OPTIMIZING CONGESTION AVOIDANCE AND CONGESTION CONTROL IN WIRED AND WIRELESS NETWORKS

Thesis

Submitted in partial fulfillment of the requirements for the degree of

DOCTOR OF PHILOSOPHY

by

TAHILIANI MOHIT PRAKASH

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

NATIONAL INSTITUTE OF TECHNOLOGY KARNATAKA,

SURATHKAL, MANGALORE - 575025

MARCH, 2013

To,
my beloved ancestors,
parents and Rohit

Acknowledgements

In my journey towards the completion of my doctoral thesis, I was assisted and shown the path by numerous people.

Things are definitely a lot easier when there is a person who gives you ideas, who goes through each and every word written by you over and over, correcting it, improving it, a person whom you turn to each time you have a doubt or some problem. Words are insufficient to express my sincere gratitude and thanks towards my research guide, **Prof. K. C. Shet**, for his constant guidance, support and encouragement. He gave me a lot of freedom in my research and allowed me to work on the problems of my interest. His passion for teaching and research showed me how academia can be exceedingly rewarding and great fun at the same time. I have learnt a lot and still have plenty to learn from him.

I am highly grateful towards my Research Progress Assessment Committee (RPAC) Members **Prof. U. Sripati** and **Prof. S. S. Kamath** for their greatly valuable comments. Their sharp questions forced me to think harder about my research. I wholeheartedly thank **Prof. T. G. Basavaraju** who gave me an opportunity to join NITK, Surathkal. I consider myself privileged for being a part of NITK, Surathkal. This would not have been possible without his belief and confidence in me. I am immensely thankful to **Prof. K. Chandrasekaran** for his strong support and confidence in my research abilities. His encouraging feedback has always motivated me in my research work. I humbly thank **Dr. Annappa**, HOD/Chairman, DRPC for his regular inputs which have helped me me to do the work in the right direction. I would also like to thank the ex-Head of our Department **Sri Vinay Kumar** for providing the facilities needed to carry out the research work.

Assistant Professor **Mr. B. R. Chandavarkar** has been an excellent company throughout my Ph.D studies. I am grateful to him for giving me an opportunity to work with Pearson Publishers and contribute to its International Editions. He was instrumental in collaborating and establishing Wireless Information Networking Group (WiNG) and organizing SWN-2011 and SWN-2012 at NITK Surathkal. It is hard to imagine a support better than the one provided by Mr. B. R. Chandavarkar.

It has been a pleasure to share the Research Lab with my colleagues. I would like to thank **Mrs. Raksha Prabhu**, **Mrs. Saumya Hegde** and **Dr. Prashanth C. M.** for their encouragement and kind support at the start of my Ph.D studies; **Mrs. Geetha V.** for the interesting discussions on life and phiolosophy and **Usha Divakarla** for always finding a way to arrange parties. I would like to thank **Ganesh Reddy**, **Rino Cherian**, **Manoj Thomas**, **Anitha** and **Bane Raman** for being wonderful department mates, **Fr. Johny Jose** for always bringing a smile on everyones face, **Mr. Shivamurthy** for his interesting queries on *ns*-2 and **Mrs. Poornalatha**, **Mr. Melwyn**, **Mr. Kiran**, **Mrs. Megha** and **Mrs. Pushpalatha** for their encouragement and support.

I would like to thank a few post graduate and under graduate students with whom I was associated at NITK, Surathkal. I thank **Shivaraj Shetty**, **Chirag Raval**, **Vaidehi Panwala**, **Raviteja B. L.**, **Tanusha Nadkarni**, **Rabna Rajendran**, **Rochak Gupta**, **Sutanu Maiti**, **Abhishek Gupta**, **Anupma Raj**, **Madhulata H** and **Rakesh Aggarwal** for the wonderful moments we spent together while working on several projects. Shivaraj Shetty always amazed me with his wonderful abilities of problem solving; I thank him for the several discussions we had on the design and implementation of XECN. He brought to my notice, a minor bug in the XECN mechanism, for which I am highly grateful to him.

Manasa was my colleague for six years during my undergraduate and postgraduate studies and more importantly, my best friend for almost a decade now. During the not-so-great times of my studies, she was right there and provided excellent support. She has always been an integral part of my every success. I dont have words to express my gratitude for the enormous amount of time, support and advice she gave me whenever I needed it. She always motivated me to strive for the best. It is tough to imagine a better friend than Manasa. Thank you, Manasa.

I am grateful to **Mumtaz Mohammed Hussain** and his family for their excellent and boundless support throughout my nine years in Karnataka. I wholeheartedly thank them and Manasa's parents for recommending me to join Nitte for my postgraduate studies; which provided an outstanding platform to explore my capabilities and made me worthy to be a part of NITK. Mumtaz Hussain has been a great mentor, an excellent critic and a fantastic blackboard off which I'd frequently bounce off my ideas and dumb questions at times. He taught me the usefulness of the daily grind in the life of a Ph.D. student and always motivated me to strive for the best, for which I am deeply indebted to him. Thank you, Mumtaz bhai for always bringing a smile on my face with your interesting comments and poems and every other thing.

My stay at Usha Kiran, Nitte would not have been so comfortable and enjoyable without the constant support of **Late Sri G. K. Shetty** and his son **Mr. Sandeep Shetty**; **Bharati Kamath** and her sons **Hitesh Kamath** and **Sandesh Kamath**. I would like to thank them all for making my stay at Nitte a memorable one. Although the journey from Nitte to NITK is quite long, for me, it was always short and sweet. I would like to thank the Vishal bus staff **Mr. Jayant**, **Mr. Krishna**, **Mr. Umesh**, **Mr. Sunil**, **Mr. Afzal**, **Mr. Sudarshan**, **Mr. Santu** and **Mr. Yashodhar** for their kindness and concern towards me. They were very kind to pick/drop me near my apartment, especially during the heavy rains in the monsoon. I would like to thank **Hegde uncle**, **Mr. Jeevan**, **Mr. K. Ramakrishna Rao**, **Mr. Prashant**, **John uncle**, **Mr. Prasad**, **Amit**, **Nelson Tellis**, **Oneal Mascarenhas**, **Varun**, **Wilson Tellis** and **Pawan kumar** for sharing the lighter, fun filled moments in the bus and helping me to beat the stress.

I would like to thank **Mr. Abid** for being an excellent company and providing infinite support during the last five years. His galvanizing presence in my life has added to my development as a researcher and as a human being. I am grateful to him for giving me a bike which immensely helped in managing the time for my research work. Thank you, Abid bhai for the wonderful support, fun filled moments and every other thing.

I would like to thank the staff of our Deparment for all their help and support. **Mrs. Seema Shivaram** was wonderful in helping me with all the official work. **Mr.**

Dayanand was very helpful in ensuring my Progress Seminars went well by taking care of every other thing. I would also like to thank **Mrs. Yeshavanti**, **Hema**, **Vanita**, **Jaya**, **Mr. Kamath**, **Mr. Balachandra** and **Mr. Vairavnathan** for their help during my stay at NITK.

A special thanks to my beloved friends **Meha Vaidya**, **Birju Ransariya**, **Dharmendra Patel** and **Pavan Kumar Rambatla** who have been a pillar of support since my schooling days, **Kumar Rohit** - my apartment mate during my undergraduate studies, **Dhiraj Minocha** and **Tuhin Chakraborty** - my classmates during my undergraduate studies.

Whatever I am today, is because of the life long support of my father **Dr. Prakash U. Tahiliani** and my mother **Mrs. Meena P. Tahiliani**. I dedicate the successful completion of my doctoral thesis to my father's dream. My fathers great confidence in my abilities encouraged me to always go an extra mile in my research. My little brother, **Rohit P. Tahiliani** has been a rock of support in my life. His amazing support and constant encouragement provided me with a lot of strength during my research work. This thesis would not have been possible without the amazing support, patience, constant encouragement, care, unfettered belief and prayers of my family. No words can express my gratitude for them.

Finally, my deepest gratitude towards the Almighty for making this thesis possible.

Place: NITK, Surathkal **Mohit P. Tahiliani**

Date: 18th March, 2013

Abstract

Internet over the past few years has transformed from an experimental system into a gigantic and decentralized source of information. The success of the Internet can be partly attributed to the congestion control mechanisms implemented in Transmission Control Protocol (TCP). TCP has been the de-facto transport protocol for Internet since its inception. Although TCP constantly evolved over a period of two decades, the diversity in the characteristics of present and next generation networks and a variety of application requirements have posed several challenges to TCP congestion control mechanisms. As a result, the shortcomings in the fundamental design of TCP have become increasingly apparent. In this dissertation, we propose solutions to overcome these shortcomings and increase the robustness of TCP by carefully optimizing the fundamental TCP congestion control algorithms.

We focus on the middle ground between *end-to-end transport protocols* and *network based transport protocols*. The major goal is to optimize the performance of TCP while ensuring minimum deployment complexity. The motivation stems from the fact that the need for deployment of Active Queue Management (AQM) and Explicit Congestion Notification (ECN) has become apparent, owing to the drastic impact of "persistently full buffers" on the performance of Internet. We aim to leverage the benefits of AQM/ECN mechanisms and provide a richer explicit feedback to the end-hosts to aid them in making efficient congestion control decisions.

Although Random Early Detection (RED) has proved to be an effective AQM mechanism, its performance is highly sensitive to the appropriate settings of the parameters. Rather than tuning the parameters of original RED, this dissertation instead, aims to improve the performance of Adaptive RED (ARED) and Refined Adaptive RED (Re-ARED) by proposing two new AQM mechanisms. First, we demonstrate that neither ARED's conservative approach alone nor Re-ARED's aggressive approach alone suffices to improve the throughput and reduce the packet drop rate. Hence, we have designed and implemented two new AQM mechanisms, namely "Fast Adapting RED (FARED)" and "Cautious Adaptive RED (CARED)" to combine the benefits of ARED and Re-ARED. Extensive simulation results show that while FARED fails to achieve the desired goal, CARED offers robust performance in a wide variety of scenarios and outperforms ARED and Re-ARED. Unlike other RED variants, CARED requires only algorithmic modifications and is easy to deploy.

Second, a new congestion signaling mechanism called "eXtended ECN (XECN)" is designed and implemented to provide richer feedback to the end-hosts. XECN does not require additional bits in TCP or IP header since it re-uses the bits already allocated for ECN efficiently and unambiguously. Moreover, it requires modifications only at the sender and the receiver. It does not require any modification in the working of the router.

Third, we develop a new variant of TCP called "TCP Surathkal" which leverages the benefits of AQM and XECN. TCP Surathkal takes congestion control decisions based on the *severity of congestion* in the network. We present a modified design of a fluid model which is based on Poisson Counter Driven Stochastic Differential Equations to validate the working of TCP Surathkal. Results obtained by conducting extensive simulations and mathematical modeling show that TCP Surathkal achieves high utilization, reduces the packet drop rate and incurs less oscillations in the router queues. The advantages of TCP Surathkal make it suitable for deployment in a wide variety of networks.

Table of Contents

List of Tables

List of Figures

Abbreviations and Nomenclature

max_p	Maximum Drop Probability
max_{th}	Maximum Threshold
min_{th}	Minimum Threshold
p_d	Packet Drop Probability
w_q	Queue weight factor for Exponential Weighted Moving Average
avg	Average Queue Length
$cwnd$	Congestion Window
$newavg$	Current Average Queue Length
ns-2	Network Simulator - 2
$oldavg$	Previous Average Queue Length
ACKs	Acknowledgments
AIAD	Additive Increase Additive Decrease
AIMD	Additive Increase Multiplicative Decrease
AQM	Active Queue Management
ARED	Adaptive Random Early Detection
BER	Bit-Error Rate
BRED	Balanced Random Early Detection

CARED	Cautious Adaptive Random Early Detection
CBT-RED	Class-Based Threshold Random Early Detection
CC	Count Codepoint
CE	Congestion Experienced
CWR	Congestion Window Reduced
DCTCP	Data Center Transmission Control Protocol
DRED	Dynamic Random Early Detection
DSRED	Double Slope Random Early Detection
ECE	ECN Echo
ECN	Explicit Congestion Notification
ECT	ECN Capable Transport
EWMA	Exponential Weighted Moving Average
FARED	Fast Adapting Random Early Detection
FRED	Flow Random Early Detection
FTP	File Transfer Protocol
GAIMD	General AIMD
GRED	Gentle Random Early Detection
HSTCP	High Speed Transmission Control Protocol
ICMP	Internet Control Message Protocol
IETF	Internet Engineering Task Force
LRED	Loss-ratio based Random Early Detection
MIMD	Multiplicative Increase Multiplicative Decrease

NLRED	Nonlinear Random Early Detection
PQM	Passive Queue Management
RCP	Rate Control Protocol
Re-ARED	Refined Adaptive Random Early Detection
RED	Random Early Detection
RED-PD	Random Early Detection with Preferential Dropping
RTO	Retransmission Timeout
RTT	Round Trip Time
SACK	Selective Acknowledgment
SARED	Stabilized ARED
SRED	Stabilized RED
STCP	Scalable Transmission Control Protocol
TCP	Transmission Control Protocol
VoIP	Voice over Internet Protocol
Wi-Fi	Wireless Fidelity
WRED	Weighted Random Early Detection
XCP	eXplicit Contol Protocol
XECN	eXtended Explicit Congestion Notification

Chapter 1

Introduction

Internet over the past few years has experienced phenomenal growth in terms of the penetration rate and the diversity of the applications. It has transformed from an experimental system into a gigantic and decentralized source of information. According to a 2011 study (Akamai 2011), the average connection bandwidth globally is 2.6Mbps while more than 27% of users having bandwidth above 5Mbps and 65% of users having bandwidth above 2Mbps. The usage of narrowband (<256Kbps) has shrunk to about 2.9% of the users. The success of the Internet can be partly attributed to the congestion control mechanisms implemented in Transmission Control Protocol (TCP). These congestion control mechanisms are widely deployed in well known operating systems and are extensively used by a variety of Internet applications. However, tremendous growth in the range of bandwidth, increase in Bit-Error Rates (BER) and increased diversity in applications have challenged the congestion control mechanisms of TCP. These mechanisms must be able to accomodate and leverage the diversity in the characteristics of different networks (See Table 1.1), support a variety of application requirements and different traffic workloads (See Table 1.2). Thus, the need for optimizing these mechanisms has become extremely important.

Transport protocols with congestion control mechanisms are mainly classified into three categories: (i) *end-to-end protocols* (e.g., TCP Newreno) that rely on implicit congestion signals such as packet loss and/or latency, (ii) *network based protocols* (e.g., eXplicit Contol Protocol (XCP)) that rely on explicit feedback from the network and (iii) *end-to-end protocols with explicit feedback* (e.g., TCP+AQM/ECN) that rely on a few bits of explicit feedback from the network to aid end-hosts in making congestion

Table 1.1: Characteristics of different networks (Qazi 2010)

Network	**Capacity**	**Latency**	**BER**
Wired LANs (e.g., Ethernet)	10Mbps - 10Gbps	$<$ 1ms	$\leq 10^{-12}$
Data Centers	1Gbps - 1Tbps	100μs - 1ms	10^{-12}
Wired WANs	$\approx$10Mbps - 14Tbps	10ms - 300ms	10^{-12}
802.11 WLAN/Mesh Networks	$<$1Mbps - 600Mbps	1ms - 200ms	$>10^{-5}$
Cellular Data Networks (e.g., 3G)	384Kbps - 3Mbps	$\approx$ 100ms - 1s	10^{-5}
Satellite Networks	100Kbps - 155Mbps	250ms - 1s	10^{-10}

Table 1.2: Performance Requirements of Internet Applications (Qazi 2010)

Application	**Examples**	**Requirements**
Interactive	Voice over IP, Video Conferencing	Minimal latency, small jitter and less throughput variations
Short flows ($<$ 100KB)	Google Search, Facebook	Short response times
Medium sized transfers (100KB - 5MB)	Picasa, YouTube, Facebook photos	Low latency
Large transfers ($>$ 5MB)	Software updates, Video On-demand	Consistent high throughput

control decisions. While *end-to-end protocols* have performance limitations, *network based protocols* have been considered as hard to deploy because they need to maintain per-flow state at the routers. Since routers are complex and expensive devices, modifying them is a difficult task. Moreover, *network based protocols* require more bits for explicit feedback than are available in the IP header (Qazi 2010). On the other hand, *end-to-end protocols with explicit feedback* have lower deployment complexity than *network based protocols* since they require modifications mainly at the end-hosts, with incremental support from the routers (e.g., deployment of AQM/ECN) (Qazi 2010).

The performance of TCP-based applications, apart from the congestion control mechanisms, critically depends on the choice of queue management scheme implemented in the routers. Queue management mechanisms control the length of the queues by dropping packets when necessary. Passive Queue Management (PQM) (e.g., tail-drop) is the most widely deployed queue management mechanism in Internet routers (Hassan and Jain 2004). PQM does not employ any preventive packet drop before the router buffer gets full and hence, is easily deployable. However, due to the inherent problems of PQM such as global synchronization (Floyd and Jacobson 1993), lock-out (Hassan and Jain 2004), etc, Internet Engineering Task Force (IETF) recommends Active Queue Management (AQM) for the next generation Internet routers (Hassan and Jain 2004). Moreover, another limitation of PQM called *"persistently full buffer problem"* (recently exposed as a part of *"bufferbloat"* (Gettys 2011)) has proved the acute need of widespread deployment of AQM. The expected advantages of AQM deployment are: (i) overall performance improvement in terms of increase in throughput, reduced delay, minimal packet loss and overcoming the limitations of PQM. (ii) end-hosts can get richer explicit feedback which, infact, is required to meet the challenges of the present and next-generation networks.

This dissertation focuses on improving the performance of *end-to-end protocols with explicit feedback*, especially TCP+AQM/ECN. The major goals are (i) to optimize the performance of existing AQM mechanisms for *congestion avoidance* while ensuring minimal deployment complexity (ii) enable congestion signaling mechanisms to provide richer feedback to end-hosts by using already available bits in the TCP and IP header and (iii) to optimize the *congestion control* laws of TCP. We start by providing a brief description of the standard AQM mechanisms.

1.1 AQM for Congestion Avoidance

TCP provides congestion control by four main algorithms namely Slow Start, Additive Increase/Multiplicative Decrease (AIMD)[1], Fast Retransmit and Fast Recovery. Slow start and AIMD are used for dynamically changing the size of a congestion window (*cwnd*). Slow Start increases the *cwnd* exponentially to quickly bring a newly started flow to the desired speed. In steady state, TCP uses AIMD to vary the *cwnd* in conjunction with fast retransmit and fast recovery. Fast Retransmit and fast Recovery are triggered in the event of a packet loss and are used to quickly recover from the state of congestion. These four algorithms, though modified several times in the

[1]AIMD algorithm is also known as *Congestion Avoidance* algorithm.

recent past, have been the cornerstones of TCP congestion control research.

Congestion avoidance mechanisms differ from congestion control mechanisms, since former are proactive while latter are reactive. Though AIMD is also known as *Congestion Avoidance* algorithm, it is a misnomer since AIMD does not try to *avoid* congestion proactively (Bhandarkar and Reddy 2007). Henceforth, we consider AIMD algorithms as *Congestion Control mechanisms* and AQM mechanisms as *Congestion Avoidance mechanisms* since AQM mechanisms proactively inform the sender about network state and *avoid* congestion.

The deployment of AQM mechanisms in the Internet has significantly increased in the recent past, because PQM mechanisms have a few limitations such as:

- Global Synchronization (Floyd and Jacobson 1993): Traditional tail-drop gateways do not provide an early congestion notification. This leads to *global synchronization*, a phenomenon in which all senders sharing the bottleneck gateway reduce their sending rate at the same time, thereby under-utilizing the network resources.

- Lock-out (Hassan and Jain 2004): PQM mechanisms (e.g., tail-drop) allow a single connection or a few connections to monopolize the buffer space in the router queues. This results in unfair sharing of the network resources among the connections, thereby giving rise to fairness problems.

- Bufferbloat (Gettys 2011): Since memory costs have reduced in the recent past, modern Internet routers are designed with extremely large buffers. As a result, today's Internet suffers from poor network performance because TCP variants implemented in modern operating systems are *end-to-end protocols* and hence, do not reduce the sending rate unless a packet drop is encountered. Since the packet drop occurs only when these large buffers overflow, queueing delay experienced by each packet increases drastically, thereby degrading the Quality of Service for delay sensitive applications such as DNS queries, Voice over IP (VoIP) and other multimedia applications. This problem has been termed as *Bufferbloat*.

AQM mechanisms have been extensively studied to monitor and limit the growth of the queue at routers. These mechanisms *avoid* congestion by proactively informing the sender about congestion, either by dropping a packet or by marking a packet. Random Early Detection (RED) (Floyd and Jacobson 1993) is the most widely deployed AQM mechanism in the routers and Explicit Congestion Notification (ECN) (Ramakrishnan et al. 2000) is the most popular marking mechanism (also known as signaling mechanism) used in conjunction with RED. However, it has been shown that the effectiveness of RED largely depends on appropriately setting atleast four parameters, namely: minimum threshold (min_{th}), maximum threshold (max_{th}), queue weight factor (w_q) for exponential weighted moving average and maximum drop probability (max_p) (Feng et al. 1999; Floyd et al. 2001). Optimal values for these parameters differ for different scenarios and are dependent on several other factors such as number of flows passing through same bottleneck gateway (Feng et al. 1999), packet size (Misra et al. 2000), etc. Table 1.3 shows the values of these parameters used in Cisco 12000 Series routers that implement a modified RED called Weighted RED (WRED). C is the capacity of the link in packets where mean packet size is 1500 bytes.

Table 1.3: WRED parameter setting in Cisco 12000 Series Router

Link Speed	C	min_{th}	max_{th}	w_q	max_p
DS3	3666	110	367	9	1
OC3	12917	388	1292	10	1
OC12	51666	1550	5167	12	1

Adaptive RED (ARED) (Floyd et al. 2001) addresses the parameter sensitivity of RED by dynamically varying max_p and automatically setting min_{th}, max_{th} and w_q parameters. ARED requires setting of only one parameter - *target queueing delay*, defined as the maximum amount of time a packet is delayed in the queue. In this dissertation, we show that though ARED dynamically varies max_p, the adaptation is conservative and leads to the loss of throughput for sometime, especially when the level of congestion changes sharply. We further show that a combination of conservative and aggressive approach to vary max_p yields robust performance in a wide range of scenarios.

1.2 TCP Congestion Control Laws

Though the scale of the Internet and its usage increased by several orders of magnitude in the recent past, TCP has evolved to keep up with the changing network conditions and has proven to be scalable and robust. Table 1.4 shows the implementation of various TCP variants in modern operating systems (Afanasyev et al. 2010). However, since the nature of Internet applications changes frequently, some of the assumptions made during the early design of TCP may not be valid in today's networks. As an example, TCP congestion control mechanisms assume network to be a *black box* and rely on a packet loss as an implicit notification of congestion (Ramakrishnan et al. 2000). While in most traditional networks this assumption is true, newer network environments, especially wireless, challenge it. Moreover, using packet loss as a congestion signal results in coupling of congestion control with loss recovery; thus makes different types of losses difficult to discern (Qazi 2010). This forces senders to respond in the same manner to all types of losses and worsens the performance of TCP in wireless networks. Non-congestion losses are mistaken for congestion losses and the sending rate is unnecessarily reduced.

Table 1.4: TCP Variants in modern Operating Systems

Operating System	**Default TCP**
Linux Kernel > 2.6	CUBIC
Macintosh	TCP Newreno + SACK
Windows XP/Vista/7	TCP Newreno + SACK
Android	CUBIC

Another area of concern that arises from the early design of TCP is the conservative approach adopted by TCP's AIMD algorithm. When an acknowledgement (ACK) is received in AIMD phase, the *cwnd* is increased as shown in Eq. 1.2.1. This is known as Additive Increase phase of the AIMD algorithm.

$$cwnd = cwnd + \frac{1}{cwnd} \tag{1.2.1}$$

When congestion is detected either through timeout or duplicate acknowledgements (*dupacks*) or Selective Acknowledgements (SACK), *cwnd* is updated as shown in Eq. 1.2.2. This is known as Multiplicative Decrease phase of the AIMD algorithm.

$$cwnd = \frac{cwnd}{2} \tag{1.2.2}$$

After reducing the *cwnd* by half in the event of congestion, TCP linearly increases the *cwnd* which takes considerable number of Round Trip Times (RTTs) to reach the original sending rate i.e., the *cwnd* size when congestion is detected. While decreasing the *cwnd* by half certainly reduces the load, "by half" is an ad-hoc decision (Yang and Lam 2000).

In this dissertation, we design and implement a new TCP variant called "TCP Surathkal". TCP Surathkal employs an efficient multiplicative decrease mechanism which reduces the *cwnd* based on the *severity of congestion* in the network, rather than reducing it by half. We show that backing off as a function of the *severity of congestion* can lead to better congestion responsiveness, faster fairness convergence and less fluctuations in queues at the routers.

1.3 Motivation and Assumptions

In this section, we briefly discuss the motivation for AQM and the major goals of congestion control mechanisms.

1.3.1 Motivation for AQM

The main motivation for using AQM is that it provides preventive measures to manage a queue and eliminates the limitations associated with PQM. The goals of AQM are specified as follows (Hassan and Jain 2004):

- Minimal packet loss - The number of packets losses due to buffer overflow in routers must be reduced to improve the throughput.
- Low latency - The queueing delay must be kept low by maintaining a small queue size to provide better service to interactive apllications.

- Fairness - Fair bandwidth allocation among the competing flows to avoid the lock-out problem.

1.3.2 Goals of Congestion Control mechanisms

The major goal of congestion control mechanisms is to use the network as efficiently as possible. There are a few performance goals that congestion control mechanisms must achieve (Qazi 2010).

- Efficiency - Congestion control protocols should be able to achieve maximum link utilization across a broad range of link capacities.
- Negligible Packet Loss - Packet losses due to buffer overflows should be negligible. These losses result in retransmissions which waste the network bandwidth.
- Stability - When networks face transient erratic behaviors caused by sudden increases in traffic (e.g., flash-crowds), congestion control mechanisms should be able to detect this behavior and move to a stable operating point.
- Fairness - Congestion control mechanisms should ensure fair usage of the available link capacity. Several definitions of fairness exist in the literature; the focus of this dissertation is on the Jain's fairness index.
- Easy to deploy - Ideally, the new congestion control protocol should be amenable to deployment in the current Internet architecture. It should be able to coexist with TCP without requiring complex router-level computations. Moreover, its incremental deployment must be feasible without affecting the performance of the protocols that already exist.
- Low Router Complexity - Protocols that require the routers to maintain per-flow queues and classify every packet are considered complex. Such protocols have more memory requirements and hence, incur more cost.

1.3.3 Assumptions

The scope of this dissertation has been limited to congestion in connectionless packet-switched networks, and in particular, to networks with the same architecture as the Internet. The assumptions made towards the network under consideration are as follows (Toomey 1997):

- The network is a wide-area network with nodes passing data to other nodes along links. The nodes are arbitrarily connected; the network does not have a particular topology (e.g., spanning tree, hypercube, etc).
- The network is connectionless. There is no reservation of network bandwidth or resources between a source of data transmission and its intended destination.
- The network is packet-switched. Packets are individually routed from a source to a destination.
- The network uses semi-static routing. Routes can change slowly over time, but most routes are static.
- The network sets no bandwidth constraints on data sources. A source host may attempt to transmit data at a rate which will exceed the bandwidth or resources on the links and the network elements.
- The network makes no transmission guarantee. The links on the network have finite bandwidth and the nodes have finite buffer space for packets waiting to be routed. If network components are unable to forward data on to the destination for any reason, the data may be dropped.

Justification:

Currently, Internet provides only the *best-effort* delivery of data i.e. the network only attempts to deliver the packets, without guaranteeing any performance bound. Moreover, the Internet does not have a particular topology. These are, in general, the main characteristics of the Internet. The assumptions made above are in-line with the characteristics of the Internet.

1.4 Outline of the Thesis

The rest of the dissertation is organized as follows:

Chapter 2 provides the background of TCP and AQM, their limitations and a brief survey of the related work. We also discuss the different types of explicit feedback mechanisms, highlighting their merits and demerits.

Chapter 3 provides a detailed explanation of the performance of ARED and Refined Adaptive RED (Re-ARED) (Kim and Lee 2006). We have highlighted the shortcomings of ARED and Re-ARED by carrying out extensive simulations in varying scenarios. Further, we attempt to eliminate those shortcomings by designing two new ARED based AQM mechanisms: Fast Adapting RED (FARED) and Cautious Adaptive RED (CARED).

Chapter 4 discusses the eXtended Explicit Congestion Notification (XECN) mechanism which is newly designed to enable the TCP sender to infer the *severity of congestion* in the network. This chapter also describes the working of TCP Surathkal, which leverages the benefits of AQM mechanisms and XECN to improve the overall performance of the network.

In Chapter 5, we present a modified design of a fluid model which is based on Poisson Counter Driven Stochastic Differential Equations to validate the working of TCP Surathkal. The effectiveness of TCP Surathkal is demonstrated by studying the stability behavior of TCP Surathkal.

Chapter 6 presents the Conclusions and the possible future directions.

Chapter 2

Literature Review

The shortcomings of PQM schemes have posed a fundamental limitation in achieving the objectives of congestion control. Recently, a lot of work has focused on optimizing the performance of AQM mechanisms to *avoid* congestion. Since RED is widely implemented and is the default AQM mechanism recommended by IETF for the next generation Internet routers, we limit the scope of this dissertation to RED based AQM mechanisms. This chapter reviews the important features of RED based active queue management and TCP congestion control mechanisms. Further, we briefly review the different types of mechanisms to provide explicit feedback to the end-hosts and categorize them based on their deployment requirements.

2.1 RED based AQM mechanisms

The parameter sensitivity of RED has been addressed by several researchers and as a result, RED has been extended and enhanced by adopting many different approaches. The basic mechanism of RED, however, still remains the same.

On arrival of every packet, RED gateways calculate the average queue size (avg) by using Exponential Weighted Moving Average (EWMA). If avg is less than min_{th}, the packet is enqueued. If avg is more than max_{th}, the packet is dropped[1]. However, if avg is between min_{th} and max_{th}, the packet is dropped randomly with a certain probability. RED, therefore, has two computational parts: computation of avg and

[1]In the presence of ECN, the router may choose to mark the packet instead of dropping. Hence, we use the terms dropping and marking interchangeably.

calculation of packet drop probability (p_d). The following equations show avg and p_d calculation of RED respectively:

$$avg = ((1 - w_q) \times oldavg) + (w_q \times cur\_q) \tag{2.1.1}$$

where $oldavg$ = average queue size during previous packet arrival
cur_q = current queue size

$$p_d = \begin{cases} 0 & avg < min_{th} \\ \frac{avg - min_{th}}{max_{th} - min_{th}} \times max_p & min_{th} \le avg < max_{th} \\ 1 & avg \ge max_{th} \end{cases} \tag{2.1.2}$$

The probability with which a packet is dropped is a linear function of the avg. Hence, when avg varies from min_{th} to max_{th}, the drop probability varies from 0 to max_p. If avg increases above max_{th}, drop probability becomes 1 i.e. all incoming packets are dropped. Figure 2.1 shows the marking function of RED.

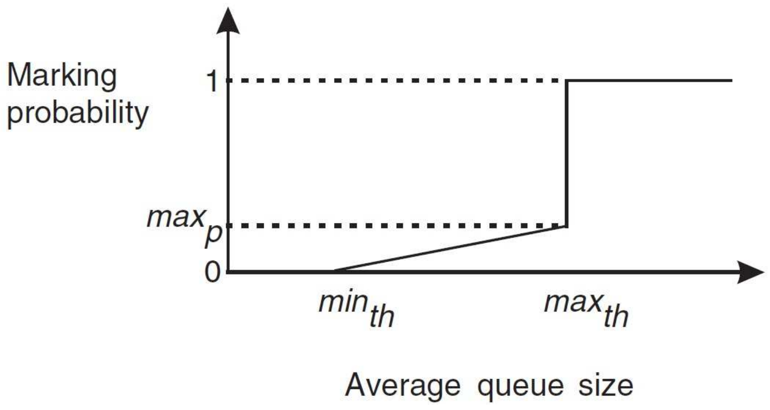


Figure 2.1: Marking function of RED

The effectiveness of RED highly depends on the appropriate setting of its parameters. Moreover, it is difficult to find the appropriate values of parameters that enable RED to perform equally well in different scenarios. RED may in fact perform worse than PQM if its parameters are not correctly tuned. We now discuss the issues and prior work done in appropriately setting these parameters.

Setting max_p

The choice of max_p significantly affects the performance of RED. If max_p is too small, the number of active packet drops becomes less and hence, cannot prevent the queue overflow. If max_p is too large, the number of active packet drops becomes more and significantly affects the throughput.

(Feng et al. 1999) demonstrates that the choice of max_p depends not only on the bandwidth delay product but also on the number of flows. An algorithm called Self Configuring RED is developed and implemented in (Feng et al. 1999) to vary max_p parameter based on the *avg* dynamics. The main idea is to modulate the packet dropping behavior of RED by monitoring the variations in the *avg*. If the *avg* oscillates around min_{th}, the value of max_p is decreased to make RED less aggressive. Similarly, if the *avg* oscillates around max_{th}, the value of max_p is increased to make RED more aggressive. max_p is carefully varied so as to keep the *avg* between min_{th} and max_{th}. This algorithm performs well in different traffic scenarios since it reduces the oscillations in the instantaneous queue length.

As an extension to Self Configuring RED, an Adaptive RED (ARED) is developed in (Floyd et al. 2001) and its effectiveness over the original RED[2] is demonstrated in terms of improved throughput and reduced oscillations in the queue size. Unlike Self Configuring RED, ARED is designed to keep the *avg* in *target range* between min_{th} and max_{th} and thus, max_p is varied accordingly. Moreover, ARED automatically sets min_{th}, max_{th} and w_q parameters. The choice of *target queuing delay*, which determines the trade-off between delay and link utilization, is left to the network operators.

ARED follows an Additive Increase/Multiplicative Decrease (AIMD) policy to vary max_p. While ARED adopts a conservative approach to vary max_p, Refined Adaptive RED (Re-ARED)(Kim and Lee 2006) adopts an aggressive approach to bring *avg* within its *target range* more *quickly.* However, like ARED, Re-ARED too employs an AIMD policy to adapt max_p. Contrary to ARED and Re-ARED, a modified ARED algorithm based on Multiplicative Increase/Multiplicative Decrease

[2]Original RED refers to the RED proposed in (Floyd and Jacobson 1993)

(MIMD) policy to adapt max_p is designed in (Marquez et al. 2007). However, the results show that MIMD policy to adapt max_p yields similar results as the AIMD policy.

Setting w_q for EWMA

The *avg* in RED is required to filter out the transient congestion, while at the same time, detect congestion that has persisted for several RTTs. If w_q is too small, the AQM may fail to detect the incipient congestion and lead to overall performance degradation by causing queues to overflow. If w_q is too large, *avg* tracks the instantaneous queue and leads to more oscillations in the queue, thereby degrading the performance of AQM.

ARED, as discussed above, automatically sets the w_q as a function of the link bandwidth. It is shown in as

$$w_q = 1 - exp(\frac{-1}{C}) \tag{2.1.3}$$

where C is the link capacity in packets/second, computed for packets of the specified default size (Floyd et al. 2001).

An ARED based algorithm[3] that adaptively varies w_q along with max_p is designed in (Verma et al. 2002). The main goal of the algorithm is to modulate the aggressiveness of RED by varying the value of w_q based on the changes in *avg*. If the change in *avg* is negligible, smaller value for w_q (w_{q1}) is chosen to give more weight to the *oldavg*. On the other hand, if the change in *avg* is significant, larger value for w_q (w_{q2}) is chosen to give more weight to the instantaneous queue length. However, w_{q1} and w_{q2} are fixed values and must be predefined.

Similar mechanisms, Stabilized ARED (SARED)(Javam and Analoui 2006) and Self Tuning RED (Chen et al. 2011) focus on assigning different queue weights, w_q, to ARED instead of one fixed queue weight. The major limitation of these approaches is that they introduce several new parameters to achieve performance gain. Setting these additional parameters adds to the complexity.

[3]We call this algorithm as w_q+max_p algorithm.

Setting min_{th} and max_{th}

It is recommended that the min_{th} for a RED router that carries only TCP traffic should be around five packets. max_{th} should be at least three times min_{th} (Floyd et al. 2001). However, a different set of values are required for min_{th} and max_{th} to achieve fairness when non-TCP traffic (e.g., UDP traffic) co-exists with the TCP traffic. This approach is adopted by Class-Based Threshold RED (CBT-RED) (Parris et al. 1999). CBT-RED sets the min_{th} and max_{th} thresholds according to the traffic type and its priority. UDP traffic is assigned a separate drop threshold than the one assigned for TCP traffic.

Balanced RED (BRED) (Anjum et al. 1999) achieves fairness among TCP and UDP traffic by regulating the bandwidth of a flow based on the other active flows. However, it requires per-flow accounting and hence, has scalability and deployment issues.

ARED automatically sets min_{th} based on the Eq.(2.1.4). max_{th} is set to three times of min_{th}.

$$min_{th} = max(5, \frac{d_t * C}{2}) \tag{2.1.4}$$

where d_t represents the *target queuing delay* set by the network operators.

Setting min_{th} as per Eq.(2.1.4) ensures that for high bandwidth links, min_{th} is set sufficiently high and for low bandwidth links, min_{th} is set accordingly low.

