The Pennsylvania State University

The Graduate School

Department of Computer Science and Engineering

QUALITY OF SERVICE (QoS) PROVISIONING IN THE INTERNET USING FLOW ESTIMATION

A Thesis in

Computer Science and Engineering

by

Sungwon Yi

Submitted in Partial Fulfillment
of the Requirements
for the Degree of

Doctor of Philosophy

August 2005

The thesis of Sungwon Yi has been reviewed and approved* by the following:

Chita R. Das
Professor of Computer Science and Engineering
Thesis Co-Advisor
Co-Chair of Committee

George Kesidis
Associate Professor of Electrical Engineering & Computer Science and Engineering
Thesis Co-Advisor
Co-Chair of Committee

Thomas F. La Porta
Associate Professor of Computer Science and Engineering

Guohong Cao
Associate Professor of Computer Science and Engineering

Constantino M. Lagoa
Associate Professor of Electrical Engineering

Ray Acharya
Professor of Computer Science and Engineering
Head of the Department of Computer Science and Engineering

*Signatures are on file in the Graduate School.

Abstract

Quality-of-service (QoS) provisioning in the Internet in the presence of unpredictable traffic dynamics is an important, but admittedly complex problem. The main goal of this thesis is to investigate flow estimation based techniques for better QoS support in the Internet. In this context, we discuss five related topics in this thesis. First, we propose a flow estimation scheme, called Hash-based Two-level Caching (HaTCh), to accurately estimate the number of active connections, which can be used for better congestion control. It is shown that the HaTCh scheme provides more accurate flow estimation than the exising (SRED)scheme under various workloads. Second, we investigate the design issues of drop functions in an AQM scheme and propose a new AQM scheme, called HaTCh based RED (HRED), based on the new flow estimation technique. Third, we propose a HaTCh-based Dynamic Quarantine (HaDQ) scheme, an extension of the HaTCh scheme, to detect and penalize unresponsive TCP flows, which pose a serious threat to conforming TCP users through Cenial of Service (DoS) attacks. Then, the HaDQ scheme is further extended to dynamically detect and block worm propagation. It is shown through extensive simulation that the HADQ mechanism is an effective and feasible solution for controlling bandwidth attack of unresponsive TCP flows and worms. Finally, we extend our AQM framework to wireless networks, and present a new AQM scheme, called Proxy-RED, for Wireless Local Area Networks (WLANs). The Proxy-RED scheme can enhance the performance of WLANs in terms of goodput and packet loss rate.

Table of Contents

List of Tables

List of Figures

Acknowledgments

This thesis represents six years of my life at Penn State, and I indebted to many people for their continual support, love, and help. First of all, I am grateful to my thesis advisors, Professor Chita R. Das and Professor George Kesidis, for the large doses of guidance, patience, and encouragement they have provided. They were not just advisors but also friends, brothers, and parents to me during my time at Penn State. I thank my committee members, Professor Thomas LaPorta, Professor Guahong Cao, and Professor Constantino Lagoa, for their insightful commentary on my work. I thank to Martin Kappes and Sachin Garg for giving me valuable opportunity to work at the Avaya Research Lab, and for their guidance toward more realistic aspects of my research. It was a great pleasure for me to collaborate with them. My thank also goes to my colleagues in the High Performance Computing Lab. for discussing wide range of topics, offering friendship, and sharing pleasant memories in Penn State. There are many friends in Korea including the members of Handabal, who have supported and encouraged me all the time. I would like to express my gratitude to all of them.

I especially feel a deep sense of gratitude to my parents, Taibok Yi and Kumja Kim for their love and care, and my brother and sisters, Heewon, Miwon, Jinwon and Myungwon, for caring my parents on my behalf and for their continuous support. Finally, I thank my wife, Heesung Kim, for her continual support, and my adorable son, Hojoon Andrew Yi, for giving me an additional dimension of pleasure in my life.

Chapter 1

Introduction

Recently, a number of empirical studies on the traffic measurement showed that a variety of network traffic exhibit *self-similarity* and *long-range dependency* [16] [71] [14]. These traffic characteristics, which imply time-invariant bursts, can cause serious performance degradation such as longer queueing delay and higher packet loss rate at the congested routers [7] [52], and thus make congestion control quite complex. In addition, millions of Internet users have recently experienced Denial-of-Service (DoS) due to sever network congestion, which is attributed not only to traffic dynamics and volume but due to malicious users or software. Therefore, developing an efficient mechanism that handles network congestion, whether it is naturally formed or maliciously intended, is essential for not only improving the network performance but also for providing Quality of Service (QoS) in the Internet.

Although congestion issues have traditionally been considered in wired network, the need for efficient congestion control in wireless networks is quite obvious. Recently, wireless networks based on the IEEE 802.11 standard have been widely deployed in enterprises and university campuses mostly to provide wireless data access to laptops, PDAs, etc., to the wired infrastructure such as Internet. However,the available bandwidth in IEEE 802.11 networks is much smaller than in wired local area networks. Therefore, the disparity in the link speed between wired and wireless networks makes the wireless

access point a significant potential bottleneck in the downstream direction, and finding solutions, considering wireless networks characteristics such as longer delay and higher packet loss rate, is critial for successful deployment of wireless services.

The main motivation of this research is to design and analyze an AQM scheme for better congestion control, to develop security measures to protect networks from malicious users and softwares, and to extend AQM concept to wireless networks to improve wireless network performance.

1.1 Active Queue Management

Internet architects addressed congestion problems by extending the TCP scheme to incorporate congestion control mechanisms such as slow start and fast retransmit. The main idea of the TCP congestion control mechanism is to regulate the packet injection rate according to the estimated level of congestion. A TCP source detects packet loss (i.e, congestion) by monitoring the transmitted and acknowledged packets, and backs off the transmission rate to avoid successive packet loss. If all the packets are successfully delivered, the sender *slowly* increases the transmission rate until it reaches the maximum rate. However, the TCP congestion control mechanism often results in *congestion collapse* [32] due to delayed congestion detection and activation of the back-off mechanism, and synchronized packet transmission from different senders.

Along with the TCP's congestion control mechanism, the Internet Engineering Task Force (IETF) recommended to deploy an Active Queue Management (AQM) scheme such as Random Early Detection (RED) [9]. Since then, several AQM schemes have been proposed to minimize the packet loss rate, to stabilize buffer occupancy, and

to prevent global synchronization in the Internet [22] [38] [50] [19]. The main role of an AQM scheme is to inform senders about the onset of any congestion in order to activate the TCP congestion control mechanism before the queue overflows, resulting in successive packet loss at the congested router. This can be achieved by detecting the level of congestion accurately and deploying a packet drop mechanism based on the level of congestion. However, none of these schemes are very effective due to limited information about the level of congestion and inefficient design of the drop mechanism.

1.2 Internet Security

According to the Census Bureau of the Department of Commerce, e-commerce accounted for 19.6% of U.S. manufacturing shipment, 11.7% of U.S merchant whole sale, and 28.1% of electronic shopping (including mail order) in 2002 [4]. Considering the fact that e-commerce handled 0.8% of U.S. retails in 2000 and it increased to 1.9% in 2004 [6], it is clear that U.S. economy will heavily rely on Internet technology in the near future. On the other hand, 83% of senior information technology executives acknowledged that their system had been compromised during 2003 in Global Security Survey 2004 [5]. In addition, 40% of them stated that their organization experienced financial loss due to these activities.

Today, one of the most common ways to breach security is a Denial-of-Service (DoS) attack. A DoS attack is a malicious attempt to cripple a target infrastructure, commonly by flooding packets, resulting in denial of service to legitimate users. Since February 2000's, DoS attacks on major Web sites such as Amazon.com, Yahoo.com, and Ebay.com have been one of the most serious threats to millions of users who rely on

the Internet for their daily business. DoS Attack technology has continued to evolve and recently include large scale attacks called worms, self-propagating mal-codes. Since the advent of the worm, known as morris worm [60], new and faster worms have been constantly released and threatened the Internet community. In July 2001, Code Red II infected more than 359,000 hosts within 14 hours [45], and it was followed by a series of fast spreading worms such as Nimda [66], SQLSlammer [44], and Sasser [67]. It is believed that a worm can theoretically infect more than hundreds of thousands of vulnerable hosts within a couple of minutes. A worm can therefore initiate any destructive activities on an infected machine such as corrupting data, killing processes, and installing Trojan horses and back doors [64] [65]. In addition, worms can consume serious amount of network resources including buffer and bandwidth, leading to sever congestion and denial of service to legitimate network users.

Although several solutions have been proposed for handling these problems, the performance of these schemes are seriously limited by assuming the specifics of a DoS/worm attack or by overlooking the implementation details. To our knowledge, there is no effective technique to handle DoS attacks in Internet.

1.3 Overview of Thesis

This thesis focused on finding practical solutions for these problems by i) designing and analyzing an Active Queue Management (AQM) scheme for better congestion control, ii) developing an algorithm, which can detect and penalize malicious flows used in bandwidth attack and worm propagation, and iii) extending the AQM concept to the

wireless domain such as Wireless Local Area Networks (LANs). The proposed research addresses the following five issues:

- To develop an accurate congestion estimation mechanism
- To design a new AQM scheme by combining accurate congestion estimation and an efficient drop mechanism
- To develop a router mechanism for controlling unresponsive flows
- To develop a network-based worm defense mechanism
- To study the performance of AQM in wireless LANs and improve wireless LAN performance using an AQM based approach

These ideas are discussed in greater detail in the rest of the thesis. Chapter 2 presents a new AQM scheme called HRED (HaTCh-based RED). The improved performance of HRED results from the accurate estimation of the congestion level by the Hash-based Two-level caChing (HaTCh). To accurately estimate the severity of congestion, we extend the SRED [48] scheme that estimates the number of active flows and use it to indicate the level of congestion. The proposed scheme can eliminate SRED's innate problems such as fluctuation in estimating the number of flows and under-estimation when misbehaving flows are mixed with TCP flows. The HaTCh scheme uses hashing and a two-level caching mechanism to accurately estimate the number of active flows under various workloads. Then, the estimated number of flows is used to determine the packet drop/marking probability of HRED along with the current queue capacity.

However, HRED provides better control in buffer occupancy and achieves low loss rate through efficient design of the dropping function.

The architecture of the HaTCh scheme is extended in Chapter 3 to detect and penalize unresponsive flows. The proposed scheme, called HaTCh-based Dynamic Quarantine (HaDQ), identifies high bandwidth flows without the collecting per-flow information, as is done in many cases. The identified flows are monitored using a small Content Addressable Memory (CAM), called the quarantine memory, and are rate-limited based on the number of active flows estimated by the HaTCh scheme.

Chapter 4 presents ongoing research on a worm defense mechanism based on the HaDQ architecture along with preliminary simulation results. The proposed scheme exploits HaTCh's hashing mechanism to sample multiple flows originated from the same source node. The sampled flows are quarantined using the quarantine memory of HaDQ, and the number of destination nodes are estimated using the similar technique used in HaTCh. The main feature of the proposed scheme is that it is independent of the underlying transport layer protocol and minimizes the memory requirement for detecting worm traffic. The preliminary simulations results show that the proposed scheme can detects a worm's probing traffic less than 3 seconds.

In Chapter 5, we study the use of RED [], a well known active queue management (AQM) scheme, and explicit congestion notification (ECN) [] to handle bandwidth disparity between a wired and the wireless interface of an access point. Then, we propose the Proxy-RED scheme as a solution for reducing the AQM overhead from the access point. Simulations-based performance analysis indicates that the proposed Proxy-RED scheme improves the overall performance of the network. In particular, the Proxy-RED

scheme significantly reduces packet loss rate and improves goodput for a small buffer, and minimizes delay for a large buffer size. The concluding remarks of the thesis are drawn in Chapter 6.

Chapter 2

An AQM Scheme based on the Number of Active Flows

An Active Queue Management (AQM) Scheme consists of two main functions. The one is to estimate the level of congestion and the other is to deploy random drop/marking. In this chapter, we first present an accurate and robust congestion estimation technique, and then extend it to design a complete AQM scheme, called HRED.

2.1 Introduction

Internet congestion control is an important, but admittedly complex problem primarily because of the unpredictable traffic dynamics. Several active queue management (AQM) schemes have been proposed for congestion control to minimize high packet loss rates and global synchronization in the Internet [19] [22] [36] [38] [40] [50]. Two crucial functions of an AQM scheme are to estimate the level of congestion and to respond accordingly either by randomly dropping or marking the packets. In these schemes, the average queue length [22] [38] [40] , link idle time or packet loss due to buffer overflow [19] was used as an indication of the congestion level. However, none of these schemes are very effective since they give limited information about the level of congestion.

Recently a new approach, called Stabilized RED (SRED)[1] [48], drew wide attention since it proposed to use the number of active flows as an indication of the congestion level. It is believed that the number of active flows is a better indicator of network congestion compared to other parameters such as average queue length and packet loss event. Furthermore, the number of active flows can be used as a configuration parameter to stabilize the AQM control systems [31] [36]. Therefore, an accurate estimation of this number will lead to improved congestion control.

In SRED, a small cache memory, called the *zombie list*, is used to record the M most recently seen flows. Each cache line (*zombie*) contains the source and destination address pair, last arrival time, and hit count of the flow. Each arriving packet (source and destination address) is compared with a randomly selected cache line. If the addresses match (called a hit), the hit count of the cache line is increased by 1. Otherwise (called a miss), the selected cache line is replaced by the arriving flow's address with a replacement probability r. To estimate the number of active flows, SRED maintains a hit frequency $f(t)$ and updates $f(t)$ with $(1-\alpha)f(t-1)+\alpha$ on a hit, and with $(1-\alpha)f(t-1)$ on a miss, where α is a time constant. Then, the inverse of the hit frequency ($f(t)^{-1}$) is used as the estimation of the number of active flows.

Although the SRED concept and the use of number of active flows as an indication of the severity of congestion are quite novel, a detailed performance analysis showed several limitations. For example, one of the problems in SRED is that the estimated number of flows fluctuates as the number of flows increases in the network. This implies that the severity of the congestion is not accurately captured. Second, although misbehaving

[1] In this chapter, we use "SRED" to denote the flow estimation capability of SRED.

flows such as UDP can be identified by computing the hit count and calculating the *total occurrence* of the flow as described in [48], SRED still underestimates the number of active flows when the traffic mix includes both TCP and aggressive UDP connections.

To address these problems, we first present a mathematical model to analyze the estimation capability of SRED, and show how the steady-state hit frequency of the cache model can be used to estimate the number of active flows. We then propose a modified SRED scheme, called HaTCh (Hash-based Two-level Caching), that uses hashing and a two-level caching mechanism to accurately estimate the number of active flows under various workloads. Unlike the original SRED, where the entire cache is a single block, the proposed scheme divides the cache into a fixed number of sub-blocks. With the hashing scheme, each arriving packet is hashed into one of the partitioned subcaches, and the hit frequency is maintained for each subcache. Due to the reduced size of the subcache and the number of flows per subcache, the hit probability of an arriving packet is improved. The hashing scheme stabilizes the estimation through this improved hit probability.

The proposed two-level caching scheme consisting of a smaller "Level 1" (L1) cache and a larger "Level 2" (L2) cache, basically works similar to the general two-level cache used in processor architecture design. It implies that an L1 cache miss results in an access to the L2 cache. However, the cache inclusion property is not satisfied here. An arriving packet is first compared with a randomly selected L1 cache line. If the addresses match, the hit count is incremented, otherwise a randomly selected L2 cache line is compared with the packet ID (the source and destination addresses). A hit in the L2 cache results in bringing the cache line to the corresponding cache line in the L1 cache. If the addresses do not match in the L2 cache (a miss), the L2 cache

line is replaced with the packet ID with a given replacement probability. The purpose of the L1 cache in HaTCh is to isolate the misbehaving flows from the L2 cache. The two-level caching scheme accurately estimates the number of active flows by isolating the misbehaving flows to prevent monopolization of the L2 cache, and to yield more room in the L2 cache for the conforming flows.

We extend the SRED analytical model for the proposed two-level caching scheme to demonstrate its effectiveness in isolating the misbehaving flows. We then analyze the performance of the HaTCh scheme through extensive simulations using the ns-2 simulator [3]. Estimation accuracy and stability of estimation in the presence of burst traffic and misbehaving flows are used as the main performance metrics to compare the proposed scheme with SRED. The simulation results indicate that the two-level scheme not only stabilizes the estimation but also improves the accuracy of estimation for various workloads. In particular, HaTCh out-performs SRED in the presence of misbehaving flows.

The rest of this chapter is organized as follows: In Section 2.2, we present a mathematical model to analyze the estimation capability of SRED. We describe the limitations of SRED in Section 2.3. The proposed estimation scheme, HaTCh, is detailed in Section 2.4. In Section 2.5, the simulation results are presented followed by the concluding remarks in Section 2.7.

2.2 SRED Model for Estimating the Number of Active Flows

The key idea of SRED is to relate the cache hit frequency to the number of active flows. However, this was not proved formally in the original paper [48] that relied on

simulation to arrive at the conclusion. In this section, we present a Markov model to understand the concept of SRED and analyze its estimating behavior.

Consider a small cache memory (*zombie list*) with M lines, and N independent flows, each with a rate λ_i packets/s. Here, the packet arrival process may *not* be necessarily Poisson. We *only* assume that the flow IDs of the multiplexed sequence of arriving packets are independent. If we assume X_i is the number of cache lines with the flow ID i in the cache and $\bar{X}$ is the vector that maintains the number of cache lines occupied by each flow, then $\bar{X}$ can be presented as a Markov chain whose transitions occur at packet arrival time. $\bar{X}$ and its state space can be defined as:

$$\sum_{i=1}^{N} X_i = M, \text{and } \bar{X} \in \{\bar{m} = \begin{pmatrix} m_1 \\ \cdots \\ m_i \\ \cdots \\ m_N \end{pmatrix} \mid 0 \leq m_i \leq M, \sum_{i=1}^{N} m_i = M\} \equiv \mathcal{S}_{M,N}.$$

Here, the transition rates of the Markov chain are dependent on both the replacement probability and the outcome of the cache comparison between an arriving packet and a randomly selected cache line. Based on the SRED's functionality, for a given cache state $\bar{m}$, the number of cache lines for each flow in the cache remains the same either in the event of a cache hit, or in the event of a cache miss with probability $1 - r$. However, the state of the cache changes from $\bar{m}$ to $\delta_{ij}\bar{m}$ in the event of a cache miss with replacement probability r.

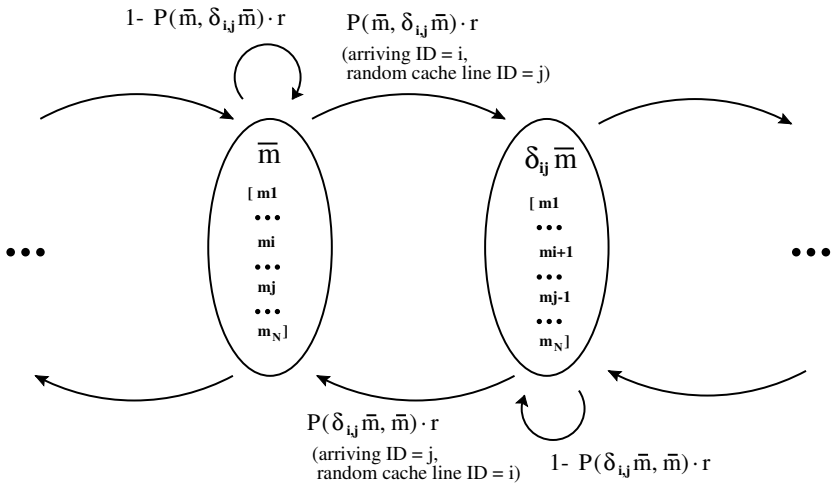


Fig. 2.1. The state transition diagram of $\bar{X}$.

In this figure, $\bar{m}$ and $\delta_{ij}\bar{m}$ represent the present and the next states. $P(\bar{m}, \delta_{ij}\bar{m})$ denotes the cache miss probability when a flow ID i is compared with a cache line j, and r is the replacement probability.

A pictorial view of the state space transition is given in Figure 2.1. When an arriving packet has a flow ID i, and a randomly chosen cache line belonging to flow ID j in the given cache state $\bar{m}$ is replaced by ID i with the replacement probability of r, the new state $\delta_{ij}\bar{m}$ is defined as:

$$\delta_{ij}\bar{m} = \begin{pmatrix} m_1 \\ \cdots \\ m_i + 1 \\ \cdots \\ m_j - 1 \\ \cdots \\ m_N \end{pmatrix} \text{ if i } < \text{ j}, \tag{2.1}$$

for all the cache states $\bar{m}$ such that $m_i < M$ and $0 < m_j$. If $i > j$, then the i and j terms are swapped. For a given state $\bar{m}$, the probability of a cache hit is the product of the probability that an arriving packet has the flow ID i and the probability that a randomly selected cache line has the same flow ID. Similarly, the probability of a cache miss is the product of the probability that an arriving packet has the flow ID i and the probability that a randomly selected cache line has a different flow ID than i. Therefore, the transition probability of this cache model can now be written for a cache hit or a cache miss without replacement as:

$$\begin{aligned} P(\bar{X}(t+1) = \bar{m} \mid \bar{X}(t) = \bar{m}) &\equiv P_{hit}(\bar{m}, \bar{m}) + P_{miss}(\bar{m}, \bar{m})\,(1-r) \\ &= \sum_{i=1}^{N} \frac{m_i}{M} \cdot \frac{\lambda_i}{\sum_{k=1}^{N} \lambda_k} + \sum_{\substack{i,j=1 \\ i \neq j}}^{N} \frac{m_j}{M} \cdot \frac{\lambda_i}{\sum_{k=1}^{N} \lambda_k}\,(1-r). \end{aligned} \tag{2.2}$$

For a cache miss that is replaced with probability r, the expression is:

$$P(\bar{X}(t+1) = \delta_{ij}\bar{m} \mid \bar{X}(t) = \bar{m}) \equiv P(\bar{m}, \delta_{ij}\bar{m})\, r = \sum_{\substack{i,j=1 \\ i \neq j}}^{N} \frac{m_j}{M} \cdot \frac{\lambda_i}{\sum_{k=1}^{N} \lambda_k}\, r. \tag{2.3}$$

In the above expressions, $\frac{\lambda_i}{\sum_{k=1}^{N} \lambda_k}$ denotes the probability that an arriving packet has a flow ID i, and the $\frac{m_i}{M}$ and $\frac{m_j}{M}$ terms represent the probability that a randomly selected cache line from the *zombie list* has the same and different IDs, respectively.

For a given state $\bar{m}$ such that $m_i < M$ and $0 < m_j$, the detailed balance equation of this system for any r becomes:

$$\pi(\bar{m})P(\bar{m}, \delta_{ij}\bar{m}) = \pi(\delta_{ij}\bar{m})P(\delta_{ij}\bar{m}, \bar{m}), \tag{2.4}$$

where

$$P(\bar{m}, \delta_{ij}\bar{m}) = \frac{m_j}{M} \cdot \frac{\lambda_i}{\sum_{k=1}^{N} \lambda_k}$$

$$\text{and } P(\delta_{ij}\bar{m}, \bar{m}) = \frac{m_i + 1}{M} \cdot \frac{\lambda_j}{\sum_{k=1}^{N} \lambda_k}.$$

Here, $\pi(\bar{m})$ is the steady-state distribution of $\bar{X}$. Thus, (2.4) becomes

$$\pi(\bar{m}) = \pi(\delta_{ij}\bar{m}) \frac{\lambda_j}{m_j} \cdot \frac{m_i + 1}{\lambda_i}. \tag{2.5}$$

We have found that the following distribution solves the detailed balance equation (2.4):

$$\pi(\bar{m}) = \frac{\prod_{n=1}^{N} \frac{\lambda_n^{m_n}}{m_n!}}{G_{MN}}, \tag{2.6}$$

where the normalizing constant is

$$G_{MN} = \sum_{\bar{m} \in \mathcal{S}_{M,N}} \prod_{n=1}^{N} \frac{\lambda_n^{m_n}}{m_n!}. \tag{2.7}$$

Thus, we can conclude that the process $\bar{X}$ is time reversible. From (2.5), (2.6) and (2.7) we get:

$$\pi(\delta_{ij}\bar{m}) = \frac{\prod_{n \neq i,j} \frac{\lambda_n^{m_n}}{m_n!} \cdot \frac{\lambda_i^{m_i+1}}{(m_i+1)!} \cdot \frac{\lambda_j^{m_j-1}}{(m_j-1)!}}{G_{MN}}.$$

Now, let us denote $f(t)$ as the hit frequency, and H as the steady-state hit probability of the cache. Then, H includes the summation of all states as:

$$H = \sum_{\bar{m} \in \mathcal{S}_{M,N}} \pi(\bar{m}) P_{hit}(\bar{m}, \bar{m}). \tag{2.8}$$

In practice, H can be estimated by a first order autoregressive process, defined as:

$$f(t) = (1-\alpha) f(t-1) + \alpha \cdot \mathit{1}\ \{\text{hit at } t^{th} \text{ packet }\}$$

for $0 < \alpha < 1$. *1* in the above expression represents an indicator function of a cache hit. It is clear that H is a limiting point of the process f. ie, $\lim_{t \to \infty} f(t) \equiv H$. We assume that the choice of α is such that $f(t)$ converges faster than the rate of change of N. (TCP estimates the round trip time using an autoregressive process too.) Here, α is a time constant that determines the speed of the model to reach the steady-state. Finally,

from (2.2), (2.6) and (2.8), we express the steady-state hit frequency of the given cache model as the following theorem.

THEOREM 1. *Under the assumption of independent packet flow identifiers, the hit frequency, calculated by a first order autoregressive process for the single cache system (SRED), converges to*

$$H = \sum_{\bar{m} \in \mathcal{S}_{M,N}} \left(\frac{\prod_{n=1}^{N} \frac{\lambda_n^{m_n}}{m_n!}}{G_{MN}} \sum_{i=1}^{N} \left(\frac{m_i}{M} \cdot \frac{\lambda_i}{\sum_{k=1}^{N} \lambda_k} \right) \right) \tag{2.9}$$

in the steady-state.

Observe that if each λ_i in the summand of the numerator of (2.9) is replaced by the maximum value of λ_{max}, then H is less than or equal to $\frac{\lambda_{max}}{\sum_{k=1}^{N} \lambda_k}$. Similarly, if each λ_i in the summand of the numerator of (2.9) is replaced by the minimum value of λ_{min}, then H is greater than or equal to $\frac{\lambda_{min}}{\sum_{k=1}^{N} \lambda_k}$. Therefore, we have the following two corollaries.

COROLLARY 1.

$$\frac{\lambda_{min}}{\sum_{k=1}^{N} \lambda_k} \leq H \leq \frac{\lambda_{max}}{\sum_{k=1}^{N} \lambda_k}. \tag{2.10}$$

COROLLARY 2. *If all the arriving rates, λ_i, are equal for all N, then the upper and lower bounds on H in (2.10) are equal, leading to*

$$H = \frac{1}{N}.$$

The above expression shows that one can compute the number of active flows (N) from the steady-state hit frequency (H). We calculated the steady-state hit frequency and the number of flows using (2.9) and compared them with the simulation results. Due to the large state space ($_{M+N-1}\mathrm{C}_{\mathrm{M}}$), we first investigate the accuracy of our model with a relatively small cache memory (10 cache lines) and a small number of flows (10 or less) by enumerating the entire state space. We then extend the validation for large cache size (up to 800 cache lines) and more number of flows (up to 100) by using various state space truncation techniques.

When the same arrival rate is used for all flows, the estimation using (2.9) was exactly the same as the actual number of flows. Table 2.1 shows the results of the comparison between the estimation by (2.9) and the simulation results. (We used the ns-2 simulator and the simulation environment is described in Section 2.3.) The results indicate that the model is quite robust in estimating the number of flows. However, the simulation results showed that the accuracy of estimation depends on the cache size (M) and the replacement probability (r). A larger cache size, at least equal to greater than the number of flows, and a smaller replacement probability (r) help in better estimation.

