NUI MAYNOOTH

Ollscoil na hÉireann Má Nuad

An Experimental Investigation of TCP Performance in High Bandwidth-Delay Product Paths

Baruch Even,
Hamilton Institute,
baruch@ev-en.org

Supervisor: Professor Douglas Leith
Department of Computer Science
Faculty of Science
National University of Ireland, Maynooth
Maynooth, Ireland

February, 2007

This thesis is dedicated to my parents who made my education priority and to my wife who made it a pleasure.

ACKNOWLEDGMENTS

I would like to acknowledge those who helped make this thesis transition from thought to action:

First and foremost, My supervisor Prof. Douglas Leith whose guidance helped me steer in the right direction and find the way on the long journey, and whose insight helped me tackle some seemingly dead-ends.

Prof. Robert Shorten for his help during the research and the writing stages.

Prof. Barak Pearlmutter for watering the seed of thought of going for a Masters degree and his help to make this effort bear fruits.

Rosemary Hunt and Kate Moriarty for all their administrative help that let me focus on my work rather than form filings.

And also my friends at the Hamilton Institute who made my stay in Ireland much more enjoyable: Anthony, Carlos, David, Eric, Florian, Gavin, Ian, Mehmet, Parisa, Peter, Rade, Santiago, Selim, Steven and Tianji.

This work was supported by Science Foundation Ireland grants 00/PI.1/C067 and 04/IN3/I460.

Contents

1 Introduction **1**

1.1 Introductory Remarks 1

1.2 Structure of Thesis 2

1.3 Contribution of thesis 2

2 An Overview of Congestion Avoidance in TCP **4**

2.1 TCP Overview 4

2.2 TCP Variants 9

2.3 Additional TCP Details 13

2.4 High speed protocols 16

2.4.1 Scalable-TCP [13] 17

2.4.2 HS-TCP [8] 18

2.4.3 H-TCP [14] 18

2.4.4 BIC-TCP [26] 19

2.4.5 FAST-TCP [12] 20

2.5 Summary 20

3 Network Stack **21**

3.1 Introduction 21

3.2 Slow-path processing 22

3.3 Test Setup 23

3.4 Baseline 25
3.5 Proposed fixes 27
3.5.1 Walk SACKed Holes List 27
3.5.2 Retransmit hints 30
3.5.3 SACK hints 35
3.6 Conclusions 35

4 The cost of aggressive window growth **38**
4.1 Introduction 38
4.2 Properties of standard TCP flows 39
4.2.1 A model of a network of TCP flows 40
4.2.2 Properties of standard TCP 45
4.3 Test setup 48
4.4 Properties of H-TCP 49
4.4.1 Long-term λ Unfairness 50
4.4.2 Short-term Unfairness 51
4.5 Summary 53

5 Summary and Conclusions **61**

Bibliography **62**

List of Figures

2.1 Exponential Growth of TCP Slow Start 7

2.2 TCP Flow States 8

2.3 Tahoe TCP 11

2.4 Reno TCP 11

2.5 TCP Packet Format 14

2.6 TCP sliding window 16

2.7 *cwnd* history of a standard TCP flow on a network with 500Mbps bandwidth and 220ms RTT 17

3.1 Network Topology 23

3.2 Processing stages of an incoming packet and location of the receive queue 26

3.3 Overall performance in Mbit/s for a specified line rate. Before the changes and after. 26

3.4 Baseline Kernel at 950Mbps, 220ms rtt and 20% queue size 28

3.5 TCP performance with SACK hole list modification at 950Mbps, 220ms rtt and 20% queue size 32

3.6 TCP performance with SACK Holes and Retransmit Hints modifications at 950Mbps, 220ms rtt and 20% queue size 34

3.7 With SACK Holes, Retransmit Hints and SACK Hints at 950Mbps, 220ms rtt and 20% queue size 37

4.1 Evolution of window size 40

4.2 Average time between congestion events for standard TCP for one flow as the bottleneck link bandwidth is varied. Measurements taken from experimental testbed, RTT is 220ms, router queue is sized at 20% of the bandwidth-delay product 46

4.3 Throughput ratios with λ unfairness for standard TCP with bandwidths of 50Mbps, 100Mbps and 300Mbps. Network with 10 flows in total, 5 flows with $\lambda = 1$ and 5 flows with a λ ranging between 1 and 0.145. Measurements taken from experimental testbed, RTT is 220ms, router queue sized at 20% of bandwidth-delay product 48

4.4 Distribution of peak window sizes for standard TCP at 50Mbps and 100Mbps. Measurements taken from experimental testbed, network with 10 flows, RTT is 220ms, router queue sized at 20% of bandwidth-delay product 54

4.5 Average time between congestion events for H-TCP for one flow as the bottleneck link bandwidth is varied. Measurements taken from experimental testbed, RTT is 220ms, router queue is sized at 20% of the bandwidth-delay product. 55

4.6 Throughput ratios with λ unfairness for H-TCP with bandwidths of 100Mbps and 300Mbps. Network with 10 flows in total, 5 flows with $\lambda = 1$ and 5 flows with a λ ranging between 1 and 0.145. Measurements taken from experimental testbed, RTT is 220ms, router queue sized at 20% of bandwidth-delay product. 55

4.7 H-TCP $alpha$ and $cwnd$ evolution for 15 seconds from congestion event 56

4.8 Example evolution of flow cwnds. Network with 10 flows in total, five flows with $\lambda = 0.5$ and five with $\lambda = 1$. Measurements taken from experimental testbed, RTT is 220ms, router queue sized at 20% of bandwidth-delay product. 56

4.9 Actual versus demanded synchronisation rate as number of flows is varied. Measurements taken from experimental testbed, RTT is 220ms, bandwidth 300Mbps, router queue sized at 20% of bandwidth-delay product. 57

4.10 $cwnd$ distribution for different number of flows. Measurements taken from experimental testbed, RTT is 220ms, bandwidth 300Mbps, router queue size at 20% of bandwidth-delay product. 57

4.11 Time between congestion events for different number of flows. Measurements taken from experimental testbed, RTT is 220ms, bandwidth 300Mbps, router queue size at 20% of bandwidth-delay product. . . . 58

4.12 Impact of BDP on aggressiveness of H-TCP flows, 5 flows on each host all with 220ms RTT and bandwidth of 100 or 300 Mbps 59

4.13 Impact of backoff factor at 300Mbps, 220ms RTT with $\lambda = 0.5$ 60

List of Tables

3.1 Hardware and Software Configuration. 24

Chapter 1

Introduction

1.1 Introductory Remarks

The performance of the Internet is determined not only by the network and hardware technologies that underlie it, but also by the software protocols that govern its use. In particular, the TCP transport protocol is responsible for carrying the great majority of traffic in the current internet, including web traffic, email, file transfers, music and video downloads. TCP provides two main functions. First, it provides functionality to detect and retransmit packets lost during a transfer thereby providing a reliable transport service to higher layer applications. Second, it enforces congestion control. That is, it seeks to match the rate at which packets are injected into the network to the available network capacity. A particular aim here is to avoid so-called congestion collapse, prevalent in the late 1980s prior to the inclusion of congestion control functionality in TCP.

Over the last decade or so, the link speeds within networks have increased by several orders of magnitude. While the TCP congestion control algorithm has proved remarkably successful, it is now recognised that its performance is poor on paths with high bandwidth-delay product, e.g. see [13, 8, 14, 26, 12] and references therein. With the increasing prevalence of high speed links, this issue is becoming of widespread concern. This is reflected, for example, in the fact that the Linux operating system now employs an experimental algorithm called BIC-TCP[26] while Microsoft are actively studying new algorithms such as Compound-TCP[25].

While a number of proposals have been made to modify the TCP congestion control algorithm, all of these are still experimental and pending evaluation as they change the

congestion control in new and significant ways and their effects on the network are not well understood. In fact, the basic properties of networks employing these algorithms may be very different to networks of standard TCP flows. The aim of this thesis is to address, in part, this basic observation.

1.2 Structure of Thesis

This thesis is organised as follows. In chapter 2 we give an overview of congestion control mechanisms in the TCP protocol. In chapter 3 we construct and test an experimental testbed for investigating the performance of high-speed TCP algorithms. A key issue here is that the computational burden created by the standard Linux network stack (similar comments also apply to other operating systems) is too great to support proper TCP operation at speeds above about 100Mb/s. Before we can study the behaviour of the TCP congestion control algorithm, it is therefore necessary to develop a more efficient network stack implementation. In chapter 4 we use the developed testbed to study in detail the performance of the H-TCP high-speed algorithm in unsynchronised network conditions. A summary and conclusion are presented in chapter 5.

1.3 Contribution of thesis

The contribution of this thesis includes the following:

(i) We document the performance degradation of the standard Linux network stack on high bandwidth-delay product paths. By careful instrumentation of the network stack it is established that this degradation is primarily associated with the excessive computation burden imposed by the standard SACK processing algorithm. A modified SACK processing implementation is developed and its performance validated on commodity hardware for paths with delay up to 220ms and bandwidth up to 1Gbs (corresponding to a maximum bandwidth-delay product of approximately 20000 packets).

(ii) Based on this modified Linux kernel an instrumented testbed network is developed. To allow controlled study of the impact of synchronisation rate on behaviour, a modified FreeBSD dummynet implementation is also developed.

(iii) Using the developed testbed network we investigate the performance of both standard TCP and H-TCP in unsynchronised conditions. We demonstrate that the unfairness in long-term average throughput between flows with different synchronisation rates is amplified by the more aggressive increase rate of the H-TCP algorithm. Large short-term fluctuations in the rate of a flow are often a feature of networks in which high-speed protocols are deployed, leading to short-term unfairness between competing flows. The distribution of rate variation depends, amongst other things, on the network backoff factors, with larger reducing short-term unfairness but reducing the responsiveness of the network.

In terms of publications, the testbed development work in this thesis was reported at the Terena Networking Conference, 2005 in a presentation entitled *Fair & useful comparisons: how should we evaluate new TCP proposals ?* and in a paper *Evaluating the performance of TCP stacks for high-speed networks* that appeared in the Proceeding of the Workshop on Protocols for Fast Long Distance Networks (PFLDnet), 2006.

Chapter 2

An Overview of Congestion Avoidance in TCP

In this chapter we describe the features of TCP relevant to congestion control. This chapter is organised as follows. Section 2.1 provides some background information on TCP and describes the features that can exist in a typical TCP flow. In section 2.2 we look at the main TCP variants. Section 2.3 provides additional details of TCP that are relevant to our discussion. Section 2.4 reviews recent proposals for changes to the TCP congestion control algorithm to improve performance in high bandwidth-delay product networks. Finally section 2.5 summarises the contents of this chapter.

2.1 TCP Overview

TCP is one part of two well known protocol standards commonly referred to as TCP/IP. TCP sits on top of the IP layer and passes segments onto the IP layer for further processing. These segments are then passed onto the lower level layers and eventually onto the network. TCP was officially adopted as a standard in RFC[1] 793 [19] in 1981 and was designed to deal with message flow control and error correction, ensuring reliable delivery of a message from a source application to a destination application. IP was also officially adopted as a standard in RFC 791 [18] in 1981. IP deals with logical addressing and specifies source and destination addresses. These addresses are used to

[1]The Requests for Comments (RFC) document series is a set of technical and organizational notes pertaining to the Internet. These documents are maintained by the Internet Engineering Task Force (IETF).

route a message to its destination and to provide a return address for any response.

The origins of TCP/IP stem from DARPA research into resilient networks for use in a battlefield environment [5]. The goal of this research was to design a protocol suite that could cope with network link failure and ensure delivery of data to its destination. As TCP/IP evolved it moved from the research environment to be deployed in isolated networks that were eventually interconnected to become what we now know as the Internet[5].

TCP is a bi-directional, reliable, end-to-end protocol for controlling data transmission. TCP sources break messages from higher protocol layers into datagrams that are encapsulated in packets which are then transmitted over the network. These packets are reassembled by the TCP receiver into the original message and passed onto the higher level protocol layers. For every packet sent on the network by a source an acknowledgement (ACK) is expected to be transmitted back from the destination. This ACK (or lack thereof) is used by the source to determine if the acknowledged packet was successfully received at the destination. In this manner packets can be tracked and retransmitted if required.

To facilitate further discussion of TCP, the general features of TCP will be highlighted. We do not describe a specific TCP variant here but provide an overview of features a TCP variant can possess. These features will be dealt with in an historical and chronological manner. Refer to [6] for a more detailed description of the TCP protocol. In order to describe the TCP protocol some concepts are needed:

- Round Trip Time (RTT): the time taken for a packet to be sent from a source to a destination and for the corresponding acknowledgement to be received by the source, assuming no packet loss.
- Advertised window (*wnd*): the amount of data a destination has advertised that it is willing to receive. This is reported in every ACK sent by the destination to the source[2].
- Source congestion window (*cwnd*): the number of packets in flight i.e. transmitted but not yet acknowledged, assuming the source is not restricted by the advertised window.
- Send window (*swnd*): the minimum of *wnd* and *cwnd*.

[2]For ease of understanding, all window sizes are described in terms of packets in this thesis. In a real TCP implementation these window sizes can be in terms of bytes.

- Additive Increase Multiplicative Decrease (AIMD): TCP increases its congestion window by adding to it on receipt of ACK's and decreases the send window by a multiplicative factor on receipt of an indication of packet loss.
- Slow Start: In Slow Start mode the congestion window is doubled every RTT which leads to an exponential rate of increase.
- Slow start threshold (*ssthresh*): the threshold in packets below which the source remains in slow start mode.
- Fast Retransmit: a mechanism whereby the source retransmits a packet after receiving a number of duplicate ACK's (normally three) rather than waiting for a retransmit timer to timeout.
- Fast Recovery: the mode entered after a packet drop is detected via Fast Retransmit. In this mode the congestion window is restricted in value until the dropped packet is successfully retransmitted.
- Retransmission Time-Out (RTO): the interval a source waits without receiving an ACK before marking a packet as lost, retransmitting it and entering Slow Start mode. TCP calculates RTO based on current RTT and RTT variance.
- Congestion Avoidance: the mode the source enters after Fast Recovery. The source uses an AIMD strategy, linearly increasing its congestion window at a rate of one packet per RTT.
- TCP state machine: TCP is state based and maintains a local data structure to track the state of a connection. States include CLOSED, ESTABLISHED, LISTEN, and other intermediary states.
- TCP Control Block (TCB): an internal data structure that holds TCP state information and internal variables.
- Maximum Sized Segment (MSS): the maximum packet size that can be sent by a TCP source.

During the initial phase of a TCP connection the receiver (or receivers when the data flow is bi-directional) provides details of the amount of incoming data that it can process (*wnd*). This informs the source of the maximum data that the destination is currently willing to receive. In early implementations of TCP, sources transmitted data in a burst with further bursts on receipt of change in the advertised window size. This

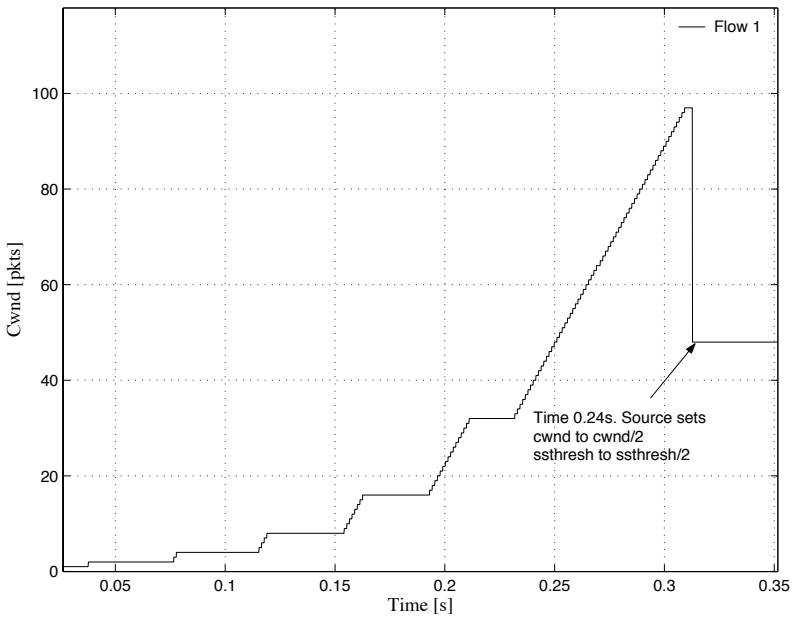


Figure 2.1: Exponential Growth of TCP Slow Start

caused problems as the source tended to overwhelm the network. To address this issue Jacobson et al [11] proposed a series of measures.

Jacobson et al [11] proposed Slow Start mode. In this mode the congestion window increases on the receipt of an ACK according to the following formula

$$cwnd_{(n+1)} = cwnd_{(n)} + 1 \tag{2.1}$$

where n indexes the ACK's received. Assuming the advertised window is greater than the current *cwnd*, the source will send as many packets onto the network as allowed by its *cwnd*. The source will transmit new packets every time a new ACK is received and increments *cwnd* according to (2.1). As *cwnd* packets are sent in an RTT, the congestion window increases by *cwnd* packets per RTT. This leads to a doubling of the amount of packets sent in the next RTT causing exponential growth of *cwnd*, see for example Figure 2.1. The source increases its send window in this manner until one of two conditions is met:

1. *ssthresh* is reached. In this case the source transitions to Congestion Avoidance mode and will continue in this mode until condition 2 is met.

