LEVERAGING INNOVATIVE TRANSPORT LAYER SERVICES FOR IMPROVED APPLICATION PERFORMANCE

by

Preethi Natarajan

A dissertation submitted to the Faculty of the University of Delaware in partial fulfillment of the requirements for the degree of Doctor of Philosophy in Computer & Information Sciences

February 2009

LEVERAGING INNOVATIVE TRANSPORT LAYER SERVICES

FOR IMPROVED APPLICATION PERFORMANCE

by

Preethi Natarajan

Approved: _____

B. David Saunders, Ph.D.
Chair of the Department of Computer and Information Sciences

Approved: _____

Tom Apple, Ph.D.
Dean of the College of Arts and Sciences

Approved: _____

Debra Hess Norris, M.S.
Vice Provost for Graduate and Professional Education

I certify that I have read this dissertation and that in my opinion it meets the academic and professional standard required by the University as a dissertation for the degree of Doctor of Philosophy.

Signed: _____

Paul D. Amer, Ph. D.
Professor in charge of dissertation

I certify that I have read this dissertation and that in my opinion it meets the academic and professional standard required by the University as a dissertation for the degree of Doctor of Philosophy.

Signed: _____

Adarshpal S. Sethi, Ph.D.
Member of dissertation committee

I certify that I have read this dissertation and that in my opinion it meets the academic and professional standard required by the University as a dissertation for the degree of Doctor of Philosophy.

Signed: _____

Phillip T. Conrad, Ph. D.
Member of dissertation committee

I certify that I have read this dissertation and that in my opinion it meets the academic and professional standard required by the University as a dissertation for the degree of Doctor of Philosophy.

Signed: _____

Stephan Bohacek, Ph. D.
Member of dissertation committee

I certify that I have read this dissertation and that in my opinion it meets the academic and professional standard required by the University as a dissertation for the degree of Doctor of Philosophy.

Signed: _____

Randall R. Stewart
Member of dissertation committee

ACKNOWLEDGMENTS

I am most grateful to my advisor, Professor Paul Amer, for his incomparable guidance and patience over the past years. He provided an apt environment to learn and hone my skills, and always showed enthusiasm to help improvise my ideas, paper drafts, and presentation slides. His "hands off" approach to advising has been both a challenging and a rewarding experience. Thanks to Prof. Amer, I would cherish my metamorphosis at PEL.

I am thankful to my committee members: Prof. Stephan Bohacek, Prof. Phillip Conrad, Prof. Adarsh Sethi and Randall Stewart for reviewing and suggesting improvements to this work. Special thanks to Randall Stewart and Prof. Phillip Conrad for commuting significant distances to be on my committee. I am also thankful to Fred Baker for his excellent advice during my internship at Cisco Systems and later.

I am fortunate to have worked with a set of brilliant people at PEL. I thank Armando Caro and Janardhan Iyengar for their patient responses and insightful discussions during their stay at PEL and in the subsequent years. I thank Jon Leighton for his shrewd comments, draft reviews, and also for the interesting discussions on a wide range of topics from gun control to non-profit organizations. Nasif Ekiz and Ertugrul Yilmaz have made my PEL years more fun and memorable. I thank Nasif, Ertugrul, and Joe Szymanski for the many times they rebooted machines and restarted experiments on my behalf.

I would like to acknowledge the financial support that made this dissertation possible. This research was sponsored in part by the U.S. Army Research Laboratory, and by Cisco System's University Research Program.

This dissertation would not have been possible without the support from family and friends. My parents, Raju and Gowri, were steadfast on giving me good education, even if it meant pushing their needs to the back burner. Raju never ceases to be a source of inspiration, and made me realize the significance of planning ahead. Gowri taught me the importance of hard work and commitment. I am eternally indebted to them for their emotional support, and am looking forward to spending more time with them in the coming years.

Finally, I thank my confidant and husband, Prasanna, for his rock solid backing during my PhD years. He helped me stay effective, focused, and balanced in many ways, including relocating his job, feeding me during deadlines, and forfeiting even college football to chauffeur me to the lab ☺. I cannot even imagine pulling through the last few years without him by my side.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ABSTRACT

We investigate three issues related to the transport layer, and address these issues using the innovative transport layer services offered by the Stream Control Transmission Protocol (SCTP) [RFC4960].

In the first issue, we explore the benefits from SCTP's multistreaming service for HTTP-based applications. The current web transport – TCP, offers a sequential bytestream, and in-order data delivery within the bytestream. Transferring *independent* web objects over a single TCP connection results in head-of-line (HOL) blocking, and worsens web response times. On the contrary, transferring these objects over different SCTP streams eliminates inter-object HOL blocking. We propose a design for HTTP over SCTP streams, and implement this design in the open source Apache web server and Firefox browser. Using emulation, we show that persistent and pipelined HTTP 1.1 transfers over a single multistreamed SCTP association improves web response times when compared to similar transfers over a single TCP connection. The difference in TCP vs. SCTP response times increases and is more visually perceivable in high latency and lossy browsing condition, as found in the developing world.

The current workaround to improve an end user's perceived WWW performance is to download an HTTP transfer over multiple TCP connections. While we expect multiple TCP connections to improve HTTP throughput, emulation results show that the competing and bursty nature of multiple TCP senders degrade HTTP performance especially in end-to-end paths with low bandwidth last hops. In such

browsing conditions, a single multistreamed SCTP association not only eliminates HOL blocking, but also boosts throughput compared to multiple TCP connections.

In the second issue, we explore how SCTP's (or TCP's) SACK mechanism degrades end-to-end performance when out-of-order data is non-rengable. Using simulation, we show that SACKs result in inevitable send buffer wastage, which increases as the frequency of loss events and loss recovery durations increase. We introduce a fundamentally new ack mechanism, Non-Renegable Selective Acknowledgments (NR-SACKs), for SCTP. An SCTP receiver uses NR-SACKs to explicitly identify some or all out-of-order data as being non-renegable, allowing a sender to free up send buffer sooner than if the data were only SACKed. Simulation comparisons show that NR-SACKs enable more efficient utilization of a transport sender's memory, and also improve throughput in Concurrent Multipath Transfer (CMT) [Iyengar 2006].

The third issue explores CMT performance during path failures. Using simulation, we demonstrate how CMT suffers from significant "rbuf blocking" which degrades performance during permanent and short-term path failures. To improve performance, we introduce a new destination state called the "Potentially Failed" (PF) state. CMT's failure detection and (re)transmission policies are augmented to include the PF state, and the modified CMT is called CMT-PF. Using simulation, we demonstrate that CMT-PF outperforms CMT during failures – even under aggressive failure detection thresholds. We also show that CMT-PF performs on par or better but never worse than CMT during non-failure scenarios. In light of these findings, we recommend CMT be replaced by CMT-PF in existing and future CMT implementations and RFCs.

Chapter 1

INTRODUCTION

1.1 Dissertation Scope

This dissertation investigates three issues related to the transport layer, and addresses these issues to improve application performance. While these issues are explored using the Stream Control Transmission Protocol (SCTP) [RFC4960], different subsets of the proposed ideas and performance conclusions would be applicable to any reliable transport that provides services similar to SCTP. The rest of this section outlines the three issues.

1.1.1 Issue (1): Web over Multistreamed Transport

Transport layer multistreaming is the ability of a transport protocol to support multiple streams, where each stream is a logical data flow with its own sequencing space. Within each stream, the transport receiver delivers data in-sequence to the application, without regard to the relative order of data arriving on other streams. This property makes streams ideal for transferring *independent* web objects. When each web object is transmitted on a different stream, the processing and display of one object does not depend on the successful transfer and delivery of other object(s).

The current web transport – TCP, does not support transport layer multistreaming. At the time TCP was designed, congestion and flow control were the crucial transport layer services required by network applications. Later, when HTTP's

design required a reliable transport protocol, TCP was the only available option and was 'chosen' for HTTP transfers. However, transferring independent web objects over TCP results in sub-optimal response times, since, a TCP connection (i) offers a single sequential bytestream to the application, and (ii) provides *in-order* delivery within the bytestream — if a piece of one web object is lost in the network, successively transmitted web objects will not be delivered to the client until the lost piece is retransmitted and received.

Though it is believed that transport layer streams can improve web response times [Gettys 2002], no experimentation or analysis exists to support this hypothesis. This dissertation provides some of the analysis. When we started working on this issue, SCTP was the only transport that supported multistreaming. Hence, this dissertation considers SCTP streams for HTTP transfers. More recently, [Ford 2007] proposed the Structured Stream Transport (SST) protocol that functions similar to SCTP (discussed in Chapter 2).

1.1.2 Issue (2): Reneging and Selective Acks

Reliable transport protocols (such as TCP and SCTP) employ two kinds of data acknowledgment mechanisms: (i) cumulative acks indicate data that has been received in-sequence, and (ii) selective acks (SACKs) indicate data that has been received out-of-order. While cum-acked data is a receiver's responsibility, SACKed data is not. SACKed out-of-order data is implicitly *renegable*; that is, a receiver may SACK data and later discard it. The possibility of reneging, however remote, forces a transport sender to maintain copies of SACKed data in the send buffer until they are cum-acked.

Data that has been delivered to the application, by definition, is non-renegable by the transport receiver. Unlike TCP which never delivers out-of-order data to the application, SCTP's *multistreaming* and *unordered data delivery* services result in out-of-order data being delivered to the application and thus becoming non-renegable. Interestingly, TCP and SCTP implementations can be configured such that the receiver is not allowed to and therefore never reneges on out-of-order data.

This dissertation investigates the negative effects of the SACK mechanism when out-of-order data is non-renegable. While non-renegable out-of-order data is possible in both TCP and SCTP, note that the possibility is innate to SCTP due to SCTP's out-of-order data delivery services. Therefore, our investigations focus on SCTP.

1.1.3 Issue (3): CMT during Path Failures

A host is multihomed if it can be addressed by multiple IP addresses [RFC1122], as is the case when the host has multiple network interfaces. Multiple active interfaces also suggest the simultaneous existence of multiple paths between the multihomed hosts. CMT [Iyengar 2006] exploits these multiple paths for simultaneous transfer of new data between end hosts, and increases a network application's throughput. [Iyengar 2006] evaluated CMT over paths with asymmetric delay and loss characteristics. But [Iyengar 2006] did not consider path failures, which is the scope of our work.

Both TCP and UDP are unaware of multihoming. Hence, [Iyengar 2006] used the multihomed-aware transport protocol – SCTP, to perform CMT at the transport layer. Since this research is a continuation of [Iyengar 2006], our investigations also use SCTP. Incidentally, SCTP also supports path failure detection.

1.2 An SCTP Primer

SCTP was originally developed to carry telephony signaling messages over IP networks. With continued work, SCTP evolved into a general purpose transport protocol with advanced delivery options [RFC4960]. Similar to TCP, SCTP provides a reliable, full-duplex, congestion and flow-controlled connection, called an *association*. An SCTP packet, or more generally, protocol data unit (PDU), consists of one or more concatenated building blocks called *chunks*: either control or data. For the purposes of reliability and congestion control, each data chunk in an association is assigned a unique Transmission Sequence Number (TSN). Since chunks are atomic, TSNs are associated with chunks of data, as opposed to TCP which associates a sequence number with each data octet in the bytestream.

Unlike TCP, SCTP offers innovative transport layer services such as multihoming and multistreaming.

1.2.1 SCTP Multistreaming

An SCTP stream is a unidirectional data flow within an SCTP association. Independent application objects can be transmitted in different streams to maintain their logical separation during transfer and delivery. All SCTP streams within an association are subject to shared congestion control, and thus SCTP's multistreaming adheres to TCP's fairness principles.

Figure 1.1 illustrates a multistreamed association between hosts A and B. In this example, host A uses three output streams to host B (numbered 0 to 2), and has only one input stream from host B (numbered 0). The number of input and output streams in an SCTP association is negotiated during association setup. SCTP uses *Stream Sequence Numbers* (SSNs) to preserve data order within each stream.

However, maintaining order of delivery between transport protocol data units (TPDUs) transmitted on different streams is not a constraint. That is, data arriving in-order within an SCTP stream is delivered to an application without regard to data arriving on other streams.

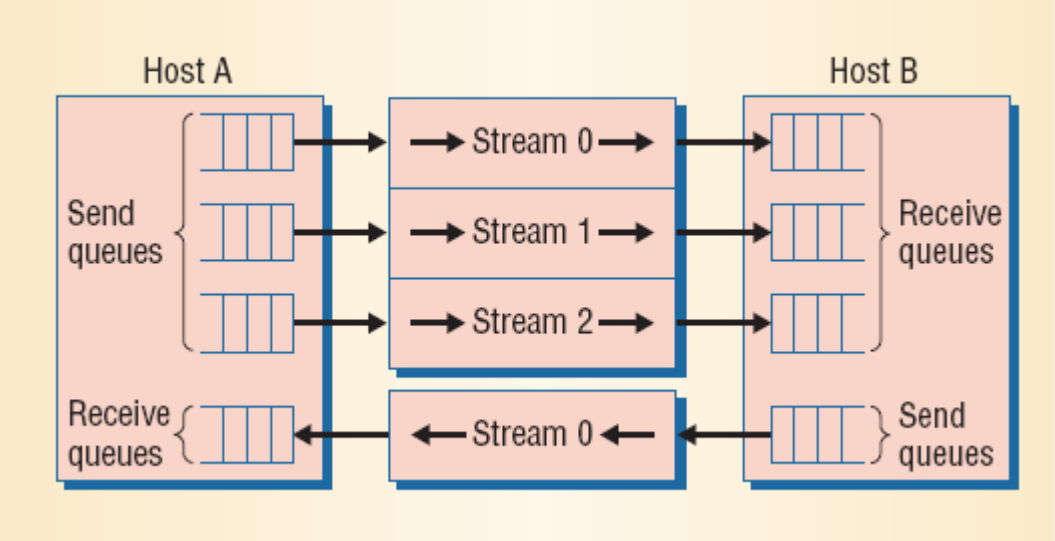


Figure 1.1: Multistreamed Association between Hosts A and B

1.2.2 SCTP Multihoming

To benefit from network interface redundancy and provide end-to-end network fault tolerance, SCTP supports multihoming at the transport layer. An SCTP endpoint may bind to multiple IP addresses during association initialization. Referring to Figure 1.2, let us contrast SCTP with TCP to further explain SCTP's multihoming feature. Four distinct TCP connections are possible between Hosts A and B: (A_1,B_1), (A_1,B_2), (A_2,B_1), (A_2,B_2). SCTP, on the other hand, is not forced to choose a single IP address on each host. Instead, a single SCTP association could consist of two sets of IP addresses, which in our example would be: ({A_1,A_2}, {B_1,B_2}). Each endpoint chooses a single destination address as a primary destination address, which is used for

transmission of new data. Note that a single port number is used at each endpoint regardless of the number of IP addresses.

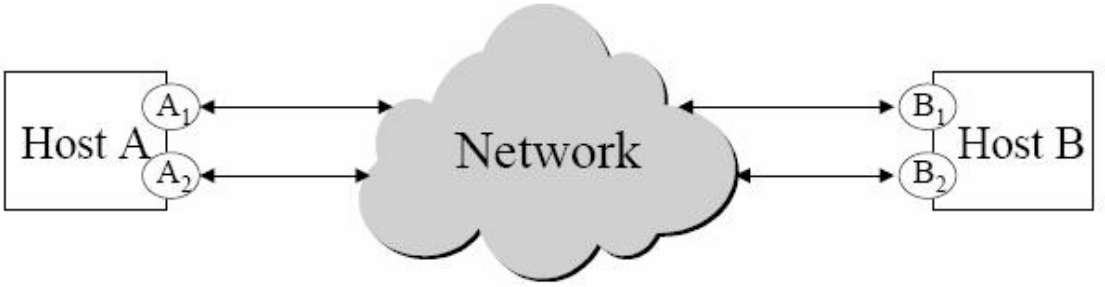


Figure 1.2: Example Multihomed Topology

SCTP monitors the reachability of each destination address through two mechanisms: acks of data and periodic probes known as heartbeats. Failure in reaching the primary destination results in failover, where an SCTP endpoint dynamically chooses an alternate destination to transmit the data, until the primary destination becomes reachable again.

1.2.3 Concurrent Multipath Transfer

Multihoming among networked machines and devices is a technologically feasible and increasingly economical proposition. Multihomed nodes may be simultaneously connected through multiple end-to-end paths to increase resilience to path failure. For instance, users may be simultaneously connected through dial-up/broadband, or via multiple wireless technologies such as 802.11b and GPRS. Concurrent Multipath Transfer (CMT) [Iyengar 2006] is an experimental extension to SCTP that assumes multiple *independent* paths, and exploits these paths for *simultaneous* transfer of new data between end hosts. A naïve version of CMT, where a data sender simply transfers new data over multiple paths, increases data reordering and adversely affects performance. [Iyengar 2006] investigates these negative effects

and proposes algorithms and retransmission policies that improve application throughput.

1.3 Dissertation Overview

A structural overview of the dissertation is shown in Figure 1.3. The three issues are discussed in Chapters 2, 3 and 4, respectively. The references cited for each chapter represent the author's publications for each topic.

Chapter 2 presents our work on the first issue – web over multistreamed SCTP. The chapter proposes a design for HTTP over SCTP streams, and discusses our efforts to implement the design in the popular Apache web server and Firefox browser. Using emulation, we show that persistent and pipelined HTTP 1.1 transfers over a single multistreamed SCTP association improves web response times when compared to similar transfers over a single TCP connection. The difference in TCP vs. SCTP response times increases and is more visually perceivable in high latency and lossy browsing condition, as found in the developing world.

The current workaround to improve an end user's perceived WWW performance is to download an HTTP transfer over multiple TCP connections. While we expect multiple TCP connections to improve HTTP throughput, emulation results show that the competing and bursty nature of multiple TCP senders degrade HTTP performance especially in end-to-end paths with low bandwidth last hops. In such browsing conditions, a single multistreamed SCTP association not only eliminates HOL blocking, but also boosts throughput compared to multiple TCP connections. These experiments were performed as part of this author's summer 2008 internship at Cisco Systems.

Our body of work in HTTP over SCTP has triggered significant interest in the area. The Protocol Engineering Lab has secured additional funding from Cisco Systems to pursue some of the ongoing and future work discussed in Chapter 2.

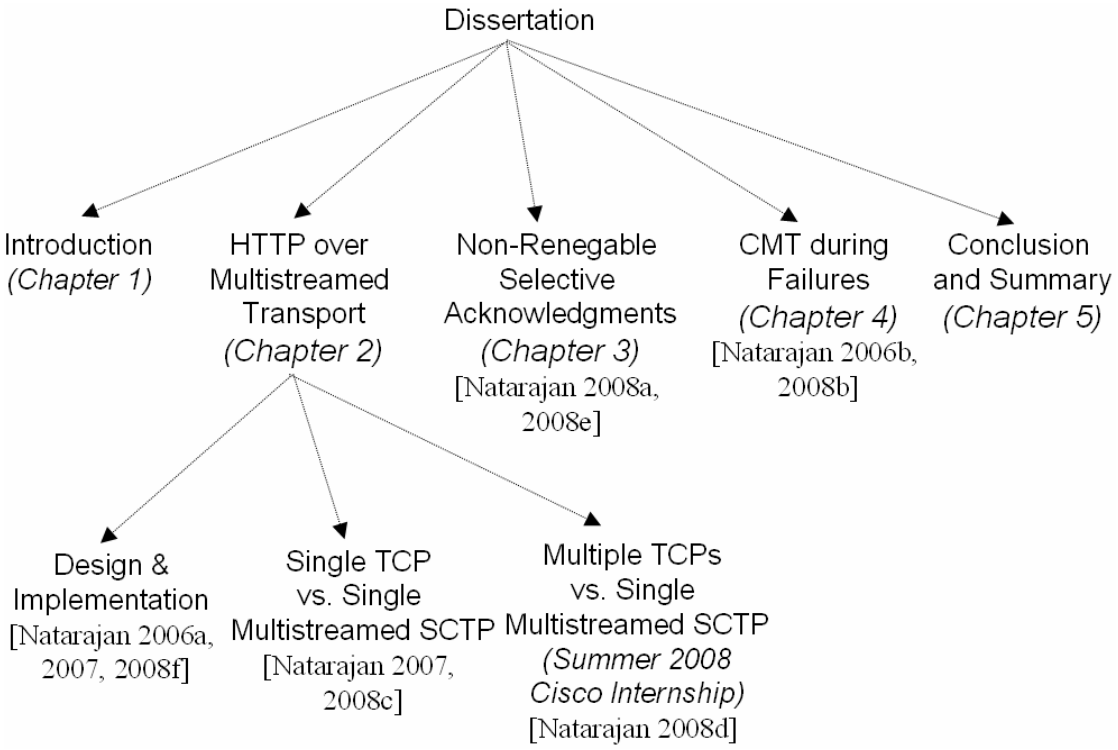


Figure 1.3: Dissertation Structure

Chapter 3 discusses the second issue – how the existing SACK mechanism degrades end-to-end performance when out-of-order data is non-rengable. Using simulation, we show that SACKs result in inevitable send buffer wastage, which increases as the frequency of loss events and loss recovery durations increase. We introduce a fundamentally new ack mechanism, Non-Renegable Selective Acknowledgments (NR-SACKs), for SCTP. An SCTP receiver uses NR-SACKs to explicitly identify some or all out-of-order data as being non-renegable, allowing a sender to free up send buffer sooner than if the data were only SACKed. Simulation

comparisons show that NR-SACKs enable more efficient utilization of a transport sender's memory. Further investigations show that NR-SACKs also improve throughput in CMT. The final section of Chapter 3 discusses ongoing activity, including our efforts within the IETF to standardize NR-SACKs for SCTP, and at UD to implement NR-SACKs in FreeBSD SCTP.

Chapter 4 presents our work on the third issue – CMT performance during path failures. Using simulation, we demonstrate how CMT suffers from significant "rbuf blocking" which degrades performance during permanent and short-term path failures. To improve performance, we introduce a new destination state called the "Potentially Failed" (PF) state. CMT's failure detection and (re)transmission policies are augmented to include the PF state, and the modified CMT is called CMT-PF. Using simulation, we demonstrate that CMT-PF outperforms CMT during failures – even under aggressive failure detection thresholds. We also show that CMT-PF performs on par or better but never worse than CMT during non-failure scenarios. In light of these findings, we recommend CMT be replaced by CMT-PF in existing and future CMT implementations and RFCs. Chapter 4 finishes with a discussion of our on-going effort to implement CMT-PF in FreeBSD SCTP.

Finally, Chapter 5 summarizes our contributions, and concludes this dissertation.

Chapter 2

HTTP OVER MULTISREAMED TRANSPORT

This chapter discusses the first problem – HTTP over SCTP streams. Sections 2.1 and 2.2 explain the head-of-line (HOL) blocking problem and its negative consequences in HTTP over TCP. Section 2.3 describes our design of HTTP over multistreamed SCTP. Sections 2.4 and 2.5 discuss HTTP over SCTP implementation specifics in the Apache web server and Firefox web browser, respectively. Section 2.6 explains evaluation preliminaries and Sections 2.7 and 2.8 present results. Section 2.9 concludes and presents ongoing and future work. Section 2.10 discusses related work.

2.1 Introduction

HTTP [RFC2616] requires a reliable transport protocol for end-to-end communication. While historically TCP has been used for this purpose, HTTP does not require TCP. A TCP connection offers a single sequential bytestream to a web server. In the case of HTTP 1.1 with persistence and pipelining, the independent HTTP responses are serialized and sent sequentially over a single connection (i.e., one TCP bytestream). In addition, a TCP connection provides *in-order* delivery within the bytestream ─ if a TPDU containing HTTP response *i* is lost in the network, successive TPDUs containing HTTP responses $i+n$ $(n \geq 1)$ will not be delivered to the web client until the lost TPDU is retransmitted and received. This situation, known as *head-of-line (HOL) blocking*, occurs because TCP cannot logically separate independent HTTP responses in its transport and delivery mechanisms.

Transport layer multistreaming is the ability of a transport protocol to support multiple streams, where each stream is a logical data flow with its own sequencing space. Within each stream, the transport receiver delivers data in-sequence to the application, without regard to the relative order of data arriving on other streams. SCTP [RFC4960] is a standardized reliable transport protocol which provides multistreaming. Independent HTTP responses transmitted over different streams of an SCTP association can be delivered to the web browser without HOL blocking.

While most web users in developed nations experience excellent browsing conditions, a large and growing portion of WWW users in developing nations experience high end-to-end delays and loss rates. In such network conditions, persistent and pipelined HTTP 1.1 transfers over TCP suffer from exacerbated HOL blocking, resulting in poor browsing experience (discussed in the next section). In this work, we evaluate multistreamed web transport's ability to reduce HOL blocking and improve a web user's browsing experience in developing regions.

2.2 Head-of-line Blocking

This section introduces a model for persistent and pipelined HTTP 1.1 transfer to formulate head-of-line (HOL) blocking. This section also discusses various factors that aggravate HOL blocking.

2.2.1 Model for HTTP 1.1, and HOL Blocking

We consider the following model to understand HOL blocking in an HTTP 1.1 persistent, pipelined transfer containing N embedded objects (Figure 2.1).

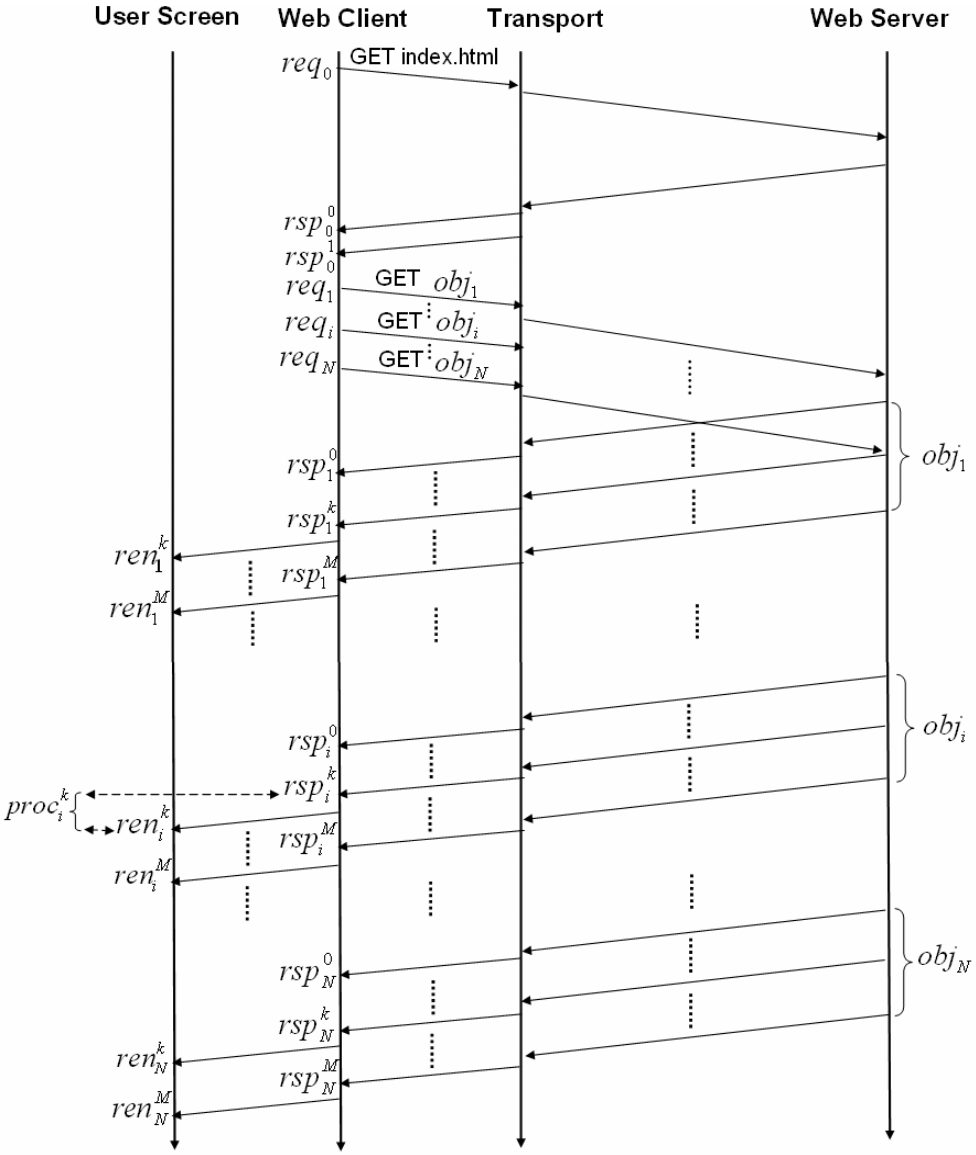


Figure 2.1: Model for HTTP 1.1 Persistent, Pipelined Transfer

obj_i = object i, $0 \leq i \leq N$. obj_0 denotes index.html, $obj_{1..N}$ denote N embedded objects in index.html.

req_i = time when the web client generates the HTTP GET request for obj_i, and writes the request to the transport layer.

obj_i^k = k^{th} piece of obj_i, $0 \leq k \leq M$; obj_i^0 denotes the response header, and $obj_i^{1..M}$ denote the different pieces of obj_i. Note that M depends on the size of obj_i. In our emulations, we assume all objects are the same size (M).

rsp_i^k = time when transport delivers obj_i^k to the web client.

ren_i^k = time when web client renders obj_i^k on user's monitor.

$proc_i^k = (ren_i^k - rsp_i^k)$ denotes the web client's processing time (e.g., decoding, decompression, rendering) for obj_i^k.

In HTTP over TCP, if obj_i^k is lost and recovered after x time units, pieces of obj_j $(j > i)$ could be HOL blocked for x time units. Assuming the web client is currently rendering obj_i^{k-1}, if $(x < proc_i^{k-1})$, this instance of HOL blocking does not affect response time for obj_{j+1}. Otherwise, the HOL blocking increases obj_{i+1}'s response time by $(x - proc_i^{k-1})$ time units [Diot 1999]. Thus, the duration of HOL blocking depends on the loss recovery period, x.

In both TCP and SCTP, the duration of loss recovery based on retransmission after 3 duplicate acks (fast retransmit) takes ~1 round-trip time (RTT), and retransmission after timeout expiration (timeout retransmit) takes between the initial retransmission timeout value (RTO) of 3 seconds and the maximum of (1RTT, min RTO (1 second)) [RFC2988]. Note that the loss recovery period increases as the path's RTT increases. Also, the frequency of HOL blocking increases as the loss rate on the end-to-end path increases. Intuitively, HOL blocking would be exacerbated over a high RTT, lossy path.

Apart from end-to-end path characteristics, individual object sizes also influence the degree of HOL blocking. As object size increases, the probability that a

piece of the object is lost also increases. Hence, a large object in a pipelined transfer is more likely to block delivery of subsequent objects than a smaller object would.

2.2.2 Browsing Conditions in Developing Regions

Unlike web users in developed nations, a large and growing portion of WWW users in developing regions experience Internet delays ranging from 100's of milliseconds to a few seconds. Such high delays transpire from low bandwidth and/or high propagation delay last hops, such as VSAT/3G/GPRS links.

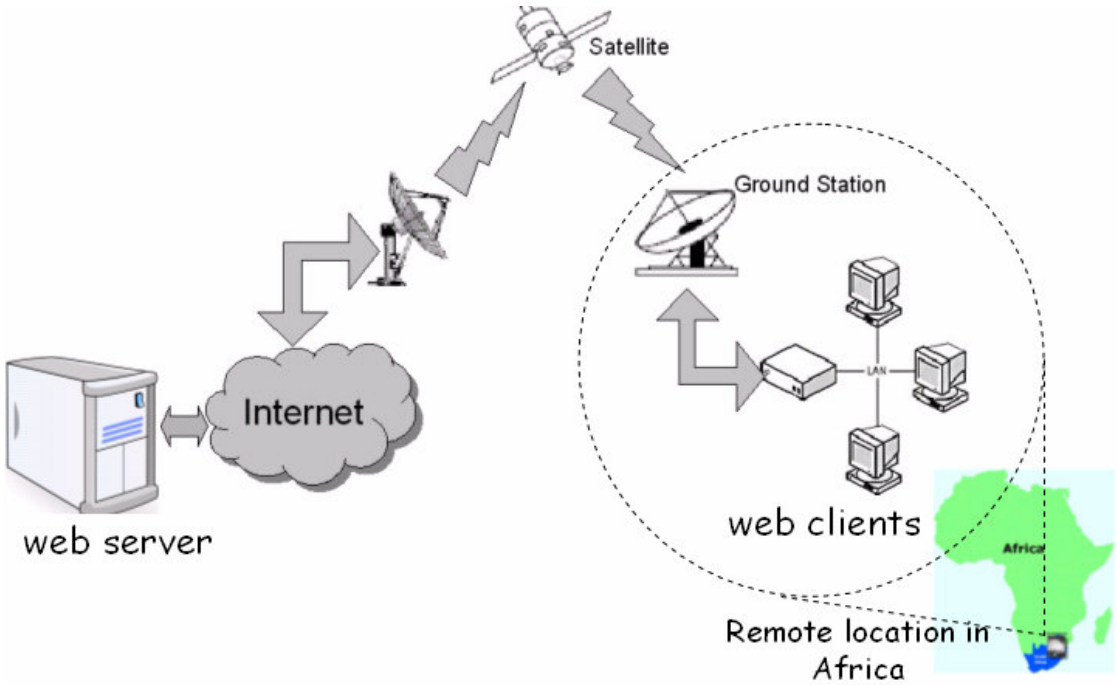


Figure 2.2: Internet Connectivity via VSAT Link

Due to a multitude of factors, VSAT solutions (Figure 2.2) are the most cost-effective and efficient method of providing Internet connectivity for commercial customers, governments and consumers in developing nations and other areas where a land-based infrastructure does not exist [WiderNet, CAfrica, Tarahaat, VSAT-

systems]. The successful deployment of VSAT systems and services in more than 120 countries provides communities with access to information, knowledge, education and business opportunities, and has been crucial in the communities' socio-economic development [Rahman 2002].

The propagation delay from ground station to geostationary satellite to ground station is ~280ms [Gurtov 2004, RFC2760]. Therefore, the delay over a VSAT link increases the RTT by ~560ms. The bandwidth-limited VSAT link is most likely the bottleneck in the transmission path. Any resulting queuing and/or processing delays within the satellite further increase the RTT. The delay caused by shared channel access over a VSAT link can sometimes increase the RTT on the order of few seconds [RFC3135].

GPRS and 3G links are characterized by variable and high latencies; the RTTs in such networks can vary between a few hundreds of milliseconds to 1 second [Chakravorty 2002, Chan 2002, RFC3481]. The proliferation of mobile phones in developing regions, and the increasing use of web browsers and other web applications on mobile phones is another example of web transfers over high latency paths. High Speed Download Packet Access (HSDPA) technology is the successor to 3G, and is emerging from research to deployment. HSDPA offers improved broadband Internet access (~1Mbps per user per cell), and is targeted as a viable option for regular Internet connectivity to both residential and mobile customers. However, channel access and/or propagation delay on an HSDPA link adds ~80ms to the path RTT [Jurvansuu 2007], which is significantly higher than current wired last hop delays.

