TCP–Carson: A Loss-event Based Adaptive AIMD Protocol for Long-lived Flows

by

Hariharan Kannan

A Thesis

Submitted to the Faculty

of the

WORCESTER POLYTECHNIC INSTITUTE

In partial fulfillment of the requirements for the

Degree of Master of Science

in

Computer Science

by

___

May 2002

APPROVED:

___

Professor Mark Claypool, Thesis Advisor

___

Professor Robert Kinicki, Thesis Co-Advisor

___

Professor David Finkel, Thesis Reader

___

Professor Micha Hofri, Head of Department

Jayamma and Daddy, who have provided me with everything.
Muralijaan, brother and dear friend.
Kanchana, who makes it all worthwhile.

Abstract

The diversity of network applications over the Internet has propelled researchers to rethink the strategies in the transport layer protocols. Current applications either use UDP without end-to-end congestion control mechanisms or, more commonly, use TCP. TCP continuously probes for bandwidth even at network steady state and thereby causes variation in the transmission rate and losses. This thesis proposes TCP Carson, a modification of the window-scaling approach of TCP Reno to suit long-lived flows using loss-events as indicators of congestion. We analyzed and evaluated TCP Carson using NS-2 over a wide range of test conditions. We show that TCP Carson reduces loss, improves throughput and reduces window-size variance. We believe that this adaptive approach will improve both network and application performance.

Acknowledgements

This thesis work is a result of the support of friends, peers and guides. I take this opportunity to thank you all for your faith and confidence in my abilities.

I express my sincere gratitude to Prof. Mark Claypool. He has been my advisor and guide. His dedication and sincerity are an inspiration for me. ”Thank You Mark!!”

I thank my co-advisor, Prof. Robert Kinicki, and reader, Prof. David Finkel. This thesis work is punctuated with their invaluable insights and suggestions.

I thank everyone at PEDS and CC. This thesis work would not have been complete without their comments and criticisms.

I am indebted to Prof. Micha Hofri and the entire faculty of the CS department for making my stay at WPI a learning and enjoyable experience. I would like to thank Sharon, Mike and Jesse for their assistance on countless occasions.

I thank Jae Chung, my partner and friend for the numerous occasions he has helped and *tolerated* me. Finally, I would like to thank all my friends at WPI for making the last two years memorable.

Contents

1 Introduction **1**

1.1 Transport Protocols 2

1.2 TCP Carson 4

1.3 Organization 6

2 Related Work **7**

2.1 TCP 7

2.1.1 Slow-Start 8

2.1.2 Congestion Avoidance 9

2.1.3 TCP Variants 9

2.2 Window-Based And Rate-Based Approaches 10

2.3 Application Specific Transport Protocols 12

3 Methodology **13**

3.1 TCP Characteristics 14

3.2 State Detection Mechanism 14

3.3 AIMD Characteristics 14

3.4 TCP Carson Algorithm 15

3.5 TCP Carson Evaluation 15

3.6 Simulation Testbed 15

4 Network State 17

4.1 TCP Behavior . . . 17

4.2 Steady State . . . 22

4.2.1 EWMA . . . 23

4.2.2 History Window . . . 24

4.3 Unsteady State . . . 29

4.3.1 Reduction In Bandwidth . . . 29

4.3.2 Retransmission Timeout . . . 29

4.3.3 Increase In Bandwidth . . . 30

4.4 Weighted Average Loss Interval . . . 31

4.5 State Detection . . . 32

4.5.1 Algorithm : WALI-32 . . . 32

4.5.2 Evaluation . . . 34

4.6 Conclusion . . . 37

5 TCP Carson 38

5.1 AIMD Behavior . . . 39

5.2 Algorithm : Adaptive AIMD . . . 41

6 Evaluation 43

6.1 TCP Carson Behavior . . . 43

6.1.1 1 TCP Carson . . . 44

6.1.2 1 TCP Carson, 1 CBR . . . 45

6.1.3 1 TCP Carson, 1 TCP Reno . . . 47

6.1.4 4 TCP Carsons, 4 TCP Renos . . . 48

6.1.5 8 TCP Carsons . . . 49

6.1.6 Summary Results . . . 51

6.2 Sensitivity To Allowed Variance 54
6.3 TCP Carson, TCP Reno and AIMD 55

7 Conclusions **58**

7.1 Future Work 60
7.1.1 Steady State Detection Algorithm 60
7.1.2 Adaptive AIMD Algorithm 60
7.1.3 Router Support 60
7.1.4 Application Performance 61

List of Figures

1.1 Traditional TCP congestion window at steady-state 4

1.2 TCP Carson congestion window at steady-state 4

3.1 Topology for the Simulation 16

4.1 Congestion Window in packets :: Single Reno flow 18

4.2 Loss interval in packets :: Single Reno flow 19

4.3 Loss interval in RTTs :: Single Reno flow 19

4.4 Congestion window in packets :: 1-16-1 TCP Reno 20

4.5 Loss interval in packets :: 1-16-1 TCP Reno 21

4.6 Loss interval in RTTs :: 1-16-1 TCP Reno 22

4.7 Exponentially Weighted Moving Average of loss intervals :: 1-16-1 . . . 23

4.8 Loss interval history size = 4 :: 1-16-1 TCP Reno 25

4.9 Loss interval history size = 8 :: 1-16-1 TCP Reno 25

4.10 Loss interval history size = 16 :: 1-16-1 TCP Reno 26

4.11 Loss interval history size = 32 :: 1-16-1 TCP Reno 26

4.12 Loss interval history size = 64 :: 1-16-1 TCP Reno 27

4.13 Steady state detection using WALI : Effect of allowed variance 31

4.14 Unsteady state detection using WALI: Effect of allowed variance 32

4.15 State Detection Algorithm : WALI-32 33

4.16 Congestion window for flow 7 among 10 competing TCP Reno flows . . 34
4.17 Loss interval and state detection for flow 7 among 10 competing TCP Reno flows 34
4.18 Congestion window for flow 1 :: 4-8-16-8-4 TCP Reno flows 35
4.19 Loss interval and state detection for flow 1 :: 4-8-16-8-4 TCP Reno flows 36

5.1 TCP vs AIMD : Congestion Windows for flows(4,5,9) :: 4-8-16-8-4 . . . 40
5.2 TCP Carson Algorithm : Adaptive AIMD 42

6.1 Congestion Window for a Single TCP Carson Flow 44
6.2 Loss Interval (in packets) for a Single TCP Carson Flow 45
6.3 Congestion Window for TCP Carson flow :: 1 TCP Carson with 1 CBR (0.4 Mb) 46
6.4 Loss interval for TCP Carson flow :: 1 TCP Carson with 1 CBR (0.4 Mb) 46
6.5 Congestion Window for TCP Reno Flow :: 1 TCP Carson with 1 TCP Reno 47
6.6 Congestion Window for TCP Carson Flow :: 1 TCP Carson with 1 TCP Reno 47
6.7 Loss interval for TCP Carson Flow :: 1 TCP Carson with 1 TCP Reno . . 48
6.8 Congestion Window for TCP Reno flow :: 4 TCP Carson with 4 TCP Reno 48
6.9 Congestion Window for TCP Carson flow :: 4 TCP Carson with 4 TCP Reno 49
6.10 Congestion Window for TCP Carson flow :: 8 TCP Carson 50
6.11 Loss interval for TCP Carson flow :: 8 TCP Carson 50
6.12 TCP Carson Sensitivity to Allowed Variance :: 4-8-16-8-4 TCP Carson . 54
6.13 TCP Reno vs AIMD(0.148, 0.125) vs TCP Carson : Congestion Windows for Flows 1 and 5 56

List of Tables

5.1 AIMD Table 42

6.1 1 TCP Carson flow vs 1 TCP Reno flow 51

6.2 TCP Carson in combination with CBR(0.4Mb) vs TCP Reno in combination with CBR(0.4Mb) 52

6.3 1 TCP Carson flow along with 1 TCP Reno flow 52

6.4 4 TCP Carson flows along with 4 TCP Reno flows 52

6.5 8 TCP Carson flows vs 8 TCP Reno flows 52

6.6 Sensitivity to Allowed Variance 55

6.7 TCP Reno vs AIMD vs TCP Carson 57

Chapter 1

Introduction

Carson City: State capital of Nevada, USA. Founded as a community in 1858, seven years after the first settlement of Eagle Station.

In the past thirty years the Internet has been transformed from an experimental system that provided email and news messaging services, into a gigantic, decentralized warehouse of information and entertainment. The current Internet has a multitude of network applications which differ both in terms of transport layer requirements and services provided to the users. These applications maybe broadly classified as:

- *Short and bursty Web transfers*. Almost 50-70% of the flows in the Internet today are HTTP transfers which are short-lived and bursty [CJOS00].
- *Streaming media applications*. Streaming audio and video flows typically last much longer than HTTP transfers. Audio streaming applications last for 78 minutes on an average [MH00].
- *Real-time applications*. Voice over IP (VOIP) and Internet Telephony applications are being marketed as next generation "killer" Internet applications. According

to an estimate from Jeff Pulver, an independent analyst of the Web-based voice communication industry, "Roughly 15 million people in the United States use voice communication over the Internet."[1] Typical voice conferences over the Internet are long lived flows.

- *Long lived data flows*. Most traffic in the Internet today (nearly 80%), is still carried by a small number of flows [Mat01]. These flows are typically long lived file or object transfers.

While Web transfers constitute the majority of flows on the Internet, the majority of the bandwidth consumption comes from long lived flows [Mat01]. Long lived multimedia applications require a steady transmission rate, while, long lived data flows require a high throughput.

1.1 Transport Protocols

The application's network requirements are handled by the transport layer protocols. The Internet today is driven primarily by two transport layer protocols: UDP and TCP.

The User Datagram Protocol (UDP) is a light-weight transport layer protocol without connection setup delays, flow control or error control, providing applications with a flexible interface to the network. For these reasons, UDP has been used for delay-bound, real-time applications and multimedia applications. However, UDP does not respond to network congestion. Hence, applications sitting on top of UDP need to perform their own flow-control and retransmission mechanisms.

The Transmission Control Protocol (TCP), on the other hand, is a heavy-weight transport layer protocol. Currently, more than 80% of the Internet bandwidth usage is from

[1] http://www.pulver.com

TCP-based applications [SHS97]. TCP establishes an end-to-end connection before transmitting data. It is an acknowledged datagram service which retransmits on detecting packet losses. At the same time, TCP considers packet drops as an implicit indication of congestion and modifies its transmission rate (window-size) based on packet drops. TCP, therefore, has been the choice of most network applications.

There are two distinct phases in TCP - *slow-start* and *congestion avoidance*. The goals and behavior of TCP are very different in each of these phases.

During slow-start, TCP attempts to estimate of the available bandwidth by probing aggressively for bandwidth (a window-size (W) increase of one packet, W+1, for every packet successfully transmitted) until it reaches a threshold value. However, during this exponential growth period, the probable loss is also correspondingly high. If the network conditions do not change, up to W/2 packets could be lost in a round-trip time. Traditional TCP shrinks back its window-size to 1 on the loss of one or more packets in the slow-start phase and sets the threshold to W/2. It then starts the probing process again. Slow-start continues until during a particular probing the window-size reaches the threshold without the loss of a packet. At that time, the second phase is entered.

The second phase is referred to as congestion avoidance. Congestion avoidance attempts to optimize the window-size and reacts to congestion in the network. During congestion avoidance, increase in window size is linear (window size increases by a packet for every round trip time worth of successful transmission, called *additive increase*). Also in this phase, for each packet loss the window size is reduced to half of its current size, called *multiplicative decrease*. This is further illustrated in Figure 1.1 where we plot the congestion window over time for a particular flow. In this case, the congestion window size gradually increases from 8 to 16. This is the additive increase phase. When the congestion window size is 16, the flow experiences loss and the window size is decreased to 8. This is multiplicative decrease. During congestion avoidance a fixed pattern of additive

increase, packet loss, multiplicative decrease, additive increase is followed.

While TCP's probing pattern is suited for flows under network congestion, it is not beneficial for long lived flows under a steady network state. In a steady-state network, packet losses during congestion avoidance are due to TCP's continuous probing for bandwidth. Again, in Figure 1.1, we observe an unnecessary oscillation in the congestion window size, thereby, a variation in the transmission rate. TCP Carson is an enhancement to the TCP protocol in the congestion avoidance phase to suit long lived flows in a steady network state.

1.2 TCP Carson

TCP Carson is a modification of TCP Reno and shares the same characteristics as traditional TCP. It uses a window-based approach to monitor its transmission rate, is an end-to-end, fully reliable, acknowledged datagram service, and uses loss as an indicator of network congestion.

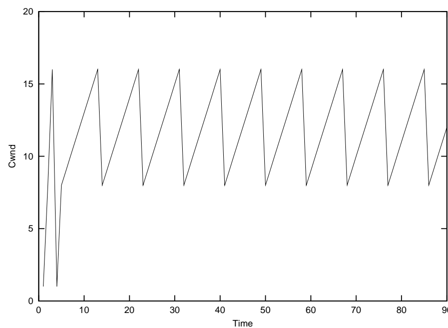


Figure 1.1: Traditional TCP congestion window at steady-state

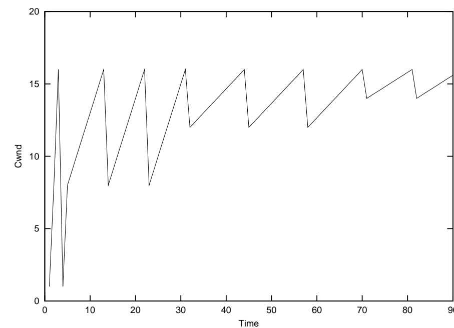


Figure 1.2: TCP Carson congestion window at steady-state

TCP Carson detects the network state using a history of loss-intervals (interval in packets between two consecutive loss events). On detecting steady state, TCP Carson adapts the change in the transmission rate in the congestion avoidance phase. During steady state, it gradually probes less aggressively for bandwidth. Conversely, it also re-

duces its decrease in transmission rate in response to a packet drop, as depicted in Figure 1.2. Initially the TCP Carson flow's congestion window oscillates between 8 and 16. However, on detecting steady state, TCP Carson changes its additive increase and multiplicative decrease factors. Thus, on a packet drop instead of reducing its window size to 8, it reduces it to 12. Again, its rate of increase in window size is reduced, taking it longer to reach the window size of 16. On detecting steady state again, instead of reducing its window size to 12, it reduces it to 14 and so on. TCP Carson thus achieves a steadier transmission rate, a higher average congestion window and experiences fewer packet drops as compared to traditional TCP algorithms.

The motivation for TCP Carson stems from the following key observations:

- The justification to improving TCP, rather than UDP, is that the principle threat to the stability of the Internet is from protocols that do not adopt any kind of congestion control mechanisms [FHPW00].

- Long lived TCP flows typically spend more time in the congestion avoidance phase than in the slow-start phase. Hence, modification in the congestion avoidance phase has a greater impact on the performance of long lived flows.

- An important characteristic of the current Internet is its inherent dynamic nature. A protocol which adapts to the changing network conditions may be more conducive to application performance as well as the performance of the network itself.

