ENHANCED QUEUE MANAGEMENT MECHANISM FOR DIFFERENTIATED SERVICES NETWORKS

MAJID HAMID ALI ALBAYATI

UNIVERSITI UTARA MALAYSIA
2012

ENHANCED QUEUE MANAGEMENT MECHANISM FOR DIFFERENTIATED SERVICES NETWORKS

A project submitted to Dean of Awang Had Salleh Graduate School in Partial Fulfilment of the requirement for the degree Master of Science in Information Technology Universiti Utara Malaysia

By

MAJID HAMID ALI ALBAYATI

KOLEJ SASTERA DAN SAINS
(College of Arts and Sciences)
Universiti Utara Malaysia

PERAKUAN KERJA KERTAS PROJEK
(Certificate of Project Paper)

Saya, yang bertandatangan, memperakukan bahawa
(I, the undersigned, certifies that)

MAJID HAMID ALI
(806281)

calon untuk Ijazah
(candidate for the degree of) **MSc. (Information Technology)**

telah mengemukakan kertas projek yang bertajuk
(has presented his/her project of the following title)

ENHANCED QUEUE MANAGEMENT MECHANISM FOR DIFFERENTIATED SERVICES NETWORKS

seperti yang tercatat di muka surat tajuk dan kulit kertas projek
(as it appears on the title page and front cover of project)

bahawa kertas projek tersebut boleh diterima dari segi bentuk serta kandungan dan meliputi bidang ilmu dengan memuaskan.
(that this project is in acceptable form and content, and that a satisfactory knowledge of the field is covered by the project).

Nama Penyelia Utama:
(Main Supervisor) : **DR. MASSUDI MAHMUDDIN**
Tandatangan
(Signature) : _____ Tarikh (Date) : _____

DR. MASSUDI BIN MAHMUDDIN
Senior Lecturer
School of Computing
UUM College of Arts and Sciences
Universiti Utara Malaysia

Nama Penyelia Kedua:
(Co-Supervisor) : **DR. MOHAMMED M. KADHUM**
Tandatangan
(Signature) : _____ Tarikh (Date) : _____

Dr. Mohammed M. Kadhum
Visiting Senior Lecturer
Information Technology
UUM College of Arts and Sciences
Universiti Utara Malaysia
06010 UUM Sintok, Kedah.

Nama Penilai 1
(Name of Evaluator 1) : **PROF. DR. RAHMAT BUDIARTO**
Tandatangan
(Signature) : _____ Tarikh (Date) : 16.01.2012

Nama Penilai 2
(Name of Evaluator 2) : **MR. SHAHRUDIN AWANG NOR**
Tandatangan
(Signature) : _____ Tarikh (Date) : 17/1/2012

PERMISSION TO USE

In presenting this project in partial fulfilment of the requirements for a postgraduate degree from the University Utara Malaysia, I agree that the University Library may make it freely available for inspection. I further agree that permission for copying of this project in any manner in whole or in part, for scholarly purposes may be granted by my supervisor(s) or in their absence by the Dean of Postgraduate Studies and Research. It is understood that any copying or publication or use of this project or parts thereof for financial gain shall not be allowed without my written permission. It is also understood that due recognition shall be given to me and to University Utara Malaysia for any scholarly use which may be made of any material from my project.

Requests for permission to copy or to make other use of materials in this project, in whole or in part, should be addressed to

Dean of Awang Had Salleh Graduate School

College of Arts and Sciences

Universiti Utara Malaysia

06010 UUM Sintok

Kedah Darul Aman

Malaysia

ABSTRACT

In the Internet, it is supposed that all connections are treated equally in the network. Due to the limitation of network resources are limited, providing guarantees on performance measures imposes declining new connections if resources are not available. Assigning network resources to connections according to their classes requires differentiating between the connection classes. For this reason, the Differentiated Services (DiffServ) has been proposed. Many of the QoS mechanisms have been developed which allow different services carried by the Internet to co-exist. Many of these mechanisms were both complex and failed to scale to meet the demands of the Internet. MRED is the common mechanism used in DiffServ routers. It suffers from large queue length variation and untimely congestion detection and notification. These consequences cause performance degradation due to high queuing delays and high packet loss. In this project, enhanced version of MRED is developed to improve the performance of DiffServ networks that use TCP as the transport layer protocol. Enhanced MRED includes average packet arrival rate when computing the packet drop probability. Enhanced MRED showed a good performance compared to that of MRED, in term of fast congestion detection and notification. The limitation of the new mechanism is that it works only with responsive connections which play a big role in avoiding and controlling the congestion. The major contribution of this project is to provide an improved queue management mechanism for DiffServ networks that responds to congestion more quickly, delivers congestion notification timers, and controls the queue length directly to congestion which results in minimizing queue length variation. All these would help improve the DiffServ networks performance.

ACKNOWLEDGEMENTS

Praise to Allah for his guidance and blessing for giving me the strength and perseverance to complete this project. I would like to thank my supervisors: Dr. Massudi bin Mahmuddin and Dr. Mohammed M .Kadhum for all the guidance, stimulus, and practical advice provided over the past time and he gave me many interesting, valuable and sincere feedbacks throughout his supervision. I am thankful to him for his support and motivation without which completion of the work presented in this project would not have been possible. I shall always remember Dr. Muhammad for the efforts he has spent in strengthening my understanding about topics related to my research, and giving me enough leeway to help me in managing my research.

I am also thankful to the Information Technology Department – the faculty and staff. Being a postgraduate student at UUM has been an incredible experience. I shall always remember the time I have spent here as one of the best phases of my life. I wish to thank the Ministry of Higher Education of Iraq for their financial support awarded to me. I am thankful to all friends, whose love, blessings and well wishes have shown me the success that I have achieved in the form of this master's degree. Finally yet importantly, I am extremely grateful to my beloved father, my affectionate mother, and my precious brothers who always provided me the encouragement to acquire the education I wanted. Special thanks are due to my faithful wife, and my two kids, Layth, and Lena. Without your love and support I am sure that I would not have been able to achieve so much throughout the two years of my study abroad. I dedicate the accomplishment of this project to my beloved father, my affectionate mother, and to the twin of my spirit, my wife. May Allah bless all of you.

TABLE OF CONTENTS

LIST OF FIGURES

APPENDIX

CHAPTER ONE
INTRODUCTION

This project is about enhancing the queue congestion management mechanism used in Differentiated Services environment to help providing good quality of service to end users based on their requirements. The goal of this chapter is to place the project in its context. In this chapter, an introduction to Differentiated Services, its issues, and the role of queue management mechanism in enhancing the network performance are provided in Section 1.1 and 1.2, respectively. The research problem is presented in Sections 1.3. Sections1.4, 1.5, and 1.6 of this chapter, respectively, include the research questions, research scope, and objectives of the research presented in this project. The importance of the work done in this project is stated in Section 1.7 while the project organization is presented in Section 1.8 of this chapter.

1.1 Introduction

In recent years, important investments have been made in the planning and development of computer networks. The rapid growth of the Internet provides a good opportunity for creating new mechanisms for internet infrastructure to service the increase of new applications, such as web surfing, network monitoring, desktop sharing and video conferencing. The delay variations in network system affect network applications. In an acknowledgement and time-out-based congestion control mechanism, e.g., TCP, performance is related to the delay-bandwidth product of the connection (Durresi, Sridharan, Jain, Liu, & Goyal, July 2001). Furthermore, TCP round-trip time (RTT) measurements are sensitive to delay variations, which may cause wrong timeouts and retransmissions.

Internet Protocol (IP) based network was designed to provide users with best effort service that allows user packets to share network resources. The rapid increase in the IP applications resulted in a significant burden on restricted network resources, such as bandwidth and buffer space, leading to high degree of congestion (Qadeer, Sharma, Agarwal, & Husain, 2009). IP applications, such as real-time and mission-critical, are the influenced ones due to high packet loss in the network. In the Internet, all sources get the same handling in the network. While network resources are limited, providing guarantees on performance measures requires rejecting new connections if resources are not available. To assign resource to connections according to their class, connection classes should be differentiated. Therefore, IP Quality of Service (QoS) was developed to allow network operators offering different levels of packets treatment according to user requirements. QoS Routing, as defined in (Qadeer et al., 2009), is a routing mechanism as per which paths for flows are established based on some knowledge of resource availability in the network as well as the QoS requirement of flows. It attempts to perform routing by computing multiple paths between two nodes which could satisfy different service requirements; and change routing when the availability of resources in the shortest path is insufficient.

Since Internet carries many different types of services, including voice, video, streaming data, web pages and email, many of the proposed QoS mechanisms that allowed these services to co-exist were both complex and failed to scale to meet the demands of the Internet. For that, the Differentiated Services (DiffServ) has been proposed.

1.2 Differentiated Services (DiffServ)

Differentiated Service (DiffServ) is an IP QoS architecture that allows prioritizing packets according to the type of service the user desires. According to (Kimura, Kamei, & Okamoto, 2002), DiffServ is a computer networking architecture that specifies a simple, scalable and coarse-grained mechanism for classifying, managing network traffic and providing quality of service (QoS) guarantees on modern IP networks. DiffServ can, for example, be used to provide low-latency, guaranteed service (GS) to critical network traffic such as voice or video while providing simple best-effort traffic guarantees to non-critical services such as web traffic or file transfers.

By marking packets at the edge of the network according to the performance level that the network wishes to provide them, the network's nodes treat the packets differently (El Hachimi, Abouaissa, Lorenz, & Sathya, 2003). A general way to distinguish packets is by using RED buffers and use different parameters for different packets (Stankiewicz & Jajszczyk, 2007). Thus, applications over the internet could benefit of lesser delays and larger throughputs.

A packet belonging to a flow may get three possible priority levels within the flow. This can be used to provide a lower loss probability to SYNC packets in a TCP connection, as in contrast with other packets, the losses of SYNC packets result in very long time-outs. Additional to differentiation within each flow, all flows are grouped to some classes (not more than four), and different treatment can be given to different classes (Peng, Hongchao, Binqiang, & Hui, 2009).

Furthermore, it is possible to differentiate between flows. Four classes of flows are defined, and packets of a given class are queued in a class-dependent queue. To differentiate between packets belonging to same class, three virtual queues are implemented in each of the four queues. To

each of the 12 combinations of the four flow class and the three internal priority levels within a flow correspond a code point that a packet is given when entering the network. Actually, not all queues and all priority groups need to be implemented (Qian, 2008).

Nodes in DiffServ environment are equipped with some functional units that allow Per-Hop Behaviors (PHBs), packet classification, marking, shaping, and policing. The encodings recommended for DiffServ enable a network operator with great flexibility in defining different classes of data traffic. Actually, network operators can configure their networks according to any of the following commonly-defined Per-Hop Behaviors:

- Default PHB (Per hop behavior)—which is typically best-effort traffic
- Expedited Forwarding (EF) PHB—dedicated to low-loss, low-latency traffic
- Assured Forwarding (AF) PHB—gives assurance of delivery under prescribed conditions
- Class Selector PHBs—which maintain backward compatibility with the IP Precedence field.

The DiffServ is scalable because of that tasks, such as multi-flow classification, policing, shaping and marking, are performed at the border (edge) routers networks. This is because the border routers deal with the end user links that are slow as a result of which it has time to do the costly functions like Multi-field Classifier (MFC) and traffic conditioning as mention in (Sundaresan, 1999). In contrast, core routers do the forwarding according to the DiffServ Code Point (DSCP) stated in the packet header. DSCP is the first six bits in the Type of Service byte in the IP header.

According to Lain-Chyr et al., Assured Forwarding (AF) (Lain-Chyr, Hsu, Cheng-Yuan, & Chun-Shin, 2004) can provide many QoS services, and compared to the Expedited Forwarding (EF) PHB (Makkar et al., 2000), Assured Service provides a statistical bandwidth guarantee to end users, and allows them to claim a share of the excess network bandwidth in addition to the subscribed bandwidth. The main problem in the process of assured forwarding is the stability of the average queue length and the latency. Solving this problem can be done by deploying the RED with In and Out (RIO) algorithm (or other improved RIO queue management) in the interior node. However, several studies (Du, Qiu, & Guo, 2009) found that these algorithms are hard to set parameters and easy to make some mistakes, the vibration of the average queue size and the latency are big when the speed changes.

1.3 Problem Statements

DiffServ-capable router utilizes virtual buffers called MRED (multi-RED) (Jahon, Byunghun, Kwangsue, Hyukjoon, & Hyunkook, 2001) in each physical queue allows to its performance and to create dependence between their operation. MRED probability of dropping each packet is based on the size of its virtual queue (Qadeer et al., 2009). MRED drop probability function uses the average queue length, which is collected over long period, to make its control decisions. However, the use of average queue length makes MRED reacts to congestion slowly. This results in large queue length variation and untimely congestion detection and notification which would cause performance degradation due to high queuing delays and high packet loss (Nagendran, Kartick, Sayee Ram, SenthilKumar, & Sudha, 2010). MRED suffers from low bandwidth utilization, low throughput under poorly setting parameters, and large queuing delay variance (jitter) because of the fluctuation of the queue level, being unable to handle

unresponsive connections, and high number of consecutive drop. Thus, the quality of service observed by the end system is lowered significantly.

1.4 Research Questions

i. How can we enhance the queue management mechanism used in DiffServ-capable router that ensures good quality of service?

ii. How can we evaluate and validate the DiffServ-capable router that employs the enhanced the queue management mechanism?

1.5 Research Scope

This research focuses on improving Multi-Random Early Detection (MRED) queue management mechanism in order to improve the performance of differentiated service environment. Therefore, the focus will be on developing a drop probability function for MRED queue management mechanism utilized by in the DiffServ-capable routers. The implementation of the mechanisms and all the test experiments will be performed using the version 2.32 of the network simulator (NS-2) software on a machine running the CentOS 5.2 version of the Linux operating system.

1.6 Research Objectives

The aim of this research is to improve the performance of differentiated service network by enhancing DiffServ-capable router scheduling mechanism. In order to achieve this research aim, we come up with the following research objectives:

i. To develop an enhanced drop probability function for DiffServ-capable router that utilizes MRED mechanism in order to improve the throughput and decrease packet drop.

ii. To implement the enhanced MRED mechanism in simulated differentiated service network by using network simulator 2 (ns-2).

iii. To analyze the results obtained from the simulations in terms of throughput, packet loss, queue length, and link utilization.

1.7 Research Significance

DiffServ has been introduced to differentiate between connection classes and to allocate resources to connections according to their class. Therefore, As DiffServ is based on marking data packets at the edge router of the network according to the performance level (quality of service) that the network wishes to provide, packets are handled differently at the network routers. This requires efficient and reliable buffering and scheduling mechanism to meet the user or subscriber requirements. Enhanced MRED mechanism proposed in this project can help improving DiffServ performance .

1.8 Organization of the Project Report

This project is organized in six chapters as follows:

Chapter 1 provides an overview of the project. It presents an introduction to the importance of queue management mechanism and the need for improving the current mechanism used in DiffServ network's routers. The chapter presents the objectives and contributions of this project as well.