2.3 Limitations of SRED

In order to examine the capability of SRED in estimating the number of active flows, we performed a number of simulations using the network, shown in Figure 2.2. In the simulations, all the connection requests are generated at the leftmost nodes and

Table 2.1. Estimated Number of Flows

Number of Flows	Memory Size	Estimated Number by Equation (9)	Estimated Number by Simulation		
			$r = 1.0$	$r = 0.25$	$r = 0.01$
10	5	10	31.8063	12.5903	10.3349
	10	10	16.1963	11.6975	10.2145
	20	10	13.1511	10.4804	10.3596
	40	10	11.5175	10.1427	9.9184
	80	10	10.5701	9.4195	9.9970
50	25	50	161.7820	63.3137	52.8626
	50	50	88.0117	56.6543	51.3475
	100	50	66.5306	54.1983	52.2865
	200	50	59.6492	50.2373	51.7769
	400	50	52.7332	52.1293	50.8571
100	50	100	334.7113	137.3253	109.6963
	100	100	182.7144	115.3155	108.0594
	200	100	143.9651	118.6892	100.2126
	400	100	112.5784	104.5773	106.1109
	800	100	116.2514	110.4582	105.9616

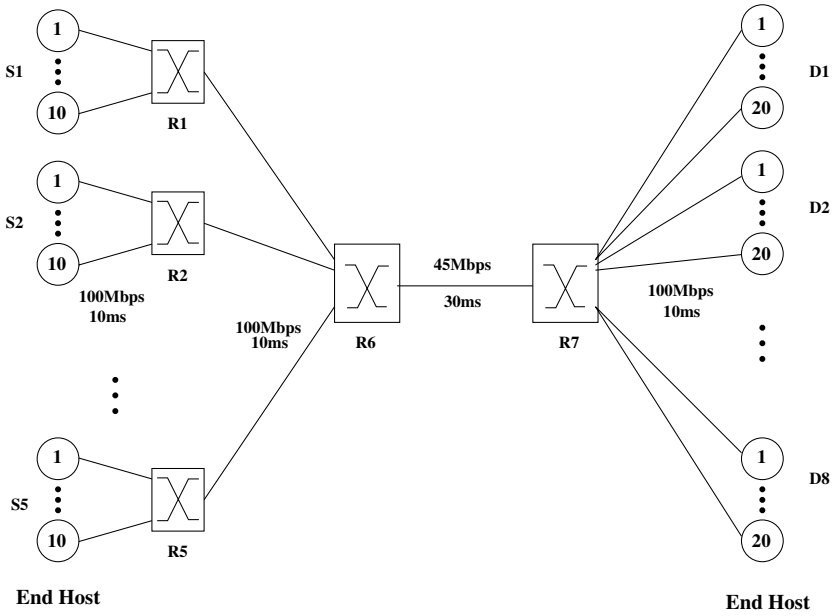


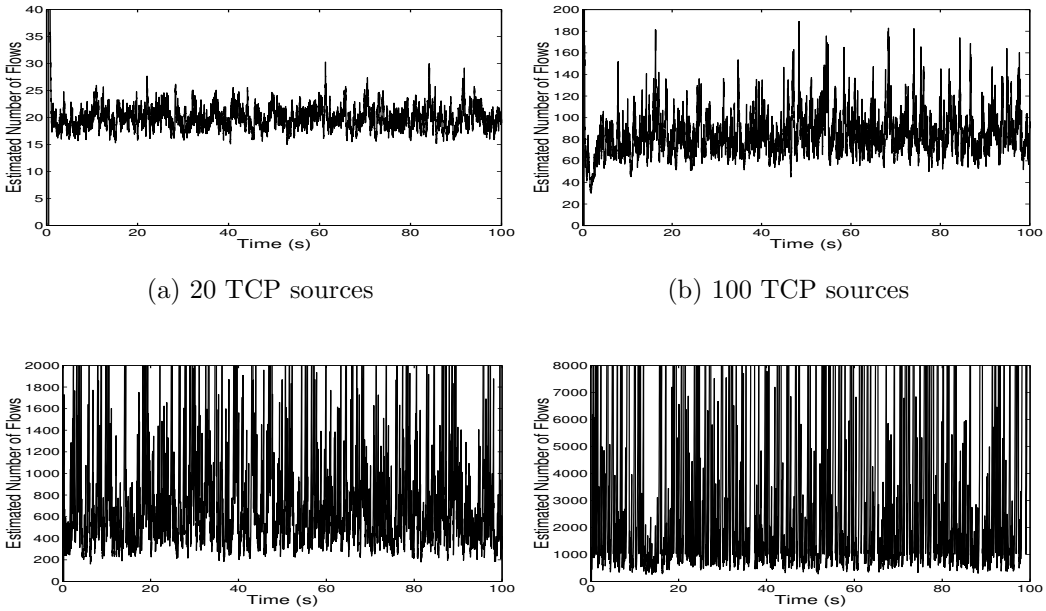
Fig. 2.2. The network topology used for simulation

terminate at the rightmost nodes, and all the sources randomly initiate packet transmission between 0 to 1s. Each intermediate node has a buffer size of 600 packets, while the packet size is fixed at 1 K bytes. SRED is deployed at R6 to estimate the number of active flows with a cache size of 1000 lines, replacement probability $(r) = 0.25$, and $\alpha = 0.001$, as used in [48].

We investigated three important factors that mostly affect the estimation performance of SRED: stability of the estimation, impact of burst traffic, and impact of misbehaving flows.

Stability : Figure 2.3 shows the estimated number of active flows with SRED when 20, 100, 1000, and 4000 flows are used. The estimation capability of SRED is quite accurate with a small number of flows, but it highly fluctuates as the number of flows increases. The fluctuation results from the low hit probability when a large number of flows is used.

Impact of burst traffic : Figures 2.4 (a) and (b) show the estimated number of flows for 500 TCP and 500 UDP connections, respectively. The estimation behavior of SRED was not the same for the two types of traffic. Successive arrival of the burst traffic sources (TCP connections) caused more hits within a short period of time, and this resulted in more stable but lower estimation in Figure 2.4 (a) compared to that in Figure 2.4 (b). Unlike the argument in SRED, SRED works properly only under proper configurations. The replacement probability (r) implies not only the lifetime of the cache line, but also the sampling frequency of the caching mechanism in that the cache line is updated (sampled) roughly for every $\frac{M}{r}$ packets. By lowering the sampling frequency, the number of packets sampled from the burst flows was reduced.



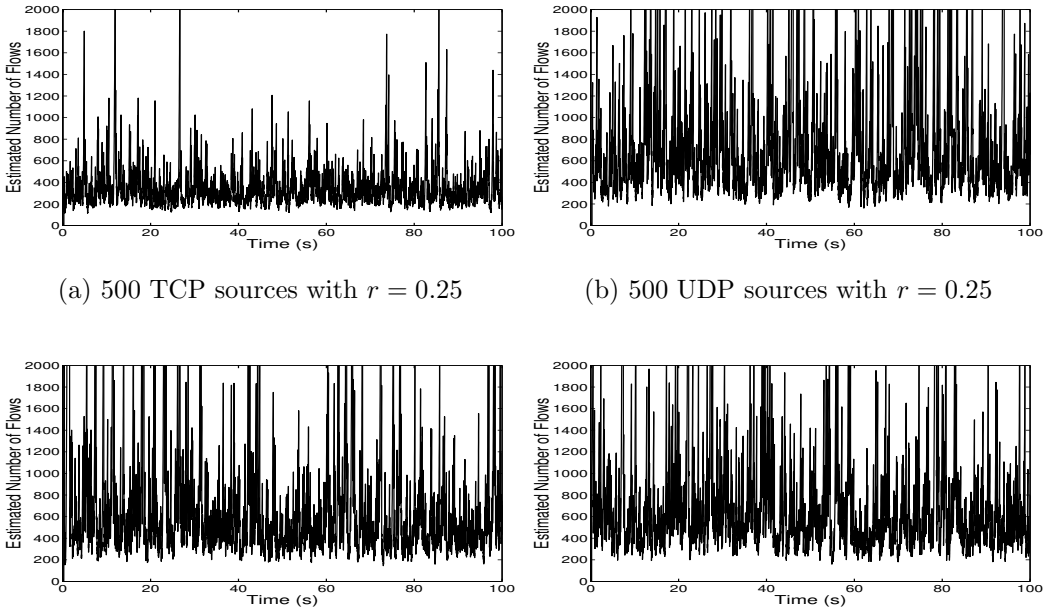
(a) 20 TCP sources

(b) 100 TCP sources

(c) 1000 TCP sources

(d) 4000 TCP sources

Fig. 2.3. Estimated number of flows with SRED



(a) 500 TCP sources with $r = 0.25$

(b) 500 UDP sources with $r = 0.25$

(c) 500 TCP sources with $r = 0.01$

(d) 500 UDP sources with $r = 0.01$

Fig. 2.4. Impact of burst traffic in estimating the number of flows

Now, we changed the replacement probability to 0.01. Figures 2.4 (c) and (d) show that the effect of the burst flows was significantly reduced without degrading the response time, but the fluctuation still remained as a problem. Note from Figures 2.4 (c) and (d) that both the TCP and UDP flows exhibit similar performance unlike in Figures 2.4 (a) and (b).

Impact of misbehaving flows : The effect of misbehaving flows (generally UDP), when mixed with TCP flows has been extensively studied in the context of AQM schemes [38] [40]. These studies tried to detect the misbehaving flows with a minimum amount of per flow information. Another approach, called SFB [19], controls the misbehaving flows without using the per flow information via a group of hash tables.

A problem of both SFB and SRED is that when a large number of misbehaving flows is present in a network, the hash table (or *zombie list*) is contaminated by these flows, and the performance of these schemes is significantly degraded. We demonstrate the impact of misbehaving flows using the SRED mathematical model in Table 2.2 and through a simulation study in Section 2.5.

We again used a cache size of 10 lines, with 10 flows, and set the arrival rate of misbehaving flows (λ_m) as 2 and 3 times that of the conforming flows (λ_c) to mimic the behavior of misbehaving flows in the mathematical model (equation (2.9)). The small memory size and number of flows are not enough to capture the exact effect of misbehaving flows, but it helps us to predict the tendency when a large number of flows is used. Table 2.2 shows that as the fraction of misbehaving flows increases, the number of active flows is underestimated. When misbehaving flows become a dominant part of the traffic, the estimated number starts to recover from underestimation. These results

have a similar trend as that of the simulation result with a total 500 flows (TCP + UDP) presented in Section 2.5 (Figure2.9).

Table 2.2. Impact of misbehaving flows in estimating the number of flows

Fraction of Misbehaving Flows in Total Workload	Estimated Number of Flows (when $\lambda_m = 2 \cdot \lambda_c$)	Estimated Number of Flows (when $\lambda_m = 3 \cdot \lambda_c$)
0 %	10	10
10%	9.309	8.001
20%	9.000	7.541
30%	8.897	7.529
40%	8.907	7.712
50%	9.002	8.000

In summary, SRED exhibits unstable estimation with a large number of flows, different estimation behavior for different traffic characteristics, and underestimation in the presence of misbehaving flows.

2.4 The Proposed Hash-based Two-level Caching Scheme (HaTCh)

Based on the discussions in the previous section, we propose a new active flow estimation scheme, called HaTCh (Hash-based Two-level Caching) that can minimize SRED's innate problems as discussed in the previous section. The HaTCh scheme consists of two parts. The first part is a hashing scheme to stabilize the estimation, and the

other is a two-level caching scheme to isolate the misbehaving flows that contaminate the *zombie list* and lead to underestimation.

2.4.1 A Hash-Based Estimation

The key idea of the hashing scheme comes from the observation that SRED's hit probability is low for large number of flows. This problem can be alleviated by using a hashing scheme. To implement hashing, the single cache memory (*zombie list*) is partitioned into k small chunks, called subcaches. Whenever a packet arrives, the packet is hashed into a subcache using the source and destination addresses, and a cache line is randomly selected for comparison from the subcache. Note that a connection is always hashed to the same subcache with this technique. Each subcache maintains its own hit frequency and the estimated number of flows, and the estimated number of flows per subcache is aggregated to find the total number of estimated flows. The performance of the hash-based estimation may degrade when most active flows are hashed into one or two subcaches, but this can be alleviated by periodically scattering the hash function as noted in [42]. When all the flows have the same sending rate and round trip time, the hit probability of a flow is $(\frac{1}{N})^2$ under SRED, but the hit probability increases to $(\frac{k}{N})^2$ when k subcaches are used in HaTCh. This improved hit probability helps in getting an accurate and stable flow estimation.

2.4.2 A Two-level Caching Scheme

The motivation for a two-level caching comes from the fact that the misbehaving flows tend to send more packets than the conforming flows, and this increases the hit

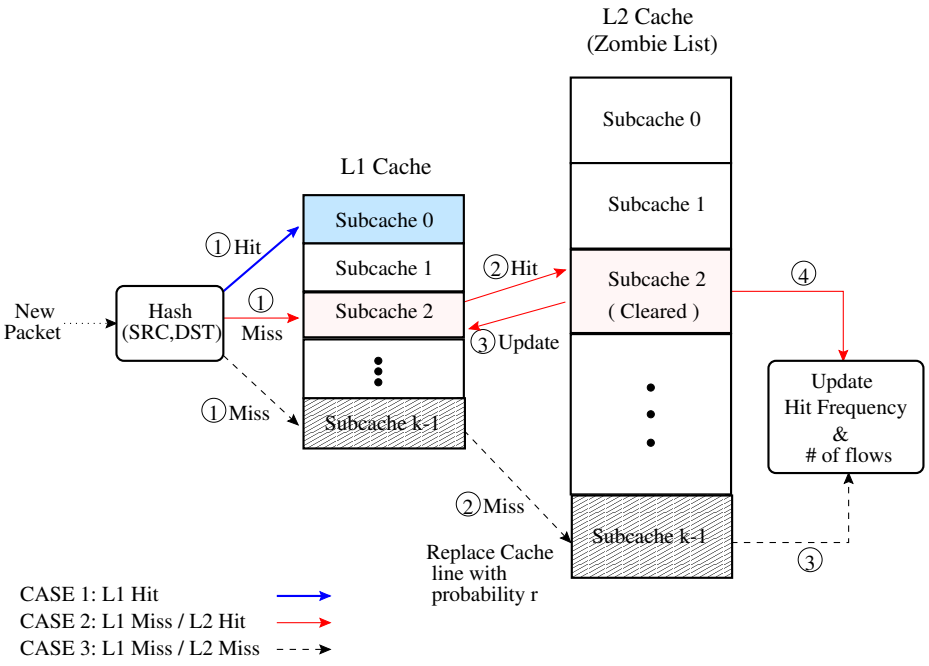


Fig. 2.5. HaTCh (Hash-based Two-level Caching) Architecture

rate. Therefore, developing an efficient scheme to isolate *excess* packets from the memory is the key for accurate estimation. We accomplish this through a two-level cache design.

Figure 2.5 shows the basic organization of HaTCh, which combines hashing and the two-level caching. The structure of the two-level caching proposed here is similar to that of the general two-level caching scheme. The major differences are that the inclusion property is not necessarily satisfied in the two-level cache model proposed here, and the L1 and L2 cache update operations are also different. L1 is a smaller cache compared to the second level L2 cache, and each of the two caches are divided into k subcaches (blocks). Note that corresponding to each subcache in L1, there is a subcache in L2.

An arriving packet is hashed into one of the L1 subcaches using the source and destination addresses. Then a randomly selected cache line from the L1 subcache is compared with the arriving packet. If there is a L1 cache hit (Case 1 in Figure 2.5), the hit count of the cache line is increased by 1, but the hit frequency of this subcache remains the same. Otherwise, the corresponding L2 subcache is selected. Then, a randomly selected L2 cache line in the corresponding subcache is compared with the arriving packet. If there is a L2 cache hit (Case 2 in Figure 2.5), the previously selected L1 cache line is updated with this L2 cache line, and the hit count of L1 cache line is set to 1, and the L2 cache line is then cleared. If the L2 cache misses (Case 3 in Figure 2.5), the L2 cache line is replaced with the arriving packet with a probability r. Irrespective of whether there is a hit or miss in the L2 cache, the hit frequency and the number of estimated flows for the subcache are recalculated, and also the total number of flows. The sequence of operations for the three cases is shown by the solid, dotted and dashed lines in Figure 2.5.

The key features of HaTCh are following: First, HaTCh takes advantage of the improved hit probability though hashing in both L1 and L2 caches. This stabilizes the estimation process of HaTCh. Second, the hit frequency is recalculated only when there is a L1 cache miss. The L1 cache is updated only when a flow hits the L2 cache and thus, the L1 cache is generally shared by the flows that hit the L2 cache. Filtering these flows to the L1 cache provides a fair chance for all the flows to update the hit frequency. Third, on an L2 cache hit, the selected cache line is cleared to yield room for the following conforming flows; a mechanism called L2 cache cleaning. Ideally, the L2 cache (zombie list) should be shared uniformly among all competing flows to yield an accurate flow estimation. Although the misbehaving flows are filtered in the L1 cache, the flows that missed the L1 cache will fill the L2 cache more aggressively than conforming flows. Therefore, the cleaning mechanism in the L2 cache also contributes to fair sharing of the L2 cache.

2.4.3 A Preliminary Model of the HaTCh Scheme

In this section, we present a mathematical model for the proposed HaTCh scheme to demonstrate its effectiveness in isolating the misbehaving flows. Unlike the SRED model presented earlier, where we were able to compute the hit frequency from the steady-state distribution of the cache lines occupied by each flow, here we simply show how the proposed scheme controls the misbehaving flows. As we will see, the model for the HaTCh is extremely complex, and thus, the state space enumeration and computation is expensive. Thus, in stead of focusing on the solution of the steady-state

probabilities, here we show through a Markov model why the two level caching is better than SRED.

The model extends the single cache design to capture the two-level cache memory with M_1 lines for the L1 cache and M_2 lines for the L2 cache ($M_1 \ll M_2$), and N independent flows. If we assume that (X_i, Y_i) be the number of cache lines with the flow ID i in the L1 and L2 caches respectively, and ($\bar{X}$, $\bar{Y}$) be a pair of vectors representing the number of cache lines in L1 and L2 occupied by each flow, then ($\bar{X}$, $\bar{Y}$) represents the discrete state model of the system. Now the state space for $\bar{X}$, $\bar{Y}$ can be defined as:

$$\bar{X} \in \{\bar{c} = \begin{pmatrix} c_1 \\ \cdots \\ c_i \\ \cdots \\ c_N \end{pmatrix} \mid 0 \leq c_i \leq M_1, \sum_{i=1}^{N} c_i = M_1\} \equiv S_X,$$

$$\bar{Y} \in \{\bar{m} = \begin{pmatrix} m_1 \\ \cdots \\ m_i \\ \cdots \\ m_N \end{pmatrix} \mid 0 \leq m_i \leq M_2, \sum_{i=1}^{N} m_i \leq M_2\} \equiv S_Y,$$

$$\text{and } \mathcal{S}_{M_1,M_2,N} \equiv S_X \times S_Y.$$

Let us assume an arriving packet has a flow ID i, a randomly selected cache line from the L1 cache has a flow ID j, and a randomly selected cache line from the L2 cache

has a flow ID k. For a given cache state ($\bar{c}$, $\bar{m}$), if there is a L1 cache hit or a miss in both the caches and the cache line is not replaced in the L2 cache, the number of cache lines for each flow in both caches remains the same, ($\bar{c}$, $\bar{m}$). On the other hand, if there is an L1 cache miss and an L2 cache hit, the state changes to a next state $\gamma_{ij}(\bar{c}, \bar{m})$, where $c_i = c_i + 1, c_j = c_j - 1$, and $m_i = m_i - 1$. If both L1 and L2 caches incur miss and the L2 cache is replaced with the replacement probability of r, the state changes to another state $\eta_{ij}(\bar{c}, \bar{m})$, where $m_i = m_i + 1$, and $m_k = m_k - 1$.

For a given state $(\bar{c}, \bar{m})$, the probability of an L1 cache hit is the product of the probability that an arriving packet has the flow ID i and the probability that a randomly selected L1 cache line has the same flow ID. Accordingly, the probability of an L1 cache miss and L2 cache hit is the product of the probability that an arriving packet has the flow ID i and the probability that a randomly selected L1 cache line has a flow ID other than i, and the probability that a randomly selected L2 cache line has the flow ID i. Similarly, the probability of the L1 and L2 cache miss is computed by considering the probabilities that both the L1 and L2 cache lines have different flow IDs other than i.

Now, the transition probability of the two-level cache model can be written for an L1 cache hit or for miss in the both caches and without replacement in the L2 cache as:

$$
\begin{aligned}
&P((\bar{X}, \bar{Y})(t+1) = (\bar{c}, \bar{m}) \mid (\bar{X}, \bar{Y})(t) = (\bar{c}, \bar{m})) \\
&\quad \equiv P_{hit}((\bar{c}, \bar{m}), (\bar{c}, \bar{m})) + P_{miss}((\bar{c}, \bar{m}), (\bar{c}, \bar{m}))\ (1 - r) \\
&= \sum_{i=1}^{N} \frac{c_i}{M_1} \cdot \frac{\lambda_i}{\sum_{z=1}^{N} \lambda_z} + \sum_{\substack{i,j,k=1 \\ i \neq j, i \neq k}}^{N} \frac{c_j}{M_1} \cdot \frac{m_k}{\sum_{z=1}^{N} m_z} \cdot \frac{\lambda_i}{\sum_{z=1}^{N} \lambda_z}\ (1 - r). \qquad (2.11)
\end{aligned}
$$

For an L1 miss and L2 hit, the state transition probability becomes,

$$P((\bar{X},\bar{Y})(t+1)=\gamma_{i,j}(\bar{c},\bar{m}) \mid (\bar{X},\bar{Y})(t)=(\bar{c},\bar{m})) \equiv P((\bar{c},\bar{m}),\gamma_{ij}(\bar{c},\bar{m}))$$

$$= \sum_{\substack{i,j=1 \\ i\neq j}}^{N} \frac{c_j}{M_1} \cdot \frac{m_i}{\sum_{z=1}^{N} m_z} \cdot \frac{\lambda_i}{\sum_{z=1}^{N} \lambda_z} = \sum_{i=1}^{N} (1-\frac{c_i}{M_1}) \frac{m_i}{\sum_{z=1}^{N} m_z} \cdot \frac{\lambda_i}{\sum_{z=1}^{N} \lambda_z}. \quad (2.12)$$

For the miss in the both caches and the L2 cache line being replaced with probability r, the equation becomes,

$$P((\bar{X},\bar{Y})(t+1)=\eta_{i,j}(\bar{c},\bar{m}) \mid (\bar{X},\bar{Y})(t)=(\bar{c},\bar{m})) \equiv P((\bar{c},\bar{m}),\eta_{ij}(\bar{c},\bar{m}))\, r$$

$$= \sum_{\substack{i,j,k=1 \\ i\neq j, i\neq k}}^{N} \frac{c_j}{M_1} \cdot \frac{m_k}{\sum_{z=1}^{N} m_z} \cdot \frac{\lambda_i}{\sum_{z=1}^{N} \lambda_z}\, r. \quad (2.13)$$

Note that these equations are derived using the same context that we used for SRED model. Therefore, for the two-level cache under HaTCh, we can prove the following theorem following the idea of (2.8).

THEOREM 2. *Under the assumption of independent packet flow identifiers, the hit frequency calculated by a first order autoregressive process for the two-level caching system (HaTCh) converges to*

$$H = \sum_{\substack{(\bar{c},\bar{m})\in \\ \mathcal{S}_{M_1,M_2,N}}} \pi(\bar{c},\bar{m}) P((\bar{c},\bar{m}),\gamma_{ij}(\bar{c},\bar{m}))$$

$$= \sum_{\substack{(\bar{c},\bar{m})\in \\ \mathcal{S}_{M_1,M_2,N}}} \pi(\bar{c},\bar{m}) \sum_{i=1}^{N} (1-\frac{c_i}{M_1}) \frac{m_i}{\sum_{z=1}^{N} m_z} \cdot \frac{\lambda_i}{\sum_{z=1}^{N} \lambda_z} \quad (2.14)$$

in the steady-state.

In the above equation $\pi(\bar{c}, \bar{m})$ represents the steady-state distribution of $(\bar{X}, \bar{Y})$. Unlike the SRED model, it is difficult to find a closed form expression for $\pi(\bar{c}, \bar{m})$. Thus, instead of computing the state probability distribution, we explain how equation (2.14) captures the misbehaving flows.

The steady-state hit frequency is determined by the steady-state cache line distribution and the steady-state hit probability as shown in (2.9) and (2.14). In (2.9) for the SRED model, the effect of the misbehaving flows is magnified by both the arrival rate of the misbehaving flows, λ_i, and the number of cache lines occupied by the flows, $\frac{m_i}{M}$, which is proportional to the arrival rate. This leads to the underestimation of SRED when misbehaving flows are present. In contrast, (2.14) clearly indicates how HaTCh effectively isolates the misbehaving flows and yields more accurate estimation. First, HaTCh clears the selected L2 cache line on an L2 cache hit (L2 cache cleaning mechanism) to create more room for the following conforming flows, and this contributes to a fair distribution of the L2 cache lines for all active flows. The term $\frac{m_i}{\sum_{z=1}^{N} m_z}$ in (2.14) captures this because it represents the probability that the flow ID in the L2 cache is i. Second, the distribution of the L1 cache lines, which is occupied by other flows, the $(1 - \frac{c_i}{M_1})$ term in (2.14) mitigates the effect of arrival rate of the misbehaving flows, λ_i. As a result, HaTCh yields more accurate and stable estimation even in the presence of misbehaving flows. We validate our claim about the effectiveness of HaTCh through simulation results in the next section.

2.4.4 Limitations of HaTCh

Although the HaTCh scheme stabilizes the estimation and isolates the misbehaving flows, understanding the limitations of HaTCh is important to avoid possible performance penalty. For example, if all the N active flows are hashed into the first half of the subcaches, the estimated number of flows becomes N. Later, these flows could terminate and N new flows could be hashed into the other half of the subcache. Although the actual number of flows is N, the estimation becomes $2N$. Another problem comes from the two-level caching. Assume that N flows are initiated and hashed into all different subcaches, and the L1 cache and the L2 cache are filled. Once the L1 cache is completely filled, each of the 10 flows will keep on hitting at the L1 cache before the hit frequency reaches the steady-state. The estimation thus becomes much larger than the actual number of flows.

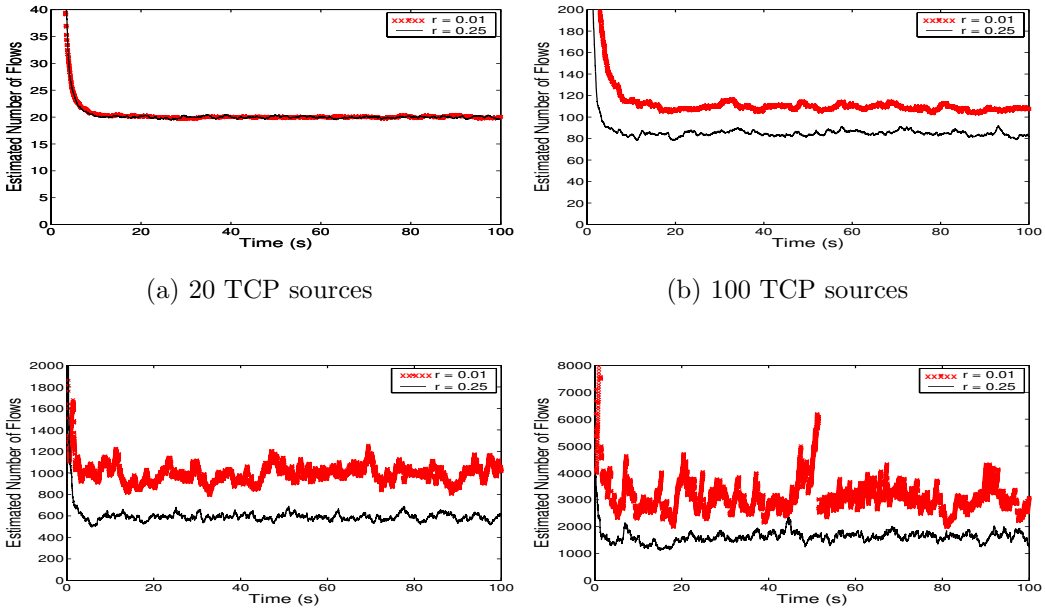
We propose two simple solutions for these problems: Periodic reset and Exponential back off of the hit frequency per subcache. For example, each subcache can keep track of the last time when the hit frequency was updated, and it is periodically compared with the current time. The number of estimated flows per subcache is then reset to 1 (if there is an L1 cache hit in the period, otherwise reset to 0) or is backed off to half when there is no update during the period. However, we did not perform further investigation of this problem since these cases are very rare in practice, and are not relevant to our investigation of the estimation performance.

2.5 Performance Evaluation

The proposed HaTCh scheme was simulated to analyze estimation stability, impact of burst flows, and impact of misbehaving flows, as has been done in Section 2.3. In our simulation, we configured HaTCh with a hash size (k) of 10, and L1 and L2 cache sizes as 100 and 1000 lines, respectively. The L1 cache size is fixed at 10% of the L2 cache size, which in turn is kept closer to the number of flows (N). The results are summarized below.

Stability : Figure 2.6 shows the estimated number of active flows with HaTCh for two different replacement probabilities (r = 0.25 and 0.01) when 20, 100, 1000, and 4000 TCP flows are used. The estimated number of flows is remarkably stabilized compare to Figure 2.3 at the cost of a little bit of more delay. However, HaTCh also showed a tendency to underestimate the number of flows for a replacement probability of 0.25 as the workloads increased. The accuracy of the estimation is significantly improved with a replacement probability of 0.01 as was shown in the previous section. Up to 1000 flows, the estimated number of flows oscillated within 20% range with HaTCh. As the number of flows increased, the fluctuation was more and also the number of flows was underestimated due to the traffic burst and insufficient cache size.

Theoretically, SRED like mechanisms (including HaTCh) can keep track of $\frac{N}{r}$ flows as discussed in [48]. We observed that if the number of active flows (N) is greater than the number of L2 cache lines (M_2), the HaTCh performance starts to degrade (oscillation exceeds 20% ranges). However, unlike SRED, HaTCh showed smooth (damped)



(a) 20 TCP sources

(b) 100 TCP sources

(c) 1000 TCP sources

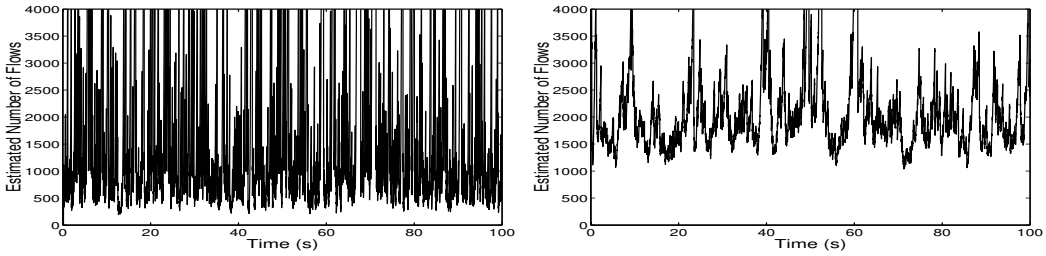
(d) 4000 TCP sources

Fig. 2.6. Estimated number of flows with HaTCh

oscillation even for 8000 TCP flows, but the oscillation gradually increased as we increased the number of flows from 1000 to 8000. Since the memory requirement for each cache line is very marginal (about 16 bytes), we believe that the L2 cache can accommodate a large number of flows (16M bytes L2 cache can support 1 million flows with about 20% error bound).