We implemented the TCP Carson protocol in NS-2 [oCB]. We evaluated TCP Carson on a range of test conditions. Our findings indicate that TCP Carson achieves a higher throughput, lower loss-rate and reduced window size variance in comparison with TCP Reno for long lived flows. On a 1 Mb bottleneck link, 8 TCP Carson flows achieved nearly 20% higher average throughput, 15% higher average congestion window size and

30% lower average loss rate as compared to 8 TCP Reno flows.

1.3 Organization

The rest of the thesis report is organized as follows. Chapter 2 explores the related work in the area of TCP congestion avoidance. Chapter 3 explains the methodology adopted for building, evaluating and analyzing TCP Carson. Chapter 4 explores traditional TCP behavior in terms of congestion window and loss characteristics, and develops a mechanism to detect steadiness in the network. Chapter 5 discusses the algorithm for the TCP Carson adaptive AIMD protocol. Chapter 6 presents the evaluation and analysis of the protocol under different network conditions. Finally, Chapter 7 concludes with our findings and future work.

Chapter 2

Related Work

The men of experiment are like the ant, they only collect and use; the reasoners resemble spiders, who make cobwebs out of their own substance. But the bee takes the middle course: it gathers its material from the flowers of the garden and field, but transforms and digests it by a power of its own.

–Francis Bacon, 1561-1626

Congestion control at the end-host has been a topic of active research. Various enhancements to the traditional TCP algorithm have been proposed. These may be broadly classified as window-based approaches, rate-based approaches and application specific approaches. In this chapter we explore end-to-end congestion control protocols and their impact on applications and the network itself.

2.1 TCP

Congestion control at the end-host is controlled by the transport protocols which govern the transmission rate on a particular flow. Transport Control Protocol (TCP) is the primary protocol used by majority of the applications on the Internet to provide congestion and

flow control. Hence, the robustness of TCP congestion control plays a critical factor in the stability of the Internet. TCP transmits a window of packets in every round trip time (RTT). Packet drops are considered as an implicit indication of network congestion; thereby, TCP modifies the window size based on packet drops. As discussed in Chapter 1, TCP has two distinct phases: slow-start and congestion avoidance. Much research has gone into studying and enhancing the behavior of TCP in both these phases.

2.1.1 Slow-Start

During slow-start, TCP attempts to estimate the bandwidth available for a particular flow. In this phase, there is an exponential increase in the window size on successful packet delivery. On a loss, TCP sets its window size back to 1. Slow-start continues until it reaches a threshold on the window-size. [AFP98] investigates the drawbacks of setting the window size to 1 and proposes other initial window sizes.

Enhancements suggested to the slow-start algorithm fall in two categories, those that involve the network in choosing the initial window size, and, those that use the threshold parameter to adjust the window size. Congestion Manager (CM) [BRS99], presents a framework for managing network congestion at the end-hosts. CM exposes a API to enable transport protocols to learn about network characteristics. Thus, a new flow can learn about the available bandwidth on a particular link using the CM API instead of performing slow-start. TCP Fast-start [PK98] is another proposed mechanism for short-lived web transfers, that suggests reusing network information cached in the recent past and reducing the overhead due to slow-start. TCP smooth-start [WS00], on the other hand, is a variant of the slow-start algorithm, that provides a smooth transition between the exponential and linear growth phases of TCP congestion window.

TCP Carson uses the traditional TCP slow-start algorithm.

2.1.2 Congestion Avoidance

During congestion avoidance, TCP attempts to optimize the window size and react to network congestion. In this phase, there is a linear additive increase to the window size (window size increment of 1) on successful packet deliveries for a round trip time and a multiplicative decrease (window size decrement to half current size) on encouuntering packet loss.

The additive increase parameter is represented by a and the multiplicative decrease parameter is represented by b. That is on a loss event, the congestion window is decreased from W to $(1\text{-}b)W$, and otherwise the congestion window is increased from W to $W + a$ each round-trip time [FHP00]. In the rest of the paper, we refer to traditional TCP as TCP(1, 0.5) where the *1* is the increment value, a, and *0.5* is the decrement factor, b.

There are many variations to the traditional TCP algorithm. The more well-known variants include: Tahoe, Reno, Vegas, and more recently, New-Reno and SACK [FF96]. All these variants of TCP have slow-start and congestion avoidance algorithms in common.

2.1.3 TCP Variants

Tahoe was among the first implementations of TCP and was introduced in version 4.3 of BSD in 1988 and implements the Fast-Retransmit [APS99] algorithm. The Fast Retransmit algorithm uses the arrival of three duplicate acknowledgments (ACKs) as an indication of loss of a segment and retransmits the segment without waiting for the retransmission timer to expire. In 1990, TCP Reno was introduced in version 4.3 of the BSD TCP suite. In addition to the Fast-Retransmit algorithm, TCP Reno implements the Fast-Recovery [APS99] algorithm. The Fast-Recovery algorithm suggests that following a Fast-Retransmit, the transmission may remain in the congestion avoidance phase

instead of entering slow-start. New-Reno is an enhancement of TCP Reno which avoids many of the retransmit timeouts that occur in Reno when multiple packets are lost from a window. In TCP SACK, the receiver reports back to the sender the reception of any non-contiguous set of packets in a single acknowledgment. Thus, the sender can retransmit non-contiguous lost packets. TCP Vegas [LPW01] is an alternative source-based congestion control mechanism. It anticipates the onset of congestion by monitoring the difference between the rate it is expecting and the rate it is realizing, given its congestion window.

TCP Carson is an enhancement to the congestion avoidance phase of TCP Reno. It is a variant of TCP to suit long lived flows and implements the steady-state detection algorithm and the adaptive AIMD algorithm in addition to slow start and congestion avoidance.

2.2 Window-Based And Rate-Based Approaches

Many algorithms have been suggested as an alternative to the TCP congestion control algorithm. Some of these require network participation, such as Rate Adaptive TCP (RATCP). RATCP [And00] is a modification of TCP congestion control in which the congestion window is adapted to *explicit* bottleneck rate feedback. The other algorithms can be broadly sub-classified as either rate-based or window-based.

The Additive Increase Multiplicative Decrease (AIMD) [FF99, FHP00] is a window based approach that has its basis on the deterministic response function of TCP. [FHP00] indicates that the $(1, 0.5)$ approach of TCP may be unnecessarily severe and suggests other values for a and b which behave like TCP over a long duration. This family of protocols still exhibit the same probing-loss pattern as exhibited by traditional TCP, even though it is not as accented. The speed to use available bandwidth is slower than TCP [YKXL00].

In addition, because of their small fixed decrease factors, their reduction in bandwidth to an actual congestion event is small. Different relationships have been suggested between a and b. [FHP00] suggests using:

$$a = \frac{3b}{2-b}$$

while [YL00] suggests the following.

$$a = \frac{4(1-b^2)}{3}$$

Chapter 5 discusses some of the issues of using the AIMD approach.

Rate based protocols, such as TCP-friendly rate control (TFRC), Rate Adaptive Protocol (RAP) and TCP Emulation At Receiver (TEAR) try to maintain a relatively steady sending rate while still being responsive to congestion. RAP [RHE99] is a rate adaptive protocol specific to pre-encoded real time streams in which the congestion control and the error control mechanisms work separately. TEAR [ROY00] suggests a flow-control approach for multimedia streams where the receiver determines the rate of transmission based on the congestion signals in their forward paths. TFRC [FHPW00] suggests using a control equation that relies on the loss event rate[1] to determine the transmission rate. In TFRC, an upper bound on the sending rate is determined as a function of the packet size, round-trip time, steady-state loss event rate, and the TCP retransmit time-out value. In its simplified form, the TCP response function is given by [FHPW00]:

$$T = \frac{\sqrt{1.5}s}{R\sqrt{p}}$$

[1] A loss event is defined as a loss of one or more packets in the same RTT. The concept of loss event has its basis in TCP-SACK, which can indicate multiple packets lost in a single round trip time.

where, T is the transmission rate in Mbps, s is the packet size in bits, R is the round trip time in seconds and p is the loss event rate.

Rate-based protocols are less responsive to congestion events, primarily because the sending rates are calculated at discrete intervals of time, during which they transmit at a fixed rate. The transmission rate (window size) of TCP Carson, on the other hand, is a continuous function of the current network state and observed loss events. Hence, TCP Carson is highly responsive to bandwidth changes and network congestion.

2.3 Application Specific Transport Protocols

The unsuitability of TCP or UDP for all applications has led to research in the area of application specific transport protocols. Various alternatives to TCP and UDP have been proposed to suit audio and video streaming applications on the Internet. SCP [CPW98] is another proposed media streaming transmission protocol that detects state of a flow using variations in the RTT, and modifies its transmission rate accordingly. MM-Flow [CC00] is a transport protocol built on top of UDP that provides better performance for multimedia applications.

Chapter 3

Methodology

The fundamental principle of science, the definition almost, is this: the sole test of the validity of any idea is experiment.

–Richard P. Feynman, 1918-1988

In this chapter we outline the methodology adopted for developing TCP Carson. We also specify the simulation setup that was used in the development process. Development of TCP Carson involved the following steps:

- Study TCP congestion avoidance behavior.
- Develop and evaluate algorithm to detect steady/unsteady network state.
- Study AIMD congestion avoidance behavior and drawbacks.
- Develop TCP Carson algorithm.
- Evaluate TCP Carson.

3.1 TCP Characteristics

Under steady network state, TCP continues to probe for bandwidth, thereby causing losses and variation in transmission rate. TCP Carson attempts to reduce the losses and provide a steadier transmission rate during congestion avoidance. Hence, the first step towards building TCP Carson was characterizing the TCP congestion avoidance phase. Section 4.1 inspects the loss and loss-event patterns of TCP during the congestion avoidance phase and correlates it to the congestion window oscillations of TCP. We define the terms *steady* and *unsteady* and discuss the behavior of TCP in each state.

3.2 State Detection Mechanism

The next step involved developing an algorithm to detect the current state of the network. Section 4.1 defines the terms *steady* and *unsteady* network state. Section 4.2 evaluates metrics to detect steady state in a network and analyzes their advantages and drawbacks. Section 4.3 discusses the probable causes of unsteadiness in the network and suggests mechanisms to detect an unsteady state. Finally, Sections 4.4 and 4.5 presents and evaluates the proposed state detection algorithm.

3.3 AIMD Characteristics

Section 5.1 reviews the behavior of the basic AIMD algorithm, discusses the issues involved and lays the foundation for TCP Carson. The performance of the AIMD approach is evaluated and the advantages and drawbacks of AIMD are discussed.

3.4 TCP Carson Algorithm

TCP Carson uses an adaptive AIMD approach after detecting the state of the network. It adapts its increase and decrease parameters, depending on the current network state detected. Section 5.2 proposes TCP Carson's extension to basic AIMD. It outlines the features of the TCP Carson algorithm and present the algorithm of adapting AIMD based on detected network state.

3.5 TCP Carson Evaluation

Chapter 6 presents the analysis and evaluation of the TCP Carson protocol under a wide range of network conditions. We analyze the sensitivity of TCP Carson with respect to its parameters. We also present a comparison of TCP Carson, AIMD and TCP Reno.

In Chapter 7 we summarize our thesis work and suggest areas of further development.

3.6 Simulation Testbed

We used *Network Simulator-2* (ns-2) for our thesis work. ns-2 [oCB] is an event-driven network simulator from the University of California, Berkeley useful for simulating local and wide area networks. Numerous network protocols have been implemented in ns-2. We used ns-2 for characterizing network behavior, analyzing TCP and AIMD protocols and for developing and evaluating TCP Carson.

We use a dumb-bell topology (Figure 3.1) to conduct our experiments. *R1* is the bottleneck router to which the senders *(s1, s2 ... sn)* are connected on 2 Mb links with a delay of 10 ms. *R1* has a queue size of 15 and is a drop-tail router. The link *R1—R2* has a capacity of 1 Mb and a delay of 40 ms. Again, the receivers *r1, r2 ... rn* are connected to *R2* on 2 Mb links with a delay of 10ms. Depending on the test scenario, we changed

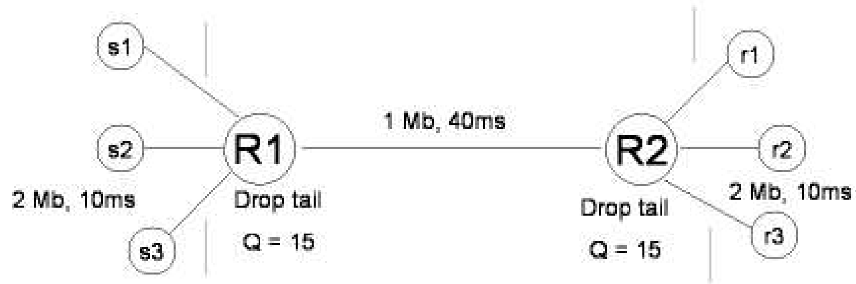


Figure 3.1: Topology for the Simulation

the number of senders and receivers and also the transport protocol used by them. The packets size is set to 1000 bytes in our experiments.

Chapter 4

Network State

There are some enterprises in which a careful disorderliness is the true method.

–Moby Dick, by Herman Melville, 1851

In this chapter we discuss the concept of network state. We define the terms *steady* and *unsteady* and characterize the network behavior in these states. We analyze various mechanisms to detect the current state of the network. Finally, we describe our proposed algorithm to detect network state and evaluate it under different network conditions.

4.1 TCP Behavior

TCP works on the principle that when a flow is experiencing congestion in the network, it has losses. Thereby, it modifies its transmission rate based on loss. The behavior of TCP in the congestion avoidance phase can be characterized by the size of the congestion window and the interval between successive losses.

Figure 4.1 is a plot of congestion window over time for a solo TCP Reno flow on a network. Under such a condition the congestion window (cwnd) oscillates continously

between fixed values, 16 and 32 in this case. This oscillating pattern of *cwnd* results in a pattern for the loss interval, Figure 4.2 in this example. We observe that during this phase the losses are occurring at regular intervals of packets, 450 packets in this case as depicted in Figure 4.2

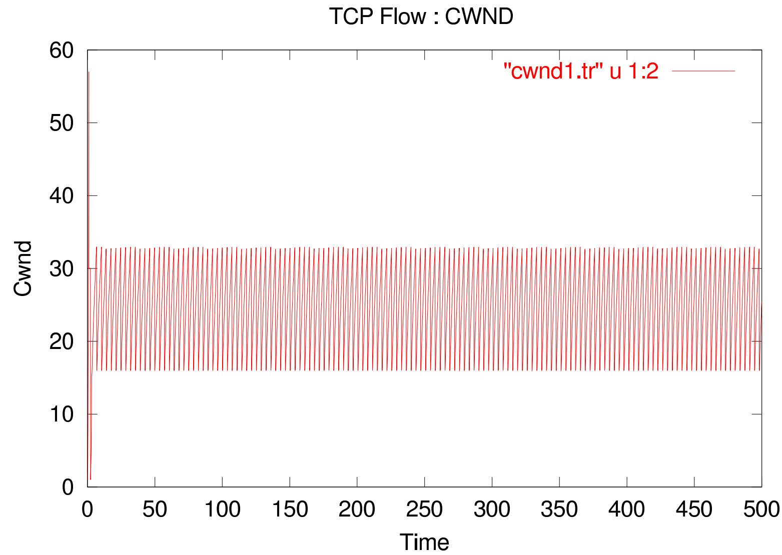


Figure 4.1: Congestion Window in packets :: Single Reno flow

Another, characteristic of the congestion avoidance phase is the number of RTTs between two successive loss events as illustrated in Figure 4.3. We define a *round* to be a RTT worth of packets. Again, we observe that the number of rounds between two successive loss events also follows a fixed pattern. In this case, a loss event is recorded every 28—32 rounds. We call such a state of predictable oscillation of congestion window as *steady state* from the perspective of the flow. A steady network state is characterized by a small variability in the loss intervals, both in terms of packets and in terms of rounds.

