COMPLEMENTING CURRENT ACTIVE QUEUE MANAGEMENT SCHEMES WITH RECHOKE AND RECEIVER-WINDOW MODIFICATION (RWM)

by

VISVASURESH GOVINDASWAMY

Presented to the Faculty of the Graduate School of

The University of Texas at Arlington in Partial Fulfillment

of the Requirements

for the Degree of

DOCTOR OF PHILOSOPHY

THE UNIVERSITY OF TEXAS AT ARLINGTON

December 2006

ACKNOWLEDGEMENTS

First I am extremely gratefully and thankful to the Almighty, God, for giving me strength, knowledge and providing me an opportunity to witness this incredible day when I am writing this final page of my dissertation. Without His blessings nothing could have been possible for me.

I am extremely grateful to Gergely Záruba, my dissertation advisor, for giving me the right amount of freedom and guidance during my graduate studies. From the very beginning, he treated me as a peer and as a friend. He was instrumental in maintaining my focus, and not only taught me how to become a better researcher, but also helped me to become a better person. His engaging arguments and strong feedback have contributed greatly to this dissertation. I hope and look forward to continued collaboration with him in the future.

I am grateful to the other members of my committee, Ramez Elmasri, Hao Che, Leonidas Fegaras, Manfred Huber and David Levine, for their feedback and advice that helped to shape my research skills.

I would like to express my earnest gratitude to my parents for their love and support, without which any of my achievements would not have been possible. Thanks to my father, J. Govindaswamy, who sparkled my early interest in science and engineering. His undaunting confidence gave me the strength to overcome any difficulties and to maintain high goals. Thanks to my mother, Saroja, for her love and the countless sacrifices she made to raise me and to give me the best possible education. I must also thank my brothers, Jeganathan and Balasekaran, my sister, Bama, and my nephews Ramrajiv, Harisanjiv, Kirtiranjiv and Hemasagjiv. Their encouragement and loving support certainly helped me through my PhD. I am deeply indebted to my lifelong friend, Shiva, for all his help and support, without which it would not have been possible to reach this stage in my career.

October 30, 2006

ABSTRACT

COMPLEMENTING CURRENT ACTIVE QUEUE MANAGEMENT SCHEMES WITH RECHOKE AND RECEIVER-WINDOW MODIFICATION (RWM)

Publication No. ______

Visvasuresh Govindaswamy, PhD.

The University of Texas at Arlington, 2006

Supervising Professor: Gergely Záruba

Explicit Congestion Notification (ECN) and Active Queue Management (AQM) schemes have been proposed for present-day TCP/IP networks to better manage network congestion. ECN enabled AQMs were shown to have a promising advantage over existing drop-tail queues. However, when implemented, they were susceptible to the following problems: 1) the timeout mechanism or the duration of the reception of three duplicate acknowledgements (ACKs), due to early-dropped packets by these AQMs, delays the response time of TCP (in reducing the offered rate) 2) using ECN with these AQMs has its downsides: i) its messages may get delayed or dropped due to congestion in downstream routers; and ii) TCP implementations at both the source and the destination have to be ECN-compliant (which presents a significant problem in today's cores and end systems); and 3) these AQM schemes, with or without ECN, fail to protect TCP-friendly flows adequately in the presence of non TCP-friendly (e.g., UDP) or malicious flows.

This dissertation presents solutions to these problems by proposing two novel AQM modification schemes called Receiver-Window Modification (RWM) and RECHOKe (REpeatedly CHOose and Keep for responsive flows, REpeatedly CHOose and Kill for unresponsive flows). By combining these two schemes with RED, we produce a new AQM scheme called RCUBE (Receiver-Window Modified Random Early Detection queues with RECHOKe). By using RECHOKe as a component, RCUBE easily identifies, controls and punishes malicious flows, by requiring only a small amount of information, approximately proportional to the order of magnitude of malicious flows. By using RWM, we reduce the average TCP queue sizes in the queues and in doing so, not only make it easier to identify malicious flows but also reduce the queuing delay resulting in significant improvements in one-way end-to-end packet delays, delay jitter, throughput and number of dropped packets for TCP-friendly flows. We compare RED, CHOKe, xCHOKe, RECHOKe and RCUBE schemes and show that RCUBE outperforms these schemes in identifying, controlling and punishing malicious flows and in protecting TCP-friendly flows. We also provide a theoretical analysis for RCUBE, RECHOKe and RWM schemes to validate our claims.

TABLE OF CONTENTS

LIST OF ILLUSTRATIONS

LIST OF TABLES

CHAPTER 1

INTRODUCTION

The Transmission Control Protocol (TCP) is the major connection oriented *transport layer protocol* used in today's Internet [1]. It uses the services provided by the Internet Protocol (IP, at the network layer, which is responsible for end-to-end transfer of datagrams). Since IP does not inherently provide reliable service to applications, TCP is tailored to provide reliability and transmission rate control to applications using the Internet. The two transmission control features embedded in TCP are flow control and congestion control:

- Flow control is responsible to provide means for transmitting nodes to reduce (and increase) their transmission rate in order to prevent buffer overload (and underload) at the receiving node. In TCP, flow control is enabled by the receiving node directly informing the transmitter about its available buffer size using designated fields in the TCP header of a packet.
- Congestion control on the other hand is a process to deal with buffer overload at routers along the transmission path. If the buffer of a router on the relay path is saturated then the router may drop packets determined by its buffer control policy. If such congestion control was not available, TCP would retransmit the dropped packets thus increasing its transmission rate, in a positively fed-back manner.

This dissertation addresses congestion related to present-day networks, i.e. TCP/IP networks with best-effort traffic. This means that networks today provide a service that does not make any promise of whether a packet is actually delivered to the destination, or whether the packets are delivered in order or not however it will do its best to indeed deliver datagrams. The main goal in our research is to significantly improve the performance of the present-day networks while minimizing the change required in bringing about these improvements so as to minimize the cost of such changes. Hence, we do not consider traffic protected by Quality of Service (QoS) mechanisms such as Differentiated [2] and Integrated [3] Services although some of our proposed improvements can complement these mechanisms. To achieve our overall goal, additional mechanisms are only needed at routers to protect the network from congestion from both TCP-friendly and "misbehaving" (e.g., UDP) flows. The difference between these two types of flows is that the latter are flows that do not use conformant TCP congestion control. "Misbehaving" flows are also known in network literature as non-responsive or malicious flows while TCP-friendly flows are known as responsive flows or non-malicious flows.

Several solutions, such as new Active Queue Management (AQM) [4] schemes, have been proposed to replace the existing drop-tail queues which dominate routers in present-day IP networks. Explicit Congestion Notification (ECN) [5][6] enabled AQMs were shown to have a promising advantage over existing drop-tail queues. However, when implemented, they were susceptible to the following problems:

1) the timeout mechanism or the duration of the reception of three duplicate acknowledgements (ACKs), due to early-dropped packets by these AQMs, delays the response time of TCP (in reducing the offered rate)
2) using ECN with these AQMs has its downsides: i) its messages may get delayed or dropped due to congestion in downstream routers; and ii) TCP implementations at both the source and the destination have to be ECN-compliant (which presents a significant problem in today's cores and end systems)
3) these AQM schemes, with or without ECN, fail to protect TCP-friendly flows adequately in the presence of non TCP-friendly (e.g., UDP) or malicious flows.

In our research, we make use of flow control as well as congestion control to achieve transmission rate control using a scheme called Receiver-Window Modification (RWM). We also present a mathematical model to illustrate the benefits of using RWM on queues such as RED [7]. To protect TCP-friendly flows from "misbehaving" or malicious flows, we also propose a scheme called RECHOKe (**RE**peatedly **CHO**ose and **K**eep for responsive flows, **RE**peatedly **CHO**ose and **K**ill for unresponsive flows) to detect, control and punish malicious flows, thereby protecting the TCP-friendly or responsive flows. Finally, we present RCUBE (**R**eceiver Window Modified **R**andom Early Detection queues with **R**ECHOKe) scheme that combines the benefits of RWM and RECHOKe. Although both the RWM and the RECHOKe schemes are used in RCUBE, they can also be used with most other AQM schemes. Hence, this dissertation presents solutions to the above-mentioned problems. To solve problems identified with the first and the second points above we introduce RWM and to address the third issue RECHOKe is presented.

The remainder of this chapter is organized as follows. Sections 1.1 to 1.4 present a bird's eye view of the remaining chapters, outlining the motivations and contributions, reviewing current solutions briefly and providing brief overviews for each scheme. Thus these sections serve as extended abstracts for the rest of this dissertation with an aim to ease the reader into the discussed topics. Sections 1.1 and 1.2 discuss the Receiver-Window Modification (RWM) Scheme and its mathematical model whereas the RECHOKe and RCUBE schemes are presented briefly in Sections 1.3 and 1.4 respectively. Finally this chapter ends with information describing the organization of this dissertation.

1.1 Receiver-Window Modification (RWM) Scheme

In common TCP implementations congestion control employs indirect feedback [8], i.e., the transmitting node will deduce that a segment was dropped (or marked in the case of ECN), due to congestion if no (within a time period) or negative (in the form of duplicate positive acknowledgements) acknowledgements have been received. The transmission rate of the TCP sender is then drastically reduced to help the routers to recover. More precisely, TCP congestion control consists of two phases called *Congestion Avoidance* (CA) and *Slow Start* (SS) [9]. TCP senders thus need to probe the network as to how high of a rate it can support. TCP flows start off with the SS algorithm, where the number of allowable pending segments (the rate) is increased binary exponentially within each round trip time (RTT). When a predefined threshold is reached the congestion control mechanism changes over to the CA. CA increases the rate with one segment per RTT. If congestion is encountered, then the rate is halved for each RTT a loss event has happened. If the rate drops below a dynamically adjusted threshold, the SS algorithm is invoked again. This unique combined behavior of the CA and SS algorithms may lead to what is known as the *Global Synchronization Problem* [10]. This problem occurs whenever multiple flows lose packets at the bottleneck router, leading to the flows "restarting" concurrently, resulting in reduced aggregate throughput. It has also been shown that the sudden changes in the TCP transmission window caused by the interaction of SS and CA create a chaotic process introducing self-similarity in the traffic [11]. Present day, queue management solutions, such as RED [7], use more sophisticated solutions in the queues of routers to determine which packets to drop thus trying to compensate for the Global Synchronization Problem. We provide a brief description of such techniques in Chapter 2.

In the presence of bursty Internet traffic, routers along the path may buffer packets in their queues to absorb/reduce the burstiness. Queuing theory implies that the bigger the queue's capacity is, the more a queue can deal with the burstiness without dropping a packet. Furthermore, recent results have also shown that queuing and aggregation of packets can reduce the order of self-similarity of Internet traffic [12]. Unfortunately, TCP's congestion control can introduce a high variance (resulting in a high delay jitter) in the current buffer sizes. On the other hand, if the maximum buffer size of routers is increased, the TCP senders will be notified about congestion much later. This will lead to an increase in the flow's end-to-end delay. This delay is the sum of two components:

i) the transmission, propagation and processing delays (which can be considered fixed delays from our perspective);

ii) the queuing delays at the routers (which is variable and depends on the distribution and magnitude of rates).

Hence, the greater the current queue size, the greater the queuing delay will be in a case of saturation, increasing the observed end-to-end packet delay. On the other hand, if the queue size is too small, packets will be dropped regularly due to buffer overflow. Hence, there is a need for solutions to determine an optimal (or sub-optimal) queue sizes. The research area dealing with the above phenomenon is called *TCP Queue Management* (QM). The aim of QM is to minimize the congestion in the network by properly managing routers' queues. QM needs to differentiate between transient and long-term congestion. Transient congestion takes place when the congestion period is shorter than the reaction time needed for the TCP congestion control mechanisms to kick in. On the other hand, long-term congestion is due to overall heavy traffic load. QM approaches can be divided into two categories:

i) passive queue management algorithms, where the queue of the routers needs to fill up before corrective action is taken; and

ii) active queue management (AQM) [13], where preventive action is taken before the queue is fully saturated.

AQM algorithms can again be subdivided into:

- Load based AQM approaches, including Random Exponential Marking (REM) [14], Adaptive Virtual Queue (AVQ) [15] and PI controller [16] algorithms. Load based approaches are out of the focus of this paper. They are merely mentioned for the completeness of discussion.
- Queue-based AQM, including Random Early Detection (RED) [7], Stabilized RED (SRED) [17], Flow RED (FRED) [18], Adaptive RED (ARED) [19], and BLUE [20].

1.1.1 Motivation

Changing implementation of transport and network layers in today's TCP/IP networks to implement some of these schemes is difficult and expensive. Moreover, network administrators and users are more likely to be resisting such change. The present idea is to replace drop-tail queues with AQMs such as RED at congested regions of the network, i.e., at edge routers. The goals of these AQMs are ideally: no packet loss, 100% link utilization, low queuing delay and low jitter rate. Hence, our motivation behind this research is to improve these AQMs to get as close as possible to these ideal goals.

Using AQM schemes that employ average queue length measures to manage congestion come with their own problems, e.g., such problems for RED are well-documented [17][19][21][22]. The level of congestion at routers with their immediate links (neighbours), compounded with RED's complex parameter tuning [22] induce large variations in RED queue sizes. The queue size is near the low threshold parameter

- Th_{min}, whenever the link is lightly congested or when the maximum blocking probability - P_{max} is high. However, the queue size increases to around the high threshold parameter - Th_{max} with an increase in the congestion levels or if P_{max} is set to a low value. Both these situations result in a degraded throughput, i.e., RED may underperform even the Drop-Tail mechanism [10]. RED may also introduce jitter into non-bursty streams. All of these disadvantages have generated more research using RED as a starting point for further improvements [14][19][20].

1.1.2 Major Research Contribution

This paper deals with queue-based AQM, presenting a novel approach called Receiver-Window Modification (RWM) for congestion avoidance in TCP. Using our RWM scheme, the problems mentioned in the previous section are minimized in schemes that use the average queue length to manage congestion. In AQMs, such as RED, ARED and BLUE, the timeout mechanism or the duration of the reception of three duplicate ACKs, due to early-dropped packets, delays the response time of the TCP congestion scheme in reducing the network congestion. ECN may be used to mark packets instead of dropping them, however it also has its downsides:

i) ECN marked messages may get delayed or worse, lost; and

ii) TCP implementations at both the source and the destination (as well as all routers) have to be ECN-compliant (which presents a significant problem in today's implementations).

To minimize these problems, RWM, which can be used together with RED, ARED, BLUE or other AQM queues, is used for congestion avoidance in packet switched networks, especially at the ingress and gateway routers. RWM works with RED, ARED and BLUE algorithms together relaxing their need for a modified TCP layer at servers and clients since it does not require modification to TCP implementations at servers or clients, i.e., no “RWM-compliancy” is needed. Our approach is specifically designed for ingress and gateway routers which are common locations of heavy congestion. The main idea of our approach is to restrict the TCP transmission window with the flow control window instead of the congestion control window, thus controlling the transmission window with a finer granularity. We will show that RWM modified queues improve on the average queue size, the one-way packet delay, number of dropped packets and throughput as compared to RED, RED-ECN, ARED, ARED-ECN, BLUE and BLUE-ECN queues, especially in paths that have non-ECN compliant routers (which is a true reflection of present-day networks). Moreover, we also show that AQMs using ECN can be greatly influenced by the congestion state in downstream routers.

1.2 Modeling of RWM

There are two basic ways to perform congestion control for flows:

1) End-to-end: where congestion is inferred from end-system observed segment loss and/or delay. There is no explicit feedback from the network routers; this is the current approach in the Internet.

2) Network-assisted: where routers provide feedback to end systems or explicitly specify rate at which the sender should transmit. Extensions to TCP/IP have

been proposed for network-assisted congestion control, e.g., using a single bit in the headers indicating congestion (TCP/IP's ECN).

Modeling and analysis of TCP connections with queues has been an active research area. A Markov chain based model [23] has been presented for *N* TCP connections using either Tail-Drop or RED gateways, where the state space is the vector of window size of all of the *N* connections, the queue size is a function of the sum of the size of congestion windows and state transition probabilities are dependent on packet loss probability at the router buffer, the slow-start (SS) and congestion avoidance (CA) phases of TCP. Many of the past works assume that the packet loss probability is constant but it is in fact dependent on the buffer size and packet discarding discipline at the router and the window size of the transmission. In [24], the authors use a previously developed nonlinear dynamic model of TCP to analyze and design RED by relating its parameters such as the low-pass filter break point and loss probability profile to the network parameters. Finally, Tinnakornsrisuphap et al. [25][26][27][28][29][30][31] have modeled a RED queue with TCP connections that explicitly incorporate complete packet level operations in TCP, probabilistic packet marking mechanism in RED, heterogeneous round trip times (RTTs) of TCP flows and session-layer dynamics, *i.e.,* connection arrivals and departures. Their model avoids the state space explosion of [23] as well, however assuming that the RED queue does not drop packets.

1.2.1 Motivation

AQM schemes, when implemented, have parameters that are difficult to configure and hence, they are usually sub-optimally set. The main goal behind the introduction of these AQM schemes is to ease congestion within the networks. However, when configured poorly, they may make the congestion worse or the networks significantly underutilized. Thus, having a mathematical model that captures the behavior of TCP and its interaction with the AQM schemes can help to elevate the understanding of configuring such schemes to suit networks. We will model RWM in the context of RED, however the modeling work presented could be easily extended to other types of AQMs.

1.2.2 Major Research Contribution

We are going to present a mathematical model for the Receiver-Window Modification (RWM) scheme that can be used to complement the RED scheme at the ingress and gateway routers. Our model extends the modeling work done in [25][26][27][28][29][30][31] to the RWM scheme, henceforth known as RED-RWM, relaxing the assumption that the buffer capacities are infinite. Our model has the following features:

1. Incorporates a more accurate model for the Additive Increase/Multiplicative Decrease (AIMD) window mechanism of TCP. The AIMD algorithm controls the window size in response to the modified acknowledgement packet from the bottleneck RED-RWM gateway.
2. Provides more realistic session-level dynamics, *i.e.,* arrivals and terminations of flows, and the variability of round-trip delays with modification of the receiver window field by the RWM scheme.
3. Provides a realistic interaction between TCP and RED-RWM gateway.

4. Presents a RED-RWM model that deals with TCP transmission rate in terms of packet-level operations.
5. Unlike the RED model in [25][26][27][28][29][30][31], our model incorporates packet drops.

Using this model, we will provide:

1. A weak convergence theorem for the number of connections; thus leading the way to more relaxed configuration of RED-RWM gateway parameters. We will show that the average queue size of RWM weakly converges to the number of connections N at steady state provided that N is large and, N > Th_{min}, and N < Th_{max}.
2. A Monte Carlo simulation based on our model to compare outcomes to a discrete event simulation model of NS2 [32]. We will show that the two models validate each other.
3. An asymptotic and steady-state analysis, showing that the average TCP dynamics for a large number of flows competing for a RED-RWM queue is closely related to that of a single flow using the queue utilizing the same TCP congestion control mechanism.
4. The sessions become asymptotically independent as the number of flows becomes large, suggesting that RED-RWM alleviates the synchronization problem among flows.
5. We thus have both theoretical and experimental proofs that RWM aids RED in reducing the average queue length, thereby lowering queuing delay.

1.3 RECHOKe

To control TCP-friendly flows from non-adaptive sources such as UDP and non-TCP-friendly sources, we are proposing a new preferential dropping scheme called RECHOKe (REpeatedly CHOose and Keep for responsive flows, REpeatedly CHOose and Kill for unresponsive flows). It aims to improve on proposed schemes such as CHOKe [33] and its variants such as Back CHOKe [33], Front CHOKe [33], Multi-drop CHOKe (M-CHOKe) [33], Adaptive CHOKe (A-CHOKe) [33], CHOKe+ [34] and xCHOKe [35][36] for router buffer management.

CHOKe is a buffer management scheme that enables routers in an IP network to control congestion in the case when TCP segments are not the only segments queued. CHOKe compares each newly arriving packet with a randomly selected packet from the queue. If they are from the same flow (referred to as a CHOKe hit), then both packets are dropped; otherwise the arriving packet is allowed to enter the queue (referred to as a CHOKe miss). CHOKe has been shown to be erratic in recognizing malicious flows [35][36] while often punishing non-malicious flows. Its advantage of being stateless can become a disadvantage; if it had kept some state, it would be able to make better and more accurate decisions. To improve upon CHOKe, xCHOKe was introduced. In xCHOKe, hits are stored in a table which is refreshed periodically (e.g., every t ms). xCHOKe uses this table to check whether the arriving packet's flow label is already in it. If it is (referred to as a table hit), the arriving packet is dropped or marked for dropping with a probability $p*$. After this step the packet is compared with a randomly selected packet from the queue. If this results in a hit (referred to as a CHOKe hit), then both packets are dropped or marked for dropping and the flow label is added to the lookup table (associated with a hit counter of one). If the flow is already in the table, the associated hit counter is incremented. The hit counter of flows in the lookup table is used to compute the probability p^*.

1.3.1 Motivation

Since xCHOKe is dependent on CHOKe hits to drop packets and update its table, it often punishes non-malicious flows as well. xCHOKe performs better than CHOKe by keeping partial state, however it (similarly to CHOKe) does not control malicious flows with bandwidths smaller or slightly greater than the link capacity. xCHOKe was shown to behave like CHOKe when

1) the number of malicious flows is small and/or

2) the bandwidth of malicious flows are at or around the link capacity.

There are other approaches to punish unresponsive flows but most of the time they incur too much overhead. Schemes like RED-PD [37] require proper values for their parameters such as the target RTT. The overhead of RED-PD and RED-NB [38] in today's networks lies in that it is too complex for the router to constantly calculate flows' RTTs, as RTTs are heavily dependent on the ever-changing state of the network. Schemes like RED-NB and Self Adjustable CHOKe [39] also need to access the *protocol* field in the IP packet header to differentiate between TCP and UDP flows and hence, are not safe as these fields can be disguised in denial-of-service attacks. Another example is the Core-Stateless Fair Queuing (CSFQ) scheme [40]. Although CSFQ is promising, it requires modifications to the entire network and an additional field in the

IP header; making CSFQ unpractical for today's networks. CSFQ requires that packets be marked with an estimate on their current transmission rate at the edge router upon entering the network. Upon receiving these packets, core routers estimate the flow's fair share and preferentially drop a segment if the fair share is less than the rate estimate carried by the packet. Yet other examples are schemes like Flow Random Early Detection (FRED) [18], which incur too much overhead as they seek to insure that all flows receive a fair share of the link's capacity by logically managing the queue on a per-flow basis and maintaining statistics for every flow that has a packet queued in the router. Another scheme, which uses approximate fair queuing, is Stochastic Fair Queuing (SFQ) [41]. SFQ classifies packets into a smaller number of queues than Fair Queuing using a hash function, requiring around 1000 to 2000 queues in a typical router to achieve its full potential [42], thereby making it too costly. Hence, we need a scheme, which is not only reliant on CHOKe hits or RED hits, but is computationally inexpensive controlling and punishing both high- and low-bandwidth malicious flows while protecting TCP-friendly flows.

1.3.2 Major Research Contribution

Our scheme combines the ideas behind xCHOKe and RED-PD to detect, control and punish malicious flows which, we believe, are faster than xCHOKe and RED-PD. Since xCHOKe, uses the CHOKe hit history and RED-PD the RED drop/mark history, our scheme uses both these histories to isolate, control and punish malicious flows. We extend xCHOKe, by using RED drop history; this makes intuitive sense since the probability that a gateway chooses a particular flow to mark or drop during congestion is roughly proportional to that flow's share of the bandwidth at the router. We believe (and show) that using both histories helps to lower the number of “false alarms” and detect malicious flows faster, especially when the numbers of flows using the queue are large. Hence, in xCHOKe, CHOKe hits are entered as CHOKe hit history in a table, while our scheme - RECHOKe both CHOKe hit and CHOKe-RED drop/mark histories are used to detect and thwart malicious flows. We denote the RED queue used by the CHOKe scheme by affixing “RED” after the name of the CHOKe, thus we will be talking about CHOKe-RED queues.

Since these schemes are dependent on the accuracy of CHOKe hits, CHOKe misses and CHOKe-RED drops/marks, we analyze all three of them using NS2, verify their accuracy and provide suggestions. We also present two enlightening ad hoc variations of CHOKe called Half1- and Half2-CHOKe based on our analysis to show that CHOKe should adapt to the rate of flows. Both of these ad hoc schemes, like CHOKe, use a single selection from among the queued packets for comparison with each new packet that is arriving. Our results will show that RED, CHOKe and xCHOKe are limited in what they can achieve since malicious flows still get significantly more than their fair share and non-malicious flows get mistakenly penalized. We also show the unreliability of RED, CHOKe and xCHOKe in protecting malicious flows. RECHOKe helps to isolate, control and punish malicious flows and it can be used with any active queue management scheme such as Random Early Detection (RED) and Receiver-Window Modification (RWM), an approach detailed in the next section.

1.4 RCUBE

In the previous section we have discussed how malicious and non-responsive flows can have a negative effect on TCP flows. These "non-TCP" flows do not reduce their rates even when their packets are dropped. Hence, over time, they dominate the network, choking and robbing bandwidth from TCP-friendly flows.

1.4.1 Motivation

Several schemes (as mentioned in the previous section) have been proposed to protect TCP-friendly flows from non-adaptive flows such as those from UDP, non TCP-friendly and malicious sources. However, these schemes either incur too much overhead or do not sufficiently protect TCP-friendly flows from non-adaptive flows. Hence, we need a better method.

1.4.2 Major Research Contribution

RCUBE (Receiver Window Modified Random Early Detection queues with RECHOKe) combines the advantages of both Receiver Window Modification (RWM) and RECHOKe schemes. RECHOKe helps to isolate, control and punishes malicious flows when used with active queue management schemes such as Random Early Detection (RED) [7], Stabilized RED (SRED) [17], Adaptive RED (ARED) [19] and BLUE [20]. The RWM active queue management scheme, which can be implemented with other queue management schemes such as RED, BLUE, ARED, does not require modification to all end system TCP/IP stacks but can be solely implemented in heavily-congested ingress and gateway routers. By using RECHOKe, the RWM scheme can be implemented anywhere in the network without any modification to all end system

TCP/IP stacks unlike Explicit Congestion Notification (ECN) [5][6]. We will show that by reducing the average queue sizes, RWM queues will be able to reduce the queuing delay resulting in significant improvements in one-way end-to-end packet delays, delay jitter, throughput and number of dropped packets.

1.5 Dissertation Organization

The rest of this dissertation is organized as follows: In Chapter 2, we present a thorough description of previous work as found in the research literature relating to our research. In Chapter 3, we introduce the Receiver Window Modification (RWM) scheme and present an extensive simulation based analysis, evaluating and comparing AQM algorithms with and without ECN, and with our proposed RWM scheme. In Chapter 4, we present a mathematical time-difference model for RWM and use that model to perform asymptotic and steady-state analysis, for an RED-RWM gateway with *N* TCP connections competing for bandwidth. Chapters 5 and 6 present the RECHOKe and RCUBE schemes respectively together with experimental results and analysis supporting our claims from the previous sections. Finally, in Chapter 7, we summarize the contributions of the Ph.D. dissertation and discuss future research directions.

CHAPTER 2

BACKGROUND

This chapter provides an overview of the various major topics involved in active queue management research. Sections 2.1 through 2.6 provide an insight into the Transmission Control Protocol's (TCP) congestion control mechanisms, Conventional Queue Management Techniques, Random Early Detection (RED) AQM, Adaptive Random Early Detection (ARED) AQM, BLUE AQM, and Explicit Congestion Notification (ECN) respectively, while Section 2.7 provides an overview of Rate Control/Enforcement techniques. These sections provide relevant background information for RWM (as described in Chapter 3) and its model (as described in Chapter 4). Sections 2.8, 2.9 and 2.10 discuss the three main preferential dropping schemes proposed in network literature to identify, control and punish malicious flows, i.e., CHOKe, RED-Preferential Drop (RED-PD), and xCHOKe respectively.

2.1 TCP Congestion Control

As outlined in the previous sections current TCP implementations [8][9] contain a number of algorithms aimed at controlling network congestion. The algorithms used by TCP include among others SS (Slow Start), CA (Congestion Avoidance), fast-retransmit, and fast recovery. For the purpose of this paper, only the SS and CA algorithms are investigated. The CA and SS algorithms are meant to work together.

The CA algorithm has a variable called *ssthresh* determining the actual SS threshold size. At the start of a transmission, the congestion window - *cwnd* is initialized to one segment while *ssthresh* to 64Kbytes. When congestion occurs (indicated by timeout or reception of duplicate ACKs), half of the current window size (the minimum of *cwnd* and the receiver's advertised window, but at least two segments) is saved in *ssthresh*. Additionally, if the congestion is indicated by a timeout, *cwnd* is set to one segment size. When new data is acknowledged by the other end *cwnd* is increased depending on whether SS or CA is being performed. If the *cwnd* is less or equal to *ssthresh*, then SS is activated otherwise CA continues. SS lasts until it reaches the *ssthresh*, and then CA takes over. SS increments the cwnd by one segment, every time an ACK is received (thus increasing the *cwnd* approximately binary exponentially within a roundtrip time). On the other hand, CA dictates *cwnd* to be incremented by *1/cwnd* each time an ACK is received (thus increasing the *cwnd* by approximately one segment size in each round trip delay). Thus, packet drops are important indication of congestion for the TCP layer to decrease the transmission rate of flows. The reader is referred to [8] for a more detailed discussion.

2.2 Conventional Queue Management Techniques

In the Internet routers are responsible to store and forward datagrams containing segments of flow. Race conditions inside routers generated by their switching core's speeds, or by the bandwidth of ports require datagrams to be buffered at the input and/or the output ports. These buffers may filled up as their temporal arrival processes are faster than their service process and thus can force datagrams in the buffer or potential datagrams to be dropped. Initial congestion queue management research has focused on Tail Drop (Drop Tail)[18], Drop-from-front [43], Random Drop [7] and Early Random Drop [18] [43] techniques.

In *Drop Tail* queues, packets arriving after the queue becomes saturated are dropped. Packets will continue to be dropped until buffer space becomes available in the queue. The advantages of this type of QM are that it is simple, fast and easy to implement. However Drop Tail queues have the following drawbacks:

1) congestion control is not invoked until queues become saturated, thus congestion is not signaled to sources (which are still increasing their transmission rate) until queues are full;

2) global synchronization, i.e., it does not allow maximal utilization of link capacity because of synchronous oscillation of TCP transmission rates of several flows;

3) there is no control over which packets are dropped;

4) there is a fairness problems since a sudden traffic burst from one source may fill up all the available buffer space and as a consequence:

5) Drop Tail suffers from the lockout phenomenon whereby a single flow may not be able to get any packets through, if a window of segments arrives just when the queue becomes saturated.

To solve the fairness problem, the queue could be subdivided into many sub-queues with different service classes. However, this will lead to packets being discarded even if there was available buffer space in the queue.

In *Drop-from-front* queues, congestion is signaled earlier to the TCP sources by dropping packets at the front of the queue when it becomes saturated, resulting in faster invocation of congestion control action by TCP. However Drop-from-front introduces a major fairness-flaw [43].

In *Random Drop*, the packet to be dropped is uniform randomly selected, as the queue becomes saturated. This can alleviate global synchronization since it is highly unlikely that it will drop packets from most or all of the flows at the same time, as was in the case of the Drop Tail. It also prevents the lockout problem since the dropping probability of packets belonging to the same flow is proportional to the bandwidth consumption of the flow.

One of the first solutions that reacted to congestion before the queue became saturated is the *Early Random Drop* (ERD) mechanism. This scheme defines a queue length threshold value at some fraction of the total available queue size. Segments arriving after the queue size has reached this threshold will be randomly dropped (with a fixed probability). The disadvantage of this scheme is that it does not guarantee a fair treatment between flows nor does it guarantee that the contents of the queue will reflect the flow distribution of different sources. Moreover, ERD is bias against large bursts [44] as it favors flows with small amounts of segments.

2.3 Random Early Detection

Random Early Detection (RED) [7] prevents global synchronization by sacrificing a specific data flow whenever the average queue length increases indicating that overflow may occur soon. RED picks among flows fairly, discards a segment, forcing the flow into congestion avoidance and hence, decreasing its aggregate transmission rate. The probability that the gateway chooses a particular flow to mark or drop during congestion is roughly proportional to that flow's share of the bandwidth at the router. In short, it is valid to say that if a flow has a large fraction of the recently dropped packets, then it has also most likely received a large portion of the recent bandwidth. RED also avoids biasness against bursty traffic. However, despite these advantages, there are some well-documented problems with RED [17][19][21][22]. The level of congestion with its immediate links (neighbors), compounded with its complex parameter tuning [22] induces large variations in the queue size. The queue size is near Th_{min}, whenever the link is lightly congested or the P_{max} is high. However, the queue size increases to around Th_{max} with an increase in the congestion levels or if P_{max} is set to a low value. Both these situations result in a degraded throughput, i.e., RED may even underperform a Drop-Tail mechanism [10]. RED may also introduce jitter into non-bursty streams. All these disadvantages have generated more research using RED as a starting point for further improvements [14][19][20].

There are two phases to the RED algorithm; the estimation of average queue size and the decision on which packet to drop (see pseudo-codes in figures 2.1 and 2.2 respectively). The purpose of RED is to make the network more capable of accommodating bursty traffic rather than shaping bursty traffic to accommodate the needs of the network. It uses FIFO scheduling due to FIFO’s low overhead, lack of scaling problems, and reduction of weight of the tail of the delay distribution.

```
For each packet arrival
    If the queue is not empty
        $avq = (1- w_q)\ avq + w_q q$
    Else using a table lookup
        $avq = (1- w_q)^{(\text{time-}q\_time)/\text{s}}\ avq$
Where
        $avq$: average queue size
        $w_q$: queue size
        q: current queue size
        s: typical transmission time
        time: current time
        $q\_time$: start of the queue idle time
```

Figure 2.1 Average queue size estimation algorithm

In RED, if the average queue length is smaller than Th_{min}, then the arriving segment is accepted. If the average queue length is between Th_{min} and Th_{max}, then the segment is dropped with probability P_a. P_a is a linear function of the average queue length, possibly increasing to a pre-determined maximum value P_{max}. However, if the calculated average queue length is larger than Th_{max}, then incoming segments are dropped with probability one. This behavior results in a larger virtual queue size for short-term bursts compared to long-term bursts. As the average queue length (*avq*) varies from Th_{min} to Th_{max}, the packet-marking probability P_a varies linearly from 0 to P_{max} with $P_a = P_{max} * (avq - Th_{min})/(Th_{max} - Th_{min})$.

```
For each packet arrival
    Calculate $avq$
    If $Th_{min} \leq avq < Th_{max}$
        Calculate $P_a$
        Mark/drop the arriving packet with $P_a$
    Else if $Th_{max} < avq$
        Drop the arriving packet
    Else
        Accept the arriving packet
Where
     $avq$: average queue size
     $Th_{min}$: minimum queue threshold
     $Th_{max}$: maximum queue threshold
     $P_a$: marking/dropping probability
```

Figure 2.2 Dropped packet selection algorithm

There are a number of variations of RED, such as Adaptive RED (ARED) [7], Random Exponential Marking (REM) [14], BLUE [20] and RED with Preferential

Dropping (RED-PD) [37]. We are going to briefly outline ARED and BLUE in subsequent section as they are the most advanced AQM schemes available today.

2.4 Adaptive Random Early Detection

Adaptive RED (ARED) [19] is one of the many versions of RED. It is geared to achieve a predefined target queue length and hence lower packet loss and minimum variance in queuing delay. More importantly, it aims to solve the complex parameter setting problem in RED.

In ARED, P_{max} is not constant but is changed (adapted) at half-second intervals to keep the average queue around $(Th_{min} + Th_{max})/2$. The pseudo code of P_{max} adaptation is shown in figure 2.3 (an additive increase multiplicative decrease (AIMD) technique is used for adapting P_{max}).

2.5 BLUE

The BLUE active queue management algorithm [20] uses packet loss and link utilization history to manage congestion by detecting and adjusting the rate of packets being dropped or marked. It uses the probability, P_a, to mark (by using ECN) or drop queued packets. P_a is increased whenever packets are dropped from the queue and decreased when the link is underutilized.

The factor of increase to P_a is denoted by δ_1 while δ_2 represents the factor of decrease. The update on P_a takes place, whenever the queue length exceeds a certain threshold *L,* at the rate of 1/*freeze_time*. The parameter *freeze_time* represents the time interval between successive updates on P_a. Figure 2.4 shows a sample pseudo code of the BLUE algorithm.

```
Execute the following at every interval of 0.5 seconds:
    If ($avq$ > target and $P_{max} < 0.5$)
        $P_{max} = P_{max} + \alpha$
    Else if ($avq$ < target and $P_{max} > 0.01$)
        $P_{max} = P_{max} * \beta$
Where
    $avq$: average queue size
    target: desired queue size
    $P_{max}$: maximum probability
    $\alpha$: min(0.01, $P_{max}/4$)
    $\beta$: 0.9
```

Figure 2.3 Algorithm to adapt P_{max}

2.6 Explicit Congestion Notification

The Explicit Congestion Notification (ECN) [5][6] mechanism is used to notify the sender TCP process about congestion, ahead of time thus preventing unnecessary packet drops. In case of an ECN modified RED queue, if the average queue size is between Th_{min} and Th_{max} and the datagram is selected then it will be marked in its IP and TCP headers instead of being dropped. An ECN-compliant TCP source, on receiving the marked ACK packets (echoed from the ECN-compliant TCP receiver), will behave as if it had encountered a dropped packet. Just like in the case of detecting a packet drop, the response of the TCP source to the congestion should happen at most once per round trip time (RTT).

```
Upon packet loss:
    If ((now-last_update) > freeze_time) then
        $P_a := P_a + \delta 1$
        Last_update = now
Upon link idle:
    If ((now-last_update) > freeze_time) then
        $P_a := P_a - \delta 2$
        Last_update = now
Where
        $P_a$: marking/dropping probability
        $\delta 1$:  amount of increase by $P_a$
        $\delta 2$:  amount of decrease by $P_a$
        now: current time
        last_update: last time $P_a$ was changed
        freeze_time: amount of time for $P_a$ to take effect
```

Figure 2.4 BLUE algorithm

ECN avoids unnecessary packet drops and accelerates the detection of congestion by the source, without having to wait to detect a dropped packet (a timeout from the transmit timer or on receiving three duplicate ACKs). However, as a downside ECN messages may get lost, and TCP implementations at both the source and the destination have to be ECN-compliant (which presents a significant problem in today's implementations). Currently there is no practical benefit in setting ECN bits in Internet packets; as this requires ECN capable routers (at least at bottleneck points), servers and clients throughout a network. In [45], experiments were conducted using TBIT (the TCP Behavior Inference Tool) showing that on the 13th of September, 2000, 21 out of 26447 (0.07%) sites responded positively with an appropriate SYN/ACK to ECN. In March 2002, only 7 out of 12364 sites (0.05%) responded positively. These tests included end systems running a representative distribution of nearly all types of operating systems.

A TCP sender reacts to an ECN flagged segment by halving both the congestion window *cwnd* and the slow-start threshold *ssthresh* at most once per RTT, while ignoring succeeding ECNs within that particular RTT. This is to ensure that the TCP source does not reduce its window repeatedly within that particular RTT. However, when the ECN-compliant TCP sender receives three duplicate ACKs without any ECNs, it will undergo Fast Retransmit and Fast Recovery procedures (see TCP Reno [9]). On the other hand, if the ECN-compliant TCP sender receives three duplicate ACKs after a reaction to an ECN notification within the RTT, it will not react since it has already reacted to the ECN notification.

To facilitate the ECN scheme, TCP uses a 2-bit ECN field [6] in the IP header (as shown in figure 2.5), with a possibility of marking four ECN codepoints from ‘00’ to ‘11’. Bit sequence ‘11’ is known as the Congestion Experienced (CE) codepoint which is used to indicate congestion to the end nodes. Codepoints ‘10’ and ‘01’, (also known as ECT(0) and ECT(1) respectively,) are used by the TCP sender to denote that it is ECN-capable. An ECN-compliant sender will use both of these codepoints to ensure that the routers along the path are not erasing the CE codepoint and that an ECN-compliant receiver is not erroneously reporting to the ECN-compliant sender the reception of packets with CE codepoints. Routers (that are ECN compliant) along the path treat both of these codepoints similarly. The ‘00’ codepoint (default assignment in any IP datagram) indicates that the packet is not using ECN.

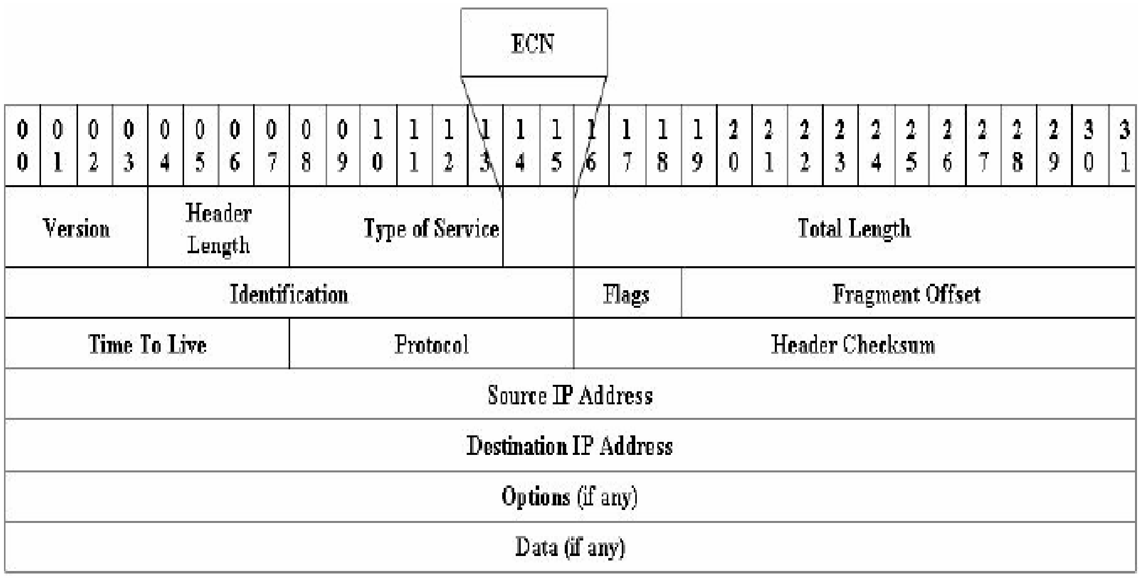

Figure 2.5 ECN bit in IP header

The TCP header will also have to contain information for ECN. Flags: Congestion Window Reduced (CWR) and ECN-Echo (ECE) are introduced into the TCP header as shown in figure 2.6. ECE is used for negotiating ECN-capability between TCP senders and receiver, whereas the CWR is used by the TCP sender to inform the TCP receiver that the congestion window has been reduced (so that the TCP receiver can determine when to stop setting the ECN-Echo flag).

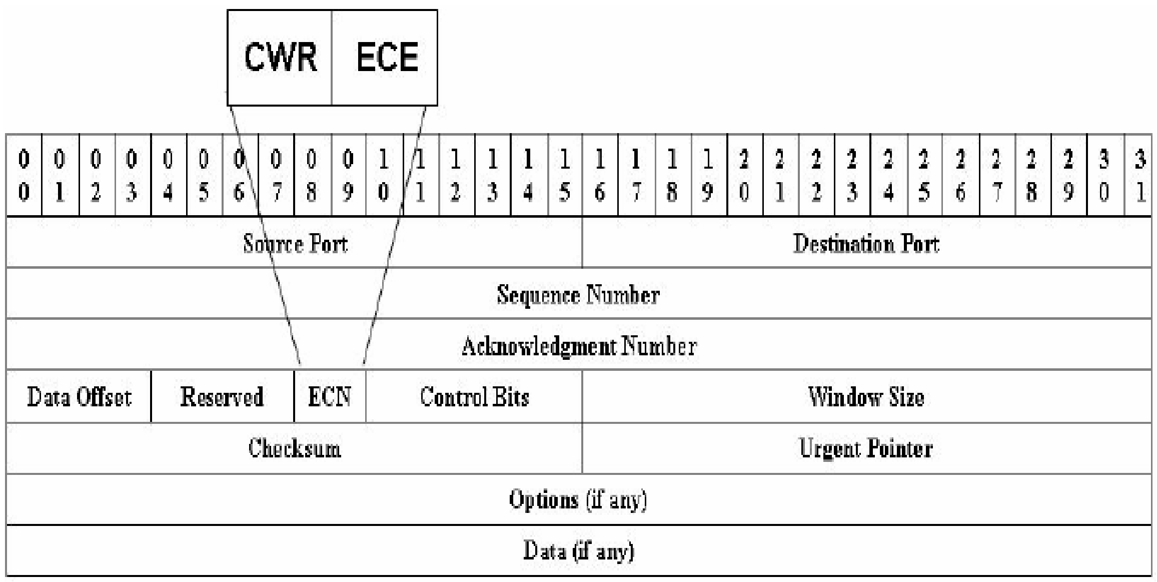

Figure 2.6 The CWR and ECE bits

A typical sequence of events is as follows. After positive negotiation between the sender and the receiver about using ECN, the sender sets the ECN-Capable Transport (ECT) codepoint in the packet's IP header. When an ECN-capable router detects congestion, it selects a packet. If the packet had its ECT codepoint set, instead of dropping it, the router sets the packet's Congestion Experienced (CE) codepoint in its IP header, to indicate to its sender of the impending congestion. On receiving this packet, the TCP receiver sets the ECN-Echo (ECE) flag in its next ACK packet to the TCP sender. When the TCP sender receives this ACK packet, it will react as if it had encountered a packet dropped. It will also set the Congestion Window Reduced (CWR) flag in the TCP header of the next packet it sends to the receiver acknowledging the reception of the ACK with its ECE flag set (and that it had reacted to the impending congestion).

2.7 Rate Control/Enforcement Techniques

As outlined in the first chapter, a TCP stream may either be constrained by flow or congestion control (i.e., whichever window is smaller). Flow control compared to congestion control is a direct feedback process where the TCP receiver tells the TCP sender explicitly what its transmission window's maximum value should be. Modifying the receiver window (the corresponding TCP header field) at intermediate routers can thus provide a mean to set TCP senders' transmission window explicitly[1]. To improve overall communication latency an intermediate network element may modify the receiver's advertised window in the returned TCP acknowledgments. This rate control or enforcement technique has been used in [46][47][48][49] and by Packeteer, Inc. [50] in 1995. This technique is referred to as Explicit Window Adaptation in [46], where the objective is to match the sum of the windows of active TCP connections sharing the buffer in the edge router to the effective network bandwidth delay product, thus

[1] Some researchers claim that routers in general should not look at layer-4 headers as their function is limited to layer-3. Yet, enabling them to do so can provide significant benefits (see, e.g., NAT).

avoiding packet losses whenever possible. Hence, the receiver's advertised windows in TCP acknowledgments are modified at the edge router. The feedback loop in their scheme extends only from the ATM Access Points (AAPs) to the sources of the TCP connections. In [51], Karandikar et al use a similar technique, as part of their TCP Rate Control, to achieve high throughput, fairness and low delay. They calculate the rate feedback to fit ATMs' Available Bit Rate (ABR) service [52][53][54]; and then translate the rate into a window value to give feedback to the TCP source.

Narvaez et al in [48] extend [54] by defining a complex TCP emulation engine at the rate-controller using an acknowledgment bucket scheme In [49], Satyavolu et al suggest a rate-to-window translation scheme based upon another ATM ABR algorithm, ERICA [53]. All these authors explored such ideas from the point of view of TCP/IP-ATM internetworking, i.e. extending ATM ABR type rate control from ATM edges to TCP end systems.

2.8 CHOKe

CHOKe [33] has been proposed to punish non-responsive, i.e. non TCP-friendly, flows. It does so by calculating the average occupancy of the FIFO buffer in a manner similar to RED, i.e., by using an exponential moving average - *avq*. Hence, in that context it can be said that CHOKe is an extension to RED. It uses the RED's thresholds, Th_{min} and Th_{max}, to monitor the *avq*. If the *avq* is smaller than Th_{min}, then the arriving segment is accepted. If the *avq* is greater than Th_{min}, each arriving packet is compared with a randomly selected packet already in the queue, called *drop candidate packet*, from the FIFO buffer. If they have the same flow ID, they are both dropped in which case a CHOKe *hit* has taken place (otherwise it is a CHOKe *miss*). When a CHOKe *miss* happens, the arriving packet is dropped with a probability of P_a; as in RED, P_a is a linear function of the average queue length, possibly increasing to P_{max}. Figure 2.7 shows a simple flowchart of CHOKe.

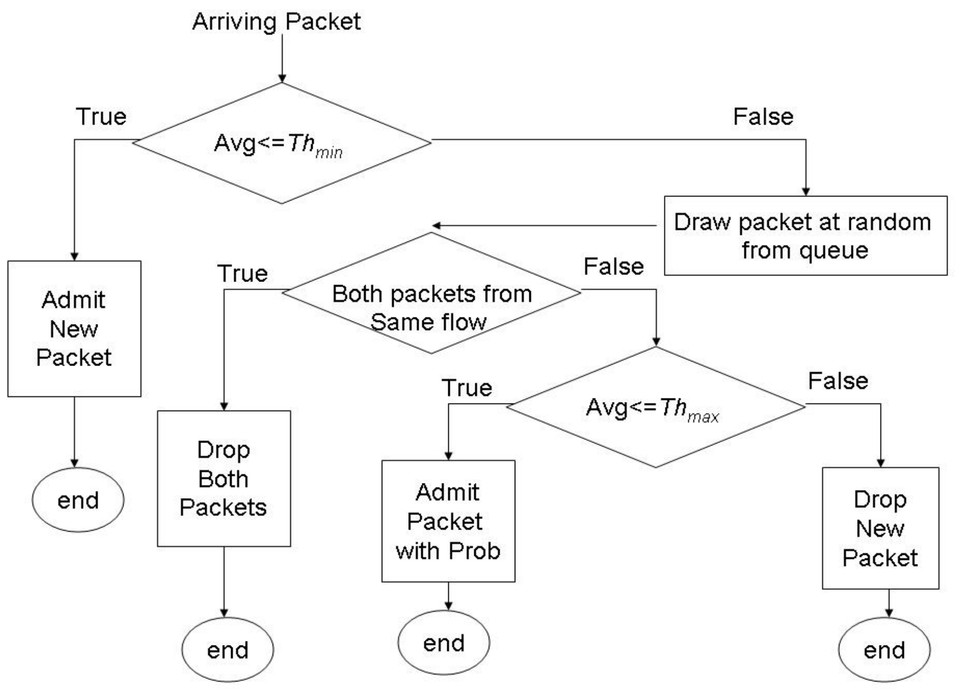

Figure 2.7 Flowchart of CHOKe

There have been several CHOKe variants proposed in the literature:

- Multi-drop CHOKe (M-CHOKe) [33]: Here, instead of one drop candidate packet, *m* candidate packets are chosen and compared with each arriving packet, thus improving the chances of getting hits. The disadvantage of this scheme is that it might punish responsive flows at a greater extent as well.

- Adaptive CHOKe (A-CHOKe) [33]: Here, the buffer between the two thresholds is partitioned into *k* regions and *m* is automatically set to $2i$ $(i = 1, 2, \ldots, k)$.
- Front CHOKe [33]: The drop candidate packet is chosen from the head of the queue.
- Back CHOKe [33]: The drop candidate packet is chosen from the tail of the queue.
- CHOKe+ [34]: aims to solve the high-drop rate of CHOKe; its pseudocode is depicted in figure 2.8.

```
For each incoming packet P
    Pick $k$ packets at random from queue
    Let $m$ be # of packets from the same flow as P
    Let $0 <= g_2 < g_1 <= 1$ be constants
    If $m > g_1 k$,
            P and the $m$ packets are dropped
    Else if $g_2 k <= m < g_1 k$,
            Drop P and the $m$ packets only if RED were to drop P
    Else
            Just drop P according to RED
```

Figure 2.8 CHOKe+ algorithm

2.9 RED-PD

By keeping partial flow state for high-bandwidth flows, RED-PD [37] detects and thwarts malicious flows by monitoring a subset of flows at the queue of the router and comparing their rates with a *targeted bandwidth.* If the rate of a flow is greater than the targeted bandwidth, then it is considered a malicious flow. This target bandwidth *f(r,p)*, above which a flow is identified, is defined as the bandwidth obtained by the *reference TCP* flow with the *target RTT r* and the current drop rate *p* at the output queue and is:

$$f(r,p) \approx \frac{\sqrt{1.5}}{r\sqrt{p}}$$

RED-PD keeps the drop history over *K * CL(R, p)* seconds, where *K* = 3 and *CL(R, p)* is given by

$$CL(r,p) = \frac{1}{f(r,p)p} = \frac{r}{\sqrt{1.5p}}$$

RED-PD keeps *M* lists as well, where a list is given by

$$\frac{K \times CL(r,p)}{M} = \frac{1}{Mf(r,p)p} = \frac{r}{M\sqrt{1.5p}}, \text{ where } M = 5.$$

By using the packet drop history at the router to detect high bandwidth flows in times of congestion, RED-PD drops packets from these flows preferentially. By restricting these high-bandwidth flows, it improves the performance of low-bandwidth flows. Figure 2.9 shows the architecture of a RED-PD router. The difficulty of using the RED-PD router in today's networks is that it is too time-consuming for the router to calculate the RTT of a flow since RTT is heavily dependent on the state of the network. In addition, predefining *r* is neither realistic nor easy.