Chapter 2 is a literature review that contains a background material on Quality of Service and queue management in DiffServ that defines the general framework for this research. The issues in DiffServ and the efforts done to alleviate them are covered in this chapter.

Chapter 3 presents the experimental tools and methodologies. The former introduces popular TCP/IP performance measurement tools, such as ns-2, with description of their usage while the latter covers network topology and settings used in the experiments. It presents the development of the enhanced MRED mechanism in the simulation.

Chapter 4 introduces the enhanced MRED mechanism proposed in this project. The chapter describes the details of enhanced MRED's structural design. It discusses the enhanced mechanism implementation issues as well.

Chapter 5 presents a detailed performance evaluation of the enhanced MRED mechanism based on the numerical results obtained through simulations. It studies the behavior of the enhanced MRED a performance comparison of enhanced MRED to RED.

Chapter 6 states the global conclusions of the research work presented in this project and provide some suggestions for further studies.

CHAPTER TWO
LITERATURE REVIEW

While the issues of Differentiated Services (DiffServ) environment were generally described in Chapter 1, this chapter provides the background and some related research on queue management in DiffServ network that defines the general framework of this research. This chapter explains the function of queue management mechanism in performance optimization of DiffServ networks that utilize TCP/IP protocols; and it provides performance analysis of the current mechanism used in DiffServ. In this chapter, the Quality of Service (QoS) concept is presented in Section 2.1 A general introduction about DiffServ architecture is presented in Section 2.2 Queue management ant its important role in DiffServ is presented in Section 2.3 Multiple RED queue management mechanism is presented in Section 2.4. The details of RED queue management algorithm including its function and structure are covered in Sections 2.5, 2.6, 2.7, 2.8, and 2.9 respectively. Section 2.6 summarizes the topics covered in this chapter.

2.1 Quality of Service

Quality of Service (QoS) QoS is defined as the proficiency of a network element to furnish some degree of commitment for congenial network data delivery as stated in (Qadeer et al., 2009). Network should meet the service requirements of QoS when transporting packets from a source to their destination. QoS goals are to meet the user application requirements, providing a network that is transparent to its users.

According to Qadeer et al. (Qadeer et al., 2009),The common QoS factors are:

- Bandwidth: the average usable and available bandwidth over the link at any time;
- Delay: the average end-to-end delay caused at network level at any time;
- Delay jitter: the average difference of the various delay times over the link;
- Packet loss probability: the average probability of packet loss over the link over a length of time.

Over Internet, different applications have different requirements for packet loss, delay, and bandwidth. Network service providers (ISPs) provide QoS to users based on an agreement between them. The agreement is known as a Service Level Agreement (SLA). QoS manage traffic across a network according to the applications requirement. Some applications, such as voice applications, require bandwidth and delay guarantees, referred to as quantitative applications, while others, such as file transfer applications, are more qualitative. Voice applications have strict delay requirements and can tolerate minimum packet loss. On the other hand, a file transfer application is very sensitive to packet drops but can endure delays. According to Xipeng et al. (Xipeng & Ni, 1999), to deal with such differences, QoS assigns flows to one of the following two categories:

- Guaranteed service keeps a specific amount of bandwidth from end to end and can guarantee a specified delay tolerance for the exclusive use of an application or even aggregated sessions.
- Differentiated service provides simple prioritization. Applications are detected at the ingress and assigned SLAs, which in turn decide the QoS mechanisms to be employed by the router, like which queue will be used to place traffic, and which drop priority will be designated in case of congestion requiring a packet drop.

2.2 Differentiated Services (DiffServ) Architecture

DiffServ uses six bits of the DS field in the IP header to make up the DSCP (Differentiated Service Code Point) field. DSCP is used to select the per-hop behaviour (PHB) a packet experiences at each node. The mapping of DSCPs to PHBs at each node is not fixed. Before a packet enters a DiffServ domain, its DSCP field is marked by the end-host or the first-hop router according to the service quality the packet is required and entitled to receive. Within the DiffServ domain (see Figure 1), each router only needs to look at DSCP to decide the proper treatment for the packet. No complex classification or per-flow state is needed. DiffServ has two important design principles, namely pushing complexity to the network boundary and the separation of policy and supporting mechanisms. The network boundary refers to application hosts, leaf (or first hop) routers, and edge routers. Figure 2.1 shows the DiffServ domain that includes

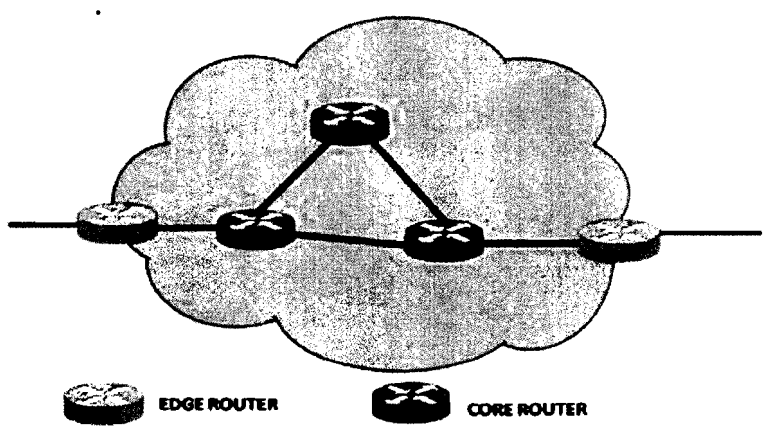

Figure 2.1. DiffServ Domain

DiffServ design has the following elements:

- **Policy and resource manager:** it creates policies and distributed them to DiffServ routers in the DiffServ domain. A policy determines which level of services in the network is assigned to which packets. This assignment may depend on the behavior of the source of the flow (e.g. its average rate and its burstiness) and special network elements are therefore added at the edge of the network so as to measure the source behavior.

- **Edge routers**: responsible to mark (assign code points) packets according to the policy specified by the network administrator. The mark that a packet receives identifies the class of traffic to which it belongs. After being marked, a packet may then be immediately forwarded into the network, delayed for some time before being forwarded, or it may be discarded.

- **Core routers**: When marked packet arrives at DiffServ-capable router, the packet is forwarded to its next hope according to the per-hop behavior associated with that packet's class. Routers within the network have to assign the right priority to packets according to their code mark. The priority translates to parameters of scheduling and of dropping decisions in the core routers.

2.3 Queue Management in DiffServ Network

Queue management is essential to provide good quality of service to end users. Queue management enables bandwidth control traffic treatment. Therefore, two queue types are needed. They are *weighted fair bandwidth distribution* and *priority*.

Depending on the available buffer space and the desired steady state queue length at the router, packets are admitted to the router. Queue management mechanisms allocate the available buffer space at the router between the flows being multiplexed over the outgoing transmission links and control the length of the packet queues created within the buffers. This is accomplished by deciding whether admitting the newly arrived packet to the router or discarding it from the network. Incoming packet may be allowed to enter the router without changing the queue status or it may be admitted after dropping a packet from the queue, or the arriving packet itself may be dropped as highlighted by Xiaojie et al. (Xiaojie, Kamal, & Leonard, 2004).

To guarantee that higher priority flows are given priority over lower priority ones, the queue management mechanism used in the router drops lower priority packets when congestion occurs. Currently, a modified RED queue management mechanism, namely Multiple RED (MRED), is used in DiffServ networks

2.4 Multiple RED Queue Management Mechanism

DiffServ provides QoS by classifying traffic flows into different categories. Each packet is marked with a code point indicating its unique category. Packets are scheduled according to their code points. The *Assured Forwarding* mechanism (Bianchi & Blefari-Melazzi, 2001) is a group of code points that can be used to classify four classes of traffic in a DiffServ network. Each class has three drop precedences that enable traffic treatment within a single class. Assured Forwarding uses, redQueue, a modified RED which put all packets for a single class in one physical queue. This physical queue is consists of three virtual queues (see Figure 2.2), one for each drop precedence, or RED queues (called Multi RED). MRED can have more than one physical queue.

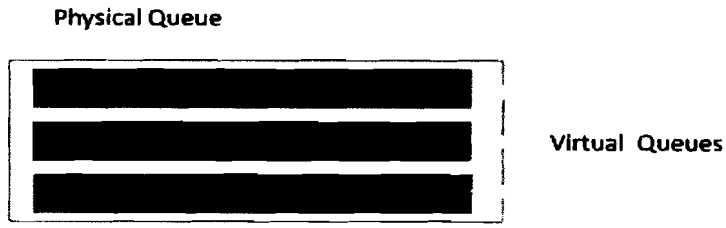

Figure 2.2 Single Physical Buffer in DiffServ

Assured Forwarding is recommended for applications that need a better reliability than the best-effort service. Assured Forwarding Service is implemented where classification and policing are done at the ingress routers of the ISP networks. If the Assured Service traffic does not exceed the bit-rate specified by the SLA, they are considered as in profile. Otherwise, the excess packets are considered as out of profile. All packets, in and out, are put into an Assured Queue to avoid out of order delivery. The queue is managed by a queue management mechanism called RED with In and Out, or RIO.

Assured Forwarding mechanism (Makkar et al., 2000) provides different levels of forwarding assurances for IP packets by dropping more packets that have low priority compared to packets with high priority.

MRED has many versions such as Rio Coupled (RIO C), in which the probability of dropping low priority packets, called "out-of-profile packets", is based on the weighted average lengths of all virtual queues (Yang, Chen, & Zhang, 2011), whereas the probability of dropping a high priority ("in-profile") packet is based only on the weighted average length of its own virtual queue (Yang, Chen, & Zhao, 2008). It basically maintains two RED algorithms, one for in packets and one for out packets. There are two thresholds for each queue. When the queue size is below the first threshold, no packets are dropped. When the queue size is between the

two thresholds, only out packets are randomly dropped. When the queue size exceeds the second threshold, indicating possible network congestion, both in and out packets are randomly dropped, but out packets are dropped more aggressively.

Another version of MRED is called RIO De-couple (RIO D) (Wen-Ping & Zhen-Hua) which has the probability of dropping each packet is based on the size of its virtual queue. Another version is the WRED (Weighted RED) in which all probabilities are based on a single queue length (Bianchi & Blefari-Melazzi, 2001) . It is possible to use the dropTail queue.

While MRED mechanism is RED with multi virtual queues, this means that MRED inherits all problems associated with RED. In the following section, the Random Early Detection (RED) mechanism is reviewed.

2.5 Random Early Detection (RED)

RED uses a Triple Threshold Average Queue Occupancy Level (*Minth*, *Maxth*, 2 *Maxth*) activation function for congestion detection. Using two thresholds in Basic RED and three thresholds in Gentle RED, enables RED to differentiate between different congestion levels experienced at the router based on the router queue condition. The use of average queue length allows RED to better differentiate between temporary queue oscillations due to short-term data traffic increases and persistent queue growth due to long-term data traffic overload. As concluded by Nga (Nga, Iu, Ling, & Lam, 2008), this enables RED to properly detect persistent congestion and to house and endure short term data traffic increases even though the algorithm's ability for early congestion detection is compromised and the average queue length is controlled.

RED routers accept all incoming packets until the queue length reaches Minth, and then it drops a packet with a linear distribution function. When the queue length reaches Maxth, all incoming packets are dropped with probability one. RED router buffer is shown in Figure 2.3

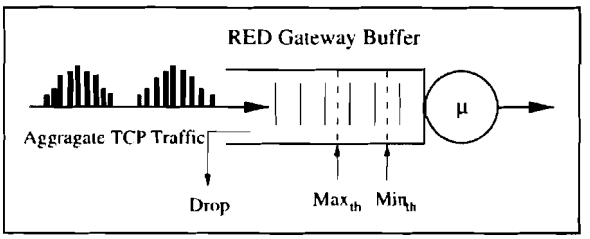

Figure 2.3 RED Router Buffer (Adopted from (Ryu, Rump, & Qiao, 2004))

A router implementing RED detects congestion early by computing the average buffer length avg and sets the two queue thresholds *Maxth* and Minth for packet drop. Upon the arrival of a new packet to the router, RED calculates the average queue length using EWMA process. The average queue length is defined as

$$avg = (1 - w)\, avg + wq \qquad (2.1)$$

where *avg* is the new value of the average queue length at a given time, q is the current queue length, and w, which is normally less than one, is the weight parameter that is used for calculating the average queue length, avg, from the instantaneous queue length as stated in (Ryu et al., 2004). *ave* is used as a control variable to activate packet drop/mark process. The average buffer length tracks the current buffer length. However, the average queue length fluctuates much slower than q because w value is much less than one. Thus, the average queue length tracks the long-term fluctuations of q to reflect the congestion in networks.

As mentioned in (Floyd, November 1997), the packet dropping probability function determines how frequently the router will send congestion notifications by dropping packets, given the current level of congestion. The probability function of RED allows it to tune the packet dropping probability based on the congestion level at the router. The level of the congestion is proportional to the level by which the lower threshold is exceeded by the average queue length. RED gives proper packet drop/mark probabilities to different packets depending on the average queue length, packet length, and the number of undropped/unmarked packets since last dropping/marking.

According to Mahbub et al. (Mahbub & Raj, 2003), RED algorithm includes two computational parts:

- Computation of the average queue length
- Computation of the packet drop/mark probability

In RED, the initial packet drop/mark probability (Pini) is computed as a linear function of the average queue length which reflects different congestion severity levels. The larger the computed average queue length is, the greater the probability with which an incoming packet is dropped or marked.

The RED initial packet drop/mark probability, Pini, is calculated by

$$P_{ini} = \left(\frac{Max_{drop}}{Max_{th} - Min_{th}} \right) * avg - \left(\frac{Min_{th} * Max_{drop}}{Max_{th} - Min_{th}} \right) \quad (2.2)$$

The adjusted initial packet drop/mark probability is computed by scaling the initial packet drop/mark probability by a fraction which reflects the relative length of a packet with respect to the maximum packet length. The initial packet drop/mark probability is defined as:

$$(P_{ini} = (CurrentPacketLength / MaxPacketLength) \cdot P_{ini}) \quad (2.3)$$

This is to ensure that the packet drop/mark probability is proportional to the packet length in bytes. RED incorporates the number of undropped packets since last dropping, count, to compute the final packet drop probability that can be expressed as:

$$(P_f = P_{ini} / (2 - count.P_{ini})) \quad (2.4)$$

As stated in (Floyd, November 1997), the application to final drop probability to incoming packets ensures a uniformly distributed packet intermarking interval with packet drops at evenly spaced intervals. This helps to avoid clustered packet drops which cause global synchronization. It also helps to reduce the occurrence of long periods during which no packets are dropped so as to enable effective control of the queue length and prevention of congestion.

Having several congestion detection thresholds together with dynamically adjusted packet drop probabilities allow RED routers to set the frequency of congestion notification delivery to the intensity of the congestion detected (Firoiu & Borden, 2000).