When the bandwidth available to a particular flow either decreases or increases significantly, we define the transition period as being *unsteady*. This might happen either because new flows have entered the network or because certain flows have left the network. A period of high-congestion leading to retransmission timeouts is also considered

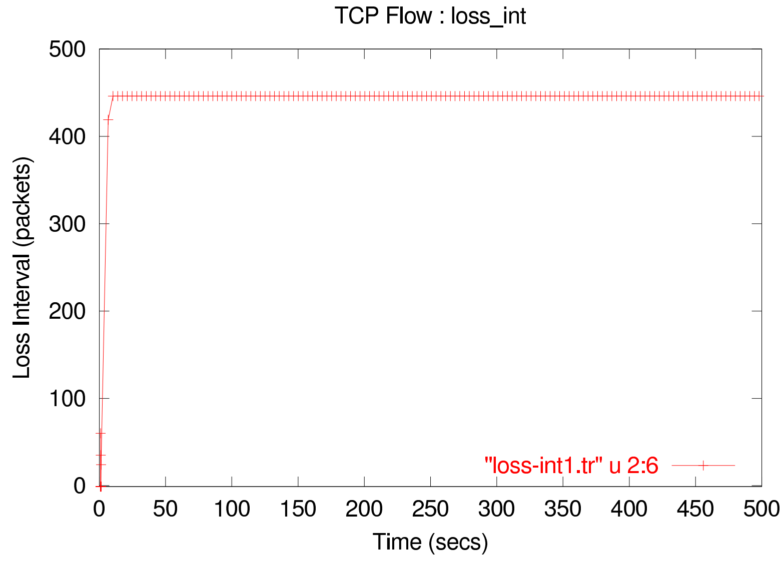


Figure 4.2: Loss interval in packets :: Single Reno fbw

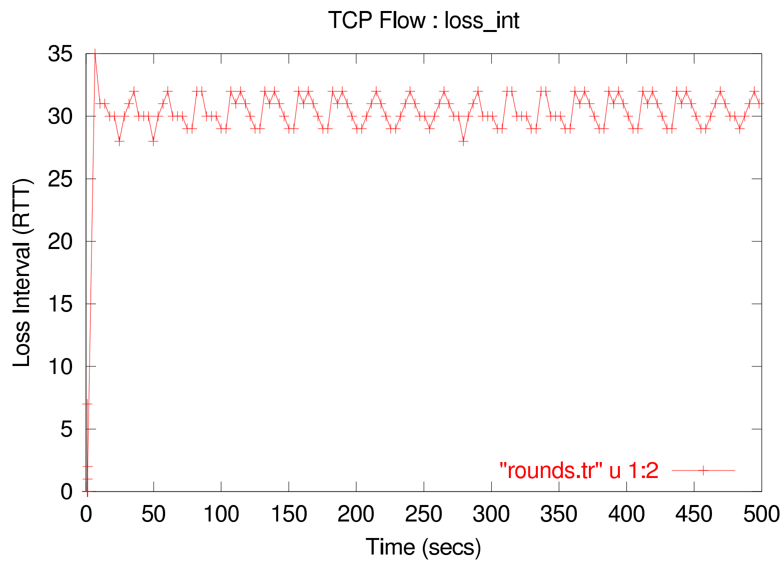


Figure 4.3: Loss interval in RTTs :: Single Reno fbw

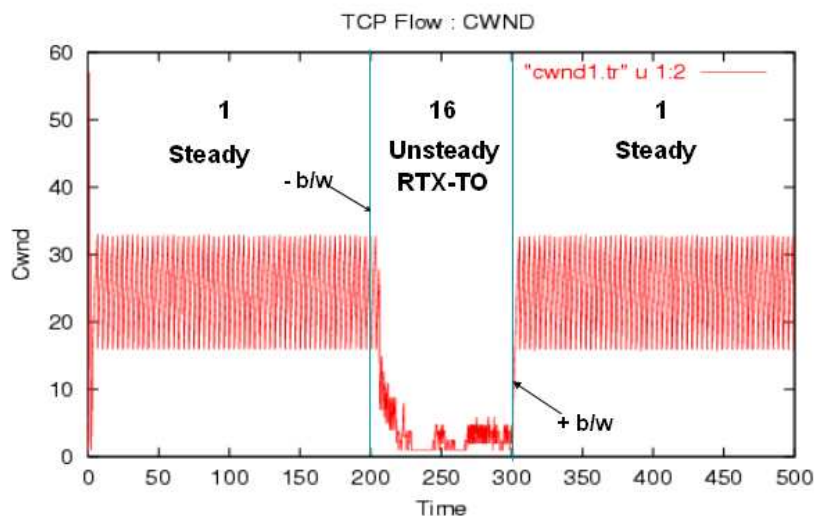


Figure 4.4: Congestion window in packets :: 1-16-1 TCP Reno

as an unsteady state.

Any significant change to the network condition resulting in an unpredictable variance in the congestion window is called an *unsteady state*. There are many causes of unsteadiness in the network:

- *New flows entering the network.* When a number of new flows enter the network, the bandwidth available to the existing flows suddenly decreases.

- *Flows leaving the network.* When flows leave the network, more bandwidth is availble to the remaining flows.

- *Retransmission timeouts.* When a flow experiences a retransmission timeout, its window size is set to 1 and it enters the slow start mode again. Usually this happens when there is a severe network congestion. Other causes include link failure, routing changes and bursty traffic.

To illustrate this concept, we ran a single TCP Reno flow for a period of 500 seconds. During this period, after 200 seconds, we introduced 15 new flows in the network. This

caused a sudden reduction to the bandwidth available for the flow, which is reflected in its congestion window being reduced. (Figure 4.4) The 15 flows were stopped at time 300, causing an increase in the bandwidth and thereby, an increase in the congestion window. During the period from 200—300, we observe that the congestion window remains at one for certain periods. These are retransmission timeouts experienced by the flow. We call this scenario *1-16-1*.

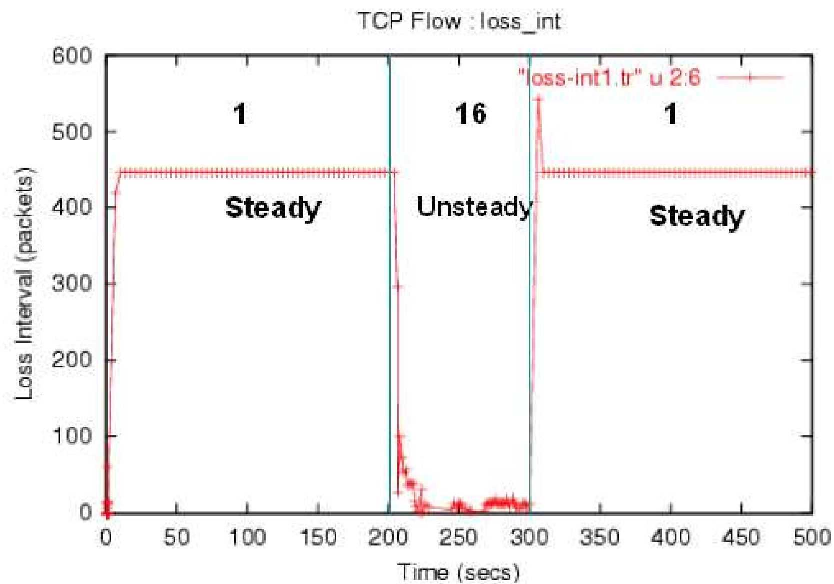


Figure 4.5: Loss interval in packets :: 1-16-1 TCP Reno

The loss interval follows the behavior of the congestion window closely. Figure 4.5 shows that when the congestion window drops at time 200, the losses experienced by the flow arrive more closely; that is the interval between loss events reduces. At time 300, when more bandwidth was available, the loss-interval increased. Thus, a change in the loss interval pattern indicates a transition period (unsteadiness) in the network. Figure 4.6 explores the same concept on the basis of difference in rounds. Again, we observe that the round interval pattern also follows the congestion window pattern closely.

The state of the network can be determined by the following observations:

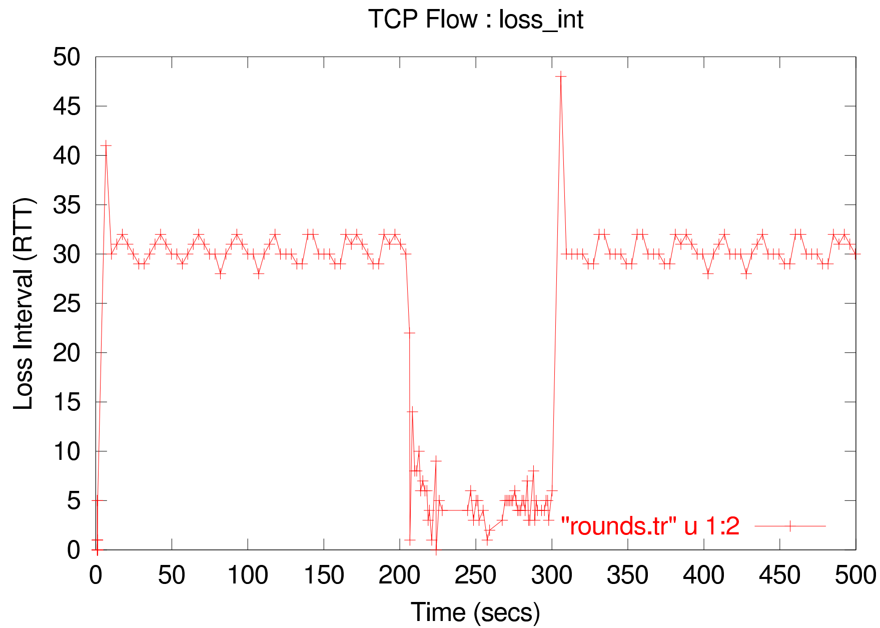


Figure 4.6: Loss interval in RTTs :: 1-16-1 TCP Reno

- if the variability in the loss intervals is small, such as during the period, 0 — 200 in Figure 4.5, the flow is in a steady state.

- if the loss interval pattern changes, time 200 in Figure 4.5 for instance, it implies that the flow is in an unsteady state.

- if the flow experiences retransmission timeouts, such as during the period 200 — 300 in Figure 4.4, the flow is in an unsteady state.

4.2 Steady State

Any metric to detect steady network state should be consistent with the above observations. The instantaneous loss interval pattern cannot be used for detecting the state. This is because transient bandwidth changes have a large effect on the loss interval. Again, the loss intervals may have a fixed oscillation pattern in steady state. While the metric should

be able to detect transitions in available bandwidth, it should also be resilient to transient spikes or valleys in bandwidth. We evaluated two different sets of metrics to detect steady state based on loss-intervals.

- Exponentially Weighted Moving Average (EWMA)
- History Window

4.2.1 EWMA

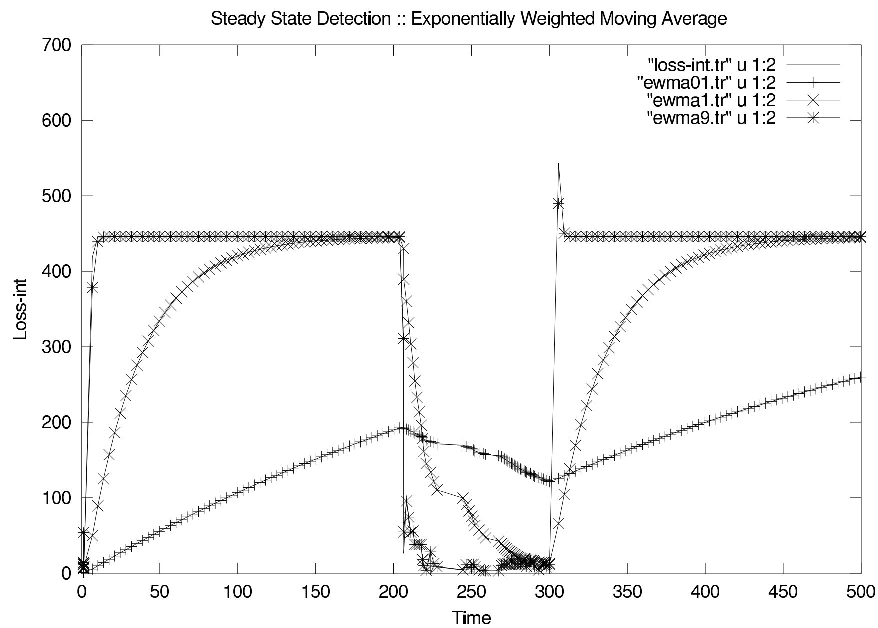


Figure 4.7: Exponentially Weighted Moving Average of loss intervals :: 1-16-1

The Exponentially Weighted Moving Average(EWMA) uses the following formula:

$$ave = ((1-wt) \times ave) + (loss--int \times wt)$$

where, *ave* is the the exponentially weighted moving average, *wt* is the weight and *loss-int* is the instantaneous loss interval value.

We evaluated the EWMA method for the weights — 0.01, 0.1 and 0.9 (Figure 4.7). The EWMA-0.9 follows the loss-interval pattern very closely. While it is successful in determining bandwidth changes, it also reacts considerably to transient spikes. EWMA-0.01 on the other hand does not react fast enough to bandwidth changes. EWMA-0.1 suffers from the drawback that it takes a long time to converge to the loss-interval value. Depending on the weights assigned, this method either puts too much weight on the the most recent loss-interval, or takes too much history into account and is slow to respond to changes. Determining the proper weight to be assigned is a difficult problem and hence, the EWMA method is not suited for detecting the current state of the network. Similar results were noted in [FHPW00].

4.2.2 History Window

In the history window method, we kept track of the most recent n loss intervals for different history sizes (n = 4, 8, 16, 32, 64). We then performed statistical operations on the history window such as mean, median, standard deviation and weighted averages[1] and analyzed them in different scenarios.

Figure 4.8 to Figure 4.12 are plots of mean, median, standard deviation and weighted average for the 1-16-1 scenario with different history sizes. This section analyzes the effect of history size and choice of metric while detecting steady state.

Effect of History Size

As the history size increases, we observe the following effects:

- It takes longer to gather the history of loss intervals.
- The gap between two successive values in every metric decreases.