In addition to propagation delays, sub-optimal traffic routing increases latency of Internet traffic in developing nations [Baggaley 2007, Cottrell 2006]. For

example, sub-optimal routing for intra-African traffic results in Internet traffic traversing multiple VSAT links, and/or being routed through North America or Europe, leading to RTTs as high as 2.5 seconds [PingER]. Furthermore, Internet traffic to/from developing regions traverses through lossy paths, and experiences significant end-to-end loss rates [Cottrell 2006, PingER].

Online U.S. shoppers consider 4 seconds as the maximum acceptable page download time before potentially abandoning a retail site [Akamai 2006]. Response times above 4 seconds interrupt the user experience, causing the user to leave the site or system. While web users over high latency and lossy paths in developing nations must be more tolerant to response times, these users will prefer to use a system that provides better browsing experience.

2.3 Design of HTTP over SCTP Streams

Several experts agree that the best transport scheme for HTTP would be one that supports datagrams, provides TCP compatible congestion control on the entire datagram flow, and facilitates concurrency in GET requests [Gettys 2002]. When we started this work, SCTP was the only available multistreamed transport, and hence became our default choice [Natarajan 2006a]. Afterward, [Ford 2007] proposed a new TCP-based multistreamed web transport. This new transport protocol is similar to SCTP and is discussed in Section 2.10.

Apart from multistreaming, SCTP offers other features that are well suited for a web transport. Unlike TCP, SCTP's state transition does not require a TIME_WAIT state [RFC793], since the *Initiation* and *Verification tags* help to associate SCTP PDUs with the corresponding SCTP associations [RFC4960]. Note that TCP's TIME_WAIT state increases memory and processing overload at a busy

web server [Faber 1999]. Also, SCTP's COOKIE mechanism prevents SYN attacks, and SCTP multihoming provides fault-tolerance and the possibility of multipath transfer [Natarajan 2006a].

Two guidelines governed our HTTP over SCTP design:

- Make no changes to the existing HTTP specification, to reduce deployment concerns.
- Minimize SCTP-related state information at the server so that SCTP multistreaming does not further contribute to the server being a performance bottleneck.

The independent nature of HTTP responses is most exploited by downloading them on different SCTP streams. Accordingly, the important design question to address was: which end (client or server) should decide the SCTP stream to be used for an HTTP response? Having the web server manage the SCTP stream scheduling is undesirable, as it involves maintaining additional state information at the server. Further, the client is better positioned to make scheduling decisions that rely on user perception and the operating environment. We therefore concluded that the client should decide object scheduling on streams.

We considered two designs by which the client conveys the selected SCTP stream to the web server: (1) the client specifies an SCTP stream number in the HTTP GET request and the server sends the corresponding response on this stream, or (2) the server transmits the HTTP response on the same stream number on which the corresponding HTTP request was received. Design (1) requires just one incoming stream and several outgoing streams at the server, but requires modifications to the HTTP GET request specification. Design (2) requires the server to maintain as many

incoming streams as there are outgoing streams, increasing the memory overhead at the server. Every inbound or outbound stream requires additional memory in the SCTP Protocol Control Block (PCB), and the amount of memory required varies with the SCTP implementation. The reference SCTP implementation on FreeBSD (version 6.1), requires 25 bytes for every inbound stream and 33 bytes for every outbound stream [FreeBSD]. We considered this memory overhead per stream to be insignificant compared to the effort to modify the HTTP specification, and chose option (2).

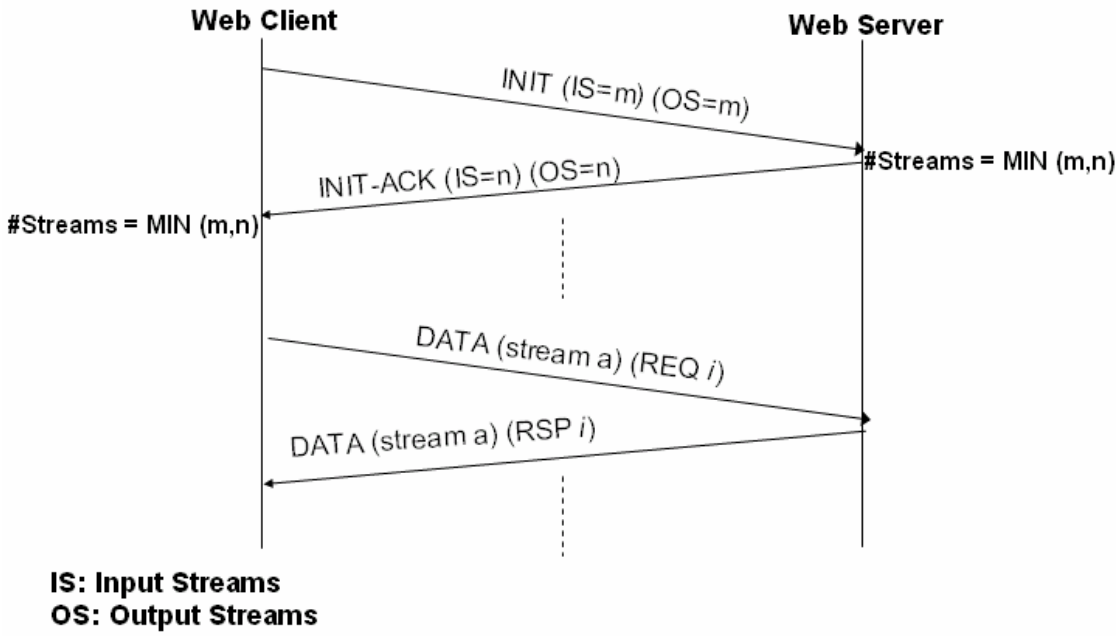


Figure 2.3: Design of HTTP over SCTP Streams

Figure 2.3 gives an overview of our HTTP over SCTP design. A web client and server first negotiate the number of SCTP streams to use for the web transfer. During association establishment, the web client requests *m* inbound and *m* outbound streams. The INIT-ACK from the server carries the web server's offer on the number of inbound/outbound streams (*n*). After association establishment, the number of inbound and number of outbound streams available for HTTP transactions, *s* =

MIN(m,n). Note that an SCTP end point can initially offer a lower number of streams and later increase the offer using the streams reset functionality [Stewart 2008a].

When a web server receives a request on an inbound SCTP stream a ($a< s$), the server sends the corresponding response on the outbound stream a. If $s <$ number of pipelined requests, the web client must schedule the requests over the available SCTP streams using a scheduling policy, such as round-robin.

2.4 Implementation in the Apache Web Server

We chose the popular open source Apache web server (version 2.0.55) [Apache] for our task. In this section, we give an overview of Apache's architecture, and our adaptations [Natarajan 2006a].

2.4.1 Apache Architecture

The Apache HTTP server (*httpd*) has a modular architecture. The main functions such as server initialization, HTTP request parsing, and memory management are handled by the *core* module. Accessory functions such as request redirection, authentication, dynamic content handling are performed by separate modules. The *core* module relies on Apache Portable Runtime (APR), a platform independent API, for network, memory and other system dependent functions.

Apache uses *filters* – functions through which different modules process an incoming HTTP request (*input filters*) or an outgoing HTTP response (*output filters*). The *core* module's input filter calls APR's read API to read HTTP requests. During request processing, all state information related to the request are maintained in a *request structure*. Once the response is generated, the *core* module's output filter calls APR's send API for transmitting the response.

Apache has a set of multi-processing architectures that can be enabled at compile time. We considered the following architectures: (1) *prefork* ─ non-threaded pre-forking server and (2) *worker* ─ hybrid multi-threaded multi-processing server. With *prefork*, a configurable number of processes are forked during server initialization, and are setup to listen for connections from clients. With *worker*, a configurable number of server threads and a listener thread are created per process. The listener thread listens for incoming connections from clients, and passes the connection to a server thread for further processing. In both architectures, the server processes or threads handle requests sequentially from a transport connection.

2.4.2 Adapting Apache

Apache's *core* module and the APR were modified to support SCTP streams. APR's read and send API implementations were modified to read and transmit data on a specific SCTP stream. Each time APR reads an HTTP request, the SCTP input stream number is stored in the corresponding *request structure*, so that the response can be written on the equivalent SCTP output stream.

Apache uses *directives* that allow a web administrator to configure various parameters during server initialization. The syntax of the *Listen directive* was modified so that a web admin can configure the transport protocol (TCP or SCTP) during initialization.

2.5 Implementation in the Firefox Web Browser

We chose the Firefox (version 1.6a1) browser since it is a widely used open-source browser. Firefox belongs to the Mozilla suite of applications which have a layered architecture [Mozilla]. Mozilla applications such as Firefox and Thunderbird

(mail/news reader), belong to the top layer, and rely on the services layer for access to network and file I/O. The services layer uses platform independent APIs offered by the Netscape Portable Runtime (NSPR) library.

Firefox has a multi-threaded architecture. To render a web page, the HTTP module in the services layer parses the URL, uses NSPR to open a TCP connection to the appropriate web server, and downloads the web page. While parsing the web page, the HTTP module opens additional TCP connections as required, and pipelines HTTP GET requests for the embedded objects.

Adapting Firefox to work over SCTP streams involved modifications to both NSPR and the HTTP module.

2.5.1 Adapting NSPR

We first modified NSPR to create and setup an SCTP socket instead of a TCP socket. During association establishment, NSPR requests a specific number of SCTP input and output streams. Note that this request can be negotiated down by the server. Therefore, after association establishment, NSPR queries SCTP for the number of input/output streams available for HTTP transactions. Also, NSPR was modified to include new SCTP related send and receive methods.

In the current implementation, HTTP request scheduling over SCTP streams is handled within NSPR. Since the HTTP module is more knowledgeable about the web page contents and user preferences, future implementations could consider HTTP request scheduling at the HTTP module.

In current HTTP request scheduling, the requests are transmitted in a round-robin fashion over SCTP streams. Other scheduling policies can also be considered. For example, in a lossy network environment, such as wide area wireless

connectivity through GPRS, a better scheduling policy might be 'smallest pending object first' where the next GET request goes on the SCTP stream that has the smallest sum of object sizes pending transfer. Such a policy reduces the probability of HOL blocking among the responses downloaded on the same SCTP stream.

2.5.2 Adapting the HTTP Module

Modifying the HTTP module turned out to be more challenging than expected, primarily due to Firefox's assumptions about in-order data delivery within a transport connection. Within the HTTP module, an nsHttpPipeline object is responsible for sending pipelined requests and reading pipelined responses. As shown in Figure 2.4, nsHttpPipeline creates an nsHttpTransaction object for each request. An nsHttpTransaction object is associated with an nsHttpConnection object, which reads the HTTP responses from NSPR. Since pipelined responses are read back-to-back, nsHttpPipeline uses the response length information (available in the response header) to distinguish the end of current response from the beginning of next response. In effect, an nsHttpPipeline object assumes the following about a transport layer connection:

1. All pieces of one response will be delivered before any piece of another response is delivered. That is, pieces of responses will not be delivered in an interspersed fashion.
2. Responses are delivered in the same sequence in which the pipelined requests were transmitted.

These assumptions hold when the underlying transport is TCP – a reliable protocol delivering in-order data to nsHttpPipeline. However, various factors result in out-of-order response delivery in HTTP over SCTP streams.

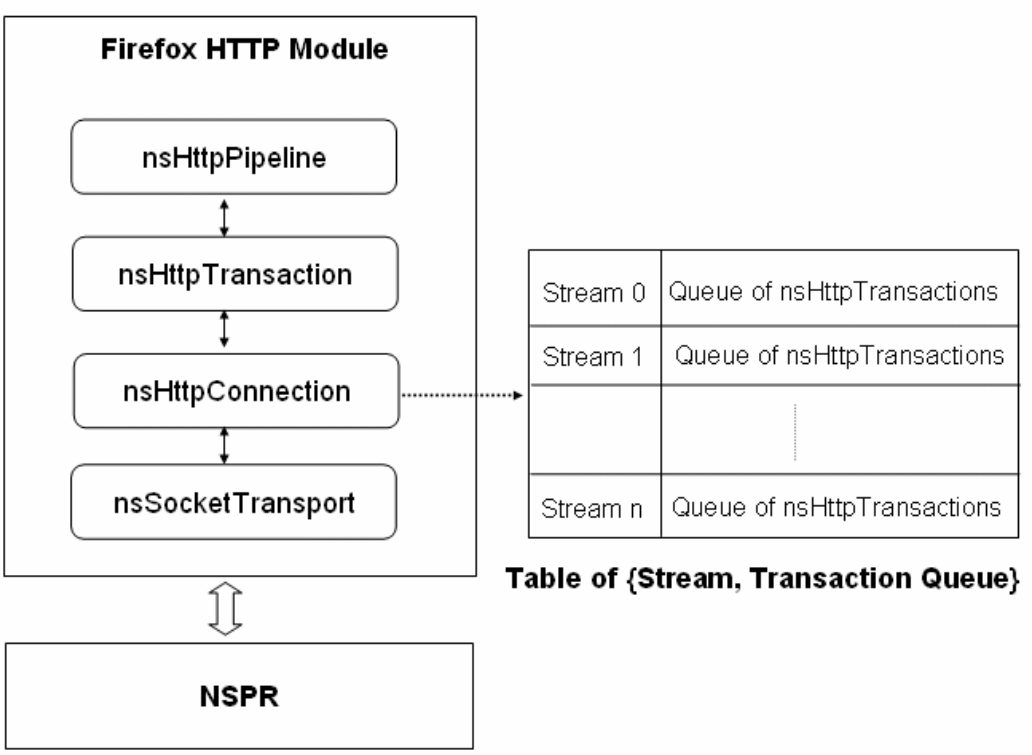


Figure 2.4: Modifications to Firefox HTTP Module

2.5.2.1 Factors Affecting Response Delivery in HTTP over SCTP streams

As mentioned in Section 2.4.1, the current Apache implementation reads and processes requests in succession (one after the other) within a transport connection. Therefore, Apache generates responses in the same sequence that it reads requests, i.e., Apache's response sequence (*server_response*) equals its request sequence (*server_request*). Also, for the following discussions, let the HTTP module's transmitted request sequence be *client_request,* and the delivered response sequence be *client_response*.

2.5.2.1.1 Non HOL Blocked Requests

Loss of HTTP requests transmitted on stream *i*, does not prevent delivery of successfully received requests on stream *j*. During request losses, *server_request* will be different from *client_request*. Therefore, the generated *server_response*, and *client_response* will be different from *client_request*, violating nHttpPipeline's assumption (2).

2.5.2.1.2 Non HOL Blocked Responses

At Firefox's SCTP layer, the loss of a response on stream *i*, does not prevent delivery of successfully received responses on stream *j*. During response losses, *client_response* can be different from *client_request*, also violating nsHttpPipeline's assumption (2).

2.5.2.1.3 Interaction between Apache and FreeBSD SCTP

SCTP preserves message boundaries. At Apache, data in each *write()* translates to an application message, and this message is delivered in its entirety to the receiving application. SCTP fragments a message into Path MTU (PMTU) sized TPDUs before transmission. SCTP's fragmentation and reassembly process is designed such that all message fragments must be assigned consecutive Transmission Sequence Numbers (TSNs). Therefore, all message fragments must be transmitted sequentially. The receiving SCTP uses the (i) (B)eginning fragment bit, (ii) sequential TSNs, and (iii) (E)nding fragment bit for correct reassembly [RFC4960]. In effect, SCTP's fragmentation and reassembly creates *dependencies* in message transmission. *A fragment of message i+1 cannot be transmitted until all fragments of message i have been transmitted.*

Apache's request processing rate is often higher than SCTP's data transmission rate, especially when SCTP's data transmission is limited by low bandwidth/high latency links and/or packet losses. In such scenarios, as long as the SCTP socket's send buffer allows, Apache writes multiple HTTP responses on the socket, and these responses await transmission at the SCTP send buffer. If Apache writes a 100K response on stream *i* followed by a 1K response on stream *j*, SCTP will not transmit the 1K response until all fragments of the 100K response are successfully transmitted. Note that the transmission time of the 100K response increases in low bandwidth/high latency/high loss scenarios. Since the 100K and 1K responses are self-regulating, it is highly desirable that browser's rendering of the 1K response does not depend on transmission/arrival/rendering of the 100K response.

To overcome this issue, we relocated message fragmentation from the SCTP layer to *HTTP response fragmentation* at Apache. Apache writes an HTTP response as multiple application messages, such that, each message at the SCTP layer results in a PMTU-sized TPDU, and is not fragmented further by SCTP. An application can use either the SCTP_PEER_ADDR or the SCTP_STATUS socket options to get the association's PMTU [Stewart 2008b].

HTTP response fragmentation results in the following interesting interaction between Apache and FreeBSD SCTP. The FreeBSD SCTP maintains a queue of application messages for each outbound stream in an association. Note that during HTTP response fragmentation, the messages in these queues translate to a piece of an HTTP response. The FreeBSD SCTP transmits messages from the stream queues in a round-robin fashion. If an SCTP association has *m* outbound streams, once an application message from stream *i*'s queue is transmitted, a message from stream (*i+1*

mod m)'s queue is considered for transmission. When Apache's request processing rate is higher than SCTP's transmission rate, multiple SCTP stream queues contain messages (pieces of HTTP responses) awaiting transmission. Due to FreeBSD SCTP's round-robin transmission, the HTTP response pieces are transmitted in an interspersed fashion, and arrive in the same fashion at Firefox's SCTP layer. In fact, *even under no loss conditions*, delivery of a piece of response *i* can be followed by a piece of response *j*, violating nsHttpPipeline's assumption (1).

2.5.2.1.4 Web Server Architecture

Currently, Apache's multi-threaded architecture dedicates a server thread to each transport connection, and the server thread services requests in succession. We envision a multi-threaded server architecture, where multiple server threads concurrently serve requests on a transport connection [Natarajan 2006a]. To understand our motivation for the new architecture, consider the following two cases: (i) current architecture, where a single server thread serves responses 1 and 2 in succession, and (ii) new architecture, where two server threads concurrently serve responses 1 and 2. Note that the server communicates over a single SCTP association in both cases. However, the concurrency in case (ii) causes the initial pieces of both responses to be transmitted sooner (and rendered sooner by the client) than case (i). We call case (ii) *object interleaving* and discuss its advantages in [Natarajan 2006a].

Now, assume that the web server does HTTP response fragmentation and both responses are transmitted on the *same* SCTP stream. In case (i), the server writes all pieces of response 1 on the stream before writing response 2. Therefore, all pieces of response 1 are transmitted (and delivered) to Firefox before any piece of response 2. However, in case (ii), the two server threads write concurrently over the same SCTP

stream. Therefore, the response pieces can be transmitted and delivered in an interspersed fashion at Firefox, violating nsHttpPipeline's assumption (1).

2.5.2.2 Modifications to the HTTP Module

Based on our experience with Apache and Firefox, we feel that adapting Apache and Firefox to handle object interleaving is a complex task, and it might be easier to develop a new server and browser from the scratch. Nevertheless, we reiterate that a multistreamed web transport opens up new possibilities such as object interleaving, which can further improve HTTP performance.

In our Firefox adaptation over SCTP, nsHttpPipeline's assumptions on response delivery are similar as before, but, this time the assumptions are w.r.t. an SCTP stream instead of a transport connection. nsHttpPipeline assumes that, within an SCTP stream, (1) all pieces of one response will be delivered in-order, before any piece of another response is delivered, and (2) responses are delivered in the same sequence in which the pipelined requests were transmitted.

The HTTP module was modified as follows (Figure 2.4):

- nsHttpConnection maintains a table data structure as shown in Figure 2.4. Each entry in the table is a set of {SCTP stream number, *queue* of requests (nsHttpTransactions objects) transmitted over the stream}.
- After transmitting a request over an SCTP output stream, nsHttpConnection appends the corresponding nsHttpTransaction object to the tail of the stream's queue.
- Whenever data can be read from the SCTP socket, NSPR first notifies nsHttpConnection about the SCTP input stream number. NSPR uses the

MSG_PEEK flag and/or SCTP's extended receive information structure [Stewart 2008b] to gather this information.

- Once nsHttpConnection knows the SCTP input stream, nsHttpConnection associates the received piece of response to the nsHttpTransaction at the head of the stream's queue.
- When the nsHttpTransaction object is read completely, nsHttpConnection deletes this transaction from the head of the stream queue, so that the next piece of response on the stream is delivered to the new head of queue.

2.6 Evaluation Preliminaries

The SCTP-enabled Apache and Firefox were used to evaluate improvements to web users' browsing experience in Internet conditions found in the developing world. This section discusses evaluation preliminaries such as the nature of web workloads and experimental setup.

2.6.1 Nature of Web Workloads

Several web characterization studies have identified certain key properties of the WWW. These properties have led to a better understanding of WWW's nature, and the design of more efficient algorithms for improved WWW performance.

Using server logs from six different web sites, Arlitt et. al. identified several key web server workload attributes that were common across all six servers [Arlitt 1997]. Their work also predicted that these attributes would likely "persist over time". Of these attributes, the following are most relevant to our study: (i) both file size and transferred file size distributions are heavy-tailed (Pareto), and (ii) the median transferred file size is small (≤5KB). A similar study conducted several years later

confirmed that the above two attributes remained unchanged over time [Williams 2005]. Also [Williams 2005] found that the mean transferred file size had slightly increased over the years, due to an increase in the size of a few large files. Other studies such as [Houtzager 2003, Williamson 2003] agree on [Arlitt 1997]'s findings regarding transferred file size distribution and median transferred file size.

These measurement studies lead to a consensus that unlike bulk file or multimedia transfers, HTTP transfers are short-lived flows, where, a typical web object consists of a small number of TPDUs and can be transferred in a few RTTs.

2.6.2 Experimental Setup

The emulations were performed on the FreeBSD platform which had the kernel-space reference SCTP implementation. The experimental setup, shown in Figure 2.5 uses three nodes running FreeBSD 6.1: (i) a node running the in-house TCP or SCTP HTTP 1.1 client, (ii) a server running Apache, and (iii) a node running Dummynet [Rizzo 1997] connecting the server and client. Dummynet's traffic shaper configures a full-duplex link between client and server, with a queue size of 50 packets in each direction. Both forward and reverse paths experience Bernoulli losses with loss rates varying from 0%-10% – typical of the end-to-end loss rates observed in developing regions [Cottrell 2006, PingER].

FreeBSD TCP's default initial cwnd is 4MSS [FreeBSD, RFC3390]. The recommended initial cwnd in SCTP is 4MSS as well. FreeBSD TCP implements packet counting, while SCTP implements Appropriate Byte Counting (ABC) with L=1 [RFC4960, RFC3465]. Additionally, FreeBSD TCP implements Limited Transmit [RFC3042], which enhances loss recoveries for flows with small cwnds. Both transports implement SACKs and delayed acks.

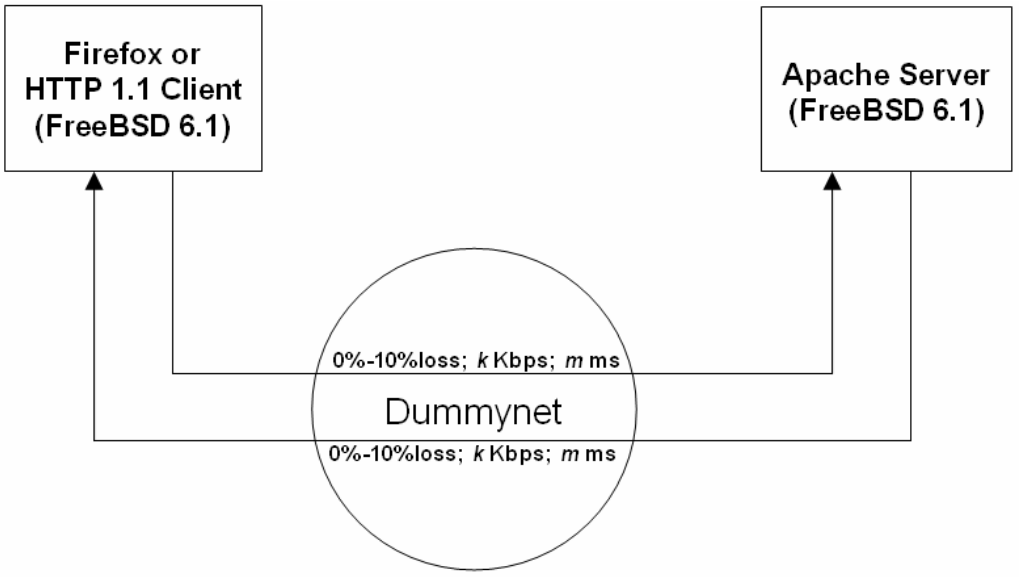


Figure 2.5: Emulation Setup

2.7 Single TCP Connection vs. Single Multistreamed SCTP Association

This section compares an HTTP 1.1 persistent, pipelined transfer over a single TCP connection vs. over a single multistreamed SCTP association. The impact of multiple transport connections is discussed in Section 2.8.

2.7.1 Experiment Parameters

Every pipelined transfer comprises of an *index.html* with N equal sized embedded objects of following sizes: 3KB, 5KB, 10KB, and 15KB. The number of embedded objects (N) varies: 5, 10, and 15. We believe these values reflect current trends in web pages. For example, the number of embedded images in web pages of online services such as maps.google.com and flickr.com vary from 8 to 20. At both client and server nodes, we assume that the transport layer send and receive buffers are not the bottlenecks; they are large enough to hold all data of pipelined transfer.

The following high latency browsing environments are considered for evaluation [Cottrell 2006, PingER]. Results for other high latency environments such as High Speed Download Packet Access (HSDPA) links are available in [Natarajan 2007].

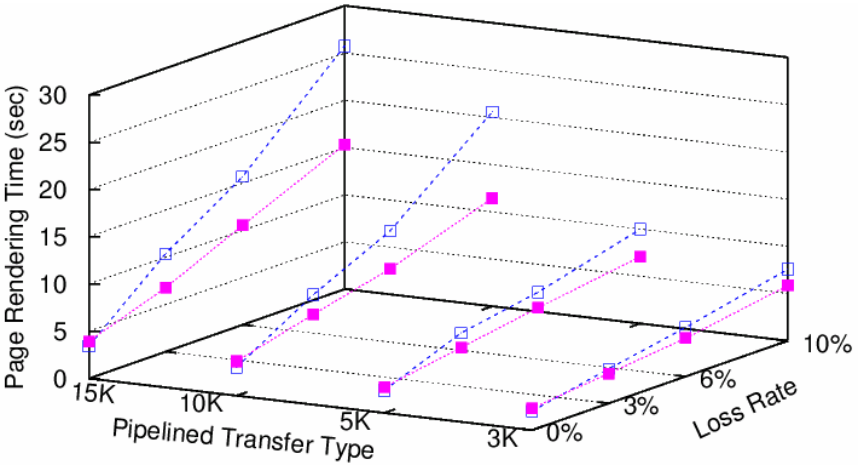
- 1Mbps link with 350ms RTT (*1Mbps.350ms*): User in South Asia, accessing a web server in North America over a land line.
- 1Mbps link with 850ms RTT (*1Mbps.850ms*): User in Africa, sharing a VSAT link to access a web server in North America.
- 1Mbps link with 1100ms RTT (*1Mbps.1100ms*): User in Africa, sharing a VSAT link to access a web server within Africa. The web traffic traverses at least 2 VSAT links; the RTT over each VSAT link is ~550ms.

2.7.2 Results: Page Rendering Times

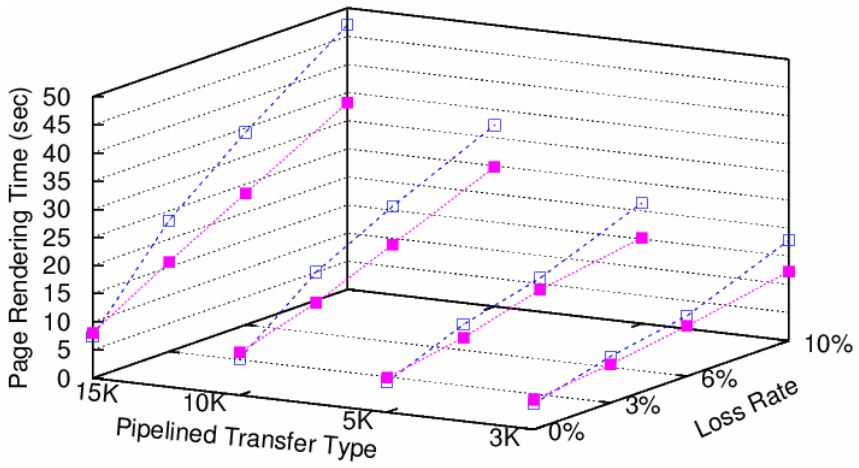
A web page is considered completely *downloaded* when Firefox receives the last piece of pipelined transfer from the transport layer (Figure 2.1). The web page is completely *rendered* when Firefox processes and draws this last piece on the user's screen. In HTTP over TCP (HTTP/TCP), the last piece of data always belongs to the last pipelined object, whereas in HTTP over SCTP streams (HTTP/SCTP), the last piece of data could belong to *any* pipelined object. In both schemes, rendering the last piece of an object depends on the throughput of the underlying transport connection.

Using terminology defined in Section 2.2 (see Figure 2.1), page rendering time is defined as the time from when the browser sends the first GET request (*index.html*), to the time when the last piece of the web page is painted on the screen.

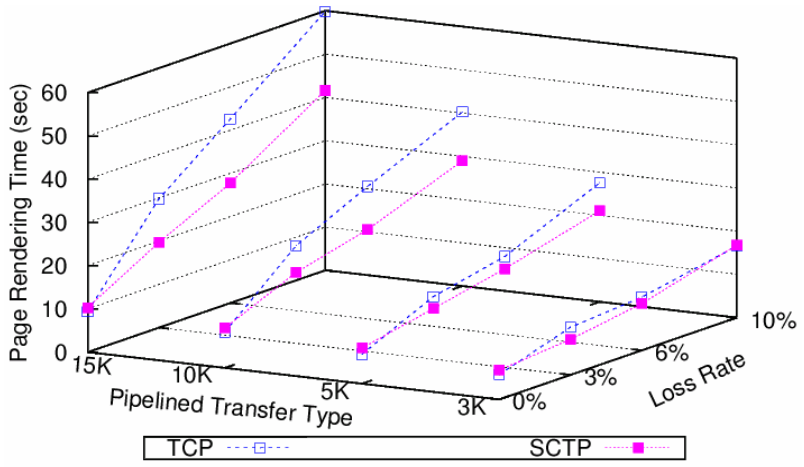
$$\textit{Page rendering time (T)} = (ren_N^M - req_0)$$



(a): 1Mbps.350ms



(b): 1Mbps.850ms



(c): 1Mbps.1100ms

Figure 2.6: Page Rendering Times (N=10)

Our initial hypotheses about SCTP and TCP's page rendering times were as follows:

- Both SCTP and TCP have similar values for their initial cwnd, and employ delayed acks with a 200ms timer. Therefore, we expected both TCP and SCTP's page rendering times to be identical when no losses occur.
- Though SCTP and TCP congestion control are similar, minor differences enable better loss recovery and increased throughput in SCTP [Alamgir 2002]. Unlike TCP whose SACK info is limited by the space available for TCP options, the size of SCTP's SACK chunk is larger (only limited by the path MTU), and therefore at times contains more information about lost TPDUs than TCP's SACK. Also, FreeBSD's SCTP stack implements the Multiple Fast Retransmit algorithm (MFR), which reduces the number of timeout recoveries at the sender [Caro 2006]. Therefore, as loss rates increase, we expected the enhanced loss recovery features to help SCTP outperform TCP.

Figure 2.6 shows the page rendering times for N=10, averaged over 50 runs with 95% confidence. Similar results for N=5 and 15 can be found in [Natarajan 2007]. Interestingly, in all 3 graphs, the results for the no loss case contradict (i), and TCP's rendering times are slightly (but not perceivably) better than SCTP's. Detailed investigation revealed the following difference between the FreeBSD 6.1 SCTP and TCP implementations. SCTP implements Appropriate Byte Counting (ABC) with L=1. During slow start, a sender increments cwnd by 1MSS bytes for each delayed ack. The TCP stack does packet counting which results in a more aggressive cwnd increase when the client acks TCP PDUs smaller than 1MSS (such as HTTP response headers).

We expect SCTP to perform similar to TCP when the TCP stack implements ABC with L=1.

As the loss rate increases, SCTP's enhanced loss recovery offsets the difference in SCTP vs. TCP cwnd evolution. SCTP begins to perform better; the difference even more pronounced for transfers containing larger objects (10K and 15K). For the 1Mbps.1100ms case, the difference between SCTP and TCP page rendering times for 10K and 15K transfers is ~6 seconds at 3% loss, and as high as ~15 seconds at 10% loss. For the same types of transfers, the difference is ~8-10 seconds for 10% loss in 1Mbps.350ms scenario. Similar trends are observed in results for N=5 and 15 as well [Natarajan 2007].

To summarize, SCTP's page rendering times are comparable to TCP's during no loss, and SCTP's enhanced loss recovery enables faster page rendering times during lossy conditions. More importantly, the absolute page rendering time difference increases, and is more visually perceivable as the end-to-end delay, loss rate, and pipelined transfer size increase.

2.7.3 Results: Response Times for Pipelined Objects

Persistent and pipelined HTTP 1.1 transfers over a single TCP connection results in *sequential rendering* at Firefox – even if Firefox's TCP layer has downloaded *all* objects in the pipelined transfer, these *independent* objects are delivered to Firefox only in a *sequential* manner, such that Firefox processes and renders at most one object at a time. Packet losses cause HOL blocking and further delay the sequential delivery of independent objects. On the other hand, SCTP streams provide concurrency in the transfer and delivery of independent objects – an SCTP receiver can deliver object *i+1* to Firefox even before object *i* is completely delivered as long as these two objects are

transmitted over different SCTP streams. This concurrency enables Firefox to render *multiple* objects in parallel, a.k.a., *concurrent rendering*.

While browsers have to open multiple TCP connections to achieve concurrent rendering, concurrent rendering is innate to a multistreamed web transport. The browser tunes the concurrency level by simply adjusting the number of streams. An SCTP association with one stream provides the same concurrency as a single TCP connection, and results in sequential rendering. An SCTP association with two streams provides twice as much concurrency as sequential rendering. A multistreamed association provides *maximum concurrency* for a pipelined transfer when the number of streams equals the number of objects in the transfer. Note that concurrent rendering remains unaffected by a further increase in concurrency.

In our initial investigations, we discovered that a multistreamed web transport enables concurrent rendering *even during no losses*. Irrespective of packet losses, the interaction between Apache's HTTP response fragmentation and FreeBSD SCTP (Section 2.5.2.1.3) causes Firefox's SCTP layer to receive pieces of multiple objects in an interleaved fashion. The SCTP receiver delivers these pieces of multiple objects in an interspersed fashion to Firefox, resulting in concurrent rendering even during no losses. During packet losses, SCTP streams eliminate or reduce HOL blocking, thus increasing the degree of concurrent rendering. Concurrent rendering is demonstrated in a number of movies available online at [Movies].

