneck Link (byte)



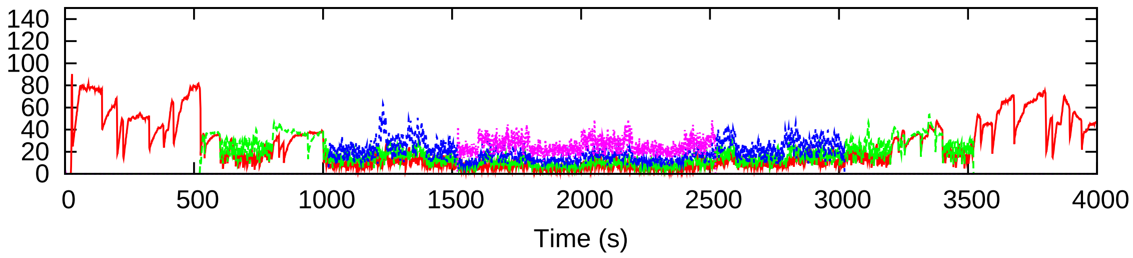
(a) TCP



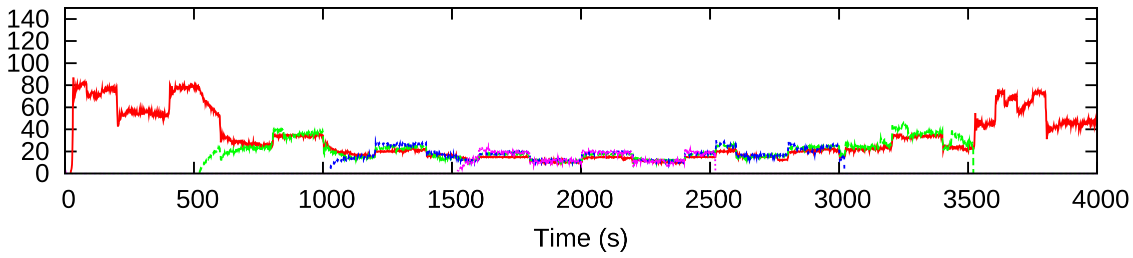
(b) CUBIC



(c) CTCP

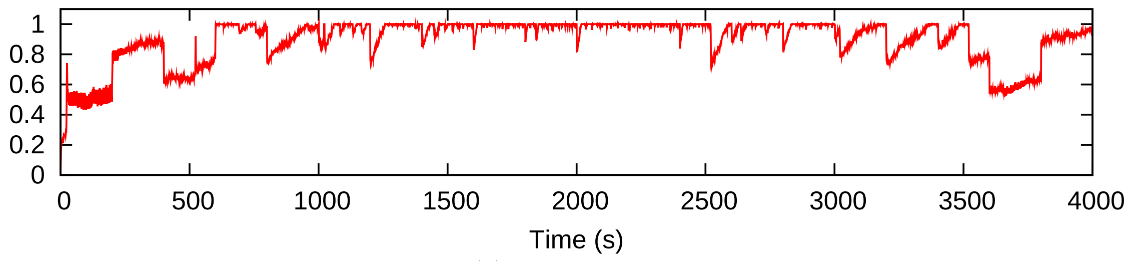


(d) FAST

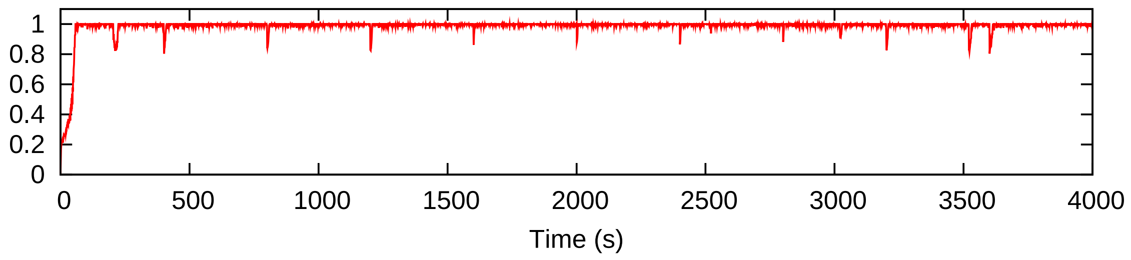


(e) SYNC

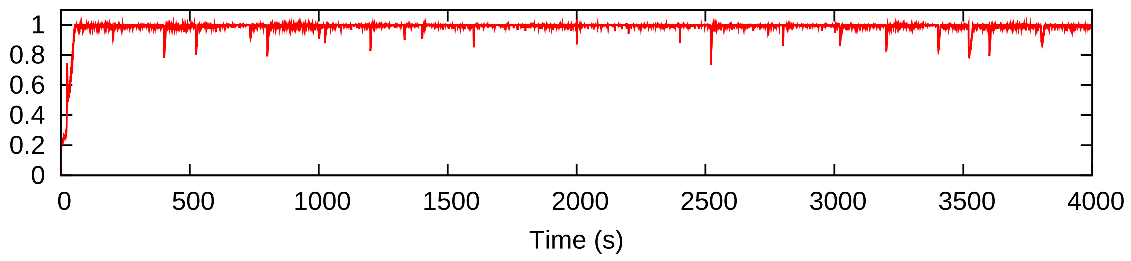
Figure A.8: Tower Scenario with Varying Background Traffic: Throughput Trajectories of Flows 0, 10, 20, 30 (Mbps)