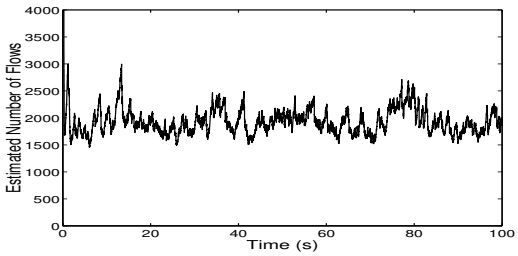
Assuming a perfect hash function, the optimal hash size could be the same as the L2 cache size, since all the flows could be hashed into exactly one cache line. In the following simulation, we examined the impact of hash size (k) in the estimation behavior of HaTCh with 2000 TCP flows, while the total memory requirement remains the same (1000 cache lines). Figure 2.7 depicts the estimation results of HaTCh with different hash sizes. As noted in the comparison between SRED and HaTCh, hashing with L2 cache cleaning mechanism generally degrades the response time especially when a small number of flows is used. The delay is due to the time required to fill the L2 cache lines. Although the hash size of 40 outperformed all other cases in terms of accuracy, it also has the longest response time. We leave this as a design study that optimizes the accuracy and response time of the system.

Impact of burst traffic : As discussed in Section 2.3, burst nature of TCP traffic affects the estimation accuracy of SRED. The impact of burst traffic is significantly reduced by configuring SRED with appropriate parameters. Therefore, we evaluated the performance of the two-level caching in the presence of burst traffic in this section. To investigate the impact of more realistic and burst traffic in the estimation, we used 500 Pareto On/Off sources with a shape parameter $s = 1.2$ (corresponding to the Hurst parameter of 0.9), which represents *long-range dependency* [47] [71]. Here, we defined a

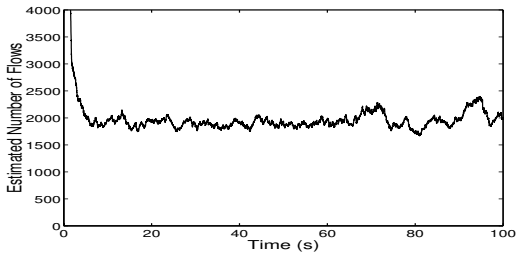


(a) No Hash (Single Block)

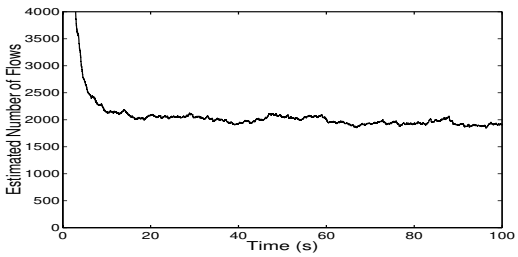
(b) Hash size = 5



(c) Hash size = 10

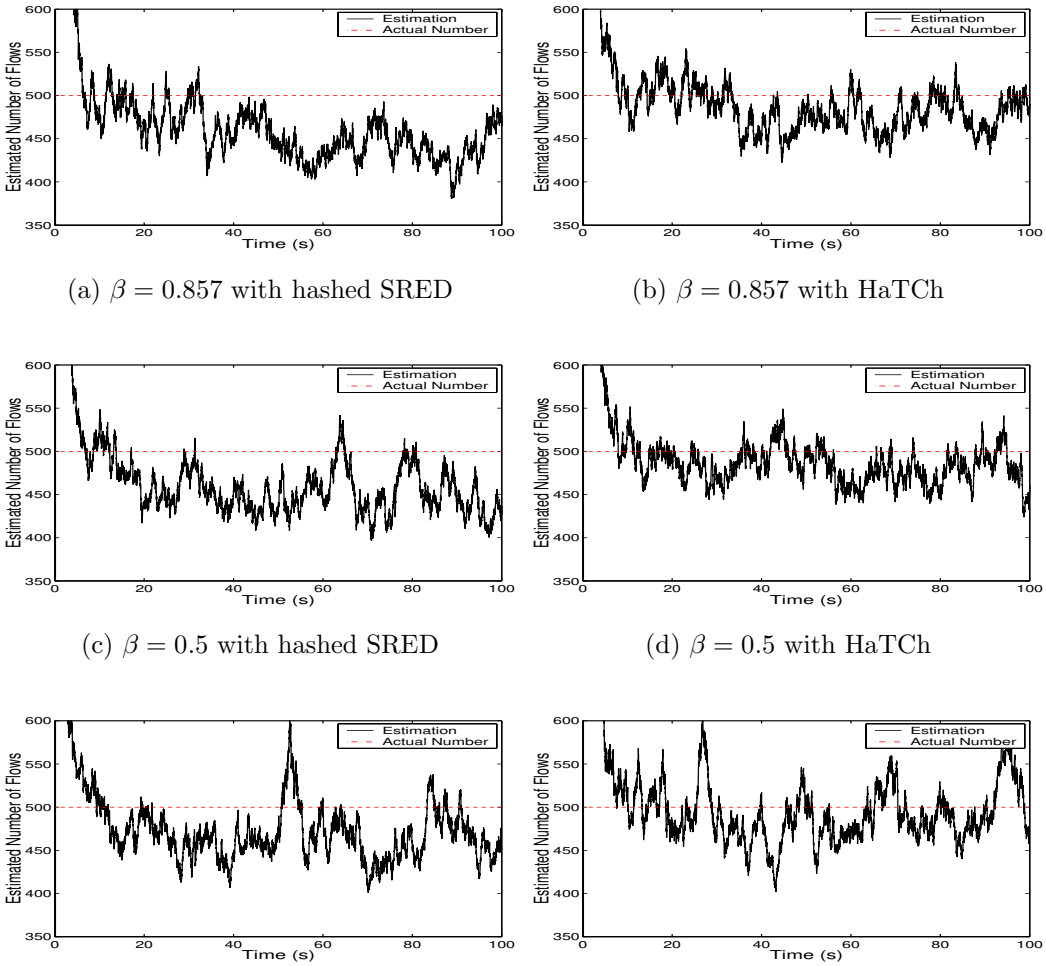


(d) Hash size = 20



(e) Hash size = 40

Fig. 2.7. Impact of hash size in the estimated number of flows ($r = 0.01$)



(a) $\beta = 0.857$ with hashed SRED

(b) $\beta = 0.857$ with HaTCh

(c) $\beta = 0.5$ with hashed SRED

(d) $\beta = 0.5$ with HaTCh

(e) $\beta = 0.167$ with hashed SRED

(f) $\beta = 0.167$ with HaTCh

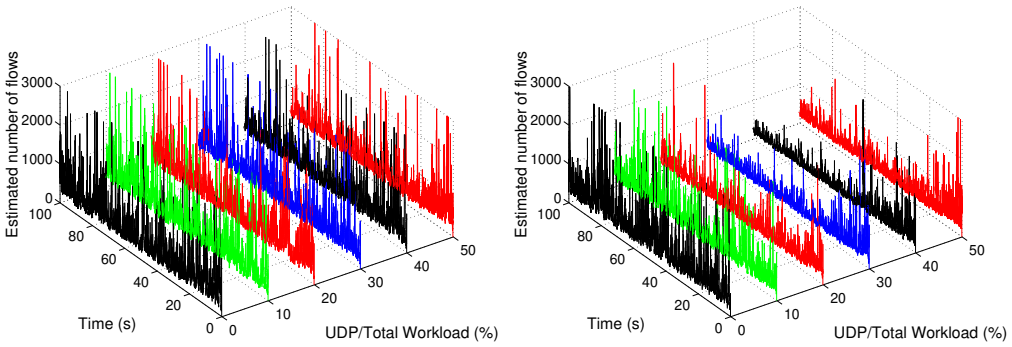
Fig. 2.8. Estimated number of flows when 500 Pareto On/Off sources are used.

parameter that affects the burst size as:

$$\beta = \frac{\bar{\epsilon}}{\bar{\epsilon} + \bar{\omega}}$$

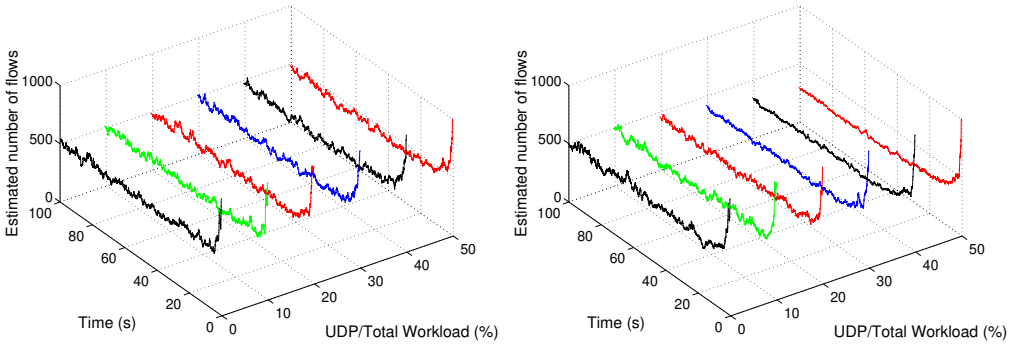
where, $\bar{\epsilon}$ and $\bar{\omega}$ are the average On and Off periods. We varied β from 0.167, which represents the maximum burst case to 0.857 (the minimum burst), and compared the performance of hashed SRED (by implementing hashing scheme in the *zombie list* of SRED) and HaTCh. Figure 2.8 shows the simulation results with $r = 0.01$. In both cases, the gap between the peak and nadir increases as the burst size increases. However, hashed SRED still slightly underestimated the number of flows, whereas HaTCh showed more accurate and stable estimation.

Impact of misbehaving flows : To demonstrate the effect of misbehaving flows on HaTCh performance, we performed simulations with different sending rates of misbehaving UDP flows. We also varied the UDP traffic workload from 0 to 50% of the total workload. Figure 2.9 shows the results of SRED estimation with 500 total flows. In Figure 2.9 (a), we set the sending rate of the misbehaving UDP flows (λ_{UDP}) as 2 times of the fair share of the link bandwidth (λ_{fair}). Although the estimated number of flows went down slightly, the fluctuation was not alleviated. The impact of misbehaving flows was more pronounced when we increased the UDP sending rate to 3 times of the fair share of the link bandwidth in Figure 2.9 (b). Here, SRED began to underestimate the number of flows as the UDP workload increased up to 30%, and the estimation recovered after the UDP flows became a dominant part of the traffic (40 and 50%).



(a) $\lambda_{UDP} = 2 \cdot \lambda_{fair}$ (b) $\lambda_{UDP} = 3 \cdot \lambda_{fair}$

Fig. 2.9. Impact of misbehaving flows in estimating the number of active flows with SRED



(a) $\lambda_{UDP} = 2 \cdot \lambda_{fair}$ (b) $\lambda_{UDP} = 3 \cdot \lambda_{fair}$

Fig. 2.10. Impact of misbehaving flows in the estimating the number of active flows with HaTCh

We repeated the same experiments using HaTCh. Note that we used different scaling factors in the graphs to show the results. Clearly in Figure 2.10 (a), HaTCh's estimation is remarkably stable, and the misbehaving flows were successfully isolated as expected. HaTCh showed the same performance up to 20% UDP workload in Figure 2.10 (b) as well. As the UDP workload increased, HaTCh also showed graceful performance degradation since a large number of excess packets from misbehaving flows could not be captured in the L1 cache. However, the estimation error was significantly reduced with HaTCh compared to SRED.

We also found that the performance of HaTCh was not affected significantly by the size of L1 cache as long as the number of L1 cache lines is greater than the number of misbehaving flows. The size of L1 and L2 cache lines can be determined based on the target number of misbehaving flows and the target number of active flows. In our experiments, we set the L1 cache size to 100 lines to control up to 100 misbehaving flows. This configuration provided the best performance for our simulation environment, where HaTCh effectively isolated up to 100 UDP flows (20% of total workload) in Figure 2.10 (b).

The impact of TCP sources with heterogeneous round trip times (RTTs) is an important factor that affects the performance of AQM schemes. TCP flows with short RTTs not only slow down the transmission rates quickly after a congestion notification (packet drop) of an AQM scheme, but also recover quickly compared to those with long RTTs. Therefore, an AQM scheme has to react differently for TCP sources with different RTTs to achieve fair share of bandwidth. However, flow estimators such as HaTCh have no control on transmission rates of traffic sources. Therefore, HaTCh performance on

TCP sources with heterogeneous RTTs can be predicted by investigating the impact of misbehaving flows. The simulation results with TCP sources that experience different RTTs (varied between 60ms to 540ms) exhibited the same trend as shown in Figure 2.9 and Figure 2.10.

Table 2.3. Impact of L_2 Cache Size and Hash Size

L2 Cache Size (M_2)	Hash Size (k)	Number of Flows per Subcache when $M_2 = N$	Ratio of L2 Cache Size to Number of Flows		
			1:1	1:2	1:4
200	2	100	0.116178	0.200499	0.295529
	4	50	0.084269	0.138910	0.268778
	8	25	0.061544	0.071426	0.256025
500	5	100	0.119863	0.189939	0.258613
	10	50	0.053280	0.102669	0.258538
	20	25	0.016502	0.087497	0.200555
1000	10	100	0.081638	0.124901	0.240245
	20	50	0.072930	0.093449	0.229931
	40	25	0.020743	0.077896	0.243097
2000	20	100	0.058761	0.106871	0.259732
	40	50	0.024954	0.090319	0.267330
	80	25	0.016792	0.083062	0.281995

Impact of HaTCh configuration parameters : Finally, we examine the impact of two major configuration parameters (the second level cache size M_2 and the hash size k) on the performance of HaTCh. Initially, we set the L2 cache size (M_2) to 200 lines, and increased it up to 2000 lines. We adjusted the the hash size (k) to make the number of active flows per subcache constant (100, 50, and 25 flows) for each L2 cache size when the L2 cache size is equal to the number of active flows N. Then we

increased N twice and four times that of M_2. In all these experiments the L_1 cache size was set at 10% of the L_2 cache size since that provided the optimal configuration.

Here, we summarize the performance of HaTCh by presenting the relative error of the estimated number of flows, defined as the ratio of the standard deviation (σ) to the mean (μ) of the estimated number of flows. In Table 2.3, the relative error was bounded within 11% regardless of the hash size when the L2 cache size and the number of flows were the same. Notably, HaTCh performs better as the number of flows (accordingly the L2 cache size is larger) increases, because a large number of flows multiplexed at the router reduces the impact of burst traffic. A large hash size (k) helped in the accurate estimation of N at the cost of little additional delay. As the numbers indicate, the relative error was about 6% when the the hash size k was 20, and was reduced to less than 2% for $k = 80$ when $N = 2000$. The trend also indicates that HaTCh can provide more accurate estimate of N if the number of flows is higher than 2000, which is likely to be true in many practical cases.

As the number of flows (N) became two to four times M_2, the accuracy of estimation suffered although the overall trend in the impact of N and k remained valid. For the 1:2 ratio, the error was limited to 8% when $N = 2000$ and $k = 80$. However, the relative error was high for the 1:4 ratio suggesting that number of flows should not exceed twice the size of the L_2 cache to limit the flow estimation error to 10%. Moreover, the impact of the hash size on the estimation accuracy was reduced because of more contending flows for the cache lines.

2.6 A HaTCh-based AQM Scheme (HRED)

This section presents a new AQM scheme, called HaTCh-based RED (HRED), which uses the HaTCh scheme to estimate the level of congestion, i.e, the number of active flows (N). HRED then deploy a random drop mechanism (at the enqueue point of packet memory) using N and the instantaneous queue length to regulate the packet injection rate.

2.6.1 Previous Work

We briefly reviews currently proposed Active Queue Management (AQM) schemes before describing the proposed scheme.

RED: The basic idea of RED [22] is to detect the incipient stage of a congestion,

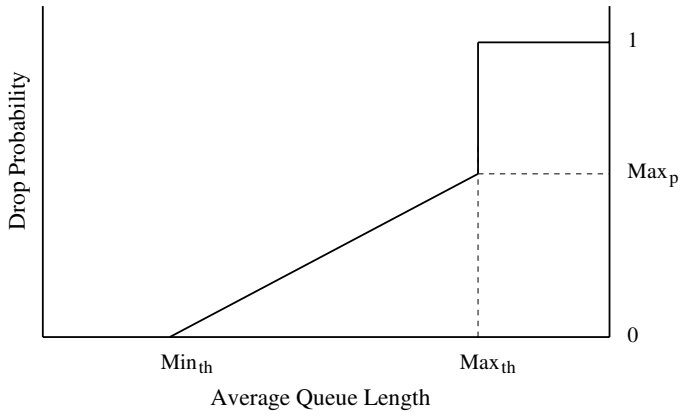


Fig. 2.11. The drop function of RED

and notify the sources to reduce the packet injection rate by deploying a random drop mechanism. RED calculates the Exponentially Weighted Moving Average (EWMA) on each packet arrival, and uses this number to find the drop rate. Figure 2.11 shows the drop function of RED. To find the drop probability, RED maintains parameters such as min_{th}, max_{th}, and max_p. When the average queue length is less than or equal to min_{th}, no packets are dropped, whereas all the packets are dropped when the average queue length exceeds max_{th}. Otherwise, the drop probability is determined by max_p, which is statically configured and is a linear function of the average queue length. Although RED queue outperforms the traditional drop-tail queue in terms of packet loss rate and preserving the burst flows, the performance of RED can easily degrade to that of the drop-tail queue due to the statically configured parameters. To address this problem, adaptive versions of RED (ARED) have been proposed [18] [21]. The main idea of the adaptive REDs is to adjust max_p dynamically based on the level of congestion (the average queue length). ARED increases max_p when the average queue length exceeds the max_{th}, and decreases it when the average queue length goes below the min_{th}.

Although the ARED scheme eliminates many problems of RED, it has the following two problems. First, ARED still relies on the average queue length to indicate the level of congestion. As noted in [19], the average queue length gives very little information about the level of congestion at a persistent queue. Second, the drop probability is decoupled from the current capacity of the target queue (instant queue length).

BLUE: W. Feng et al. [19] proposed a scheme, called Blue, to overcome the shortcomings of RED. The main idea of Blue is that it uses the packet loss rate and the link utilization to indicate the level of congestion rather than the average queue length.

With the Blue scheme, the target queue is monitored during short periods of time, and the packet drop rate is adjusted either at queue overflow or when a link is idle. Although Blue showed better performance especially in terms of packet loss rate compared to RED, it still suffers from severe queue oscillation with a small queue size, since it waits for the under or over utilization of the queue to adjust the drop rate.

SRED: Stabilized RED (SRED) [48] proposed by T. Ott et al. uses the number of active flows as an indication of the level of congestion. Under SRED, a rough range of the packet drop (or marking) probabilities, P_{sred}, is initially calculated based on the instantaneous queue length (q) (see Figure 2.12). If the queue length is greater than or equal to 1/3 of the queue capacity B, P_{sred} is set to P_{max}. Else, if the queue length is less than 1/6 of the queue capacity, then P_{sred} is set to $P_{max}/4$. Otherwise, P_{sred} is set to zero. The actual packet drop probability, P_{zap}, is then calculated as $P_{zap} = P_{sred} \times \min(1, 1/(N_{max} \times f(t))^2)$, where $1/f(t)$ is the estimated number of active flows using the packet memory at time t. The above expression implies that if the estimated number of flows $1/f(t)$ is greater than N_{max}, then the drop probability is only determined by the instantaneous queue length. Otherwise, the drop probability is set to $P_{sred} \times (N/N_{max})^2$. In [48], P_{max} was set to 0.15 and N_{max} was 256.

Although SRED generally shows fine control on the queue occupancy for a certain range of congestion, we observed several shortcomings in the SRED drop mechanism through extensive simulation. First, the buffer utilization is very low since the queue length converges to $B/6$. Second, finding optimal configuration parameters such as P_{max} and N_{max} for different workloads and network environments are very difficult. Third,

SRED degrades to Tail-Drop when the severity of congestion exceeds the level that can be controlled by the maximum drop probability (P_{max}).

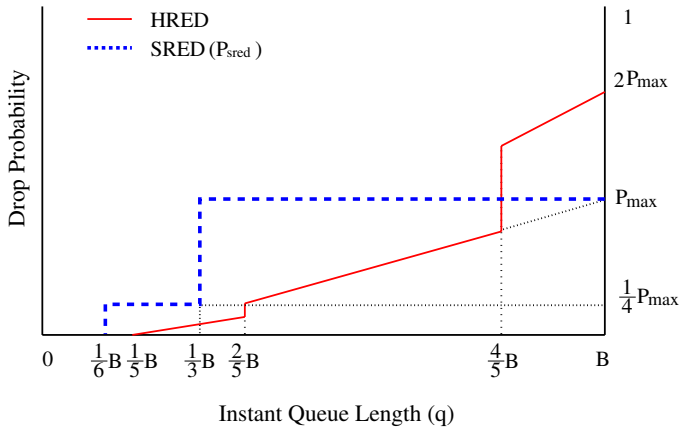


Fig. 2.12. The drop functions of the SRED and the HRED schemes

2.6.2 The Proposed HaTCh-based RED (HRED) Scheme

In this section, we propose a new AQM scheme based on the HaTCh concept. First, we examine the impact of a drop function on the packet loss rate before proposing the new AQM scheme.

2.6.2.1 Impact of a Packet Drop/Marking Function

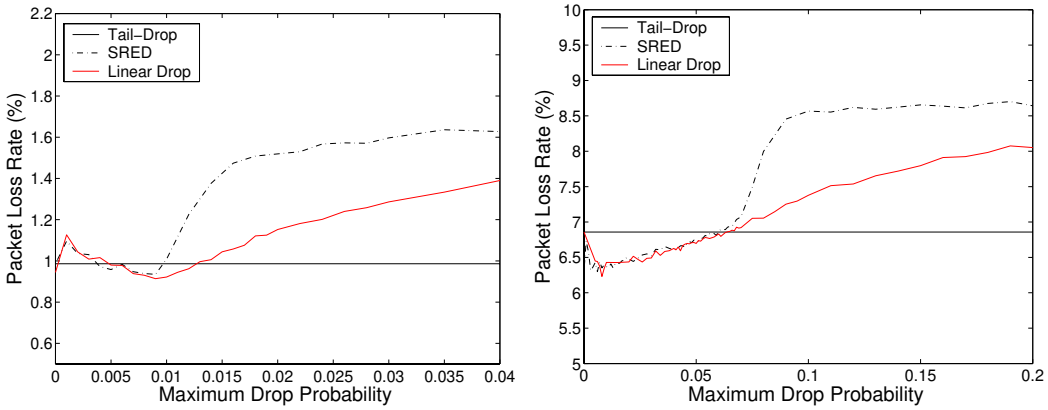
The role of the "drop function" in an AQM schemes is to find the drop probability based on the estimated level of congestion. Drop functions of AQM schemes can be

classified into two categories. One is a flat or step type functions used in SRED and BLUE, and the other is a linear type function used in RED. In order to examine the impact of the drop functions on the performance of an AQM scheme, we performed the following simulation using the similar environment described in the previous section. Figure 2.13 shows the loss rate with a drop-tail queue, and queues with step type and liner type drop functions for different values of maximum drop probabilities. The experiments are conducted with 100 and 500 TCP connections. The simulation results show that the random drop mechanism outperformed the drop-tail queue only in a small range of maximum drop probabilities. However, the loss rate of the queue with a linear drop function is less than that of the queue with step type function, and the range of the maximum drop probability that leads to performance gain is wider. Moreover, the packet loss rate of the step type function seriously increases when the maximum drop probability is aggressively configured whereas the linear drop function shows graceful degradation.

2.6.2.2 HRED Drop Mechanism

Based on the discussion in the previous subsection, we present a new drop mechanism. The overall structure of the proposed drop mechanism is similar to that of RED. The main differences are that i) the proposed scheme uses the number of active flows to indicate the severity of congestion, whereas RED uses the average queue length[2], and

[2]An averaging of the queue length is conducted using a typical first-order autoregressive mechanism. Averaging reduces the false-positive congestion-detection probability at the expense of a higher missed-detection probability.



(a) 100 TCP connections

(b) 500 TCP connections

Fig. 2.13. Impact of Drop Function

ii) the proposed drop mechanism also considers the instantaneous queue length to stabilize queue occupancy as well as minimize the packet loss rate. (Note that SRED also used an estimated number of active flows and the instantaneous queue length in finding the drop probability. However, SRED's performance showed severe dependency on the configuration parameters due to the inefficiency of the drop function.)

To overcome the shortcomings of SRED, we propose a new packet drop mechanism, which uses a piece-wise linear function of the instantaneous queue length, q as depicted in Figure 2.12. This design of the drop function helps in steering the instantaneous queue length to the target range and in improving the buffer utilization. In particular, if the buffer is empty or almost empty ($q < B/5$), no packet is dropped. If the buffer is underutilized ($B/5 \leq q < 2B/5$), HRED reduces the maximum drop probability to half of its original value, i.e., $P_{\max} \rightarrow P_{max}/2$. If q begins to exceed the target range ($4B/5 \leq q$), which is assumed to imply imminent buffer overflow, HRED doubles the drop probability ($P_{max} \rightarrow 2P_{max}$).

2.6.2.3 Finding the severity of congestion (max_p)

Here, we complete the HRED design by showing how the estimated number of active flows is used to determine P_{max}. Assuming that all the flows are TCP based, several AQM schemes [48] [28] relate the number of active flows in finding the maximum drop probability. These techniques merely relied on a simple relationship, called the square-root law, more precisely $BW = \frac{1}{RTT}\sqrt{\frac{3}{2p}}$ [49]. Here, BW is the throughput of a single TCP flow, RTT is the round-trip time, and p is the packet loss rate of the flow. However, the above equation only incorporats the packet loss rate from *triple*

duplicate ACKs without considering the TCP retransmission timeouts. This is a simpler solution, but can result in over-estimation of throughput at a severely congested router. Therefore, we use the throughput equation with retransmission timeout [49], given by $BW = \frac{1}{RTT\sqrt{\frac{2p}{3}}+RTO\min(1,3\sqrt{\frac{3p}{8}})p(1+32p^2)}$, to find the maximum drop probability as follows.

$$C = \frac{N}{RTT\sqrt{\frac{2p}{3}} + RTO\min(1, 3\sqrt{\frac{3p}{8}})p(1 + 32p^2)}, \tag{2.15}$$

where C is the link bandwidth, RTO is the retransmission timeout. Given C and N, we use Newton's method to find the loss rate p and, in turn, use this quantity to indicate the maximum drop probability P_{max}. Clearly, we assume that good estimates of average RTT and RTO are available at a router for this computation. We believe that the complexity and the overhead of the computation may not be a serious concern since it is not computed per-packet.

Finally, the HRED scheme is summarized as follows:

- Upon each packet arrival, the HaTCh scheme estimates the number of active flows, N.
- The maximum drop probability (P_{max}) is periodically calculated based on N using equation (2.15), which needs the number of active flows (N).
- An arriving packet is dropped with a probability calculated from the maximum drop probability, P_{max}, and the instantaneous queue length according to the proposed drop mechanism given in Figure 2.12.

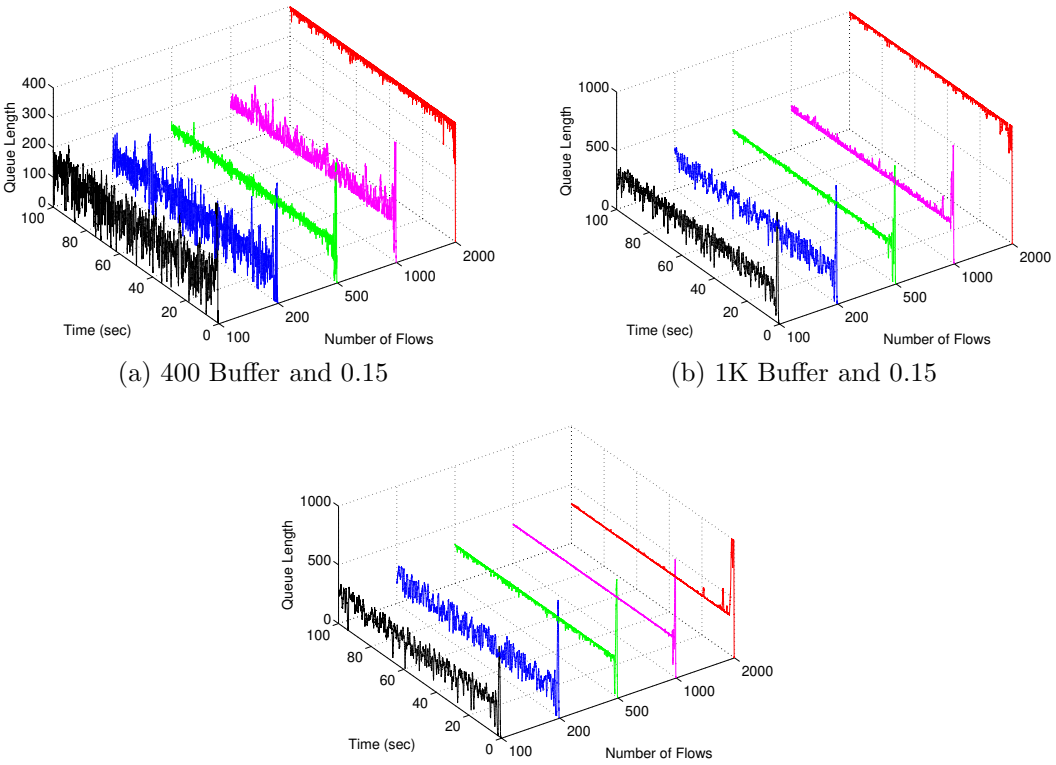
The next subsection presents performance analysis of the proposed HRED scheme.