To reiterate, the fundamental difference between sequential and concurrent rendering is that in sequential rendering, a piece of object *i* is rendered only after objects 1 through *i-1* are completely rendered, whereas in concurrent rendering, pipelined objects are displayed independent of each other. We use the following metric

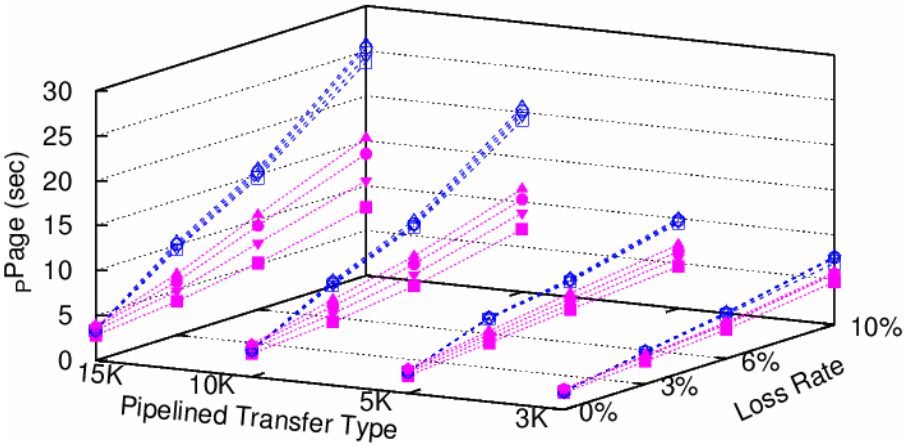
to capture the concurrency and progression in the appearance of all pipelined objects on the user's screen. Recall terminology from Section 2.2,

req_0 = time when browser sends HTTP GET request for *index.html*.

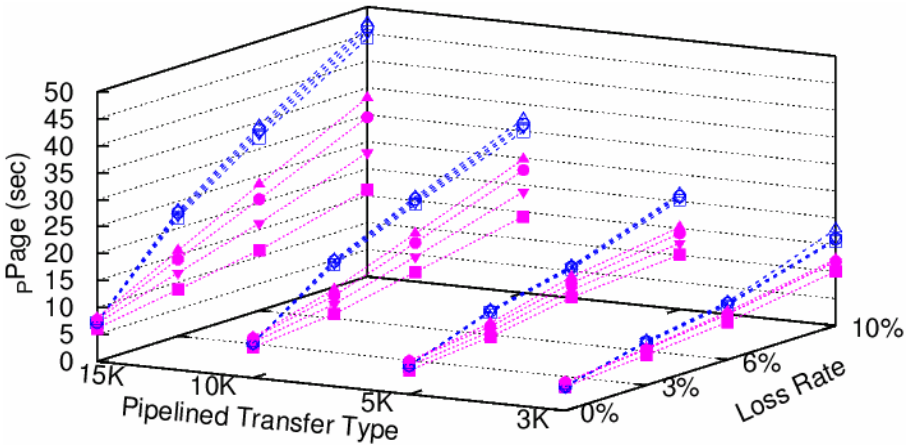
$({}_P ren_i - req_0)$ = time elapsed from the beginning of the page download (req_0) to the earliest time when at least *P%* of object *i* is rendered.

${}_P Page$ is defined as the time elapsed from the beginning of page download to the earliest time when at least *P%* of *all* pipelined objects are rendered on the screen, i.e., ${}_P Page = \text{MAX}\ [({}_P ren_i - req_0);\ 1 \leq i \leq N]$

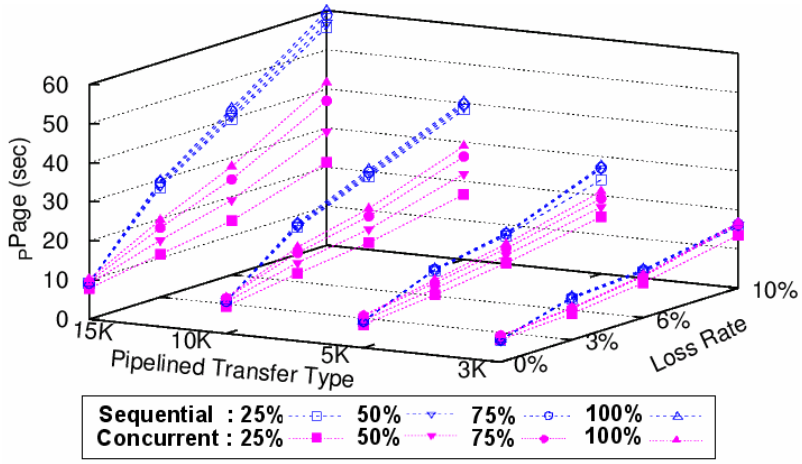
Figure 2.7 plots the ${}_{25\%}Page$, ${}_{50\%}Page$, ${}_{75\%}Page$ and ${}_{100\%}Page$ values for N=10, averaged over 50 runs. Transfers over SCTP consider maximum concurrency, i.e., enough SCTP streams are opened so that every pipelined object is downloaded on a different stream. Results for N=5 and 15 can be found in [Natarajan 2007]. As expected, ${}_{100\%}Page$ values for both concurrent (solid points connected by dotted lines) and sequential (hollow points connected by dashed lines) rendering equal the corresponding transport's page rendering times (*T*). Also, the ${}_P Page$ times in concurrent rendering are spread out vs. clustered together in sequential rendering. Concurrent rendering's dispersion in ${}_P Page$ values signifies the parallelism in the appearance of all 10 pipelined objects.



(a): 1Mbps.350ms



(b): 1Mbps.850ms



(c): 1Mbps.1100ms

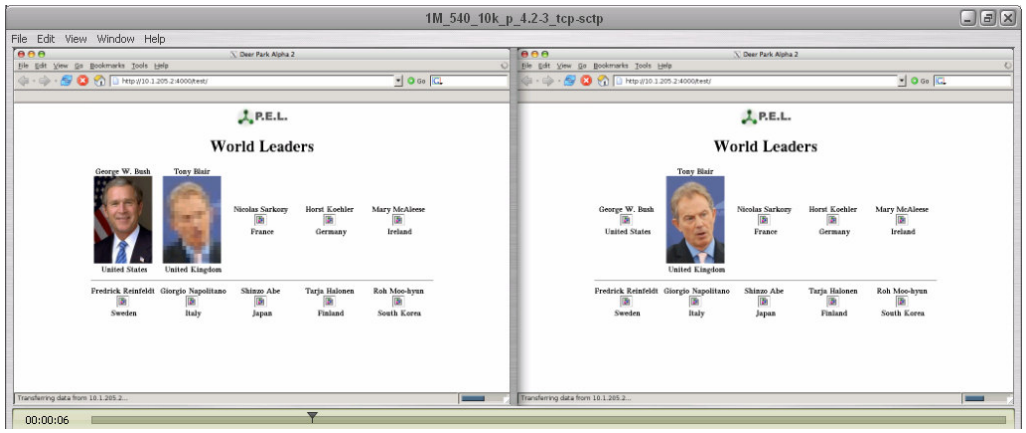
Figure 2.7: $_{P}$Page Values for N=10

Both sequential and concurrent rendering schemes' values are comparable at 0% loss. As loss rate increases, the difference in two rendering schemes' $_{P}Page$ values increase. In addition, we find that concurrent rendering displays 25%-50% of all pipelined objects much sooner (relative difference ~4 – 2 times for 15K, 10K and 5K objects) than sequential rendering. This result holds true for N=5 and 15 as well. In the following subsection, we demonstrate how this result can be leveraged to significantly improve response times for objects such as progressive images, whose initial 25%-50% contain sufficient information for the human eye to perceive the object contents.

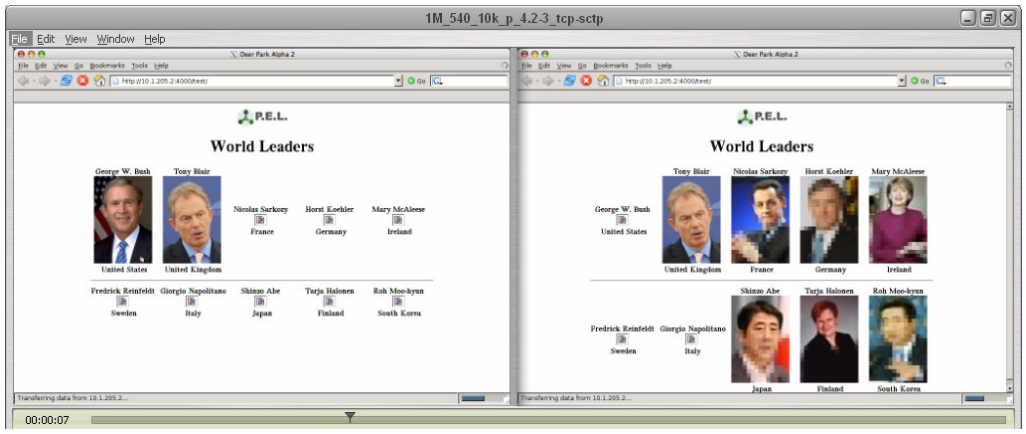
2.7.4 Concurrent Rendering and Progressive Images

Progressive images (e.g., JPEG, PNG) are coded such that the initial TPDUs approximate the entire image, and successive TPDUs gradually improve the image's quality/resolution. Via simple experiments, we demonstrate how concurrent rendering considerably improves user perception of progressive images. The example web page consists of an initial 1K image of our lab's logo, followed by 10 progressive JPEG images of world leaders, each of size 10K.

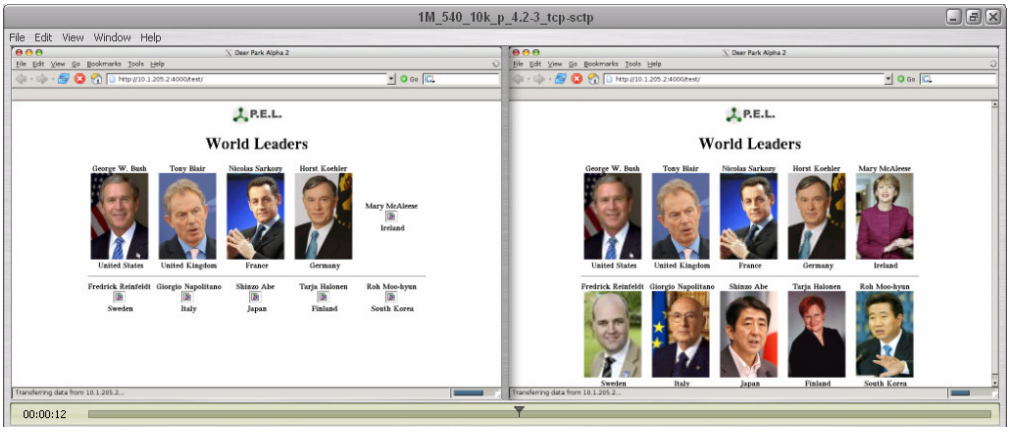
Both Firefox over TCP (sequential) and Firefox over SCTP (concurrent) download the example web page over a 56Kbps link with 1080ms RTT. The full page downloads were captured as movies, and are available online at [Movies]. In the snapshots shown in Figure 2.8, both sequential (left) and concurrent (right) runs experienced ~4.3% loss. Both rendering schemes start the download at t=0s. At t=6s (Figure 2.8a), the sequential scheme rendered a complete image followed by a good quality 2nd image, and the concurrent scheme displayed a complete image on the browser window.



(a): t=6 seconds



(b): t=7 seconds



(c): t=12 seconds

Figure 2.8: Concurrent Rendering of Progressive Images (56Kbps.1080ms; 4.3% loss)

At t=7s (Figure 2.8b), sequential rendering displays 2 complete images, vs. concurrent rendering's 7 partial images, at least 4 of which are of good quality. At

t=12s (Figure 2.8c), sequential rendering displays 4 complete images, whereas concurrent rendering presents the user with all 10 images of good quality. With concurrent rendering, the complete page is rendered only ~t=23s. From t=12s to 23s, all 10 images get refined, but the value added by the refinement is negligible to the human eye. Therefore, the user "perceives" all images to be complete by t=12s, while the page rendering time is actually t=23s. In the sequential run, all 10 images do not appear on the screen until t=26s.

2.7.5 SCTP Implementation and Concurrent Rendering

As mentioned earlier, our primary reason for choosing the FreeBSD platform is the availability of the SCTP reference implementation on FreeBSD. Section 2.5.2.1.3 discussed the unique interaction between Apache server and FreeBSD SCTP's round-robin scheduling of application messages over stream send queues. This interaction enabled concurrent rendering even during no packet losses, and increased the degree of concurrent rendering during lossy conditions. Consequently, absence of this interaction may lower the degree of concurrent rendering. For example, on platforms where SCTP implementations do FIFO or some other scheduling of application messages, concurrent rendering's $_{P}Page$ values may not be as dispersed as shown in Figure 2.7, but will be more dispersed than the corresponding values for sequential rendering.

2.8 Multiple TCP Connections vs. Single Multistreamed SCTP Association

The current workaround to reduce HOL blocking and improve an end user's perceived WWW performance is to download an HTTP transfer over multiple TCP connections. This section compares the two approaches proposed to improve

HTTP performance – multiple TCP connections vs. a *single* multistreamed SCTP association. Similar to Section 2.7, investigations here focus on browsing conditions most likely to exist in the developing world.

2.8.1 Background

In congestion-controlled transports such as TCP and SCTP, the amount of outstanding (unacknowledged) data is limited by the data sender's cwnd. Immediately after connection establishment, the sender can transmit up to initial cwnd bytes of application data [RFC3390, RFC4960]. Until congestion detection, both TCP and SCTP employ the slow start algorithm that doubles the cwnd every RTT. Consequently, the higher the initial cwnd, the faster the cwnd growth and more data gets transmitted every RTT. When an application employs N TCP connections, during the slow start phase, the connections' aggregate initial cwnd and their cwnd growth increases N-fold. Therefore, until congestion detection, an application employing N TCP connections can, in theory, experience up to N times more throughput than an application using a single TCP connection.

When a TCP or SCTP sender detects packet loss, the sender halves the cwnd, and enters the congestion avoidance phase [Jacobson 1988, RFC4960]. If an application employing N TCP connections experiences congestion on the transmission path, not all of the connections may suffer loss. If *M* of the *N* open TCP connections suffer loss, the multiplicative decrease factor for the connection aggregate is *(1 - M/2N)* [Balakrishnan 1998a]. If this decrease factor is greater than one-half (which is the case unless all *N* connections experience loss, i.e., $M<N$), the connections' aggregate cwnd and throughput increase after congestion detection is *more* than N times that of a single TCP connection.

On the whole, an application employing multiple TCP senders exhibits an aggressive sending rate, and consumes a higher share of the bottleneck bandwidth than an application using fewer or single TCP connection(s) [Mahdavi 1997, Balakrishnan 1998a]. Multiple TCP connections' aggressive sending behavior has been shown to increase throughput for various applications so far. [Tullimas 2008] employs multiple TCP connections to maintain the data streaming rate in multimedia applications. [Sivakumar 2000] proposes the PSockets library, which employs parallel TCP connections to increase throughput for data intensive computing applications. Likewise, we expect multiple TCP connections to improve HTTP throughput.

2.8.2 In-house HTTP 1.1 Client

The original plan was to use the Apache web server and the Firefox browser for the evaluations. But, following initial investigations, we decided to employ a custom built HTTP 1.1 client instead of Firefox due to the following reason.

In Firefox, the number of open transport connections to a server/proxy can be easily modified via user configuration. Firefox parses an URL, opens the first transport connection to the appropriate web server, and retrieves *index.html*. After parsing *index.html*, Firefox opens the remaining connection(s) to the server, and pipelines further requests across all connection(s) in a round-robin fashion. Initial investigations revealed that Firefox delays pipelining requests on a new transport connection. Specifically, the first HTTP transaction on a transport connection is always non-pipelined. After the successful receipt of the first response, subsequent requests on the same transport connection are then pipelined. We believe this behavior is Firefox's means of verifying whether a server supports persistent connections [RFC2616 Section 8]. However, this precautionary behavior increases the per connection transfer time by

at least 1 RTT, and packet losses during the first HTTP transaction further increase the transfer time. Clearly, this behavior is detrimental to HTTP throughput over multiple TCP connections. Also, this behavior interferes in the dynamics we are interested in investigating – interaction between multiple TCP connections and HTTP performance. Therefore, we developed a simple HTTP 1.1 client, which better models the general behavior of HTTP 1.1 over multiple transport connections, and does not bias results against multiple TCP connections.

The in-house client reproduces most of Firefox's transaction model, except that this client immediately starts pipelining on each new transport connection. The client employs either TCP or SCTP for the HTTP transfer. While one or more TCP connections are utilized for the HTTP 1.1 transfer, the complete page is downloaded using a single multistreamed SCTP association with maximum concurrency (each pipelined transaction is retrieved on a different SCTP stream). Additionally, the client mimics all of Firefox's interactions with the transport layer such as non-blocking reads/writes, and disabling the Nagle algorithm [RFC896]. The following algorithm describes the client in detail:

1. Setup a TCP or SCTP socket.
2. If SCTP, set appropriate data structures to request the required number of input and output streams during association establishment.
3. Connect to the server.
4. Timestamp "Page Download Start Time".
5. Request for *index.html*.
6. Receive and process *index.html*.
7. Make the socket non-blocking, and disable Nagle.

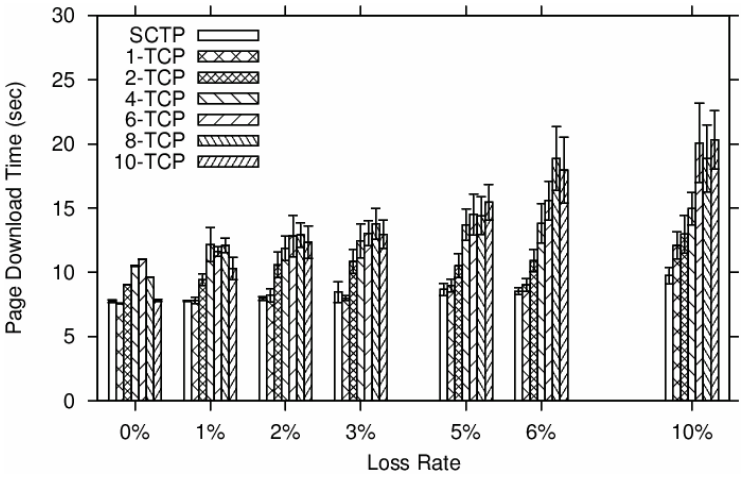
8. While there are more transport connections to be opened:
    8.1. Setup a socket (non-blocking, disable Nagle).
    8.2. Connect to the server.
9. While the complete page has not been downloaded:
    9.1. Poll for read, write, or error events on socket(s).
    9.2. Transmit pending requests on TCP connections or SCTP streams in a round-robin fashion.
    9.3. Read response(s) from readable socket(s).
10. Timestamp "Page Download End Time".

2.8.3 Experiment Parameters

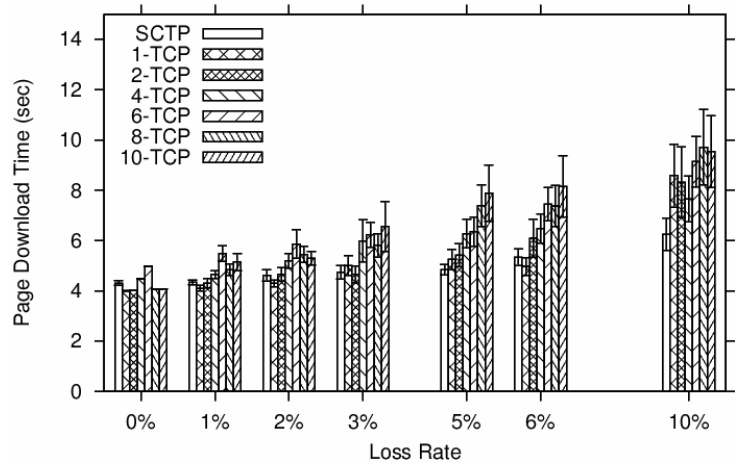
The sample web page used in the emulations comprises an *index.html* with 10 embedded objects. All embedded objects are the same size – 5KB. The impact of varying object sizes is discussed in Section 2.8.4.3.

Evaluations in Section 2.7 considered a 1Mbps last hop bandwidth, which is deemed to be a costly, high-end option for an average user in the developing world. Therefore, apart from a1Mbps last-hop, the following more limited last-hop bandwidths found in developing regions are considered [Du 2006]: 64Kbps, 128Kbps, and 256Kbps. Also, the following end-to-end propagation delays are considered [Cottrell 2006, PingER]:

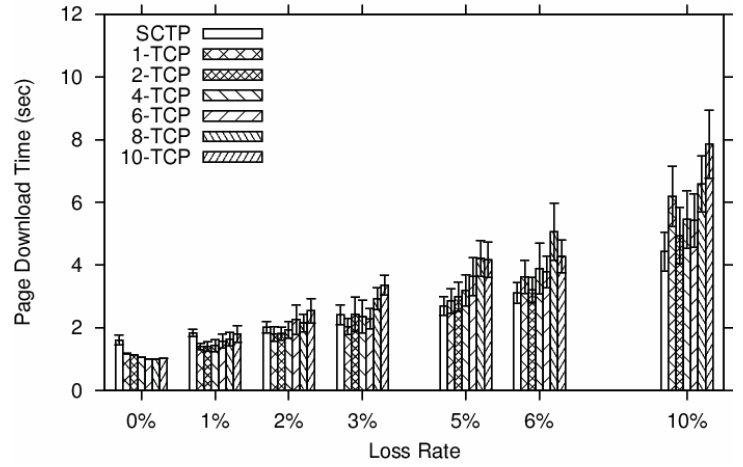
- 200ms RTT: User in East Asia, accessing a web server in North America over a land line.
- 350ms RTT: User in South Asia, accessing a web server in North America over a land line.
- 650ms RTT: User accessing a web server over a shared VSAT link.



(a): 64Kbps.200ms



(b): 128Kbps.200ms



(c): 1Mbps.200ms

Figure 2.9: HTTP Throughput (Object Size = 5K)

The FreeBSD TCP implementation tracks numerous sender and receiver related statistics including the number of timeout recoveries, and fast retransmits. After each TCP run, some of these statistics were gathered either directly from the TCP stack or using the *netstat* utility.

2.8.4 Results: HTTP Throughput

The HTTP page download time is measured as "Page Download End Time" – "Page Download Start Time" (Section 2.8.2). Figure 2.9 shows the HTTP page download times over a single multistreamed SCTP association (a.k.a. SCTP) vs. N TCP connections (N=1, 2, 4, 6, 8, 10; a.k.a. N-TCP) for the 64Kbps, 128Kbps and 1Mbps bandwidth scenarios. Results for 256Kbps bandwidth scenario can be found in [Natarajan 2008d]. Note that each embedded object is transmitted on a different TCP connection in 10-TCP, and employing more TCP connections is unnecessary. The values in Figure 2.9 are averaged over 40 runs (up to 60 runs for the 10% loss case), and plotted with 95% confidence intervals.

2.8.4.1 During No Congestion

Evaluations with 0% loss (Figure 2.9) help understand the behavior of multiple TCPs during congestion. As mentioned earlier, the initial cwnds of both TCP and SCTP are similar ─ 4MSS. Since there is no loss, both transports employ slow start during the entire page download. This equivalent behavior results in similar throughputs between SCTP and 1-TCP in 64Kbps and 128Kbps bandwidths. Recall from Section 2.7.2 that the packet-counting FreeBSD 6.1 TCP sender increases its cwnd more aggressively than an SCTP sender. As the available bandwidth increases

(256Kbps, 1Mbps), this difference in cwnd growth facilitates 1-TCP to slightly outperform SCTP [Natarajan 2008d].

As mentioned in Section 2.8.1, N-TCP's aggressive sending rate can increase an application's throughput by up to N times during slow start. Therefore, as the number of TCP senders increase, we expected multiple TCPs to outperform both 1-TCP and SCTP. Surprisingly, the results indicate that multiple TCPs perform similar to 1-TCP at 1Mbps and 256Kbps bandwidths [Natarajan 2008d]. As bandwidth decreases, multiple TCPs perform similar or worse (!) than both 1-TCP and SCTP. Further investigation revealed the following reasons.

2.8.4.1.1 Throughput Limited by Bottleneck Bandwidth

Low bandwidth pipes can transmit only a few packets per second. For example, a 64Kbps bottleneck cannot transmit more than ~5.3 1500byte PDUs per second or roughly 1 PDU per 200ms RTT. A single TCP sender's initial cwnd allows the server to transmit 4MSS bytes of pipelined responses back-to-back, causing a low bandwidth pipe (64Kbps, 128Kbps, and 256Kbps) to be fully utilized during the entire RTT. More data transmitted during this RTT cannot be forwarded, and gets queued at the bottleneck router. Therefore, data transmitted by $N \geq 2$ TCP senders do not contribute to reducing page download times, and N-TCPs perform similar to 1-TCP in 64Kbps (N=10), 128Kbps (N=8, 10), and 256Kbps (N>2) bandwidths [Natarajan 2008d]. The 1Mbps bottleneck is completely utilized by the initial cwnd of N=4 TCP senders (~16 1500byte PDUs per RTT). Therefore, $2 \leq N \leq 4$ TCP senders slightly improve page download times when compared to 1-TCP and N>4 TCP senders do not further reduce page download times.

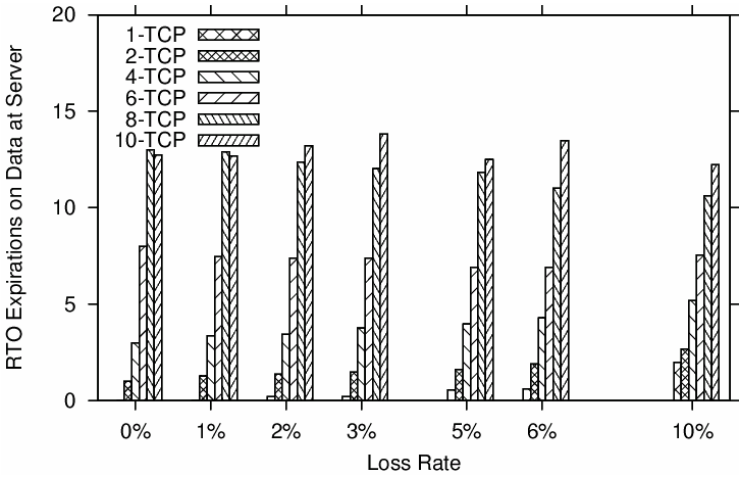
As the propagation delay and RTT increase, the bottleneck router forwards more packets per RTT. For example, the 1Mbps pipe can transmit ~53 PDUs per RTT in the 650ms scenario vs. ~16 PDUs per RTT in the 200ms scenario. Consequently, more TCP senders help fully utilize the 1Mbps pipe at 650ms RTT, and N-TCPs decrease page download times [Natarajan 2008d]. However, similar to the 200ms RTT scenario, lower bandwidths limit HTTP throughput, and N-TCPs perform similar to 1-TCP in the 350ms and 650ms RTTs [Natarajan 2008d]

To summarize, *HTTP throughput improvement is limited by the available bandwidth in a low bandwidth last hop. As bandwidth decreases, fewer TCP senders will fully utilize the available bandwidth, and additional TCP senders just increase the queuing delay and decrease throughput.*

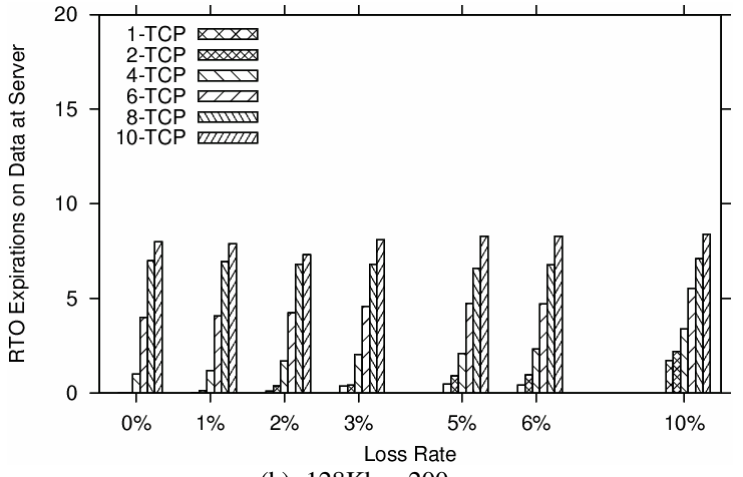
2.8.4.1.2 Queuing Delay at the Bottleneck

Figure 2.10 shows the mean number of timeout expirations on data at the server for the 64Kbps, 128Kbps and 1Mbps bandwidth scenarios. Note that the values plotted are the *mean timeouts per HTTP transfer*. When N>1 TCP senders are employed for the HTTP transfer, the plotted values denote the sum of timeouts across *all N* senders. We first focus on the values at 0% loss. Surprisingly, except 1Mbps, some TCP sender(s) in the other bandwidth scenarios undergo timeout recoveries. Since no packets were lost, these timeouts must be spurious, and are due to the following.

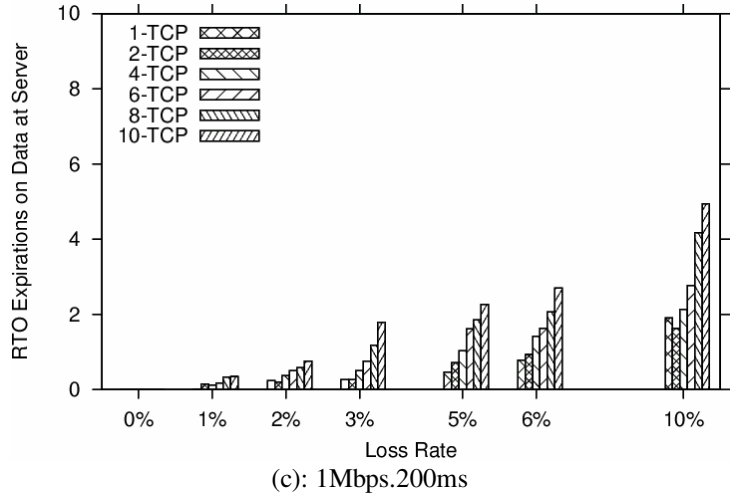
During connection establishment, a FreeBSD TCP sender estimates the RTT, and calculates the retransmission timeout value (RTO) [FreeBSD, RFC2988]. For a 200ms RTT, the calculated RTO equals the recommended minimum of 1 second [RFC2988].



(a): 64Kbps.200ms



(b): 128Kbps.200ms



(c): 1Mbps.200ms

Figure 2.10: RTO Expirations on Data at Server (Object Size = 5K)

Connection establishment is soon followed by data transfer from the server. Lower bandwidth translates to higher transmission and queuing delays. In a 64Kbps pipe, the transmission of one 1500byte PDU takes ~186ms, and a queue of ~5 such PDUs gradually increases the queuing delay and the RTT to more than 1 second. When outstanding data remains unacknowledged for more than the 1 second RTO, the TCP sender(s) (wrongly) assume data loss, and spuriously timeout and retransmit unacknowledged data.

As the number of TCP senders increase, more packets arrive at the bottleneck, and the increased queuing delay triggers spurious timeouts at a greater number of TCP senders. Of the 4 bandwidth scenarios considered, the1Mbps transfers experience the smallest queuing delay, and do not suffer from spurious timeouts. *As the bottleneck bandwidth decreases, queuing delay increases. Therefore HTTP transfers over smaller bandwidths experience more spurious timeouts.*

A spurious timeout is followed by unnecessary retransmissions and cwnd reduction. *If the TCP sender has more data pending transmission,* spurious timeouts delay new data transmission, and increase page download times (N=2, 4, 6, 8 TCP in 64Kbps, and N=4, 6 TCP in 128Kbps). As the number of TCP connections increase, fewer HTTP responses are transmitted per connection. For example, each HTTP response is transmitted on a different connection in 10-TCP. Though the number of spurious timeouts (and unnecessary retransmissions) is highest in 10-TCP, the TCP receiver delivers the first copy of data to the HTTP client, and discards the spuriously retransmitted copies. Therefore, 10-TCP's page download times are unaffected by the spurious timeouts. Nonetheless, spurious timeouts cause wasteful retransmissions that compete with other flows for the already scarce available bandwidth.

As the propagation delay increases, the RTO calculated during connection establishment is increased (> 1 second). Since transmission and queuing delays remain unaffected, they impact the RTT less at higher propagation delays. Consequently, spurious timeouts slightly decrease at 350ms and 650ms RTTs, but still remain significant at lower bandwidths, and increase page download times [Natarajan 2008d].

To summarize, the aggressive sending rate of multiple TCP senders during slow start does NOT necessarily translate to improved HTTP throughput in low bandwidth last hops. Bursty data transmission from multiple TCP senders increases queuing delay causing spurious timeouts. The unnecessary retransmissions following spurious timeouts (i) compete for the already scarce available bandwidth, and (ii) adversely impact HTTP throughput when compared to 1-TCP or SCTP. The throughput degradation is more noticeable as the bottleneck bandwidth decreases.

2.8.4.2 During Congestion

Though SCTP and TCP congestion control are similar, minor differences such as SCTP's byte counting and more accurate gap-ack information improve SCTP's loss recovery and throughput (Section 2.7.2). As the loss rate increases, SCTP's better congestion control offsets FreeBSD TCP's extra ack advantage during no loss, and SCTP outperforms 1-TCP.

Recall from Section 2.8.1 that N-TCPs' (N>1) aggressive sending rate during congestion avoidance can, in theory, increase throughput by more than N times. Therefore, we expected multiple TCPs to outperform both 1-TCP and SCTP. *On the contrary, multiple TCP connections worsen HTTP page download times, and the degradation becomes more pronounced as loss rate increases.* This observation is true

for all 4 bandwidth scenarios studied. Further investigation revealed the following reasons.

2.8.4.2.1 Increased Number of Timeout Recoveries at the Server

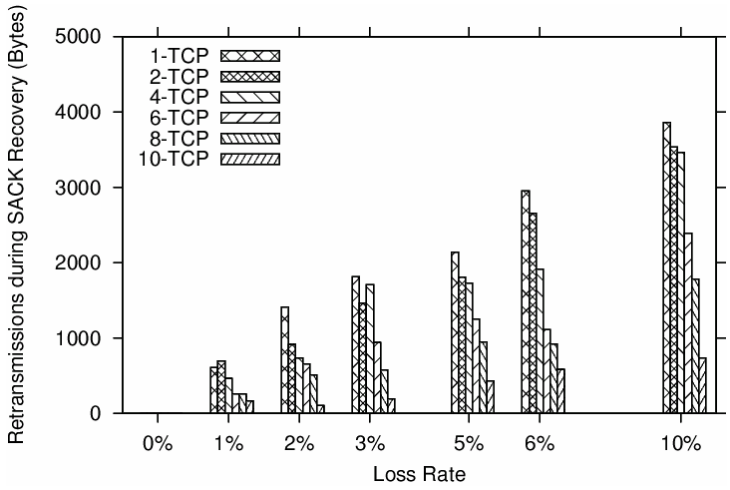
For every loss rate, the mean number of timeout expirations at the server increases as the number of TCP senders increases (Figure 2.10). Section 2.8.4.1.2 discussed how increased queuing delays cause spurious timeouts even at 0% loss. Such spurious timeouts, observed during lossy conditions as well, delay new data transmission, thus worsening HTTP page download times.

Recall that the 1Mbps transfers did not suffer spurious timeouts (0% loss in Figure 2.10c). However, multiple TCPs still amplify timeout expirations in 1Mbps transfers. Further investigation revealed that multiple TCPs reduce ack information which is crucial for fast retransmit-based loss recoveries.