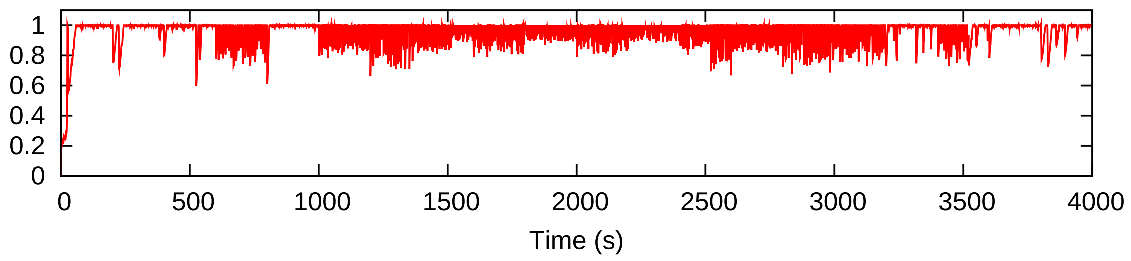
(a) TCP



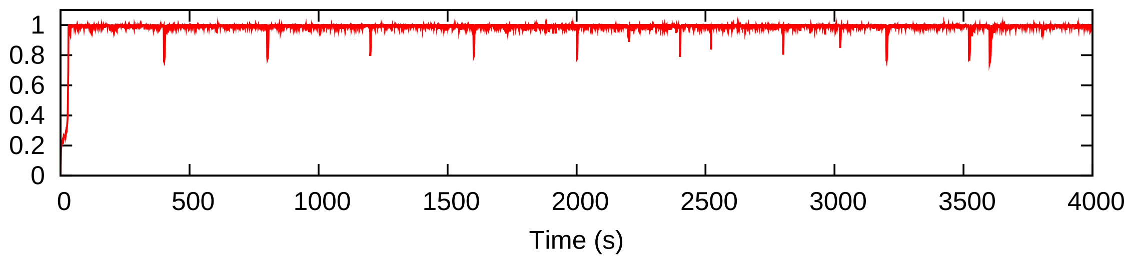
(b) CUBIC



(c) CTCP

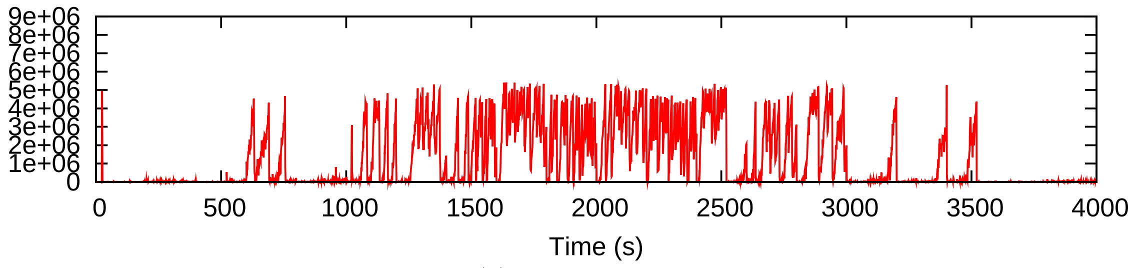


(d) FAST

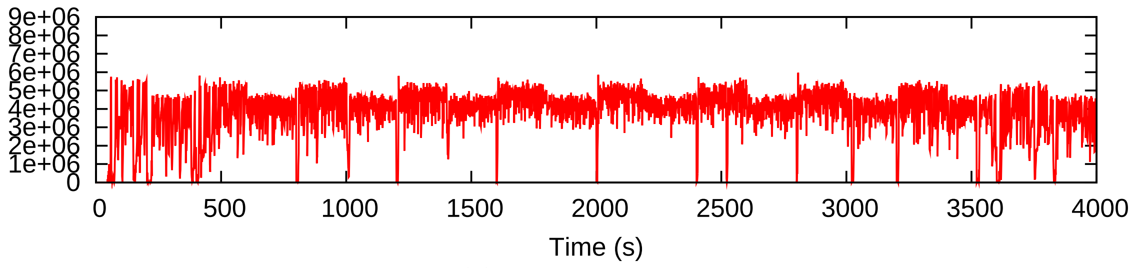


(e) SYNC

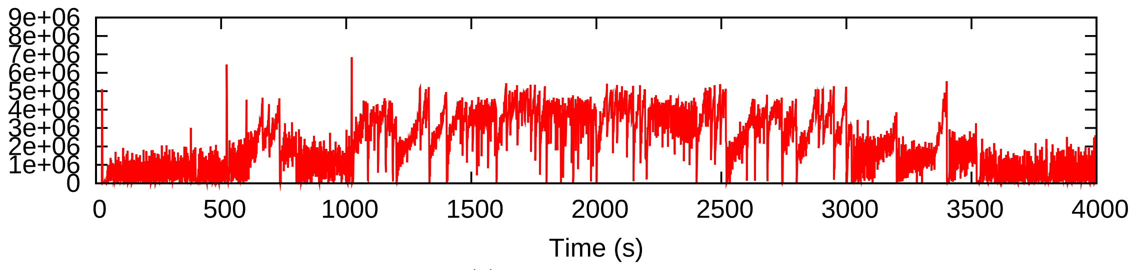