The RED algorithm involves four parameters to regulate its performance. Minth and Maxth are the queue thresholds to perform packet drop, Maxdrop is the packet drop probability at Maxth, and w is the weight parameter to calculate the average queue length from the instantaneous

queue length. The average buffer length follows the instantaneous buffer length. However, because w is much less than one, avg changes much slower than q. Thus, avg follows the long-term changes of q, reflecting persistent congestion in networks. By making the packet drop probability a function of the level of congestion, RED router has a low racket-drop probability during low congestion, while the drop probability increases the congestion level increases (Hassan & Jain, 2004).

The packet drop probability of RED is small in the interval Minth and Maxth. Furthermore, the packets to be dropped are chosen randomly from the arriving packets from different flows. Consequently, packets coming from different sources are not dropped simultaneously. Hence, RED gateways avoid global synchronization by randomly dropping packets. The performance of RED significantly depends on the values of its four parameters (May, Diot, Lyles, & Bolot, 2000) (Wu-Chang & Dilip, 1999), Maxdrop, Minth, Maxth, and w. It is very hard to find optimal values for these parameters as they would to depend on the typical round-trip times in the system (Welzl, 2005) (Floyd, http://www.icir.org/floyd/red.html#parameters, November 2008).

2.6 RED Parameters

In this subsection, we examine the effect of the RED parameters and how should be set:

- **The Weight Parameter, w.**

This parameter determines the reactiveness of the Exponential Weighted Moving Average (EWMA) process to traffic oscillations (Welzl, 2005). RED uses the average (and not the instantaneous) queue length as a control variable to control active packet drop. Computing the

average queue length involves the previous average queue length and the instantaneous queue length modified by a weight parameter w. Hence, average queue length works as low pass filter (LPF) (Floyd & Jacobson, 1993). The average queue length is required to track persistent congestion that occurs over long time range while, at the same time, filtering out short time congestion (Mahbub & Raj, 2003). Now consider what would happen if w were 1: only the instantaneous queue would be used, and the impact of preceding values would be completely eliminated. Setting this parameter to 0, conversely, would mean that the average queue length would remain fixed at some old value and not react to queue oscillations at all (Welzl, 2005). This means that, if w is very small, the average queue length does not catch up with the long range congestion that may result in the failure of active queue management. If w is very large, the average queue length tracks the instantaneous queue length, which also degrades the performance of active queue management (Bing & Mohammed, 2008).

In a realistic model for determining w, where aggregate TCP traffic has been taken into account, the values (0.05, 0.07) (Bing & Mohammed, 2008) give better performance than the values (0.001, 0.002) (Floyd, 1997) (Floyd & Jacobson, 1993) in certain cases.

- **Buffer Thresholds, Minth and Maxth.**

The desired (required) average queue length determines the values of these two parameters. Dropping incoming packets when the average queue length surpasses Maxth prevents the queue from growing further. If this parameter is set to a small value the queue will be small (and therefore short delay). Conversely, the parameter Minth depends on the burstiness of data traffic - if the bursty data traffic should be accommodated in the buffer fairly, Minth should be set to a rather large value - and at the same time, (Maxth - Minth) must not be very small to allow for

the randomness to take effect (Welzl, 2005). For a RED carrying only TCP traffic, Minth should be around five packets, and Maxth should be at least three times Minth (Floyd, 1997). A different set of values are required for Minth and Maxth to protect TCP traffic from non-TCP traffic which does not employ the congestion control mechanisms of TCP (Parris, Jeffay, & Smith, January 1999).

- **Maximum Packet Drop Probability, Max_{drop}.**

The selection of this parameter significantly affects the performance of RED. If Maxdrop is very small, then active packet drops are not enough to prevent global synchronization. Very large value of Maxdrop decreases the throughput. Even though a Maxdrop value of 0.1 is generally suggested (Floyd, 1997), the selection of an optimal value of Maxdrop according to network traffic situation is still an open issue (May et al., 2000) (Wu-Chang & Dilip, 1999).

It was demonstrated that the value of Maxdrop depends on the number of flows as well as the bandwidth delay product (Feng, Kandlur, Saha, & Shin, 1999). The upper bound of packet drop probability (Maxdrop) can be expressed as:

$$Max_{drop} \leq ((N * SS * C) \ / \ B\tau) \tag{2.5}$$

where N is the number of flows, B is the total bandwidth, SS is the segment size, τ is the round-trip time, and C is a constant. From Eq. 2.5, it is not possible to fix a value of Maxdrop for a dynamically changing the network environment (Mahbub & Raj, 2003). Finally, Maxdrop should be small because the general goal of RED is not to drop a large number of packets once Minth is exceeded but only drop a packet occasionally, as a result forcing senders to reduce their transmission rates (Welzl, 2005).

2.7 Improving the Response Time

RED uses four parameters and one state variable to regulate its performance. Using the average queue length in controlling the active packet drop provides the following advantages (Mahbub & Raj, 2003):

- Accumulating short-term congestion.
- Tracing long-term congestion.

The low pass filter characteristic of average queue is also featured with slow-time response to the changes of long-term congestion in networks. This is harmful to the throughput and delay performance of RED gateway. For example, after a long-term congestion, the average buffer length stays high even if the instantaneous queue is back to normal or low; RED will, therefore, continue dropping packets even after the end of congestion (May et al., 2000) resulting in low throughput. The slow response of the average queue length will result in the throughput restoring slowly after heavy congestion (Christiansen, Jeffay, Ott, & Smith, 2001). A larger value of w can improve the response time, but at the expense of the RED queue tracing short-term congestion, which is against the proactive queue management mechanisms principle.

Low Pass Filter/Over Drop Avoidance (LPF/ODA) is an efficient algorithm for calculating the average queue length. It has shown that LPF/ODA algorithm improves the response time, throughput, and reduces the delay of RED routers (Zheng & Atiquzzaman, 2005). LPF/ODA calculates the average queue length as follows:

During long-term congestion, and the average queue length is calculated by

$$avg = avg + w_q(q - avg) \tag{2.6}$$

- And, during this period, the RED queue is in the active drop phase.
- If the average queue length is high at the end of long-term congestion, halve the average queue length. During this period, the RED queue is in the over drop avoidance (ODA) phase.
- If the average queue length is below a specific threshold value after the end of long-term congestion, renew the value of the average queue length using the LPF model.

Figure 2.4 shows the flowchart of LPF/ODA algorithm.

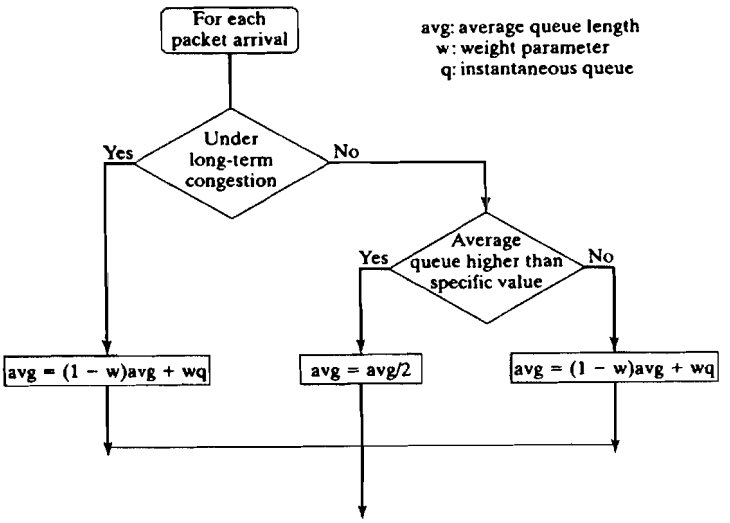

Figure 2.4 LPF/ODA Algorithm

2.8 RED Algorithm Description

This subsection describes the details of the RED algorithm, shown in Figure 2.6. The line numbers, enclosed in parenthesis, appearing throughout this section, refer to line numbers in Figure 2.6.

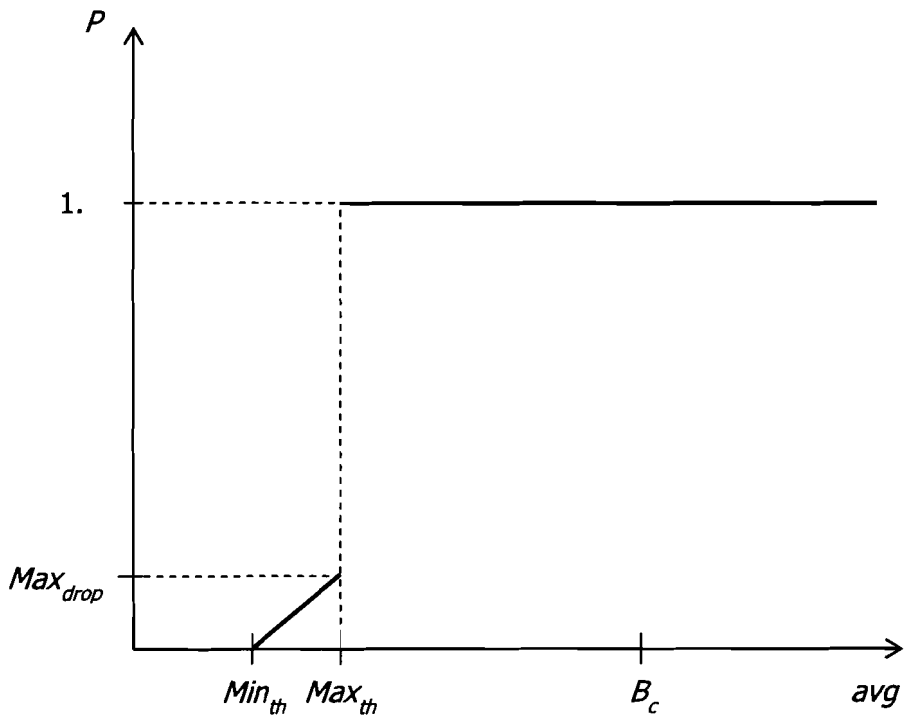

Figure 2.5 RED Packet Drop/Mark Function

The pseudo code given in Figure 2.6 describes the RED algorithm.

for (each arriving packet (Pkt_i)) {
/* calculate the average queue size (avg) */
[1] $avg^{(i)} = (1 - w) * avg^{(i-1)} + w * q_i$
/* normal operation */
if ($avg^{(i)} < Min_{th}$) {
/* admit packet to buffer */
[2] enqueue(Pkt_i) }
/* congestion avoidance */
else if ($Min_{th} \le avg^{(i)} < Max_{th}$) {
/* compute initial packet dropping/marking probability (P_{ini}) */
[3] $P_{ini}^{(i)} = \left(\frac{Max_{drop}}{Max_{th} - Min_{th}}\right) * avg^{(i)} - \left(\frac{Min_{th} * Max_{drop}}{Max_{th} - Min_{th}}\right)$
/* normalize initial packet dropping/marking probability */
[4] $P_{ini}^{(i)} = \left(\frac{getPktLength(Pkt_i)}{MaxPktLength}\right) P_{ini}^{(i)}$
/* compute final dropping/marking probability (P_f) */
[5] $P_f^{(i)} = \frac{P_{ini}^{(i)}}{2 - count * P_{ini}^{(i)}}$
/* drop/mark the packet with probability (P_f) */
[6] dropPacket(Pkt_i, $P_{ini}^{(i)}$)
[7] markPacket($Pktcn_i$, $P_{ini}^{(i)}$) }
/* congestion control */
else /*($avg^{(i)} \ge Max_{th}$)*/ {
/* drop/mark the packet with probability 1 */
[8] dropPacket(Pkt_i, 1.0)
[9] markPacket($Pktcn_i$, 1.0) } }

Figure 2.6 RED Algorithm

As mentioned earlier, Basic RED uses two thresholds, Maxth and Minth, on the average queue length for drop/mark activation and congestion detection. The Basic RED's initial packet dropping/marking probability is a piece-wise continuous linear increasing function of the average queue length and varies linearly from (0) to (Maxdrop) as the average queue length varies from (Minth) to (Maxth) as shown in Figure 2.5. Basic RED computes the average queue length (avg) upon every packet's arrival using an Exponential Weighted Moving Average

(EWMA) low pass filter (line 1). The average queue length is compared to the minimum threshold (Minth) and the maximum threshold (Maxth). If the average queue length is less than the minimum threshold, the packet is admitted to the buffer (line 2). If the average queue length is between the minimum and the maximum thresholds, the initial (line 3), the scaled initial (line 4), and finally the final (line 5) packet dropping/marking probability is computed and the packet is dropped/marked with the final dropping/marking probability (Pf) (line 6 if ECN-capable) and (line 7 if non-ECN-capable). If the average queue length is larger than the maximum threshold, the packet is dropped/marked with probability 1.0 (line 8 if ECN-capable) and (line 9 if non-ECN-capable).

Numerous studies were carried out widely to investigate the performance of TCP/IP over RED. The studies revealed that even though RED can improve the TCP performance under certain parameter settings and network conditions, the basic RED algorithm is still susceptible to several problems, such as bandwidth unfairness, low throughput under poorly setting parameters, and large queuing delay variance (jitter) because of the fluctuation of the queue level, being unable to handle unresponsive connections, and a high number of consecutive drop. In addition, according to Nagendran et al. (agendran et al., 2010), RED has fairness issue when UDP flows demand assured service. Therefore, it is important to stabilize the queue length to protect the responsive connections.

It can be concluded that even the use of multiple RED (MRED) with different parameter settings cannot solve the problems mentioned above due to the wide number of parameters that have an impact on the system's performance in DiffServ network. It is realized that there is a

dire need for more research in this area of networking to improve the system's performance and resource utilization in DiffServ environment.

2.9 Summary

This chapter began with a description of quality of service and differentiated services (DiffServ) environment. The chapter showed how important is the role of queue management mechanism in DiffServ. It described the function and structure details of RED queue management algorithms.

In this chapter, it has been revealed that MRED algorithm used in DiffServ has many problems such as bandwidth unfairness, low throughput under poorly setting parameters, and large queuing delay variance (jitter) because of the fluctuation of the queue level, being unable to handle unresponsive connections, and a high number of consecutive drop. These problems have motivated the researchers to improve MRED in order to improve DiffServ. We have observed and concluded that the current congestion control mechanisms cannot solve above mentioned problems due to the wide number of parameters that have an impact on the system's performance. We also realized that there is a dire need to improve the queuing management in DiffServ to provide good QoS to end users.

In the next chapter, the experimental tool and the research methods for performance evaluation of the enhanced version of MRED will be presented.

CHAPTER THREE
RESEARCH METHODOLOGY

Chapter 2 reviewed the background material on queue management that forms the basis of the general framework for this research, and revealed the necessity for the necessity for improving the queue management mechanism used in DiffServ networks, namely Multiple RED (MRED), in order to improve the performance of these networks. . As stated in Chapter 1, one of the objectives of this project aim at evaluating the enhanced MRED queue management mechanism. In this chapter, research methodology for developing and evaluating the performance of the enhanced MRED is presented.