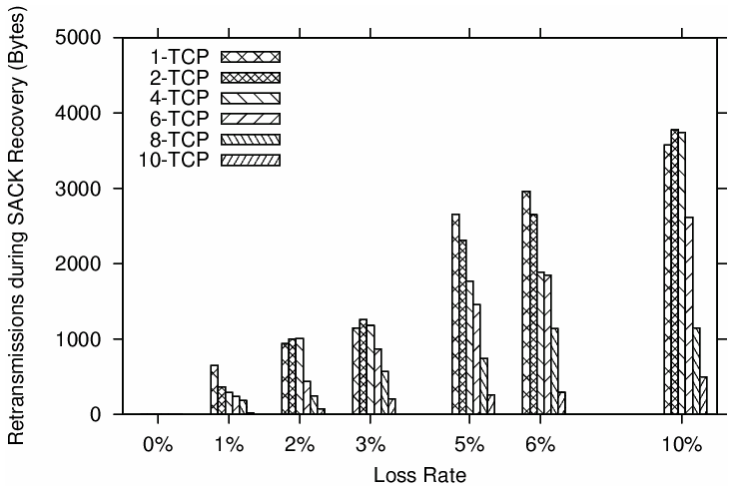
Figure 2.11 shows the average number of bytes retransmitted during TCP SACK recovery episodes (fast recovery) in the 64Kbps and 1Mbps transfers, respectively. (Results for the other intermediate bandwidths were similar and hence not shown.). Each value represents retransmissions from the server to client, and does not include retransmissions after timeout expirations. Similar to values in Figure 2.10, each value in Figure 2.11 represents the *average bytes retransmitted per HTTP transfer*, i.e., bytes retransmitted by *all N* TCP senders.

During 0% loss, data is always received in-order at the client. The acks from client to server contain no SACK blocks, and the server does not undergo SACK recoveries (Figure 2.11). During loss, data received out-of-order at the client triggers dupacks containing SACK blocks. On receiving 3 dupacks, a TCP sender enters SACK recovery and fast retransmits missing data [FreeBSD]. Higher loss rates trigger more

SACK recovery episodes, and increase retransmissions during SACK recoveries (Figure 2.11). However, for a given loss rate, the retransmissions decrease as the number of TCP connections increase. That is, for the same fraction of lost HTTP data (same loss rate), loss recoveries based on fast retransmits decrease as the number of TCP senders increase.



(a): 64Kbps.200ms



(b): 1Mbps.200ms

Figure 2.11: Fast Retransmits during SACK Recovery (Object Size = 5K)

Note that loss recovery based on fast retransmit relies on dupack information from the client. As the number of TCP connections increase, data

transmitted per connection decreases, thus reducing the number of potential dupacks arriving at each TCP sender. Ack losses on the reverse path further decrease the number of dupacks received. While the TCP senders implement Limited Transmit [RFC3042] to increase dupack information, the applicability of Limited Transmit diminishes as the amount of data transmitted per TCP connection decreases.

In summary, *increasing the number of TCP connections decreases per connection dupack information. Fewer dupacks reduce the chances of fast retransmit-based loss recovery, resulting in each sender performing more timeout-based loss recoveries.*

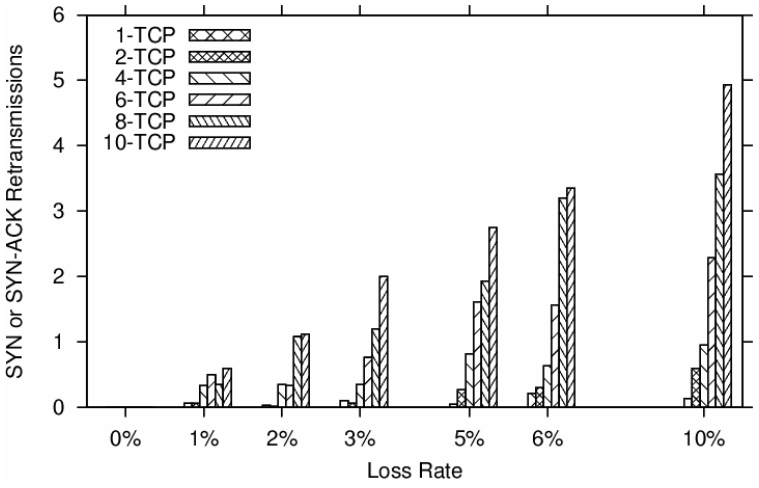
2.8.4.2.2 Increased Connection Establishment Latency

The in-house HTTP client, which closely resembles Firefox's transaction model, first opens a single TCP connection to the server, and retrieves and parses *index.html*. Then, the client establishes more TCP connection(s) for requesting embedded objects in a pipelined fashion. Note that HTTP requests can be transmitted over these connections only after successful connection establishment, i.e., only when the TCP client has successfully sent a SYN and received a SYN-ACK. Any delay in connection establishment due to SYN or SYN-ACK loss delays HTTP request (and response) transmission.

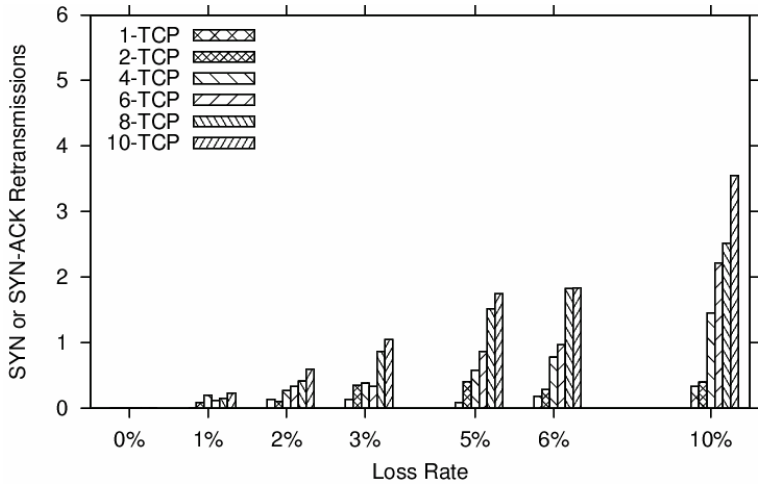
Figure 2.12 shows the average number of SYN or SYN-ACK retransmissions for the 64Kbps and 1Mbps transfers, respectively. (Results for the other intermediate bandwidths were similar and hence not shown.) When multiple TCP connections are employed for an HTTP transfer, the number of SYN, SYN-ACK packets increase, and the probability of a SYN or SYN-ACK loss increases. Therefore,

the number of SYN or SYN-ACK retransmissions tends to increase as the number of TCP connections increase.

A SYN or SYN-ACK loss can be recovered only after the recommended initial RTO value of 3 seconds [RFC2988], and increases the HTTP page download time by at least 3 seconds. Consequently, losses during connection establishment degrade HTTP throughput more when the time taken to download HTTP responses (after connection establishment) is smaller compared to the initial RTO value.

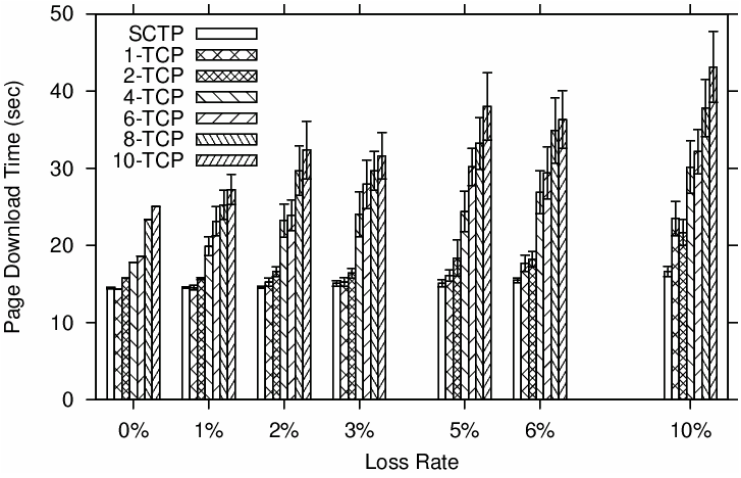


(a): 64Kbps.200ms

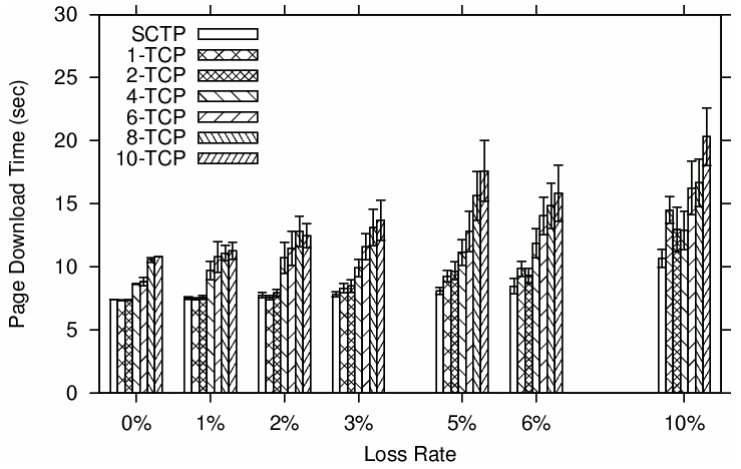


(b): 1Mbps.200ms

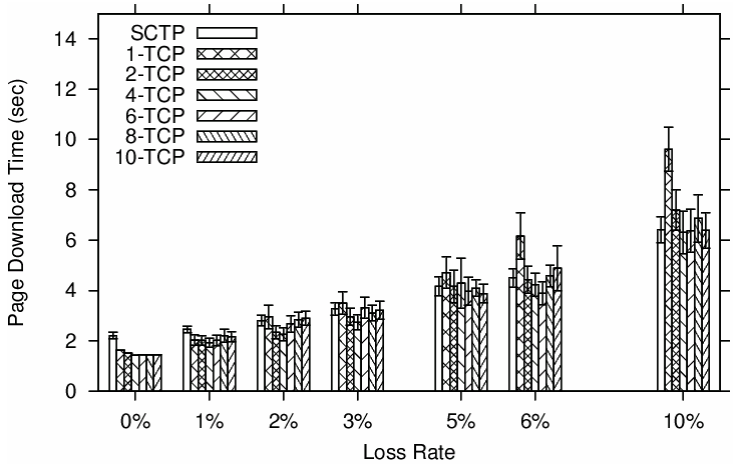
Figure 2.12: SYN or SYN-ACK Retransmissions (Object Size = 5K)



(a): 64Kbps.200ms



(b): 128Kbps.200ms



(c): 1Mbps.200ms

Figure 2.13: HTTP Throughput (Object Size = 10K)

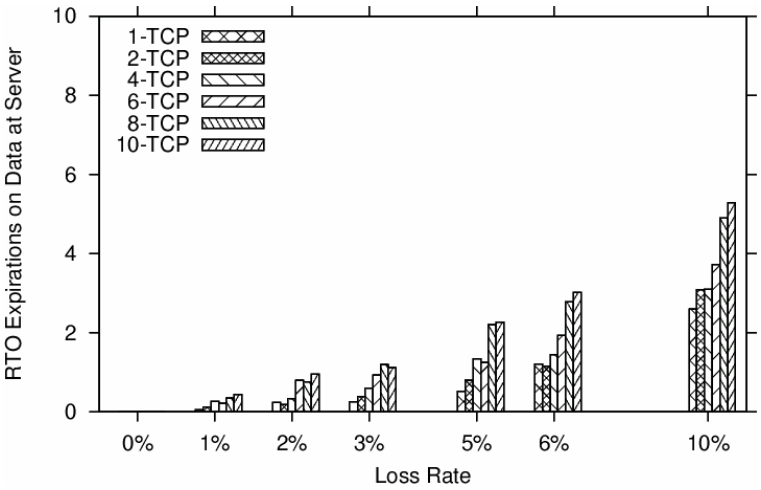


Figure 2.14: RTO Expirations on Data at Server (1Mbps.200ms; Object Size = 10K)

2.8.4.3 Impact of Varying Object Sizes

To investigate how object size impacts HTTP throughput, we repeated the emulations with larger (10K) embedded objects. The results are shown in Figure 2.13. Comparing Figures 2.9 and 2.13, we see that the trends between 1-TCP and multiple TCPs remain similar between the 5K and 10K transfers for all bandwidth scenarios except 1Mbps. In 1Mbps, N-TCPs perform better than 1-TCP, and the improvement is more pronounced at higher loss rates.

Figure 2.14 shows the server's mean timeout recoveries for the 10K transfers in the 1Mbps scenario. Comparing values in Figure 2.14 with Figure 2.10c, we see that the 10K transfers suffered fewer timeout recoveries *per transfer time unit* than 5K transfers. In the 10K transfers, each TCP sender transfers more data and receives more dupacks per TCP connection than the 5K transfers (Section 2.8.4.2.1). The increased flow of acks in the 10K transfers triggered more fast-retransmissions in SACK recovery episodes, and fewer timeout-based recoveries compared to the 5K transfers (Figure 2.14). Consequently, N-TCPs improved HTTP throughput in the 10K

transfers. However, as the last hop bandwidth decreases, the negative consequences of multiple TCP senders, such as increased queuing delay and connection establishment latency, increase the page download times, and N-TCPs perform similar to or worse than 1-TCP. *More importantly, note that, SCTP's enhanced loss recovery helps outperform N-TCPs even in the 10K transfers.*

To summarize, object size affects HTTP throughput over multiple TCP connections. *Smaller objects reduce dupack information per TCP connection and degrade HTTP throughput more than bigger objects. However, the impact of object size decreases, and the negative consequences of multiple TCP senders dominate more and bring down HTTP throughput at lower bandwidths.*

2.9 Conclusion, Ongoing and Future Work

We examined HOL blocking and its effects on web response times in HTTP over TCP. We proposed a multistreamed web transport such as SCTP to alleviate HOL blocking, and designed and implemented HTTP over SCTP in the Apache web server and Firefox browser. Emulation evaluations demonstrate that HTTP over TCP suffers from exacerbated HOL blocking which worsened response times in the high latency and lossy browsing conditions found in the developing world. On the contrary, SCTP streams eliminate inter-object HOL blocking, and improve web response times. The improvements are more visually perceivable in high latency and lossy end-to-end paths found in the developing world.

The current workaround to improve an end user's perceived WWW performance is to download an HTTP transfer over multiple TCP connections. While we expected multiple TCP connections to improve HTTP throughput, emulation results showed that the competing and bursty nature of multiple TCP senders degraded

HTTP performance especially in low bandwidth last hops. In such browsing conditions, a single multistreamed SCTP association not only eliminates HOL blocking, but also boosts throughput compared to multiple TCP connections.

Our body of work in HTTP over SCTP has stimulated significant interest in the area. The Protocol Engineering Lab has also secured funding through Cisco Systems' University Research Program for some of the ongoing activity discussed below.

2.9.1 IETF Internet Draft

We have proposed an Internet Draft (ID) to standardize our HTTP over SCTP streams design [Natarajan 2008f]. This ID was presented at the 73rd IETF Meeting held at Minneapolis in November 2008. The objectives of this ID are: (i) to highlight SCTP services that better match the needs of HTTP-based applications, (ii) to propose the HTTP over SCTP streams design, and (iii) to share important lessons learnt while implementing HTTP over SCTP in Apache and Firefox.

2.9.2 SCTP-enabled Apache and Firefox

Jonathan Leighton is heading this on-going effort to integrate our HTTP over SCTP design and implementation into the Firefox distribution from mozilla.org, and the Apache distribution from apache.org. The current activity is focused on integrating SCTP related APIs in the Netscape Portable Runtime (NSPR) API and the Apache Portable Runtime (APR) API, which offer platform independent network implementations to Firefox and Apache, respectively. Subsequent work will focus on modifying Firefox and Apache to take advantage of these SCTP related APIs, and

enabling appropriate SCTP related compile options for various platforms and SCTP implementations.

2.9.3 Minimizing Resource Requirements

As mentioned in Section 2.8, today's web browsers reduce HOL blocking in HTTP over TCP by downloading an HTTP transfer over multiple TCP connections. In contrast, a browser over SCTP eliminates HOL blocking by simply increasing the number of streams in the SCTP association. Each TCP connection or a pair of SCTP streams (inbound/outbound) increases the processing and resource overhead at the web server or proxy. However, the resources required to support a new pair of SCTP streams is much less compared to a new TCP connection. For example, on FreeBSD each inbound or outbound SCTP stream requires an additional 28 or 32 bytes, respectively, in the SCTP Protocol Control Block (PCB), while a new TCP PCB requires ~700 bytes [FreeBSD]. The difference in TCP vs. SCTP resource requirements increases with the number of clients, and can be significant at a web server farm handling thousands of clients. This difference can also be significant at intermediate entities such as web caches that serve many web clients and/or other caches [Squid].

The absolute difference in TCP vs. SCTP resource requirements depends not only on the respective protocol implementations but also on how optimal the implementations are. While the TCP stack has been optimized over the past two decades, the SCTP stack is relatively new, and the SCTP reference implementation on FreeBSD can be optimized further. For example, Randall Stewart, the designer of FreeBSD SCTP estimates that the FreeBSD SCTP PCB size can be reduced by ~600

bytes. Evaluating TCP vs. SCTP resource usage make more sense after such optimizations are in place.

2.9.4 Impact on Developing Regions

While HTTP over SCTP promises better response times in high propagation delay/low bandwidth/lossy browsing conditions, it is impractical to expect all web servers to provide web over SCTP in the immediate future, without which SCTP's benefits cannot be leveraged. To address this issue, we propose a realistic, low cost, gateway-based solution that translates HTTP over TCP to HTTP over SCTP streams for easier and localized deployment. The solution assumes that the web browser is capable of HTTP over SCTP, similar to the SCTP-enabled, freely available Firefox browser used in our emulations. The gateway is physically positioned between the server and client, such that, the gateway talks SCTP to clients over the last hop with high propagation delay and/or low bandwidth, and talks TCP to web servers in the outside world. For the architecture shown in Figure 2.2, the gateway is positioned between the VSAT ground station (on the left) and the Internet cloud. We believe that the "proxy" configuration in the SCTP-enabled Apache server is a good starting point to achieve the gateway functionality at minimal monetary cost [Apache].

At a minimum, a gateway solution should provide faster page downloads than HTTP over TCP. This solution can be extended to further enhance pipelined objects' response times. For example, the gateway could use batch image conversion software [Gimp] to convert embedded non-progressive JPEG or PNG images to their corresponding progressive versions before forwarding them to the clients. Image conversion at the gateway takes on the order of milliseconds per image, but can improve a user's response times on the order of seconds.

2.10 Related Work

Significant interest exists for designing new transport and session protocols that better suit the needs of HTTP-based client-server applications than TCP. As mentioned earlier, several experts agree that the best transport scheme for HTTP would be one that supports datagrams, provides TCP compatible congestion control on the entire datagram flow, and facilitates concurrency in GET requests [Gettys 2002]. WebMUX [Gettys 1998] was one such session management protocol that was a product of the (now historic) HTTP-NG working group [HTTP-NG]. WebMUX proposed using a reliable transport protocol to provide web transfers with streams for transmitting independent objects. However, the WebMUX effort did not mature.

[Ford 2007] proposes the use of Structured Stream Transport (SST) for web transfers. SST was proposed after [Natarajan 2006a] and functions similar to SCTP streams. SST extends TCP to provide multiple streams over a TCP-friendly transport connection. Simulation-based evaluations in [Ford 2007] show that SST provides similar page download times as TCP. The primary contribution of a multistreamed web transport is the reduction in HOL blocking, which is the focus of our work. Using real implementations, we show that reduced HOL blocking in HTTP over SCTP results in visually perceivable improvements to individual objects' response times in browsing conditions typical of developing regions. Also, we note that SCTP is a standardized IETF protocol with many fine-tuned kernel space implementations, while SST is a research protocol yet to be standardized.

Apart from new session and transport protocols, other sender-side techniques focus on reducing the adverse effects of the current workaround to reduce HOL blocking – parallel TCP connections. The Congestion Manager (CM) [RFC3124] is a shim layer between the transport and network layers which aggregates congestion

control at the end host, thereby enforcing a fair sending rate when an HTTP transfer employs multiple TCP connections. "TCP Session" [Padmanabhan 1998] proposes integrated loss recovery across multiple TCP connections to the same web client (these multiple TCP connections are together referred to as a TCP session). All TCP connections within a session are assumed to share the transmission path to the web client. A Session Control Block (SCB) is maintained at the sender to store information about the shared path such as its cwnd and RTT estimate. While CM and TCP Session reduce the adverse effects of parallel TCP connections on the network and the application, these solutions still require a web browser to open multiple TCP connections, thereby increasing the web server's resource requirements.

Content Delivery Networks (CDNs) replicate web content across geographically distributed servers, and reduce response times for web users by redirecting requests to a server closest to the client. [Krishnamurthy 2001] confirms that CDNs reduce average web response times for web users along USA's east coast for static content. Unfortunately, little research exists on the prevalence of CDNs for content providers and web users outside of developed nations. Also, CDNs cannot lessen web response times when latency is due to (i) propagation delay and/or low bandwidth last hop, as is the case in developing regions, or (ii) sub-optimal traffic routing that increases end-to-end path RTTs [Baggaley 2007].

Chapter 3

NON-RENEGABLE SACKS (NR-SACKS) FOR SCTP

This chapter discusses a fundamentally new transport layer acknowledgment mechanism called Non-Renegable Selective Acks (NR-SACKs). Sections 3.1 and 3.2 introduce reneging in current transport protocol implementations and the inefficiencies with TCP and SCTP SACK mechanisms when received data is non-renegable. Section 3.3 proposes NR-SACKs for SCTP, and discusses the specifics of SCTP's NR-SACK chunk. Sections 3.4 and 3.5 discuss simulation preliminaries and present results comparing SACKs vs. NR-SACKs in both SCTP and CMT. Finally, Section 3.6 concludes and presents ongoing and future work.

3.1 Introduction

Reliable transport protocols such as TCP and SCTP employ two kinds of data acknowledgment mechanisms: (i) cumulative acks (cum-acks) indicate data that has been received in-sequence, and (ii) selective acknowledgments (SACKs) indicate data that has been received out-of-order. In both TCP and SCTP, while cum-acked data is the receiver's responsibility, SACKed data is not, and SACK information is *advisory* [RFC3517, RFC4960]. While SACKs notify a sender about the reception of specific out-of-order TPDUs, the receiver is permitted to later discard the TDPUs. Discarding data that has been previously SACKed is known as *reneging*. Though reneging is a possibility, the conditions under which current transport layer and/or

operating system implementations renege, and the frequency of these conditions occurring in practice (if any) are unknown and needs further investigation.

Data that has been delivered to the application, by definition, is non-renegable by the transport receiver. Unlike TCP which never delivers out-of-order data to the application, SCTP's multistreaming and unordered data delivery services (Chapter 1) result in out-of-order data being delivered to the application and thus becoming non-renegable. Interestingly, TCP and SCTP implementations can be configured such that the receiver is not allowed to and therefore never reneges on out-of-order data (details in Section 3.2). In these configurations, even non-deliverable out-of-order data becomes non-renegable.

The current TCP or SCTP SACK mechanism does not differentiate between out-of-order data that "has been delivered to the application and/or is non-renegable" vs. data that "has not yet been delivered to the application and is renegable". In this work, we introduce a fundamentally new third acknowledgment mechanism called Non-Renegable Selective Acknowledgments (NR-SACKs) that enable a transport receiver to explicitly convey non-renegable information to the sender on some or all out-of-order TPDUs. While this work introduces NR-SACKs for SCTP, the NR-SACKs idea can be applied to any reliable transport protocol that uses selective acknowledgments and/or permits delivery of out-of-order data, or where a receiver never reneges on previously acked data.

3.2 Problem Description

This section investigates the effect of SCTP's SACK mechanism in situations when out-of-order data is non-renegable, and identifies conditions under which SACKs hurt performance in an SCTP or CMT association.

3.2.1 Background

The SCTP (or TCP) send buffer, or the sender-side socket buffer (Figure 3.1), consists of two kinds of data: (i) new application data waiting to be transmitted for the first time, and (ii) copies of data that have been transmitted at least once and are waiting to be cum-acked, a.k.a. the *retransmission queue* (RtxQ). Data in the RtxQ is the transport sender's responsibility until the receiver has guaranteed their delivery to the receiving application, and/or the receiver guarantees not to renege on the data.

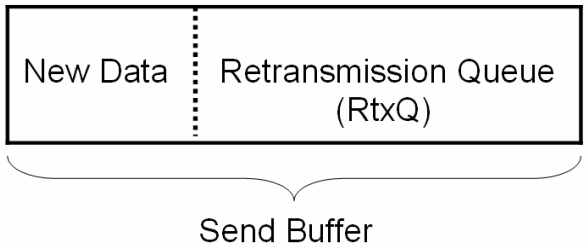


Figure 3.1: Transport Layer Send Buffer

In traditional in-order data delivery service, a receiver cum-acks the latest in-order data. Cum-acked data has either been delivered to the application or is ready for delivery. In either case, cum-acks are an explicit assurance that the receiver will not renege on the corresponding data. Upon receiving a cum-ack, the sender is no longer responsible, and removes the corresponding data from the RtxQ. In the current SACK mechanism, cum-acks are the only means to convey non-renegable information; all selectively acked (out-of-order) data are by default renegable.

As discussed in Chapter 1, SCTP's multistreaming service divides an end-to-end association into independent logical data streams. Data arriving in-sequence within a stream can be delivered to the receiving application even if the data is out-of-order relative to the association's overall flow of data. Also, data marked for unordered delivery can be delivered immediately upon reception, regardless of the data's position

within the overall flow of data. Thus, SCTP's data delivery services result in situations where out-of-order data is delivered to the application, and is thus non-renegable.

Operating systems allow configuration of transport layer implementations such that received out-of-order data is never reneged. For example, in FreeBSD, the *net.inet.tcp.do_tcpdrain* or *net.inet.sctp.do_sctp_drain* sysctl parameters can be configured to never revoke kernel memory allocated to TCP or SCTP out-of-order data, such that non-deliverable out-of-order data is non-renegable. Thus, out-of-order data can also be rendered non-renegable through simple user configuration.

In the following discussions, "non-renegable out-of-order data" refers to data for which the transport receiver takes full responsibility, and guarantees not to renege either because (i) the data has been delivered (or is deliverable) to the application, or (ii) the receiving system (OS and/or transport layer implementation) guarantees not to revoke the allocated memory until after the data is delivered to the application. With the current SACK mechanism, non-renegable out-of-order data is selectively acked, and is (wrongly) deemed renegable by the transport sender. Maintaining copies of non-renegable data in the sender's RtxQ is unnecessary.

3.2.2 Unordered Data Transfer using SACKs

Using a timeline diagram, this section discusses the effects of SACKs in transfers where all out-of-order is non-renegable. The discussion is applicable to any type of reliable data delivery service (in-order, partial-order, unordered) where all out-of-order data is non-renegable, but uses the simple unordered SCTP data transfer example shown in Figure 3.2.

In this example, the SCTP send buffer denoted by the rectangular box can hold a maximum of eight TPDUs. Each SCTP PDU is assigned a unique Transmission

Sequence Number (TSN). The timeline slice shown in Figure 3.2 picks up the data transfer at a point when the sender's cwnd C=8, allowing transmission of 8 TPDUs (arbitrarily numbered with TSNs 11-18). Note that when TSN 18 is transmitted, the RtxQ grows to fill the entire send buffer.

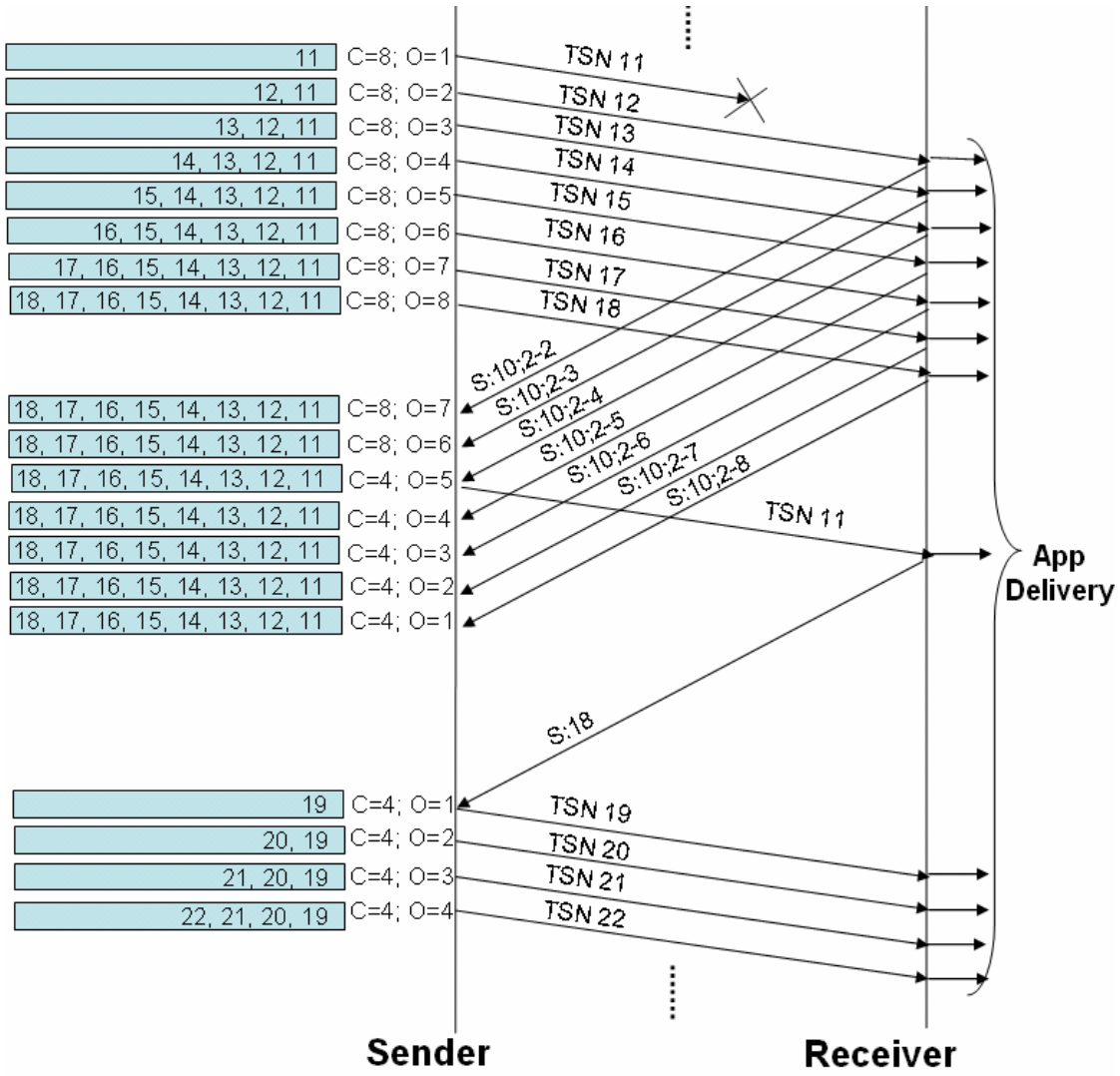


Figure 3.2: Unordered SCTP Data Transfer using SACKs

In this example, TSN 11 is presumed lost in the network. The other TSNs are received out-of-order and immediately SACKed by the SCTP receiver. The SACKs

shown have the following format: (S)ACK: CumAckTSN; GapAckStart-GapAckEnd. Each gap-ack start and gap-ack end value is relative to the cum-ack value, and together they specify a block of received TSNs.

At the sender, the first SACK (S:10;2-2) is also a dupack and gap-acks TSN 12. Though data corresponding to TSN 12 has been delivered to the receiving application, the SACK does not convey the non-renegable nature of TSN 12, requiring the sender to continue being responsible for this TSN. Starting from the time this SACK arrives at the sender, the copy of TSN 12 in the sender's RtxQ is *unnecessary*. The gap-ack for TSN 12 reduces the amount of outstanding data (O) to 7 TPDUs. Since O<C, the sender could in theory transmit new data, but in practice cannot do so since the completely filled send buffer *blocks* the sending application from writing new data into the transport layer. We call this situation *send buffer blocking*. Note that send buffer blocking prevents the sender from fully utilizing the cwnd.

The second and third dupacks (S:10;2-3, S:10;2-4) increase the number of unnecessary TSNs in the RtxQ, and send buffer blocking continues to prevent new data transmission. On receipt of the third dupack, the sender halves the cwnd (C=4), fast retransmits TSN 11, and enters fast recovery. Dupacks received during fast recovery further increase the amount of unnecessary data in the RtxQ, prolonging inefficient RtxQ usage. Note that though these dupacks reduce outstanding data (O<C), send buffer blocking prevents new data transmission.

The sender eventually exits fast recovery when the SACK for TSN 11's retransmission (S:18) arrives. The sender removes the unnecessary copies of TSNs 12-18 from the RtxQ, and concludes the current instance of send buffer blocking. Since send buffer blocking prevented the sender from fully utilizing the cwnd before, the new

cum ack (S:18) does not increase the cwnd [RFC4960]. The application writes data into the newly available send buffer space and the sender now transmits TSNs 19-22.

Based on the timeline in Figure 3.2, the following observations can be made regarding transfers with non-renegable out-of-order data:

- The unnecessary copies of non-renegable out-of-order data waste kernel memory (RtxQ). The amount of wasted memory is a function of *flightsize* (amount of data "in flight") during a loss event; a larger flightsize wastes more memory.
- When the RtxQ grows to fill the entire send buffer, send buffer blocking ensues, which can degrade throughput.

3.2.3 Implications to CMT

As discussed in Chapter 1, CMT is an experimental SCTP extension that exploits SCTP multihoming for simultaneous transfer of new data over multiple independent paths [Iyengar 2006]. Similar to an SCTP sender, the CMT sender uses a single send buffer and RtxQ for data transfer. However, the CMT sender's total flightsize is the sum of flightsizes on each path. Since the amount of kernel memory and the probability of send buffer blocking increase as the transport sender's flightsize increases (previous subsection), we hypothesize that a CMT association is even more likely than an SCTP association to suffer from the inefficiencies of the existing SACK mechanism.

3.3 Solution: Non-renegable Selective Acks

Non-Renegable Selective Acknowledgments (NR-SACKs) [Natarajan 2008a, Natarajan 2008e] enable a receiver to explicitly convey non-renegable

information on out-of-order data. In SCTP, NR-SACKs provide the same information as SACKs for congestion and flow control, and the sender is expected to process this information identical to SACK processing. In addition, NR-SACKs provide the added option to report some or all of the out-of-order data as being non-renegable.

3.3.1 NR-SACK Chunk Details

Before sending/receiving NR-SACKs, the endpoints first negotiate NR-SACK usage during association establishment. An endpoint supporting the NR-SACK extension lists the NR-SACK chunk in the Supported Extensions Parameter carried in the INIT or INIT-ACK chunk [RFC5061]. During association establishment, if both endpoints support the NR-SACK extension, then each endpoint acknowledges received data with NR-SACK chunks instead of SACK chunks.