Figure A.9: Tower Scenario with Varying Background Traffic: Utilization Ratio of the Bottleneck Link



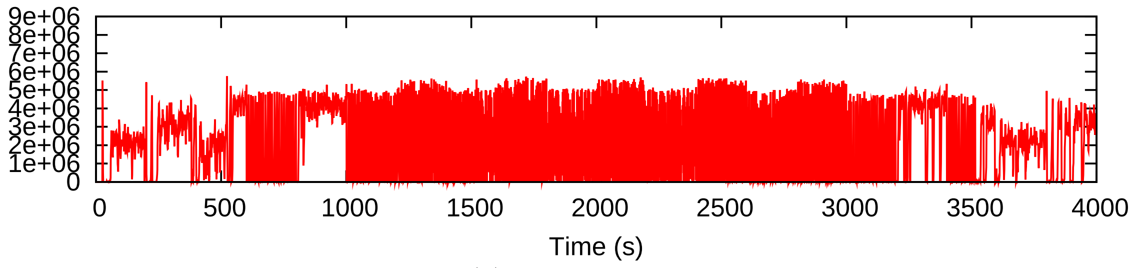
(a) TCP



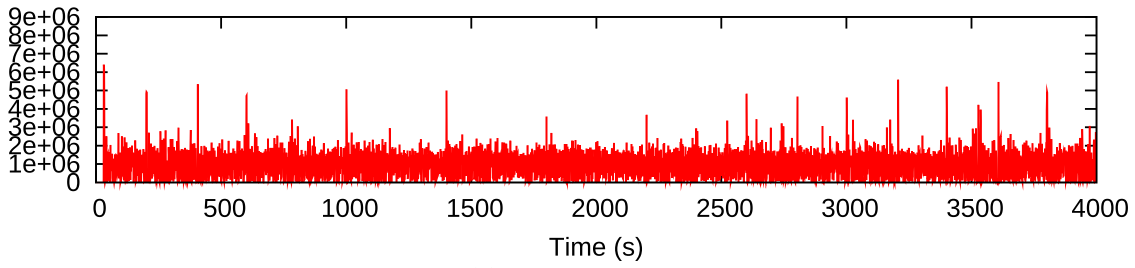
(b) CUBIC



(c) CTCP



(d) FAST



(e) SYNC

Figure A.10: Tower Scenario with Varying Background Traffic: Queue Dynamics at the Bottleneck Link (byte)