2.6.3 Performance Evaluation

We performed a variety of simulations by varying the number of connections (from 20 to 2000), buffer size (400, 600, or 1000 packets), and link capacity (5Mbs or 45Mbs). We monitored the instantaneous queue length and the packet loss rate, which are the most commonly used metrics in evaluating the performance of AQM schemes. We first analyzed the simulation results of SRED and ARED, and then compared them with the proposed scheme, HRED.

Although the drop probability of SRED is adaptively determined according to the level of congestion (the number of active flows), its performance heavily relies on its two configuration parameters, P_{max} and N_{max}. In Figures 2.14(a) and (b), for a 45Mbs link and $P_{max} = 0.15$, SRED degraded to a Tail-Drop mechanism when there are a large number of flows. For a large number of flows, SRED showed better control on the queue occupancy with a larger P_{max} ($= 0.25$) as in Figure 2.14(c). However, this configuration resulted in poor buffer utilization for the samll number of flows (100 to 200). For a small number of flows (100 to 200), the SRED queue suffered from oscillation when the buffer size was also small (400 packets) as shown in Figure 2.14 (a). Furthermore, SRED exhibited poor buffer utilization in most of the experiments.

Next, we changed the capacity of the bottleneck link to 5Mbs; Figure 2.15(a) depicts how SRED attempts to keep the buffer full, a behavior noted in [25]. Assuming that the maximum number of flows is limited to 256 ($N_{max} = 256$) for a 45Mbs link, the average throughput that each flow can get is 175.8Kbs, but this number decreases to 19.5Kbs for a 5Mbs link when the same maximum number of flows is used. Therefore,

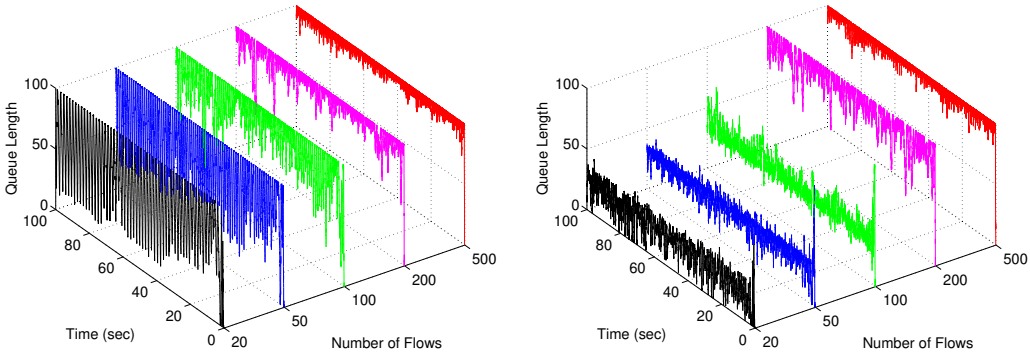


(a) 400 Buffer and 0.15

(b) 1K Buffer and 0.15

(c) 1K Buffer and 0.25

Fig. 2.14. Impact of buffer size and P_{max} under SRED at 45Mb Link



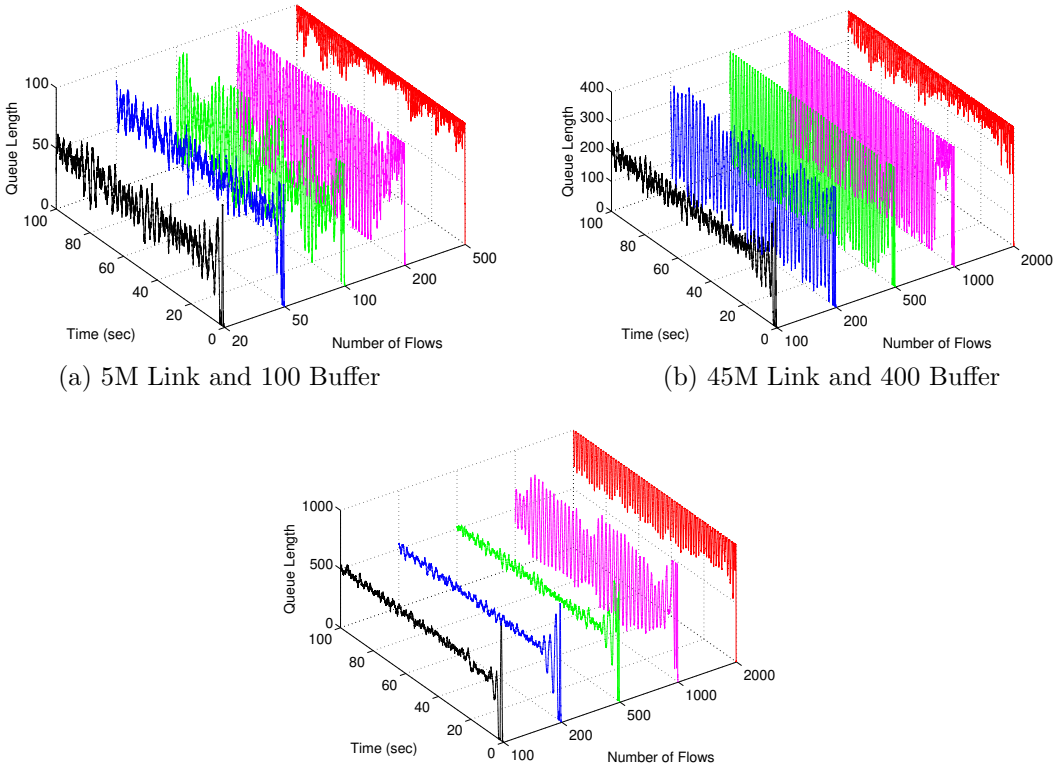
(a) 5M Link, $N_{max} = 256$ (b) 5M Link, $N_{max} = 25$

Fig. 2.15. Impact of N_{max} with SRED

the drop probability, used in SRED based on $N_{max} = 256$, is too conservative. As a result, the SRED queue again suffered severe oscillation. With a smaller value of N_{max} (=25) (targeting approximately 175.8kbs of average throughput), SRED showed better performance in Figure 2.15(b), but the queue occupancy behavior with high workloads remained similar to that of the same queue under a Tail-Drop mechanism.

In summary, SRED used an unambiguous indicator (N) for the severity of congestion, and showed some level of adaptability in reacting to congestion. Nevertheless, its performance degraded due to the inaccuracy in estimating N and inefficient design of the packet dropping function.

The key idea of ARED is to adaptively change the maximum drop probability according to the target average queue length. However, finding an optimal value of the maximum drop probability is difficult in ARED. In Figure 2.16(a) and (b), the ARED queue suffered from severe oscillation and degraded to a Tail-Drop queue as the number



(a) 5M Link and 100 Buffer

(b) 45M Link and 400 Buffer

(c) 45M Link and 1K Buffer

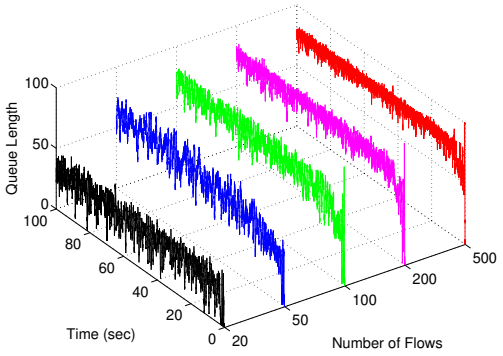
Fig. 2.16. Instantaneous Queue Length under ARED

of flows increased. As with the original RED mechanism, ARED also performed better with a large buffer size as shown in Figure 2.16(c), but the queue occupancy again became unstable as the workload increased. On the other hand, HRED showed fine control of the queue occupancy regardless of the change in workloads, link speed, and buffer size as depicted in Figure 2.17.

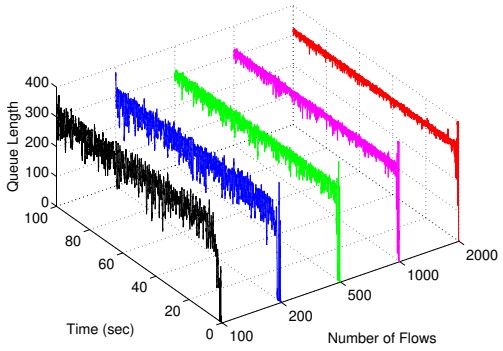
In Figure 2.18, the packet loss rates are plotted for different buffer sizes (400 and 1000 packets) under three AQM schemes (ARED, SRED and HRED). Most of the simulation results demonstrated that HRED outperforms the other two AQM schemes regardless of the buffer size. Note that the packet loss rate of ARED is smaller than that of HRED for the 1000 flows case in Figure 2.18(a). In this case, severe oscillation of the ARED queue (Figure 2.16(a)) resulted in poor link utilization with a lower packet loss rate. Although it is believed that the performance of ARED can improve with a large buffer size, the proposed HRED scheme outperformed the other two schemes in terms of packet loss rate and queue occupancy behavior even with a large buffer as shown in Figure 2.18(b).

2.7 Concluding Remarks

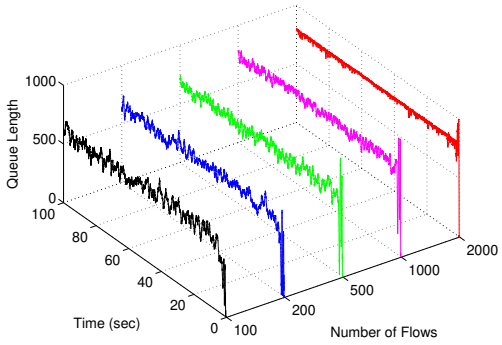
Design of AQM schemes has been an active area of research in the Internet community to minimize network congestion, and thus, improve performance. However, the complexity of the Internet traffic dynamics has eluded researchers in finding an efficient solution. Recently, SRED [48] was proposed to provide better congestion control by using the number of active flows as an indicator of congestion. It was shown through



(a) 5M Link and 100 Buffer

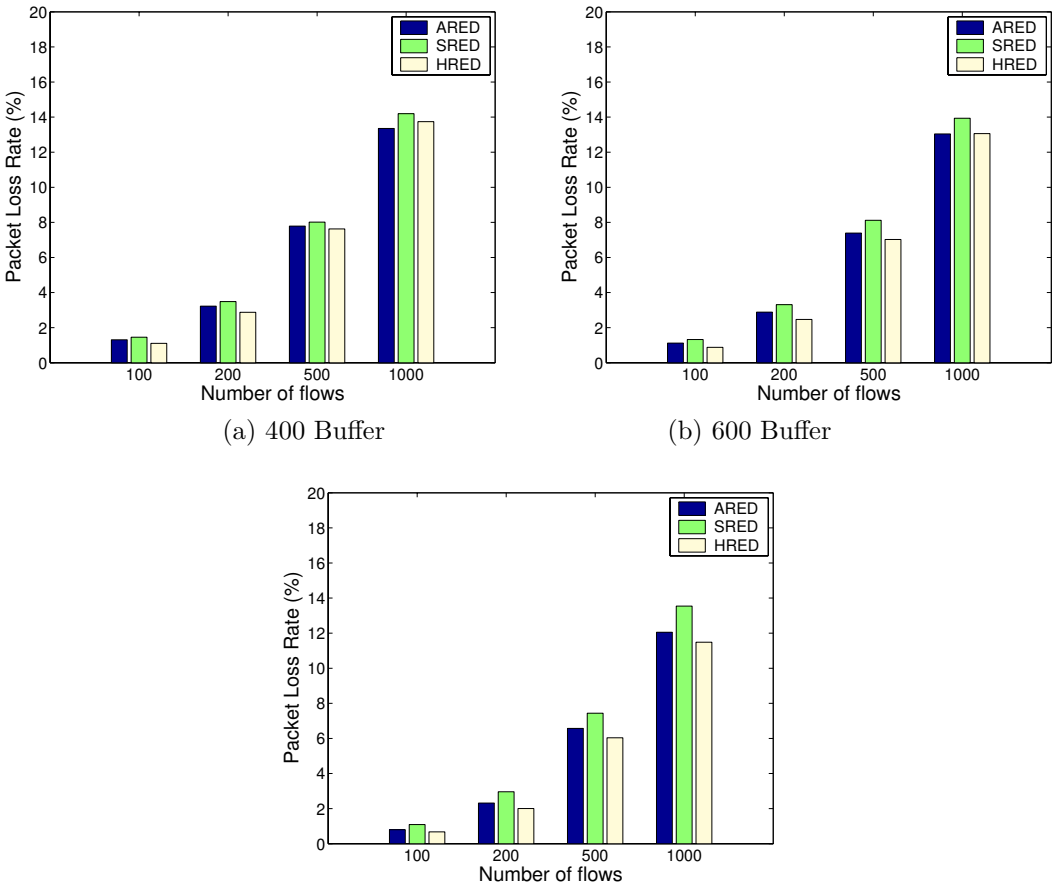


(b) 45M Link and 400 Buffer



(c) 45M Link and 1K Buffer

Fig. 2.17. Instantaneous Queue Length under HRED



(a) 400 Buffer

(b) 600 Buffer

(c) 1K Buffer

Fig. 2.18. Packet Loss Rate under Different AQM Schemes

a simulation study that an accurate estimation of the number of active flows is a very useful indication of the level of congestion.

In this chapter, we have presented a mathematical framework for analyzing the estimation behavior of SRED. The model showed that the steady-state hit frequency of the SRED cache model can be used to estimate the number of active flows. The model is accurate enough to capture the effect of misbehaving flows in the estimation. In order to alleviate the drawbacks of SRED, we have proposed a modified SRED, called HaTCh, that uses hashing and a two-level caching mechanism. A preliminary model of the proposed scheme was presented to demonstrate its effectiveness. Also, we conducted extensive simulation for an in-depth performance analysis of both the schemes. It was observed that the proposed HaTCh scheme improves not only the estimation accuracy and stability compared to SRED, but also improves the robustness of the estimation by effectively isolating the misbehaving flows to avoid cache contamination.

The HaTCh scheme is practically viable since the two caches can be implemented using standard hardware. A standard configuration of HaTCh could be the following: (i) L2 cache size = the target number of flows supported by the link (N); (ii) L1 cache size = 10% of L2 cache size or the target number of misbehaving flows; (iii) Replacement probability (r) = 0.01, and (iv) Keep the hash size k as large as possible based on the implementation complexity.

We, then, extended HaTCh into a complete AQM scheme, called HaTCh-based RED (HRED), which uses the number of active flows (N) to indicate the level of congestion. HRED effectively minimizes the dependency of configuration parameters through

a new dropping mechanism. As a result, HRED can provide a much stable queue occupancy and low loss rate compared to SRED and ARED.

Our future work involves in-depth performance study or HRED with more realistic workloads. Also, the model needs fine-tuning to include other system/design parameters.

Chapter 3

Controlling Unresponsive Flows

3.1 Introduction

Proliferation of unresponsive flows is becoming a major concern for providing at least a satisfactory level of service to millions of users who rely on the Internet for their daily business. Recent traffic analysis conducted by CAIDA (Cooperative Association for Internet Data Analysis) shows that unresponsive flows contribute to as high as 70-80% of the overall Internet traffic, although the byte volume is limited to about 20% [2]. Most of these unresponsive flows are UDP applications, which unlike their TCP counter parts, do not respond to network congestion. Thus, UDP flows can effectively shut out the responsive TCP flows by occupying almost the entire bandwidth and can ultimately lead to *congestion collapse* [32].

Although the end-to-end TCP congestion control mechanism [57] and an Active Queue Management (AQM) scheme (such as RED, SRED, and AVQ [22][48][36]) can help in preventing global synchronization and minimizing packet loss, the effectiveness of these schemes still heavily relies on the voluntary use of the congestion control mechanism by the end-users. Since UDP flows naturally do not respond to the network status, TCP congestion control is quite vulnerable in their presence.

Recently, several studies have analyzed the impact of unresponsive UDP flows on network performance [41][30][23], and argued the need for a router mechanism that

detects and penalizes them. In practice, network processors on the ingress line cards of Internet routers can easily discriminate between TCP and UDP packets without causing extra packet processing overhead. Furthermore, separate queues could be allocated for TCP data packets and TCP control packets. Thus, UDP flows can be effectively isolated from the TCP flows in packet memory by diverting them to a small dedicated queue as discussed in [12]. This technique provides a partial remedy for the bandwidth starvation of TCP flows, and also can reduce the delay of real-time traffic over UDP.

However, there is another class of misbehaving flows, called *unresponsive TCP sessions*, which can be interpreted as a kind of Denial-of-Service (DoS) attack on "honest" flows that employ the standard TCP congestion control mechanisms. In [58][15], it is demonstrated that malicious users at the TCP receiver side can exploit the vulnerabilities in TCP congestion control to obtain better service or to initiate DoS attack. The authors proposed a modified congestion control scheme to handle these users within the TCP framework. On the other hand, such activity at the sender side has more serious impact on the network security. For example, malicious users can easily compromise TCP's communal congestion control mechanism by deactivating the slow-start and retransmission timeout (RTO) mechanisms in their TCP/IP stack. Modifying the TCP source code to deactivate these functionalities, for example, in an open-source Linux context, is clearly not difficult. In addition, greedy users can launch multiple parallel TCP sessions to reduce their download time, a technique known as "turbo-TCP". In fact, a number of applications such as FlashGet, GolZilla, ReGet and Download Accelerator [39] bypass the TCP congestion control scheme and open multiple connections per

object [61]. We believe that such activities are likely to spread and will pose a serious security concern for the Internet.

Relatively little work has been targeted directly at ensuring that TCP flows conform to the assumed protocols. Although this problem was understood, the AQM techniques often exacerbated it by allowing a nonconformant TCP session to obtain even more than its fair share of available resources. Recent research, therefore, has been directed towards game theoretic modeling of end-system flow control [34][35][8][61]. In this context, a user is typically *expected* to act in a greed fashion (interpreted as a nonconformant in an AQM context); the network, in turn, uses a pricing mechanism to control user behavior. Although it seems promising, the pricing and billing technology is currently in its infancy in the Internet context.

In this chapter, we propose a scalable *policing* technique, which is deployable in network buffers, to detect and control a smaller, but potentially significant, number of nonconformant TCP flows for ensuring fair bandwidth sharing to conformant TCP sessions. Although the proposed scheme can be used for controlling all types of high bandwidth flows, we mainly restrict our discussion to malicious TCP flows here. We propose a comprehensive solution that involves sampling, detection, and punitive measures for such flows. The proposed scheme, called HaDQ (HaTCh-based Dynamic Quarantine), builds upon our proposed HaTCh technique [73], which can provide an accurate and robust estimation of the number of active flows compared to the single cache-based SRED mechanism [48]. To estimate the number of active TCP flows through a packet memory in an Internet router, the HaTCh scheme uses a small L_1 cache that primarily captures

the aggressive flows and a larger L_2 cache that monitors the normal flows for estimating the "hit" frequency and subsequently the number of flows.

Dynamic quarantine requires sampling of the aggressive flows to measure their arrival rates. Here, we exploit the advantage of using the L_1 cache since it isolates the aggressive flows from the L_2 cache. Therefore, any incoming flow that results in a high hit count in the L_1 cache is a possible "attacker" and is quarantined in a separate small Content Addressable Memory (CAM), called quarantine memory. The quarantined flows are monitored for a certain duration (monitoring period) to compute the drop probability for each flow based on the measured arrival rate and the fair share of the bandwidth provided by HaTCh. All other flows are directed to the L_1 cache of the HaTCh scheme to identify possible attackers. To avoid false positives, a flow is released from the quarantine memory if its drop probability is less than the drop probability of the underlying queuing policy. However, the proposed HaDQ scheme is independent of the underlying queue management scheme since it does not require an AQM scheme for quarantining and punitive actions. It can work even with a simple Tail-Drop mechanism. However, since most prior research have used AQM-based approach to handle aggressive UDP flows, we use a similar technique in this research. In this context, we use HRED, presented in the previous chapter, as an underlying AQM scheme since it showed better performance in handling network congestion.

We simulated our proposed scheme along with two prior schemes (CHOKe [50] and FRED[38]), targeted to handle unresponsive flows, by injecting a certain number of misbehaving TCP flows, which deactivated their congestion control and retransmission timeout mechanisms. The simulation results indicate that HaDQ is very effective in

penalizing the unresponsive flows. In fact, it provided almost equal bandwidth to the conforming and aggressive TCP flows, while CHOKe and FRED were less effective in controlling unresponsive sessions. Furthermore, analysis of the proposed scheme as a function of the monitoring period and the detection threshold indicates that, with a proper configuration, the false positive probability can be kept very low (less than 0.1%). We believe that HaDQ is a scalable technique in that per-flow states are only required for the small number of quarantined flows, which in turn, are minimized by the accurate HaTCh-based sampling technique.

The rest of this chapter is organized as follows: We demonstrate the impact of unresponsive TCP flows, and briefly discuss related solutions in Section 3.2. The proposed dynamic quarantine scheme, HaDQ, is detailed in Section 3.3. In Section 3.4, simulation results are presented followed by the concluding remarks in Section 3.5.

3.2 Motivation

In this section, we analyze the impact of unresponsive TCP flows on the conforming flows, and summarize previous solutions for handling unresponsive flows.

3.2.1 Impact of Unresponsive TCP flows

In order to demonstrate the impact of unresponsive TCP flows on the conforming TCP connections, we performed a number of simulations using ns2 [3] for the network shown in Figure 3.1. In the simulations, all the connection requests are generated at the leftmost nodes and terminate at the rightmost nodes except for the acknowledgments, and all the sources randomly initiate packet transmission using FTP over TCP reno [43]

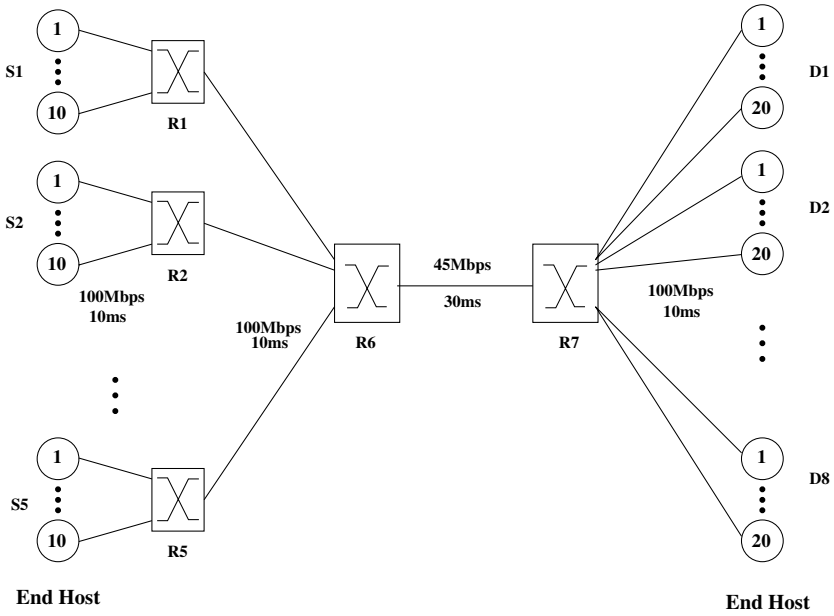


Fig. 3.1. The network topology used for simulation

between 0 to 2s. We injected a single unresponsive TCP flow that can operate in two modes. In the first mode, we deactivated the window control mechanism, and in the second mode, we deactivated the window control and kept the retransmission timeout (RTO) very low (0.1ms) to mimic an aggressive flow. Therefore, the second case is more aggressive than the first case. We measured the average throughput of the conforming flows and the unresponsive TCP flow under two different queue management policies (Tail-Drop and Adaptive RED (ARED) [21]) at R1.

Table 3.1 summarize the simulation results. Two important observations from theses results are the following: First, RTO has a more serious impact on bandwidth sharing than the window control (WC) mechanism in TCP. The bandwidth consumption of the malicious flow increased considerably when both the window control and RTO mechanisms were tampered with. Unlike UDP flooding, the unresponsive TCP flow occupied a significant portion of the bandwidth both at light load (100 flows case: 80% of total bandwidth) as well as at heavy load (1000 flows case: 38% of total bandwidth).

Second, the Tail-Drop (TD) scheme provided better protection to the conforming flows than ARED. This is consistent with the conclusion reported in [41]. This is because the unresponsive flow quickly fills up the buffer, and thus, the available buffer size for the arriving TCP flows becomes small. Therefore, most TCP flows lose their packets due to insufficient buffer size, and suffer from high packet loss rate. In addition, unlike Tail-Drop, the high average queue length will force ARED to activate the random drop mechanism, which in turn, will lead conforming flows to bandwidth starvation. As a result, ARED exhibits poor protection for conforming TCP users against malicious users.

In summary, unresponsive TCP flows can cause a serious DoS attack against the conforming TCP users. A standard AQM scheme like ARED is not adequate to provide protection against such flows. Therefore, in the next subsection, we summarize the handful of existing techniques that have been proposed to protect honest users.

3.2.2 Previous Work

The prior research for handling unresponsive UDP flows can be classified into two types as summarized below.

1) AQM based techniques

Here, we summarize four schemes: RED-PD [40], FRED [38], CHOKe [50], and SFB [19]. These techniques use an AQM scheme to penalize the unresponsive flows to ensure fair share of bandwidth.

The main idea of RED-PD [40] is to use the property of RED that the amount of packet loss for each flow is proportional to its bandwidth share. In this approach, M separate lists of recently dropped packets are maintained, and a flow is identified as a high bandwidth flow when it loses packets from K out of the total M lists. Measuring only a subset of the traffic is feasible and may be effective for detecting high-bandwidth flows. Maintaining the history of recently dropped packets can, however, be costly during periods of congestion since most of the active flows lose packets at a congested router.

The detection mechanism of FRED [38] is similar to RED-PD in that it maintains partial flow information. However, FRED collects bandwidth sharing information based on the currently queued packets. With FRED, the average per-flow queue length is calculated and is used to identify and penalize high bandwidth flows. As a result, FRED

Table 3.1. The Impact of a Single Unresponsive TCP flow

Traffic Mix	Throughput with Tail-Drop (Mbs)	Throughput with ARED (Mbs)
100 Standard TCPs	0.433	0.431
1 TCP w/o WC	1.581	1.842
100 Standard TCPs	0.367	0.091
1 TCP w/o WC and RTO	8.194	35.824
500 Standard TCPs	0.087	0.087
1 TCP w/o WC	1.342	1.147
500 Standard TCPs	0.078	0.039
1 TCP w/o WC and RTO	5.620	25.138
1000 Standard TCPs	0.044	0.044
1 TCP w/o WC	0.271	0.790
1000 Standard TCPs	0.041	0.027
1 TCP w/o WC and RTO	3.502	16.989

* WC : Window Control, RTO : Retransmission Timeout
* Throughput for Standard TCP is the average value

requires a larger queue to make the detection mechanism work properly. Moreover, the buffer usage often does not capture bandwidth sharing accurately.

In CHOKe [50], an arriving packet is compared with a randomly selected packet in the queue, and both packets are dropped when they belong to the same flow. Otherwise, the arriving packet is dropped with a probability determined by RED. Although CHOKe is relatively simple to implement, its level of protection (or the degrees of fairness) is coarse, and packet loss from conforming flows is unavoidable.

A kind of Bloom filter mechanism, called SFB [19], was proposed using multiple hash functions to detect and penalize unresponsive flows without maintaining per-flow states. Although SFB is a scalable mechanism, finding good configuration parameters for SFB is not easy. SFB uses X levels of hash functions with a bin size of L. It allocates buffer for each hash bin, e.g., $1.5B/L$, and uses this quantity as the threshold value for packet dropping. (Here, B denotes the buffer size.) Even with a large number of levels X, most of the bins will suffer from high packet drop rates with a small bin size when number of flows is large. Therefore, a large value for L is desirable but, in this case, a good threshold value is difficult to ascertain since B/L is very small. In addition, the bandwidth consumed by unresponsive flows is controlled by a statically configured parameter regardless of the number of actively competing flows.

2) Network Security based technique

Gil et al. proposed a data-structure, called Multi-Level Tree for Online Packet Statistics (MULTOPS), as a "bandwidth attack" detection mechanism [27]. To detect high bandwidth flows or bandwidth attackers, MULTOPS maintains packet rate statistics using a 4-level tree. For example, MULTOPS keeps the destination address of the

packets going in a forward direction, and keeps the source address of the packets going in the reverse direction. If the ratio of forward packets with an IP address prefix x to the reverse packets with the same address prefix goes below the minimum threshold or exceeds the maximum threshold, the subnet with the prefix x becomes a candidate for an attacker-victim pair. This approach is based on the assumption of symmetric traffic. However, today's Internet traffic is often asymmetric with extremely wide variation as shown in [53]. To maintain a low false positive rate, MULTOPS needs to incorporate precise traffic statistics for a specific location, which is difficult to achieve in practice.

In this chapter, we consider solutions for unresponsive TCP flows. In summary, all prior techniques may work in specific scenarios, but the performance of these schemes can be seriously limited by the inefficient/inaccurate detection mechanism or parameter dependency. Moreover, the effectiveness of these schemes for handling unresponsive TCP flows is not known.