The proposed NR-SACK chunk for SCTP is shown in Figure 3.3. Since NR-SACKs extend SACK functionality, the NR-SACK chunk has several fields identical to the SACK chunk: the *Cumulative TSN Ack,* the *Advertised Receiver Window Credit, Gap Ack Blocks,* and *Duplicate TSNs*. These fields have identical semantics to the corresponding fields in the SACK chunk [RFC4960]. NR-SACKs also report non-renegable out-of-order data chunks in the *NR Gap Ack Blocks*, a.k.a. "nr-gap-acks". Each NR Gap Ack Block acknowledges a continuous subsequence of non-renegable out-of-order data chunks. All data chunks with TSNs ≥ (Cumulative TSN Ack + NR Gap Ack Block Start) and ≤ (Cumulative TSN Ack + NR Gap Ack Block End) of each NR Gap Ack Block are reported as non-renegable. The *Number of NR Gap Ack Blocks (M)* field indicates the number of NR-Gap Ack Blocks included in the NR-SACK chunk.

Note that each sequence of TSNs in an NR Gap Ack Block will be a subsequence of one of the Gap Ack Blocks, and there can be more than one NR Gap Ack Block per Gap Ack Block. Also, non-renegable information cannot be revoked. If a TSN is nr-gap-acked in any NR-SACK chunk, then all subsequent NR-SACKs gap-acking that TSN should also nr-gap-ack that TSN. Complete details of NR-SACK chunk can be found in [Natarajan 2008a].

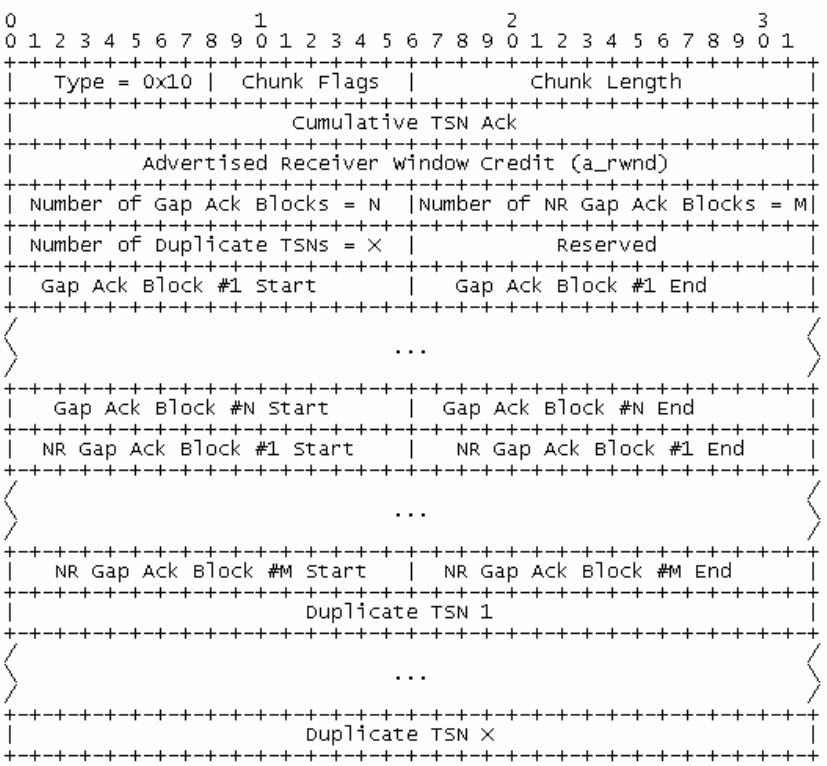


Figure 3.3: NR-SACK Chunk for SCTP

The second least significant bit in the *Chunk Flags* field is the *(A)ll* bit. If the ‘A’ bit is set to '1', all out-of-order data blocks acknowledged in the NR-SACK chunk are non-renegable. The ‘A’ bit enables optimized sender/receiver processing and reduces the size of NR-SACK chunks when all out-of-order TPDUs at the receiver are non-renegable.

3.3.2 Unordered Data Transfer using NR-SACKs

NR-SACKs provide an SCTP receiver with the option to convey non-renegable information on out-of-order data. When a receiver guarantees not to renege an out-of-order data chunk and nr-gap-acks the chunk, the sender no longer needs to keep that particular data chunk in its RtxQ, thus allowing the sender to free up kernel memory sooner than if the data chunk were only gap-acked.

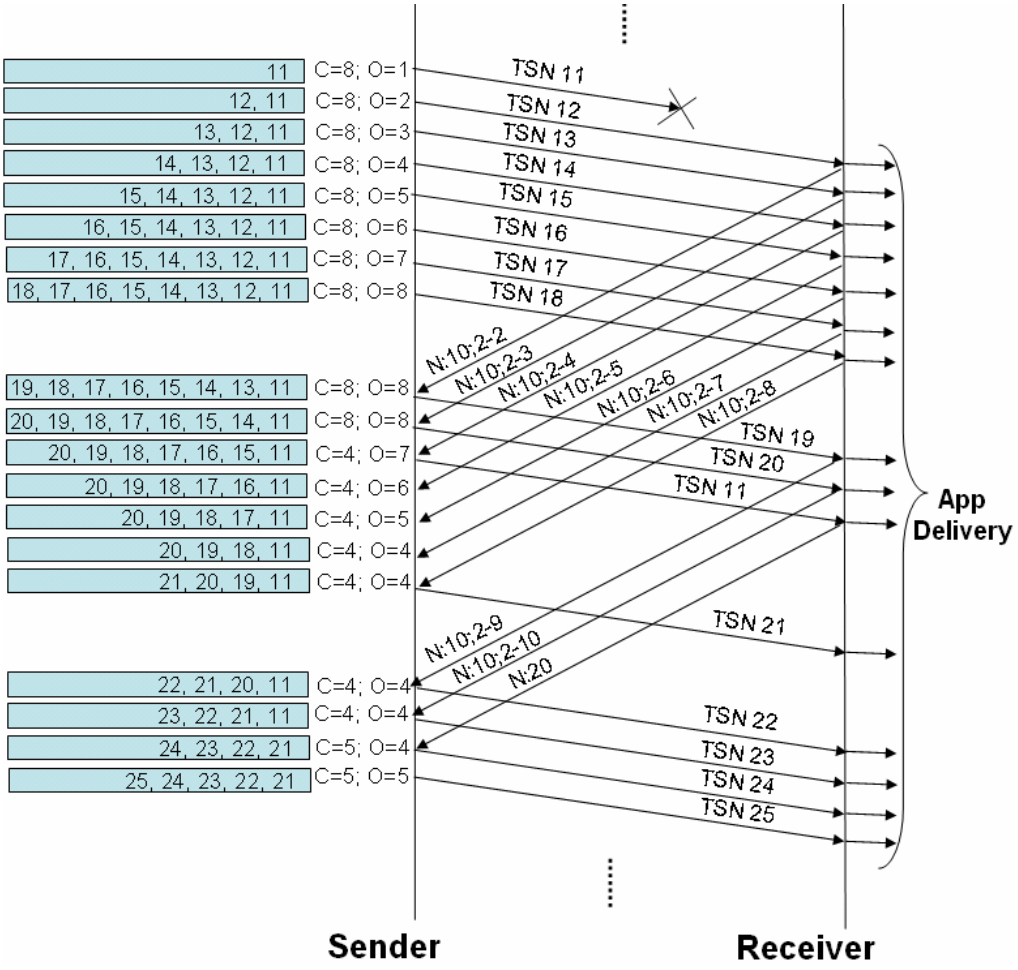


Figure 3.4: Unordered SCTP Data Transfer using NR-SACKs

Figure 3.4 is analogous to Figure 3.2's example, this time using NR-SACKs. The sender and receiver are assumed to have negotiated the use of NR-

SACKs during association establishment. As in the example of Figure 3.2, TSNs 11-18 are initially transmitted, and TSN 11 is presumed lost. For each TSN arriving out-of-order, the SCTP receiver transmits an NR-SACK chunk instead of SACK chunk. Since all out-of-order data are non-renegable in this example, every NR-SACK chunk has the 'A' bit set, and the nr-gap-acks report the list of TSNs that are both out-of-order and non-renegable.

All NR-SACKs in Figure 3.4 have the following format: (N)R-SACK: CumAckTSN; NRGapAckStart-NRGapAckEnd. The first NR-SACK (N:10;2-2) is also a dupack. This NR-SACK cum-acks TSN 10, and (nr-)gap-acks TSN 12. Once the data sender is informed that TSN 12 is non-renegable, the sender frees up the kernel memory allocated to TSN 12, allowing the application to write more data into the newly available send buffer space. Since TSN 12 is also gap-acked, the amount of outstanding data (O) is reduced to 7, allowing the sender to transmit new data – TSN 19.

On receipt of the second and third dupacks that newly (nr-)gap-ack TSNs 13 and 14, the sender removes these TSNs from the RtxQ. On receiving the second dupack, the sender transmits new data – TSN 20. On receipt of the third dupack, the sender halves the cwnd (C=4), fast retransmits TSN 11, and enters fast recovery. Dupacks received during fast recovery (nr-)gap-ack TSNs 15-20. The sender frees RtxQ accordingly, and transmits new TSNs 21, 22 and 23. The sender exits fast recovery when the NR-SACK with new cum-ack (N:20) arrives. This new cum-ack increments C=5, and decrements O=3. The sender now transmits new TSNs 24 and 25.

The explicit non-renegable information in NR-SACKs ensures that the RtxQ contains only *necessary* data – TPDUs that are actually in flight or "received and

renegable". Comparing Figures 3.2 and 3.4, we observe that NR-SACKs use the RtxQ more efficiently.

3.4 Evaluation Preliminaries

The ns-2 SCTP and CMT modules [NS-2, Ekiz 2007] were extended to support and process NR-SACK chunks. The simulation-based evaluations compare long-lived SCTP or CMT flows using SACKs vs. NR-SACKs under varying cross-traffic loads. This section discusses the experiment setup and other evaluation preliminaries in detail.

3.4.1 Simulation Setup

[Andrew 2008] recommends specific simulation setups and parameters for realistic evaluations of TCP extensions and congestion control algorithms. These recommendations include network topologies, details of cross-traffic generation, and delay distributions mimicking patterns observed in the Internet. We adhere to these recommendations for a realistic evaluation of SACKs vs. NR-SACKs.

The SCTP evaluations use the dumb-bell topology shown in Figure 3.5, which models the access link scenario specified in [Andrew 2008]. The central bottleneck link connects routers R_1 (left) and R_2 (right), has a 100Mbps capacity, and 2ms one-way propagation delay. Both routers employ drop tail queuing and the queue size is set to the bandwidth-delay product of a 100ms flow. Each router is connected to three cross-traffic generating edge nodes via 100Mbps edge links with the following propagation delays: 0ms, 12ms, 25ms (left) and 2ms, 37ms, 75ms (right). Each left edge node generates cross-traffic destined to every right edge node and vice-versa.

Thus, without considering queuing delays, the RTTs for cross-traffic flows sharing the bottleneck link range from 8ms—204ms.

[Andrew 2008] recommends application level cross-traffic generation over packet level generation, since, in the latter scenario, cross-traffic flows do not respond to the user/application/transport behavior of competing flows. Also, [Andrew 2008] proposes the use of Tmix [Weigle 2006] traffic generator. However, the recommended Tmix *connection vectors* were unavailable at the time of performing our evaluations. Therefore, we decided to employ existing ns-2 application level traffic generation tools, recommended by [Wang 2007a, Wang 2007b]. Since our simulation setup uses application level cross-traffic, we believe that the general conclusions from our evaluations will hold for evaluations using the Tmix traffic generator.

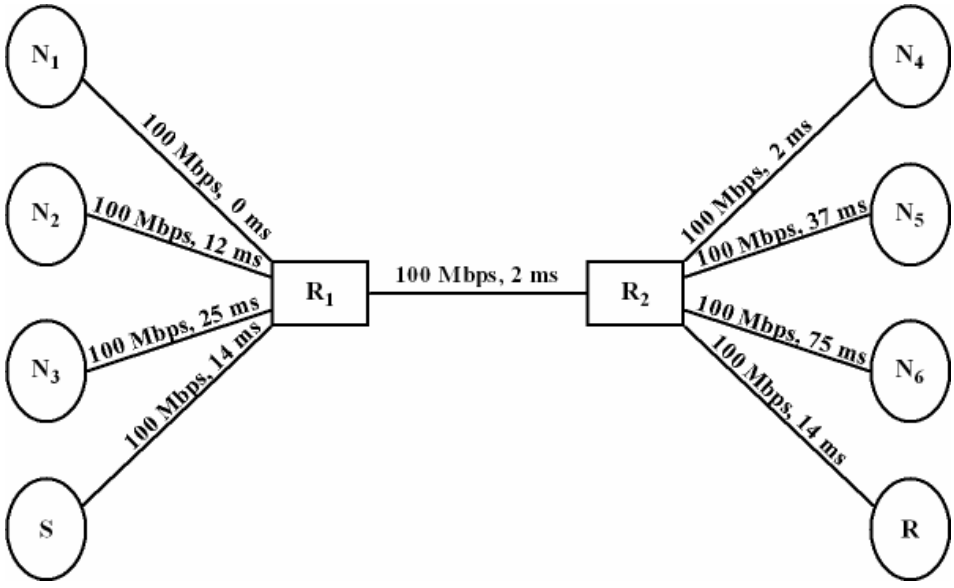


Figure 3.5: Topology for SCTP Experiments (Topology 1)

Cross-traffic generated by three kinds of applications are considered: (i) non-greedy, responsive HTTP sessions generated by PackMime implementation [Cao 2004], (ii) rate controlled, unresponsive video sessions over UDP, and (iii) greedy,

responsive bulk file transfer sessions over TCP. We are unaware of existing measurement studies on the proportion of each kind of traffic observed in the Internet. Therefore, the simulations assume a simple, yet reasonable rule for the traffic mix proportion – more HTTP traffic than video or FTP traffic.

Each edge node runs a PackMime session to every edge node on the other side, and the amount of generated HTTP traffic is controlled via the PackMime *rate* parameter. Similarly, each edge node establishes video and FTP sessions to every edge node on the other side, and the number of video/FTP sources on each node impacts the amount of video/FTP traffic. To avoid synchronization issues, the PackMime, video, and FTP sessions start at randomly chosen times during the initial 5 seconds of the simulation. The default segment size for all TCP traffic results in 1500 byte IP PDUs; the segment size for 10% of the FTP flows is modified to result in 576 byte IP PDUs. Also, the PackMime *request* and *response* size distributions are seeded in every simulation run, resulting in a range of packet sizes at the bottleneck [Andrew 2008].

The bottleneck router load is measured as (L) = (mean queue length ÷ total queue size). Four packet-level load/congestion variations are considered: (i) Low (~15% load, < 0.1% loss), (ii) Mild (~45% load, 1-2% loss), (iii) Medium (~60% load, 3-4% loss), (iv) Heavy (~85% load, 8-9% loss).

Topology 1 (Figure 3.5) is used to evaluate SCTP flows. CMT evaluations are over the dual-dumbbell topology shown in Figure 3.6 (topology 2). Topology 2 consists of two independent bottleneck links between routers R_1-R_2 and R_3-R_4. Similar to topology 1, each router in topology 2 is attached to 3 cross-traffic generating edge nodes, with similar bottleneck and edge link bandwidth/delay characteristics. In both topologies, nodes S and R are the SCTP or CMT sender and receiver, respectively. In

topology 2, both S and R are multihomed, and the CMT sender uses the two independent paths (paths 1 and 2) for simultaneous data transfer. In both topologies, S and R are connected to the bottleneck routers via 100Mbps duplex edge links, with 14ms one-way delay. Thus, the one-way propagation delay experienced by the SCTP or the CMT flow corresponds to 30ms, approximating the US coast-to-coast propagation delay [Shakkottai 2004].

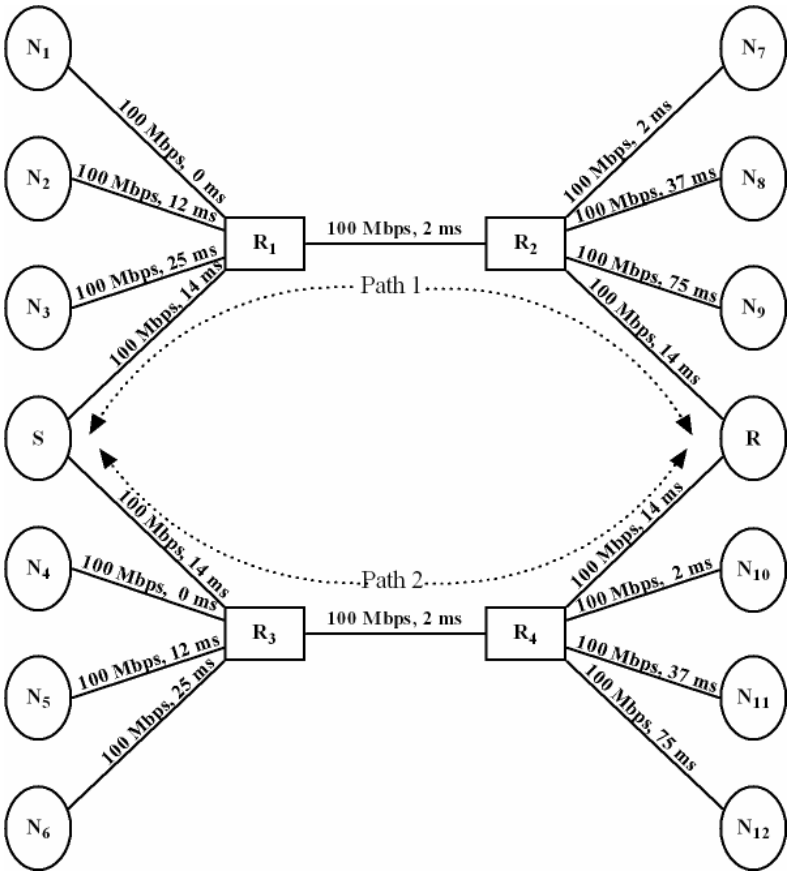


Figure 3.6: Topology for CMT Experiments (Topology 2)

In both topologies, the bottleneck links experience bi-directional cross-traffic; the cross-traffic load is similar on both forward and reverse directions. In topology 1, the cross-traffic load varies from low to heavy. For CMT evaluations using

topology 2, the bottlenecks experience asymmetric path loads; path 1 cross-traffic load varies from low to heavy, while path 2 experiences low load.

The SCTP or CMT flow initiates an unordered data transfer ~18-20 seconds after the simulation begins such that, all data received out-of-order at R is deliverable, and thus, non-renegable. Trace collection begins after a 20 second warm-up period from the start of SCTP or CMT traffic, and ends when the simulation completes after 70 seconds. The CMT sender uses the recommended RTX-SSTHRESH retransmission policy, i.e., retransmissions are sent on the path with highest ssthresh [Iyengar 2006].

3.4.2 Metric: Efficient Retransmission Queue Utilization

In transfers using SACKs, the RtxQ consists of two kinds of data (Figure 3.2): (i) *necessary* data – data that is either "in flight" and has not yet reached receiver's transport layer, or data that has been received but is renegable by the transport receiver, and (ii) *unnecessary* data – data that is received out of order and is non-renegable. *The RtxQ is most efficiently utilized when all data in the RtxQ are necessary*. As the fraction of unnecessary data increases, the RtxQ is less efficiently utilized.

The transport sender modifies the RtxQ as and when SACKs or NR-SACKs arrive. The RtxQ size varies during the course of a file transfer, but can never exceed the send buffer size. For time duration t_i in the transfer, let,

r_i = size of retransmission queue, and

k_i = amount of necessary data in the RtxQ.

During t_i, only $k_i \div r_i$ of the RtxQ is efficiently utilized, and the efficiency changes whenever k_i or r_i changes.

Let $\frac{k_0}{r_0}, \frac{k_1}{r_1}, \ldots \frac{k_n}{r_n}$ be the efficient RtxQ utilization values during time durations $t_0, t_1, \ldots t_n \left(\sum t_i = T\right)$, respectively. The time weighted efficient RtxQ utilization averaged over *T* is calculated as $RtxQ\_Util = \left(\sum t_i \times \frac{k_i}{r_i} \right) \div T$. To measure RtxQ utilization, the ns-2 SCTP (or CMT) sender tracks k_i, r_i, and t_i until association shutdown. Let,

W = time when trace collection begins after the initial warm-up time, and

E = simulation end time.

In the following discussions, the time weighted efficient RtxQ utilization averaged over the *entire trace collection time, i.e., T = (E – W),* is referred to as *RtxQ_Util*.

In an unordered transfer using NR-SACKs, all out-of-order data will be nr-gap-acked and the RtxQ should contain only necessary data. Therefore, we expect an SCTP or CMT flow using NR-SACKs to most efficiently utilize the RtxQ (*RtxQ_Util* = 1) under all circumstances.

3.4.3 Retransmission Queue Utilization during Loss Recovery

Typically, in SCTP transfers, data is always received in-order during no losses, unless the intermediate routers reorder packets. Consequently, during no losses, SCTP flows employing either SACKs or NR-SACKs utilize the RtxQ most efficiently, and the corresponding *RtxQ_Util* values equal unity. The two acknowledgment mechanisms differ in RtxQ usage only when data is received out-of-order, which ensues when an SCTP flow suffers packet losses. Specifically, in SCTP, *the duration of NR-SACKs' impact on the RtxQ is limited to loss recovery periods*. To evaluate the impact of the two ack schemes during loss recovery periods, the ns-2 SCTP sender timestamps

every entry/exit to/from loss recovery. Since none of the routers reorder packets in our simulations, the SCTP sender uses the following naive rule – the sender enters loss recovery on the receipt of SACKs (or NR-SACKs) with at least one gap-ack block, and exits loss recovery on the receipt of SACKs (or NR-SACKs) with a new cum-ack and zero gap-acks. We found that this simple rule resulted in a good approximation of the actual loss recovery periods.

Let $\frac{k_0}{r_0}, \frac{k_1}{r_1}, \ldots \frac{k_m}{r_m}$ be the efficient RtxQ utilization values during the loss recovery periods $l_0, l_1, \ldots l_m \left(\sum l_i = L\right)$, respectively. The time weighted efficient RtxQ utilization averaged over *only the loss recovery durations of trace collection (L)* is refereed to as *RtxQ_Util_L,* and is calculated as $RtxQ\_Util\_L = \left(\sum l_i \times \frac{k_i}{r_i}\right) \div L$.

An SCTP sender tracked both *RtxQ_Util* and *RtxQ_Util_L.* Depending on the paths' bandwidth/delay characteristics, a CMT association experiences data reordering even under no loss conditions. Data transmitted on the shorter delay path will be received out-of-order w.r.t. data transmitted on other path(s). Therefore, the naïve rule mentioned above cannot be employed to estimate entry/exit of CMT sender's loss recovery, and the CMT sender tracked only *RtxQ_Util*.

3.5 Results

For each type of sender (SCTP or CMT), different send buffer sizes imposing varying levels of memory constraints are considered: 32K, 64K and INF (unconstrained space) for SCTP, and 128K, 256K and INF for CMT. The results presented here are averaged over 30 runs, and plotted with 95% confidence intervals. In the following discussions, an SCTP flow using SACKs or NR-SACKs is referred to

as SCTP-SACKs and SCTP-NR-SACKs, respectively. Similarly, a CMT flow using SACKs or NR-SACKs is referred to as CMT-SACKs and CMT-NR-SACKs.

3.5.1 Retransmission Queue Utilization

As the end-to-end path gets more congested, SCTP-SACKs' *RtxQ_Util_L* remains fairly consistent ~0.5 (Figure 3.7), while the *RtxQ_Util* decreases (Figure 3.8).

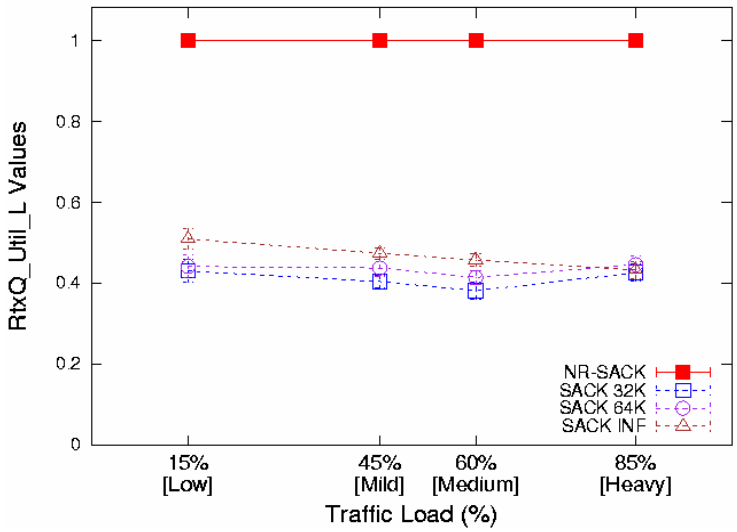


Figure 3.7: RtxQ Utilization during Loss Recovery in SCTP

The *RtxQ_Util_L* values indicate that irrespective of path loss rate, SCTP-SACKs efficiently utilize only ~50% of RtxQ during loss recovery; ~50% of RtxQ is wasted buffering unnecessary data. At lower congestion levels (lower cross-traffic), the frequency of loss events and the fraction of transfer time spent in loss recovery are smaller, resulting in negligible RtxQ wastage during the entire trace collection period (*RtxQ_Util*). As loss recoveries become more frequent, SCTP-SACKs' inefficient RtxQ utilization during loss recovery lowers the corresponding *RtxQ_Util* values. The simulation results show that, on average, SCTP-SACKs waste ~20% of the RtxQ during moderate congestion and ~30% during heavy congestion conditions. The

amount of wasted kernel memory increases as the number of transport connections increase, and can be significant at a server handling large numbers of concurrent connections, such as a web server.

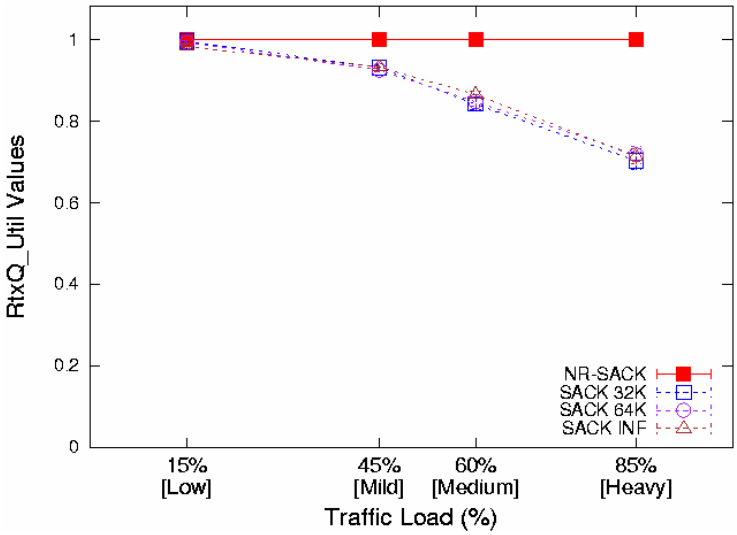


Figure 3.8: RtxQ Utilization in SCTP

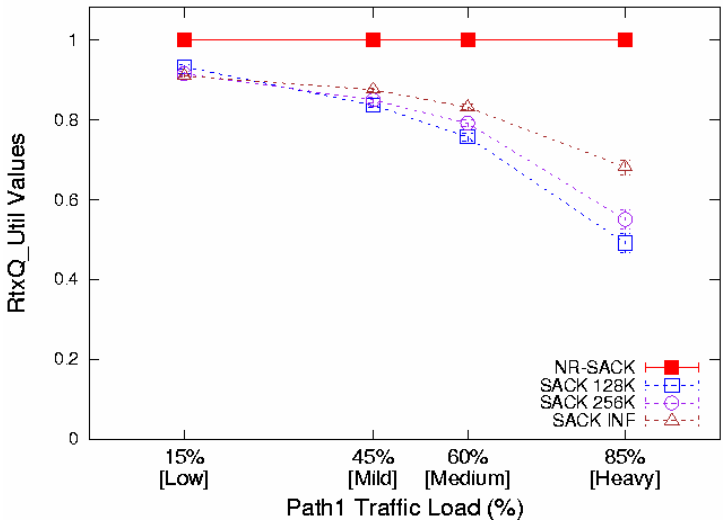


Figure 3.9: RtxQ Utilization in CMT

By definition of the *RtxQ_Util* metric, NR-SACKs are expected to utilize the RtxQ most efficiently, even during loss recovery periods (Section 3.4.2). The

simulation results confirm this hypothesis. Under all traffic loads, *RtxQ_Util* values for both SCTP-NR-SACKs and CMT-NR-SACKs (Figure 3.9) are unity.

In CMT evaluations, path 2 experiences low traffic load, while path 1's traffic load varies from low to heavy (Figure 3.6). Recall that a CMT sender transmits data concurrently on both paths. Asymmetric path congestion levels aggravate data reordering in CMT. As path 1 congestion level increases, TPDU losses on the higher congested path 1 cause data transmitted on the lower congested path 2 to arrive out-of-order at the receiver. CMT congestion control is designed such that losses on path 1 do not affect the cwnd/flightsize on path 2 [Iyengar 2006]. While losses on path 1 are being recovered, sender continues data transmission on path 2, increasing the amount of non-renegable out-of-order data in the RtxQ. As the paths become increasingly asymmetric in their congestion levels, the amount of non-renegable out-of-order data in the RtxQ increases, and brings down CMT-SACKs' *RtxQ_Util* (Figure 3.9).

Increasing the send buffer/RtxQ space improves SCTP-SACKs' or CMT-SACKs' kernel memory (RtxQ) utilization only to a certain degree. In Figures 3.8 and 3.9, *RtxQ_Util* for the INF send buffer is essentially the upper bound on how efficient SCTP or CMT employing SACKs utilizes the RtxQ. Therefore, we conclude that *TPDU reordering results in inevitable RtxQ wastage in transfers using SACKs. The amount of wasted memory increases as TPDU reordering and loss recovery durations increase.* Also, smaller send buffer sizes further degrade *RtxQ_Util_L* and *RtxQ_Util* values in transfers using SACKs. This degradation is more pronounced in CMT (Figure 3.9). Further investigations reveal this effect to be due to send buffer blocking, discussed next.

3.5.2 Send Buffer Blocking in CMT

When the RtxQ grows to fill the entire send buffer, send buffer blocking ensues, preventing the application from writing new data into the transport layer (Section 3.2.2). In both SCTP and CMT, send buffer blocking increases as the send buffer is more constrained (decreases). In addition, CMT employs multiple paths for data transfer, increasing a sender's total flightsize in comparison to SCTP. Therefore, we hypothesized that CMT would suffer more send buffer blocking than SCTP (Section 3.2.3). Indeed, in the simulations, CMT suffered significant send buffer blocking even for 128K and 256K send buffer sizes. In this section, we focus on the effects of send buffer blocking in CMT.

CMT using either acknowledgment scheme suffers from send buffer blocking for 128K and 256K buffer sizes. In CMT-SACKs, send buffer blocking continues until cum-ack point moves forward, i.e., until loss recovery ends. As path 1 congestion level increases, timeout recoveries become more frequent, causing longer loss recovery durations. Therefore, as congestion increases, the CMT-SACKs sender is blocked for longer periods of transfer time. On the other hand, send buffer blocking in CMT-NR-SACKs is unaffected by the congestion level on path 1. As and when NR-SACKs arrive (on path 2), the CMT-NR-SACK sender removes nr-gap-acked data from the RtxQ, allowing more data transmission. CMT-SACKs' longer send buffer blocking durations adversely impact performance as discussed below.

3.5.2.1 Ineffective Use of Send Buffer Space

Send buffer blocking limits RtxQ growth and reduces throughput. The impact on throughput is minimized when the available send buffer space is utilized *as much as possible.*

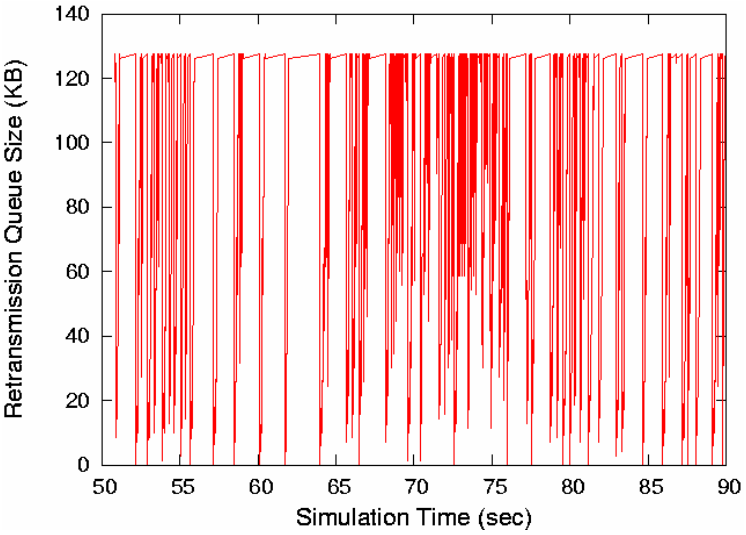


Figure 3.10: RtxQ Evolution in CMT-SACKs

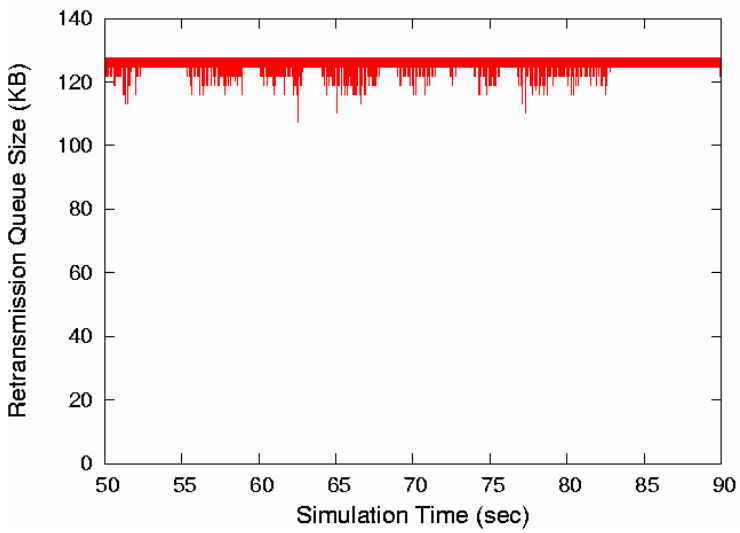


Figure 3.11: RtxQ Evolution in CMT-NR-SACKs

Figures 3.10 and 3.11 illustrate CMT sender's RtxQ evolution over 40 seconds of a transfer using SACKs and NR-SACKs, respectively. The figures show that both CMT-SACKs and CMT-NR-SACKs suffer from send buffer blocking – the maximum RtxQ size in the figures corresponds to 100% of send buffer (128K). However, the RtxQ evolution in CMT-SACKs (Figure 3.10) exhibits more variance –

reaches the maximum and drops to 0 multiple times, while CMT-NR-SACKs' RtxQ size is closer to 128K most of the time (Figure 3.11).