A.3 Multiple Congested Links Fairness

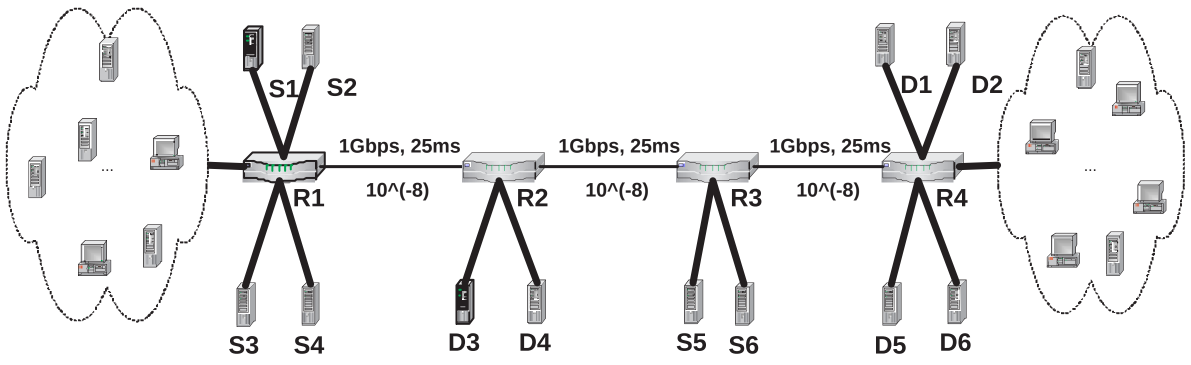


Figure A.11: Parking-lot Network Topology

In order to investigate MCL unfairness of Sync-TCP, a typical parking-lot topology shown in figure A.11 is used. Web-like cross traffics (described at the beginning of section 3.5) are also generated between the two clouds. The flow arrival and departure sequence shown in block scenario (figure 3.7) is used too, and six flows are generated between S_i and D_i.

In the park-lot topology, flow 1 and 2 are the two flows that pass through multiple (two) congested links, the link between R_1 and R_2 and the link between R_3 and R_4. As for flow 3, 4, 5, and 6, they pass through only one of the two congested links.

	TCP	CUBIC	CTCP	FAST	SYNC
Flow 1 and 2	55.5	88.8	92.6	104.3	14.8
Flow 3 and 4	199.1	268.9	260.5	253.2	336.6
Flow 5 and 6	232.7	268.7	262.1	253.7	336.4

Table A.1: Average Throughput (Mbps) of Different Flows

Table A.1 shows the average throughput of different kinds of flows when different congestion control algorithms are adopted. It indicates that, in the metric of MCL unfairness, Sync-TCP is even much worse than TCP. The following is a simple explanation.

Sync-TCP detects congestion by comparing queue delay with Th_{qd}. In this experiment, $qd_{flow3} = qd_{r1}$, $qd_{flow5} = qd_{r3}$, and $qd_{flow1} = qd_{r1} + qd_{r3}$. Here, qd_{r1} is the queue delay at the router $R1$ and qd_{r3} is the queue delay at the router $R3$. Flow 1 may detect congestion and reduce *cwnd* even when both flow3 and flow5 do not detect congestion. Hence, congestion signals detected by flow 1 can be much more than the sum of congestion signals detected by flow 3 and flow 5. In addition, flow 1 also increases λ frequently and uses a small β. Consequently, flow 1 receives much less throughput.

The above results are reasonable since flow 1 consumes more network resources for transmitting the same amount of data. In addition, MCL unfairness of Sync-TCP may motivate large content providers, such as YouTube, to deploy more mirrors and reduce the load of core networks. Finally, Sync-TCP will switch back to TCP when throughput is too low. Hence, Sync-TCP flows, which pass through MCLs, will not be totally starved.