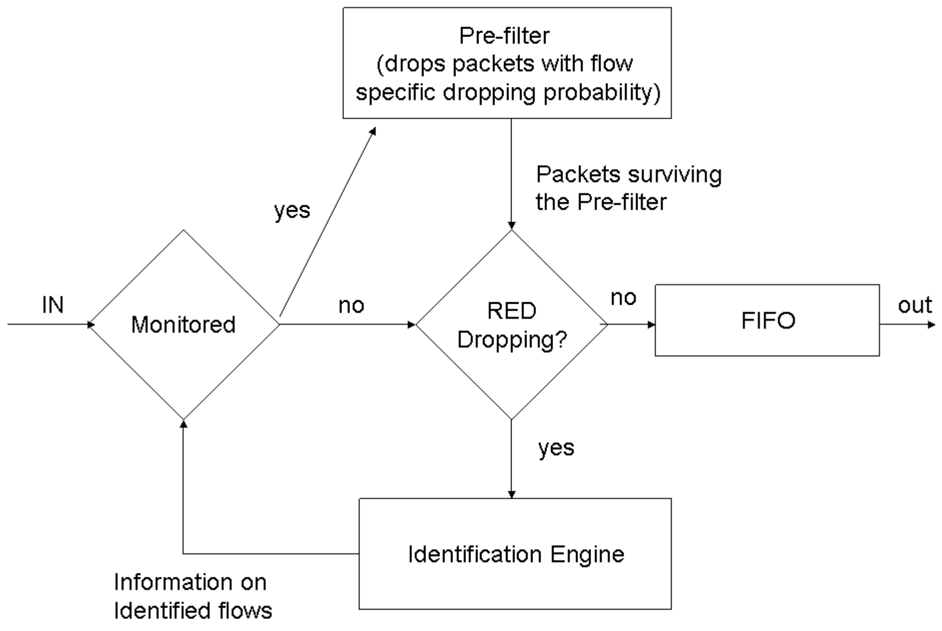
Pre-filter
(drops packets with flow specific dropping probability)
Packets surviving the Pre-filter
yes
IN
Monitored
no
RED Dropping?
no
FIFO
out
yes
Identification Engine
Information on Identified flows

Figure 2.9 The architecture of RED-PD

2.10 xCHOKe

xCHOKe [35][36], an extension to CHOKe and RED, uses partial state to identify malicious flows. It also uses the RED's thresholds, minimum threshold (Th_{min}) and maximum threshold (Th_{max}), to monitor the average queue length (*avq*).

xCHOKe works as follows: If the *avq* is smaller than Th_{min}, then the arriving segment is accepted. If the *avq* is greater than the Th_{min}, a lookup table is referenced to see whether the arriving packet's flow label is present in it. If it is indeed present (table *hit*), the arriving packet is dropped or marked for dropping. A packet is then selected at random from the queue and its flow label is compared with that of the arriving packet. If the flow labels are the same (CHOKe *hit*), both packets are dropped or marked for dropping and the flow label is added to the lookup table with an initial value of one for an associated hit counter. If the flow is already in the table, the associated hit counter is incremented. The hit counter n of an entry in the lookup table is used to compute the probability $p*$ with which an incoming packet with the same flow label as the entry will be dropped; $p*$ is computed as follows: $p^* = \min(1, p_a \times 2^n)$, where p_a is RED's dropping probability. If the *avq* is greater than Th_{max}, then the packet is dropped or marked for dropping. Figure 2.10 shows a flowchart of the xCHOKe scheme. xCHOKe associates a time-to-live (TTL) with each entry in the lookup table. Upon expiry of its TTL, an entry in the table is discarded. TTLs are not refreshed when an incoming packet has hits in the lookup table (table hits) nor if hits occur as an outcome to a comparison with a randomly chosen packet from the buffer (CHOKe *hits*).

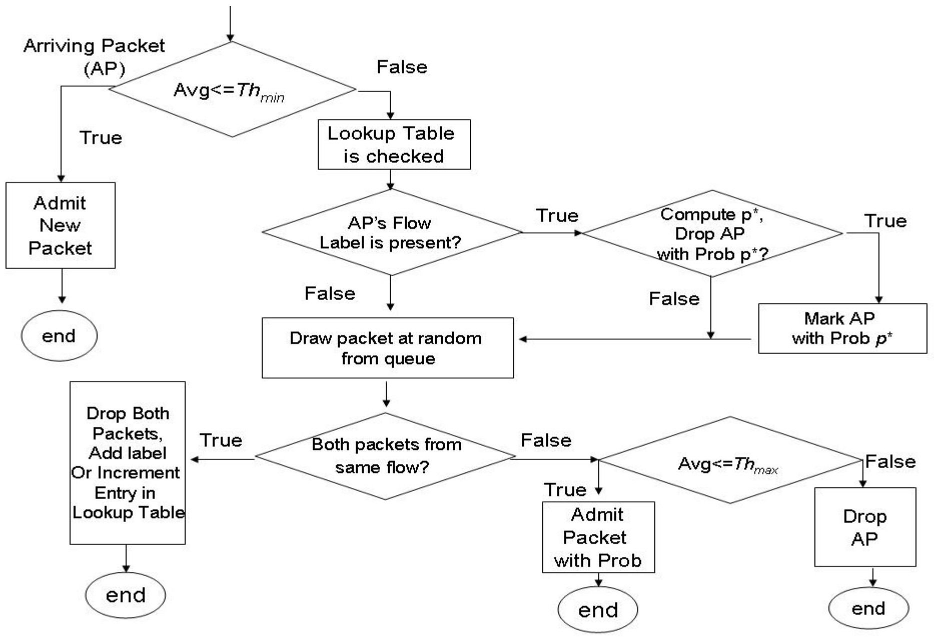

Figure 2.10 Flowchart of the xCHOKe

CHAPTER 3

RECEIVER-WINDOW MODIFICATION (RWM) SCHEME

In this chapter we present our solution to solve the first two problems as identified in Chapter 1, faced by active queue management (AQM) schemes. To refresh the reader's memory these problems are:

1) the congestion detection, i.e., the timeout mechanism or the duration of the reception of three duplicate acknowledgements (ACKs), due to early-dropped packets by AQMs, delays the response time of TCP in reducing the network congestion, and

2) The employment of ECN within AQMs has the following drawbacks: ECN messages may get delayed or worse, lost; and TCP implementations at both the source and the destination have to be ECN-compliant.

Our goals are to provide solutions that work with present-day TCP/IP networks and at the same time, reduce the average TCP queue sizes in the queues and in doing so, reduce the queuing delay resulting in significant improvements in one-way end-to-end packet delays, delay jitter, throughput and number of dropped packets for TCP-friendly flows. If deployed, this enhancement should only be needed at the routers. In this chapter, we present this enhancement as a solution to address the above-mentioned problems at the ingress/gateway routers, and call it the Receiver-Window Modification

(RWM) scheme. We provide a further enhancement to this scheme in Chapter 5 so that it can be deployed at any router within the TCP/IP network.

The rest of this chapter is organized as follows. The next section describes RWM while Section 3.2 presents simulation based analysis and comparisons of our RWM scheme to several proposed AQM schemes with and without ECN. Finally, Section 3.3 concludes the chapter by summarizing our results.

3.1 Description of RWM

AQM solutions use packet drops or marks at router queues to manipulate the TCP sender into decreasing its transmission rate. RED, ARED and BLUE queues prevent a queue in any intervening router from reaching its limit by dropping packets early. With ECN-compliant RED, ARED or BLUE queues, packets are not unnecessarily dropped but marked as if they were dropped. We denote these types of queues as RED-ECN, ARED-ECN and BLUE-ECN respectively.

Using ECN, a queue accelerates the detection of congestion by the source, without having to wait to detect a dropped packet (a timeout from the transmit timer or on receiving three duplicate ACKs). However, as a downside: i) ECN messages may get delayed by congested downstream routers or lost; and ii) TCP implementations at the router and both the source and the destination have to be ECN-compliant. Currently there is no practical benefit in setting ECN bits in Internet packets; as this requires ECN capable routers (at least at bottleneck points such as ingress and gateway routers), servers and clients throughout a network. In present-day TCP/IP networks, few servers and clients are “ECN-compliant”. Hence, “ECN-compliant” routers are not practical in present-day TCP/IP networks due to a lack of “ECN-compliant” servers and clients.

Moreover, AQM schemes have several drawbacks as outlined in the second chapter. Research has shown that RED may underperform even a Drop-Tail [10] queue. Our Receiver Window Modification (RWM) scheme alleviates these disadvantages of RED.

We are proposing to use both flow and congestion control feedbacks to reduce congestion at routers. In this research, we will deal with congestion occurring especially at the ingress and gateway routers – two major congestion areas within the network. Instead of setting ECN related bit in the chosen packets of RED, ARED and BLUE queues, we propose the use of RWM at these routers. RWM sets the receiver window field to one maximum segment size (MSS) in the ACK packets that are going towards the sender from the receiver, instead of early-dropping or marking the packets at the queue. (The only exception that it will not modify the receiver window field is when the field has a value of 0. This occurs when a TCP application wants to tell its peer not to send any more data). We denote RWM modification to AQM schemes by affixing “RWM” after the name of the AQM, thus we will be talking about RED-RWM, ARED-RWM and BLUE-RWM respectively.

In the case of RED-RWM and ARED-RWM, the field is set to 1, instead of early-dropping or marking the packets, only if the average queue length (*avq*) is between Th_{min} and Th_{max}, where Th_{min} and Th_{max}, are the minimum and maximum threshold values respectively. However, if the *avq* is greater than Th_{max}, the packet is not only dropped but the field in the ACK is also set to 1. The idea here is to reduce the rate as soon as possible without waiting for the timeout mechanism or the duration of the reception of three duplicate ACKs to take effect or elapse, thus cutting down on the number of packets that would have otherwise been dropped. The threshold values used are the same as in RED and ARED queues.

In the case of RED-RWM and ARED-RWM, the recommended P_{max} ranges are bounded from 0.3 to 1 respectively. For example, the RED algorithm (as shown in figure 2.2), is modified to what is shown in figure 3.1.

In the case of BLUE-RWM, the probability, P_a is used (instead of marking (by using ECN) or early-dropping of queued packets (without ECN)) to modify the ACK packets. P_a is increased whenever ACK packets are modified or/and data packets are dropped from the queue due to overflow and decreased when the link is underutilized. If the queue is continually dropping packets due to buffer overflow, BLUE-RWM increments P_a, thus increasing the rate at which it modifies the field to 1. Conversely, if the queue becomes empty or if the link is idle, BLUE-RWM decreases P_a, thereby decreasing the rate of ACK modification. This effectively allows BLUE-RWM to "learn" the correct rate it needs to modify the field to. The *freeze_time* is randomized to avoid global synchronization.

```
For each packet arrival
    Calculate avq
    If $Th_{min} \le$ avq $< Th_{max}$
        Calculate $P_a$
        Set receiver window field in ACKs to 1
            MSS with $P_a$
    Else if $Th_{max} <$ avq
        Drop the arriving packet
    Else
        Accept the arriving packet
Where
    avq: average queue size
    $Th_{min}$ : minimum queue threshold
    $Th_{max}$: maximum queue threshold
    $P_a$: marking/dropping probability
```

Figure 3.1 Dropped packet selection algorithm

Upon receiving a modified ACK packet, the sender will transmit the minimum of the congestion and the advertised received window sizes. (Whenever the TCP header of an ACK packet is modified, the checksum in the TCP header needs to be adjusted for error control.) The advantages of RWM queues are that they work with the current implementation of TCP in end systems and do not require changes to the existing network schemes except at the ingress and gateway routers where the RWM scheme is to be implemented. Hence, they overcome the disadvantage of the ECN queues since in ECN; TCP/IP stacks at both ends require modifications to be “ECN-compliant” in addition to the router. Since, the feedback loop extends from the RWM queue on the router to the sender; the response time of the algorithm is determined by the delay between the router and the sender rather than the RTT of the connection, which is true for queues using the ECN mechanism. This implies that the longer the RTT is, the greater the advantage that RWM has over ECN queues. The delayed response of the sender to an ECN is further compounded if the congestion worsens at the downstream routers where ECN messages could be delayed or worse, dropped. Moreover, the response time will also be heavily dependent on the types of queue implementations at these routers. Presently, almost all the routers along a path use drop tail queues. We will show that using RED, ARED, BLUE AQM, and drop tail queues will only increase the response time when compared to RWM. The queues using the RWM mechanism enjoy the advantage of faster response times to the impending congestion since the notification of congestion arrives at the source quicker when compared to those using timeout or ECN mechanisms.

In the next section, we model a small TCP/IP network by an NS2 simulator and show extensive results to support our claims.

3.2 Simulation Analysis of RWM

We have conducted an extensive set of simulation experiments to evaluate and compare AQM algorithms with and without ECN to the proposed RWM. Our simulations were based on an extended and corrected (the receiver window – congestion window interaction in NS2 does not follow common, linux-type TCP implementations) NS2 simulator. The subsequent subsections explain the details and discuss the results of the different simulation scenarios. We compare AQM schemes on the simple network topology of 4 sources connected to a sink via 4 routers as shown in figure 3.2; the nearest router to the sources employs the AQM to be investigated while the other 3 routers contain Drop Tail queues which are kept large enough not to discard any packets. We use TCP Reno as most modern TCP implementations are "Reno" based.

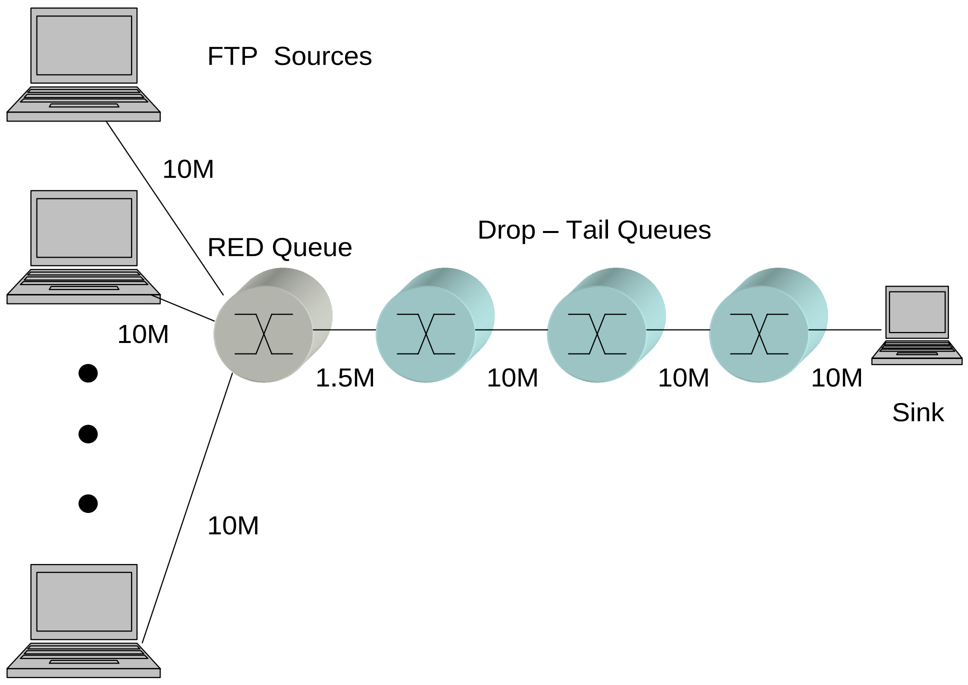

Figure 3.2 Simulation network topology

3.2.1 Scenario 1 – RED AQM

We first study the performance of TCP with RED, RED-ECN and RED-RWM. We start four File Transfer Protocol (FTP) connections and monitor them for the next 150 seconds; the size of the RED queue is set to accommodate 35 packets. Other RED parameters were set as follows:

- Queue weight given to current queue size sample = 0.002
- Minimum threshold of average queue size (Th_{min}) = 5
- Maximum threshold of average queue size (Th_{max}) = 15
- Max probability of dropping a packet = 1/*linterm*, where *linterm* = 3

Figure 3.3 depicts the average queue size versus the simulation time of the investigated AQMs. RED-RWM reduces the average queue size by about 25% compared to RED and RED-ECN. In the case of the RED-ECN queues, the larger queue sizes are partly due to marked packets suffering congestion along the path, increasing the delay along their way to the destination and then being echoed back to the source via ACK packets. On the other hand, the feedback loop for the RED-RWM is much shorter and hence, the average queue sizes are closer to the Th_{min} of average queue size of 5 packets. The duration of the reception of three duplicate acknowledgements (ACKs), due to early-dropped packets as in the case of RED delayed the response time of the TCP congestion scheme causing the larger average queue sizes. The same observation was made when we had used timeouts in our experiments.

Keeping the same topology and RED parameters but varying the number of sources, we investigated the performance of RED, RED-ECN and RED-RWM by collecting essential data and analyzing them in terms of

- delay jitter (the variation in the time taken for packets to be transmitted from the source to the destination in a network.),
- average packet delay,
- number of packets dropped, and
- throughput.

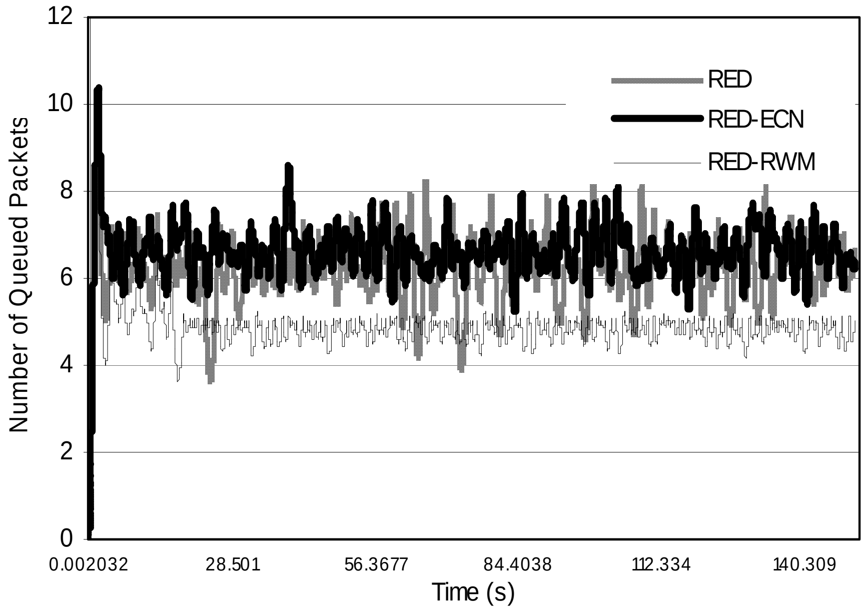

Figure 3.3 Average queue sizes for RED, RED-ECN and RED-RWM

Figures 3.4, 3.5, 3.6 and 3.7 show the respective results obtained for 1000 seconds of simulation time.

Larger average queue sizes mean greater buffering delay. When the buffering delay is increased, the corresponding round-trip times increase and cause the aggregate TCP behavior to be less aggressive. Likewise, when the buffering delay is decreased, the corresponding round-trip times decrease and cause the aggregate TCP behavior to be more aggressive. Hence, smaller average queue sizes imply smaller queuing delay and as a result, there is an improvement in the average one-way average packet delay with RED-RWM as compared to those obtained with RED-ECN and RED queues, as can be seen in figure 3.4. Since its average queue length is the least, the average packet delay of RED-RWM is 15-20% less than that of RED-ECN and RED. Moreover, RWM significantly and consistently reduces the variation of the delay by about 35% (as shown in figures 3.3 and 3.5); the variation in the average queue length is the least in RWM. RWM outperforms its counterparts in terms of overall throughput (as shown in figure 3.6) and total packet drops (as shown in figure 3.7) as well. Since it modifies instead of early dropping, RED-RWM drops less packets than RED whereas the disadvantages of ECN, especially its dependence on the performance of downstream routers, leads to RED-ECN dropping more packets than RED-RWM. Similar results were also obtained with ARED (as shown in Section 3.3.2) and BLUE (as shown in Section 3.3.3).

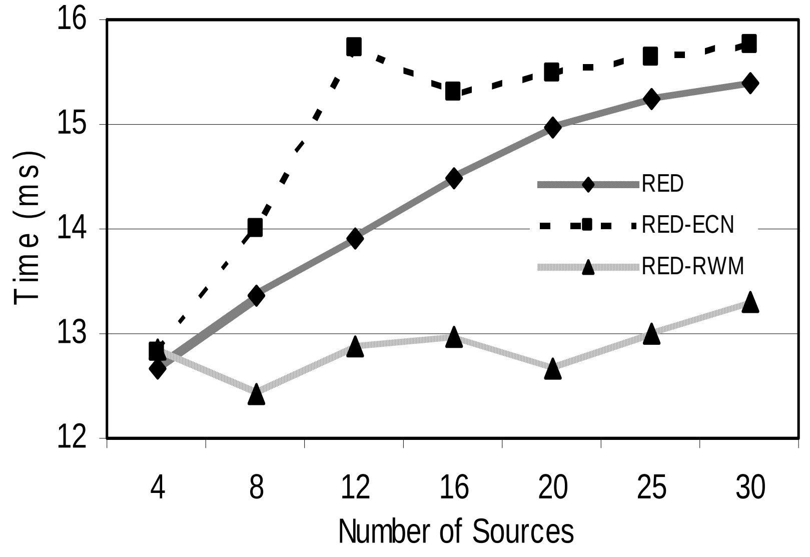

Figure 3.4 Average packet delay for RED, RED-ECN and RED-RWM

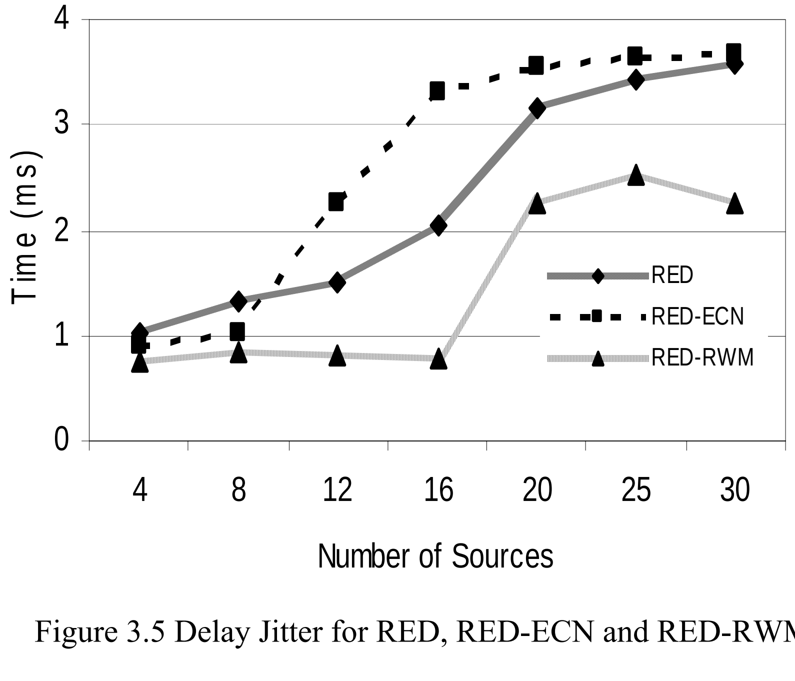

Figure 3.5 Delay Jitter for RED, RED-ECN and RED-RWM

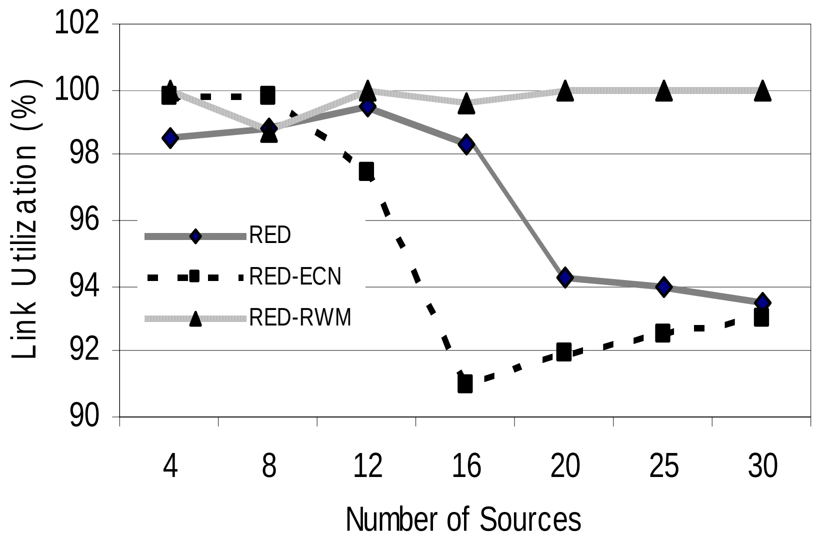

Figure 3.6 Throughput for RED, RED-ECN and RED-RWM

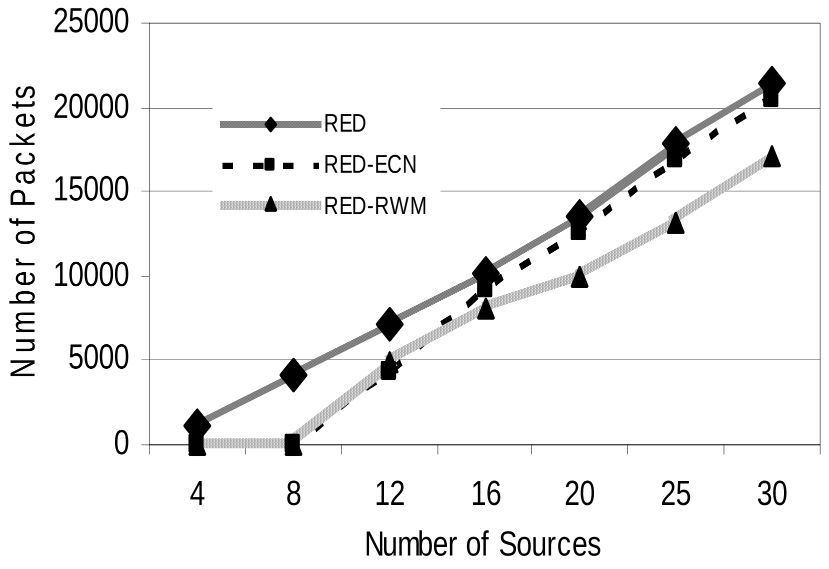

Figure 3.7 Total packet drops for RED, RED-ECN and RED-RWM

3.2.2 Scenario 2 – ARED AQM

Here, we investigate the performance of ARED, ARED-ECN and ARED-RWM queues with the same network topology and RED parameters as in our previous scenario. The average queue sizes for ARED, ARED-ECN and ARED-RWM are compared with each other. Generally, it can be observed that the average queue size for ARED-RWM is smaller than those of ARED and ARED-ECN (as shown in figure 3.8). (Note that the average queue length curves of ARED and ARED-ECN are comparable and semi-overlapping). The reasons for the variation between RWM, ARED and ARED-ECN are the same as they were in the previous scenario. It is due to faster response times to the impending congestions since the notification of congestion for ARED-RWM queue arrives at the source faster when compared to the ECN (in the case of ARED-ECN) mechanisms and the response time for the three duplicate ACKs (in the case of ARED). Note, that similarly to RED-ECN (as shown in Section 3.3.1) and BLUE-ECN (as shown in Section 3.3.3), in ARED-ECN, the notification is delayed by all of the queuing delays of the downstream routers. Hence, the average queue lengths for ARED-RWM are about 50% smaller compared to those of ARED-ECN and ARED.

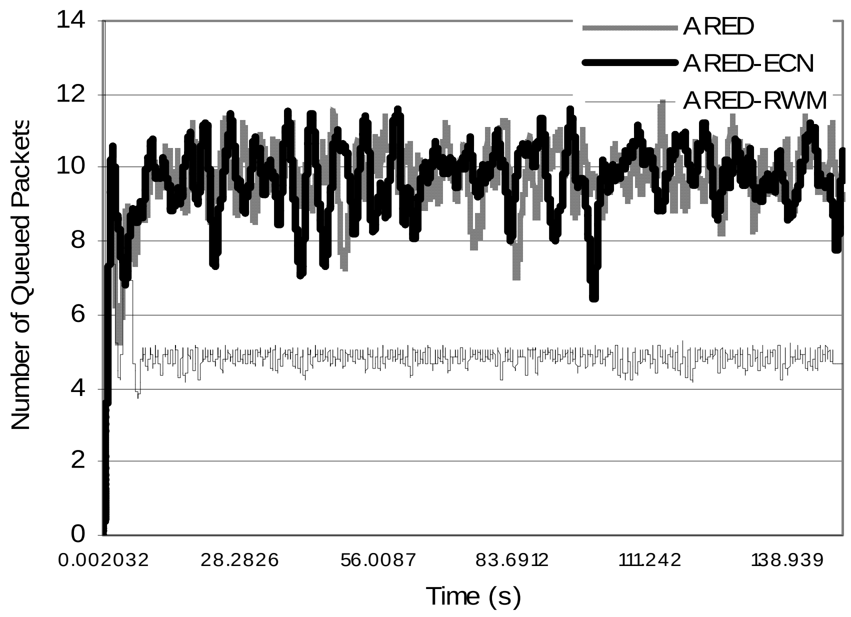

Figure 3.8 Average queue sizes for ARED, ARED-ECN and ARED-RWM

Keeping the same topology and RED parameters but varying the number of sources (increasing the duration of the experiments to 1000s), ARED, ARED-ECN and ARED-RWM queues were compared based on their one-way average packet delay, delay jitter, number of packet drops and throughput. Figures 3.9, 3.10, 3.11 and 3.12 show the results that confirm that ARED-RWM outperforms ARED and ARED-ECN in all of these categories.

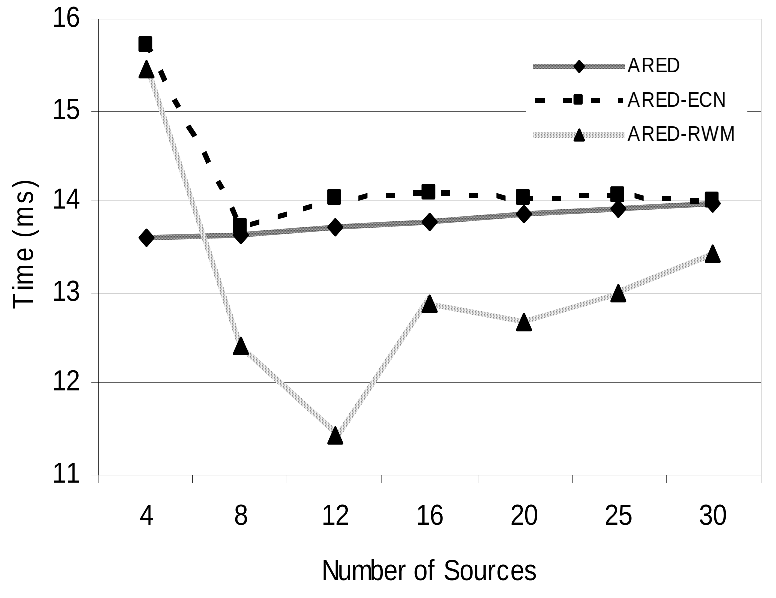

Figure 3.9 Average packet delay for ARED, ARED-ECN and ARED-RWM

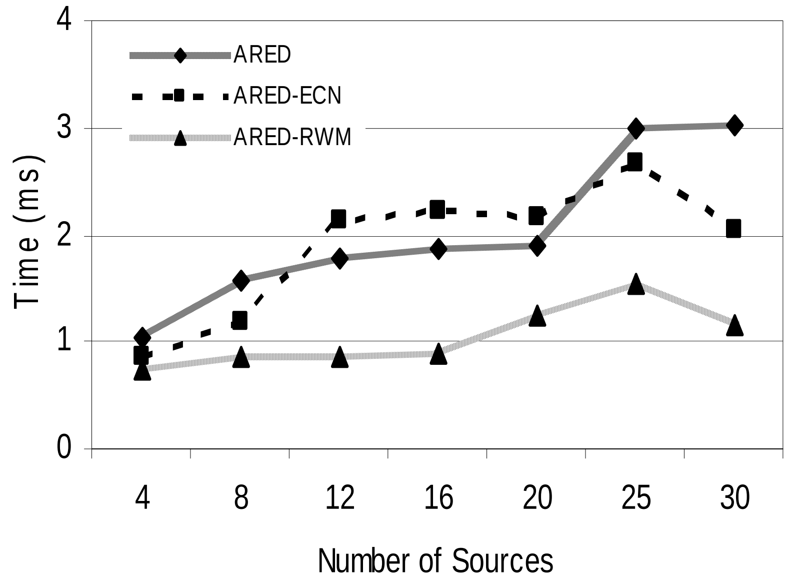

Figure 3.10 Delay jitter for ARED, ARED-ECN and ARED-RWM

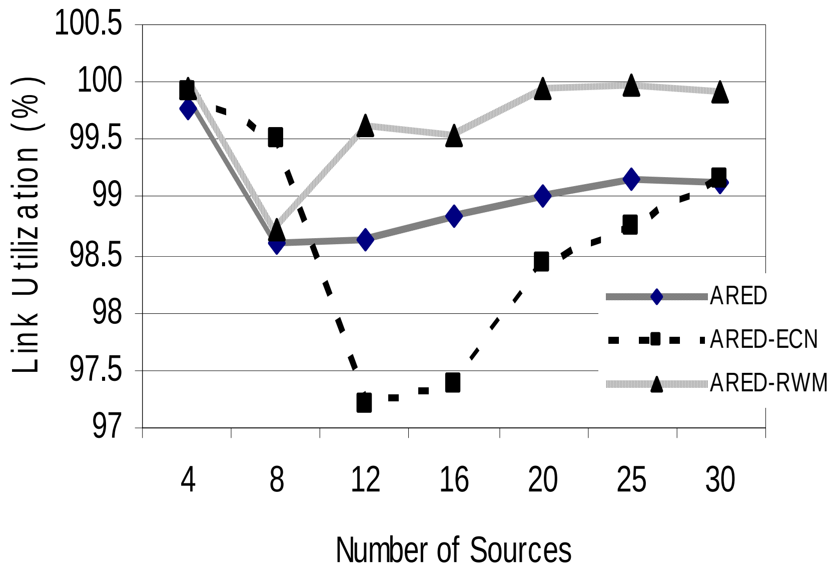

Figure 3.11 Throughput for ARED, ARED-ECN and ARED-RWM

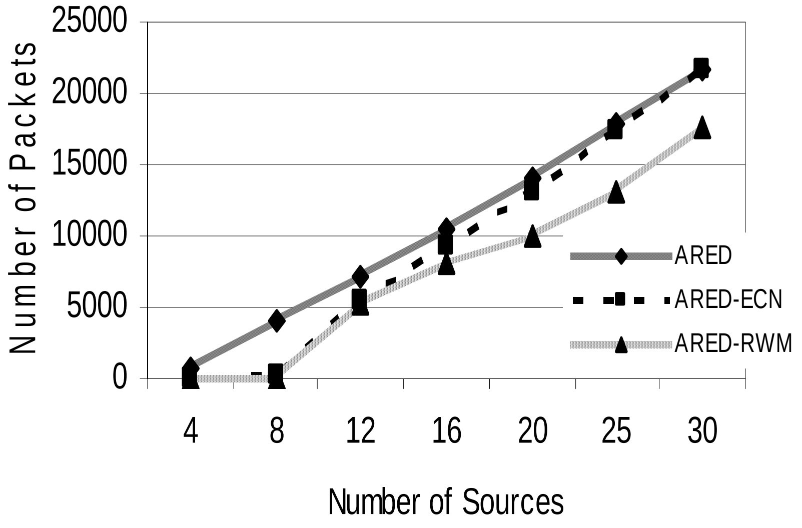

Figure 3.12 Total packet drops for ARED, ARED-ECN and ARED-RWM

3.2.3 Scenario 3 – BLUE AQM

Here, we investigate the performance of BLUE, BLUE-ECN and BLUE-RWM queues using the network topology of our previous scenarios. We use the following BLUE parameters [20]:

- *Freeze_time* = 10ms
- $\delta_1 = 0.02$
- $\delta_2 = 0.002$
- threshold L = 15 packets

The average queue sizes for BLUE, BLUE-ECN and BLUE-RWM are shown in figure 3.13. The average queue size for BLUE-RWM is slightly better than those of BLUE and BLUE-ECN.

Keeping the same topology and BLUE parameters but varying the number of sources, BLUE, BLUE-ECN and BLUE-RWM queues are also evaluated in terms of their one-way average packet delay, delay jitter, throughput and number of total packet drops; figures 3.14, 3.15, 3.16 and 3.17 show the respective results. In terms of the packet delay, delay jitter, throughput and number of total packet drops, BLUE-RWM significantly outperforms BLUE and BLUE-ECN.

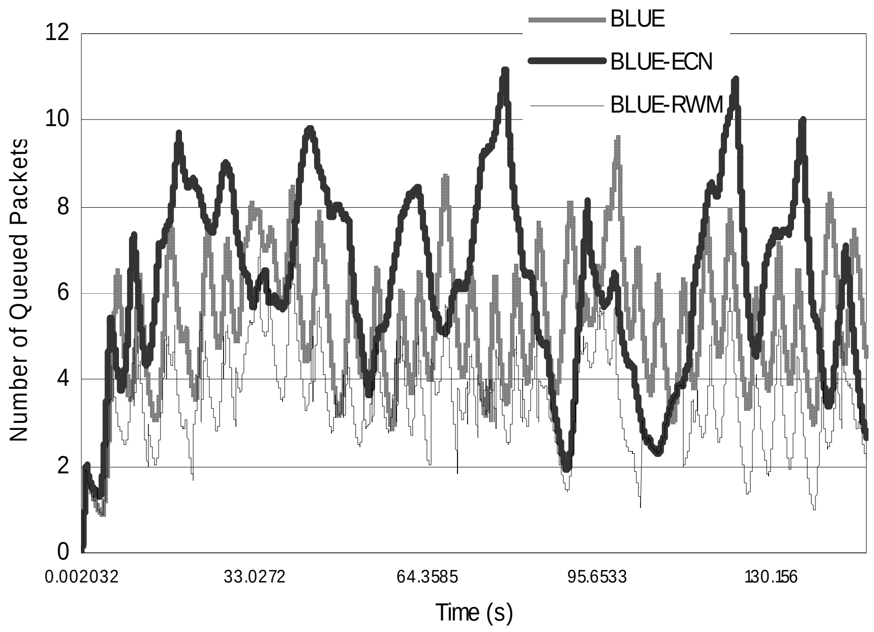

Figure 3.13 Average queue sizes for BLUE, BLUE-ECN and BLUE-RWM

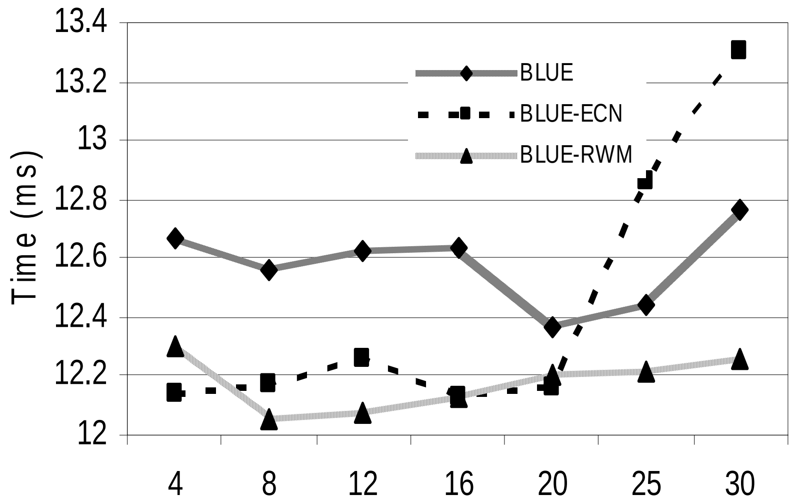

Figure 3.14 Average packet delay for BLUE, BLUE-ECN and BLUE-RWM

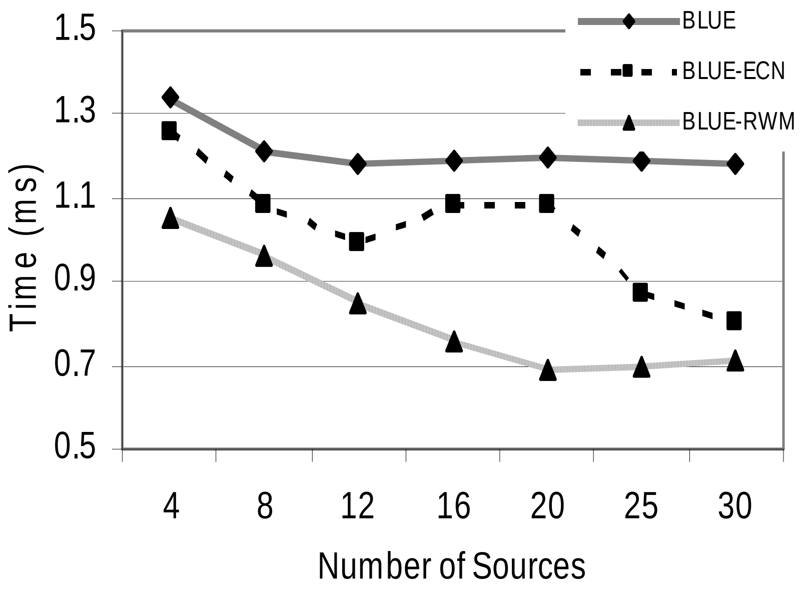

Figure 3.15 Delay jitter for BLUE, BLUE-ECN and BLUE-RWM

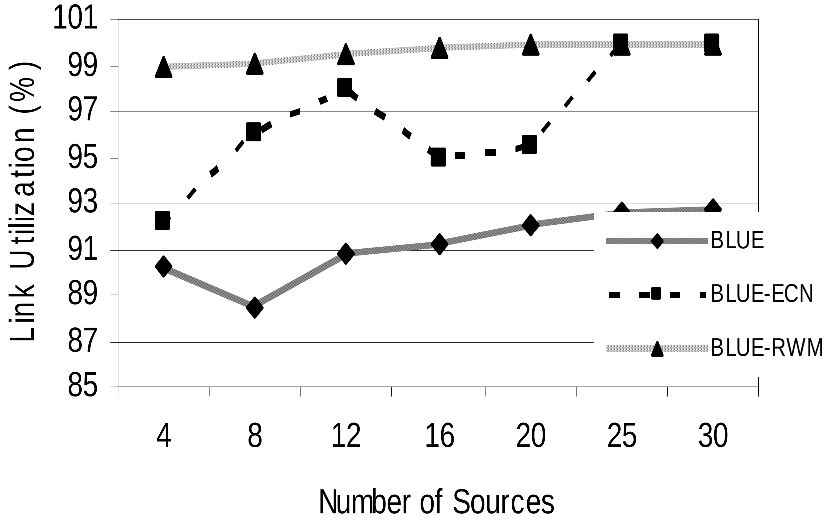

Figure 3.16 Throughput for BLUE, BLUE-ECN and BLUE-RWM

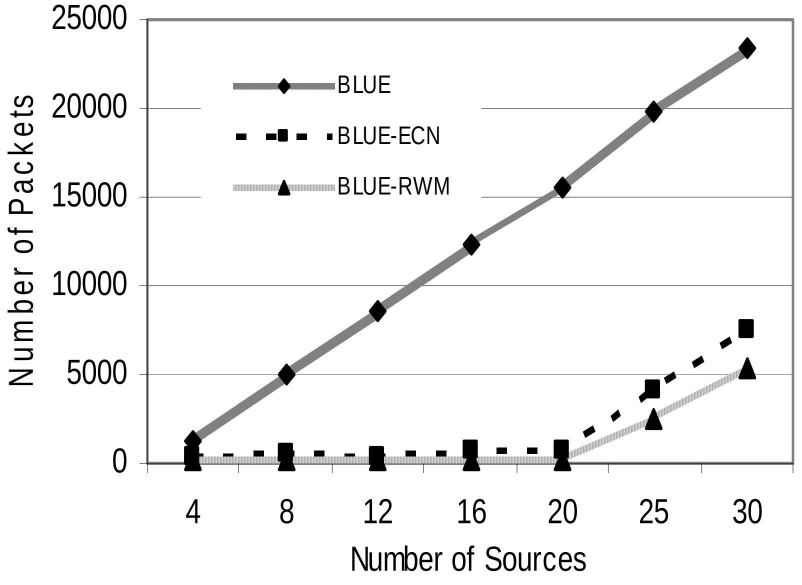

Figure 3.17 Total packet drops for BLUE, BLUE-ECN and BLUE-RWM

3.2.4 Scenario 4 – Comparing RWM AQMs

In this section we are comparing RED-RWM, ARED-RWM, and BLUE-RWM AQMS in terms of one-way average packet delay, delay jitter, number of total packet drops and throughput. Figures 3.18, 3.19, 3.20 and 3.21 show the respective simulation results. Overall, BLUE-RWM outperforms both RED-RWM and ARED-RWM as the number of sources increases.

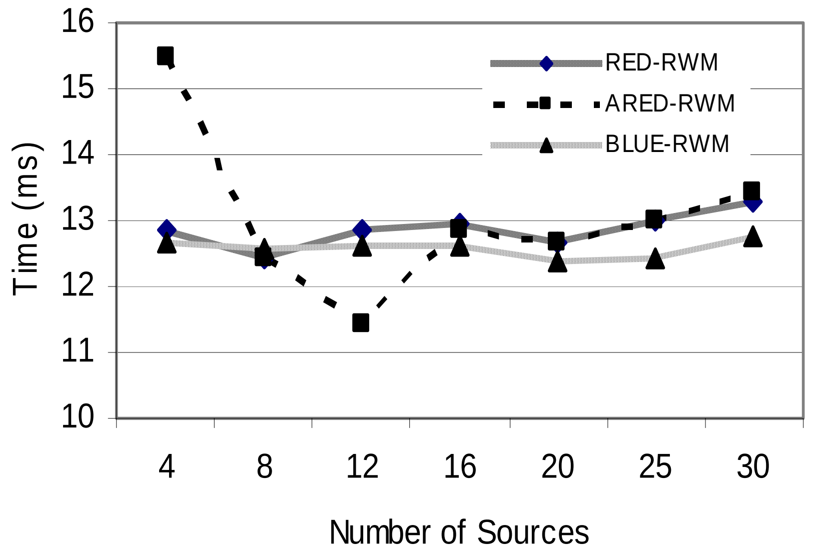

Figure 3.18 Average packet delay for RED-RWM, ARED-RWM and BLUE-RWM

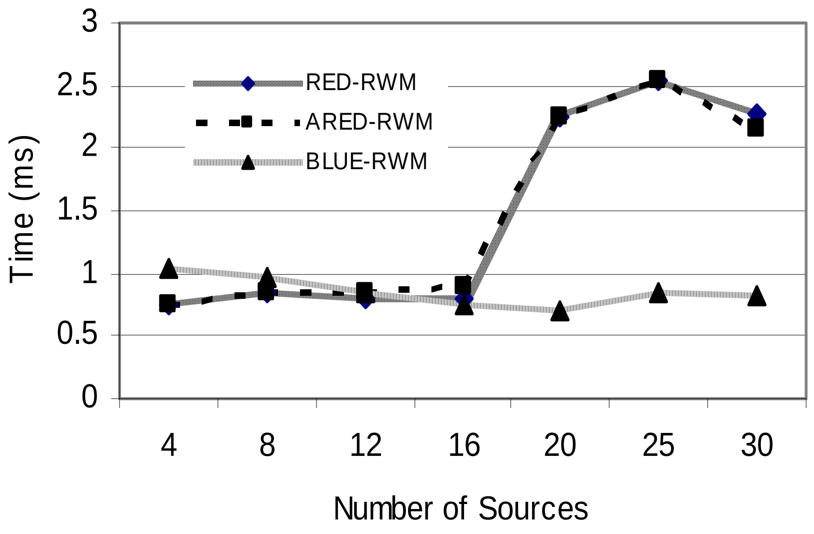

Figure 3.19 Delay jitter for RED-RWM, ARED-RWM and BLUE-RWM

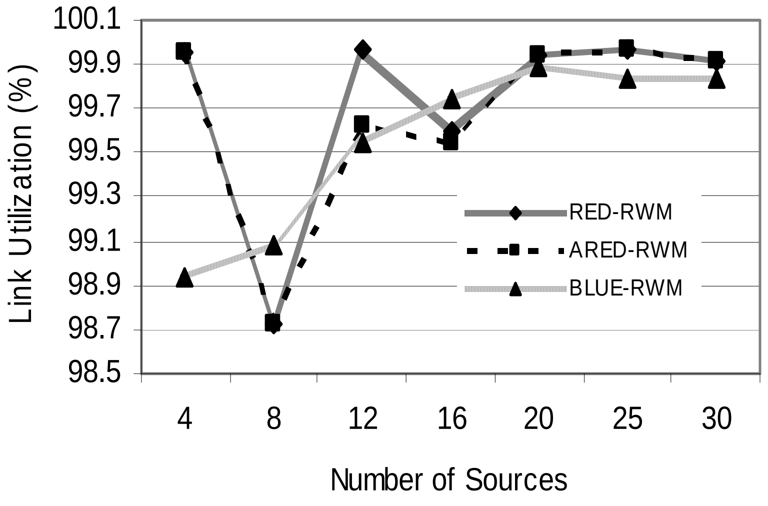

Figure 3.20 Throughput for RED-RWM, ARED-RWM and BLUE-RWM

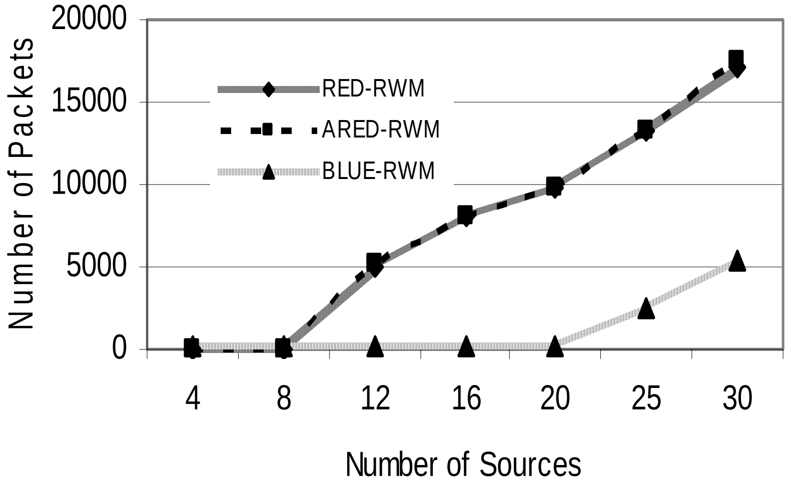

Figure 3.21 Total packet drops for RED-RWM, ARED-RWM and BLUE-RWM

The reasons why BLUE-RWM performs better than RED-RWM and ARED-RWN are due the fact that BLUE-RWM inherits BLUE's ability to perform queue management based directly on packet loss and link utilization rather than on the instantaneous or average queue lengths as in the case of RED-RWM and ARED-RWM. Unlike RED-RWM and ARED-RWM, BLUE-RWM reduces congestion with a minimal amount of buffer space; hence it is performs better. The experiments also show that there is only slight difference between RED-RWM and ARED-RWM.

3.3 Summary

In this chapter we had proposed a modification to existing adaptive queue management protocols (AQM) called Receiver-Window Modification (RWM). RWM does not require modification to all end system TCP/IP stacks but could be solely implemented in heavily-congested ingress and gateway routers. This means that RWM does not require both the sources and receivers to be "RWM-compliant" as was the case for ECN-compliant queues. Our RWM scheme helped to reduce the average queue sizes of the RED, ARED, BLUE, ARED-ECN, RED-ECN and BLUE-ECN queues. By reducing the average queue sizes, RWM queues reduced the queuing delay resulting in significant improvements in one-way end-to-end average packet delays, delay jitter, throughput and number of dropped packets. We have also shown that the performance of RED-ECN, ARED-ECN and BLUE-ECN queues was heavily dependent on the queues of the downstream routers. RWM queues in ingress and gateway routers were not influenced by the number and state of the downstream router as they would piggyback congestion information to the source in the next available acknowledgement packet.

CHAPTER 4

RECEIVER-WINDOW MODIFICATION MODEL

This chapter presents the mathematical model, with asymptotic and steady-state analysis, for the RWM scheme, introduced in Chapter 3, using the RED gateway, i.e. RED-RWM gateway, with *N* TCP connections competing for bandwidth. We expand on the model developed for the RED-ECN gateway in [25][26][27][28][29][30][31] to the RWM extension, henceforth to the RED-RWM, relaxing the assumption that the buffer size should be infinite. Here, upon selection of packets, the RED-RWM queue, instead of marking them, sets the receiver window field to one maximum segment size (MSS) in the ACK packets that are going towards the sender from the receiver. The main idea of the RWM approach is to restrict the TCP transmission window with the flow control window instead of the congestion control window, thus controlling the transmission window with a finer granularity. Results from simulations in Chapter 3 have shown that RWM extended queues outperform their counterparts (functioning with and without ECN) in terms of number of total packet drops, average packet delay, delay jitter and throughput.

The rest of this chapter is organized as follows. The next section describes our time difference RWM model while Section 4.2 and 4.3 present an asymptotic and a steady-state analysis respectively. Section 4.4 provides Monte Carlo simulation results on our formulas, analyzing and comparing them to discrete event simulation results. Finally, Section 4.5 concludes the chapter.

4.1 The RWM Time-Difference Model

We now present our model. We will discretize time T with a uniform duration ΔT and make use of timeslots represented by $[t*\Delta T, (t+1)*\Delta T)$ where t is an integer starting at 0 (thus we will be talking about timeslot t or time $t*\Delta T$). The round-trip times (RTTs) of TCP connections are approximated as integer multiples of the length of timeslots (ΔT). We define a logical indicator function we denote: 1[L] which evaluates to 1 if L is true and to 0 if L is false. We also define the following variables:

1) $f_m^{(N)}(t)$ **:** $\mathbf{R}_+ \rightarrow [0,1]$: the feedback (segment acknowledgement modifying) probability function in the RED-RWM gateway when there are x number of packets in the queue being used by N number of sources. Hence, the probability of acknowledgements being modified or segments being dropped by the RED-RWM gateway is $f_m^{(N)}(t)$. ($f_m^{(N)}(t)$ should scale with N.)
2) $f_d^{(N)}(t)$**:** $\mathbf{R}_+ \rightarrow [0,1]$: the segment dropping probability function in the RED-RWM gateway when there are x number of packets in the queue being used by N number of sources. The same rules apply to $f_d^{(N)}(t)$ as to $f_m^{(N)}(t)$.
3) $Q^{(N)}(t)$: the queue size at time $\Delta T*t$ with N number of TCP connections. During timeslot t, this size will increase or decrease if the number of incoming packets at the queue is greater or less (respectively) than the number of outgoing packets.