2. Packet loss is detected by the source. In this case the source reduces its *cwnd* by

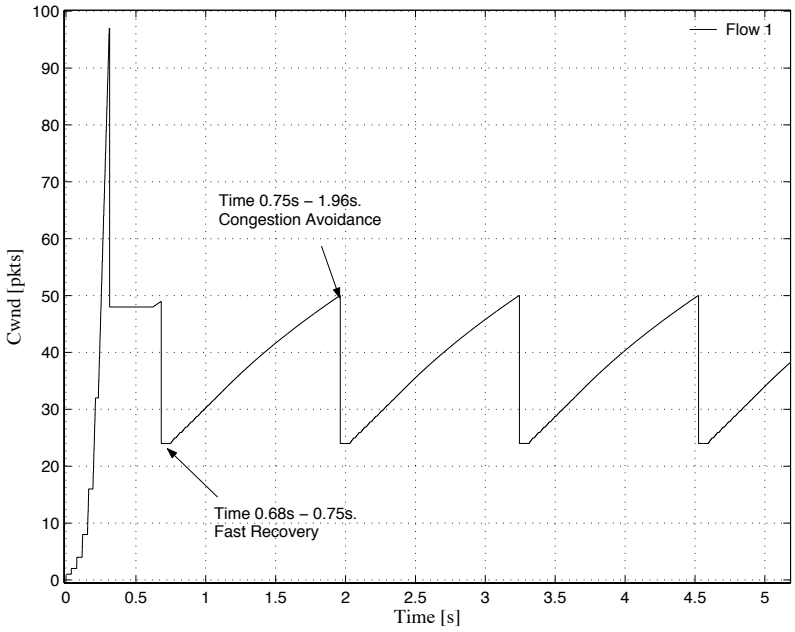


Figure 2.2: TCP Flow States

half and transitions to Fast Recovery mode.

Packet loss was originally detected at the source by the expiry of a retransmission timer. This timer is set based on current network conditions and is tied to variations in RTT. See [6] [11] for more details. This was found to be an inefficient mechanism for detection of packet loss and the Fast Retransmit mechanism was proposed to address this issue [11]. With Fast Retransmit a packet is retransmitted if three duplicate ACK's are received. This occurs as the destination only ACK's the last packet in sequence that it has received. If the destination receives packets out of order, due to a packet being dropped, it will send duplicate ACK's for the last packet received in order. The receipt of three of these duplicate ACK's is an indication of a dropped packet as detailed in RFC 2001 [24]. The source then sets *ssthresh* and *cwnd* to half its current value of *cwnd*. See, for example, time period 0.31s in Figure 2.1.

Following Fast Retransmit the source enters into Fast Recovery mode. The dropped packet(s) are retransmitted and the source remains in Fast Recovery mode until the retransmitted packet is acknowledged by the destination. See time period 0.68s - 0.75s in Figure 2.2.

Following successful retransmission of lost packet(s)the source then enters into Con-

gestion Avoidance mode. The congestion window increases according to the following formula

$$cwnd_{(n+1)} = cwnd_{(n)} + 1/cwnd_{(n)} \tag{2.2}$$

As with Slow Start the source transmits a new packet every time an ACK is received but in this mode *cwnd* is only increased by $1/cwnd$. As *cwnd* packets are sent in an RTT, *cwnd* increases by one packet per RTT. This leads to a near linear increase in *cwnd* with a slope of 1[3]. See time period 0.75s - 1.96s in Figure 2.2. The source remains in Congestion Avoidance until a packet drop is detected. In this way the TCP flow will cycle between Fast recovery and Congestion Avoidance until the end of the flow unless there is a retransmission timeout for a packet, in which case the TCP flow control will reset *cwnd* to one and restart the flow in Slow Start mode.

As the TCP source send rate is clocked by incoming ACK's the source can react to prevalent network conditions. This feedback control mechanism, called the TCP self clocking mechanism, leads to a situation whereby the source increases its congestion window at a slower rate under heavy network load than under light network load.

2.2 TCP Variants

In order to understand the current status of TCP it is important to look at its development and in particular the reasoning behind specific design features. Early implementations of TCP used a go-back-n model (send sequence goes back n packets) when packets were lost. These implementations had no congestion control and led to a series of 'congestion collapses' on the Internet. During these congestion collapses the data throughput of connections was severely reduced due to excessive retransmission of packets. These issues were addressed by a version of TCP called Tahoe [11] in which the problem of congestion was approached by a 'Conservation of Packets' principle whereby new packets were not put into the network until the old ones left. Tahoe is thus a self clocking system, formed by the transmission of data and the receipt of acknowledgements.

Tahoe offered a means of combating congestion through dynamically altering the size of the protocol's send window. As can be seen in Figure 2.3 the algorithm follows

[3]The increase is linear in RTT but not in time as the RTT will vary over time as network queues fill and empty.

these simple rules:

1. As an initial condition or if the RTO expires, set *cwnd* to one.
2. In Slow Start increase *cwnd* by one packet for each ACK until *ssthresh* is reached.
3. On reaching *ssthresh* enter Congestion Avoidance mode.
4. In Congestion Avoidance increase *cwnd* by $1/cwnd$ for each ACK received.
5. On detection of a packet loss, *cwnd* is reset to one, Slow Start is entered and *ssthresh* is set to half its current value.

Congestion Avoidance and Slow Start are independent algorithms with different objectives but in practice they are implemented together. Slow Start probes the network so that the TCP source can get an initial indication of the network bandwidth available. Congestion Avoidance more gently probes the network so that the TCP source can adapt to changing network conditions. A TCP connection will start in Slow Start mode but switch to Congestion Avoidance mode after *cwnd* reaches the value of *ssthresh*. In addition to these enhancements Tahoe also includes Fast Retransmit, better RTT variance estimation, and exponential retransmit timer back-off. These enhancements dramatically enhanced the throughput performance of TCP [11].

Historically, the next major variant of TCP is called Reno TCP [24]. This variant of TCP is similar to the Tahoe TCP, except it also includes Fast Recovery [24]. Reno TCP does not return to Slow Start after Fast Recovery (which ends on the receipt of the retransmitted packet), instead it reduces the congestion window to half the current window size as can be seen in Figure 2.4. Note that in this example the TCP flow goes into Timeout mode following Slow Start due to excessive packet transmission during Slow Start. Reno also includes delayed ACKs which will be discussed in the next section.

TCP Tahoe and Reno experience poor performance when multiple packets are lost from one window (*cwnd*) of data[4]. With the limited information available from cumulative acknowledgments, a TCP source can only learn about a single lost packet per round trip time. An aggressive source could choose to retransmit packets early, but such retransmitted packets may have already been successfully received. TCP NewReno [9] address this issue by modifying the action taken when receiving new

[4]Note that *cwnd* represents the number of packets in flight for the flow until a packet(s) is dropped.

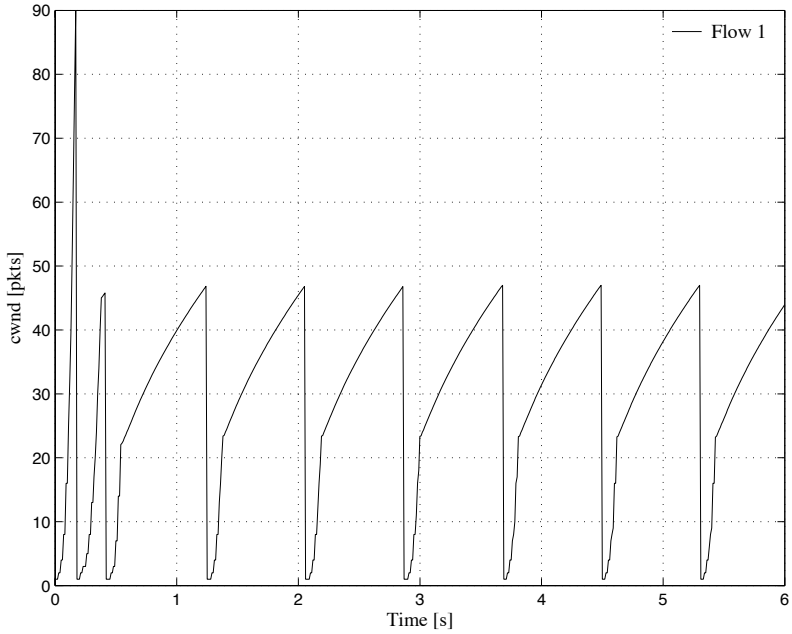


Figure 2.3: Tahoe TCP

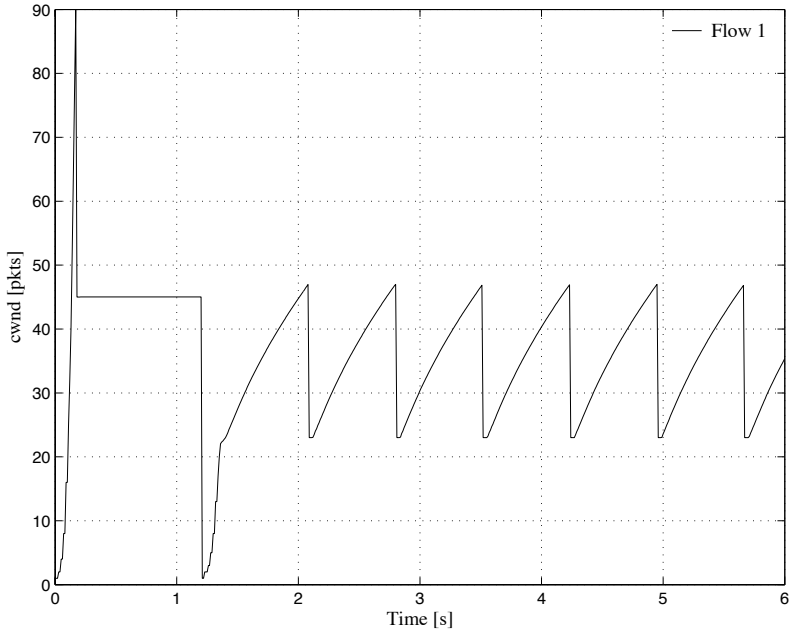


Figure 2.4: Reno TCP

ACK's. In order to exit Fast Recovery, the source must receive an ACK for the highest sequence number sent before entering Fast Recovery. Thus, unlike TCP Reno, new "partial ACK's" (those which represent new ACK's but do not represent an ACK of all outstanding data) do not take TCP NewReno out of Fast Recovery. In this way, Reno retransmits one packet per RTT until all lost packets are retransmitted.

Although TCP NewReno addresses the issue of multiple drops within a window of data, it does not use all the information on dropped packets available at the receiver. It takes a full round trip for a sender to learn about each lost packet. When multiple segments are lost this inefficiency can reduce throughput substantially. This is of particular importance on high bandwidth-delay product paths as many of the proposed TCP algorithms for such paths are more aggressive than standard TCP and frequently generate multiple packet losses during normal operation. Improving performance with multiple drops is addressed by the Selective Acknowledgment (SACK) mechanism [15], combined with a selective repeat retransmission policy.

SACK is an extension to TCP intended to improve TCPs handling of the multiple packet loss case. When the sender and receiver support SACK, the receiver transmits TCP ACK packets that include a list of up to four ranges of packet sequence numbers. These ranges specify packets that have been successfully received, and the gaps between these ranges can be used by the sending machine to infer which packets are likely to have been lost and therefore should be retransmitted. This has the benefit of allowing the source to intelligently retransmit packets and react more efficiently to multiple dropped packets.

In more detail, the SACK data in a TCP ACK consists of the start and stop packet sequence numbers of up to four non-overlapping blocks of packets. The SACK blocks are not necessarily listed in sequence order as the first SACK block is required to include the sequence number of the most recently received packet. The receiver is not guaranteed to keep the packet that it SACKed in previous ACKs and it may do so if facing a memory pressure. The sender can't drop the packets that were SACKed since it might need to retransmit them anyway if the receiver did drop the packets eventually.

Often the TCP packet includes the time-stamp option for sequence wrapping detection, in which case the ACK packet has space for only three SACK blocks.

2.3 Additional TCP Details

In this section we briefly describe some additional features of TCP relevant to this thesis. To start with we look at the TCP packet format which is shown in Figure 2.5. The TCP packet comprises the following fields:

- Source Port and Destination Port. Identifies points at which upper-layer source and destination applications receive TCP services.
- Sequence Number. Specifies the number assigned to the first byte of data in the current message. In the TCP handshake phase, this field also can be used to identify an initial sequence number to be used in an upcoming transmission.
- Acknowledgment Number. Contains the sequence number of the next byte of data the source of the packet expects to receive.
- Length. Indicates the number of 32-bit words in the TCP header.
- Unused. Reserved for future use.
- Flags. Carries a variety of control information, including the SYN and ACK bits used for connection establishment, and the FIN bit used for connection termination.
- Window Size. Specifies the size of the source's receive window (that is, the buffer space available at the destination for incoming data).
- Checksum. Used to determine whether the header was damaged in transit.
- Urgent Data Pointer. Points to the first urgent data byte in the packet.
- Options. Specifies various TCP options such as the MSS, the window scale value and the time the packet was sent. Also used by SACK to inform the source of non-contiguous blocks of data that have been received in order to allow retransmission of only the lost segments.
- Payload. Contains upper-layer information

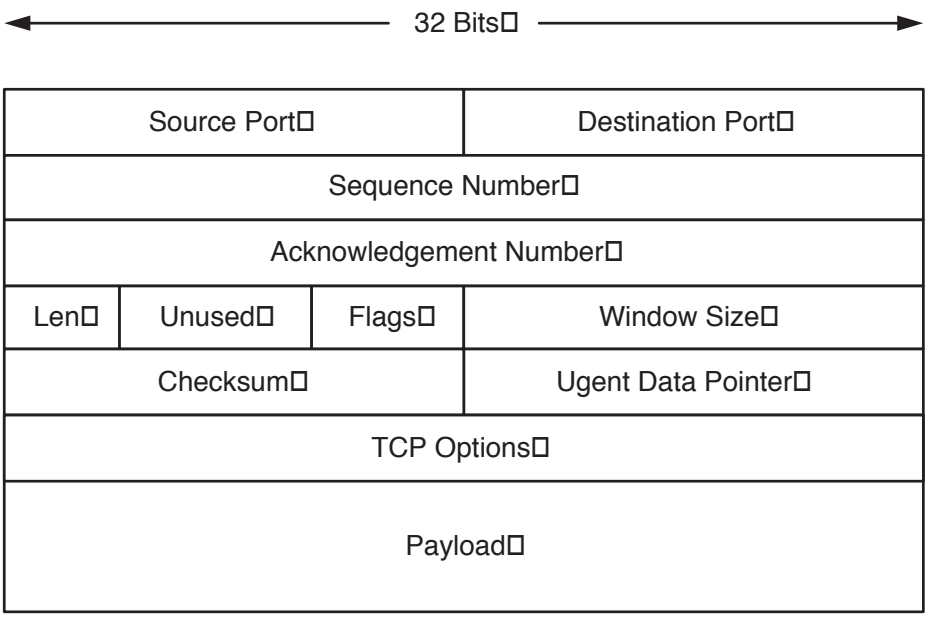


Figure 2.5: TCP Packet Format

TCP maintains an internal state machine to track the state of a TCP connection. This information is stored in the TCB and is updated as the connection changes state such as whether a connection is established or closed or in a variety of other states [6]. For example, TCP initiates a three part handshake to establish a connection with a destination. This handshake is used to set up initial conditions for the connection and works in the following way:

- The source sends a synchronisation packet (SYN) to the receiver so that its sequence numbers can be captured by the receiver and used to track incoming packets. The source's state is SYN SENT. The receiver's state is SYN RECVD.
- The receiver sends a SYN ACK packet so that its sequence numbers can be captured by the source and used to track incoming packets. From the receiver's perspective the connection is complete and its state is now ESTABLISHED.
- The source sends an ACK packet and the connection is completed. The source's state is now ESTABLISHED.

The TCB also contains per connection information such as the *cwnd*, *wnd* and the RTT. TCP uses these variables to manage flow control in order to ensure a suitable

amount of data is transmitted onto the network and to reduce packet retransmission to a minimum.

In addition to congestion control the areas addressed by TCP flow control includes receiver overflow. If traffic is sent at too fast a rate for the receiver to process, its buffers may overflow and packets will need to be retransmitted. The receiver buffer size is first advertised during the initial TCP handshake and subsequently updated in every ACK transmitted to the source.

To manage a flow, TCP uses a sliding window mechanism to control the sending of packets as shown in Figure 2.6. The send window (*swnd*) slides over the data stream as ACK's are received or *wnd* changes and TCP thereby uses the send window to control the amount of packets that can be in transit in the network.

Another characteristic of TCP worthy of note deals with the unnecessary transmission of packets both from the source and the destination that affected early versions of TCP. This phenomenon known as 'Silly Window Syndrome' causes inefficient usage of the network and impacts the source and destination as they have to process a large number of small packets. On the receiver side two mechanisms are used to address this issue. The first is used after advertising a zero window. The receiver waits to transmit an ACK until a minimum of half of the receiver's buffer becomes free or it has MSS bytes ready to transmit (in this case the receiver is also a source). With the second mechanism the receiver delays sending new advertised windows until two packets have been received or for a certain time (normally 200ms). For further details see [4].