A.4 Coexistence with TCP Flows

In this group experiments, we will investigate the coexistence between a HSCC algorithm and TCP. In another word, part of bandwidth-greedy and elastic applications adopt a HSCC algorithm and the others adopt the legacy TCP under the assumption that socket buffer is large enough and window scale option is enabled for supporting high speed data transmission.

Dumbbell topology (figure 3.6) and block scenario (figure 3.7) are still used. The web-like background traffics (described at the beginning of section 3.5) are also generated between the two clouds, and *delay* of side links are all set to 5ms. As for the bottleneck link, *bw*=1Gbps, *delay*=50ms, *qsize*=0.5BDP, and *per* is set to 10^{-8}, 10^{-7}, 10^{-6}, 10^{-5}, 10^{-4}, and 10^{-3}. The number of flows is set to 4, flow 0 and 1 use TCP, and the other two flows use a HSCC algorithm. Figure A.12 shows the average throughput of flow 0 and 1 and the average throughput of flow 2 and 3. The average throughput of the four flows, in the case that they all use TCP, is also plotted.

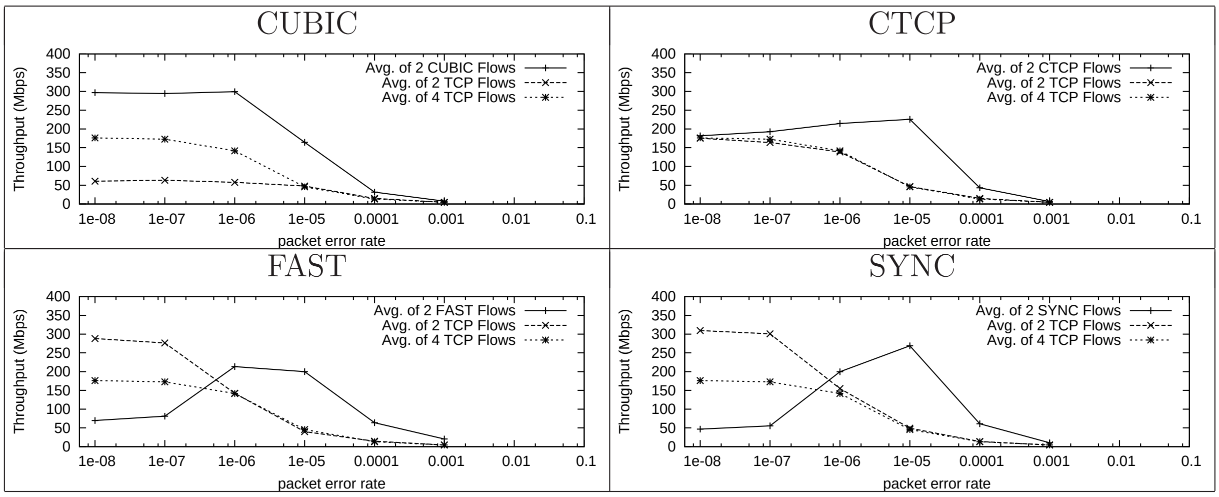


Figure A.12: Coexistence with TCP

Figure A.12 indicates that Cubic-TCP flows always steal a lot of bandwidth from the competing TCP flows. CTCP flows do not steal bandwidth from the competing TCP flows. When *per* is very low, TCP flows cannot steal bandwidth from CTCP flows too since CTCP is designed to never receive less throughput than TCP. When *per* is high, CTCP can also utilize the bandwidth that cannot be utilized by TCP flows. Hence, CTCP performs very good in the aspect of coexistence with TCP. However, the cost is that it cannot drive network to operate around the knee. Their convergence behaviors are also very complex, and competing CTCP flows may not share network resources fairly, especially when their life-span is short.

As for Fast TCP and Sync-TCP, when *per* is very low, TCP flows can steal bandwidth from Fast TCP or Sync-TCP flows. The reason is that TCP is a loss-based congestion control algorithm that only reduces its sending rate when segment loss is detected. Sync-TCP and Fast TCP are delay-based HSCC algorithms which will reduce *cwnd* when the earlier congestion signal, queue delay, is detected.