4) $A_i^{(N)}(t)$: the number TCP segments received by the RED-RWM gateway from TCP connection i ($i \in \{1,\ldots,N\}$) within timeslot t; thus, the total number incoming segments at the RED-RWM queue is:

$$A^{(N)}(t) = \sum_{i=1}^{N} A_i^{(N)}(t) \qquad 4.1$$

$A_i^{(N)}(t)$ can be modeled by an integer-valued random variable that encodes the transmission window size (in packets) at time $\Delta T{*}t$; we assume that $A_i^{(N)}(t) \in \{0,\ldots,W_{max}\}$, where W_{max} represents the receiver advertised window size of the connection (the transmission window size of an idle connection is zero).

5) $C^{(N)}(t)$: the number of TCP segments served by the RED-RWM gateway during time slot t.

6) Since the queue is always non-negative, using the above variables, the queue dynamics can be represented by the following recursive function:

$$Q^{(N)}(t+1) = [Q^{(N)}(t) + A^{(N)}(t) - C^{(N)}(t)\ ]^{+} \qquad 4.2$$

In addition, both A(N)(t)and C(N)(t) follow some stochastic recursions which depend on the current queue size thereby forming a stochastic feedback system. If we denote the average queue serving capacity per connection by C, then C(N)(t)=NC, thus equation 4.1 becomes:

$$Q^{(N)}(t+1) = [Q^{(N)}(t) + A^{(N)}(t) - NC\ ]^{+} \qquad 4.3$$

7) $W_i^{(N)}(t)$: the transmission window size at connection i at the beginning of timeslot t. The transmission windows size is the minimum of *cwnd* and *rwnd*. We assume that

each connection will transmit all TCP segments that it can during slot *t*. Note, that $A_i^{(N)}(t) \leq W_i^{(N)}(t)$. (More precisely if $\Phi_i^{(N)}(t)$ is the number of pending TCP segments for connection *i* at time $\Delta T*t$ and connections have an unlimited amount of segments to transmit, then $A_i^{(N)}(t)=W_i^{(N)}(t)- \Phi_i^{(N)}(t)$.)

8) $X_i^{(N)}(t)$: the remaining workload (in packets) at the TCP sending process of connection *i* at the beginning of timeslot *t*; thus, $A_i^{(N)}(t) \leq \min(W_i^{(N)}(t),X_i^{(N)}(t))$. (More precisely, $A_i^{(N)}(t)=\min(W_i^{(N)}(t)-\Phi_i^{(N)}(t),X_i^{(N)}(t))$)

9) $D_i^{(N)}(t)$: the round trip time (RTT) at time $t*\Delta T$ as observed by the TCP sending process of connection *i*; $D_i^{(N)}(t)$ will be given in integer multiples of ΔT. We model $D_i^{(N)}(t)$ as a random variable of possible outcomes: $\{2,\ldots,D_{max}\}$.

10) We will assume that any congestion-control action occurs at the end of a round-trip. We also simplify the segment transmissions to the queue by assuming that all segments allowed within an RTT are transmitted in the same slot to the queue. Let βi(N)(t) represent the number of timeslots (at timeslot t) since the last time connection i has transmitted a segment to the queue; then:

$$\beta_i^{(N)}(0) = 1[X_i^{(N)}(0) > 0]$$

$$\beta_i^{(N)}(t+1) = (1+ \beta_i^{(N)}(t) \ 1[\beta_i^{(N)}(t) < D_i^{(N)}(t)]) \ 1[X_i^{(N)}(t) > 0]$$

With these assumptions our formula for $A_i^{(N)}(t)$ will evolve to:

$$A_i^{(N)}(t) = \min(W_i^{(N)}(t), X_i^{(N)}(t)) 1[\beta_i^{(N)}(t) = 1]$$

where the indicator function $1[\beta_i^{(N)}(t)=1]$ is to ensure that the connection transmits only once per RTT. (Note, that the above assumptions eliminate the need for variable $\Phi_i^{(N)}(t)$.) Now our formula for the queue evolution can be rewritten as:

$$Q^{(N)}(t+1)=[Q^{(N)}(t)-NC+\sum_{i=1}^{N}\min(W_i^{(N)}(t),X_i^{(N)}(t))1[\beta_i^{(N)}(t)=1]] \qquad 4.4$$

11) Let us introduce function $G_{i,t}(a,b)$ evaluating to a if the last transmission action of connection i at time $t*\Delta T$ was within the then applying roundtrip time, and evaluating to b otherwise. G will be a useful tool to change variables only after a RTT has passed.

$$G_{i,t}(a,b)=a*1[\beta_i^{(N)}(t)>1]+b*1[\beta_i^{(N)}(t)=1] \qquad 4.5$$

12) Let the modifying and dropping probability functions be given by :

$$f_m^{(N)}(\hat{Q}^{(N)}(t))=\begin{cases}0 & Th_{\min}>\hat{Q}\\ P_{\max}(\dfrac{\hat{Q}-Th_{\min}}{Th_{\max}-Th_{\min}}) & Th_{\min}\le\hat{Q}\le Th_{\max}\\ 0 & \hat{Q}>Th_{\max}\end{cases}$$

$$f_d^{(N)}(\hat{Q}^{(N)}(t))=\begin{cases}0 & Th_{\max}>\hat{Q}\\ 1 & \hat{Q}>Th_{\max}\end{cases}$$

where P_{max} is the maximum marking probability, Th_{min} is the minimum threshold, Th_{max} is the maximum threshold.

13) Since the RED-RWM scheme complements the RED queue, it uses the same queue averaging filter as the RED queue (known as Exponentially Weighted Moving Average - EWMA) to evaluate the average queue size based on the instantaneous queue size with a past-weighting parameter α ($0 < \alpha \leq 1$). Let $\hat{Q}^{(N)}(t)$ represent average queue at time $t*\Delta T$, then:

$$\hat{Q}^{(N)}(t+1) = (1-\alpha)\hat{Q}^{(N)}(t) + \alpha Q^{(N)}(t+1) \qquad 4.6$$

14) Let $z_{i,j}^{(N)}(t+1)$ represent random indicator variables for dropped packets, where $j \in \{1, ..., A_i^{(N)}(t)\}$; $z_i^{(N)}(t+1)$ is zero if the j^{th} packet of connection i is dropped in timeslot t (and 1 otherwise). Thus:

$$z_{i,j}^{(N)}(t+1) = 1[(L_{i,j}(t+1) > f_d^{(N)}(\hat{Q}^{(N)}(t))]$$

where $L_{i,j}$(t)s are I..I.D. (in all their dimensions) uniform random variables taking values from [0, 1]. Hence, the indicator function of the event that no packet of connection i is dropped in timeslot t can be written as:

$$z_i^{(N)}(t+1) = \begin{cases} \prod_{j=1}^{A_i^{(N)}(t)} z_{i,j}^{(N)}(t+1) & A_i^{(N)}(t) \geq 1 \\ 1, & A_i^{(N)}(t) = 0 \end{cases}$$

$$z_i^{(N)}(0) = 1$$

If $z_i^{(N)}(t+1) = 0$, then at least one packet from connection i in timeslot t has been dropped by the RED-RWM gateway. This information may only be used one RTT later to halve the value of the congestion window size, thus we need to propagate a zero value to the end of the roundtrip time. We will use variable $Z_i^{(N)}(t)$ for such propagation. $Z_i^{(N)}(t)$ thus evolves according to:

$$Z_i^{(N)}(t+1) = G_{i,t+1}(Z_i^N(t) * z_i^{(N)}(t+1), 1)$$

$$Z_i^{(N)}(0) = 1$$

Now we can define variable $\hat{z}_i^{(N)}(t)$ to reflect that there was a drop during the previous RTT. $\hat{z}_i^{(N)}(t)$ evolves according to

$$\hat{Z}_i^{(N)}(t+1) = G_{i,t+1}(\hat{Z}_i^N(t), Z_i^{(N)}(t)) \qquad 4.7$$

15) Similarly, let the modification of acknowledgement packets by the RED-RWM gateway be represented by an indicator random variable $m_{i,j}^{(N)}(t+1)$ where $j \in \{1, \ldots, A_i^{(N)}(t)\}$; $m_{i,j}^{(N)}(t+1)$ is zero if the j^{th} packet of connection i causes the acknowledgement packet to be modified by the RED-RWM gateway to one MSS in timeslot t:

$$m_{i,j}^{(N)}(t+1) = 1[(V_{i,j}(t+1) > f_m^{(N)}(\hat{Q}^{(N)}(t))]$$

where $V_{i,j}$(t)s are I.I.D. (in all their dimensions) uniform random variables taking values from [0, 1]. Hence, the indicator function of the event that no packet of connection i is RWM-modified in timeslot t can be written as

$$m_i^{(N)}(t+1)=\begin{cases}\prod_{j=1}^{A_i^{(N)}(t)} m_{i,j}^{(N)}(t+1) & A_i^{(N)}(t)\geq 1\\ 1, & A_i^{(N)}(t)=0\end{cases}$$

$$m_i^{(N)}(0)=1$$

If $m_i^{(N)}(t+1)=0$, then at least one packet from connection *i* in timeslot *t* has been RWM-modified by the RED-RWM gateway. This information may only be used one RTT later to modify the transmission to 1 MSS, thus we need to propagate a zero value to the end of the roundtrip time. We will use variable $M_i^{(N)}(t)$ for such propagation. $M_i^{(N)}(t)$ thus evolves according to:

$$M_i^{(N)}(t+1)=G_{i,t+1}(M_i^N(t)*m_i^{(N)}(t+1),1)$$

$$M_i^{(N)}(0)=1$$

Now we can define variable $\hat{M}_i^N(t)$ to reflect that there was a modification during the previous RTT. $\hat{M}_i^N(t)$ evolves according to:

$$\hat{M}_i^{(N)}(t+1)=G_{i,t+1}(\hat{M}_i^N(t),M_i^{(N)}(t)) \qquad 4.8$$

16) Connection *i*, after receiving the modified acknowledgement from the RED-RWM gateway, will transmit only one MSS for one RTT. After the RTT the source will transmit the minimum of the previous congestion window before modification and the new advertised receiver window (unless it receives more modified acknowledgements from the RED-RWM gateway). Thus, we need to save the

transmission window's size before the RWM modification has happened using variable $\Omega_i^{(N)}(t)$. $\Omega_i^{(N)}(t)$ evolves according to:

$$\Omega_i^{(N)}(t+1) = G_{i,t+1}(\Omega_i^{(N)}(t), W_i^{(N)}(t+1)\hat{M}_i^{(N)}(t+1) + \Omega_i^{(N)}(t)(1-\hat{M}_i^{(N)}(t+1))) \qquad 4.9$$

$$\Omega_i^{(N)}(0) = 1$$

17) We also need an indicator variable at each roundtrip's end time, showing whether there has been a marking two roundtrips ago while there has been none in the roundtrip that was just finished.

$$E_i^{(N)}(t+1) = G_{i,t+1}(0,(1-\hat{M}_i^{(N)}(t))M_i^{(N)}(t)) \qquad 4.10$$

18) Each connection may be in one of two basic modes i.e., either in an active or an idle mode. At the beginning of timeslot t if a session has no packet to transmit then it is in the idle mode. An idle connection may become active by the beginning of the next timeslot with probability p_{ar} independently of previous events. Hence the duration of an idle period is *geometrically* distributed with parameter p_{ar} (and a mean of $1/p_{ar}$). We can use this to capture the dynamics of connection arrivals, where the interarrival times are exponentially distributed (as the geometric distribution is the discrete equivalent to the exponential distribution), we let i) $U_i(t)$ ($i \in \{1,\ldots,N\}$) represent a collection of I.I.D. uniform random variables on [0, 1] and ii) $\mathbf{1}[U_i(t+1) \leq p_{ar}]$ represent an indicator function for an idle connection i for the event of the arrival of a new packet in timeslot t+1.

19) Let $F_i(t)$ represent the total number of TCP segments (the session size) connection i will have to transmit if it becomes active at the beginning of timeslot t; $F_i(t)$ can be modeled as a collection of I.I.D. non-negative integer-valued random variables with some general probability mass function (usually a geometric or a discretized exponential pmf). Recall, that we used $X_i^{(N)}(t)$ to denote the remaining workload (expressed in TCP segments) of connection i at the beginning of timeslot t; thus:

$$X_i^{(N)}(t+1) = 1[X_i^{(N)}(t)>0](X_i^{(N)}(t)-A_i^{(N)}(t)) \qquad 4.11$$

$$+1[X_i^{(N)}(t)=0]1[U_i(t+1)\le P_{ar}]F_i(t+1)$$

20) RTT will not change for a connection during the same RTT, thus we can formulate it using the *$G_{i,t}(a,b)$* function. At the beginning of time slot *t*+1 (if *t*+1 is the timeslot for connection *i* to send its data) the expected roundtrip time is determined by the proportion of how many segments are in the queue at time *$\Delta T^*(t+1)$* plus how many are expected during slot *t*+1 (which is *$A^{(N)}(t+1)$*) and the rate with which the queue is served:

$$D_i^{(N)}(t+1) = \max(G_{i,t+1} = (D_i^{(N)}(t),(Q^{(N)}(t+1)+$$

$$A^{(N)}(t+1))/(N*C)),2) \qquad 4.12$$

Note, that this can lead to synchronization if we assume that all connections arrive at the same time. This synchronization will be removed by the random marking and dropping as *t* grows. The above formulation also requires *$D_i^{(N)}(t)$* to be at least two as we expect the acknowledgments to come back in a different time slot than what was used to send their segments.

21) The evolution of the window mechanism for connection *i* can now be described through recursion. The congestion-control mechanism of TCP operates either in the *slow start* (SS) or the *congestion avoidance* (CA) phase. A new connection starts in SS in order to probe the available bandwidth of the network (the queue in our case). While in SS, the congestion window size is doubled every round-trip time until the receiver window field of one or more acknowledgement packets are modified by the RED-RWM gateway to one MSS or one or more packets are dropped by the RED-RWM gateway. Note that the transmission window size is limited by the receiver advertised window size W_{max}. Thus, if the connection is in SS then the transmission window of connection *i* evolves according to

$$\begin{aligned} W_{i,SS}^{(N)}(t+1) = G_{i,t+1}(W_{i,SS}^{(N)}(t), \min(\max(2W_i^{(N)}(t),1), W_{\max})\hat{Z}_i^{(N)}(t+1)\times \\ (1-E_i^{(N)}(t+1))\hat{M}_i^{(N)}(t+1) + \\ E_i^{(N)}(t+1) * \Omega_i^{(N)}(t) + (1-\hat{M}_i^{(N)}(t+1)) + \\ \left\lceil \frac{W_i^{(N)}(t)}{2} \right\rceil \hat{M}_i^{(N)}(t+1)(1\text{-}\hat{Z}_i^{(N)}(t+1))(1-E_i^{(N)}(t+1))) \end{aligned} \quad 4.13$$

22) In CA, the congestion window size in the next timeslot (*t*+1) is incremented if no modified acknowledgement packet was received in timeslot t, while if one or more

segments are modified in timeslot *t* the congestion window in the next timeslot is set to one segment. The congestion window size in CA is then given by:

$$W_{i,CA}^{(N)}(t+1) = G_{i,t+1}(W_{i,CA}^{(N)}(t), \min(W_i^{(N)}(t)+1, W_{\max})\hat{Z}_i^{(N)}(t+1)\times$$
$$(1-E_i^{(N)}(t+1))\hat{M}_i^{(N)}(t+1)+$$
$$E_i^{(N)}(t+1)*\Omega_i^{(N)}(t)+(1-\hat{M}_i^{(N)}(t+1))+$$
$$\left\lceil \frac{W_i^{(N)}(t)}{2} \right\rceil (1\text{-}\hat{Z}_i^{(N)}(t+1))(1\text{-}E_i^{(N)}(t+1))\hat{M}_i^{(N)}(t+1)) \qquad 4.14$$

23) Let the $S_i^{(N)}(t)$ random indicator variable denote the state of TCP connections ($S_i^{(N)}(t)$ is zero if connection *i* is in CA and one if it is in SS). Therefore, combining equations 4.14 and 4.15, the complete recursion of the congestion window size can be written as:

$$W_i^{(N)}(t+1) = (\, S_i^{(N)}(t) W_{i,SS}^{(N)}(t+1) +$$
$$(1\text{ - } S_i^{(N)}(t)) W_{i,CA}^{(N)}(t+1)) 1[X_i^{(N)}(t) - A_i^{(N)}(t) > 0\,]$$

The last indicator function is used to reset the congestion window size to zero when connection *i* runs out of data and returns to its idle state.

24) Finally, the evolution of $S_i^{(N)}(t)$ can be described as:

$$S_i^{(N)}(t+1) = 1[X_i^{(N)}(t) \le W_i^{(N)}(t)\,] +$$
$$1[X_i^{(N)}(t) > W_i^{(N)}(t)\,]\, S_i^{(N)}(t)\hat{Z}_i^{(N)}(t+1) \qquad 4.15$$

where connection *i* will be in the SS state during timeslot *t* if there is either i) no packet left to transmit (so the connection resets) at the beginning of the timeslot or ii) the connection was active and in the SS state then a packet drop or marking leads to the CA state. Equation 4.15 assumes that a new TCP connection, in the SS state, is set up one timeslot after the previous connection is torn down upon finishing its workload, and the new TCP connection becomes active when a new file/object arrives, initiating a three-way handshake. Moreover, it implies that the state is updated in the next timeslot following a transmission despite the window size being updated one RTT after transmission using the appropriate SS/CA state as in the correct operation of TCP.

4.2 Asymptotic Analysis

Here, we present a theorem for the asymptotic analysis for the RED-RWM queue when the number of connections using the queue is large. First we present the initial conditions and then the theorem.

(**Initial condition 1**) For i = 1, . . .,N, the initial conditions of random variables in the model are:

$$Q^{(N)}(0) = \hat{Q}^{(N)}(0) = W_i^{(N)}(0) = E_i^{(N)}(0) = \beta_i^{(N)}(0) = 0;$$

$$D_i^{(N)}(0) = 2;$$

and

$$z_i^{(N)}(0) = \hat{Z}_i^{(N)}(0) = Z_i^{(N)}(0) = S_i^{(N)}(0) = \Omega_i^{(N)}(0) = m_i^{(N)}(0) = \hat{M}_i^{(N)}(0) = M_i^{(N)}(0) = 1.$$

We denote the vector of state variables for connection *i* in timeslot *t* by

$$Y_i^{(N)}(t) := (W_i^{(N)}(t), X_i^{(N)}(t), E_i^{(N)}(t), S_i^{(N)}(t), D_i^{(N)}(t), \beta_i^{(N)}(t), \Omega_i^{(N)}(t), m_i^{(N)}(t), \hat{M}_i^{(N)}(t), M_i^{(N)}(t), z_i^{(N)}(t), \hat{Z}_i^{(N)}(t), Z_i^{(N)}(t)). \quad 4.16$$

(Initial condition 2) All TCP connections are responsive to congestion control and the number of connections N is greater than the value of *Th_{min}*.

Theorem 1.

Assume that both Initial Conditions (1 and 2) are valid and

$$Y_i^{(N)}(t) \Rightarrow_N Y(t),\ Q_i^{(N)}(t) \Rightarrow_N Q(t) \text{ and } \hat{Q}_i^{(N)}(t) \Rightarrow_N \hat{Q}(t),$$

where $\Rightarrow_N$ *denotes weak convergence or convergence in distribution with large N, then the sequences* $\{Y_i^{(N)}(t), Q_i^{(N)}(t), \hat{Q}_i^{(N)}(t)\}$ *and* $\{Y(t), Q(t), \hat{Q}(t)\}$ *with t= 0, 1, 2, 3…. are time-homogeneous Markov chains, i.e. their future states are dependent on the present state and not on past states.*

Given the recurrence of the RWM model as

$$Y_i^{(N)}(t+1) = (W_i^{(N)}(t+1), X_i^{(N)}(t+1), E_i^{(N)}(t+1), S_i^{(N)}(t+1), D_i^{(N)}(t+1), \beta_i^{(N)}(t+1), \Omega_i^{(N)}(t+1), m_i^{(N)}(t+1), M_i^{(N)}(t+1), \hat{M}_i^{(N)}(t+1), z_i^{(N)}(t+1), Z_i^{(N)}(t+1), \hat{Z}_i^{(N)}(t+1))$$

$$=_{st} P(Y_i^{(N)}(t))$$

$$:= (P_1(Y_i^{(N)}(t)), P_2(Y_i^{(N)}(t)), \ldots, P_{13}(Y_i^{(N)}(t)))$$

with the state space Y given by

$$y := \{0,1,\ldots,W_{\max}\}\times\{0,1,\ldots,X_{\max}\}\times\{0,1\}\times\{0,1\}\times\{2,3,\ldots,D_{\max}\}\times\{0,1,\ldots,D_{\max}\}$$

$$\times\{0,1,\ldots,X_{\max}\}\times\{0,1\}\times\{0,1\}\times\{0,1\}\times\{0,1\}\times\{0,1\}\times\{0,1\}.$$

Then it holds by analysis that $P_1(Y_i^{(N)}(t)), P_2(Y_i^{(N)}(t)),\ldots,P_{13}(Y_i^{(N)}(t))$ *are given by*

$$P_1(Y_i^{(N)}(t)) = S_i^{(N)}(t)W_{i,SS}^{(N)}(t+1) + (1-S_i^{(N)}(t))W_{i,CA}^{(N)}(t+1)1[X_i^{(N)}(t)-A_i^{(N)}(t)>0],$$

$$P_2(Y_i^{(N)}(t)) = 1[X_i^{(N)}(t)>0](X_i^{(N)}(t)-A_i^{(N)}(t)) + 1[X_i^{(N)}(t)=0]\,1[U_i(t+1)<P_{ar}]F_i(t+1),$$

$$P_3(Y_i^{(N)}(t)) = G_{i,t+1}(0,(1-\hat{M}_i^{(N)}(t))M_i^{(N)}(t)),$$

$$P_4(Y_i^{(N)}(t)) = 1[X_i^{(N)}(t)-W_i^{(N)}(t)\le 0] + 1[X_i^{(N)}(t)-W_i^{(N)}(t)>0]\,S_i^{(N)}(t)\hat{Z}_i^{(N)}(t+1),$$

$$P_5(Y_i^{(N)}(t)) = \max(G_{i,t+1} = (D_i^{(N)}(t),(Q_i^{(N)}(t+1)+A_i^{(N)}(t+1))/(N*C)),2),$$

$$P_6(Y_i^{(N)}(t)) = (1+\ \beta_i^{(N)}(t)\ 1[\beta_i^{(N)}(t)<D_i^{(N)}(t)])\ 1[X_i^{(N)}(t)>0]$$

$$P_7(Y_i^{(N)}(t)) = G_{i,t+1}(\Omega_i^{(N)}(t), W_i^{(N)}(t+1)\hat{M}_i^{(N)}(t+1) + \Omega_i^{(N)}(t)(1-\hat{M}_i^{(N)}(t+1))),$$

$$P_8(Y_i^{(N)}(t)) = (\textstyle\prod_{j=1}^{A_i^{(N)}(t)} 1[V_{i,j}(t+1)>f_m(\hat{Q}^{(N)}(t))])\hat{Z}_i^{(N)}(t+1),$$

$$P_9(Y_i^{(N)}(t)) = G_{i,t+1}(M_i^{(N)}(t)*m_i^{(N)}(t+1),1),$$

$$P_{10}(Y_i^{(N)}(t)) = G_{i,t+1}(\hat{M}_i^{(N)}(t), M_i^{(N)}(t)),$$

$$P_{11}(Y_i^{(N)}(t)) = (\textstyle\prod_{j=1}^{A_i^{(N)}(t)} 1[L_{i,j}(t+1)>f_d(\hat{Q}^{(N)}(t))])\hat{Z}_i^{(N)}(t+1)\hat{M}_i^{(N)}(t+1),$$

$$P_{12}(Y_i^{(N)}(t)) = G_{i,t+1}(Z_i^{(N)}(t)*z_i^{(N)}(t+1),1),$$

$$P_{13}(Y_i^{(N)}(t)) = G_{i,t+1}(\hat{Z}_i^{(N)}(t), Z_i^{(N)}(t)),$$

where

$$G_{i,t}(a,b) = a*1[\beta_i^{(N)}(t)>1] + b*1[\beta_i^{(N)}(t)=1] \qquad a,b\in\Re,$$

$$W_{i,SS}^{(N)}(t+1) = G_{i,t+1}(W_{i,SS}^{(N)}(t), \min(\max(2W_i^{(N)}(t),1), W_{\max})\hat{Z}_i^{(N)}(t+1)\times$$

$$(1-E_i^{(N)}(t+1))\hat{M}_i^{(N)}(t+1)+$$

$$E_i^{(N)}(t+1)*\Omega_i^{(N)}(t)+(1-\hat{M}_i^{(N)}(t+1))+$$

$$\left\lceil\frac{W_i^{(N)}(t)}{2}\right\rceil\hat{M}_i^{(N)}(t+1)(1\text{-}\hat{Z}_i^{(N)}(t+1))(1-E_i^{(N)}(t+1)))$$

and

$$W_{i,CA}^{(N)}(t+1) = G_{i,t+1}(W_{i,CA}^{(N)}(t), \min(W_i^{(N)}(t)+1, W_{\max})\hat{Z}_i^{(N)}(t+1)\times$$

$$(1-E_i^{(N)}(t+1))\hat{M}_i^{(N)}(t+1)+$$

$$E_i^{(N)}(t+1)*\Omega_i^{(N)}(t)+(1-\hat{M}_i^{(N)}(t+1))+$$

$$\left\lceil\frac{W_i^{(N)}(t)}{2}\right\rceil(1\text{-}\hat{Z}_i^{(N)}(t+1))(1\text{-}E_i^{(N)}(t+1))\hat{M}_i^{(N)}(t+1))$$

for i.i.d. [0, 1]-uniform rvs { $L_{i,j}(t + 1)$, $U_i(t + 1)$, $V_{i,j}(t + 1)$; t = 0, 1, . . .}.

i. *For any bounded function* $g:\mathrm{Z}_+^{13}\to\Re$, *we have*

$$\frac{1}{N}\sum_{t=1}^{N} g(Y_i^{(N)}(t)) \xrightarrow{P}_N \mathrm{E}[g(Y(t))].$$

where $Y(t)=(W(t),X(t),E(t),S(t),D(t),\beta(t),\Omega(t),m(t),M(t),\hat{M}(t),z(t),Z(t),\hat{Z}(t))$ *and*

$\xrightarrow{P}_N$ *denotes convergence in probability when N tends to infinity, with the state space Y given by*

$$y := \{0,1,\ldots,W_{\max}\}\times\{0,1,\ldots,X_{\max}\}\times\{0,1\}\times\{0,1\}\times\{2,3,\ldots,D_{\max}\}\times\{0,1,\ldots,D_{\max}\}$$

$$\times\{0,1,\ldots,X_{\max}\}\times\{0,1\}\times\{0,1\}\times\{0,1\}\times\{0,1\}\times\{0,1\}\times\{0,1\}.$$

and

$$Y_i^{(N)}(t) \Rightarrow_N Y(t).$$

where $\Rightarrow_N$ *denotes weak convergence or convergence in distribution with large N.*

Here, we are implying that the average TCP dynamics for a large number of flows using a RED-RWM queue is closely related to that of a single flow utilizing the same TCP congestion control mechanism.

ii. *For any integer i* = 1, 2, . . . , *the random vector {* $Y_1^{(N)}(t)$ *, i* = 1, . . . , *I} becomes asymptotically independent as N becomes large, with*

$$\lim_{N\to\infty} \mathrm{P}[Y_i^{(N)}(t)=y_i, i=1,\ldots,I] = \prod_{i=1}^{I} \mathrm{P}[Y(t)=y_i]$$

for any $\mathbf{y}_i \in Y$, *i* = 1, . . . , *I.*

Here, we are implying that as sessions become asymptotically independent as N becomes large, suggesting that the RED-RWM queue alleviates the synchronization problem among the flows.

iii. *For large N, we have*

$$\frac{Q^{(N)}(t)}{N} \xrightarrow{P}_N q(t),$$

$$\frac{\hat{Q}^{(N)}(t)}{N} \xrightarrow{P}_N \hat{q}(t),$$

and

$$A_i^{(N)}(t) \Rightarrow_N NA(t),$$

Resulting in

$$q(t+1) = (q(t) - C + A(t))^+$$

$$\hat{q}(t+1) = (1-\alpha)\hat{q}(t) + \alpha q(t+1)$$

iv. *For any bounded mapping* $g : \mathrm{Y} \to \Re,$ *there exists a bounded and continuous mapping* $F_g : [0,1] \times [0,1] \times \mathrm{Y} \to \Re$ *such that*

$$E[g\left(Y_i^{(N)}(t+1)\right) | \Omega_t, \tau_t] = F_g\left(\Psi_i^{(N)}(t), \chi_i^{(N)}(t), Y_i^{(N)}(t)\right) \tag{4.17}$$

This assumption states that given the events leading up to the beginning of timeslot [t, t + 1), the expected behavior of session i leading to the beginning of timeslot [t + 1, t + 2) can also be determined using the expected knowledge of the conditional receiver window modifying probability, $\Psi_i^{(N)}(t)$, *the conditional dropping probability* $\chi_i^{(N)}(t)$ *and* $Y_i^{(N)}(t)$. *Hence, (Assumption 3) implies that*

$$E[g\left(Y_i^{(N)}(t+1)\right)] = E[F_g\left(\Psi_i^{(N)}(t), \chi_i^{(N)}(t), Y_i^{(N)}(t)\right)] \tag{4.18}$$

This leads to (from iii)

$$E[g\left(Y_i^{(N)}(t+1)\right)] \Rightarrow_N E[F_g\left(\Psi(t), \chi(t), Y(t)\right)] \qquad 4.19$$

implying that for large N, given the events leading up to the beginning of timeslot [t, t + 1), the expected behavior of session i leading to the beginning of timeslot [t + 1, t + 2) can also be determined using the expected value of the conditional receiver window modifying probability, $\Psi(t)$*, the conditional dropping probability* $\chi(t)$ *and* $Y(t)$.

The proof of Theorem 1 is presented in Appendix A.

4.2.1 Limiting Cases

Next, consider the resulting model from Theorem 1 in the regime when *C* is either very large or very small.

<u>C is large:</u> the modifying probability per flow converges to zero for all *t*. Therefore, each incoming flow will always operate in the SS mode with each connection doubling its window sizes every round-trip.

<u>C is very small:</u> the average queue size increases with the modifying probability. The rate of modifying acknowledgements increases as the modifying probability approaches 1. As a result, an increasing number of TCP connections will receive modified acknowledgements and will approximately transmit a single packet. If the dropping probability is 1, packets will be dropped.

<u>4.3 Steady-State Analysis</u>

Here, we present a theorem for the steady-state analysis for the RED-RWM model when the number of connections using the queue is large.

(Initial condition 3) The sequence $\{Y(t), Q(t), \hat{Q}(t)\}$ with t= 0, 1, 2, 3…. admits a steady state such that $(Y(t), q(t), \hat{q}(t)) \Rightarrow_t (Y^*, q^*, q^*)$ where q^* is a constant and

$Y^* = (W^*, X^*, E^*, S^*, D^*, \beta^*, \Omega^*, m^*, M^*, \hat{M}^*, z^*, Z^*, \hat{Z}^*)$ is a y-valued rv.

Theorem 2. *Assume that both Initial conditions (1, 2 and 3) and Theorem 1 are valid. Let us use the notation:* $\Rightarrow_N$ *to denote a weak convergence in N to a steady state. Then:*

$$q^* \Rightarrow_N N.$$

The proof of Theorem 2 is presented in Appendix B.

Intuitively, since RWM complements RED, it inherits RED's property of choosing a particular flow, to mark or drop during congestion, which roughly is proportional to that flow's share of the bandwidth at the router. However, the ideal packet distribution in a RWM queue is one per flow for N TCP flows in steady state, especially for large N and, N > Th_{min} and N < Th_{max}. Hence, RWM's potential is limited by the RED algorithm.

4.4 Simulation Results And Analysis

We have created an NS2 model of the RED-RWM scheme. We used a simple barbell topology with a single congested link and varied the number of connections N from 10 to 30. All connections were modeled as "File Transfer Protocol (FTP)" flows, i.e., as constant and inexhaustible source of information. We used the following RED parameters.

- Queue weight given to current queue size sample =0.002
- Minimum threshold of average queue size (Th_{min}) = N-1
- Maximum threshold of average queue size (Th_{max}) = 3N
- Max probability of dropping a packet = 1/*linterm*, where *linterm* = 3

To verify our formulation we have also created a custom Monte Carlo simulation and implemented our time difference equations. We have performed simulation with parameters similar to those of the discrete event simulation. The average queue sizes were observed with both methods and are depicted in figure 4.1; curves with a DE suffix are results of the discrete event simulations (NS2) while curves with an MC suffice show the Monte Carlo results (analytic). (Note that the time frame of these two simulations is not identical however we plot them on the same figure, i.e., their variation may not be on the same time scale). We can observe a good match, validating our mathematical model (note, that more extensive simulations have been performed with changes in other parameters and those results underline this conclusion as well).

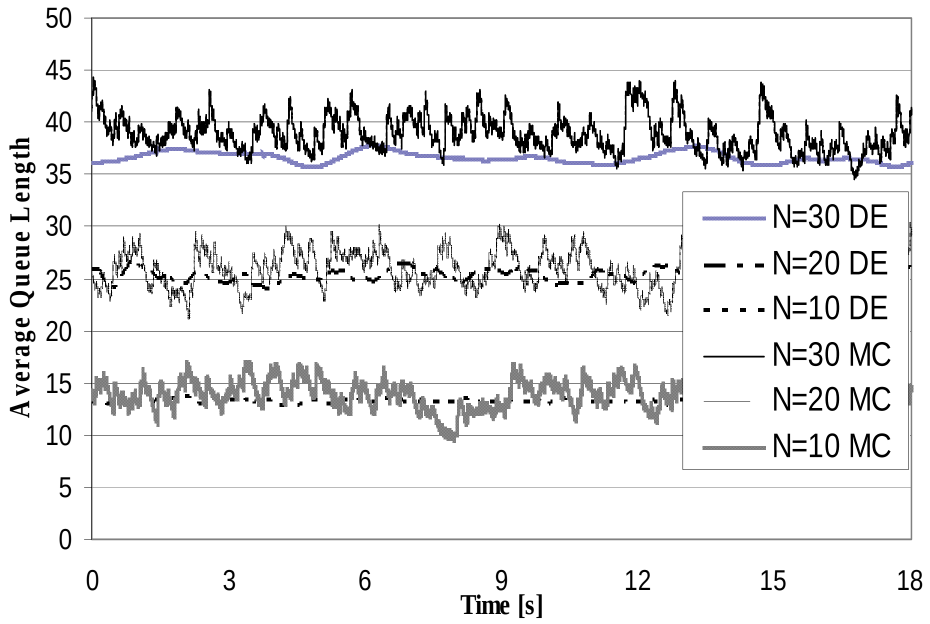

Figure 4.1 Average queue length at a RED-RWM queue

4.5 Summary

In this chapter, we have presented a mathematical model for the RED-RWM queue by extending the work done previously for RED-ECN queues. We have verified our model by comparing Monte Carlo simulations of our model to discrete event simulations of an NS2 model. We "theoretized" that the RWM modified RED queue will weakly converge to a steady state and that as the number of clients (TCP connections) grows, the queue size will weakly converge to the number of clients, assuming that the minimum marking threshold (Th_{min}) is less than the number of clients.

CHAPTER 5

ReCHOKe

In this chapter, we are proposing a scheme called RECHOKe (REpeatedly CHOose and Keep for responsive flows, REpeatedly CHOose and Kill for non-responsive flows) to be used for detecting, controlling and punishing malicious flows in TCP/IP networks. It is an extension of xCHOKe [35][36], CHOKe [33] and RED-PD [37] schemes, combining both the CHOKe hit and RED [7] Drop/Mark histories, to detect, control and punish these flows more accurately while providing better protection to non-malicious flows. However, unlike xCHOKe and CHOKe, it does not drop packets during CHOKe hits; thereby eliminating the complexity of dropping or marking randomly selected packets already queued. We also analyze xCHOKe and RECHOKe in detail by using NS2 and show that RECHOKe performs better than RED, CHOKe and xCHOKe which are limited in what they can achieve since malicious flows still get much more than their fair share and non-malicious flows get mistakenly penalized.

CHOKe, xCHOKe and RECHOKe are preferential dropping schemes that have been proposed or are being proposed for detection, control and punishment of malicious flows at routers in IP networks. They use CHOKe hits, CHOKe misses and/or CHOKe-RED drops to carry out these tasks. In this chapter we also investigate the accuracy of malicious flow detection by using these hits, misses and drops (using NS2). We also point out the unreliability of CHOKe hits and misses, when compared to CHOKe-RED drops, as they affect TCP-friendly flows adversely. By doing so, we present two ad hoc enlightening variations of CHOKe called Half1- and Half2-CHOKe to improve CHOKe and compare them with CHOKe. As we will show Half1- and Half2-CHOKe outperform CHOKe when the combined rates of malicious flows are less or greater than the link capacity respectively. These two variations are used by RECHOKe to detect, control and punish malicious flows.

The rest of this chapter is organized as follows. The next section outlines our solution while Section 5.2 presents simulation results and compares the RECHOKe scheme to several proposed AQM schemes. Finally, Section 5.3 concludes the chapter.

5.1 Description of RECHOKe

This section presents the functional description of RECHOKe; a flowchart of RECHOKe is depicted in figure 5.1. If the average queue length (*avq*) is smaller than the minimum threshold, Th_{min}, then the arriving packet is accepted. If the *avq* is greater than Th_{min} but smaller than Th_{max}, the lookup table is checked to see whether the arriving packet's flow label is present there. If it indeed is (called a table hit), the arriving packet is marked for dropping and the associated hit counter is incremented. A packet is then selected at random from the queue and its flow label is compared with that of the arriving packet. If the flow labels are the same (called a CHOKe hit), the flow label is added to the lookup table with an initial value of 1 for an associated hit counter. If the flow is already in the table, the associated hit counter is incremented. Unlike in xCHOKe, packets are not dropped or marked for dropping as a result of a

CHOKe hit. They are allowed to enter the FIFO buffer. This removes the complexity of dropping the random packet after it has been admitted into the buffer and the unreliability of CHOKe hits. To remove this complexity from CHOKe, the authors in [33] have proposed to add one extra bit to the packet header so that the bit is set to one if the random packet is to be dropped. When this packet advances to the head of the RED buffer, the status of this bit determines whether it is to be immediately discarded or transmitted on the outgoing line. We feel that this idea leads to a wasteful use of valuable buffer space and the addition of complexities due to the 1-bit modification to the present-day packet header and the need for the routers to check each and every packet on the outgoing line.

If the packet is dropped or marked (in the presence of ECN [5][6]) by RED (called a RED hit), then its label is added to a lookup table with an initial value of 1 for an associated hit counter. If the flow is already in the table, the associated hit counter is incremented.

If avq is greater than Th_{max}, then the packet is dropped and its label is added to a lookup table with an initial value of 1 for an associated hit counter. If the flow is already in the table (also called a table hit), the associated hit counter is incremented. The hit counter n of an entry in the lookup table is used to compute the probability p^* and a boolean value c^* with which an incoming packet with the same flow label as a table entry will be dropped. These values are computed as follows:

$$p^* = \min(1, p_a \times 2^n) \quad \text{and} \quad c^* = f\left(n \geq \frac{\sum_{i=1}^{m} n_i}{(2 \times m)} \right)$$

where p_a is RED's dropping probability and m is the number of entries in the table. If c^* is true then the packet will be marked or dropped with probability p^*. The purpose for c^* is that we do not want to punish a TCP-friendly flow, whose previous packets just "accidentally" happened to experience a CHOKe hit leading to one of its recent packets experiencing a table hit. This scenario is common in CHOKe and xCHOKe, which results in TCP-friendly or responsive flows getting punished unnecessarily.

Hence, in RECHOKe, we update the table when an incoming packet has:

i) hits in the lookup table (table hits),

ii) hits with a randomly chosen packet from the buffer (CHOKe hits), or

iii) it has been selected to be dropped or marked by RED (RED hits).

The first two cases are similar to xCHOKe, while the third is similar to RED and RED-PD behavior. We can use RED drop or mark history in the presence of ECN. In Section 5.2.1, we will analyze the RED buffer to verify how accurate these RED drops are in identifying malicious flows. The unreliability of CHOKe hits is mentioned in [35][36] but it is not analyzed. Hence, in Sections 5.2.2 and 5.2.3, we carry out studies of CHOKe and CHOKe-RED to verify their accuracy and to find ways to improve their accuracy. To improve on CHOKe and xCHOKe, we use two new versions of the CHOKe algorithm. We call them Half1 and Half2. In Half1 CHOKe, the packets are randomly chosen from the first half of the buffer while in Half2 CHOKe, they are randomly chosen from the second half of the buffer. RECHOKe starts off with the original CHOKe algorithm, and then it samples the first quarter of the buffer with the entries in the table at *t* ms. It computes the occupancy percentage at the first quarter of the occupied queue by the entries in the table. If it is greater than 30%, then RECHOKe uses Half1-CHOKe, otherwise it uses Half2-CHOKe (see Section 5.3 for more details).

In RECHOKe, we associate a time-to-live (TTL) with each entry in the lookup table. However, TTLs are not refreshed only when an incoming packet has:

i) table hits,

ii) CHOKe hits, or

iii) RED hits.

The first two cases again are similar to xCHOKe while the third case is unique to RECHOKe.

5.1.1 Complexity Cost

The computational complexity for RECHOKe is limited to:

i) a hash lookup in the table for each packet which takes a constant time,

ii) the computation of p^* and c^* if there is a table hit for that packet, and

iii) the sampling of the first quarter of the occupied queue at *t* ms.

The space complexity of RECHOKe is the size of the lookup table which can be either implemented as a variable table which grows as needed to approximately the order of magnitude of malicious flows or based on a Least Recently Used or Least Frequently Used scheme [55].

5.1.2 Theoretical RECHOKe Model

The approximate analysis for RECHOKe is an extension of the analysis for xCHOKe [35] where the system is modeled as a queue with a FIFO queuing discipline and assumes that each source is independent and identically distributed, and the arrival process is Poisson. Likewise, we denote the arrival rate of the *i*th source as λ_i and the mean service time for each packet as $1/\mu$.

An incoming packet may suffer a hit in three phases of the RECHOKe algorithm:

i) the flow label of the packet is in the lookup table (table hit),

ii) the flow label of the packet matches that of a randomly drawn packet from the buffer (CHOKe hit) and

iii) a RED drop (RED hit).

We will use a similar notation to [35], where

1) P_{TABLE} denotes the probability that an incoming packet will suffer a table hit,

2) P_{CHOKe} denotes the probability that the packet will suffer a CHOKe hit, and

3) P_{RED} denotes the RED drop probability.

Let us denote the number of Table, CHOKe and RED hits for a table entry by *n*. If the incoming packet's flow label is in the lookup table, then the probability of dropping an incoming packet on a table hit can be given by

$$p^* = \min(1, P_{RED} \times 2^n) \times c^*$$

where n is a function of P_{TABLE}, P_{CHOKe} and P_{RED} and c* is Boolean function represented by

$$c^* = f\left(n \geq \frac{\sum_{i=1}^{m} n_i}{(2 \times m)} \right)$$

where *m* is the number of entries in the table.

Assuming that no TTLs are refreshed, the maximum value of *n* can be approximated by:

$\lambda_i * TTL * (P_{TABLE} + P_{CHOKe} + P_{RED})$

where TTL is the initial time to live value for an entry in the lookup table. This is greater than xCHOKe's

$k = \lambda_i * TTL * P_{CHOKe}$

This is reflected in figure 5.38 implying the identification process is more rigorous in RECHOKe than in xCHOKe. Hence, the chances of the malicious flow's packet getting dropped are greater in RECHOKe than in xCHOKe. The probability that a packet is not dropped by the RECHOKe algorithm is

$P_{NoDrop} = (1 - P_{TABLE} \times P_{RED} \times 2^n \times c^*) \times (1 - P_{RED})$, thus:

$$P_{NoDrop} = (1 - P_{TABLE} \times P_{RED} \times 2^n \times f\left(n \geq \frac{\sum_{i=1}^{m} n_i}{(2 \times m)} \right)) \times (1 - P_{RED}) \qquad 5.1$$

xCHOKe's P_{NoDrop} can be given by:

$$P_{NoDrop} = (1 - P_{TABLE} \times P_{RED} \times 2^{k}) \times (1 - P_{CHOKe}) \times (1 - P_{RED}) \qquad 5.2$$

The value of P_{NoDrop} in equation 5.1 is smaller than in equation 5.2, since at any instant of time, $n > k$ and if c^* is one, then it implies that the chances of dropping packets from malicious flows is greater for RECHOKe than for xCHOKe. This is reflected in figures 5.39 and 5.42. If c^* is zero, it implies that the chances of dropping packets from low-bandwidth flows is greater for xCHOKe than for RECHOKe. This is reflected in figure 5.40 and 5.43.

5.2 Simulation Results and Analysis

This section presents the experiments and analysis to verify the accuracy of CHOKe hits, CHOKe misses and CHOKe-RED drops/marks. We have also performed experiments on RED, CHOKe and xCHOKe and compare them to RECHOKe.

Using NS2, we simulate the bar-bell topology shown in figure 5.2. Here, we analyze the effects on the RED, CHOKe, xCHOKe and RECHOKe buffers in terms of buffer occupancy by a malicious flow, a constant-bit-rate UDP flow, and competing with 10 TCP flows over a 1-Mbps R1-R2 link. All TCP flows have the same round trip propagation delay of 20ms with each output link having a latency of 1ms and capacity of 10Mbps. The parameters for the RED, CHOKe, xCHOKe and RECHOKe buffers are: $Th_{min} = 10$, $Th_{max} = 50$, $P_{max} = 0.1$.

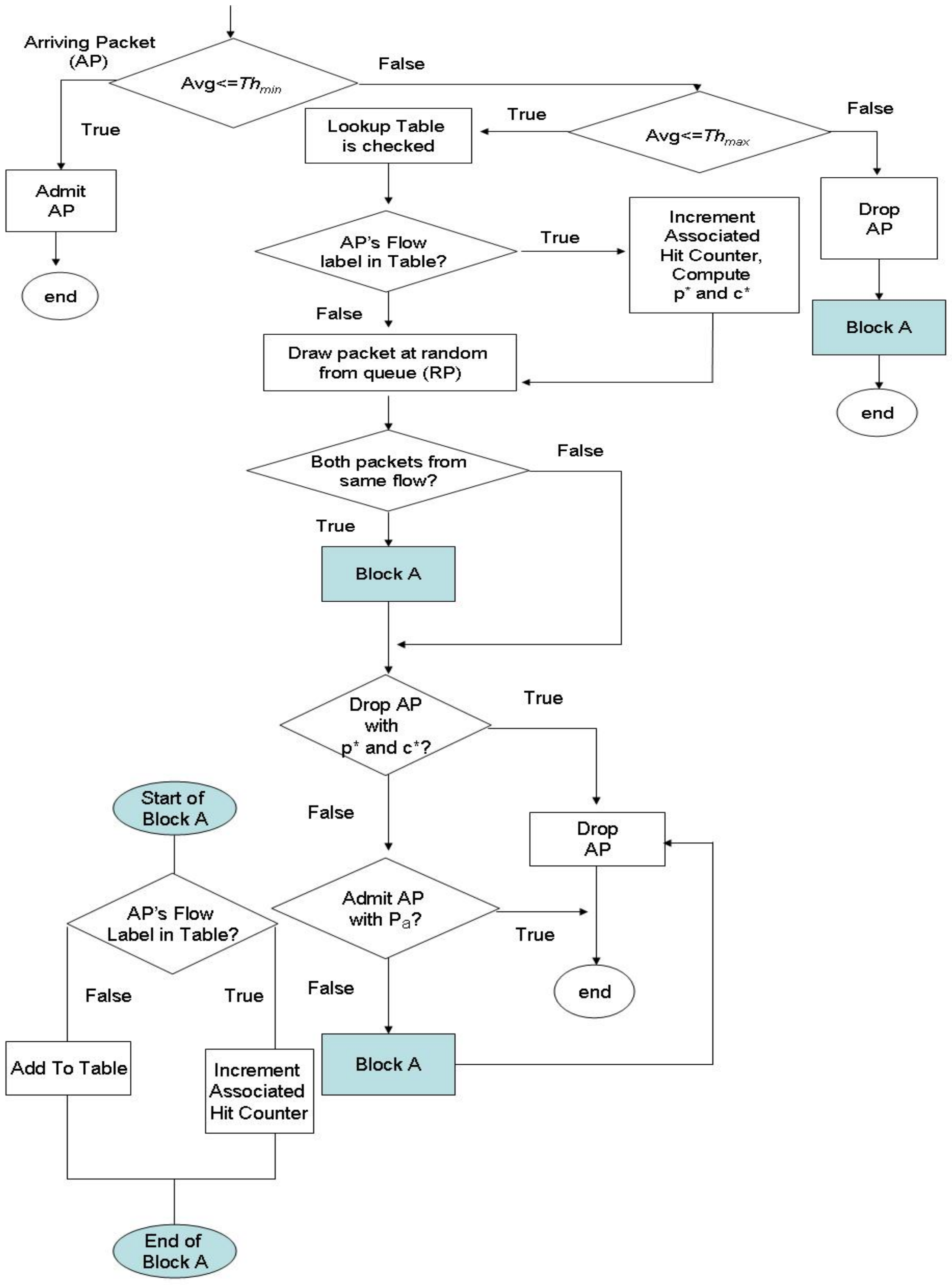

Figure 5.1 The flowchart of RECHOKe

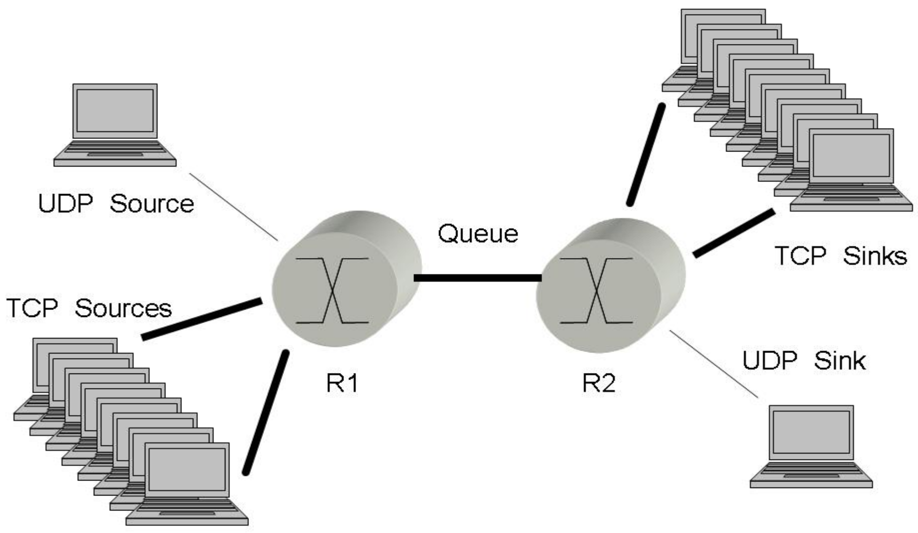

Figure 5.2 Simulation Network Topology

We ran 11 simulation sets, starting with a 0.5 and then 1Mbps UDP flow rates and then increasing the rate of the malicious flows in 1Mbps increments. Each simulation was long enough for initial transients to settle and be insignificant. If simulation results do not have a time axis, then we have run enough simulation instances to claim a 95% confidence that our results are no more than 5% off.

5.2.1 Experiments on RED

In order to analyze the RED buffer to verify how accurate RED drops are in identifying malicious flows, we divide RED Drops into *forced* and *unforced drops*. Forced drops occur when the average queue length (*avq*) of the RED buffer is greater than Th_{max}, whereas unforced drops occur when $Th_{min} < avq < Th_{max}$.

To analyze the RED buffer, we define the following terms. RED victim: the packet that is randomly selected to be dropped from the FIFO buffer. A RED hit occurs when the RED victim represents the flow whose packets dominate the buffer at the time of selection of the RED victim. This can be either a good (proper) or a bad RED hit. Good RED hit: during a RED hit, the RED victim belongs to a malicious flow. Bad RED hit: during a RED hit, the RED victim belongs to a non-malicious (or TCP-friendly) flow. It is a bad hit because we are punishing the non-malicious flow by dropping one of its packets. A RED miss occurs when the RED victim does not match the flow whose packets are dominating the buffer at the time of selection of the RED victim; this can again be a good or bad RED miss. Good RED miss: during a RED miss, the RED victim belongs to a malicious flow. Bad RED miss: during a RED miss, the RED victim belongs to a non-malicious flow.