3.3 The HaTCh-based Dynamic Quarantine Scheme (HaDQ)

This section presents the proposed policing scheme, called HaTCh-based Dynamic Quarantine (HaDQ), which dynamically quarantines and penalizes the unresponsive TCP flows based on HaTCh concept. The detail description on the HaTCh scheme is presented in Section 2. The process of mitigating the effect of unresponsive TCP flows can be divided into three parts: sampling, detection, and punitive measures. Several practical limitations make it difficult to find efficient solutions for these tasks. First, to accurately identify and penalize an attack, maintaining a per-flow state is impractical considering the extremely high volume of TCP sessions at any given point and time in

today's Internet. Second, identification of the misbehaving flows should be based on the fair sharing of the available transmission bandwidth to prevent the malicious users, who exploit a statically configured parameter. Third, a detection mechanism should provide protection for the wrongly detected flows (false positives). Finally, punitive measures should be conducted in real-time to protect the network and all the active users at the time of attack.

We now describe the HaDQ scheme in such terms: sampling, detection, and punitive measures.

3.3.1 Sampling

The first step in detecting an attacker is to monitor the arrival rate of each candidate. One of the simplest solutions is to deploy a classical random sampling scheme as used in [17]. With this scheme, incoming packets are randomly sampled with a low probability and the sampled flows are more carefully studied using a separate memory over a certain period of time. Although random sampling is simple, accurate detection may require longer time when a small memory is used to sample from a large number of competing flows.

To improve the sampling accuracy, we use the hit count of the *smaller* L_1 cache of HaTCh. The purpose of the L_1 cache (a RAM) is to protect the much larger L_2 cache (also a RAM) from the aggressive flows. Indeed, any flow that results in a high hit count in the L_1 cache is a suspect and a good candidate for more careful observation. Once a flow is sampled and classified as a possible unresponsive flow, the flow is quarantined from the HaTCh RAM and its session identifier is registered in a separate Content Addressable

Memory (CAM), called the quarantine memory. Note that, although we maintain per-flow state in the quarantine memory, the size of the memory is very small compared to the number of active flows (N). We assume that the size of the quarantine memory is comparable to the size of the L_1 cache since the L_1 cache size is configured according to the target number of aggressive flows, which is about 10% of the target number of flows. Considering today's lowest Internet access rate of 56Kbs, we used $C/56000$ as a guideline for the target number of flows, where C represents the link bandwidth.

To control the number of sampled (quarantined) flows in the quarantine memory, the sampling threshold (θ_{smpl}) for the hit count in the L_1 cache is periodically adjusted according to the number of flows registered in the quarantine memory and the target number of quarantined flows. Here, the target number of quarantined flows has a lower bound (NQ_{low}) and an upper bound (NQ_{up}). The quarantine memory needs to be filled, but not too full, to maintain a proper sampling sensitivity. Therefore, the sampling threshold (θ_{smpl}) is decremented by 1 when the number of sampled flows is less than the lower bound (NQ_{low}) of the target number of flows, and it is incremented by 1 when the number of sampled flows is greater than the upper bound (NQ_{up}). If NQ_{low} is too small, the quarantine memory can be easily filled with false positive conforming flows, and if it is too large, an attacker may not be sampled.

When the quarantine memory becomes full, we assume that the sampling process (by which new flows could be quarantined) is turned off. This is a simple yet efficient mechanism to control the sampling sensitivity, but false positive conforming flows can block real attackers from being sampled when the quarantine memory is full. Thus, there

is a need to quickly "exonerate" or time-out mistakenly quarantined flows[1]. Instead of deactivating sampling when the quarantine memory is full, we could sort the session identifiers in the quarantine memory according to their estimated arrival rates. Thus, a newly sampled flow with a high hit count could *push out* a flow with the *lowest* arrival rate. This solution can effectively avoid the quarantine memory overflow problem, but sorting adds significantly more implementation complexity to the HaDQ design. Note that "turbo-TCP" can be easily handled with HaDQ since all the sessions share the same session identifier (source-destination address pair).

3.3.2 Detection

There are three factors that should be considered in detecting unresponsive flows. First, to accurately measure the arrival rate of quarantined flows, which may include highly bursty but conforming (mis-quarantined) flows, the traffic monitoring period should be large enough to cover multiple round-trip times. A larger monitoring period would, however, increase the system response time to unresponsive flows. Second, detection of an aggressive flow should be based on the number of active flows as this quantity obviously determines the fair share of available bandwidth. Third, errors in HaTCh's estimation of the number of active flows should be considered. Although HaTCh is a more robust estimator than SRED, its estimate also exhibits oscillation [73]. Therefore,

[1] Also, separately queuing TCP control packets will reduce the total number of session identifiers under consideration for the TCP data queue, which is especially helpful if, e.g., the link is a conduit for a TCP SYN attack on a server during which time many spurious session identifiers will be transmitted.

we use certain multiples of C/N as the detection threshold, θ_{det}, in identifying the unresponsive flows to reduce the false positives. Here, C is the link bandwidth and N is the estimated number of flows.

A flow registered in the quarantine memory is isolated from the process that computes N (HaTCh). The quarantined flows are then monitored for a period of time, t_{mon}, to measure a flow's actual arrival rate (computed by simply counting the number of packets arrived during the monitoring period). At the end of a monitoring period t_{mon}, the drop probability P_{ID} for each quarantined flow is updated based on the measured arrival rate and the estimated fair share of the bandwidth, C/N. If the measured arrival rate of a flow (BW_{ID}) is greater then θ_{det}, the flow is identified as an aggressive flow and its drop probability P_{ID} is computed proportional to the excess bandwidth. If the measured arrival rate is less than θ_{det}, the drop probability P_{ID} is reduced by half as long as it is greater than the drop probability P_Q of the underlying queuing policy. When the drop probability of a flow is less than P_Q, it is released from the quarantine memory. All the flows to be released are measured for the entire monitoring period to prevent releasing an unresponsive flow sampled in the middle of a monitoring period.

The drop probability, P_Q, can be computed in a variety of ways. In the simplest form, P_Q represents the packet loss rate due to buffer overflow with a Tail-Drop queue, and is easily computed by counting the number of packet forwarded and dropped. Alternatively, it can be replaced by the maximum drop probability (P_{max}) when a sophisticated queue management scheme such as ARED [21] and SRED [48] is used. Therefore,

the proposed scheme is quite generic and is independent of any AQM algorithm. However, in this chapter, we used HRED, presented in the previous chapter, in order to compare HaDQ to two prior scheme, CHOKe and FRED.

3.3.3 Punitive Measures

The simplest approach to protect honest, conforming TCP users is to completely block the unresponsive flows by dropping all of their packets. Instead of using total blocking, we adopt a random drop mechanism that enforces fair share of the bandwidth among all competing flows and, in particular, does not starve a mis-quarantined flow. Instead of a random drop, a sophisticated traffic measuring mechanism such as a time sliding window [13] or a leaky bucket mechanism[68] could be used to throttle a flow. If we assume that there are three queues in a packet memory, one each for TCP control, TCP data, and UDP, then another alternative that avoids starvation is to isolate the unresponsive TCP flows by simply demoting them to the UDP queue.

HaDQ with a probabilistic drop mechanism as a punitive measure is summarized in Figure 3.2. **We reiterate that HaDQ does not require an AQM mechanism to enact punitive measures.** When a new packet with a session identifier i arrives at a router, the quarantine CAM is searched first. If the flow is found in the quarantine memory, the flow's arrival rate estimation is updated. If the flow's drop probability P_{ID} is greater than *zero*, the packet is dropped with a probability P_{ID} (CASE 1). Otherwise, the packet is directed to the packet memory (CASE 2). If the flow is not found in the quarantine memory, it is directed to the L_1 cache of HaTCh and the estimated number of active flows is recalculated in HaTCh (CASE 3).

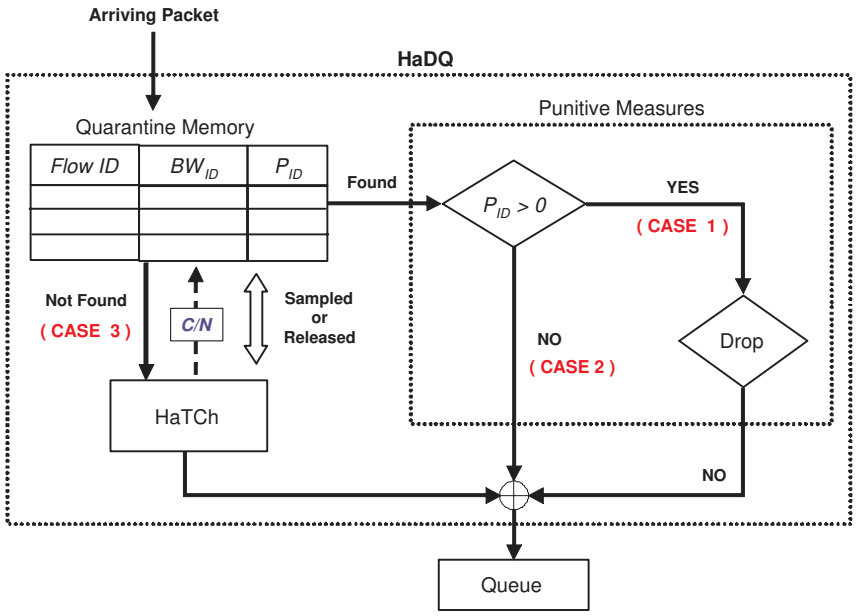


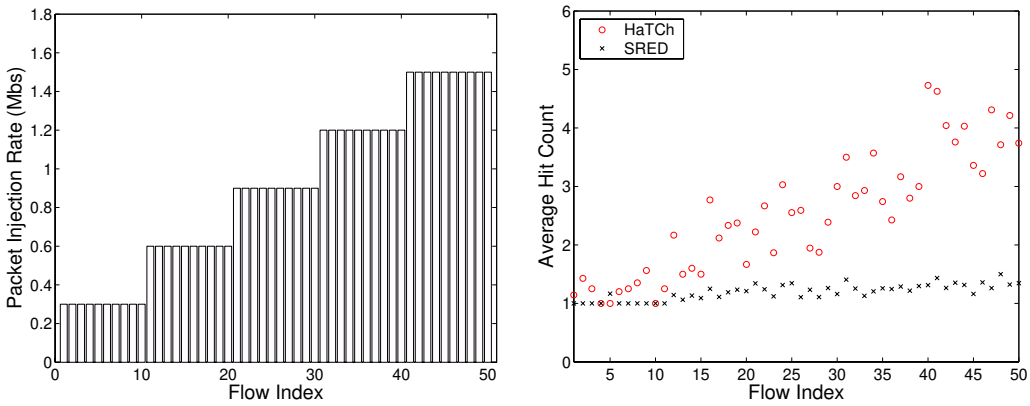
Fig. 3.2. HaDQ Design and Control Flow

For every hit in the L_1 cache of HaTCh, the hit count of the cache line is compared against the sampling threshold, θ_{smpl}, and the flow is registered in the quarantine memory if the hit count is greater than the sampling threshold. At this time, the flow's drop probability is set to zero. The sampling threshold is periodically updated based on the target number of quarantined flows (NQ_{low} and NQ_{up}) to be monitored. At the end of a monitoring period, the drop probability for each quarantined flow is updated based on the measured arrival rates and the fair share of the bandwidth. If the number of flows in the quarantine memory is less than NQ_{low}, the threshold θ_{smpl} of the L_1 cache is reduced by 1; while if the number of quarantine flows is greater than NQ_{up}, θ_{smpl} is incremented by 1. We used 4 as the initial value of θ_{smpl}, and limited its range ($3 \leq \theta_{smpl} \leq 12$) to avoid extremely low and high sampling rate. However, the initial value was not very critical in the detection accuracy in our simulations.

3.4 Performance Evaluation

The proposed scheme was evaluated through extensive simulation in terms of the efficiency of the sampling mechanism, the accuracy of detection, and the protection capability (punitive measure). Although the proposed HaDQ scheme is independent of the choice of the AQM schemes, we used HRED described in previous chapter, along with HaDQ because the traffic workload used in the following simulation covers from normal network conditions to heavily congested workload. We used the same simulation environment described in Section 3.2.1. The size of the quarantine memory of HaDQ was configured equal to HaTCh's L_1 cache targeting a maximum of 100 unresponsive flows (note that this is less conservative than "maximum aggressive flow" limit in [73]).

3.4.1 Sampling Efficiency and Scalability



(a) Packet Injection Rate (b) Average Hit Count

Fig. 3.3. Average Hit Count

First, we investigated the sampling efficiency of SRED [48] and HaTCh [73], which is used in HaDQ. Under SRED or HaTCh, an accurate estimation of the number of active flows (N) is essential since N is used in indicating the severity congestion. Therefore, the accuracy of the sampling mechanism depends on the performance of the estimation process. Aggressive flows can be efficiently sampled with these schemes either by counting the number of cache lines occupied by each flow or by comparing the hit counts as noted in [48]. In SRED, some of the active flows occupy cache lines more aggressively than others. This unfair share of cache lines among the competing flows results in underestimation of N, and therefore, in an inaccurate target fair bandwidth

sharing. As a result, the sampling mechanism will suffer from unacceptably high false positives.

Now, consider using the hit count to detect the aggressive flows. If the hit counts of aggressive and conforming flows are similar, the sampling mechanism will suffer from high false positives. To compare the characteristics of the hit count under SRED and HaTCh, we performed the following simulation. First, we divided 50 UDP flows into 5 groups, where each group had the same packet injection rate. The packet injection rate increased as the flow index increased as shown in Figure 3.3(a). Next, we took 100 snapshots of the SRED and the L_1 cache of HaTCh between 100s and 200s from 10 simulations, and calculated the average hit count for each flow. In Figure 3.3(b), the average hit counts of all flows with SRED are quite similar regardless of the packet injection rate. On the other hand, the average hit count in HaTCh clearly increased as the packet injection rate increased. Although the hit count in HaTCh did not exhibit the exact amount of shared bandwidth, it is accurate enough to discriminate relatively aggressive flows as we will see in greater detail in the following section.

Next, we investigated the memory requirement of the sampling/detection technique used in RED-PD [40]. As explained in Section 3.2.2, RED-PD maintains M separate lists of recently dropped packets to identify high bandwidth flows. Clearly, the accuracy of detection will improve with a larger M, but the memory requirement will also increase accordingly. Therefore, M is a critical parameter that determines the sampling/detection performance of RED-PD (RED-PD recommends $M = 5$ as a default value). In this simulation, we measured the number of per-flow states required for maintaining the drop history of RED-PD for different values of M (1,3, and 5) when 100,

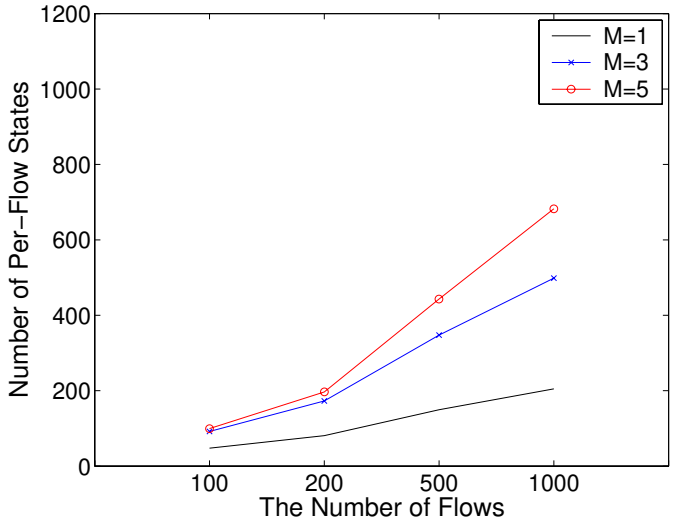


Fig. 3.4. The number of per-flow states required for RED-PD

200, 500, and 1000 TCP flows are used. In Figure 3.4, the memory requirement is less than 20% of the total number of flows when $M = 1$, but the *K out of M concept* [40] is no longer valid since all these flows are mis-sampled when a single list is used. With a larger value of M (3 and 5), the number of per-flow states increases almost linearly. As a result, the number of pef-flow states required for sampling reaches up to 70% of the number of active flows ($N = 1000$). Considering the characteristics of today's Internet traffic [52][14], which often results in persistent congestion, this can bring serious scalability problem.

The memory requirement for maintaining the per-flow sates with SRED also can grow to N in the worst case. On the other hand, the memory requirement of HaDQ is limited to the quarantine memory size, which is 10% of N assuming that the number of unresponsive flows is limited to 10%. In addition, HaDQ exhibits accurate sampling property as shown in Figure 3.3(b).

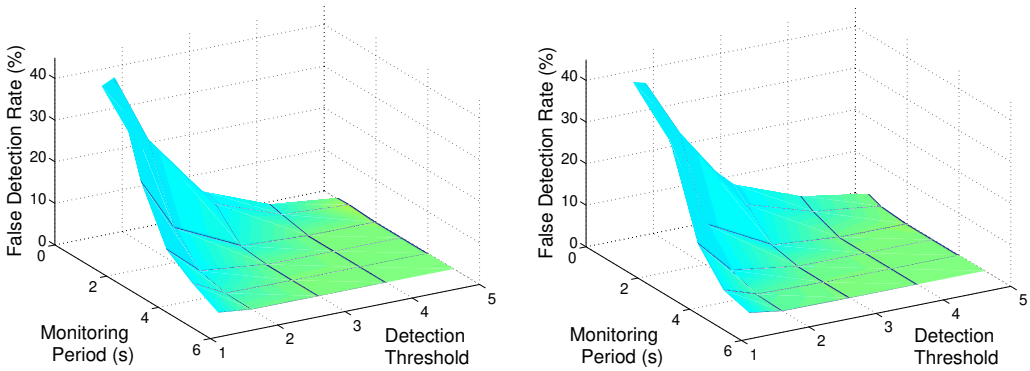
3.4.2 Detection Performance

Table 3.2. HaDQ Configuration Parameters

Parameter	Values
Monitoring Period (t_{mon}) (Sec)	0.5, 1, 2, 3, 4, 5
Detection Threshold (θ_{det})	1.5, 2, 3, 4, 5
Replacement Probability (r) for HaTCh	0.01, 0.1

In these simulations, we used 500/1000 standard TCP flows and an unresponsive TCP flow, and varied NQ_{low} from 30% to 50% and NQ_{up} from 70% to 90% of the quarantine memory size. Although the overall performance was similar with these values, the sampling threshold θ_{smpl} was more stable with $NQ_{low} = 40\%$ and $NQ_{up} = 80\%$ of the quarantine memory. Therefore, we used these values in the rest of the simulations, and the other HaDQ configuration parameters are shown in Table 3.2. We measured the false detection and false drop rates, which are defined as S_{fls}/S_{tot} and D_{fls}/D_{tot} respectively. Here, S_{fls} and S_{tot} represent the total number of false detections and the total number of samplings in HaTCh's L_1 cache respectively; and D_{fls} and D_{tot} represent the total number of false packet drops and the total number of packet drops, respectively. A false detection occurs when a conforming TCP flow is quarantined and mis-classified as an aggressive flow (i.e., false positives in detection), whereas a false drop occurs when a packet is dropped from conforming TCP flows due to the quarantine mechanism (false positives in punitive action).

Figure 3.5 and Figure 3.6 show the simulation results for the false detection rate and false drop rate, respectively. For 500 TCP flows and an unresponsive flow case, the false detection rate decreased from 38% to 0.3% as θ_{det} increased from 1.5 to 5 when $t_{mon} = 0.5$s in Figure 3.5 (a). As t_{mon} increased, the false detection rate decreased, and it became almost zero (less than 0.1%) in most simulations for $t_{mon} > 3s$. In Figure 3.6, the false drop rate also followed the same trend as the false detection rate. In addition, we found that both the false detection rate and the false drop rate significantly reduced as the number of flows increased (up to 1000 standard TCP flows and 25 unresponsive TCP flows).



(a) 500 TCP flows with $r = 0.01$

(b) 1000 TCP flows with $r = 0.01$

(c) 500 TCP flows with $r = 0.1$

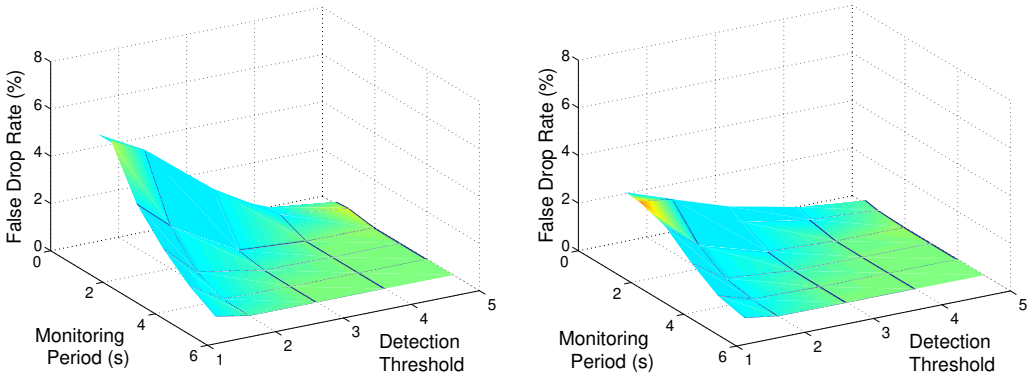
(d) 1000 TCP flows with $r = 0.1$

Fig. 3.5. False Detection Rate of HaDQ with different configurations

While the false detection rate and false drop rate are maintained in acceptable ranges by using proper configuration parameters, it is interesting to note that the dynamic quarantine scheme introduced a little inaccuracy in the estimation process of HaTCh, and resulted in an increased false detection rate. That is, if a flow is missampled and quarantined, the cache lines belonging to that flow remain in the L_2 cache, and HaTCh consequently underestimates the number of active flows. To minimize this side-effect, we increased the replacement probability r of HaTCh from 0.01 to 0.1, resulting in a shorter life-time of L_2 cache lines. Now, the false detection rate reduced to 28% (from 38.8% to 28%) with the new configuration in Figure 3.5 (c) compared to Figure 3.5 (a). Thus, a higher value of the replacement probability (r) decreases the false detection and false drop rates. Irrespective of the variation in false detection/drop rate, HaDQ successfully limited the bandwidth of unresponsive flow to the fair share of the bandwidth.

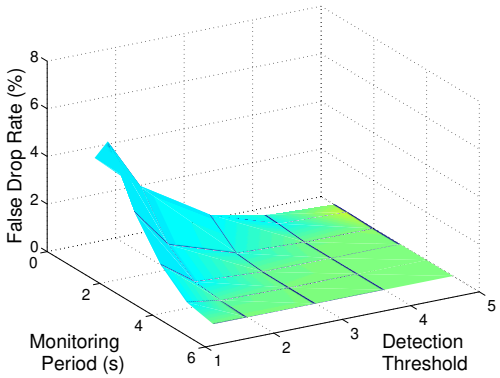
We configured HaDQ with $r = 0.1$, $t_{mon} = 2$s, and $\theta_{det} = 3$ for the following simulations, since the false sampling and false drop rates were negligible with these values, while maintaining a good sampling sensitivity.

The detection speed of HaDQ was also investigated. In each simulation, we began with 500 TCP flows, and added unresponsive TCP flows (from 1 to 80 flows) between 100 to 102s. Figure 3.7 depicts the quarantine delay, i.e., the time between when an unresponsive flow commences and when it is quarantined, and shows that HaDQ managed the detection time within 2.3 seconds regardless of the number of unresponsive flows. In addition, the detection time showed very small variation.

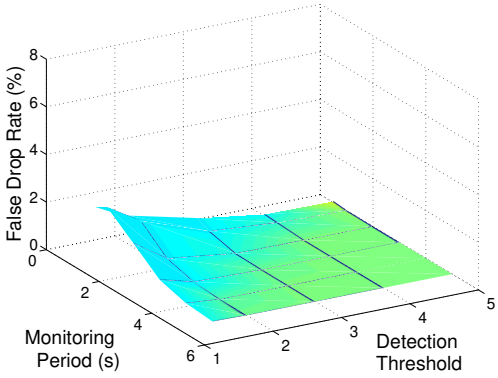


(a) 500 TCP flows with $r = 0.01$

(b) 1000 TCP flows with $r = 0.01$



(c) 500 TCP flows with $r = 0.1$



(d) 1000 TCP flows with $r = 0.1$

Fig. 3.6. False Drop Rate of HaDQ with different configurations

3.4.3 Impact of the Punitive Measure

To show the effectiveness of HaDQ in protecting conforming TCP users, we compared the average throughput of the conforming TCP flows and unresponsive flows under CHOKe, FRED and HaDQ. CHOKe and FRED are built upon ARED, and all these schemes are deployed at the congested link in Figure 3.1. We did not simulate RED-PD in this study. Intuitively RED-PD should provide fair bandwidth share like HaDQ, since it also monitors the aggressive flows. However, as showed in Section 3.4.1, the memory required for the sampling mechanism is much higher than the HaDQ scheme, resulting in poor scalability. We also simulated SFB [19], but its results were very much dependent on the system configuration parameters and the simulations environments. Since the SFB results showed wide variation, we only discuss CHOKe and FRED results here.

The simulation results for 500 to 1000 conforming TCP flows with an unresponsive flow are depicted in Figures 3.8 (a) and (b). FRED showed slightly better protection of the standard TCP flows compared to CHOKe, but both the schemes failed to sufficiently penalize the unresponsive TCP flow. Under FRED, the single unresponsive TCP flow occupied 13% and 11% of the total bandwidth, respectively for 500 and 1000 TCP flows; these numbers increased to 39% and 31%, respectively, under CHOKe. As a result, conforming TCP flows suffered from continuous retransmissions and timeouts. Moreover, none of the packet drops applied to the conforming TCP flows was caused by the random drop mechanism of the underlying AQM scheme (ARED). This implies that both CHOKe and FRED failed to differentiate between the conforming TCP flows and the unresponsive flows, and subjected the former to unnecessary packet drops. On

the other hand, HaDQ performed extremely well as the figures show. HaDQ precisely activated the punitive measure against the unresponsive TCP flow and enforced fair sharing of the available bandwidth in both cases. In contrast to CHOKe and FRED, all the packet drops applied to the standard TCP flows were triggered by the underlying AQM scheme (HRED), not by HaDQ, which fairly increased the packet drop rate for the unresponsive flow.

Figures 3.8 (c) and (d) depict the simulation results when we increased the number of unresponsive flows up to 5% of the total number of standard TCP flows. As can be seen, significant performance degradation resulted under CHOKe and FRED: 5% of the unresponsive TCP flows occupied more than 99% of the total available link bandwidth. HaDQ, again, fairly and accurately limited the total bandwidth of unresponsive flows to around 5%, and provided excellent protection to conforming TCP flows. Finally, we added an UDP flow, whose injection rate is 16 times the fair sharing of the available bandwidth, to an unresponsive TCP and 500/1000 TCP flows assuming that all the flows are multiplexed in a queue. The drop rate of CHOKe was again proportional to the bandwidth share, and FRED showed slightly better performance in Figures 3.8 (e) and (f). However, Under CHOKe and FRED, the UDP and unresponsive TCP flows again consumed significant amount of bandwidth, whereas HaDQ effectively enforced the fair bandwidth sharing.

3.5 Concluding Remarks

Internet users can easily compromise the TCP congestion control mechanism by deactivating the slow-start and retransmission timeout or by launching multiple parallel TCP sessions. These activities can be interpreted as a DoS attack on the honest, conforming users and, thus, pose a serious security concern. Finding an efficient and feasible solution to control these unresponsive flows is made difficult primarily because of the large traffic volume and the complexity of the Internet traffic dynamics.

In this chapter, we propose a novel solution for handling unresponsive TCP flows. The proposed policing technique, called HaDQ, uses our proposed HaTCh scheme to sample, detect and dynamically quarantine the aggressive flows, which are initially captured in the first level (L_1) cache of HaTCh. The HaTCh scheme accurately estimates the number of active TCP sessions, which ,in turn, is used to compute the per-session fair share of bandwidth. The proposed scheme uses a small CAM, called the quarantine memory, to monitor the aggressive flows and take punitive measures when their estimated arrival rates significantly exceed the fair bandwidth sharing. The main attractive features of the proposed scheme are its accurate sampling technique that helps in minimizing the per-flow state, and thus, helps in scalability, and the detection and punitive mechanisms that are based on the number of competing flows.

Extensive performance evaluation indicated that the proposed dynamic quarantine scheme maintains a very low false drop (false positive) rate of less than 0.1%, and can provide much better protection to conforming TCP flows compared to CHOKe and FRED. Simulations with up to 5% unresponsive TCP sessions and with a mix of UDP

and an unresponsive TCP sessions showed that HaDQ can provide almost equal bandwidth allocation to all of the competing sessions. Although HaDQ was integrated with an AQM scheme (HRED) in order to compare with CHOKe and FRED in this chapter, it does not require any AQM scheme in detecting and taking punitive action.

Our future work involves further investigation of adaptive HaDQ configurations under more realistic workloads consisting of heterogeneous traffic.