We should note that the results are measured in the scenario that the load of background traffic is almost constant during the simulation. By activating a 240Mbps CBR UDP flow

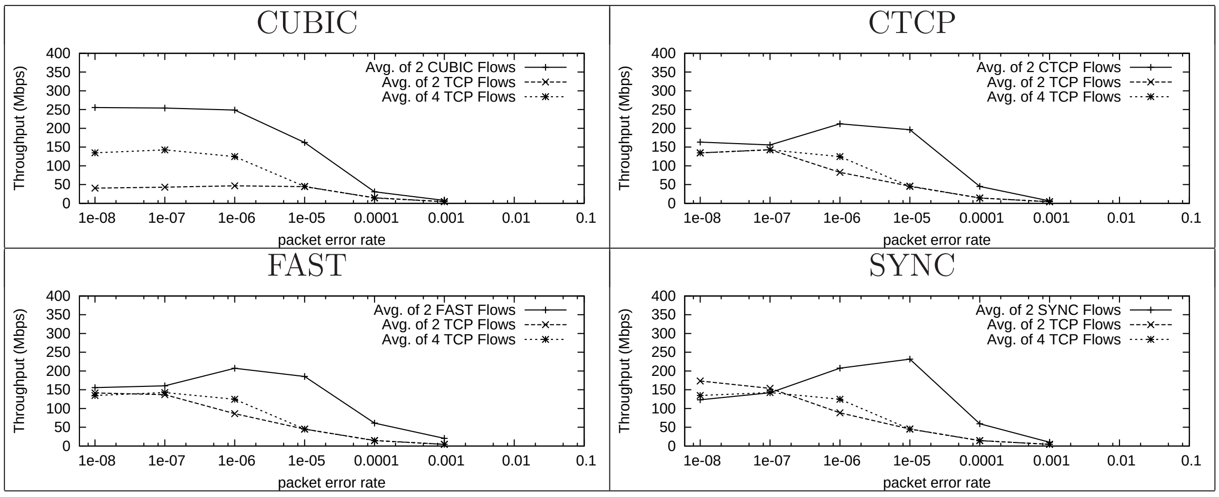


Figure A.13: Coexistence with TCP When the Load of Background Traffic Varies

during the periods ([200,300], [400,500], [600,700], [800,900]), as shown in figure A.13, even when *per* is low, the average throughput of Sync-TCP/Fast TCP flows is still comparable to that of the competing TCP flows. The reason is that when the load of background traffic is reduced, Sync-TCP/Fast TCP can quickly acquire the suddenly increased available bandwidth.

In addition, Sync-TCP will switch back to TCP if throughput is too low. Hence, Sync-TCP flows will not be totally starved by the competing TCP flows.

As for the coexistence with loss-based HSCC algorithms, it has been reported that CTCP can be starved by Bic-TCP [105]. As for Sync-TCP, it should perform even worse than CTCP. According to *The Tragedy of the Commons* in game theory [62], it is impossible for endpoints to simultaneously solve this unfairness issue and drive network to operate around the knee. For solving the unfairness issue, Sync-TCP must also try to drive network to operate around the cliff and it cannot keep its friendliness to cross traffic. Standardization and/or queue mechanisms, such as ZL-RED [63] that catches and punishes loss-based flows, should be adopted for solving this fundamental confliction.

A.5 Cross Traffic and the Value of λ

In Sync-TCP, λ is used to calculate β based on equation 3.9 in subsection 3.4.5. For emptying the queue of the bottleneck link while not under-utilizing the network, λ should be slightly larger than 1. According to analysis in subsection 3.3.2, it is better if λ could be adjusted based on the kind and the load of cross traffic. In this subsection, we will demonstrate the effects of the kind & the load of cross traffic and the necessity of adjusting λ based on network environment.

Dumbbell topology (figure 3.6) and block scenario (figure 3.7) are used here. As for the bottleneck link, *bw*=1Gbps, *delay*=50ms, *qsize*=1.0BDP, and *per* is set to 10^{-6}. The number of competing Sync-TCP flows is set to 16 and *delay* of side links are all set to 5ms.