3.1 Introduction

There are several ways to conduct research, and this depends on the purpose of study. It is common to use either a descriptive or a prescriptive approach in research on information technology (Nyame-Asiamah & Patel, 2009). Descriptive research seeks knowledge about the nature of reality, and improves performance of the system (Aken, 2004). Our research adapts *Research Design* because it is accepted among many researchers in the information and communication processing systems (Venable, 2006). Research Design can address the problem in a unique and efficient way (Khosrow-Pour, 2006). Many researchers have used Research Design approach depending on Vaishnavi & Kuechler (Vaishnavi & Kuechler, 2005). Vaishnavi and Kuechler methodology that is used in our research is shown in Figure 3.1.

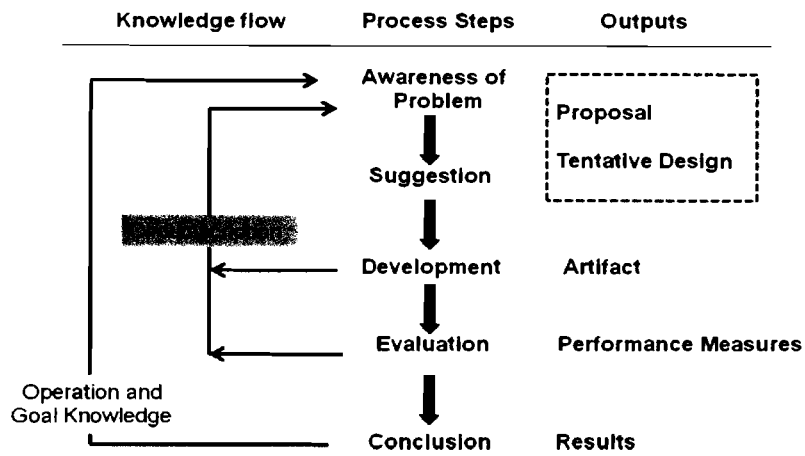

Figure 3. 1 General Methodology for Design Science Research (Vaishnavi & Kuechler, 2008)

The methodology illustrated in Figure 3.1 consists of five phases namely: Awareness of a problem, Suggestion, Development, Evaluation and Conclusion. Each of these phases is elaborated afterwards.

3.2 Awareness of Problem Step

The research work presented in this project is motivated by the need for a developed queue management mechanism to improve the performance of the Diffserv in the Internet as the current mechanism suffers from low throughput, and high number of consecutive drop in addition to large queuing delay variance (jitter) that make it being unable to handle unresponsive connections. This degrades the performance level or quality of service required by the user at the end system. Therefore, it is important to come up with an enhanced queue management mechanism to increase the throughput and reduce packet drops and delay in order to improve Diffserv networks.

3.3 Suggestion Step

Developing a drop probability function for MRED, which uses a measure of packet arrival rate with a measure of the queue length for its control decisions will provide good quality of service and show better ability in realizing the goals of controlling the packet arrival rate to the DiffServ-capable router, router queue lengths, and network congestion, while achieving a higher performance. Therefore, the enhanced MRED drop probability function presented in this project use the average queue length and the average packet arrival rate for making its dropping decision to accomplish the goal of providing good quality of service.

3.4 Development Step

In this section, the system requirements, the design, and the implementation of the enhanced MRED are presented.

3.4.1 System Requirements

Queue management mechanism in the network routers has to deal with the packets arrived at the input interface and take decision on whether to admit arrived packets into the buffer. To do so, the queue management should have information about the status of the buffer which can be done by calculating the buffer occupancy level compared to the buffer size. This information is required once packets arrive. Therefore, a model is required to provide this information. Based on the decision, another model is required to admit or drop the packet. As the drop can be randomly from the buffer, a model for calculating the drop probability function applied to a specific packet is required as well.

The enhanced MRED has to do the following tasks:

- Controlling the aggregate the packet arrival rate to maintain a higher average packet arrival rate at the router's buffer, with smaller rate variations, which assists to avoid congestion and improves link utilization.
- Controlling the instantaneous queue length to decrease the queuing delay and avoid buffer overflows while maintaining high link utilization and low packet drop ratio.
- Providing early congestion detection, based on prediction, and timely congestion notification, via packet dropping or marking based on the value resulting from the packet drop/mark probability computation, to instruct the traffic senders to reduce their transmission rates to help control the queue length.

3.4.2 System Design

Enhancing the MRED queue management mechanism requires designing a new packet drop/mark probability function that helps the queue management mechanism to take a proper decision or action in whether to allow or drop the packet to control the queue length efficiently. The current MRED uses an Exponentially Weighted Moving Average (EWMA) of the queue length to decide when to drop packets. It requires the use of a small queue averaging weight to make it less sensitive to very short-term increases in the packet arrival. A packet may be not allowed to the buffer while there is plenty space available or it may be allowed to the buffer while there is little room left before the buffer overflows. Therefore, the use of the average queue length does not allow the exercise of tight control over the instantaneous queue length, but allows only the average queue length to be controlled. This could lead to higher packet loss

and excessive oscillations in the instantaneous queue length which would generate large delay variations and delay jitter as well as poor link utilization.

Considering the average packet arrival rate when computing the packet drop/mark probability function, can provide information that helps the queue management mechanism to act properly.

3.4.3 Implementation

In this project, the enhanced MRED is developed by using C++ and implemented in network Simulator 2.32 software on a machine running the CentOS 5.2 version of the Linux operating system. The code is debugged and verified several times by conducting many simulation experiments to ensure that the models are working properly. The enhanced MRED is validated using ns-2 validation program, Run-time Trace, and Incremental Implementation. For every simulation, the Run-time Trace is checked to ensure it runs as expected. The detailed information regarding the development of the enhanced MRED is provided in Chapter four.

3.5 Evaluation Step

Evaluation is performed to ensure that the enhanced MRED is working properly and efficiently. The results gained from simulations are analyzed statistically to evaluate the performance of the enhanced MRED. The evaluation details are presented in Chapter five.

3.5.1 Simulation Topology and Scenario

The aim of this experiment is to show that it is possible to achieve prioritization of important packets without any use of information provided by transport layer, and to test and evaluate the

modified version of MRED under different Committed Information Rate (CIR) levels varied at the source edge nodes.

In this experiment, two priority levels are defined, the higher "In packets" or "green packets" and the lower "Out packets" or "red packets". We use the time-sliding window (TSW2CM) policer. For each edge router, a CIR is defined. All packets will be marked as high priority if the rate of the connection is below CIR. If the rate exceeds CIR, packets will be marked probabilistically such that the rate of packets marked with high priority corresponds to the CIR. The transmission rate is computed as the rate averaged over the "TSW window"; in this experiment, the simulation its duration is 20msec.

We use the network topology shown in Figure 3.2 below.

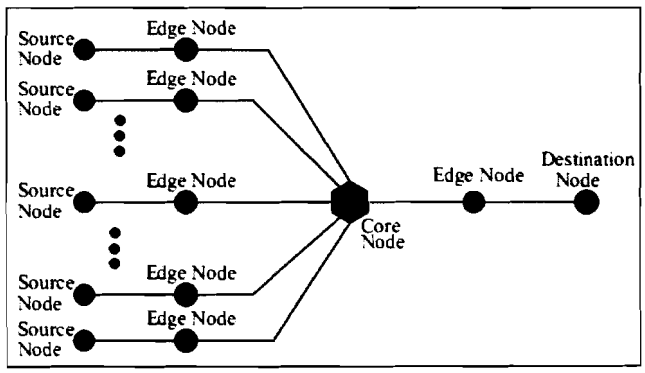

Figure 3. 2 Network Topology

50 sources are connected to their edge nodes via links of 100Mbps bandwidth and 0.5msec delay each. Links between the edge nodes and the bottleneck core node have 30Mbps bandwidth 2msec delay. The bandwidth of link between the core node and the edge router node

toward the destination node is 10Mbps with 5msec delay. The link between the edge node and the destination has 100Mbps bandwidth and 0.5msec delay.

Traffic coming from sources is marked according to parameters that are specified at their edge routers. The traffic represents a file that has a Pareto distribution with shape parameter 1.25 and an average size of 10kbytes (average transferred file over the internet). And the average packet size is 1040 bytes, of which 1000 are data and 40 bytes are header.

Files to be transmitted arrive at each source node according to a Poisson process with an average rate of five files per second. Many sessions from the same source node can be active at the same time.

The size of the queue at the bottleneck router is 100 packets. Therefore, the queue management parameters at other routers will not have any affects on the results. The modified Multi-RED version queue management is used at the bottleneck core node queue.

The same parameters for both priority levels are selected. The reason behind this selection is to create conditions that allow us to study the effect of the modified version of MRED on diminishing the loss probabilities of vulnerable packets, and on TCP performance in terms of delay and throughput. Giving the same parameters to both priorities, allows realizing the direct effect of protecting vulnerable packets on the TCP performance.

For each color of packets (red, green), the averaged queue length is monitored (this is done using the standard exponential averaging with parameter $w_q = 0.01$). Packets of a given color start to be dropped when the average number of queued packets of this color exceeds min_{th} of 15. This allows the drop probability to increase linearly with the averaged queue length until it

reaches max_{th} value of 45, where the drop probability, max_p, is chosen to be 0.5. The drop probability will be equal to 1 whenever max_p is exceeded.

The average packet size is 1040 bytes, of which 1000 are data and 40 bytes are header. An average ftp file is assumed to contain 10^4 bytes of data, which means that its total average size (including the headers) is about 1.04 x 10^4 x 8 bits. After multiplying by the number of source nodes and dividing by the average time between arrivals of files at a node, the packet arrival rate to the bottleneck core node is

$$avgR = \frac{50*1040*10^4*8}{0.22} = 18.909 Mbps \qquad (3.1)$$

Thus, it is obvious that the traffic rate is higher than the bottleneck link (which is 10 Mbps). Hence, the congestion is expected to occur, and therefore, it is required to have an active queue management mechanism.

The simulation duration is chosen to be 100 seconds to discard warming up period and avoid overlapping phenomenon.

When run the scenario, the behavior of MRED is recorded and examined by using performance metrics explained below.

3.5.2 Performance Metrics

The term metrics refers to the criteria used to evaluate the performance of the system as stated in (Jain, 1991). In this research, the following performance metrics are used for the quantitative performance comparison between the enhanced mechanism and standard UDP. This is to validate the new mechanism.

- **Outgoing Transmission Link Utilization**

Utilization of a link refers to an amount of data carried by a link relative to the link's maximum capacity. In this research, link utilization measures the ratio of time the link is utilized, it is defined as:

$$U = \frac{BusyTime\,[\text{seconds}]}{SimulationTime\,[\text{seconds}]} \cdot 100\% \tag{3.2}$$

- **Packet Drop**

Packet loss results in very noticeable performance issues. Network applications such as voice over IP, online gaming, and videoconferencing experience quality of service degradation when the packet loss is high as mentioned in (Imadud & Nazar Abbas, 2008). In this research, packet loss measures the ratio of the number of packets discarded at the bottleneck link to the total number of packets inserted into the bottleneck link buffer for all source, it is defined as:

$$L = \frac{DroppedPackets\,[\text{bits}]}{TotalPackets\,[\text{bits}]} \cdot 100\% \tag{3.3}$$

- **Queue Length**

Since the role of the proactive queue management mechanism is to keep the queue length as small as possible to provide space for accommodating temporary increases in the multimedia traffic to avoiding packet loss, queue length is an important metric through which the effectiveness of a queue management mechanism can be shown. In this research, statistical average (expected value (E(Q)) of the instantaneous queue length is used and defined as:

$$\bar{Q} = \frac{1}{N} \sum_{i=1}^{N} Q_i \tag{3.4}$$

- **Throughput**

Throughput is defined as the rate (requests per unit of time) at which the requests can be serviced by the system. In data network, throughput is defined as the amount of data transferred successfully from host to another in a given time period. Throughput, which is essentially bound by the BDP, is measured in number of bits per second (bps). In this research, throughput is measured as the number of data packets received correctly at the server host in a unit of time (in bit per second). Throughput for of a connection is calculated using the following formula:

$$thr = \frac{n}{t} R_i \qquad (3.5)$$

where n is the number of bytes received by the server at the end of the simulation, and t is the simulation time.

Analyze the results is typically done by post-analysis of the trace information produced by the ns–2 program execution. The trace files will usually have enough information to compute average link utilization on the communication links in the simulation, average queue sizes at the various queue, and drop rate in the queues, just to name a few.

The Diffserv network efficiency, with which the network resources are utilized, can be measured based on the link utilization and packet loss values. This information is important to network owners and operators. They can maximize their profits by minimizing their cost based on this information.

The average queue length identifies the average queuing delay experienced by the real time traffic passing through the DiffServ-capable router. The quality of service provided to the network users is indicated by the queuing delay and the packet loss. In other words, the queuing

delay and the packet loss determine the reliability and response time offered by the DiffServ network.

3.6 Conclusion Step

This step is the final step in this research. The results gained from the simulations and the analysis of the results showed that the enhanced MRED is work properly and the expected performance is achieved. More details and potential future work are presented in Chapter six.

3.7 Summary

This chapter presented the research method that is used to conduct this project. It presented the steps that are followed in enhancing MRED for DiffServ networks. The simulation scenario used in this research and based on a particular network topology is presented in this chapter as well. Also, the performance metrics, which are utilized by well-known network researchers and previous research works, used for the quantitative performance evaluation of the enhanced MRED queue management mechanism are described in this chapter. With the intention of running a reliable simulation experiments, the network simulator (ns-2) and the newly implemented ns-2 module were validate and verified.

After specifying the methodology and the experimental tool and scenario that are used to implement the enhanced MRED queue management mechanism in this chapter, the design and the implementation issues of the enhanced MRED will be presented in the next chapter.

CHAPTER FOUR
DESIGN AND IMPLEMENTATION OF ENHANCED MRED

After setting up the research methodology for evaluating the performance of the enhanced MRED queue management mechanism in Chapter 3, this chapter introduces the design and the implementation of the enhanced MRED.

4.1 Enhanced MRED Implementation Design

The main objective of enhanced MRED is to ensure that packets get their right treatment while providing the routers with congestion control capabilities in order to improve DiffServ network. Enhanced MRED achieves this by controlling packet arrival rate and the average queue length at the router. Rate and queue control are exercised through early congestion detection & notification, rate reduction, and queue growth (increase in the queue level) avoidance by dropping lower priority packets randomly.

The router should differentiate between different congestion levels and to practice congestion control of different levels, accordingly. The enhanced MRED mechanism controls the aggregate the packet arrival rate to maintain a higher average packet arrival rate at the router's buffer, with smaller rate variations, which assists to avoid congestion and improves link utilization and impose fairness between connections based on their packets priorities. Also, it controls the average queue length to decrease the queuing delay and avoid buffer overflows while maintaining high link utilization and low packet drop ratio generally. In addition, it provides early congestion detection and timely congestion notification by packet dropping or marking based on the value resulting from the packet drop/mark probability computation, to instruct the

traffic senders to reduce their transmission rates to help control the queue length. All this would help improving DiffServ. Enhanced MRED queue management monitors the average queue length to detect congestion and to compute the packet drop probability (P) which is applied to the arriving packet. Enhanced MRED queue manager determines the rate at which packets are discarded from the network in order to reduce the average packet arrival rate (R) at the router near the outgoing link capacity (C). When a packet is arrived at the router, the average packet arrival rate (R) will be computed by enhanced MRED queue manager.