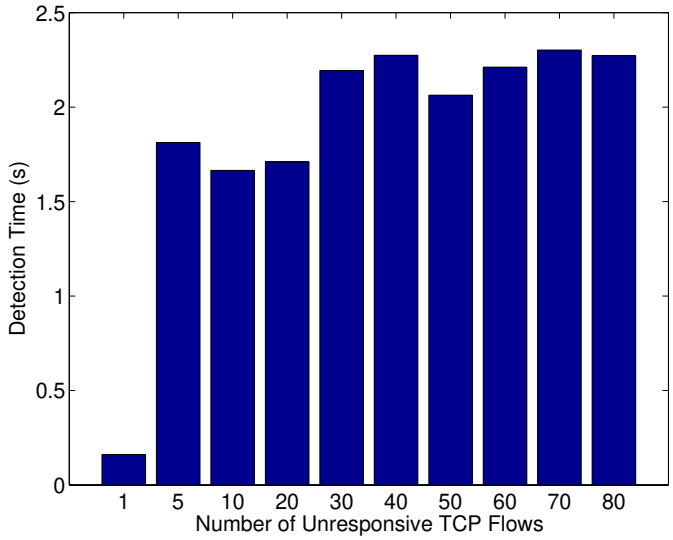
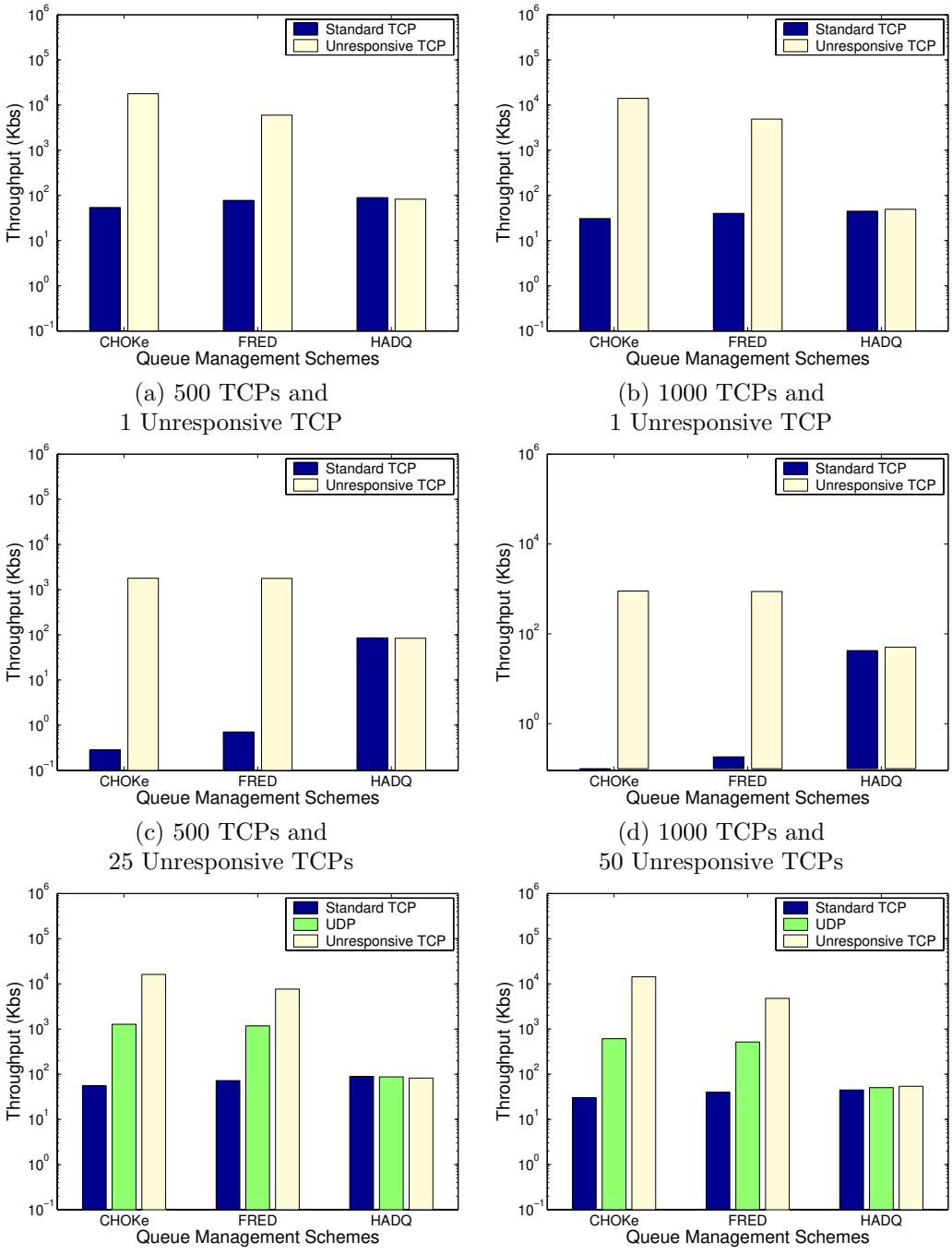


Fig. 3.7. Detection Time of HaDQ



(a) 500 TCPs and 1 Unresponsive TCP

(b) 1000 TCPs and 1 Unresponsive TCP

(c) 500 TCPs and 25 Unresponsive TCPs

(d) 1000 TCPs and 50 Unresponsive TCPs

(e) 500 TCPs, 1 UDP, and 1 Unresponsive TCP

(f) 1000 TCPs, 1 UDP, and 1 Unresponsive TCP

Fig. 3.8. Throughput comparison of CHOKe, FRED, and HaDQ

Chapter 4

Worm Defense Mechanism

4.1 Introduction

Since November 1988, worms have been one of the most serious threats to the millions of the Internet community. A worm is a self-propagating program, which contains malicious payloads that rapidly spread using the network connections. Although the potential impact of worms is significant enough to bring down the entire network service, defending against worms remains a largely open problem. Difficulties in developing an efficient worm defense mechanism lie in its ferocious nature. First, as has been shown in [44] [63], the propagation speed of today's worms is extremely fast, and thus can easily outpace human response. For example, SQL Slammer, also known as Sapphire, infected more than 90 percent of vulnerable host within 10 minutes [44]. Second, worms are becoming sophisticated in that variations (and imaginations) of each worm outbreaks in short period of time after the initial instance. For example, the Code Red worm was initially released in July 13 2001. Although the first version of Code Red infected many hosts and consumed network bandwidth by spreading itself, it did not cause serious damage due to a poor scanning (or probing) mechanism. A week later, the second version of Code Red, Code Red II, was released with a new scanning mechanism resulting in hours of network breakdown [45]. Therefore, signature-based worm detection is vulnerable to the presence of new worms. Third, considering the fact that

false positives can cause denial-of-service to legitimate network users, a worm defense mechanism should be highly reliable.

Current trends in worm defense technology are largely rely on *containment* framework [62] [70] [46]. The main idea of the containment technique is to quarantine the worm activities within a limited area such as an enterprise network or a subset of an enterprise network, called a cell, to prevent Internet-wide infections. When probing activities are detected, all traffic from the suspected hosts are blocked, either totally or partially. Therefore, a reliable and robust detection mechanism, which can minimize the false positive, is a key success component in deploying a containment technique.

We present a novel worm detection mechanism in this chapter. The proposed detection mechanism is based on the HaTCh scheme presented in Section 2, which estimates the number of active flows at the Internet router, and HaDQ scheme described in Chapter 3, designed to detect and penalize bandwidth attacker. The main feature of the proposed scheme is that it minimizes the amount of per-flow state required to detect probing activities using HaTCh. Once the probing activity is detected, then all the traffics from the suspected source address are blocked using a mechanism similar to that used in HaDQ. Initial simulation results show that the proposed scheme is very effective in detecting malicious worm traffic. For example, the detection time for SQL Slammer type of worm is less than 3 seconds without any observed false positives.

The rest of this chapter is organized as follows: We briefly describe previous work on worm defense mechanisms in Section 4.2. The proposed approach is detailed in Section 4.3. In Section 4.4, simulation results are presented followed by the concluding remarks in Section 4.5.

4.2 Related Work

Williamson [70] proposed to implement a kind of filter on the network stack to regulate the rate of connection requests to new hosts. In rate regulation, a small buffer called the delay queue, is used to allow a fixed rate of new requests. Therefore, legitimate connections with occasional bursts experience a small delay and loss, whereas port probing traffic of worms is rate-limited with possible heavy loss. One limitation of this approach is that most of the Internet end-hosts must upgrade their network stack to make this approach successful. Another problem is that this technique itself can be compromised by a worm that may gain system level control, since this technique is implemented in network stack of individual hosts.

In [46], Moore et al. studied worm containment techniques, particularly address blacklisting and content filtering. Address blacklisting is a technique to limit access of a set of IP addresses, which are identified as infected hosts. It can be easily deployed at an Internet router, but list management operations such as update and remove can be very expensive computationally considering the volume of traffic in the Internet.

Unlike address blacklisting, the content filtering approach actively detect worm propagation by investigating content signature of packets. For this, an Internet router needs to maintain a database of content signatures for each identified worm. This approach should be accompanied by additional technologies such as characterizing and generating content signatures of worms, and only work under the assumption that worm is not polymorphic. However, this assumptions is likely to be broken in the near future [45] [69].

Recently, Staniford [62] proposed a cell-based containment technique. Here, an enterprise network is divided into smaller units, called cells. Each cell is equipped with containment device, which limits the number of new destinations, at the boundary of cells. Although he claims that for most port the number of new destination can be managed within a constant (10 flows), cells that include web servers, game servers, and p2p participants (e.g., Kazaa) can easily exceed such limitations. In addition, UDP-based worms such as SQL Slammer are difficult to control under this scheme.

Within the worm containment concept, a few worm detection techniques have recently been proposed. S. Chen et al. proposed the DAW technique to detect and limit the propagation rate of a worm [10]. The main idea of DAW is that a worm-infected host will show higher connection failure rate than an uninfected host due to its fast but inaccurate scanning behavior. A limitation of this approach is that DAW only works for worms that use TCP as the transport layer protocol. However, SQL Slammer, which is a UDP based worm, has infected more than 90% of vulnerable hosts within ten minutes, and other UDP based worms are likely to outbreak in the near future due to the simplicity and effectiveness of their scanning strategies. Another problem is that implementation details have not been discussed in [10]. An Internet router is a point where millions of connections converge. To analyze the connection failure rate with DAW, maintaining significant amount of per-flow state is unavoidable. Therefore, DAW also should be accompanied by an effective technique to minimize the amount of per-flow state to be deployed at the Internet router.

Recently, X. Chen et al. proposed a new worm detection and suppression technique called DEWP [11]. DEWP's detection mechanism is based on the port matching

and the number of different destination addresses using a port. However, today's Internet traffic is often asymmetric with extremely wide variation as shown in [53]. In this case, the DEWP detection mechanism should be only rely on the number of unique destinations for a specific port in detecting worm activity, but the implementation details on regarding per-flow maintenance has not been investigated. More importantly, DEWP has to block all packets that use the specific port number. Considering the fact that worms such as Code Red, Nimda and Welchia [54] have exploited port 80, which is also used by web traffic, a worm detection mechanism based on this specific port number can be easily hampered.

In summary, worm containment based approach may be one of effective solutions in limiting and suppressing worm propagation. However, its performance is heavily dependent on the detection mechanism. Prior worm detection techniques explored in this section may work in the specific scenario, but they depends on a specific transport layer characteristics. In addition, the implementation issues has not been fully investigated.

4.3 The Proposed Scheme

This section presents the proposed worm detection and quarantine mechanism, which detects and suppress scanning activities in an enterprise network.

4.3.1 Worm Detection and Quarantine Mechanism

A network-based worm defense mechanism needs to collect and analyze per-flow or per-source statistics to determine whether the specific flows are malicious or not. In addition, per-flow or per-source state maintenance is necessary to reduce false positives.

However, a modern Internet router is a place where millions of connections converge at any given time. Collecting information on all the connections can not only raise the scalability issues but also cause significant overhead that can result in serious performance degradation. Therefore, developing scalable techniques to ascertain *suspicious flows* and to collect information only for *suspicious flows* is critical in deploying a network-based worm defense mechanism. For this reason, a network-based worm defense mechanism requires two steps: sampling and detection. A main purpose of sampling is to select only a subset of active flows that show abnormal behavior to reduce per-flow or per-source information management, and then the sampled flows are closely investigated and classified as either worms or normal flows.

The main idea of our the proposed Worm-DQ scheme comes from the investigation of general worm behavior. First, the total size of a worm payload is very small ranging from hundreds bytes to a few kilo bytes. Second, today's worms can transmit enormous amount of probing packets either at the link speed (SQL Slammer) or by invoking multiple threads (Code Red) in searching for possible victims. Although some worms such as blaster exploit address co-relation among the workstations within a network, target addresses of scanning packets are generally randomly generated. For example, the blaster worm uses linear scanning after a successful scan. However, these worms still rely on random scanning until the first successful compromise. As a result, a router located in the attack path will see many short-lived connections from the same source address toward millions of different destination address.

The proposed worm defense mechanism is based on both the HaTCh [73] and HaDQ [74] schemes (See Section 2 and Chapter 3). The HaTCh scheme estimates the

number of active flows without maintaining per-flow states and this quantity is used to dynamically detect and penalize unresponsive flows in HaDQ. The first step in worm detection is to sample the possible scanning traffic since since it requires significant resources including CPU and memory to trace all the active flows in Internet router.

Sampling: Under HaTCh, worm traffic cannot be registered at the L1 cache due to so many different destination addresses. As a result, worm packets will tend to *miss* at the L1 cache, but will occupy the L2 cache lines very aggressively. Therefore, we use the L2 cache of HaTCh to sample possible scanning traffic. All the flows registered in the L2 cache represent a short history of the recent packets, precisely $\frac{M}{r}$ packets, that have arrived at the HaTCh device. Theoretically, these cache lines should be shared equally by all the competing flows. Therefore, the "unfair" distribution of the L2 cache lines per source address implies the presence of an aggressive flow. Here, we exploit the hash mechanism of the HaTCh scheme. The L1 and L2 Cache of HaTCh is partitioned into small chunks called subcaches. When a packet arrives, it is hashed into a subcache using its source and destination addresses. Thus, every given flow is always hashed into the same subcache. On each packet arrival, the per-source L2 cache lines count for the corresponding subcache is updated. If the per-source L2 cache lines count for the subcache exceeds the fair cache line share, the source address becomes *suspicious*. A flow is quarantined from HaTCh if a source address is suspicious at more than c out of k subcaches, where k is the hash size. Therefore, the quarantined flow is always assured that there are at least c different connections from the same source address.

Detection: A quarantined flows is isolated from the HaTCh operation, and registers to the quarantine memory as has been done in HaDQ. Unlike HaDQ where

actual bit rate is measured for each quarantined flows, here, we estimate the number of different destination addresses by using the HaTCh technique. Here, we define this quantity as the degree of destination. In estimation, only a single memory cell is used for a source address. Assuming that the degree of destination from a single source should be small generally, a large this number imply that the host is highly suspicious. Then, all the packets form the source address are blocked. Since the performance of the estimation mechanism is critical in the accurate detection, we modified the estimation process of HaTCh for more accurate and stable estimation. HaTCh calculate the hit frequency to estimate the number of active flows using a first order autoregressive process as follows:

$$f(t) = (1 - \alpha)f(t-1) + \alpha \cdot 1 \{\text{hit at } t^{th} \text{ packet }\}$$

for $0 < \alpha < 1$. 1 in the above expression represents an indicator function of a cache hit. Then, the inverse of the hit frequency ($f(t)$) is used as the estimate of number of flows. It is clear that the smaller α gives more stable estimation with a longer response time, since it is the time constant of the estimation process. When the number of flows is small, a single miss in HaTCh contributes a very small change in the total estimated number of flows. However, the estimation process over-reacts for the large number of flow with a static value of α. For example, assuming $\alpha = 0.001$, the hit frequency for ten competing flows is 0.1, and the estimate becomes 0.1009 after a cache hit. On the other hand, the hit frequency of a thousand competing flows is 0.001, and the estimate becomes 0.00199 (500 flows) after a cache hit. As a result, the estimated number of flows

severely fluctuates under a large number of competing flows. Therefore, we adaptively adjust α based on the current hit frequency as shown in Figure 4.1.

```
if (adaptive )
        if ( f(t) > 0.4 )
                alpha = 0.01
        else if ( f(t) > 0.2 )
                alpha = 0.005
        else if ( f(t) > 0.02 )
                alpha = 0.001
        else
                alpha = 0.0001
```

Fig. 4.1. The Adaptive Estimation Algorithm

The proposed scheme can be summarized as follows: When a packet arrives at a router, the quarantine memory is searched first looking for the same source address. If the flow is found in the quarantine memory and the estimated number of flows for the source address exceeds the threshold (θ_b), the packet is dropped. Otherwise, the packet is processed by the HaDQ mechanism. For the each miss in the L2 cache, the number of cache lines per source address is updated. If there are more than θ_s subcaches, which has more cache lines for the source address, then the flow is registered in the quarantine memory. We used the threshold (θ_b) value for blocking suspicious flow as 400, which is the four times of the size of a subcache in the following section. The quarantined flows are released if the estimated number of flows is alway less than θ_b for a period of time (θ_r).

4.3.2 Discussion

Although the proposed scheme may generally be effective in detecting high speed worm traffic, there are a few practical limitations. Within an enterprise network, there can be several high-profile end-hosts. For example, a game server, web server or database server can legitimately support more than a thousand of concurrent users. To reduce the false positives, the proposed scheme should maintain a high-profile server list. When these hosts are identified as a worm infected under the proposed scheme, the detection can be suppressed. However, this approach leaves two problems. First, router resources can be meaninglessly consumed since these hosts will be keep on being detected and released from the quarantine memory. Second, it will leave another vulnerability for possible attacks that target theses hosts. Therefore, we investigate the subset of the traffic for these flows. When these hosts are registered at the quarantine memory, the hit frequency is calculated only for the subset of the flows instead of all the active flows. Here, a simple hash function can be used to select a subset of the traffic, and the size of the hash function can be determined by the traffic statistics for the particular host.

4.4 Performance Evaluation

In this section, the simulations environment is described and then preliminary simulation results are presented.

4.4.1 Simulation Environments

The proposed scheme was evaluated through extensive simulations using ns2 [3] for the topology given in Figure 4.2. The main structure of the topology is based on one

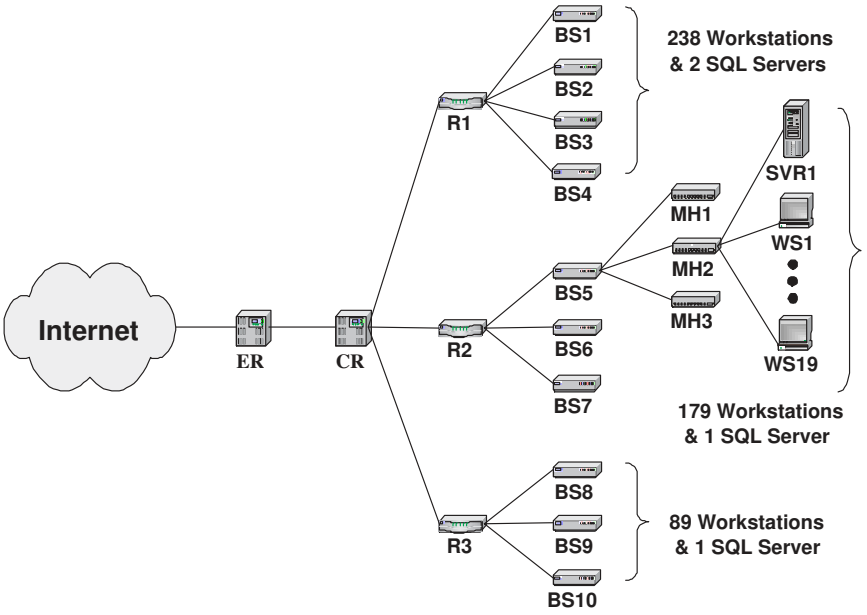


Fig. 4.2. The network topology used for simulation

at Oregon State University [1], and it is scaled down to represent an enterprise network with five hundred end nodes. There are 506 work stations and 4 SQL servers, and these nodes are connected to the Internet using a hierarchy of routers, such as a multi-port hub (MH), building switch (BS), router (R), center router (CR) and edge router (ER). Links between end node and MH are 10Mbs with 10ms delay, and all other nodes are connected with links of 100Mbs with 10ms delay. For simulations, we used the ns2 worm model [11] that is based on the SIR (Susceptible-infectious-removal) model [46] [63] [37] [29]. In the ns2 worm model, an entire simulation network is divided into two parts: an abstract network, which represents the Internet and an detailed network where the packet level details is simulated.

The overall behavior of an abstract network is similar to the SIR model. Thus, the number of infected, vulnerable, and removed hosts are calculated based on the scan rate, the number of probing packets received from the detailed networks, and the total number of hosts. In addition, probing packets are periodically released toward the detailed network. When a host in the detailed network receives a probing packet, the node is infected if it is vulnerable. The host then immediately starts to generate the probing packets at the predefined rate. Here, worm traffic uses UDP for sending 404-byte probing packets to mimic the behavior of SQL Slammer, and the average scanning rate is set to 4000 [44].

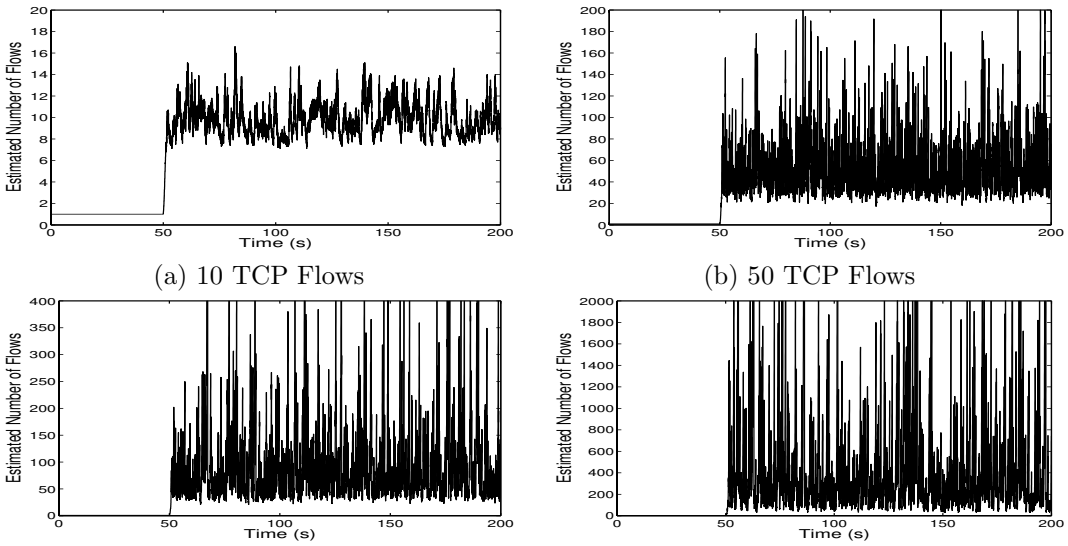
The proposed worm defense mechanism is deployed at the edge router (ER) with HaDQ configured according to [74] as shown in Table 4.1.

Table 4.1. Configuration Parameters

Parameter	Values
L_1 Cache Size	200
L_2 Cache Size	2000
Quarantine Memory Size	200
Hash Size	20

4.4.2 Simulation Results

Before we evaluate the performance of the proposed scheme, we first evaluate the accuracy of the modified estimation process using a single memory since the performance of the proposed worm detection relies on it. A simple *dumbbell*-like topology was used for the simulations. There is a source node on the left side of the network and five hundreds node on the right side of the network are used as the destinations, and the proposed mechanism is deployed in the middle to estimated the number of active flows. TCP traffic was generated at the beginning of the simulations and the proposed scheme activated after 50 seconds and lasted for 150 seconds. Figure 4.3 summarizes the estimation process with the static time constant ($\alpha = 0.001$). For small number of flows, the estimation is quite accurate as shown in Figure 4.3 (a). However, the estimation severely oscillates as the number of flows increases in Figure 4.3 (b), (c), and (d). On the other hand, the adaptively configured estimation process shows significantly reduced oscillation resulting in more stable and accurate estimation. The proposed scheme also accurately estimated up to 50 flows in Figure 4.4 (a) and (b). Note that we use a different scale for the y-axis in Figure 4.4 to show the estimation in detail. In addition, the response time of



(a) 10 TCP Flows

(b) 50 TCP Flows

(c) 100 TCP Flows

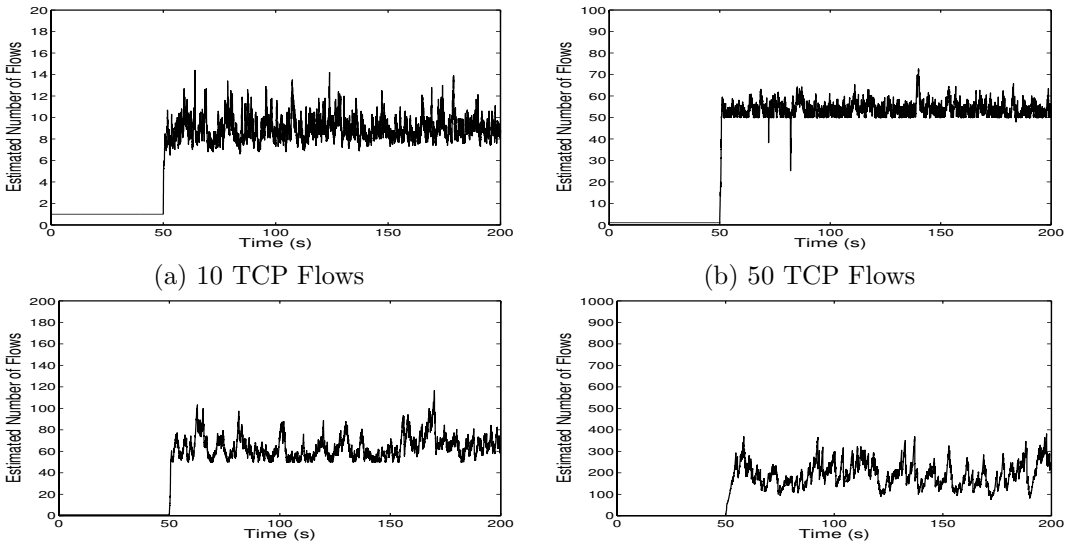
(d) 500 TCP Flows

Fig. 4.3. The Estimated Number of Flows with $\alpha = 0.001$

the estimated process also improved. However, a large number of flows brings under-estimation in Figure 4.4(c) and (d) as noted in [73].

To evaluate the performance of the proposed scheme, we performed a number of simulations by varying the amount of background traffic from 250 to 1000 TCP flows and configuration parameters, such as the sampling threshold (θ_s) from 2 to 20. Then, we randomly selected 5 end-hosts and generated 300Kbs of UDP traffic from 2 to 20 different destinations (degree of destinations) to cause false samplings (we refer to these flows as the controlled traffic in the following). Figure 4.5 summarizes the simulation results. Figure 4.5 (a) and (c) shows the sampling time, which is defined as the time when the host is infected to the time when the flow is quarantined. It is clear that the average sampling time gradually increases with the larger value of the sampling threshold (θ_s) since a larger subcache should be assigned a larger sampling threshold. However, the difference is less than 1.3 seconds. Note that the sampling time with large background traffic (250 flows) in Figure 4.5 (c) is smaller than that of the 1000-flows case in Figure 4.5 (a). As the number of flows increase, the background TCP traffic backs off due to the congestion, but the worm traffic does not respond to it. In addition, since the fair per-source=- cache lines are reduced due to the large number of flows, the worm traffic is quickly sampled. Once the worm traffic is sampled, all of this traffic is blocked in less than two seconds in all the simulations we ran. Therefore, the maximum delay for the proposed scheme to block worm traffic is 2.8 seconds for 250 background TCP flows and 2.5 seconds for 1000 TCP flows case.

We also investigate the impact of false sampling. As describe previously, we used the controlled traffic to cause false sampling. Obviously, when the degree of destination is



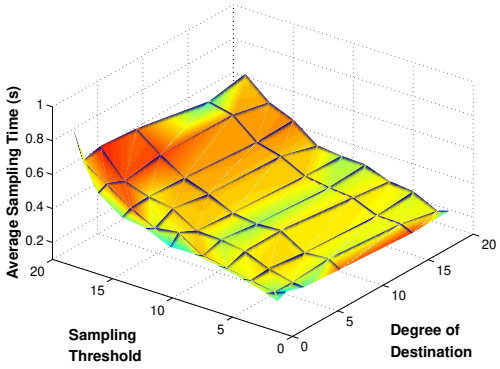
(a) 10 TCP Flows

(b) 50 TCP Flows

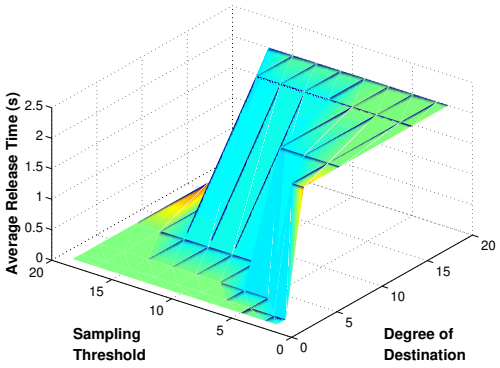
(c) 100 TCP Flows

(d) 500 TCP Flows

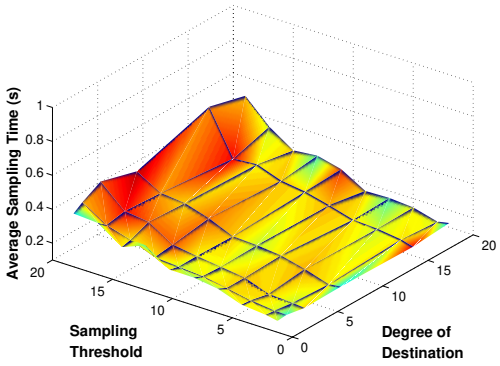
Fig. 4.4. The Estimated Number of Flows with adaptively configured α



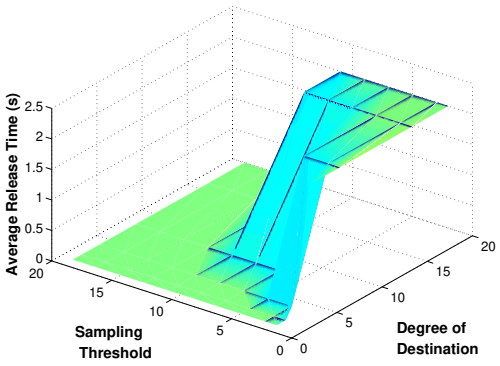
(a) Average Sampling Time with 250 TCPs



(b) Average Release Time with 250 TCPs



(c) Average Sampling Time with 1000 TCPs



(d) Average Release Time 1000 TCPs

Fig. 4.5. Detection Time with 250 TCP flows

smaller than the sampling threshold (θ_s), the source nodes are not sampled. In Figure 4.5 (b) and (d), these cases are presented as the zero sampling time, all the mis-sampled flows are released after 2 seconds, which is a design parameter (θ_r). The number of mis-sampled flows with 1000 background TCP flows in Figure 4.5 (d) is smaller than that for 250 TCP flows shown in Figure 4.5 (b). This is because that the controlled flows are competing with a large amount of background traffic. As a result, some controlled flows are not sampled for the large sampling threshold ($\theta_s > 10$). In all simulations, there was no false detection.