In the first group of experiments, cross traffic is generated by web surfing, and the load

Time (s)	[0:200]	[200:400]	[400:600]	[600:800]	[800:1000]
HTTP Transactions per second)	3200	6400	9600	6400	3200
Approximate Data Rate (Mbps)	300	600	900	600	300

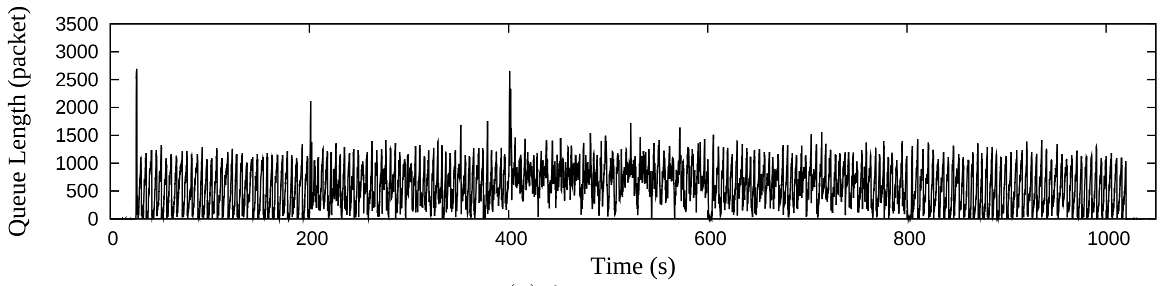
Table A.2: The Load of Cross Traffic Generated by Web Surfing

Time (s)	[0:200]	[200:400]	[400:600]	[600:800]	[800:1000]
Number of Legacy FTP Flow	50	100	150	100	50
Approximate Data Rate (Mbps)	240	480	720	480	240

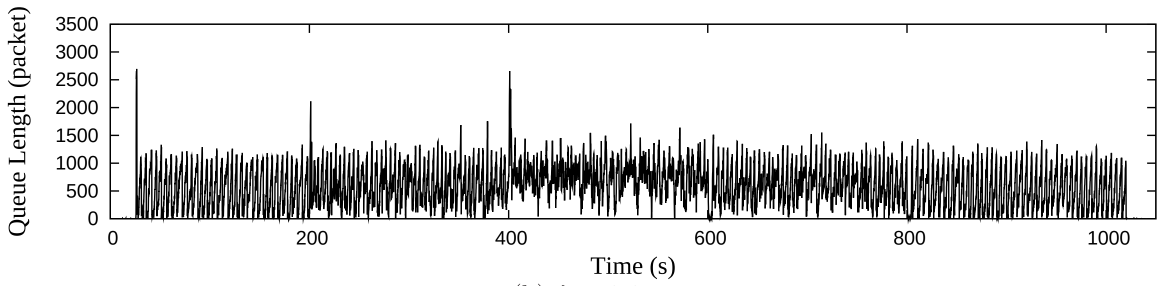
Table A.3: The Load of Cross Traffic Generated by the Legacy FTP Applications

varies according to table A.2. Figure A.14 illustrates queue dynamics of the bottleneck link when λ is fixed to 1.25, is fixed to 2.0, or is adjusted based on network environment. In the second group of experiments, cross traffic are generated by legacy FTP applications whose socket buffer is 64KB, and the load varies according to table A.3. Figure A.15 illustrates queue dynamics of the bottleneck link when λ is fixed to 1.25, is fixed to 2.0, or is adjusted based on network environment.

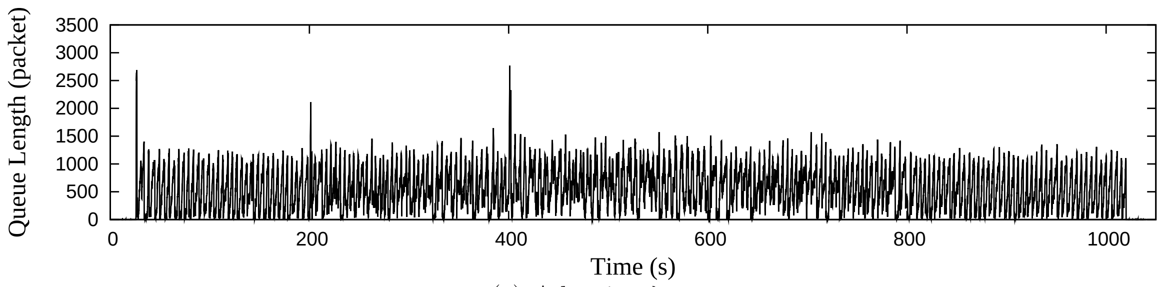
Figure A.14 and A.15 indicate that the kind and the load of cross traffic do affect the value of λ that should be adopted for emptying the queue of the bottleneck link periodically. A fixed value of λ cannot work well in the heterogeneous Internet. Figure A.14 and A.15 also indicate that, through making λ adaptive to network environment, Sync-TCP can empty the queue periodically, and hence drive the bottleneck link to operate around the knee under more scenarios.