On the source side, the Nagle algorithm [16] dictates when to transmit a packet. If the source has less than one MSS of data to send, a new packet will not be sent if there is any outstanding unacknowledged data. If the data to be sent is greater than one MSS, it is sent immediately. This algorithm deals with situations where applications are sending data at either slow or fast rates. When an application is sending at a slow rate data is 'clumped' and transmitted when required. This situation typically happens with interactive applications such as telnet. For applications with a requirement for faster data rates, the data is transferred in full MSS segments. This typically occurs in an application such as FTP where large data transfers are required. For more detail on both these mechanisms see [6].

The features above affect TCP network congestion analysis. In particular, for fast data rate applications, the receiver side changes cause one ACK to be sent for every two packets sent. This means that the update rule for *cwnd*, (2.1) and (2.2), are only called

[6]This diagram is based on a paper on TCP performance by Geoff Huston [10]

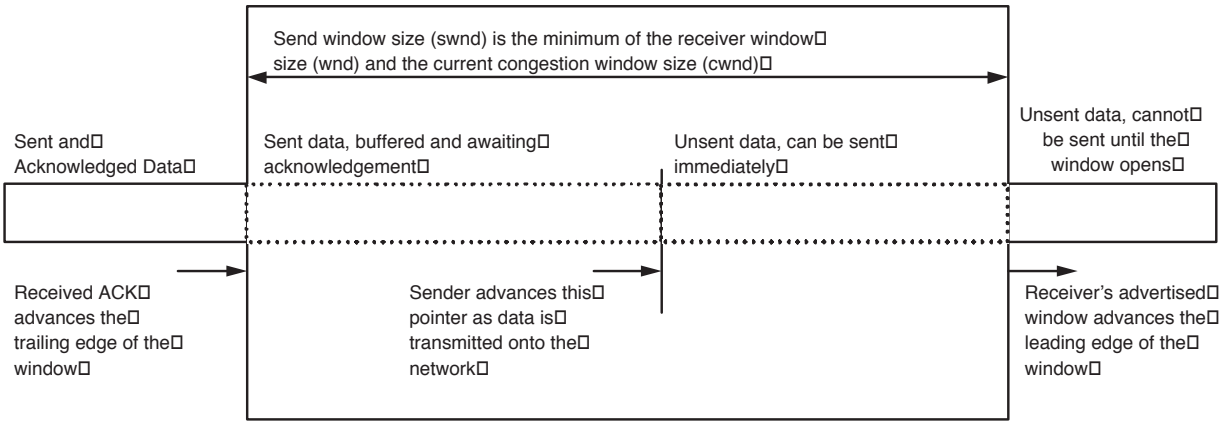


Figure 2.6: TCP sliding window[6]

at half the rate of the situation where the ACK's are not delayed. Turning this feature on or off for a TCP flow can have a large effect on its performance.

2.4 High speed protocols

A basic design property limits the scalability of the existing TCP congestion control algorithm on bandwidths with a high Bandwidth Delay Product (BDP). Namely, the linear rate of increase whereby the congestion window is increased by one packet per round-trip time means that following backoff of the congestion window it can take a substantial period of time before the window recovers. For example, this can take about 20 minutes for a network with bandwidth of 500Mbps and 220ms RTT as can be seen in Figure 2.7 on the following page. With the increasing prevalence of high speed links, this issue is becoming of widespread concern. This is reflected, for example, in the fact that the Linux operating system now employs an experimental algorithm called BIC-TCP[26] while Microsoft are actively studying new algorithms such as Compound-TCP[25].

A number of proposals have been made to modify the TCP congestion control algorithm to improve performance in high bandwidth-delay product paths. All of these are still experimental and pending evaluation as they change the congestion control in new and significant ways and their effects on the network are not well understood. Here we give a brief overview of several proposed modifications that have received considerable attention over the last few years.

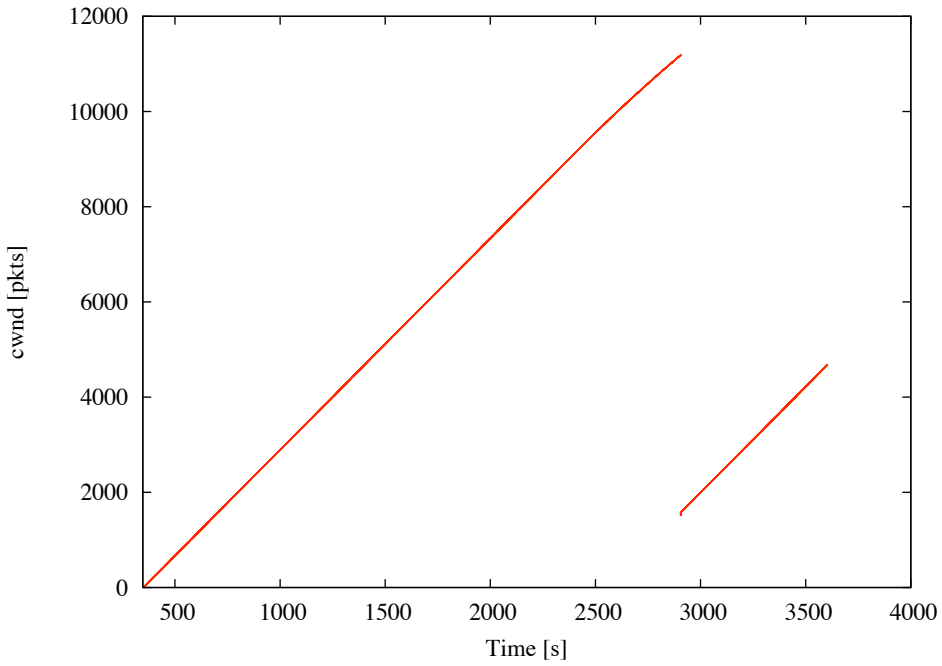


Figure 2.7: *cwnd* history of a standard TCP flow on a network with 500Mbps bandwidth and 220ms RTT

2.4.1 Scalable-TCP [13]

The basic idea in Scalable-TCP is to make the recovery time after a congestion event independent of window size. Specifically, Scalable-TCP proposes that the TCP $cwnd$ be updated as follows

$$\begin{aligned} \text{Ack:} \quad & cwnd \leftarrow cwnd + \alpha \\ \text{Loss:} \quad & cwnd \leftarrow \beta \times cwnd \end{aligned}$$

Suggested values for the parameters α and β are 0.01 and 0.875, respectively. A mode switch is used whereby the standard TCP $cwnd$ update rules are used when $cwnd$ is less than a threshold, Low_Window, and the Scalable-TCP update rules are used for larger $cwnd$ values.

2.4.2 HS-TCP [8]

HS-TCP uses the current TCP $cwnd$ value as an indication of the bandwidth-delay product on a path. The AIMD increase and decrease parameters are then varied as functions of $cwnd$:

$$\begin{array}{rrl} \text{Ack:} & cwnd & \leftarrow cwnd + \dfrac{f_\alpha(cwnd)}{cwnd} \\ \text{Loss:} & cwnd & \leftarrow g_\beta(cwnd) \times cwnd \end{array}$$

In [8] logarithmic functions are proposed for $f_\alpha(cwnd)$ and $g_\beta(cwnd)$, whereby $f_\alpha(cwnd)$ increases with $cwnd$ and $g_\beta(cwnd)$ decreases. Similarly to Scalable-TCP, HS-TCP uses a mode switch so that the standard TCP update rules are used when $cwnd$ is below a specified threshold.

2.4.3 H-TCP [14]

HTCP uses the elapsed time Δ since the last congestion event, rather than $cwnd$, to indicate path bandwidth-delay product and the AIMD increase parameter is varied as a function of Δ. The AIMD increase parameter is also scaled with path round-trip time to mitigate unfairness between competing flows with different round-trip times. The AIMD decrease factor is adjusted to improve link utilisation based on an estimate of the queue provisioning on a path. In more detail,

$$\begin{array}{rrl} \text{Ack:} & cwnd & \leftarrow cwnd + \dfrac{2(1-\beta)f_\alpha(\Delta)}{cwnd} \\ \text{Loss:} & cwnd & \leftarrow g_\beta(B) \times cwnd \end{array}$$

with

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

$$g_\beta(B) = \begin{cases} 0.5 & |\frac{B(k+1)-B(k)}{B(k)}| > \Delta_B \\ min(\frac{T_{min}}{T_{max}}, 0.8) & otherwise \end{cases}$$

where Δ_L is a specified threshold such that the standard TCP update algorithm is used while $\Delta \le \Delta_L$. A quadratic increase function $\bar{f}_\alpha$ is suggested in [14], namely $\bar{f}_\alpha(\Delta) = 1 + 10(\Delta - \Delta_L) + 0.25(\Delta - \Delta_L)^2$. T_{min} and T_{max} are measurements of

the minimum and maximum round-trip time experienced by a flow. $B(k+1)$ is a measurement of the maximum achieved throughput during the last congestion epoch.

2.4.4 BIC-TCP [26]

BIC-TCP employs a form of binary search algorithm to update $cwnd$. Briefly, a variable w_1 is maintained that holds a value halfway between the values of $cwnd$ just before and just after the last loss event. The $cwnd$ update rule seeks to rapidly increase $cwnd$ when it is beyond a specified distance S_{max} from w_1, and update $cwnd$ more slowly when its value is close to w_1. Multiplicative backoff of $cwnd$ is used on detecting packet loss, with a suggested backoff factor β of 0.8. In more detail,

$$\text{Ack:} \quad \begin{cases} \delta = (w_1 - cwnd)/B \\ cwnd \leftarrow cwnd + \frac{f_\alpha(\delta, cwnd)}{cwnd} \end{cases}$$

$$\text{Loss:} \quad \begin{cases} w_1 = \begin{cases} \frac{1+\beta}{2} \times cwnd & cwnd < w_1 \\ cwnd & otherwise \end{cases} \\ w_2 = cwnd \\ cwnd \leftarrow \beta \times cwnd \end{cases}$$

with

$$f_\alpha(\delta, cwnd) = \begin{cases} \frac{B}{\sigma} & (\delta \leq 1, cwnd < w_1) \\ & \text{or } (w_1 \leq cwnd < w_1 + B) \\ \delta & 1 < \delta \leq S_{max}, cwnd < w_1 \\ \frac{w_1}{B-1} & B \leq cwnd - w_1 < S_{max}(B-1) \\ S_{max} & otherwise \end{cases}$$

BIC-TCP also implements an algorithm whereby upon low utilisation detection, it increases its window more aggressively. This is controlled with the Low_Util and $Util\_Check$ parameters. In order to maintain backwards compatibility, it uses the standard TCP update parameters when $cwnd$ is below threshold Low_Window.

2.4.5 FAST-TCP [12]

FAST-TCP is a delay based algorithm. In outline,

$$\begin{aligned} \text{Each RTT:} \quad cwnd &\leftarrow [cwnd + \frac{T_{min}}{\bar{T}} cwnd + f_\alpha(B)]/2 \\ \text{Loss:} \quad cwnd &\leftarrow 0.5 \times cwnd \end{aligned}$$

where T_{min} and $\bar{T}$ are the minimum and average observed latencies of the flow respectively. The function $f_\alpha(B)$ depends upon the measured throughput B achieved by the flow: currently, $f_\alpha(B)$ is set to 8, 20 and 200 for achieved throughputs of less than 10Mbit/sec, less than 100Mbit/sec and greater than 1Gbit/sec respectively. (These thresholds are specified by the $sysctl$ entries $(m0a, m0u, m1l)$, $(m1a, m1l, m1u)$ and $(m2a, m1l, m2u)$ respectively). FAST-TCP also includes rate pacing. Note that rate-pacing is a functional change and is thus viewed here as being part of the congestion control algorithm (unlike network stack issues such as efficient SACK processing implementation which fundamentally involve no functional change, only a change in computational burden).

2.5 Summary

In this chapter we provided an overview of TCP congestion control. We noted that there are many variants of TCP. The main variants of TCP in use today are flavours of TCP Reno and TCP Sack. The main difference between these TCP variants lies in the manner in which they deal with lost packet recovery. For this thesis we concentrated on New Reno and Sack. We also discussed some general features of TCP such as the TCP packet format, the TCP state machine, TCP receiver overflow, the TCP sliding window and the 'Silly Window Syndrome' that are relevant to congestion control. Recently proposed changes to improve performance on high bandwidth-delay product paths are reviewed.

Chapter 3

Network Stack

3.1 Introduction

In this paper we focus on the Linux 2.4/2.6 network stack implementation as it is widely used in TCP research for high-speed networks. Specifically, measurements were taken using commodity high-end servers, see Table 3.1 on page 24 for details, and a Linux 2.6.6 kernel modified to include instrumentation of network stack operation and timing. Many of the issues discussed are, however, also relevant to other operating system network stacks. The efficiency of the TCP implementation for high-speed networks has received considerable attention with, for example, widespread support within gigabit-speed (and above) network interface cards for hardware offload to reduce the processing burden within the operating system kernel. However, the bulk of this work has focused on fast path optimisation. While fast path performance is key in situations where packet loss is rare (e.g. server farms), we demonstrate that slow path performance is now the bottleneck in wide-area transfers where the probing action of the AIMD strategy in TCP's congestion control action means that packet loss is an intrinsic feature of normal operation.

We offer a look into the performance issues and offer possible fixes to enable a commodity machine to handle a 1Gbps link with maximum utilisation.

3.2 Slow-path processing

The terms slow-path and fast-path describe two paths in the Linux kernel code to handle received segments. The slow-path is the basic code path which is used to handle all possible cases and is thus encumbered with many conditionals that may reduce performance. The fast-path is a subset of the slow-path that, after verifying some basic conditions, then assumes that it can skip treatment for special cases such as lost or out-of-order packets, thereby reducing the number of conditionals and the amount of code needed to run in that path and reducing the time it takes to process a "normal" packet.

On the sending side, the slow path is executed for TCP ACK packets received following a packet loss. In particular, the slow path is executed for all ACK packets containing SACK information. SACK processing is memory and computationally expensive due to the need to walk the list of packets in flight in order to mark the appropriate packets as SACKed, and additional walks to retransmit segments as needed. This processing has received little attention for performance, most of the effort so far being directed at the fast-path.

In Linux 2.6.6 processing of a TCP ACK packet containing SACK information requires the following main operations:

- SACK marking (`tcp_sacktag_write_queue`) walks the list from the start for each SACK block in the TCP ACK and marks the appropriate packets as SACKed. We have found that in the normal loss case there are frequently several runs of lost packets and thus three SACK blocks to process. Hence, the scoreboard code will routinely walk the segment list three times for each TCP ACK packet.

- Marking head lost packets (`tcp_mark_head_lost`) will mark as lost packets that are not SACKed. To do this it walks the list of packets from the start and marks packets in sack holes as lost.

- Cleaning the receive queue (`tcp_clean_rtx_queue`) walks the list again to remove any and all packets that were acknowledged for each ACK packet.

- Retransmit (`tcp_xmit_retransmit_queue`) walks the whole list of packets in flight for each ACK packet received in recovery mode and tries to resend packets marked as lost and not yet retransmitted. This code then also walks the

list a second time from the start in order to find packets for forward transmits (i.e. transmission of new packets rather than retransmission of previously sent packets) during the recovery stage.

The result of these SACK processing actions can be a walk of up to five times of the scoreboard which is a simple linked list of all packets in flight. When the number of packets in flight is large, as it is in high bandwidth-delay product paths, walking such a list five times places a significant burden not only on CPU resources but also on memory resources as the scoreboard is too large for the CPU cache leading to memory stalls waiting for data.

3.3 Test Setup

In order to study TCP operation, an experimental testbed has been developed. The testbed implements a simple dumbbell topology as shown in Figure 3.1. We recognise that this topology is a limited one, but it does provide a good test-bed for raw performance testing. The machines and software setup is described in Table 3.1 on the next page.

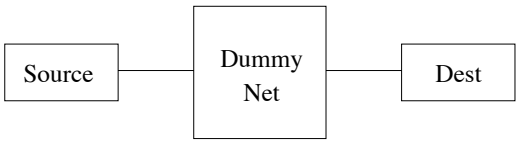


Figure 3.1: Network Topology

A FreeBSD DummyNet router was used to emulate paths with propagation delays of 220ms and bandwidths ranging from 10Mb/s-950Mb/s. The value of 220ms should be representative of trans-continental links and provides for a good test-case, e.g. Ireland-California. The router was configured to have a 20% capacity of the link BDP.

Test durations are chosen long enough to provide consistent measurements of average throughput. We have found that 10 minutes are sufficient to get repeatable results with H-TCP. We have chosen H-TCP as an example high-speed protocol but many of our results apply equally to other algorithms.

To achieve the information needed to understand the dynamics of the software we used OProfile[1] for kernel profiling and a specialised tracing system to log information from within the kernel.

	Description
CPU	Intel Xeon CPU 3GHz
Memory	512 Mbytes
Motherboard	Dell PowerEdge 1850
FreeBSD DummyNet	4.10
Linux Kernel	Linux 2.6.6
txqueuelen	1,000
max_backlog	300
NIC	Intel 82540EM Gigabit Ethernet
NIC Driver	e1000 version 5.2.39-k2
TX & RX Descriptors	4096
Congestion Control Algorithm	H-TCP

Table 3.1: Hardware and Software Configuration.

OProfile is a statistical profiler that uses CPU features to detect various events, the events of interest to us being power and cache misses. The power event is emitted when the CPU is running an instruction, while the cache miss event is emitted for any miss to the L1 CPU cache. Both events give an impression of where the CPU is spending its time and where algorithms might need improvement with regard to their memory access patterns. Each of these events is logged once every configurable number of emits, thus giving a statistical profiling of the code and for a long enough test case this highlights the hot-spots in the code. The data collected includes the EIP processor register which points to the instruction that was executed at the event firing time. At the end of a test OProfile analyses the data collected and provides a report showing the number of events flagged for each function and a breakdown of the events per source line.