Figures 5.3 and 5.4 show the percentages of RED hits and misses for unforced and forced drops respectively. As it can be observed, the proportion of hits is greater in both figures when compared to that of the misses. This implies that the RED victim, whether it is due to a forced or unforced drop, belongs to the same flow whose packets dominate the RED buffer. To find the type of RED hits, we analyze both the good and bad RED hits for both types of drops. We have found that nearly all the RED hits are good hits for both types of drops. This means that the RED victims belong to the malicious flow and the malicious flow represents the dominant flow whose packets are occupying the buffer. Hence, this shows that the probability that a gateway picks a

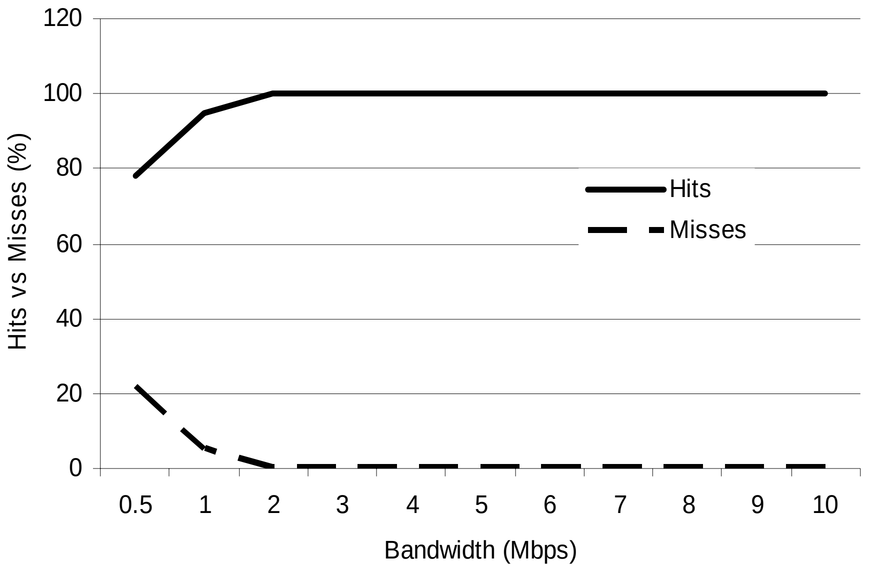

Figure 5.3 The percentage of RED hits and misses for unforced drops

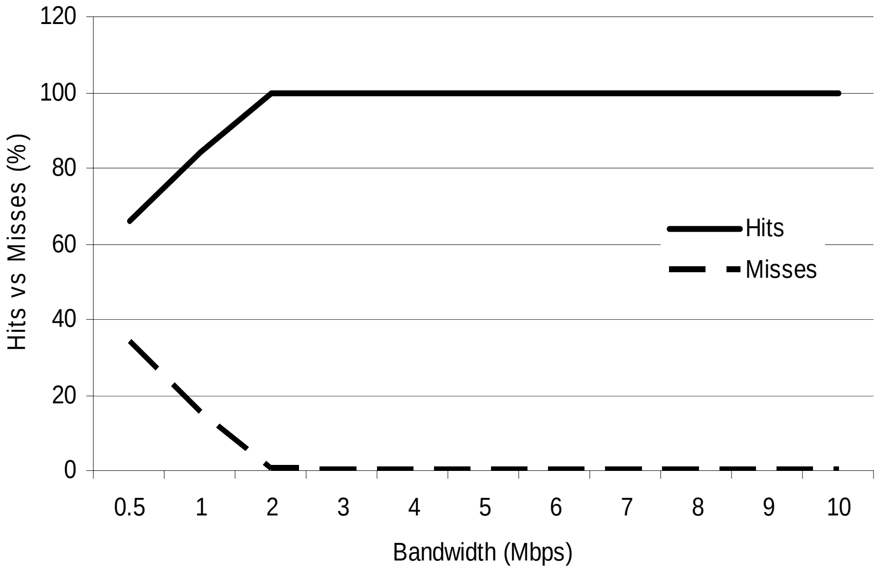

Figure 5.4 The percentage of RED hits and misses for forced drops

particular flow to mark or drop during congestion is roughly proportional to that flow's share of the bandwidth at the router.

In both cases, there are prominent RED misses for 2 simulations, which occur when the rates of the UDP flow are at 0.5 and 1Mbps. Figures 5.5 and 5.6 show the number of good and bad RED misses for both types of drops. When the rate of the UDP flow is at 0.5Mbps, we have several good misses, implying that the RED victim, belonging to a malicious flow, did not match the flow whose packets dominate the buffer at the instance of selection of the victim packet. On the other hand, the number of bad misses (however very small, compared to those of good RED hits) gets reduced when the rate of the UDP flow increases for both types of drops. This means that the number of instances when the RED victim belongs to a non-malicious flow (and the non-malicious flow itself did not represent the dominant flow whose packets are occupying the buffer) are very small.

With the RED experiments, we have shown that we can easily identify malicious flows by keeping track of the RED drops since the number of good RED hits is far greater than the number of RED misses and bad RED hits. The problem with RED hits is that they only increase as congestion increases. Hence, they do not control malicious flows on occasions when the congestion is light. They cannot be used alone to accomplish the goal of detecting, controlling and punishing malicious flows.

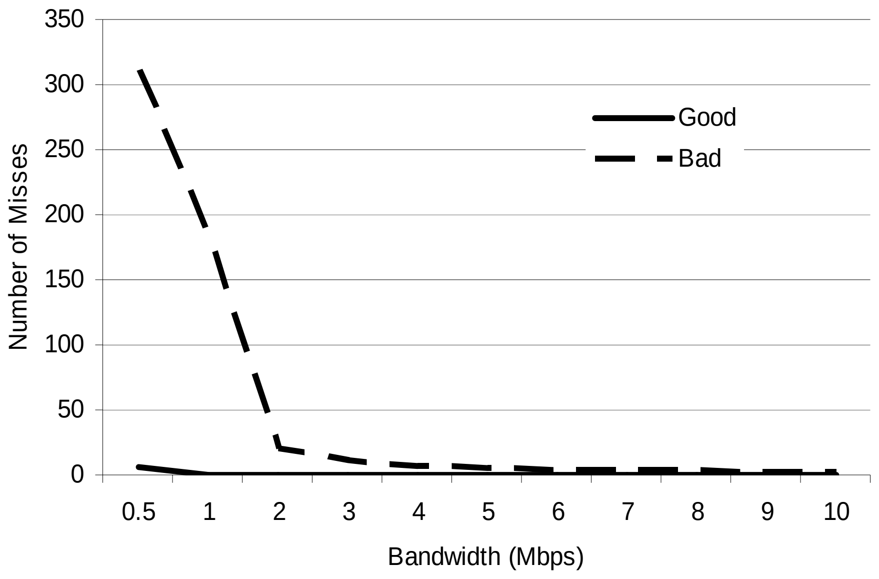

Figure 5.5 The number of RED good and bad misses for unforced drops

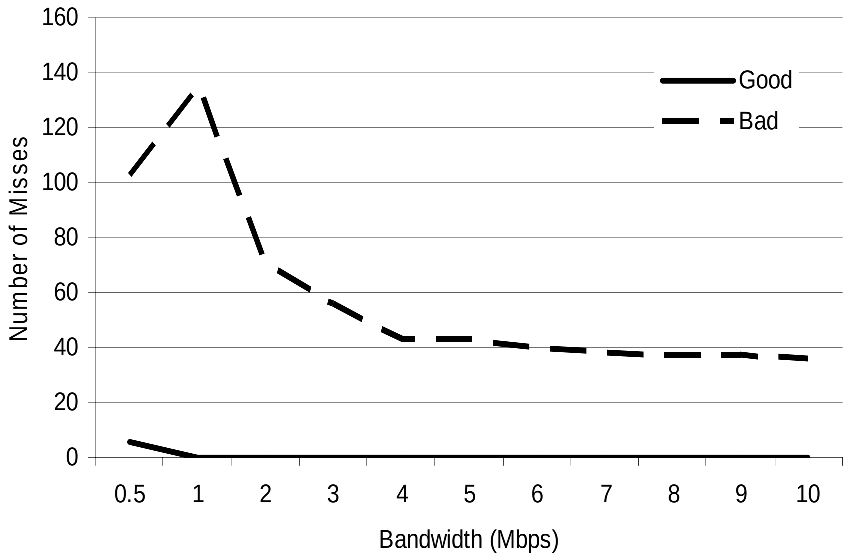

Figure 5.6 The number of RED good and bad misses for forced drops

5.2.2 Experiments on CHOKe

In this section, we study the CHOKe buffer by dividing the occupied portion of the buffer into four quarters to observe the following:

i) the percentages of hits and misses, good and bad hits and misses,

ii) the percentages of buffer occupancy within each quarter by the malicious flow for every packet arrival (i.e. in our experiment we use UDP flows as malicious flows),

iii) the position of the victim packet within the buffer that are responsible for the hits and misses,

iv) the relationship between the variation of the UDP rate with a constant R1-R2 link and the buffer occupancy by the UDP flow in each quarter of the buffer.

To analyze the CHOKe buffer, we define the following. CHOKe victim: the packet that is randomly selected from the FIFO buffer to be compared with each arriving packet. CHOKe hit occurs when the flow label of the CHOKe victim matches that of the arriving packet. A CHOKe hit can be either a good or bad CHOKe hit. Good CHOKe hit occurs when the arriving packet belongs to a malicious flow and hence, the CHOKe victim is also from that malicious flow. Bad CHOKe hit occurs when the arriving packet belongs to a TCP-friendly flow and hence, the CHOKe victim is also from that particular flow. CHOKe miss: this occurs when the CHOKe victim does not match the arriving packet. A CHOKe miss can be either a good or bad CHOKe miss. Good CHOKe miss occurs when the arriving packet belongs to a TCP-friendly flow and the CHOKe victim is from any other but that particular flow. Bad CHOKe miss occurs when the arriving packet belongs to a malicious flow and the CHOKe victim is from any other but that particular flow.

Figure 5.7 shows the percentages of CHOKe hits and misses. We can observe that when the UDP flow is less than three times the link capacity, the percentages of misses is about 5-40% greater than the percentages of hits. However, they drop rapidly from 68% to 53% when the UDP flow rate is increased from 0.5Mbps to 2Mbps. Increasing the rate of the UDP flow further only decreases the percentages of misses to about 3-7%.

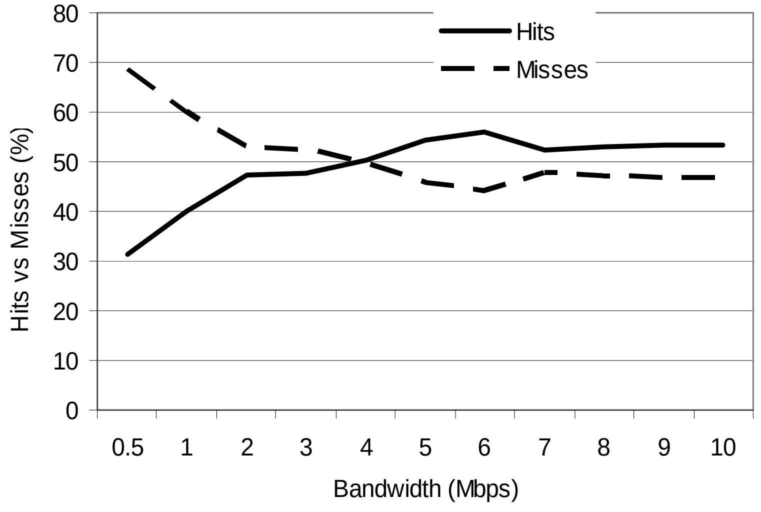

Figure 5.7 Hits vs. misses

As for hits, its percentages increase rapidly from 31% at 0.5Mbps to about 47% at 2Mbps. After 2Mbps, the increase hovers around 1-8% and improves only about 1-

6% over that of the misses. This increase is small compared to that of the UDP rate. The increase in its rate should imply that more UDP packets should occupy the buffer enabling more hits and less misses. However, figure 5.7 shows that this was not the case.

Figure 5.7 presents an interesting phenomenon. To investigate this, we divide the packets within the buffer into 4 quarters, where the first quarter is at the head of the queue. We call these quarters as Qtr1, Qtr2, Qtr3 and Qtr4. Figure 5.8 shows the results for this experiment. The tail quarter UDP occupancy percentage increases from 63% to 98% as the UDP rate is varied whereas Qtr3 increases from 41% to 70%. In contrast the head quarter's occupancy decreases rapidly from 42% to about 20% whereas those of Qtr3 are changing between 48% and 63%. The high UDP buffer occupancy percentages at the tail of the queue are expected since the UDP packets arrive at the tail section of the buffer. However, this results in choking TCP flows. The UDP flows advance in the queue is tapered off slowly due to the effects of the CHOKe hits. The head of the queue has the least UDP buffer occupancy percentages since it is at the head of the queue where the packets leave the buffer and it shows that CHOKe has been successful in reducing the number of packets in the buffer. However, overall, as shown in figure 5.8 CHOKe does not lead to an overall increase in TCP packets in the buffer. The explanation is that CHOKe also affected the TCP-friendly flows by dropping TCP packets with bad misses (as shown in figures 5.9 and 5.10) and bad hits (as shown in figure 5.11). Although bad hits are small in numbers, they are effective in forcing the

TCP sources into reducing their rates. On the other hand, since the number of bad misses is large, UDP packets tend to occupy more buffer space than they should.

An interesting observation is that the greater the rate of the malicious flows, the greater the buffer space they occupy at Qtr4. If there are more than one, let say, two UDP flows, one whose rate is at 10Mbps while another at 2Mbps. Then the packets from the UDP flow at 10Mbps will occupy greater buffer space at Qtr4 than those from the UDP flow at 2Mbps. The reverse is true at the head of the queue.

Another observation is that the percentage at the head of the queue can provide a good upper bound in estimating the bandwidth share of the flows using the buffer. Here, in figure 5.8, Qtr1 gives an upper bound of about 42% of the bandwidth used by the UDP flow when its rate is at 0.5Mbps.

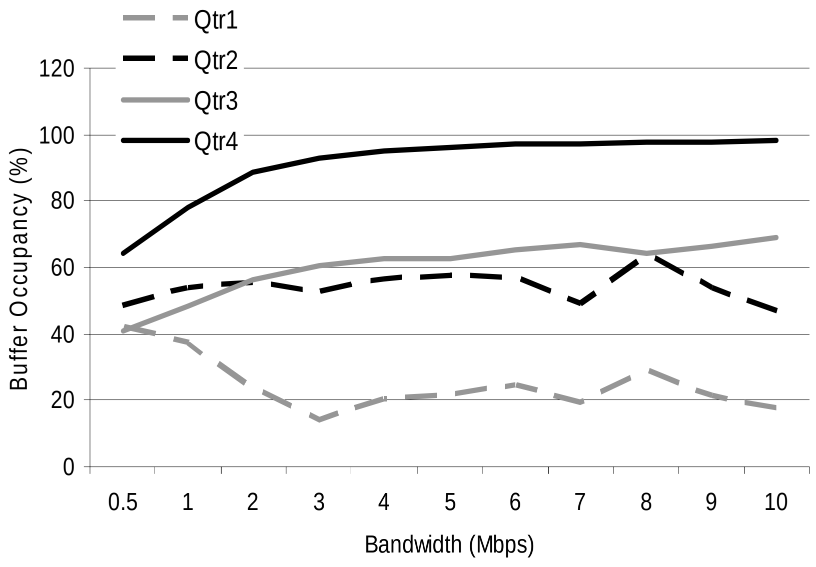

Figure 5.8 The percentage of average buffer occupancy by the UDP flow during CHOKe hits

Figure 5.9 shows CHOKe's ratio of bad misses over good misses. At 0.5Mbps UDP rate the ratio is at 1.4 implying that the number of bad misses is only slightly higher than those of good misses, meaning that the number of arriving packets, belonging to the malicious flow and the victim that are chosen are from any other but that particular flow, is greater than the number of arriving packets, belonging to a TCP-friendly flow and the victim are chosen are from any other but that particular flow. However this ratio increases rapidly with a growing UDP transmission rate. The growing UDP rate should imply that instead of more bad misses, there should be a greater number of good hits. Hence, there should be less bad hits as the rate of the UDP flow is increased as evident from figure 5.11. Furthermore, there should be a greater number of good misses since there are many more UDP packets in the buffer that should have been chosen as victims to be compared with the incoming TCP packets. *However, less good misses imply that TCP flows have encountered choke hits.* This behavior has an adverse effect on the throughput of TCP flows. In fact, figure 5.9 should ideally be a negatively sloped plot. Figure 5.10 shows the similar results on a different scale; obviously the victims are not chosen appropriately to ensure more good hits and good misses.

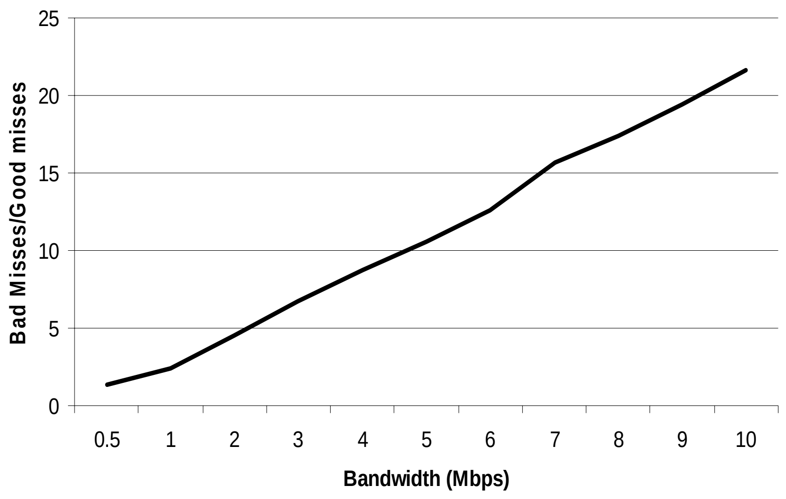

Figure 5.9 CHOKe’s ratio of bad misses over good misses

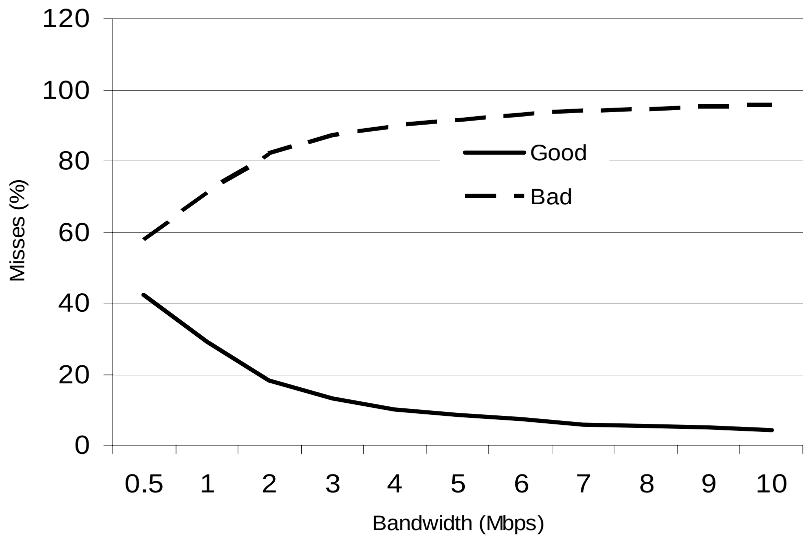

Figure 5.10 The percentage of bad and good misses

Figure 5.11 shows CHOKe's ratio of bad hits over good hits. At low UDP rates this ratio is around 0.05 implying that the number of good hits is about 14 times greater than those of bad hits. This means that the conditions favoring that both the arriving packets and their randomly picked victims are from the same TCP-friendly flows are far smaller than the conditions favoring that both arriving packets and their randomly picked victims are from the UDP flow. This makes sense since as the rate of UDP flows increase, the number of good hits should also increase as more UDP packets are occupying the buffer. However, the number of bad hits (although small) has an adverse effect on the throughput of the TCP flows.

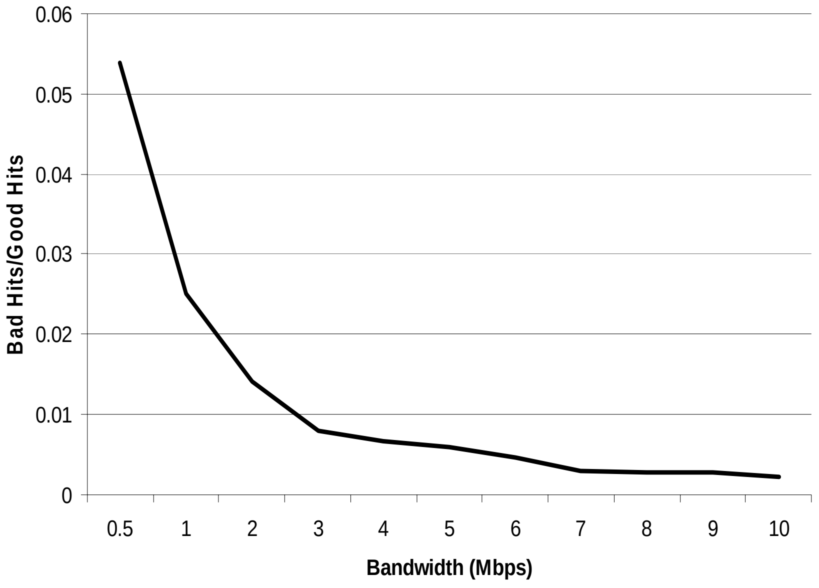

Figure 5.11 Ratio of bad and good hits against the UDP's bandwidth

To verify the effects of CHOKe hits on the buffer in terms of reducing the rate of malicious flows, we measure the percentages of average buffer occupancy of the UDP flow at each buffer quarter during CHOKe hits for every 10s of a 30s simulation. The percentages are shown in figures 5.12, 5.13 and 5.14 respectively. High UDP buffer occupancy percentages at the tail of the queue are expected since UDP packets arrive at the tail section of the buffer. However, this may also result in choking TCP flows. As UDP flows advance in the queue they are tapered off slowly due to the effects of CHOKe hits. The head of the queue has the least UDP buffer occupancy percentages showing that CHOKe has been successful in reducing the number of packets in the buffer. Although there are some reductions before the first 10 seconds, the average UDP occupancy in Qtr1 is between 22 and 58% as the UDP rate is varied. After the first 10 seconds, increasing the duration of the flows has no big effect on the outcome. The reduction in Qtr1 reflects the reduction in the bandwidth of the UDP flow but the unresponsiveness in the reduction is due to bad misses (as shown in figures 5.9 and 5.10) and bad hits (as shown in figure 5.11).

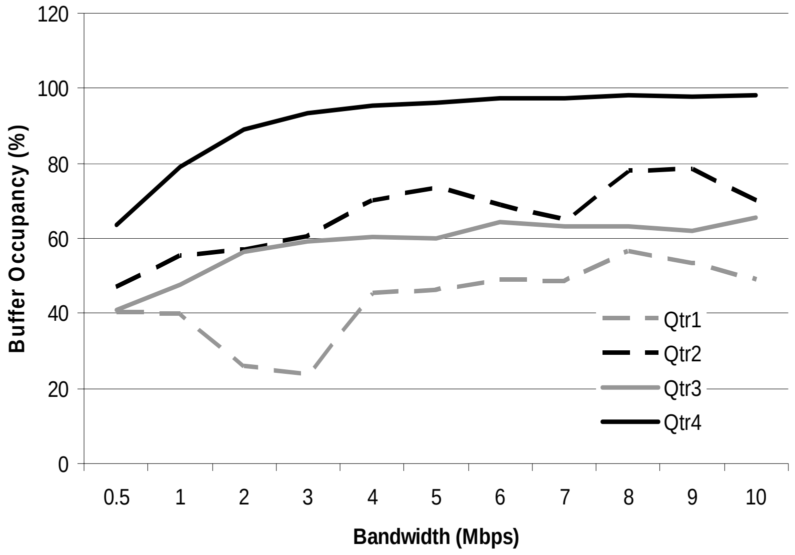

Figure 5.12 The percentages of average buffer occupancy by the UDP flow during CHOKe hits (0-10s)

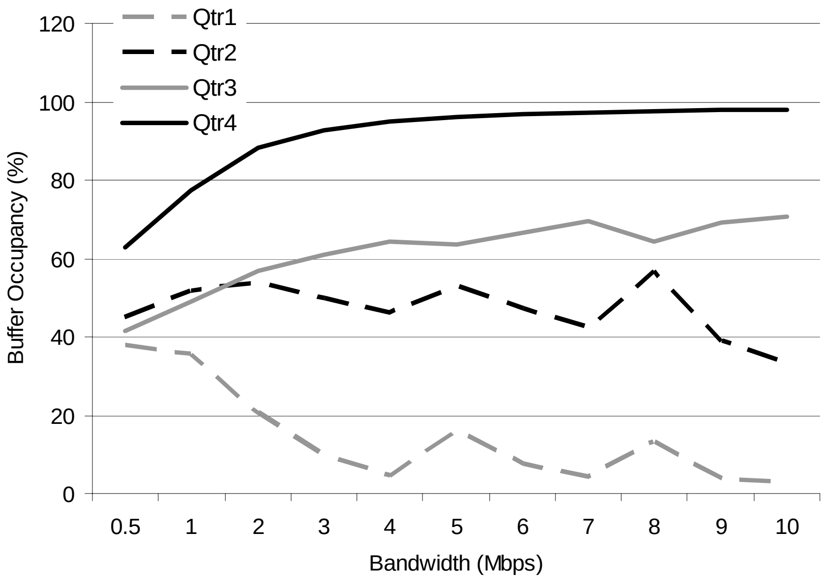

Figure 5.13 The percentages of average buffer occupancy by the UDP flow during CHOKe hits (10-20s)

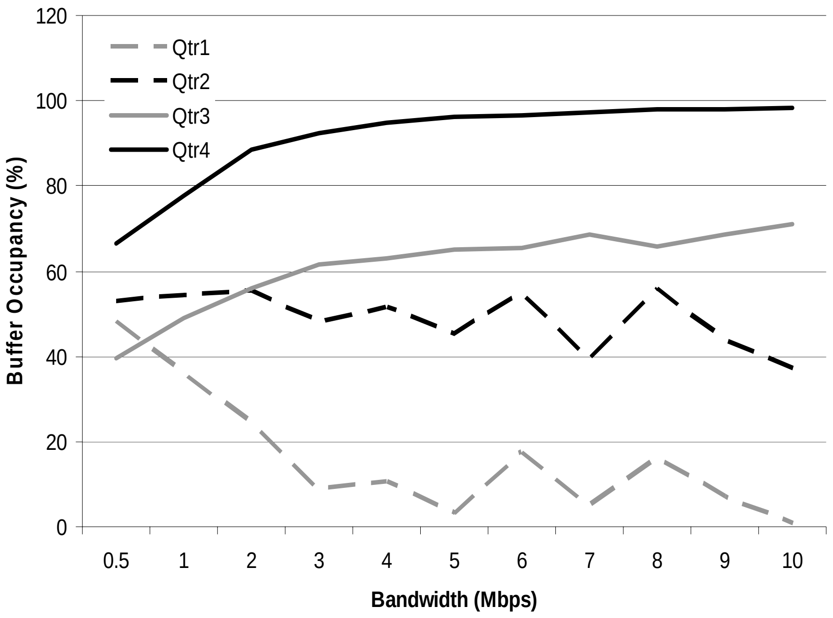

Figure 5.14 The percentages of average buffer occupancy by the UDP flow during CHOKe hits (20-30s)

Figure 5.15 shows the average UDP buffer occupancy percentages during CHOKe misses. The queue's last quarter's average UDP occupancy percentages increase from 46% to 82% compared to the 63% to 98% during CHOKe hits. In fact, as expected, all 4 quarters show reduced average UDP buffer occupancy during CHOKe misses when compared to cases with CHOKe hits. Of all the quarters, Qtr4 reduces the least. This is mainly due to the fact that most of the random packets are chosen from the first 3 quarters, leading to a large number of misses. As is in figure 5.8, similar observation is observed with respect to Qtr4, that is, the greater the rate of the UDP flow, the more buffer space it will occupy in Qtr4.

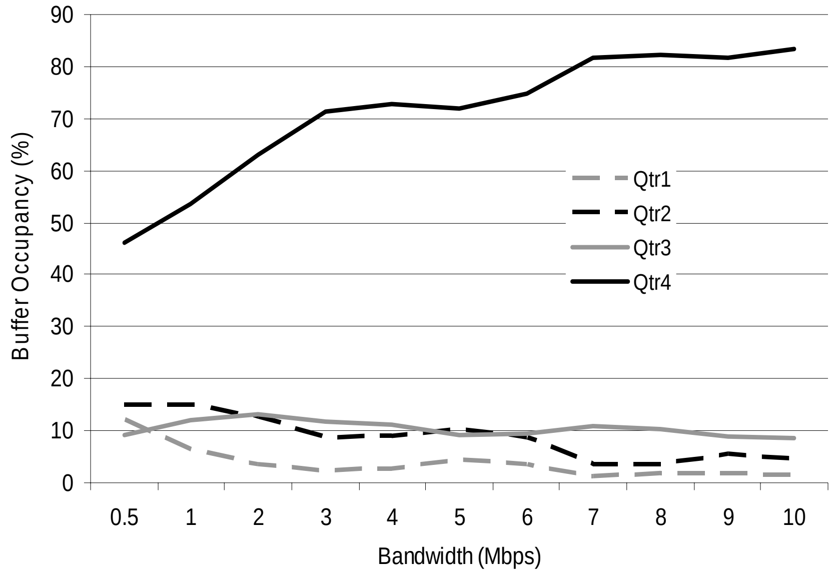

Figure 5.15 The percentages of average buffer occupancy by UDP flow during CHOKe misses

We then plot the average percentage of buffer occupancy by the UDP flow at each quarter during CHOKe misses for every 10 seconds of a 30 second simulation. Figures 5.16, 5.17 and 5.18 show the average percentage of buffer occupancy by the UDP flow at each quarter during CHOKe misses for the first, second and third 10 seconds respectively. From the 3 figures, we notice that the average UDP buffer occupancy rate for Qtr3 and Qtr4 remains quite the same. Qtr1 reduces after the first 10 seconds, quite slowly due to the tremendous amounts of bad misses (as shown in figure 5.10). After the first 10 seconds, increasing the time for the simulations has an insignificant effect on the outcome as could be seen in figures 5.17 and 5.18. Although there were some reductions just before the first 10 seconds, the average UDP occupancy in Qtr1 was between 8 and 37% as the UDP rate was varied. Qtr2 reduced too but not as drastically as Qtr1. The reduction in Qtr1 reflected the reduction in the bandwidth of the UDP flow. Our aim is to reduce this occupancy it faster than what is achieved using CHOKe.

Hence, figure 5.8, and figures 5.12 to 5.18 indicate that it would make sense to choose the random packet from Qtr4, thereby increasing the chances of achieving good hits.

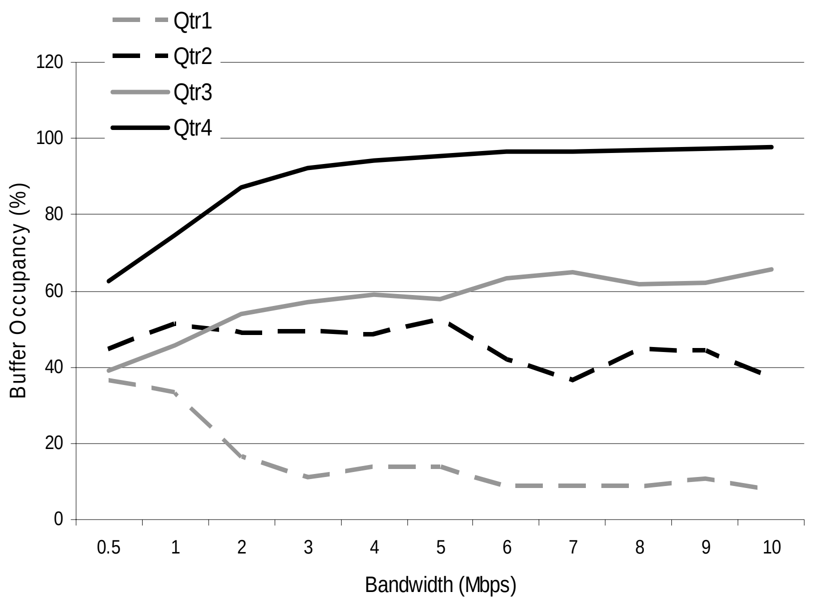

Figure 5.16 The percentage of average buffer occupancy by the UDP flow during CHOKe misses (0-10s)

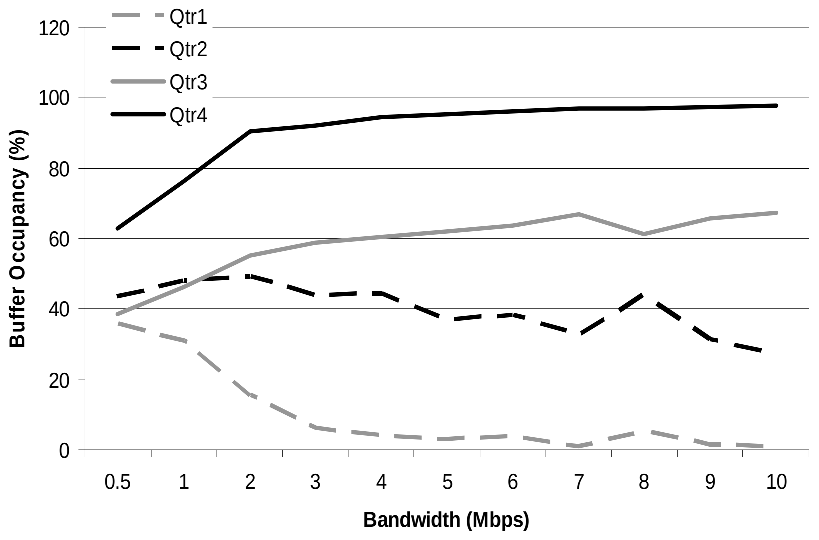

Figure 5.17 The percentage of average buffer occupancy by the UDP flow during CHOKe misses (10-20s)

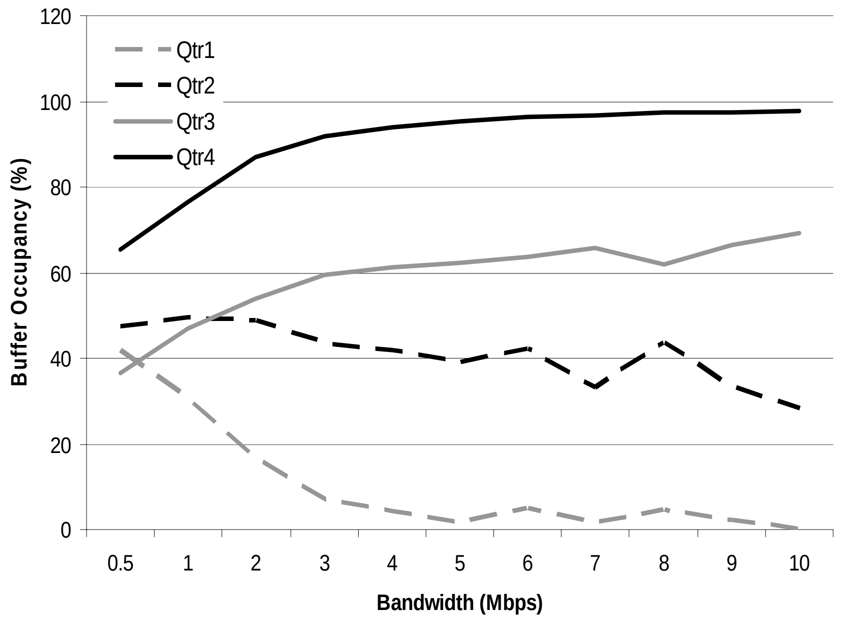

Figure 5.18 The percentage of average buffer occupancy by the UDP flow during CHOKe misses (20-30s)

Figure 5.19 shows the percentages of the location of victims within each quarter. As expected, victims were chosen almost proportionately from the 4 regions. Figures 5.20 and 5.21 show the percentages of location of victims within each quarter; when the UDP flow's rate is changed, during CHOKe hits and misses respectively. Figure 5.20 clearly shows that victims chosen from Qtr3 and Qtr4 increasingly matched the arriving packets as the rate of the UDP flow increased. The rate of increase, however, was more in Qtr 4 than Qtr3 since more UDP packets occupied the region (as shown in figure 5.8). We need to analyze these CHOKe hits to check whether they are good or bad CHOKe hits (as shown in figures 5.22 and 5.23). Figure 5.22 shows more good CHOKe hits for Qtr4 and Qtr3 as the rate is increases due to increasing occupancy of Qtr4 and Qtr3 by UDP packets. On the other hand, good CHOKe hits decline for Qtr2 and Qtr1 since there is a decreasing occupancy of Qtr2 and Qtr1 by UDP packets as the UDP rate increases. In figure 5.23, Qtr4 suffered more bad CHOKe hits when the UDP flow was at and around the R1-R2 link's capacity. As a result, the TCP flows suffered losses leading to low throughput resulting in a reduced number of bad CHOKe hits at the other quarters. As the UDP rate increased, the number of bad CHOKe hits in Qtr4 has reduced due to the large increase in UDP packets.

Hence, the number of bad CHOKe hits increases when

i) the number of flows using the buffer decreases,

ii) the rates of the TCP-friendly flows using the buffer increases, and

iii) the rates of the malicious flows using the buffer decreases.

On the other hand, the number of good CHOKe hits increases when

i) the number of flows using the buffer decreases,

ii) the rates of the TCP-friendly flows using the buffer decreases and

iii) the rates of the malicious flows using the buffer increases.

In figure 5.20, the victim hits are very small for Qtr1 since this quarter has the lowest number of UDP packets among the 4 regions (as shown in figure 5.8). Conversely, in figure 5.21, the victim misses are very high for Qtr1 since this quarter has the lowest UDP packets among the 4 regions. However, we need to analyze these victim misses again to check whether they are good or bad CHOKe misses (as shown in figures 5.24 and 5.25 respectively).

In figure 5.24, the percentages of good CHOKe misses are inversely proportional to the UDP rate. As the UDP rate is increased, the percentages of good CHOKe misses decrease. The percentages of good CHOKe misses for all the quarters are large at and around the link capacity since the number of packets from different flows using the buffer are greater at low UDP rates.

In figure 5.25, the percentages of bad CHOKe misses were directly proportional to the UDP rate for Qtr1 and Qtr2 while it was inversely proportional to the UDP rate for Qtr3 and Qtr4. This was mainly because of the concentration of UDP packets in Qtr3 and Qtr4 (as shown in figure 5.8). Hence, the number of bad CHOKe misses increases when

i) the number of flows using the buffer increases,

ii) rates of TCP-friendly flows using the buffer increase, and

iii) rates of malicious flows using the buffer decrease.

On the other hand, the number of good CHOKe misses increases when

i) the number of flows using the buffer increases,

ii) rates of TCP-friendly flows using the buffer decrease, and

iii) rates of malicious flows using the buffer increase.

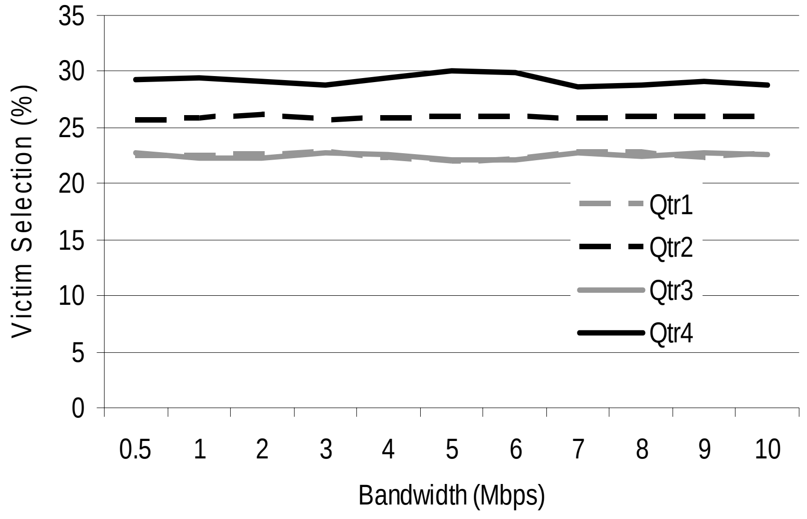

Figure 5.19 The percentages of location of victims within each quarter

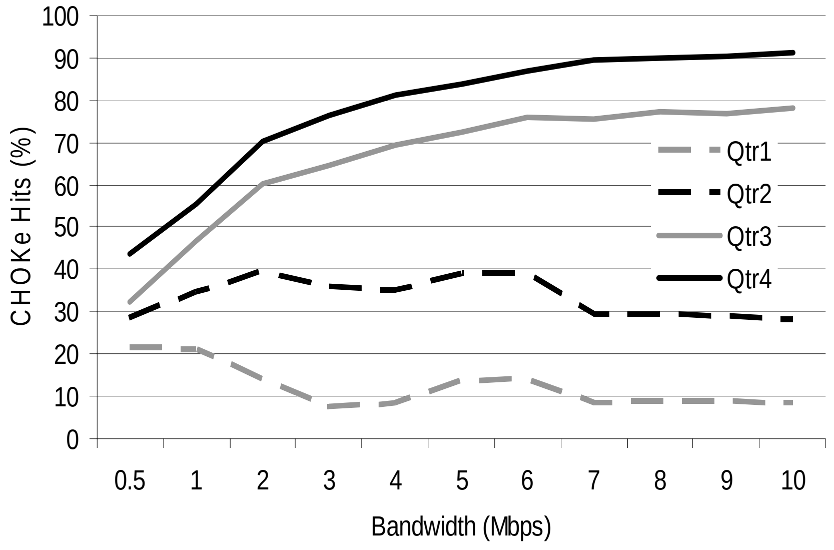

Figure 5.20 The percentage of location of CHOKe hits within each quarter

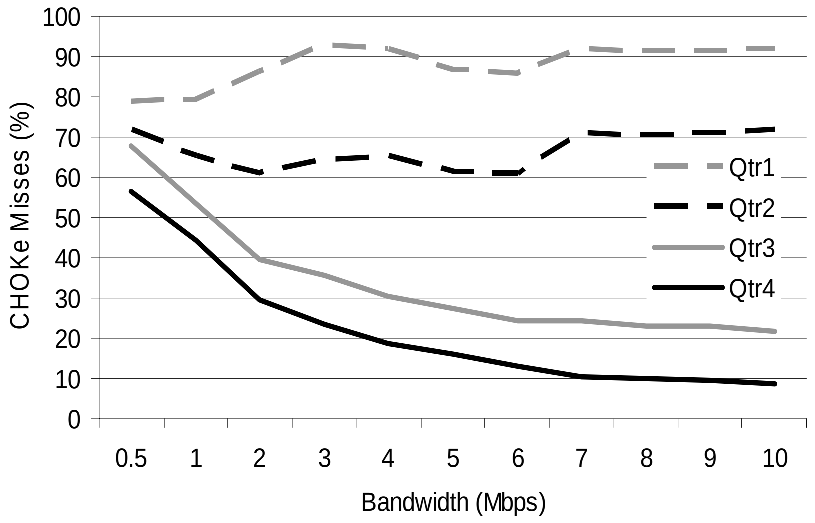

Figure 5.21 The percentage of location of CHOKe misses within each quarter

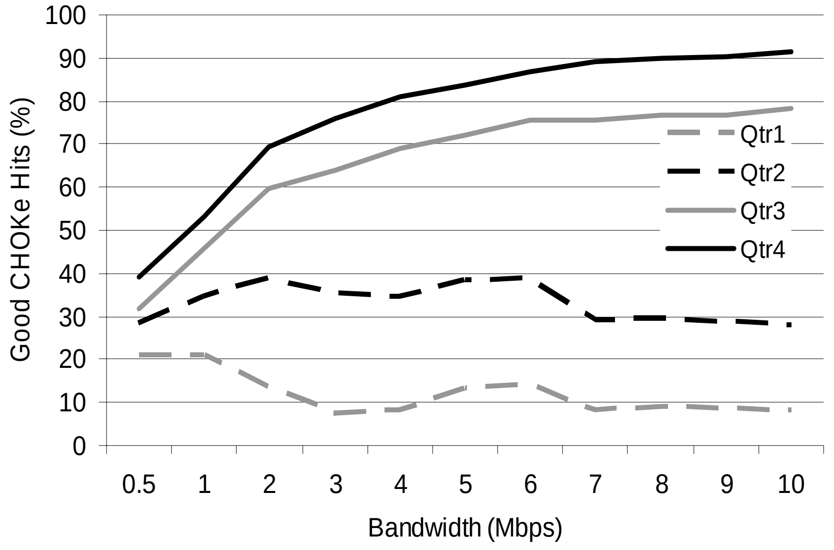

Figure 5.22 The percentage of location of good CHOKe hits within each quarter

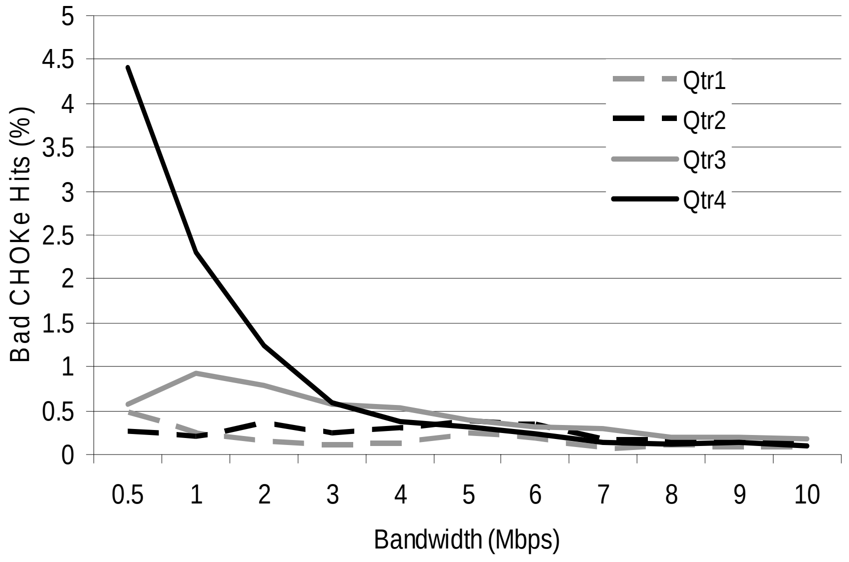

Figure 5.23 The percentage of location of bad CHOKe hits within each quarter

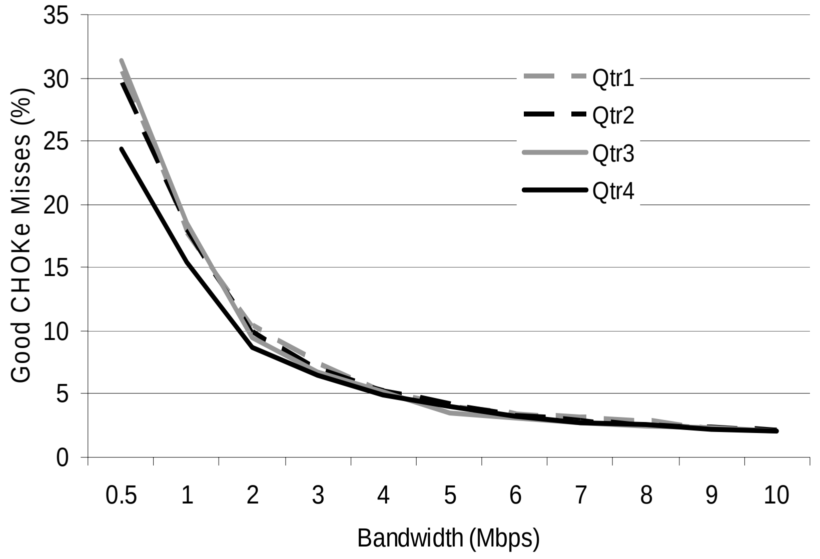

Figure 5.24 The percentage of location of good CHOKe misses within each quarter

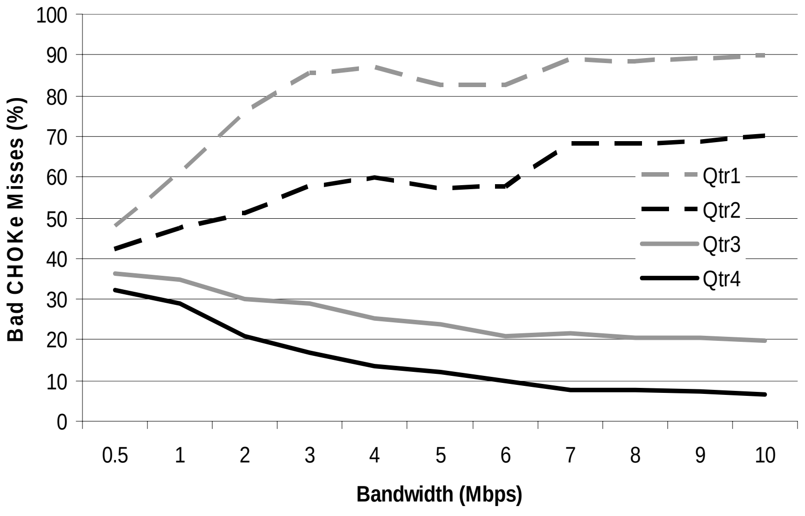

Figure 5.25 The percentage of location of bad CHOKe misses within each quarter

5.2.3 Experiments on CHOKe-RED

CHOKe victims that escape being dropped (Figure 5.7 - those involved in CHOKe misses) are received by the RED queue which either accepts or drops them. If $avq > Th_{max}$ then packets are dropped; if $Th_{min} < avq < Th_{max}$ then packets are dropped or marked probabilistically; if $Th_{min} > avq$ then packets are accepted. To analyze CHOKe-RED, we define the following terms for RED drops. CHOKe miss victim is a packet which after experiencing a CHOKe miss enters the RED queue. CHOKe miss RED hit occurs when the flow id of the CHOKe miss victim matches the flow whose packets dominate the buffer at the time of selection of the CHOKe miss victim. A CHOKe miss RED hit could be either a good or bad CHOKe miss RED hit. Good CHOKe miss RED hit: during a CHOKe miss RED hit, the CHOKe miss victim belongs to a malicious flow. Bad CHOKe miss RED hit: during a CHOKe miss RED hit, the CHOKe miss victim belongs to a non-malicious flow. CHOKe miss RED miss occurs when the flow id of the CHOKe miss victim does not match the flow whose packets dominate the buffer at the time of selection of the CHOKe miss victim. This again could be either a good or bad CHOKe miss RED miss. Good CHOKe miss RED miss: during a CHOKe miss RED miss, the CHOKe miss victim belongs to a non-malicious flow. Bad CHOKe miss RED miss: during a CHOKe miss RED miss, the CHOKe miss victim belongs to a malicious flow.

Figure 5.26 shows that out of the packets that are dropped by the RED buffer, a large majority experience RED hits. Almost all RED hits are good CHOKe miss RED hits, thus RED was accurate in identifying malicious flows from CHOKe misses. It seems that at lower rates, there is a large number of CHOKe misses. However, these are very few since at these rates, the queue drops or marks few packets. We also find that all CHOKe miss RED misses, although few in number, were bad. Hence, RED also acts as an additional filter, after CHOKe, to weed out the packets belonging to malicious flows. This implies that both CHOKe and RED history could (should) be used together to identify malicious flows faster than using one or the other. Although CHOKe-RED drops/marks are more accurate than CHOKe hits and misses, they occur less often since in RED, a packet is either accepted or dropped whenever *avq* is between Th_{min} and Th_{max}. Moreover, only one packet is dropped while in CHOKe, two packets are dropped for each hit. Hence, in the next section, we aim to improve on the accuracy of CHOKe hits and misses.

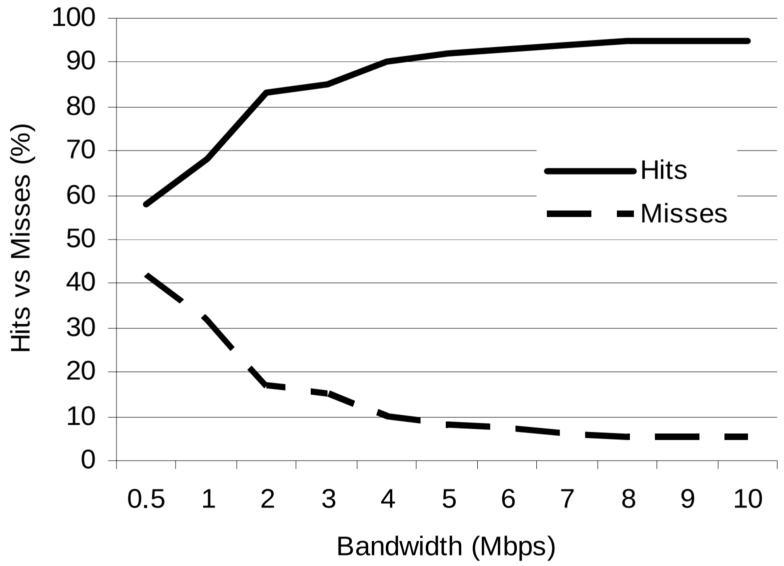

Figure 5.26 The percentage of RED hits and misses after CHOKe misses

5.3 Half1- and Half2-CHOKes

From the experiments in the previous section, we see that if UDP flow rates are high and a victim packet is chosen from Qtr3 and Qtr4 (or Half2), that will improve the chances of having

i) a good CHOKe hit if the arriving packet belongs to a malicious flow, or

ii) a good CHOKe miss if the arriving packet belongs to a TCP-friendly flow, while decreasing the chances of

iii) bad CHOKe hits if the arriving packet belongs to a malicious flow, or

iv) bad CHOKe misses if the arriving packet belonged to a TCP-friendly flow. However, for UDP rates at or below the link capacity, the victim packet should be chosen from Qtr1 and Qtr2 (or Half1). Although at low rates, Half2 has slightly greater percentages in good CHOKe hits than at Half1, the latter gives better protection to non-malicious flows (as shown in figure 5.23). Here we show by experiments that at lower rates, good CHOKe hits are more effective in Half1 than at Half2. Hence, we propose two versions of CHOKe. The first, Half2-CHOKe, chooses victim packets from Half2 whereas the second, Half1-CHOKe, chooses from Half1. From figure 5.8, UDP packets occupancy of Qtr1 during CHOKe hits is consistently greater than 30% when the UDP rate is at and less than the link capacity. This is true whenever the sum of UDP rates is at or below the link capacity.

Using the same network topology as before, we carried out simulations using NS2 to compare Half1, Half2 and CHOKe. Figures 5.27 and 5.28 illustrate two examples when the UDP rate is set to a rate greater than the link capacity using the network topology in figure 5.2 and the number of TCP flows is set at 10. Here, the CHOKe version using Half2 (Half2-CHOKe) performs better than both regular CHOKe and Half1-CHOKe. Even when the number of UDP flows is increased, as long as the sum of their bandwidth is greater than the link capacity, Half2-CHOKe outperforms the other two. Figures 5.29, 5.30 and 5.31 illustrate an example of one of the many experiments that we have performed confirming this result. Figure 5.32 shows a comparison of bandwidths of malicious flows for all three schemes.