[1] The weighted average method was proposed in [FHPW00]. We explain it in detail in Section 4.4

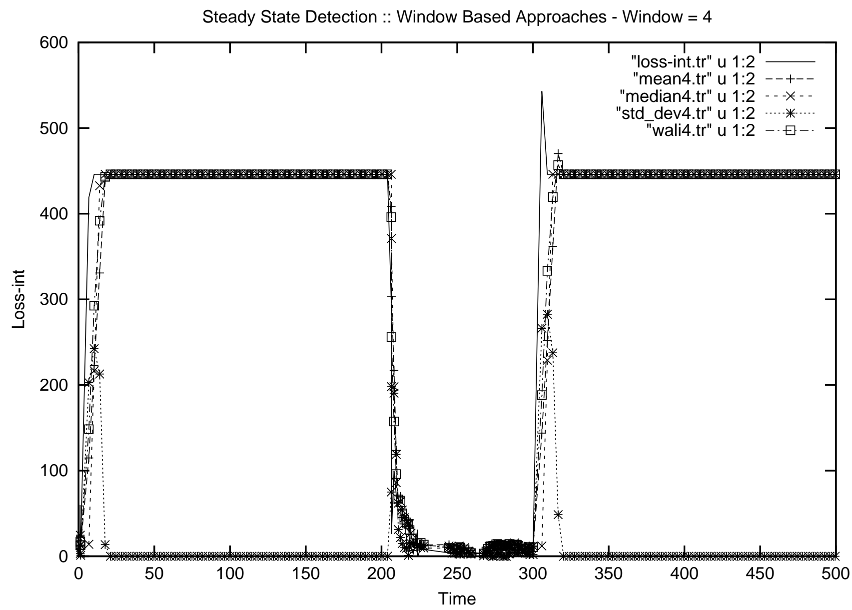


Figure 4.8: Loss interval history size = 4 :: 1-16-1 TCP Reno

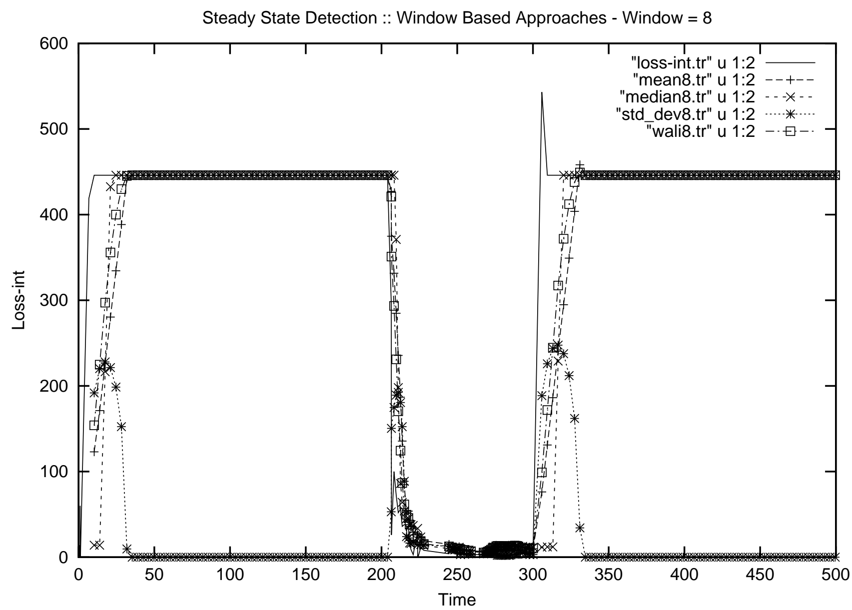


Figure 4.9: Loss interval history size = 8 :: 1-16-1 TCP Reno

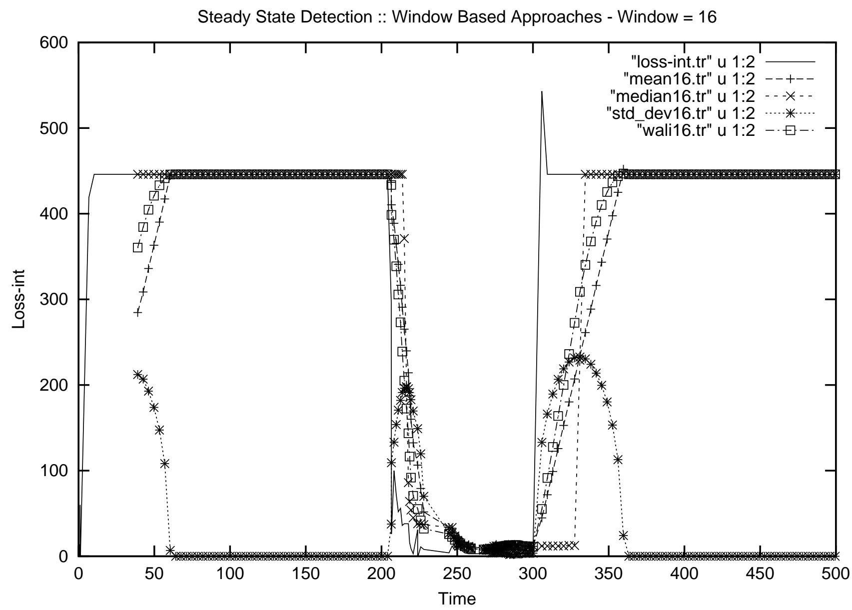


Figure 4.10: Loss interval history size = 16 :: 1-16-1 TCP Reno

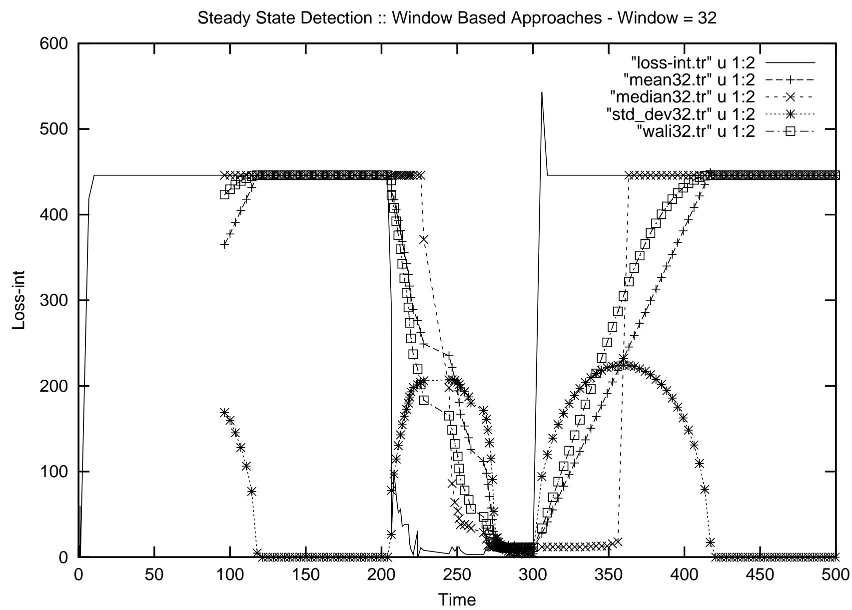


Figure 4.11: Loss interval history size = 32 :: 1-16-1 TCP Reno

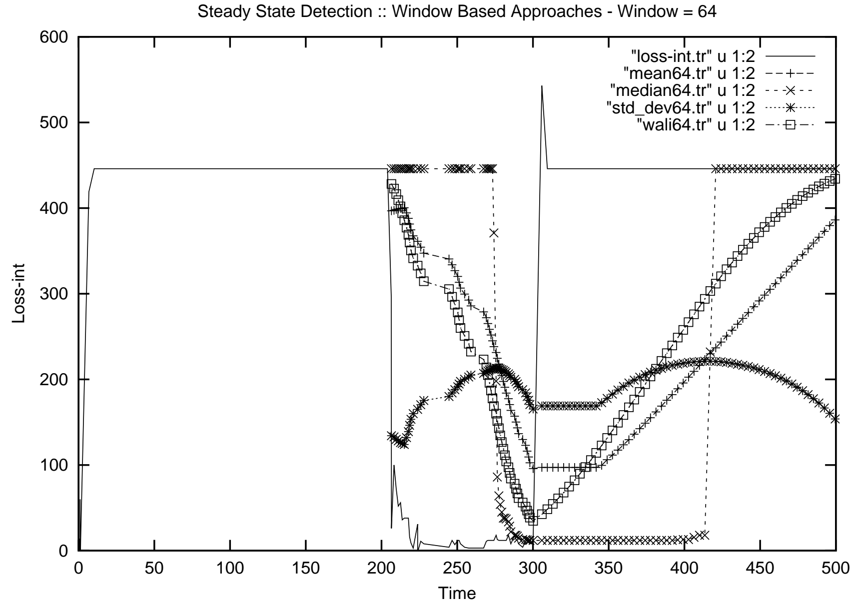


Figure 4.12: Loss interval history size = 64 :: 1-16-1 TCP Reno

- The metrics become more and more resilient to bandwidth changes.

Thus, there exists various trade-offs in choosing the right history size. A small history size has the advantages of a smaller initialization time, responds rapidly to bandwidth changes and has sizeable difference between successive values, but suffers from the drawback of not being resilient to transient congestion and temporary spikes or valleys in available bandwidth. A large history size, on the other hand, is resilient to transient congestion, but is slower in detecting bandwidth changes and requires more initialization time. Empirically, we found that a history of 32 loss intervals was well-suited in most of our test cases, and we use this history size for all further evaluation. An optimal value of the history size is a subject of future study.

Effect of Metric

In this section, we analyze the behavior of the mean, median, standard deviation and the weighted average of the loss interval history. The weighted average loss interval (WALI)

to detect loss-rate was proposed in [FHPW00]. For a given history size, n, the weighted average of the loss interval (WALI) is calculated using exponential weights (w_i) given by:

$$w_i = 1 \qquad 1 \leq i \leq n/2$$

$$w_i = 1 - \frac{i - n/2}{n/2 + 1} \qquad n/2 \leq i \leq n$$

Analysis of Figure 4.11 suggests the following characteristics of the metrics:

- The *standard deviation* is the fastest to detect bandwidth changes. While this is advantageous in the case of a change in bandwidth, transient spikes and valleys in the bandwidth have a significant effect on the standard deviation.

- The median has a sharp transition to a change in bandwidth. It is also the most resilient to transient spikes and valleys in available bandwidth since it ignores the outliers. However it is the slowest in detecting bandwidth changes.

- The *mean* and the *weighted average* perform reasonably well, both, in detecting network changes and in being resilient to transient changes in bandwidth. The difference between the mean and the weighed average is that the mean gives equal weights to all values in the window, while the weighted average gives an exponentially decreasing weight to older values. The weighted average, thereby, gives more importance to the instantaneous network state than to older values. The advantage of the weighted average over the mean is that a transient spike or valley will affect its value for a shorter duration.

Taking into consideration the above observations, we chose the weighted average method to detect the state of the network. We decide on the state of the network using the difference between two consecutive values of the weighted average. We explain this method in further detail in Section 4.4.

4.3 Unsteady State

There are three causes for an unsteady network state:

- Reduction in a flow's share of bandwidth
- Retransmission timeout
- Increase in a flow's share of bandwidth

In this section, we consider each cause, understand the resulting behavior and present a mechanism to detect the unsteadiness.

4.3.1 Reduction In Bandwidth

When new flows enter a network, the bandwidth available to each flow reduces. This results in a reduction in the cwnd size. The effect of such a change is observed in the loss interval pattern. The losses occur more frequently during the reduction in bandwidth, that is, the loss-interval and the WALI reduces. Such a transition results in a temporary unsteadiness. The difference in the WALI values can be used to detect such a reduction in bandwidth.

4.3.2 Retransmission Timeout

A period of high congestion leading to retransmission timeouts results in an unsteady state. During this state, the cwnd is set to 1, the flow goes into slow-start mode and starts probing for bandwidth again. The expiration of the retransmission timer signals such an unsteady state.

4.3.3 Increase In Bandwidth

When flows leave the network, there is an increase in the available bandwidth for the remaining flows. Under such a condition the congestion window size increases. The losses start occurring more frequently and the loss interval and the WALI increases. Hence, the WALI can be used to detect such a transition. The drawback of using this approach is that the flow needs to report a loss before the WALI changes. In some cases this might result in a significant lag, from the time the bandwidth transition occurs until the time it is actually detected. Therefore, we use a RTT based timer to detect this transition. As explained in Section 4.1, the difference in rounds between consecutive losses follows the loss-interval pattern. We use the difference between consecutive round intervals to set a timer for detecting this transition. If a particular flow does not experience a loss for x number of rounds then we decide that it is an unsteady state. x is determined by multiplying a factor by the difference in rounds for the previous loss interval. There exists a tradeoff in choosing the value of the multiplication factor. If x is set too large than it will take longer to detect a transition in the network; if set too small than transient spikes in bandwidth and variations in the loss intervals can be potentially detected as unsteady. Again, a history can be used to further tune the value for x. In our evaluation we used the following value for x.

$$x = 1.75 \times r$$

where, r is the round interval corresponding to the previous loss-interval.

When a bandwidth transition occurs, the history of loss intervals is no longer reflective of the current network state. When a flow has retransmission timeouts, it goes back into slow-start mode, making the history obsolete. Hence, whenever an unsteady network state

is detected, we reset the history window.

4.4 Weighted Average Loss Interval

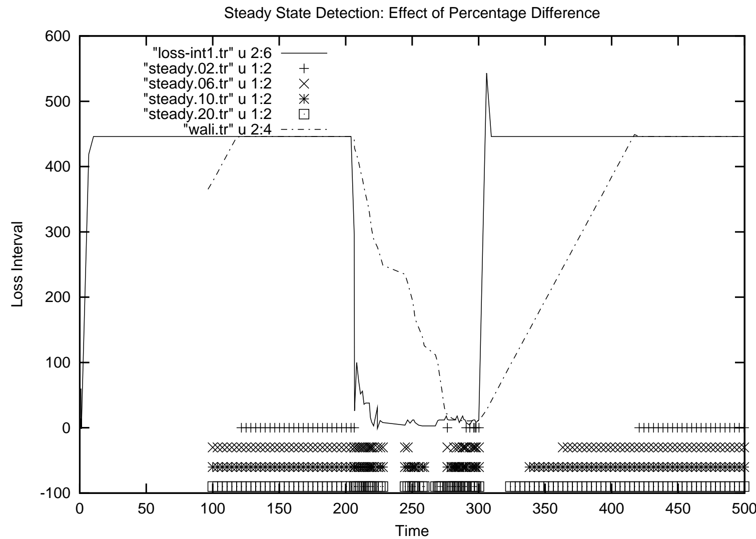


Figure 4.13: Steady state detection using WALI : Effect of allowed variance

For detecting the state of the network, we use the difference (in percentage) between two successive WALI values. Figure 4.13 plots steady state detection and Figure 4.14 plots the unsteady state detection for a range (2%, 6%, 10% and 20%) of allowed differences.