In addition to OProfile we used a simple logging system based on RelayFS[27] to transfer fine-grained monitoring data from the kernel to user space for later analysis. The data was logged in binary format and converted into usable text logs after the test finished in order to have the minimal impact on the test. The data logged from the kernel included

- TCP internal variables for the test flows. This includes the congestion window $cwnd$, snd_una and changes in TCP state (e.g. between Congestion Avoidance and Fast Recovery).
- Network stack variables. In particular, the receive queue size.

- SACK processing timing information. High precision time stamp information for entry and exit to the SACK code elements allowed fine-grained measurement of processing time of functions such as `tcp_sacktag_write_queue`. We also logged accounting information on the number of SACKs walked in the scoreboard marking code and on the SACK options received in order to be able to make better informed decisions on caching.

The method used to add the logging code was to have a Python script running in the build process to modify the source file to include the logging required for the test so we could have different kernels with different logs as needed.

3.4 Baseline

Our results are all for the Linux Kernel 2.6.6. They can be ported forward but we needed a stable environment to work on in order to offer comparable results.

The baseline kernel is already a modified kernel, we have applied the following patches to it:

Increase rate-halving limit. The cwnd limit used for rate halving during a loss event was increased from one half of ssthresh to ssthresh. This was to enable easy comparison of TCP graphs and is not deemed to affect the overall performance.

Disable Throttle action. The host receive queue is where incoming packets are buffered before processing. In Linux the receive queue includes a throttle action whereby once the queue fills all subsequent packets are dropped until the queue empties. This throttle action is a DoS protection but has serious adverse effects on performance and caused a great disruption to the TCP ACK clock. In all of our tests throttle action is therefore disabled so that the receive queue operates a pure FIFO discipline. We note that this change has now become standard in later Linux distributions.

Initial performance results for Linux 2.6.6 on a 950Mbps 220ms rtt link show that after about 100Mbps the sender side fails to fill the link, see Figure 3.3 on the following page. We see that up to the 100Mbps mark the actual rate of the transfer is very close to the line rate, but after that the transfer rate is unable to keep up with the line rate.

Figure 3.4 on page 28 shows the broken behaviour in more detail. Figure 3.4(a) on page 28 shows the $cwnd$ time history from which it can be seen that the cwnd stays at very small value for extended periods (many 10s of seconds) following a loss event.

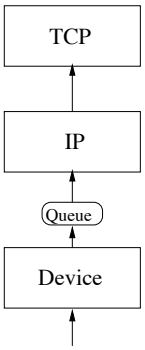


Figure 3.2: Processing stages of an incoming packet and location of the receive queue

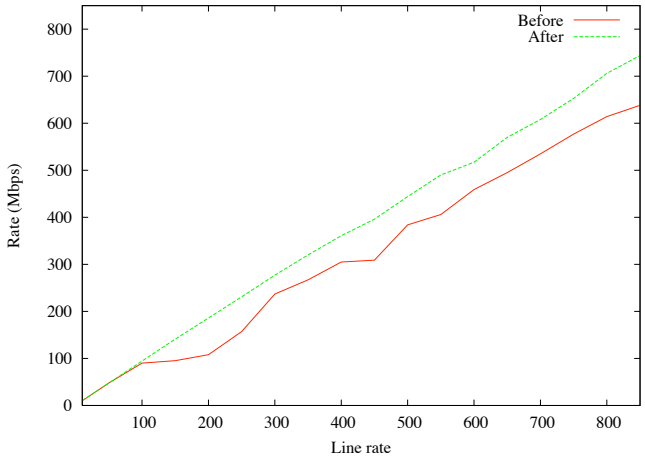


Figure 3.3: Overall performance in Mbit/s for a specified line rate. Before the changes and after.

During these periods, the packet sequence number is not advanced and the transfer is stalled. In Figure 3.4(b) on page 28 we zoom-in into the fifth loss-event of Figure 3.4(a) on page 28. This plot shows us the total time to process an ACK packet within the kernel (measured as the time spent in the `tcp_ack` function), the time to walk the list for marking the SACK scoreboard and the time spent in the retransmit function. We also see the number of packets traversed in the send buffer during our scoreboard handling and we see that it has a strong correlation to the total time spent processing ACKs.

It can be seen from figure 3.4(b) on page 28 that the number of packets walked in the SACK scoreboard marking reaches 60,000 and at that the time to process a single ACK reaches about 333 μs whereas the time between arrival of TCP ACK packets is only about 13 μs. This results in the host receive queue (where incoming packets are

buffered before processing, see figure 3.2 on the preceding page) overflowing very quickly since the kernel is unable to process the received ACKs fast enough. This results in the loss of many ACKs, losing us SACK information, damaging the TCP ACK clock and stalling the transfer.

Algorithm 3.4.1 Scoreboard marking (original)

```
for i = 0 to n_sacks do
  for j = 0 to n_pkts do
    if end(pkt_j) >= end(sack_i) then
      break //End of SACK block, go to next SACK block
    end if
    if end(pkt_j) < snd_una then
      continue //Old SACK information, skip
    end if
    if pkt_j is in sack_i then
      continue //Packet not in SACK block, skip
    end if
    if pkt_j is not SACKed then
      Mark as SACKed
    end if
  end for
end for
```

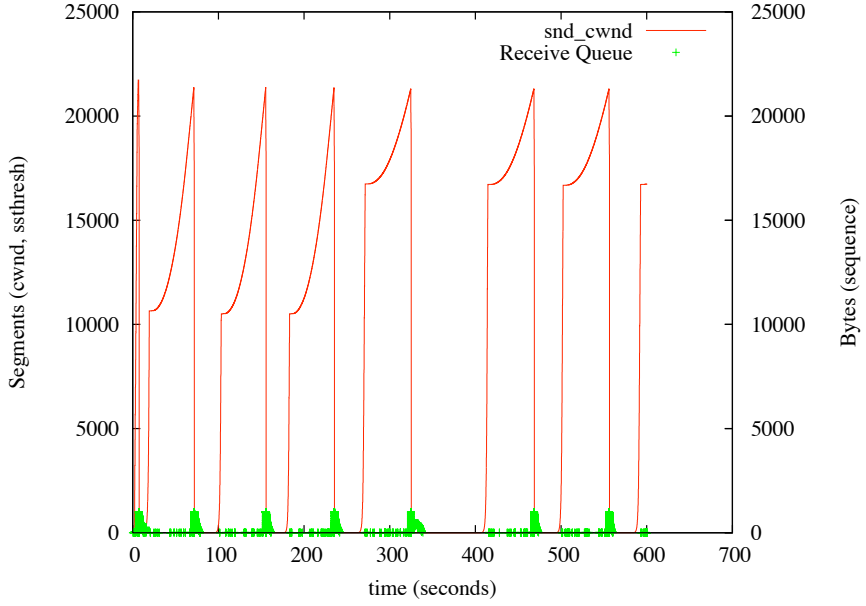
3.5 Proposed fixes

In this section we discuss the modifications we have developed to improve SACK process performance. Before proceeding, however, as a reference point we provide pseudo-code description of the algorithms in the base kernel in Algorithm 3.4.1 and Algorithm 3.4.2. These are simplifications of the actual code which omit details of the exact representation of the data structures but they are complete enough for present purposes.

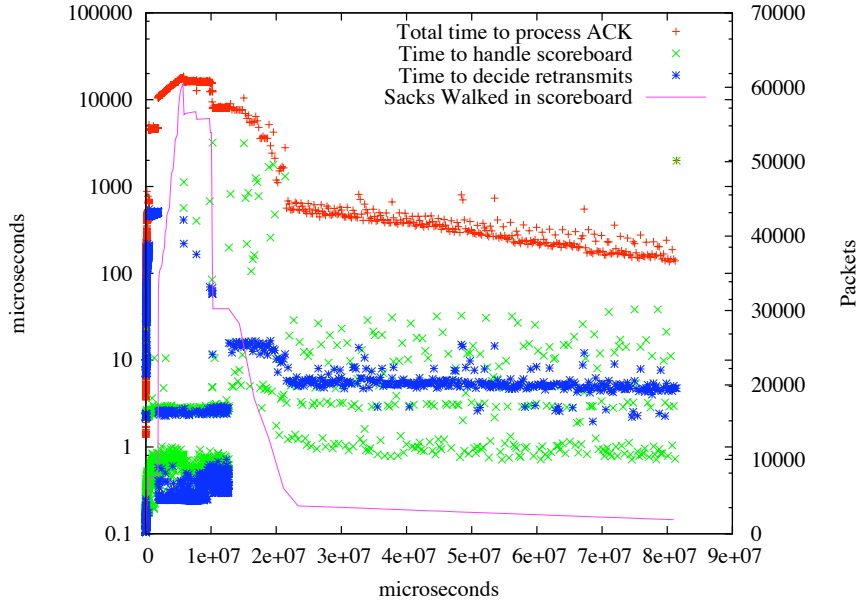
We consider the following proposed fixes.

3.5.1 Walk SACKed Holes List

In the scoreboard marking process we require to mark packets as SACKed or lost based on the SACK information contained in the TCP ACK packet. We can safely



(a) CWND trace



(b) Trace of loss event 5

Figure 3.4: Baseline Kernel at 950Mbps, 220ms rtt and 20% queue size

Algorithm 3.4.2 Retransmit (original)

```
count ← lost_pkts
if count > 0 then
    for i = 0 to n do
        if pkts_in_flight > snd_cwnd then
            return
        end if
        if pkt_i is marked lost and was not retransmitted then
            retransmit pkt_i
            count ← count − 1
            if count == 0 then
                break
            end if
        end if
    end for
end if
if can't send more then
    return
end if
for i = 0 to n do
    if cannot send forward retransmits then
        break
    end if
    if pkts_in_flight > snd_cwnd then
        break
    end if
    if pkt_i was SACKed, marked lost or already retransmitted then
        continue
    end if
    retransmit pkt_i
end for
```

ignore packets that are already SACKed since marking them again will not have any effect. Since we expect that the number of packets lost is small relative to the number of packets received we can make use of a linked-list containing only the packets not yet SACKed. A walk of this list will then scale with the number of packets lost rather than the number of packets in flight, which should yield a significant performance improvement in high bandwidth-delay paths where we may have many thousands of packets in flight.

The solution that we employ is insert a new pointer field into the data structure for a packet. We then use this to create a new linked list which is a subset of the full packets in flight list. The new list can be kept up to date by removing packets from it as they are SACKed, so that we maintain a linked-list only of packets that are not SACKed. By using this new list for SACK scoreboard marking, our walks can completely ignore all of the SACKed packets.

In addition to this major change, rather than walking the scoreboard list for every SACK block since we know that SACK blocks are non-overlapping we can sort the SACK blocks into order and then walk the list a single time. This immediately yields roughly a factor of three speedup.

The pseudo-code of the resulting modified algorithm is shown in Algorithm 3.5.1.

The impact of this change on the processing time for ACK packets is shown in Figure 3.5(a) on page 32, and detail from a trace of a single loss event is also shown in Figure 3.5(b) on page 32.

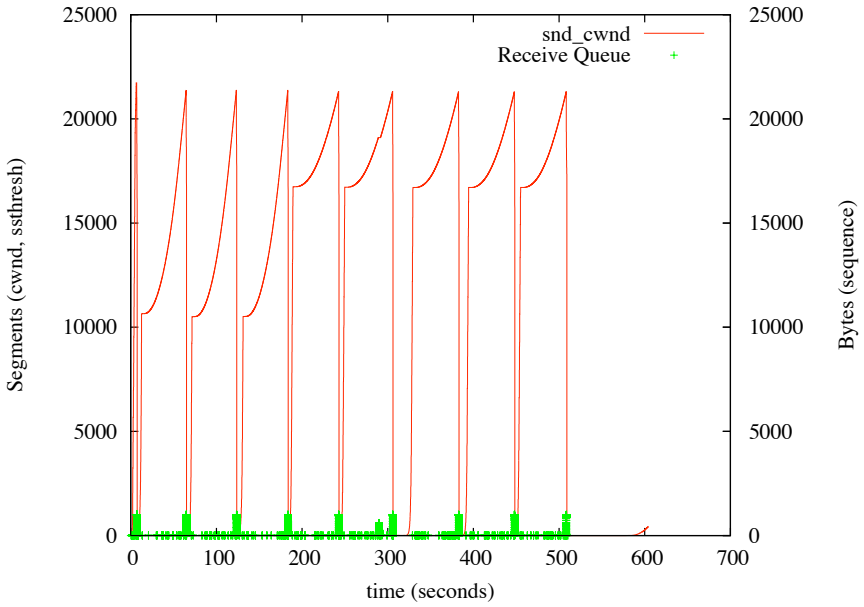
3.5.2 Retransmit hints

We can see in Figure 3.5(b) on page 32 that the retransmit process is now the bottleneck with processing times between 100 μs to 1,000 μs. The retransmit process starts from the beginning of the packets in flight list and tries to find a packet that wasn't SACKed and wasn't retransmitted already in order to retransmit it. As we advance in time it takes longer and longer to find a packet that qualifies since all the ones at the start were either SACKed or retransmitted already (those that were retransmitted already will not be eligible for further retransmission for at least another RTT to allow feedback on the success or otherwise of the first retransmission to return).

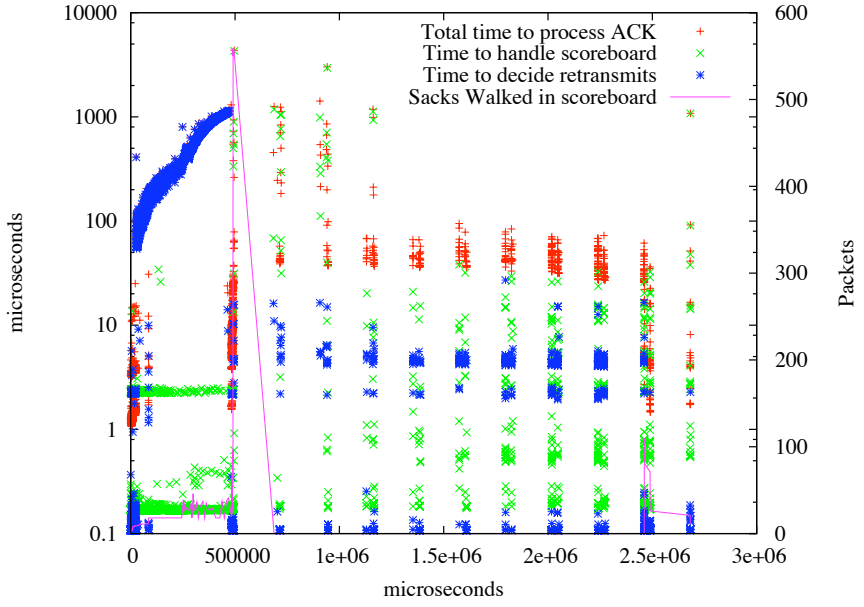
The solution that we employed is to cache a pointer to the packet that was last retransmitted. This provides a hint to the retransmit algorithm as to where it can start the

Algorithm 3.5.1 Scoreboard marking (SACK holes)

```
Sort SACK blocks sack_0 to sack_n
sackhole ← head of sackhole list
if sackhole == NULL then
    sackhole − head ← pkt − head
    sackhole ← sackhole − head
end if
pkt ← sackhole
for i = 0 to n_sacks do
    while not end of scoreboard do
        if sackhole == NULL then
            while pkt is SACKed do
                pkt ← next(pkt)
            end while
            sackhole ← pkt
        end if
        pkt ← sackhole
        if end(sackhole) >= end(sack_i) then
            break //End of SACK block, go to next SACK block
        end if
        if end(sackhole) < snd_una then
            sackhole ← next(sackhole)
            continue //Old SACK information, skip
        end if
        if sackhole is in sack_i then
            Mark as SACKed
        else //sackhole not in sack_i
            Mark sackhole as lost //Used to be in a different code section
        end if
        sackhole ← next(sackhole)
    end while
end for
```



(a) CWND trace



(b) Trace of loss event 5

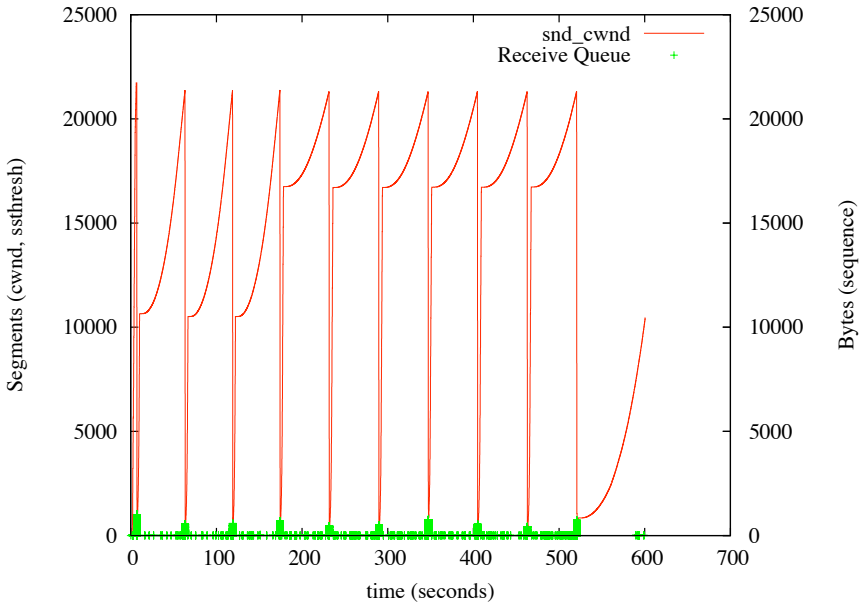
Figure 3.5: TCP performance with SACK hole list modification at 950Mbps, 220ms rtt and 20% queue size

search from next time. That is, when we walk the scoreboard we update the hint and use it in the next call to the function as the starting point when searching for packets to retransmit. The hint is reset when entering a loss event or when the scoreboard management modifies the status of a retransmitted packet. The pseudo-code of the modified algorithm is shown in Algorithm 3.5.2.