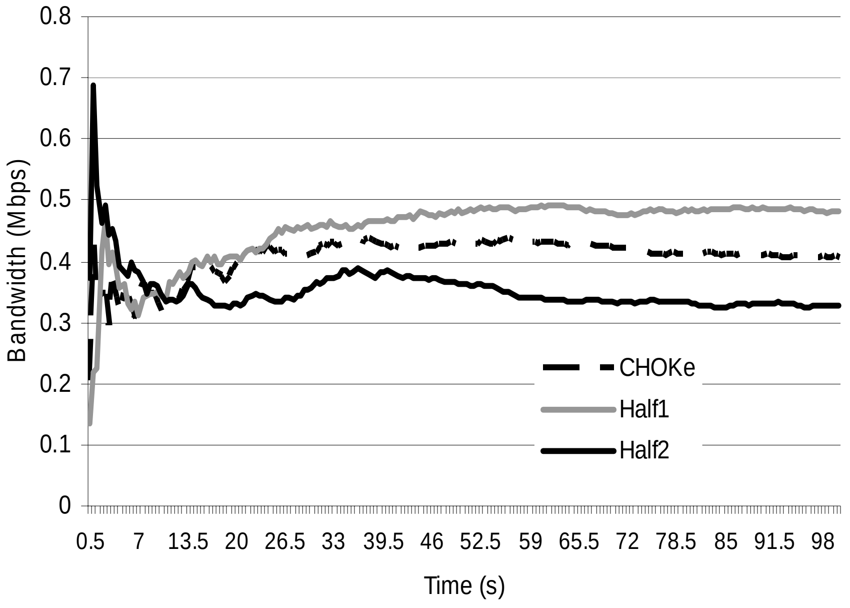

Figure 5.27 Link utilization with a UDP flow of 2Mbps

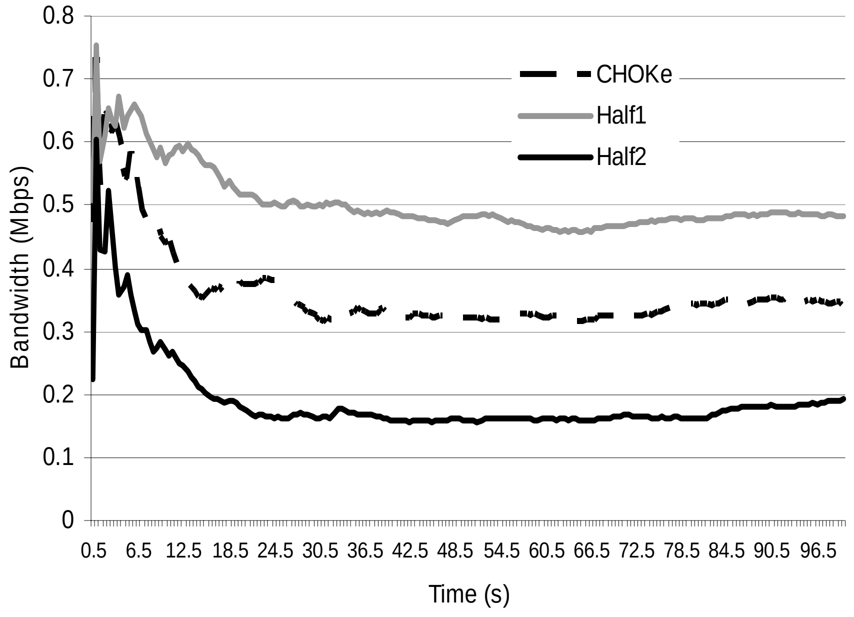

Figure 5.28 Link utilization with a UDP flow of 10Mbps

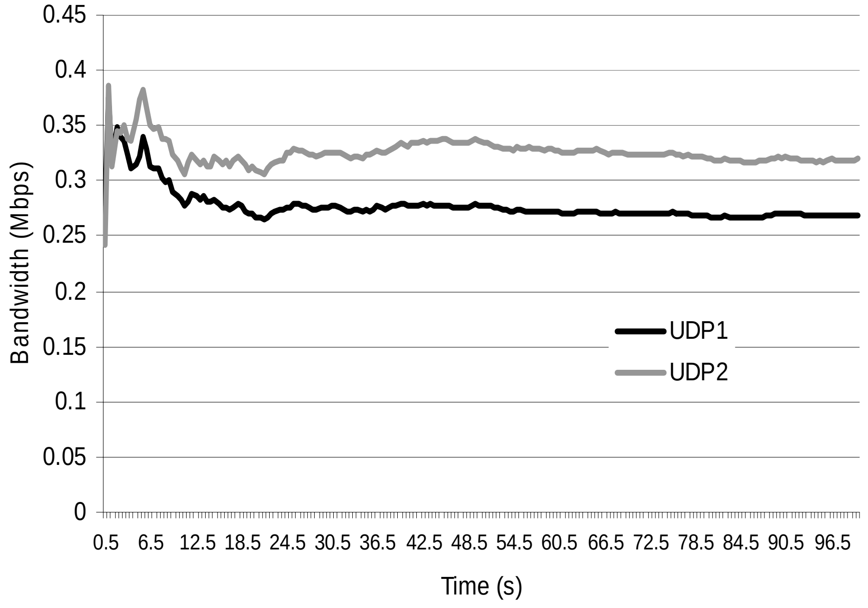

Figure 5.29 Link utilization of 2 UDP flows (1Mbps each) using CHOKe

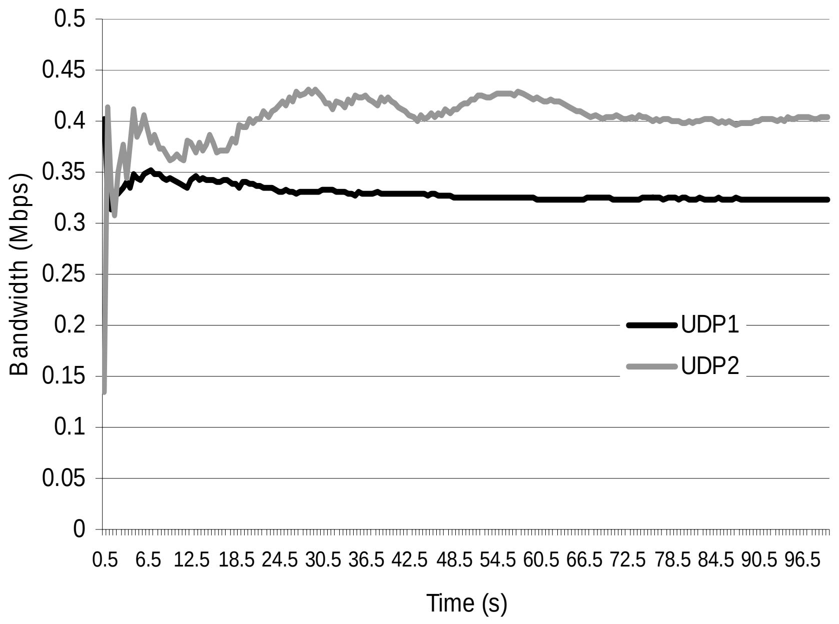

Figure 5.30 Link utilization of 2 UDP flows (1Mbps each) using Half1

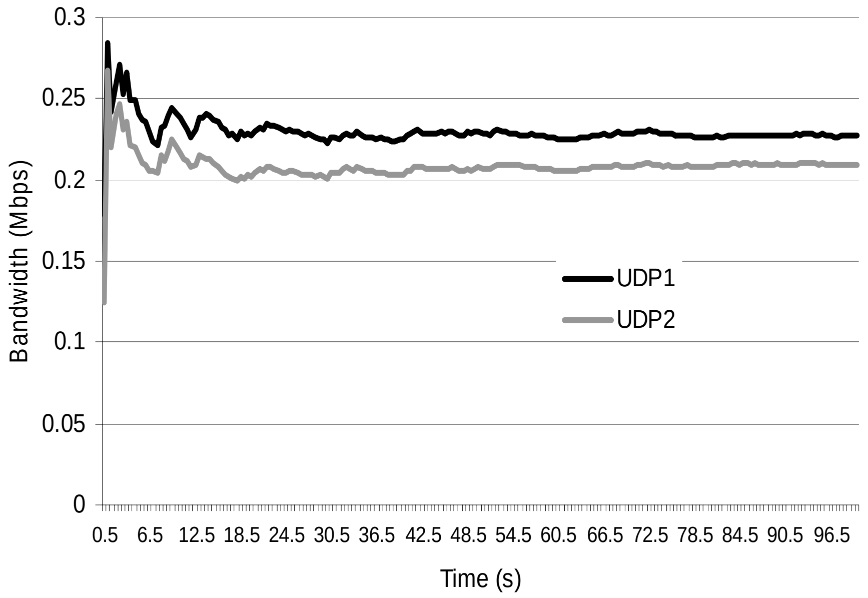

Figure 5.31 Link utilization of 2 UDP flows (1Mbps each) using Half2

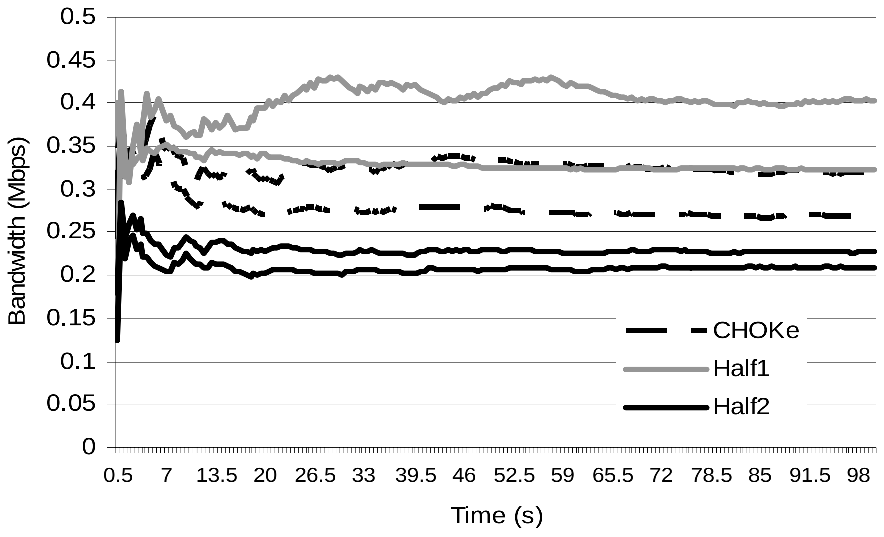

Figure 5.32 Link utilization of 2 UDP flows at 1Mbps

Figure 5.33 illustrates the situation when the UDP rate is set below the link capacity, i.e. at 0.5Mbps, using the same network topology as before. Here, Half1-CHOKe performs better than both regular and Half2-CHOKe. When the number of UDP flows is increased, as long as the sum of their rate is below the link capacity Half1-CHOKe outperforms the other two variants. Figures 5.34, 5.35, and 5.36 show some sample experiment outcomes from the many experiments we performed confirming this result. Figure 5.37 shows a comparison of bandwidths of the malicious flows for the 3 schemes for 3 UDP flows at 0.3Mbps. Here, Half1-CHOKe decreases the UDP occupancy of Qtr1 and Qtr2 faster than regular CHOKe. The effects of CHOKe hits are far greater for Half1-CHOKe than for Half2-CHOKe as the former focuses on the head of the queue, providing an approximate upper bound for the bandwidth of the outgoing UDP flow.

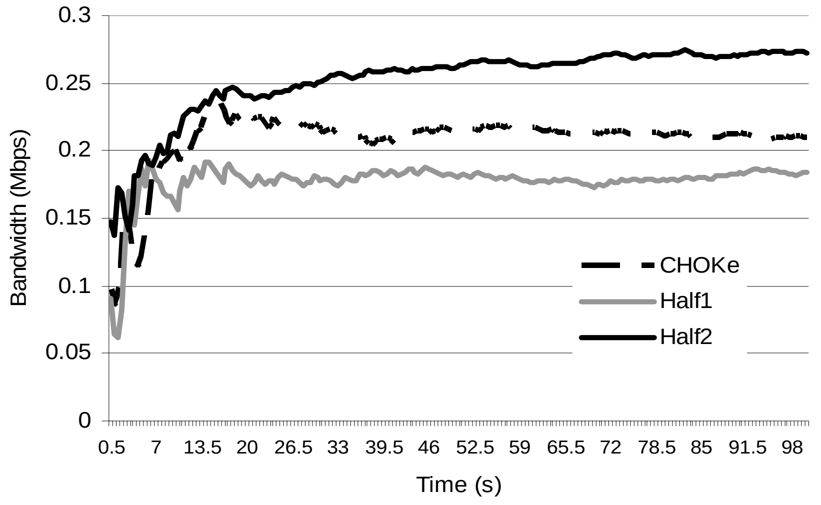

Figure 5.33 Link utilization of the UDP flow at 0.5Mbps

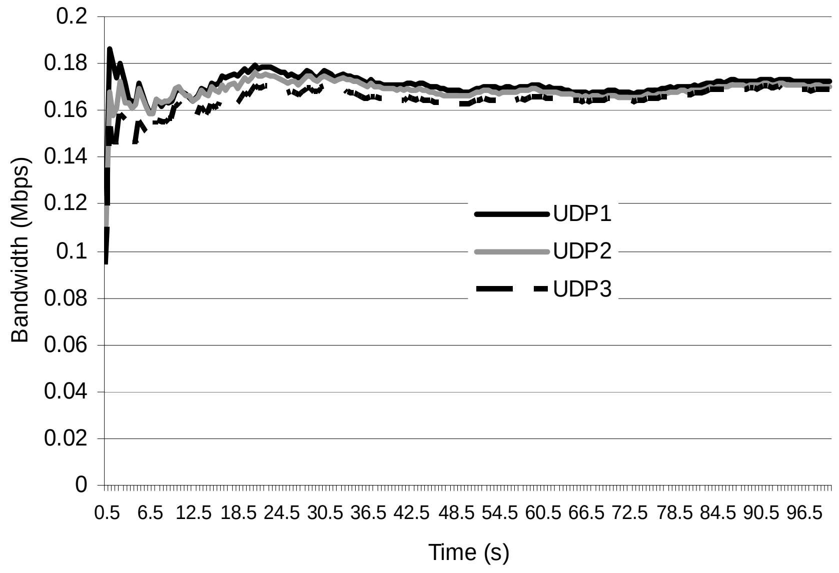

Figure 5.34 Link utilization of 3 UDP flows (0.3Mbps each) using CHOKe

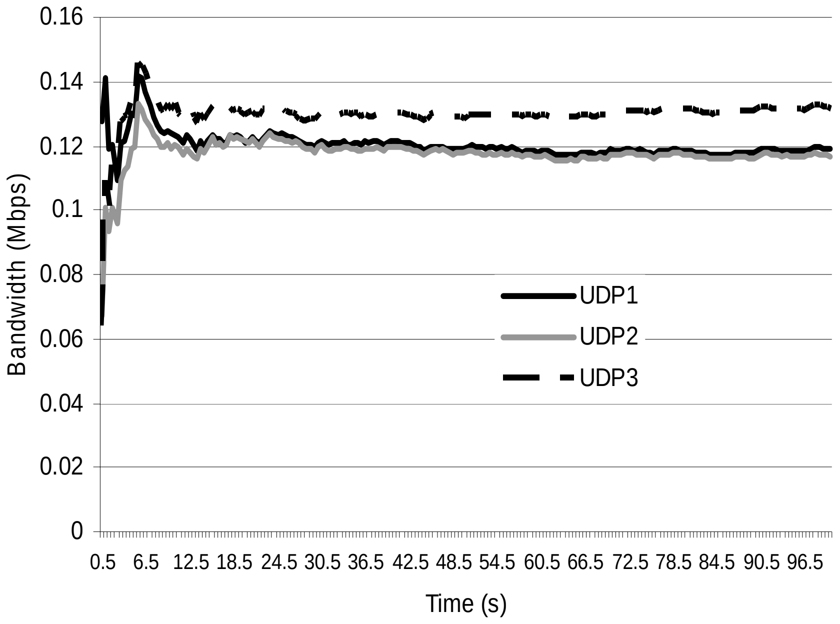

Figure 5.35 Link utilization of 3 UDP flows (0.3Mbps each) using Half1

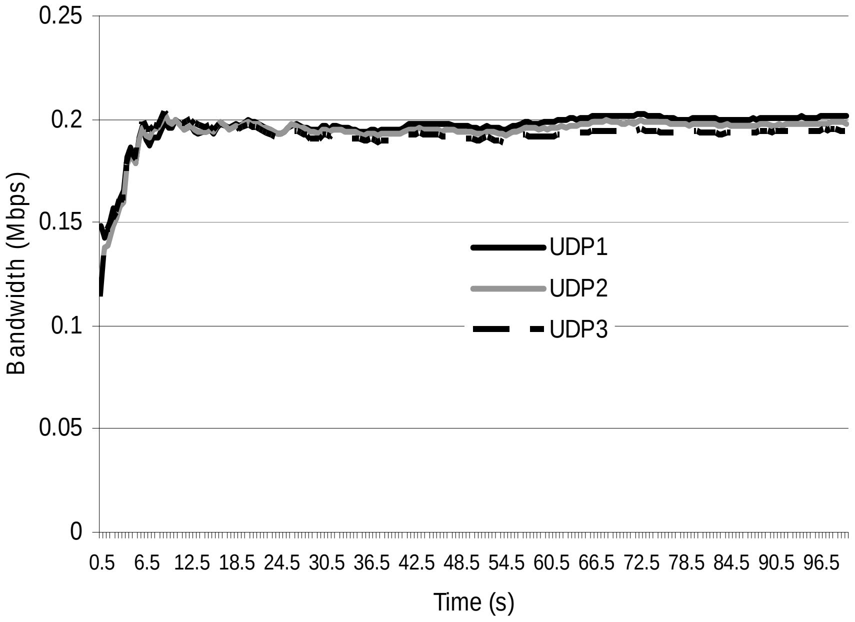

Figure 5.36 Link utilization of 3 UDP flows (0.3Mbps each) using Half2

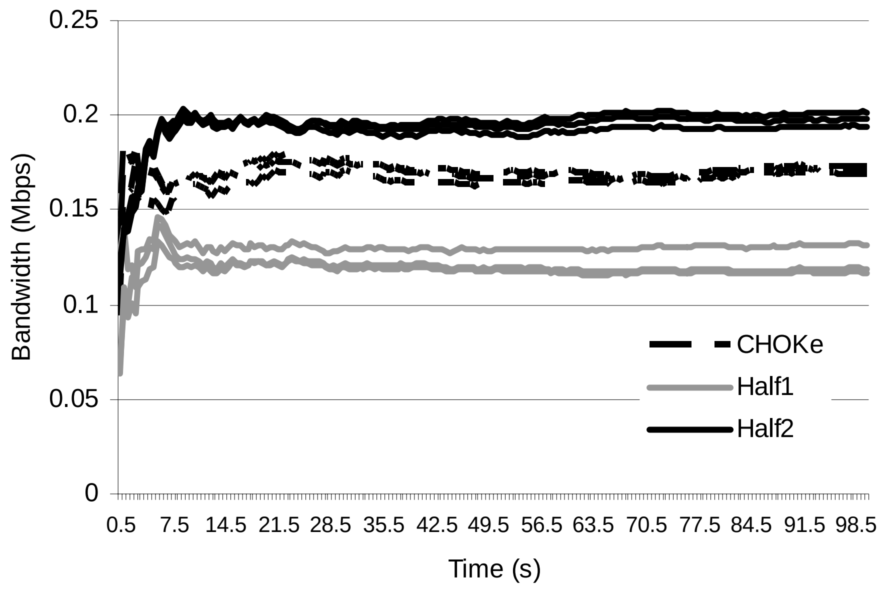

Figure 5.37 Link utilization of 3 UDP flows at 0.3Mbps

Hence, for RECHOKe, we will start with Half2-CHOKe and at certain intervals (e.g., every quarter second) we monitor the first quarter of the buffer to check the occupancy of the buffer by the malicious flows recorded in the table. If the occupancy rate is greater than 30%, we can proceed with Half1-CHOKe otherwise we switch to Half2-CHOKe.

5.4 Analysis of xCHOKe and RECHOKe

In this section we analyze xCHOKe and RECHOKe buffers; to do this, we define the following terms. A table hit occurs when the flow id of the arriving packet matches a flow id entry already in the table. A table hit can be either good or bad. The table hit is good if the flow that the hit packet belongs to is malicious; it is a bad hit otherwise. Table drops are table hits that were selected for dropping by xCHOKe (after drawing a random with probability $p*$); for RECHOKe, these are table hits that were selected after evaluating $p*$ and $c*$. Good table drops occur when the dropped packet belongs to a malicious flow; bad table drops occur otherwise. A CHOKe miss victim is a packet which after experiencing a CHOKe miss enters the RED queue. CHOKe miss RED hit occurs when the flow id of the CHOKe miss victim matches the flow whose packets dominate the buffer at the time of selection of the CHOKe miss victim; CHOKe miss RED miss occur otherwise. A CHOKe miss RED hit could be either a good or bad: it is good if the victim packet belongs to a malicious flow; it is bad otherwise. A CHOKe miss RED miss again could be either a good or a bad CHOKe miss RED miss. Good CHOKe miss RED miss is when the victim packet belongs to a non-malicious flow; it is bad otherwise.

Using NS2, we simulate a bar-bell topology and analyze the xCHOKe and RECHOKe buffers in terms of buffer occupancy by a malicious flow (a constant-bit-rate UDP flow) competing with 10 TCP flows over a 3-Mbps link. All TCP flows have the same round trip propagation delay of 20ms with each output link having a latency of 1ms and capacity of 10Mbps. The parameters for the xCHOKe and RECHOKe buffers are: $Th_{min} = 10$, $Th_{max} = 50$, $P_{max} = 0.1$.

We ran simulations in 11 steps, starting with a malicious UDP rate of 0.5Mbps, increasing the rate in 1Mbps increments. Each simulation was long enough for initial transients to settle and be insignificant. Where appropriate, enough simulations were run to claim a 95% confidence that the confidence intervals are less than 5% of the mean.

Figure 5.38 shows the number of table hits for both the xCHOKe and RECHOKe algorithms. The number of hits in RECHOKe was nearly 50-65% greater than when xCHOKe was used. The combination of CHOKe and RED histories allow RECHOKe to register a large number of hits and using these hits, RECHOKe can isolate these malicious flows faster and more accurately. RECHOKe does punish malicious flows more rigorously while not penalizing TCP-friendly flows as it can be observed by comparing figures 5.55 and 5.56. The majority of the hits shown in figure 5.38 are good table hits as witnessed by figures 5.39 and 5.40. This points to RECHOKe identifying malicious flows better than xCHOKe. Figure 5.39 also shows that the number of good table hits for RECHOKe is 50-65% greater than that of xCHOKe; Figure 5.40 shows that the number of bad table hits for RECHOKe is 2-16 times less as well.

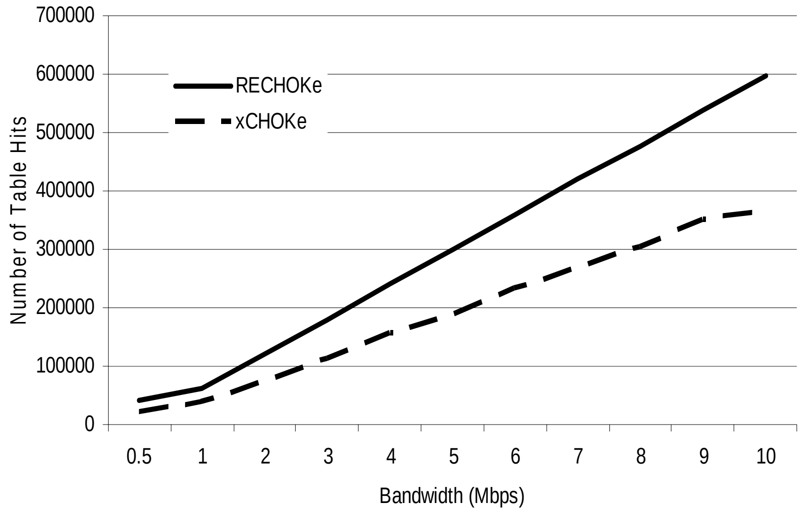

Figure 5.38 The number of table hits vs. UDP rate (RECHOKe and xCHOKe)

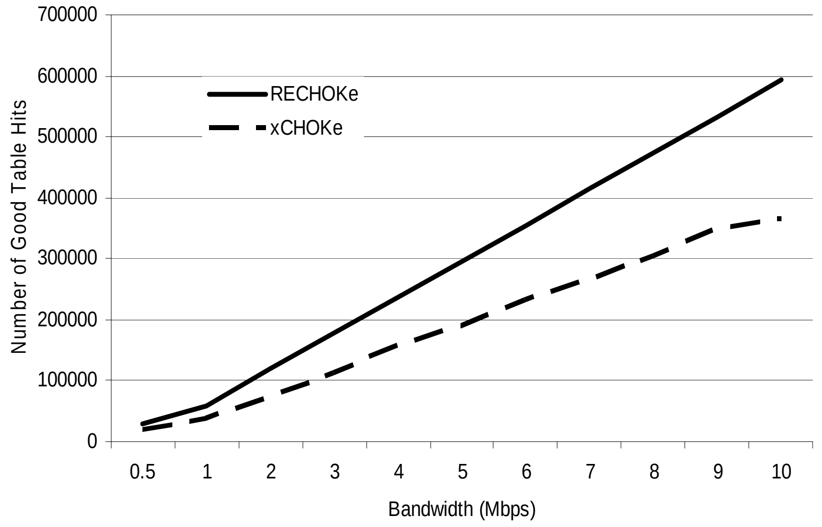

Figure 5.39 The number of good table hits vs. UDP rate (RECHOKe and xCHOKe)

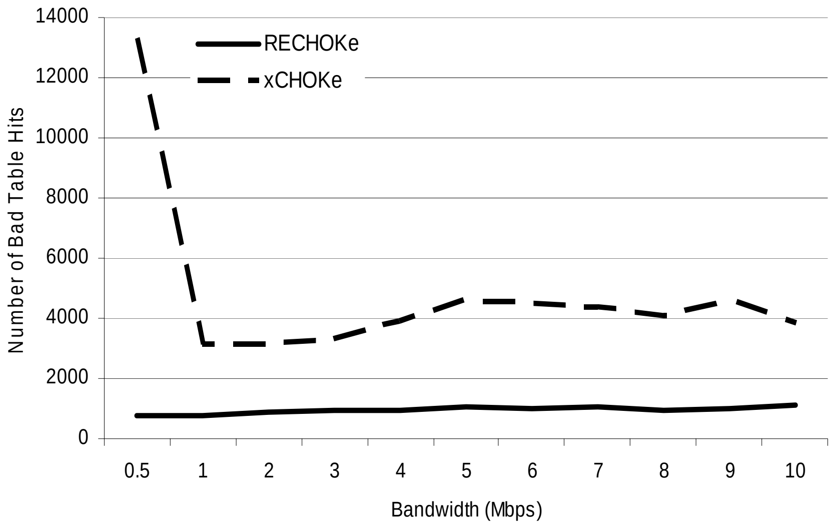

Figure 5.40 The number of bad table hits vs. UDP rate (RECHOKe and xCHOKe)

Next, we analyze the number of table drops for both algorithms. Ideally drops should only consist of good table drops. Figure 5.41 shows the number of table drops for both algorithms. Figure 5.42 shows the number of good table drops; we can observe that RECHOKe outperforms xCHOKe significantly; while figure 5.43 shows that the same is true in terms of bad table drops. Nearly all the table misses (not shown here) for both algorithms are good table misses. We can conclude that RECHOKe drops fewer packets from TCP-friendly flows than xCHOKe, thereby penalizing TCP-friendly flows less significantly than xCHOKe.

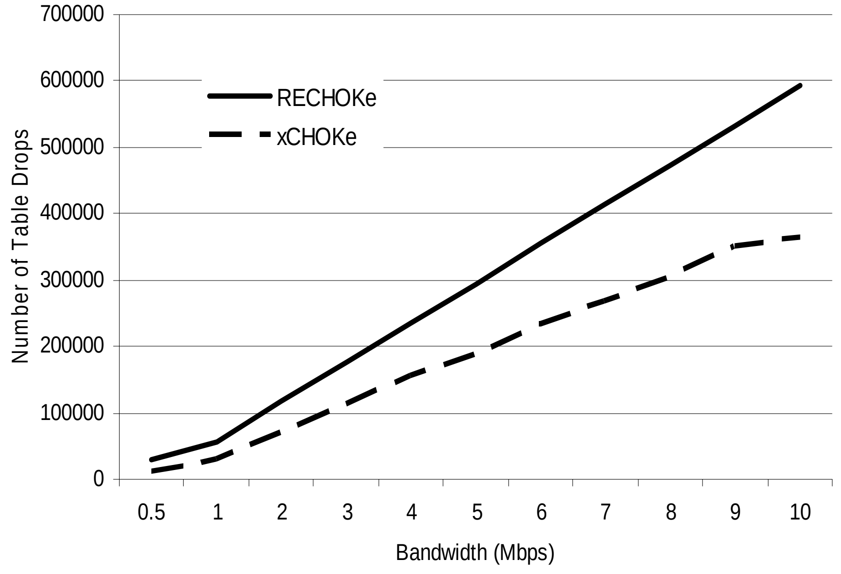

Figure 5.41 The number of table drops vs. UDP rate (RECHOKe and xCHOKe)

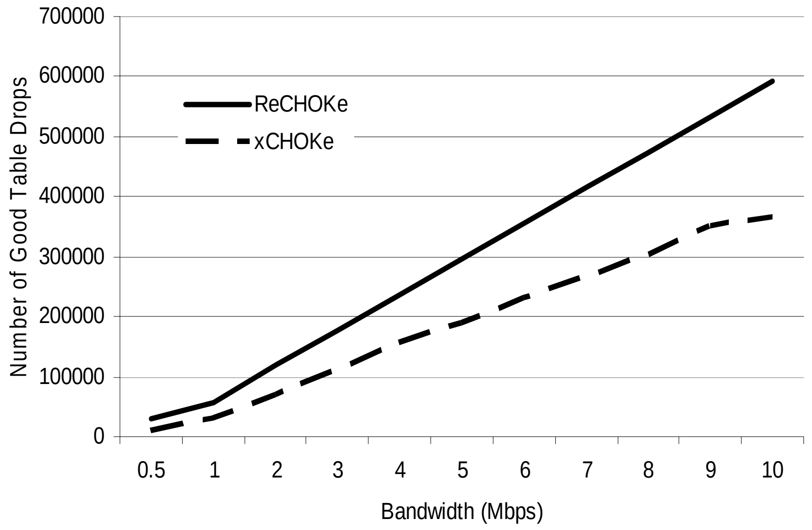

Figure 5.42 The number of good table drops vs. UDP rate (RECHOKe and xCHOKe)

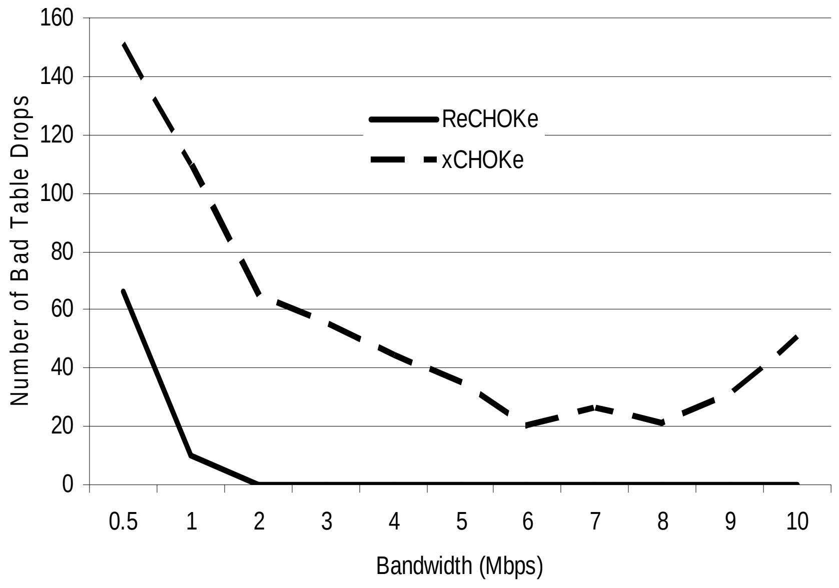

Figure 5.43 The number of bad table drops vs. UDP rate (RECHOKe and xCHOKe)

Due to the large number of good table drops and the almost insignificant number of bad table drops, RECHOKe is faster and more accurate in the identification of flows. Unlike xCHOKe, RECHOKe updates the table after table hits. This is necessary because advancing farther in the simulations CHOKe hits become more unreliable as the number of packets belonging to the malicious flow in the buffer are reduced drastically leading to a greater number of bad CHOKe hits (as shown in figures 5.46 and 5.47). The same phenomenon can be observed in xCHOKe and CHOKe, which leads to unnecessary punishment of TCP-friendly flows. Hence, in RECHOKe, we do not drop packets during CHOKe hits. Figures 5.44 and 5.45 show that the number of CHOKe hits and misses for both algorithms, reflect this trend. CHOKe hits did not affect RECHOKe as the packets responsible for these hits were not dropped but passed on to the RED buffer. Although these hits were updated in the table, the c* condition prevents unnecessary drops of TCP-friendly packets.

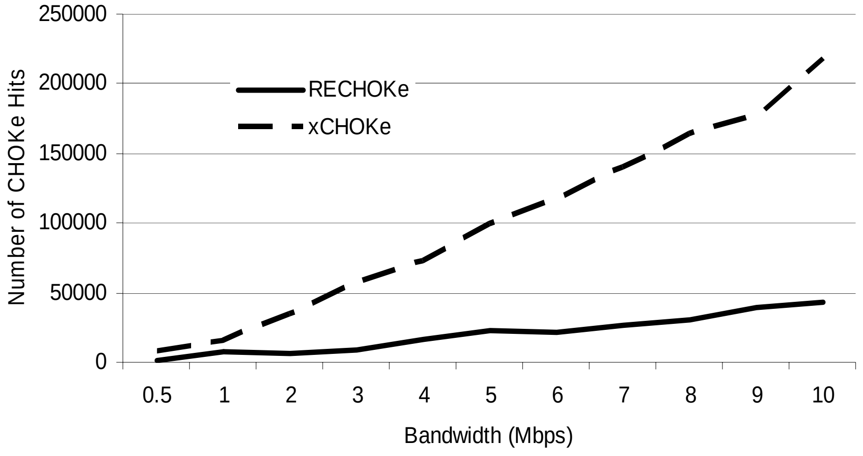

Figure 5.44 The number of CHOKe hits vs. UDP rate (RECHOKe and xCHOKe)

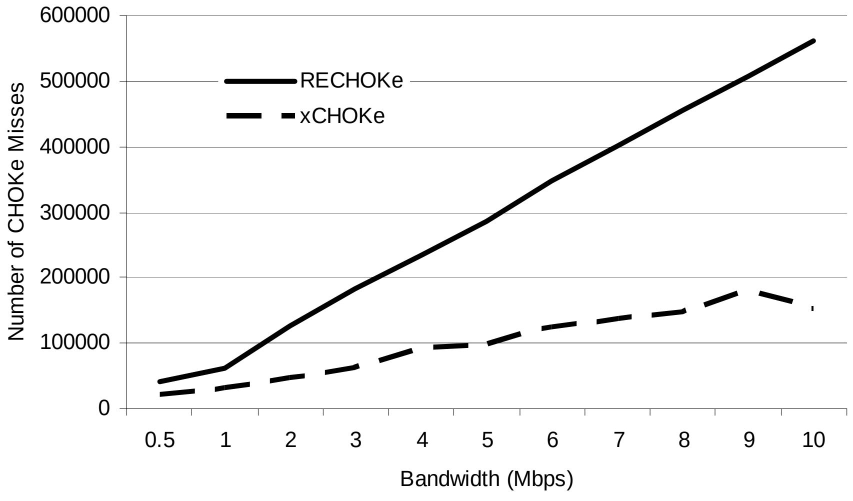

Figure 5.45 The number of CHOKe misses vs. UDP rate (RECHOKe and xCHOKe)

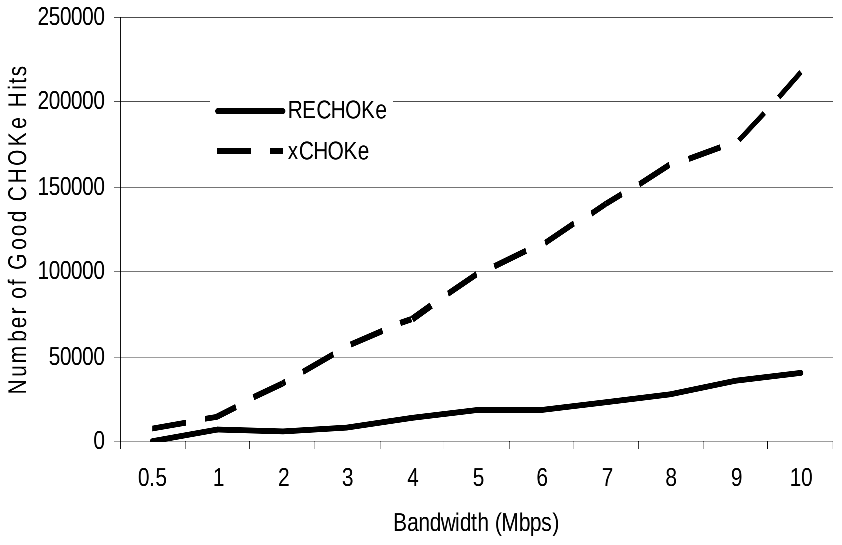

Figure 5.46 The number of good CHOKe hits vs. UDP rate (RECHOKe and xCHOKe)

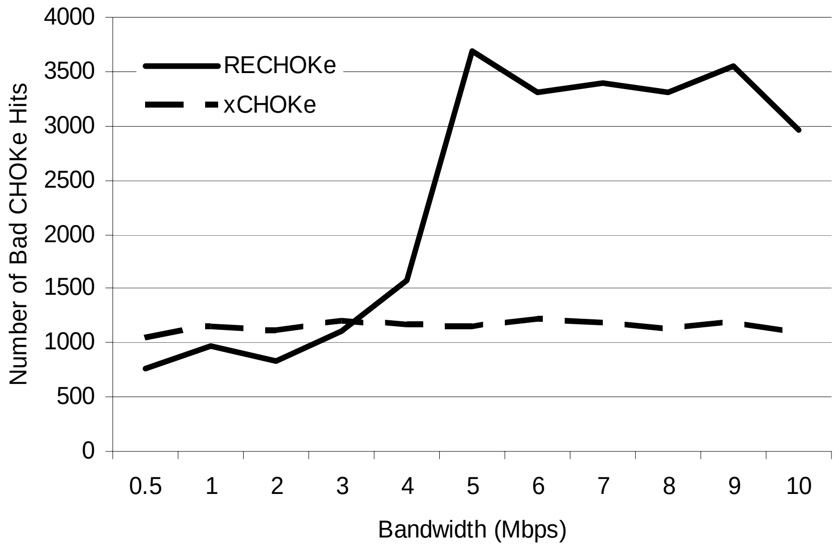

Figure 5.47 The number of bad CHOKe hits vs. UDP rate (RECHOKe and xCHOKe)

Upon analyzing the CHOKe misses in figure 5.48 and 5.49, we have found that RECHOKe significantly outperforms xCHOKe in both the number of bad CHOKe misses and good CHOKe misses despite having the larger number of CHOKe Misses. This shows that RECHOKe allows packets from TCP-friendly flows to enter the RED buffer more frequently than xCHOKe.

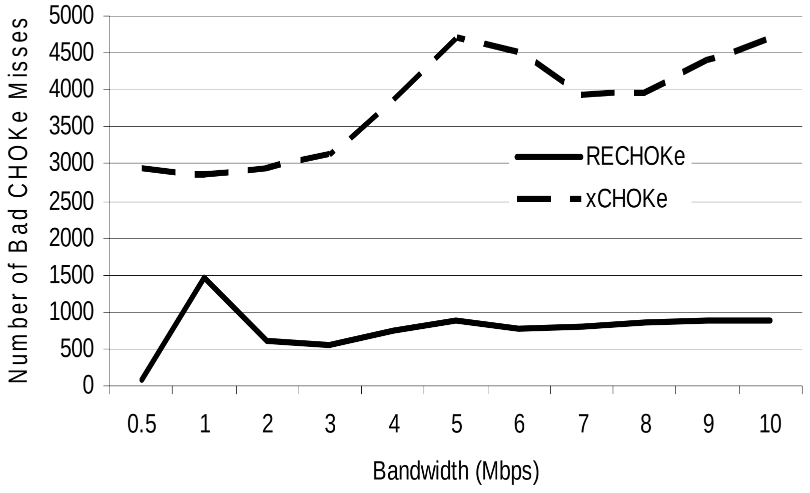

Figure 5.48 The number of bad CHOKe misses vs. UDP rate (RECHOKe and xCHOKe)

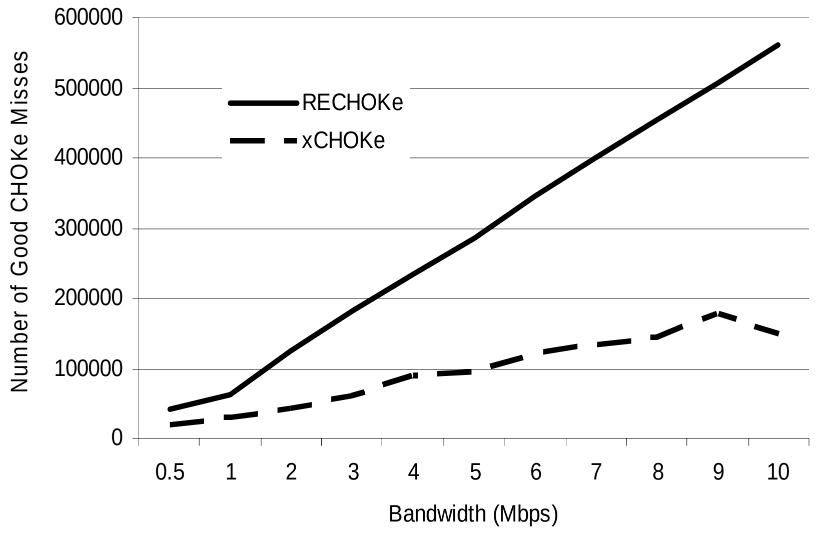

Figure 5.49 The number of good CHOKe misses vs. UDP rate (RECHOKe and xCHOKe)

Next, we analyze RED hits and misses for both RECHOKe and xCHOKe. Figure 5.50 shows that the number of dropped packets was greater when using RECHOKe. This is true because during CHOKe hits packets are not dropped but admitted into the buffer; as a result, more packets need to be dropped by the RED buffer. Analyzing both figures 5.51 and 5.52 we can see that the most RED hits with RECHOKe are good, i.e., most dropped packets belong to malicious flows. This happens in spite of most of the admitted packets being from the ten TCP-friendly flows. After the good RED hits, the malicious flows became no longer dominant. Hence, RED took out more UDP packets from (the already the few) that had been admitted into the buffer.

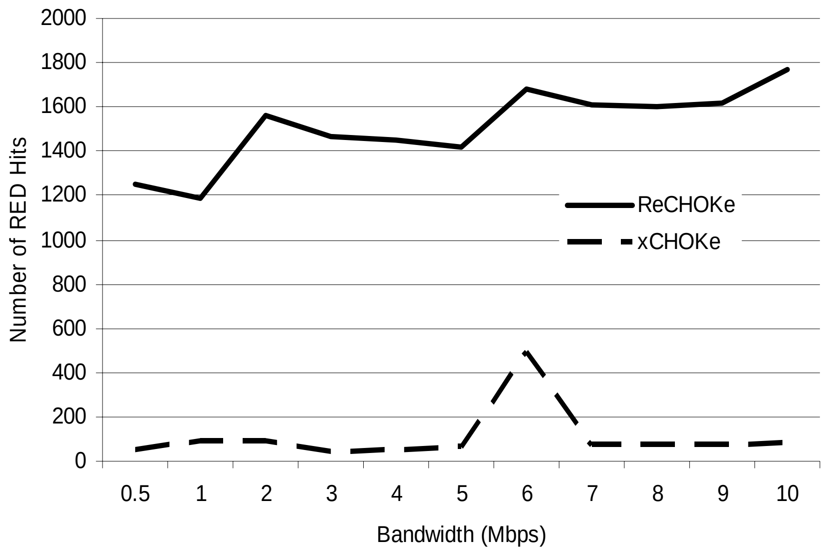

Figure 5.50 The number of RED hits vs. UDP rate (RECHOKe and xCHOKe)

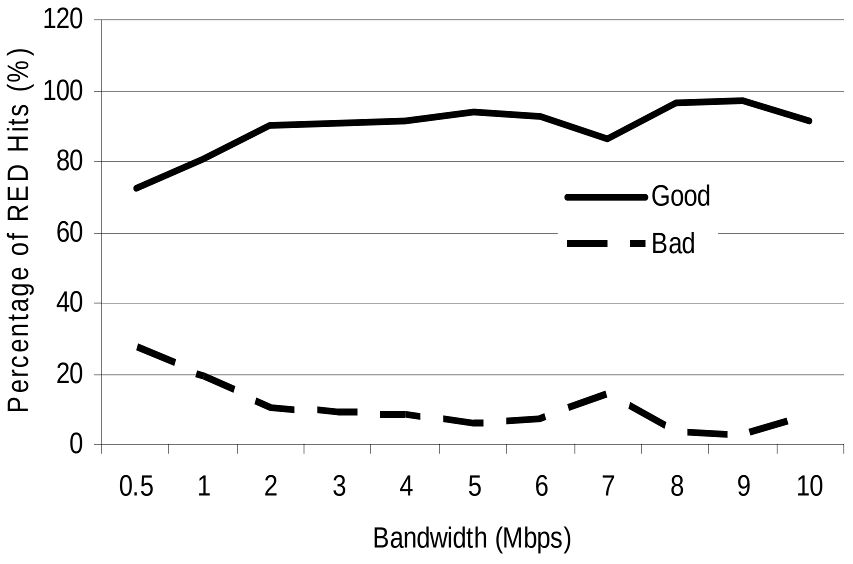

Figure 5.51 The percentages of good and bad RED hits vs. UDP rate (RECHOKe)

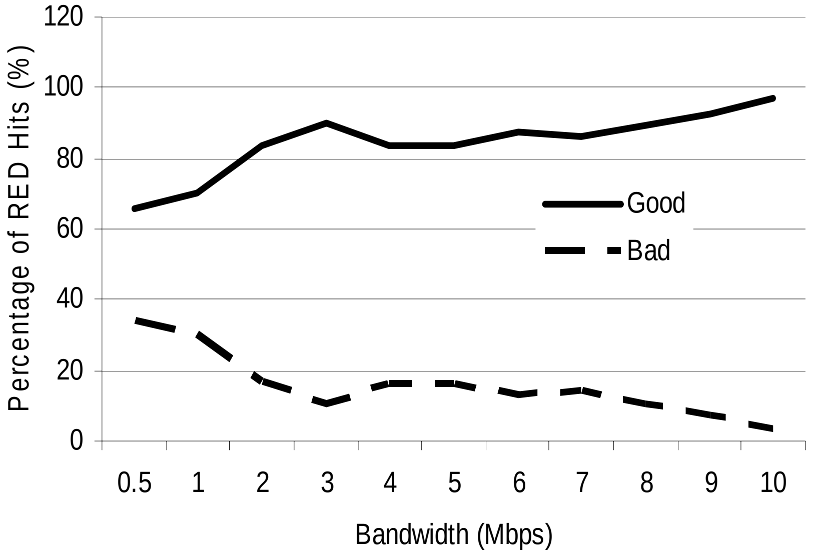

Figure 5.52 The percentages of good and bad RED hits vs. UDP rate (xCHOKe)

5.5 Comparing RECHOKe with RED, CHOKe and xCHOKe

In this section, we present experiments to compare RECHOKe with RED, CHOKe and xCHOKe. In these sets of experiments we again use the same barbell topology as before (as shown in figure 5.2), however with two malicious flows. Each of these flows is a constant-bit-rate UDP flow set at a) 1.2Mbps and b) 2Mbps, competing with 10 TCP flows over a 3-Mbps R1-R2 link. (UDP rates of 1.2Mbps result in flows with a capacity less than the link capacity, while using 2Mbps flows put us beyond this capacity.) All TCP flows have the same round trip propagation delay of 20ms with each output link having a latency of 1ms and a capacity of 10Mbps. The TTL is 10ms for both xCHOKe and RECHOKe. The RED parameters for RED, CHOKe, xCHOKe and RECHOKe buffers are: $Th_{min} = 10$, $Th_{max} = 50$, $P_{max} = 0.1$.

We present two simulation scenarios; in Scenario 1, we study the impact of two malicious flows with non-identical rates on a set of well-behaved TCP flows with identical RTTs while in Scenario 2, we deal with non- identical RTTs.

5.5.1 Scenario 1

Figures 5.53, 5.54, 5.55 and 5.56 show the bandwidth allocation on the congested link for each flow using RED, CHOKe, xCHOKe and RECHOKe buffer respectively when the UDP rate is low. Figure 5.57 shows a comparison of bandwidths of the malicious flows for the 4 schemes. RED, as expected, does not control the malicious flows (UDP1 and UDP2) from gaining control of almost the entire link bandwidth. The bandwidths of the TCP-friendly flows (TCP1 to TCP10) are almost zero. CHOKe and xCHOKe do control malicious flows, however, during the process of controlling these malicious flows, TCP flows suffer CHOKe hits and as a result suffer bandwidth losses. On the other hand, by reducing bad CHOKe hits, RECHOKe controls malicious flows better than CHOKe and xCHOKe. It also offers better protection for TCP-friendly flows.

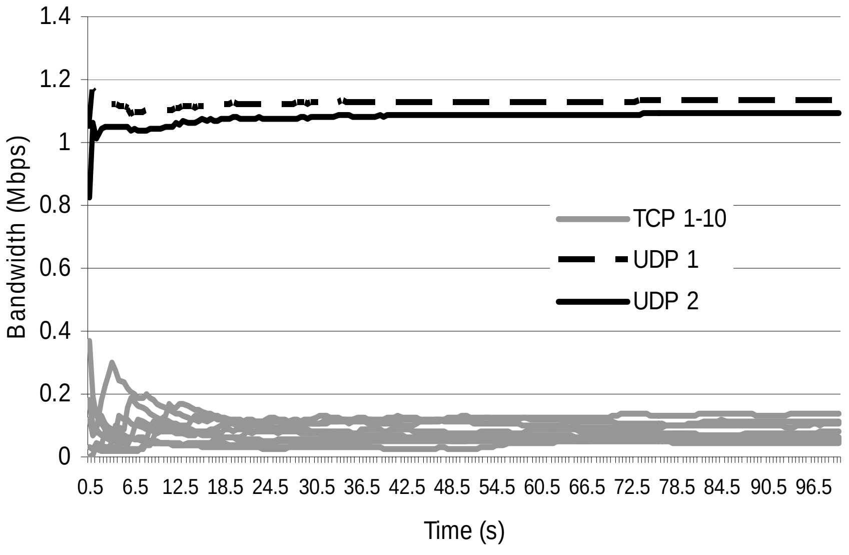

Figure 5.53 Link utilization with RED

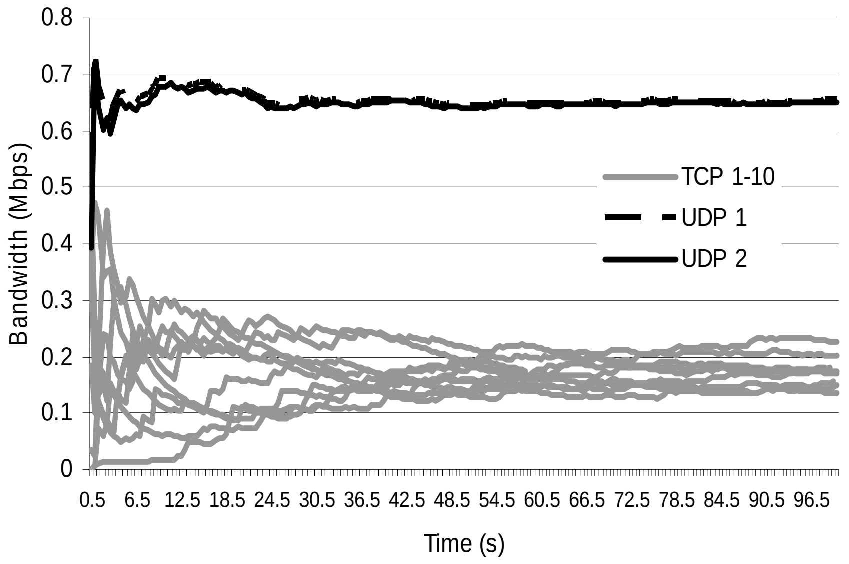

Figure 5.54 Link utilization with CHOKe

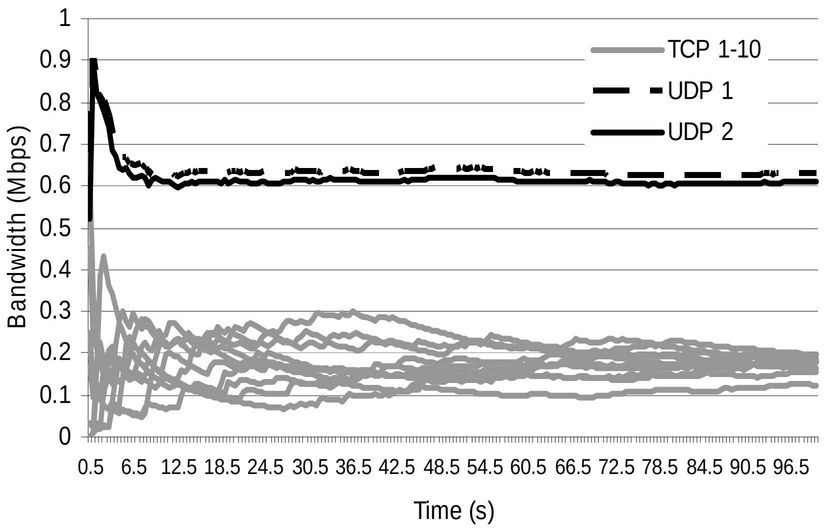

Figure 5.55 Link utilization with xCHOKe

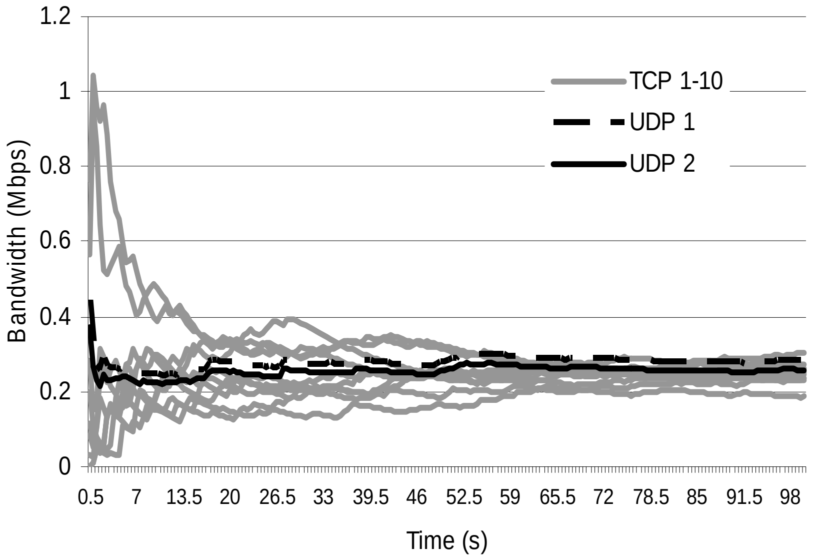

Figure 5.56 Link utilization with RECHOKe

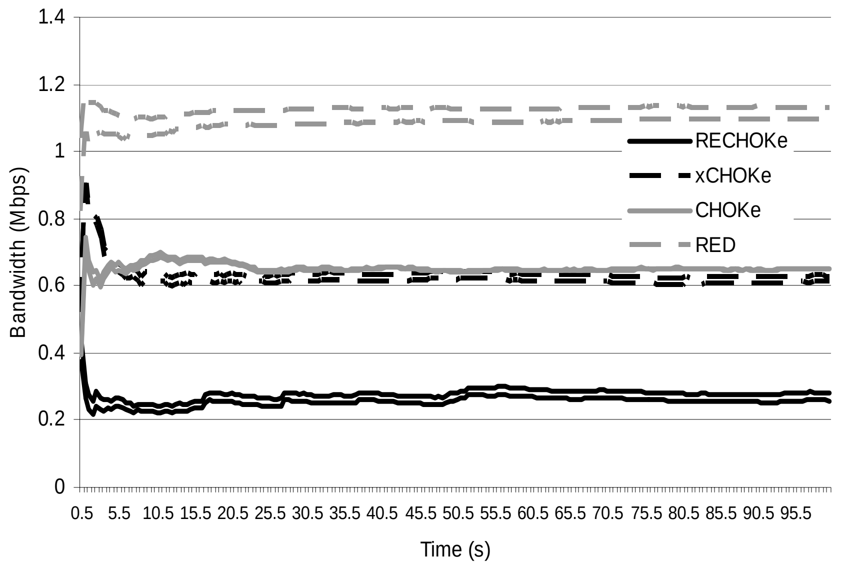

Figure 5.57 UDP flow rates with RECHOKe, xCHOKe, CHOKe and RED

Figures 5.58 to 5.62 show results for a similar setup but with the high UDP flow rates. The results are also similar to the previous simulation results; RED, CHOKe and xCHOKe cannot control the two malicious flows as well as RECHOKe (which thus offers better protection to TCP-friendly flows).