After the packet's arrival, immediately, enhanced MRED makes a prediction of the expected changes in the queue length over a subsequent period of time, of length (T), presuming that the traffic arrival characteristics will remain unchanged over this period. Based on this prediction, enhanced MRED computes the initial packet marking probability (Pini) as the fraction of traffic arrival that needs to be dropped over this period taken into account high priority packets.

4.1.1 Average Packet Arrival Rate Estimation

In practice, a packet sliding window is used to calculate the average packet arrival rate of the aggregate data traffic. The packet sliding window technique uses a sliding window of packets which moves forward upon every packet arrival. A packet sliding window of size Ws packets calculates the average packet arrival rate based on the packet length and the interarrival time for the last Ws packets. It computes the average packet arrival rate as a fraction of the total arrived packets over the total elapsed time. If the k^{th} packet's arrival instant is represented by t_k, its interarrival time by T_k (= t_k - t_k-1), and its length by l_k, the average packet arrival rate is computed as:

$$R_i = \frac{L_{W_S}^{(i)}}{T_{W_S}^{(i)}} = \frac{\sum_{k=i-W_S+1}^{i} l_k}{\sum_{k=i-W_S+1}^{i} T_k} \tag{4.1}$$

Which can be implemented as:

$$R_i = \frac{L_{W_S}^{(i-1)} + (l_i - l_{i-W_S})}{t_i - t_{i-W_S}} \tag{4.2}$$

A packet sliding window is easy to implement and using actual data points for computing the average packet arrival rate.

If the packet arrival and queue states are stable and do not alter very much, it is worthwhile to drop lower priority packets at fairly regular intervals. It is important not to have too many packets dropped close to each other or to have long periods of time where no packets are dropped or marked. Too many dropped/marked packets close together can cause global synchronization, and also too long packet intermarking times between dropped packets can cause large queue sizes and congestion as mentioned in (Zheng & Atiquzzaman, 2005).

According to (Floyd, November 1997), when the traffic and queue states are stable, the number of undropped/unmarked packets between two adjacent dropping would be exponentially distributed if the probability (P_{ini}) is directly applied to individual arriving packets independently.

Therefore, if the initial packet dropping/marking probability (P_{ini}) is used directly to drop packets, packets could get dropped or marked in large numbers close to each other or not get dropped for very long periods of time. This is avoided by modifying the initial marking probability based on the number of undropped packets since the last dropped/marked packet

(count) to obtain a packet intermarking interval that is uniformly distributed over {1. 2, ..., 2/Pini} as in (Floyd, November 1997). This distributes the dropped/marked packets consistently over the incoming packets.

The pseudo code given in Figure 4.1 describes the enhanced MRED algorithm.

for (each arriving packet (Pkt_i)) {

/* compute average packet arrival rate to the router buffer (R)*/

$$R = \frac{PktLength_{-W_s+1} + \ldots + PktLength_i}{PktIntervalTime_{-W_s+1} + \ldots + PktIntervalTime};$$

/* calculate the average queue size (avg) */

[1] $avg^{(i)} = (1 - w_s) * avg^{(i-1)} + w_s * q_i$

/* normal operation */

if ($avg^{(i)} < Min_{th}$) {

/* admit packet to buffer */

[2] enqueue(Pkt_i) }

/* congestion avoidance */

else if ($Min_{th} \le avg^{(i)} < Max_{th}$) {

/* compute initial packet dropping probability (P_{mi}) */

[3] $P_{ini}^{(i)} = \left(\frac{Max_{drop}}{Max_{th} - Min_{th}}\right) * avg^{(i)} * R^{(i)} - \left(\frac{Min_{th} * Max_{drop}}{Max_{th} - Min_{th}}\right)$

/* normalize initial packet dropping/marking probability*/

[4] $P_{ini}^{(i)} = \left(\frac{getPktLength(Pkt_i)}{MaxPktLength}\right) P_{ini}^{(i)}$

/* compute final dropping probability(P_f)*/

[5] $P_f^{(i)} = \frac{P_{ini}^{(i)}}{2 - count.P_{ini}^{(i)}}$

/* drop the packet with probability (P_f)*/

[6] dropPacket(Pkt_i, $P_{ini}^{(i)}$)

/* congestion control */

else /*($avg^{(i)} \ge Max_{th}$)*/ {

/* drop/mark the packet with probability 1 */

[7] dropPacket(Pkt_i, 1.0)

Figure 4. 1 Enhanced MRED Algorithm

4.2 Enhanced MRED Implementation

The Diffserv environment that we are implementing follows the "Assured forwarding" approach. A packet that belongs to a connection could receive three levels of priority within the connection. These levels are called "drop precedences". It can be used to provide a lower loss probability to sync packets in a TCP connection. According to Noureddine and Tobagi (Noureddine & Tobagi, 2002), the losses of sync packets result in very long time-outs.

Implementing enhanced MRED for DiffServ required five modules to be compiled in ns2 simulator. One for RED-based queuing, one for policing, one for the base DiffServ router functionality (dsRED), and one for each the edge and core routers. Each of these modules defines a single class.

The essential module for the DiffServ implementation is dsRED module. It is included in "dsred.h" and "dsred.cc files." The dsRED Queue class is the parent class for the edgeQueue and coreQueue classes as shown in Figure 4.2 below.

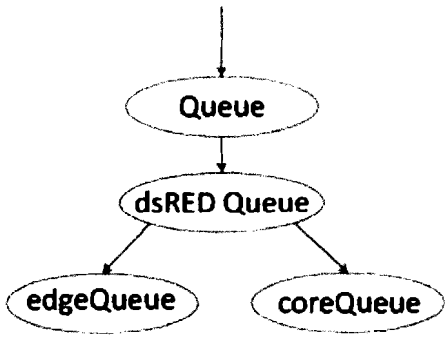

Figure 4. 2 The class hierarchy of dsREDQueue

In DiffServ architecture, dsRED Queue responsible for implementing functionality and declaring the parameters that are common to edge and core routers. The edge router module is defined by edgeQueue class. It is included in "edge.h" and "edge.cc." files. It is responsible for marking the code point for each packet. Also, it defines and managing policies which determine the kind of dealing that a packet receives at edge router. In addition, maintaining multiple physical and virtual queues and setting parameters for each these queues are done by this module. The maximum bandwidth a queue can use over a link to the core router can be determined by this module as well. coreQueue class that defines the core router module forwards packets based on the value of the code points marked on each packet. Core module is included in the "core.h" and "core.cc" files.

While differentiation of packets can be within each connection, all connections can be classified to many classes (at most four) which can be given different dealing as well.

Moreover, it is possible to differentiate between connections. Four classes of classes are defined (see Figure 4.3), and packets of a given class are queued in a class-dependent queue. To distinguish between packets that belong to the same class, in each of the four queues there are three internal virtual queues. To each of the 12 combinations of the four flow class and the three internal priority levels within a flow correspond a code point that a packet is given when entering the network.

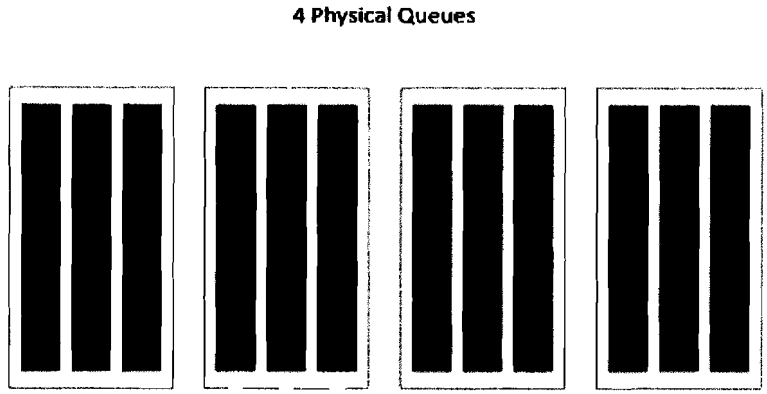

Figure 4. 3 dsREDQueue illustration

Rio Coupled RIO C is used in enhanced MRED, in which the probability of dropping low priority packets (called "out-of-profile packets") is based on the weighted average lengths of all virtual queues, whereas the probability of dropping a high priority ("in-profile") packet is based only on the weighted average length of its own virtual queue.

4.2.1 Configuration of Enhanced MRED in Network Simulator 2 (ns-2)

To simulate Diffserv using, the policy should be fully determined in the tcl script. To determine the number of physical queues, we use the following command:

```
$dsredq set numQueues_ $m
```

where **m** can take values between 1 and 4.

Configuring queue 0 to be a RIO-C is done with the following command:

```
$dsredq setMREDMode RIO-C 0
```

All queues are set to be RIO-C if the last argument is not given. Likewise, types other than RIO-C can be defined. To specify the number **$n** of virtual queues, we use the command:

```
$dsredq setNumPrec $n
```

RED has 6 parameters: the parameter queue weight, queue number, virtual queue number, min_{th}, max_{th} and max_P. If queue weight parameter, q_w, is not stated then it is taken to be 0.002 by default.

Red parameters can be configured using the following command:

```
$dsredq configQ $queueNum $virtualQueueNum $minTh $maxTh $maxP
```

The DropTail queue can also be used with the following command:

```
$dsredq setMREDMode DROP
```

The configuration then is given as before with only the first three parameters:

```
$dsredq configQ $queueNum $virtualQueueNum $minTh
```

All arriving packets are dropped when the min_{th} value is reached.

To compute the drop probability of RED, we need an estimate of the packet size. For a packet of size 1000 bytes, we can use the following command:

```
$dsredq meanPktSize 1000
```

Regarding the scheduling of packets, specific scheduling regimes can be defined. For example the weighted round robin with queue weights 5 and 1 respectively can be defined using the following command:

```
$dsredq setSchedularMode WRR

$dsredq addQueueWeights 1 5
```

Other possible scheduling are Weighted Interleaved Round Robin (WIRR), Round Robin (RR) which is the default scheduling, and the strict priorities (PRI).

The set of four queues along with the virtual queues is supplemented with a Per Hop Behavior (PHB) table. Its entries are defined by

- the code point,
- the class (physicalqueue), and
- the "precedence" (virtual queue).

An entry is assigned by using the following command

```
$dsredq addPHBEntry 11 0 1
```

which means that code point 11 is mapped to the virtual queue 1 of the physical queue 0.

The following command results in bringing the PHB table:

```
$dsredq printPHBTable
```

The number of physical and virtual queues can be brought using the following command:

```
$dsredq printStats
```

The following command can bring the RED weighted average size of the specified physical queues (0 in our case):

```
$dsredq getAverage 0
```

4.2.2 Defining Enhanced MRED Policies

All connections within the same source and going to same destination are subject to a regular policy. A policy defines many specific parameters such as policer type and target rate. It specifies at least two code points. The selection between them depends on the difference between the connection's current sending rate and its target, and possibly on the policy dependent parameters (such as burstiness). The policy specifies meter types that are used for measuring the relevant input traffic parameters. A packet arriving at the edge router causes the meter to update the state variables corresponding to the connection, and the packet is then marked according to the policy. The packet has an initial code point corresponding to the required service level; the marking can result in downgrading the service level with respect to the initial required one.

A policy table is used in ns-2 to store the policy type of each connection. Information stored in the policy table include Source Node ID, Destination Node ID, Policer Type, Meter Type, Initial Code Point, CIR (Committed Information Rate), CBS (Committed Burst Size), C Bucket

(Current Size Of The Committed Bucket), EBS (Excess Burst Size), E Bucket (Current Size of the Excess Bucket), PIR (Peak Information Rate), PBS (Peak Burst Size), P bucket (current size of the peak bucket), Arrival Time of Last Packet, Average Sending Rate, and TSW Window Length (TSW).

TSW is a policer based on average transmission rates and the averaging is performed over the window length, in seconds, of data. The default value is 1 sec. Possible policer types can be:

- **TSW2CM (TSW2CMPolicer)**

It uses a CIR and two drop precedences as well. The used probabilistically when the CIR is exceeded.

- **TSW3CM (TSW3CMPolicer)**

It uses a CIR, a PIR and three drop precedences. The medium priority level is used probabilistically when the CIR is exceeded, and the lowest one is used probabilistically when the PIR is exceeded.

- **Token Bucket (TokenBucketPolicer)**

It uses CIR and a CBS, and two drop precedences.

- **Single Rate Three Color Marker (srTCMPolicer)**

It uses CIR, CBS and EBS to choose from three drop precedences.

- **Two Rate Three Color Marker (trTCMPolicer)**

It uses CIR, CBS, EBS and PBS to choose from three drop precedences.

Each of the policer type mentioned above defines the meter it uses. The initial code point and one or two downgraded code points are defined by a policer table for each policy type. The initial code point is called "green code" and the lowest downgraded code is "red". If there is another code point in-between, it is called "yellow".

The configuration of the polices in ns-2 can be done through TCL. We can update the policy table by using the "addPolicyEntry" command which contains the edge queue variable denoting the edge queue, the source and destination nodes of the connection, the policer type, its initial code point, and then the values of the parameters that it uses; these are some or all of CIR, CBS, PIR and PBS as mentioned above. CIR and PIR are given in bps, and CBS, EBS and PBS in bytes.

For example:

```
$edgeQueue addPolicerEntry [$n1 id] [$n8 id] trTCM 10 200000 1000 300000 1000
```

By doing this, a policy for the connection that originates in **$n1** and ends at **$n8** is added. If the TSW policers are used, one can add at the end the TSW window length. If not added, it is taken to be 1sec by default.

We can use "addPolicyEntry" command specificcally to the policy and to the initial code point defines the downgraded code points which are same to all connections that use the policy with the common initial code point. It can be used as follows,

```
$edgeQueue addPolicerEntry srTCM 10 11 12
```

The command that used in bringing the entire policer table is:

```
$edgeQueue printPolicyTable
```

We can use the following command to bring the entire policer table:

```
$edgeQueue printPolicerTable
```

To get the current size of the C buckets in bytes, the following command can be used:

```
$edgeQueue getBucket
```

4.3 Summary

This chapter presented the design and the implementation of the enhanced MRED queue management mechanism. It presented the drop/mark probability function which is applied upon packet arrival to address the state of congestion and decide the probability with which the packet should be denied entrance to the queue based on priorities. The chapter covered how enhanced MRED uses a packet sliding window technique for computing the average packet arrival rate of the aggregate data traffic upon every packet arrival in order to take a proper decision and ensuring packet treatment. It show how enhanced MRED can be configured in NS2 for DiffServ networks.