Calculating packet drop probability (p_d)

It is observed that sharply increasing the drop probability to 1 when *avg* crosses max_{th} (see Fig.2.1) results in high number of packet drops. Hence, a modified RED known as Gentle RED (GRED) is recommended by Floyd that varies the drop probability from max_p to 1 when *avg* varies from max_{th} to twice max_{th} so as to reduce the number of active packet drops. Figure 2.2 shows the marking function of GRED.

Stabilized RED (SRED) (Ott et al. 1999) has been designed to make the router queue stable over a wide range of load levels. Instead of calculating the *avg*, SRED

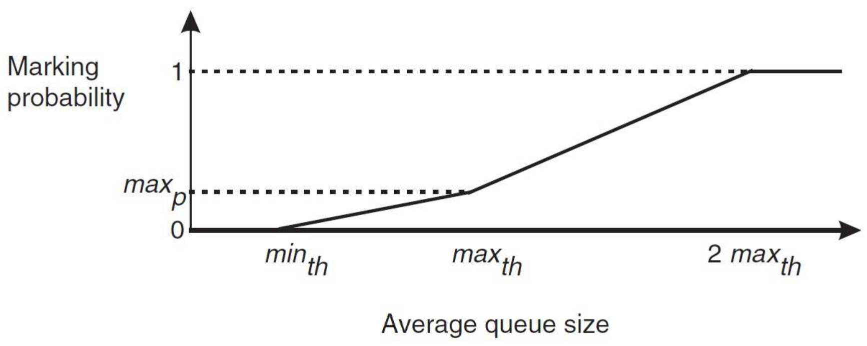


Figure 2.2: Marking function of Gentle RED

drops packets depending on the instantaneous queue length and the number of active flows. Eq.(2.1.5) and Eq.(2.1.6) show the packet drop probability (p_d) and the final packet drop function (p_{sred}) of SRED respectively. B represents the buffer capacity. SRED achieves the goal of stabilizing the queue, however, suffers from low throughput even for a small number of active flows.

$$p_d = \begin{cases} max_p & \frac{B}{3} \le cur\_q < B \\ \frac{1}{4}max_p & \frac{B}{6} \le cur\_q < \frac{B}{3} \\ 0 & 0 \le cur\_q < \frac{B}{6} \end{cases} \tag{2.1.5}$$

$$p_{sred} = \begin{cases} p_d & \textit{for a large number of active flows} \\ \frac{p_d}{65536}(numberofflows)^2 & \textit{for a small number of active flows} \end{cases} \tag{2.1.6}$$

Double Slope RED (DSRED)(Zheng and Atiquzzaman 2000) implements two linear drop functions with different slopes to improve the throughput and delay of RED. The main idea is to divide the queue between min_{th} and max_{th} into two segments and use a separate linear function for each segment. DSRED adapts to the level of congestion by changing the slope of the drop function. The equation governing the packet drop probability (p_d) is given by Eq.(2.1.7).

$$p_d = \begin{cases} 0 & avg < min_{th} \\ \alpha(avg - min_{th}) & min_{th} \leq avg < mid_{th} \\ 1 - \gamma + \beta(avg - mid_{th}) & mid_{th} \leq avg < max_{th} \\ 1 & avg \geq max_{th} \end{cases} \tag{2.1.7}$$

where α and β are given by Eq.(2.1.8) and Eq.(2.1.9) respectively. mid_{th} is a threshold for the *avg* to change the slope of the drop function and γ is used as a mode selector to adjust the slopes of the drop function.

$$\alpha = \frac{2(1-\gamma)}{max_{th} - min_{th}} \tag{2.1.8}$$

$$\beta = \frac{2\gamma}{max_{th} - min_{th}} \tag{2.1.9}$$

Nonlinear RED (NLRED) (Zhou et al. 2006), on the other hand, replaces the linear packet dropping function of RED by a nonlinear quadratic function to improve the effectiveness of RED. Eq.(2.1.10) shows the packet drop probability (p_d) of NLRED.

Though DSRED and NLRED outperform the original RED, their parameter sensitivity remains same as the original RED because they do not vary max_p and use the default value of w_q.

$$p_d = \begin{cases} 0 & avg < min_{th} \\ (\frac{avg - min_{th}}{max_{th} - min_{th}})^2 \times max_p & min_{th} \leq avg < max_{th} \\ 1 & avg \geq max_{th} \end{cases} \tag{2.1.10}$$

Other RED Variants

There are another category of RED based AQM mechanisms that not only take *average queue size* into consideration but also consider the *instantaneous queue size* to reduce the packet drop rate and improve the overall throughput. Examples of such mechanisms include Modified RED (Feng et al. 2004) and Effective RED (Abbasov and Korukoglu 2009). Appropriately setting thresholds for *instantaneous* *queue size* is a challenging issue in these mechanisms. Moreover, since these mechanisms are completely based on the original RED algorithm, the parameter sensitivity of these mechanisms remains same as that of the original RED.

Other AQM mechanisms designed based on RED include: Flow RED (FRED) (Lin and Morris 1997), Dynamic RED (DRED) (Aweya et al. 2001), RED with Preferential Dropping (RED-PD) (Mahajan and Floyd 2001), Exponential RED (Liu et al. 2005), Loss-ratio based RED (LRED) (Wang et al. 2007), AQM mechanism based on Neural Networks (NN-RED) (Hariri and Sadati 2007), etc. There are some concerns on the suitability of approaches followed by all these mechanisms since they do not eliminate the parameter sensitivity of RED. Moreover these mechanisms are more complicated to deploy than the original RED algorithm.

We summarize this section by classifying the RED variants based on the parameters they focus to improve the performance of original RED. This dissertation focuses on designing robust ARED based mechanism that varies max_p efficiently to improve the performance in terms of throughput and packet drop rate.

Table 2.1: Classification of RED Variants

max_p	w_q	min_{th} and max_{th}	p_d
Self Configuring RED	w_q+max_p algorithm	CBT-RED	GRED
ARED	ARED	BRED	SRED
Re-ARED	SARED	ARED	DSRED
ARED with MIMD	Self Tuning RED		NLRED

2.2 Mechanisms for Explicit feedback

Explicit feedback mechanisms have gained a lot of attention in the recent past because of the increasing number of problems experienced with implicit congestion notification mechanisms and partly due to the success of the ECN. Taking into consideration the large diversity in the characteristics of Internet, absence of explicit feedback may significantly affect the performance of TCP. We briefly discuss a few mechanisms designed to provide explicit feedback to the end-hosts. Table 2.2 shows the possible combinations to provide explicit feedback (Welzl 2006).

Table 2.2: Possible Combinations to provide Explicit Feedback

end-host generates	**router generates**	**router updates**	**Example**
×	✓	×	Choke packets / Source Quench
✓	×	✓	Explicit Congestion Notification
×	✓	✓	Hop-by-hop Choke packets
✓	✓	✓	Choke packets + ECN

Choke packets

Choke packets (Nagle 1984) are generated by the router as soon as it notices congestion and sent to the source. There are two types of choke packets: (i) which only notify the source about the congestion e.g., Source Quench conveyed with Internet Control Message Protocol (ICMP) and (ii) which provide additional information to the source e.g., the *degree of congestion.* The most widely deployed, though its usage is not recommended anymore (Welzl 2006), is the Source Quench message. Fig.2.3 (Welzl 2006) shows the working of choke packets.

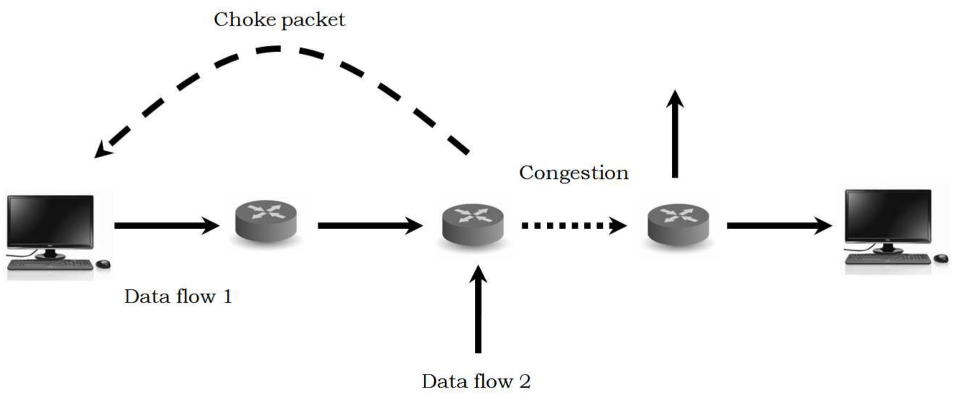


Figure 2.3: Choke packets

Sending choke packets is the fastest method to notify sources about the congestion. Moreover, the arrival of a choke packet at the source clearly indicates the presence of congestion in the network and hence eliminates the ambiguity between a congestion loss and a non-congestion loss.

The disadvantage of this method, however, is that the packet generation at routers is a very complex task because it involves memory allocations and header initializations. Moreover, injecting additional packets in the network in a state of congestion is not recommended, even if these packets travel in the other direction (Welzl 2006).

Explicit Congestion Notification

Explicit Congestion Notification (ECN)(Ramakrishnan et al. 2000) is one of the most popular congestion signaling mechanisms in communication networks. It is widely deployed in a large variety of operating systems at end hosts, modern Internet routers and used by a variety of transport protocols. Apart from being used as a congestion signaling mechanism, ECN in wireless networks is considered as an efficient mechanism to distinguish congestion losses from non-congestion losses (Ramani and Karandikar 2000). Recently, the advantages of ECN have been demonstrated in data center networks by a newly designed transport protocol called Data Center Transmission Control Protocol (DCTCP)(Alizadeh et al. 2010).

As shown in Fig.2.4 and Fig.2.5, ECN uses two bits in the IP header, namely ECN Capable Transport (ECT) and Congestion Experienced (CE), and two bits in the TCP header, namely Congestion Window Reduced (CWR) and ECN Echo (ECE), for signaling congestion to the end-hosts. ECN is an industry standard and its detailed mechanism is described in RFC 3168. Fig.2.6 (Welzl 2006) shows in brief,

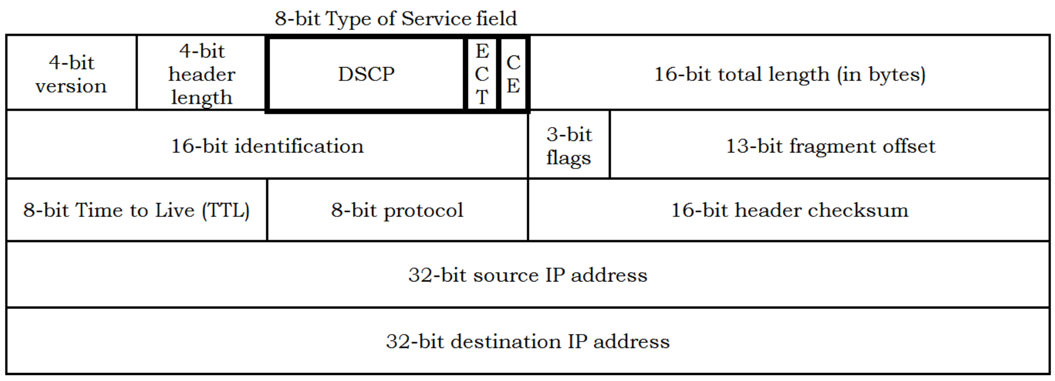


Figure 2.4: ECN bits in IP header

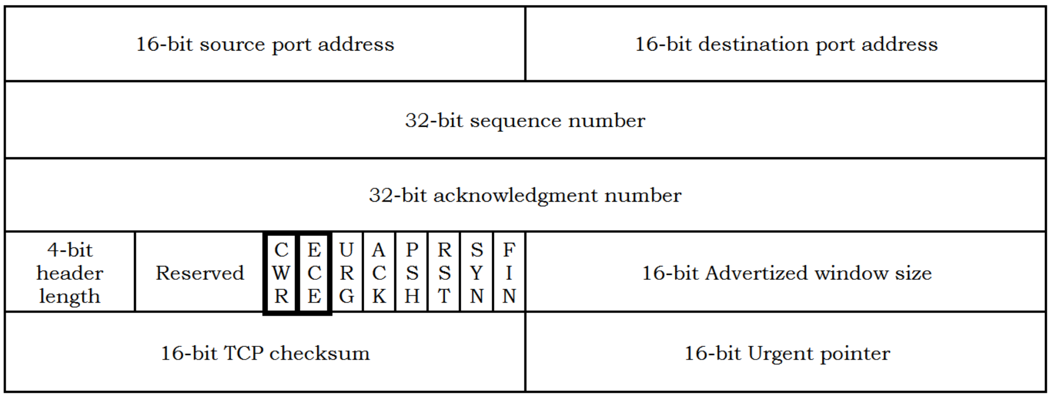

16-bit source port address
16-bit destination port address
32-bit sequence number
32-bit acknowledgment number
4-bit header length
Reserved
CWR
ECE
URG
ACK
PSH
RST
SYN
FIN
16-bit Advertized window size
16-bit TCP checksum
16-bit Urgent pointer

Figure 2.5: ECN bits in TCP header

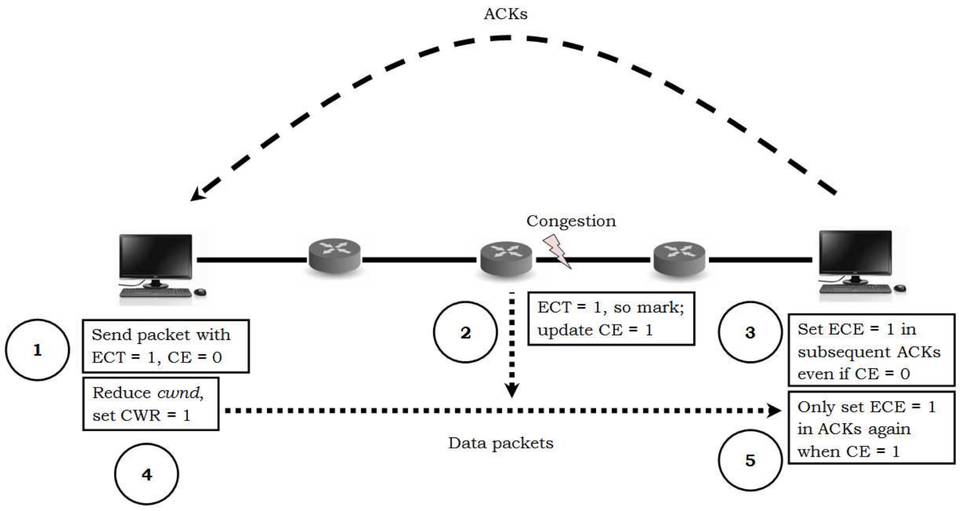

ACKs
Congestion
1
Send packet with ECT = 1, CE = 0
2
ECT = 1, so mark; update CE = 1
3
Set ECE = 1 in subsequent ACKs even if CE = 0
4
Reduce *cwnd*, set CWR = 1
Data packets
5
Only set ECE = 1 in ACKs again when CE = 1

Figure 2.6: Explicit Congestion Notification

the steps involved in the working of ECN mechanism.

Despite the fact that ECN can substantially improve network performance, network measurements reveal poor usage of this option in today's Internet. ECN is generally used in conjunction with RED. Since network operators still hesitate to enable RED, ECN too is disabled. In this dissertation, we leverage the advantages of ECN and extend it further to provide even better feedback to the end-hosts than it provides currently.

Hop-by-hop Choke packets

In hop-by-hop choke packets method (Gerla and Kleinrock 1980), every congested router on the path from a source to a destination sends the congestion feedback to its directly preceding router (see Fig.2.7). Based on this congestion feedback, the preceding router immediately controls the sending rate which reduces the load on the congested router and improves the network performance. This method has similar drawbacks as that of "Choke packets" since it imposes significant amount of work on the routers.

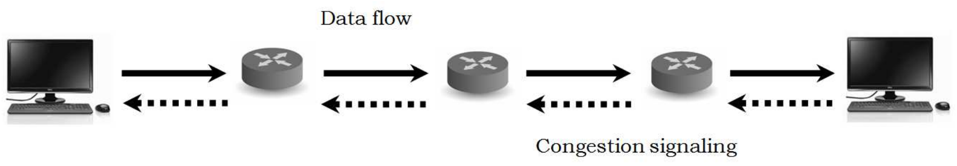


Figure 2.7: Hop-by-hop Choke packets

Choke packets + ECN

This method stems from the fact that while ECN is being deployed gradually in the modern Internet routers, the old routers still continue to use the choke packets. As a result, a combination of choke packets + ECN is being used currently to provide the explicit feedback to the end-hosts. The advantages of ECN, however, cannot be leveraged unless every router on the path from source to destination enables the ECN option.

2.3 TCP Congestion Control

A series of congestion collapse in Internet is observed for the first time in October 1986. Since then the congestion control mechanisms have evolved significantly. A simple congestion control mechanism called "Congestion control Using Timeouts at the End-to-end layer (CUTE)" is proposed in (Jain 1986) which is the first mechanism based on increase and decrease algorithms (Chiu and Jain 1989). This mechanism maintains a variable called congestion window (*cwnd*) for each source. Parameters such as maximum value of *cwnd*, minimum value of *cwnd*, *cwnd* initialization policy, *cwnd* increase-policy and *cwnd* decrease-policy are considered in the design of the CUTE mechanism.

In (Jacobson 1988), Jacobson developed a new principle called "Conservation of Packets", which means that a new packet is not injected into the network until an old packet leaves the network. This principle leads to the formation of a key mechanism called "Self-Clocking", which means that the source uses returning acknowledgements (ACKs) as a clock to determine when to send new packets into the network. Three algorithms are designed and implemented: Slow-Start, AIMD and Fast Retransmit. Slow-Start algorithm is designed to start the Self-Clocking mechanism. This algorithm quickly fills the empty pipeline (network is viewed as a pipeline) at the beginning of transmission or after a retransmission timeout (RTO) to bring the connection towards its equilibrium (a connection is said to be in equilibrium if it is running stably with a full window of data in transit). AIMD closely obeys the "Conservation of Packets" principle once the connection is in equilibrium. Fast Retransmit algorithm considers *dupacks* as a sign of packet loss in the network and retransmits the lost packet without waiting for a retransmission timer to expire.

Since the pioneering work of Raj Jain and Van Jacobson, a lot of research effort has gone into the design of fair and efficient congestion control protocols for the Internet. Fig.2.8 (Welzl 2006) shows some of the mechanisms / protocols standardized and recommended for implementation by the IETF.

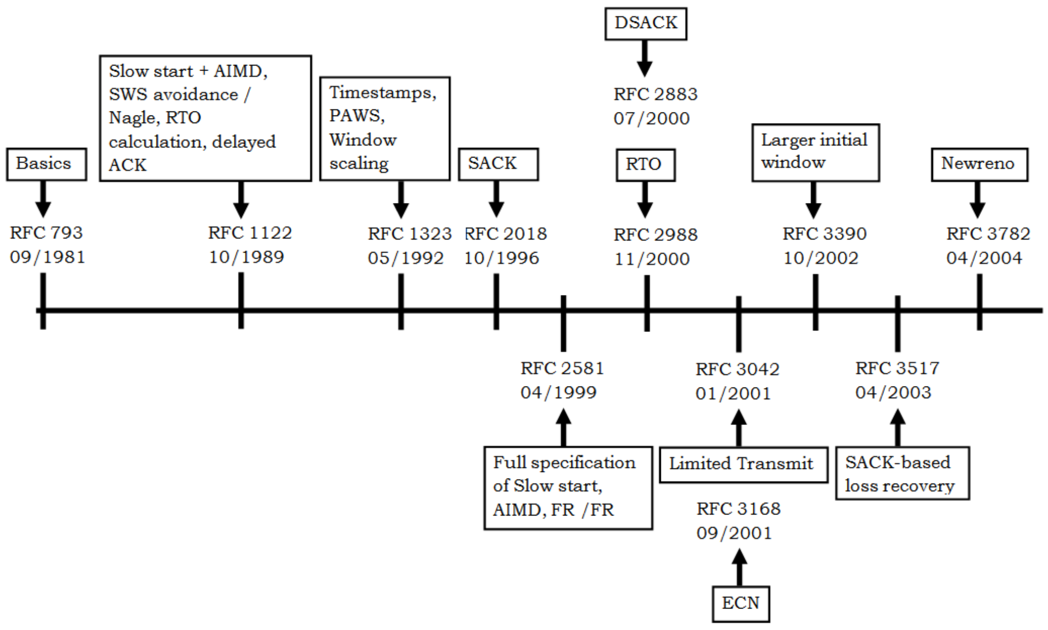


Figure 2.8: Congestion control protocols standardized by IETF

As discussed in the previous chapter, transport protocols with congestion control mechanisms can be mainly classified into three categories (Qazi 2010): (i) *end-to-end protocols* (ii) *network based protocols* and (iii) *end-to-end protocols with explicit feedback*. In the following sections we discuss the work done in each of these categories.

2.3.1 End-to-end protocols

This section briefly reviews the congestion control protocols that preserve the original *end-to-end* semantics of TCP i.e., neither source nor destination relies on any explicit notification from the routers. *End-to-end protocols* treat the network as a *black box* and rely on a packet loss and/or delay as an indication of congestion.

A lot of TCP variants that follow *end-to-end* semantics have been designed and implemented. Protocols that rely on packet loss for congestion indication are termed as loss-based or reactive protocols (Afanasyev et al. 2010). On the other hand, protocols that rely on delay for congestion indication are termed as delay-based or proactive protocols (Afanasyev et al. 2010). (Afanasyev et al. 2010) provide an excellent classification and a detailed description of all such TCP variants, with their strengths and weaknesses. Fig.2.9, extracted from (Afanasyev et al. 2010), shows an evolutionary graph of end-to-end TCP protocols.

As shown in Fig.2.9, TCP Tahoe (Jacobson 1988) is the first TCP implementation that includes Slow-Start, AIMD and Fast Retransmit algorithms. TCP Reno (Jacobson 1990) is the first TCP implementation that includes Fast Recovery along with the algorithms implemented in TCP Tahoe. TCP Vegas (Brakmo and Peterson 1995), on the other hand, includes a modified Slow Start algorithm and uses Additive Increase / Additive Decrease (AIAD) instead of AIMD. The initial design of these protocols laid the foundation for all currently known *end-to-end* TCP protocols. Since then a number of TCP variants have been designed and implemented to: (i) provide robustness against packet reordering (ii) efficiently utilize the available bandwidth in wireless networks by distinguishing congestion losses from non-congestion losses, (iii) achieve high utilization in high-speed/long delay networks, etc.

Several research studies have shown that using packet loss and/or delay as congestion notification pose fundamental limitations in achieving high utilization and fairness. Some of the limitations are listed below:

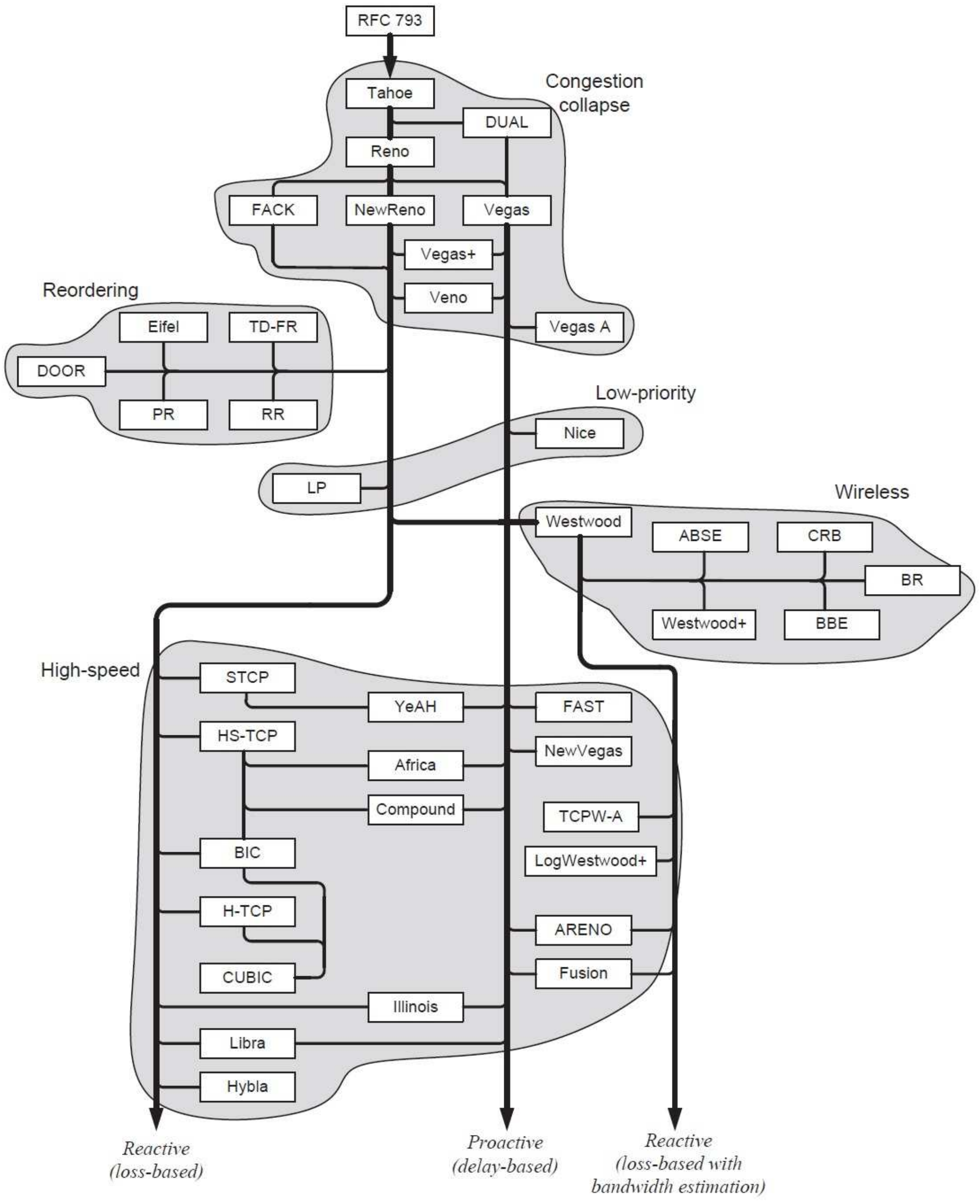


Figure 2.9: Evolution of *end-to-end* TCP protocols

- *Assuming every packet loss indicates congestion* - TCP congestion control mechanisms are fine tuned to perform well in wired networks where packet losses are mostly due to congestion. However, wireless networks such as 802.11 Wireless

LAN, multi-hop wireless mesh networks, etc also suffer from significant packet losses due to reasons such as transmission errors, collisions and link failures. The inability to differentiate between congestion and non-congestion losses significantly degrades the performance of TCP in wireless networks (Balakrishnan et al. 1997).

- *Packet loss is a binary feedback* - A packet loss is a binary feedback of congestion which only indicates whether the network is congested or not congested. It does not provide any additional information about the *degree of congestion.* This forces the sources to be conservative in their increase-policy and aggressive in their decrease-policy, e.g., in AIMD phase, TCP increases the *cwnd* by 1 packet per RTT and when a packet loss is detected, reduces the *cwnd* by half. While this conservative approach fails to utilize the available bandwidth, it also increases the oscillations in the queues and causes instability.

- *Overflows buffers of any capacity* - A packet loss due to congestion occurs only when the buffers overflow. Hence, to obtain such a signal, the network needs to be driven to a point of extreme congestion (Qazi 2010). This leads to the *bufferbloat* problem and degrades the performance of delay sensitive Internet applications.

- *Hard to measure delay reliably* - The protocols that use delay as an indicator of congestion rely on accurate calculation of RTT. There are several factors such as network asymmetry, use of delayed ACKs, etc that affect the calculation of RTT. Moreover, delay-based protocols cannot fairly compete with loss-based protocols, since the former are proactive while latter are reactive.

2.3.2 Network based protocols

Network based protocols overcome most of the limitations of *end-to-end protocols* by relying on explicit feedback from the network. Examples of such protocols include: eXplicit Control Protocol (XCP) (Katabi et al. 2002), Rate Control Protocol (RCP) (Dukkipati 2008), etc.

The main idea of XCP is to generate precise feedback about window increase / decrease from the routers. This feedback assists the sources to regulate their sending rates. However, XCP requires additional 128 bits in the IP header.

RCP is a *network based protocol* in which each router assigns a single rate to all flows passing through it. Calculating this single rate based on the number of active flows passing through a router is a difficult task, considering the dynamic nature of the Internet (Qazi 2010). Moreover, RCP needs additional 96 bits in the existing IP header, making it extremely complicated to deploy.

There are several other *network based protocols* that maintain per-flow accounting and provide richer feedback to the end-hosts. However, the deployment of such protocols is a complex task since they impose significant overhead on the routers. Moreover, the interaction of such protocols with TCP leads to unfairness because *network based protocols* generally tend to avoid congestion. TCP variants deployed in the Internet are loss-based and hence may starve the *network based protocols.*

2.3.3 End-to-end protocols with explicit feedback

This category of protocols rely on a few bits of explicit feedback from the routers to efficiently regulate the congestion control laws at the sources e.g., TCP+AQM/ECN (Hollot et al. 2002). Unlike *network based protocols*, *end-to-end protocols with explicit feedback* have minimal deployment complexity since AQM mechanisms are independent of the congestion control laws. Moreover, *end-to-end protocols with explicit feedback* typically require modifications at the end-hosts.

The success of TCP+AQM/ECN however, relies upto a great extent on the widespread deployment and usage of AQM/ECN. Though the network measurements show a poor usage of AQM/ECN in the Internet, the severe impact of *bufferbloat* on the Internet performance has drawn the attention of several researchers towards the deployment of AQM/ECN.

2.3.4 Slowly responsive TCPs

It has been observed that decrease-by-half is not a fundamental requirement of congestion control (Yang and Lam 2000). In Digital Equipment Corporation bit (DECbit) (Ramakrishnan and Jain 1988), also based on AIMD, a flow reduces its *cwnd* to 7/8 of the old value in response to a packet loss. In (Yang and Lam 2000), a General AIMD (GAIMD) approach is proposed which states that the *cwnd* is increased by α for every window of packets acknowledged and it is decreased to β of the current value when there is a congestion indication. Unlike traditional TCP which implements (α, β) = (1, 0.5), authors in (Yang and Lam 2000) suggest (α, β) = (0.31, 0.875) to improve TCP's friendliness with other competing flows and reduce the fluctuations as compared to traditional TCP flows. (Kwon and Fahmy 2002) further propose a modification to GAIMD by differentiating congestion indications obtained through ECN marks and congestion indications obtained either by a timeout or by three *dupacks*. If congestion indication is obtained through an ECN mark, (Kwon and Fahmy 2002) recommend (α, β) = (0.2, 0.875). However, it retains the basic TCP's response of (α, β) = (1, 0.5) if congestion indication is obtained either by a timeout or by three *dupacks*.

These algorithms which modify the (α, β) parameters of AIMD are termed as *slowly responsive* because unlike TCP, these mechanisms refrain from halving their *cwnd* in response to a packet loss (Bansal et al. 2001). (Bansal et al. 2001) addresses the fundamental question of whether these algorithms are safe to deploy in Internet by investigating the behavior of these algorithms under more realistic dynamic network conditions. The paper concludes that deployment of such algorithms in Internet is safe and does not affect other congestion control mechanisms that co-exist.

Though all the *slowly responsive* algorithms demonstrate the effectiveness of appropriately setting (α, β) parameters for AIMD, none of them attempt to adaptively vary (α, β) based on explicit network feedback about congestion. Rather, these algorithms use a fixed value for (α, β) parameters. Moreover, to the best of our knowledge, the advantages of *slowly responsive* algorithms have not yet been fully studied in the context of wireless networks.

2.4 Problem statement

To optimize congestion avoidance and congestion control mechanisms to maximize the throughput by efficiently allocating resources of a network.

2.5 Objectives

The different objectives of the work are as listed below:

- To design ARED based efficient Active Queue Management (AQM) mechanisms.
- To design an ECN based efficient congestion signaling mechanism that not only differentiates congestion and non-congestion losses in wireless networks, but also provides an information about the *severity of congestion* in the network.
- To optimize the exising AIMD algorithm by adaptively setting the (α, β) parameters based on the *severity of congestion* in the network.
- To study the overall performance improvement in above cases.

Chapter 3

ARED based robust AQM mechanisms

The main objective of AQM mechanisms is to stabilize the router queues and maintain a good trade-off between high throughput and packet drop rate. Although RED has proved to be an effective AQM mechanism, its performance is highly sensitive to the appropriate settings of the parameters. Moreover, RED's performance significantly depends on the amount of traffic load and it has been shown that the average queue length (*avg*) exhibits nonlinear instability (Ranjan et al. 2002, La et al. 2003). Setting the parameters of original RED to achieve equally well performance in different scenarios is yet not clearly known. Rather than tuning the parameters of original RED, this dissertation instead, aims to improve the performance of Adaptive RED (ARED). ARED is an efficient and promising AQM mechanism which overcomes a majority of the problems related to the deployment of original RED.

3.1 Overview of ARED

Adaptive RED (ARED) is a variant of RED that dynamically varies max_p and automatically sets min_{th}, max_{th} and w_q parameters. The main idea of ARED is to modulate max_p between 0.01 (1%) and 0.5 (50%) to maintain the average queue length (*avg*) between a specified *target range*. ARED algorithm and its parameters are shown in Algorithm 1 and Table 3.1 respectively.

Algorithm

If *avg* is below the *target range*, max_p is decreased by β to bring *avg* up within the *target range*. On the other hand, if *avg* is above the *target range*, max_p is increased by α to bring *avg* down within the *target range*. To ensure that a single modification of max_p does not shift *avg* from *below target* to *above target* or vice versa, ARED specifies an upper bound and lower bound for α and β respectively. These bounds are derived based on the choice of the *target range*. Following equations show the upper bound for α and lower bound for β:

$$\alpha < 0.25 \times max_p \tag{3.1.1}$$

$$\beta > 0.83 \tag{3.1.2}$$

Table 3.1: Variables and Fixed Parameters of ARED

Variables	**Fixed Parameters**
max_p	*interval*: 0.5 seconds
avg: average queue length	*target*: $[min_{th} + 0.4 \times (max_{th} - min_{th}), min_{th} + 0.6 \times (max_{th} - min_{th})]$
	β: decrease parameter, 0.9
	α: increase parameter, min(0.01, 0.25*max_p)

Algorithm 1: ARED Algorithm

```
every interval seconds :
if $avg < target$ and $max_p \geq 0.01$ then
    decrease $max_p$ :
    $max_p = max_p \times \beta$
end
else if $avg > target$ and $max_p \leq 0.5$ then
    increase $max_p$ :
    $max_p = max_p + \alpha$
end
```

Advantages

Based on the empirical observations and simulation analysis, (Floyd et al. 2001) show that ARED can stabilize the queue within the specified *target range* and achieve high throughput. Moreover, (La et al. 2003) analytically demonstrate that ARED has more desirable properties of an AQM mechanism as compared to RED such as: (i) the stability of ARED is independent of the number of flows (ii) ARED is less sensitive to round trip propagation delay and (iii) the choice of *target range* provides a trade-off between the stability and smaller queueing delay.

Limitations

Despite several advantages, it can be noted from Algorithm 1 that ARED updates max_p at a slower time scale than *avg*. Hence, max_p converges slowly to a value that stabilizes the *avg* within the *target range*. ARED's conservative approach of updating max_p has two disadvantages: (i) it degrades the throughput, especially when the level of congestion varies sharply, and (ii) since *avg* remains out of the *target range* for sometime, it leads to instability in the queue length.

When the level of congestion varies sharply, it takes atleast 24.5 seconds for ARED to increase max_p from 0.01 to 0.5 or atleast 20.1 seconds to decrease max_p from 0.5 to 0.01 (Floyd et al. 2001). This is a significant amount of time, considering the dynamic nature of the Internet wherein the congestion level keeps varying frequently. Refined Adaptive RED (Re-ARED), described in the next section, addresses this limitation of ARED.

3.2 Overview of Refined Adaptive RED

Refined Adaptive RED (Re-ARED) aims to improve the performance of ARED by reducing the convergence time of max_p i.e. the time taken by max_p to converge to a value that stabilizes the *avg* within the *target range*. The main idea is to update max_p based on the ratio of the change in *avg*. This enables Re-ARED to quickly stabilize the *avg* when the level of congestion varies sharply. Re-ARED algorithm and its parameters are shown in Algorithm 2 and Table 3.2 respectively.

Unlike ARED, Re-ARED sets the *target range* as $[min_{th} + 0.48 \times (max_{th} - min_{th}), min_{th} + 0.52 \times (max_{th} - min_{th})]$ to minimize the queueing delay. However, note that the choice of *target range* provides a trade-off between smaller queueing delay and

Table 3.2: Variables and Fixed Parameters of Re-ARED

Variables	**Fixed Parameters**
max_p	*interval*: 0.5 seconds
avg: average queue length	*target*: $[min_{th} + 0.48 \times (max_{th} - min_{th}), min_{th} + 0.52 \times (max_{th} - min_{th})]$
β: decrease parameter	
α: increase parameter	

Algorithm 2: Re-ARED Algorithm

```
every interval seconds :
if $avg < target$  and $max_p \geq 0.01$ then
    decrease $max_p$ :
    $\beta = 1 - (0.17 \times \frac{target-avg}{target-min_{th}})$
    $max_p = max_p \times \beta$
end
else if $avg > target$  and $max_p \leq 0.5$ then
    increase $max_p$ :
    $\alpha = 0.25 \times max_p \times \frac{avg-target}{target}$
    $max_p = max_p + \alpha$
end
```

stability of *avg*. In the next section, we compare the performance of ARED and Re-ARED over a wide variety of scenarios in *ns*-2 to study the benefits and limitations of each.