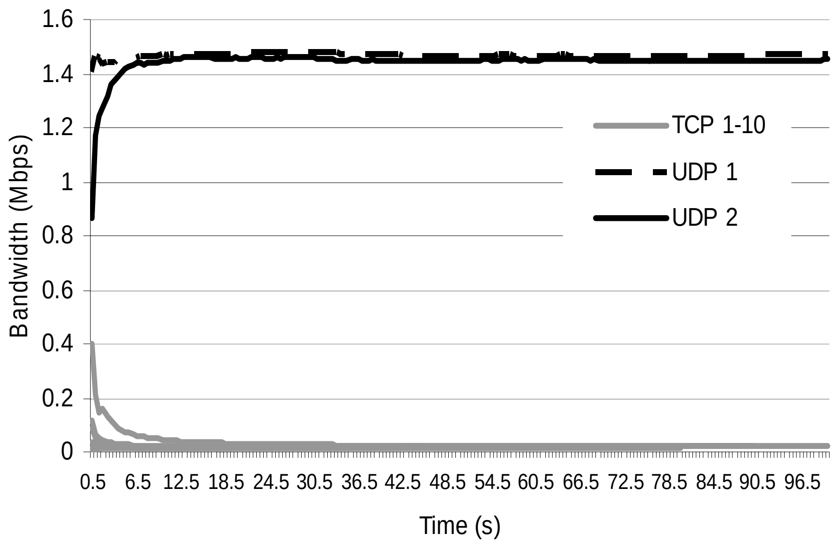

Figure 5.58 Link utilization with RED

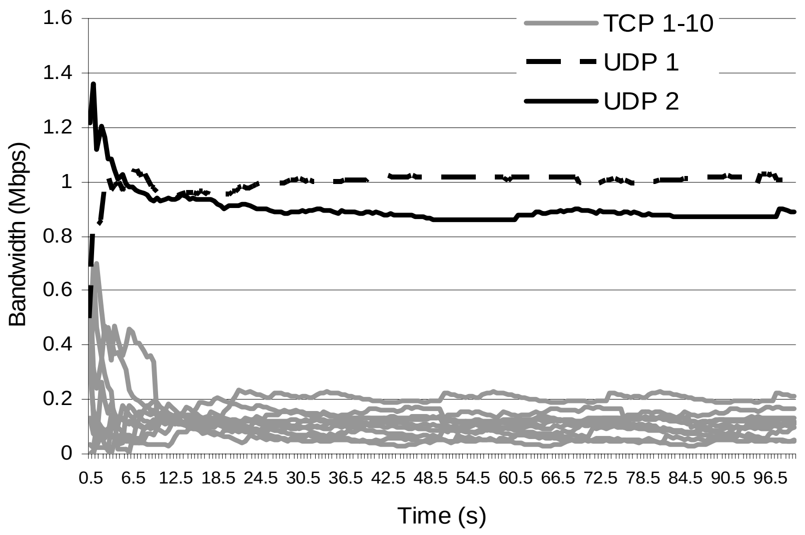

Figure 5.59 Link utilization with CHOKe

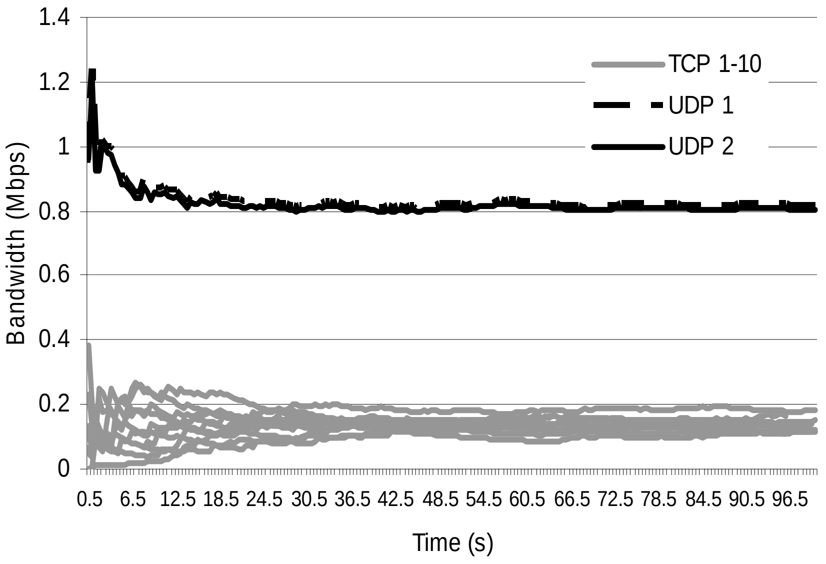

Figure 5.60 Link utilization with xCHOKe

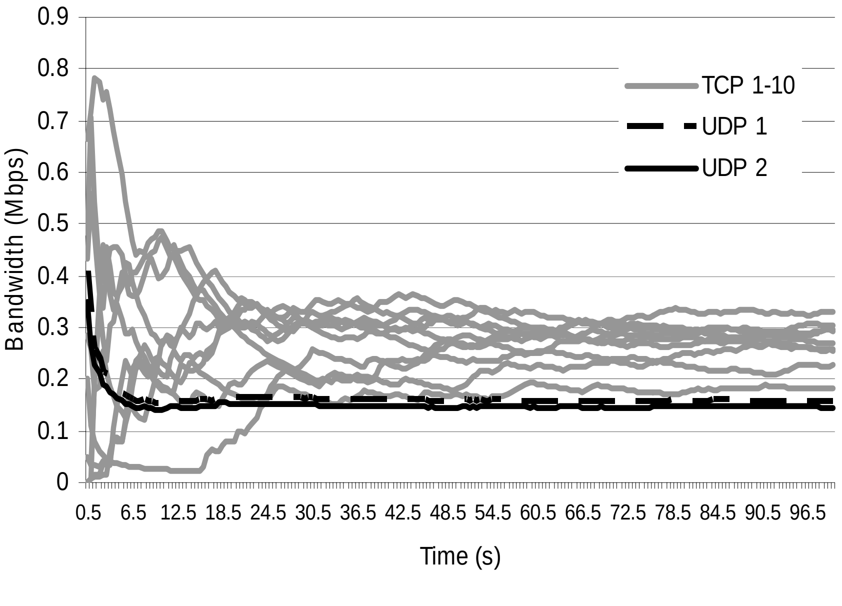

Figure 5.61 Link utilization with RECHOKe

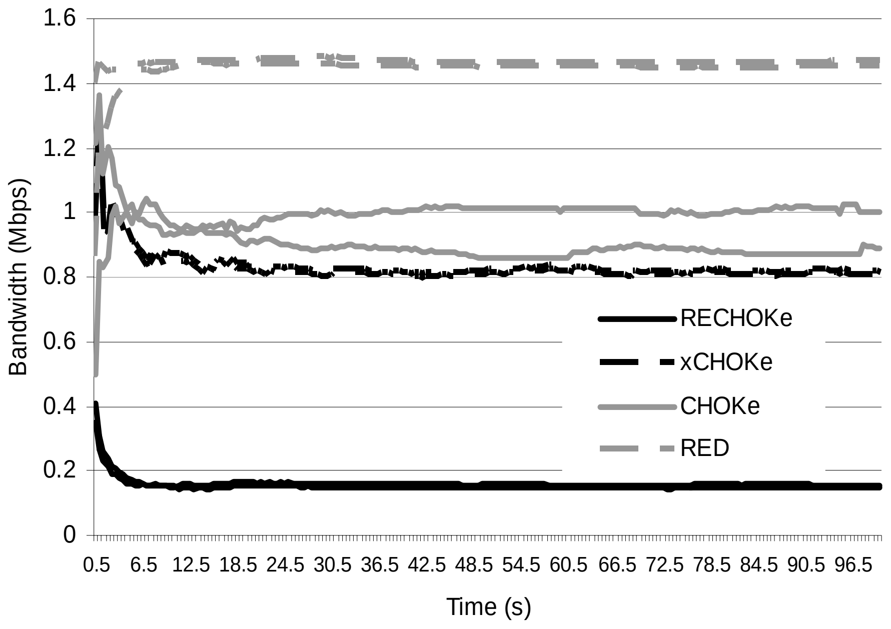

Figure 5.62 UDP flow rates with RECHOKe, xCHOKe, CHOKe and RED

Next, we study the behavior of UDP flows when the number of TCP flows is increased. In these sets of experiments we again use the same barbell topology as before (as shown in figure 5.2), however with the two malicious flows at 2Mbps each, competing with N number of TCP flows over a 3-Mbps R1-R2 link.

The results are shown in Table 5.1. Although the increase in the number of TCP flows implied an increase in the unreliability of CHOKe hits in affecting these TCP flows, it does not affect RECHOKe for the following reasons:

1) Packets are not dropped as a result of CHOKe hits and

2) The boolean function c*

In fact, the increase in congestion leads to an increase in the number of RED hits. RED hits are more reliable in detecting malicious flows (Section 5.2.1). Packets dropped by RED are mostly UDP packets and the increase in updates in the Table due to these RED hits increases the chances of dropping UDP packets as a result of Table hits.

Table 5.1 Affects of Increasing Number of TCP Flows

N number of TCP flows	Ave Link utilization by UDP	Ave Link utilization by TCP
10	0.15	0.28
20	0.06	0.14
30	0.04	0.097
40	0.025	0.073

5.5.2 Scenario 2

In this set of experiments, the network setup is similar to that of the previous section; however, we make the two UDP flows rates asymmetric by setting them at 1 and 3Mbps respectively. In addition, the ten TCP flows are simulated with non-identical round trip delays at 2ms delay increments across the flows. The R1-R2 link is set to a bandwidth of 3Mbps; RED parameters are the same as before.

Figures 5.63 to 5.66 show the bandwidth allocation for the congested link for each flow for RED, CHOKe, xCHOKe, and RECHOKe respectively. Figure 5.67 shows a comparison of bandwidths of the malicious flows for all 4 schemes. Again, as expected, RED does not control malicious flows from gaining control of almost the entire link bandwidth. CHOKe and xCHOKe do control these malicious flows however they also punish the TCP flows with bad CHOKe hits. By minimizing the bad CHOKe hits, RECHOKe controls malicious flows best (thus protecting TCP-friendly flows best).

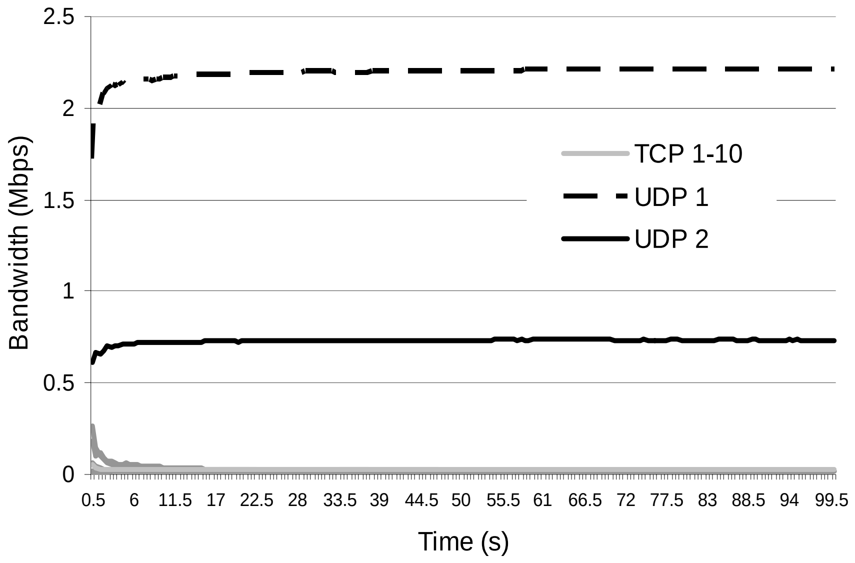

Figure 5.63 Link utilization with RED

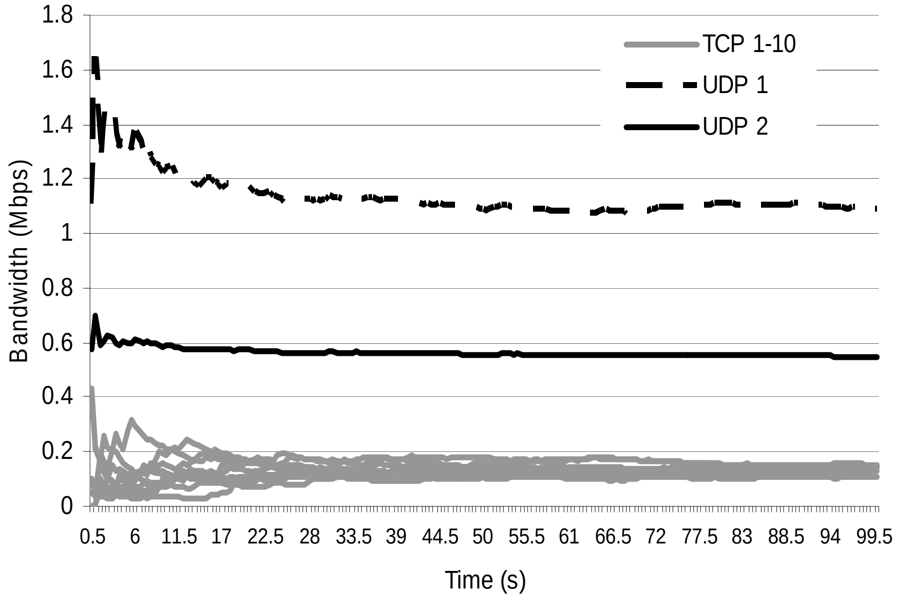

Figure 5.64 Link utilization with CHOKe

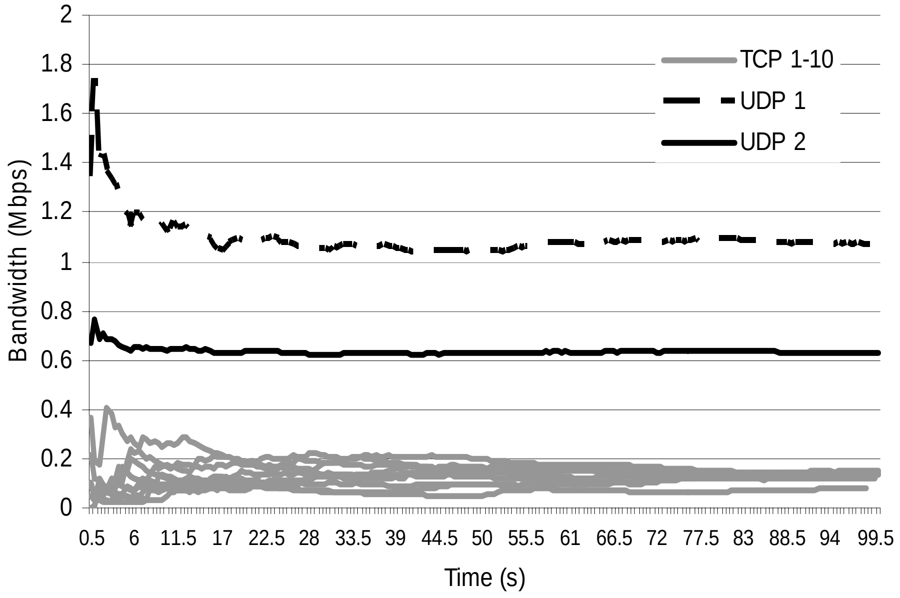

Figure 5.65 Link utilization with xCHOKe

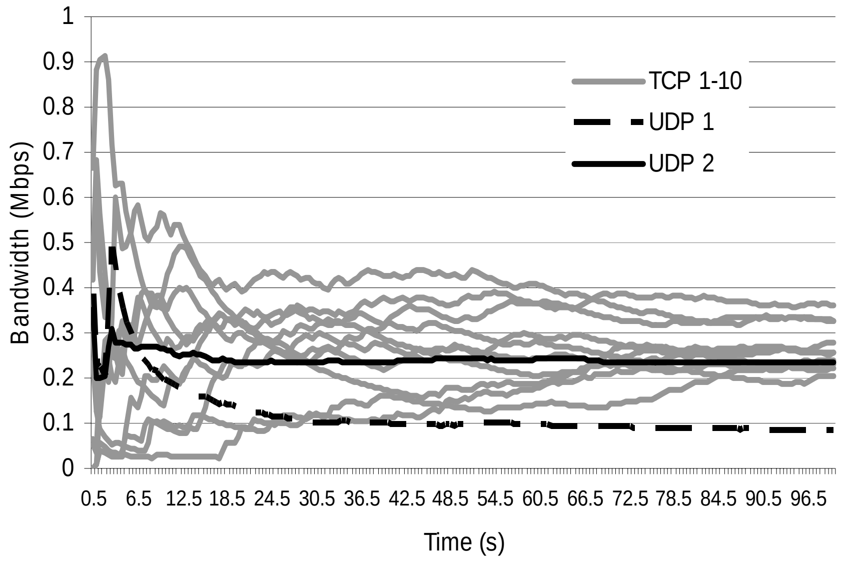

Figure 5.66 Link utilization with RECHOKe

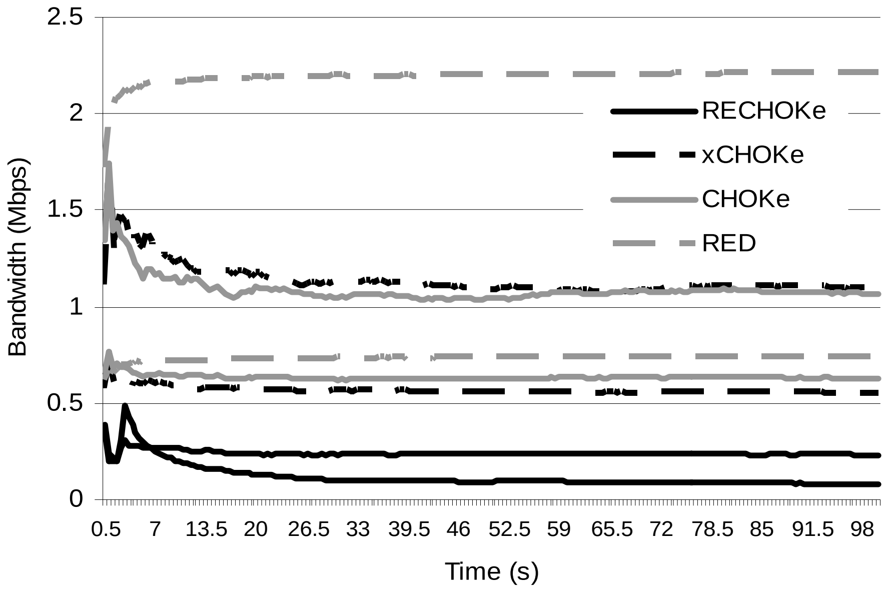

Figure 5.67 UDP flow rates with RECHOKe, xCHOKe, CHOKe, and RED

5.6 Summary

In this chapter, we analyzed the accuracy of CHOKe hits, CHOKe misses and CHOKe-RED drops/marks. We found that

1) CHOKe hits, in the form of bad CHOKe hits, were unreliable in that they affected non-malicious flows adversely by dropping their packets,
2) CHOKe misses, in the form of bad CHOKe misses, were unreliable in that they allowed UDP packets to steal more bandwidth from TCP flows.

To further investigate the cause of these problems, we presented and analyzed two variations of CHOKe called Half1 and Half2 to improve upon CHOKe.

In this chapter, we have also presented RECHOKe based on Half1- and Half2-CHOKe, a scheme for detecting, controlling and punishing malicious flows in IP networks. We showed that RECHOKe outperforms RED, CHOKe and xCHOKe by combining the techniques used by xCHOKe and RED-PD in identifying and punishing malicious flows while eliminating the complexity of dropping or marking randomly selected packets already queued (a method used by both CHOKe and xCHOKe) and the unreliability of CHOKe hits.

CHAPTER 6

RCUBE

In this chapter, we merge all our previous work to propose RCUBE (**R**eceiver Window Modified **R**andom Early Detection queues with **R**ECHOKe) which combines the advantages of RWM (Receiver Window Modification - Chapter 4) with the RECHOKe scheme (introduced in Chapter 5). By using the RECHOKe scheme, RCUBE easily identifies malicious flows by using CHOKe hit and CHOKe-RED histories, while requiring only a small amount of information, approximately proportional to the order of magnitude of malicious flows. By using the RWM scheme, we reduce the average TCP queue sizes in the queues and in doing so, not only make it easier to identify malicious flows using RECHOKe but also reduce the queuing delay resulting in significant improvements in one-way end-to-end packet delays, delay jitter, throughput and number of dropped packets for TCP-friendly flows. We analyze xCHOKe, RECHOKe and RCUBE in detail using NS2 and show that RCUBE easily outperforms RED, either used by itself or with CHOKe, xCHOKe, or RECHOKe, in identifying, controlling and punishing malicious flows and in protecting TCP-friendly flows.

The rest of this chapter is organized as follows. The next section describes RCUBE, while Section 6.2 presents a comparative analysis of xCHOKe, RECHOKe, and RCUBE. Section 6.3 details simulation results and compares RCUBE to several other schemes. Finally, Section 6.4 concludes the chapter.

6.1 Description of RCUBE

In this section we are proposing RCUBE, an active queue management scheme that merges RWM and RECHOKe, thus combining the advantages of both of these schemes. As RECHOKe requires modifications to network routers we are going to investigate a situation when RWM is implemented at all network routers and not just at ingress or gateway routers. RECHOKe selects a segment to be dropped whenever the flow of that segment dominates the buffer. Hence, TCP flows whose acknowledgements did not follow a symmetrical reverse path will have their packets selected for dropping if they dominate the buffer.

RCUBE works the following way: if the average queue length (*avq*) is greater than the minimum threshold, Th_{min}, and smaller than the maximum threshold, Th_{max}, then the lookup table is checked to see if the arriving packet's flow label is present in it. If there is a table hit, the arriving packet is marked for dropping and the associated hit counter is incremented. A packet is then selected at random from the queue and its flow label is compared with that of the arriving packet. If the flow labels are the same (CHOKe hit), the flow label is added to the lookup table with an initial value of one for the associated hit counter. If the flow is already in the table, the associated hit counter is incremented. Unlike in xCHOKe, packets are not dropped or marked for dropping as a result of CHOKe hits; they are allowed to enter the FIFO buffer. If the packet would be dropped by RED or the ACK would be modified by RWM, then the flows label is added to a lookup table with an initial value of one for an associated hit counter or if the flow is already in the table, the associated hit counter is incremented. This behavior is depicted in the flowchart of figure 6.1.

6.1.1 Complexity Cost

The computational complexity for RCUBE is limited to:

i) a hash lookup in the table for each packet which takes a constant time,

ii) the computation of p^* and c^* if there is a table hit for that packet,

iii) the sampling of the first quarter of the occupied queue at t ms, and

iv) the modification of ACKs by the RWM scheme.

The space complexity of RCUBE is the size of the lookup table which can be either implemented as a variable table which grows as needed to approximately the order of magnitude of malicious flows or based on a Least Recently Used or Least Frequently Used scheme [55].

6.1.2 Theoretical RCUBE *Model*

The approximate analysis for RCUBE is an extension of the analysis for xCHOKe [35][36] where the system is modeled as a queue with a FIFO queuing discipline and assumes that each source is independent and identically distributed, and the arrival process is Poisson. Likewise, we denote the arrival rate of the ith source as λ_i and the mean service time for each packet as $1/\mu$.

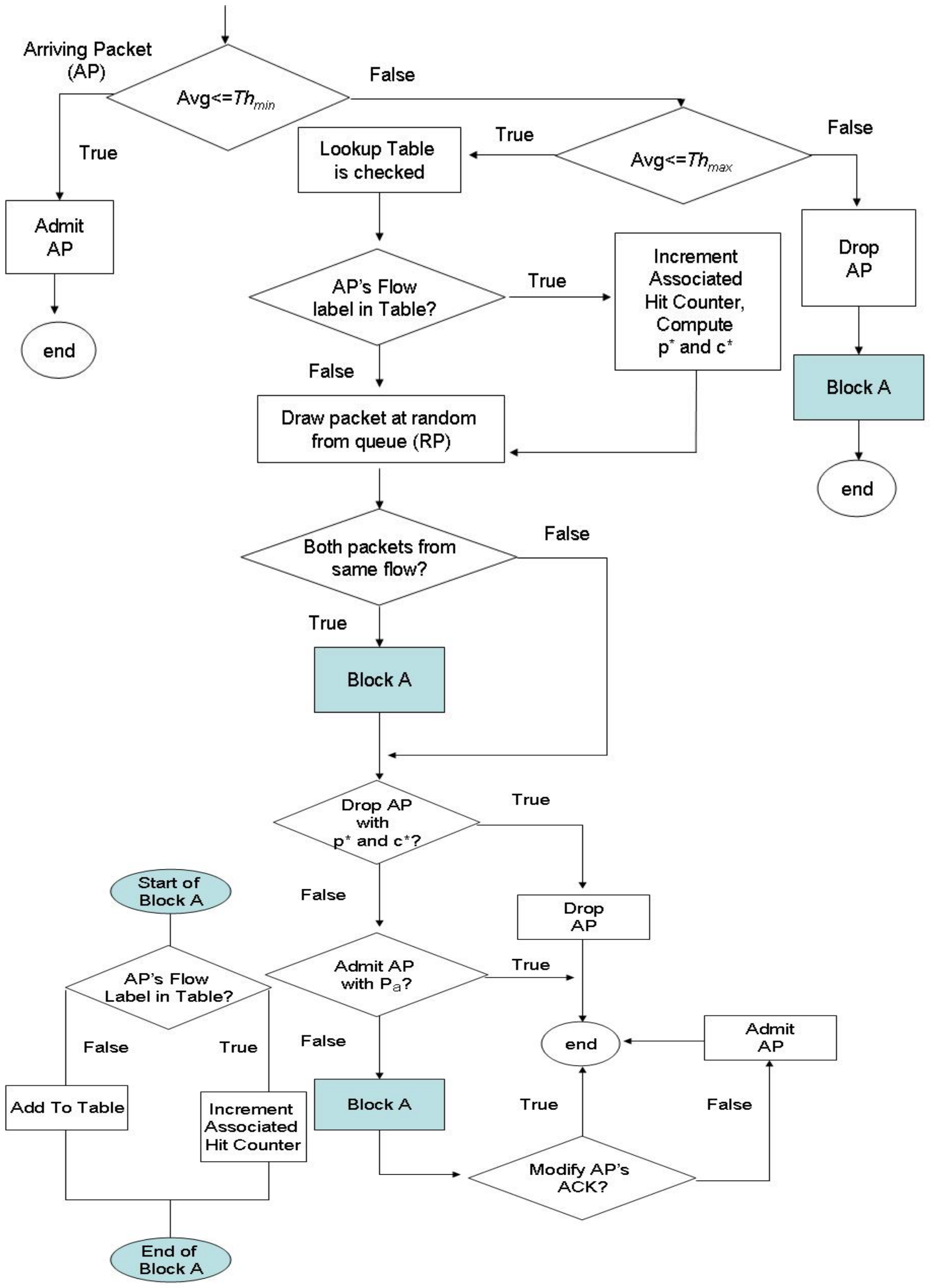

Figure 6.1 The flowchart of RCUBE

An incoming packet may suffer a hit in three phases of the RCUBE algorithm:

i) the flow label of the packet is in the lookup table (table hit),

ii) the flow label of the packet matches that of a randomly drawn packet from the buffer (CHOKe hit) and

iii) a RED drop and/or RWM ACK modification (RED hit).

We will use a similar notation to [35][36], where

1) P_{TABLE} denotes the probability that an incoming packet will suffer a table hit,

2) P_{CHOKe} denotes the probability that the packet will suffer a CHOKe hit, and

3) P_{RED} denotes the RED drop probability, and

4) P_{RWM} denotes the RED-RWM ACK modifying probability.

Let us denote the number of table, CHOKe and RED hits using RCUBE for a table entry by *r*. If the incoming packet's flow label is in the lookup table of size *m*, then the probability of dropping an incoming packet using RCUBE on a table hit can be given by:

$$P^*_{RCUBE} = \min(1, P_{RWM} \times 2^r) \times s^*$$

where *r* is a function of P_{TABLE}, P_{CHOKe} and P_{RWM} and *s** is Boolean function represented by

$$s^* = f\left(r \geq \frac{\sum_{i=1}^{m} r_i}{(2 \times m)} \right)$$

Similarly, for RECHOKe,

$$P^*_{RECHOKe} = \min(1, P_{RED} \times 2^n) \times c^*$$

where n is a function of P_{TABLE}, P_{CHOKe} and P_{RED} and c^* is Boolean function represented by

$$c^* = f\left(n \geq \frac{\sum_{i=1}^{m} n_i}{(2 \times m)} \right)$$

Assuming that no TTLs are refreshed, the maximum value of r and n can be approximated by:

$r = \lambda_i * \text{TTL} * (P_{TABLE} + P_{CHOKe} + P_{RWM})$ and

$n = \lambda_i * \text{TTL} * (P_{TABLE} + P_{CHOKe} + P_{RED})$

where TTL is the initial time to live value for an entry in the lookup table. This is greater than xCHOKe's

$k = \lambda_i * \text{TTL} * (P_{CHOKe})$

Whenever avq is between Th_{min} and Th_{max}, the RWM values for P_{max} is greater than those of RED. Hence, since $P_{RWM} > P_{RED}$ whenever avq is between Th_{min} and Th_{max}, r is always greater than n.

This is reflected in figure 6.2 implying the identification process is more rigorous in RCUBE than in RECHOKe and xCHOKe. Hence, the chances of the malicious flow's packet getting dropped are greater in RCUBE than in RECHOKe and xCHOKe.

The probability that a packet is not dropped by the RCUBE is

$$P_{NoDrop} = (1 - P_{TABLE} \times P_{RWM} \times 2^r \times s^*) \times (1 - P_{RWM}) \qquad 6.1$$

and that of RECHOKe is

$$P_{NoDrop} = (1 - P_{TABLE} \times P_{RED} \times 2^{n} \times c^{*}) \times (1 - P_{RED}) \qquad 6.2$$

xCHOKe's PNoDrop can be given by:

$$P_{NoDrop} = (1 - P_{TABLE} \times P_{RED} \times 2^{k}) \times (1 - P_{CHOKe}) \times (1 - P_{RED}) \qquad 6.3$$

The value of P_{NoDrop} in equation 6.3 is smaller than in equations 6.1 and 6.2, since at any instant of time, $r > n > k$ and if s^* and c^* are one, then it implies that the chances of dropping packets from high-bandwidth flows is greater for RCUBE than for RECHOKe and xCHOKe. This is reflected in figures 6.3 and 6.6. If s^* and c* is zero, it implies that the chances of dropping packets from TCP-friendly flows is greater for xCHOKe than for RECHOKe and RCUBE. This is reflected in figures 6.4 and 6.7.

6.2 Evaluation of RCUBE, xCHOKe and RECHOKe

In this section we analyze xCHOKe, RECHOKe and RCUBE; to do so, we define the following terms. A table hit occurs when the flow id of the arriving packet matches a flow id entry already in the table. A table hit can be either good or bad. The table hit is good if the flow that the hit packet belongs to is malicious; it is a bad hit otherwise. Table drops are table hits that were selected for dropping by xCHOKe (after drawing a random with probability p*); for RCUBE and RECHOKe, these are table hits that were selected after evaluating p^* and c^*. Good table drops occur when the dropped packet belongs to a malicious flow; bad table drops occur otherwise. A CHOKe miss victim is a packet which after experiencing a CHOKe miss enters the RED queue. CHOKe miss RED hit occurs when the flow id of the CHOKe miss victim matches the flow whose packets dominate the buffer at the time of selection of the CHOKe miss victim; CHOKe miss RED miss occur otherwise. A CHOKe miss RED hit could be either a good or bad: it is good if the victim packet belongs to a malicious flow; it is bad otherwise. A CHOKe miss RED miss again could be either a good or a bad CHOKe miss RED miss. Good CHOKe miss RED miss is when the victim packet belongs to a non-malicious flow; it is bad otherwise.

Using NS2, we simulate a bar-bell topology and analyze the xCHOKe, RECHOKe and RCUBE buffers in terms of buffer occupancy by a malicious flow (a constant-bit-rate UDP flow) competing with 10 TCP flows over a 3Mbps link. All TCP flows have the same round trip propagation delay of 20ms with each output link having a latency of 1ms and capacity of 10Mbps. The parameters for the xCHOKe, RECHOKe and RCUBE buffers are: $Th_{min} = 10$, $Th_{max} = 50$, $P_{max} = 0.1$.

We ran 11 sets of simulations, starting with a malicious UDP rate of 0.5Mbps, increasing the rate in 1Mbps increments. Each simulation was long enough for initial transients to settle and be insignificant. Where appropriate, enough simulations were run to claim a 95% confidence that the confidence intervals are less than 5% of the mean.

Figure 6.2 shows the number of table hits for the xCHOKe, RECHOKe and RCUBE algorithms. As it can be observed, the number of hits for RCUBE is greater than those for xCHOKe and RECHOKe. The number of hits for RCUBE is 55-65% greater than with xCHOKe and slightly greater than RECHOKe. RCUBE’s dominance is due to:

1) RWM’s ability of keeping a lower average queue length than RED for TCP originated packets and

2) the combination of CHOKe and CHOKe-RED histories, as RWM allows RCUBE to register a large number of hits and using these hits, RCUBE can isolate malicious flows faster and with a higher accuracy.

RCUBE’s performance dominance increases when the number and/or rate of TCP-friendly and/or UDP flows increases. RCUBE has the:

3) greatest number of good table hits (as shown in figure 6.3) and

2) smallest number of bad table hits (as shown in figure 6.4), thus providing the best protection for TCP flows.

Next, we analyze the number of table drops for table hits for all three algorithms. Again, ideally, drops should only consist of good table drops. Figures 6.5 to 6.7 show the number of table drops, good table drops and bad table drops respectively. We can observe that RCUBE outperforms xCHOKe and RECHOKe in all of these figures. We have also observed, that nearly all table misses (not shown) for all algorithms were good misses. From figures 6.6 and 6.7 we can deduce that RCUBE drops the greatest number of packets from the malicious flow while dropping the least number of packets from TCP-friendly flows. Hence, figure 6.6 shows that RCUBE is the fastest and most accurate in punishing malicious flows while figure 6.7 shows that it affects the TCP-friendly flows the least. By affecting the TCP-friendly flows the least and at the same time, punishing malicious flows, RCUBE provides the best protection for TCP-friendly flows.

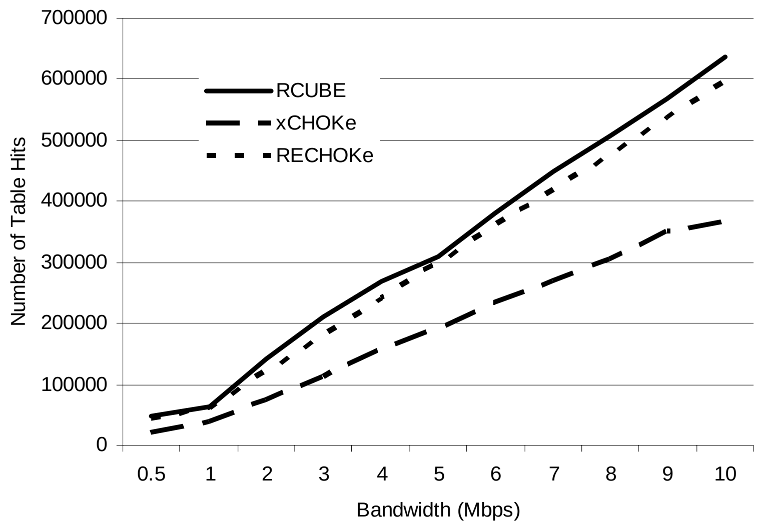

Figure 6.2 The number of table hits vs. UDP rate

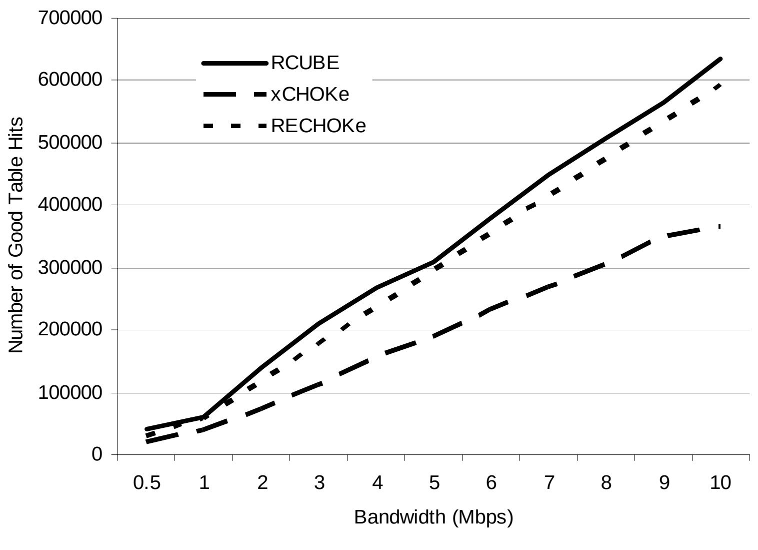

Figure 6.3 The number of good table hits vs. UDP rate

Figures 6.8, and 6.9 show the number of CHOKe hits and misses respectively. xCHOKe has the greatest number of CHOKe hits, followed by RCUBE and RECHOKe. To analyze these hits, we looked at the number of good (as shown in figure 6.10) and bad (as shown in figure 6.11) CHOKe hits. xCHOKe and RCUBE had the greatest number of good and bad CHOKe hits respectively. When compared to RCUBE and RECHOKe, xCHOKe has the greatest number of UDP packets in the buffer and hence, has a higher probability of getting many good hits. Meanwhile, RCUBE needs the smallest queue size among the three resulting in the greatest number of bad CHOKe hits. This is due to less TCP segments placed in the buffer due to the RWM scheme. Unlike xCHOKe, both RCUBE and RECHOKe update their tables after table hits. This is necessary as with the lifetime of flows, CHOKe hits become more unreliable (since the number of packets belonging to the malicious flow in the buffer are reduced drastically leading to a greater number of bad CHOKe hits). The same phenomenon can be observed in xCHOKe and CHOKe, leading to an unnecessary punishment of TCP-friendly flows if packets are to be dropped during CHOKe hits. Hence, in both RCUBE, and RECHOKe packets are not dropped during CHOKe hits but passed on to the RED buffer. Although these hits are updated in the table, the $c*$ condition prevents unnecessary drops of TCP-friendly packets.

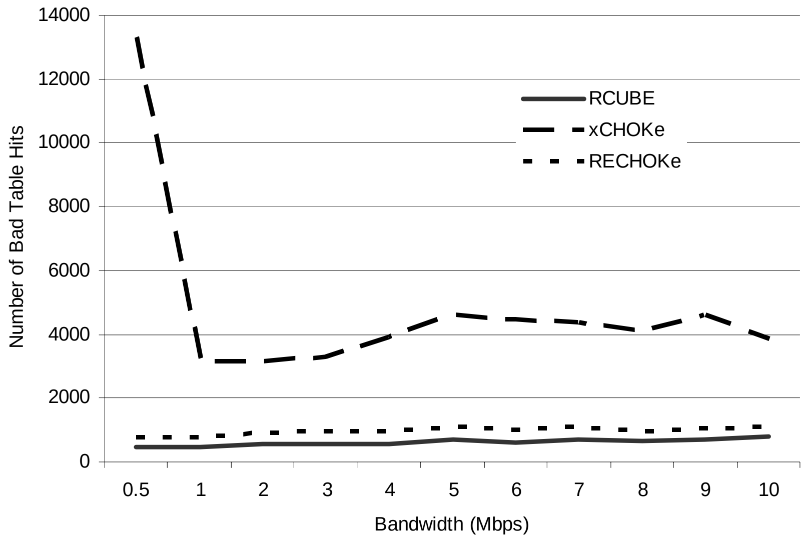

Figure 6.4 The number of bad table hits vs. UDP rate

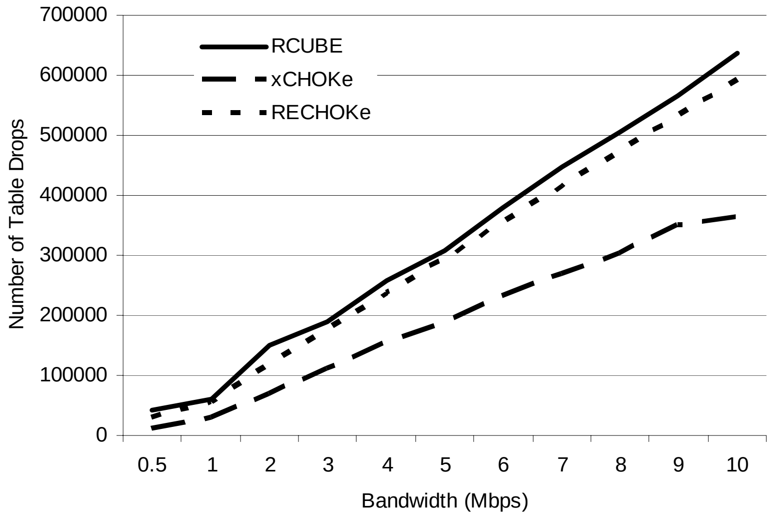

Figure 6.5 The number of table drops vs. UDP rate

Upon analyzing the CHOKe misses in figures 6.9, 6.12, and 6.13, we have found that RCUBE significantly outperforms xCHOKe and RECHOKe in the number of bad CHOKe misses while RECHOKe performs the best in the number of good CHOKe misses. Hence, both RECHOKe and RCUBE provide better protection than xCHOKe. Analyzing CHOKe misses (as shown in figure 6.9) we find that RECHOKe, compared with RCUBE, had more number of good CHOKe misses due to its larger TCP average queue size. The greater number of table hits (as shown in figure 6.3) for RCUBE imply that fewer packets from malicious flows are allowed to enter the RED buffer, leading to overall smaller average queue sizes. This again leads to significant improvements in one-way, end-to-end packet delays, delay jitter, throughput and number of dropped packets, similarly to the claims of [56][57][58].

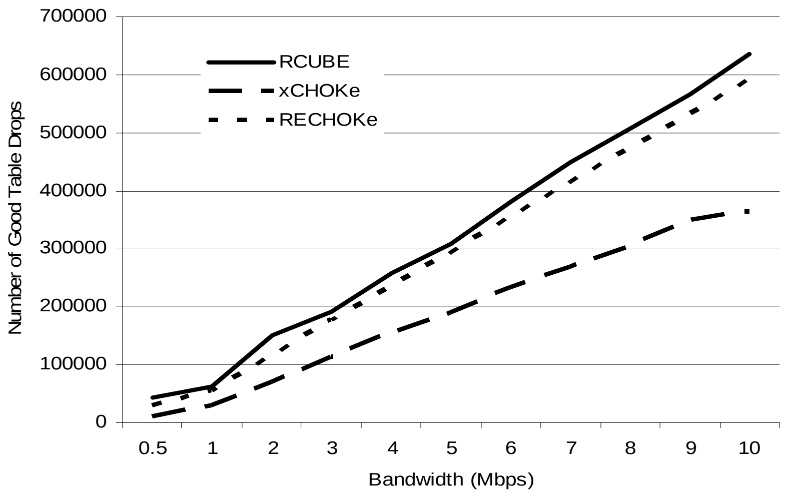

Figure 6.6 The number of good table drops vs. UDP rate

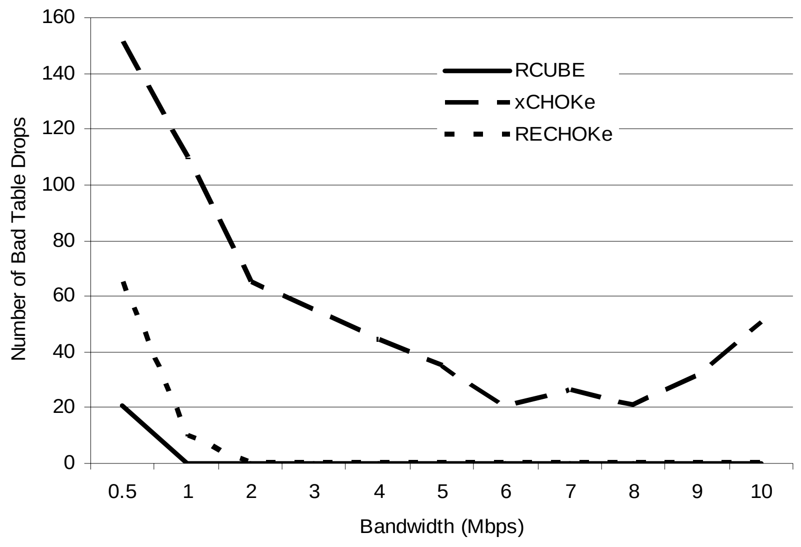

Figure 6.7 The number of bad table drops vs. UDP rate

Next, we analyze RED hits and misses for all three algorithms. Figure 6.14 shows that the number of dropped packets due to RED hits is the largest when using RECHOKe. It is greater than in RCUBE because of RECHOKe's larger queue sizes. The amount of dropped packets is greater for both RECHOKe and RCUBE buffers when compared to xCHOKe. This is because during CHOKe hits packets are not dropped but admitted into the RECHOKe and RCUBE buffers and as a result more packets are dropped by the RED buffer after RECHOKe and RCUBE than after xCHOKe.

Figures 6.15 to 6.17 show the number of RED hits for RECHOKe, xCHOKe, and RCUBE. RED hits are mainly good, implying that most of the dropped packets are from malicious flows. This happens despite the fact, that most of the packets admitted into the buffer are from the 10 TCP-friendly flows. After the good RED hits malicious flows become no longer dominant. Hence, RED drops even more UDP packets from the few that have been admitted into the buffer. Bad hits happen mainly because a TCP-friendly flow was dominating the buffer at the time of drop or window modification and its packet was chosen to be dropped.