4.5 Concluding Remarks

In this chapter, we presented a new worm defense mechanism based on the number of flows estimated by the technique described in Chapter 2. A simulation-based performance study showed that the proposed scheme detected and quarantined worm propagation traffic quickly without causing false positives. We plan to extend the proposed scheme and to explore the following issues:

- The impact of transport layer protocol used by worm traffic on the detection performance
- The impact of the worm scanning rate on the detection performance
- The design of a detection mechanism for slowly propagating worms

For this purpose, we implemented TCP-based worms, which mimic the behavior of the blaster worm. We believe that the proposed scheme will provide a viable solution to protect networks from scanning worms.

Chapter 5

Proxy-RED: An AQM Scheme for Wireles LAN

5.1 Introduction

Wireless networks based on the IEEE 802.11 standard have been widely deployed in enterprises and university campuses mostly to provide wireless data access to Laptops, PDAs, etc., to the wired infrastructure. However, the available bandwidth in IEEE 802.11 networks is much smaller than in a wired local area networks since IEEE 802.11 networks are non-switched half-duplex, i.e. only one participant can transmit at a time. Interference from radio sources such as microwaves, cordless phones and other 802.11 networks further reduces the available bandwidth. These networks also suffer from hidden-station problem which results in collisions and hence lowered channel performance [59].

The peak transmission rate possible in 802.11a/g stations is 54 Mbs. However, as earlier analytical [33] and experimental studies [26] have shown, due to the large fixed overhead per frame transmission, the maximum channel efficiency is only 50–60%. Moreover, the peak data rate can only be used in close proximity to the Access Point (AP); stations further away from the access point fall back to lower data rates that in turn diminish the maximum channel throughput. The actual channel throughput also heavily depends on the frame payload size. When only frames as are typical for VoIP

traffic are sent, the maximal throughput on the wireless channel can drop to below 1 Mbs even at data rate of 11 Mbs [33].

Bandwidth in wireless media will remain a limited resource as compared to wired networks. Increasing bandwidth by adding other base stations covering the same area does not scale as there is only a limited number of non-interfering channels. While an IEEE working group (IEEE 802.11e) is working on alleviating some of the Quality of Service (QoS) issues that come with the use of wireless networks and their limited bandwidth, MAC layer approaches will not solve the congestion problem arising from the disparate link speeds in an access point. An access point has two interfaces, an 802.11 wireless interface to transmit/receive frames on the air and a wired interface to the Distribution System (DS). In an enterprise, the DS typically is 100 Mbs switched Ethernet. The disparity in channel capacity of these two interfaces makes the access point a significant potential bottleneck in the downstream direction.

In such scenarios, it is very likely that the outgoing link gets oversubscribed resulting in frequent output buffer overflow. In the absence of any special mechanism, congestion notification to the TCP sources using the link will occur in the so called "tail-drop" manner, and the performance of Tail-Drop queue is well documented to be poor [22]. Furthermore, synchronization of multiple TCP connections flowing through the access point may result in substantial lower throughput exacerbating the overhead of 802.11 networks. The throughput problem could be alleviated to a certain extent by having large buffers on the outgoing "air-interface" of the access point. However, this causes excessive delays for packets with real-time constraints. Further, which packets are dropped is not controlled by the access point.

In wired networks, as a solution for congestion, Active Queue Management (AQM) schemes, especially RED [22] and its variants [18] [21] [38] [50] [40] have been widely studied. The basic idea of RED is to detect the incipient stage of a congestion, and notify the sources to reduce their packet injection rates by deploying a random drop/marking mechanism. To measure the severity of congestion, RED calculates the Exponentially Weighted Moving Average (EWMA) of queue length at each packet arrival, and uses this number to find the drop/marking probability. RED is implemented in network switches and in routers which are likely to suffer from congestion due to disparate interface speeds. It thus appears well motivated to study the effect of implementing AQM schemes in a wireless access point to address the congestion issues.

As deployment of 802.11 networks has increased, so has the concern for supporting enterprise applications over these networks. The lack of sufficient support for mobility (subnet roaming), security and QoS are key concerns that have led to the development of wireless architectures which include a gateway (GW) shown in Figure 5.1. This gateway is a network element that sits one or more hops away from the access point to provide mobility, security and QoS support. All traffic in-out of the access point is routed through this gateway, with some sort of tunneling/NATing mechanism if the gateway is server-based and multiple hops away from the access point. Next hop gateways are switch based and provide the same services as server-based gateways. Switch-based solutions can also supply power-over-Ethernet (POE) to the access point.

Another overriding concern in wireless deployments is cost. As access points get loaded with functionality, they get more and more expensive, thereby making radio coverage provisioning in an enterprise very expensive. An alternative architecture, called

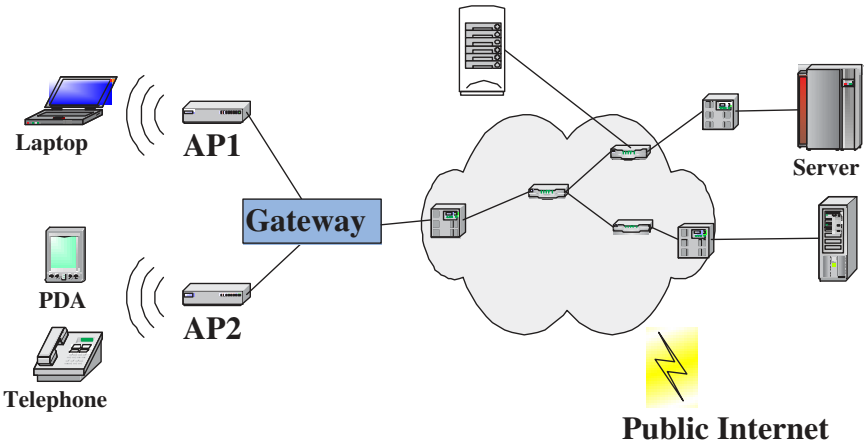


Fig. 5.1. Architectural trend in Enterprise 802.11 deployments

light weight access points, is again the use of a gateway in which the bulk of networking functionality is provided by a gateway, limiting the access point to do basic 802.11 channel access along with simple bridging to 802.2 frames. Features like QoS, mobility, security, management, location services etc. are supported in the gateway which can handle more than one access-port. The economies of scale deliver significant cost benefits for medium and large scale deployments.

In this chapter, we first study the impact AQM/ECN schemes on wireless Local Area Networks (WLAN). Simulation results shows that an AQM scheme such as RED aggravates the packet loss rate and goodput due to the large delay at the access point. However, we found that the performance of RED could be significantly improved when used with ECN. We then investigate the feasibility of implementing AQM at the gateway. In particular, we study whether it is possible to achieve effective congestion avoidance

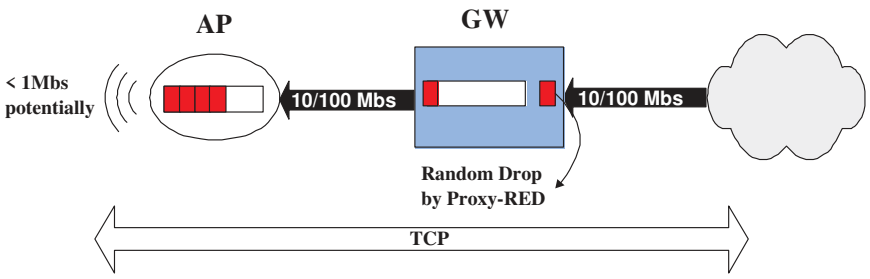


Fig. 5.2. High level solution architecture

by implementing RED in the gateway, not the access point. Since in this scheme, the gateway performs RED on behalf of the access point, we call this scheme *Proxy-RED*. The basic idea behind this approach is depicted in Figure 5.2. In the downstream direction, Ethernet frames cross the ingress and egress interfaces of the gateway and get queued up at the access point for transmission over the air. For RED to be effectively implemented in the gateway, the gateway should be aware of the queue state in the access point. Here, we uses a periodically sampled instantaneous queue length of the access point to calculate the estimated average queue length at the gateway. Once this is achieved, the gateway can implement RED in the usual manner with the RED parameters configured based on the maximum queue size at the access point.

In the original RED, the average queue length is calculated on each packet arrival whereas it is done periodically in Proxy-RED. The average queue length is an important parameter that determines the overall AQM performance since it indicates the severity of congestion, and affects the stability of the random drop mechanism. Simulation results show that the estimated average queue length in Proxy-RED depicts the behavior of

original average queue length very accurately. As a result, the proposed Proxy-RED scheme results in overall performance improvement in WLAN. In particular, the Proxy-RED scheme significantly improves packet loss rate and goodput for a small buffer, and delay for a large buffer size.

The remainder of this chapter is organized as follows: We demonstrate the impact of an AQM/ECN scheme on WLAN with a brief discussion in Section 5.2. The proposed AQM scheme for WLAN, called Proxy-RED, is detailed in Section 5.3. In Section 5.4, simulation results are presented followed by the concluding remarks in Section 5.5.

5.2 An AQM scheme in WLAN

Although congestion due to the speed mismatch between a wired network and wireless LAN (WLAN) at the access point (AP) is regarded as a critical problem that affects overall performance of WLAN, only a handful of research have investigated the AQM issues in WLAN. H. Xu et al. proposed an AQM scheme for WLAN in [72], but the performance analysis of the proposed AQM scheme was limited only to issue of delay (from wired network to WLAN). On the other hand, goodput and packet loss rate are generally accepted as more important metrics in evaluating an AQM performance. In [51], the authors mainly focused on the comparative analysis of different versions of TCPs, particularly TCP veno [24] and TCP reno under RED and Tail-Drop (TD) Queue. This study concluded that RED does not help improve goodput in WLAN. However, it lacks the detailed analysis of the reason why RED can result in the performance degradation.

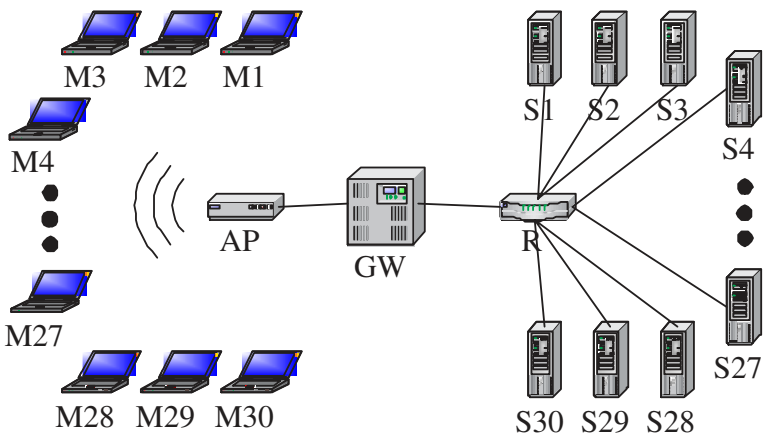
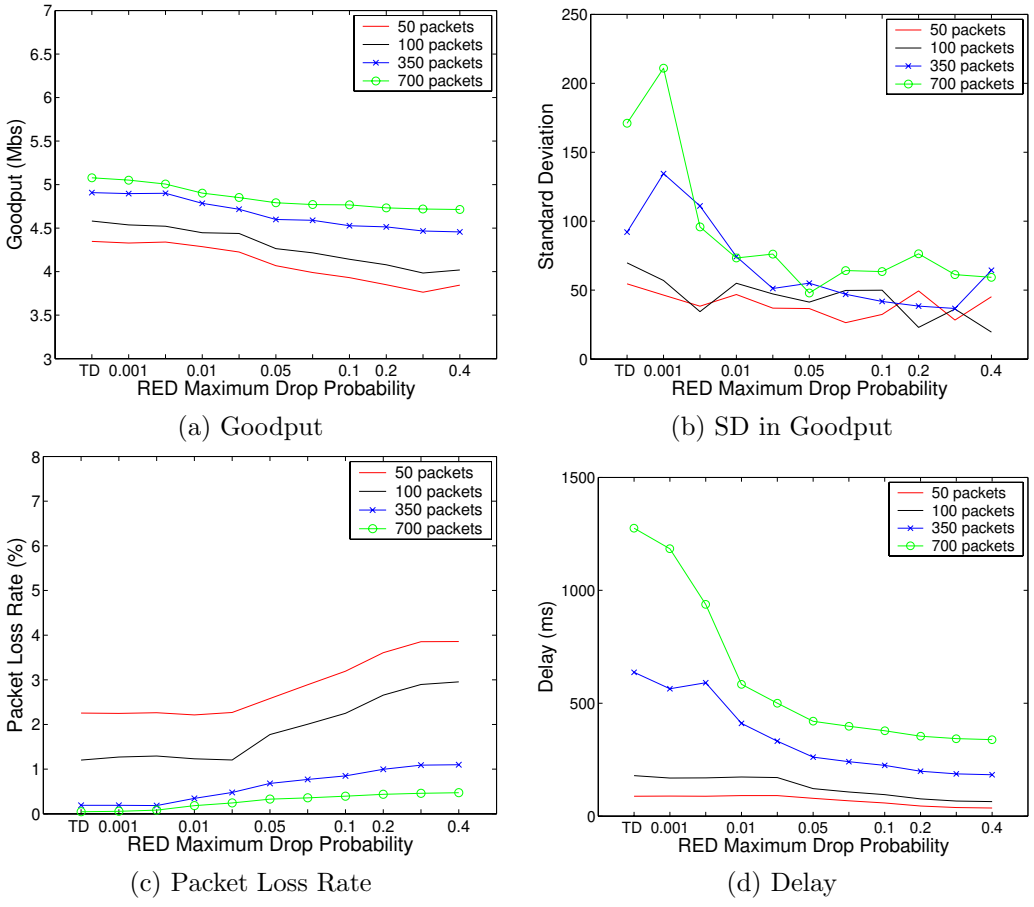


Fig. 5.3. The network topology used for simulation

To investigate the impact of an AQM scheme in WLAN, in particular RED, we performed extensive simulation using ns2 [3] for the network shown in Figure 5.3. In the simulations, all the data packets are generated at the wired nodes and terminate at the mobile nodes except for the ACK packets, and all the sources randomly initiate packet transmission between 150 to 152s (for the ns2 simulator reach the stable state), and connections are terminated 400s later. Each intermediate node has a buffer size of 500 packets, while TCP payload is fixed at 1460 bytes. RED/Tail-Drop is deployed at the the access point whose buffer size varies from 50 to 700 packets, which roughly target up to 1 Mbyte of memory. We set the minimum (θ_{min}) and maximum (θ_{max}) threshold of RED as 30% and 70% of the buffer size respectively, and the maximum drop probability (p_{max}) is varied from 0.001 to 0.4.

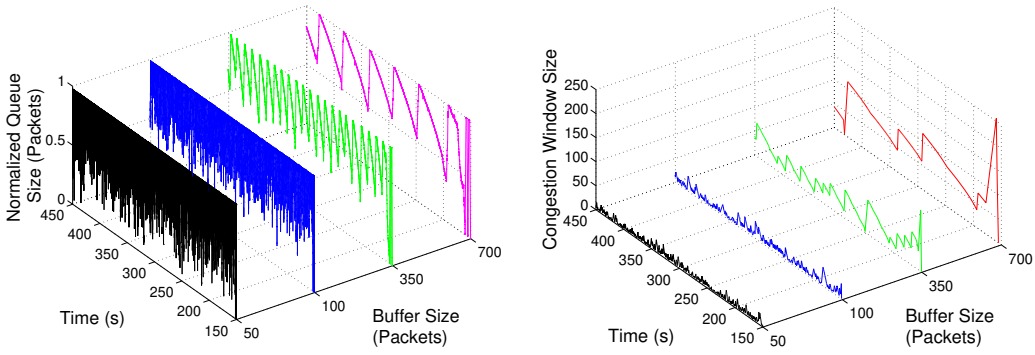
Figures 5.4, 5.5 and 5.6 summarize the simulation results of 10 TCP connections for the downlink (access point to mobile nodes). Although throughput roughly converges



(a) Goodput (b) SD in Goodput

(c) Packet Loss Rate (d) Delay

Fig. 5.4. Tail-Drop and RED with 10 TCP connections

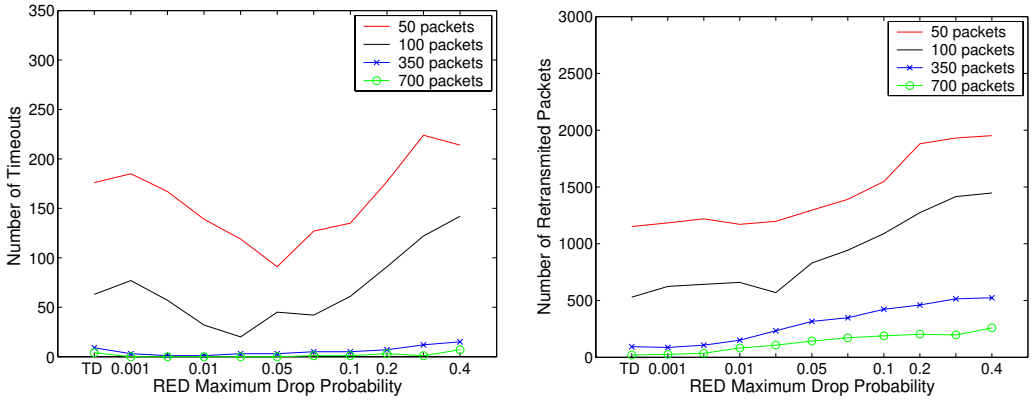


(a) Normalized Queue Behavior (b) Congestion Window Size

Fig. 5.5. Tail-Drop with 10 TCP connections under different buffer size

to 5.25 Mbs regardless of the queue management scheme, gootput and packet loss rate with Tail-Drop in Figure 5.4 are better compared to RED as reported in [51], and the performance degradation increases as p_{max} of RED increases. (Note that the Tail-Drop (TD) results are the leftmost points in most of the graphs.) However, the delay experienced at the access point and the standard deviation for each competing connections are significantly improved with RED.

To get a better understanding of the Tail-Drop/RED behavior in the WLAN, we first investigated the queue behavior and the congestion window size of a connection. In Figure 5.5, for a small buffer size, the instantaneous queue length fluctuated severely and the congestion window size was very small. On the other hand, the queue behavior exhibited the typical saw-tooth shape, and the window size became large for a large buffer size. A large congestion window size and buffer size imply that packet injection is much more bursty resulting in a larger delay compared to a network with a small

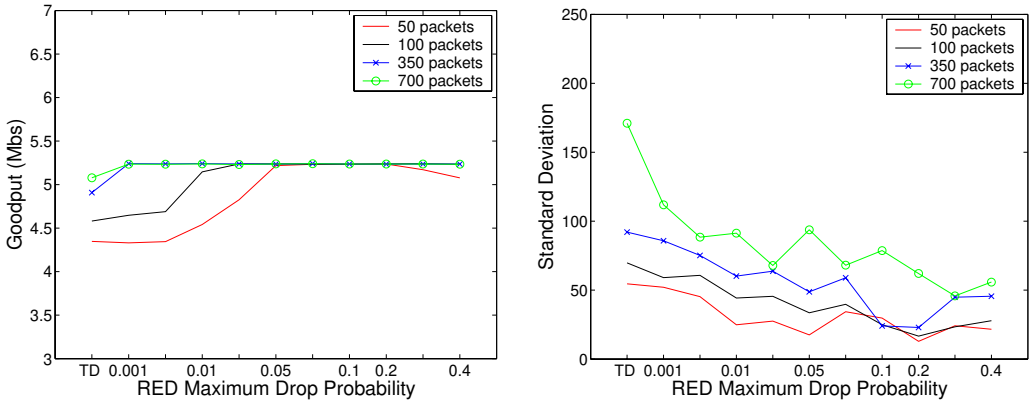


(a) Number of Timeout (b) Number of Retransmitted Packets

Fig. 5.6. The impact of Tail-Drop and RED on TCP sources

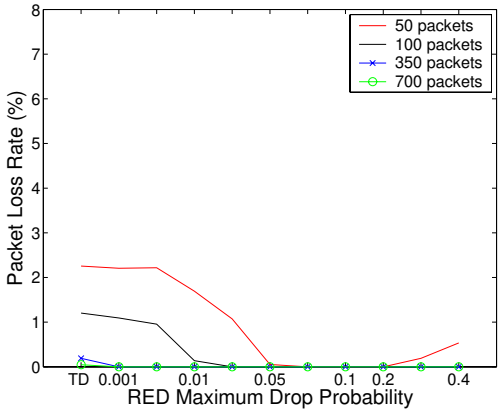
window size and buffer. A large bandwidth delay product induces instability in the TCP/RED mechanism, and it is extremely difficult for RED to find the optimal operating points [56]. Next, we measured the number of TCP time-outs and the number of packets retransmitted at TCP sources with Tail-Drop and RED. Figure 5.6 (a) clearly shows that the number of TCP time-outs decreases with RED for a range of p_{max}, but the number of retransmitted packets increases as p_{max} does in Figure 5.6 (b). This implies that although RED helps in reducing the number of TCP time-outs, the number of dropped packets, i.e., retransmitted packets due to the RED mechanism, is larger than that caused by the buffer overflow with the Tail-Drop Queue. This explains why Tail-Drop shows better goodput and packet loss rate than RED in Figure 5.4.

To reduce the effect of packet loss by the random drop mechanism, we performed the same simulation with the ECN [20] [55] enabled. When ECN is enabled, the RED queue sets the explicit notification (ECN) bit in the header of an arriving packet instead

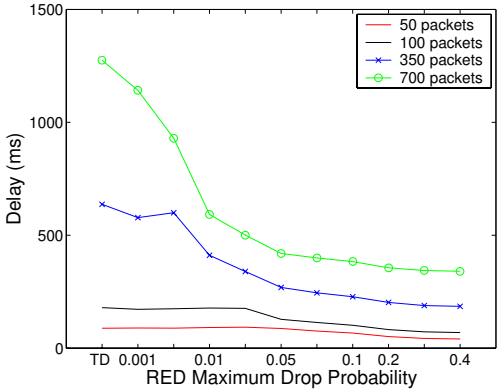


(a) Goodput

(b) SD in Goodput



(c) Packet Loss Rate



(d) Delay

Fig. 5.7. Tail-Drop and ECN enabled RED with 10 TCP connections

of dropping it, and then the receiver copies the ECN bit to the ACK packet to notify the possibility of congestion to the source. When a TCP source receives an ACK packet with the ECN bit activated, it was interpreted as a sign of congestion, and reduced its congestion window size by half of the original value. In Figure 5.7, RED with ECN significantly improves the packet loss rate and goodput compared to Tail-Drop and RED without ECN in Figure 5.4. Also, RED with ECN maintains a comparable delay and standard deviation of RED queue.

In summary, RED does not help in a WLAN environment as was reported in [51], but with ECN, RED can provide better performance than Tail-Drop. Therefore, in the rest of the experiments, we use RED with the ECN marking. However, implementing the RED scheme at an access point may not be practically feasible considering the current architectural trends of *light weight access points*. While all prior studies have considered the impact of RED at the access point, we investigate the modified RED scheme that works on the behalf of the access point.

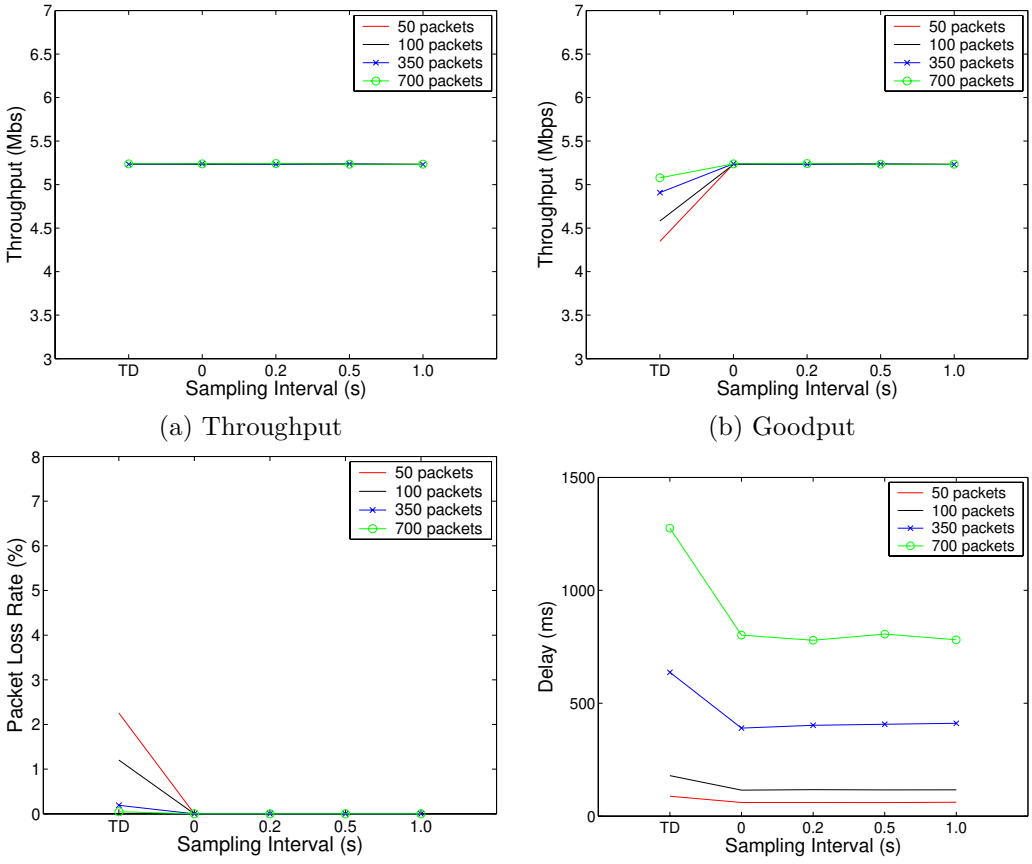
5.3 The Proposed Proxy-RED Scheme

The main idea of proxy AQM is to reduce the overhead of the access point by implementing the AQM functionality at the gateway as shown in Figure 5.2. An AQM scheme deploys a random drop/marking mechanism based on the information such as the average queue length [22] or the outgoing rate [36] that indicates the severity of congestion. In WLAN, the outgoing rate is dependent on the number of mobile nodes, which participate in communication. Therefore, estimating the outgoing rate or the

queue state at a gateway without explicit notification from an access point is a practically difficult problem.

In this thesis, we extend the RED/ARED scheme to a proxy mode since RED is the most widely studied AQM scheme, and is commercially used in CISCO routers. To investigate the feasibility of this idea, we slightly modify ARED as follows. The access point calculates the average queue length and update p_{max} of ARED, and then these values are transmitted to the wireless gateway periodically (every t_{sample} seconds). When the wireless gateway receives these values, it simply deploys a random drop mechanism. The simulation results show that the proposed scheme improves goodput and packet loss rate in Figures 5.8 (b) and (c). Especially, the delay at the access point, which is the most critical factor in deploying VoIP in WLAN, is significantly improved in Figure 5.8 (d) compared to Tail-Drop (TD).

However, calculating the average queue length still takes a significant amount of overhead in the RED/ARED scheme than just deploying the random drop mechanism. As a solution to this problem, we propose to use the sampled instantaneous queue length of the access point to calculate the average queue length at the gateway. One way to achieve this is to block the egress interface on the gateway as long as there is a frame in the access point to transmit. This leads to queue buildup in the gateway (as opposed to the access point) and the gateway can then implement RED in the usual manner. In other words, the service rate of the egress Ethernet interface is made to mimic the wireless egress interface by blocking it from time to time. If the egress Ethernet interface on the gateway is blocked as soon as there is a frame in the access point to be sent out, and is unblocked as soon as that frame is sent on the air, then the



(a) Throughput

(b) Goodput

(c) Packet Loss Rate

(d) Delay at access point

Fig. 5.8. Tail-Drop and ARED with random drop at Gateway

queue in the gateway will exactly mimic the instantaneous queue length in the access point (minus 1). Another approach is to simply transmitting the instantaneous queue length to the gateway periodically (every t_{sample} seconds). Although there are multiple ways to obtain the sampled instantaneous queue length of the access point, we used the later since it is simpler, and the choice of the implementation detail is beyond the focus of our study. Once this is achieved, the gateway can implement RED with the RED parameters configured based on the maximum queue size at the access point.

Here, we slightly modify the ARED's drop function considering the WLAN characteristics. First, we use the smaller maximum drop probability for Proxy-RED. In simulations, we observed that the packet loss rate and goodput started to degrade quickly when p_{max} exceeds 0.1 for 10 and 30 TCP connections. We believe that a large value of p_{max} only results in higher packet loss rate and lower goodput by driving most of the TCP sources to *timeout*. In practice, using more than 30 mobile node for an access point is very rare, and the impact of aggressive packet dropping is much more serious for a network with longer RTTs, such as in WLAN . Therefore, we limit the maximum drop probability (p_{max}) of Proxy-RED to 0.1.