As the allowed variation in WALI values increases, it takes less time to detect steady state. As seen in Figure 4.13, after the 15 flows leave the network at time 300, for an allowed variation of 2%, steady state is detected at time 420; for an allowed variation of 6%, steady state is detected at time 360; for an allowed variation of 10%, steady state is detected at time 340; and for an allowed variation of 20%, steady state is detected at time 320. Conversely, as the allowed variation in WALI values increases, it takes more time to detect an unsteady state. A higher allowed variation is resilient to a larger spike or

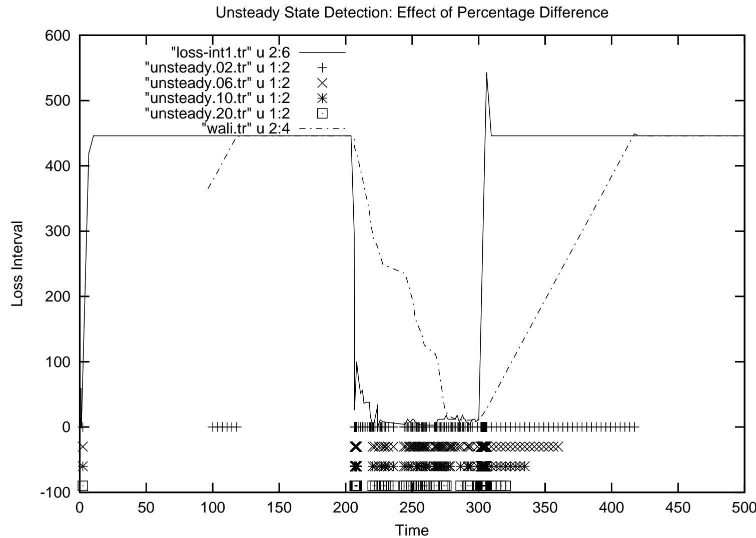


Figure 4.14: Unsteady state detection using WALI: Effect of allowed variance

valley in bandwidth; however, it takes longer to detect an increase or decrease in available bandwidth. For our evaluation, we chose a 10% difference as the threshold for steadiness. We evaluate the sensitivity of the allowed variation in Chapter 6.

4.5 State Detection

In this section we present the State Detection Algorithm that decides on the current state of the network. We also present the evaluation of the steady state detection algorithm in various scenarios.

4.5.1 Algorithm : WALI-32

There are two parts to the state detection algorithm:

- Receiver Functionality: The receiver detects a loss using a gap in sequence numbers. It considers all losses in the same round together as a single loss-event. It calculates the loss interval and embeds it in the acknowledgment to the sender.

- Sender Functionality: For every packet sent to the receiver, the sender includes the round number in the header. The sender increments the round number using a RTT timer. For every incoming packet, the receiver considers two cases:
  - If a loss is reported, then the sender calculates the WALI using the loss interval value sent by the receiver. It then detects the state of the network as described in Section 4.4.
  - If no loss is reported, then the sender checks whether it has been long enough to indicate an increase in bandwidth. If so, it decides that it is an unsteady network state.

  If an unsteady state is detected the history window is reset.

The complete algorithm for detecting the state of the network is presented in Figure 4.15.

```
Receiver Functionality:
   For each incoming packet
        if (gap in sequence numbers && different round) {
             calculate loss interval
             report loss interval in acknowledgment
        }

Sender functionality:
   For each outgoing packet
        calculate round number
        report round number to receiver in header

   For each incoming acknowledgment
        if (loss interval reported by receiver) {
             calculate WALI
             detect network state using difference in WALI
             enter the detected network state (steady/unsteady)
        }
        else if ( no loss for long time) {
             enter unsteady state
        }
        if (unsteady network state detected) {
            reset history window
        }
```

Figure 4.15: State Detection Algorithm : WALI-32

4.5.2 Evaluation

In this section we evaluate the state detection algorithm in various scenarios.

10 TCP Reno flows

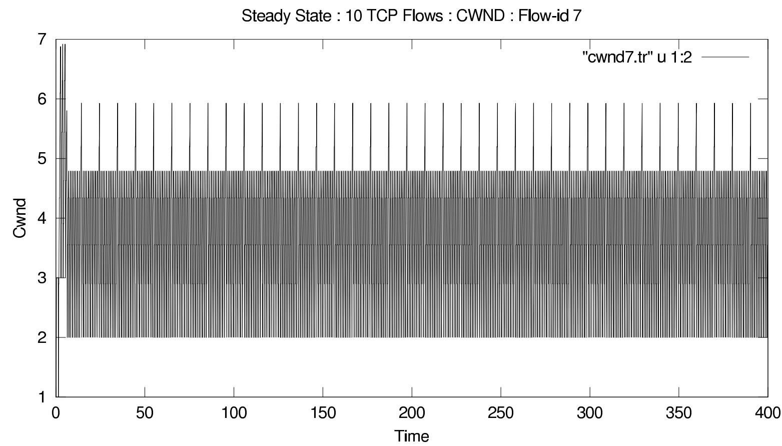


Figure 4.16: Congestion window for fbw 7 among 10 competing TCP Reno fbws

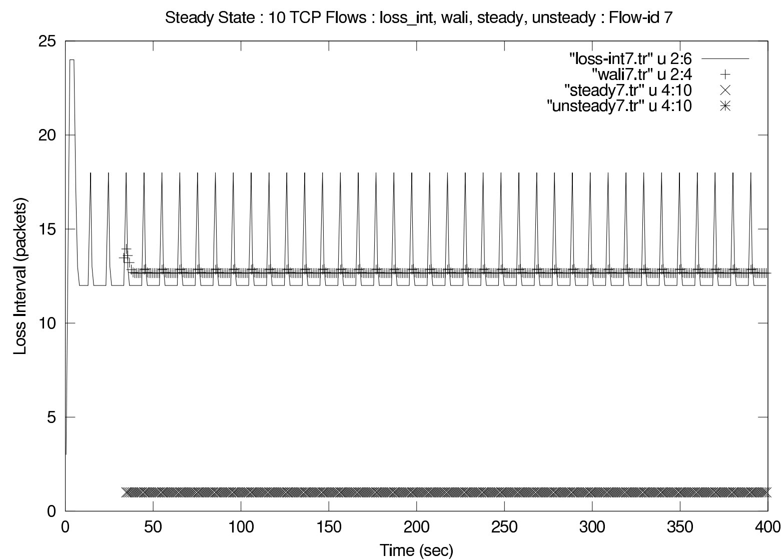


Figure 4.17: Loss interval and state detection for fbw 7 among 10 competing TCP Reno fbws

We ran a simple test case of 10 TCP Reno flows for 500 seconds. Figure 4.16 plots

the congestion window versus time and Figure 4.17 plots the state detection for one flow. We observe that the congestion window follows a fixed oscillation pattern between 2 and 6 packets for the entire duration. This oscillation is reflected in the loss interval pattern which oscillates between 12 and 18 packets. The weighted average loss interval remains close to 13 for the entire duration. As a result, the state detection algorithm detects the entire period as steady state, as expected.

4-8-16-8-4 TCP Reno flows

To evaluate how the state detection algorithm performed under changing network conditions we next evaluated it in the following scenario. 4 TCP Reno flows ran for a period of 875 seconds. At time 175, we introduced 4 more flows into the network which ran for 525 seconds. At time 350, 8 more flows were introduced into the network, making the number of flows equal to 16. These 8 flows were stopped at time 525. We call this scenario *4-8-16-8-4*.

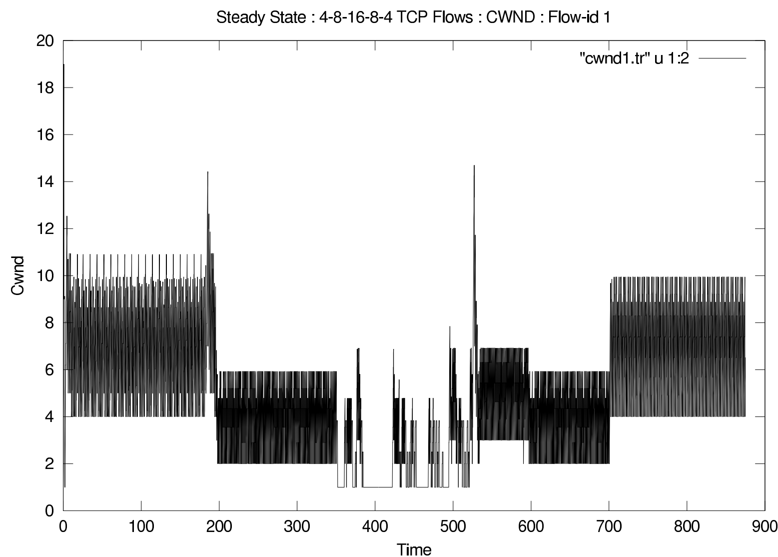


Figure 4.18: Congestion window for flow 1 :: 4-8-16-8-4 TCP Reno flows

Figure 4.18 is a plot of the cwnd versus time for Flow 1 which lasted the entire 875

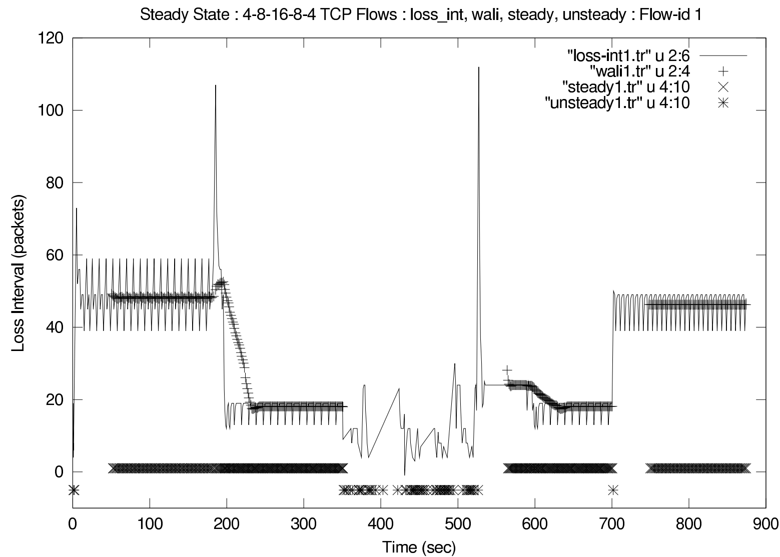


Figure 4.19: Loss interval and state detection for fbw 1 :: 4-8-16-8-4 TCP Reno fbws

seconds. Figure 4.19 is the corresponding plot of the loss interval and the state detection algorithm. The weighted average values are plotted on top of the loss interval values. We observe that the weighted average follows the trend of the loss interval values. The series of marks corresponiding to 0 on the y-axis are instances of time when steady state is detected and the marks corresponding to y-axis of -5 are instances when an unsteady state is detected.

During the period from 30 to 175, when there are only 4 flows in the network, steady state is detected. For instance, the mark at (50,0) indicates a steady state. We observe that when the number of flows increases from 4 to 8, the weighted average loss interval does not change significantly enough and an unsteady state is not detected. However, when the number of flows increases to 16, an unsteady state is detected. The mark at (350, -5) corresponds to this unsteadiness. During the period when there were 16 flows in the network, Flow 1 experiences retransmission timeouts which are detected as unsteady state. The mark at (375, -5) is an instance of this case. When the 16 flows leave the network, Flow 1 detects a steady network state. The mark at (560, 0) indicates steadiness

in the network. Again, when the number of flows reduced from 8 to 4, the state detection algorithm detects a change in available bandwidth resulting in an unsteady state (700, -5). Finally, when there are only 4 flows in the network, Flow 1 is able to detect a steady state (750, 0).

4.6 Conclusion

The loss interval pattern follows the transition in available bandwidth closely. We use a loss-interval history based approach to detect the state of the network. The state detection algorithm uses the following parameters:

- The size of the history window required to detect state of the network.
- Number of rounds of no loss to detect transition in bandwidth.
- Allowed variance in loss interval.

Our results indicate that this algorithm is effective in detecting the current state of the network.

Chapter 5

TCP Carson

A capacity to change is indispensable. Equally indispensable is the capacity to hold fast to that which is good.

– John Foster Dulles, 1888-1959

TCP Carson is a hybrid of the Additive Increase, Multiplicative Decrease (AIMD) algorithm and the TCP Reno algorithm. In Chapter 1, we discussed the probing problem of TCP, that is under a steady network state, TCP continues to probe for bandwidth aggressively, thereby causing loss and variation in transmission rate. TCP Carson detects the state of the network using the algorithm presented in Chapter 4. It adapts the increase and the decrease parameters of AIMD according to the state of the network detected. In the first part of this chapter we discuss the drawbacks of the AIMD algorithm. We then present the TCP Carson adaptive AIMD algorithm.

5.1 AIMD Behavior

[FHP00] indicates that the TCP(1,0.5) approach may be unnecessarily severe and suggests other values for a and b governed by the equation,

$$a = \frac{3b}{2-b}$$

where a is the window increase on successful delivery of a round trip worth of packets, and (1-b) is the multiplicative window decrease on encountering a packet loss[1]. By reducing the value of a, AIMD probes less aggressively for available bandwidth. Also, by decreasing b, AIMD responds less rapidly to packet loss.

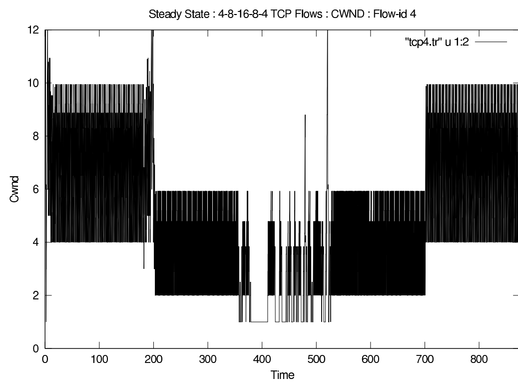
While this is beneficial when in network steady state, under an unsteady network condition, TCP is more fair then AIMD, since it reduces its transmission rate more than AIMD on a packet loss. It is observed that AIMD blocks out new flows entering the network. TCP probes for new bandwidth more aggressively than AIMD. Hence, when flows leave the network TCP reaches the steady transmission rate faster than AIMD.