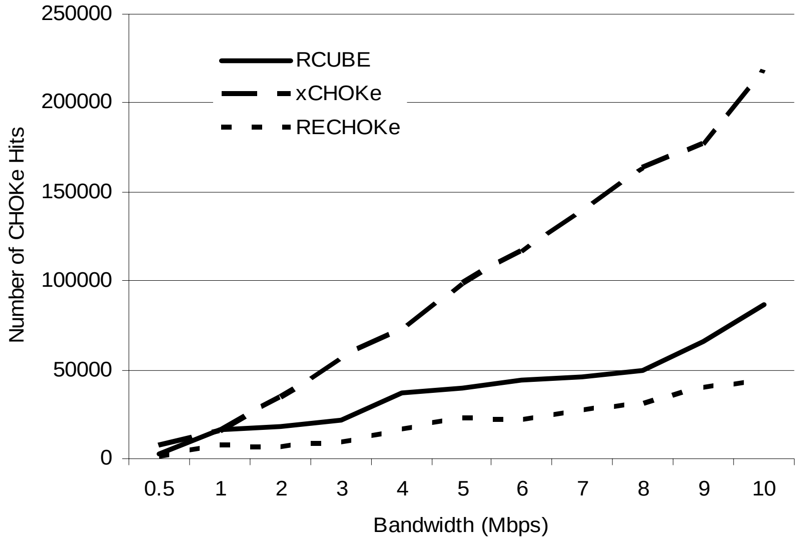

Figure 6.8 The number of CHOKe hits vs. the UDP rate

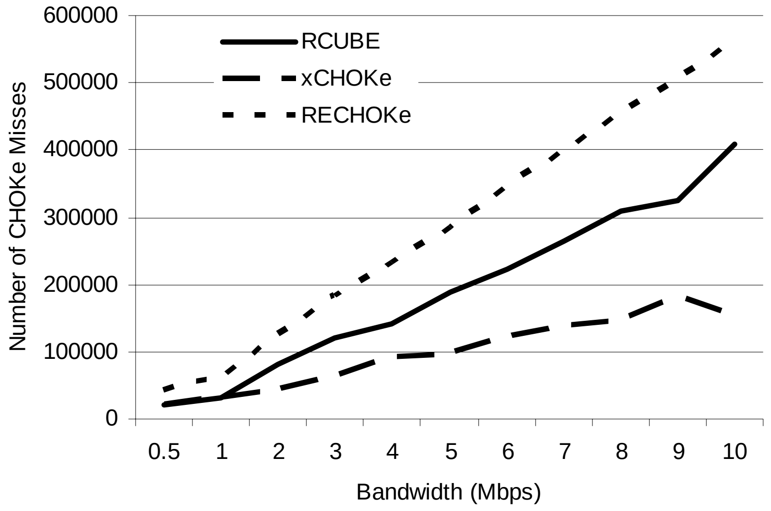

Figure 6.9 The number of CHOKe misses vs. the UDP rate

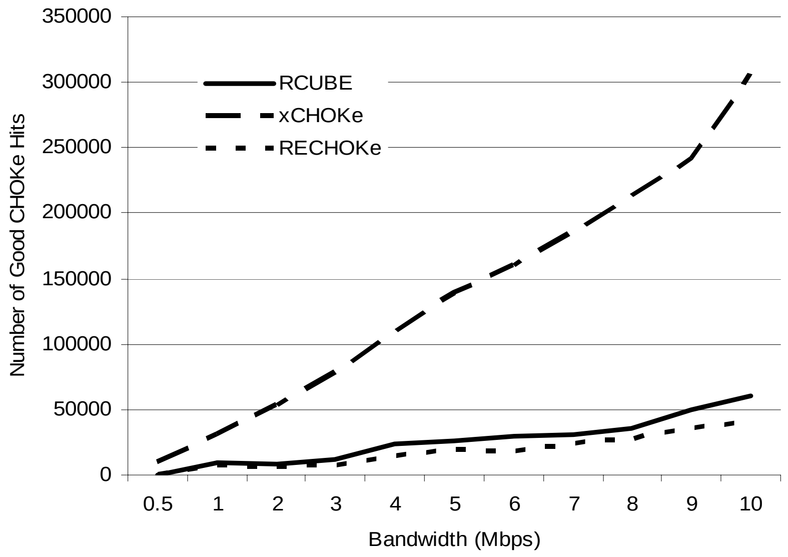

Figure 6.10 The number of good CHOKe hits vs. the UDP rate

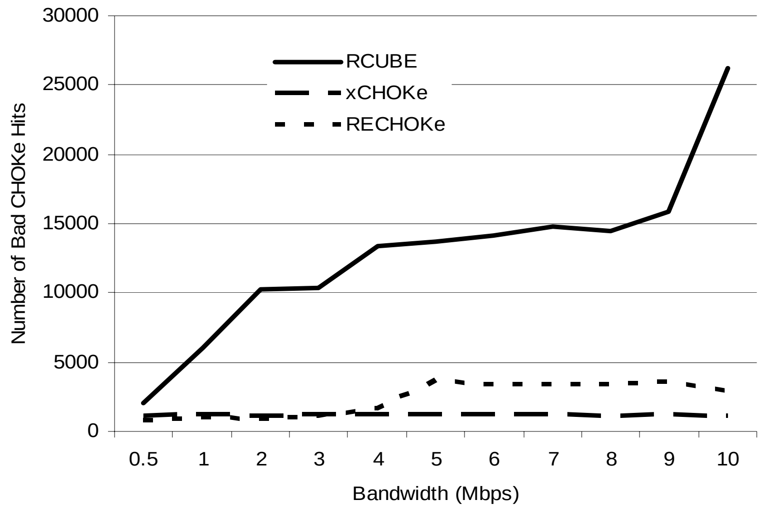

Figure 6.11 The number of bad CHOKe hits vs. the UDP rate

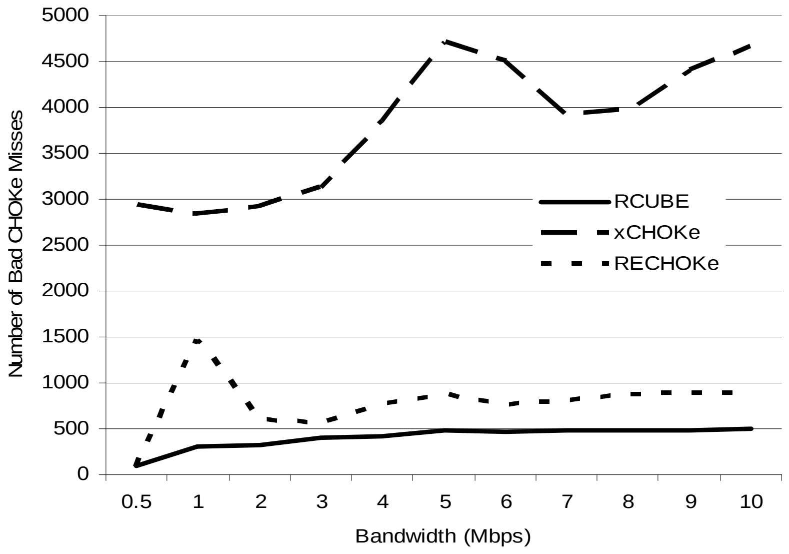

Figure 6.12 The number of bad CHOKe misses vs. the UDP rate

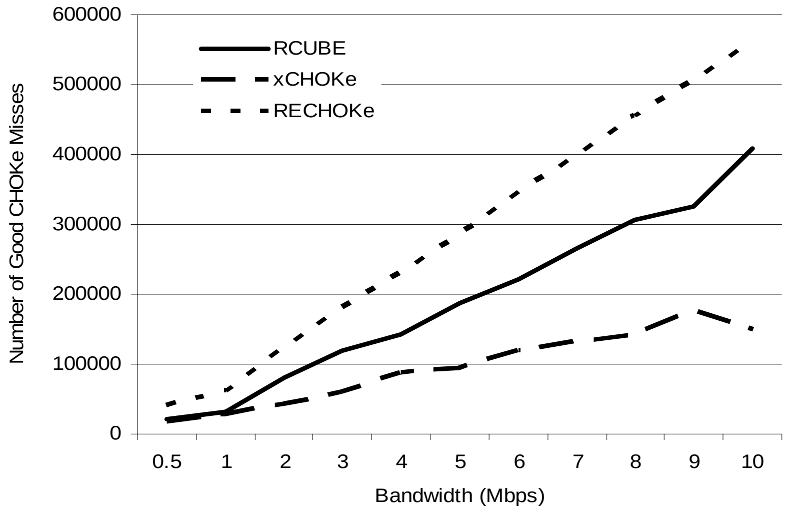

Figure 6.13 The number of good CHOKe misses vs. the UDP rate

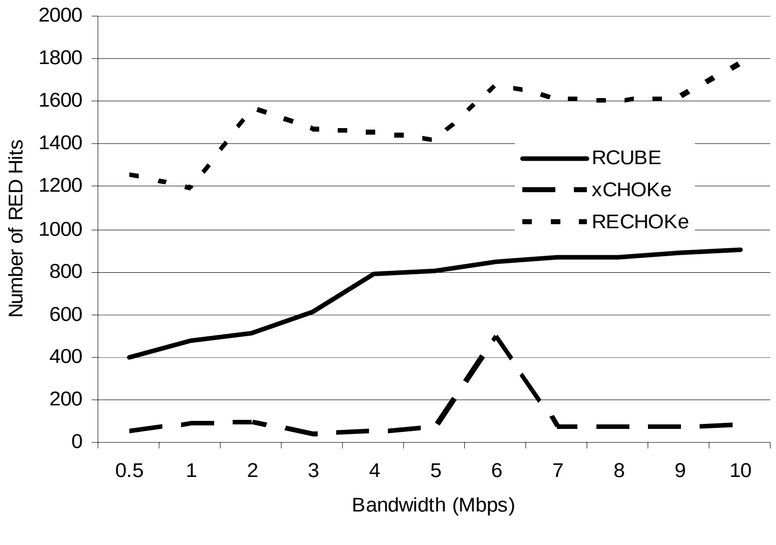

Figure 6.14 The number of RED hits vs. the UDP rate

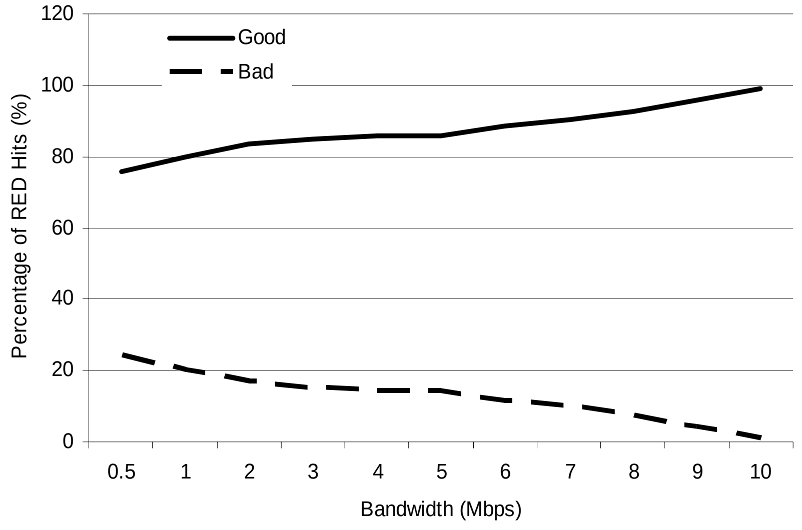

Figure 6.15 The percentages of good and bad RED hits vs. the UDP rate (RCUBE)

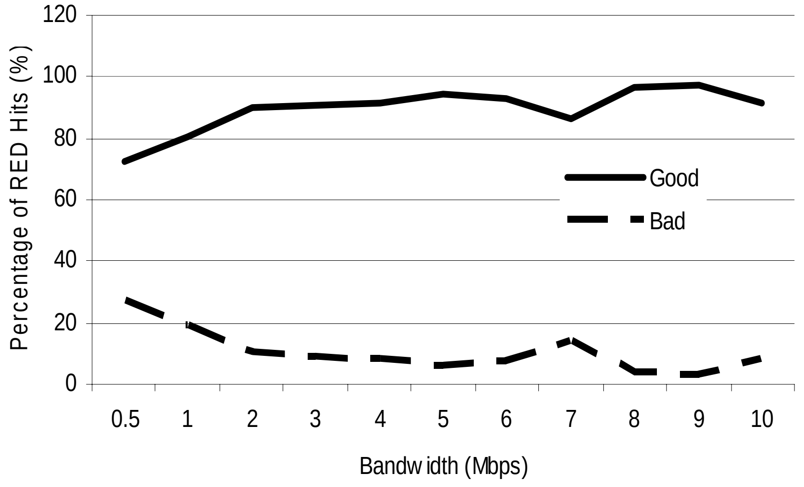

Figure 6.16 The percentages of good and bad RED hits vs. the UDP rate (RECHOKe)

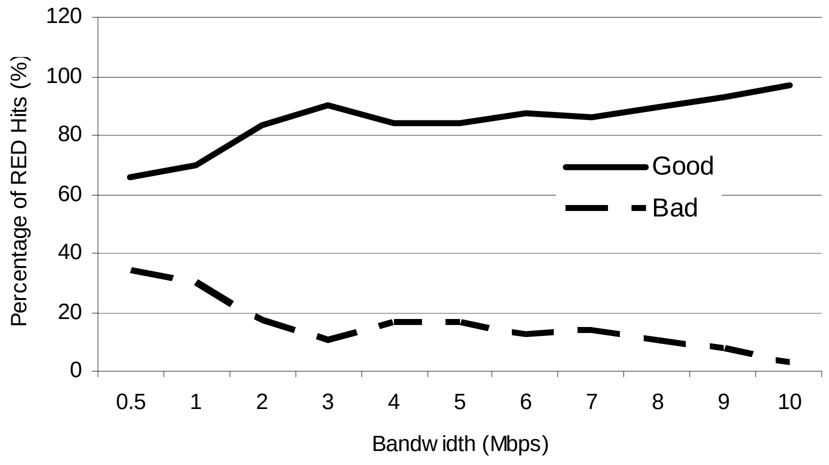

Figure 6.17 The percentages of good and bad RED hits vs. the UDP rate (xCHOKe)

6.3 Comparing RCUBE with RED, CHOKe, xCHOKe and RECHOKe

In this section, we present more experiments to compare RCUBE with RED, CHOKe, xCHOKe, and RECHOKe. In these sets of experiments we again use the same bar-bell topology as before (see figure 5.2), however with two malicious flows. Each of these flows is a constant-bit-rate UDP flow set at a) 1.2Mbps and b) 2Mbps, competing with 10 TCP flows over the 3Mbps “R1-R2” link. (UDP rates of 1.2Mbps result in flows with a capacity less than the link capacity, while using 2Mbps flows put us beyond this capacity.) The TTL is 10ms for both xCHOKe and RECHOKe. The RED parameters for the RED, CHOKe, xCHOKe, RECHOKe and RCUBE buffers are: $Th_{min} = 10$, $Th_{max} = 50$, $P_{max} = 0.1$.

We again consider two scenarios; in the first scenario, we study the impact of two malicious flows with non-identical rates on a set of well-behaved TCP flows with identical RTTs while in Scenario 2, we deal with non- identical RTTs.

6.3.1 Scenario 1

Figures 6.18 to 6.22 show the bandwidth allocation for the congested link for each flow with RED, CHOKe, xCHOKe, RECHOKe, and RCUBE buffers respectively at the low UDP rate (less than the link capacity). Figure 6.23 shows a comparison of bandwidths of the malicious flows for all five schemes while figure 6.24 compares RCUBE, and RECHOKe. RED, as expected, does not control the malicious flows (UDP1 and UDP2) from gaining control of almost the entire link bandwidth; the bandwidth shares of all TCP-friendly flows are almost zero. CHOKe and xCHOKe do control malicious flows; however TCP flows suffer CHOKe hits and as a result suffer bandwidth share losses. On the other hand, by reducing bad CHOKe hits, RECHOKe controls malicious flows thus offering a better protection for TCP-friendly flows. However, RCUBE does control the two malicious flows better than RECHOKe. Although the average queue lengths of RCUBE are less than those of RECHOKe, a larger number of good table drops (as shown in figure 6.6) and smaller number of bad table drops (as shown in figure 6.7) enable RCUBE to control the two malicious flows better.

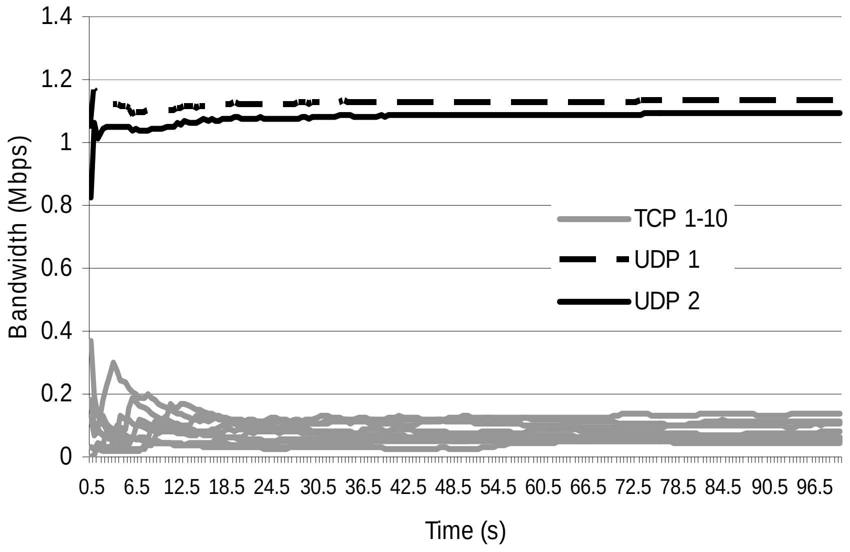

Figure 6.18 Link utilization with RED

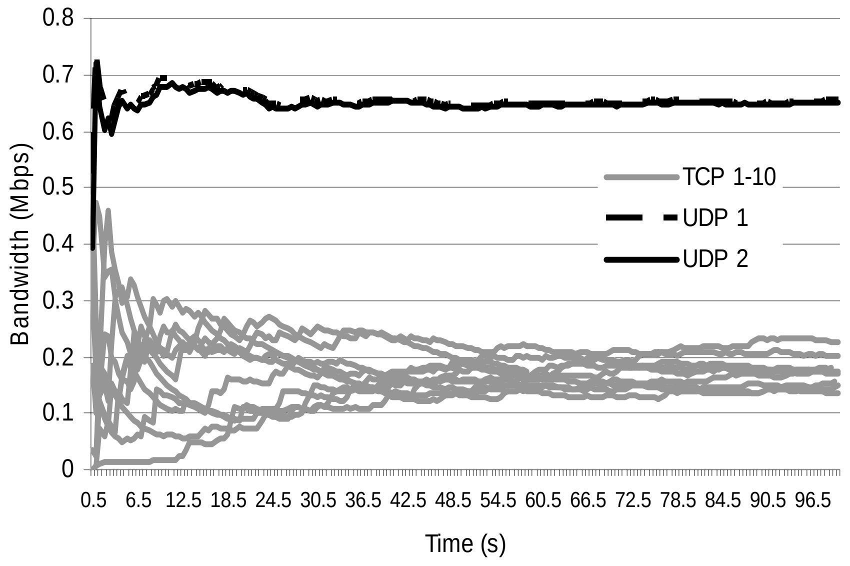

Figure 6.19 Link utilization with CHOKe

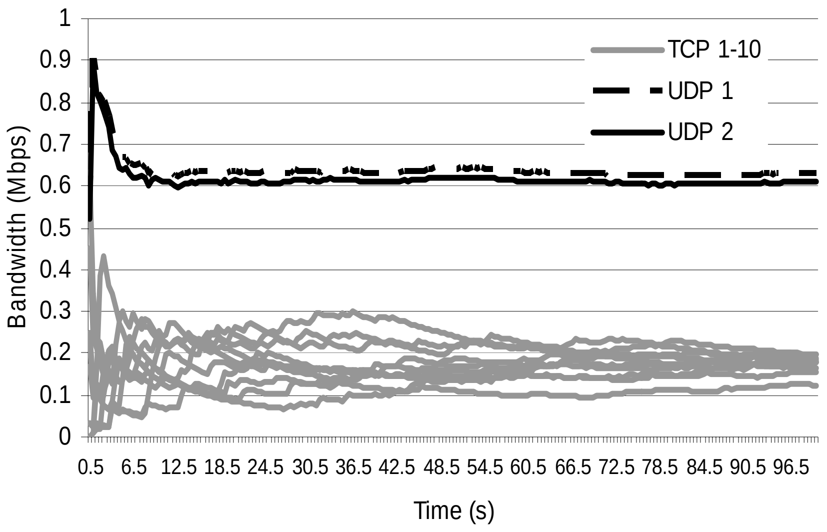

Figure 6.20 Link utilization with xCHOKe

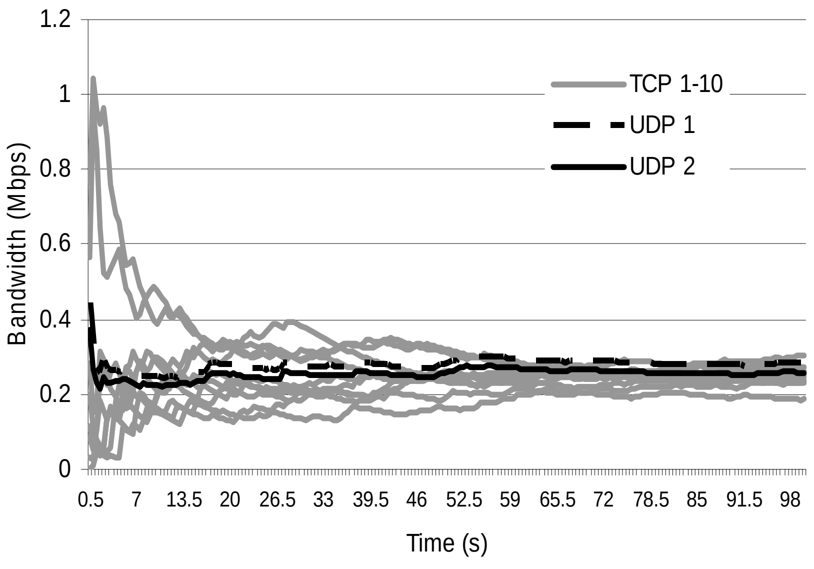

Figure 6.21 Link utilization with RECHOKe

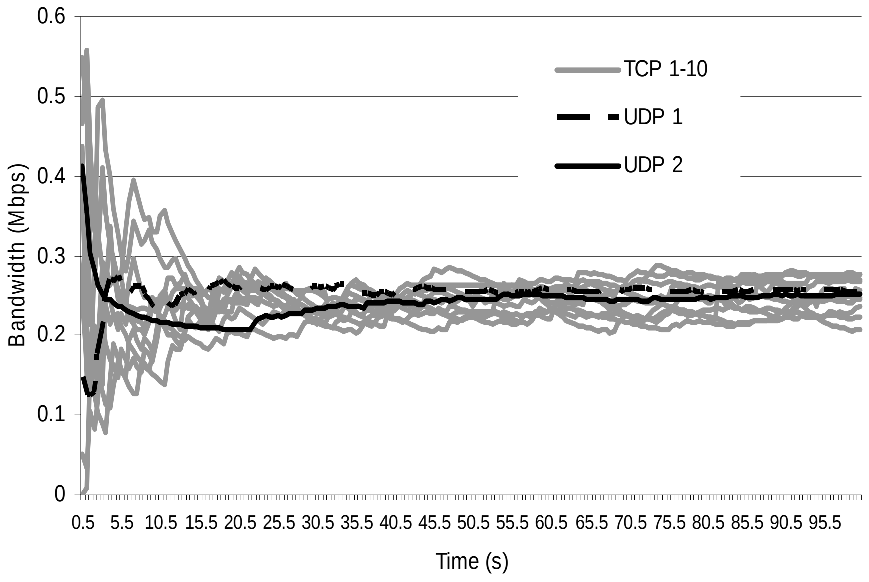

Figure 6.22 Link utilization with RCUBE

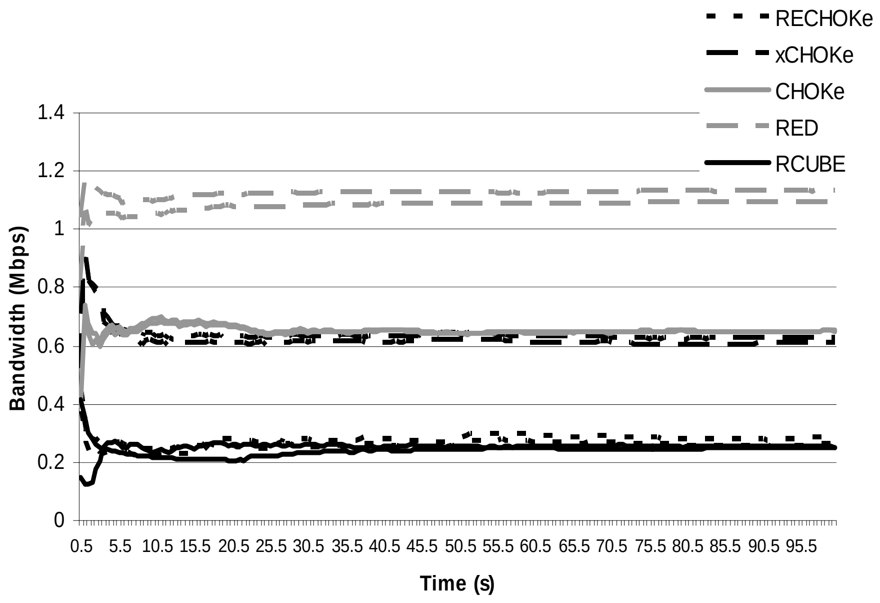

Figure 6.23 UDP flow rates with RCUBE, RECHOKe, xCHOKe, CHOKe, and RED

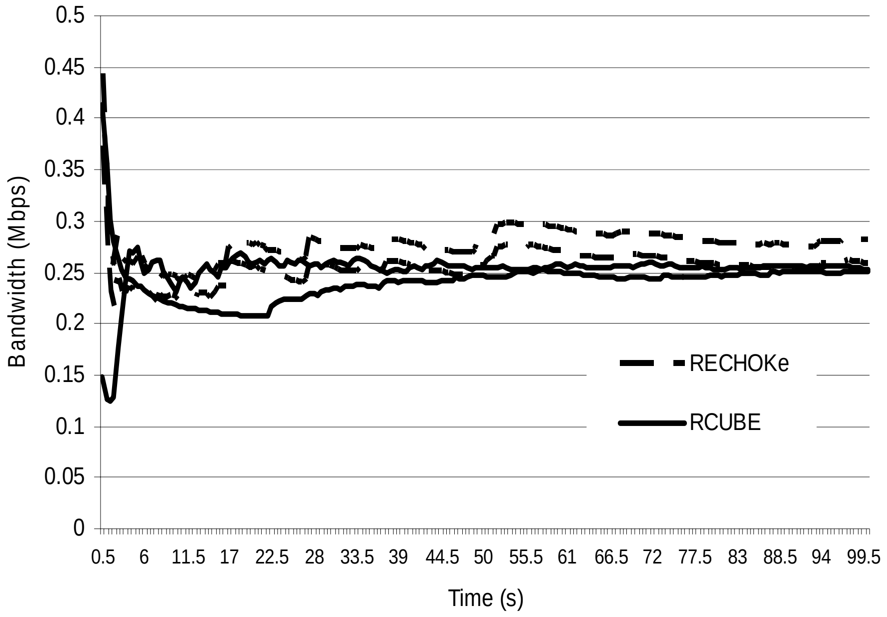

Figure 6.24 Link utilization of UDP flows for RECHOKe and RCUBE

Figures 6.25 to 6.29 show results for the congested link for each flow with RED, CHOKe, xCHOKe, RECHOKe and RCUBE buffers respectively when the UDP flows are set to their high rate. Figure 6.30 shows a comparison of bandwidths of the malicious flows for all five schemes while Figures 6.31 to 6.35 present comparisons of scheme at rates of 1.2Mbps and 2Mbps. Finally, figure 6.36 compares RCUBE and RECHOKe at the high UDP flow rate. Figures 6.31, 6.32, and 6.33 show that as the rate of the malicious flows is increased from 1.2Mbps to 2Mbps, TCP flows lose more of their bandwidth share to the malicious flows. However, figures 6.34 and 6.35 show that both RECHOKe and RCUBE increase their control and punishment of the two malicious flows in these cases. Figure 6.36 shows that RCUBE controls malicious flows better than RECHOKe. Although the average queue length of RCUBE is less than that of RECHOKe, more good table drops (as shown in figure 6.6) and less bad table drops (as shown in figure 6.7) enable RCUBE to control the malicious flows better. Figures 6.28 and 6.29 also show that RCUBE drops fewer packets from TCP-friendly flows than RECHOKe.

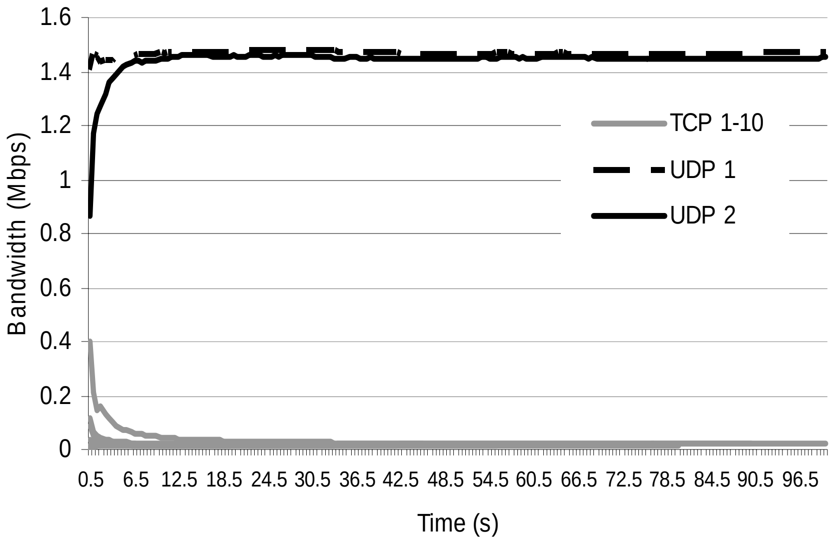

Figure 6.25 Link utilization with RED

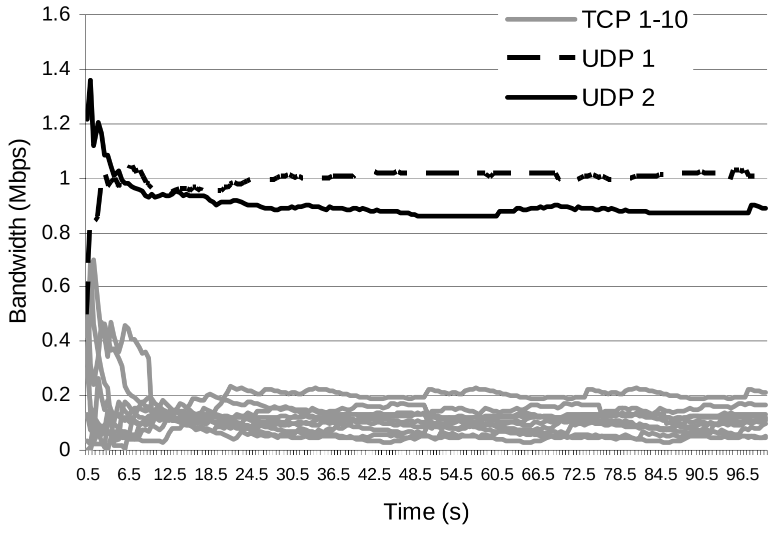

Figure 6.26 Link utilization with CHOKe

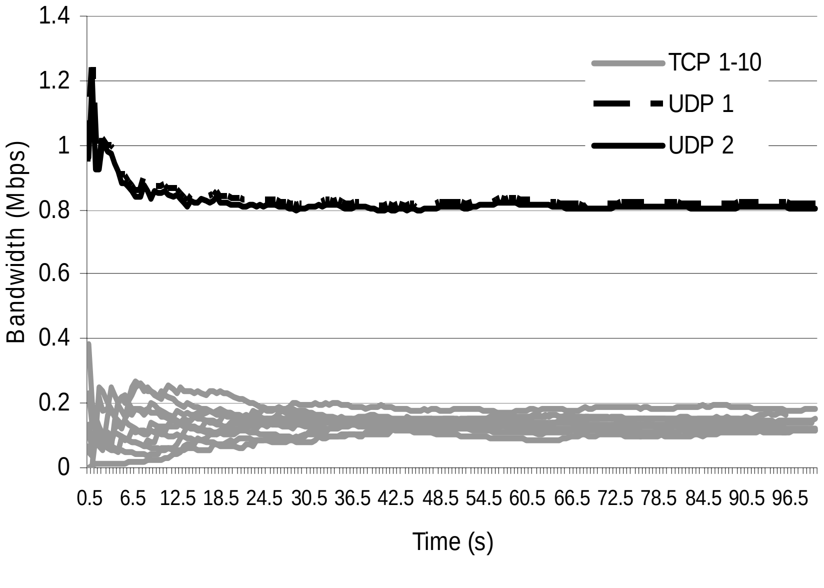

Figure 6.27 Link utilization with xCHOKe

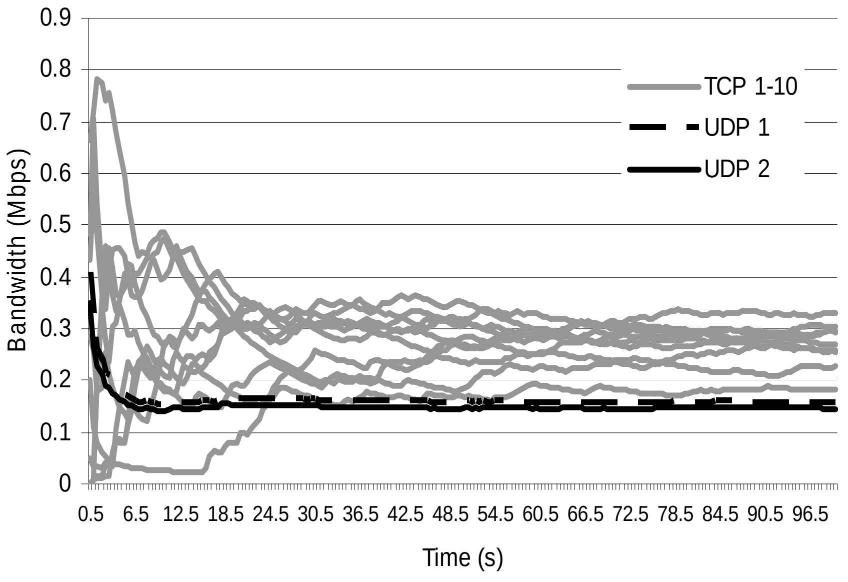

Figure 6.28 Link utilization with RECHOKe

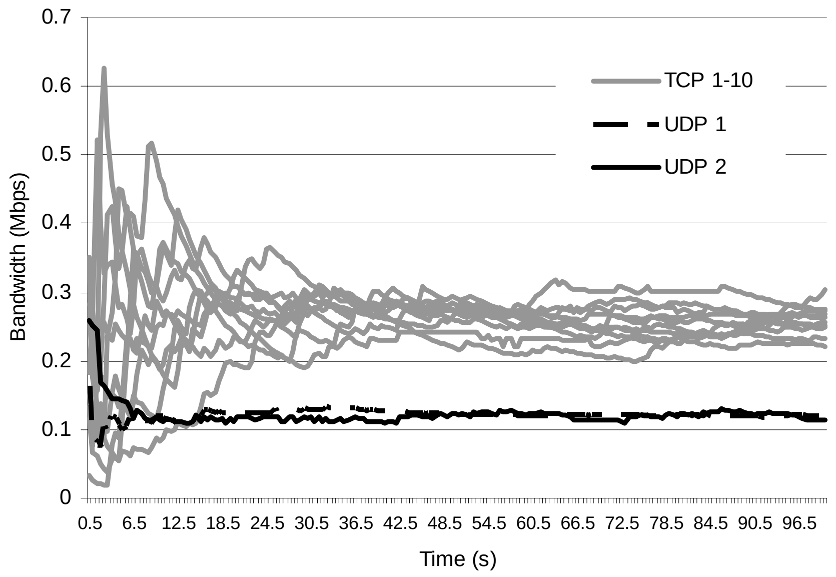

Figure 6.29 Link utilization with RCUBE

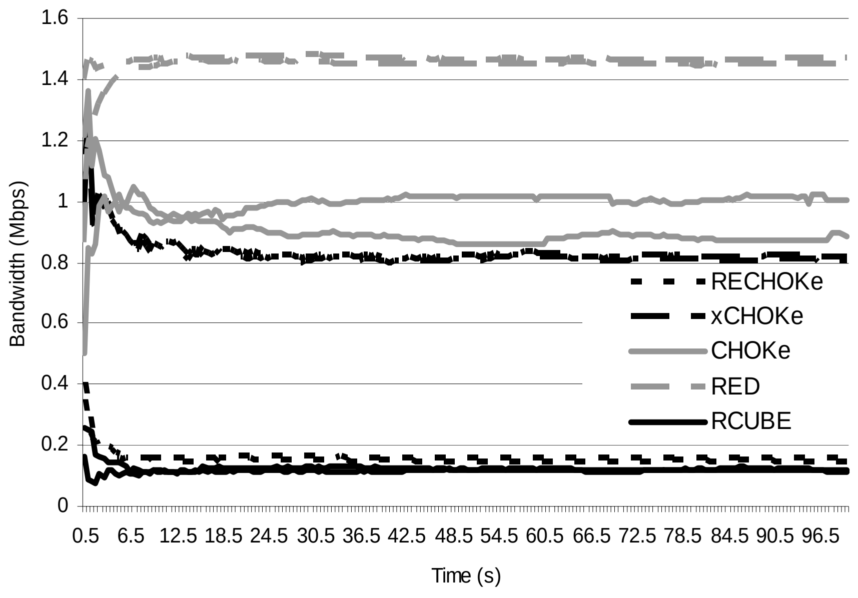

Figure 6.30 UDP flow shares with RCUBE, RECHOKe, xCHOKe, CHOKe, and RED

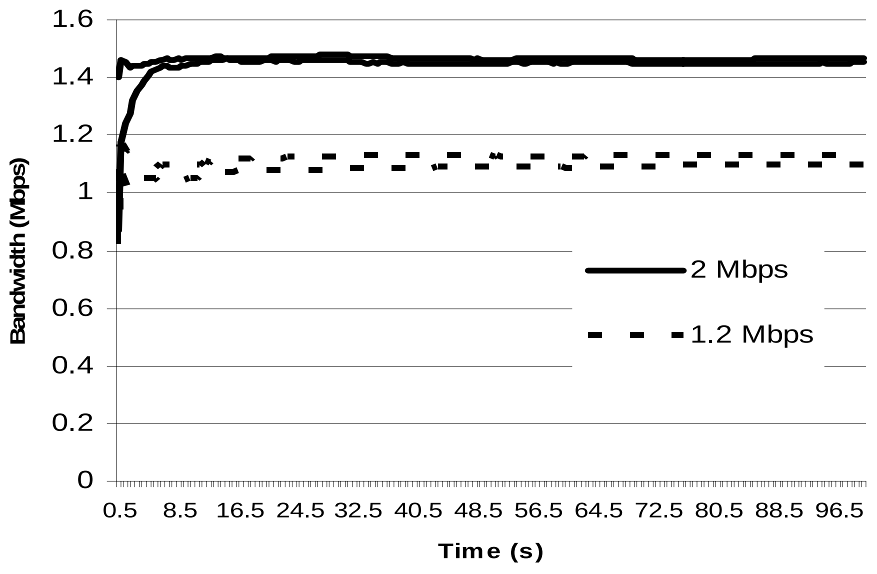

Figure 6.31 Link utilization of UDP flows at 1.2Mbps and 2 Mbps with RED

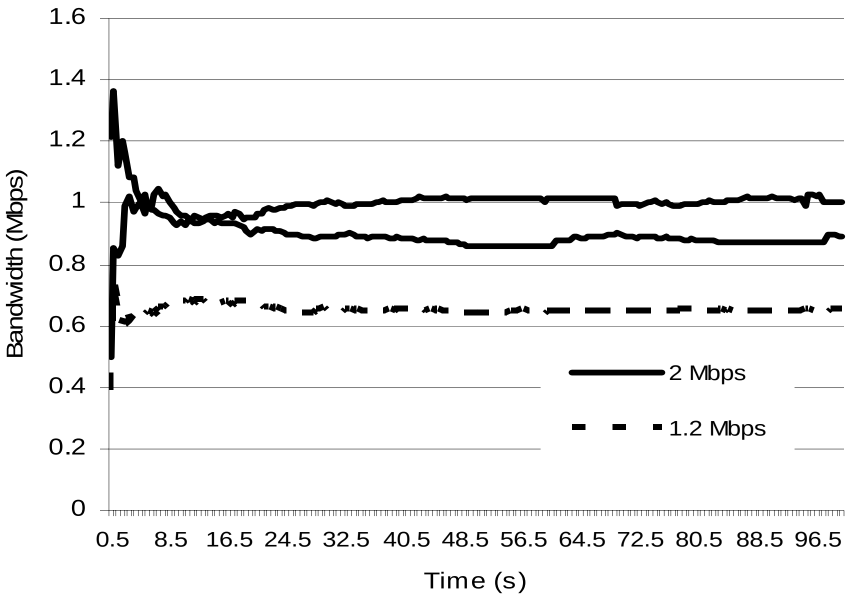

Figure 6.32 Link utilization of UDP flows at 1.2Mbps and 2 Mbps with CHOKe

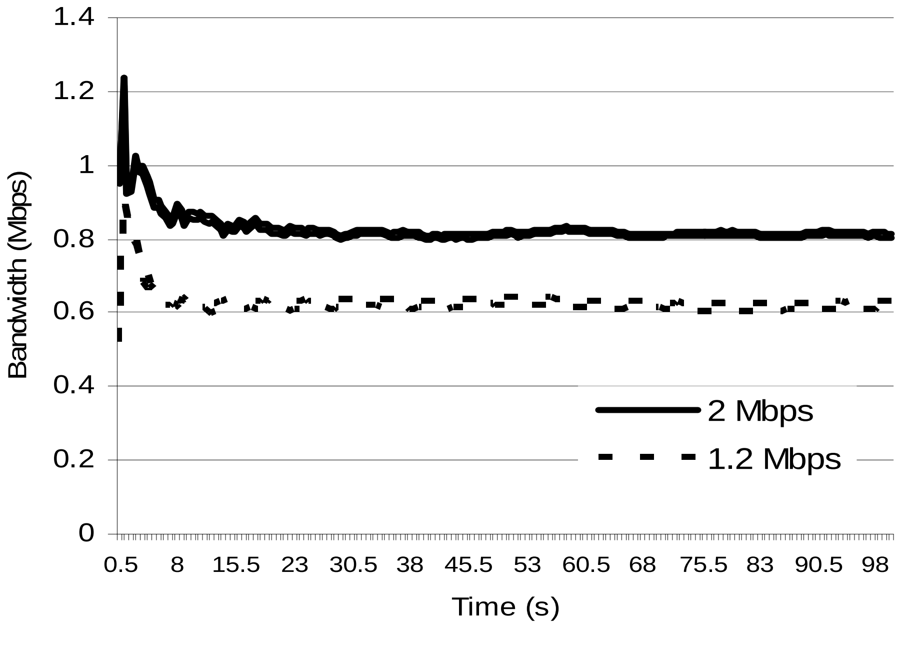

Figure 6.33 Link utilization of UDP flows at 1.2Mbps and 2 Mbps with xCHOKe

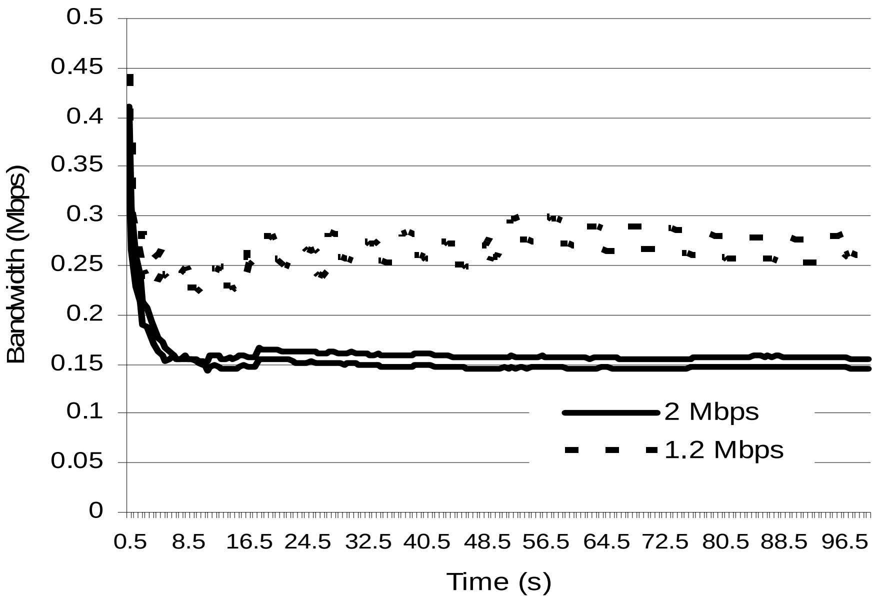

Figure 6.34 Link utilization of UDP flows at 1.2Mbps and 2 Mbps with RECHOKe

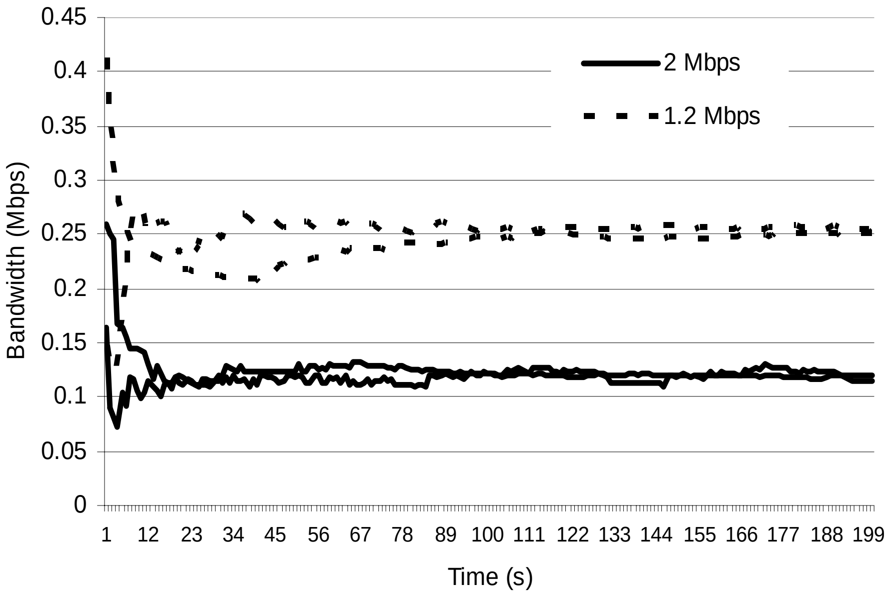

Figure 6.35 Link utilization of UDP flows at 1.2Mbps and 2 Mbps with RCUBE

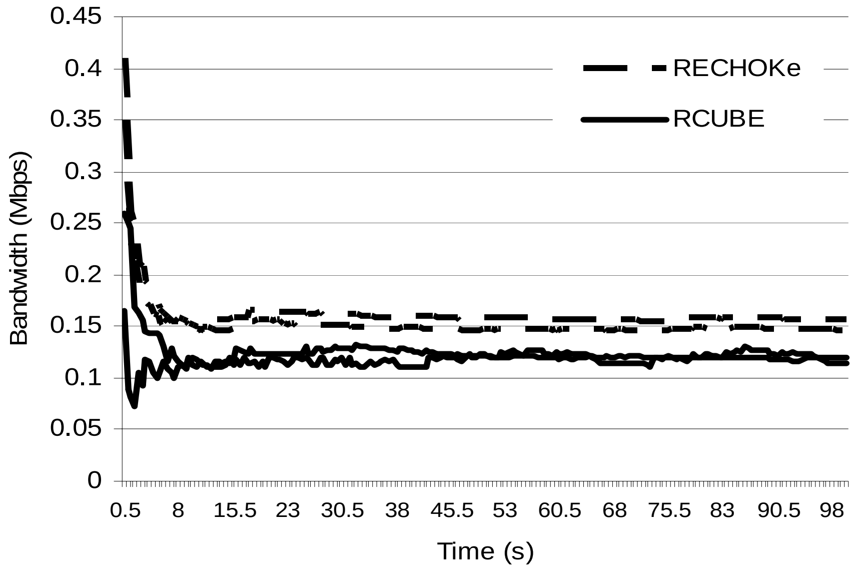

Figure 6.36 Link utilization of UDP flows with RECHOKe and RCUBE

6.3.2 Scenario 2

In these experiments, the network setup is similar to that of the previous section; however, we make the two UDP flows rates asymmetric by setting one at 1Mbps while the other at 3Mbps. In addition, the ten TCP flows are simulated with non-identical round trip delays at 2ms delay increments across the flows. The “R1-R2” link’s bandwidth is set at 3Mbps; RED parameters are the same as before.

Figures 6.37 to 6.41 show the bandwidth allocation of the congested link for each flow for RED, CHOKe, xCHOKe, RECHOKe and RCUBE buffers respectively. RCUBE again controls malicious flows better than RECHOKe. Although the average queue length in RCUBE is less, the larger number of good table drops (as shown in figure 6.6) and the fewer number of bad table drops (as shown in figure 6.7) enable RCUBE to achieve better control. Figures 6.42 and 6.43 also show that the RCUBE drops fewer packets from TCP-friendly flows leading to more stable bandwidth allocation even though TCP flows have heterogeneous RTTs.

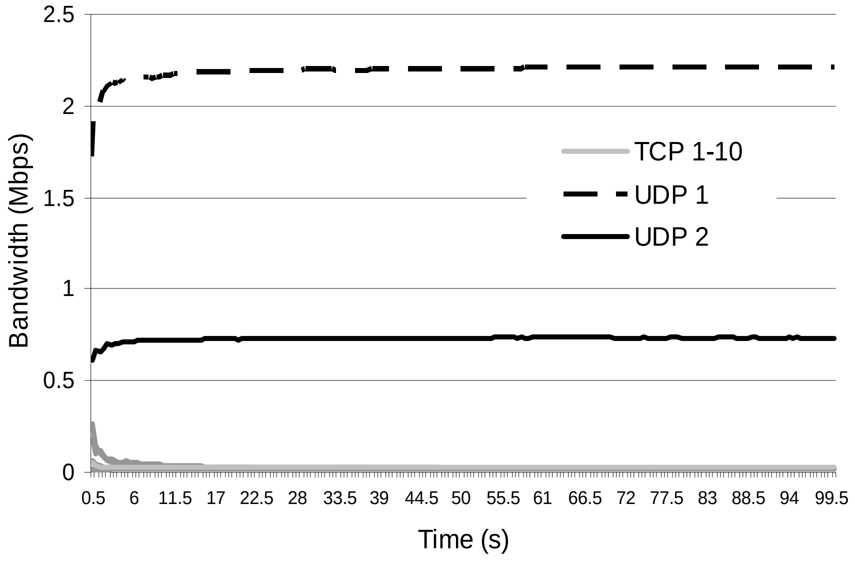

Figure 6.37 Link utilization with RED

6.4 Summary

In this chapter, we have presented RCUBE which combined the advantages of both RWM and RECHOKe. We analyzed xCHOKe, RECHOKe and RCUBE in detail using NS2 and showed that RCUBE better all of the other schemes studied and provided with a discussion on why that is. We have also compared RCUBE with RED, either used alone or with RECHOKe, xCHOKe, and CHOKe; we have shown that RCUBE performed the best among all these schemes.

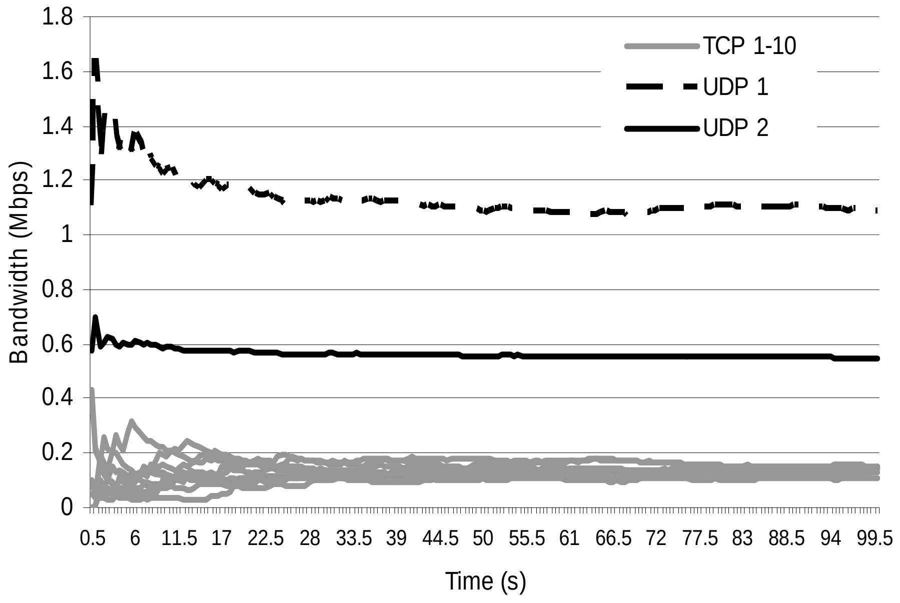

Figure 6.38 Link utilization with CHOKe

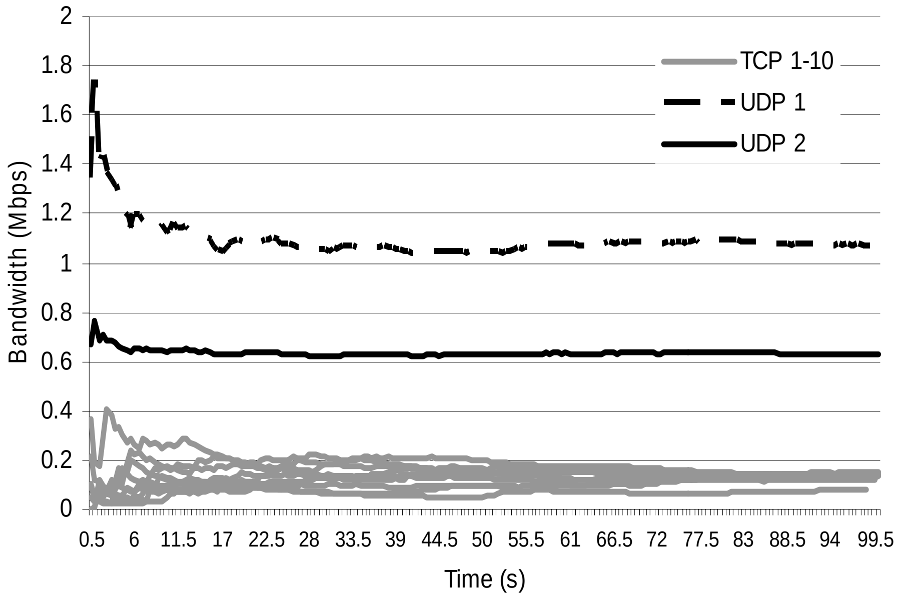

Figure 6.39 Link utilization with xCHOKe

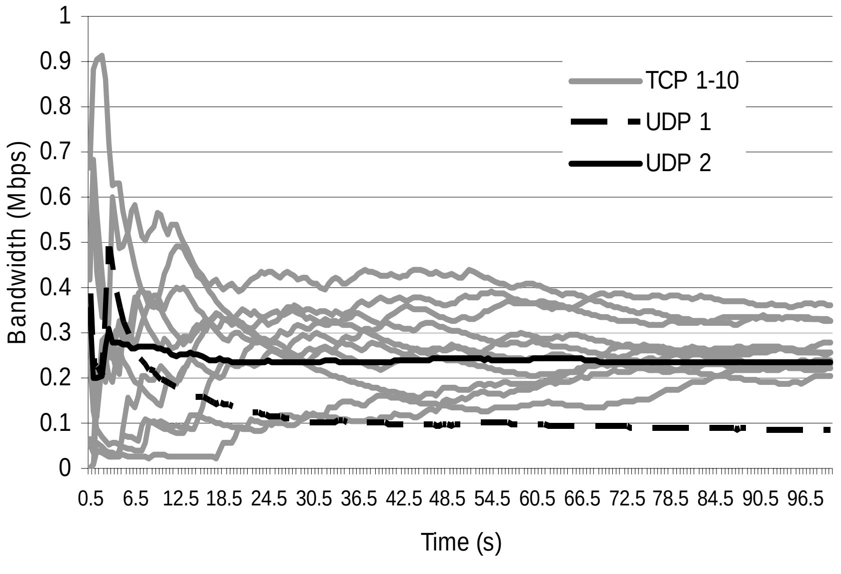

Figure 6.40 Link utilization with RECHOKe

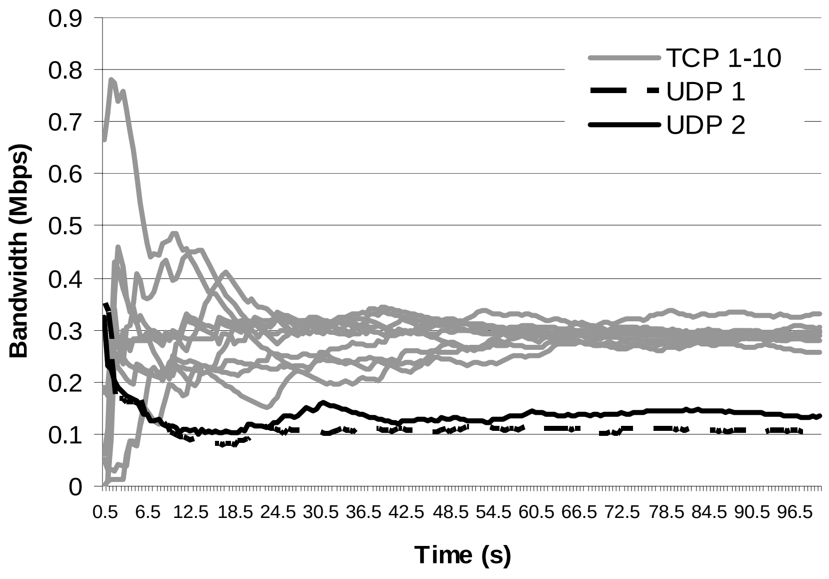

Figure 6.41 Link utilization with RCUBE

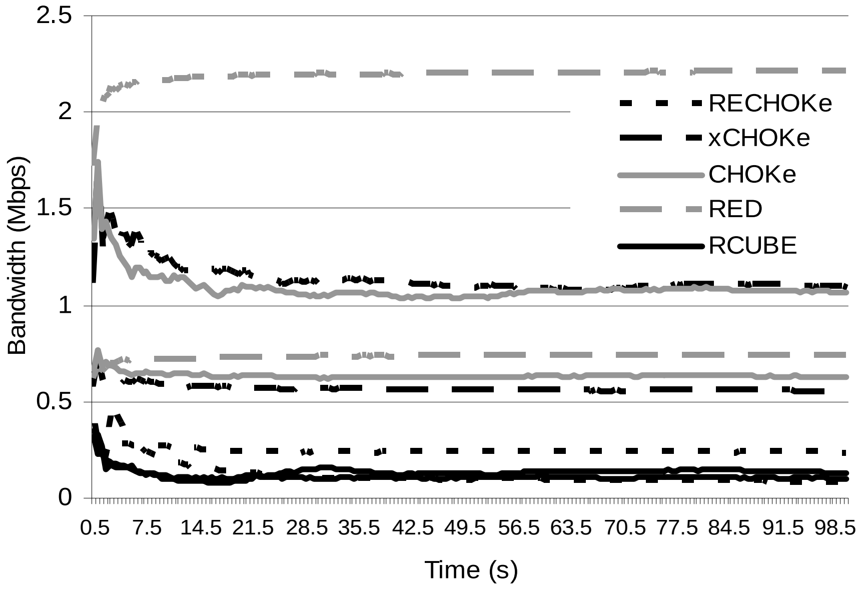

Figure 6.42 UDP flow shares with RCUBE, RECHOKe, xCHOKe, CHOKe, and RED

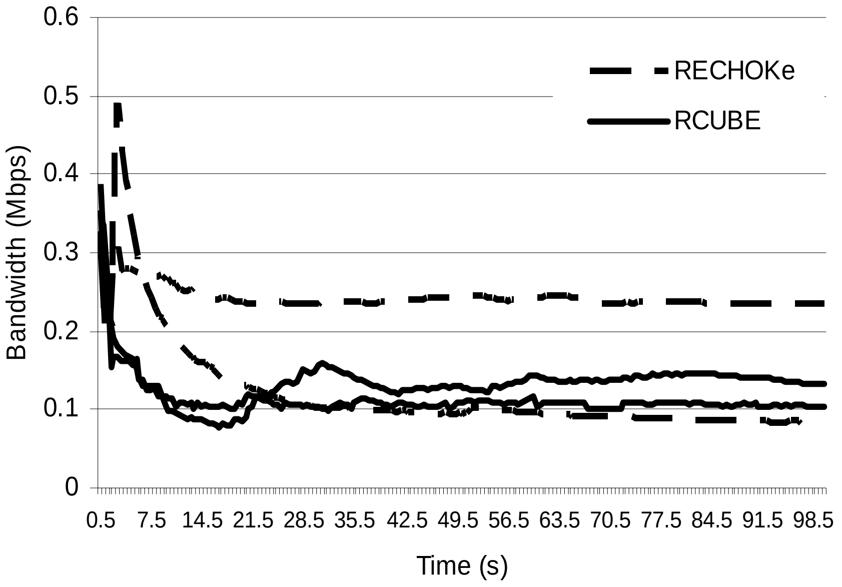

Figure 6.43 Link utilization of UDP flows with RECHOKe and RCUBE

CHAPTER 7

CONCLUSIONS

In this dissertation, we had proposed three major schemes to solve three main problems that present-day networks encounter. These problems are:

1) the TPC congestion detection, i.e., the timeout mechanism or the duration of the reception of three duplicate acknowledgements, due to early-dropped packets by queue management protocols in routers delays the response time of TCP in reducing the network congestion;

2) using ECN with active queue management schemes has its downsides: i) ECN messages may get delayed or dropped by downstream routers; and ii) TCP implementations at both the source and the destination have to be ECN-compliant;

3) active queue management schemes, with or without ECN, fail to protect TCP-friendly flows adequately in the presence of non TCP-friendly or malicious flows.