Second, RED/ARED drops all the arriving packets when the average queue length exceed the maximum threshold[1]. Since the instantaneous queue length is periodically transmitted to the wireless gateway, all the connections may suffer from severe packet loss when the estimated queue exceeds the maximum threshold value for the entire sampling interval (t_{sample}). To prevent this undesirable packet loss, we double the p_{max} value

[1]ARED in gentle mode chooses the drop probability between p_{max} and 1.0 when the q_{ave} moves th_{max} to $2th_{max}$. Then, RED/ARED's drop probability is too conservative for the large th_{max}.

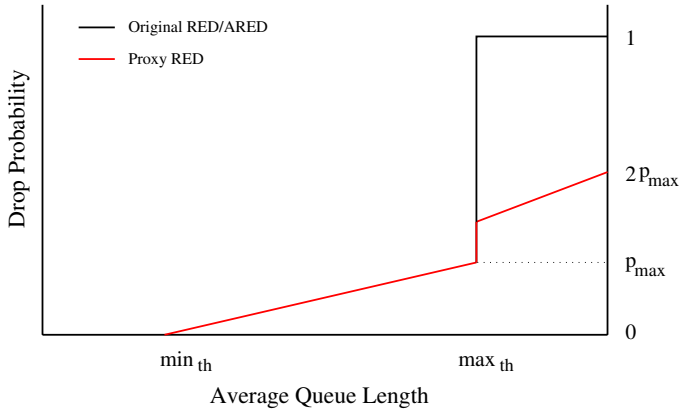


Fig. 5.9. The drop function of RED/ARED and Proxy-RED

when the average queue length exceeds the maximum threshold in Proxy-RED as shown in Figure 5.9.

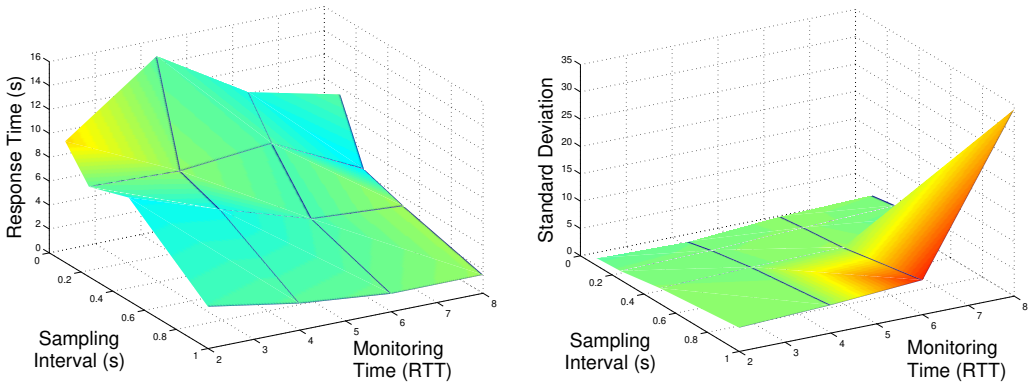
5.4 Performance Evaluation

The average queue length of RED is calculated by $q_{ave} \leftarrow (1 - q_w)q_{ave} + q_w q$, where q_w represents the queue weight, and q and q_{ave} represent the instantaneous and the average queue lengths, respectively. Here, the overall characteristics of the average queue length is dependent on the queue weight. When the queue weight is large, the average queue length fluctuates severely according to the instantaneous queue length. On the other hand, the average queue is more stable, but requires a longer time to reach the steady state behavior for a small value of the queue weight. In [21], the queue weight is determined by the link capacity since the average queue length is calculated on each packet arrival. ARED configures the queue weight to represent a time constant

for the queue average of one second, which is equivalent to 10 RTTs of queue average assuming $RTT = 100ms$, and uses $w_q = exp(-1/C)$, where C is the link capacity (packets/second) for the default configuration. However, Proxy-RED uses the sampled instantaneous queue length of the access point to calculate the average queue length. Therefore, we first investigate the accuracy of the estimated average queue length of the proposed scheme.

In this simulation, we varied the sampling interval (t_{sample}) from 0.1 to 1.0s and configured the queue weight to target 2 to 10 RTTs of queue average[2]. In the beginning, 10 TCP flows started at 150s to 152s and lasted for 300s. Later, 20 TCP flows are added at 250s and lasted for 100s. Figure 5.10 summarizes the simulation results with 100 and 350 packet buffers at the access point. Here, we define the response time as the time between when 20 TCP connections were added to the time when the average queue length reached the peak value. It is clear that the response time was better with a larger sampling interval (t_{sample}). However, the standard deviation of the average queue length was also large. On the other hand, a smaller sampling interval exhibits more stable estimation at the cost of the communication overhead between the AP to the wireless gateway. From the simulations, we found that the estimated average queue length is very accurate when the sampling interval is less then 1 second. For the rest of the simulations, we configured t_{sample} and w_q with 500ms and 0.167 respectively, which represent the estimated average queue length of 6 RTTs since these value showed the best result in our simulation.

[2]We deactivated the adaptive algorithm in the Proxy-RED to eliminate the impact of p_{max}, thus we used $p_{max} = 0.1$ to target maximum 30 TCP flows.



(a) Response time with 100 buffer

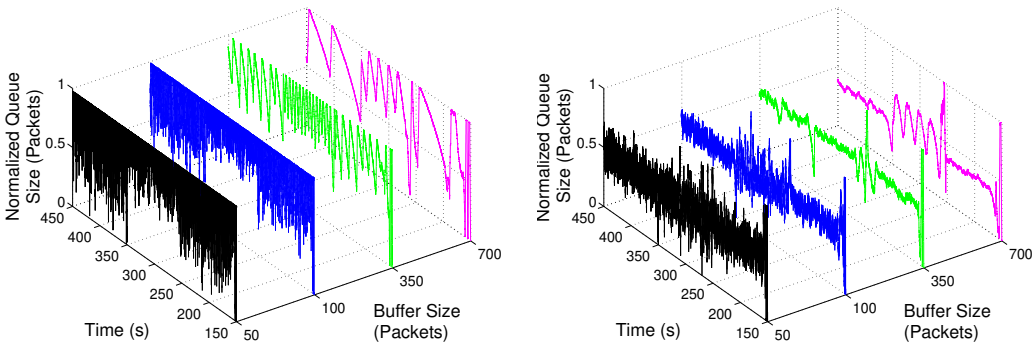
(b) Standard Deviation with 100 buffer

(c) Response time with 350 buffer

(d) Standard Deviation with 350 buffer

Fig. 5.10. Proxy-RED performance for different parameter settings

Next, we activated the adaptive mode for Proxy-RED, and investigated the queue behavior of the Proxy-RED scheme. In Figure 5.11 (a), the access point queue severely fluctuates regardless of the buffer size with the Tail-Drop queue, and suffers from the continuous buffer overflows, especially when traffic is increased. On the other hand, Proxy-RED effectively reduces the buffer overflow, and the queue behavior becomes more stable for the larger the buffer size in Figure 5.11 (b).



(a) Normalized buffer under Tail-Drop (b) Normalized buffer under Proxy-RED

Fig. 5.11. The impact of Tail-Drop and Proxy-RED on queue behavior

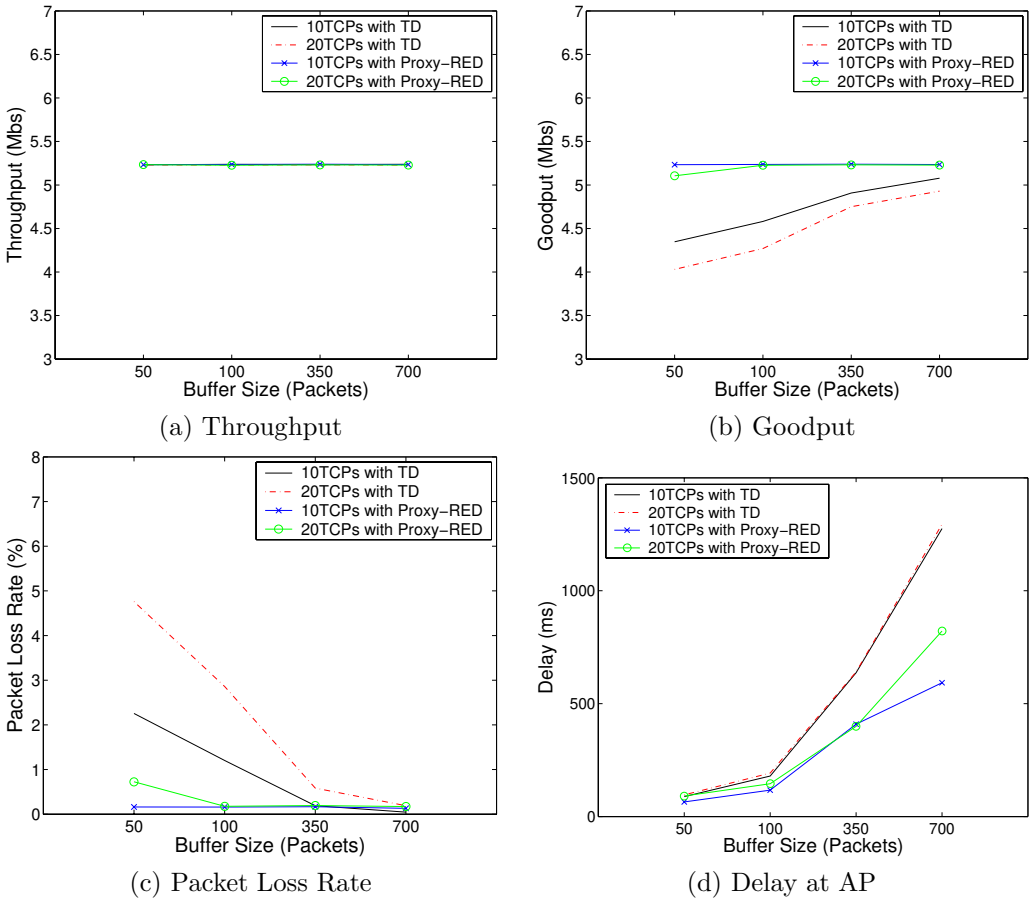
Figure 5.12 finally summarizes the simulation results with 10 and 20 TCP flows with the Tail-Drop queue and Proxy-RED. Although the WLAN throughput is not affected by the AQM scheme, the goodput with Proxy-RED shows significant improvement compared to the Tail-Drop queue in Figures 5.12 (a) and (b). Moreover, the goodput with Tail-Drop queue degrades as the number of a connection increases, but Proxy-RED

effectively maintains a stable goodput regardless of the number of connections when buffer size is larger than 100 packets. Proxy-RED also shows a lower and stable packet loss rate and delay. Although packet loss rate with the Tail-Drop queue significantly is reduced with a larger buffer size, the larger buffer size also contributes to a longer delay in Figures 5.12 (c) and (d). On the other hand, Proxy-RED keeps the packet loss rate under 1% regardless the buffer size. Since the Proxy-RED is based on ARED, which aims to keep the queue size around the middle of the buffer, the delay still increases even with Proxy-RED as the buffer size grows, but not as much as the Tail-Drop queue.

5.5 Concluding Remarks

Although wireless networks based on the IEEE 802.11 standard have been widely deployed in enterprises and university campuses, the congestion arising from the disparity in channel capacity of the wireless and the wired interface of an access point poses a serious challenge in providing Quality of Service (QoS) to the wireless network users. In this chapter, we first studied the feasibility of an AQM scheme to handle the congestion at the wireless access point, and observed that an AQM scheme such as RED can bring significant performance improvement in WLAN when used with ECN marking. We, then, proposed the proxy AQM scheme, called Proxy-RED, that performs the AQM functionality at the gateway on the behalf of wireless access points considering the current architectural trend of *a light-weight access points*.

Simulation results showed that the proposed Proxy-RED scheme can bring overall performance improvement in WLAN. In particular, the Proxy-AQM scheme significantly



(a) Throughput (b) Goodput

(c) Packet Loss Rate (d) Delay at AP

Fig. 5.12. Tail-Drop and Proxy-RED with 10/20 TCP connections

improved packet loss rate and goodput for a small buffer, and delay for a large buffer size.

In the future, we plan to investigate the effect of proxy RED schemes on converged wireless local area networks, i.e., wireless networks used for Voice over IP (VoIP) and data. Initial simulation results indicate that the quality of the VoIP connections could largely benefit from the use of such a scheme. Furthermore, although we limited our focus on the queue length based AQM scheme, i.e., RED, in implementing the proxy-RED scheme in this chapter, our future work will involve a comparative study with other classes of AQM schemes such as AVQ [36] and HRED [74].

Chapter 6

Conclusions

With the proliferation of different types of applications that need customized service, Quality-of-Service (QoS) provisioning has become an active research area in the Internet community. However, the complexity of the Internet traffic dynamics has eluded researchers in finding an efficient solution. The main theme of this thesis is to investigate flow estimation based techniques to support QoS in the Internet.

In this thesis, we investigated five related topics aimed at improving the stability and security of the Internet, which are critical components in supporting QoS in the Internet. First, we developed a flow estimation scheme, called HaTCh, to accurately measure the congestion level since congestion control is an efficient way of improving network stability. In-depth performance evaluation including analytical modelings and extensive simulations showed that the HaTCh scheme improved not only the estimation accuracy and stability, but also improved the robustness of the estimation compared to the SRED mechanism. Second, the HaTCh scheme was extended to design a new AQM scheme (HRED). HRED effectively minimized the dependency of configuration parameters through a new dropping function, resulting in a much stable queue occupancy and low packet loss rate compared to existing AQM schemes such as SRED and ARED. Third, we developed a new DoS defense mechanism, called HaDQ, based on the flow estimations technique to detect and penalize the bandwidth attack of non-responsive

flows. The main advantage of the HaDQ scheme is that it is a scalable technique since it minimizes the per-flow state by exploiting HaTCh's accurate sampling techniques. Simulation based performance evaluation showed that the HaDQ scheme maintained very low false positives, while providing much better protection to conforming users compared to CHOKe and FRED. Then, we developed a worm defense mechanism, called the worm-DQ scheme, which inherited the main advantages of HaTCh and HaDQ. Performance study including various background traffic showed that the worm-DQ scheme was very effective in detecting and penalizing worm scanning activities. Finally, we investigated the impact of an AQM scheme in wireless networks, and presented a new AQM scheme, called Proxy-RED, which is tailored for Wireless Local Area Networks (WLANs). The Proxy-RED scheme is shown to be an effective technique for improving the goodput and delay of access points in WLANs.

In the future, we plan to extend these works in following directions. First, based on the understanding of today's Internet traffic characteristics, we plan to optimize the performance of HaTCh/HRED and HaDQ schemes by properly configuring or by extending them to self-configuring to various traffic conditions. Second, in security measures, false positive is critical since it can cause Denial-of-Service (DoS) to legitimate users. Therefore, diversifying the evaluation tools is important in evaluating security measures. As an on-going work, we plan to perform trace-driven performance evaluations to minimize possible false positives. We believe these techniques, when fully fetched, can provide a viable solution for supporting QoS in Internet.

References

[1] http://www.net.oregonstate.edu/I2/presentations/I2webpics/text0.htm.

[2] The cooperative association for internet data analysis (caida). http://www.caida.org.

[3] Network simulator (Ns). On-line document. Available from http://www.isi.edu/nsnam.

[4] 2002 E-commerce Multi-sector Report. U.S. Census Bureau, April 2004. Available from http://www.census.gov/eos/www/papers/2002/2002finaltext.pdf.

[5] 2004 Global Security Survey, May 2004. Available from http://www.deloitte.com.

[6] Estimated Quarterly U.S. Retail E-commerce Sales. U.S. Census Bureau, May 2004. Available from http://http://www.census.gov/mrts/www/current.html.

[7] R. Addie, M. Zukerman, and T. Neame. Fractal traffic: measurements, modelling and performance evaluation. In *Proceedings of IEEE INFOCOM*, pages 977–984, March 1995.

[8] Aditya Akella, Srinivasan Seshan, Richard Karp, Scott Shenker, and Christos Papadimitriou. Selfish behavior and stability of the internet:: a game-theoretic analysis of tcp. In *Proceedings of the 2002 conference on Applications, technologies, architectures, and protocols for computer communications*, pages 117–130, 2002.

[9] R. Braden, D. Clark, J. Crowcroft, B. Davie, S. Deering, D. Estrin, S. Floyd, V. Jacobson, G. Minshall, C. Partridge, L. Paterson, K. Ramakrishnan, S. Shenker, J. Wroclawski, and L. Zhang. Recommendations on Queue Management and Congestion Avoidance in the Internet. RFC2309, April 1998.

[10] S. Chen and Y. Tang. Internet Quarantine: Requirements for Containing Self-Propagating Code. In *Proceedings of International Conference on Distributed Computing Systems (ICDCS)*, March 2004.

[11] X. Chen and J. Heidermann. Detecting Early Worm Propagation through Packet Matching. USC Technical Report, ISI-TR-2004-585, February 2004.

[12] D. Clark and V. Jacobson. Flexible and efficient reource management for datagram networks. unpublished manuscript, April 1991.

[13] David D. Clark and Wenjia Fang. Explicit Allocation of Best-Effort Packet Delivery Service. *IEEE/ACM Transactions on Networking*, 6(4):362–373, August 1998.

[14] Mark E. Crovella and Azeer Bestavros. Self-Similarity in World Wide Web Traffic: Evidence and Possible Causes. *IEEE/ACM Transactions on Networking*, 5(6):835–846, December 1997.

[15] D. Ely, N. Spring, D. Wetherall, S. Savage, and T. Anderson. Robust Congestion Signaling. In *Proceedings of the International Conference on Network Protocols (ICNP)*, pages 332–341, November 2001.

[16] A. Erramilli, O. Narayan, and W. Willinger. Experimental Queuing Analysis with Long-Range Dependent Traffic. *IEEE/ACM Transactions on Networking*, April 1996.

[17] C. Estan and G. Varghese. New Directions in Traffic Measurement and Accounting. In *Proceedings of the ACM SIGCOMM*, August 2002.

[18] W. Feng, D. Kandlur, D. Saha, and K. Shin. A Self-Configuring RED Gateway. In *Proceedings of IEEE INFOCOM*, pages 1320–1328, March 1999.

[19] W. Feng, D. D. Kandlur, S. Debanjan, and Kang Shin. Stochastic Fair Blue - A Queue Management Algorithm for Enforcing Fairness. In *Proceedings of IEEE INFOCOM*, pages 1520–1529, April 2002.

[20] S. Floyd. TCP and Explicit Congestion Notification. *Computer Communication Review*, 24(5):10–23, October 1994.

[21] S. Floyd, F. Gummadi, and S. Shenker. Adaptive RED: An Algorithm for Increasing the Robustness of RED's Active Queue Management. Under submission, August 2001. Available from `http://www.icir.org/floyd/papers/adaptiveRed.pdf`.

[22] S. Floyd and V. Jacobson. Random Early Detection Gateways for Congestion Avoidance. *IEEE/ACM Transactions on Networking*, 1(4):397–413, August 1993.

[23] Sally Floyd and Kevin Fall. Promoting the use of end-to-end congestion control in the internet. *IEEE/ACM Transactions on Networking*, 7(4):458–472, 1999.

[24] Cheng Peng Fu and Soung Chang Liew. Tcp veno: Tcp enhancement for wireless access networks. *IEEE Journal on Selected Areas in Communications*, 21(2), February 2003.

[25] Y. Gao, G. He, and J. C. Hou. On Exploiting Traffic Predictability in Active Queue Management. In *Proceedings of IEEE INFOCOM*, volume 3, pages 1630–1639, June 2002.

[26] Sachin Garg and Martin Kappes. An Experimental Study of Throughput for UDP and VoIP Traffic in IEEE 802.11b Networks. In *Proceedings of the IEEE Wireless Communications and Networking Conference (WCNC)*, March 2003.

[27] T. M. Gil and M. Poletto. MULTOPS: a date-structure of bandwidth attack detection. In *Proceedings of the USENIX*, August 2001.

[28] S. Ha, S. Han, and V. Bharghavan. A Scalable Router Mechanism for Load Adaptive Fair Packet Dropping. In *Proceedings of IEEE Global Telecommunications Conference*, pages 304–308, November 2000.

[29] H. W. Hethcote. The mathematics of infectious diseases. SIAM Review, October 2000.

[30] C. V. Hollot, Y. Liu, V. Misra, and D. Towsley. Unresponsive Flows and AQM Performance. In *Proceedings of IEEE INFOCOM*, volume 1, pages 85–95, March 2003.

[31] C.V. Hollot, Vishal Misra, Don Towsley, and Wei-Bo Gong. A Control Theoretic Analysis of RED. In *Proceedings of IEEE INFOCOM*, pages 1510–1519, April 2001.

[32] V. Jacobson. Congestion Avoidance and Control. *Proceedings of the ACM SIGCOMM*, pages 314–329, August 1988.

[33] Martin Kappes and Sachin Garg. Can I Add a VoIP Call? In *Proceedings of the IEEE Internationcal Conference on Communications (ICC)*, May 2003.

[34] F.P. Kelly. Charging and rate control for elastic traffic. *European Transactions on Telecommunications*, 8:33–37, 1997.

[35] F.P Kelly, A.K. Maulloo, and D.K.H. Tan. Rate control for communication networks: shadow prices, proportional fairness and stability. *Journal of the Operational Research Society*, 49:237–252, 1998.

[36] S. Kunniyur and R. Srikant. Analysis and Design of an Adaptive Virtual Queue(AVQ) Algorithm for Active Queue Management. In *Proceedings of the ACM SIGCOMM*, pages 123–134, August 2001.

[37] M. Lijenstam, Y. Yuan, BJ Premore, and D Nicol. A mixed abstraction level simulations model of large-scale internet worm infestations. In *Proceedings of the International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems*, October 2002.

[38] D. Lin and R. Morris. Dynamics of Random Early Detection. In *Proceedings of the ACM SIGCOMM*, pages 127–137, September 1997.

[39] Y. Liu, W. Gong, and P. Shenoy. On the Impact of Concurrent Downloads. In *Proceedings of the 2001 Winter Simulation Conference*, pages 1300–1305, December 2001.

[40] R. Mahajan and S. Floyd. RED-PD: Controlling High Bandwidth Flows at the Congested Router. In *Ninth International Conference on Network Protocols*, pages 192–201, November 2001.

[41] M. May, J. Bolot, C. Diot, and B. Lyles. Reasons not to deploy RED. In *Proceedings of IWQoS*, pages 260–262, March 1999.

[42] P. McKenney. Stochastic Fairness Queueing. In *Proceedings of IEEE INFOCOM*, pages 733–740, March 1990.

[43] J. Mo, R. La, V. Anantharam, and J. Walrand. Analysis and Modeling of TCP Reno and Vegas. In *Proceedings of IEEE INFOCOM*, pages 1556–1563, March 1999.

[44] D. Moore, V. Paxson, S. SSavage, C. Shannon, S. Staniford, and N. Weaver. Inside the Slammer Worm. In *IEEE Security and Privacy*, volume 1, pages 33–39, July 2003.

[45] D. Moore, C. Shannon, and k claffy. Code-red: a case study on the spread and victims of an internet worm. In *Proceedings of ACM SIGCOMM Workshop on Internet measurment*, pages 273–284, November 2002.

[46] D. Moore, C. Shannon, G. M. Voelker, and S. Savage. Internet Quarantine: Requirements for Containing Self-Propagating Code. In *Proceedings of IEEE INFOCOM*, March 2003.

[47] Philippe Nain. Impact of Bursty Traffic on Queues. To appear in Statistical Inference for Stochastic Processes, 2001.

[48] T. Ott, T. Lakshman, and L. Wong. SRED: Stabilized RED. In *Proceedings of IEEE INFOCOM*, pages 1346–1355, March 1999.

[49] J. Padhye, V. Firoiu, D. Towsley, and J. Kurose. Modeling TCP Throughput: A Simple Model and its Empirical Validation. In *Proceedings of the ACM SIGCOMM*, pages 632–637, August 1998.

[50] R. Pan, B. Prabhakar, and K. Psounis. CHOKe - A Stateless Active Queue Management Scheme for Approximating Fair Bandwidth Allocation. In *Proceedings of IEEE INFOCOM*, pages 942–951, March 2000.

[51] Q. Pang, S. C. Liew, C. P. Fu, W. Wang, and V O.K. Li. Performance Study of TCP Veno over WLAN and RED Router . In *Proceedings of IEEE Global Telecommunications Conference*, December 2003.

[52] K. Park, G. Kim, and M. Crovella. On the Effect of Traffic Self-similarity on Network Performance . In *In Proceedings of the SPIE International Conference on Performance and Control of Network Systems*, pages 296–310, November 1997.

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

[54] F. Perriot and D. Knowles. Symantec Security Response - W32 Welchia Worm. Available from `http://securityresponse.symantec.com/avcenter/venc/data/w32.welchia.worm.html`, July 2004.

[55] K. K. Ramakrishnan, S. Floyd, and D. Black. The addition of Explicit Congestion Notification (ECN) to IP. RFC3168, September 2001.

[56] J. Wang S. Adlakha J. C. Doyle S. H. Low, F. Paganini. Dynamics of TCP/RED and a Scalable Control. In *Proceedings of IEEE INFOCOM*, volume 1, pages 239–248, June 2002.

[57] J. Wang S. Adlakha J. C. Doyle S. H. Low, F. Paganini. Internet Congestion Control. In *IEEE Control Systems Magazine*, volume 22, pages 28–43, February 2002.

[58] S. Savage, N. Cardwell, D. Wetherall, and T. Anderson. TCP Congestion Control with a Misbehaving Receiver. In *ACM Computer Communication Review*, volume 29, October 1999.

[59] J. Schiller. Mobile Communications. ISBN 0-201-39836-2, 2000.

[60] D. Seeley. A tour of the worm. In *Proceedings of the Winter Usenix Conference*, 1989.

[61] J. Shu and P. Varaiya. Pricing Network Services. In *Proceedings of IEEE INFOCOM*, April 2003.

[62] S. Staniford. Containment of Scanning Worms in Enterprise Networks. to appear Journal of Computer Security, 2004.

[63] S. Staniford, V. Paxson, and N. Weaver. How to own the internet in your spare time. In *Proceedings of the 11th USENIX Security Symposium*, pages 149–167. USENIX Association, 2002.

[64] Symantec. Symantec Security Response - Trojan Horse. Available from `http://securityresponse.symantec.com/avcenter/venc/data/trojan.horse.html`, November 2004.

[65] Symantec. Symantec Security Response - Trojan.Helemoo. Available from `http://securityresponse.symantec.com/avcenter/venc/data/trojan.helemoo.html`, July 2005.

[66] Computer Emergency Response Team. CERT Advisory CA-2001-26 Nimda Worm. Available from `http://www.cert.org/advisories/CA-2001-26.html`, July 2001.

[67] U.S Computer Emergency Readiness Team. US-CERT Current Activity. Available from `http://www.us-cert.org/current/current_activity.html`, September 2004.

[68] J. Turner. New directions in communications, or which way to the information age? *IEEE Communication Magazine*, 24:8–15, October 1986.

[69] N. Weaver. Potential Strategies for High Speed Active Worms: A Worst Cas Analysis. Web Published. Available from `http://www.cs.berkeley.edu/ nweaver/worms.pdf`, March 2002.

[70] M. Williamson. Throttling Viruses: Restricting Propagation to Defeat Malicious Mibile Code. HP Laboratories Technical Report, HPL-2002-172., February 2002.

[71] W. Willinger, M. S. Taqqu, R. Sherman, and D. V. Wilson. Self-Similarity Through High-Variability: Statistical Analysis of Ethernet LAN Traffic at the Source Level. *IEEE/ACM Transactions on Networking*, 5(1):71–86, February 1997.

[72] Heng Xu, Qi Xue, and Aura Ganz. Adaptive Congestion Control in Infrastructure Wireless LANs with Bounded Medium Access Delay. In *Proceedings of the International Mobility and Wireless Access Workshop*, October 2002.

[73] S. Yi, X. Deng, G. Kesidis, and C. R. Das. HaTCh: A Two-level Caching Scheme for Estimating the Number of Active Flows. In *Proceedings of IEEE Conference on Decision and Control (CDC)*, volume 3, pages 2829–2834, December 2003.

[74] S. Yi, X. Deng, G. Kesidis, and C. R. Das. A Dynamic Quarantine Scheme for Controlling Unresponsive TCP flows. The Pennsylvania State University Technical Report CSE04-004, February 2004.

Vita

Sungwon Yi was born in Seoul, Korea, on July 3 1969. He received the B.E. degree from the Department of Information and Control Engineering, Kwangwoon University in Korea 1995. From 1995 to 1998, he worked for LG-CNS (former LG-EDS) as a system engineer, where he developed a carrier settlement system, a sub-system of the Korea Telecom's new billing system. In 1999, He joined the Department of Computer Science and Engineering the Pennsylvania State University, and received an M.S degree for his thesis, *Providing Fairness in Diffserv Architecture*, in 2004. He is currently a Ph.D candidate in the Department of Computer Science and Engineering, the Pennsylvania State University. His research interests include the areas of Computer Networks, with emphasis on Internet congestion control and Quality-of-Service (QoS), Network Security (DoS and WORM defense), and Mobile Computing. He has served as a technical referee for numerous journals and conferences including IEEE Infocom, Globecom, and IEEE Transactions on Multimedia. He also served as a member of technical program committee of GLOBECOM 2005.