3.3 Comparison of ARED and Re-ARED

We compare the performance of ARED and Re-ARED in a diverse set of scenarios such as: varying the number of long flows, the presence of web traffic, reverse traffic and sharply increasing or decreasing the level of congestion. The parameters taken into consideration are throughput, packet drop rate and stability of average queue length (*avg*), with and without sharp variations in the level of congestion. A standard bottleneck topology as shown in Fig. 3.1 is used for all the scenarios.

3.3.1 Throughput and packet drop rate

In this experiment, we vary the amount of traffic load that results in sharp increase in *avg* and observe its impact on the throughput and packet drop rate of ARED and

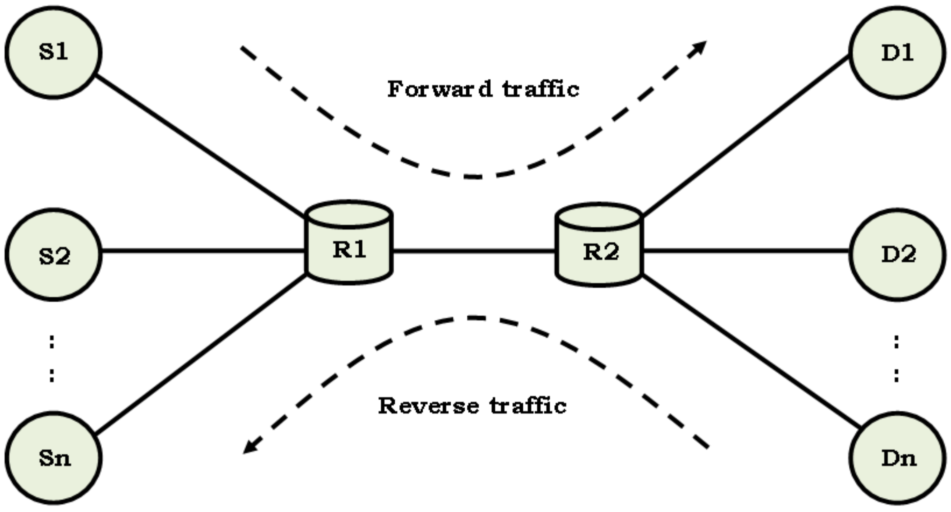


Figure 3.1: Dumbbell topology

Re-ARED. Bottleneck bandwidth is fixed to 10 Mbps and RTT propagation delay is set to 80 milliseconds (ms). File Transfer Protocol (FTP) is used to simulate the long flows. The number of FTP flows is varied from 10 through 100. All flows start at the same time and hence, result in a sharp increase in *avg*. The size of a TCP packet is fixed to 1000 bytes excluding the headers. The target queueing delay is set to 5ms as per the guidelines stated in (Floyd et al. 2001). We do not set min_{th}, max_{th}, w_q and max_p because they are automatically configured by ARED and Re-ARED. Table 3.3 and Table 3.4 show the performance of ARED and Re-ARED in terms of throughput and packet drop rate respectively. R-A represents the improvement of Re-ARED over ARED.

It can be observed that neither ARED nor Re-ARED has consistent better performance when compared to the other. Re-ARED's aggressive approach of updating max_p improves throughput upto 0.13% (12 Kbps) (See Table 3.3) and reduces the packet drop rate upto 0.38% (Table 3.4). On the other hand, ARED's conservative approach of updating max_p achieves better throughput upto 0.12% (See Table 3.3) and incurs less packet drops upto 0.28% (Table 3.4) in a few scenarios. These results reveal that neither a conservative approach alone nor an aggressive approach alone of updating max_p suffices to improve the overall performance.

Table 3.3: Throughput(kbps) of ARED and Re-ARED

FTP Flows	**ARED**	**Re-ARED**	**R-A (%)**
10	9177	9189	**0.13**
20	9171	9182	0.12
30	9175	9186	0.12
40	9186	9185	-0.01
50	9178	9178	0.00
60	9166	9171	0.05
70	9162	9160	-0.02
80	9144	9153	0.10
90	9145	9139	-0.06
100	9108	9097	**-0.12**

Table 3.4: Packet Drop Rate(%) of ARED and Re-ARED

FTP Flows	**ARED**	**Re-ARED**	**R-A(%)**
10	0.23	0.22	0.01
20	0.27	0.27	0.00
30	3.50	3.31	0.19
40	8.21	8.25	-0.04
50	10.41	10.54	-0.13
60	12.49	12.77	**-0.28**
70	13.94	14.02	-0.08
80	15.67	15.29	**0.38**
90	17.00	16.75	0.25
100	17.84	17.80	0.04

3.3.2 Stability of the *avg*

In this section, we compare the behavior of ARED and Re-ARED with respect to the stability of *avg*. We analyze the performance of ARED and Re-ARED in five different types of scenarios: (i) varying the number of long flows (ii) fixed number of long flows without the background traffic (iii) fixed number of long flows with the background traffic (iv) sharp increase in the congestion level and (v) sharp decrease in the congestion level.

Scenario 1: Varying the number of long flows

In this experiment, we analyze the performance of ARED and Re-ARED in a scenario similar to the one designed for analyzing the throughput and packet drop rate. The main idea is to study the impact of one-time burst on the stability of *avg* with ARED and Re-ARED. Figure 3.2a through 3.2d and Figure 3.3a through 3.3d show the comparison of ARED and Re-ARED in terms of stability of *avg* as the number of FTP flows increases i.e. the amount of burstiness increases.

With the increase in the number of FTP flows, ARED takes more time to converge max_p and hence, *avg* tends to remain outside *target range*[1] for a longer time. On the other hand, Re-ARED's aggressive approach converges max_p faster and its performance is robust even when the number of FTP flows is large. This robustness, however, comes at the cost of degraded throughput and/or increased packet drop rate for a few scenarios as shown in Table 3.3 and Table 3.4.

Scenario 2: Fixed number of long flows without the background traffic

In this scenario, we further investigate the stability of *avg* by increasing the bottleneck bandwidth and RTT propagation delay to 15Mbps and 250ms respectively. A simulation scenario similar to the one described in (Floyd et al. 2001) is designed. 100 one-way FTP flows are simulated without any reverse traffic or short flows.

Figure 3.4 and Figure 3.5 show the oscillations in the *avg* with ARED and Re-ARED respectively. The *avg* exhibits smaller oscillations with Re-ARED because it sets a smaller *target range* of $[min_{th} + 0.48 \times (\text{max}_{th}$ - $min_{th}), min_{th} + 0.52 \times (\text{max}_{th}$ - $min_{th})]$[2]. ARED, instead, sets the *target range* as $[min_{th} + 0.4 \times (\text{max}_{th}$ - $min_{th}), min_{th} + 0.6 \times (\text{max}_{th}$ - $min_{th})]$[3]. Both ARED and Re-ARED perform equally well in this scenario by stabilizing the *avg* around their *target range.*

Scenario 3: Fixed number of long flows with the background traffic

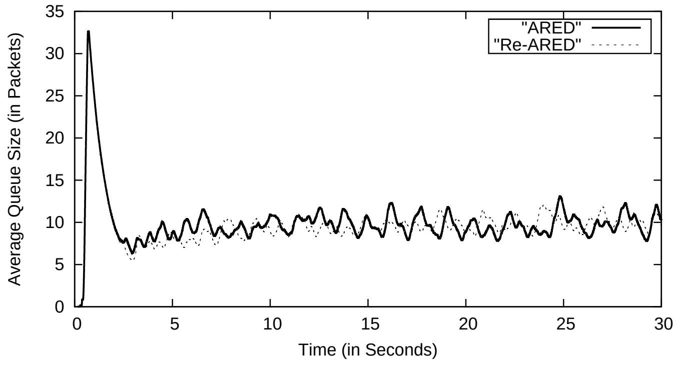
We repeat the above experiment with a slightly more realistice traffic which, apart from 100 long flows, also includes 20 short flows generating web-traffic and one long flow generating the reverse traffic.

Figure 3.6 and Figure 3.7 show the oscillations in the *avg* with ARED and Re-ARED respectively. It can be observed that, although a few irregularities exist, ARED

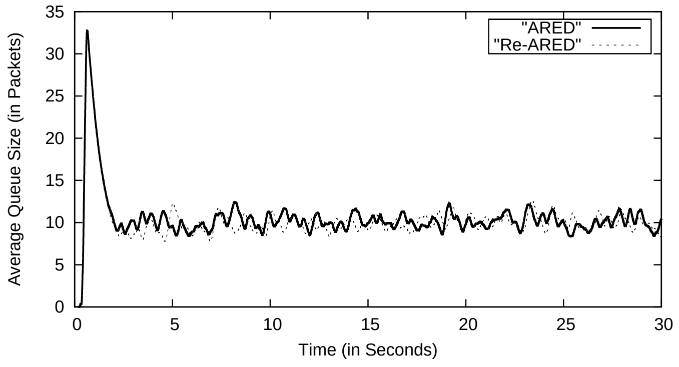
[1]The *target range* for this scenario turns out to be [9, 11]

[2]For min_{th} = 20 and max_{th} = 80, *target range* turns out to be [48.8, 51.2]

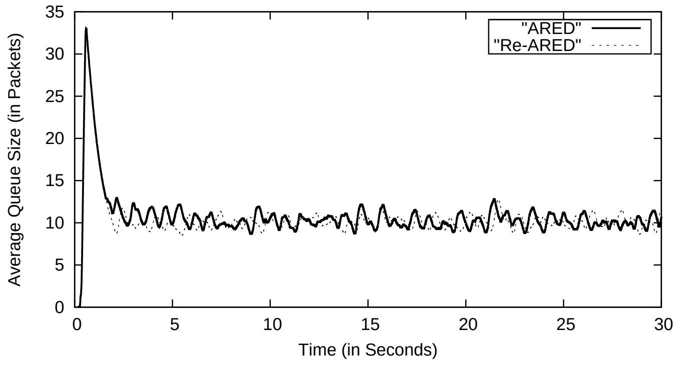
[3]For min_{th} = 20 and max_{th} = 80, *target range* turns out to be [44, 56]



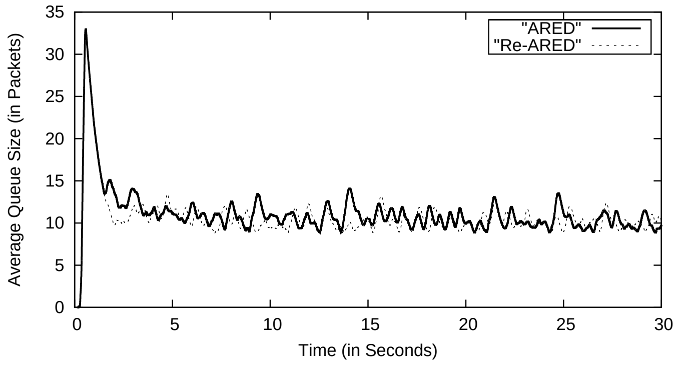
(a) 10 FTP Flows



(b) 20 FTP Flows

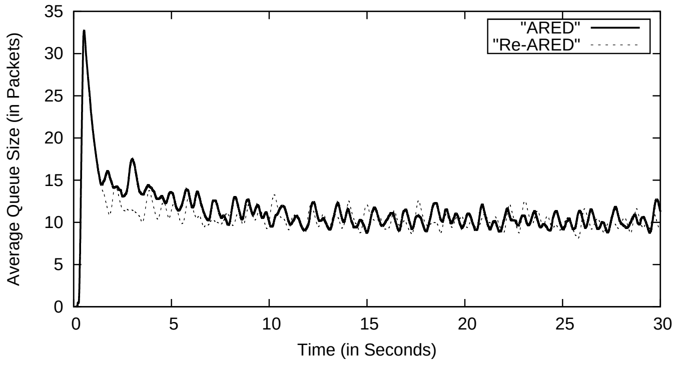


(c) 30 FTP Flows

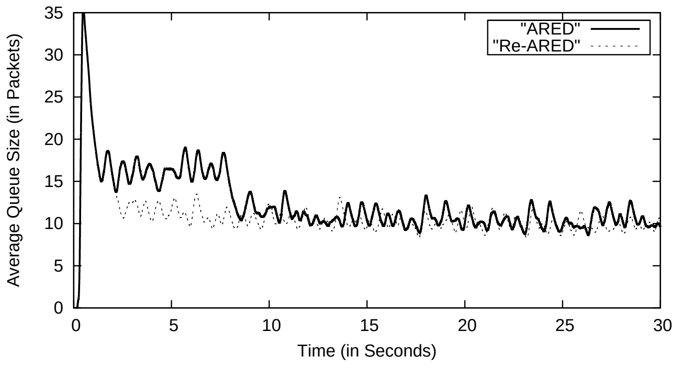


(d) 40 FTP Flows

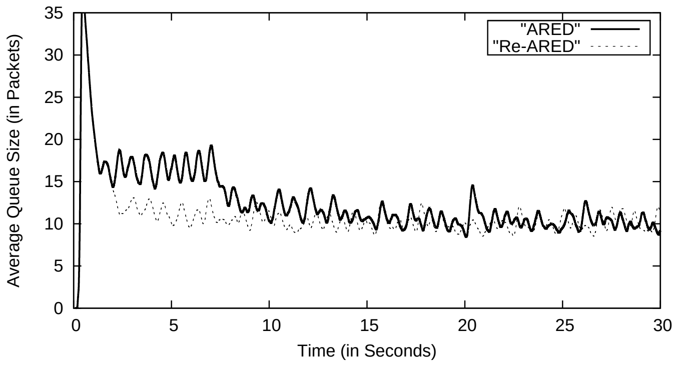
Figure 3.2: *avg* dynamics of ARED and Re-ARED: 10 to 40 FTP Flows



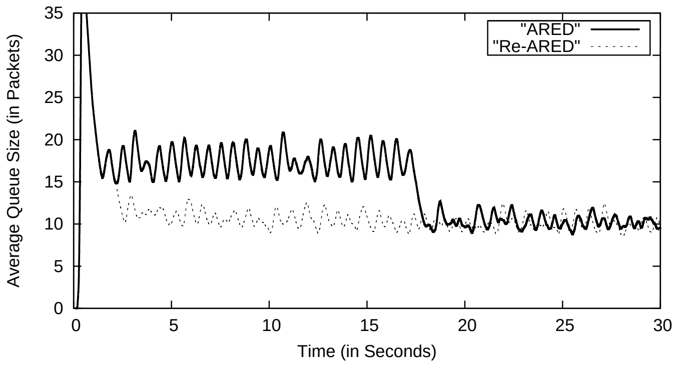
(a) 50 FTP Flows



(b) 60 FTP Flows



(c) 70 FTP Flows



(d) 80 FTP Flows

Figure 3.3: *avg* dynamics of ARED and Re-ARED: 50 to 80 FTP Flows

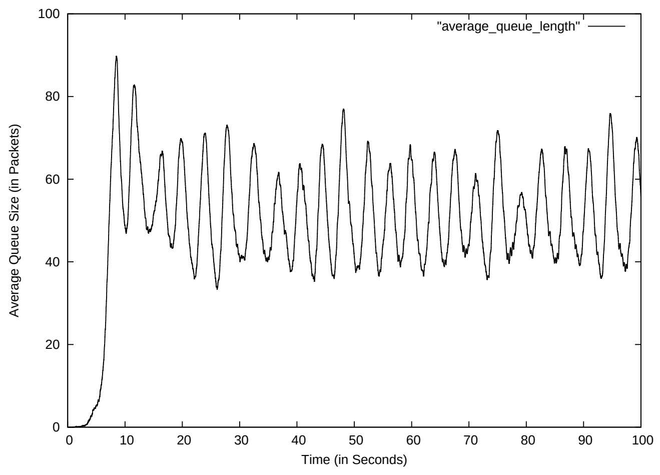


Figure 3.4: Stability of *avg* with ARED - without background traffic

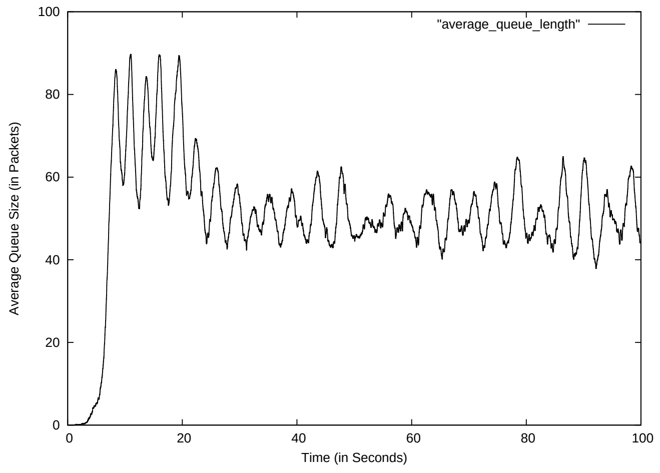


Figure 3.5: Stability of *avg* with Re-ARED - without background traffic

maintains the *avg* within its desired *target range* in the presence of background traffic. On the other hand, *avg* with Re-ARED oscillates out of its desired *target range* and is largely affected by the presence of background traffic. The presence of reverse traffic in the network affects the rate at which the ACKs arrive at the source. This causes short time bursts in the forward TCP traffic which subsequently reflects in the *change of avg*. Since Re-ARED depends on the *ratio of change in avg* to update max_p, its queue length exhibits larger fluctuations when compared to ARED.

Scenario 4: Sharp increase in the congestion level

In this experiment, the major focus is to study the oscillations in instantaneous queue length and *avg* of ARED and Re-ARED when the congestion level increases sharply.

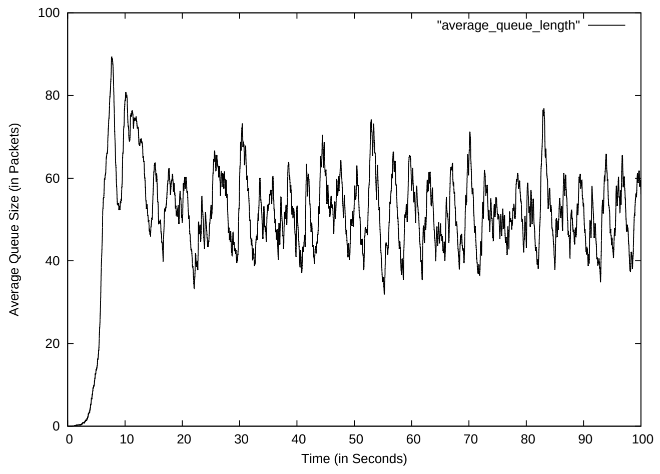


Figure 3.6: Stability of *avg* with ARED - with background traffic

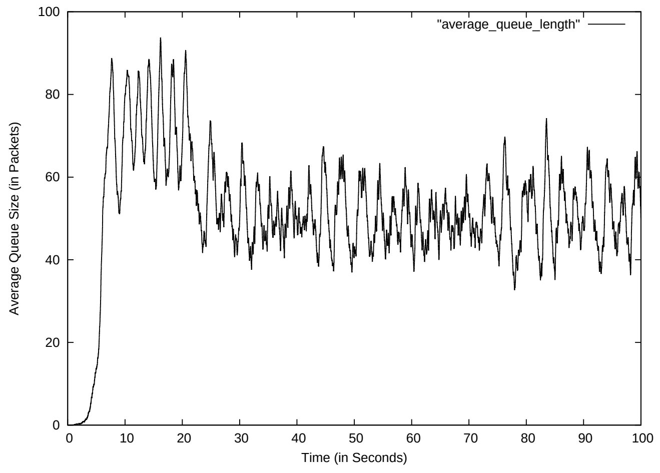


Figure 3.7: Stability of *avg* with Re-ARED - with background traffic

A simple scenario is designed in which the forward traffic consists of two long flows and reverse traffic consists of one long flow. The total simulation time is 50 seconds. To simulate a sharp increase in the congestion level, 20 new flows are started, one every 0.1 seconds, after 25 seconds of the simulation. The scenario designed here is analogous to the one used in (Floyd et al. 2001). Figure 3.8 and Figure 3.9 show the performance of ARED and Re-ARED respectively.

It can be observed that ARED takes 5 seconds more than Re-ARED to stabilize the *avg* within the *target range* because it updates max_p on a longer time scale than *avg*. The *avg* increases sharply with the sudden increase in the level of congestion, but since max_p is updated on a longer time scale than *avg*, it takes considerable amount of time for ARED to bring the *avg* within the *target range*. On the other

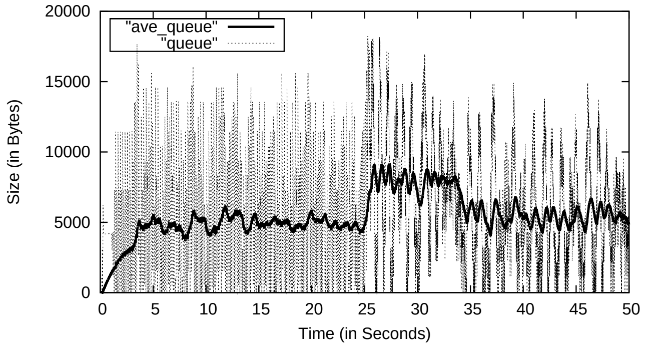


Figure 3.8: ARED with sharp increase in the congestion level

hand, Re-ARED updates max_p aggressively based on the *the ratio of change in avg* and stabilizes the *avg* faster than ARED.

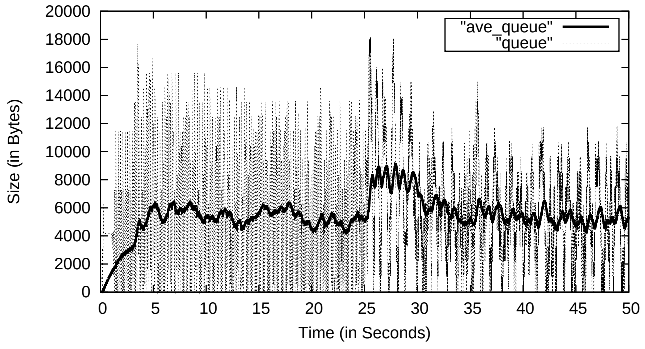


Figure 3.9: Re-ARED with sharp increase in the congestion level

Scenario 5: Sharp decrease in the congestion level

We repeat the above experiment to study the oscillations in instantaneous queue length and *avg* of ARED and Re-ARED when the congestion level decreases sharply. Figure 3.10 and Figure 3.11 show the performance of ARED and Re-ARED respectively.

The aggressive approach to update max_p leads to a significant degradation in the performance of Re-ARED when the congestion level decreases sharply. The queue becomes empty and the bottleneck link is under utilized for a few seconds. Moreover, Re-ARED takes around 15 seconds to stabilize the *avg* within the *target range*. Although ARED takes almost the same time as Re-ARED to stabilize the *avg*, it has a robust performance and utilizes the queue efficiently.

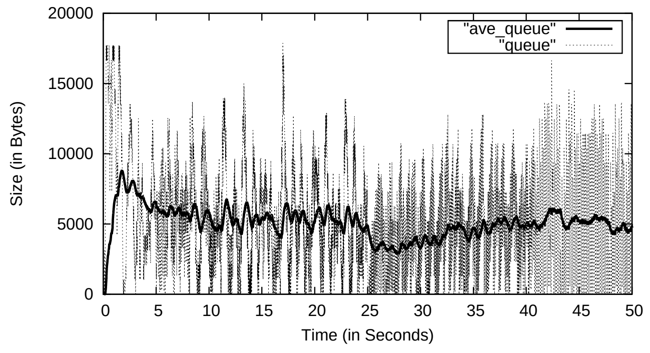


Figure 3.10: ARED with sharp decrease in the congestion level

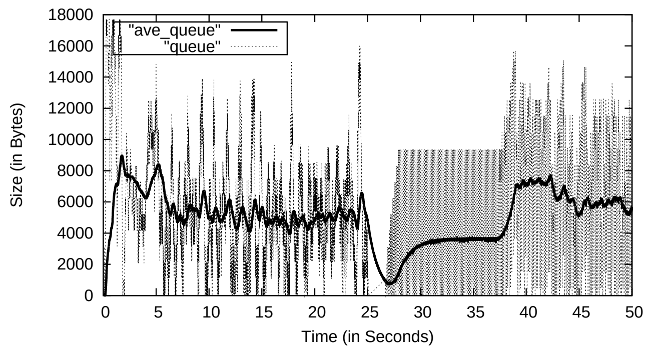


Figure 3.11: Re-ARED with sharp decrease in the congestion level

3.3.3 Inference

The comparative study of ARED and Re-ARED brings out a few important observations of both the AQM mechanisms. We infer the following:

- Neither ARED's conservative approach alone nor Re-ARED's aggressive approach alone of updating max_p suffices to improve the overall performance in terms of throughput and packet drop rate.
- Re-ARED is more robust than ARED in terms of max_p convergence time; especially when there is a sharp increase in the congestion level (See Scenario 1).
- ARED is more robust than Re-ARED in terms of max_p convergence time; especially when there is a sharp decrease in the congestion level (See Scenario 5).
- In the presence of reverse traffic and short flows, ARED offers better stability

of *avg* than Re-ARED. However, in the absence of background traffic, stability of *avg* with both mechanisms is the same (See Scenario 2).

An AQM mechanism that combines the advantages of ARED and Re-ARED is highly desired. In this dissertation, we design and implement two new ARED based AQM mechanisms, namely Fast Adapting RED (FARED) and Cautious Adaptive RED (CARED). The major goal is to stabilize the *avg* in a wide variety of scenarios and at the same time, achieve high throughput and minimize the packet drop rate.

3.4 Fast Adapting RED (FARED)

3.4.1 Overview

If a single modification of max_p shifts *avg* from *below target* to *above target* or vice versa, it may cause instability in *avg*. Hence, while adapting max_p it must be ensured that a single modification of max_p does not exceed *avg* from *below target* to *above target* or vice versa. This can be achieved by appropriately selecting upper bound and lower bound for α and β respectively. Note that these bounds depend on the *target range*.

Eq. (3.1.1) and Eq. (3.1.2) specify an upper bound and lower bound for α and β respectively for ARED's *target range* of [min_{th} + 0.4 $\times$(max_{th} - min_{th}), min_{th} + 0.6 $\times$(max_{th} - min_{th})]. These bounds ensure that single modification of max_p in ARED would not exceed *avg* from *below target* to *above target* or vice versa.

In Re-ARED, bounds for α and β are same as that of ARED, but *target range* is modified to [min_{th} + 0.48 $\times$(max_{th} - min_{th}), min_{th} + 0.52 $\times$(max_{th} - min_{th})]. Since bounds on α and β depend on *target range*, if *target range* changes, even bounds must change. Hence, Re-ARED does not ensure that a single modification of max_p would not exceed *avg* from *below target* to *above target* or vice versa. This may cause instability in the *avg* and due to this reason, we believe, Re-ARED drops the *avg* to zero when there is a sharp decrease in the congestion level (See Fig. 3.11). As a result, we develop Fast Adapting RED (FARED) mechanism which makes minor modifications to Re-ARED by appropriately selecting the bounds for α and β.

3.4.2 Design of FARED Algorithm

FARED algorithm retains the *target range* as specified in Re-ARED algorithm but modifies the upper bound and lower bound for α and β respectively. New bound for α is derived as follows:

Packet drop probability (p) in RED is calculated as follows (Floyd and Jacobson 1993):

$$p = max_p \times (\frac{avg - min_{th}}{max_{th} - min_{th}}) \tag{3.4.1}$$

Before adapting max_p

$$avg_1 = min_{th} + \frac{p}{max_p} \times (max_{th} - min_{th}) \tag{3.4.2}$$

and after adapting max_p

$$avg_2 = min_{th} + \frac{p}{max_p + \alpha} \times (max_{th} - min_{th}) \tag{3.4.3}$$

Subtracting (3.4.3) from (3.4.2)

$$avg_1 - avg_2 = \frac{\alpha}{max_p + \alpha} \times \frac{p}{max_p} \times (max_{th} - min_{th}) \tag{3.4.4}$$

Hence to ensure *avg* does not exceed *above target* to *below target*

$$\frac{\alpha}{max_p + \alpha} < (0.52 - 0.48) \tag{3.4.5}$$

$$\Rightarrow \frac{\alpha}{max_p + \alpha} < 0.04 \tag{3.4.6}$$

$$\Rightarrow \alpha < 0.042 \times max_p \tag{3.4.7}$$

Similarly for β, before adapting max_p

$$avg_1 = min_{th} + \frac{p}{max_p} \times (max_{th} - min_{th}) \tag{3.4.8}$$

and after adapting max_p

$$avg_2 = min_{th} + \frac{p}{max_p \times \beta} \times (max_{th} - min_{th}) \tag{3.4.9}$$

Subtracting (3.4.9) from (3.4.8)

$$avg_1 - avg_2 = \frac{1-\beta}{\beta} \times \frac{p}{max_p} \times (max_{th} - min_{th}) \tag{3.4.10}$$

Hence to ensure *avg* does not exceed *below target* to *above target*

$$\frac{1-\beta}{\beta} < (0.52 - 0.48) \tag{3.4.11}$$

$$\Rightarrow \frac{1-\beta}{\beta} < 0.04 \tag{3.4.12}$$

$$\Rightarrow \beta > 0.9615 \tag{3.4.13}$$

Based on Eq. (3.4.7) and Eq. (3.4.13), the FARED algorithm is shown in Algorithm 3. Note that the only difference between FARED and Re-ARED lies in appropriate selection of upper bound and lower bound for α and β respectively. The parameters of FARED, hence, are same as that of the Re-ARED.

Algorithm 3: FARED Algorithm

```
every interval seconds :
if $avg < target$ and $max_p \geq 0.01$ then
    decrease $max_p$
    $\beta = 1 - (0.0385 \times \frac{target-avg}{target-min_{th}})$
    $max_p = max_p \times \beta$
end
else if $avg > target$ and $max_p \leq 0.5$ then
    increase $max_p$
    $\alpha = 0.042 \times max_p \times \frac{avg-target}{target}$
    $max_p = max_p + \alpha$
end
```

We have implemented the FARED algorithm in *ns*-2 and compared its performance with ARED and Re-ARED. The next section discusses the results.

3.4.3 Results

We verify the performance of FARED against ARED and Re-ARED in terms of throughput, packet drop rate and stability of *avg* by repeating a few experiments carried out in Section 3.3.

Throughput and packet drop rate

In this experiment, we analyze the performance of FARED in terms of throughput and packet drop rate and compare its performance with ARED and Re-ARED. A scenario similar to the one designed in Section 3.3.1 is used for simulations. Table 3.5 and Table 3.6 show the performance of FARED in terms of throughput and packet drop rate respectively. F-A represents the improvement of FARED over ARED and F-R represents the improvement of FARED over Re-ARED.

Table 3.5: Throughput(kbps) and Improvement(%) of FARED

FTP Flows	**ARED**	**Re-ARED**	**FARED**	**F-A**	**F-R**
10	9177	9189	9193	0.17	0.04
20	9171	9182	9189	**0.20**	0.08
30	9175	9186	9186	0.12	0.00
40	9186	9185	9195	0.10	0.11
50	9178	9178	9187	0.10	0.10
60	9166	9171	9181	0.16	0.11
70	9162	9160	9168	0.07	0.09
80	9144	9153	9156	0.13	0.03
90	9145	9139	9139	-0.06	0.00
100	9108	9097	9108	0.00	**0.12**

Table 3.6: Packet Drop Rate(%) and Improvement(%) of FARED

FTP Flows	**ARED**	**Re-ARED**	**FARED**	**F-A**	**F-R**
10	0.23	0.22	0.22	0.01	0.00
20	0.27	0.27	0.26	0.01	0.01
30	3.50	3.31	2.42	**1.08**	**0.89**
40	8.21	8.25	8.03	0.18	0.22
50	10.41	10.54	10.34	0.07	0.20
60	12.49	12.77	12.34	0.15	0.43
70	13.94	14.02	13.75	0.19	0.27
80	15.67	15.29	15.27	0.40	0.02
90	17.00	16.75	16.93	0.07	-0.18
100	17.84	17.80	17.82	0.02	-0.02

Results demonstrate that FARED algorithm achieves consistently better performance than ARED and Re-ARED in terms of throughput and packet drop rate. When compared to ARED, FARED improves the throughput upto 0.20% (18 Kbps) (See

Table 3.5) and reduces packet drop rate upto 1.08% (See Table 3.6). Similarly when compared to Re-ARED, FARED improves the throughput upto 0.12% (11 Kbps) (See Table 3.5) and reduces packet drop rate upto 0.89% (See Table 3.6). However, in the next section we show that the FARED's improvement in throughput and reduction in packet drop rate is due to the over utilization rather than the efficient utilization of the queue.

Varying the number of long flows

In this experiment, we study the impact of one-time burst on the stability of *avg* with FARED. Fig. 3.12a through 3.14d show the comparison of FARED, ARED and Re-ARED in terms of stability of *avg* as the number of FTP flows increases i.e. the amount of burstiness increases.

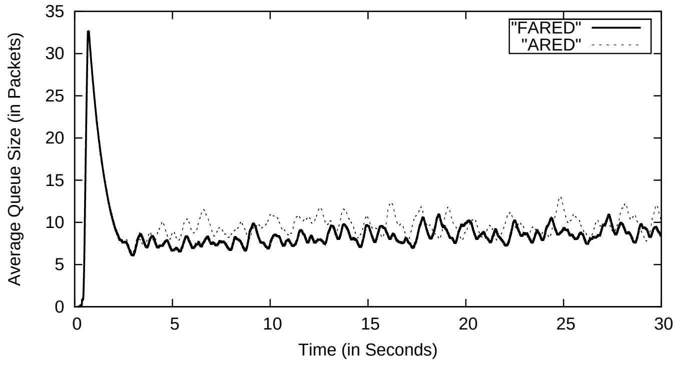
Although the main goal of FARED is to stabilize the *avg* faster (by ensuring that a single modification of max_p does not exceed *avg* from *below target* to *above target*), we observe that it stabilizes *avg* much slower than Re-ARED. This mismatch can be understood by comparing Eq.(3.4.7) with Eq.(3.1.1) and Eq.(3.4.13) with Eq.(3.1.2).

Eq.(3.4.7) limits the increase in max_p to 4.2% of current max_p value whereas Eq.(3.1.1) allows max_p to increase by 25% of its current value. Similarly, Eq.(3.4.13) limits the decrease in max_p to 96.15% of current max_p value whereas Eq.(3.1.1) allows max_p to decrease by 83% of its current value. Thus, setting appropriate values for α and β in FARED, infact, leads to diminishing returns because the increase and decrease of max_p is restricted by the choice of a smaller *target range*. This is confirmed by the results obtained in the next few sections.

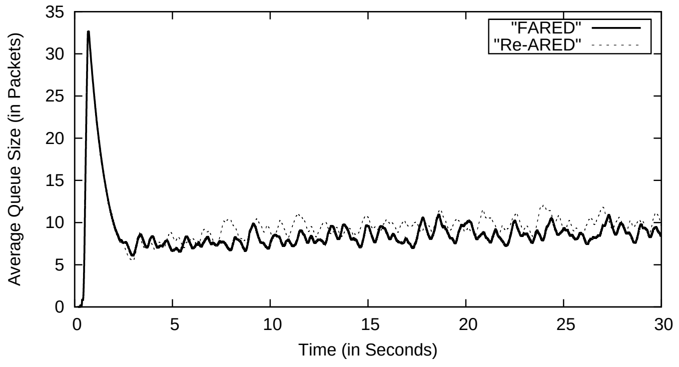
Sharp increase in the congestion level

The focus of this experiment is to study the oscillations in the instantaneous queue length and *avg* of FARED when the congestion level increases sharply. The simulation scenario is similar to the one designed in scenario 4 of Section 3.3.2.