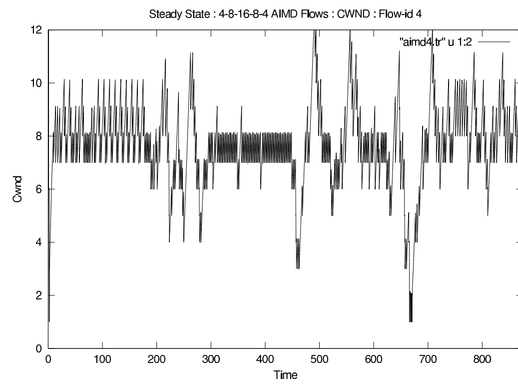
To illustrate the behavior of AIMD, we ran the *4-8-16-8-4* scenario with AIMD (0.148, 0.125)[2] flows and TCP(1, 0.5) flows respectively. Figures 5.1 depicts comparative plots of the congestion windows for flows 4, 5 and 9. (Flows 1—4 lasted the entire 875 seconds, Flows 5—9 lasted for the middle 525 seconds and Flow 9—16 lasted the middle 175 seconds.) We make the following observations from the graphs:

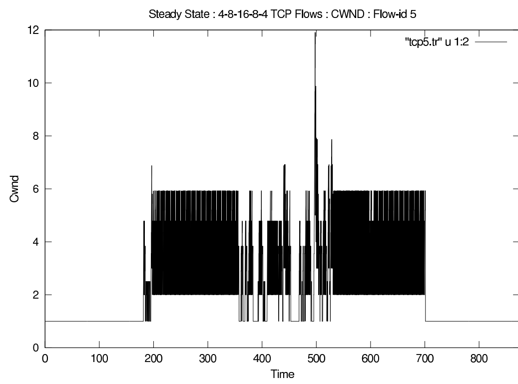
- TCP responds rapidly to bandwidth changes. The congestion window of Flow 4 drops reduces substantially when new flows enter the network at time 175. Again, when flows leave the network, congestion window of flow 4, rapidly increases at time 700. TCP is fair in the sense that over extended periods of time, TCP flows

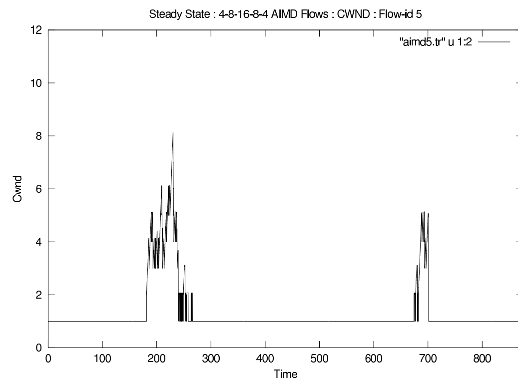
[1] Other equations between a and b have been also been suggested. We present a preliminary discussion in Section 2.2

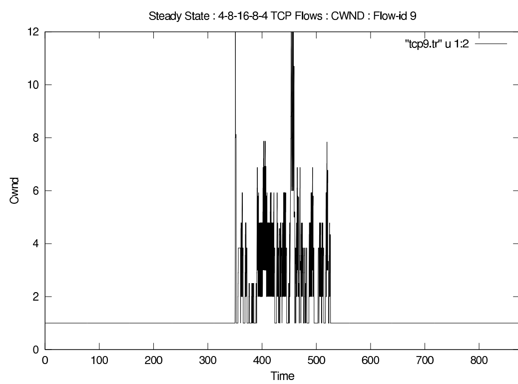
[2] The choice of these values is discussed in Section 5.2

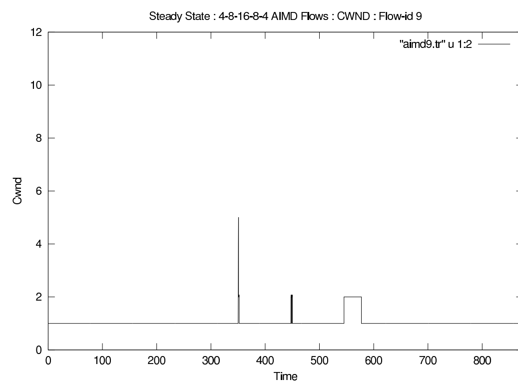












TCP AIMD

Figure 5.1: TCP vs AIMD : Congestion Windows for fbws(4,5,9) :: 4-8-16-8-4

share the bandwidth equally. As can be seen by the plots of the congestion windows for flows 4 and 5, the congestion window of both the flows oscillate between 2 and 6 for the period 175—350 and 525—700.

- AIMD(0.148, 0.125) is slow in grabbing available bandwidth. Hence, AIMD flows take much longer to reach steady transmission rate. In this experiment, flows 5—9 do not reach a steady transmission rate at all.
- AIMD(0.148, 0.125) flows do not reduce their transmission rate substantially to packet drops. As a result of this, when new AIMD flows enter the network, the existing flows do not give up their bandwidth, thus blocking out the new flows. Here flows 5 and 9 are blocked out for substantial periods of time, while flow 4 continues to transmit at a high rate. This makes AIMD unfair in some cases.

5.2 Algorithm : Adaptive AIMD

TCP Carson tries to benefit from the behavior of both TCP and AIMD. While in unsteady network state, it behaves like TCP - aggressively probing for available bandwidth and responding rapidly to congestion. However, on detecting steady state it gradually decreases its probing and tries to attain a steady transmission rate with fewer drops. The TCP Carson sender maintains a table (Table 5.1) of corresponding a and b values. We evaluated the performance of various values of a and b and found out that these values are TCP-friendly.

TCP Carson starts by behaving like TCP Reno, index 1 of the table, setting its a and b values to 1 and 0.5 respectively. It uses the WALI-32 (Section 4.5) algorithm to detect the state of the network. On detecting steady state, it sets a and b to the next index value in the AIMD table, until it reaches the last index on the AIMD table. Once it reaches the last index, it uses those values for a and b until it encounters an unsteady state. On detecting

unsteadiness, the sender resets the a and b values to 1 and 0.5 respectively, thus behaving like TCP. We call this algorithm *Adaptive AIMD* (AAIMD).

On detecting steady state, TCP Carson resets 16 of the least recent values in the history window, thus requiring 16 more losses before it detects steady state again. This is because the loss interval pattern changes for every pair of a and b values. TCP Carson waits to gather enough history before it tries to detect steady state again. However, during this period it continues to check for unsteadiness.

On detecting unsteady state, it is not necessary for the Adaptive AIMD algorithm to reset the loss interval history, since the state detection algorithm resets the entire history every time it detects an unsteady state. The Adaptive AIMD algorithm is presented in Figure 5.2

```
Sender Functionality:
   Enter current network state using WALI-32 algorithm (Section 4.5.1)
   if ( in steady state ) {
        decrease a and b using AIMD table (Table 5.1)
        reset 16 least recent values in loss interval history
   }
   if( in unsteady state ) {
        reset a and b to 1 and 0.5 respectively (Table 5.1)
   }
```

Figure 5.2: TCP Carson Algorithm : Adaptive AIMD

In the next chapter we present the evaluation and analysis of TCP Carson under a range of test conditions.

Index	a	b
1	1.000	0.500
2	0.750	0.400
3	0.250	0.250
4	0.148	0.125

Table 5.1: AIMD Table

Chapter 6

Evaluation

Now, what I want is Facts.

–Hard Times, by Charles Dickens, 1854

In this chapter we present a detailed evaluation of TCP Carson. Section 6.1 shows the behavior of TCP Carson under a range of network conditions. For detecting the state of the network, TCP Carson uses the difference between two succesive weighted average values, as discussed in Section 4.4. Section 6.2 presents the sensitivity of the TCP Carson protocol to the percentage difference. Section 6.3 contrasts TCP Carson with TCP Reno and AIMD.

6.1 TCP Carson Behavior

In this section we present the behavior of TCP Carson. We evaluated TCP Carson under a range of test cases. Section 6.1.6 presents the comparison of throughput, loss-interval and average congestion window size of TCP Carson and TCP Reno in our test cases.

6.1.1 1 TCP Carson

A single TCP Carson flow was allowed to run for 500 seconds. Figure 6.1 is a plot of TCP Carson's congestion window across time and Figure 6.2 plots its loss interval pattern.

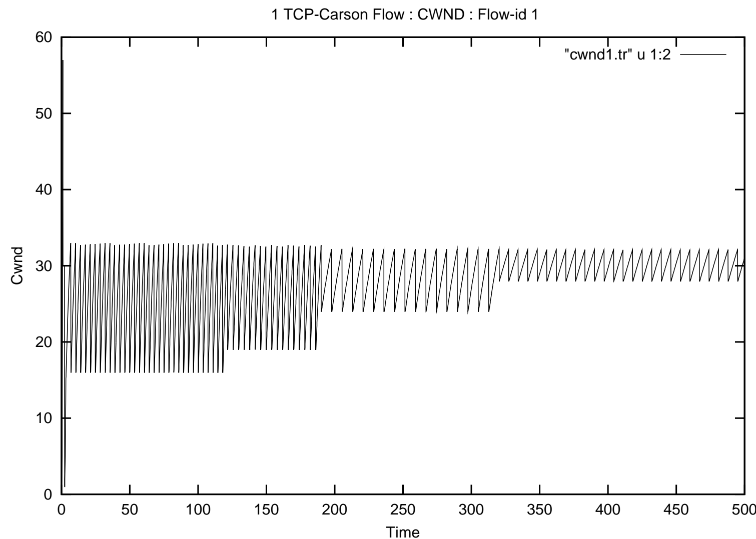


Figure 6.1: Congestion Window for a Single TCP Carson Flow

As seen in Figure 6.2, steady state was first detected at time 120. TCP Carson reduced its a and b values. This resulted in a reduced variance in the congestion window as seen in Figure 6.1 between time 120—185. The loss interval increased from 450 to 500 at time 125 corresponding to this reduced variance. Steady state is again detected at times 185, 325 and 440. As expected, there was no unsteady state detection during the entire run. Every time it detected a steady state, TCP Carson waited to gather history before it detected steady state again. We can observe that TCP Carson, on detecting steady state, reduces its congestion window variance. The loss intervals are farther apart, indicating that TCP Carson encounters fewer losses. (Table 6.1)

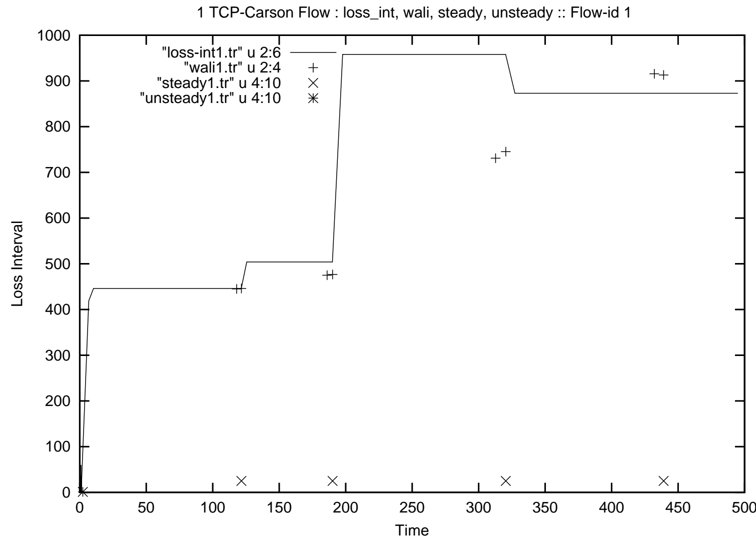


Figure 6.2: Loss Interval (in packets) for a Single TCP Carson Flow

6.1.2 1 TCP Carson, 1 CBR

To illustrate the behavior of TCP Carson under unsteady state, we ran a TCP Carson flow along with a constant bit rate (CBR) flow for 1500 seconds. The CBR flow was stopped at time 500 and restarted at time 1000, thus forcing unsteadiness while the TCP Carson flow lasted for the entire 1500 seconds. The CBR flow was sending data at the rate of 0.4 Mb. (The bottleneck link was 1 Mb.)

Figure 6.3 and Figure 6.4 are plots of the congestion window and loss interval pattern for the TCP Carson flow. We observe that TCP Carson was able to detect unsteadiness whenever there was a bandwidth transition due to the stopping and starting of the CBR flows. TCP Carson reset its *a* and *b* values to 1 and 0.5 when it detected the unsteadiness and behaved like TCP until it gathers enough history to detect steady state. On detecting steady state, TCP Carson reduced its window size variance. Thus TCP Carson adapts its increase and decrease parameters to the state of the network detected. We ran the same test case with a TCP Reno flow instead of the TCP Carson flow. Table 6.2 compares the throughput, loss interval and average congestion window size of TCP Carson with TCP

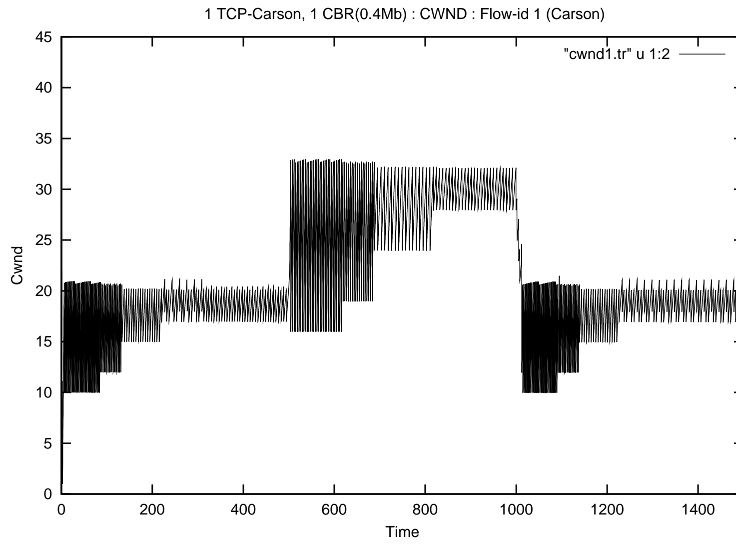


Figure 6.3: Congestion Window for TCP Carson fbw :: 1 TCP Carson with 1 CBR (0.4 Mb)

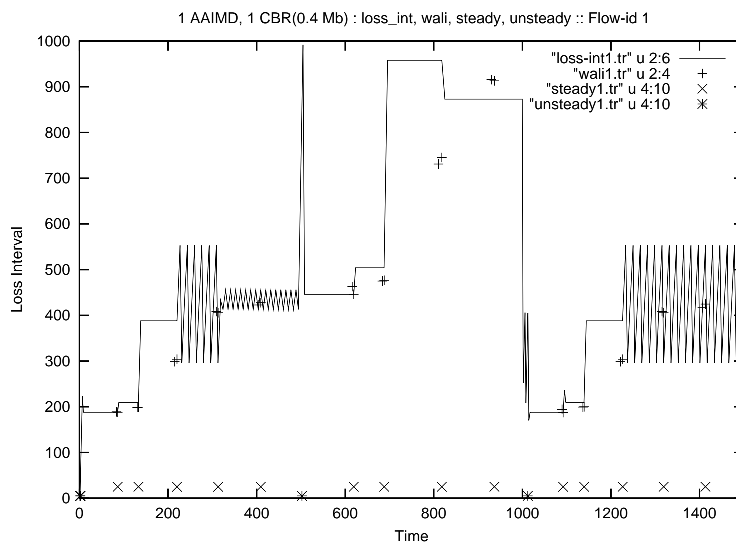


Figure 6.4: Loss interval for TCP Carson fbw :: 1 TCP Carson with 1 CBR (0.4 Mb)

Reno.

6.1.3 1 TCP Carson, 1 TCP Reno

Next, we ran TCP Carson along with TCP Reno for 400 seconds to understand the mutual effect of TCP Reno's congestion window variance and TCP Carson's adaptive window variance.