CHAPTER FIVE
EVALUATION & RESULTS

This chapter is dedicated to discuss the evaluation of the enhanced MRED enhanced designed for DiffServ networks. The chapter provides the performance of the enhanced MRED regarding the protection of vulnerable packets. The goal of this chapter is to show that, with the enhanced MRED, we can achieve prioritization of sensitive packets without any use of transport layer information. The chapter also shows the enhanced MRED performance in terms of Committed Information Rate (CIR), average queue length, packet loss, the packet arrival rate, bandwidth utilization and throughput, based on the results obtained from the simulation.

5.1 Introduction

In TCP/IP networks, some packets are very important and the loss of them can affect the performance of TCP seriously. These packets include (i) packets responsible for TCP connection establishment, (ii) packets sent when the connection has a small window, and (iii) packets sent after a timeout or a fast retransmission. These packets are called "vulnerable" or "sensitive" packets. Marking those packets with a higher priority and implementing the priority using DiffServ architecture can help improve the performance of the TCP connection significantly. Marking those packets requires that network layer be aware of transport layer information such as the state of the TCP connection. The enhanced MRED presented in this project allows prioritizing the sensitive packets without the need for transport layer information. As mention in Chapter Three, two priority levels can be defined. The higher "In packets" and lower "Out packets" using Time-Sliding Window (TSW2CM).

5.2 The Effect of the Committed Information Rate (CIR) Variation

Committed Information Rate (CIR) is defined for each edge router. All packets are marked as high priority providing the TCP connection's rate is below CIR. Once the rate exceeds CIR, packets are marked probabilistically such that at the average, the rate of packets marked with high priority corresponds to the CIR. The transmitted rate is computed as the rate averaged over the "TSW window".

In evaluating the enhanced MRED, many experimentations were conducted with different CIR levels at the source edge nodes to study the effect CIR variation on performance. The CIR variation is 100Kbps, 300Kbps, 1Mbps, and 10Mbps.

We check the effect of the CIR marking rate on the loss probabilities of the SYN packets and of the first data in a connection, the effect is shown in Figures 5.1 to 5.4 below. In the figures,

CP means DiffServ code point; TotPkts means total packets; TxPkts means transmitted packets;. Ldrops means late drops; Edrops means early drops.

CP	TotPkts	TxPkts	ldrops	edrops
All	37800	36640	112	1048
10	4090	4090	0	0
11	33710	32550	112	1048

Figure 5. 1 CIR of 100Kbps

CP	TotPkts	TxPkts	ldrops	edrops
All	37513	36573	92	848
10	9421	9421	0	0
11	28092	27152	92	848

Figure 5. 2 CIR of 300Kbps

CP	TotPkts	TxPkts	ldrops	edrops
--	-------	------	------	------
All	37428	36738	0	690
10	20519	20200	0	319
11	16909	16538	0	371

Figure 5. 3 CIR of 1Mbps

CP	TotPkts	TxPkts	ldrops	edrops
--	-------	------	------	------
All	38061	36726	7	1328
10	38061	36726	7	1328

Figure 5. 4 CIR of 10Mbps

While the performance of enhanced MRED was the best when the CIR is 1Mbps, the original MRED for the same CIR is considered for comparison. The performance of the original MRED when CIR equals to 1Mbps is illustrated in Figure 5.5 below.

All	38022	36725	17	1280
10	1470	1470	0	0
11	36552	35255	17	1280

Figure 5. 5 CIR of 1Mbps for Original MRED

As mentioned in Chpater Three, the amount of packet loss results in very noticeable performance issues. It degrades the performance of the TCP applications significatly. From the figures we noticed that even with varied CIR levels, the losses of SYN packets is decreased as an implication of more packets transmitted and the better performance was achieved when CIR is 1Mbps.

5.3 Average Queue Length

This section provides the performance of enhanced MRED in terms of the average queue length compared to the original MRED. The average queue length can show how often the buffer is occupied which can help realizing how much it is controlled by the queue management employed at the router. The queue management mechanism used imposes its rules, such as packet drop or mark, to keep the queue length as small as possible to accommodate the sudden increases in the data traffic.

Figure 5.6 shows the average queue length of the bottleneck buffer when using enhanced MRED while Figure 5.7 shows the average queue length of original MRED, respectively.

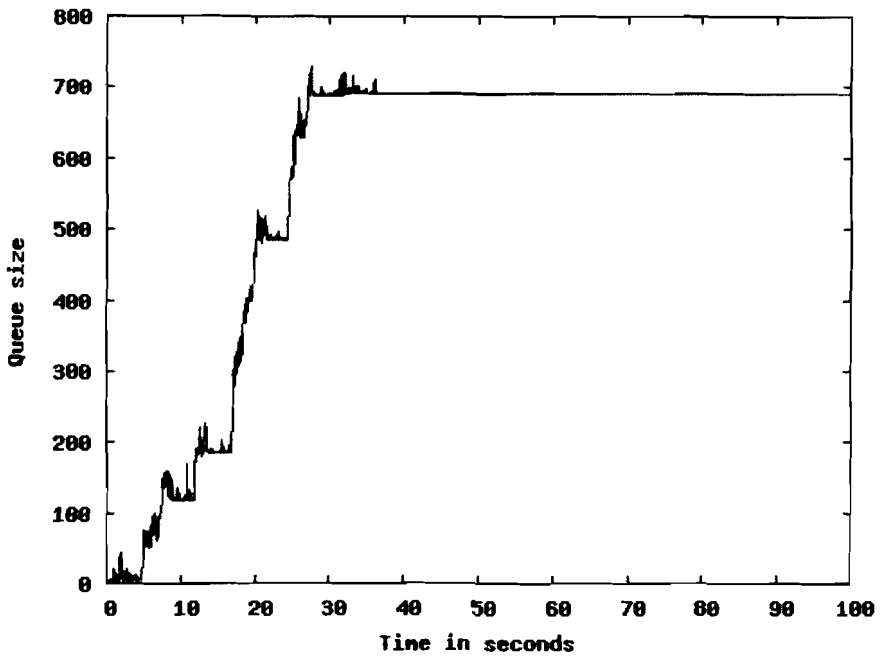

Figure 5. 6 The actual queue length of the enhanced MRED

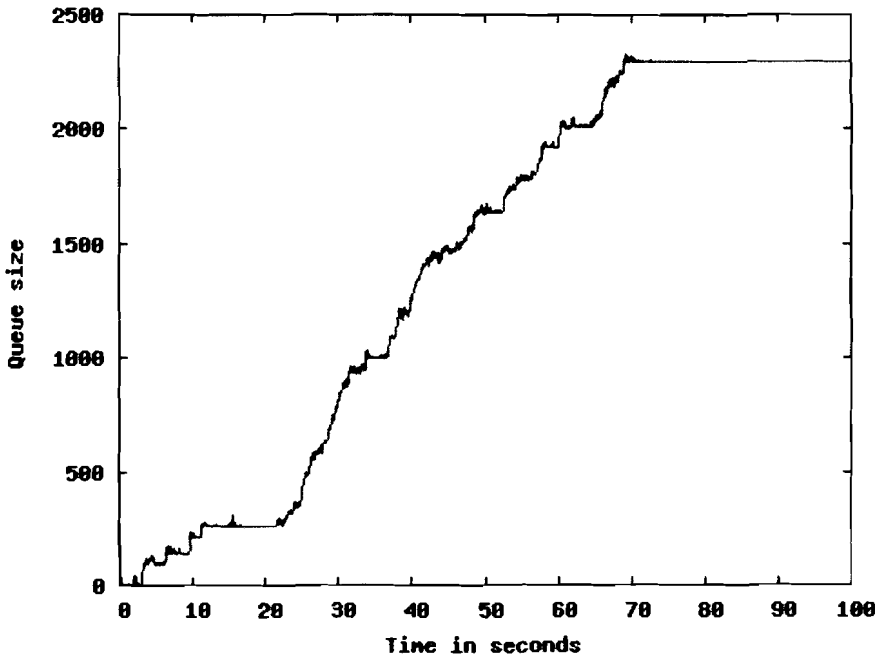

Figure 5. 7 The actual queue length of the original MRED

From the Figure 5.6, we can realize that enhanced MRED control the queue length efficiently compare to the case of the original MRED. As shown in Figure 5.6, the average queue size of the enhanced MRED is around 700 packets. Taking into account the packets size of 1000 and the bottleneck link bandwidth of 20Mbps, the average queuing delay equals:

$$D_q = \frac{700*1000*8}{10*10^3*10^3} = 560ms$$

For the case of original MRED, as shown in Figure 5.7, the average queue size is around 2300 packets. Therefore, the average queuing delay is:

$$D_q = \frac{2300*1000*8}{10*10^3*10^3} = 1840ms$$

Which is much higher than the case of enhanced MRED. It is clear that enhanced MRED control the queue length efficiently which leads to better network performance.

5.4 Outgoing Link Utilization

It is well known that the proportion of time the buffer having packets waiting to be transmitted on the outgoing transmission link determines the level to which the outgoing transmission link capacity is utilized. Link utilization is affected by the packet arrival pattern and speed of the packet arrival reaching the router buffer. Therefore, the link utilizations obtained on the outgoing transmission link can be understood in the view of the averages of the packet arrival rate and average queue size.

The link utilization obsereved throughout enhanced MRED and original MRED experiments are shown in Figure 5.8 and 5.9, respectively.

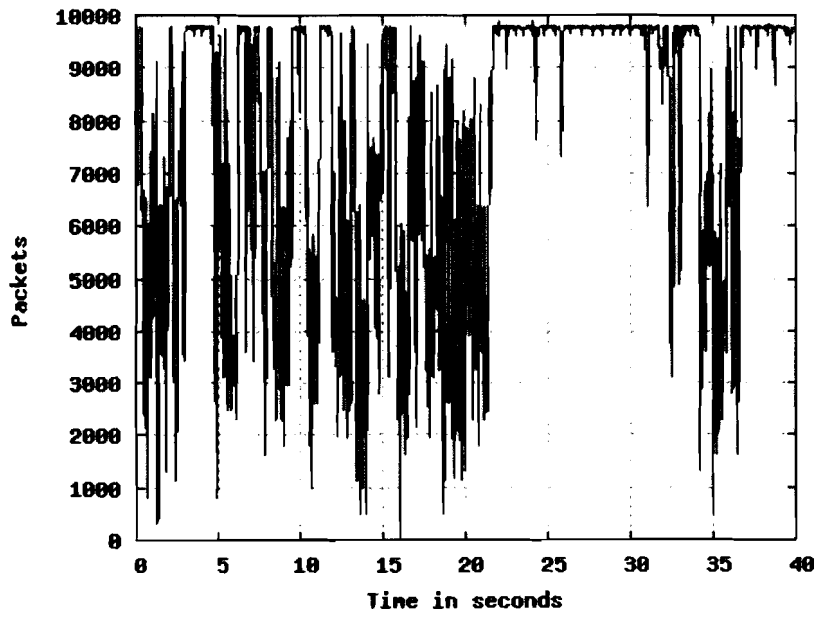

Figure 5. 8 Link utilization using enhanced MRED

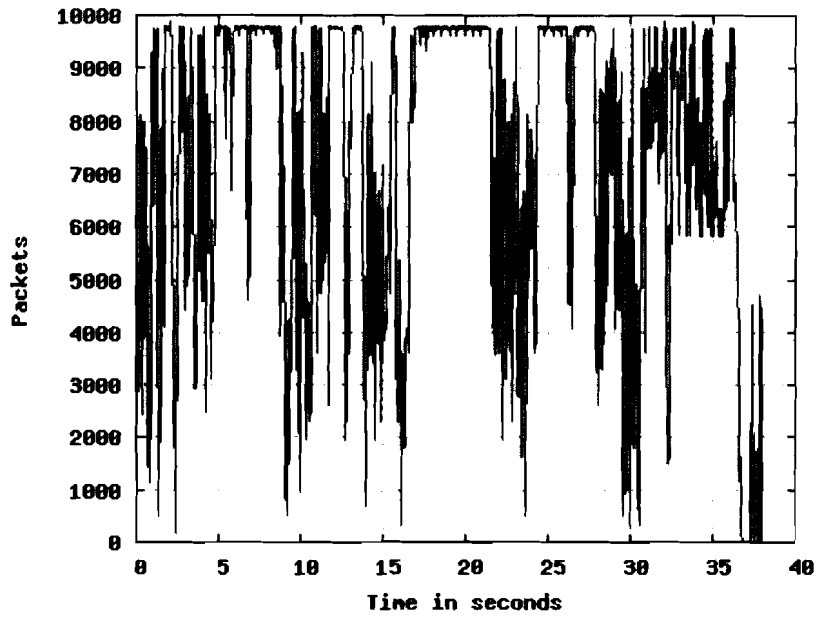

Figure 5. 9 Link utilization using original MRED

The figures show that enhanced MRED utilizes the bandwidth of bottleneck link somewhat better than original MRED even though there is a flactuation in the link utilization which can be due to the controlled arrival rate by the drop probability function of enhanced MRED.

5.5 Packet Arrival Rate

The aggregated packet arrival rate to the router buffer when using enhanced MRED is shown in Figure 5.10 while Figure 5.11 shows arrival rate for the case of original MRED.