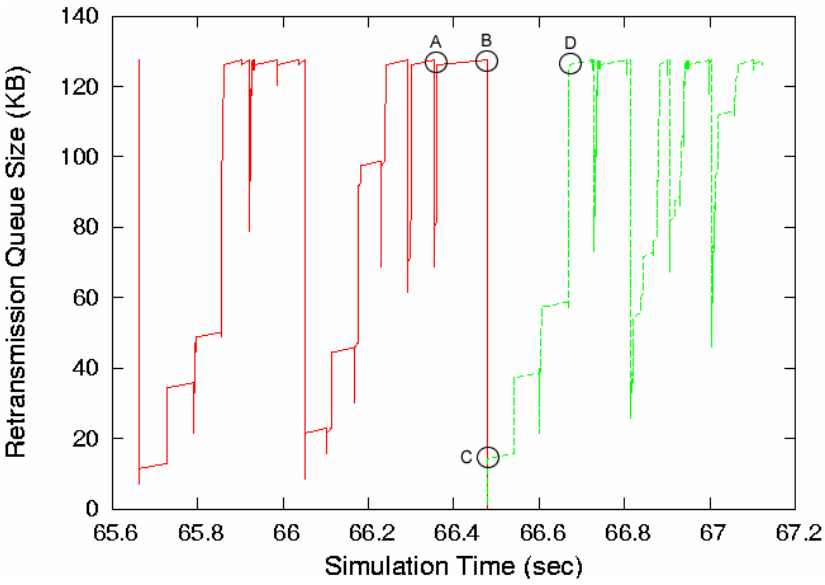


Figure 3.12: RtxQ Evolution in CMT-SACKs (~1.5 sec)

Figure 3.12 is a zoom of CMT-SACKs' RtxQ evolution over an arbitrary 1.5 second period. At point A (time 66.36sec), RtxQ size hits the maximum, and the sender is blocked from transmitting any more data. Subsequent SACKs reduce the amount of outstanding data, but send buffer blocking prevents the sender from clocking out new data. At time 66.42sec, path 1's retransmission timer expires; the sender detects loss, and retransmits TSN 20369 on path 2. At time 66.48sec (point B), sender receives a SACK with a new cum-ack (TSN=20457) and completely clears RtxQ contents, ending the current instance of send buffer blocking. The sender immediately transmits new data on both paths, and the RtxQ evolution after the new cum-ack (TSN=20457) is shown by the (green) dashed line. The cwnd on path 1 allows transmission of 2 MTU sized TPDUs (TSNs 20458 and 20459). The cwnd on path 2 is 127162 bytes, but the *Maxburst* parameter [RFC4960] limits the sender to transmit

only 4 MTU sized TPDUs – TSNs 20460-20463. Once the sender transmits data on both paths, RtxQ size increases to ~8.6K, shown by point C. Subsequent SACKs allow more data transmission and at point D the sender's RtxQ reaches the maximum causing the next instance of send buffer blocking.

Though CMT-NR-SACKs also incurs send buffer blocking (Figure 3.11), nr-gap-acks free up RtxQ space allowing the sender to steadily clock out more data. *A constrained send buffer is better utilized, and the transmission is less bursty with NR-SACKs than SACKs.* The improved send buffer use contributes to throughput improvements (discussed later).

3.5.2.2 Efficient Retransmission Queue Utilization

In Figure 3.9, CMT-SACKs' *RtxQ_Util* worsens as send buffer blocking increases (send buffer size decreases). As discussed earlier, in CMT-SACKs, send buffer blocking prevents new data transmission until loss recovery ends. Lack of new data transmission resulted in fewer and sometimes insufficient acks to trigger fast retransmits. Consequently, blocked CMT-SACKs experienced more timeout recoveries (RTOs) at heavy traffic loads than non-blocked CMT-SACKs (Figure 3.13). As the send buffer is more constrained, the average number of RTOs increase, and the fraction of transfer time spent in loss recovery increases. Longer loss recovery durations increase the duration of inefficient RtxQ utilization, and bring down blocked CMT-SACKs' *RtxQ_Util* values compared to non-blocked (INF) CMT-SACKs' *RtxQ_Util*.

On the other hand, CMT-NR-SACKs steadily clock out data, and do not incur excessive RTOs during send buffer blocking. CMT-NR-SACKs' mean number of RTOs for 128K and 256K buffer sizes are similar to the INF case (Figure 3.13). To summarize, *send buffer blocking worsens CMT-SACKs' RtxQ utilization. Blocked*

CMT-SACKs' inefficient send buffer usage increases the number of timeout recoveries, and degrades throughput when compared to CMT-NR-SACKs.

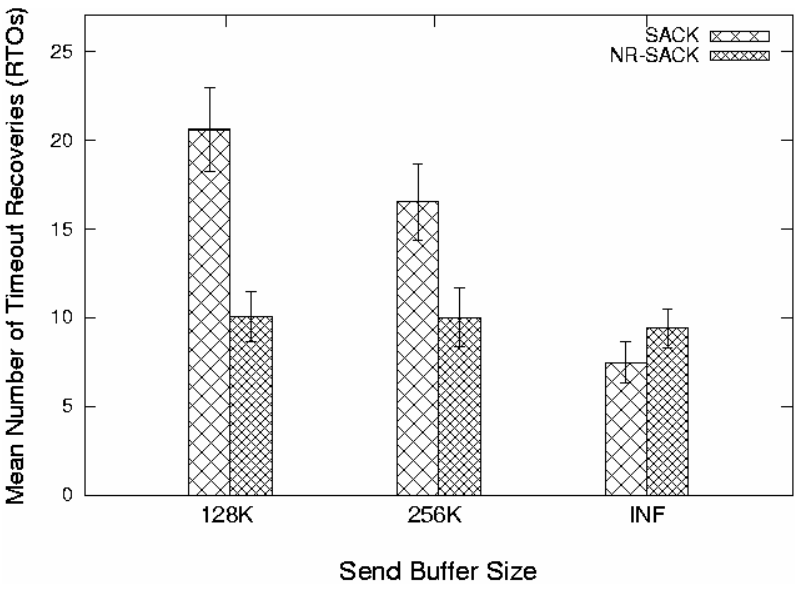


Figure 3.13: Mean Number of RTOs during Heavy Cross-traffic in CMT

3.5.2.3 Throughput

When the send buffer never limits RtxQ growth (INF send buffer size), both CMT-SACKs and CMT-NR-SACKs do not experience send buffer blocking, and perform similarly (Figure 3.14). However, CMT-SACKs achieve the same throughput as CMT-NR-SACKs at the cost of larger RtxQ sizes.

Using terminology defined in Section 3.4.2, the average RtxQ size, *RtxQ* over the entire trace collection period (*T*) is calculated as, $RtxQ = \left(\sum t_i \times r_i\right) \div T$. Figure 3.15 plots CMT-SACKs vs. CMT-NR-SACKs *RtxQ* for the INF case. As path 1 cross-traffic load increases, the bandwidth available for the CMT flow decreases, and CMT-NR-SACKs' *RtxQ* decreases (Figure 3.15).

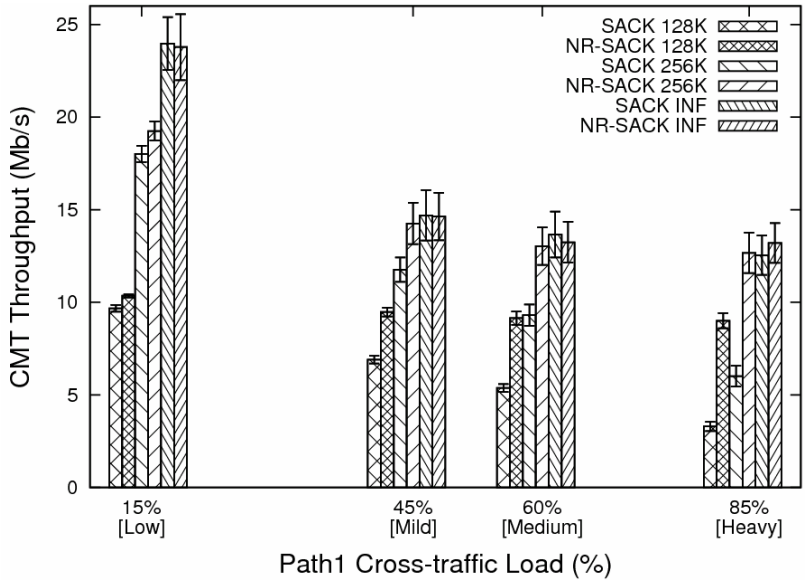


Figure 3.14: CMT-SACKs vs. CMT-NR-SACKs Throughput

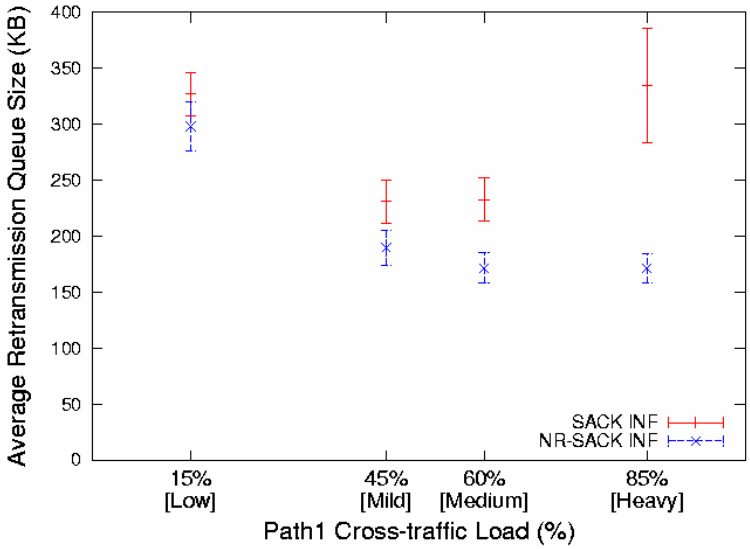


Figure 3.15: CMT-SACKs vs. CMT-NR-SACKs Average RtxQ Size

Similarly, CMT-SACKs' *RtxQ* decreases as traffic load increases from low to mild. However, a different factor dominates and increases CMT-SACKs' *RtxQ* during medium and heavy traffic conditions. Note that RtxQ growth is never constrained in the INF case, enabling the CMT sender to transmit as much data as possible on path 2

while recovering from losses on path 1. At medium and heavy cross-traffic loads, loss recovery durations increase due to increased timeout recoveries, and the CMT-SACKs sender transmits more data on path 2 compared to mild traffic conditions. This factor increases CMT-SACKs' *RtxQ* during medium and heavy traffic conditions.

Going back to Figure 3.14, when the send buffer size limits RtxQ growth, CMT-NR-SACKs' efficient RtxQ utilization enables CMT-NR-SACKs to perform better than CMT-SACKs. *The throughput improvements in CMT-NR-SACKs increase as conditions that aggravate send buffer blocking increases.* I.e., NR-SACKs improve throughput more as send buffer becomes more constrained and/or when the paths become more asymmetric in the congestion levels. Alternately, *CMT-NR-SACKs achieve similar throughput as CMT-SACKs using smaller send buffer sizes.* For example, during mild, medium and heavy path 1 cross-traffic load, CMT-NR-SACKs with 128K send buffer performs similar or better than CMT-SACKs with 256K send buffer. Also, CMT-NR-SACKs with 256K send buffer performs similar to CMT-SACKs with larger (unconstrained) send buffer.

3.6 Conclusion, Ongoing and Future Work

This work investigated the effects of existing transport layer SACK mechanism when data received out-of-order is non-renegable. We conclude that SACKs cause inevitable sender memory wastage, which worsens as data reordering and loss recovery durations increase. We proposed a new ack mechanism, Non-Renegable Selective Acknowledgments (NR-SACKs) for SCTP, which provides the transport receiver with the option to convey non-renegable information on some or all out-of-order data. The concept of NR-SACKs is applicable to any reliable transport employing SACKs and/or provides out-of-order data delivery.

Note that a *transfer employing NR-SACKs never performs worse than a transfer using SACKs.* When out-of-order data is non-renegable, NR-SACKs perform better than SACKs. Simulations confirmed that in both SCTP and CMT, NR-SACKs utilize send buffer and RtxQ space most efficiently. Send buffer blocking in CMT with SACKs adversely impacts end-to-end performance, while efficient send buffer use in CMT with NR-SACKs alleviates send buffer blocking. Therefore, NR-SACKs not only reduce sender's memory requirements, but also improve throughput in CMT. The only negative with NR-SACKs is the added complexity of implementation, and the extra overhead to generate and process NR-SACKs. We argue these negatives are negligible.

3.6.1 IETF Internet Draft

We plan to standardize the design and processing specifics of the SCTP NR-SACK chunk, and have proposed the same as an IETF Internet Draft in the transport area working group (TSVWG) [Natarajan 2008a]. The details of the NR-SACK chunk and the simulation results were presented at the 71st and 72nd IETF meetings. Based on the positive feedback from the TSVWG members, the proposal has been modified to be an *experimental* item, and is currently being implemented in the reference SCTP implementation on FreeBSD. As future work, we also plan on conducting empirical studies to gather information on how often reneging occurs, if any, in practice.

3.6.2 NR-SACKs Implementation in FreeBSD

Ertugrul Yilmaz is heading the on-going effort to implement NR-SACKs in the FreeBSD SCTP stack. This effort involves defining the NR-SACK chunk structure, modifying the sender and receiver code to generate and process NR-SACKs,

respectively, and defining a test suite to debug the NR-SACKs implementation. In the future, we plan to draw on the FreeBSD implementation to compare SACKs vs. NR-SACKs performance for both SCTP and CMT.

Chapter 4

CMT PERFORMANCE DURING FAILURE

This chapter discusses the third problem – Concurrent Multipath Transfer (CMT) performance during path failures. Section 4.1 motivates this research by discussing the commonness of link failures in the Internet. Section 4.2 overviews CMT's failure detection process, and discusses how CMT's throughput degrades during path failures. Section 4.3 details a proposed solution to the problem – CMT with the "potentially-failed" destination state (CMT-PF). Sections 4.4 and 4.5 present simulation based evaluations of CMT vs. CMT-PF during failure and congestion, respectively. Finally, Section 4.6 concludes and presents ongoing, future and related work.

4.1 Motivation

As discussed in Chapter 1, SCTP natively supports transport layer multihoming for fault-tolerance purposes. Concurrent Multipath Transfer (CMT) [Iyengar 2006] is an experimental SCTP extension that assumes multiple independent paths between multihomed end points, and exploits the independent paths for simultaneous transfer of new data (see Chapter 1).

Path failures arise when a router or a link connecting two routers fails due to planned maintenance activities or unplanned accidental reasons such as hardware malfunction or software error. Ideally, the routing system detects unplanned link failures, and reconfigures the routing tables to avoid routing traffic via the failed link.

Using data from an ISP's routing logs, [Markopoulou 2004] observes that link failures are part of everyday operation. Around 80% of the failures are unplanned, and the time-to-repair for any particular failure can be on the order of hours. Existing research also highlights problems with Internet backbone routing that result in long route convergence times. [Labovitz 2000] shows that Internet's interdomain routers may take as long as tens of minutes to reconstruct new paths after a failure. During these delayed convergences, end-to-end Internet paths experience intermittent loss of connectivity in addition to increased packet loss, latency, and reordering.

Using probes, [Paxson 1997] and [Zhang 2000] find that "significant routing pathologies" prevent selected pairs of hosts from communicating about 1.5% to 3.3% of the time. Importantly, the authors also find that this trend has not improved with time. Reference [Labovitz 1999] examines routing table logs of Internet backbones to find that 10% of all considered routes were available less than 95% of the time, and more than 65% of all routes were available less than 99.99% of the time. The duration of these path outages were heavy-tailed and about 40% of path outages took more than 30 minutes to repair. In [Chandra 2001], the authors use probes to confirm that failure durations are heavy-tailed, and report that 5% of detected failures last more than 2.75 hours, and as long as 27.75 hours. The pervasiveness of path failures in practice motivates us to study their impact on CMT.

4.2 CMT Performance during Path Failure

This section gives an overview of CMT's failure detection procedure and describes how CMT's performance suffers during path failures.

4.2.1 Failure Detection in CMT

Since CMT is an extension to SCTP, CMT retains SCTP's failure detection process. A CMT sender uses a tunable failure detection threshold called *Path.Max.Retrans (PMR)* [RFC4960]. As shown in the finite state machine of Figure 4.1, a destination is in one of the two states – *active* or *failed* (inactive). A destination is active as long as acks come back for data or heartbeats (probes) sent to that destination. When a sender experiences more than PMR consecutive timeouts while trying to reach a specific active destination, that destination is marked as failed. Only heartbeats (i.e., no data) are sent to a failed destination. A failed destination returns to the active state when the sender receives a heartbeat ack. RFC4960 proposes a default PMR value of 5, which translates to at least 63 seconds (6 consecutive timeouts) for failure detection.

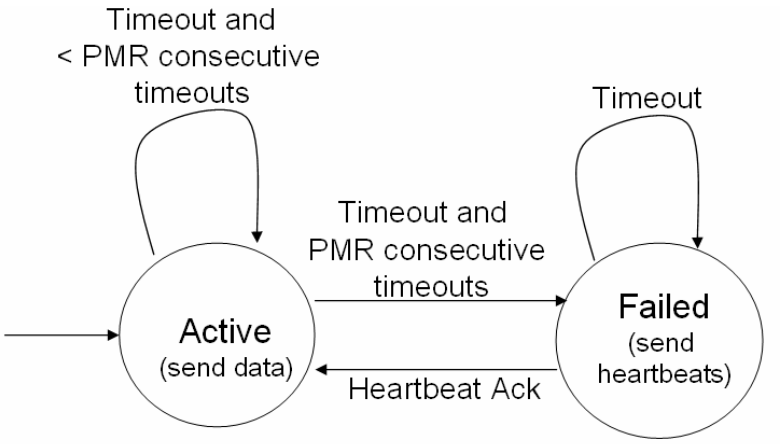


Figure 4.1: Failure Detection in CMT

4.2.2 Receive Buffer Blocking in CMT

[Iyengar 2005] explores the "rbuf blocking" problem in CMT, where TPDU losses throttle data transmission once the CMT receiver's buffer (rbuf) is filled

with out-of-order data. Even though the cwnd would allow new data to be transmitted, rbuf blocking (i.e., flow control) stalls the sender, causing throughput degradation.

Rbuf blocking problem cannot be eliminated in CMT [Iyengar 2005]. To reduce rbuf blocking's negative impact during congestion, [Iyengar 2005] proposes different retransmission policies that use heuristics for faster loss recovery. These policies consider different path properties such as loss rate and delay, and try to reduce rbuf blocking by sending retransmissions on a path with lower loss or delay. In practice, the loss rate of a path can only be estimated, so [Iyengar 2005] proposed the RTX_SSTHRESH policy, where retransmissions are sent on the path with the largest slow-start threshold. Since RTX_SSTHRESH outperformed other retransmission policies during congestion, [Iyengar 2005] recommended the RTX_SSTHRESH policy for CMT. However, [Iyengar 2005] did not consider CMT performance during path failures. As we shall show, CMT with the RTX_SSTHRESH policy suffers from significant rbuf blocking during path failures.

4.2.3 Rbuf Blocking during Path Failure

CMT's failure-induced rbuf blocking problem is modeled via the timeline shown in Figure 4.2. The CMT sender (A) has two interfaces – A_1 and A_2, and transmits data to a receiver (B) with two interfaces – B_1 and B_2. All four addresses are bound in the CMT association such that the sender employs the 2 independent paths – path 1 and path 2, for data transmission. C_i and O_i denote the cwnd in number of MTUs, and the number of outstanding TPDUs, respectively, on path *i*. The initial cwnd for each path=2 MTUs. The data transfer example also assumes the following for easier illustration: (a) each SCTP PDU contains a single data chunk resulting in a one-to-one

correspondence between an SCTP PDU and TSN, and (b) each SCTP PDU is MTU-sized.

In Figure 4.2, a SACK labeled <S*a*, *b-c*; R*d*> acknowledges all TSNs upto and including the cumulative TSN value of *a*, in-order arrival of TSNs *b* through *c* (missing report for TSNs *a+1* through *b-1*), and an advertised receiver window[1] capable of buffering *d* more TSNs. On receiving a SACK, sender A subtracts the number of outstanding TSNs from the advertised receiver window, and calculates the amount of new data that can be sent without overflowing the receive buffer. The transport layer receive buffer for this example can hold a maximum of 5 TSNs, and its contents are listed after the reception of every TSN.

In the example, both forward and reverse paths between A_1 and B_1 fail just after TSN 2 enters the network. Hence, TSN 2 and the SACK for TSN 1 are presumed lost. TSNs 3 and 4 arrive out of order, each trigger a SACK, and are stored in the receive buffer. The CMT sender uses the *Cwnd Update for CMT (CUC)* algorithm [Iyengar 2006] to decouple a path's cwnd evolution and data ordering. On receiving the SACK triggered by TSN 3, the sender uses CUC to increment C_2 to 3, and decrement O_1 and O_2 to 1. The available receive buffer space for new data is calculated as advertised receive window (4) – total outstanding TSNs in the association (2). This available receive buffer space allows the sender to transmit two TSNs, 5 and 6, on path

---

[1] Advertised receiver window (*a_rwnd*) has different connotations in TCP and SCTP. TCP's *a_rwnd* denotes the available memory in rbuf, starting from the left edge of received sequence space [RFC793]. SCTP's *a_rwnd* denotes the available memory after considering all TPDUs not yet delivered to the application layer, including the out-of-order TPDUs [RFC4960].

2. On path 1, even though 1 MTU worth of new data can be transmitted ($C_1 > O_1$), rbuf blocking, i.e., flow control stalls data transmission.

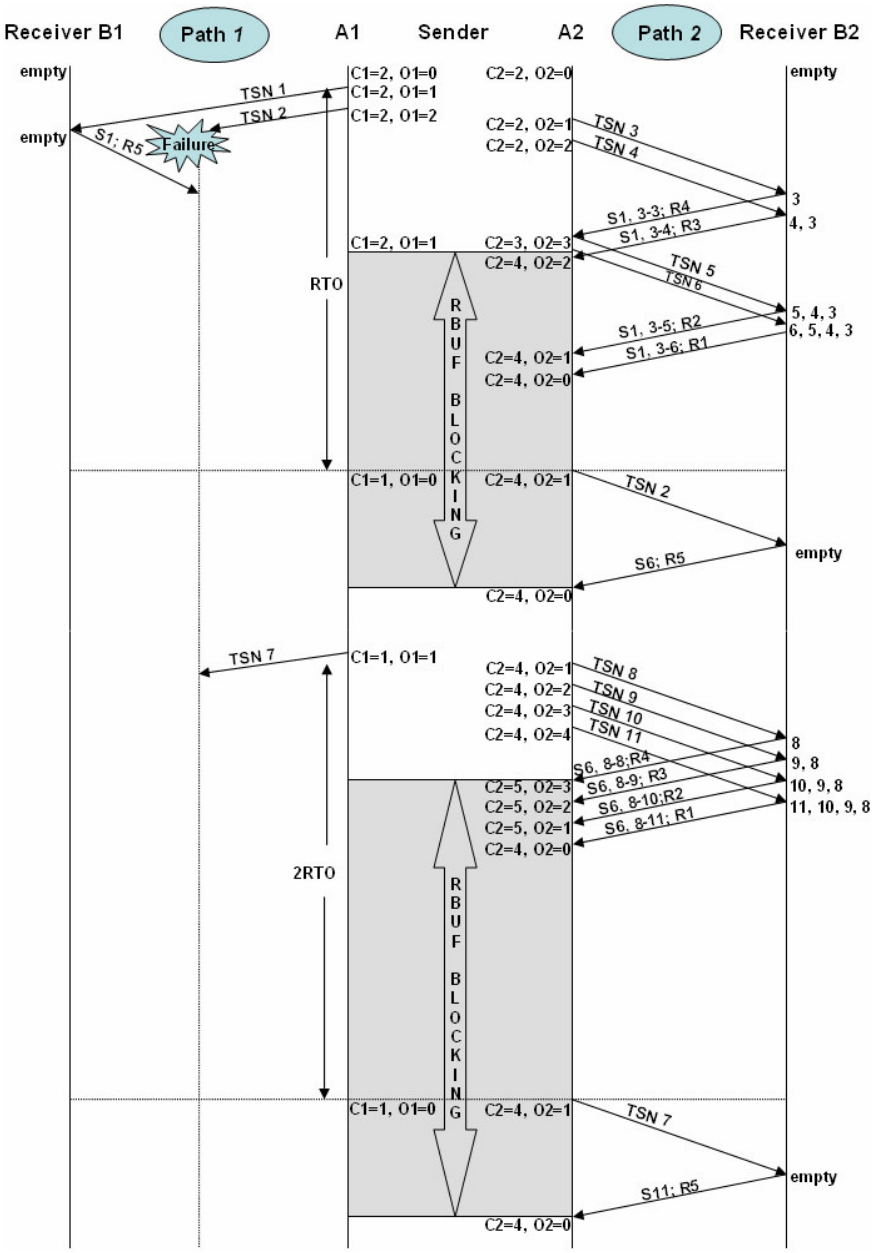


Figure 4.2: Rbuf Blocking in CMT during Failure

On receiving the SACK triggered by TSN 4, the sender increases C_2 to 4, and decreases O_2 to 2. Lack of receive buffer space (advertised receive window – total

number of outstanding TSNs) continues to prevent transmission of new data on path 2. Since $O_2 < C_2$, the SACKs triggered by TSNs 5 and 6 do not increment C_2 [RFC4960] (discussed later). But these SACKs decrement O_2. Even though $O_2 < C_2$, rbuf blocking stalls data transmission on path 2.

Path 1's retransmission timer expires and the sender detects the loss of TSN 2. Note that this timeout is the first of the 6 (PMR = 5) consecutive timeouts needed to detect path 1 failure. After this timeout, C_1 is set to 1, O_1 is set to 0, and path 1's RTO value is doubled [RFC4960]. The CMT sender employs the RTX_SSTHRESH policy and retransmits TSN 2 on path 2. Data cannot be transmitted on path 1 due to rbuf blocking.

On receiving TSN 2, the receiver delivers data from TSNs 2-6 to the application. The corresponding SACK advertises a receive window of 5 TSNs, and concludes the current rbuf blocking instance. The sender now transmits TSN 7 on path 1, and TSNs 8-11 on path 2. Due to path 1 failure, TSN 7 is lost, and TSNs 8-11 are received out-of-order and stored in the receiver's buffer. The SACK triggered by TSN 8 increments C_2 to 5 and decrements O_2 to 3. The available receive buffer space for new data=0, triggering another instance of rbuf blocking, which stalls data transmission until TSN 7 is successfully retransmitted. Note that the loss of TSN 7 can be recovered only after a timeout on path 1, and due to the exponential backoff algorithm, path 1's current RTO value is twice the previous value.

To generalize, sender A transmits new data on path 1 until (PMR + 1) number of consecutive timeouts mark path 1 as failed. During failure detection, data transmitted on non-failed path(s) arrive out-of-order, resulting in consecutive rbuf blocking instances. Each rbuf blocking instance concludes when the sender retransmits

lost TPDUs after an RTO. The length of an rbuf blocking instance is therefore proportional to the failed path's RTO. Also, each rbuf blocking instance is exponentially longer than the previous instance due to the exponential backoff of RTO values.

Rbuf blocking results in the following side-effects that further degrade CMT's throughput:

Preventing congestion window growth: Note that rbuf blocking prevents the sender from fully utilizing the cwnd. When the amount of outstanding data is less than the cwnd, RFC4960 prevents the sender from increasing the cwnd for future SACKs. For example, in Figure 4.2, when the sender receives the SACKs for TSNs 5, 6, 9-11, arrive, the sender cannot increment C_2.

Reducing congestion window: To reduce burstiness in data transmission, an SCTP sender employs a congestion window validation algorithm similar to [RFC2861]. During every transmission, the sender uses the *MaxBurst* parameter (recommended value of 4) as follows:

*If ((outstanding + MaxBurst * MTU) < Cwnd)*

*Cwnd = outstanding + MaxBurst * MTU*

This algorithm reduces the cwnd during idle periods so that at the next sending opportunity, the sender cannot transmit more than (*MaxBurst* * MTU) bytes of data. During rbuf blocking, the amount of outstanding data can become smaller than the cwnd. In such cases, the above rule is triggered and further reduces the cwnd. In Figure 4.2, when the SACK triggered by TSN 11 arrives at the sender, O_2 decrements to 0. The window validation algorithm causes C_2 to be reduced to 4 (O_2 (0) + *MaxBurst* (4)).

4.3 CMT with Potentially Failed Destination State

[Caro 2005] recommends lowering the value of PMR for SCTP flows in Internet-like environments. Correspondingly, lowering the PMR for CMT flows reduces the number of rbuf blocking episodes during failure detection. However, lowering the PMR is an incomplete solution to the problem since a CMT flow is rbuf blocked for any PMR > 0 (discussed later). Also, a tradeoff exists on deciding the value of PMR – a lower value reduces rbuf blocking but increases the chances of spurious failure detection, whereas a higher PMR increases rbuf blocking and reduces spurious failure detection in a wide range of environments.

4.3.1 Details of CMT-PF

To mitigate the recurring instances of rbuf blocking during path failures, our proposed solution introduces a new destination state called "potentially-failed" in the FSM of Figure 4.1. The new FSM, shown in Figure 4.3, is based on the rationale that loss detected by a timeout implies either severe congestion or failure en route. After a single timeout on a path, a sender is unsure, and marks the corresponding destination as "potentially-failed" (PF). A PF destination is not used for data transmission or retransmission. CMT's retransmission policies are augmented to include the PF state. CMT with the new set of retransmission policies is called CMT-PF [Natarajan 2006b]. Details of CMT-PF are:

- If a TPDU loss is detected by RFC4960's threshold number of missing reports, one of CMT's current retransmission policies, such as RTX_SSTHREH, is used to select an active destination for "fast" retransmissions.

- If a TPDU loss is detected by a timeout, the corresponding destination transitions to the PF state (Figure 4.3). The sender does not transmit data to a PF destination. However, when all destinations are in the PF state, the sender transmits data to the destination with the least number of consecutive timeouts. In case of tie, data is sent to the last active destination. This exception ensures that CMT-PF does not perform worse than CMT when all paths have potentially failed (discussed further in Section 4.6).

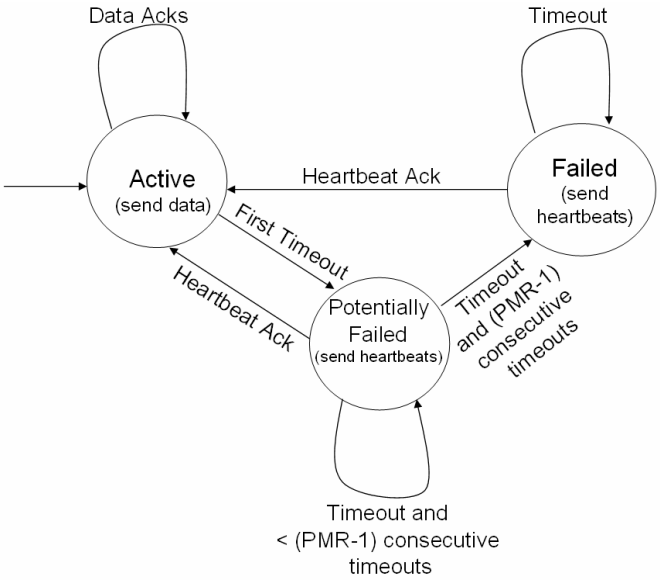


Figure 4.3: CMT-PF Failure Detection (PMR > 0)

- Heartbeats are sent to PF destination(s) with an exponential backoff of RTO after every timeout until either (i) a heartbeat ack transitions the destination back to the active state, or (ii) an additional PMR consecutive timeouts confirm the path failure, upon which the destination transitions to the failed state, and heartbeats are sent with a lower frequency as described in RFC4960.

- Once a heartbeat ack indicates a PF destination is alive, the destination's cwnd is set to either 1 MTU (CMT-PF1), or 2 MTUs (CMT-PF2), and the sender follows the slow start algorithm to transmit data to this destination. Detailed analysis on the cwnd evolution of CMT-PF1 vs. CMT-PF2 can be found in Section 4.6.
- Acks for retransmissions do not transition a PF destination to the active state, since a sender cannot determine whether the ack was for the original transmission or the retransmission(s).

4.3.2 CMT-PF Data Transfer during Failure

Figure 4.4 depicts an analogous CMT-PF timeline for the scenario described in Figure 4.2. All events are identical between the two figures up to the first timeout on path 1. After this timeout, the CMT-PF sender transitions path 1 to the PF state, transmits a heartbeat on path 1, and retransmits TSN 2 on path 2. The heartbeat loss on the failed path (path 1) is detected on the next timeout. This timeout is the second of (PMR + 1) consecutive timeouts required to detect path 1 failure. Meanwhile, receiver buffer space is released once the retransmitted TSN 2 is received on path 2. From this point onwards, data is transmitted only on path 2, without further rbuf blocking.

4.4 CMT vs. CMT-PF Evaluations during Failure

CMT-PF was implemented in the University of Delaware's SCTP/CMT module for the ns-2 network simulator [NS-2, Ekiz 2007]. This section discusses the performance of CMT vs. CMT-PF during permanent and short-term failure scenarios.

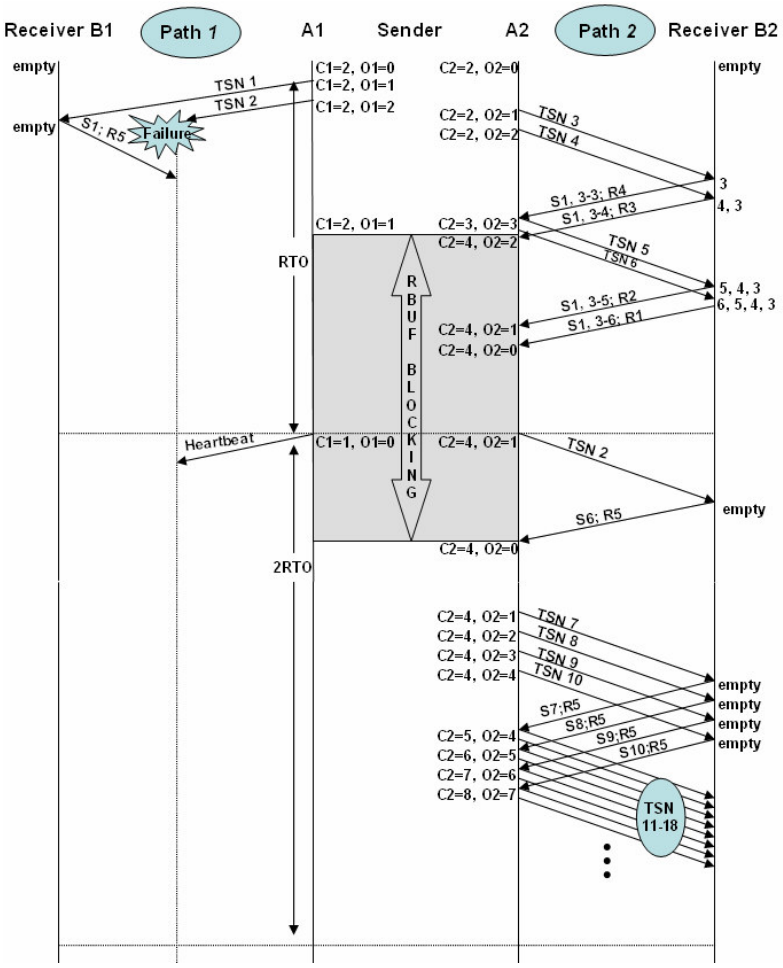


Figure 4.4: CMT-PF Reduces Rbuf Blocking during Failure

In the simulation topology (Figure 4.5), the multihomed sender, A, has two independent paths to the multihomed receiver, B. The edge links between A (or B) to the routers represent last-hop link characteristics. The end-to-end one-way delay is 45ms on both paths, representing typical coast-to-coast delays experienced by significant fraction of the flows in the Internet [Shakkottai 2004]. We note that the final conclusions regarding CMT vs. CMT-PF are independent of the actual bandwidth and delay configurations used in the topology, as long as these configurations are similar on both paths.