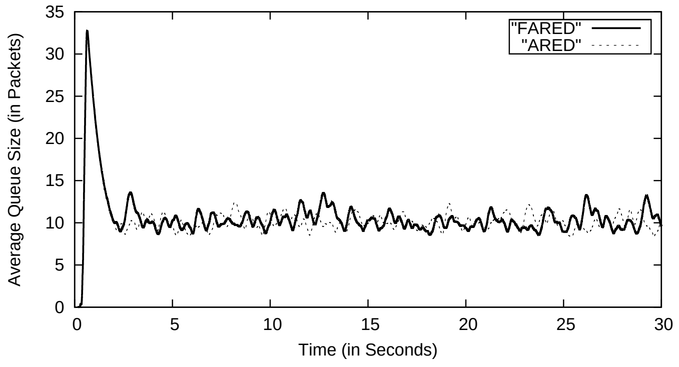
The drawback of selecting smaller *target range* in FARED is clearly visible in Fig. 3.15. FARED fails to stabilize the *avg* when there is a sudden increase in the congestion level. While this increases the queue utilization and certainly improves the



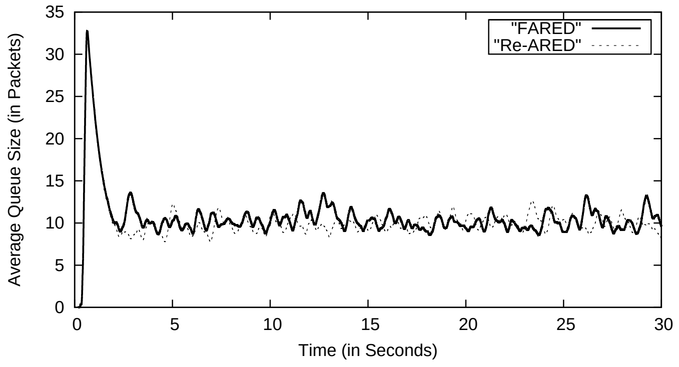
(a) 10 FTP Flows - FARED vs ARED



(b) 10 FTP Flows - FARED vs Re-ARED

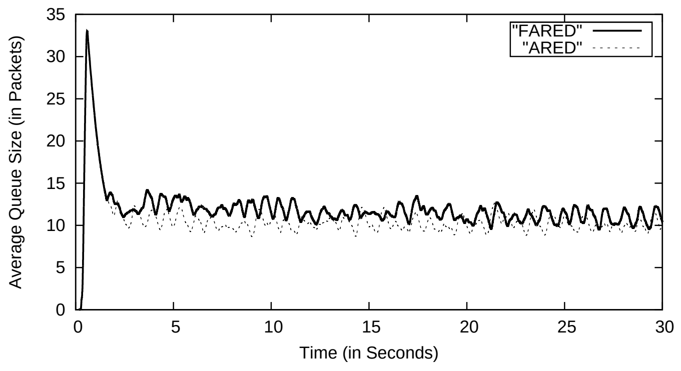


(c) 20 FTP Flows - FARED vs ARED

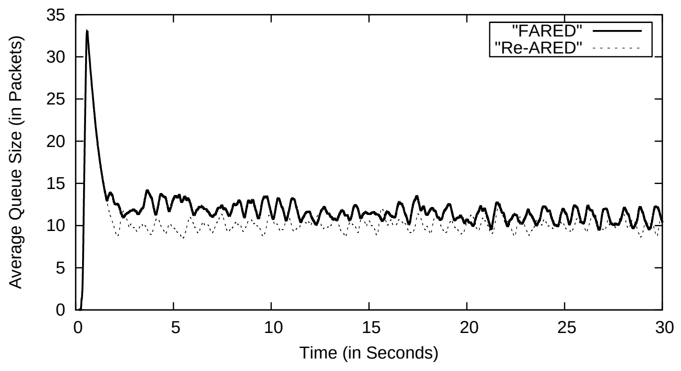


(d) 20 FTP Flows - FARED vs Re-ARED

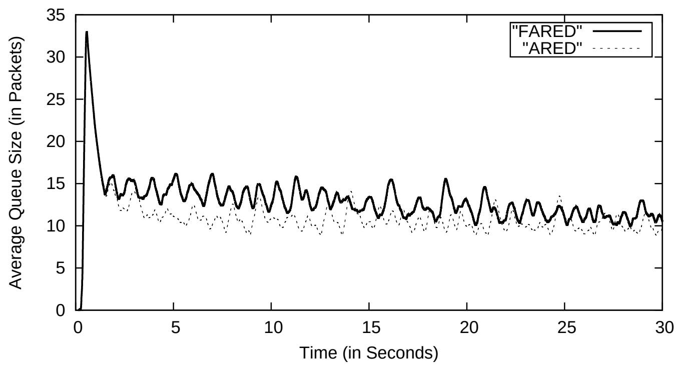
Figure 3.12: *avg* dynamics of FARED, ARED and Re-ARED: 10 to 20 FTP Flows



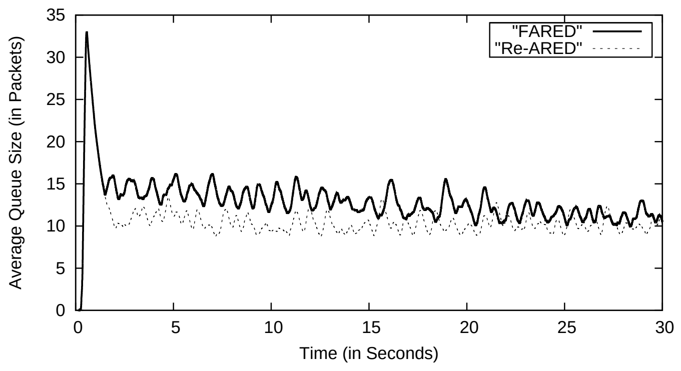
(a) 30 FTP Flows - FARED vs ARED



(b) 30 FTP Flows - FARED vs Re-ARED

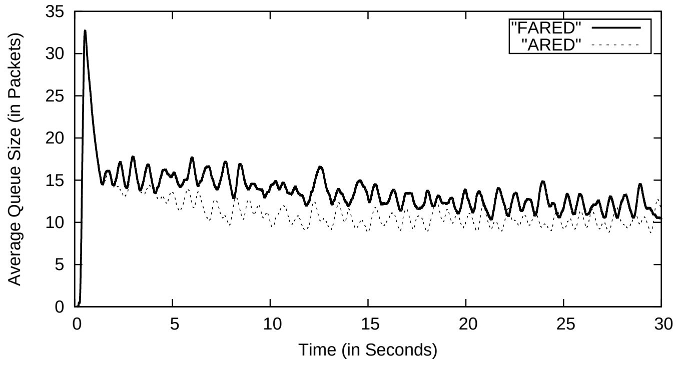


(c) 40 FTP Flows - FARED vs ARED

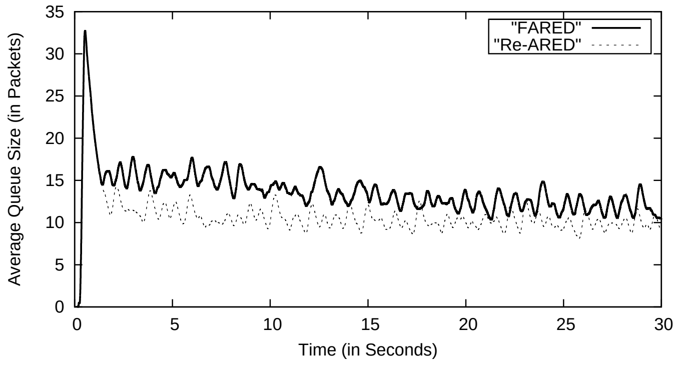


(d) 40 FTP Flows - FARED vs Re-ARED

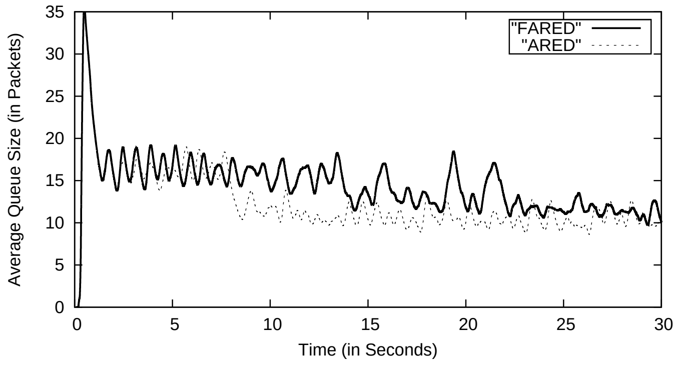
Figure 3.13: *avg* dynamics of FARED, ARED and Re-ARED: 30 to 40 FTP Flows



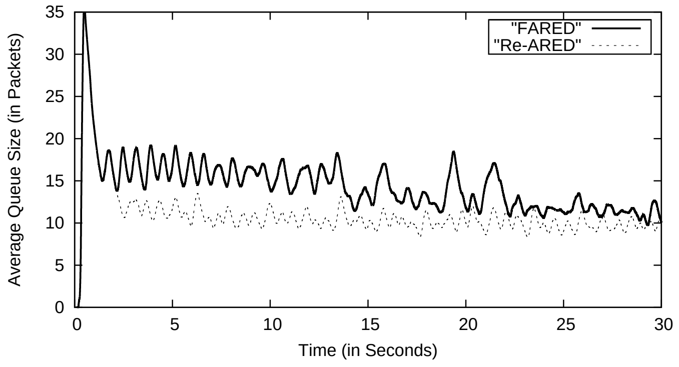
(a) 50 FTP Flows - FARED vs ARED



(b) 50 FTP Flows - FARED vs Re-ARED



(c) 60 FTP Flows - FARED vs ARED



(d) 60 FTP Flows - FARED vs Re-ARED

Figure 3.14: *avg* dynamics of FARED, ARED and Re-ARED: 50 to 60 FTP Flows

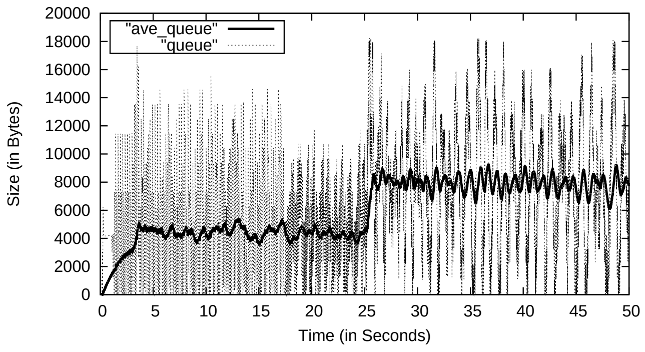


Figure 3.15: FARED with sharp increase in the congestion level

throughput, it degrades the quality for short flows by increasing the queueing delay.

Sharp decrease in the congestion level

We repeat the above experiment to study the oscillations in the instantaneous queue length and *avg* of FARED when the congestion level decreases sharply. Fig. 3.16 shows the results.

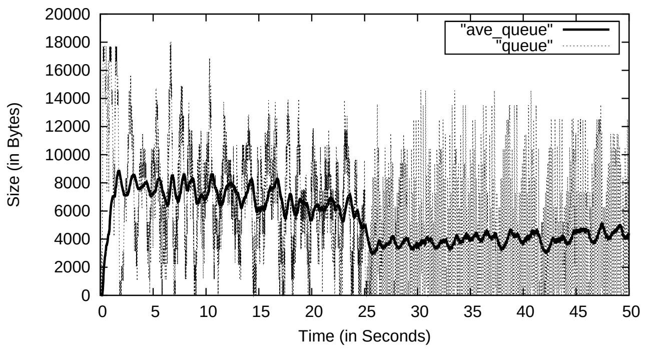


Figure 3.16: FARED with sharp decrease in the congestion level

Unlike Re-ARED, FARED prevents the *avg* from dropping to zero when there is a sharp decrease in the congestion levels. The performance of FARED, however, is not optimal since it does not stabilize the *avg* back within the *target range.*

3.4.4 Inference

Based on the results shown in the previous section, we infer the following about the performance of FARED:

- Although FARED improves the throughput and reduces the packet drop rate as compared to ARED and Re-ARED, its performance improvement is driven by the over utilization of the queue rather than the efficient utilization. Moreover, it leads to instability in the queue and causes variations in the delay.

- Selecting a smaller *target range* significantly affects the stability of *avg* because it imposes constraints on the convergence time of max_p.
- FARED does not meet the desired goal of combining the advantages of ARED and Re-ARED.

3.5 Cautious Adaptive RED (CARED)

3.5.1 Overview

The major objective of designing CARED is to combine the advantages of ARED and Re-ARED and overcome the demerits of FARED. Unlike ARED and Re-ARED that update max_p conservatively and aggressively respectively, CARED algorithm is designed to update max_p either aggressively or conservatively depending on the *level of traffic load*. The main idea of CARED is to carefully modulate max_p when the level of congestion varies sharply. We call the time period during which level of congestion varies sharply as the *"critical time period"*.

The *level of traffic load* is classified into: "up" and "down". While calculating the *avg*, if current average queue length (*newavg*) is greater than previous average queue length (*oldavg*), the *level of traffic load* is considered as "up" since the *avg* is increasing. Similarly if current average queue length (*newavg*) is less than previous average queue length (*oldavg*), the *level of traffic load* is considered as "down" since the *avg* is decreasing. Based on this notion of *level of traffic load*, CARED algorithm efficiently modulates max_p and provides robust performance in a wide variety of scenarios. Setting other parameters such as min_{th}, max_{th}, w_q and *target queuing delay* in CARED is similar to that of ARED. Like ARED, CARED also uses a *target range* of [min_{th} + 0.4 $\times$(max$_{th}$ - min_{th}), min_{th} + 0.6 $\times$(max$_{th}$ - min_{th})].

3.5.2 Design of CARED Algorithm

Table 3.7 shows the variables and fixed parameters used in the CARED algorithm. The design considerations of CARED algorithm are as follows:

- If *newavg* is below $target_{low}$ and the *level of traffic load* is up, max_p is decreased conservatively as per ARED. Aggressively decreasing max_p as per Re-ARED would result in further increase in sending rate which may take *newavg* above $target_{up}$.

- If *newavg* is below $target_{low}$ and the *level of traffic load* is down, max_p is decreased aggressively as per Re-ARED. Conservatively decreasing max_p as per ARED in this scenario would lead to under utilization.

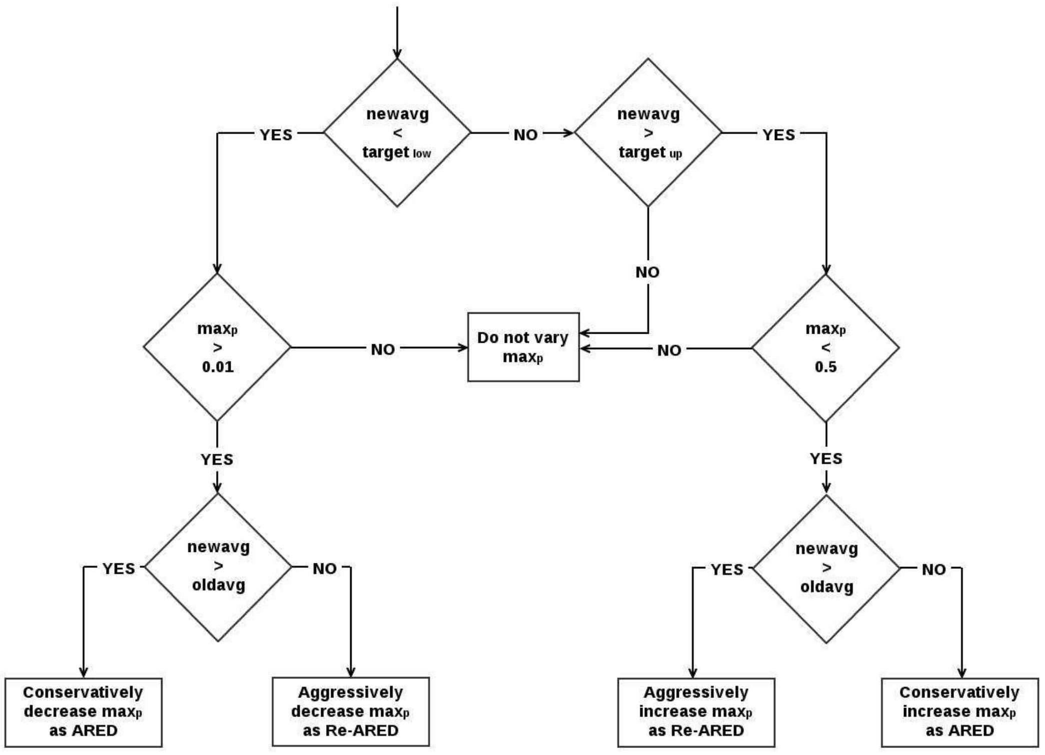
- If *newavg* is above $target_{up}$ and the *level of traffic load* is down, max_p is increased conservatively as per ARED. Aggressively increasing max_p as per Re-ARED in this scenario would drop more packets than required. Since *newavg* is already decreasing and moving towards the *target range*, active packet drops must be conservative rather than aggressive.

- If *newavg* is above $target_{up}$ and the *level of traffic load* is up, max_p is increased aggressively as per Re-ARED to avoid *newavg* from crossing max_{th}. Conservatively increasing max_p as per ARED in this scenario would take more time to bring *newavg* back within the *target range* and hence may affect the throughput and also increase the queueing delay.

Note that like ARED and Re-ARED, CARED also varies max_p within a range of 1% to 50%. CARED design and its parameters are shown in Fig. 3.17 and Table 3.7 respectively. CARED algorithm is shown in Algorithm 4.

Table 3.7: Variables and Fixed Parameters of CARED

Variables	**Fixed Parameters**
max_p	*interval*: 0.5 seconds
newavg: current *avg*	$target_{low}$: min_{th} + 0.4 × (max_{th} - min_{th})
oldavg: previous *avg*	$target_{up}$: min_{th} + 0.6 × (max_{th} - min_{th})
β: decrease parameter	$\beta = 0.9$ for ARED
α: increase parameter	α: min(0.01, 0.25*max_p) for ARED



newavg < target low
YES
NO
newavg > target up
YES
NO
max_p > 0.01
NO
Do not vary max_p
NO
max_p < 0.5
YES
YES
newavg > oldavg
YES
NO
newavg > oldavg
YES
NO
Conservatively decrease max_p as ARED
Aggressively decrease max_p as Re-ARED
Aggressively increase max_p as Re-ARED
Conservatively increase max_p as ARED

Figure 3.17: Flowchart of Cautious Adaptive RED

3.5.3 Results

We verify the performance of CARED against ARED and Re-ARED in terms of throughput, packet drop rate and stability of *avg* by repeating the experiments carried out in Section 3.3.

Throughput and packet drop rate

In this experiment, we analyze the performance of CARED in terms of throughput and packet drop rate and compare its performance with ARED and Re-ARED. A scenario similar to the one designed in Section 3.3.1 is used for simulations. Table 3.8 and Table 3.9 show the performance of CARED in terms of throughput and packet drop rate respectively. C-A represents the improvement of CARED over ARED and C-R represents the improvement of CARED over Re-ARED.

The improvement in throughput achieved by CARED is upto 0.32% (29 Kbps) and 0.2% (18 Kbps) (See Table 3.8) as compared to ARED and Re-ARED respectively. Moreover, CARED reduces the packet drop rate upto 0.51% and 0.43% (See Table

Algorithm 4: CARED Algorithm

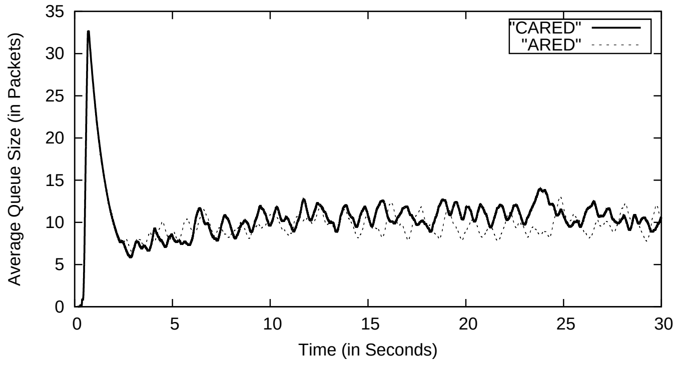
```
every interval seconds :
if $newavg < target_{low}$ and $max_p \geq 0.01$ then
    if $newavg > oldavg$ then
        decrease $max_p$ as per ARED mechanism
        $max_p = max_p \times \beta$
    end
    else if $newavg < oldavg$ then
        decrease $max_p$ as per $Re-ARED$ mechanism
        $\beta = 1 - (0.17 \times \frac{target_{low} - newavg}{target_{low} - min_{th}})$
        $max_p = max_p \times \beta$
    end
end
else if $newavg > target_{up}$ and $max_p \leq 0.5$ then
    if $newavg > oldavg$ then
        increase $max_p$ as per $Re-ARED$ mechanism
        $\alpha = 0.25 \times max_p \times \frac{newavg - target_{up}}{target_{up}}$
        $max_p = max_p + \alpha$
    end
    else if $newavg < oldavg$ then
        increase $max_p$ as per ARED mechanism
        $\alpha = min[0.01, 0.25 \times max_p]$
        $max_p = max_p + \alpha$
    end
end
```

3.9) than ARED and Re-ARED respectively. Since CARED algorithm takes into consideration *the level of traffic load* to switch from ARED to Re-ARED and vice versa, it consistently minimizes the packet drop rate and maximizes the throughput.

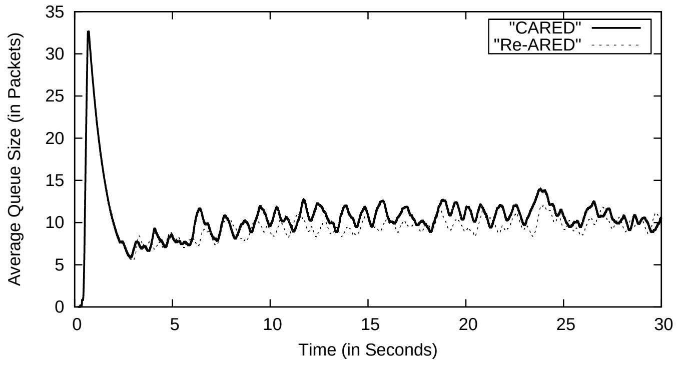
Varying the number of long flows

In this experiment, we study the impact of one-time burst on the stability of *avg* with CARED. Fig. 3.18a through 3.21d show the comparison of CARED, ARED and Re-ARED in terms of stability of *avg* as the number of FTP flows increases i.e. the amount of burstiness increases.

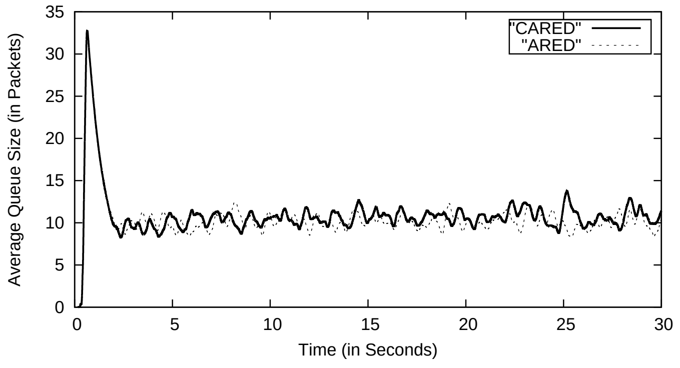
The results show that *avg* exhibits stable behavior with CARED. Fig. 3.21c and Fig. 3.21d show the exact mechanism of CARED where it combines the behavior of ARED and Re-ARED. It converges faster than ARED but slower than Re-ARED based on the *level of traffic load.* Taking a look at the throughput for the same scenario



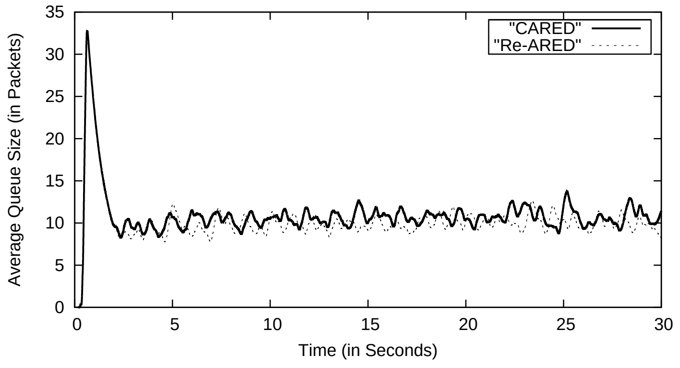
(a) 10 FTP Flows - CARED vs ARED



(b) 10 FTP Flows - CARED vs Re-ARED

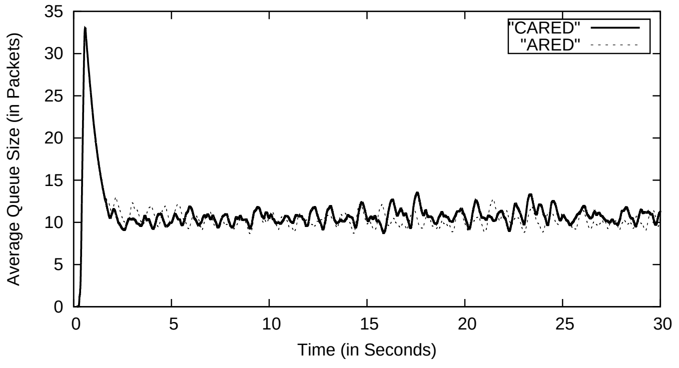


(c) 20 FTP Flows - CARED vs ARED

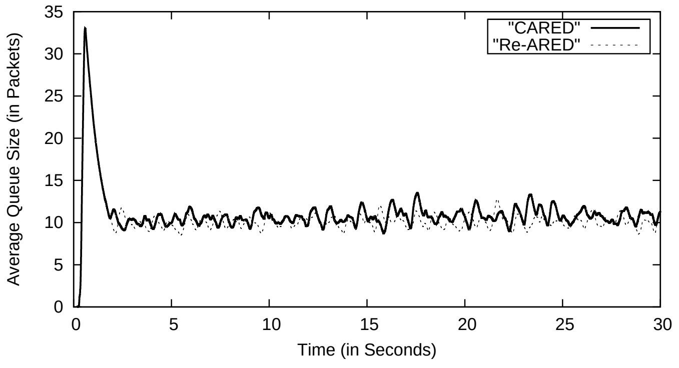


(d) 20 FTP Flows - CARED vs Re-ARED

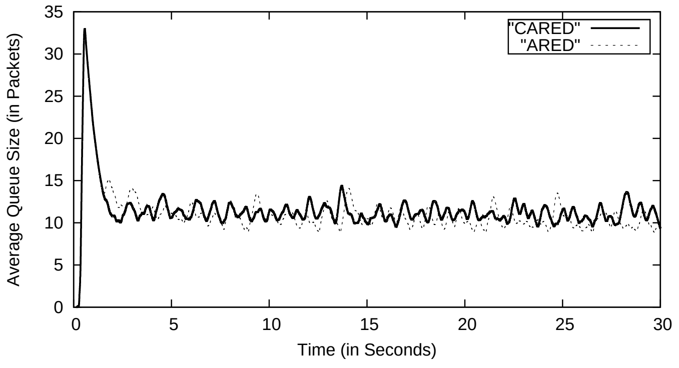
Figure 3.18: *avg* dynamics of CARED, ARED and Re-ARED: 10 to 20 FTP Flows



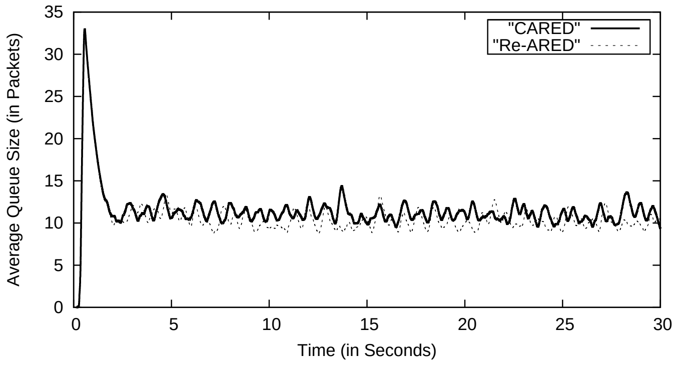
(a) 30 FTP Flows - CARED vs ARED



(b) 30 FTP Flows - CARED vs Re-ARED

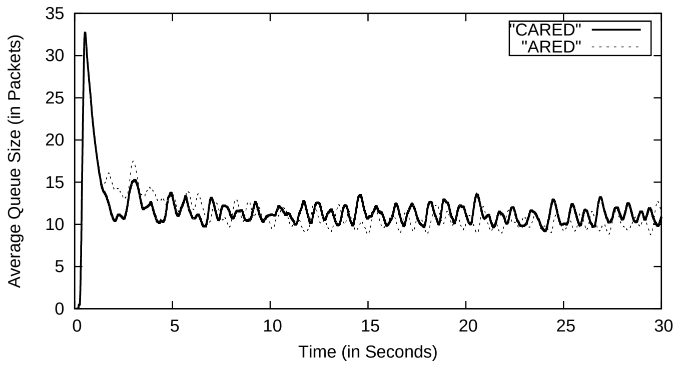


(c) 40 FTP Flows - CARED vs ARED

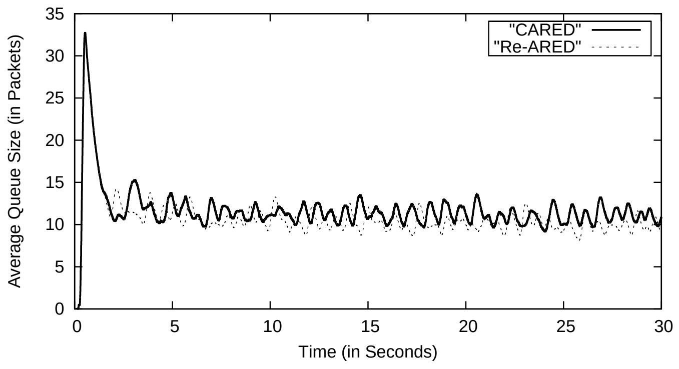


(d) 40 FTP Flows - CARED vs Re-ARED

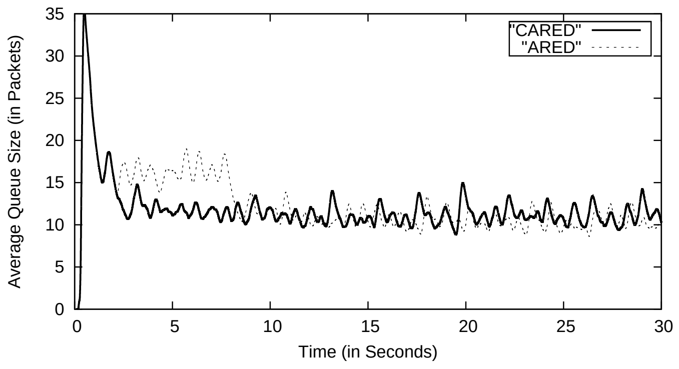
Figure 3.19: *avg* dynamics of CARED, ARED and Re-ARED: 30 to 40 FTP Flows



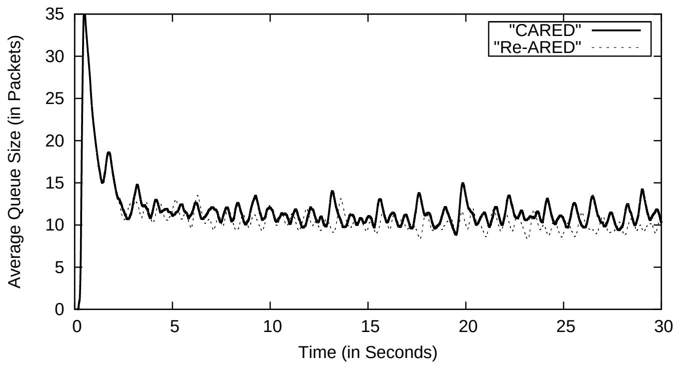
(a) 50 FTP Flows - CARED vs ARED



(b) 50 FTP Flows - CARED vs Re-ARED

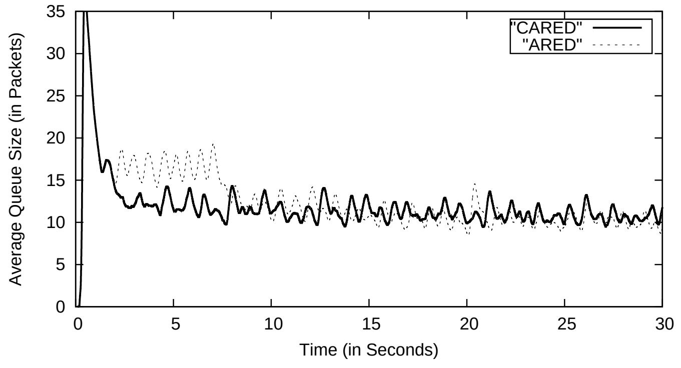


(c) 60 FTP Flows - CARED vs ARED

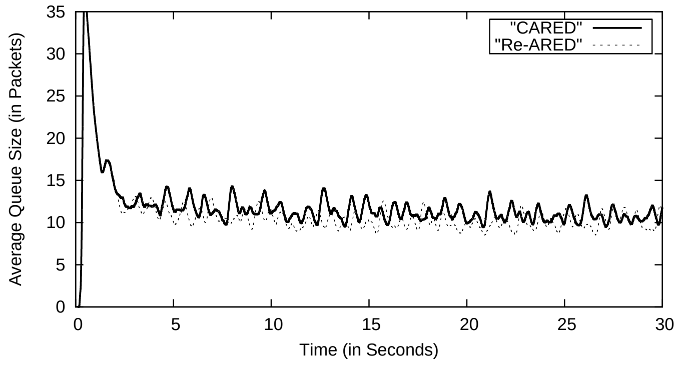


(d) 60 FTP Flows - CARED vs Re-ARED

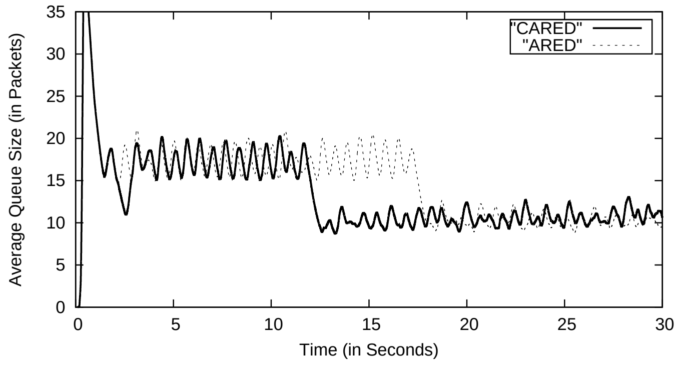
Figure 3.20: *avg* dynamics of CARED, ARED and Re-ARED: 50 to 60 FTP Flows



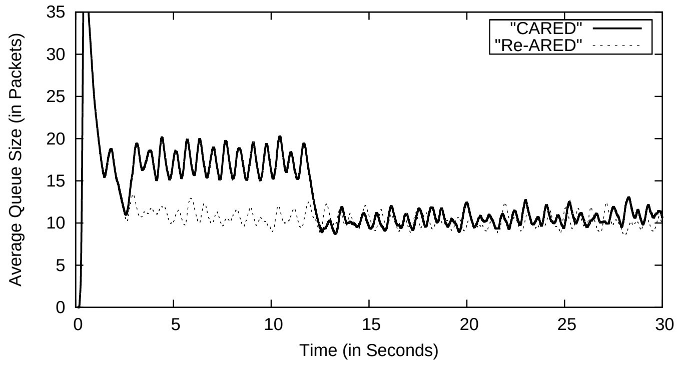
(a) 70 FTP Flows - CARED vs ARED



(b) 70 FTP Flows - CARED vs Re-ARED



(c) 80 FTP Flows - CARED vs ARED



(d) 80 FTP Flows - CARED vs Re-ARED

Figure 3.21: *avg* dynamics of CARED, ARED and Re-ARED: 70 to 80 FTP Flows

Table 3.8: Throughput(kbps) and Improvement(%) of CARED

FTP Flows	**ARED**	**Re-ARED**	**CARED**	**C-A**	**C-R**
10	9177	9189	9189	0.13	0.00
20	9171	9182	9200	**0.32**	**0.20**
30	9175	9186	9195	0.22	0.10
40	9186	9185	9195	0.10	0.11
50	9178	9178	9190	0.13	0.13
60	9166	9171	9178	0.13	0.08
70	9162	9160	9164	0.02	0.04
80	9144	9153	9159	0.16*	0.07*
90	9145	9139	9145	0.00	0.07
100	9108	9097	9110	0.02	0.14

Table 3.9: Packet Drop Rate(%) and Improvement(%) of CARED

FTP Flows	**ARED**	**Re-ARED**	**CARED**	**C-A**	**C-R**
10	0.23	0.22	0.23	0.00	-0.01
20	0.27	0.27	0.26	0.01	0.01
30	3.50	3.31	2.99	**0.51**	0.32
40	8.21	8.25	8.13	0.08	0.12
50	10.41	10.54	10.35	0.06	0.19
60	12.49	12.77	12.34	0.15	**0.43**
70	13.94	14.02	14.01	-0.07	0.01
80	15.67	15.29	15.46	0.21*	0.17*
90	17.00	16.75	16.95	0.05	-0.20
100	17.84	17.80	17.66	0.18	0.14

(i.e. 80 FTP flows) in Table 3.8, we observe that CARED achieves 0.16% and 0.07% more throughput than ARED and Re-ARED respectively. Moreover, packet drop rate with CARED for the same scenario is 0.21% and 0.17% (See Table 3.9) less as compared to ARED and Re-ARED respectively. Thus, CARED improves the overall performance of the network.

Fixed number of long flows without the background traffic

In this experiment, we further investigate the stability of *avg* by simulating CARED in a scenario similar to the one described in scenario 2 of Section 3.3.2. The scenario consists of 100 FTP flows without any background traffic.

Comparing Fig. 3.4, Fig. 3.5 and Fig. 3.22, we observe that CARED incurs

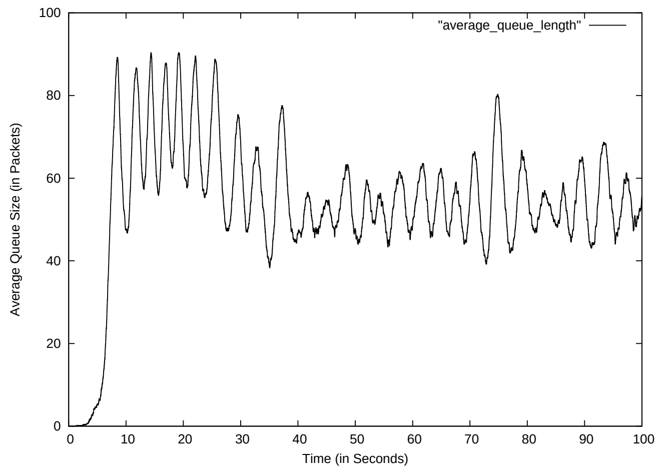


Figure 3.22: Stability of *avg* with CARED - without background traffic

smaller oscillations in *avg* than ARED and slightly larger oscillations than Re-ARED[4]. However, for a *target range* of [44, 56], CARED's performance is far better than that of the Re-ARED.