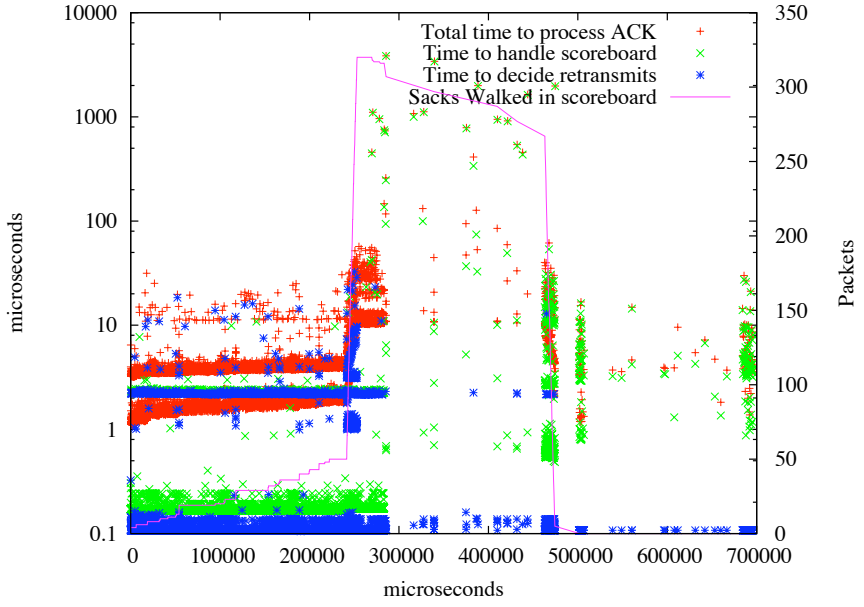
Algorithm 3.5.2 Retransmit (Retransmit Hints)

```
count ← lost_pkts
if count > 0 then
  for i = retrans_hint to n do
    if pkts_in_flight > snd_cwnd then
      return
    end if
    if pkt_i is marked lost and was not retransmitted then
      retransmit pkt_i
      retrans_hint ← i
      count ← count − 1
      if count == 0 then
        break
      end if
    end if
  end for
end if
if can't send more then
  return
end if
for i = i to n do
  if cannot send forward retransmits then
    break
  end if
  if pkts_in_flight > snd_cwnd then
    break
  end if
  if pkt_i was SACKed, marked lost or already retransmitted then
    continue
  end if
  retransmit pkt_i
end for
```

The improvement in performance can be seen in Figure 3.6(a) on the following page and Figure 3.6(b) on the next page.



(a) CWND trace



(b) Trace of loss event 9

Figure 3.6: TCP performance with SACK Holes and Retransmit Hints modifications at 950Mbps, 220ms rtt and 20% queue size

3.5.3 SACK hints

In Figure 3.6(b) on the preceding page we can see that there is still a substantial processing burden as the number of SACKs walked is in the order of a thousands. This is due to a large number of losses: while the code walks only the unSACKed packets, there are many of these.

By looking at the packet traces we found that the SACK blocks in ACK packet i and the SACK blocks in ACK packet $i + 1$ are frequently the same apart from an advance of the left edge of the first SACK block. In this case we can improve efficiency by simply starting the walk for ACK packet $i + 1$ from the same location in the list where we finished for ACK packet i. We make sure that we only use the hint from the last packet when this assumption is correct. If the SACK blocks are not a simple case of increasing the first SACK block we ignore the hint and do the full walk of the SACK holes list. The pseudo-code of the modified algorithm is shown in Algorithm 3.5.3.

The performance with this change is shown in Figure 3.7(a) on page 37 and Figure 3.7(b) on page 37. Comparing Figure 3.6(b) on the previous page to Figure 3.7(b) on page 37, it can easily be seen that there is a sharp drop in the number of SACKs walked. The resulting improvement in ACK processing time now ensures that there is no build up of backlogged ACK packets in receive queue, as can be seen if we compare the queue size in Figure 3.6(a) on the previous page and Figure 3.7(a) on page 37. We still see spikes in the number of SACKs walked, but these are only intermittent and the receive queue build-up is not large enough to result in queue overflow.

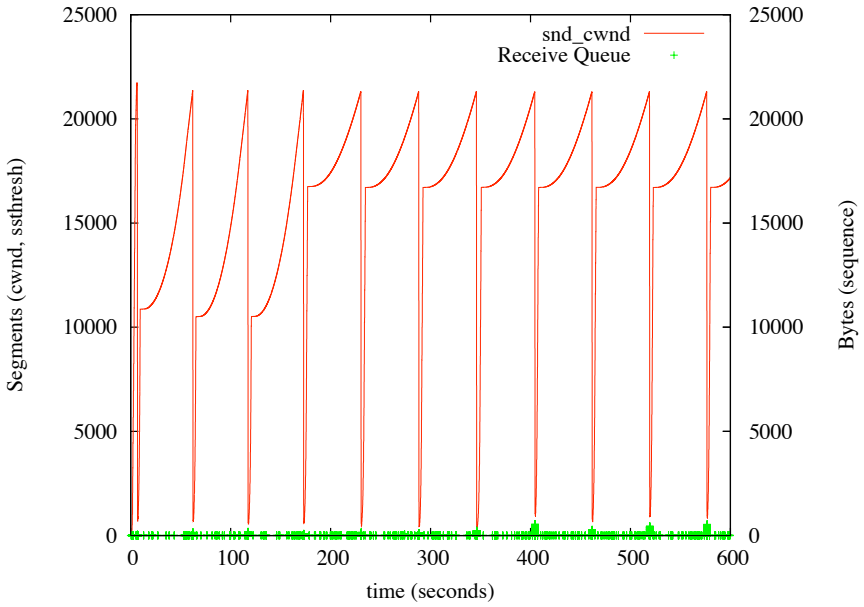
Note that in Figure 3.7(a) on page 37 we can see sharp drops in cwnd down to a few hundreds of packets, followed by a rapid rise back up to the slow-start threshold. The reason for such drops is not completely known but we suspect it has to do with the cwnd being capped to the number of packets in flight which decreases during the recovery stage.

3.6 Conclusions

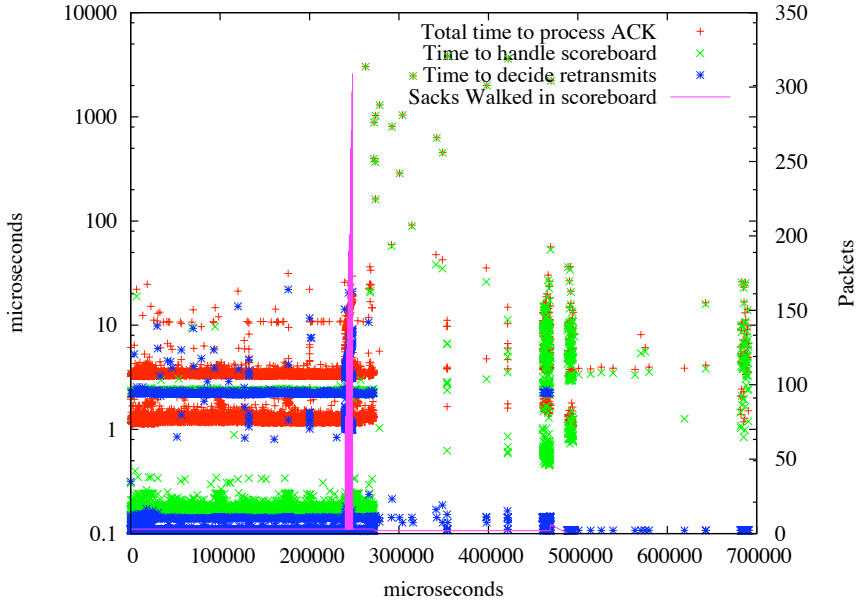
The overall impact of these changes on throughput can be seen in Figure 3.3 on page 26. Evidently, the sender is now able to completely fill the link up to speeds of 1Gb/s and delay of 220ms. This corresponds to a bandwidth-delay product of approximately 20,000 packets. Further work is needed to test performance at higher rates.

Algorithm 3.5.3 Scoreboard marking (SACK holes and SACK hints)

```
Sort SACK blocks sack_0 to sack_n
if sack_hint is set then
    sackhole ← sack_hint
else
    sackhole ← head of sackhole list
    if sackhole == NULL then
        sackhole − head ← pkt − head
        sackhole ← sackhole − head
    end if
end if
prev ← sackhole
for i = 0 to n_sacks do
    while not end of scoreboard do
        if sackhole == NULL then
            while prev is SACKed do
                prev ← next(prev)
            end while
            sackhole ← prev
        end if
        if end(sackhole) >= end(sack_i) then
            break //End of SACK block, go to next SACK block
        end if
        if end(sackhole) < snd_una then
            prev ← sackhole
            sackhole ← next(sackhole)
            continue //Old SACK information, skip
        end if
        if sackhole is in sack_i then
            Mark as SACKed
        else //sackhole not in sack_i
            Mark sackhole as lost //Used to be in a different code section
        end if
        sack_hint ← prev
        sackhole ← next(sackhole)
    end while
end for
```



(a) CWND trace



(b) Trace of loss event 9

Figure 3.7: With SACK Holes, Retransmit Hints and SACK Hints at 950Mbps, 220ms rtt and 20% queue size

Chapter 4

The cost of aggressive window growth

4.1 Introduction

In recent years, several new TCP congestion control algorithms have been proposed for deployment in long-distance and high-speed networks. As we have already discussed in earlier chapters, a primary objective in developing these algorithms has been to achieve improved scaling of performance with increasing bandwidth. In particular, a basic problem with standard TCP, when deployed on high bandwidth-delay product links, is that the time taken by a flow to recover after a back-off event can be prohibitively long, thereby leading to long data transfer times. Many authors have therefore focused on developing AIMD-like algorithms whose probing behaviour becomes more aggressive as bandwidth increases: BIC-TCP[26], Scalable-TCP[13], HS-TCP[7], and H-TCP[14] all fall into this category.

Unfortunately, adjusting the manner in which individual flows probe for available bandwidth serves not only to keep the time between consecutive congestion events short, but it also changes the way in which flows compete for available bandwidth. In fact, the basic properties of such networks may be very different to networks of standard TCP flows, and while some aspects of their behaviour have been explored, many fundamental questions pertaining to their behaviour remain unanswered. The purpose of this chapter is to address, in part, this basic observation. In particular, our objective is to explore the 'cost of missing drops' for high speed protocols. Two important questions arise in this context.

(i) The first of these is related to the long-term behaviour of networks in which

different flows have differing synchronisation rates. By synchronisation rate λ we mean the proportion of network congestion events at which a flow experiences packet loss (thus the synchronisation rate is 1 when a flow sees a drop at every network congestion event). It is known[20] that networks of TCP flows are well behaved with respect to changes in synchronisation rate; namely, long-term relative bandwidth allocation amongst competing flows scales linearly with synchronisation rate. For high-speed protocols, however, the manner in which individual flow synchronisation rates impact network behaviour is currently not clear. If the allocation of bandwidth amongst competing flows is very sensitive to synchronisation rate, then one concern is that this may lead to gross unfairness in the throughputs achieved by flows experiencing different synchronisation rates.

(ii) The second important issue is concerned with the short-term variations in rate that arise when networks are unsynchronised. For very aggressive protocols, missing a congestion event may result in an individual flow temporarily seizing a large proportion of the network bandwidth. As a result, while flow throughputs might average out to be fair over long time-scales, they may be very unfair over short time-scales.

In this chapter we investigate the above topics. We begin with a basic review of TCP and discuss some of the more desirable properties of standard TCP. Using the H-TCP high-speed proposal as an exemplar, we then present detailed experimental results that characterise the long and short term behaviour as a function of synchronisation rate. While our results are obtained for H-TCP, they identify artifacts of generic high-speed protocols that may in fact render then unsuitable in networks with drop-tail routers.

4.2 Properties of standard TCP flows

While the TCP congestion control algorithm contains a number of modes of operation, including slow-start, congestion avoidance and timeout, long-lived flows on high bandwidth-delay paths usually spend the bulk of their time in congestion avoidance mode and we therefore focus on modelling the behaviour of this mode. During congestion avoidance, the standard TCP congestion control algorithm updates the congestion window $cwnd$ according to an Additive Increase Multiplicative Decrease (AIMD) control law. In the congestion avoidance phase, when a source i receives a TCP ACK,

it increments $cwnd$ according to $cwnd \rightarrow cwnd + \alpha/cwnd$ where $\alpha = 1$ for the standard TCP algorithm. When packet loss is detected, $cwnd$ is reduced by a back-off factor β: thus $cwnd \rightarrow \beta cwnd$, where $\beta = 0.5$ for standard TCP.

The properties of networks that employ standard TCP are well known and have been reviewed in a number of publications. For convenience we review a recently developed mathematical model of TCP that exposes some of the pertinent features of networks of long-lived TCP flows.

4.2.1 A model of a network of TCP flows

Consider communication networks for which the following assumptions are valid: (i) at congestion every source experiences a packet drop; and (ii) each source has the same round-trip-time (RTT)[1]. In this case an exact model of the network dynamics may be found as follows [22].

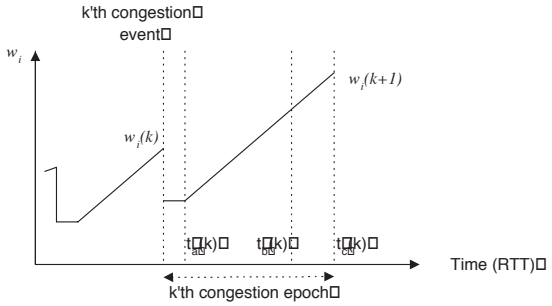


Figure 4.1: Evolution of window size

Let $w_i(k)$ denote the congestion window size of source i immediately before the k'th network congestion event is detected by the source. Over the k'th congestion epoch three important events can be discerned: $t_a(k), t_b(k)$ and $t_c(k)$; as depicted in Figure 4.1. The time $t_a(k)$ denotes the instant at which the number of unacknowledged packets in flight equals $\beta_i w_i(k)$ where β_i is the multiplicative factor of the i'th flow (α_i is the additive increase factor of this flow); $t_b(k)$ is the time at which the bottleneck queue is full; and $t_c(k)$ is the time at which packet drop is detected by the sources, where time is measured in units of RTT[2]. It follows from the definition of the *AIMD*

[1]One RTT is the time between sending a packet and receiving the corresponding acknowledgement when there are no packet drops.

[2]Note that measuring time in units of RTT results in a linear rate of increase for each of the congestion window variables between congestion events.

algorithm that the window evolution is completely defined over all time instants by knowledge of the $w_i(k)$ and the event times $t_a(k), t_b(k)$ and $t_c(k)$ of each congestion epoch. We therefore only need to investigate the behaviour of these quantities.