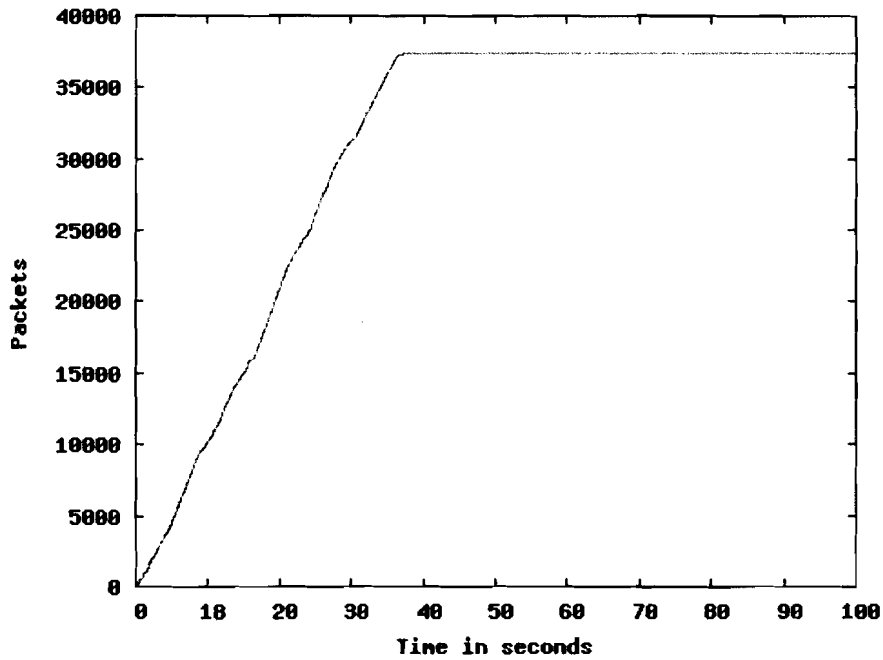

Figure 5. 10 Packet arrival using enhanced MRED

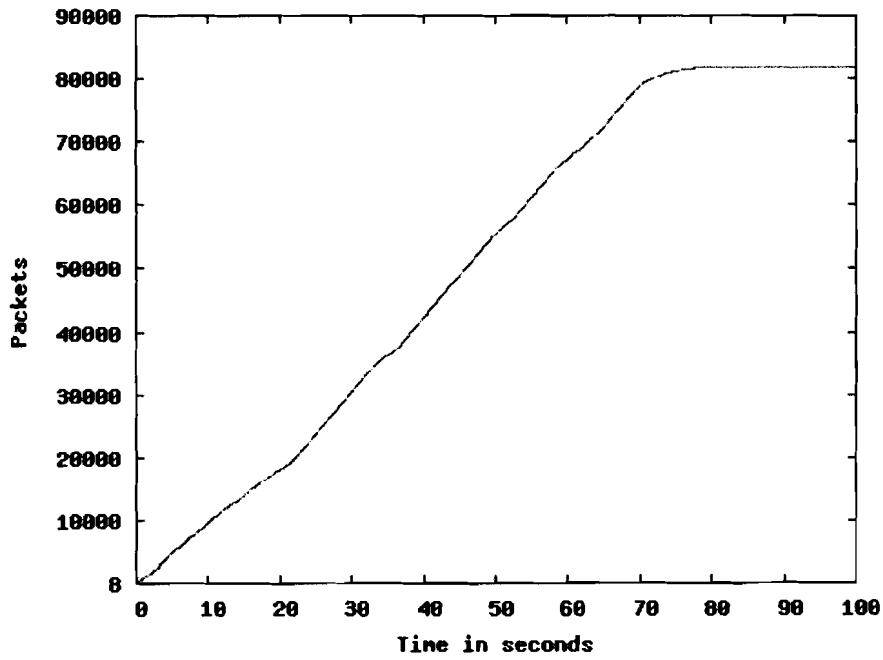

Figure 5. 11 Packet arrival using original MRED

From Figures 5.10, it is clear that enhanced MRED apply tight controll to direct the aggregated packet arrival rate to managemable level which help controlling the queue size and reducing the packet loss. Figure 5.11 indicates that the amount of packets arrived at the router was almost double than that of enhanced MRED, which justify the more packet drops that the sources experince over original MRED.

5.6 Throughput

In data network, throughput is defined as the amount of data transferred successfully from host to another in a given time period. Throughput, which is essentially bound by the Bandwidth Delay Product (BDP), is measured in number of bits per second (bps). In this project, throughput is measured as the number of data packets received correctly at the server host in a unit of time (in bit per second). Throughput is the significant performance measure for short and long-lived TCP connections.

Throughtout enhanced MRED expewriments, we have noticed that throughput is increased and packet loss is reduced. Tables 5.1 to 5.4 show that the number of data packets that where successfully transmitted during the simulations was quit independent on the CIR. This is due to the fact that arrival rate of sessions does not depend on the CIR. Figure 5.12 demonstrates the throughput gained when using enhanced MRED over CIR of 1Mbps. We can realize that with less packet losses and delay, the throughput is good as well, compared to that of original MRED.

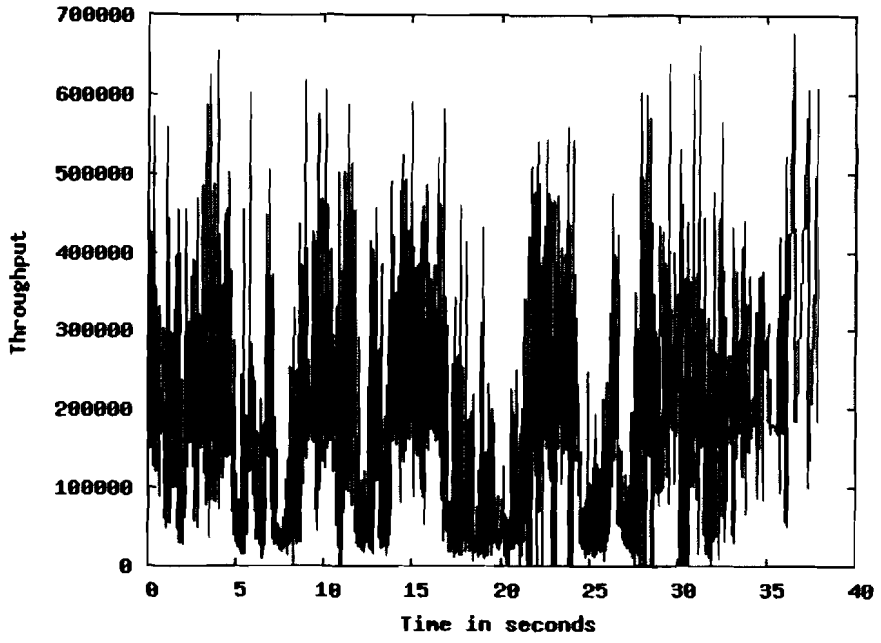

Figure 5. 12 Throughput using enhanced MRED

Figure 5.13 illustrates the throughputs gained when using the original RED.

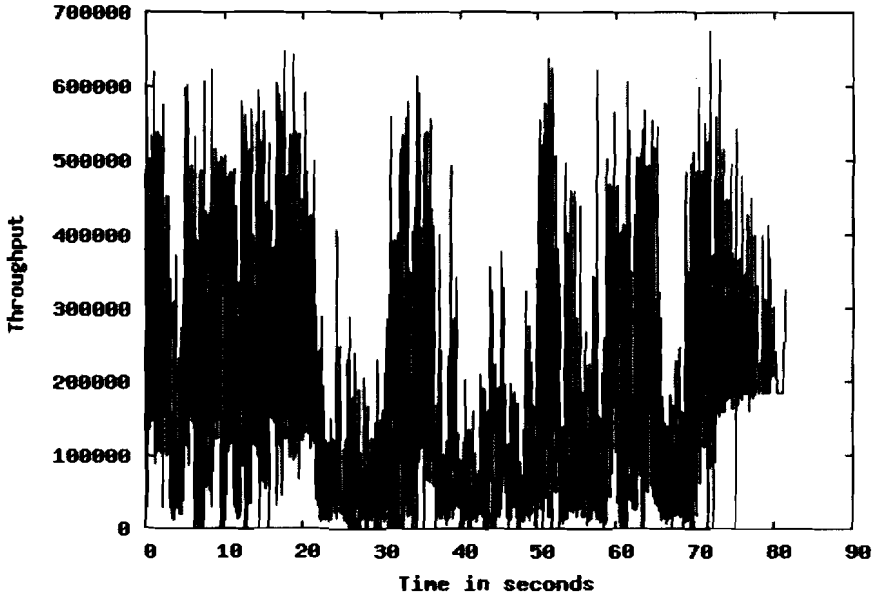

Figure 5. 13 Throughput using original MRED

From the figures, we can see that the average troughput in the use of enhanced MRED is better than original MRED.

We can conclude that a significant improvement was made by enhancing MRED mechnism to improve TCP/IP network performance using DiffServ architerchture. The enhanced MRED was validated through several simulations experiments. It was applied on specific case which is the protection of sensitive packets mentioned earlier in this chapter. Sensitive packets worsen performance of TCP considerably since they cause long time-outs. This is particularly the case for the loss of a SYN that results in a timeout of 3sec or of 6sec. In high speed networks the duration of a file transfer is short (often the whole transfer is much shorter than timeout), so we can expect to gain much by applying enhanced MRED to eliminate these long timeouts.

In simulation experiments, an average file size was chose to be 10kbytes, which is the avereaged measured file size in the Internet. Thus, approximately around 10% of the packets is a SYN packet and additionally, another 10% of the packets are first in a transfer. Therefore, in the absence of enhanced MRED mechnism, around 20% of lost packets would correspond to these types of sensitive packets. Thus, reducing these losses can result in a significant improvement in the TCP/IP performance.

5.7 Summary

This chapter presented the performance evaluation of the enhanced MRED and compared to the original MRED mechanism, used currently in DiffServ environment, based on the numerical results gained from simulation experiments. It studied the performance of the enhanced MRED in terms of the average queue length, the packet arrival rate, packet loss, the bottleneck link utilization, and the throughput gained.

It showed that the enhanced MRED helps moderating and controlling the packet arrival rate where the bottleneck link bandwidth is the limiting factor and the average arrival rate remains relatively near the maximum link capacity. It was shown how enhanced MRED can protect the sensitive packets, such as SYN, from being dropped or lost.

It was shown that the enhanced MRED offers less delay; and the packet loss is less compared to the case of using the original MRED, thus, it confirms the suitablity of the enhanced MRED for short TCP connections.

In addition, in terms of throughput, it was shown that the enhanced MRED allows TCP to improve its throughput with comparable packet loss. Forthermore, the enhanced MRED system helps to avoid congestion and improving the TCP network overall.

Finally, the enhanced MRED offers a superior performance to that of original MRED in terms of providing a lower queuing delay at a higher link utilization and a lower fraction of packet loss.

CHAPTER SIX
CONCLUSION AND FUTURE WORK

As the performance of enhanced MRED queue management mechanism was analyzed in Chapter Five based on the numerical results obtained from simulations to verify and validate the enhanced MRED developed for DiffServ networks, this chapter provides the conclusion of the research work presented in this project in Section 6.1 in addition to some suggestions for further studies in Section 6.2.

6.1 Conclusion

Recently, significant investments have been made in the planning and development of computer networks. The rapid growth of the applications over the Internet drives researchers to develop new mechanisms for internet infrastructure in order to guarantee the quality of service provided to user who use applications, such as web surfing, network monitoring, desktop sharing and video conferencing. The delay variations in network system affect in network applications. In an acknowledgement and time-out-based congestion control mechanism, e.g., TCP, performance is related to the delay-bandwidth product of the connections. In addition, TCP round-trip time (RTT) measurements are sensitive to delay variations, which may cause wrong timeouts and retransmissions.

In the Internet, all sources are supposed to have same treatment. While network resources are limited, providing guarantees on performance measures requires rejecting new connections when network resources are not available. To assign resource to connections according to their class, we have to differentiate between connection classes. For that, the Diffserv has been

proposed. Since Internet carries many different types of services, including voice, video, streaming data, web pages and email, many of the proposed QoS mechanisms that allowed these services to co-exist were both complex and failed to scale to meet the demands of the Internet.

DiffServ is a computer networking architecture that specifies a simple and scalable mechanism for classifying, managing network traffic and providing quality of service (QoS) guarantees on modern IP networks. For example, DiffServ can be used to provide low-latency, guaranteed service to critical network traffic such as voice or video while providing simple best-effort traffic guarantees to non-critical services such as web traffic or file transfers.

As Diffserv is based on marking data packets at the edge router of the network according to the performance level (quality of service) that the network wishes to provide, packets are handled differently at the network routers. This requires efficient and reliable buffering and scheduling mechanism to meet the user or subscriber requirements.

By marking packets at the edge of the network according to the performance level that the network wishes to provide them, the network's nodes treat the packets differently. A general way to distinguish packets is by using RED buffers and use different parameters for different packets. Thus, applications over the internet could benefit of lesser delays and larger throughputs.

A packet belonging to a flow may get three possible priority levels within the flow. This can be used to provide a lower loss probability to SYN packets in a TCP connection, as in contrast with other packets, the losses of SYN packets result in very long time-outs. Additional to differentiation within each connection, all connections are grouped to some classes (not more than four), and different treatment can be given to different classes.

Furthermore, it is possible to differentiate between flows. Four classes of flows are defined, and packets of a given class are queued in a class-dependent queue. To differentiate between packets belonging to same class, three virtual queues are implemented in each of the four queues. To each of the 12 combinations of the four flow class and the three internal priority levels within a flow correspond a code point that a packet is given when entering the network.

DiffServ-capable router utilizes MRED (mulit-RED) in each physical queue that allows creating dependencies between their operations. MRED probability of dropping each packet is based on the size of its virtual queue. MRED drop probability function uses the average queue length, which is collected over long period, to make its control decisions. However, the use the average queue length makes MRED reacts to congestion slowly. This results in large queue length variation and untimely congestion detection and notification which would cause performance degradation due to high queuing delays and high packet loss. MRED suffers from low bandwidth utilization, low throughput under poorly setting parameters, and large queuing delay variance (jitter) because of the fluctuation of the queue level, being unable to handle unresponsive connections, and high number of consecutive drop. Thus, the quality of service observed by the end system is lowered significantly. Therefore, the goal of the research presented in this project was motivated by the need to improve the performance of differentiated service network by enhancing DiffServ-capable router scheduling mechanism. Thus, a new drop probability function for DiffServ-capable router that utilizes MRED was develops. The main objective of the enhanced MRED is to improve the throughput and decrease packet loss.

Developing a drop probability function for MRED, which uses a measure of packet arrival rate with a measure of the queue length for its control decisions will provide good quality of service and show better ability in realizing the goals of controlling the packet arrival rate to the

DiffServ-capable router, router queue lengths, and network congestion, while achieving a higher performance. Therefore, the enhanced MRED drop probability function presented in this project use the average queue length and the average packet arrival rate for making its dropping decision to accomplish the goal of providing good quality of service. The enhanced MRED was implemented in simulated differentiated service network by using Network Simulator 2 (ns-2).Enhanced MRED was studied to analyze its performance in terms of throughput, packet loss, queue length, and link utilization based on the results obtained from the simulations.

We have concluded that a significant improvement was made by enhancing MRED mechnism to improve TCP/IP network performance using DiffServ architerchture. The enhanced MRED was validated through several simulations experiments. It was applied on specific case which is the protection of sensitive packets mentioned earlier in this chapter. Sensitive packets worsen performance of TCP considerably since they cause long time-outs. This is particularly the case for the loss of a SYN that results in a timeout of 3sec or of 6sec. In high speed networks the duration of a file transfer is short (often the whole transfer is much shorter than timeout), so we can expect to gain much by applying enhanced MRED to eliminate these long timeouts.

In simulation experiments, an average file size was chose to be 10kbytes, which is the avereaged measured file size in the Internet. Thus, around 10% of the packets is a SYN packet and additionally, another 10% of the packets are first in a transfer. Consequently, in the absence of enhanced MRED mechnism, approximately 20% of lost packets would correspond to these types of sensitive packets. Therefore, reducing these losses can result in a significant improvement in the TCP/IP performance. Enhanced MRED mechanism developed in this project can help improving Diffserv performance to ensure the user satisfaction regarding network traffic.

6.2 Suggestions for Future Work

For further research, we are going to investigate the performance of enhanced MRED over a TCP/IP network that involve ECN-capable sources. To enhance the performance further, the drop position should be changed to allow faster notification through ECN packets. Therefore, we are going to need to modify the packet drop position used by the enhanced MRED. We are going to deploy bigger network size than the one used in this project, in terms of number of nodes, router, and bottleneck links with different bandwidth and delays. Also, we are going to include different flavors of TCP to study the performance of the enhanced MRED in heterogonous network environment.