To solve the first two problems we have introduced a novel AQM modification scheme called Receiver-Window Modification (RWM). To solve the third problem a scheme called RECHOKe was introduced that can detect, control, and punish malicious flows, thereby protecting TCP-friendly flows. Finally, we have combined the RECHOKe and the RWM schemes with RED to produce a new AQM scheme called

RCUBE (Receiver Window Modified Random Early Detection queues with RECHOKe).

RWM does not require modification to all end system TCP/IP stacks but could be solely implemented in heavily-congested ingress and gateway routers. This means that RWM does not require both the sources and receivers to be "RWM-compliant" as was the case for ECN-compliant queues. Our RWM scheme helped to reduce the average queue sizes of the RED, ARED, BLUE, ARED-ECN, RED-ECN and BLUE-ECN queues. By reducing the average queue sizes, RWM queues reduced the queuing delay resulting in significant improvements in one-way end-to-end average packet delays, delay jitter, throughput and number of total dropped packets. We have also shown that the performance of RED-ECN, ARED-ECN and BLUE-ECN queues was heavily dependent on the queues of the downstream routers. RWM queues in ingress and gateway routers were not influenced by the number and state of the downstream router as they would piggyback congestion information to the source in the next available acknowledgement packet. We have shown simulation experiments validating the performance of RWM.

We have also presented a mathematical model for the RED-RWM queue by extending the work previously done for RED-ECN queues. We have verified our model by comparing Monte Carlo simulations of our model to discrete event simulations of an NS2 model. We claimed and proofed that an RWM modified RED queue weakly converges to a steady state and that as the number of clients (TCP connections) grows, the queue size weakly converges to the number of clients, assuming that the minimum marking threshold (*Th_{min}*) was less than the number of clients. We have also shown, using an asymptotic and steady-state analysis, that

1) the average TCP dynamics for a large number of flows using a RED-RWM queue is closely related to that of a single flow utilizing the same TCP congestion control mechanism and that

2) sessions become asymptotically independent as number of flows using the queue become large, suggesting that the RED-RWM queue alleviates the synchronization problem among flows.

We have also proposed a scheme called RECHOKe, a scheme for detecting, controlling and punishing malicious flows while protecting non-malicious flows. It worked by combining the techniques used by xCHOKe and RED-PD in identifying and punishing malicious flows. Hence, it used both the CHOKe hit and RED's drop/mark histories to detect and control more quickly and accurately when compared to RED, CHOKe, and xCHOKe.

We have presented RCUBE which combined the advantages of both RWM and RECHOKe schemes. We had analyzed xCHOKe, RECHOKe, and RCUBE in detail using NS2 and showed why RCUBE outperforms both xCHOKe and RECHOKe. We have also compared RCUBE with RED, either used alone or with RECHOKe, xCHOKe and CHOKe, and showed that RCUBE outperformed all previous schemes.

APPENDIX A

PROOF OF THEOREM 1

Proof of Theorem 1

The sequence $\{Y_i^{(N)}(t), Q_i^{(N)}(t), \hat{Q}_i^{(N)}(t)\}$ is a stochastic process with a finite number of states in which the probability of occurrence of a future state is conditional only upon current state (Markovian property). Considering an arbitrary bounded mapping $g : \mathrm{Z}_+^{13} \to \mathbb{R}$ and writing $g(Y_i^{(N)}(t+1))$ as function of $Y_i^{(N)}(t)$ through a case analysis, it can be seen that the above argument is correct. The states that $Y_i^{(N)}(t+1)$ can transition from [t,t+1) are as follows:

$$g(Y_i^{(N)}(t+1))$$

$$= 1[X_i^{(N)}(t) = 0]$$

$$\times g(0, 1[U_i(t+1) < P_{ar}]F_i(t+1), 0, 1, 0, 0, 1, 1, 1, 1, 1, 1, 1)$$

$$+1[X_i^{(N)}(t) > A_i^{(N)}(t) > 0, \beta_i^{(N)}(t) \geq D_i^{(N)}(t)]$$

$$\times g(W_i^{(N)}(t+1), X_i^{(N)}(t) - A_i^{(N)}(t), (1-\hat{M}_i^{(N)}(t))M_i^{(N)}(t), S_i^{(N)}(t)\hat{Z}_i^{(N)}(t+1)), D_i^{(N)}(t), 1,$$

$$W_i^{(N)}(t+1)\hat{M}_i^{(N)}(t+1) + \Omega_i^{(N)}(t)(1-\hat{M}_i^{(N)}(t+1)), m_i^{(N)}(t+1), 1, M_i^{(N)}(t), z_i^{(N)}(t+1), 1, Z_i^{(N)}(t))$$

$$+1[0 < X_i^{(N)}(t) \leq A_i^{(N)}(t), \beta_i^{(N)}(t) \geq D_i^{(N)}(t)]$$

$$\times g(W_i^{(N)}(t+1), X_i^{(N)}(t), (1-\hat{M}_i^{(N)}(t))M_i^{(N)}(t), S_i^{(N)}(t)\hat{Z}_i^{(N)}(t+1)), D_i^{(N)}(t), 1,$$

$$W_i^{(N)}(t+1)\hat{M}_i^{(N)}(t+1) + \Omega_i^{(N)}(t)(1-\hat{M}_i^{(N)}(t+1)), m_i^{(N)}(t+1), 1, M_i^{(N)}(t), z_i^{(N)}(t+1), 1, Z_i^{(N)}(t))$$

$$+1[X_i^{(N)}(t) > A_i^{(N)}(t) > 0, \beta_i^{(N)}(t) < D_i^{(N)}(t)]$$

$$\times g(W_i^{(N)}(t),\ X_i^{(N)}(t)-A_i^{(N)}(t),0,S_i^{(N)}(t)\hat{Z}_i^{(N)}(t+1)),\ D_i^{(N)}(t),\beta_i^{(N)}(t)+1,\Omega_i^{(N)}(t),m_i^{(N)}(t+1),$$

$$M_i^{(N)}(t)m_i^{(N)}(t+1),\hat{M}_i^{(N)}(t),z_i^{(N)}(t+1),Z_i^{(N)}(t)z_i^{(N)}(t+1),\hat{Z}_i^{(N)}(t))$$

$$+1[0<X_i^{(N)}(t)\leq A_i^{(N)}(t),\beta_i^{(N)}(t)<D_i^{(N)}(t)]$$

$$\times g(W_i^{(N)}(t),\ X_i^{(N)}(t),0,S_i^{(N)}(t)\hat{Z}_i^{(N)}(t+1)),\ D_i^{(N)}(t),\beta_i^{(N)}(t)+1,\Omega_i^{(N)}(t),m_i^{(N)}(t+1),$$

$$M_i^{(N)}(t)m_i^{(N)}(t+1),\hat{M}_i^{(N)}(t),z_i^{(N)}(t+1),Z_i^{(N)}(t)z_i^{(N)}(t+1),\hat{Z}_i^{(N)}(t))$$

A1

Where

$$g(W_i^{(N)}(t+1),\ X_i^{(N)}(t),(1-\hat{M}_i^{(N)}(t))M_i^{(N)}(t),S_i^{(N)}(t)\hat{Z}_i^{(N)}(t+1),\ D_i^{(N)}(t),1,$$

$$W_i^{(N)}(t+1)\hat{M}_i^{(N)}(t+1)+\Omega_i^{(N)}(t)(1-\hat{M}_i^{(N)}(t+1)),m_i^{(N)}(t+1),1,M_i^{(N)}(t),z_i^{(N)}(t+1),1,Z_i^{(N)}(t))$$

$$=S_i^{(N)}(t)(1-\hat{Z}_i^{(N)}(t))\hat{M}_i^{(N)}(t)(1-E_i^{(N)}(t))\,F_g^1(W_i^{(N)}(t),X_i^{(N)}(t),D_i^{(N)}(t),\beta_i^{(N)}(t),\Omega_i^{(N)}(t),m_i^{(N)}(t),M_i^{(N)}(t),$$

$$z_i^{(N)}(t),Z_i^{(N)}(t))$$

$$+(1-S_i^{(N)}(t))Z_i^{(N)}(t)\hat{M}_i^{(N)}(t)(1-E_i^{(N)}(t))\,F_g^2(W_i^{(N)}(t),X_i^{(N)}(t),D_i^{(N)}(t),\beta_i^{(N)}(t),\Omega_i^{(N)}(t),m_i^{(N)}(t),M_i^{(N)}(t),$$

$$z_i^{(N)}(t),Z_i^{(N)}(t))$$

$$+(1-Z_i^{(N)}(t))\hat{M}_i^{(N)}(t)(1-E_i^{(N)}(t))\,F_g^3(W_i^{(N)}(t),X_i^{(N)}(t),D_i^{(N)}(t),\beta_i^{(N)}(t),\Omega_i^{(N)}(t),m_i^{(N)}(t),M_i^{(N)}(t),$$

$$z_i^{(N)}(t),Z_i^{(N)}(t))$$

$$+E_i^{(N)}(t)F_g^4(W_i^{(N)}(t),X_i^{(N)}(t),D_i^{(N)}(t),\beta_i^{(N)}(t),\Omega_i^{(N)}(t),m_i^{(N)}(t),M_i^{(N)}(t),z_i^{(N)}(t),Z_i^{(N)}(t))$$

The mappings $F_g^1, F_g^2, F_g^3,$ and $F_g^4 : \mathrm{Z}_+^7$ are associated with g and defined by:

$$F_g^1(W_i^{(N)}(t), X_i^{(N)}(t), D_i^{(N)}(t), \beta_i^{(N)}(t), \Omega_i^{(N)}(t), m_i^{(N)}(t), M_i^{(N)}(t), z_i^{(N)}(t), Z_i^{(N)}(t))$$

$$= g(\min(\max(2W_i^{(N)}(t),1), W_{\max}), X_i^{(N)}(t), 0, 1, D_i^{(N)}(t), 1, \Omega_i^{(N)}(t), m_i^{(N)}(t), M_i^{(N)}(t),$$

$$1, z_i^{(N)}(t), Z_i^{(N)}(t), 0)$$

$$F_g^2(W_i^{(N)}(t), X_i^{(N)}(t), D_i^{(N)}(t), \beta_i^{(N)}(t), \Omega_i^{(N)}(t), m_i^{(N)}(t), M_i^{(N)}(t), z_i^{(N)}(t), Z_i^{(N)}(t))$$

$$= g(\min(W_i^{(N)}(t)+1, W_{\max}), X_i^{(N)}(t), 0, 0, D_i^{(N)}(t), 1, \Omega_i^{(N)}(t), m_i^{(N)}(t), M_i^{(N)}(t),$$

$$1, z_i^{(N)}(t), Z_i^{(N)}(t), 0)$$

$$F_g^3(W_i^{(N)}(t), X_i^{(N)}(t), D_i^{(N)}(t), \beta_i^{(N)}(t), \Omega_i^{(N)}(t), m_i^{(N)}(t), M_i^{(N)}(t), z_i^{(N)}(t), Z_i^{(N)}(t))$$

$$= g(\left\lceil \frac{X_i^{(N)}(t)}{2} \right\rceil, X_i^{(N)}(t), 0, S_i^{(N)}(t), D_i^{(N)}(t), 1, \Omega_i^{(N)}(t), m_i^{(N)}(t), M_i^{(N)}(t), 1, z_i^{(N)}(t), Z_i^{(N)}(t), 0)$$

$$F_g^4(W_i^{(N)}(t), X_i^{(N)}(t), D_i^{(N)}(t), \beta_i^{(N)}(t), \Omega_i^{(N)}(t), m_i^{(N)}(t), M_i^{(N)}(t), z_i^{(N)}(t), Z_i^{(N)}(t))$$

$$= g(\Omega_i^{(N)}(t), X_i^{(N)}(t), 1, S_i^{(N)}(t), D_i^{(N)}(t), 1, \Omega_i^{(N)}(t), m_i^{(N)}(t), M_i^{(N)}(t), \hat{M}_i^{(N)}(t), z_i^{(N)}(t),$$

$$Z_i^{(N)}(t), \hat{Z}_i^{(N)}(t))$$

We introduce the following terminology to facilitate the proof of the theorem: For each t = 0, 1, … the statements **[A:t]**, **[B:t]**, **[C:t]** and **[D:t]** refer to the following convergence statements:

[A:t] There exist some non-random constants $q(t)$ and $\hat{q}(t)$ such that

$\frac{Q^{(N)}(t)}{N} \xrightarrow{P}_N q(t)$ and $\frac{\hat{Q}^{(N)}(t)}{N} \xrightarrow{P}_N \hat{q}(t)$,

[B:t] For some {0, 1, …, W$_{\max}$} g-valued rv $W(t)$, non-negative integer-valued rvs $X(t), E(t), D(t), \beta(t), \Omega(t)$ and $\{0,1\}$-valued rvs $S(t), m(t), M(t), \hat{M}(t), z(t), Z(t), \hat{Z}(t)$, it holds that

$$Y_i^{(N)}(t) \Rightarrow_N Y(t) = (W(t), X(t), E(t), S(t), D(t), \beta(t), \Omega(t), m(t), M(t), \hat{M}(t), z(t), Z(t), \hat{Z}(t))$$

[C:t] For any integer i = 1, 2,… , the random vector $\{ Y_i^{(N)}(t) \}$ becomes asymptotically independent as N becomes large, with

$$\lim_{N\to\infty} \mathrm{P}[Y_i^{(N)}(t) = y_i, i = 1, \ldots, I] = \prod_{i=1}^{I} \mathrm{P}[Y(t) = y_i]$$

where $Y_i^{(N)}(t)$ is given as in **[B:t].**

[D:t] For any bounded function $g : \mathrm{Z}_+^{13} \to \Re$, the convergence

$$\frac{1}{N}\sum_{t=1}^{N} g(Y_i^{(N)}(t)) \xrightarrow{P}_N \mathrm{E}[g(Y(t))].$$

holds with $Y(t)$ in **[B:t].**

[E:t] For any bounded mapping $g : \mathrm{Y} \to \Re$, there exists a bounded and continuous mapping $F_g : [0,1] \times [0,1] \times \mathrm{Y} \to \Re$ such that

$$E[g\left(Y_i^{(N)}(t+1)\right) \mid \Omega_t, \tau_t] = F_g\left(\Psi_i^{(N)}(t), \chi_i^{(N)}(t), Y_i^{(N)}(t)\right) \qquad \text{A2}$$

This assumption states that given the events leading up to the beginning of timeslot [t, t + 1), the expected behavior of session i leading to the beginning of timeslot [t + 1, t + 2) can also be determined using the expected knowledge of the conditional receiver window modifying probability, $\Psi_i^{(N)}(t)$, the conditional dropping probability $\chi_i^{(N)}(t)$ and $Y_i^{(N)}(t)$ at [t, t + 1). Hence, it also implies that

$$E[g(Y_i^{(N)}(t+1))] = E[F_g(\Psi_i^{(N)}(t), \chi_i^{(N)}(t), Y_i^{(N)}(t))] \qquad \text{A3}$$

This leads to

$$E[g(Y_i^{(N)}(t+1))] \Rightarrow_N E[F_g(\Psi(t), \chi(t), Y(t))] \qquad \text{A4}$$

implying that for large N, given the events leading up to the beginning of timeslot [t, t+1), the expected behavior of session i leading to the beginning of timeslot [t + 1, t + 2) can also be determined using the expected value of the conditional receiver window modifying probability, $\Psi(t)$, the conditional dropping probability $\chi(t)$ and $Y(t)$.

Lemma 1: If **[A:t], [B:t]** and **[C:t]** hold for some t = 0, 1, …, then **[D:t]** holds.

Proof: From **[B:t]** and Proposition 4.7, Pg. 140 [59], if $Y_i^{(N)}(t) \to Y(t)$ and $g: Z_+^{13} \to \Re$ is a continuous function, then $g(Y_i^{(N)}(t)) \xrightarrow{P}_N g(Y(t))$. It immediately follows that

$$\frac{1}{N}\sum_{t=1}^{N} g(Y_i^{(N)}(t)) \xrightarrow{P}_N \mathrm{E}[g(Y(t))].$$

Hence, **[D:t]** holds.

Lemma 2: If **[A:t]** and **[B:t]** hold for some t = 0, 1, …, then **[E:t]** holds.

The random vector $Y_1^{(N)}(t)$ for a session *i* in timeslot [*t, t* + 1) takes values

$W_i^{(N)}(t), X_i^{(N)}(t), E_i^{(N)}(t), S_i^{(N)}(t), D_i^{(N)}(t), \beta_i^{(N)}(t), \Omega_i^{(N)}(t), m_i^{(N)}(t), \hat{M}_i^{(N)}(t), M_i^{(N)}(t), z_i^{(N)}(t),$

$\hat{Z}_i^{(N)}(t)$ and $Z_i^{(N)}(t)$ in a discrete space *Y*. Hence, the rv $A_i^{(N)}(t)$ can be determined as a function of $Y_i^{(N)}(t)$, i.e., $A_i^{(N)}(t) = \Phi(Y_i^{(N)}(t))$, for some bounded continuous function $\Phi : \mathrm{Y} \rightarrow \Re$.

Letting $\tau_t = \sigma_1\{\, Q^{(N)}(0), Y_i^{(N)}(0), V_{i,j}(s), 1 \le s \le t; i, j = 1, 2, \ldots\}$, we define

$$\Psi_i^{(N)}(t) := \mathrm{E}\left[m_{i,j}^{(N)}(t+1) \mid \tau_t \right]$$

$$= \left(1 - f_m^{(N)}\left(\hat{Q}^{(N)}(t)\right)\right)^{A_i^{(N)}(t)}$$

to be the conditional probability that, given τ_t, connection *i* will receive receiver window modification acknowledgements from the RED-RWM gateway during the period [t, t + 1). Under the enforced independence assumptions, it is clear that

$$E[\textstyle\prod_{j=1}^{A(t)} m_{i,j}^{(N)}(t+1) \mid \tau_t] = \Psi_i^{(N)}(t)$$

$$\textstyle\prod_{j=1}^{A(t)} m_{i,j}^{(N)}(t+1) =_{st} 1[V_{i,j}(t+1) \le \Psi_i^{(N)}(t)]$$

Letting $\Omega_t = \sigma_2\{\, Q^{(N)}(0), Y_i^{(N)}(0), L_{i,j}(s), 1 \le s \le t; i, j = 1, 2, \ldots\}$, we also define

$$\chi_i^{(N)}(t) := \mathrm{E}\left[z_{i,j}^{(N)}(t+1) \mid \Omega_t \right]$$

$$= \left(1 - f_d^{(N)}\left(\hat{Q}^{(N)}(t)\right)\right)^{A_i^{(N)}(t)}$$

to be the conditional probability that, given Ω_t, connection *i* will experience packet drops by the RED-RWM gateway during the period [t, t + 1). Under the enforced independence assumptions, it is clear that

$$E[\prod_{j=1}^{A(t)} z_{i,j}^{(N)}(t+1)] = \chi_i^{(N)}(t)$$

$$\prod_{j=1}^{A(t)} z_{i,j}^{(N)}(t+1) =_{st} 1[V_{i,j}(t+1) \le \chi_i^{(N)}(t)]$$

Finally, it follows from (R3)

$$\begin{aligned}
&E[g(Y_i^{(N)}(t+1)) \mid \Omega_t, \tau_t] \\
&= 1[X_i^{(N)}(t) = 0] \\
&\times E[\, g(0, 1[U_i(t+1) < P_{ar}] F_i(t+1), 0, 1, 0, 0, 0, 1, 1, 1, 1, 1, 1)] \\
&+ 1[X_i^{(N)}(t) > A_i^{(N)}(t) > 0, \beta_i^{(N)}(t) \ge D_i^{(N)}(t)] \\
&\times[\Psi_i^{(N)}(t)\, \chi_i^{(N)}(t) g(W_i^{(N)}(t+1), X_i^{(N)}(t) - A_i^{(N)}(t), (1-\hat{M}_i^{(N)}(t)) M_i^{(N)}(t), S_i^{(N)}(t) Z_i^{(N)}(t), \\
&(Q^{(N)}(t) + A^{(N)}(t))/(N*C), 1, W_i^{(N)}(t+1) M_i^{(N)}(t) + \Omega_i^{(N)}(t)(1 - M_i^{(N)}(t)), 1, 1, M_i^{(N)}(t), 1, 1, Z_i^{(N)}(t)) \\
&+ (1-\Psi_i^{(N)}(t))(1-\chi_i^{(N)}(t)) g(W_i^{(N)}(t+1), X_i^{(N)}(t) - A_i^{(N)}(t), (1-\hat{M}_i^{(N)}(t)) M_i^{(N)}(t), S_i^{(N)}(t) Z_i^{(N)}(t), \\
&(Q^{(N)}(t) + A^{(N)}(t))/(N*C), 1, W_i^{(N)}(t+1) M_i^{(N)}(t) + \Omega_i^{(N)}(t)(1 - M_i^{(N)}(t)), 0, 1, M_i^{(N)}(t), 0, 1, Z_i^{(N)}(t)) \\
&+ \Psi_i^{(N)}(t)(1-\chi_i^{(N)}(t)) g(W_i^{(N)}(t+1), X_i^{(N)}(t) - A_i^{(N)}(t), (1-\hat{M}_i^{(N)}(t)) M_i^{(N)}(t), S_i^{(N)}(t) Z_i^{(N)}(t), \\
&(Q^{(N)}(t) + A^{(N)}(t))/(N*C), 1, W_i^{(N)}(t+1) M_i^{(N)}(t) + \Omega_i^{(N)}(t)(1 - M_i^{(N)}(t)), 1, 1, M_i^{(N)}(t), 0, 1, Z_i^{(N)}(t)) \\
&+ (1-\Psi_i^{(N)}(t)) \chi_i^{(N)}(t) g(W_i^{(N)}(t+1), X_i^{(N)}(t) - A_i^{(N)}(t), (1-\hat{M}_i^{(N)}(t)) M_i^{(N)}(t), S_i^{(N)}(t) Z_i^{(N)}(t), \\
&(Q^{(N)}(t) + A^{(N)}(t))/(N*C), 1, W_i^{(N)}(t+1) M_i^{(N)}(t) + \Omega_i^{(N)}(t)(1 - M_i^{(N)}(t)), 0, 1, M_i^{(N)}(t), 1, 1, Z_i^{(N)}(t))] \\
&+ 1[0 < X_i^{(N)}(t) \le A_i^{(N)}(t), \beta_i^{(N)}(t) \ge D_i^{(N)}(t)]
\end{aligned} \qquad \text{A5}$$

$\times[\Psi_i^{(N)}(t)\,\chi_i^{(N)}(t)g(W_i^{(N)}(t+1),\,X_i^{(N)}(t),(1-\hat{M}_i^{(N)}(t))M_i^{(N)}(t),S_i^{(N)}(t)Z_i^{(N)}(t),$

$(Q^{(N)}(t)+A^{(N)}(t))\,/(N*C),1,W_i^{(N)}(t+1)M_i^{(N)}(t)+\Omega_i^{(N)}(t)(1-M_i^{(N)}(t)),1,1,M_i^{(N)}(t),1,1,Z_i^{(N)}(t))$

$+(1-\Psi_i^{(N)}(t))\,(1-\chi_i^{(N)}(t))g(W_i^{(N)}(t+1),\,X_i^{(N)}(t),(1-\hat{M}_i^{(N)}(t))M_i^{(N)}(t),S_i^{(N)}(t)Z_i^{(N)}(t),$

$(Q^{(N)}(t)+A^{(N)}(t))\,/(N*C),1,W_i^{(N)}(t+1)M_i^{(N)}(t)+\Omega_i^{(N)}(t)(1-M_i^{(N)}(t)),0,1,M_i^{(N)}(t),0,1,Z_i^{(N)}(t))$

$+\Psi_i^{(N)}(t)(1-\chi_i^{(N)}(t))g(W_i^{(N)}(t+1),\,X_i^{(N)}(t),(1-\hat{M}_i^{(N)}(t))M_i^{(N)}(t),S_i^{(N)}(t)Z_i^{(N)}(t),$

$(Q^{(N)}(t)+A^{(N)}(t))\,/(N*C),1,W_i^{(N)}(t+1)M_i^{(N)}(t)+\Omega_i^{(N)}(t)(1-M_i^{(N)}(t)),1,1,M_i^{(N)}(t),0,1,Z_i^{(N)}(t))$

$+(1-\Psi_i^{(N)}(t))\,\chi_i^{(N)}(t)g(W_i^{(N)}(t+1),\,X_i^{(N)}(t),(1-\hat{M}_i^{(N)}(t))M_i^{(N)}(t),S_i^{(N)}(t)Z_i^{(N)}(t),$

$(Q^{(N)}(t)+A^{(N)}(t))\,/(N*C),1,W_i^{(N)}(t+1)M_i^{(N)}(t)+\Omega_i^{(N)}(t)(1-M_i^{(N)}(t)),0,1,M_i^{(N)}(t),1,1,Z_i^{(N)}(t))]$

$+1[X_i^{(N)}(t)>A_i^{(N)}(t)>0,\beta_i^{(N)}(t)<D_i^{(N)}(t)]$

$\times[\Psi_i^{(N)}(t)\,\chi_i^{(N)}(t)g(W_i^{(N)}(t),\,X_i^{(N)}(t)-A_i^{(N)}(t),0,S_i^{(N)}(t)\hat{Z}_i^{(N)}(t),\,D_i^{(N)}(t),\beta_i^{(N)}(t)+1,\Omega_i^{(N)}(t),$

$1,M_i^{(N)}(t),\hat{M}_i^{(N)}(t),1,Z_i^{(N)}(t),\hat{Z}_i^{(N)}(t))$

$+(1-\Psi_i^{(N)}(t))\,(1-\chi_i^{(N)}(t))g(W_i^{(N)}(t),\,X_i^{(N)}(t)-A_i^{(N)}(t),0,S_i^{(N)}(t)\hat{Z}_i^{(N)}(t),\,D_i^{(N)}(t),\beta_i^{(N)}(t)+1,$

$\Omega_i^{(N)}(t),0,0,\hat{M}_i^{(N)}(t),0,0,\hat{Z}_i^{(N)}(t))$

$+\Psi_i^{(N)}(t)(1-\chi_i^{(N)}(t))g(W_i^{(N)}(t),\,X_i^{(N)}(t)-A_i^{(N)}(t),0,S_i^{(N)}(t)\hat{Z}_i^{(N)}(t),\,D_i^{(N)}(t),\beta_i^{(N)}(t)+1,\Omega_i^{(N)}(t),1,$

$M_i^{(N)}(t),\hat{M}_i^{(N)}(t),0,0,\hat{Z}_i^{(N)}(t))$

$$+(1-\Psi_i^{(N)}(t))\chi_i^{(N)}(t)g(W_i^{(N)}(t), X_i^{(N)}(t)-A_i^{(N)}(t),0,S_i^{(N)}(t)\hat{Z}_i^{(N)}(t), D_i^{(N)}(t),\beta_i^{(N)}(t)+1,\Omega_i^{(N)}(t),0,$$

$$0,\hat{M}_i^{(N)}(t),1,Z_i^{(N)}(t),\hat{Z}_i^{(N)}(t))$$

$$+1[0<X_i^{(N)}(t)\le A_i^{(N)}(t),\beta_i^{(N)}(t)<D_i^{(N)}(t)]$$

$$\times[\Psi_i^{(N)}(t)\,\chi_i^{(N)}(t)g(W_i^{(N)}(t), X_i^{(N)}(t),0,S_i^{(N)}(t)\hat{Z}_i^{(N)}(t), D_i^{(N)}(t),\beta_i^{(N)}(t)+1,\Omega_i^{(N)}(t),$$

$$1,M_i^{(N)}(t),\hat{M}_i^{(N)}(t),1,Z_i^{(N)}(t),\hat{Z}_i^{(N)}(t))$$

$$+(1-\Psi_i^{(N)}(t))\,(1-\chi_i^{(N)}(t))g(W_i^{(N)}(t), X_i^{(N)}(t),0,S_i^{(N)}(t)\hat{Z}_i^{(N)}(t), D_i^{(N)}(t),\beta_i^{(N)}(t)+1,$$

$$\Omega_i^{(N)}(t),0,0,\hat{M}_i^{(N)}(t),0,0,\hat{Z}_i^{(N)}(t))$$

$$+\Psi_i^{(N)}(t)(1-\chi_i^{(N)}(t))g(W_i^{(N)}(t), X_i^{(N)}(t),0,S_i^{(N)}(t)\hat{Z}_i^{(N)}(t), D_i^{(N)}(t),\beta_i^{(N)}(t)+1,\Omega_i^{(N)}(t),1,$$

$$M_i^{(N)}(t),\hat{M}_i^{(N)}(t),0,0,\hat{Z}_i^{(N)}(t))$$

$$+(1-\Psi_i^{(N)}(t))\chi_i^{(N)}(t)g(W_i^{(N)}(t), X_i^{(N)}(t),0,S_i^{(N)}(t)\hat{Z}_i^{(N)}(t), D_i^{(N)}(t),\beta_i^{(N)}(t)+1,\Omega_i^{(N)}(t),0,$$

$$0,\hat{M}_i^{(N)}(t),1,Z_i^{(N)}(t),\hat{Z}_i^{(N)}(t))$$

$$=F_g(\Psi_i^{(N)}(t),\chi_i^{(N)}(t),W_i^{(N)}(t),X_i^{(N)}(t),E_i^{(N)}(t),S_i^{(N)}(t),D_i^{(N)}(t),\beta_i^{(N)}(t),\Omega_i^{(N)}(t),m_i^{(N)}(t),$$

$$\hat{M}_i^{(N)}(t),M_i^{(N)}(t),z_i^{(N)}(t),\hat{Z}_i^{(N)}(t),Z_i^{(N)}(t)) \qquad \text{A6}$$

where the mapping is:

$$F_g:[0,1]\times[0,1]\times\{0,1,\ldots,W_{\max}\}\times\{0,1,\ldots,X_{\max}\}\times\{0,1\}\times\{0,1\}\times\{2,3,\ldots,D_{\max}\}\times$$

$$\{0,1,\ldots,D_{\max}\}\times\{0,1,\ldots,X_{\max}\}\times\{0,1\}\times\{0,1\}\times\{0,1\}\times\{0,1\}\times\{0,1\}\times\{0,1\}.$$

By using equation 4.16 to substitute $Y_i^{(N)}(t)$ in equation A6, we get

$$E[g(Y_i^{(N)}(t+1)) \mid \Omega_t, \tau_t] = F_g(\Psi_i^{(N)}(t), \chi_i^{(N)}(t), Y_i^{(N)}(t)) \qquad \text{A7}$$

Taking expectation on both sides, we get

$$E[g(Y_i^{(N)}(t+1))] = E[F_g(\Psi_i^{(N)}(t), \chi_i^{(N)}(t), Y_i^{(N)}(t))]$$

Since F_g is continuous on $[0,1] \times [0,1] \times y$ and using Proposition 4.7, Pg. 140 [59],

$$F_g(\Psi_i^{(N)}(t), \chi_i^{(N)}(t), Y_i^{(N)}(t)) \Rightarrow_N F_g(\Psi(t), \chi(t), Y(t))$$

Hence

$$E[g\left(Y_i^{(N)}(t+1)\right)] = E\left[F_g(\Psi(t), \chi(t), Y(t))\right] \qquad \text{A8}$$

Thus **[E:t]** holds.

Lemma 3: If **[A:t], [B:t]** and **[C:t]** hold for some t = 0, 1, …, then **[C:t+1]** holds.

Proof:

From **[C:t],** we have

$$(\Psi_i^{(N)}(t), \chi_i^{(N)}(t), Y_i^{(N)}(t)) \Rightarrow_N (\Psi(t), \chi(t), Y(t))$$

Since $Y_i^{(N)}(t+1)$ is independent of τ_t, Ω_t.

We can see that

$$E\left[\prod_{i=1}^{I} g_i\left(Y_i^{(N)}(t+1)\right) \mid \tau_t, \Omega_t\right] = \prod_{i=1}^{I} E\left[g_i\left(Y_i^{(N)}(t+1)\right) \mid \tau_t, \Omega_t\right]$$

$$= E\left[\prod_{i=1}^{I} F_{g_i}(\Psi_i^{(N)}(t), \chi_i^{(N)}(t), Y_i^{(N)}(t))\right]$$

by using [27].

By using the bounded convergence theorem:

$$\lim_{N\to\infty} E\left[\prod_{i=1}^{I} g_i\left(Y_i^{(N)}(t+1)\right)\right] = \lim_{N\to\infty} E\left[\prod_{i=1}^{I} F_{g_i}(\Psi_i^{(N)}(t), \chi_i^{(N)}(t), Y_i^{(N)}(t))\right]$$

$$= E\left[\prod_{i=1}^{I} F_{g_i}(\Psi_i(t), \chi_i(t), Y_i(t))\right]$$

$$= \prod_{i=1}^{I} E\left[F_{g_i}(\Psi_i(t), \chi_i(t), Y_i(t))\right]$$

$$= \prod_{i=1}^{I} E\left[g_i\left(Y_i(t+1)\right)\right] \qquad \text{A9}$$

From equation A9 we can see that

$$\lim_{N\to\infty} \mathrm{P}[Y_i^{(N)}(t+1) = y_i, i = 1,\ldots,I] = \prod_{i=1}^{I} \mathrm{P}[Y(t+1) = y_i]$$

Lemma 4: If **[A:t]** and **[D:t]** hold for some t = 0, 1, …, then **[A:t+1]** holds.

Proof: From **[D:t],** we can conclude that

$$\frac{1}{N}\sum_{t=1}^{N} A_i^{(N)}(t) \xrightarrow{P}_N \mathrm{E}[A(t)],$$

and from **[A:t],**

$$\frac{Q^{(t)}(t)}{N} - C + \frac{1}{N}\sum_{t=1}^{N} A_i^{(N)}(t) \xrightarrow{P}_N q(t) - C + E[A(t)]$$

Since

$$\frac{Q^{(t)}(t+1)}{N} = \frac{Q^{(t)}(t)}{N} - C + \frac{1}{N}\sum_{t=1}^{N} A_i^{(N)}(t)$$

$$\xrightarrow{P}_N q(t) - C + E[A(t)]$$

$$:= q(t+1)\,.$$

Since from **[A:t]**,

$$\frac{Q^{(N)}(t)}{N} \xrightarrow{P}_N q(t),$$

$$\frac{\hat{Q}^{(N)}(t)}{N} \xrightarrow{P}_N \hat{q}(t),$$

and

$$\hat{Q}^{(N)}(t+1) = (1-\alpha)\hat{Q}^{(N)}(t) + \alpha Q^{(N)}(t+1)\,,$$

we get

$$\frac{\hat{Q}^{(N)}(t+1)}{N} = (1-\alpha)\frac{\hat{Q}^{(N)}(t)}{N} + \alpha\frac{Q^{(N)}(t+1)}{N}$$

Since from above,

$$\frac{Q^{(t)}(t+1)}{N} \xrightarrow{P}_N q(t+1)$$

we get

$$\frac{\hat{Q}^{(N)}(t+1)}{N} \xrightarrow{P}_N (1-\alpha)\hat{q}(t) + \alpha q(t+1)$$

$$\xrightarrow{P}_N \hat{q}(t+1)$$

Lemma 5: If **[A:t]** and **[B:t]** hold for some t = 0, 1, …, then **[B:t+1]** holds with Y(t+1) related in distribution to Y(t)

Proof:

From equation A8, we have

$$E[g\left(Y_i^{(N)}(t+1)\right)] = E\left[F_g(\Psi(t), \chi(t), Y(t))\right].$$

Since g is a bounded mapping function, it follows that

$$Y_i^{(N)}(t+1) \Rightarrow_N Y(t+1) = (W(t+1), X(t+1), E(t+1), S(t+1), D(t+1), \beta(t+1), \Omega(t+1),$$
$$m(t+1), M(t+1), \hat{M}(t+1), z(t+1), Z(t+1), \hat{Z}(t+1))$$

APPENDIX B

PROOF OF THEOREM 2

Proof of Theorem 2

From Theorem 1 (equation [7]) and assuming the following:

i. Average queue size ($\hat{Q}^{(N)}(t)$) is currently N-1, where *th*$_{min}$= N-1. (This ensures that the next packet will trigger the RWM scheme.)

ii. The RWM queue modifies the receiver window in the acknowledgements to 1 MSS for every packet arrival after the *th*$_{min}$ threshold.

we get

$$q^* = N - 1 + (N-1)w_q(1-w_q) \qquad \text{B1}$$

Simplifying this further, as N is varied from 1 to 500 and setting w_q = 0.002, we get

$$(N-1)w_q(1-w_q) \rightarrow 1.$$

Hence, in equation B1,

$$q^* \Rightarrow_N N. \qquad \text{B2}$$

If, for example, the first packet, which enters after q^* has reached the *Th*$_{min}$, is from flow *i*, then the RWM queue will modify flow *i*'s acknowledgement to one MSS. This implies that in the next RTT, flow *i* will transmit only a single packet. Since there are *N* numbers of connections, at the steady state, all *N* connections will transmit a single packet. Since the number of packets in the queue will be approximately *N* (and hence greater than the *Th*$_{min}$) the RWM scheme will be triggered continuously until congestion eases.

Next, we relax the assumption *ii* to figure out the number of packets beyond the Th_{min} that can be queued without triggering the RWM scheme that will lead to q^* to just exceed N+1. Let M represent the number of packets. Table B.1 shows the results of the increase in the instantaneous queue length to render such a situation.

Table B.1 Results of the Increase in the Instantaneous Queue Length

Number of Connections	M
10	90
50	195
100	257
200	328

Since the equations 5.8, 5.13 and 5.14 (and thus the RWM scheme) is triggered often whenever the average queue is between Th_{min} and Th_{max}; well before these M values have been reached, equation B2 holds true.

REFERENCES

[1] S. Floyd, “Measurement studies of end-to-end congestion control in the internet”, 2002. http://www.icir.org/floyd/ccmeasure.html.

[2] S Blake, D Black, M Carlson, E Davies, and Z Wang, “An architecture for differentiated services”, RFC 2475, December 1998.

[3] R. Braden, D. Clark, and S. Shenker, "Integrated services in the internet architecture: an overview", RFC 1633, July 1994.

[4] V. Firoiu, and M. Borden, “A study of active queue management for congestion control”, Proceedings of Infocom 2000, pp. 1435 - 1444, vol. 3, March, 2000.

[5] S. Floyd, "TCP and Explicit Congestion Notification", ACM Computer Communication Review, pp. 10-23, vol. 24, October, 1994.

[6] K. Ramakrishnan, S. Floyd, and D. Black, “The addition of Explicit Congestion Notification (ECN) to IP”, RFC 3168, September, 2001.

[7] S. Floyd, and V. Jacobson, “Random early detection gateways for congestion avoidance”, IEEE/ACM Transactions on Networking, pp. 397-413, vol. 1, no. 4, August, 1993.

[8] V. Jacobson, “Congestion Avoidance and Control”, Proceedings of ACM Sigcomm 1988, pp. 314-329, August, 1988.

[9] W. Stevens, "TCP/IP Illustrated, Volume 1: The Protocols”, Addison-Wesley, 1994.

[10] L. Zhang, and D. Clark, “Oscillating Behavior of Network Traffic: A Case Study Simulation”, Internetworking: Research and Experience, pp. 101-112, vol. 1, June, 1990.

[11] A. Veres, and M. Boda, "The chaotic nature of TCP congestion control", Proceedings of Infocom 2000, pp. 1715 - 1723, vol.3, March, 2000.

[12] R. Huang, and G.V. Záruba, “Effect of ingress buffering on self-similarity of optical burst traffic”, Proceedings of OptiComm 2003, October, 2003.

[13] M. Kwon, and S. Fahmy, "A comparison of load-based and queue-based active queue management algorithms", Proceedings of the International Society for Optical Engineering, pp. 35-46, vol. 4866, July/August, 2002.

[14] S. Athuraliya, S.H. Low, V.H. Li and Y. Qinghe, “REM: active queue management”, IEEE Network, pp. 48-53, vol. 15, issue 3, May, 2001.

[15] S. Kunniyur, and R. Srikant, “Analysis and design of an Adaptive Virtual Queue (AVQ) Algorithm for active queue management”, Proceedings of the 2001 conference on applications, technologies, architectures, and protocols for computer communications, pp. 123–134, August, 2001.

[16] C.V. Hollot, V. Misra, D. Towsley and W.B. Gong, “On Designing Improved Controllers for AQM Routers Supporting TCP Flows”, Proceedings of Infocom 2001, pp. 1726-1734, vol. 3, April, 2001.

[17] T.J. Ott, T.V. Lakshman, and L.H. Wong, “SRED: Stabilized RED”, Proceedings of the IEEE Infocom 1999, pp. 1346-1355, vol. 3, March, 1999.

[18] D. Lin, and R. Morris, "Dynamics of random early detection", Proceedings of Sigcomm 1997, pp. 127–136, vol. 27, September, 1997.

[19] S. Floyd, R. Gummadi, and S. Shenker. "Adaptive RED: an algorithm for increasing the robustness of RED's Active Queue Management", 2001, http://www.icir.org/floyd.

[20] W. Feng, D. Kandlur, D. Saha, and K. Shin, "The BLUE active queue management algorithms", IEEE/ACM Transactions on Networking (TON) Volume 10, Issue 4 (August 2002) Pages: 513 – 528, 2002.

[21] M. May, J. Bolot, C. Diot, and B. Lyles, "Reasons not to deploy RED", Proceedings of 7th International Workshop on Quality of Service (IWQoS'99), pp. 260–262, June, 1999.

[22] M. May, T. Bonald, and J. Bolot, "Analytic evaluation of RED performance", Proceedings of Infocom 2000, pp. 1415 - 1424, vol. 3, March, 2000.

[23] G. Hasegawa, and M. Murata, "Analysis of dynamic behaviors of many TCP connections sharing Tail–Drop/RED routers", Proceedings of IEEE Globecom, November 2001.

[24] C. Hollot, V. Misra, D. Towsley, and W. Gong, "A control theoretic analysis of RED", Proceedings of IEEE Infocom, 2001.

[25] P. Tinnakornsrisuphap, and R. La, "Limiting Model of ECN/RED under a Large Number of Heterogeneous TCP Flows", Technical Report, Institute for Systems Research, University of Maryland, 2003; Proceedings of IEEE

Conference on Decision and Control (CDC) 2003, Maui, Hawaii, December 2003.

[26] P. Tinnakornsrisuphap, and A. Makowski, “Queue dynamics of RED gateways under large number of TCP flows”, Proceedings of IEEE Globecom, 2001.

[27] P. Tinnakornsrisuphap, and R. La, “Asymptotic behavior of heterogeneous TCP flows and RED gateways”, Technical report, Institute for Systems Research, University of Maryland, 2003.

[28] P. Tinnakornsrisuphap, R. La, and A. M. Makowski, “Modeling TCP traffic with session dynamics - Many sources asymptotics under ECN/RED gateways”, Proceedings of the 18th International Teletraffic Congress, Berlin, Germany, August 2003.

[29] P. Tinnakornsrisuphap, and A. M. Makowski, “Limit behavior of ECN/RED gateways under a large number of TCP flows”, Proceedings of Infocom 2003, San Francisco (CA), April 2003.

[30] P. Tinnakornsrisuphap, and A. M. Makowski, “On the behavior of ECN/RED gateways under a large number of TCP flows: limit theorems”, Submitted to QUESTA.

[31] A. Makowski, and P. Tinnakornsrisuphap, “Many flow asymptotics for TCP with ECN/RED”, IEEE Information Theory Workshop 2002, Bangalore, India, October 2002.

[32] NS2 simulator. http://www.isi.edu/nsnam/ns/.

[33] R. Pan, B. Prabhakar, and K. Psounis, “CHOKe: a stateless active queue management scheme for approximating fair bandwidth allocation”, Proceedings of IEEE Infocom, volume 2, pp. 942–951, Tel Aviv, Israel, March, 2000.

[34] A. Aditya, K. Richard, P. Christos, S. Srinivasan, and S. Scott, “Selfish Behavior and Stability of the Internet: A Game-Theoretic Analysis of TCP”, ACM Sigcomm 2002, Pittsburgh PA.

[35] P. Chhabra, A. John, H. Saran, and R. Shorey. "Controlling Malicious Sources at Internet Gateways", IEEE International Conference in Communication - Anchorage, Alaska, May 2003.

[36] C. Parminder, C. Shobhit, G. Anurag, J. Ajita, K. Abhishek, S. Huzur, and S. Rajeev, “xCHOKe: Malicious source control for congestion avoidance at Internet Gateways”, Proceedings of IEEE International Conference on Network Protocols (ICNP-02), Paris, Nov 2002.

[37] R. Mahajan, S. Floyd, and D. Wetherall, “Controlling high-bandwidth flows at the congested router”, 9th International Conference on Network Protocols (ICNP), November 2001.

[38] T. Wang, I. Matta, and A. Bestavros, “Efficiently and fairly allocating bandwidth at a highly congested link”, Technical Report, Boston University, December 2, 2003.

[39] Y. Jiang, M. Hamdi, and J. Liu, “Self Adjustable CHOKe: An active queue management algorithm for congestion control and fair bandwidth allocation”, 8th IEEE Symposium on Computers and Communications 2003.

[40] I. Stoica, “Stateless Core: A scalable approach for quality of service in the internet”, PhD thesis, Carnegie Mellon University, Department of Electrical and Computer Engineering, December 15, 2000.

[41] P. McKenny, “Stochastic fairness queueing”, Proceedings of Infocom 1990, pp 733-740.

[42] A. Manin, and K. Ramakrishnan, “Gateway congestion control survey”, IETF RFC (Informational) 1254, August 1991.

[43] T.V. Lakshman, A. Neidhardt, and T.J. Ott, “The drop from front strategy in TCP and in TCP over ATM”, Proceedings of Infocom 1996, pp. 1242-1250, vol. 3, March, 1996.

[44] S. Floyd, and V. Jacobson, “On traffic phase effects in packet-switched gateways”, Internetworking: Research and Experience, pp. 115-156, vol. 3, September, 1992.

[45] TBIT, the TCP Behavior Inference Tool. http://www.icir.org/tbit/ecn-tbit.html.

[46] L. Kalampoukas, A. Varma, and K.K. Ramakrishnan, “Explicit window adaptation: a method to enhance TCP performance”, IEEE/ACM Transactions on Networking (TON), pp. 338-350, vol. 10, June, 2002.

[47] A. Koike, “TCP flow control with ACR information”, ATM Forum/97-0998, December, 1997.

[48] P. Narvaez, and K.Y. Siu, “An acknowledgment bucket scheme for regulating TCP flow over ATM”, IEEE Globecom 1997, pp 1838-1844, vol. 3, November, 1997.

[49] R. Satyavolu, K. Duvedi, and S. Kalyanaraman, “Explicit rate control of TCP applications”, Unpublished Manuscript 1999. http://www.ecse.rpi.edu/Homepages/shivkuma/research/papers/iwqos99-rate.ps

[50] Packeteer Inc., http://www.packeteer.com/

[51] S. Karandikar, S. Kalyanaraman, P. Bagal, and B. Packer, “TCP rate control”, ACM Sigcomm Computer Communication Review, pp. 45-58, vol. 30, no. 1, January, 2000.

[52] R. Jain, S. Kalyanaraman, R. Goyal, S. Fahmy, and R. Viswanathan, “The ERICA Switch Algorithm for ABR Traffic Management in ATM Networks”, IEEE/ACM Transactions on Networking, pp. 87-98, vol. 8, no. 1, February, 2000.

[53] S. Kalyanaraman, “Traffic Management for the Available Bit Rate (ABR) service in Asynchronous Transfer Mode (ATM) networks”, Ph.D. Dissertation, Department of Computer Science and Information Services, Ohio State University, August, 1997.

[54] S. Karandikar, S. Kalyanaraman, P. Bagal, and B. Packer, “TCP Rate Control”, ACM Sigcomm Computer Communication Review, pp. 45-58, vol. 30, no. 1, January, 2000.

[55] F. Baccelli, and P. Brémaud, “Elements of Queueing Theory”, Springer, Berlin, 2002.

[56] V.V. Govindaswamy, G. Záruba, and G. Balasekaran, “Analyzing the Receiver Window Modification Scheme of TCP Queues”, IEEE Infocom Global Internet Workshop, Barcelona, Spain, April 23-29, 2006.

[57] V.V. Govindaswamy, G. Záruba, and G. Balasekaran, “Easing Congestion in Computer Networks using the Receiver-Window Modification (RWM) Scheme,” Under Review, Journal of Universal Computer Science, 2006.

[58] V. V. Govindaswamy, G Záruba, and G. Balasekaran, "Complementing Current Active Queue Management Schemes with Receiver-Window Modification (RWM),” Technical Report 2005-6, April, 2005.

[59] Paolo L. Gatti, “Probability Theory and Mathematical Statistics for Engineers,” London, New York, Spon Press, 2005.

[60] V.V. Govindaswamy, G. Záruba, and G. Balasekaran, “Receiver-window Modified Random Early Detection (RED-RWM) Active Queue Management Scheme: Modeling and Analysis”, IEEE International Conference on Communications (ICC 2006), Istanbul, Turkey, 11-15 June 2006.

[61] V.V. Govindaswamy, G. Záruba, and G. Balasekaran, “Modeling and Analysis of the Receiver-Window Modified Random Early Detection (RED-

RWM) Active Queue Management Scheme," Under Review - Journal of Network and Systems Management, 2006.

[62] V. V. Govindaswamy, G Záruba, and G. Balasekaran, "Receiver-Window Modified Random Early Detection (RED-RWM) Active Queue Management Scheme - Modeling and Analysis," Technical Report 2006-7, March, 2006.

[63] V. V. Govindaswamy, G Záruba, and G. Balasekaran, "RECHOKe: A Scheme for Detection, Control and Punishment of Malicious Flows in IP Networks", Under Review, IEEE International Conference on Communications (ICC 2007), Glasgow, Scotland, UK, 24-28 June 2007.

[64] V. V. Govindaswamy, G Záruba, and G. Balasekaran, "Analyzing the Accuracy of CHOKe Hits and CHOKe-RED Drops," Under Review, IEEE International Conference on Communications (ICC 2007), Glasgow, Scotland, UK, 24-28 June 2007.

[65] V. V. Govindaswamy, G Záruba, and G. Balasekaran, "RECHOKe and RCUBE", UTA Technical Report 2006-7, September, 2006.

[66] V. V. Govindaswamy, G Záruba, and G. Balasekaran, "Protecting TCP-friendly Flow Behavior in TCP/IP Networks", Under Review ACM SIGCOMM Computer Communication Review, 2006.

[67] V. V. Govindaswamy, G Záruba, and G. Balasekaran, "Enhancing Internet Robustness against Malicious Flows using RECHOKe", Under Review IEEE/ACM Transactions on Networking, 2006.

BIOGRAPHICAL INFORMATION

Visvasuresh “Victor” Govindaswamy (a.k.a. The Bat) was born on the island of Singapore. He received his high school education in Singapore before graduating from The University of Texas at Austin, U.S.A., with a Bachelor of Science in Electrical and Computer Engineering. He was affectionately introduced as “The Bat Never Sleeps…” to students by his professors due to his ability to work on a problem until it is solved without the need for sleep. He received his Master of Science in Computer Science and Engineering in May, 2001 from The University of Texas at Arlington, Arlington, Texas. His thesis topic was on real-time distributed systems, funded by DARPA (Defense Advanced Research Projects Agency, the central research and development organization for the Department of Defense). Upon completion of his masters, he pursued his Ph.D. studies at the same Department of Computer Science and Engineering at The University of Texas at Arlington, Arlington, Texas.

The Bat Never Sleeps ….