We assume that each source is informed of congestion one RTT after the queue at the bottleneck link becomes full; that is $t_c(k) - t_b(k) = 1$. Also when the sources detect congestion, the total window size has reached the capacity P of the pipe and each sources has increased its window size one more time before packet loss is detected. That is,

$$w_i(k) \geq 0, \quad \text{and} \quad \sum_{i=1}^{n} w_i(k) = P + \sum_{i=1}^{n} \alpha_i, \ \forall k > 0, \tag{4.1}$$

where P is the maximum number of packets which can be in transit in the network at any time; P is usually equal to $q_{max} + BT_d$ where q_{max} is the maximum queue length of the congested link, B is the service rate of the congested link in packets per second and T_d is the round-trip time when the queue is empty. At the $(k+1)$th congestion event

$$w_i(k+1) \quad = \quad \beta_i w_i(k) + \alpha_i [t_c(k) - t_a(k)]. \tag{4.2}$$

and summing over all sources yields,

$$t_c(k) - t_a(k) \quad = \quad \frac{1}{\sum_{i=1}^{n} \alpha_i} [P - \sum_{i=1}^{n} \beta_i w_i(k)] + 1. \tag{4.3}$$

Hence, it follows that

$$w_i(k+1) \quad = \quad \beta_i w_i(k) + \frac{\alpha_i}{\sum_{j=1}^{n} \alpha_j} [\sum_{j=1}^{n} (1 - \beta_j) w_j(k)] \tag{4.4}$$

and that the dynamics an entire network of such sources is given by

$$W(k+1) = AW(k), \tag{4.5}$$

where $W^T(k) = [w_1(k), \cdots, w_n(k)]$, and

$$A = \begin{bmatrix} \beta_1 & 0 & \cdots & 0 \\ 0 & \beta_2 & 0 & 0 \\ \vdots & 0 & \ddots & 0 \\ 0 & 0 & \cdots & \beta_n \end{bmatrix} + \frac{1}{\sum_{j=1}^{n} \alpha_j} \begin{bmatrix} \alpha_1 \\ \alpha_2 \\ \cdots \\ \alpha_n \end{bmatrix} \begin{bmatrix} 1-\beta_1 & 1-\beta_2 & \cdots & 1-\beta_n \end{bmatrix} \quad (4.6)$$

The matrix A is a positive matrix (all the entries are positive real numbers) and it follows that the synchronised network (4.5) is a positive linear system [2]. Many results are known for positive matrices and we exploit some of these to characterise the properties of synchronised communication networks. In particular, from the viewpoint of designing communication networks the following properties are important: (i) network fairness; (ii) network convergence and responsiveness; and (iii) network throughput. It is shown in [21, 3] that these properties can be deduced from the network matrix A. In particular:

Theorem 4.2.1 *[22, 3] Let A be defined as in Equation (4.6). Then A is a column stochastic matrix with Perron eigenvector $x_p^T = [\frac{\alpha_1}{1-\beta_1}, \ldots, \frac{\alpha_n}{1-\beta_n}]$ and whose eigenvalues are real and positive. Further, the network converges to a unique stationary point $W_{ss} = \Theta x_p$, where Θ is a positive constant such that the constraint (4.1) is satisfied; $\lim_{k\to\infty} W(k) = W_{ss}$; and the rate of convergence of the network to W_{ss} is bounded by the second largest eigenvalue of A.*

The preceding discussion illustrates the relationship between important network properties and the eigensystem of a positive matrix. Unfortunately, the assumptions under which these results are derived, namely of source synchronisation and uniform RTT, are in general, restrictive. It is therefore of great interest to extend our approach to more general network conditions. To this end, and to distinguish variables in the following discussion, we denote the nominal parameters of the sources used in the previous section by α_i^s, β_i^s, $i = 1, \ldots, n$. In this case it is readily shown that networks of flows employing AIMD algorithms may be modelled as:

$$W(k+1) = A(k)W(k),\ A(k) \in \{A_1, \ldots, A_m\}. \quad (4.7)$$

where

$$A(k) = \begin{bmatrix} \beta_1(k) & 0 & \cdots & 0 \\ 0 & \beta_2(k) & 0 & 0 \\ \vdots & 0 & \ddots & 0 \\ 0 & 0 & \cdots & \beta_n(k) \end{bmatrix} + \frac{1}{\sum_{j=1}^{n} \gamma_j \alpha_j} \begin{bmatrix} \alpha_1 \\ \alpha_2 \\ \cdots \\ \alpha_n \end{bmatrix} [\gamma_1(1-\beta_1(k)), \ldots, \gamma_n(1-\beta_n(k))]$$

and where $\beta_i(k)$ is either 1 or β_i^s. The non-negative matrices $A_2, .., A_m$ are constructed by taking the matrix A_1,

$$A_1 = \begin{bmatrix} \beta_1^s & 0 & \cdots & 0 \\ 0 & \beta_2^s & 0 & 0 \\ \vdots & 0 & \ddots & 0 \\ 0 & 0 & \cdots & \beta_n^s \end{bmatrix} + \frac{1}{\sum_{j=1}^{n} \gamma_j \alpha_j} \begin{bmatrix} \alpha_1 \\ \alpha_2 \\ \cdots \\ \alpha_n \end{bmatrix} \begin{bmatrix} \gamma_1(1-\beta_1^s), \ \ldots, \ \gamma_n(1-\beta_n^s) \end{bmatrix}$$

and setting some, but not all, of the β_i to 1. This gives rise to $m = 2^n - 1$ matrices associated with the system (4.7) that correspond to the different combinations of source drops that are possible. We denote the set of these matrices by $\mathcal{A}$.

It follows from (4.7) that $W(k) = \Pi(k)W(0)$, where $\Pi(k) = A(k)A(k-1)\ldots A(0)$. Consequently, the behaviour of $W(k)$, as well as the network fairness and convergence properties, are governed by the properties of the infinite matrix product $\Pi(k)$. While such matrix products are in general very hard, it is fortunate in our case that two assumptions that considerable simplify the analysis of this particular matrix product appear to be valid in a wide variety of network types.

Assumption 4.2.1 *The probability that $A(k) = A_i$ in (4.7) is independent of k and equals ρ_i.*

In other words Assumption 4.2.1 says that the probability that the network dynamics are described by $W(k+1) = A(k)W(k), A(k) = A_i$ over the k'th congestion epoch is ρ_i and that the random variables $A(k), k \in \mathbb{N}$ are independent and identically distributed (i.i.d.). The reader is referred to [23] for a more detailed discussion of this assumption. In summary, however, it has been found empirically to be accurate under a wide range of network conditions.

Given the probabilities ρ_i for $i \in \{1, \ldots, 2^n - 1\}$, one may then define the probability λ_j that source j experiences a back-off at the k'th congestion event as follows:

$$\lambda_j = \sum \rho_i \,,$$

where the summation is taken over those i which correspond to a matrix in which the j'th source sees a drop. Or to put it another way, the summation is over those indices i for which the matrix A_i is defined with a value of $\beta_j \neq 1$.

Assumption 4.2.2 *We assume that $\lambda_j > 0$ for all $j \in \{1, \ldots, n\}$.*

This assumption corresponds simply to the requirement that every flow eventually sees a backoff provided we wait for a long enough time.

Under the foregoing assumptions we have the following key result.

Theorem 4.2.2 *[23] Consider the stochastic system defined in the above preamble. Let $\Pi(k)$ be the random matrix product arising from the evolution of the first k steps of this system:*

$$\Pi(k) = A(k)A(k-1)\ldots A(0).$$

Then, the expectation of $\Pi(k)$ is given by

$$E(\Pi(k)) = (\sum_{i=1}^{m} \rho_i A_i)^k; \tag{4.8}$$

and the asymptotic behaviour of $E(\Pi(k))$ satisfies

$$\lim_{k\to\infty} E(\Pi(k)) \;\; = \;\; x_p y_p^T, \tag{4.9}$$

where the vector x_p is given by $x_p^T = \Theta(\frac{\alpha_1}{\lambda_1(1-\beta_1)}, \frac{\alpha_2}{\lambda_2(1-\beta_2)}, \ldots, \frac{\alpha_n}{\lambda_n(1-\beta_n)})$, $y_p^T = (\gamma_1, \ldots, \gamma_n)$. Here $\Theta \in \mathbb{R}$ is chosen such that equation chosen such that the sum of the rates of the flows at congestion equals the link rate is satisfied if w_i is replaced by $x_{pi} = \Theta\alpha_i/(\lambda_i(1-\beta_i))$.

Corollary 4.2.1 *For given $W(0)$ define random variable $\overline{W}(k)$ with:*

$$\overline{W}(k) \;\; := \;\; \frac{1}{k+1}\sum_{i=0}^{k} W(i).$$

Then expectation of $\overline{W}(k)$ is given by:

$$E(\overline{W}(k)) = \frac{1}{k+1}(I + E(A(1)) + E(A(1))^2 + \cdots + E(A(1))^k)W(0)$$

And since $E(A(1))^k \to x_p y^T$ *as* $k \to \infty$,

$$\lim_{k\to\infty} E(\overline{W}(k)) = x_p y^T W(0)$$

Comment 4.2.1 *Theorem 4.2.2 characterises the ensemble average behaviour of the congestion variable vector* $W(k)$*. This gives a measure of the short-term behaviour of the network. The corollary gives a measure of the long-term behaviour of the network. Together, they represent an Ergodicity result for the network.*

The following facts follow immediately from Theorem 4.2.2.

(i) Convergence: The congestion window vector $W(k)$ converges, on average, to the unique value $\bar{W}_{ss} = \Theta x_p$ where Θ is a positive constant such that the aggregate throughput equals the link capacity. When the $\lambda_i, i = 1, \ldots, n$ are equal, x_p is identical to the Perron eigenvector obtained in the case of synchronised networks; that is, the ensemble average in the unsynchronised case is identical to the fixed point in the deterministic situation where packet drops are synchronised.

(ii) Fairness: Window fairness is achieved, on average, when the vector x_p is a scalar multiple of the vector $[1, \ldots, 1]$; that is, when the ratio $\frac{\alpha_i}{\lambda_i(1-\beta_i)}$ does not depend on i. Observe that unlike in the synchronised case, fairness now depends upon on the relative drop probability of each flow. When the flows have equal synchronisation rate λ_i then the foregoing fairness condition is identical to that in the synchronised case.

(iii) Network responsiveness: The magnitude of the second largest eigenvalue Λ_2 of the matrix $\sum_{i=1}^{m} \rho_i A_i$ bounds the convergence properties of the network. The network rise-time when measured in number of congestion epochs is, on average, bounded by $n_r = \log(0.95)/\log(\Lambda_2)$.

4.2.2 Properties of standard TCP

The analysis of the previous section predicts the following properties. Since these are of particular importance in the context of high bandwidth-delay paths with unsynchronised drops they are explicitly validated using measurements from our experimental testbed.

(i) The time between congestion events is given by

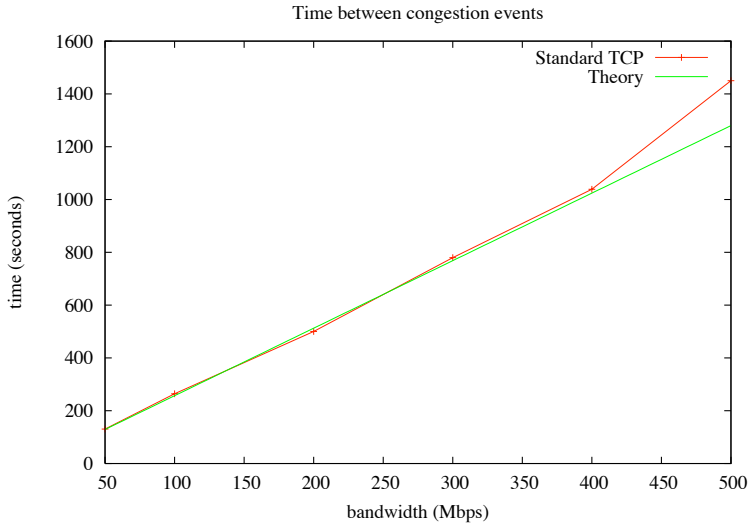


Figure 4.2: Average time between congestion events for standard TCP for one flow as the bottleneck link bandwidth is varied. Measurements taken from experimental testbed, RTT is 220ms, router queue is sized at 20% of the bandwidth-delay product

$$T(k) = \frac{1}{\sum_{i=1}^{n} \alpha_i} \Big[P - \sum_{i=1}^{n} \beta_i(k) W_i(k) \Big] + 1.$$

It can be seen that, on average, this time becomes larger as the pipe capacity increases. This behaviour is demonstrated experimentally (the testbed setup is described in detail in the next section) in Figure 4.2 where the measured time between congestion events as a function of bottleneck bandwidth is depicted. Also plotted in Figure 4.2 is the prediction made by (4.10).

(ii) The long-term average peak throughput is

$$E(w_i) = E(T) \frac{\alpha_i}{\lambda_i (1 - \beta_i) RTT_i}, \tag{4.10}$$

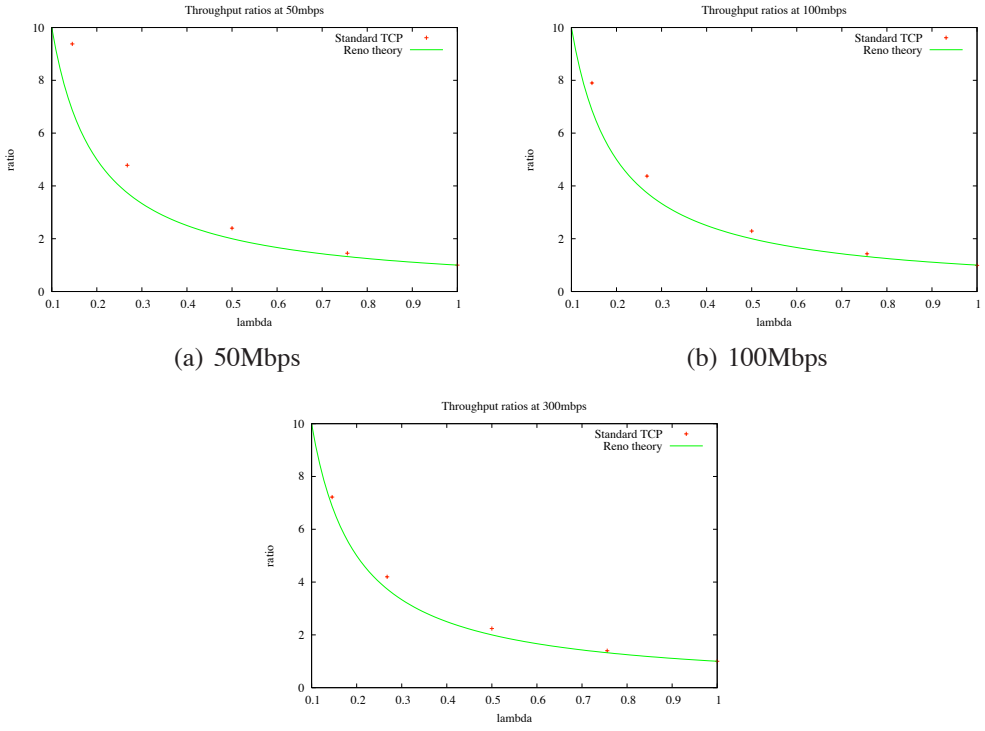
where $E(T)$ is the average time between network congestion events, λ_i is the synchronisation rate of the $i'th$ flow (assumed to be constant) and RTT_i is the round-trip-time of the i'th flow (again assumed to be approximately constant). Since $E(T)$ is the same for all flows sharing a link, the ratio of the throughputs of two competing flows is determined solely by their AIMD increase and

decrease parameters, their round-trip times and by their synchronisation rates λ_i. In particular, for two flows with same AIMD parameters round-trip times, the ratio of throughputs $E(w_1)/E(w_2)$ is equal to λ_2/λ_1. Notice that the ratio is *independent* of the bandwidth-delay product and thus high bandwidth-delay product paths behave similarly to other paths.

This behaviour is confirmed by experimental measurements in Figure 4.3. This figure shows the long-term bandwidth allocation between competing flows in a network with 10 flows in total, 5 flows with $\lambda = 1$ and 5 flows with a λ ranging between 1 and 0.145. Results are shown for link bandwidths of 50Mbps, 100Mbps and 300Mbps. It can be seen that, for standard TCP, the long-term unfairness between flows due to different synchronisation factors is an inverse linear function of synchronisation rates. It can be seen that this is indeed independent of the path bandwidth-delay product.

(iii) The matrices $A_1, \ldots, A_m$ are not dependent on the time between congestion events or the pipe capacity. Thus, not only the mean but also other statistical properties of the network as measured from the $W(k)$, are independent of these quantities. In particular, even when the long-term average throughput of flows is the same, short-term fluctuations in throughput can lead to unfairness over time-scales of a few congestion epochs. However, the foregoing analysis indicates that any such fluctuations are scale invariant with bandwidth-delay product. This remarkable property is confirmed by experimental measurements, see Figure 4.4 on page 54. This figure shows the measured distribution of peak cwnd for a flow as both the synchronisation rate and bottleneck bandwidth are varied. While the cwnd distribution varies with λ, it is invariant with bandwidth. To our knowledge, this is the first time that this type of behaviour has been documented.

Key point : *The upshot of these observations is the following (commonly) overlooked fact. While the time between congestion events increases with increasing bandwidth for standard TCP, other important quantities of interest are invariant to changing bandwidth-delay product. It remains to be seen if new protocols exhibit this or similarly nice properties.*



(a) 50Mbps

(b) 100Mbps

(c) 300Mbps

Figure 4.3: Throughput ratios with λ unfairness for standard TCP with bandwidths of 50Mbps, 100Mbps and 300Mbps. Network with 10 flows in total, 5 flows with $\lambda = 1$ and 5 flows with a λ ranging between 1 and 0.145. Measurements taken from experimental testbed, RTT is 220ms, router queue sized at 20% of bandwidth-delay product

4.3 Test setup

The test setup for this chapter is an extension of the test setup we used in section 3.3 on page 23. Our interest is in investigating the impact of synchronisation rate on the behaviour of competing TCP flows. In production networks, the synchronisation rate of a flow is determined by the pattern of packet losses that it experiences which, in turn, are typically generated by the queueing disciplines used at network routers. When buffering TCP traffic, the drop-tail queueing discipline that is ubiquitous in modern networks can lead to a complex pattern of packet drops that remains poorly understood. In order to create a controlled, reproducible test environment we therefore:

(i) Added a feature to the dummynet router software that enabled us to mark the ECN bit in buffered packets once the router queue exceeds a specified threshold

occupancy. See algorithm 4.3.1) for a pseudo-code description.

(ii) Modified the Linux kernel code to alter the sending host response to ECN marks. Normal behaviour is to backoff cwnd by half on receipt of an ECN mark. Note that, similarly to packet losses, multiple ECN marks received within a single round-trip time are treated as a single mark for backoff purposes. The modified behaviour is to backoff only with some specified probability on receipt of an ECN mark. Specifically, on receipt of an ECN mark the sending host draws a random number r with uniform distribution between 0 and 65535. The host backs off cwnd only when $r > p$ where p is a parameter that is specified by the user via a sysctl. See algorithm 4.3.2 for a pseudo-code description. The resulting synchronisation rate is $\lambda = 1 - (p/65535)$. As before, multiple ECN marks received within a single round-trip time are treated as a single mark for backoff purposes.