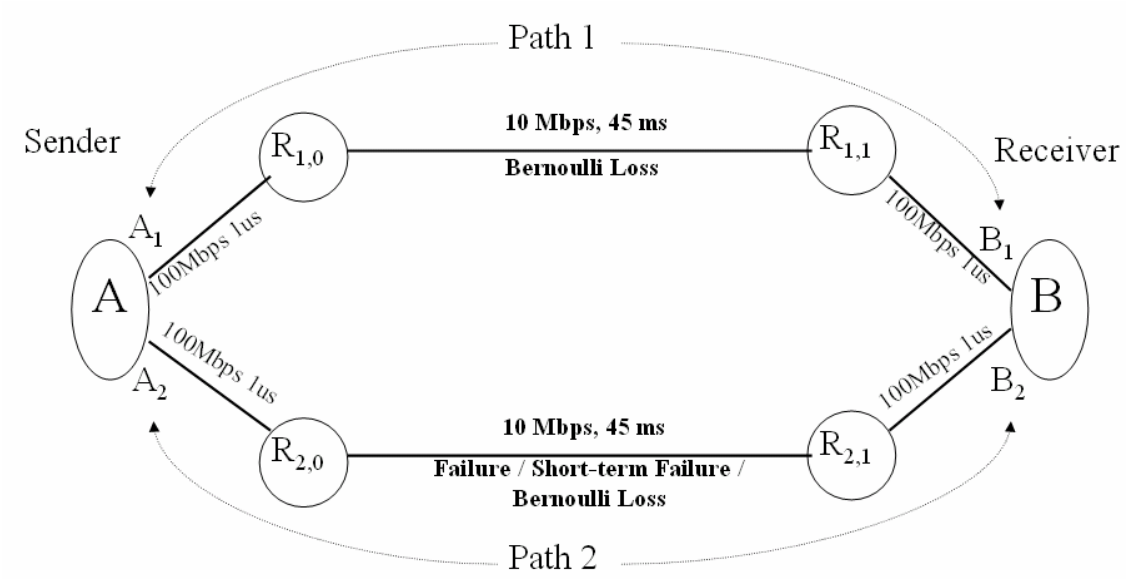


Figure 4.5: Topology for Failure Experiments

The sender A transfers an 8MB file to receiver B using both path 1 and path 2. Path 2 fails during the file transfer; this failure is simulated by bringing down the bidirectional link between routers R_{20} and R_{21}. Unless stated otherwise, the PMR=5, rbuf=64KB, and both paths experience Bernoulli losses with low loss rate (1%). We acknowledge that the Bernoulli loss model is less realistic than the nature of losses observed in the Internet. Since evaluations in this Section assume failure scenarios and rare loss events (1% or no loss), we expect the final conclusions between CMT vs. CMT-PF to remain similar even with a more realistic loss model

4.4.1 Evaluations during Permanent Failure

In the following experiments, path 2 fails permanently 5 seconds after the file transfer begins.

4.4.1.1 Evaluations during Single Permanent Failure (without Congestion)

This experiment highlights the essential differences between CMT and CMT-PF during a permanent path failure. To eliminate the influence of congestion-induced rbuf blocking, the simulation is setup such that the sender does not experience any congestion losses on either paths.

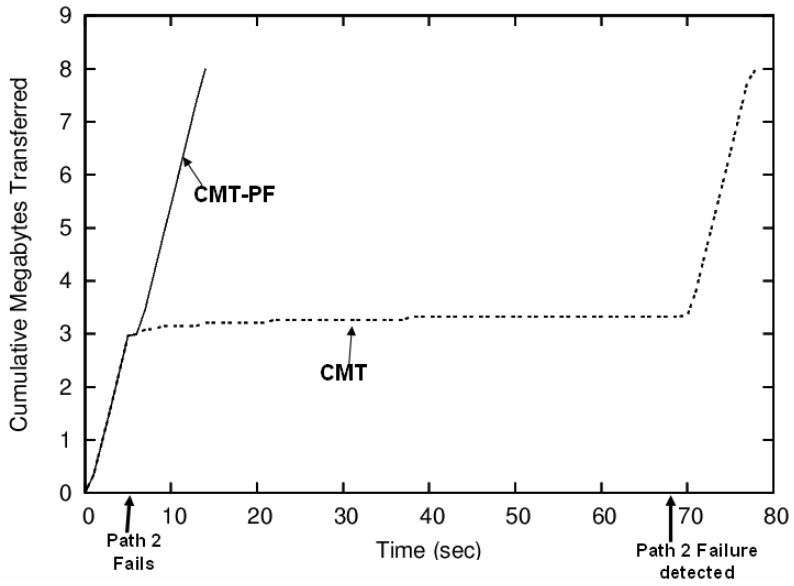


Figure 4.6: CMT vs. CMT-PF during Permanent Failure

The path 2 failure causes back-to-back timeouts at the sender. Both senders (CMT and CMT-PF) experience the first timeout on path 2 at ~6 seconds, and detect the failure after 6 back-to-back timeouts (PMR=5), at ~69 seconds (Figure 4.6). During the failure detection period, CMT continues to transmit data on path 2, experiencing consecutive timeouts and recurring rbuf blocking instances, while CMT-PF does not. CMT's throughput suffers until 69 seconds (until failure detection), after which CMT uses path 1 alone and completes the file transfer at around 80 seconds. On the other hand, CMT-PF transitions path 2 to PF state after the first timeout, and transmits only heartbeats on path 2 avoiding further rbuf blocking. Reduced rbuf

blocking helps CMT-PF to complete the file transfer (~15 seconds) using path 1 alone, even before path 2 failure is detected.

4.4.1.2 Evaluations during Varying Failure Detection Thresholds (PMR Values)

To achieve faster yet robust failure detection, [Acaro 2005] argues for varying the PMR based on a network's loss rate, and suggests PMR=3 for the Internet. Since the sender detects path failure after (PMR+1) consecutive timeouts, CMT's failure-induced rbuf blocking varies as the PMR varies. Let,

T_f = time when path 2 fails, and

T_d = time when the sender detects path 2 failure (after PMR+1 consecutive timeouts).

The *goodput during failure detection* (G) is defined as,

G = (application data received between T_f and $T_d \div (T_d - T_f)$).

Figure 4.7 plots CMT vs. CMT-PF average goodput (G) (in KB/second) with 5% error margin, for varying PMR values. The dashed line in Figure 4.7 denotes the maximum attainable goodput of an SCTP file transfer (application data received ÷ transfer time) using path 1 alone.

When the failure detection threshold is most aggressive (PMR=0), both CMT and CMT-PF detect path 2 failure after the first timeout. The senders experience similar rbuf blocking during this failure detection period and perform similarly (Figure 4.7). As PMR increases, the number of rbuf blocking instances during failure detection increases, resulting in increasing performance benefits from CMT-PF. As seen in Figure 4.7 as PMR and the failure detection duration increases, CMT-PF's goodput increases, whereas CMT's goodput decreases. Starting from PMR=3, CMT-PF's goodput is comparable or equal to the maximum attainable SCTP goodput. To conclude, *during*

permanent failure, CMT-PF performs as well as CMT for PMR=0, and better than CMT for PMR > 0.

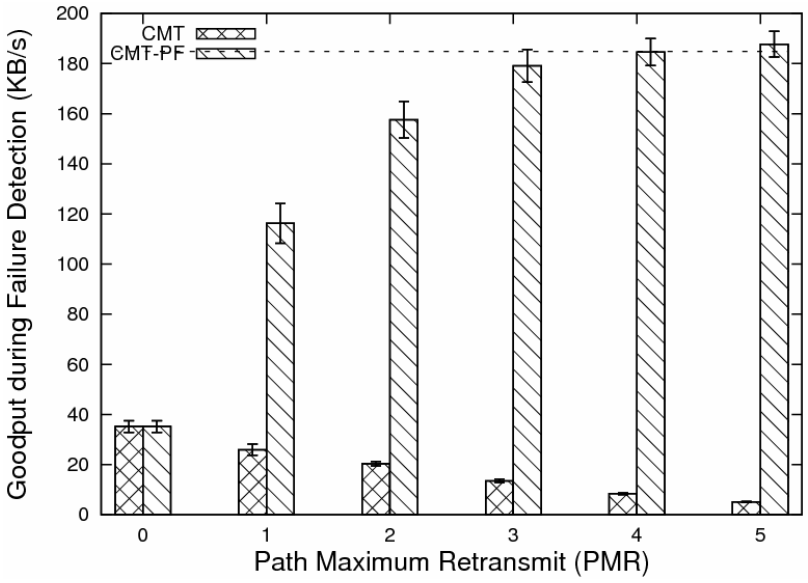


Figure 4.7: CMT vs. CMT-PF under Varying PMR Values

4.4.2 Evaluations during Short-term Failure

In the following experiments, path 2 (Figure 4.5) fails temporarily during the file transfer between A and B. The link between routers R_{20} and R_{21} goes down after 5 seconds from the start of file transfer, and is restored 5 seconds later.

4.4.2.1 Evaluations during Single Short-term Failure (without Congestion)

This experiment highlights how CMT and CMT-PF differ during a short-term failure. Neither path experiences any congestion loss. The short-term failure is long enough for the sender (CMT or CMT-PF) to experience three back-to-back timeouts on path 2. As in the failure case, CMT transmits data on path 2 after each of these timeouts, while CMT-PF does not. Therefore, CMT suffers from consecutive rbuf blocking and lower throughput than CMT-PF (Figure 4.8). Once path 2 recovers

at 10 seconds, CMT's data and CMT-PF's heartbeat transmissions on the path (after the 3rd timeout – ~12.5 seconds) are successful, and both CMT and CMT-PF complete the file transfer without further rbuf blocking.

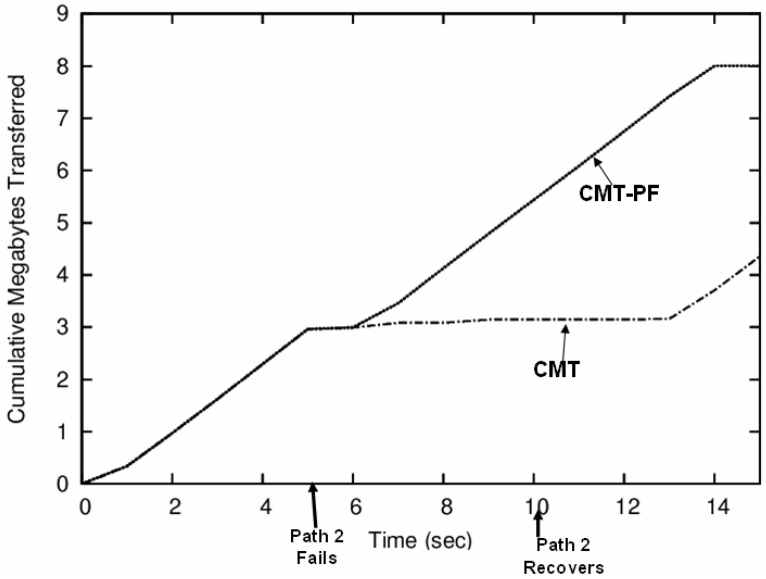


Figure 4.8: CMT vs. CMT-PF during Short-term Failure

4.4.2.2 Evaluations during Varying Receive Buffer Sizes

This second short-term failure experiment analyzes CMT vs. CMT-PF under varying levels of receive buffer constraints (receive buffer sizes). Let,

T_f = time when path 2 fails, and

T_r = time when path 2 is restored.

The *goodput during the short-term failure* (G) is defined as,

G = (application data received between T_f and $T_r \div (T_r - T_f)$).

Figure 4.9 plots CMT vs. CMT-PF average goodput (G) (in KB/second), with 5% error margin. As the receive buffer becomes more constrained, i.e., as rbuf size decreases, the chances of rbuf blocking increases. Consequently, CMT-PF's ability

to alleviate rbuf blocking is more valuable at smaller rbuf sizes, and CMT-PF performs increasingly better than CMT as rbuf size decreases.

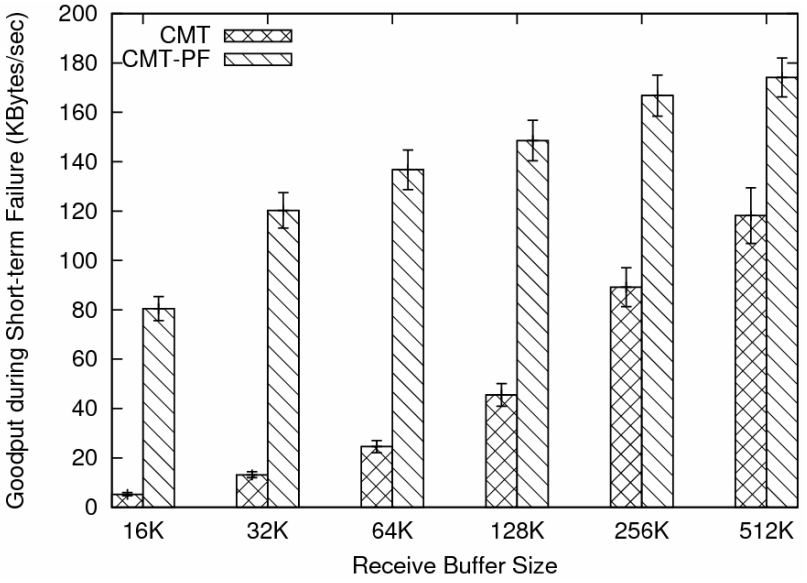


Figure 4.9: CMT vs. CMT-PF under Varying Rbuf Sizes

4.5 CMT vs. CMT-PF Evaluations during Congestion

The evaluations in the previous section confirmed that transitioning a destination to the PF state and avoiding data transmission on the PF path alleviates failure-induced rbuf blocking. During permanent and short-term failure scenarios, CMT-PF performed similar or better but never worse than CMT. We now investigate how the PF state transition fares when timeouts are caused by *non-failure scenarios* such as congestion [Natarajan 2008b].

Consider the case when timeout on a path, say p, is due to congestion rather than failure. Depending on the rbuf size and the different paths' characteristics, the transport sender may or may not be rbuf blocked until and/or after the timeout expiration, leading to the following two scenarios:

Sender is limited by rbuf: Both CMT and CMT-PF senders cannot transmit new data until the rbuf blocking is cleared, i.e., until after successful retransmission(s) of lost data. The only difference is that CMT considers *p* for retransmissions, whereas CMT-PF transmits a heartbeat on *p*, and tries to retransmit lost data on other active paths. (If all destinations are in the PF state, the CMT-PF sender transitions the destination with the least number of consecutive timeouts to the active state (Section 4.3), and retransmits lost data to this new active destination.)

Sender is not limited by rbuf: Assume that SCTP PDUs (data or heartbeats) transmitted after the first timeout on path *p* successfully reach the receiver. In CMT, the cwnd allows 1 MTU worth of new data transmission on *p* (Figure 4.10), and the corresponding SACK increments path *p's* cwnd by 1 MTU. At the end of 1 RTT after the timeout (shown by point A in Figure 4.10), (i) the cwnd on *p*=2 MTUs, and (ii) 1 MTU worth of new data has been successfully sent on *p*.

CMT-PF transmits a heartbeat on *p* and new data on other active path(s). (Note: if all destinations are marked PF, the CMT-PF sender transitions a PF destination to the active state.) Path *p* is marked active when the heartbeat ack reaches the sender. Therefore, after 1 RTT from the timeout (shown by point B in Figure 4.11), (i) cwnd on *p* =1 MTU (CMT-PF1), and (ii) no new data has been sent on *p*. Comparing points A and B in Figures 4.10 and 4.11, respectively, it can be seen that CMT has a 1 RTT "lead" in path *p*'s cwnd growth. Assuming no further losses on *p*, after *n* RTTs, the cwnd on *p* will be *2n* with CMT, and *2n-1* with CMT-PF1.

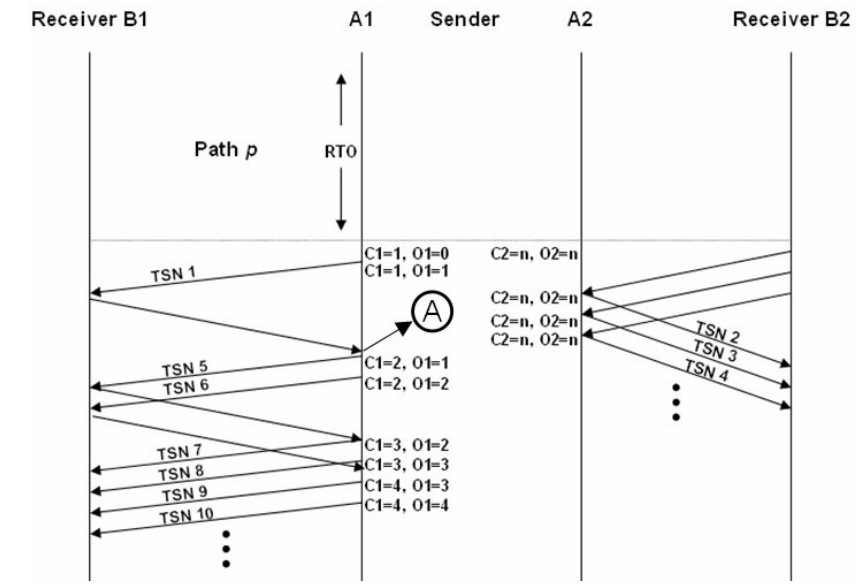


Figure 4.10: CMT Data Transfer during no Rbuf Blocking

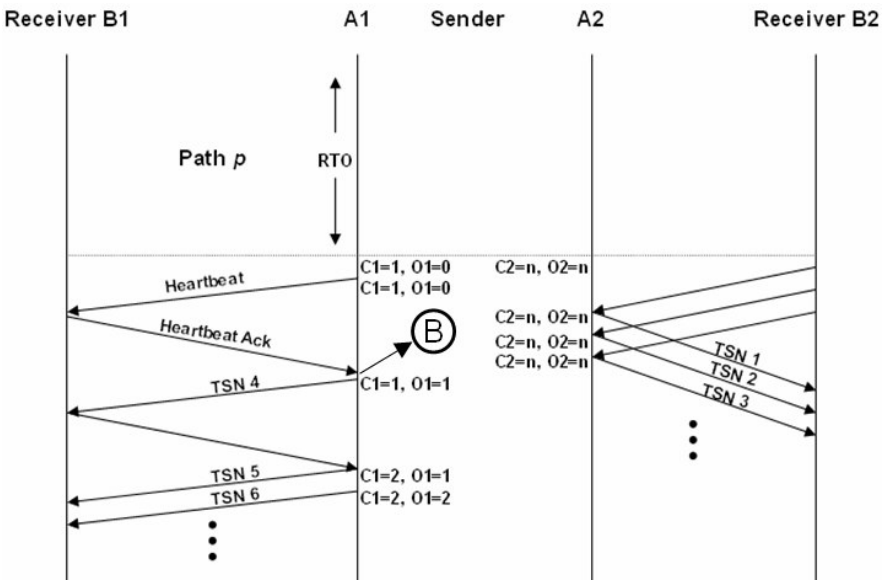


Figure 4.11: CMT-PF1 Data Transfer during no Rbuf Blocking

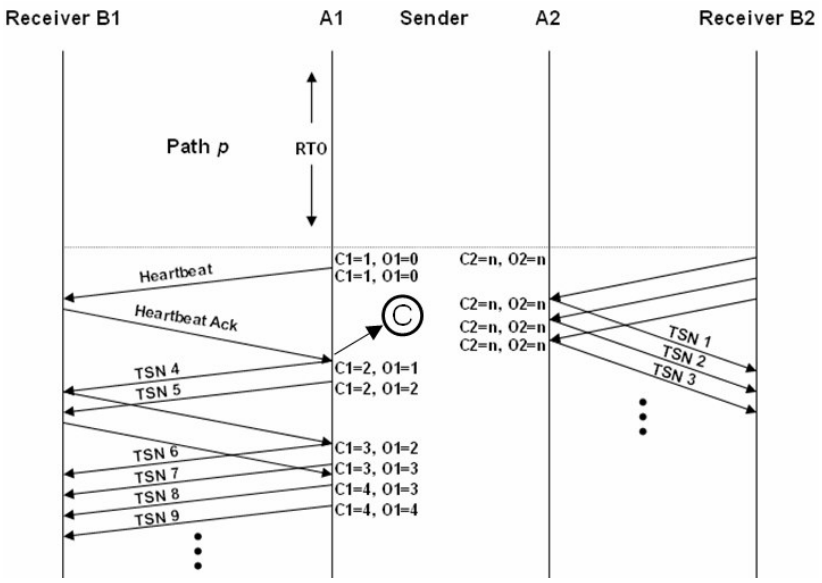


Figure 4.12: CMT-PF2 Data Transfer during no Rbuf Blocking

To avoid the 1 RTT lag in CMT-PF1's cwnd evolution, we propose CMT-PF2 which initializes path *p*'s cwnd to 2 MTUs after receiving a heartbeat ack (shown by point C in Figure 4.12). Assuming that today's Internet router queues deal with packets rather than bytes, the successful routing of a heartbeat PDU is equivalent to the successful routing of a data PDU. Hence, a heartbeat ack can be used to clock the transport layer sender in the same way as a data ack. In the following sections, any reference to CMT-PF implies CMT-PF2.

4.5.1 Simulation Setup

The simulation evaluations consider a dual-dumbbell topology with a more realistic loss model, as shown in Figure 4.13. Each router, R, is attached to five edge nodes. Dual-homed edge nodes A and B are the transport (CMT or CMT-PF) sender and receiver, respectively. The other edge nodes are single-homed, and introduce cross-traffic that instigates bursty periods of congestion and bursty congestion losses at the routers. Their last-hop propagation delays are randomly chosen from a uniform

distribution between 5-20 ms, resulting in end-to-end one-way propagation delays ranging ~35-65ms [Shakkottai 2004]. All links (both edge and core) have a buffer size twice the link's bandwidth-delay product, which is a reasonable setting in practice.

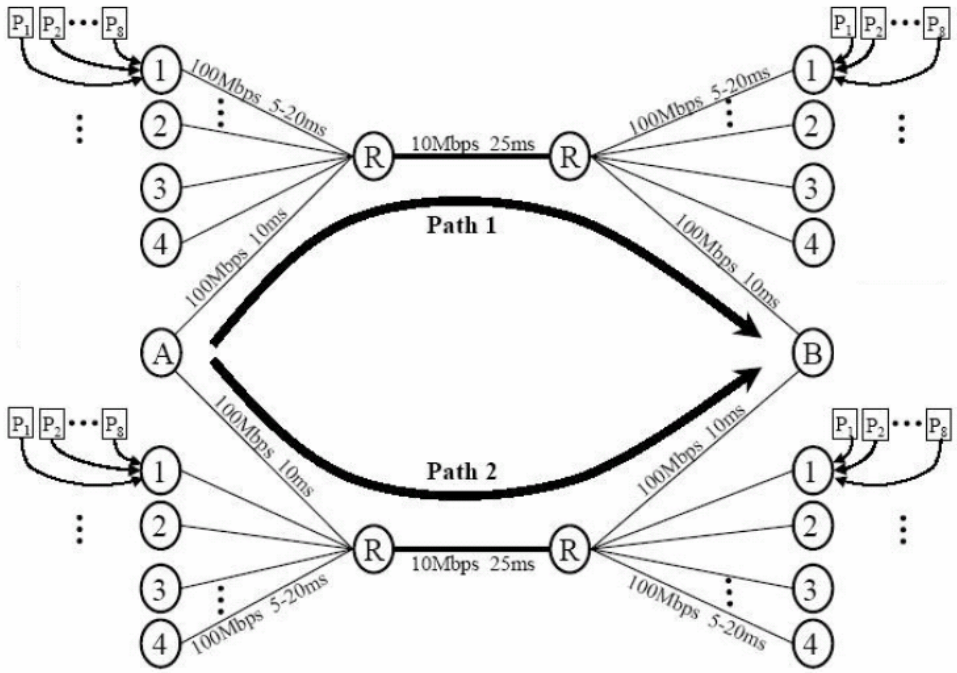


Figure 4.13: Topology for Non-failure Experiments

Each single-homed edge node has eight traffic generators, introducing cross-traffic with a Pareto distribution. The cross-traffic packet sizes are chosen to resemble the distribution found on the Internet: 50% are 44B, 25% are 576B, and 25% are 1500B [CAIDA, Fraleigh 2003]. The cross-traffic flows start at random times during the initial 5 seconds of the simulation. After an initial warm-up period of 10 seconds, sender A transmits a 32MB file to receiver B over paths 1 and 2. The result is a data transfer between A to B, over a network with self-similar cross-traffic, which resembles the observed nature of traffic on data networks [Leland 1993].

For both CMT and CMT-PF flows, rbuf=64KB, PMR=5, and loss rates are controlled by varying the cross-traffic load. The graphs in the subsequent discussions plot the average goodput (file size ÷ transfer time) of CMT vs. CMT-PF with 5% error margin.

4.5.2 Evaluations during Symmetric Loss Conditions

In the symmetric loss case, the aggregate cross-traffic load on both paths are similar and vary from 40%-100% of the core link's bandwidth.

4.5.2.1 Evaluations during Symmetric Path Delays

Both CMT and CMT-PF perform similarly (Figure 4.14) during low loss rates (i.e., low cross-traffic), since, most of the TPDU losses are recovered via fast retransmits as opposed to timeout recoveries.

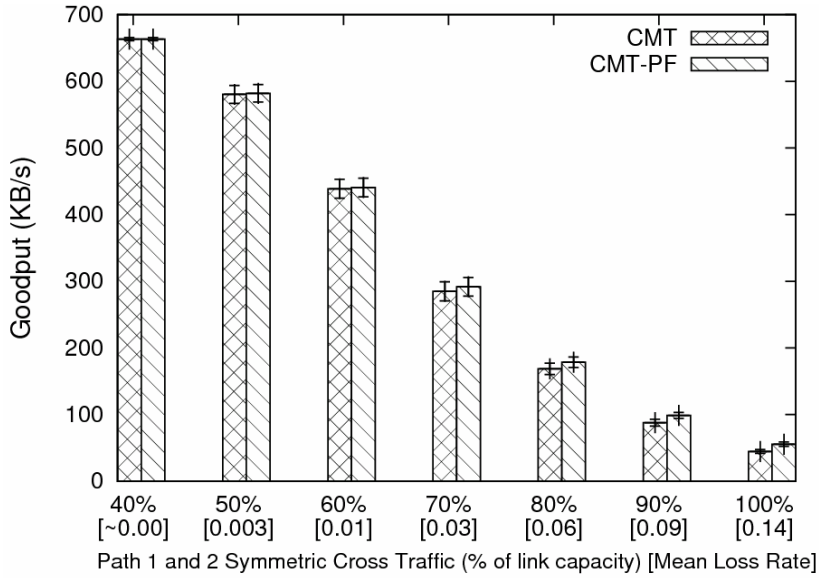


Figure 4.14: CMT vs. CMT-PF during Symmetric Loss and RTT Conditions

As the cross-traffic load and loss rate increases, the number of timeouts on each path increases. Under such conditions, the probability that both paths are *simultaneously*

marked "potentially-failed" increases in CMT-PF. To ensure that CMT-PF does not perform worse when all destinations are marked PF, CMT-PF transitions the destination with the smallest number of consecutive timeouts to the active state, allowing data to be sent to that destination (refer to Section 4.3). This modification guarantees that CMT-PF performs on par with CMT even when both paths experience high loss rates (Figure 4.14).

4.5.2.2 Evaluations during Asymmetric Path Delays

Under symmetric loss conditions, we now study how a path's RTT affects the throughput differences between CMT and CMT-PF. Note that any difference between CMT and CMT-PF transpires only after a timeout on a path. Assume that a path experiences a timeout event, and the next TPDU loss on the path takes place after n RTTs. After the timeout, CMT slow starts on the path, and the number of TPDUs transmitted on the path at the end of n RTTs = $1 + 2 + 4 \ldots + 2n = (2(n+1) - 1)$. CMT-PF uses the first RTT for a heartbeat transmission, and slow starts with initial cwnd=2 after receiving the heartbeat-ack. In CMT-PF, the number of TPDUs transmitted by end of n RTTs on the path = $0 + 2 + 4 \ldots + 2n = (2(n+1) - 2)$. Thus, after n RTTs, CMT transmits 1 TPDU more than CMT-PF, and the 1 TPDU difference is unaffected by the path's RTT. Therefore, when paths experience symmetric RTTs (a.k.a. symmetric RTT conditions), we expect the performance ratio between CMT and CMT-PF to remain unaffected by the RTT value.

We now consider a more interesting scenario when the independent end-to-end paths experience symmetric loss rates, but *asymmetric* RTT conditions. That is, path 1's RTT=x sec, and path 2's RTT=y sec ($x \neq y$). How do x and y impact CMT vs.

CMT-PF performance? More importantly, does CMT-PF perform worse when the paths have asymmetric RTTs?

Using topology in Figure 4.5, we performed the following Bernoulli loss model experiment to gain insight. The Bernoulli loss model simulations, while less realistic, take much less time than cross-traffic ones, and initial investigations revealed that both loss models resulted in similar trends between CMT and CMT-PF. Path 1's one-way propagation delay was fixed at 45ms while path 2's one-way delay varied as follows: 45ms, 90ms, 180ms, 360ms, and 450ms. Both paths experience identical loss rates ranging from 1%-10%.

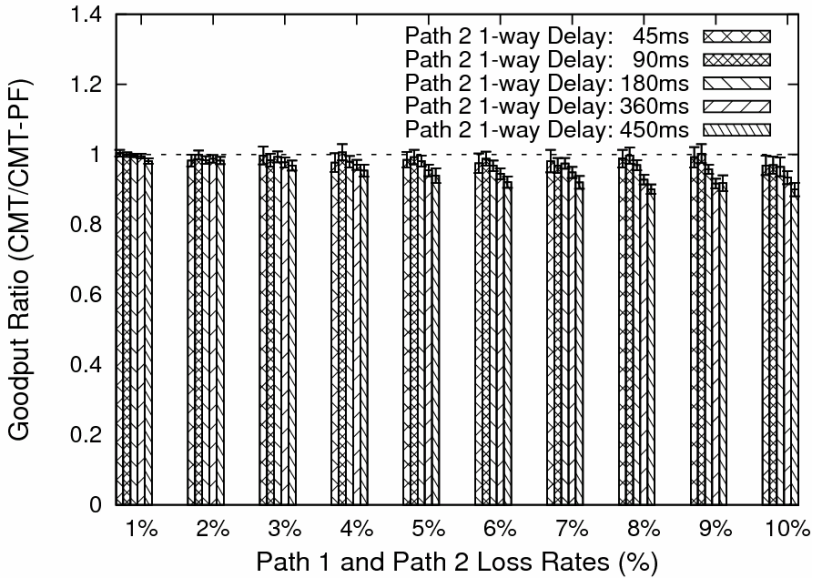


Figure 4.15: CMT vs. CMT-PF Goodput Ratios during Symmetric Loss and Asymmetric RTT Conditions

Figure 4.15 plots the ratio of CMT's goodput over CMT-PF's (relative performance difference) with 5% error margin. As expected, both CMT and CMT-PF perform equally well during symmetric RTT conditions. As the asymmetry in paths'

RTTs increases, an interesting dynamic dominates and CMT-PF performs slightly better than CMT (goodput ratios < 1).

Further investigation revealed the following about CMT vs. CMT-PF rbuf blocking durations, shown in Figure 4.16. For each combination of path 2's delay and loss rate, Figure 4.16 plots the ratio of rbuf blocked durations (CMT over CMT-PF) during timeout recoveries. As path 2 one-way delay and loss rate increases, the ratio becomes increasingly greater than 1, signifying that a CMT sender suffers longer rbuf blocking durations than CMT-PF.

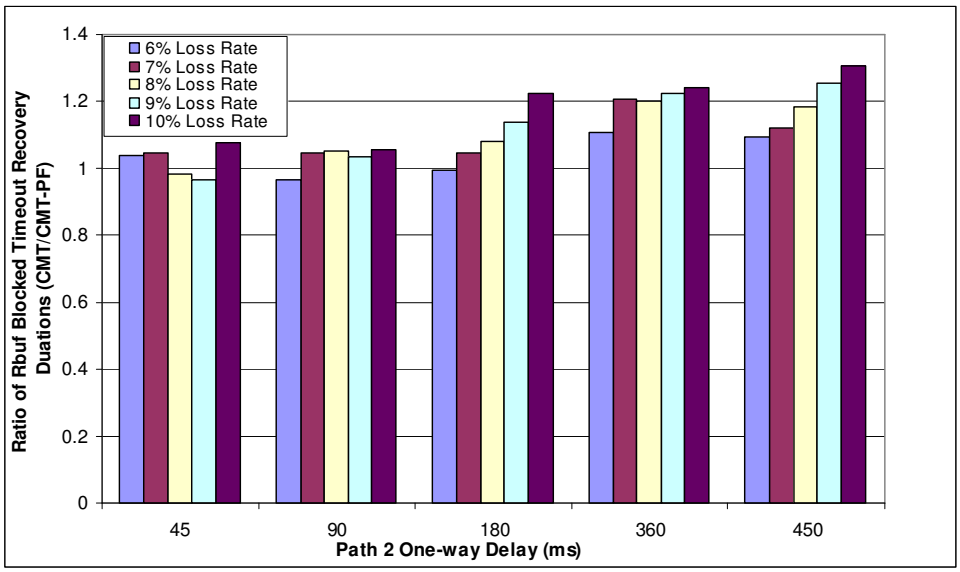


Figure 4.16: CMT vs. CMT-PF Rbuf Blocking Durations

Note that rbuf blocking depends on the frequency of loss events (loss rate), and the duration of loss recovery. As loss rate increases, the probability that a sender experiences consecutive timeout events on the path increases. After the first timeout, CMT-PF transitions the path to PF, and avoids data transmission on the path (as long as another active path exists) until a heartbeat-ack confirms the path as active. But, a CMT sender suffers back-to-back timeouts on *data* sent on the path, with exponential

backoff of timeout recovery period. As path 2's RTT increases, path 2's RTO increases, and the back-to-back timeouts on data result in longer rbuf blocking durations in CMT than CMT-PF. Therefore, as path 2's RTTs increase, CMT's goodput degrades more than CMT-PF's, and the goodput ratio decreases (Figure 4.15).

In summary, *during symmetric loss conditions, CMT and CMT-PF perform equally well when paths experience symmetric RTT conditions. As the RTT asymmetry increases, CMT-PF demonstrates a slight advantage at higher loss rates*.

4.5.3 Evaluations during Asymmetric Loss Conditions

In the asymmetric loss experiment, paths 1 and 2 experience different cross-traffic loads. The aggregate cross-traffic load on path 1 is set to 50% of the core link bandwidth, while on path 2 the load varies from 50%-100% of the core link bandwidth.