Fixed number of long flows with the background traffic

We repeat the above experiment with a slightly more realistice traffic which includes 20 short flows generating web-traffic and one long flow generating the reverse traffic.

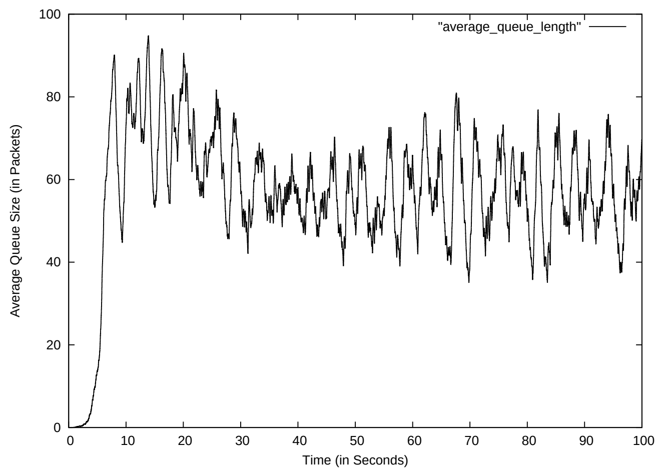


Figure 3.23: Stability of *avg* with CARED - with background traffic

It can be observed from Fig.3.23 that, apart from a few irregularities, CARED maintains the *avg* within its desired *target range* and is not largely affected by the

[4]Note that the *target range* of Re-ARED is smaller than that of CARED

presence of background traffic. This is a desirable property of an AQM mechanism since the Internet traffic is a mix of long, short and reverse flows.

Sharp increase in the congestion level

In this experiment, the major focus is to study the robustness of CARED mechanism in terms of oscillations in the instantaneous queue length and *avg* when the congestion level increases sharply. Scenario similar to the one used in Section 3.3.2 is simulated. Fig. 3.24 shows the results.

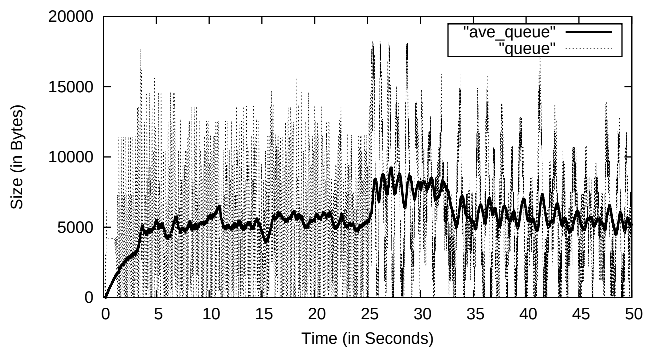


Figure 3.24: CARED with sharp increase in the congestion level

Comparing CARED with ARED and Re-ARED (see Fig. 3.8 and Fig. 3.9 respectively), it can be observed that the performance of CARED is between ARED and Re-ARED i.e. it stabilizes the *avg* a few seconds faster than ARED and a few seconds slower than Re-ARED. Thus, based on the *level of traffic load*, CARED behaves either like ARED or like Re-ARED and provides the benefits of both in a single AQM mechanism.

Sharp decrease in the congestion level

We repeat the above experiment to study the robustness of CARED mechanism in terms of oscillations in the instantaneous queue length and *avg* when the congestion level decreases sharply. Fig. 3.25 shows the results.

We observe that CARED eliminates the shortcomings of Re-ARED in this scenario (see Fig. 3.11) and does not drop the *avg* to zero. Moreover, CARED stabilizes the *avg* faster than ARED.

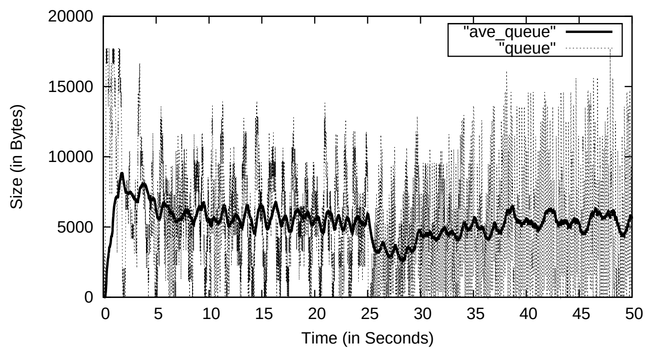


Figure 3.25: CARED with sharp decrease in the congestion level

Variations in max_p

We further validate the performance of CARED with respect to the variations in max_p. We simulate two scenarios for the same: one which is similar to scenario 2 of Section 3.3.2 and the other which includes background traffic, similar to scenario 3 of Section 3.3.2.

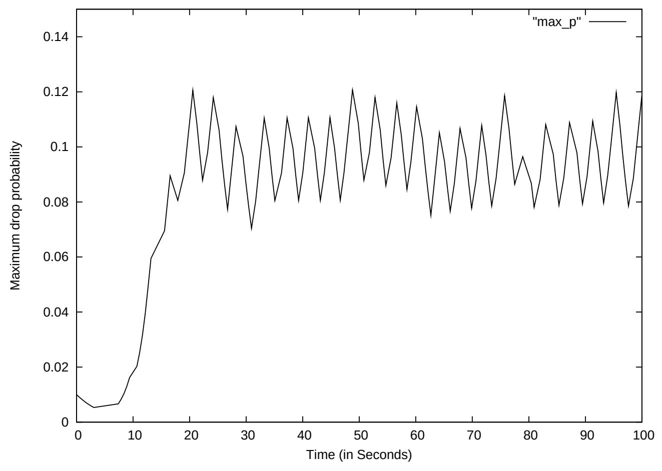


Figure 3.26: Variations in max_p with ARED - without background traffic

Fig. 3.26, Fig. 3.27 and Fig. 3.28 show the variations in max_p with ARED, Re-ARED and CARED respectively when 100 FTP flows exist without any background traffic. Fig. 3.29, Fig. 3.30 and Fig. 3.31 show the variations in max_p with ARED, Re-ARED and CARED respectively when 100 FTP flows exist along with short flows and reverse traffic.

We observe that CARED has minimal variations in both the scenarios and moreover, is not largely affected by the presence of background traffic. A constant value

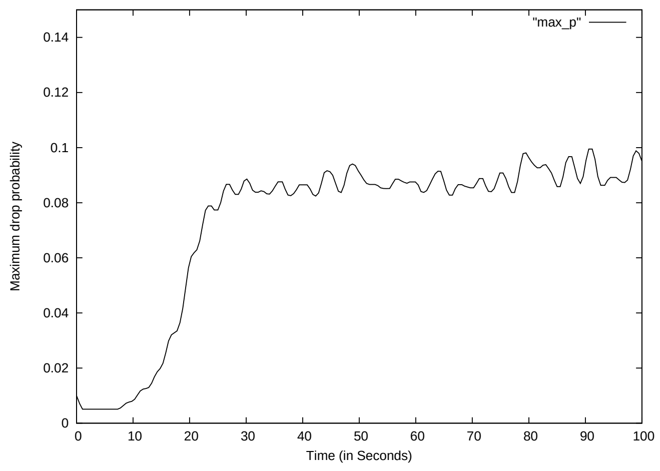


Figure 3.27: Variations in max_p with Re-ARED - without background traffic

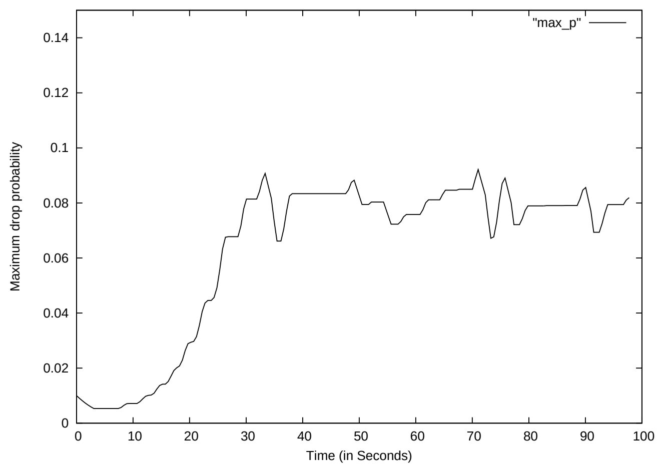


Figure 3.28: Variations in max_p with CARED - without background traffic

of max_p in CARED implies that the *avg* oscillates only within the *target range*. This ensures minimal variations in the delay, better throughput and least packet drop rate, thus satisfying the goals of an AQM mechanism.

3.5.4 Inference

Based on the results obtained in a wide variety of scenarios, we infer the following about the performance of CARED:

- CARED outperforms ARED and Re-ARED by achieving high throughput and minimizing the packet drop rate. The effectiveness of CARED algorithm comes by cautiously adapting max_p during the *critical time period*. Though CARED

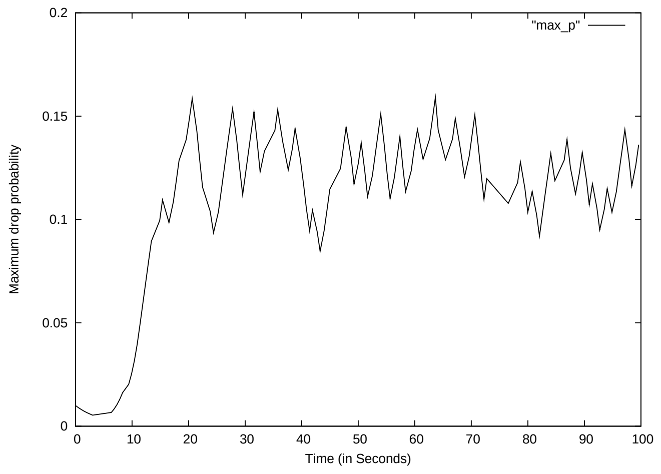


Figure 3.29: Variations in max_p with ARED - with background traffic

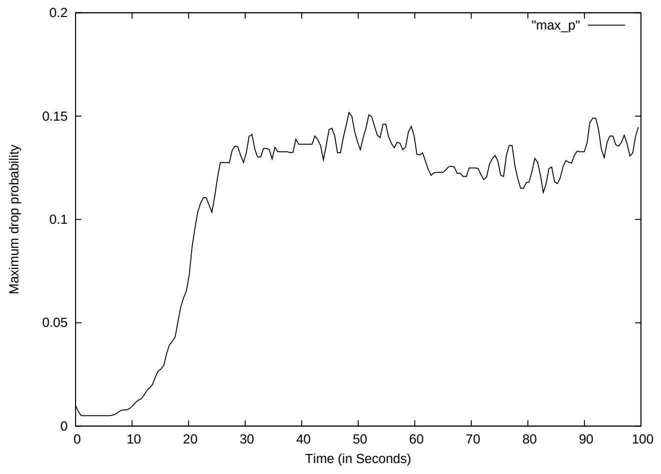


Figure 3.30: Variations in max_p with Re-ARED - with background traffic

algorithm does not achieve least packet drop rate in some scenarios, its performance in such scenarios is always between ARED and Re-ARED. CARED is designed to operate either like ARED or like Re-ARED depending on the *level of traffic load* and hence, in the worst possible scenario, its performance will resort to that of ARED or Re-ARED.

- The design of CARED algorithm gives robust performance in a wide range of scenarios in terms of stability of the *avg*.

- Unlike other RED based algorithms, CARED algorithm does not introduce new parameters to achieve the performance gain. Based on *newavg* and *oldavg*

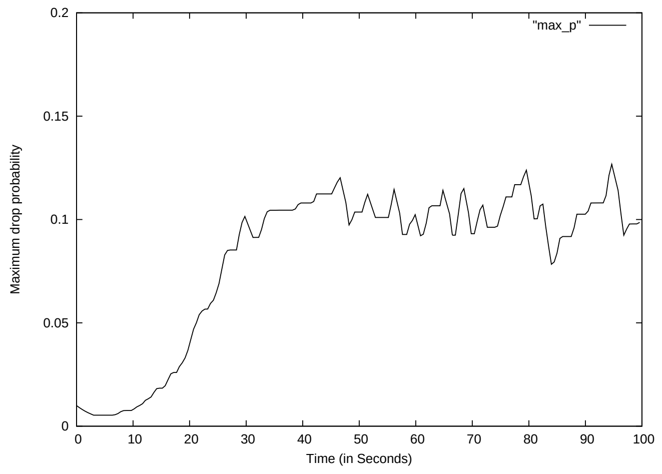


Figure 3.31: Variations in max_p with CARED - with background traffic

values, CARED algorithm infers the *level of traffic load* and varies max_p accordingly. Since there are only algorithmic changes and no new parameter settings in CARED, it can be easily deployed in modern Internet routers without any additional complexity.

To summarize, in this chapter, we have demonstrated that neither ARED's conservative approach alone nor Re-ARED's aggressive approach alone of updating max_p is sufficient to improve the performance in terms of throughput and packet drop rate. Hence, we have designed and implemented two new AQM mechanisms, namely FARED and CARED. We observe that FARED fails to achieve the desired goals of combining the advantages of ARED and Re-ARED because of the choice of smaller *target range*. CARED, on the other hand, outperforms ARED and Re-ARED in terms of throughput, packet drop rate and stability of the *avg* in a wide variety of scenarios. Thus, CARED achieves the goal of combining the benefits of ARED and Re-ARED and hence, improves the overall performance of the network. Moreover, the deployment complexity of CARED is negligible, since it requires only algorithmic modifications to achieve the performance gain and does not require setting of any new parameter.

Chapter 4

Network Controlled TCP

4.1 Overview

In this chapter, we present a newly designed network controlled TCP called "TCP Surathkal". A new variant of ECN called eXtended ECN (XECN) is also developed, which enables the sender to accurately infer the *severity of congestion* in the network. The main aim is to optimize the performance of AIMD algorithm by leveraging the benefits of AQM/ECN and consequently improve the performance of TCP across different types of networks. TCP Surathkal adaptively sets the (α, β) parameters of AIMD based on the *severity of congestion* in a network. To ensure minimal deployment complexity, XECN and TCP Surathkal are designed as incremental modifications on the existing implementations of ECN and TCP respectively.

TCP Surathkal, unlike other TCP variants, does not reduce its congestion window (*cwnd*) by half. Instead, it decreases *cwnd* adaptively and proportionally based on the *severity of congestion* inferred by using the newly designed XECN mechanism. The adaptive decrease mechanism of TCP Surathkal is similar to that of Data Center TCP (DCTCP) (Alizadeh et al. 2010), a TCP variant designed to improve the performance of TCP in Data Center Networks. However, TCP Surathkal differs from DCTCP in two major aspects: (i) DCTCP modifies the original ECN[1], but TCP Surathkal uses XECN to enable the sender to infer *severity of congestion* in the network, and (ii) Unlike DCTCP, the controller of TCP Surathkal sender is based on the instantaneous number of marked packets rather than the average number of marked packets.

[1]The "original ECN" refers to the ECN proposed in RFC 3168

The performance of TCP Surathkal is compared with the existing variants of TCP by carrying out extensive simulations using *ns*-2. The parameters taken into consideration are link utilization or throughput, fairness, stability of *cwnd*, stability of average queue length (*avg*), mean queue length and packet drop rate.

4.2 eXtended ECN (XECN)

4.2.1 Motivation

It has been noticed that the use of ECN in the Internet has increased by three folds in the last few years (Bauer et al. 2011). Moreover, recently the advantages of ECN have been demonstrated in Data Center Networks by using DCTCP. With minor modifications to the original ECN mechanism, DCTCP obtains multi-bit feedback about the *severity of congestion* in the network.

ECN uses two bits in the IP header, namely ECN Capable Transport (ECT) and Congestion Experienced (CE) (See Fig. 2.4), and two bits in the TCP header, namely Congestion Window Reduced (CWR) and ECN Echo (ECE) (See Fig. 2.5) for signaling congestion to the end-hosts. The combination of two bits of IP header or two bits of TCP header is known as a *codepoint* (Ramakrishnan et al. 2000). Table 4.1 and Table 4.2 show the ECN codepoints in the TCP header and the IP header respectively.

Table 4.1: ECN codepoints in the TCP header

Codepoint	**CWR bit value**	**ECE bit value**
Non ECN-set up	0	0
ECN-Echo	0	1
CWR	1	0
ECN-set up	1	1

As described in RFC 3168 - the sender and the receiver must negotiate the use of ECN during the three-way handshake (See Fig. 4.1). If both are ECN capable, the sender marks every outgoing data packet with either ECT(1) codepoint or ECT(0) codepoint. This serves as an indication to the router that both sender and receiver are ECN capable. Whenever congestion builds up, the router marks the data packet

Table 4.2: ECN codepoints in the IP header

Codepoint	**ECT bit value**	**CE bit value**
Non-ECT	0	0
ECT(1)	0	1
ECT(0)	1	0
CE	1	1

by replacing ECT(1) or ECT(0) codepoint by the CE codepoint. When the receiver receives a marked packet with CE codepoint, it infers congestion and hence, marks *a series of* outgoing acknowledgments (ACKs) with ECE codepoint until the sender acknowledges with CWR codepoint (See Fig. 2.6).

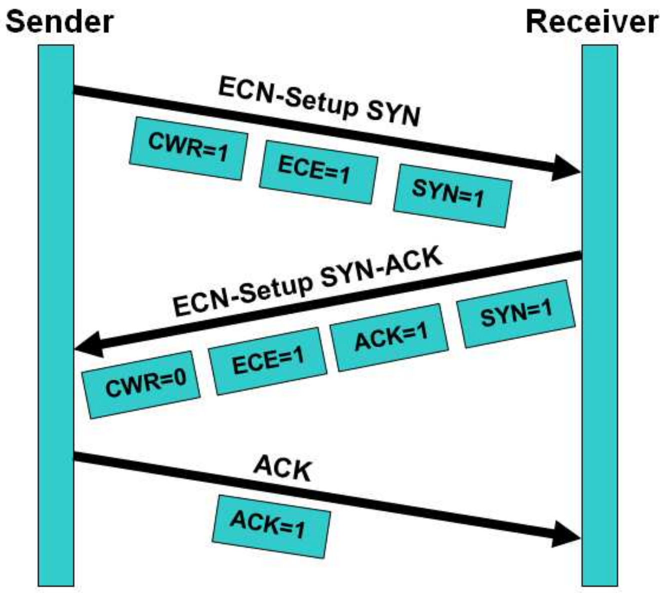


Figure 4.1: ECN Negotiation

The major observation here is that, even if the router marks just one data packet, the receiver continues to mark ACKs with ECE until it receives confirmation from the sender (See Step 3 of Fig. 2.6). This is to ensure the reliability of congestion notification; because even if the first marked ACK is lost, other marked ACKs would notify the sender about congestion. Note that this basic working of ECN aims to only notify the sender about congestion. It is not designed to provide the additional information about the *severity of congestion* to the sender.

At the receiver, counting number of packets marked by the router provides fairly accurate information about the *severity of congestion* in the network. However, conveying this information to the sender by using ECN is a complex task. One of the possible ways is to enable the sender to count the number of marked ACKs it receives from the receiver. The limitation, however, is that even if router marks just one data packet, receiver sends *a series of* marked ACKs. Hence, the number of marked ACKs counted by the sender would be much higher than the number of packets actually marked by the router. This would lead to incorrect estimation of the *severity of congestion* in the network.

To overcome this limitation, DCTCP modifies the basic mechanism of ECN[2]. Unlike TCP receiver which sends *a series of* marked ACKs, DCTCP receiver sends a marked ACK *only when* it receives a marked packet from the router i.e., it sets ECE codepoint in the outgoing ACK *only when* it receives a packet with CE codepoint. Thus, the DCTCP sender obtains an accurate number of packets marked by the router by simply counting the number of marked ACKs it receives. Note that this modification to the original ECN mechanism reduces the reliability because if a marked ACK is lost, sender remains unaware of the congestion and does not reduce the sending rate. However, since Data Center Networks are privately controlled networks, the possibility that an ACK gets lost is negligible.

While DCTCP's modifications to the original ECN are valid in Data Center Networks, these modifications pose several challenging issues if ported to the Internet. Since Internet is a public network, ACKs on the reverse path may encounter congestion and as a result, may be dropped. Hence, we propose XECN mechanism that provides information about the *severity of congestion* to the TCP sender without compromising with the reliability features of the original ECN mechanism. It is to be noted that though XECN mechanism is designed to work with the Internet, it suits equally well in Data Center Networks. Moreover, XECN mechanism does not require any changes in the current architecture and working of the routers. It requires modifications only at the sender and the receiver.

[2]We refer to it as "Modified ECN".

4.2.2 XECN Mechanism

XECN mechanism provides a unique and novel approach to convey the information about *severity of congestion* to the TCP sender without compromising with the reliablity features of the original ECN mechanism. This is achieved by *reusing* one of the four codepoints used in TCP header in an unambigious manner.

Two bits assigned for ECN in TCP header are: CWR bit(8th bit) and ECE bit(9th bit). A combination of these two bits yields four possible codepoints: CWR=0 and ECE=0 (00), CWR=0 and ECE=1 (01), CWR=1 and ECE=0 (10), CWR=1 and ECE=1 (11).

- 01 *is used by the receiver* for two purposes: one to inform the sender during handshake that it is ECN capable and other to notify the sender about congestion in the network (See Fig. 4.1 and Step 3 of Fig. 2.6).

- 10 *is used by the sender* to inform the receiver that congestion notification is received and appropriate action has been taken (See Step 4 of Fig. 2.6).

- 11 *is used by the sender* for ECN negotiation with the receiver during TCP handshake (See Fig. 4.1).

XECN aims to *reuse* 11 codepoint to provide the information about *severity of congestion* from *the receiver to the sender.* The reasons for *reusing* 11 codepoint are two folds:

- 11 codepoint is used only in SYN packet which initiates TCP handshake procedure - this implies that during the ECN negotiation, not only CWR and ECE flags are set to 1, but SYN flag is also set to 1 (See Fig. 4.1). In non-SYN packets (e.g., data packets), both CWR and ECE will never be set to 1, i.e. 11 codepoint is never used when SYN=0. Thus, it is infact a combination of three bits (CWR, ECE and SYN) which is used for ECN negotiation.

- Since 11 codepoint is used presently only with the SYN packet i.e. *from the sender to the receiver* - there will not be any ambiguity if 11 codepoint is reused to provide some information *from the receiver to the sender.*

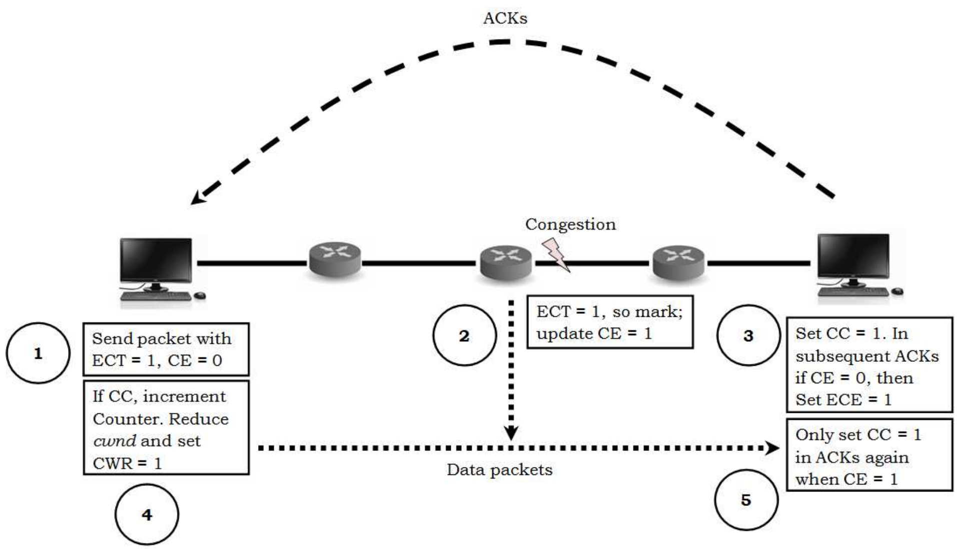

ACKs
Congestion
1
Send packet with ECT = 1, CE = 0
2
ECT = 1, so mark; update CE = 1
3
Set CC = 1. In subsequent ACKs if CE = 0, then Set ECE = 1
If CC, increment Counter. Reduce *cwnd* and set CWR = 1
4
Data packets
Only set CC = 1 in ACKs again when CE = 1
5

Figure 4.2: eXtended ECN

We call 11 codepoint as the Count Codepoint (CC) when used in a non-SYN packet. CC is used only by the receiver. Based on this notion of *reusing* 11 codepoint, XECN mechanisms at the receiver and at the sender are shown in Algorithm 5 and Algorithm 6 respectively. Fig. 4.2 explains the working of XECN in brief.

XECN mechanism at the receiver

Each time the receiver gets a marked packet from the router, it marks the ACK with CC. This approach produces an exact sequence of packets marked by the router at the sender. After sending CC, if packets are received without any mark from the router, receiver keeps marking ACKs with ECE until the sender responds with CWR. This ensures that the reliability of ECN is not compromised in XECN.

Algorithm 5: XECN mechanism at the receiver

```
On arrival of every data packet
if CE then
    Mark the ACK with Count Codepoint (CC)
end
else if sender has not yet responded to congestion then
    Mark the ACK with ECN-Echo (ECE)
end
else
    Do not mark the ACK
end
```

XECN mechanism at the sender

The sender maintains a Counter which counts the total number of packets marked by the router. Whenever the sender receives a marked ACK with CC, it increments the Counter by 1 because an arrival of CC indicates that the receiver has received a marked packet from the router. On receiving the first ACK with CC, sender reduces the *cwnd* and responds to the receiver by setting CWR in outgoing data packet. Window is reduced only once for packets of the same *cwnd*. Note that sender resets the Counter to 0 for every new *cwnd*. Sender neither responds nor increments the Counter for ACKs with ECE, once it receives an ACK with CC for that *cwnd*. However, due to unpredictable behavior of the traffic in the Internet, an ACK with CC may get lost. As a result, sender may directly get an ACK with ECE without previously receiving an ACK with CC. In such cases, sender intelligently infers that the ACK with CC might have been lost because receiver always sends an ACK with CC first and then ACKs with ECE. Hence, for the first ACK with ECE, sender increments the Counter, reduces the *cwnd* and also responds to the receiver by setting CWR in outgoing data packet.

Algorithm 6: XECN mechanism at the sender

```
Reset the Counter for new cwnd
On arrival of every ACK
if CC then
    Increment the Counter by 1
    Mark the data packet with CWR if cwnd is reduced
end
else if ECE and Counter is 0 then
    Increment the Counter by 1
    Mark the data packet with CWR if cwnd is reduced
end
else
    Do not increment the Counter
    Do not mark the data packet with CWR
end
```

Thus, by using XECN, TCP sender obtains multi-bit feedback about the *severity of congestion* from the receiver without compromising the reliability of the original ECN mechanism. However, there are few scenarios during which the sender might not be able to get the accurate information from XECN:

- When the router is highly congested, it marks almost every packet. Suppose the router marks ‘n’ consecutive packets of the same flow. Hence, for each of these ‘n’ consecutive marked packets, receiver sends an ACK with CC. Assuming the worst network state, if all ‘n’ consecutive ACKs with CC get lost, sender directly receives an ACK with ECE for which it increments the Counter by 1 and not by ‘n’. This is because sender is unaware of the loss of consecutive ACKs with CC. Although the sender can predict the total number of lost ACKs because of the cumulative nature of TCP ACKs, there is no way it can predict how many of them were carrying CC.

- Suppose the router marks alternate packets of the same flow. The resulting ACK series from the receiver would be: ACK with CC, ACK with ECE, ACK with CC, ACK with ECE, . . . and so on. Again, assuming the worst network state, if all ACKs with CC are lost, it would result in an inaccurate count at the sender.

In XECN, we assume that all the lost ACKs carried CC and hence, increment the counter accordingly. The justification for the same is as follows:

The average Round Trip Time (RTT) in the Internet is approximately 150ms to 200ms and the upper limit of TCP Retransmission Timeout (RTO) is approximately 2 seconds. It means that in every 250 milliseconds the sender receives an ACK. If an ACK does not arrive within 2 seconds, sender retransmits the packet. Hence it must be noted that if 8 consecutive ACKs are lost (i.e. $n = 8$ in example described above), TCP’s retransmission timer would expire. As a result, TCP sender retransmits the lost packet and resets *cwnd* to 1. Hence whenever $n > 7$ the information about *severity of congestion* is not required by the sender since it resets the *cwnd* to 1 based on RTO. This proves that information provided by XECN mechanism would be useful only when $n = 8$ or less. Thus, the information provided by XECN mechanism will have an inaccuracy of atmost seven packets in the worst network state. Due to this inaccuracy, the sender might reduce the *cwnd* a little more than necessary.

4.3 TCP Surathkal Control Laws

TCP Surathkal leverages the benefits of AQM mechanisms and varies its sending rate intelligently based on the information provided by XECN. The only difference between TCP Surathkal and other TCP variants is the way TCP Surathkal reduces its *cwnd* adaptively based on the network feedback. Other features of TCP such as slow start, additive increase during congestion avoidance phase and recovery of lost packets are retained in TCP Surathkal. The controller at TCP Surathkal sender works as explained below:

Additive Increase Phase

When an ACK is received, *cwnd* is updated as

$$cwnd = cwnd + \frac{1}{cwnd} \tag{4.3.1}$$

Multiplicative Decrease Phase

When the first marked ACK is received, *cwnd* is updated as

$$cwnd = cwnd \times (1 - \frac{\alpha}{2}) \tag{4.3.2}$$

where α ($0 \leqslant \alpha \leqslant 1$) is the fraction of packets marked in the last *cwnd* and is calculated as shown in Eq.(4.3.3). Thus, when congestion is low (α is near 0), *cwnd* is reduced slightly and when congestion is high (α is near 1), *cwnd* is reduced by half, just like traditional TCP.

$$\alpha = \frac{\text{total number of marked ACKs received in last RTT}}{\text{total number of packets sent in last RTT}} \tag{4.3.3}$$

Note that the *cwnd* is increased on arrival of every ACK, whereas it is decreased only once per RTT i.e., when the first marked packet is received by the sender for that *cwnd*. Marked ACK here indicates the codepoints as explained in XECN mechanism.

Features of TCP Surathkal

TCP Surathkal adaptively sets the multiplicative decrease factor of AIMD based on the *severity of congestion* in the network. It is a network controlled TCP which works with any AQM[3], provided that XECN is used as the signaling mechanism.

TCP Surathkal differs from DCTCP since the former uses “XECN” while the latter uses “Modified ECN” for congestion signaling. Moreover, TCP Surathkal sender calculates α (see Eq.(4.3.3)) based on the instantaneous number of marked packets rather than the average number of marked packets like DCTCP sender. DCTCP uses a simple AQM mechanism at the routers which marks the incoming packets based on the *instantaneous queue length* and a pre-determined marking threshold. In a burst situation, the router marks all the packets which are above the marking threshold and as a result, the DCTCP sender receives a chain of consecutively marked ACKs from the receiver. To accommodate this burst, DCTCP sender takes an *average of the marked ACKs* and decreases the *cwnd* accordingly.

On the other hand, TCP Surathkal is designed to work with AQM mechanisms that detect congestion and mark the packets at the routers based on the *average queue length* (*avg*), e.g., RED based AQM mechanisms. *avg* in these mechanisms is calculated by using EWMA which filters out the burst situations by accommodating the incipient congestion. Hence, the TCP Surathkal senders need not take care of the burst situation. Thus, rather than calculating α based on the *average number of marked ACKs*, TCP Surathkal sender uses *instantaneous number of marked ACKs* to calculate α.

Deployment Complexity

TCP Surathkal requires XECN implementation *only* at the sender and the receiver. It does not require any modifications in the working of the routers. Moreover, since XECN is only an algorithmic modification, it can be incrementally deployed over the existing implementations of ECN. Note that XECN does not introduce any additional bit(s), rather, it efficiently utilizes the existing ECN bits.

[3]Eq.(4.3.3), however, must be modified based on the type of AQM mechanism used in the routers.

4.4 Results

We have implemented the XECN mechanism and TCP Surathkal by extending the network simulator, *ns*-2. The performance of TCP Surathkal is compared with other TCP+AQM/ECN variants in a diverse set of scenarios such as: varying the bottleneck bandwidth, varying the RTT, varying the number of flows, etc. In the next section, we compare the performance of TCP Surathkal with TCP SACK[4]. TCP SACK is one of the most widely deployed TCP variant in the modern operating systems except Linux kernel > 2.6.

4.4.1 Comparison of TCP Surathkal and TCP SACK

We compare the performance of TCP Surathkal and TCP SACK in a standard network topology shown in Fig. 4.3. S, R and D represent sources, routers and destinations respectively. "n" different sources are connected to the router R1 which in turn is connected to the router R2. "n" different destinations are connected to the router R2. The link between R1 and R2 is configured to be the bottleneck link to simulate congestion. Default values of the bandwidth and delay for the links between the routers and the end nodes are fixed at 10 Mbps and 2 milliseconds respectively. The bottleneck bandwidth and delay are varied in accordance with the requirements of the experiment i.e., to study the impact of various network parameters on the performance of TCP Surathkal, we vary one parameter at a time, while keeping the other parameters fixed. Each source transfers a bulk of data to the receiver with a packet size of 1000 bytes. ARED is used as an AQM mechanism in routers since it has more desirable properties of an AQM as compared to RED. Target queuing delay is set to 5 milliseconds. ECN and XECN are used as signaling mechanisms by TCP SACK and TCP Surathkal respectively. Queue size at routers is fixed to 50 packets, unless otherwise specified.

File Transfer Protocol (FTP) sources start sending the data at time 0 seconds. All simulations are run for 1100 seconds. The receiver advertises a large window such that the sending rate is not limited by the receiver. The parameters used for

[4]TCP SACK refers to TCP Newreno + SACK

comparison are throughput, fairness, stability of the *cwnd* and stability of the average queue length (*avg*).

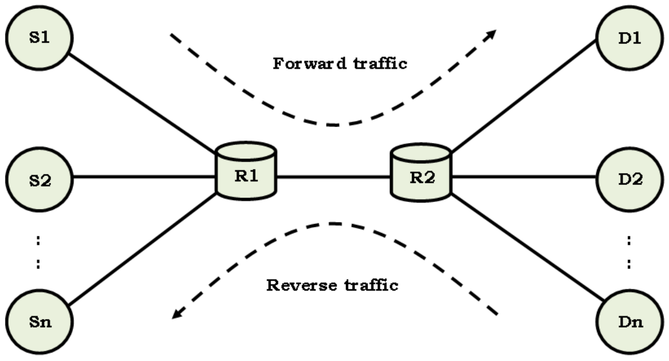


Figure 4.3: Dumbbell topology

Impact of bottleneck capacity

In this experiment, we show the impact of halving the *cwnd* on the performance of TCP SACK, as the bottleneck bandwidth is increased, and the corresponding improvement in the performance of TCP Surathkal. The propagation delay is fixed to 50ms. The topology consists of two FTP flows. The experiment is first run with TCP SACK and repeated for TCP Surathkal. Fig. 4.4 shows the results. The X-axis shows the bottleneck bandwidth in Mbps and the Y-axis shows the total throughput[5] in Mbps.

It can be observed from the graph that TCP Surathkal performs significantly better than TCP SACK. Improvement in throughput of TCP Surathkal is upto 15.82% when compared to TCP SACK. TCP SACK fails to utilize the available bandwidth since it reduces the *cwnd* by half. TCP Surathkal reduces *cwnd* depending on the *severity of congestion* in the network. As a result, when bottleneck bandwidth increases, TCP Surathkal efficiently utilizes the available bandwidth.

Impact of round trip delay

In this experiment, we show the impact of varying the propagation delay on the performance of TCP SACK and TCP Surathkal. By varying the propagation delay, we intend to vary the overall RTT and study its impact on both the protocols. The

[5]Headers are ignored in the throughput calculation.

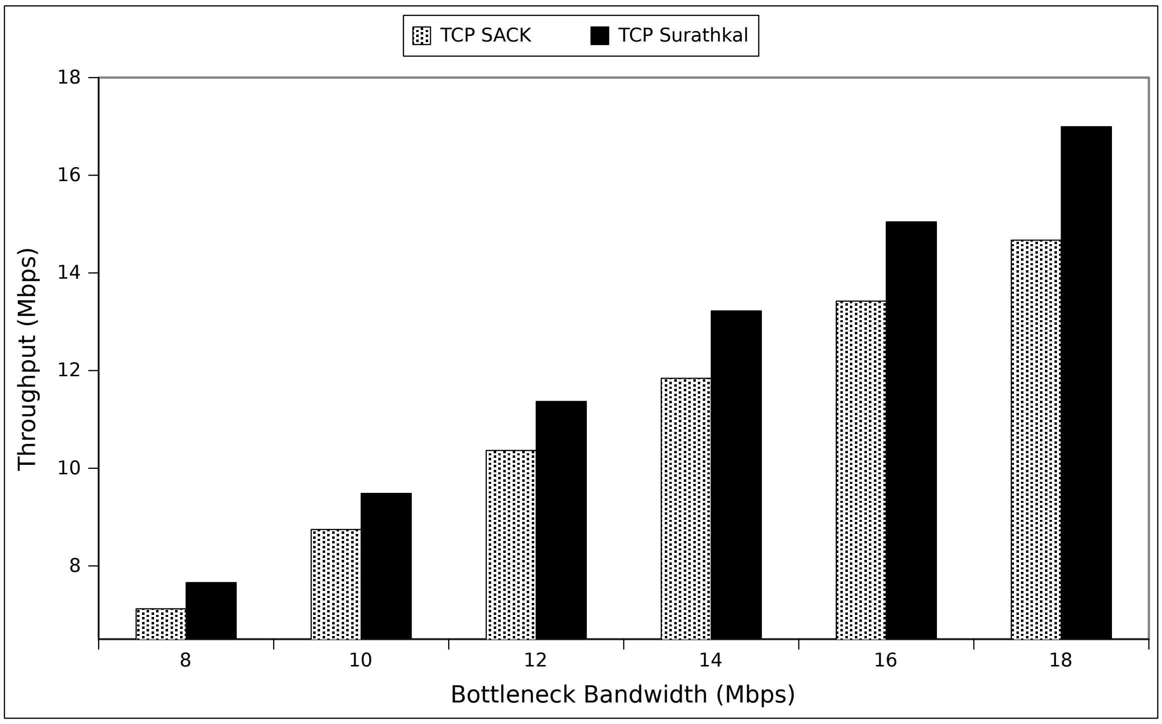


Figure 4.4: Throughput vs Bottleneck bandwidth

bottleneck bandwidth is fixed to 8Mbps. The topology consists of two FTP flows. The experiment is first run with TCP SACK and repeated for TCP Surathkal. Fig. 4.5 shows the results.