Algorithm 4.3.1 Dummynet ECN Marking

```
if pkt_size + queue_size > ecn_limit then
    Mark packet with ECN
else if pkt_size + queue_size > queue_limit then
    Drop packet
end if
```

By selecting the router buffer to be sufficiently large, and provided some hosts do in fact eventually backoff at congestion, backoff of sender cwnds before the buffer overflows ensures that packet losses are avoided. In all our tests we confirmed that the buffer was sufficiently large that very few packet losses occurred at the router.

4.4 Properties of H-TCP

Using the developed testbed, we investigate the properties of the H-TCP high-speed modification to the TCP congestion control algorithm when operating in a network with unsynchronised backoff events. We begin, however, by measuring the impact of H-TCP on the congestion epoch duration. Figure 4.5 on page 55 plots the mean time between congestion events as the network bandwidth is varied. Comparing this with figure 4.2 on page 46, the much shorter congestion epoch duration with the H-TCP algorithm is evident. In the following sections, we next study the impact of synchronisation rate on both the long and short term fairness of H-TCP flows.

Algorithm 4.3.2 Linux Random ECN Ignore

```
if ignore_ecn and snd_una > ecn_marker then
    ignore_ecn ← 0
end if
if packet is ECN marked then
    if ignore_ecn == 0 then
        ecn_marker ← snd_nxt + snd_cwnd*mss_cache
        r ← random(65535)
        if r < lambda then
            ignore_ecn ← 1
        else
            ignore_ecn ← 2
        end if
    end if
    if ignore_ecn == 1 then
        ECN reply
    else
        Original code for handling ECN
    end if
end if
```

4.4.1 Long-term λ Unfairness

Figure 4.6 on page 55 plots the ratio of throughputs of two H-TCP flows as the synchronisation rate of one flow is varied. This is the same test as was carried out for standard TCP with results shown in figure 4.3 on page 48. For comparison, the theory line from figure 4.3 on page 48 is also plotted on figure 4.6 on page 55. It is immediately evident that H-TCP exhibits a behaviour that is quite different from that of standard TCP. For example, with $\lambda = 0.145$ the ratio of flow throughputs with standard TCP is 1:7, while for H-TCP the ratio is 1:115 at 100Mbs and 1:142 at 300Mbs.

This behaviour can be attributed to the H-TCP feature of increasing α non-linearly with elapsed time since the last congestion event. Flows with a lower synchronisation rate experience, on average, a longer time between congestion events. Such flows therefore employ, on average, a larger effective α leading to greater unfairness than with standard TCP. Note that such behaviour can be expected of any algorithm that aggressively increases α based on elapsed time since the last congestion event. Moreover, a similar argument also carries over directly to algorithms such as HS-TCP that aggressively increase α based on the value of cwnd.

4.4.2 Short-term Unfairness

In addition to its impact on the long-term fairness between flows, we can expect that increasing α with elapsed time since the last congestion event will lead to increased short-term unfairness between flows. Figure 4.7(a) on page 56 shows how α grows as a function of elapsed time while figure 4.7(b) on page 56 shows the corresponding cwnd evolution. If one flows backs off while another misses the congestion event, it is clear that the latter flow may rapidly increase its cwnd until it is well above that of competing flows. If a flow misses more than one congestion event in sequence it can potentially grab a large share of the bandwidth temporarily. This type of behaviour is evident in the measured $cwnd$ time history plotted in figure 4.8 on page 56, where it manifests itself as large spikes in cwnd as one flow takes most of the bandwidth while the others are starved. Observe also that flows take alternating turns at grabbing the available bandwidth – this is required since the long-term average throughputs of the flows are identical as they all have the same synchronisation rate, see figure 4.6 on page 55. In the rest of this section we investigate this short-term unfairness behaviour in more detail.

Boundary effects

We begin by observing that there is a practical limit to the number of congestion events a flow can miss before its cwnd has increased to the point where it occupies the entire network capacity and a packet loss must necessarily occur. The impact of this constraint depends on the network conditions and, in particular, on the number of competing flows. In our tests we were therefore careful to check that the achieved synchronisation rate was close to the synchronisation rate demanded via the sender sysctl parameter p. The achieved synchronisation rate for a flow is measured as the number of backoffs by that flow divided by the number of network congestion events. The latter is measured as the number of events at which at least one flow in the network backs off. Backoffs by different flows that occur with one round-trip time of each other are taken to be the same network event, since congestion signalling can be delayed by up to one round-trip time in the network and also the TCP congestion control algorithm itself treats all congestion signals within one round-trip time as a single signal. Figure 4.9 on page 57 plots the measured versus demanded synchronisation rates under a range of network conditions. It can be seen that the measured rate remains close to the demanded rate.

Impact of number of flows

Recall that the synchronisation rate λ of a flow is the proportion of network congestion events at which that flow backs off. The value of λ does not depend on the time between congestion events. Nevertheless, with larger numbers of flows, for fixed synchronisation rate λ the time between congestion events is on average shorter. This is because since each flow tries to increase its individual cwnd each RTT, taken together the number of packets in flight in the network increases more quickly with the number of flows. This can be seen from the measurements shown in figure 4.11 on page 58 of time between network congestion events for two networks with different numbers of competing flows. Since in H-TCP the effective AIMD increase rate becomes more aggressive with increasing congestion epoch duration, we therefore expect the unfairness between flows with different synchronisation rates to decrease as the number of flows is increased. See for example figure 4.10 on page 57 which shows the distribution of peak cwnds for different number of flows.

Impact of BDP

In H-TCP the effective AIMD increase rate becomes more aggressive with increasing congestion epoch duration. Since, all other things being equal, the congestion epoch duration in turn increases with the bandwidth-delay product (see figure 4.5 on page 55), we can expect that the short-term unfairness also increases with bandwidth-delay product. Figure 4.12 on page 59 plots measured cwnd distributions and time histories for bottleneck bandwidths of 100Mbs and 300Mbs. It can be seen that the cwnd distribution has a considerably heavier tail at 300Mbps than at 100Mbps, reflecting the greater variability in cwnd at the higher speed for a given synchronisation rate.

Impact of backoff factor β

For one flow to grab bandwidth from another, it is not enough for the first flow to have an aggressive AIMD increase. It is also necessary for the second flow to backoff and release bandwidth that the first flow can then seize. That is, the ability of a flow to grab bandwidth when it misses drops is constrained by the willingness of the other network flows to yield bandwidth. We therefore expect that the short-term unfairness between flows depends on the backoff factors β used. Smaller backoff factors correspond to flows yielding bandwidth quite readily, while larger backoff factors make it harder

for a flow to grab bandwidth from the others. Hence, we expect the level of short-term unfairness to decrease with increasing β. See figure 4.13 on page 60 for measurements.

4.5 Summary

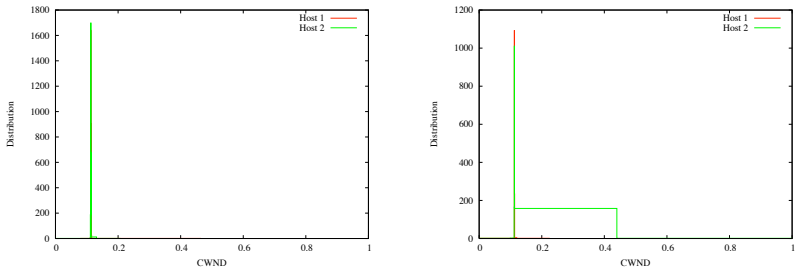
A number of important facts can be discerned from the above tests.

(i) The unfairness in long-term average throughput between flows with different synchronisation rates is amplified by the more aggressive increase rate of the H-TCP algorithm.

(ii) Large short-term fluctuations in the rate of a flow are often a feature of networks in which high-speed protocols are deployed, leading to short-term unfairness between competing flows. This unfairness can be such that a single flow can temporarily grab almost all the network capacity even though all flows have the same long-term average throughput.

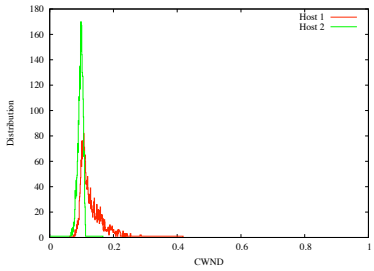
(iii) The distribution of rate variation depends on the network backoff factors. Roughly speaking, the larger the backoff factors, the smaller the variation in rate and thus the less short-term unfairness. As noted elsewhere [14, 20], increasing the backoff factor also generally reduces network responsiveness e.g. for the startup of new flows, thereby increasing the unfairness between short and long-lived flows.

We note that while the discussion here has been in terms of the H-TCP algorithm, many of the behaviours observed are expected to also be exhibited by other proposed high-speed algorithms.

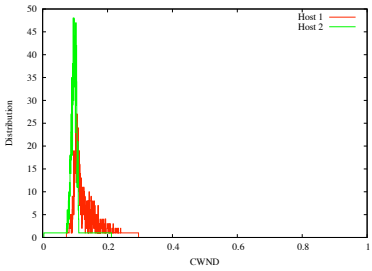


(a) 50Mbps, $\lambda = 1$

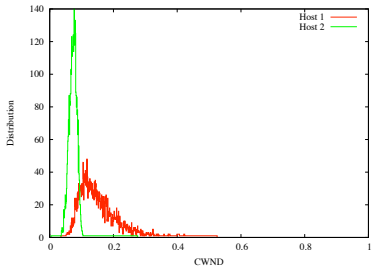
(b) 100Mbps, $\lambda = 1$



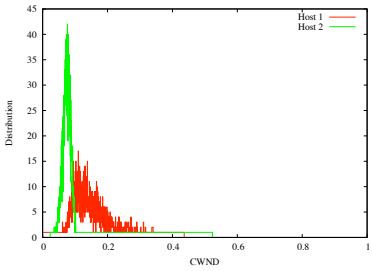
(c) 50Mbps, $\lambda = 0.76$



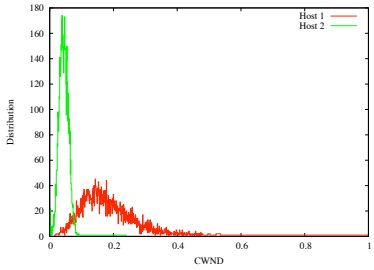
(d) 100Mbps, $\lambda = 0.76$



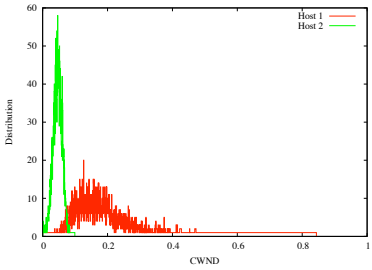
(e) 50Mbps, $\lambda = 0.5$



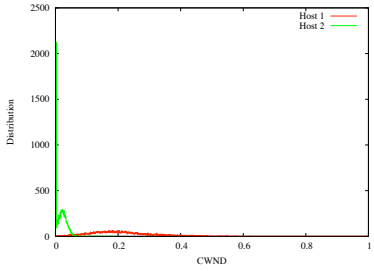
(f) 100Mbps, $\lambda = 0.5$



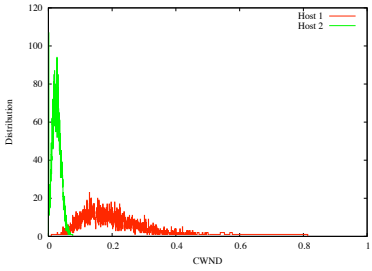
(g) 50Mbps, $\lambda = 0.26$



(h) 100Mbps, $\lambda = 0.26$



(i) 50Mbps, $\lambda = 0.14$



(j) 100Mbps, $\lambda = 0.14$

Figure 4.4: Distribution of peak window sizes for standard TCP at 50Mbps and 100Mbps. Measurements taken from experimental testbed, network with 10 flows, RTT is 220ms, router queue sized at 20% of bandwidth-delay product

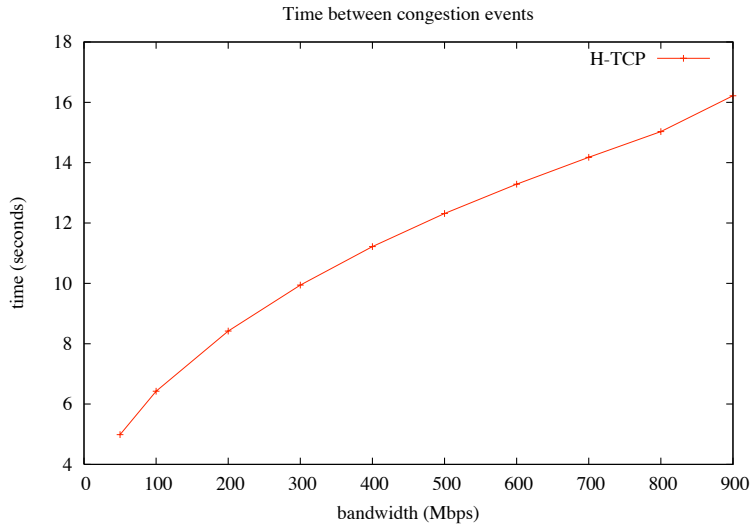
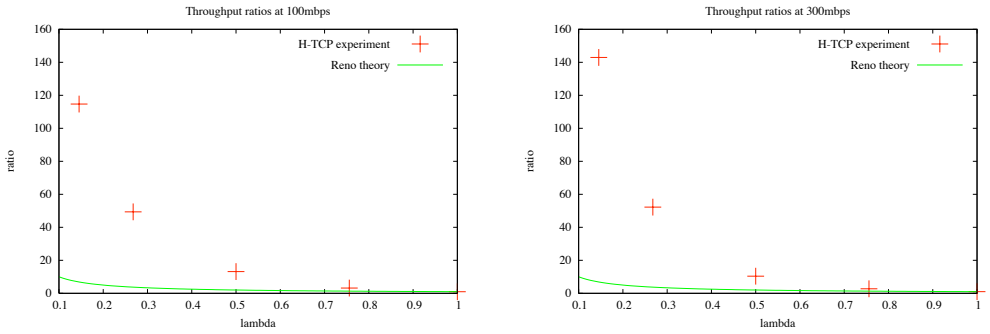
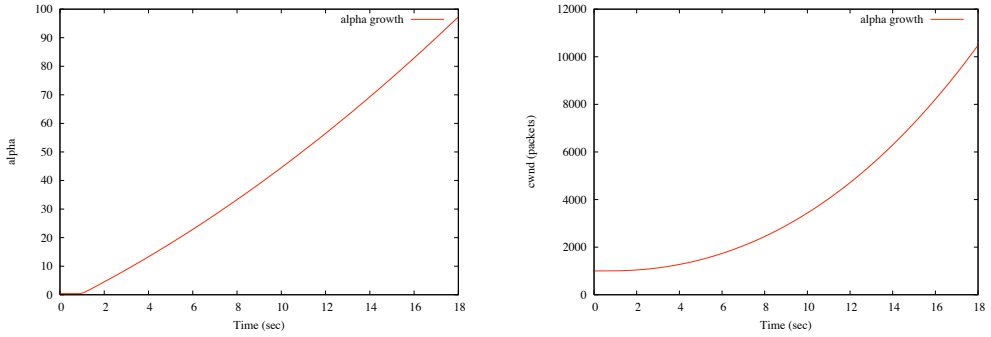


Figure 4.5: Average time between congestion events for H-TCP for one flow as the bottleneck link bandwidth is varied. Measurements taken from experimental testbed, RTT is 220ms, router queue is sized at 20% of the bandwidth-delay product.



(a) 100Mbps (b) 300Mbps

Figure 4.6: Throughput ratios with λ unfairness for H-TCP with bandwidths of 100Mbps and 300Mbps. Network with 10 flows in total, 5 flows with $\lambda = 1$ and 5 flows with a λ ranging between 1 and 0.145. Measurements taken from experimental testbed, RTT is 220ms, router queue sized at 20% of bandwidth-delay product.



(a) Evolution of α with time since congestion event

(b) Evolution of $cwnd$ growth with time

Figure 4.7: H-TCP $alpha$ and $cwnd$ evolution for 15 seconds from congestion event

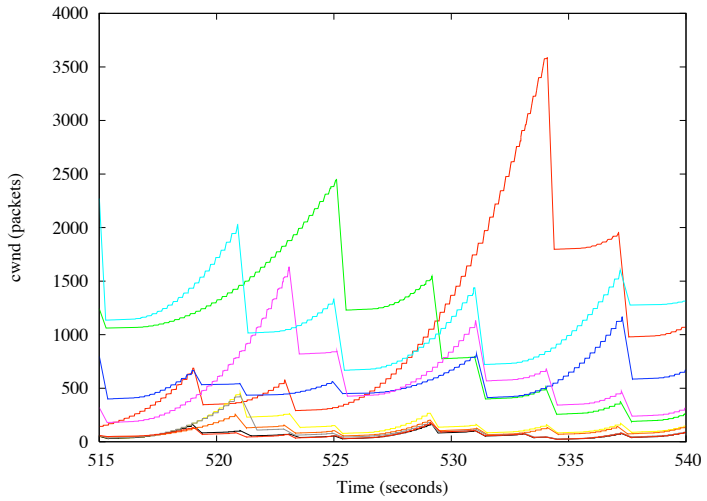


Figure 4.8: Example evolution of flow cwnds. Network with 10 flows in total, five flows with $\lambda = 0.5$ and five with $\lambda = 1$. Measurements taken from experimental testbed, RTT is 220ms, router queue sized at 20% of bandwidth-delay product.