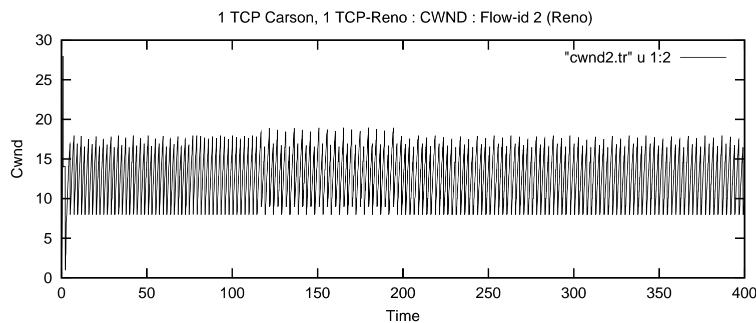


Figure 6.5: Congestion Window for TCP Reno Flow :: 1 TCP Carson with 1 TCP Reno

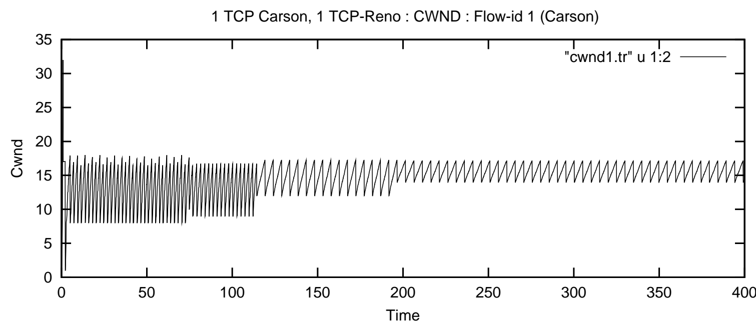


Figure 6.6: Congestion Window for TCP Carson Flow :: 1 TCP Carson with 1 TCP Reno

Figure 6.5 and Figure 6.6 plot the congestion window for the TCP Reno and the TCP Carson flow respectively. Figure 6.7 plots the loss interval pattern of TCP Carson. We observe that both TCP Reno and TCP Carson continue their default behavior. While TCP Reno oscillates its congestion window for the entire duration, TCP Carson gradually

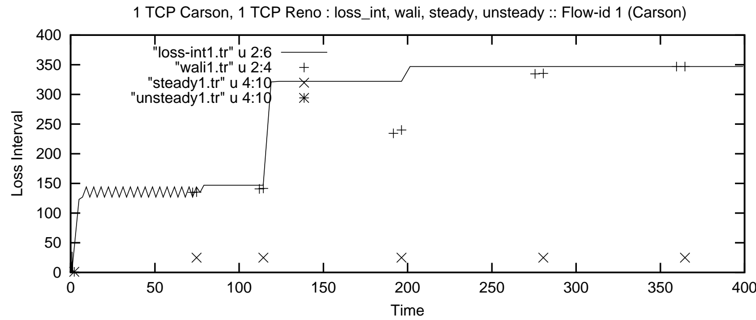


Figure 6.7: Loss interval for TCP Carson Flow :: 1 TCP Carson with 1 TCP Reno

decreases its congestion window variance. TCP Carson thus achieves a higher average transmission rate as compared to TCP Reno, while reducing its loss rate and window size variance as seen from Table 6.3.

6.1.4 4 TCP Carsons, 4 TCP Renos

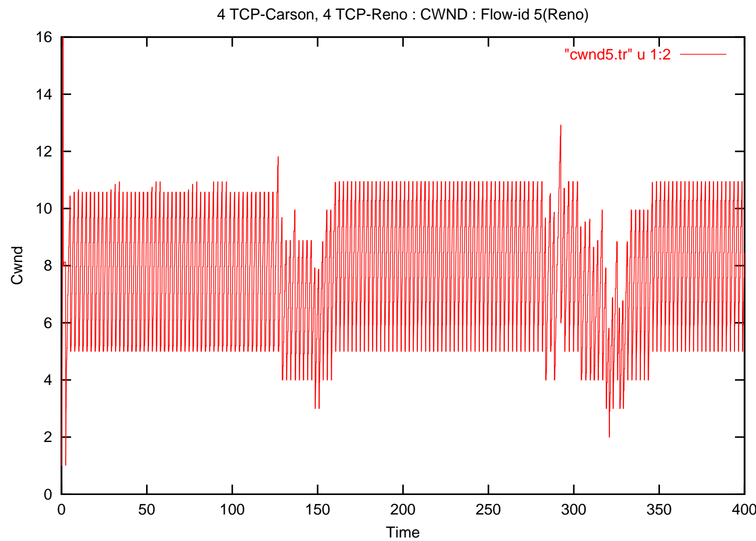


Figure 6.8: Congestion Window for TCP Reno fbw :: 4 TCP Carson with 4 TCP Reno

To evaluate the performance of TCP Carson and TCP Reno in a mix of flows, we ran 4 TCP Carson flows with 4 TCP Reno flows for 400 seconds. Figure 6.9 plots the

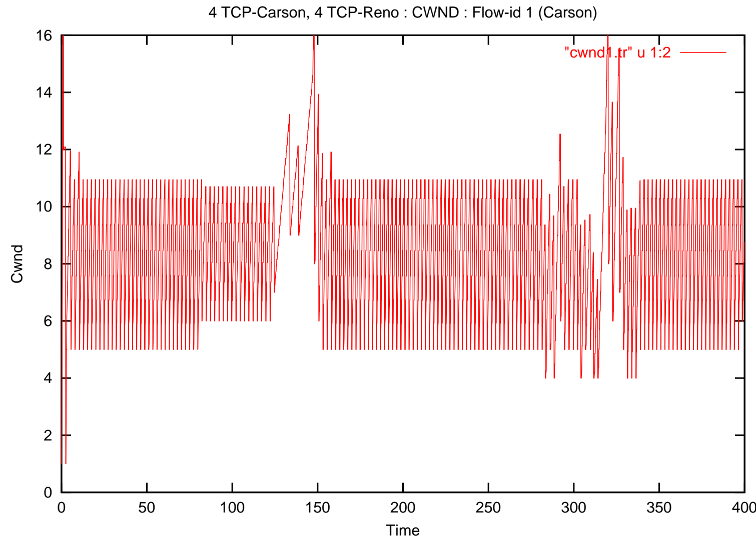


Figure 6.9: Congestion Window for TCP Carson flow :: 4 TCP Carson with 4 TCP Reno

congestion window for a TCP Carson flow and Figure 6.8 plots the congestion window for a TCP Reno flow. The bottleneck bandwidth was 1.5Mb in this case. We observe that while TCP Carson is able to detect steady state, once it changes its a and b values, the congestion window no longer oscillates in a fixed pattern. This is because TCP Carson and TCP Reno do not respond equally to a packet loss in steady state. As a result, TCP Carson detects this as an unsteady state and reverts back its a and b values to 1 and 0.5 respectively. Thus, unlike AIMD, which would cause unfairness in this scenario, TCP Carson starts behaving like TCP Reno sharing bandwidth equally with the other flows. Table 6.4 compares the throughput, the average loss interval and the average congestion window size of the TCP Carson flows with the TCP Reno flows.

6.1.5 8 TCP Carsons

We ran 8 TCP Carson flows for a period of 400 seconds in this experiment to study the performance of multiple TCP Carson flows. All the flows detected steady state for the entire period and reduced their congestion window variance. Figure 6.10 plots the

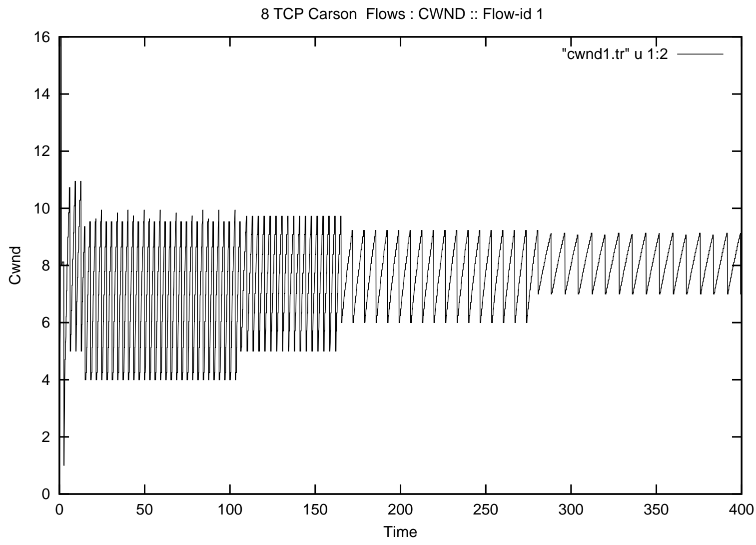


Figure 6.10: Congestion Window for TCP Carson fbw :: 8 TCP Carson

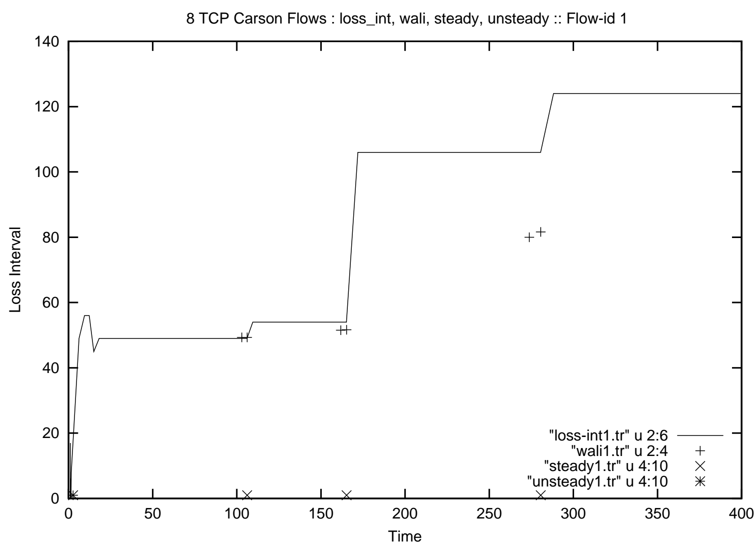


Figure 6.11: Loss interval for TCP Carson fbw :: 8 TCP Carson

	TCP Carson	TCP Reno
Avg. Cwnd. (packets)	27.78	24.80
Avg. Throughput (Mb)	0.948	0.944
Avg. Loss Interval (packets)	647.25	455.29
TCP Response Function (Mb)	1.078	0.950

Table 6.1: 1 TCP Carson flow vs 1 TCP Reno flow

congestion window for a particular flow and Figure 6.10 plots the corresponding loss interval pattern. We ran the same test with 8 TCP Reno flows instead of TCP Carson flows. Table 6.5 compares the performance of TCP Carson with TCP Reno. TCP Carson has a marked improvement over TCP Reno in terms of overall throughput and loss.

6.1.6 Summary Results

The TCP response function[FHPW00] (TRF) gives an upper bound on the transmission rate of a transport protocol for a given loss interval. In its simplified form,

$$T = \frac{\sqrt{1.5}s}{R\sqrt{p}}$$

where T is the transmission rate in megabits/second, s is the packet size in bits, emphR is the round trip time in seconds and p is the loss event rate. R is the average round trip for all the packets over the entire run. All losses that occur in the same round trip time are together considered as one loss event. The loss interval is calculated as the difference between two consecutive loss events. The loss event rate (p) is the reciprocal of the average loss interval for the entire run.

Table 6.1 to Table 6.5 contrasts the average congestion window size, average transmission rate, average loss interval and the TCP friendly throughput for TCP Carson with TCP Reno for the above experiments. We observe that:

- The average congestion window of TCP Carson is higher than that of TCP Reno in

	TCP Carson	TCP Reno
Avg. Cwnd (packets)	21.16	18.56
Avg. Throughput (Mb)	0.706	0.655
Avg. Loss Interval (packets)	418.01	320.23
TCP Response Function (Mb)	0.866	0.758

Table 6.2: TCP Carson in combination with CBR(0.4Mb) vs TCP Reno in combination with CBR(0.4Mb)

	TCP Carson	TCP Reno
Avg. Cwnd (packets)	14.64	12.68
Avg. Throughput (Mb)	0.518	0.437
Avg. Loss Interval (packets)	240.23	140.33
TCP Response Function (Mb)	0.657	0.502

Table 6.3: 1 TCP Carson flow along with 1 TCP Reno flow

	TCP Carson	TCP Reno
Avg. Cwnd (packets)	8.7	7.7
Avg. Throughput (Mb)	0.189	0.167
Avg. Loss Interval (packets)	60.00	50.54
TCP Response Function(Mb)	0.328	0.301

Table 6.4: 4 TCP Carson flows along with 4 TCP Reno flows

	TCP Carson	TCP Reno
Avg. Cwnd (packets)	8.1	7.0
Avg. Throughput (Mb)	0.121	0.109
Avg. Loss Interval(packets)	73.38	49.47
TCP Response Function (Mb)	0.363	0.298

Table 6.5: 8 TCP Carson flows vs 8 TCP Reno flows

all the cases. For instance, when one TCP Carson flow was run along with one TCP Reno flow (Table 6.3), the TCP Carson flow had an average congestion window size of 14.64 packets as compared to 12.68 packets for the TCP Reno flow.

- The average transmission rate is affected by the average congestion window size. We observe that TCP Carson achieves a higher average transmission rate, thereby a higher throughput in all cases. For instance, when one TCP Carson flow was run along with one TCP Reno flow (Table 6.3), the TCP Carson flow had an average throughput of 0.518 Mb/s as compared to 0.437 Mb/s for the TCP Reno flow.

- The average loss interval for TCP Carson is higher than that of TCP Reno in all the test cases. This implies that TCP Carson reduces the losses that occur in the congestion avoidance phase. Table 6.3 shows that the average loss interval for the TCP Carson flow was 240.23 packets as compared to 140.33 packets for the TCP Reno flow.

- TCP Carson is TCP-friendly in all cases. TCP-friendly implies that in steady state, it uses no more bandwidth than that suggested by the TCP response function. In all the test cases, the average transmission rate achieved by TCP Carson was less than the suggested transmission rate by the TCP response function. TCP response function suggests that for the observed average loss interval of 240.33 packets, a transmission rate up to 0.657 Mb/s may be achieved. TCP Carson achieves 0.518 Mb/s. (Table 6.3)

- The combination of higher throughput and lower loss rate implies that TCP Carson achieves better goodput than TCP Reno.

6.2 Sensitivity To Allowed Variance

For detecting the state of the network, TCP Carson uses the difference between two succesive weighted average values, as discussed in Section 4.4. This percentage difference is a parameter which is called *allowed variance*. In this section we evaluate the sensitivity of the allowed variance between consecutive values of the WALI. As mentioned in Section 4.4, as the allowed variance increases, it takes less time to detect steady state. Conversely, as the allowed variance increases, it takes longer to detect an unsteadiness in the network.

To understand the effect of the allowed variance, we ran the *4-8-16-8-4* scenario with 1 TCP Carson flow and 15 TCP Reno flows. The TCP Carson flow lasted the entire 875 seconds. Figure 6.2 plots the congestion window over time for TCP Carson for a range of values for the allowed variance. (0.02, 0.06, 0.10, 0.20) We observe that even under such continously changing network conditions, TCP Carson detects and adapts the increase and decrease values.