Results in the graph show that the performance of TCP SACK degrades rapidly as the propagation delay increases. Higher propagation delay leads to significant increase in the overall RTT and hence, prevents rapid increase of the *cwnd* because *cwnd* is increased only once per RTT. Moreover, TCP SACK reduces *cwnd* by half and hence, takes long time to increase *cwnd* back to the original sending rate i.e. the *cwnd* size when congestion is detected. As a result, the link is under utilized.

On the other hand, increase in the propagation delay has minimal impact on the performance of TCP Surathkal since it reduces *cwnd* depending on the *severity of congestion.* In a few RTTs, TCP Surathkal ramps up back to the original sending rate and thus, uses the available bandwidth efficiently. The improvement in the throughput of TCP Surathkal is upto 11.59% when compared to TCP SACK.

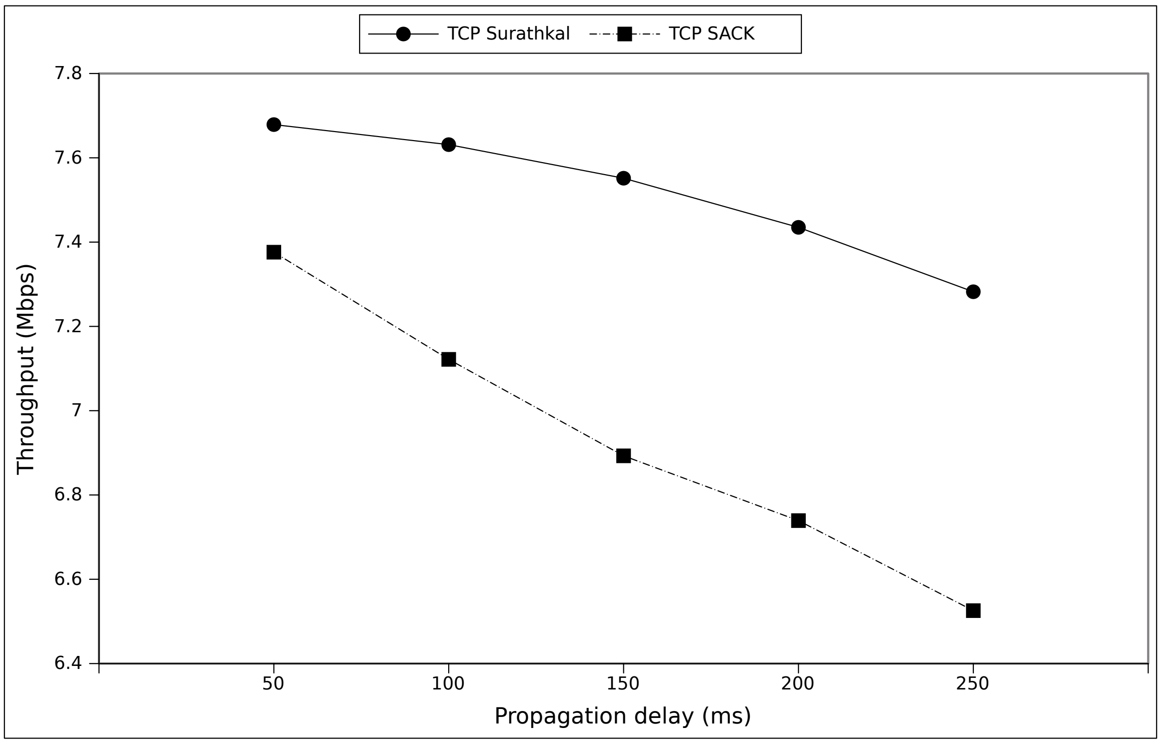


Figure 4.5: Throughput vs Bottleneck propagation delay

Fairness of TCP Surathkal

In this experiment, we study the fairness properties of TCP Surathkal by using Jain's fairness index. Jain's fairness index quantifies the fairness achieved by flows sharing a common bottleneck bandwidth. It is shown below:

$$fairness = \frac{(\sum_{i=1}^{n} x_i)^2}{n \sum_{i=1}^{n} (x_i)^2} \tag{4.4.1}$$

where 'n' represents the total number of flows and x_i represents the throughput of flow i ($1 \leq i \leq n$).

The value of fairness is between 0 and 1. When all flows achieve the same throughput, the fairness equals 1.

We compare the performance of TCP SACK and TCP Surathkal by varying the total number of FTP flows, but sharing the same bottleneck link capacity. The number of FTP flows is varied from 2 to 50, with half of the flows using TCP SACK and the other half of the flows using TCP Surathkal. The bottleneck bandwidth and propagation delay are fixed to 8Mbps and 50ms respectively. Fig. 4.6 shows the results.

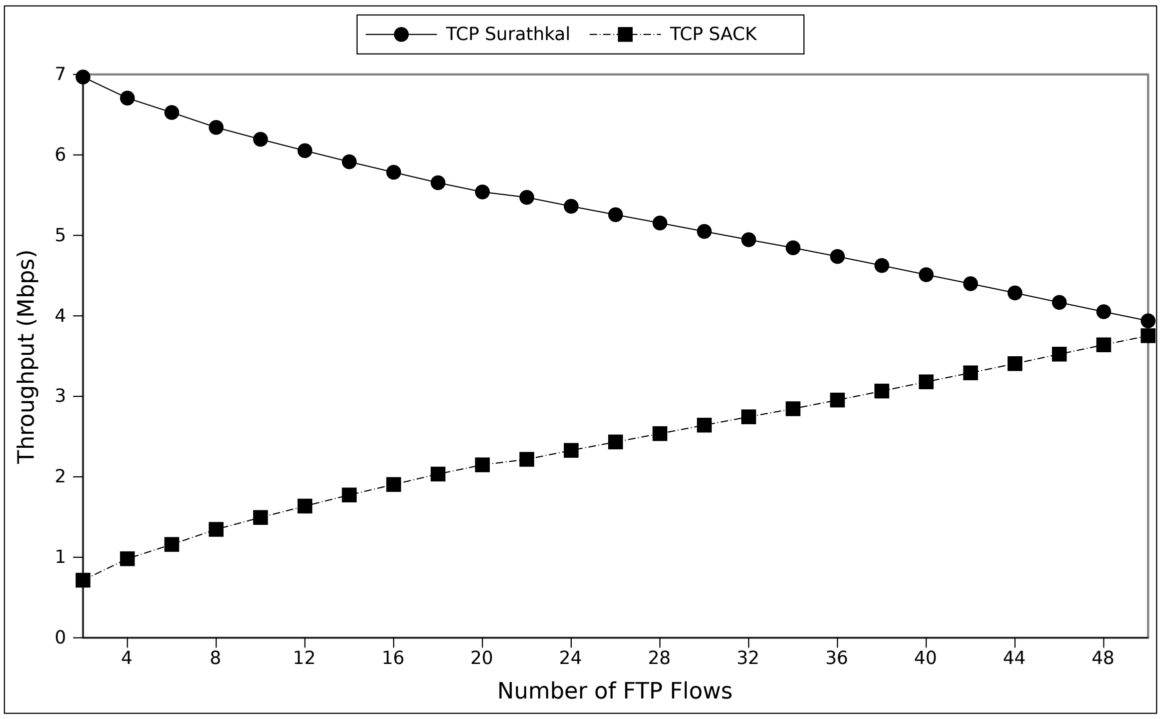


Figure 4.6: Throughput vs Number of flows

It is observed from the graph that when the number of flows are less, the average throughput of TCP Surathkal flows is significantly higher than that of the TCP SACK. It is to be noted that this is not due to TCP Surathkal's unfairness or aggreviseness, but due to the approach of decreasing *cwnd* intelligently based on the network feedback. Like TCP SACK, TCP Surathkal too probes the available bandwidth in an additive increase manner. TCP SACK flows decrease *cwnd* by half and hence leave the link under utilized. Since TCP Surathkal flows do not reduce by half, they utilize the bandwidth left over by TCP SACK flows.

As the total number of flows increase in the network, the average throughput achieved by TCP Surathkal flows is almost equal to that achieved by TCP SACK flows. When there are large number of TCP SACK flows, if one TCP SACK flow reduces its *cwnd* by half, other TCP SACK flows can claim some of this bandwidth and hence, the difference between aggregate throughput of TCP Surathkal flows and aggregate throughput of TCP SACK flows reduces. Fig. 4.7 shows the fairness ratio between TCP Surathkal and TCP SACK as the total number of flows passing through the bottleneck link increases.

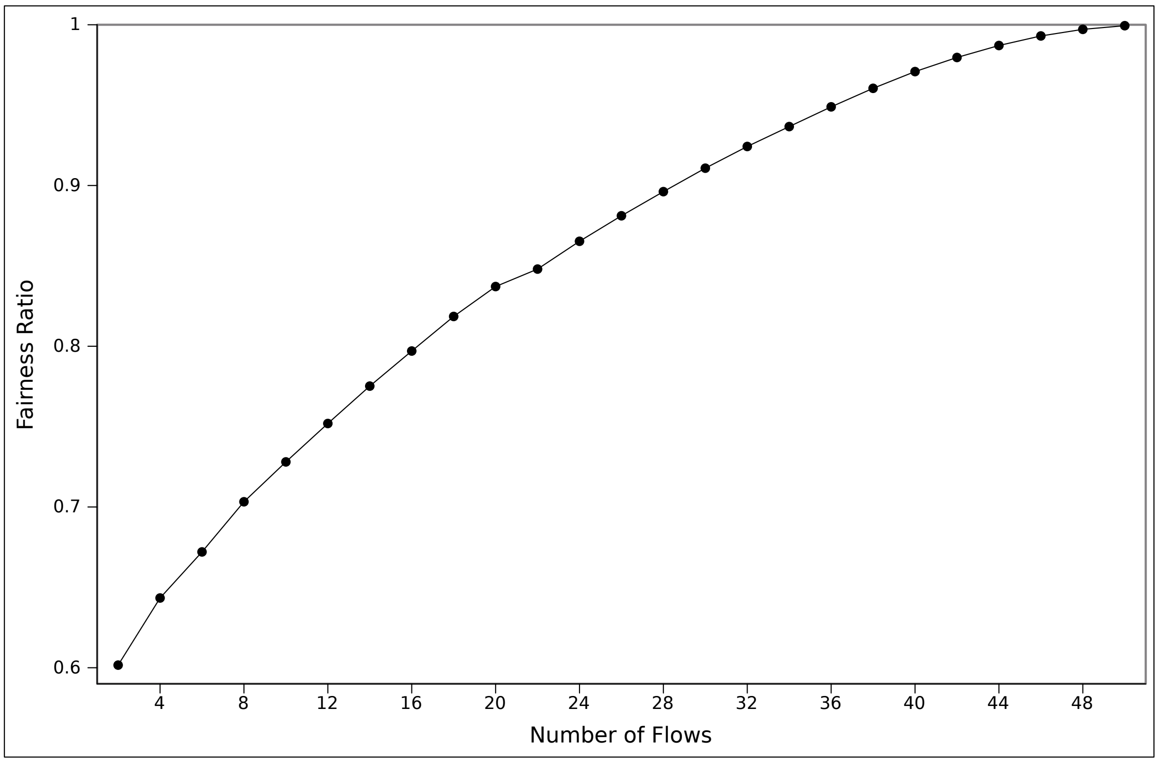


Figure 4.7: Fairness ratio

The fairness between TCP Surathkal and TCP SACK is above 90% if the total number of flows passing through the bottleneck link are 32 or more, while it is more than 95% if the total number of flows is above 40.

Stability of the *cwnd*

In this experiment, we present the benefits of TCP Surathkal in terms of stabilizing the *cwnd* as compared to TCP SACK. The bottleneck bandwidth and propagation delay are fixed to 8Mbps and 50ms respectively. The experiment is first run with TCP SACK and repeated for TCP Surathkal. Fig. 4.8 shows the results.

It can be seen from the graph that the *cwnd* of TCP SACK sender oscillates more than that of TCP Surathkal sender. Oscillations in the *cwnd* of TCP SACK are in the range of 10 packets to 100 packets whereas in case of TCP Surathkal, oscillations are in the range of 55 packets to 70 packets. Multiplicative decrease by a factor of 2 results in large oscillations in TCP SACK. These oscillations in the *cwnd* further lead to large oscillations in the queue length at routers. This causes variations in the delay

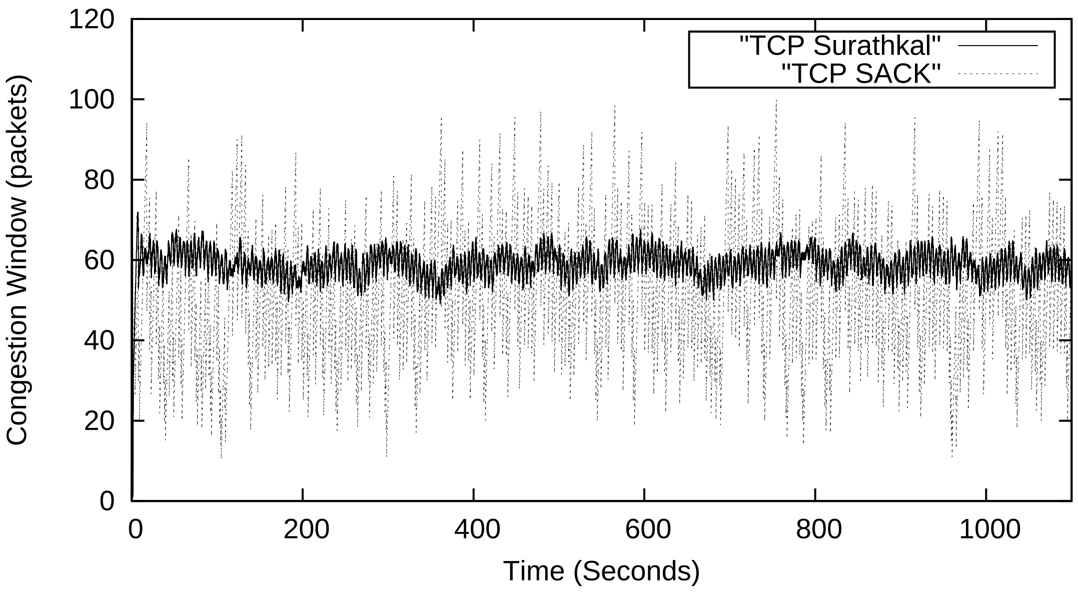


Figure 4.8: Congestion Window vs Time

and severely affects the performance of time sensitive applications e.g., multimedia applications, etc.

TCP Surathkal's adaptive decrease mechanism stabilizes the *cwnd* at the sender and consequently stabilizes the queue size at the routers (See next section for details). Thus, TCP Surathkal reduces the variations in the delay and improves the performance of time sensitive applications. In Chapter 5, we develop a fluid model for TCP Surathkal and analytically prove that TCP Surathkal significantly reduces the oscillations in the *cwnd* and the *avg*.

Stability of the *avg*

In this experiment, we measure the *avg* at routers to evaluate the performance of TCP Surathkal and TCP SACK. As described before, we configure the routers with ARED AQM mechanism. ARED addresses the parameter sensitivity of RED by automatically setting all the required parameters based on the network dynamics. ARED aims to keep the *average queue size* within a range of 40% to 60% of the minimum and maximum thresholds set for the *avg*.

Based on our network topology, ARED sets minimum threshold for *avg* to 5 packets and maximum threshold for *avg* to 15 packets. Hence in our simulations, ARED aims to keep the *avg* within *target range* which is between 9 packets and 11 packets. Fig. 4.9 shows the results.

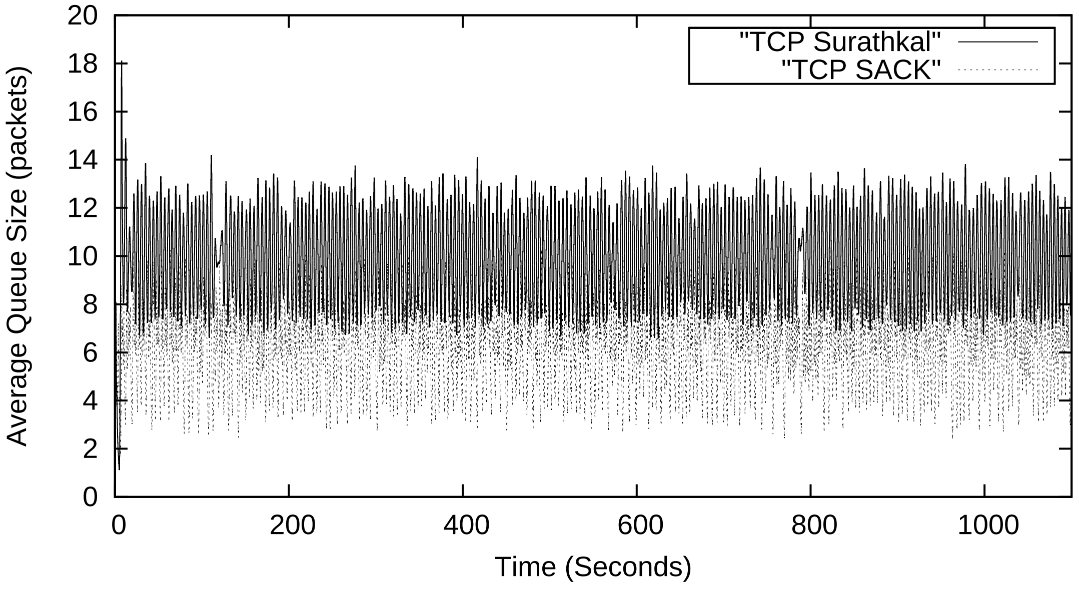


Figure 4.9: Average queue size vs Time

It can be seen from the graph that TCP Surathkal flows maintain the *avg* within the desired range whereas TCP SACK flows maintain *avg* much lower than the desired range. TCP Surathkal‘s performance is controlled by the network and hence, complements the design of ARED mechanism since the *avg* with TCP Surathkal flows is 10 packets, exactly as desired by ARED. TCP SACK under utilizes the available queue size by maintaining an *avg* of only 7 packets. Moreover, oscillations in *avg* are larger with TCP SACK flows than that of TCP Surathkal. Large oscillations create instability in the *avg* and affect the overall performance of the network.

4.4.2 Comparison with other TCP Variants

In this section, we compare the performance of TCP Surathkal with other popular TCP variants such as: SACK, High Speed TCP (HSTCP) (Floyd 2003), Scalable TCP (STCP) (Kelly 2003), CUBIC (Rhee and Xu 2005) and XCP. Moreover, we simulate all protocols with RED as the AQM mechanism to demonstrate the robustness of TCP Surathkal across different AQM mechanisms. The parameters taken into consideration are link utilization[6], mean queue length and packet drop rate.

[6]Given a certain link capacity, higher utilization values mean higher throughput.

To study the impact of various network parameters on the performance of TCP Surathkal, we vary each network parameter, one at a time, while keeping other parameters fixed. The basic network scenario consists of a dumbbell topology with bottleneck bandwidth set to 10Mbps and RTT set to 80ms. The forward traffic consists of 5 FTP flows, 5 flows carrying the interactive voice traffic (codec G.711, 64 kbps rate), 5 flows carrying the live streaming data (streaming rate is 640Kb and packet size is 840 bytes) and several short web-like flows. The reverse traffic consists of 5 FTP flows and 5 flows carrying the live streaming data. We have used a standard tool called "TCP Evaluation Suite" (Wang et al. 2007) for simulations (and the above mentioned parameters are based on the guidelines mentioned in this suite).

The reverse traffic causes congestion on the reverse path and hence, the ACKs which are traversing through that path have a high probability of getting dropped. The main idea behind simulating the reverse traffic is to demonstrate the robustness of XECN mechanism, even when the ACKs carrying CC are lost.

Impact of bottleneck bandwidth

We vary the bottleneck bandwidth in the basic scenario from 10Mbps to 1Gbps while keeping the other network parameters fixed. This covers the range of bandwidths for Wired LANs and Wired WANs as shown in Table 1.1. Fig.4.10 shows the results for link utilization.

It is observed from the graph that TCP Surathkal achieves high link utilization across a wide range of bottleneck capacities. TCP Surathkal outperforms most of the other protocols, especially HSTCP, STCP and CUBIC. However, for bandwidths $>$ 100Mbps, XCP performs better than other protocols. Note that XCP is a *network based protocol* and requires maintaining of per-flow state at the routers. Moreover, its deployment is a challenging issue since it requires additional 128 bits in the IP header. STCP, on the other hand, achieves poor utilization than other protocols.

Fig.4.11 shows the results for mean queue length. A lower mean queue length offers high burst tolerance and improves the performance of time sensitive traffic by inducing minimal queueing delay. We observe that the performance of TCP Surathkal

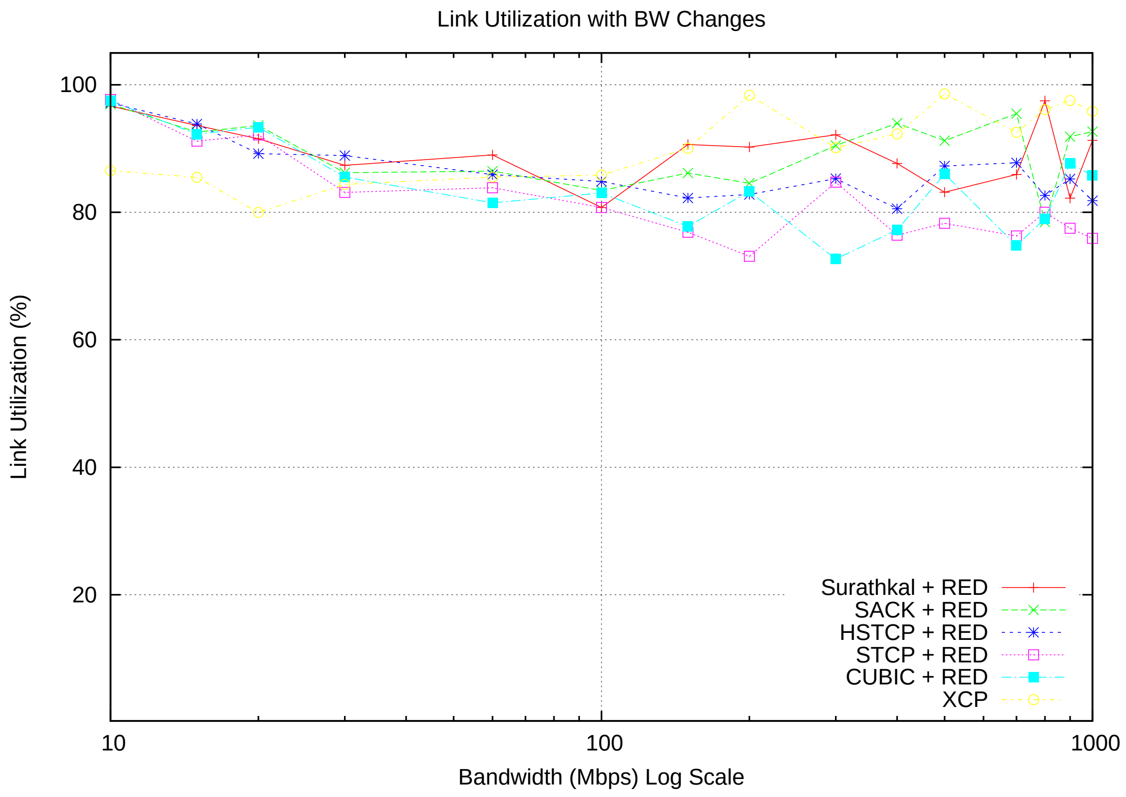


Figure 4.10: Link Utilization with Bandwidth Changes

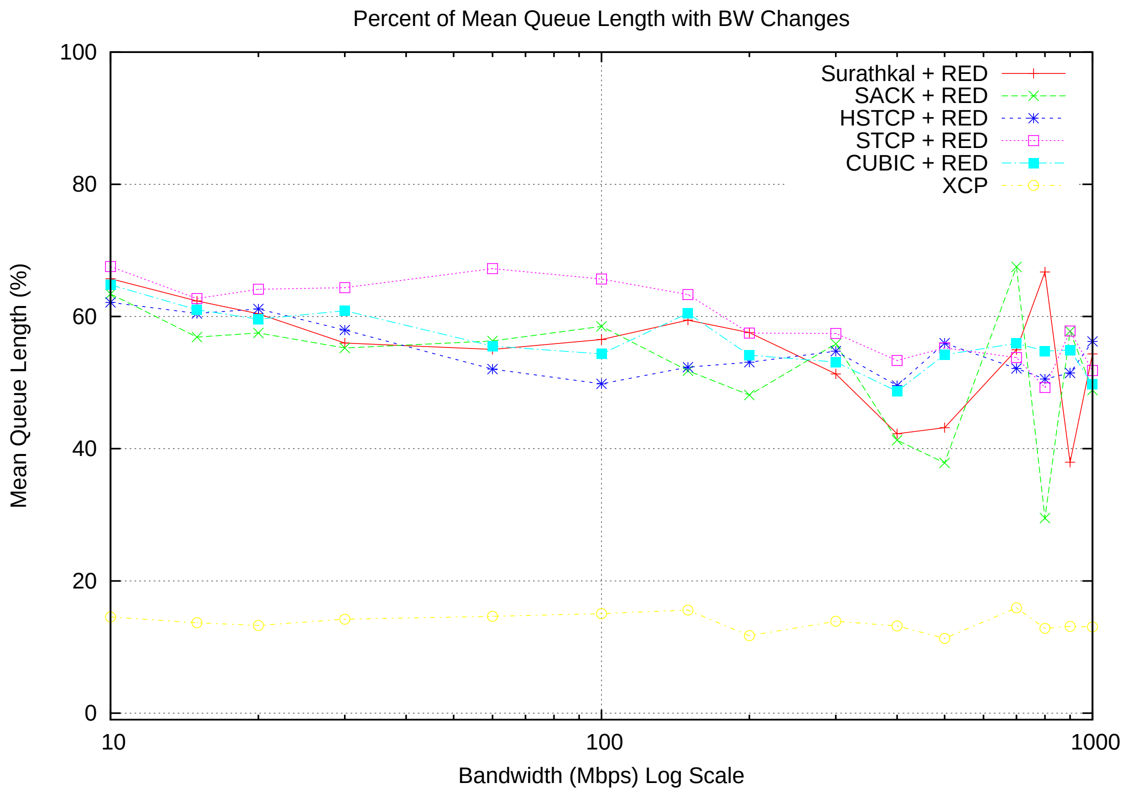


Figure 4.11: Percent of Mean Queue Length with Bandwidth Changes

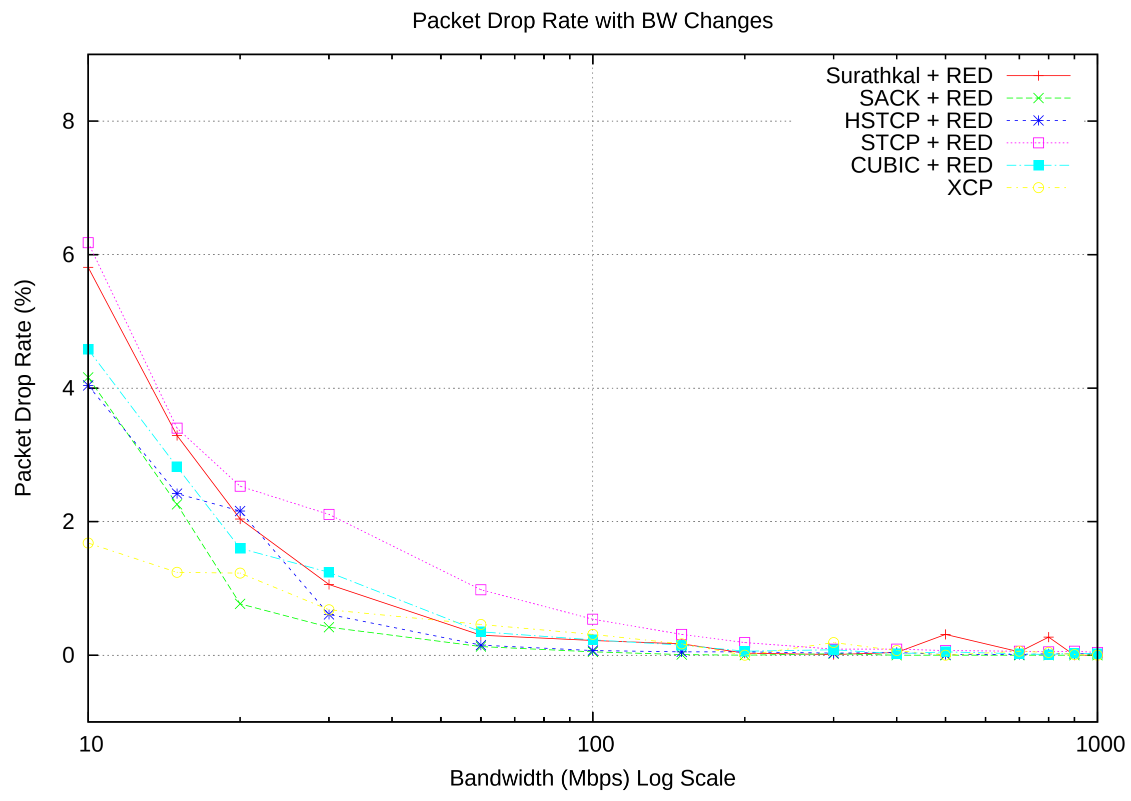


Figure 4.12: Packet Drop Rate with Bandwidth Changes

is similar to that of the other protocols. The mean queue length occupied by all protocols is almost the same, except XCP which performs exceptionally better than other protocols. On the other hand, STCP has the highest mean queue length amongst all the protocols.

Fig.4.12 shows the results for packet drop rate. Packet drops are highly undesired since they lead to re-transmissions and waste the network bandwidth. It is observed that XCP has the least packet drop rate and STCP has the highest packet drop rate. The packet drop rate of TCP Surathkal is similar to other TCP variants, except in a few scenarios where the bottleneck bandwidth is $<$ 100Mbps. When the bandwidth is less and the reverse traffic is bursty, a lot of ACKs which carry CC are dropped. This phenomenon leads to inaccuracy in estimating the *severity of congestion* and increases the convergence time of TCP Surathkal. This causes slightly more packet drop rate. When more bandwidth is provisioned, the packet drop rate of TCP Surathkal decreases sharply and becomes equal to that of the other protocols.

It is to be noted that TCP SACK performs equally well as TCP Surathkal with

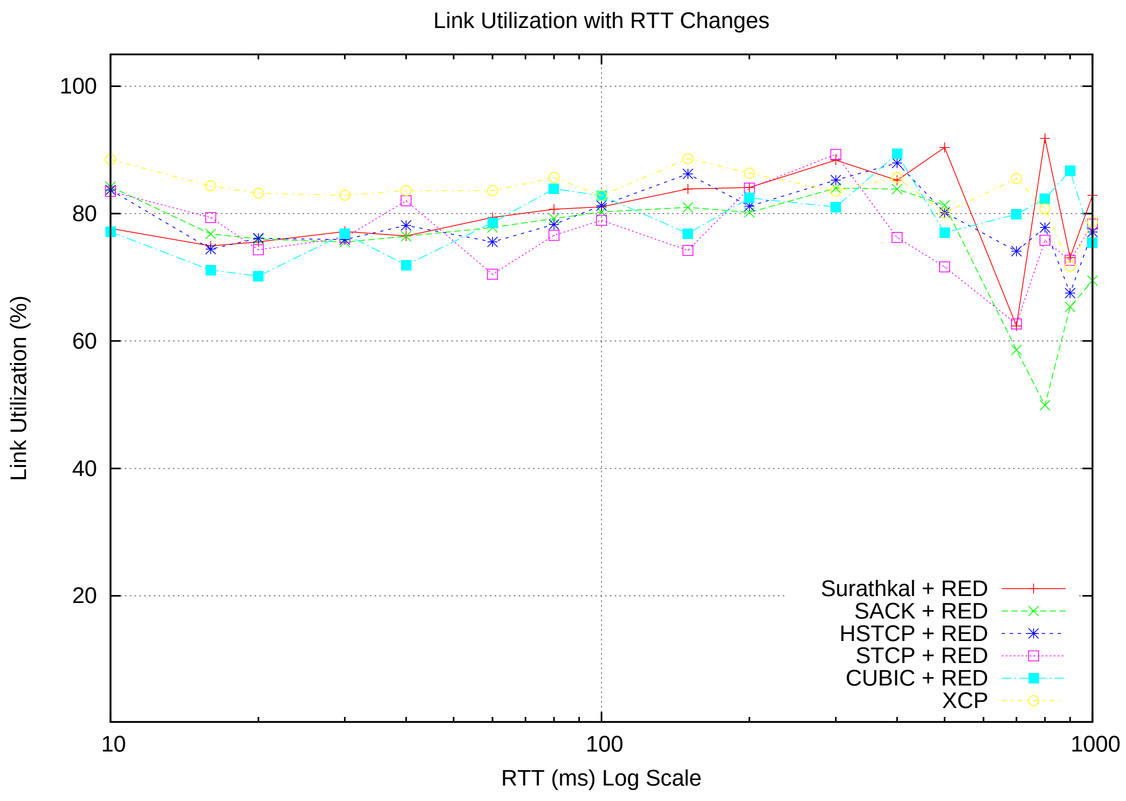


Figure 4.13: Link Utilization with RTT Changes

respect to all the parameters. However, the next section highlights the limitations of TCP SACK when RTT is varied.

Impact of round trip delay

In this section, we vary the round trip delay in the basic scenario from 10ms to 1000ms while keeping the other parameters fixed. This covers the range of latencies for Wired WANs, 802.11 WLAN/Mesh networks, cellular data networks and satellite networks. Fig 4.13 shows the results for link utilization.

It is observed from the graph that TCP Surathkal performs equally well as compared to the other protocols. TCP Surathkal, infact, achieves more utilization than other protocols when the RTT is $>$ 100ms. The average latency in Internet is around 150ms to 250ms (Paxson et al. 2011). The performance of other protocols, especially SACK and STCP, degrades sharply with the increase in RTT. Though XCP performs consistently better than other protocols, TCP Surathkal outperforms XCP in a few scenarios where the RTT is slightly $<$ 1000ms.