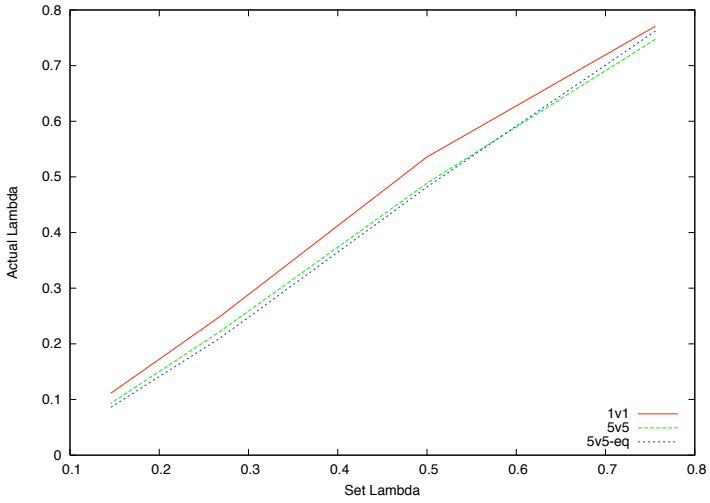
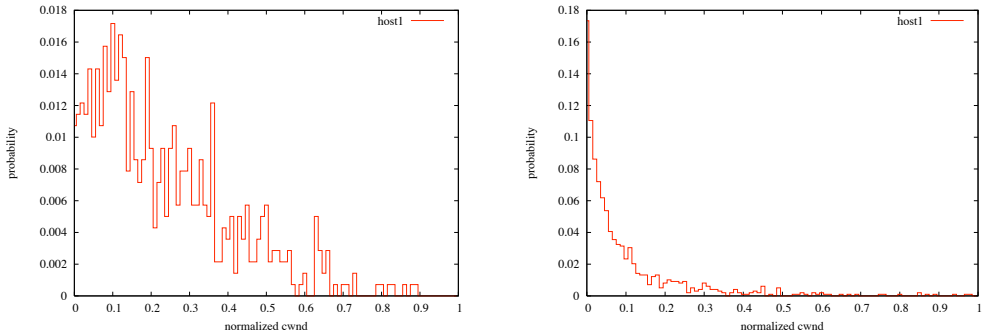
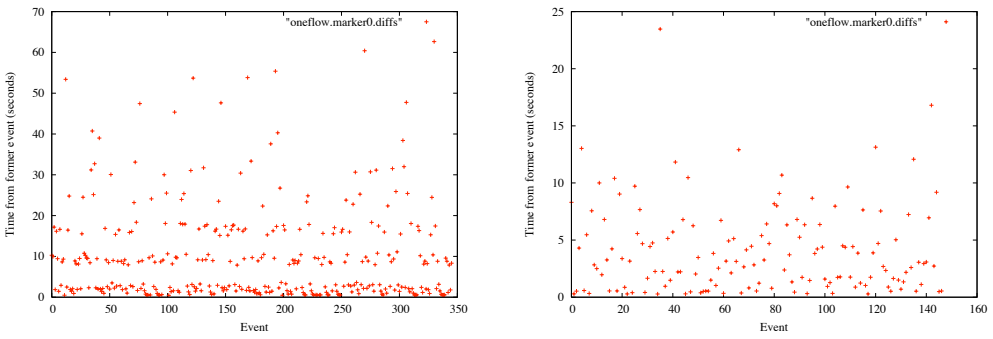


Figure 4.9: Actual versus demanded synchronisation rate as number of flows is varied. Measurements taken from experimental testbed, RTT is 220ms, bandwidth 300Mbps, router queue sized at 20% of bandwidth-delay product.



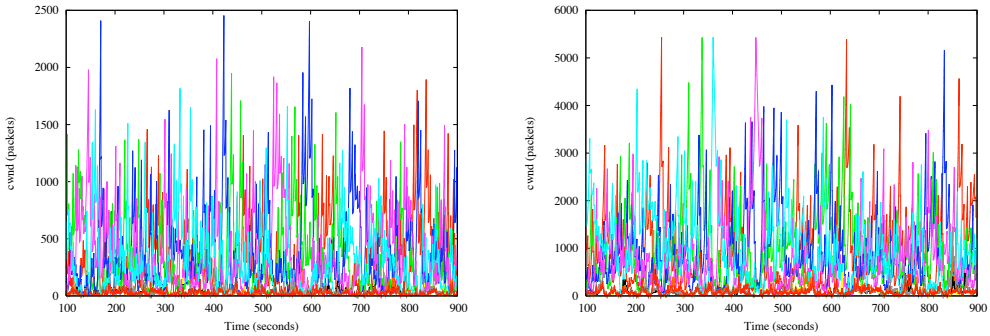
(a) 5 flows with $\lambda = 0.5$ (b) 10 flows with $\lambda = 0.5$

Figure 4.10: *cwnd* distribution for different number of flows. Measurements taken from experimental testbed, RTT is 220ms, bandwidth 300Mbps, router queue size at 20% of bandwidth-delay product.

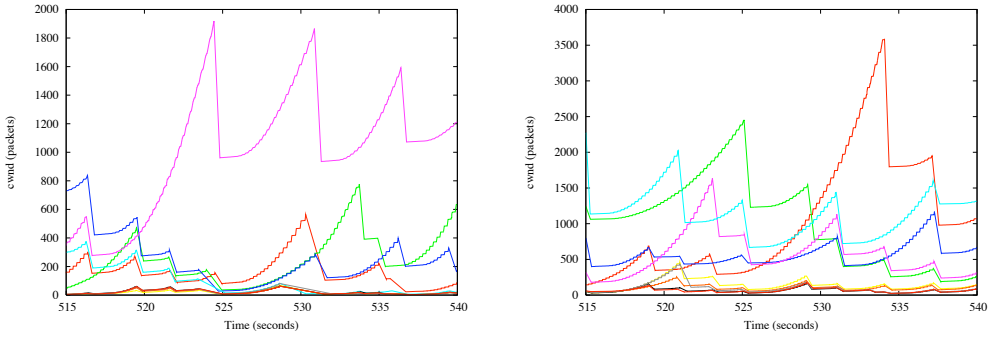


(a) 1 flow with $\lambda = 0.5$ (b) 10 flow with $\lambda = 0.5$

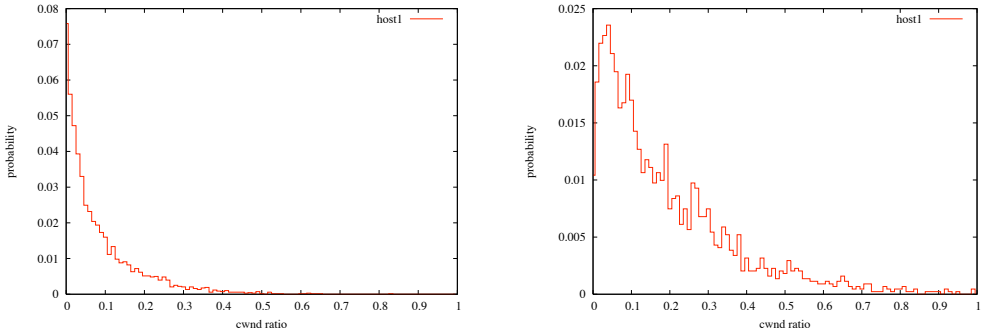
Figure 4.11: Time between congestion events for different number of flows. Measurements taken from experimental testbed, RTT is 220ms, bandwidth 300Mbps, router queue size at 20% of bandwidth-delay product.



(a) 100Mbps $\lambda = 0.5$ (b) 300Mbps $\lambda = 0.5$

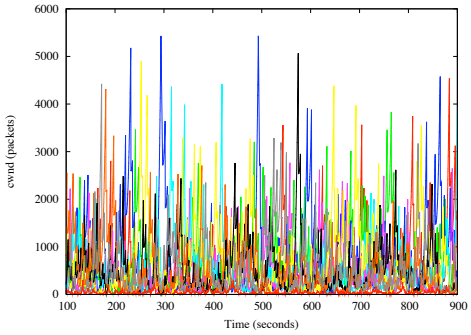


(c) 100Mbps $\lambda = 0.5$ cwnd subset (d) 300Mbps $\lambda = 0.5$ cwnd subset

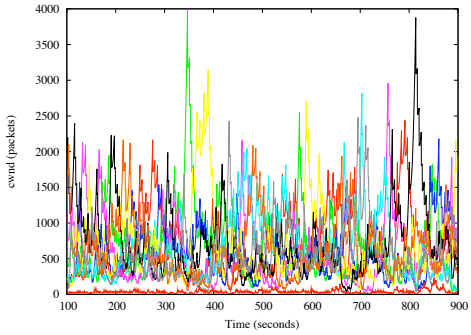


(e) 100Mbps $\lambda = 0.5$ cwnd distribution (f) 300Mbps $\lambda = 0.5$ cwnd distribution

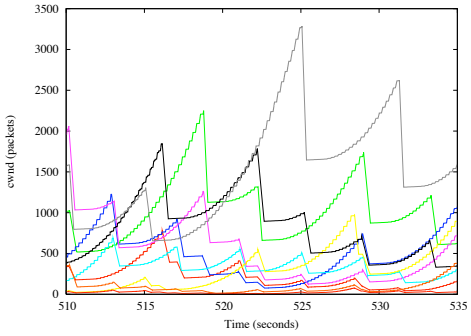
Figure 4.12: Impact of BDP on aggressiveness of H-TCP flows, 5 flows on each host all with 220ms RTT and bandwidth of 100 or 300 Mbps



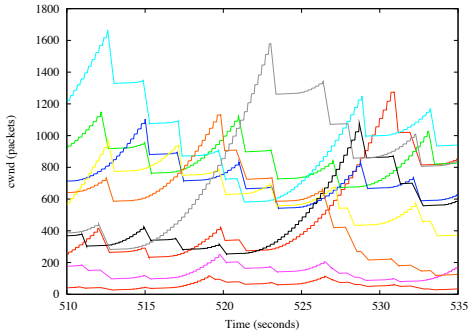
(a) $\beta = 0.5$



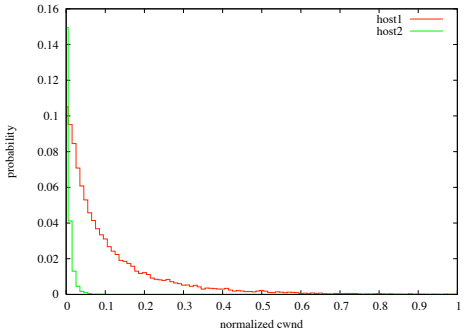
(b) $\beta = 0.8$



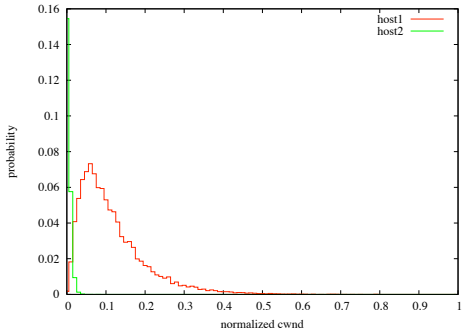
(c) cwnd history subset $\beta = 0.5$



(d) cwnd history subset $\beta = 0.8$



(e) cwnd distribution $\beta = 0.5$



(f) cwnd distribution $\beta = 0.8$

Figure 4.13: Impact of backoff factor at 300Mbps, 220ms RTT with $\lambda = 0.5$

Chapter 5

Summary and Conclusions

The objective of this thesis is to investigate some of the issues arising in the design of TCP congestion control algorithms for high bandwidth-delay product networks. In particular, we consider the fairness behaviour between competing flows in networks where packet drops are unsynchronised; that is, where not every flow sees a packet loss at each network congestion event. We review some of the main proposals for TCP congestion control in high bandwidth-delay product networks. To investigate performance, we constructed and instrumented an experimental testbed. An issue here of key practical importance is that the standard Linux network stack was found to perform poorly on high bandwidth paths due to the high computational burden associated with SACK processing. The first part of the thesis therefore focusses on more efficient network stack implementation. Using the developed testbed, we investigate the long and short term unfairness of the proposed H-TCP congestion control algorithm for high bandwidth-delay product networks. Specifically:

(i) We document the performance degradation of the standard Linux network stack on high bandwidth-delay product paths. By careful instrumentation of the network stack it was established that this degradation was primarily associated with the excessive computation burden imposed by the standard SACK processing algorithm. A modified SACK processing implementation has therefore been developed and its performance validated on commodity hardware for paths with delay up to 220ms and bandwidth up to 1Gbs (corresponding to a maximum bandwidth-delay product of approximately 18000 packets).

(ii) Based on this modified Linux kernel an instrumented testbed network was developed. To allow controlled study of the impact of synchronisation rate on

behaviour, a modified FreeBSD dummynet implementation was also developed.

(iii) Using the developed testbed network we investigated the performance of both standard TCP and H-TCP in unsynchronised conditions. We demonstrated that the unfairness in long-term average throughput between flows with different synchronisation rates is amplified by the more aggressive increase rate of the H-TCP algorithm. Large short-term fluctuations in the rate of a flow are often a feature of networks in which high-speed protocols are deployed, leading to short-term unfairness between competing flows. The distribution of rate variation depends, amongst other things, on the network backoff factors, with larger reducing short-term unfairness but reducing the responsiveness of the network.

The work performed in this thesis has been a preliminary study of scaling issues in the design of TCP congestion control algorithms for high-bandwidth delay product networks. Further work is clearly necessary.

Bibliography

[1] Oprofile. http://oprofile.sourceforge.net/.

[2] A. Berman and R. Plemmons. *Nonnegative matrices in the mathematical sciences*. SIAM, 1979.

[3] A. Berman, R. Shorten, and D. Leith. Positive matrices associated with synchronised communication networks. *Linear Algebra and its Applications*, 393(1):47–55, 2004.

[4] D. D. Clark. RFC 813: Window and acknowledgement strategy in TCP, July 1982. Status: UNKNOWN.

[5] David D. Clark. The design philosophy of the DARPA internet protocols. In *SIGCOMM*, pages 106–114, Stanford, CA, August 1988. ACM.

[6] Douglas E. Comer. *Internetworking with TCP/IP, volume I: Principles, Protocols and Architecture*. Prentice-Hall, Englewood Cliffs, NJ, fourth edition, 2000.

[7] S. Floyd. HighSpeed TCP for large congestion windows, August 2002.

[8] S. Floyd. HighSpeed TCP for large congestion windows, February 2003.

[9] S. Floyd and T. Henderson. The newreno modification to tcp's fast recovery algorithm, April 1999.

[10] G. Huston. TCP. *The Internet Protocol Journal*, 3(2), June 2000.

[11] V. Jacobson. Congestion avoidance and control. In *Proceedings of SIGCOMM*, 1988.

[12] C. Jin, D. Wei, and S. Low. FAST TCP: Motivation, Architecture, Algorithms, Performance, 2004.

[13] T. Kelly. Scalable TCP: Improving Performance in Highspeed Wide Area Networks, 2003.

[14] D.J. Leith and R.N. Shorten. H-TCP protocol for high-speed long-distance networks. In *Proc. 2nd Workshop on Protocols for Fast Long Distance Networks*, Argonne, Canada, 2004.

[15] M. Mathis, J. Mahdavi, S. Floyd, and A. Romanow. RFC 2018: TCP selective acknowledgment options, October 1996. Status: PROPOSED STANDARD.

[16] J. Nagle. RFC 896: Congestion control in IP/TCP internetworks, January 1984. Status: UNKNOWN.

[17] J. Postel. RFC 760: DoD standard Internet Protocol, January 1980.

[18] J. Postel. RFC 791: Internet Protocol, September 1981. Obsoletes RFC0760 [17]. Status: STANDARD.

[19] J. Postel. RFC 793: Transmission control protocol, September 1981. Status: STANDARD.

[20] R.N.Shorten, F. F. Wirth, and D.J. Leith. A positive systems model of tcp-like congestion control: Asymptotic results, to appear.

[21] R. Shorten, D. Leith, J. Foy, and R. Kilduff. Analysis and design of synchronised communication networks. In *Proceedings of 12th Yale Workshop on Adaptive and Learning Systems*, 2003.

[22] R. Shorten, D. Leith, J. Foy, and R. Kilduff. Analysis and design of synchronised communication networks. Accepted for publication in *Automatica*, 2004.

[23] R. Shorten, F. Wirth, and D. Leith. A positive systems model of tcp-like congestion control: asymptotic results. In *IEEE/ACM Transactions on Networking*, volume 14, pages 616–629, June 2006.

[24] W. Stevens. RFC 2001: TCP slow start, congestion avoidance, fast retransmit, and fast recovery algorithms, January 1997. Status: PROPOSED STANDARD.

[25] K. Tan, Jingmin Song, Qian Zhang, and Murari Sridharan. A compound tcp approach for high-speed and long distance networks. In *Proc. INFOCOM*, Barcelona, Spain, 2006.

[26] L. Xu, K. Harfoush, and I. Rhee. Binary increase congestion control for fast long-distance networks. In *Proc. INFOCOM*, 2004.

[27] Tom Zanussi, Karim Yaghmour, Robert W. Wisniewski, Michel Dagenais, and Richard Moore. An efficient unified approach for transmitting data from kernel to user space. In *Proc. of Ottawa Linux Symposium*, July 2003.