(a) $\lambda = 1.25$

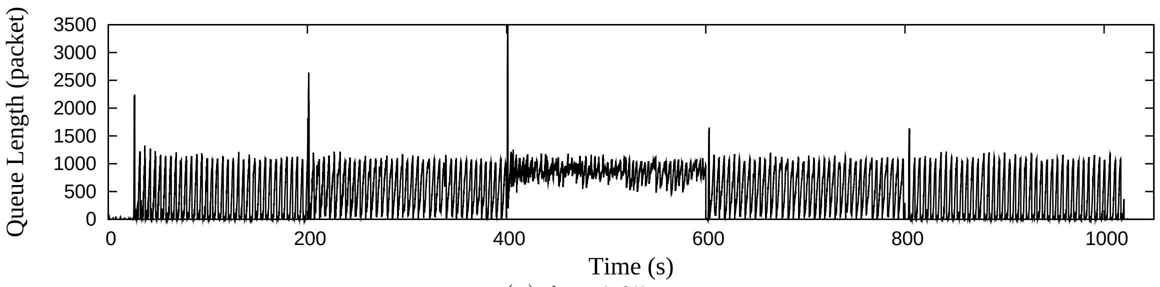


(b) $\lambda = 2.0$

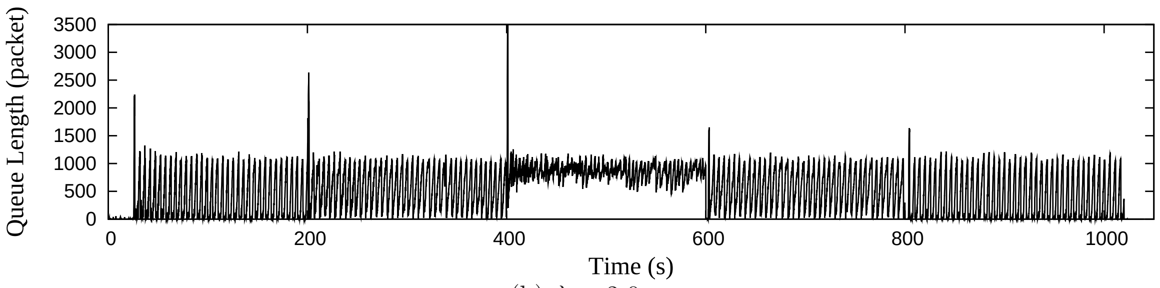


(c) Adaptive λ

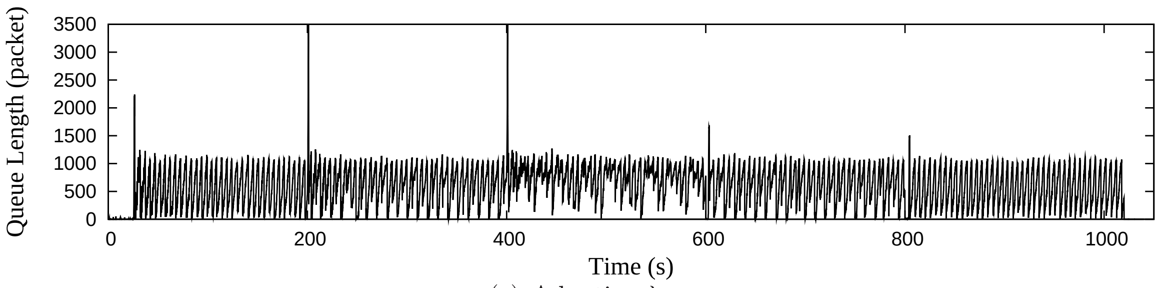
Figure A.14: Queue Dynamics at the Bottleneck Link When the Load of Web Surfing Varies



(a) $\lambda = 1.25$



(b) $\lambda = 2.0$



(c) Adaptive λ

Figure A.15: Queue Dynamics at the Bottleneck Link When the Load of Legacy FTP Cross Traffic Varies