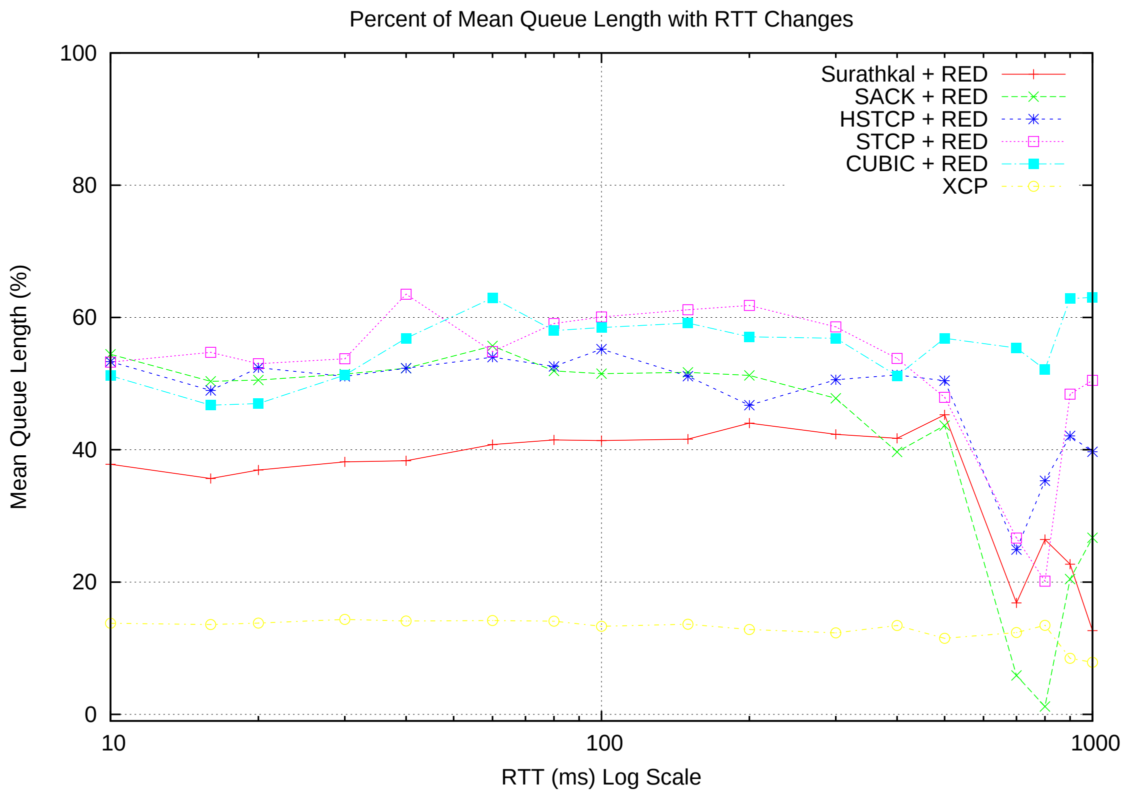


Figure 4.14: Percent of Mean Queue Length with RTT Changes

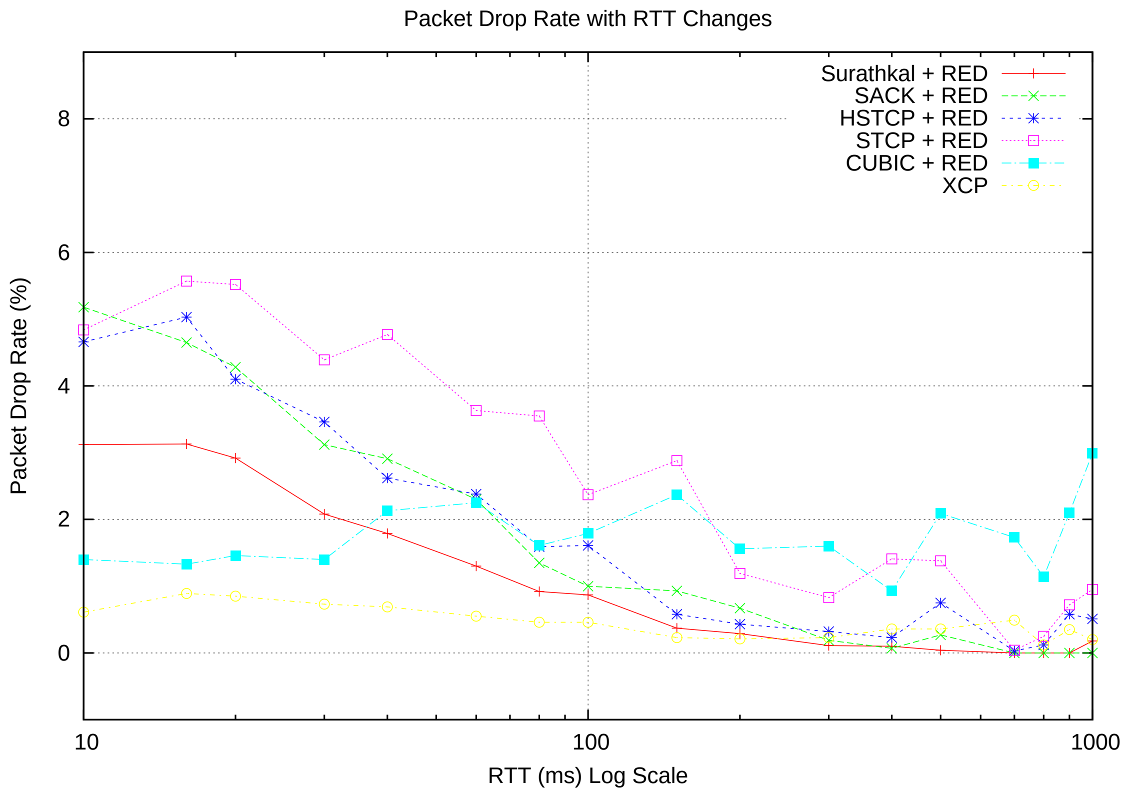


Figure 4.15: Packet Drop Rate with RTT Changes

Fig.4.14 shows the results for mean queue length when RTT is varied. It is observed that TCP Surathkal performs significantly better than other TCP variants by occupying the least mean queue length. XCP retains its exceptional performance and is the only protocol that outperforms TCP Surathkal. CUBIC and STCP have the highest mean queue length among all the protocols.

Fig.4.15 shows the results for packet drop rate when RTT is varied. TCP Surathkal consistently outperforms HSTCP, STCP and TCP SACK in all the scenarios. The packet drop rate of CUBIC increass as the RTT increases, whereas, the packet drop rate of TCP Surathkal decreases as the RTT increases. As a result, TCP Surathkal outperforms CUBIC when RTT $>$ 50ms. The average latency in the Internet being around 150ms to 250ms, we infer that TCP Surathkal achieves the best performance since it outperforms even XCP when RTT $>$ 100ms.

4.4.3 Performance evaluation of TCP Surathkal in Wireless networks

Due to the widespread deployment of Wireless Fidelity (Wi-Fi) hotspots, it has become extremely important to improve the performance of TCP in wireless networks. Wireless networks are characterized by high channel error rates as shown in Table 1.1. TCP's inability to distinguish congestion losses from non-congestion losses cause severe degradation in the throughput. In this section, we evaluate the performance of TCP Surathkal with TCP SACK and TCP Westwood (Casetti et al. 2002). While TCP SACK is the most widely used protocol, the rationale behind selecting TCP Westwood for the performance evaluation is that it is widely implemented in the Linux kernel. Moreover, it is considered highly effective in wireless networks with lossy links, especially in mixed wired and wireless networks (e.g., Wi-Fi).

The wireless network topology used for simulations is shown in Fig. 4.16. The default values for the wired link bandwidth and delay are fixed at 75Mbps and 5ms respectively. The wireless link bandwidth is kept fixed at 11Mpbs, unless otherwise specified. Link layer retransmissions are not enabled so as to demonstrate the benefits of TCP Surathkal without leveraging the advantages of lower layer functionalities. ARED is used as an AQM mechanism in routers. ECN is used as the signaling mechanism for other protocols, whereas, XECN is used for TCP Surathkal. FTP sources start sending the data at time 50 seconds. All simulations are run for 1150 seconds. The receiver advertises a large window such that the sending rate is not limited by the receiver.

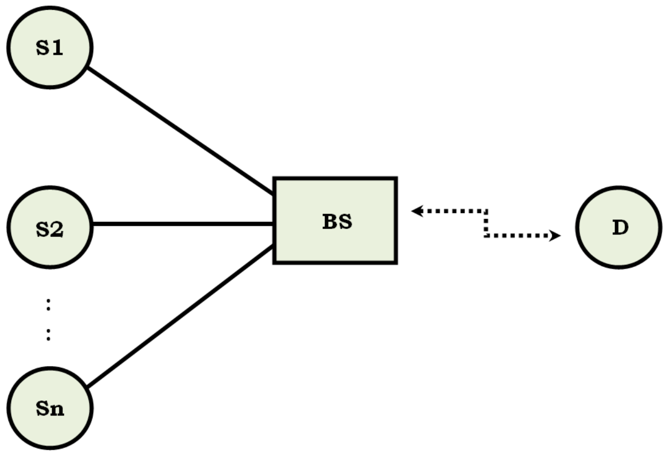


Figure 4.16: Single hop wireless topology

Impact of bottleneck capacity

In this experiment we show the effect of halving the *cwnd* on the performance of TCP SACK in wireless networks, as the bottleneck bandwidth is increased, and the corresponding improvement in the performance in case of TCP Surathkal. There is no error in the channel and hence all the packet losses occur only due to congestion. The wireless bandwidth is varied from 5Mbps to 12Mbps. The topology consists of two flows. Fig. 4.17 and Fig.4.18 show the performance comparison of TCP Surathkal with TCP SACK and TCP Westwood respectively.

It can be seen from Fig.4.17 that TCP Surathkal consistently outperforms TCP SACK while the wireless bandwidth is varied. There is an improvement of upto 4.02% in the throughput of TCP Surathkal as compared to TCP SACK. RTT variation in wireless networks is extremely high due to the interference effects and the time spent in waiting for contention. As a result, the sending rate of TCP is largely affected since updation of *cwnd* occurs in every RTT. Thus, decreasing *cwnd* blindly by half in wireless networks results in degraded throughput for TCP SACK. TCP Surathkal efficiently utilizes the bandwidth since it decreases *cwnd* based on the

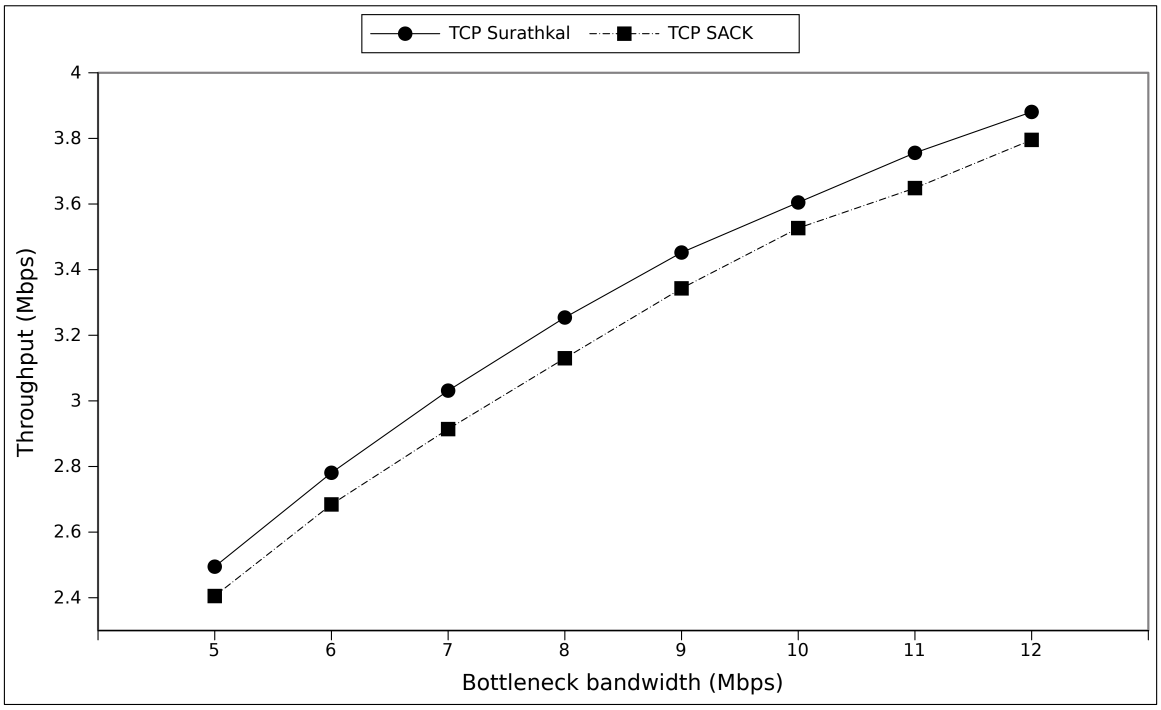


Figure 4.17: Throughput vs Wireless bandwidth

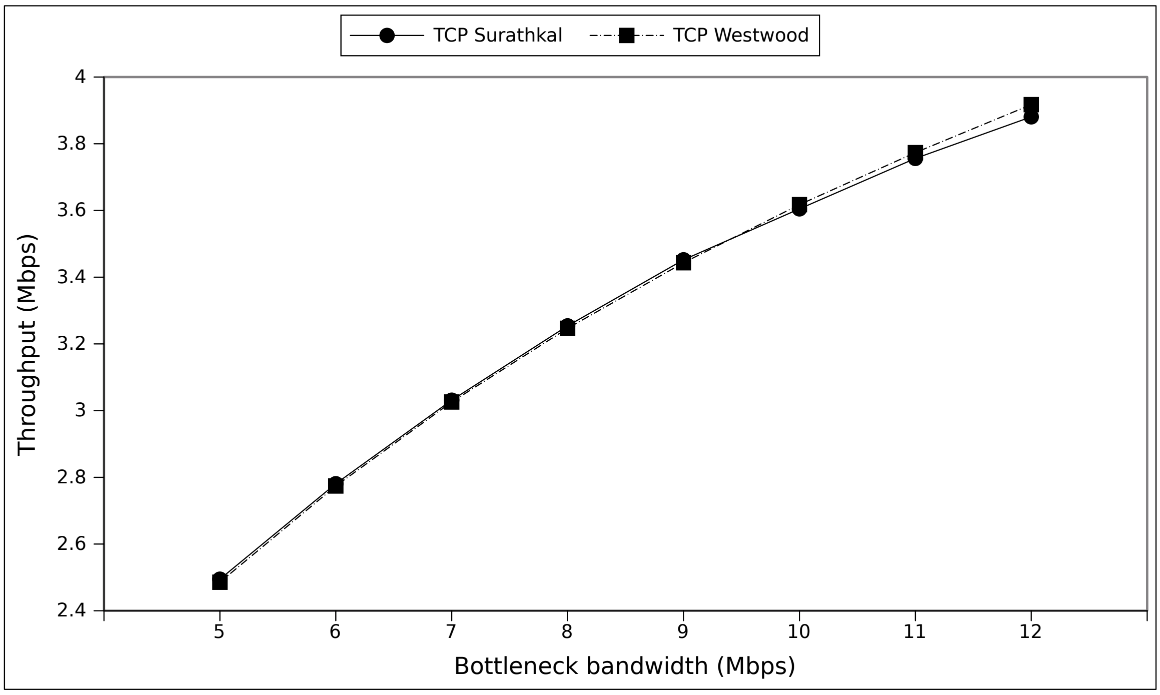


Figure 4.18: Throughput vs Wireless bandwidth

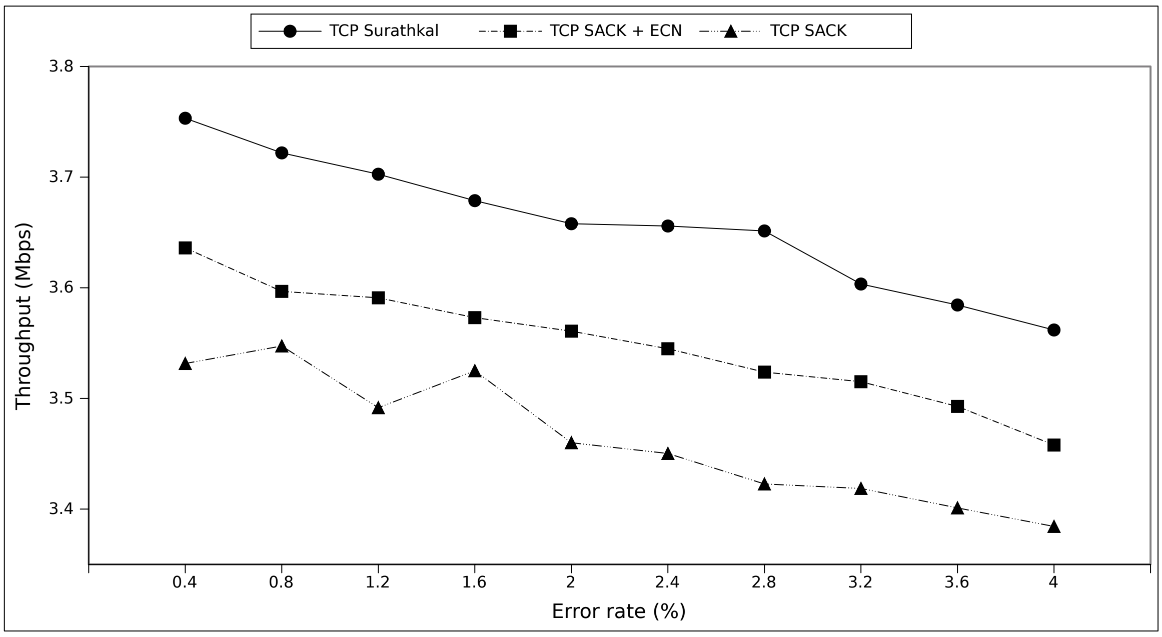


Figure 4.19: Varying error rate

network feedback about the *severity of congestion*.

On the other hand, Fig.4.18 shows that the throughput achieved by TCP Surathkal and TCP Westwood is exactly the same. This is because unlike TCP SACK, TCP Westwood does not reduce the *cwnd* by half. Instead, the main idea of TCP Westwood is to use the "bandwidth estimate" to vary the *cwnd*. The bandwidth is estimated by monitoring the returning ACKs (Casetti et al. 2002). Since both TCP Surathkal and TCP Westwood vary the *cwnd* based on the network conditions, they achieve similar throughput and effectively use the available bandwidth.

Impact of channel errors

In this experiment, we present the results for the simulation showing the performance improvement offered by TCP Surathkal at various channel error rates. The workload consists of two flows. Packet losses in this scenario may be due to channel errors, congestion or both. Fig. 4.19 shows the improvement in the performance of TCP Surathkal when compared to TCP SACK. To demonstrate the advantages of using ECN in the wireless networks, we consider two types of configuration for TCP SACK: (i) TCP SACK with ECN and (ii) TCP SACK without ECN.

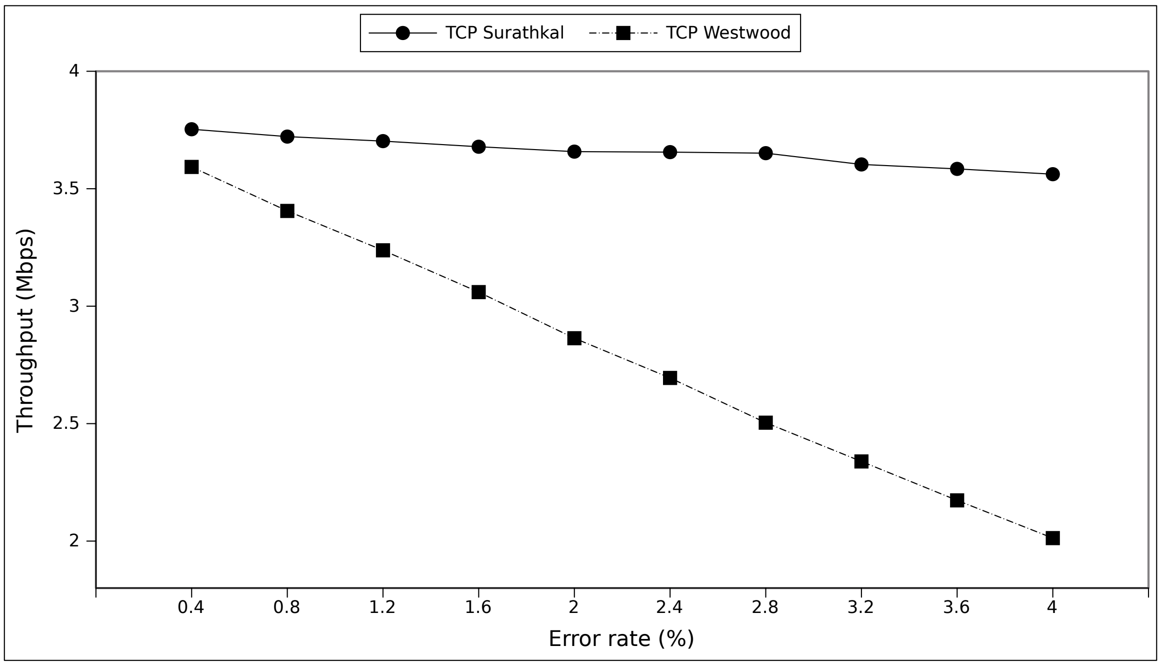


Figure 4.20: Varying error rate

It is observed that the use of ECN leads to a considerable increase in the throughput of TCP SACK. TCP Surathkal, however, outperforms both the configurations of TCP SACK and achieves maximum throughput. TCP Surathkal achieves upto 3.62% more throughput than TCP SACK+ECN and upto 6.68% more throughput than TCP SACK without ECN. It has been shown in (Tickoo et al. 2005) that maintaining sufficient *cwnd* granularity helps to improve the performance of TCP in wireless networks when the channel error rates are high. Since TCP Surathkal does not reduce *cwnd* by half like TCP SACK, its *cwnd* granularity remains higher than that of TCP SACK. Moreover, TCP SACK considers packet loss due to error as an indication of congestion and wrongly decreases the *cwnd.* TCP Surathkal relies on XECN mechanism to provide congestion notification and does not react to packet losses due to the channel error. As a result, TCP Surathkal performs consistently better than TCP SACK, with moderate channel error rates upto 4%. Although several research studies rely on link layer mechanisms to improve the performance of TCP in wireless networks, it is to be noted that the improvement in TCP Surathkal is independent of such link layer mechanisms. Combining TCP Surathkal with such link layer mechanisms may further improve the overall performance of the wireless networks. However, we are looking into this as a part of our future work.

Fig.4.20 shows that the throughput of TCP Westwood degrades drastically when the channel error rates are increased. High channel error rates lead to ACKs being dropped and hence, cause inaccuracies in the bandwidth estimation policy of TCP Westwood. On the other hand, TCP Surathkal achieves significantly better throughput than TCP Westwood. TCP Surathkal relies on an explicit feedback in the form of XECN to estimate the *severity of congestion* in the network. XECN is carefully designed to provide an additional information about congestion to the sender, while retaining the reliability of the congestion notification. Results show that TCP Surathkal achieves an improvement of upto 76.96% in throughput when compared to TCP Westwood.

4.4.4 Inference

We have demonstrated the efficacy of TCP Surathkal in a wide range of environments by comparing its performance with other popular transport protocols. Based on the results obtained in these scenarios, we infer the following:

- TCP Surathkal efficiently utilizes the available link capacity. The performance of TCP Surathkal is robust in terms of link utilization across (i) different bottleneck capacities, (ii) varying RTT and (iii) different traffic types, varying from only FTP flows to richer traffic mix including interactive voice traffic, live streaming, reverse traffic, web-traffic, etc.

- The mean queue length with TCP Surathkal is either same or less than that with the other TCP variants. Thus, TCP Surathkal does not increase the latency for delay sensitive traffic and offers high burst tolerance.

- Packet drop rate of TCP Surathkal, except in a few scenarios, is considerably less than the other protocols. The performance of TCP Surathkal is significantly better in terms of packet drop rate when the RTT is in the range of 100ms and 1000ms. This is the range for an average RTT in the Internet.

- TCP Surathkal is fair to other competing flows in the network, especially TCP SACK which is the most widely used TCP in the Internet.

- TCP Surathkal improves the overall performance of the wireless networks. Note that this improvement is achieved without using any link layer mechanism to differentiate congestion packet losses from non-congestion losses. TCP Surathkal can be easily merged with other TCP variants designed specifically for wireless networks, to further provide a robust performance.

- The above mentioned advantages of TCP Surathkal make it suitable for use across a variety of networks such as Wired WANs, 802.11 WLAN/Mesh networks and satellite networks. Moreover, since TCP Surathkal achieves high link utilization and maintains low queue length, it meets the requirements of various applications such as interactive applications, large and medium data transfers.

- The deployment complexity of TCP Surathkal is low since it requires XECN to be implemented only at the sender and the receiver. Moreover, it does not need any additional bit(s) in the TCP or IP header and makes no modifications in the working of the routers.

Chapter 5

Fluid model for TCP Surathkal

A variety of mathematical models such as the periodic model (Mathis et al. 1997), detailed packet loss model (Padhye et al. 2000), stochastic model with general loss process (Altman et al. 2000), fluid model (Misra et al. 2000), etc have been developed for modeling TCP. While some of these models focus on a particular module of TCP, others add a new level of generality to the process of TCP modeling (Hassan and Jain 2004). In this dissertation, we explore the fluid model to study the behavior of TCP Surathkal because this model is suitable for analyzing a network that consists of routers with AQM mechanisms and TCP flows.

5.1 Genesis of the Fluid model

(Misra et al. 2000) presents a fluid model based on Poisson Counter driven Stochastic Differential Equations to model the interactions of a set of TCP flows and routers with AQM mechanisms. This model is appropriate to analyze the behavior of TCP Surathkal because the presence of AQM mechanisms at routers is the fundamental requirement of TCP Surathkal.

The model is based on the assumption that packet losses to flow i are described by a Poisson process $N_i(t)$ with time varying rate $\lambda_i(t)$. $N_i(t)$ denotes the number of packet losses suffered by flow i and t denotes the time when the TCP sender *detects* the losses. Note that t is different from the time when the actual packet is dropped at the router. The time varying nature of $\lambda_i(t)$ models the independent packet dropping mechanisms generally found in AQMs (Misra et al. 2000).

The data traffic is modeled as a fluid and a set of differential equations that describe the AQM mechanism and the router queueing process are derived. This model takes into consideration the entire system in which the sending rate of TCP is closely coupled with the packet losses. Thus, the system is a closed loop control system resulting in a set of coupled differential equations (Misra et al. 2000).

Assuming that N long flows are traversing a single bottleneck router with transmission capacity C, the fluid model for TCP is given by Eq.(5.1.1), Eq.(5.1.2) and Eq.(5.1.3):

$$\frac{dW_i}{dt} = \frac{1}{R_i(q)} - (\frac{W_i}{2})\frac{W_i(t-\tau)}{R_i(q(t-\tau))}p(x(t-\tau)) \tag{5.1.1}$$

$$\frac{dx}{dt} = \frac{log_e(1-w_q)}{\delta}x(t) - \frac{log_e(1-w_q)}{\delta}q(t) \tag{5.1.2}$$

$$\frac{dq(t)}{dt} \approx -C + \sum_{i=1}^{N}\frac{W_i}{R_i(q)} \tag{5.1.3}$$

where $W_i(t)$ and $R_i(t)$ denote the *cwnd* and RTT at time t of flow i (1 $\leq$ i $\leq$ N) respectively. $q(t)$ denotes the instantaneous queue length. τ represents the round trip delay for a loss notification to reach the sender. C represents the capacity of the bottleneck router in packets. x represents the *avg* at the router. *p(x)* represents the packet drop probability as a function of x. w_q represents the smoothing constant for exponential weighted moving average x. δ represents the sampling interval.

$R_i(t)$ is given by

$$R_i(t) = \beta_i + \frac{q(t)}{C} \tag{5.1.4}$$

where β_i is the fixed propagation delay and $\frac{q(t)}{C}$ models the queueing delay.

Eq.(5.1.1) models the AIMD behavior of TCP. The first term on Right Hand Size (RHS) corresponds to the additive increase part, which shows that *cwnd* increases by one in every RTT. The second term on RHS corresponds to the multiplicative decrease part, which shows that *cwnd* decreases by half at the instant of the arrival of a loss notification i.e., when

$$\frac{W_i(t-\tau)}{R_i(q(t-\tau))}p(x(t-\tau)) = 1 \tag{5.1.5}$$

Eq.(5.1.2) provides an estimate of the average queue length (*avg*) at the router based on the samples taken every δ seconds. Note that in RED, samples are taken on arrival of every packet rather than a fixed value δ. Hence, the value of δ in the fluid model is carefully chosen to be 1/C. The rationale behind selecting 1/C is explained in detail in (Misra et al. 2000).

Eq.(5.1.3) describes the behavior of the instantaneous queue length at the router. The first term on Right Hand Size (RHS) corresponds to the decrease in the instantaneous queue length due to the servicing of the packets. The second term on RHS corresponds to the increase in the instantaneous queue length due to the arrival of packets from the TCP flows.

Thus, based on these three equations, we can get the estimate of the *cwnd* evolution, the *avg* and the instantaneous queue length. A detailed derivation of these equations is provided in (Misra et al. 2000).

5.2 Modified Fluid model for TCP Surathkal

In this section, we develop a fluid model for TCP Surathkal which is a minor modification of the fluid model discussed in the previous section.

The only difference between TCP and TCP Surathkal is that TCP decreases *cwnd* by half in the event of congestion, whereas TCP Surathkal decreases *cwnd* by an adaptive parameter, α (see Eq.(4.3.2)). Moreover, recall that TCP Surathkal does not require any modifications in the working of the routers. Based on this aspects, the modified fluid model for TCP Surathkal is given by Eq.(5.2.1), Eq.(5.2.2) and Eq.(5.2.3).

$$\frac{dW_i}{dt} = \frac{1}{R_i(q)} - (\frac{W_i\alpha_i}{2})\frac{W_i(t-\tau)}{R_i(q(t-\tau))}p(x(t-\tau)) \tag{5.2.1}$$

where α_i represents the fraction of packets marked in the last RTT of flow i.

$$\frac{dx}{dt} = \frac{log_e(1 - w_q)}{\delta} x(t) - \frac{log_e(1 - w_q)}{\delta} q(t) \tag{5.2.2}$$

$$\frac{dq(t)}{dt} \approx -C + \sum_{i=1}^{N} \frac{W_i}{R_i(q)} \tag{5.2.3}$$

Since TCP Surathkal does not require any modifications at the router, equations (5.2.2) and (5.2.3) are same as that of the TCP i.e. equations (5.1.2) and (5.1.3).

Moreover, as shown in Eq.(5.2.1), the additive increase part of TCP Surathkal (first term of RHS) too is similar to that of TCP. The only change, however, lies in the multiplicative decrease part of Eq.(5.2.1). Unlike TCP which decreases the *cwnd* as $W_i/2$, TCP Surathkal decreases the *cwnd* as $W_i \ \alpha_i/2$.

The next section presents a comparative analysis of TCP and TCP Surathkal with respect to the stability of the *cwnd* and the stability of the *avg*.

5.3 Results

We have compared the performance of TCP and TCP Surathkal by implementing both the fluid models in Matlab. A standard dumbbell topology is used for the experiments. The total number of flows in the network is fixed to 40. Bottleneck bandwidth is set to 5Mbps. Propagation delay for all flows is set to 200 ms. The buffer at the bottleneck router is sized so that all the packet losses are only because of RED. Minimum threshold (min_{th}) for the *avg* at the router is fixed to 150 packets. Maximum threshold (max_{th}) for the same is fixed to 200 packets. The exponential weighted moving average constant (w_q) is fixed to 0.0001 and maximum drop probability (max_p) is fixed to 0.1. Sampling interval δ is set to 1/C. The parameters used in the analysis are similar to the ones used in (Misra et al. 2000).

The analysis of TCP and TCP Surathkal is carried out by designing two different scenarios: Scenario 1 consists of 40 persistent long flows whereas Scenario 2 consists of 20 ON-OFF long flows and 20 persistent long flows.

Scenario 1

40 TCP flows are started at time, t = 0 seconds and stopped at time, t = 70 seconds. The same experiment is repeated for TCP Surathkal. Fig.5.1a through Fig.5.3b show the results obtained for this scenario.

It is observed from the results that the *cwnd* of TCP Surathkal sender stabilizes much quicker than that of the TCP sender. The results obtained in Fig.5.1a-5.1b are similar to the ones in Fig.4.6 obtained from *ns*-2 simulations. Since the *cwnd* stabilizes quickly with TCP Surathkal, so does the *avg* and instantaneous queue length. This leads to better performance for time sensitive traffic because the variations in the delay are reduced.

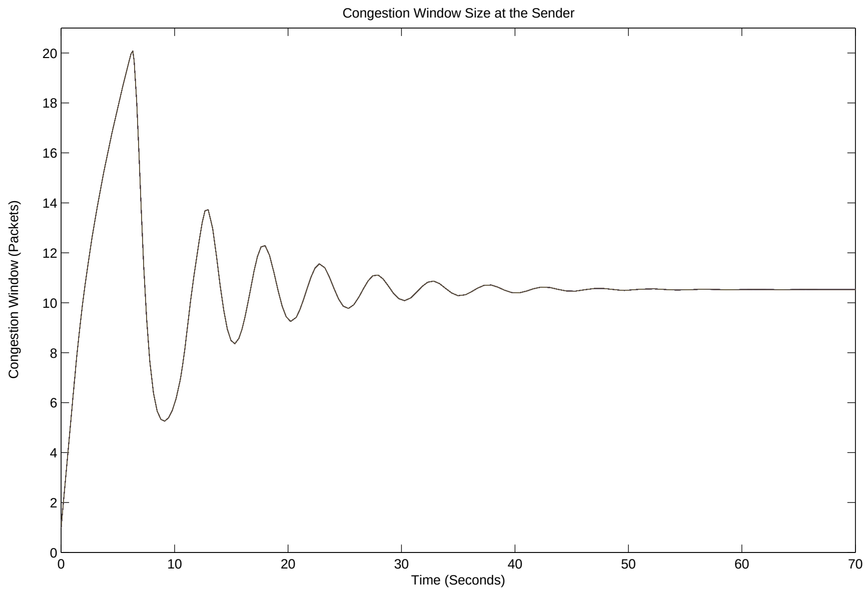
Scenario 2

40 TCP flows are started at time, t = 0 seconds. At time, t = 75 seconds, 20 TCP flows are stopped and they are re-started at time, t = 300 seconds. All 40 TCP flows are then stopped at time, t = 380 seconds. The same experiment is repeated for TCP Surathkal. Fig.5.4a through Fig.5.7b show the results obtained for this scenario.

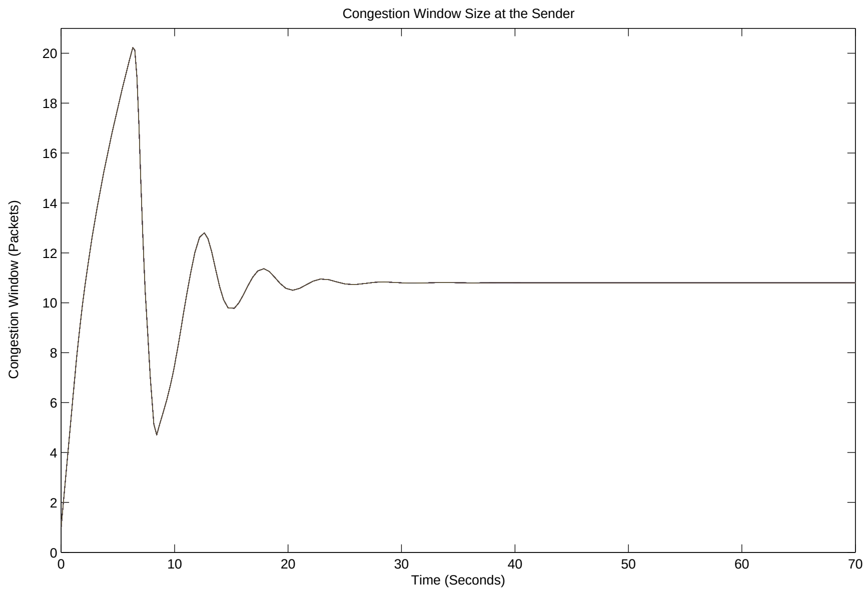
Oscillations are considered to be harmful for the network as they result in unacceptably large queue lengths and hence, incur a large variability in delays for the flows going through. Even if the mean delay turns out to be the same, these oscillations add considerable jitter to the delays. If the buffer is not large enough, then the effect of the oscillations causes buffer overflows. They also cause periodically high loss rates and affect the throughput adversely (Misra et al. 2000).

It is observed from the results that the *cwnd*, the *avg* and the instantaneous queue length exhibit large oscillations with TCP when 20 ON-OFF TCP flows are turned off. Moreover, the system does not approach a stable operating point unless 20 ON-OFF TCP flows are restarted. The traffic in the Internet is highly dynamic with large number of flows entering the network or leaving the network at the same time. It is highly desirable to bring the system to a stable operating point to reduce the impact of oscillations on the performance of time sensitive traffic.