Variant	Path 2 Cross-traffic %	# of Consecutive Timeouts			
		2	3	4	5
CMT	70	0.49	0.02	0	0
CMT-PF		0	0	0	0
CMT	80	1.13	0.07	0	0
CMT-PF		0	0	0	0
CMT	90	3.73	0.60	0.09	0.02
CMT-PF		0.02	0.02	0	0
CMT	100	9.42	1.62	0.18	0.04
CMT-PF		0.04	0.04	0	0

Table 4.1: CMT vs. CMT-PF Mean Consecutive Data Timeouts on Path 2

As discussed in the previous sub-section, as path 2's cross-traffic load increases, the probability that a sender experiences back-to-back timeouts on path 2 increases. CMT suffers a higher number of consecutive timeouts on *data* (Table 4.1) resulting in more extended rbuf blocking periods when compared with CMT-PF.

Therefore, as path 2's cross-traffic load increases, CMT-PF performs better than CMT (Figure 4.18).

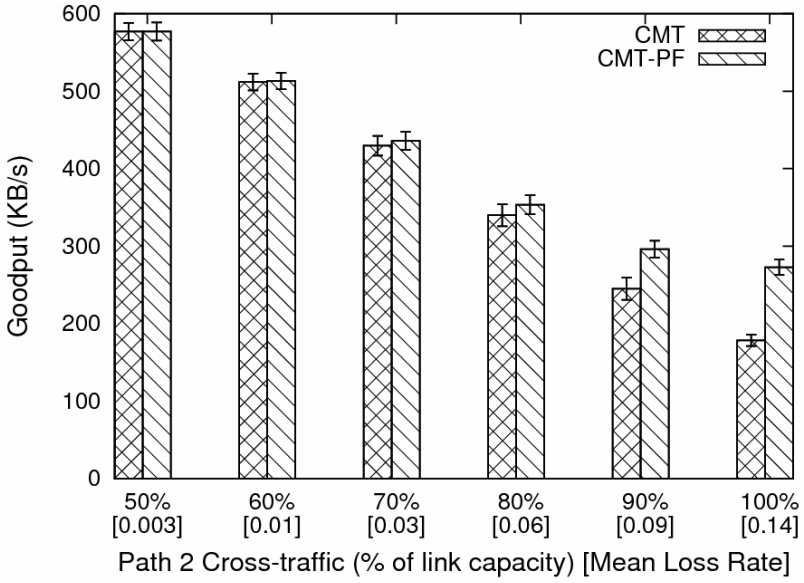


Figure 4.18: CMT vs. CMT-PF during Asymmetric Loss Conditions

Variant	Path 2 Cross-traffic %	Aggregate Transmissions		
		Path 1	Path 2	Path1/Path2
CMT	70	13,857	9,486	**1.5**
CMT-PF		14,002	9,344	**1.5**
CMT	80	15,530	7,902	**2.0**
CMT-PF		16,029	7,416	**2.2**
CMT	90	17,137	6,401	**2.7**
CMT-PF		18,153	5,362	**3.4**
CMT	100	18,093	5,508	**3.3**
CMT-PF		20,318	3,193	**6.4**

Table 4.2: CMT vs. CMT-PF Mean Number of Transmissions

The asymmetric loss experiment also helps to understand the following difference in CMT vs. CMT-PF's transmission strategy. In CMT, RTX_SSTHRESH is a *retransmission* policy, and is not applied to new data transmissions. In CMT-PF, a path is marked PF after a timeout, and as long as active path(s) exist, CMT-PF avoids retransmissions on the PF path. Once the retransmissions are all sent, CMT-PF's data

transmission strategy is applied to new data, and CMT-PF *avoids new data transmissions* on the PF path. As shown in Table 4.2, when compared to CMT, CMT-PF reduces the number of (re)transmissions on the higher loss rate path 2 and (re)transmits more on the lower loss rate path 1. This transmission difference (ratio of transmissions on path 1 over path 2) between CMT-PF and CMT increases as the paths become more asymmetric in their loss conditions.

In summary, *CMT-PF does not perform worse than CMT during asymmetric path loss conditions. In fact, CMT-PF is a better transmission strategy than CMT, and performs better as the asymmetry in path loss increases*.

4.6 Conclusion, Ongoing and Related Work

Using simulations, we demonstrated that retransmission policies using CMT with a "potentially-failed" destination state (CMT-PF) outperform CMT during permanent and short-term failures. During permanent failures, CMT-PF employs a better failure detection process than CMT even under aggressive failure detection thresholds. Investigations during symmetric loss conditions revealed that CMT-PF performs as well as CMT during symmetric path RTTs, and slightly better when the paths experience asymmetric RTT conditions. Also, CMT-PF employs a better transmission strategy than CMT during asymmetric loss conditions.

Our evaluations conclude that CMT-PF (i) reduces rbuf blocking during failure scenarios, and (ii) performs on par or slightly better than CMT during non-failure scenarios. *Since our findings demonstrate CMT-PF performs better or similar but never worse than CMT, we recommend CMT be replaced by CMT-PF in existing and future implementations and RFCs*.

4.6.1 CMT-PF Implementation in FreeBSD

Joeseph Szymanski extended the FreeBSD CMT implementation to include CMT-PF. The following emulation experiments were performed using this FreeBSD implementation.

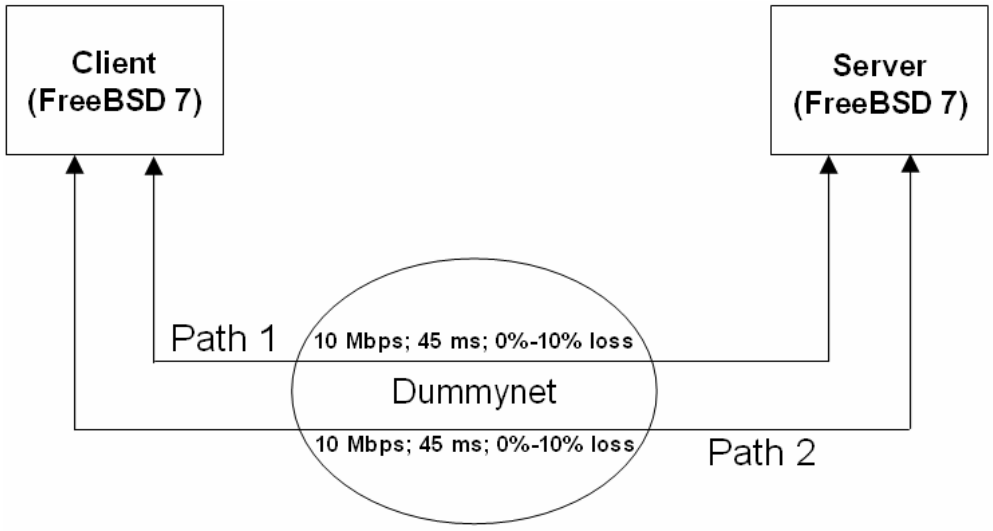


Figure 4.19: Emulation Topology for CMT vs. CMT-PF Experiments

The experimental topology, shown in Figure 4.19, consists of three nodes running FreeBSD 7 — a client node, a server node, and a third node running the Dummynet traffic shaper [Rizzo 1997]. The server and client nodes are connected by two independent paths, with symmetric bandwidth and propagation delay characteristics. The paths also experience Bernoulli losses, with loss rates varying from 0%-10%. The forward and reverse queue sizes for both paths are set to 1000KB. The transport layer receive window=64KB, and PMR=5. At time t=0, the server initiates a bulk file transfer to the client.

4.6.1.1 Single Failure Scenario

To validate the behavioral differences between CMT and CMT-PF, we emulated a single failure scenario, similar to the scenario described in Section 4.4.1.1. Neither paths experience loss in this experiment. At time t=5, path 2 fails; this failure is emulated by setting up appropriate Dummynet rules to block all packets traversing on path 2 to and from the client and server, respectively Figure 4.20 plots the cumulative bytes received at the client during this transfer.

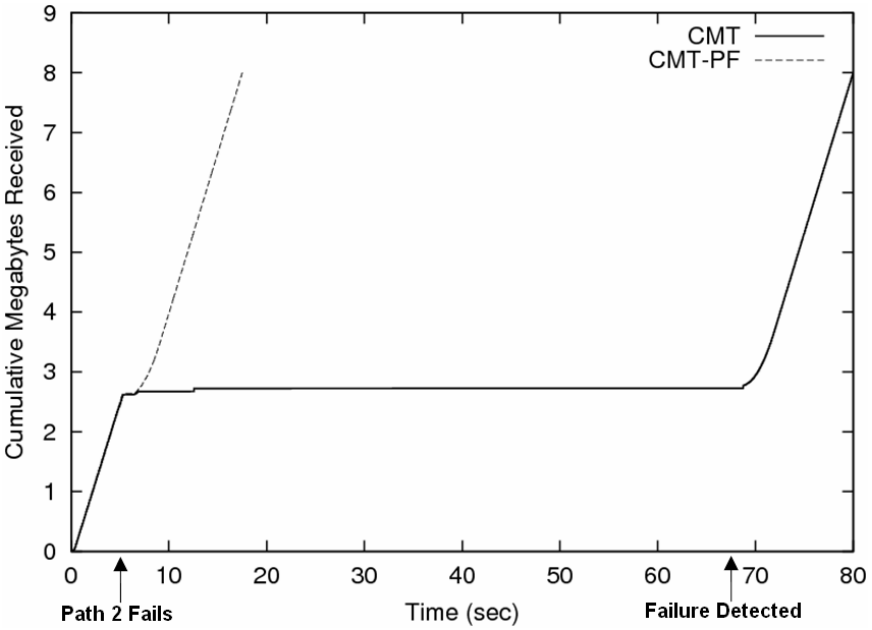


Figure 4.20: CMT vs. CMT-PF during Permanent Path Failure

Figure 4.20 can be compared with the corresponding simulation results shown in Figure 4.6. As observed in the simulations, path 2 failure causes consecutive timeouts and rbuf blocking instances in CMT, which prevents data transmission until failure detection (~69 seconds). After failure detection, CMT transmits data using only path 1, and finishes the file transfer ~80 seconds. The CMT-PF sender transitions path

2 to PF after the first timeout (~6.5 seconds), and transmits only heartbeats on path 2. Data transmission continues on path 1 and the file transfer finishes ~18 seconds.

4.6.1.2 Symmetric Loss Conditions

This experiment is designed to compare CMT vs. CMT-PF under varying congestion levels. Similar to the scenario described in Section 4.5.2.1, paths 1 and 2 experience symmetric loss rates, varying from 1%-10%. Figure 4.21 plots the average goodput (file size ÷ transfer time) of CMT vs. CMT-PF with 5% error margin.

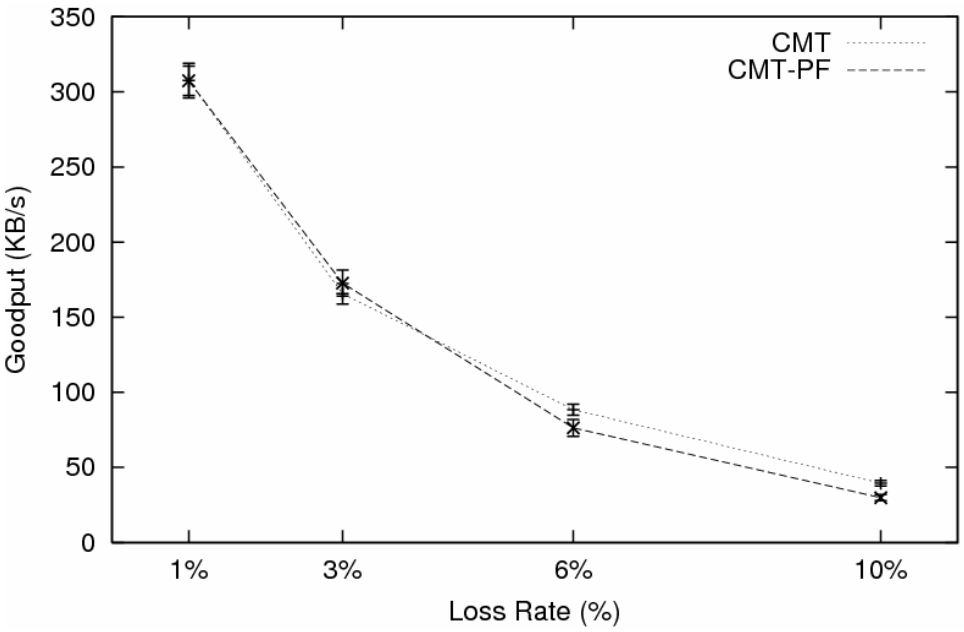


Figure 4.21: CMT vs. CMT-PF during Symmetric Loss Conditions

As observed in the simulations (Figure 4.14), both CMT and CMT-PF perform similarly during low loss rates, since, most of the TPDU losses are recovered via fast retransmits as opposed to timeout recoveries. As loss rate increases, the probability that both paths are simultaneously marked PF increases in CMT-PF. Unlike the simulation results, CMT-PF performs slightly worse than CMT during such high

loss conditions. Further investigation exposed few potential bugs in the CMT-PF implementation. We are currently exploring these issues.

4.6.2 CMT-PF Applicability during Mobile Handovers

Mobile SCTP (mSCTP) [Koh 2004, Koh 2005] provides transport layer features such as multihoming and dynamic address reconfiguration [RFC5061] to achieve seamless handover in the context of heterogeneous wireless access networks. [Budzisz 2008] investigates the possibility of using CMT to increase throughput of an mSCTP association during handover scenarios. Since path failures are common in a wireless network, [Budzisz 2008] proposes to employ CMT-PF instead of CMT.

Simulation evaluations presented in [Budzizs 2008] show that, while CMT-PF's performance during handover is sensitive to various parameters, CMT-PF reduces rbuf blocking and improves throughput for parameters typical of today's heterogeneous wireless access networks.

Chapter 5

SUMMARY AND CONCLUSIONS

This dissertation investigated three issues related to the transport layer and proposed solutions to address these issues. This chapter summarizes our contributions for each issue, and concludes the dissertation.

5.1 Issue (1): Web over Multistreamed Transport

We examined HOL blocking, and its effects on web response times in HTTP over TCP. Since a multistreamed transport such as SCTP eliminates inter-object HOL blocking, we hypothesized that SCTP streams would improve web response times. We designed and implemented HTTP over SCTP in the open source Apache server and Firefox browser. Emulation evaluations showed that persistent and pipelined HTTP 1.1 transfers over a single multistreamed SCTP association improves web response times when compared to similar transfers over a single TCP connection. The difference in TCP vs. SCTP response times increases and is more visually perceivable in the high latency and lossy browsing conditions found in the developing world.

The current workaround to improve an end user's perceived WWW performance is to download an HTTP transfer over multiple TCP connections. While we expected multiple TCP connections to improve HTTP throughput, emulation results showed that the competing and bursty nature of multiple TCP senders degraded HTTP performance especially in low bandwidth last hops. In such browsing conditions,

a single multistreamed SCTP association not only eliminated HOL blocking, but also boosted throughput compared to multiple TCP connections.

Our body of work in HTTP over SCTP has triggered significant interest in the area. We are currently working with the IETF to standardize our HTTP over SCTP streams design.

5.2 Issue (2): Reneging and Selective Acks

We investigated how the existing SACK mechanism degrades end-to-end performance when out-of-order data is non-rengable. Using simulation, we showed that SACKs result in inevitable send buffer wastage, which increases as the frequency of loss events and loss recovery durations increase. We introduced a fundamentally new ack mechanism, Non-Renegable Selective Acknowledgments (NR-SACKs), for SCTP. An SCTP receiver used NR-SACKs to explicitly identify some or all out-of-order data as being non-renegable, allowing the sender to free up send buffer sooner than if the data were only SACKed. Simulation comparisons showed that NR-SACKs enabled (i) efficient utilization of a transport sender's memory, and (ii) throughput improvements in CMT. We are currently working with the IETF to standardize NR-SACKs for SCTP.

5.3 Issue (3): CMT during Path Failures

We demonstrated that CMT suffers from significant throughput degradation during permanent and short-term path failures. We introduced a new destination state called the "Potentially Failed" (PF) state. CMT's failure detection and (re)transmission policies were augmented to include the PF state. The modified CMT, called CMT-PF, outperformed CMT during failures – even under aggressive failure

detection thresholds. During non-failure scenarios such as congestion, CMT-PF performed on par or better but never worse than CMT. In light of these findings, we recommend CMT be replaced by CMT-PF in existing and future CMT implementations and RFCs.

REFERENCES

[Almeida 1996] J. Almeida, V. Almeida, D. Yates, "Measuring the Behavior of a World-Wide Web Server," Technical Report TR-96-025, Computer Science Department, Boston University, Boston, USA, 1996.

[Akamai 2006] "Akamai and JupiterResearch Identify 4 Seconds as New Threshold of Acceptability for Retail Web Page Response Times," Press Release, Akamai Technologies Inc, November 2006. www.akamai.com/html/about/press/releases/2006/press_110606.html.

[Alamgir 2002] R. Alamgir, M. Atiquzzaman, W. Ivancic, "Effect of Congestion Control on the Performance of TCP and SCTP over Satellite Networks", NASA Earth Science Technology Conference, Pasadena, USA, June 2002.

[Andrew 2008] L. Andrew, C Marcondes, S. Floyd, L. Dunn, R. Guillier, W. Gang, L. Eggert, S. Ha, I. Rhee. "Towards a Common TCP Evaluation Suite," PFLDnet, Manchester, UK, March 2008.

[Apache] The Apache Software Foundation, October 2007. www.apache.org

[Arlitt 1997] M. Arlitt, C. Williamson, "Internet Web Servers: Workload Characterization and Performance Implications," IEEE/ACM Transactions on Networking, 5(5), pp. 631–645, October 1997.

[Baggaley 2007] J. Baggaley, B. Batpurev, J. Klaas, "Technical Evaluation Report 61: The World-Wide Inaccessible Web, Part 2: Internet Routes," International Review of Research in Open and Distance Learning, 8(2), 2007.

[Balakrishnan 1998a] H. Balakrishnan, V. N. Padmanabhan, S. Seshan, M. Stemm, R. Katz, "TCP Behavior of a Busy Internet Server: Analysis and Improvements," IEEE INFOCOM, San Francisco, USA, March 1998.

[Balakrishnan 1998b] H. Balakrishnan, R. Katz, "Explicit Loss Notification and Wireless Web Performance," IEEE GLOBECOM, Sydney, Australia, November 1998.

[Balakrishnan 1999] H. Balakrishnan, H.S. Rahul, S. Seshan, "An Integrated Congestion Management Architecture for Internet Hosts," ACM SIGCOMM, Cambridge, USA, August 1999.

[Barford 1999] P. Barford, A. Bestavros, A. Bradley, M. Crovella, "Changes in Web Client Access Patterns: Characteristics and Caching Implications," World Wide Web Journal, 2(1-2), pp. 15–28, 1999.

[Bickhart 2005] R. Bickhart, "SCTP Shim for Legacy TCP Applications", MS Thesis, Department of Computer & Information Sciences, University of Delaware, USA, August 2005.

[Briscoe 2007] B. Briscoe, "Flow Rate Fairness: Dismantling a Religion," ACM Computer Communications Review, 37(2), pp. 63–74, April 2007.

[Budzizs 2008] L. Budzisz, "Stream Control Transmission Protocol (SCTP): A Proposal for Seamless Handoff Management at the Transport Layer in Heterogeneous Wireless Networks," PhD Dissertation in progress, Department of Signal Theory and Communications, Technical University of Catalonia (UPC), Barcelona, Spain, October 2008.

[CAfrica] Connectivity in Africa, October 2007. www.connectivityafrica.ca.

[CAIDA] CAIDA: Packet Sizes and Sequencing, March 1998. www.caida.org

[Cao 2004] J. Cao, W.S. Cleveland, Y. Gao, K. Jeffay, F.D. Smith, M.C, Weigle, "Stochastic Models for Generating Synthetic HTTP Source Traffic", IEEE INFOCOM, Hong Kong, China, March 2004.

[Caro 2005] A. Caro, "End-to-End Fault Tolerance using Transport Layer Multihoming," PhD Dissertation, Department of Computer & Information Sciences, University of Delaware, USA, August 2005.

[Caro 2006] A. Caro, P. Amer, R. Stewart, "Retransmission Policies for Multihomed Transport Protocols," Computer Communications, 29(10), pp. 1798–1810, June 2006.

[Chakravorty 2002] R. Chakravorty, I. Pratt, "WWW Performance over GPRS," IEEE International conference in Mobile and Wireless Communications Networks, Stockholm, Sweden, September 2002.

[Chan 2002] M.Chan , R. Ramjee, "TCP/IP Performance over 3G Wireless Links with Rate and Delay Variation," 8th International Conference on Mobile Computing and Networking, Georgia, USA, September 2002.

[Chandra 2001] B. Chandra, M. Dahlin, L. Gao, A. Nayate, "End-to-End WAN Service Availability," 3rd USENIX Symposium on Internet Technologies and Systems, San Francisco, USA, March 2001.

[Cottrell 2006] L. Cottrell, A. Rehmatullah, J. Williams, A. Khan, "Internet Monitoring and Results for the Digital Divide," International ICFA Workshop on Grid Activities within Large Scale International Collaborations, Sinaia, Romania, October 2006.

[Diot 1999] C. Diot, F. Gagnon, "Impact of Out-of-sequence Processing on the Performance of Data Transmission," Computer Networks, 31(5), pp. 475–492, March 1999.

[Du 2006] B. Du, M. Demmer, E. Brewer, "Analaysis of WWW Traffic in Cambodia and Ghana," 15th International Conference on World Wide Web, Edinburgh, Scotland, May 2006.

[Ekiz 2007] N. Ekiz, P. Natarajan, J. Iyengar, A. Caro, "ns-2 SCTP Module," Version 3.7, September 2007. pel.cis.udel.edu.

[Faber 1999] T. Faber, J. Touch, W. Yue, "The TIME-WAIT State in TCP and Its Effect on Busy Servers," IEEE INFOCOM, New York, USA, March 1999.

[Ford 2007] B. Ford, "Structured Streams: A New Transport Abstraction," ACM SIGCOMM, Kyoto, Japan, August 2007.

[Fraleigh 2003] C. Fraleigh, S. Moon, B. Lyles, C. Cotton, M. Khan, D. Moll, R. Rockell, T. Seely, C. Diot, "Packet-level Traffic Measurements from the Sprint IP Backbone," IEEE Network, 17(6), pp. 6–16, November 2003.

[FreeBSD] FreeBSD TCP and SCTP Implementation, October 2007. www.freebsd.org/cgi/cvsweb.cgi/src/sys/netinet/

[Gettys 1998] J. Gettys, H. Nielsen, "The WebMUX Protocol," IETF Internet Draft (expired), August, 1998.

[Gettys 2002] J. Gettys, Email to End2end-interest Mailing List, October, 2002. www.postel.org/pipermail/end2end-interest/2002-October/002436.html.

[Gimp] The GNU Image Manipulation Program, www.gimp.org.

[Gurtov 2004] A. Gurtov, S. Floyd, "Modeling Wireless Links for Transport Protocols," ACM Computer Communication Review, 34(2), pp. 85–96, April 2004.

[Houtzager 2003] G. Houtzager, C. Williamson, "A Packet-Level Simulation Study of Optimal Web Proxy Cache Placement," 11th IEEE International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunications Systems, Orlando, USA, October 2003.

[HTTP-NG] HTTP-NG working group (historic). www.w3.org/Protocols/HTTP-NG/

[Iyengar 2005] J. Iyengar, P. Amer, R. Stewart, "Receive Buffer Blocking in Concurrent Multipath Transfer," IEEE GLOBECOM, St. Louis, USA, November 2005.

[Iyengar 2006] J. Iyengar, P. Amer, R. Stewart, "Concurrent Multipath Transfer using SCTP Multihoming over Independent End-to-end Paths," IEEE/ACM Transactions on Networking, 14(5), pp. 951–964, October 2006.

[Jacobson 1988] V. Jacobson, "Congestion Avoidance and Control," ACM SIGCOMM, Stanford, USA, August 1988.

[Jurvansuu 2007] M. Jurvansuu, J. Prokkola, M. Hanski, P. Perala, "HSDPA Performance in Live Networks," IEEE International Conference on Communications, Glasgow, Scotland, June 2007.

[Koh 2004] S. Koh, M. Chang, M. Lee, "mSCTP for Soft Handover in Transport Layer," IEEE Communications Letters, 8(3), pp. 189–191, March 2004.

[Koh 2005] S. Koh, Q. Xie, "Mobile SCTP (mSCTP) for IP Handover Support, draft-sjkoh-msctp, IETF Internet Draft (expired)," October 2005.

[Krishnamurthy 2001] B. Krishnamurthy, C. Wills, Y. Zhang, "On the Use and Performance of Content Distribution Networks," ACM SIGCOMM Internet Measurement Workshop, California, USA, November 2001.

[Labovitz 1999] C. Labovitz, A. Abuja, F. Jahania, "Experimental Study of Internet Stability and Wide-Area Backbone Failures," 29th International Symposium on Fault-Tolerant Computing, Madison, USA, June 1999.

[Labovitz 2000] C. Labovitz, A. Abuja, A. Bose, F. Jahanian, "Delayed Internet Routing Convergence," ACM SIGCOMM, Stockholm, Sweden, August 2000.

[Leland 1993] W. Leland, M. Taqqu, W. Willinger, D. Wilson, "On the Self-similar Nature of Ethernet Traffic," ACM SIGCOMM, San Francisco, USA, September 1993.

[Mahdavi 1997] J. Mahdavi, S. Floyd, "TCP-Friendly Unicast Rate-Based Flow Control," Technical note sent to the end2end-interest mailing list, January 1997.

[Markopoulou 2004] A. Markopoulou, G.Iannaccone, S. Bhattacharyya, C. Chuah and C. Diot, "Characterization of Failures in an IP Backbone," IEEE INFOCOM, Hong Kong, China, March 2004.

[Movies] HTTP over SCTP versus HTTP over TCP Movies, October 2007. http://www.cis.udel.edu/%7Eamer/PEL/leighton.movies/index.html

[Mozilla] Mozilla Suite of Applications, October 2007. www.mozilla.org.

[Natarajan 2006a] P. Natarajan, J. Iyengar, P. Amer, R. Stewart, "SCTP: An Innovative Transport Layer Protocol for the Web," 15th International conference on World Wide Web, Edinburgh, Scotland, May 2006.

[Natarajan 2006b] P. Natarajan, J. Iyengar, P. Amer, R. Stewart, "Concurrent Multipath Transfer using Transport Layer Multihoming: Performance under Network Failures," IEEE MILCOM, Washington D. C., USA, October 2006.

[Natarajan 2007] P. Natarajan, P. Amer, R. Stewart, "The Case for Multistreamed Web Transport in High Latency Networks," TR2007-342, Department of Computer & Information Sciences, University of Delaware, USA, October 2007.

[Natarajan 2008a] P. Natarajan, P. Amer, E. Yilmaz, R. Stewart, J. Iyengar, "Non-Renegable Selective Acknowledgements (NR-SACKs) for SCTP," draft-natarajan-tsvwg-nrsack, IETF Internet Draft (in progress), August 2008.

[Natarajan 2008b] P. Natarajan, N. Ekiz, P. Amer, J. Iyengar, R. Stewart, "Concurrent Multipath Transfer using Transport Layer Multihoming: Introducing the Potentially-failed Destination State," IFIP-TC6 Networking, Singapore, Singapore, May 2008.

[Natarajan 2008c] P. Natarajan, P. Amer, R. Stewart, "Multistreamed Web Transport for Developing Regions," ACM SIGCOMM Workshop on Networked Systems for Developing Regions (NSDR), Seattle, USA, August 2008.

[Natarajan 2008d] P. Natarajan, F. Baker, P. Amer, "Multiple TCP Connections Improve HTTP Throughput – Myth or Fact?," TR2008-333, Department of Computer & Information Sciences, University of Delaware, USA, August 2008.

[Natarajan 2008e] P. Natarajan, N. Ekiz, E. Yilmaz, P. Amer, J. Iyengar, R. Stewart, "Non-Renegable Selective Acknowledgements (NR-SACKs) for SCTP," 16th International Conference on Network Protocols, Orlando, USA, October 2008.

[Natarajan 2008f] P. Natarajan, P. Amer, J. Leighton, R. Stewart, J. Iyengar, "SCTP as a Transport Protocol for HTTP," draft-natarajan-http-over-sctp, IETF Internet Draft (in progress), October 2008.

[Nielsen 1999] J. Nielsen, "Designing Web Usability: The Practice of Simplicity," New Riders, 1999, ISBN: 156205810X

[NS-2] ns-2 documentation and software, Version 2.33, March 2008. www.isi.edu/nsnam/ns.

[Padmanabhan 1998] V. N. Padmanabhan, "Addressing the Challenges of Web Data Transport," PhD Dissertation, Computer Science Division, University of California at Berkeley, USA, September 1998.

[Paxon 1997] V. Paxson, "End-to-End Routing Behavior in the Internet," IEEE/ACM Transactions on Networking, 5(5), pp. 601–615, October 1997.

[PEL] Protocol Engineering Lab Home Page, http://pel.cis.udel.edu

[PingER] PingER Detail Reports, October 2007. http://www-wanmon.slac.stanford.edu/cgi-wrap/pingtable.pl

[Rahman 2002] S. Rahman, M. Pipattanasomporn, "Alternate Technologies for Telecommunications and Internet Access in Remote Locations," 3rd Mediterranean Conference and Exhibition on Power Generation, Transmission, Distribution and Energy Conversion, Greece, November 2002.

[RFC1122] R. Braden, "Requirements for Internet hosts – Communication Layers," RFC1122, October 1989.

[RFC1379] R. Braden, "Transaction TCP - Concepts," RFC 1379, September 1992.

[RFC2581] M. Allman, V. Paxson, and W. Stevens, "TCP Congestion Control," RFC 2581, April 1999.

[RFC2616] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach , T. Berners-Lee, "Hypertext Transfer Protocol – HTTP/1.1," RFC 2616, June 1999.

[RFC2760] M. Allman, S. Dawkins, D. Glover, J. Griner, D. Tran, T. Henderson, J. Heidemann, J. Touch, H. Kruse, S. Ostermann, K. Scott, J. Semke, "Ongoing TCP Research Related to Satellites," RFC 2760, Febraury 2001.

[RFC2861] M. Handley, J. Padhye, S. Floyd, “TCP Congestion Window Validation,” RFC2861, June 2000.

[RFC2988] V. Paxson, M. Allman, "Computing TCP's Retransmission Timer," RFC 2988, November 2000.

[RFC3042] M. Allman, H. Balakrishnan, S. Floyd, "Enhancing TCP's Loss Recovery using Limited Transmit," RFC 3042, January 2001.

[RFC3124] H. Balakrishnan, S. Seshan, "The Congestion Manager," RFC 3124, June 2001.

[RFC3135] J. Border, M. Kojo, J. Griner, G. Montenegro, Z. Shelby, "Performance Enhancing Proxies Intended to Mitigate Link-Related Degradations," RFC 3135, June 2001.

[RFC3390] M. Allman, S. Floyd, C. Partridge, "Increasing TCP's Initial Window," RFC 3390, October 2002.

[RFC3465] M. Allman, "TCP Congestion Control with Appropriate Byte Counting (ABC)," RFC3465, February 2003.

[RFC3481] H. Inamura, G. Montenegro, R. Ludwig, A. Gurtov, F. Khafizov, "TCP over Second (2.5G) and Third (3G) Generation Wireless Networks," RFC 3481, February 2003.

[RFC3517] E. Blanton, M. Allman, K. Fall, L. Wang, "A Conservative Selective Acknowledgment (SACK)-based Loss Recovery Algorithm for TCP," RFC 3517, April 2003.

[RFC4960] R. Stewart, “Stream Control Transmission Protocol," RFC 4960, September 2007.

[RFC5061] R. Stewart, Q. Xie, M. Tuexen, S. Maruyama, and M. Kozuka, "Stream Control Transmission Protocol (SCTP) Dynamic Address reconfiguration," RFC 5061, September 2007.

[RFC793] J. Postel, "Transmission Control Protocol," RFC 793, September 1981.

[RFC896] J. Nagle, "Congestion Control in IP/TCP Internetworks," RFC 896, January 1984.

[Rizzo 1997] L. Rizzo, "Dummynet: A Simple Approach to the Evaluation of Network Protocols," ACM Computer Communications Review, 27(1), pp. 31–41, January 1997.

[SCTP] Stream Control Transmission Protocol Home Page, www.sctp.org

[Shakkottai 2004] S. Shakkottai, R. Srikant, A. Broido, K. Claffy, "The RTT distribution of TCP Flows in the Internet and its Impact on TCP-based flow control," Technical Report, Cooperative Association for Internet Data Analysis (CAIDA), February, 2004.

[Sivakumar 2000] H. Sivakumar, S. Bailey, R. Grossman, "PSockets: The Case for Application-level Network Striping for Data Intensive Applications using High Speed Wide Area Networks," High-Performance Network and Computing Conference, Dallas, USA, November 2000.

[Squid] The Squid Web Cache, www.squid-cache.org.

[Stewart 2008a] R. Stewart, P. Lei, M. Tuexen, "Stream Control Transmission Protocol (SCTP) Stream Reset," draft-stewart-tsvwg-sctpstrrst, IETF Internet Draft (in progress), December 2008.

[Stewart 2008b] R. Stewart, Q. Xie, L. Yarrol, K. Poon, M. Tuexen, "Sockets API Extensions for Stream Control Transmission Protocol (SCTP)," draft-ietf-tsvwg-sctpsocket, IETF Internet Draft (in progress), January 2009.

[Tarahaat] Tarahaat Home Page, October 2007. www.tarahaat.com/tara/home.

[Tullimas 2008] S. Tullimas, T. Nguyen, R. Edgecomb, S. Cheung, "Multimedia streaming using multiple TCP connections," ACM Transactions in Multimedia Computing Communications and Applications, 4(2), pp. 1–20, 2008.

[VSAT-systems] VSAT Internet Service Provider, October 2007. www.vsat-systems.com.

[Wang 1998] Z. Wang, P. Cao, "Persistent Connection Behavior of Popular Browsers," Research Note, December 1998. www.cs.wisc.edu/~cao/papers/persistent-connection.html

[Wang 2007a] G. Wang, Y. Xia, D. Harrison, "An ns-2 TCP Evaluation Tool: Installation Guide and Tutorial," April 2007. http://labs.nec.com.cn/tcpeval.htm.

[Wang 2007b] G. Wang, Y. Xia, D. Harrison, "An NS2 TCP Evaluation Tool," draft-irtf-tmrg-ns2-tcp-tool, IETF Internet Draft (expired), November 2007.

[Weigle 2006] M. C. Weigle, P. Adurthi, F. Hern´andez-Campos, K. Jeffay, and F. D. Smith, "Tmix: a Tool for Generating Realistic TCP Application Workloads in ns-2," ACM Computer Communication Review, 36(3), pp. 65–76, July 2006.

[WiderNet] WiderNet Project Home Page, October 2007.www.widernet.org.

[Williams 2005] A. Williams, M. Arlitt, C. Williamson, K. Barker, "Web Workload Characterization: Ten Years Later," Book Chapter in "Web Content Delivery," Springer, 2005. ISBN: 0-387-24356-9.

[Williamson 2003] C. Williamson, N. Markatchev, "Network-Level Impacts on User-Level Web Performance. International Symposium on Performance Evaluation of Computer and Telecommunication Systems, Montreal, Canada, July 2003.

[Zhang 2000] Y. Zhang, V. Paxson, S. Shenker, "The Stationarity of Internet Path Properties: Routing, Loss, and Throughput," ACIRI Technical Report, May 2000.