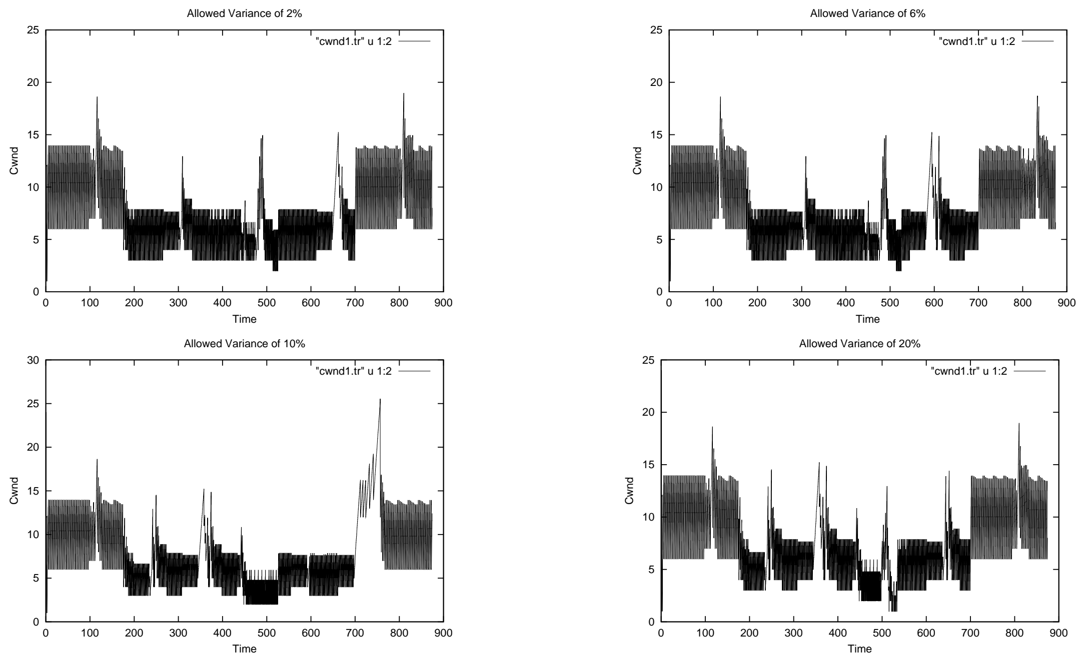


Figure 6.12: TCP Carson Sensitivity to Allowed Variance :: 4-8-16-8-4 TCP Carson

Allowed Variance	Throughput (Mb)
2%	0.172
6%	0.174
10%	0.177
20%	0.172

Table 6.6: Sensitivity to Allowed Variance

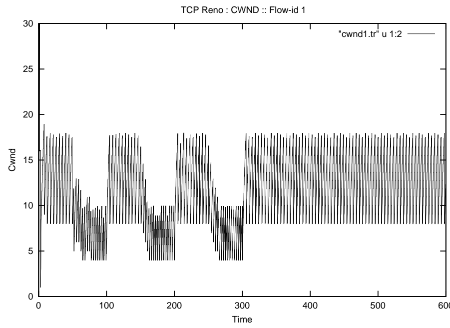
Table 6.6 compares the throughput of the TCP Carson flow in each case. We observe that as the allowed variance increases from 2% to 10%, the throughput achieved by the TCP Carson flow increases; but as the allowed variance goes up to 20% the throughput decreases. However, the difference between the throughput achieved is marginal, which suggests that over a long duration, the allowed variance does not have a significant effect on the throughput. We observed such a bell curve in the achieved throughput in other scenarios as well. While a 10% allowed variance was found to be suitable in most of our test cases, a history based approach can be used to further tune the value of allowed variance.

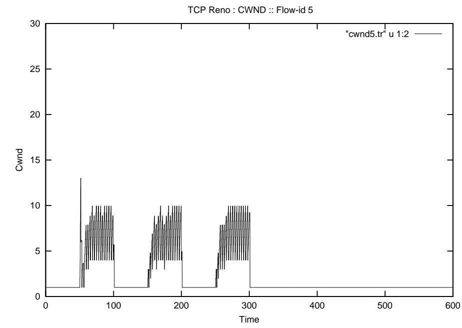
6.3 TCP Carson, TCP Reno and AIMD

TCP Carson is a combination of TCP Reno and AIMD. In this section, we compare the relative performance of TCP Carson, TCP Reno and AIMD (0.148, 0.125). For the first 300 seconds of the experiment we varied the number of flows between 4 and 8 in intervals of 50 seconds. For the remaining 300 seconds we kept the number of flows fixed at 4. Flows 1—4 lasted the entire 600 seconds, while flows 5—8 lasted for only 150 seconds. The congestion window of TCP Reno, TCP Carson and AIMD(0.148, 0.125) for flows 1 and 5 are plotted in Figures 6.13.

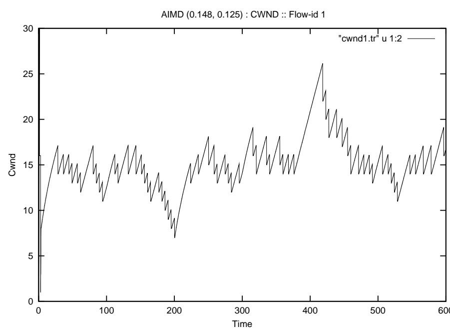
Table 6.3 gives a tabular comparison of the average throughput in each case. We observe the following:

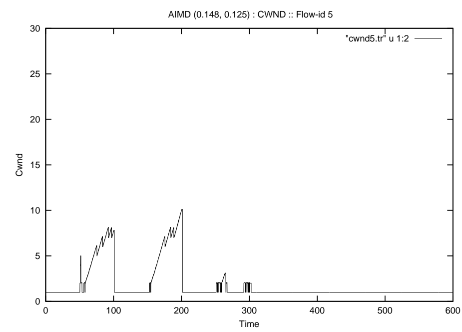
TCP Reno



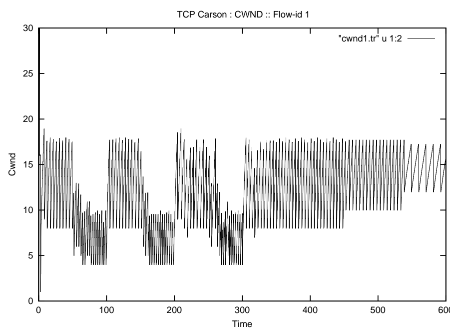


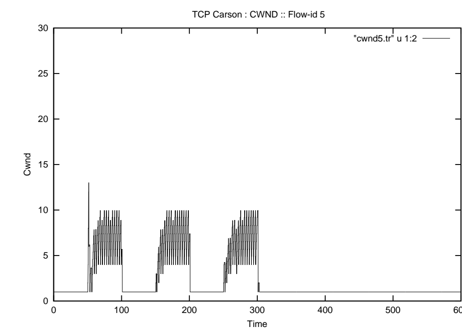
AIMD(0.148, 0.125)





TCP Carson





Flow 1 **Flow 5**

Figure 6.13: TCP Reno vs AIMD(0.148, 0.125) vs TCP Carson : Congestion Windows for Flows 1 and 5

	Avg. Throughput (Mb)	
	Flows 1–4	Flows 5—8
TCP Reno	0.198	0.100
TCP Carson	0.212	0.100
AIMD(0.148, 0.125)	0.228	0.038

Table 6.7: TCP Reno vs AIMD vs TCP Carson

- The average throughput achieved by flows 1—4 is highest for AIMD (0.228 Mb/s) and lowest for TCP Reno (0.198 Mb/s). For these flows, TCP Carson achieves a higher throughput than TCP Reno (0.212 Mb/s).

- The average throughput achieved by flows 5—8 is the same for TCP Carson and TCP Reno (0.100 Mb/s). During the periods when all the flows are in the network, both TCP Reno and TCP Carson, have comparable congestion window sizes for all the flows. However, AIMD blocks out flows 5—8 for considerable periods of time, thus being unfair. These AIMD flows achieve an average throughput of only 0.038 Mb/s.

Thus, by adapting the window increase and the decrease parameters to the current state of the network, TCP Carson achieves a higher throughput than TCP Reno and still remains fair.

On evaluation of TCP Carson in a range of network conditions and flow mixes, we observe that TCP Carson achieves a higher throughput than TCP Reno in all the test cases, remains TCP friendly, has fewer drops and is more fair than AIMD.

Chapter 7

Conclusions

... and I conclude ...

Majority of data transferred on the Internet is by long lived flows. While TCP is the most popular transport protocol, in a steady state network, TCP continues to probe for bandwidth, causing unnecessary losses and variation in the transmission rate.

TCP Carson is an enhancement to TCP Reno and shares the same characteristics as traditional TCP. TCP Carson uses a window-based approach to modify its transmission rate, is an end-to-end, fully reliable, acknowledged datagram service, and uses loss as an indicator of network congestion.

TCP Carson works on the principle that the window increase and decrease parameters of TCP can be adapted to suit the current network state. The state detection algorithm decides on the current state of the network. Any period of predictable oscillations in the transmission rate is considered *network steady state*. Conversely, varying transmission rates and periods of high congestion are considered *network unsteady state*.

We evaluated and analyzed mechanisms to detect the state of the network using the loss interval pattern. After understanding the advantages and drawbacks of these mechanisms, we chose a loss interval history based approach to detect the state of the network.

The state detection algorithm decides on the state of the network by monitoring the difference between two successive values of a Weighted Average Loss Interval(WALI). Other events such as a retransmission timeout or a long period of no loss for a flow marked an unsteady state. We evaluated our state detection algorithm over a range of network conditions and observed that the state detection mechanism performs reasonably well in most network conditions.

The *Adaptive AIMD* algorithm is an enhancement to the AIMD approach. AIMD suggests that the (1, 0.5) values for the window increase and decrease parameters of TCP are unnecessarily severe and suggests other values. However, unlike AIMD, which suggests fixed values for the window increase and decrease parameters, adaptive AIMD modifies these parameters depending on the state of the network detected.

TCP Carson starts by using (1, 0.5) for the window increase and decrease parameters. The adaptive AIMD algorithm uses the state detection algorithm to decide on the current state of the network. On detecting a steady state, it reduces the variance in the transmission rate by reducing the increase and decrease parameters. Thus, in a steady state, TCP Carson probes less aggressively for bandwidth and also responds less to a loss event. On detecting an unsteady state, TCP Carson changes back its increase and decrease parameters to (1, 0.5). TCP Carson thus adapts the increase and decrease parameters to suit the current network state.

We implemented the TCP Carson protocol in NS-2 [oCB]. We evaluated TCP Carson on a range of network conditions. Our findings indicate that TCP Carson achieves a higher throughput, lower loss-rate and a reduced window size variance in comparison with TCP Reno for long lived flows. Our evaluation also indicates that TCP Carson is more fair than the simple AIMD approach.

7.1 Future Work

In this section we propose possible extensions and enhancements to TCP Carson.

7.1.1 Steady State Detection Algorithm

Empirically, we found that a history of 32 loss intervals was well-suited in most of our test cases, and we used this history size for our evaluation. However, the history size can be tuned to suit the application and the observed network pattern. On a network that does not show a high degree of variance to available bandwidth per flow, smaller history sizes can reduce the history initialization period substantially.

7.1.2 Adaptive AIMD Algorithm

The values in the AIMD table are a result of empirical evaluation. Further research needs to be done for the proper choice of these values. Again, an equation based approach can be used instead of a tabular approach for the adaptive AIMD algorithm.

7.1.3 Router Support

We used a drop-tail router in all our evaluations. The effect of Active Queue Management techniques, such as Random Early Detection [FJ93] and Explicit Congestion Notification [Flo94], employed at the router on TCP Carson needs to be evaluated. Also, explicit bottleneck feedback from the router may improve the state detection algorithm and thereby improve the performance of TCP Carson.

7.1.4 Application Performance

Much research needs to be done to measure the relative benefits of TCP Carson over other transport protocols from the applications perspective. This would require the incorporation of the TCP Carson algorithm into an operating system and measuring the performance gain due to TCP Carson for long lived network applications. Again, the performance of TCP Carson for web traffic need to be further studied.

Bibliography

[AFP98] M. Allman, S. Floyd, and C. Partridge. Increasing TCP's Initial Window. *IETF Request for Comments (RFC) 2475*, September 1998.

[And00] Aditya Karnik And. Performance of tcp congestion control with explicit rate feedback: Rate adaptive tcp (ratcp), 2000.

[APS99] M. Allman, V. Paxson, and W. Stevens. TCP Congestion Control. *IETF Request for Comments (RFC) 2581*, April 1999.

[BRS99] Hari Balakrishnan, Hariharan S. Rahul, and Srinivasan Seshan. An Integrated Congestion Management Architecture for Internet Hosts. In *Proceedings of ACM SIGCOMM - 1999*, September 1999.

[CC00] Jae Chung and Mark Claypool. Better-Behaved, Better-Performing Multimedia Networking. In *Proceedings of SCS Euromedia*, May8-10 2000.

[CJOS00] M. Christiansen, K. Jeffay, D. Ott, and F.D. Smith. Tuning RED for Web Traffic. In *Proceedings of ACM SIGCOMM Conference*, August 2000.

[CPW98] Shanwei Cen, Calton Pu, and Jonathan Walpole. Flow and Congestion Control for Internet Streaming Applications. In *Proceedings of Multimedia Computing and Networking*, 1998.

[FF96] S. Floyd and K. Fall. Simulation-based Comparisons of Tahoe, Reno, and SACK TCP. *Computer Communication Review*, 26(3), July 1996.

[FF99] Sally Floyd and Kevin Fall. Promoting the Use of End-to-End Congestion Control in the Internet. *IEEE/ACM Transactions on Networking*, February 1999.

[FHP00] Sally Floyd, Mark Handley, and Jitendrha Padhye. A Comparison of Equation-Based and AIMD Congestion Control. Technical report, ACIRI, August 2000.

[FHPW00] Sally Floyd, Mark Handley, Jitendra Padhye, and Jorg Widmer. Equation-Based Congestion Control for Unicast Applications. In *Proceedings of ACM SIGCOMM Conference*, pages 45 – 58, 2000.

[FJ93] S. Floyd and V. Jacobson. Random Early Detection Gateways for Congestion Avoidance. *IEEE/ACM Transactions on Networking*, August 1993.

[Flo94] Sally Floyd. TCP and Explicit Congestion Notification. *Computer Communication Review*, October 1994.

[LPW01] Steven Low, Larry Peterson, and Limin Wang. Understanding TCP Vegas: A Duality Model. In *Proceedings of the ACM SIGMETRICS 2001 Conference*, pages 226–235, June 2001.

[Mat01] Liang Ibrahim Matta. The war between mice and elephants, 2001.

[MH00] Art Mena and John Heidemann. An Empirical Study of Real Audio Traffic. In *In Proceedings of the IEEE Infocom*, pages 101 – 110, March 2000.

[oCB] Universiy of California Berkeley. The Network Simulator - ns-2. Interent site
`http://www.isi.edu/nsnam/ns/`.

[PK98] V. Padmanabhan and R. Katz. Tcp fast start: a technique for speeding up web transfers, 1998.

[RHE99] Rezza Rejaie, Mark Handley, and D. Estrin. RAP: An End-to-end Rate-based Congestion Control Mechanism for Realtime Streams in the Internet. In *Proceedings of IEEE Infocom*, 1999.

[ROY00] Injong Rhee, Volkan Ozdemir, and Yung Yi. TEAR: TCP Emulation at Receivers – Flow Control for Multimedia Streaming. Technical report, Department of Computer Science, NCSU, 2000.

[SHS97] H. Sawashima, Y. Hori, and H. Sunahara. Characteristics of UDP packet loss: Effect of TCP Traffic. In *Proceedings of INET: The Seventh Annual Conference of the Internet Society*, June 1997.

[WS00] Haining Wang and Kang G. Shin. A simple refinement of slow-start of tcp congestion control. In *Proceedings of the Fifth IEEE Symposium on Computers and Communications (ISCC 2000)*, July 4-6 2000.

[YKXL00] Yang.R, Min Sik Kim, Z Xincheng, and S Lam. Two Problems of TCP AIMD Congestion Control, June 2000.

[YL00] Richard Yang and Simon Lam. General aimd congestion control. In *Proceedings ICNP 2000*, November 2000.