TCP Surathkal, on the other hand, minimizes the oscillations in the *cwnd*, the

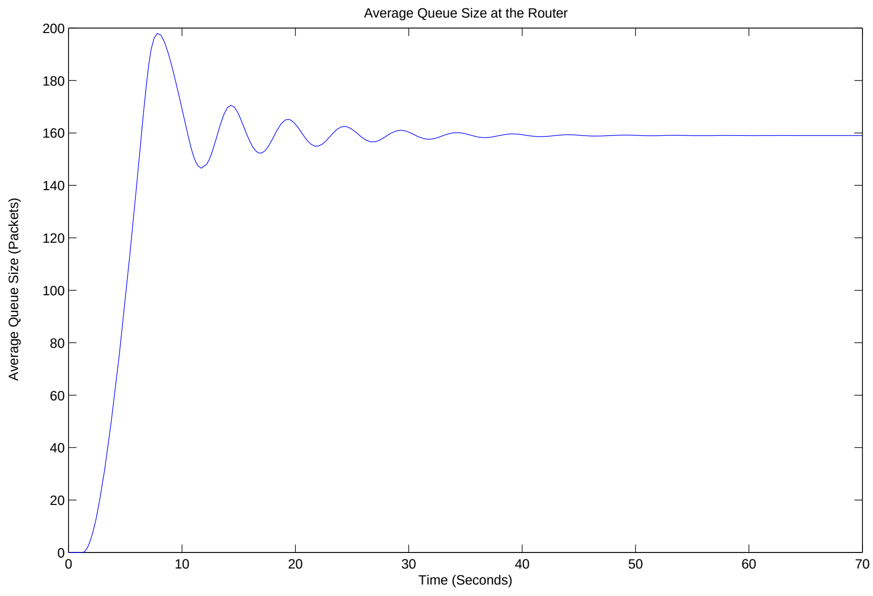


(a) Average *cwnd* of 40 TCP senders

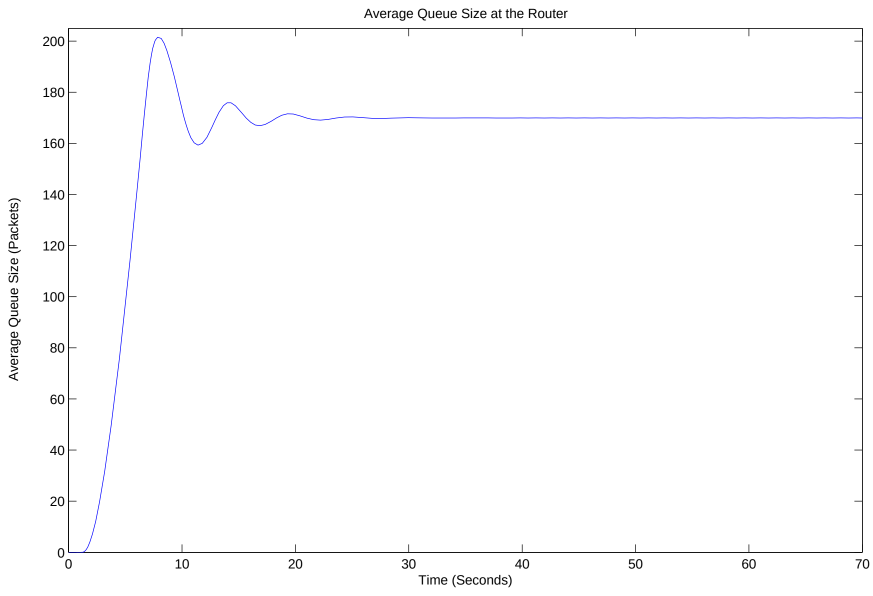


(b) Average *cwnd* of 40 TCP Surathkal senders

Figure 5.1: Oscillations in the *cwnd* with TCP and TCP Surathkal

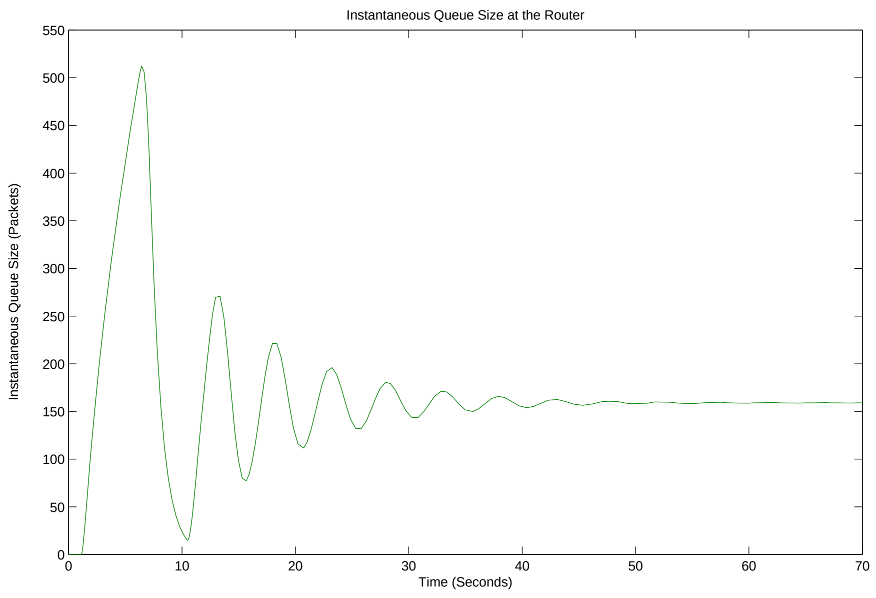


(a) *avg* with 40 TCP senders

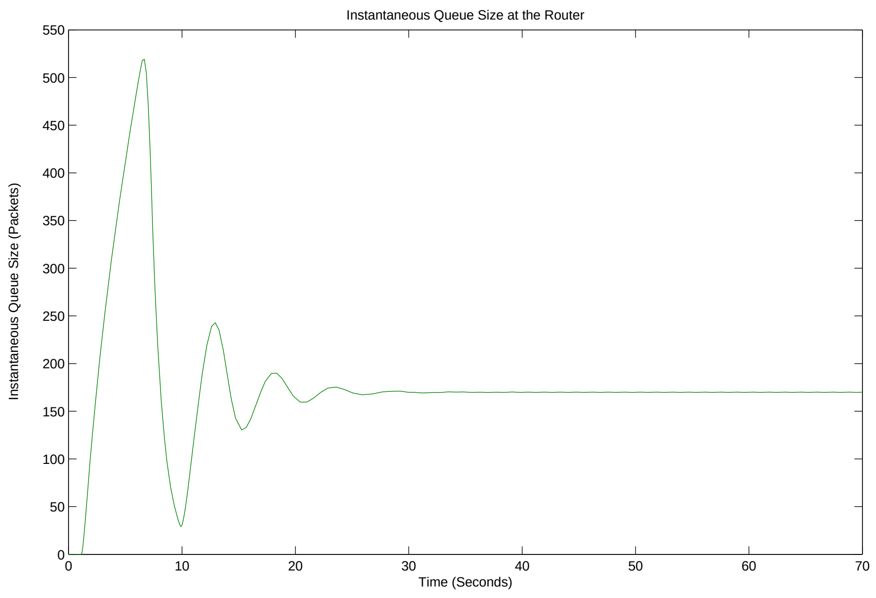


(b) *avg* with 40 TCP Surathkal senders

Figure 5.2: Oscillations in the *avg* with TCP and TCP Surathkal

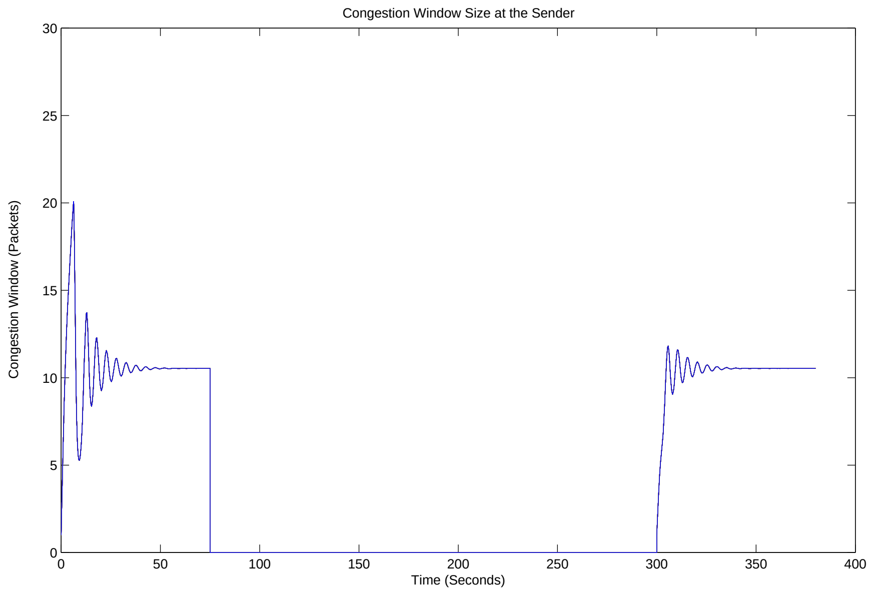


(a) Instantaneous queue size with 40 TCP senders

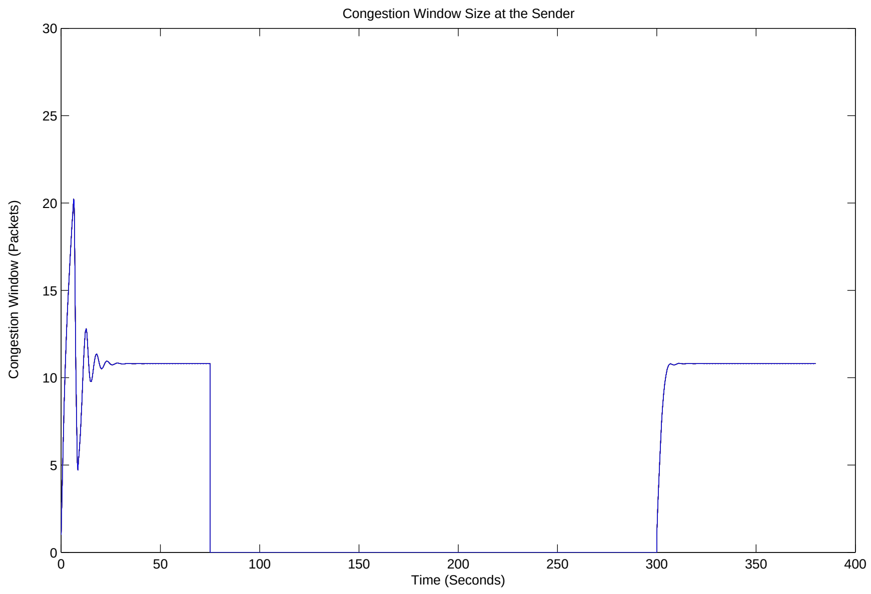


(b) Instantaneous queue size with 40 TCP Surathkal senders

Figure 5.3: Oscillations in queue length with TCP and TCP Surathkal

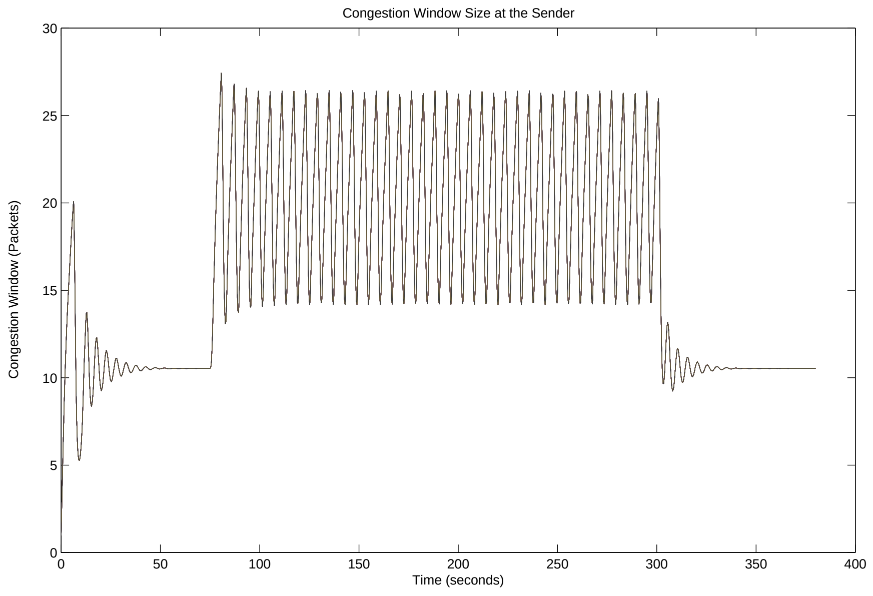


(a) Average *cwnd* of 20 ON-OFF TCP senders

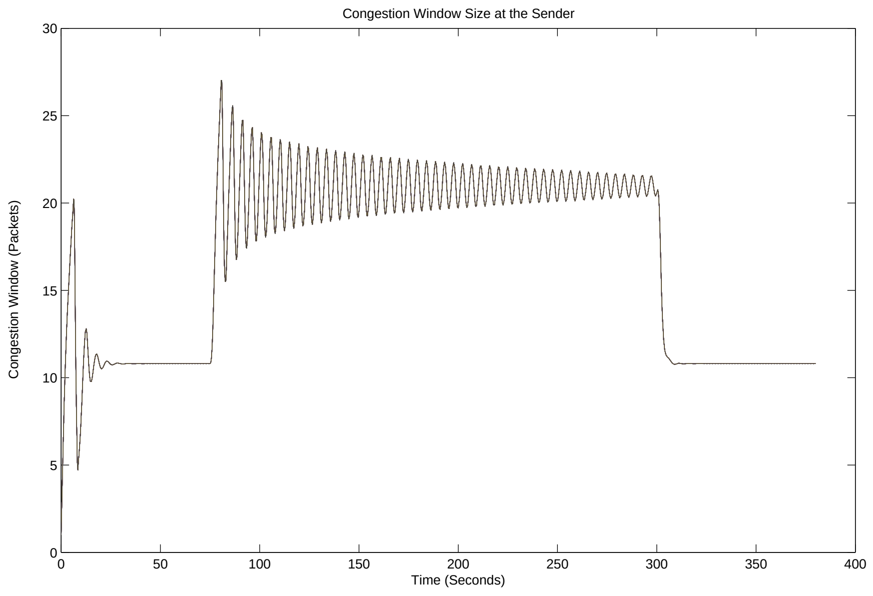


(b) Average *cwnd* of 20 ON-OFF TCP Surathkal senders

Figure 5.4: Oscillations in the *cwnd* with TCP and TCP Surathkal

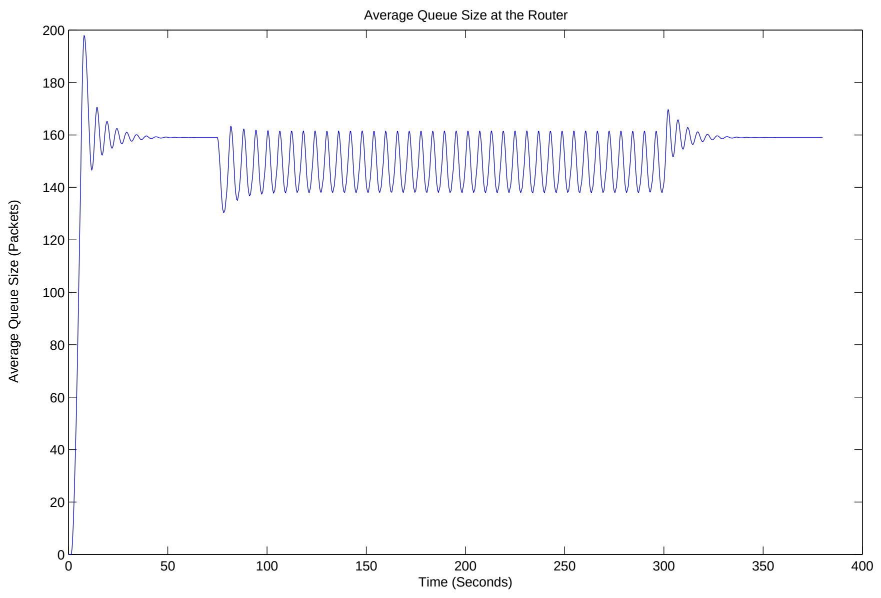


(a) Average *cwnd* of 20 persistent TCP senders

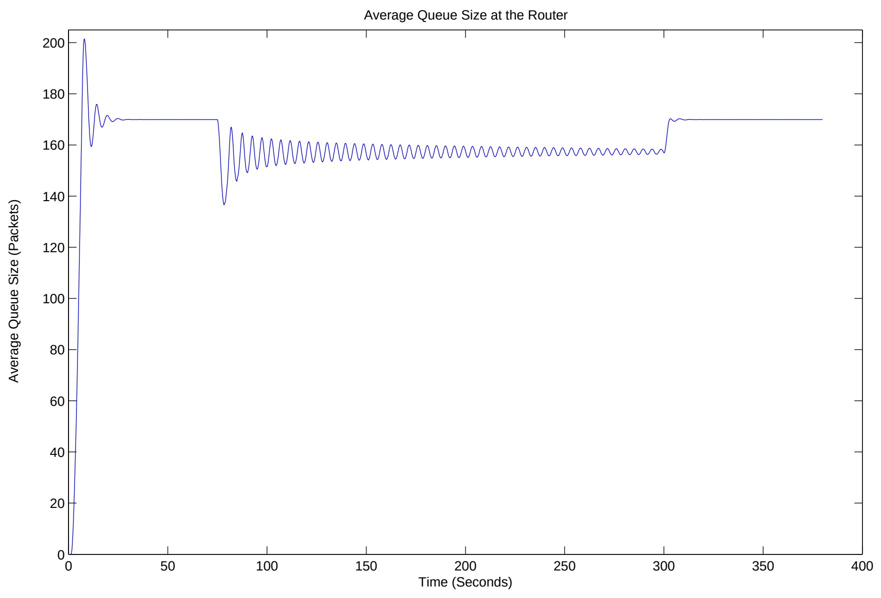


(b) Average *cwnd* of 20 persistent TCP Surathkal senders

Figure 5.5: Oscillations in the *cwnd* with TCP and TCP Surathkal

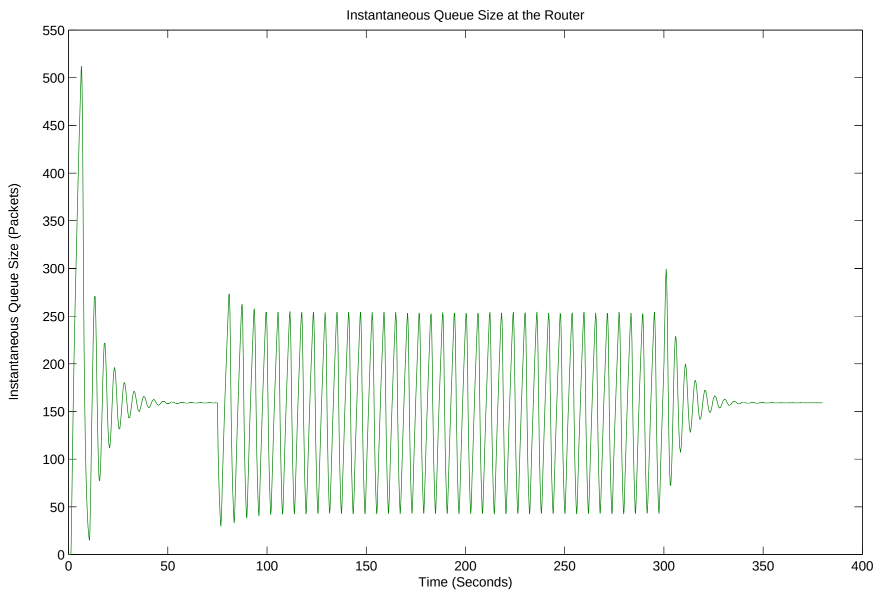


(a) *avg* with 40 TCP senders

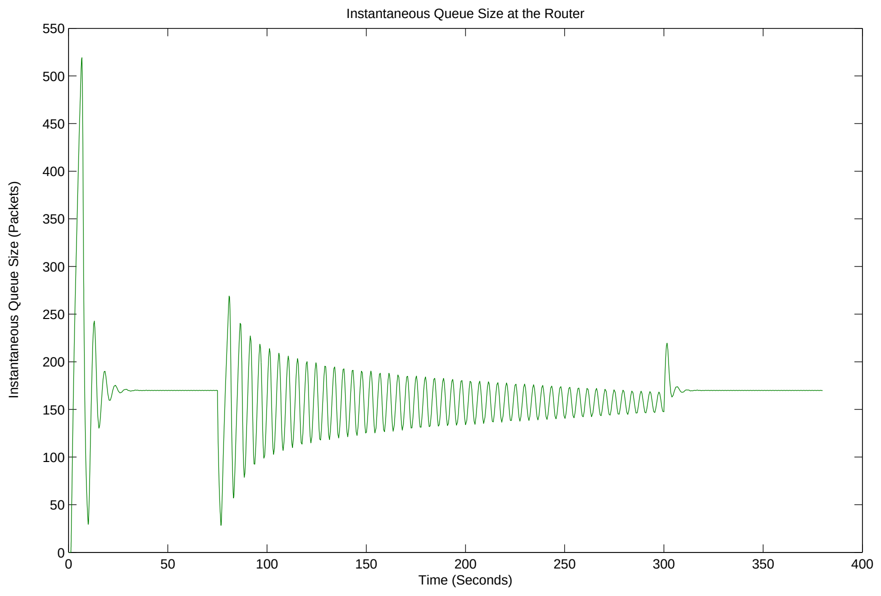


(b) *avg* with 40 TCP Surathkal senders

Figure 5.6: Oscillations in the *avg* with TCP and TCP Surathkal



(a) Instantaneous queue size with 40 TCP senders



(b) Instantaneous queue size with 40 TCP Surathkal senders

Figure 5.7: Oscillations in the queue length with TCP and TCP Surathkal

avg and the instantaneous queue length and brings the system to a stable operating point. These advantages of TCP Surathkal make it suitable for a large variety of Internet applications.

5.4 Inference

Based on the results obtained by comparing the fluid models of TCP and TCP Surathkal, we infer that by reacting based on the *severity of congestion* in the network, TCP Surathkal improves the stability of the system. TCP Surathkal stabilizes the *cwnd*, the *avg* and the instantaneous queue length and consequently improves the overall performance of the network.

Chapter 6

Conclusions and Future work

In this dissertation, we have addressed the parameter sensitivity of RED and highlighted some of the limitations of ARED and Re-ARED. We have designed and implemented two new ARED based AQM mechanisms that aim to vary the maximum drop probability efficiently, especially when the level of congestion varies sharply. We have also designed and implemented a new congestion signaling mechanism that provides a richer feedback about congestion to the end-hosts while using the bits already allocated for ECN in the TCP header. Finally, we have developed a new TCP variant which reduces its sending rate based on the *severity of congestion.* A modified fluid model is also developed to validate the performance benefits of newly designed TCP variant.

To summarize, contributions made in this research work are

- We have compared the performance of ARED with Re-ARED and demonstrated that neither a conservative approach alone nor an aggressive approach alone of updating max_p suffices to improve the overall performance in terms of throughput and packet drop rate.

- Hence, we have designed and implemented two new AQM mechanisms, namely FARED and CARED, that aim to combine the advantages of ARED and Re-ARED to improve the overall performance of the network by reducing packet drop rate and maximizing throughput. It is observed that FARED fails to achieve the desired goal because of the smaller *target range.* We note that selecting a smaller *target range* imposes constraints on the convergence time of max_p. On the other hand, CARED fulfills the desired goal of combining the benefits of ARED and Re-ARED. CARED provides robust performance in terms of throughput, packet drop rate and stability of the *avg* across a wide variety of scenarios. Moreover, since there are only algorithmic changes and no new parameter settings in CARED, it can be easily deployed in modern Internet routers without any additional complexity.

- We have designed XECN, a new congestion signaling mechanism which extends the basic functionality of the ECN mechanism. XECN not only notifies the sender about the congestion, but also enables the sender to infer the *severity of congestion* in the network by counting marked acknowledgements. XECN requires changes only at the sender and the receiver; it does not require any modification in the router. Moreover, XECN does not introduce any additional bit(s), rather, it efficiently utilizes the already available ECN bits in the TCP header. Hence, it can be incrementally deployed over the existing implementations of ECN.

- We have also designed a new variant of TCP called TCP Surathkal which decreases the *cwnd* adaptively based on the network feedback provided by XECN. We show the effectiveness of TCP Surathkal by carrying out extensive simulations using *ns*-2 and compare its performance with TCP SACK, HSTCP, STCP, CUBIC, Westwood and XCP in a wide variety of scenarios ranging from wired to wireless networks. Results demonstrate that TCP Surathkal improves overall performance of the network while being fair to the other TCP variants. Since TCP Surathkal depends on XECN for efficient utilization of network resources, it also overcomes the drawback of TCP in wireless networks by distinguishing congestion losses from non-congestion losses. Moreover, since TCP Surathkal decreases *cwnd* based on the *severity of congestion*, it achieves significant improvement even when error rates in wireless networks are moderate. The above mentioned advantages of TCP Surathkal make it suitable for use across a variety of networks such as Wired WANs, 802.11 WLAN/Mesh networks and satellite networks.

- We have developed a modified fluid model for TCP Surathkal to study its performance benefits. Results show that TCP Surathkal significantly reduces the oscillations in the queue. The effectiveness of TCP Surathkal is demonstrated by analyzing its stability behavior.

As a part of the future work, we are currently working on the modified fluid model and testing the performance of TCP Surathkal while using ARED and CARED instead of using the original RED.

Although we adapt the multiplicative decrease factor in TCP Surathkal based on the *severity of congestion*, we do not adapt the additive increase factor. We intend to explore further in this direction while keeping our major focus on the fairness issues.

The benefits of TCP Surathkal make it suitable for use in satellite networks. We aim to study the benefits of TCP Surathkal in satellite networks by comparing its performance with TCP variants which are specifically designed for satellite networks.

Moreover, the adaptive multiplicative decrease mechanism of TCP Surathkal can be easily combined with Loss Tolerant TCP (LT-TCP) (Tickoo et al. 2005) which is especially designed for extreme wireless networks and works with error rates upto 50%. It would be interesting to study the performance benefits offered by TCP Surathkal over LT-TCP in such networks.

It would be useful to evaluate the performance of CARED, XECN and TCP Surathkal in a real time implementation. CARED implementation would require collaboration with commercial vendors of routers such as Cisco, Juniper and others. XECN and TCP Surathkal can be implemented as Linux kernel modules. The results obtained in the commercial routers in the field will have a lot of bearing to validate the simulation results.

Bibliography

Abbasov, B. and Korukoglu, S. (2009). "Effective RED: An algorithm to improve REDs performance by reducing packet loss rate." *Elsevier Journal of Network and Computer Applications*, 32, 703-709.

Afanasyev, A., Tilley, N., Reiher, P., Kleinrock, L. (2010). "Host-to-Host Congestion Control for TCP." *IEEE Communication Surveys and Tutorials*, 12, 303-342.

Akamai. (2011). "The State of the Internet." *2nd Quarter, 2011 Report*, 4(2), 1-46.

Alizadeh, M., Greenberg, A., Maltz, D., Padhye, J., Patel, P., Prabhakar, B., Sengupta, S. and Sridharan, M. (2010). "Data Center TCP (DCTCP)." *ACM SIGCOMM Computer Communications Review*, 40(4), 63-74.

Altman, E., Avrachenkov, K., Barakat, C. (2000). "A stochastic model of TCP/IP with stationary random losses." *Proc., ACM SIGCOMM.*

Anjum, F. M. and Tassiulas, L. (1999). "Balanced RED: An algorithm to achieve fairness in the Internet." *Technical Report TR99-17*, Department of Electrical Engineering and Institute for Systems Research, University of Maryland at College Park.

Aweya, J., Ouellette, M., Montuno, D. Y. (2001). "A Control Theoretic approach to Active Queue Management." *Elsevier Computer Networks*, 36, 203-235.

Balakrishnan, H., Padmanabhan, V. N., Seshan, S., Katz, R. H. (1997). "A Comparison of Mechanisms for Improving TCP Performance over Wireless Links." *IEEE/ACM Transactions on Networking*, 5(6), 756-769.

Bansal, D., Balakrishnan, H., Floyd, S., Shenker, S. (2001). "Dynamic behavior of slowly-responsive congestion control algorithms." *ACM SIGCOMM Computer Communications Review*, 31(4), 263-274.

Bauer, S., Beverly, R., Berger, A. (2011). "Measuring the State of ECN Readiness in Servers, Clients, and Routers." *Proc., 2011 ACM SIGCOMM Conference on Internet Measurements*, 171-180.

Bhandarkar, S. and Reddy, A. L. N. (2007). “Emulating AQM from end hosts.” *ACM SIGCOMM Computer Communications Review*, 37(4), 349-360.

Brakmo, L. and Peterson, L. (1995). “TCP Vegas:End to End Congestion Avoidance on a Global Internet.” *IEEE Journal on Selected Areas in Communications.*

Casetti, C., Gerla, M., Mascolo, S., Sanadidi, M. Y., Wang, R. (2002). “TCP Westwood: End-to-End Congestion Control for Wired/Wireless Networks.” *Wireless Networks Journal*, 8(5), 467-479.

Chen, J., Hu, C., Ji, Z. (2011). “Self-tuning Random Early Detection algorithm to improve performance of network transmission.” *Mathematical Problems in Engineering*, 2011, Article ID. 872347.

Chiu, D.-M. and Jain, R. (1989). “Analysis of the Increase and Decrease Algorithms for Congestion Avoidance in Computer Networks.” *Computer Networks and ISDN Systems*, 17, 1-14.

Dukkipati, N. (2008). “Rate Control Protocol (RCP): Congestion control to make flows complete quickly.” *Ph.D thesis*, Department of Electrical Engineering, Stanford University.

Feng, G., Agarwal, A., Jayaraman, A., Siew, C. K. (2004). “Modified RED Gateways under Bursty Traffic.” *IEEE Communications Letters*, 8, 323-325.

Feng, W., Kandlur, D., Saha, D., Shin, K. G. (1999). “A Self Configuring RED.” *Proc., IEEE INFOCOM ’99*, 3, 1320-1328.

Floyd, S, Gummadi, R., Shenker, S. (2001). “Adaptive RED: An algorithm for increasing the robustness of RED’s Active Queue Management.” *Technical Report.*

Floyd, S, (2003). “HighSpeed TCP for Large Congestion Windows” *Request For Comments 3649, Experimental.*

Floyd, S. and Jacobson, V. (1993). “Random Early Detection Gateways for Congestion Avoidance.” *IEEE/ACM Transactions on Networking*, 1(4), 397-413.

Gerla, M. and Kleinrock, L. (1980). “Flow control: A Comparative Survey.” *IEEE Transactions on Communications*, 28(4), 553-574.

Gettys, J. (2011). “Dark Buffers in the Internet.” *IEEE Internet Computing*, 15(3), 96.

Hariri, B. and Sadati, N. (2007). “NN-RED: An AQM mechanism based on Neural Networks” *Electronic Letters*, 43, 10531055.

Hassan, M. and Jain, R. (2004). "High Performance TCP/IP Networking: Concepts, Issues and Solutions." *Pearson Education, Inc.*, New Jersey.

Hollot, C., Misra, V., Towsley, D., Gong, W. (2002). "Analysis and Design of Controllers for AQM Routers Supporting TCP Flows." *IEEE Transactions on Automatic Control*, 47(6), 945-959.

Jacobson, V. (1988). "Congestion Avoidance and Control" *Proc., ACM SIGCOMM '88*, Stanford, CA.

Jacobson, V. (1990). "Modified TCP Congestion Avoidance Algorithm." *Technical Report*, Email to the end2end-interest Mailing List.

Jain, R. (1986). "A Timeout-based Congestion Control Scheme for Window Flow-controlled Networks." *IEEE Journal on Selected Areas of Communications*, 7.

Javam, H. and Analoui, M. (2006). "SARED: Stabilized ARED." *Proc., International Conference on Communication Technology*, 1-4.

Katabi, D., Handley, M., Rohrs, C. (2002). "Internet Congestion Control for High Bandwidth-Delay Product Networks." *Proc., ACM SIGCOMM '02.*

Kelly, T. (2003). "Scalable TCP: Improving performance in highspeed wide area networks." *ACM SIGCOMM Computer Communications Review*, 33(2), 83-91.

Kim, T. and Lee, K. (2006). "Refined Adaptive RED in TCP/IP Networks." *Proc., SICE-ICASE International Joint Conference*, pp. 3722-3725.

Kwon, M. and Fahmy, S. (2002). "TCP Increase/Decrease Behavior with Explicit Congestion (ECN)." *Proc., IEEE International Conference on Communications*, 4, 2335-2340.

La, R. J., Ranjan, P., Abed, E. H. (2003). "Analysis of Adaptive Random Early Detection (Adaptive RED)." *Proc., 18th International Teletraffic Congress (ITC 03).*

Lin, D. and Morris, R. (1997). "Dynamics of Random Early Detection." *Proc., ACM SIGCOMM '97*, 127-137.

Liu, S., Basar, T., Srikant, R. (2005). "Exponential-RED: A Stabilizing AQM scheme for low and high-speed TCP protocols." *IEEE/ACM Transactions on Networking*, 13, 1068-1081.

Mahajan, R. and Floyd, S. (2001). "Controlling high bandwidth flows at the congested router." *Proc., IEEE ICNP '01.*

Marquez, R., González, I., Carrero, N., Sulbarán, Y. (2007). "Revisiting Adaptive RED: Beyond AIMD Algorithms." *Proc., 1st EuroFGI International Conference on Network Control and Optimization (NET-COOP'07)*, 30(4), 74-83.

Mathis, M., Semke, J., Mahdavi, J. (1997). "The macroscopic behavior of the TCP congestion avoidance algorithm." *ACM Computer Communications Review*, 27(3), 67-82.

Misra, V., Gong, W.-B., Towsley, D. (2000). "Fluid-based Analysis of a Network of AQM Routers Supporting TCP flows with an Application to RED." *ACM SIGCOMM Computer Communications Review*, 30(4), 151-160.

Nagle, J. (1984). "Congestion Control in IP/TCP Internetworks." *Request For Comments 896.*

Ott, T. J., Lakshman, T. V., Wong, L. (1999). "SRED: Stabilized RED." *Proc., IEEE INFOCOM '99*, 1346-11355.

Padhye, J., Firoiu, V, Towsley, D., Kurose, J. (2000). "Modeling TCP throughput: A simple model and its empirical validation." *IEEE/ACM Transactions on Networking*, 8(2), 133-145.

Parris, M., Jeffay, K, Smith, F. D. (1999). "Lightweight Active Router Queue Management for Multimedia Networking." *Proc., SPIE*, 162-174.

Paxson, V., Allman, M, Chu, J., Sargent, M. (2011). "Computing TCP's Retransmission Timer." *Request For Comments 6298, Standards Track.*

Qazi, I. A. (2010). "An Efficient Framework of Congestion Control for Next-Generation Networks." *Ph.D Dissertation*, University of Pittsburgh.

Ramakrishnan, K. K., Floyd, S., Black, D. (2001). "The Addition of Explicit Congestion Notification (ECN) to IP." *Request For Comments 3168, Standards Track.*

Ramakrishnan, K. K. and Jain, R. (1988). "A Binary Feedback Scheme for Congestion Avoidance in Computer Networks with a Connectionless Network Layer." *Proc., ACM SIGCOMM '88*, 314-329.

Ramani, R. and Karandikar, A. (2000). "Explicit Congestion Notification (ECN) in TCP over Wireless Networks." *Proc., IEEE ICPWC*, 495-499.

Ranjan, P., Abed, E. H., La, R. J. (2002). "Nonlinear instabilities in TCP-RED." *IEEE/ACM Transactions on Networking*, 12(6), 1079-1092.

Rhee, I. and Xu, L. (2005). "CUBIC: A New TCP-Friendly High-Speed TCP Variant." *Proc., PFLDNet'05.*

Tickoo, O., Subramanian, V., Kalyanaraman, S., Ramakrishnan, K. K, (2005). "LT-TCP: End-to-end framework to Improve TCP Performance over Networks with Lossy Links." *Proc., International Workshop on Quality of Service (IWQoS).*

Verma, R., Iyer, A., Karandikar, A. (2002). “Active Queue Management using Adaptive RED.” *IEEE/KICS Journal of Communications and Networks*, 5, 275-281.

Toomey, W. K (1997). “A Rate-Based Congestion Control Framework for Connectionless Packet-Switched Networks.” *University of New South Wales, Doctoral thesis.*

Wang, C., Liu, J., Li, B., Sohraby, K., Hou, Y. T. (2007). “LRED: A Robust and Responsive AQM algorithm using packet loss ratio measurement.” *IEEE Transactions on Parallel and Distributed Systems*, 18, 29-43.

Wang, G., Xia, Y., Harrison, D. (2007). “An NS-2 TCP Evaluation Suite.” *Internet Draft, Informational.*

Welzl, M. (2006). “Network Congestion Control: Managing Internet Traffic” *Wiley Series in Communication Networking and Distributed Systems.*

Yang, Y. R. and Lam, S. S. (2000). “General AIMD Congestion Control.” *University of Texas at Austin, Technical Report TR-200009.*

Zheng, B. and Atiquzzaman, M. (2000). “DSRED: an Active Queue Management scheme for Next Generation Networks” *Proc., 25th Annual IEEE Conference on Local Computer Networks (LCN’00)*, 242-251.

Zhou, K., Yeung, K. L., Li, V. O. K. (2006). “Nonlinear RED: A simple yet efficient Active Queue Management Scheme” *Elsevier Computer Networks*, 50, 3784-3794.

List of Publications

Refereed Journals

1. **Tahiliani, M. P.,** Shet, K. C. and Basavaraju, T. G. (2012). "CARED: Cautious Adaptive RED Gateways for TCP/IP Networks." *Journal of Network and Computer Applications, Elsevier*, 35, 857-864.

2. **Tahiliani, M. P.,** Shet, K. C. and Basavaraju, T. G. (2012). "Comparative Study of High-speed TCP Variants in Multi-hop Wireless Networks." *International Journal of Computer Theory and Engineering*, accepted for publication, November 2012.

3. **Tahiliani, M. P.,** and Shet, K. C. (2012). "TCP Surathkal: Network Controlled TCP for Wired and Wireless Networks." *IEEE/ACM Transactions on Networking*, submitted, January 2012.

4. **Tahiliani, M. P.,** and Shet, K. C. (2013). "TCP Surathkal: A Proactive TCP for efficient Congestion Control." *Journal of Computer Communications, Elsevier*, *to be submitted.*

Conference Proceedings

1. **Tahiliani, M. P.,** Shet, K. C. and Basavaraju, T. G. (2012). "FARED: Fast Adapting RED Gateways for TCP/IP Networks." *Proc., International Conference on Advanced Computing, Networking and Security (ADCONS 2011)*, Springer LNCS, N.I.T.K., Surathkal, 434-442.

2. **Tahiliani, M. P.,** Shet, K. C. and Basavaraju, T. G. (2010). "Performance Evaluation of TCP Variants over Routing Protocols in Multi-hop Wireless Networks." *Proc., International Conference on Computer and Communication Technology (ICCCT 2010)*, IEEE Xplore, M.N.N.I.T., Allahabad, 387-392.

3. **Tahiliani, M. P.,** and Shet, K. C. (2013). "Analysis of Cautious Adaptive RED." *to be submitted to IEEE ICACCI 2013*, IEEE Xplore, SJCE, Mysore, March 2013.

Brief Bio-Data

Mohit P. Tahiliani
Research Scholar
Department of Computer Science and Engineering
National Institute of Technology Karnataka, Surathkal
P.O. Srinivasanagar
Mangalore, 575025
Phone: 09844965159
Email: tahiliani.nitk@gmail.com

Permanent address

Mohit P. Tahiliani
S/o, Dr. Prakash U. Tahiliani,
17/B, “Udhav-Kunj”,
Alka Society, Chhapra Road,
Navsari-396445,
Gujarat, INDIA.

Qualification

M.Tech. Computer Science and Engineering, Visvesvaraya Technological University, Belgaum, 2009.

B.E. Computer Science and Engineering, Visvesvaraya Technological University, Belgaum, 2007.