REFFERENCES

Aken, J. E. (2004). Management Research Based on the Paradigm of the Design Sciences: The Quest for Field Tested and Grounded Technological Rules. *Journal of management studies, 41*(2), 219-246.

Bianchi, G., & Blefari-Melazzi, N. (2001, 2001). *Admission control over assured forwarding PHBs: a way to provide service accuracy in a DiffServ framework.* Paper presented at the Global Telecommunications Conference, 2001. GLOBECOM '01. IEEE.

Bing, Z., & Mohammed, A. (2008). A framework to determine bounds of maximum loss rate parameter of RED queue for next generation routers. *J. Netw. Comput. Appl., 31*(4), 429-445.

Christiansen, M., Jeffay, K., Ott, D., & Smith, F. D. (2001). Tuning RED for Web traffic. *Networking, IEEE/ACM Transactions on, 9*(3), 249-264.

Du, L., Qiu, Z.-Y., & Guo, Y.-L. (2009, 21-22 May 2009). *An Improved Queue Management Algorithm in DiffServ Networks.* Paper presented at the Information and Computing Science, 2009. ICIC '09. Second International Conference on.

Durresi, A., Sridharan, M., Jain, R., Liu, & Goyal. (July 2001). *Traffic management using multilevel explicit congestion notification.* Paper presented at the 5th World MultiConference on Systemics, Cybernetics and Informatics SCI'2001, ABR over the Internet.

El Hachimi, M., Abouaissa, A., Lorenz, P., & Sathya, R. (2003, 9-11 April 2003). *Control algorithm for QoS based multicast in Diffserv domain.* Paper presented at the Communication Technology Proceedings, 2003. ICCT 2003. International Conference on.

Feng, W. C., Kandlur, D. D., Saha, D., & Shin, K. G. (1999). *A self-configuring RED gateway.* Paper presented at the INFOCOM '99. Eighteenth Annual Joint Conference of the IEEE Computer and Communications Societies. Proceedings. IEEE.

Firoiu, V., & Borden, M. (2000). *A study of active queue management for congestion control.* Paper presented at the INFOCOM 2000. Nineteenth Annual Joint Conference of the IEEE Computer and Communications Societies. Proceedings. IEEE.

Floyd, S. (1997). Discussions of Setting Parameters, http://www.icir.org/floyd/REDparameters.txt.

Floyd, S. (http://www.icir.org/floyd/red.html#parameters, November 2008). Setting Parameters.

Floyd, S. (November 1997). RED: discussion of setting parameters, .

Floyd, S., & Jacobson, V. (1993). Random early detection gateways for congestion avoidance. *Networking, IEEE/ACM Transactions on, 1*(4), 397-413.

Hassan, M., & Jain, R. (2004). *High Performance TCP/IP Networking: Concepts, Issues, and Solutions*: Pearson Prentice Hall.

Jahon, K., Byunghun, S., Kwangsue, C., Hyukjoon, L., & Hyunkook, K. (2001, 2001). *MRED: a new approach to random early detection.* Paper presented at the Information Networking, 2001. Proceedings. 15th International Conference on.

Khosrow-Pour, M. (2006). *Emerging Trends and Challenges in Information Technology Management*: IGI Global.

Kimura, T., Kamei, S., & Okamoto, T. (2002, 2002). *Evaluation of DiffServ-aware constraint-based routing schemes for multiprotocol label switching networks.* Paper presented at the Networks, 2002. ICON 2002. 10th IEEE International Conference on.

Lain-Chyr, H., Hsu, S. J., Cheng-Yuan, K., & Chun-Shin, J. (2004, 23-24 March 2004). *A new scheduler for AF and EF in a DS node.* Paper presented at the Distributed Computing Systems Workshops, 2004. Proceedings. 24th International Conference on.

Mahbub, H., & Raj, J. (2003). *High Performance TCP/IP Networking: Concepts, Issues, and Solutions*: Prentice-Hall, Inc.

Makkar, R., Lambadaris, I., Salim, J. H., Seddigh, N., Nandy, B., & Babiarz, J. (2000, 2000). *Empirical study of buffer management scheme for Diffserv assured forwarding PHB.* Paper presented at the Computer Communications and Networks, 2000. Proceedings. Ninth International Conference on.

May, M., Diot, C., Lyles, B., & Bolot, J. (2000). *Influence of Active Queue Management Parameters on Aggregate Traffic Performance.*

Nagendran, A., Kartick, V., Sayee Ram, V., SenthilKumar, L., & Sudha, S. (2010, 13-14 Dec. 2010). *Study on the effect of CBR on packet marking in assured forwarding.* Paper presented at the Research and Development (SCOReD), 2010 IEEE Student Conference on.

Nga, J. H. C., Iu, H. H. C., Ling, S. H., & Lam, H. K. (2008). Comparative study of stability in different TCP/RED models. *Chaos, Solitons & Fractals, 37*(4), 977-987.

Nyame-Asiamah, F., & Patel, N. V. (2009). Research methods and methodologies for studying organisational learning.

Parris, M., Jeffay, K., & Smith, F. D. (January 1999). *Lightweight Active Router-Queue Management for Multimedia Networking.* Paper presented at the Multimedia Computing and Networking (MMCN), San Jose, CA.

Peng, Y., Hongchao, H., Binqiang, W., & Hui, L. (2009, 24-26 April 2009). *PMUF: A High-Performance Scheduling Algorithm for DiffServ Classes.* Paper presented at the Computational Sciences and Optimization, 2009. CSO 2009. International Joint Conference on.

Qadeer, M. A., Sharma, V., Agarwal, A., & Husain, S. S. (2009, 8-11 Aug. 2009). *Differentiated services with multiple random early detection algorithm using ns2 simulator.* Paper presented at the Computer Science and Information Technology, 2009. ICCSIT 2009. 2nd IEEE International Conference on.

Qian, G. (2008, 20-22 Dec. 2008). *An Integrated Approach for DiffServ Multicasting.* Paper presented at the Computer Science and Computational Technology, 2008. ISCSCT '08. International Symposium on.

Royce, W. W. (1970). *Managing the development of large software systems.*

Ryu, S., Rump, C., & Qiao, C. (2004). Advances in Active Queue Management (AQM) Based TCP Congestion Control. *Telecommunication Systems, 25*(3), 317-351.

Stankiewicz, R., & Jajszczyk, A. (2007, 26-30 Nov. 2007). *Analytical Models for Multi-RED Queues Serving as Droppers in DiffServ Networks.* Paper presented at the Global Telecommunications Conference, 2007. GLOBECOM '07. IEEE.

Sundaresan, A. (1999). Differentiate Service, http://qos.ittc.ukans.edu/DiffSpec/node5.html.

Vaishnavi, V., & Kuechler, B. (2005, 16/9/2009). Design Research in Information Systems. Retrieved 20/11/2010, 2010, from http://desrist.org/design-research-in-information-systems/

Venable, J. (2006). The role of theory and theorising in design science research. *Proceedings of DESRIST*, 24-35.

Welzl, M. (2005). *Network Congestion Control: Managing Internet Traffic*: John Wiley & Sons.

Wen-Ping, L., & Zhen-Hua, L. (11-13 Dec. 2010). *Fractional Exponent Coupling of RIO.* Paper presented at the Computational Science and Engineering (CSE), 2010 IEEE 13th International Conference on.

Wu-Chang, F., & Dilip, D. K. (1999). Adaptive packet marking for maintaining end-to-end throughput in a differentiated-services internet. *IEEE/ACM Trans. Netw., 7*(5), 685-697.

Xiaojie, G., Kamal, J., & Leonard, J. S. (2004). *Fair and efficient router congestion control.* Paper presented at the Proceedings of the fifteenth annual ACM-SIAM symposium on Discrete algorithms.

Xipeng, X., & Ni, L. M. (1999). Internet QoS: a big picture. *Network, IEEE, 13*(2), 8-18.

Yang, X., Chen, H., & Zhang, Z. (2011, 28-29 March 2011). *A Queue Management Algorithm for Differentiated Services.* Paper presented at the Intelligent Computation Technology and Automation (ICICTA), 2011 International Conference on.

Yang, X., Chen, H., & Zhao, H. (2008, 16-18 July 2008). *A Queue Management algorithm of relative discrimination for Differentiated Services.* Paper presented at the Control Conference, 2008. CCC 2008. 27th Chinese.

Zheng, B., & Atiquzzaman, M. (2005). Low pass filter/over drop avoidance (LPF/ODA): an algorithm to improve the response time of RED gateways. *International Journal of Communication Systems, 15* (10), 899-906.

APPENDIX A: Differentiated Services (Core.cc)

```
#include "dsCore.h"
/*-----------------------------------------------------------------------------
class coreClass
-----------------------------------------------------------------------------*/
static class coreClass : public TclClass {
public:
        coreClass() : TclClass("Queue/dsRED/core") {}
        TclObject* create(int, const char*const*) {
                return (new coreQueue);
        }
} class_core;

/*-----------------------------------------------------------------------------
coreQueue() Constructor.
-----------------------------------------------------------------------------*/
coreQueue::coreQueue() {
}
/*-----------------------------------------------------------------------------
int command(int argc, const char*const* argv)
    Commands from the ns file are interpreted through this interface.
-----------------------------------------------------------------------------*/
```

```
int coreQueue::command(int argc, const char*const* argv) {
        return(dsREDQueue::command(argc, argv));
}
```

APPENDIX B: Differentiated Services (Core.h)

```
#ifndef DS_CORE_H
#define DS_CORE_H
#include "dsred.h"

/*-----------------------------------------------------------------------
class coreQueue
    This class specifies the characteristics for the core router.
-----------------------------------------------------------------------*/
class coreQueue : public dsREDQueue {
public:
       coreQueue();
       int command(int argc, const char*const* argv);
protected:
};
#endif
```

APPENDIX C: Differentiated Services (red.cc)

```
#include <stdio.h>
#include "ip.h"
#include "dsred.h"
#include "delay.h"
#include "random.h"
#include "flags.h"
#include "tcp.h"
#include "dsredq.h"



/*-----------------------------------------------------------------------------
dsREDClass declaration.
    Links the new class in the TCL heirarchy.
-----------------------------------------------------------------------------*/
static class dsREDClass : public TclClass {
public:
       dsREDClass() : TclClass("Queue/dsRED") {}
       TclObject* create(int, const char*const*) {
              return (new dsREDQueue);
       }
} class_dsred;
```

```
/*-----------------------------------------------------------------------------
dsREDQueue() Constructor.
    Initializes the queue.  Note that the default value assigned to numQueues
in tcl/lib/ns-default.tcl must be no greater than MAX_QUEUES (the physical
queue array size).

-----------------------------------------------------------------------------*/
dsREDQueue::dsREDQueue() : de_drop_(NULL), link_(NULL)    {
  bind("numQueues_", &numQueues_);
  bind_bool("ecn_", &ecn_);
  int i;

  numPrec = MAX_PREC;
  schedMode = schedModeRR;

  for(i=0;i<MAX_QUEUES;i++){
    queueMaxRate[i] = 0;
    queueWeight[i]=1;
  }
    queuesDone = MAX_QUEUES;
  phbEntries = 0;          // Number of entries in PHB table

  reset();
}
```

```
// RED queues initilization
void dsREDQueue::reset() {
  int i;

  qToDq = 0;        // q to be dequed, initialized to 0

  for(i=0;i<MAX_QUEUES;i++){
    queueAvgRate[i] = 0.0;
    queueArrTime[i] = 0.0;
    slicecount[i]=0;
    pktcount[i]=0;
    wirrTemp[i]=0;
    wirrqDone[i]=0;
  }

  stats.drops = 0;
  stats.edrops = 0;
  stats.pkts = 0;

  for(i=0;i<MAX_CP;i++){
    stats.drops_CP[i]=0;
    stats.edrops_CP[i]=0;
    stats.pkts_CP[i]=0;
  }
```

```
    for (i = 0; i < MAX_QUEUES; i++)
    redq_[i].qlim = limit();

  // Compute the "packet time constant" if we know the link bandwidth.  The ptc is
the max number of (avg sized)
  // pkts per second which can be placed on the link.
  if (link_)
    for (int i = 0; i < MAX_QUEUES; i++)
      redq_[i].setPTC(link_->bandwidth());

  Queue::reset();
}

/*-----------------------------------------------------------------------
void edrop(Packet* pkt)
    This method is used so that flowmonitor can monitor early drops.
-----------------------------------------------------------------------*/
void dsREDQueue::edrop(Packet* p)
{

       if (de_drop_ != 0){
              de_drop_->recv(p);
       }
```

```
        else {
                drop(p);
        }
}

/*-----------------------------------------------------------------------
void applyTSWMeter(int q_id, int pkt_size)
Update the average rate for a physical Q (indicated by q_id).
Pre: policy's variables avgRate, arrivalTime, and winLen hold valid values;
  pkt_size specifies the bytes just dequeued (0 means no packet dequeued).
Post: Adjusts policy's TSW state variables avgRate and arrivalTime (also called
tFront) according to the bytes sent.

-----------------------------------------------------------------------*/
void dsREDQueue::applyTSWMeter(int q_id, int pkt_size) {
  double now, bytesInTSW, newBytes;
  double winLen = 1.0;

  bytesInTSW = queueAvgRate[q_id] * winLen;

  newBytes = bytesInTSW + pkt_size;

// caculate the avarage packet arrival rate to the queue
  now = Scheduler::instance().clock();
```

```
  queueAvgRate[q_id] = newBytes / (now - queueArrTime[q_id] + winLen);
  queueArrTime[q_id] = now;
}

/*---------------------------------------------------------------------------
void enque(Packet* pkt)
    The following method outlines the enquing mechanism for a Diffserv router.
This method is not used by the inheriting classes; it only serves as an
outline.
---------------------------------------------------------------------------*/
void dsREDQueue::enque(Packet* pkt) {
  int codePt, eq_id, prec;
  hdr_ip* iph = hdr_ip::access(pkt);
  //extracting the marking done by the edge router
  codePt = iph->prio();
  int ecn = 0;

  //looking up queue and prec numbers for that codept
  lookupPHBTable(codePt, &eq_id, &prec);

  // code added for ECN support
  //hdr_flags* hf = (hdr_flags*)(pkt->access(off_flags_));
  hdr_flags* hf = hdr_flags::access(pkt);
```

```
if (ecn_ && hf->ect()) ecn = 1;

stats.pkts_CP[codePt]++;
stats.pkts++;

switch(redq_[eq_id].enque(pkt, prec, ecn)) {
case PKT_ENQUEUED:
  break;
case PKT_DROPPED:
  stats.drops_CP[codePt]++;
  stats.drops++;
  drop(pkt);
       break;
case PKT_EDROPPED:
  stats.edrops_CP[codePt]++;
  stats.edrops++;
  edrop(pkt);
  break;
case PKT_MARKED:
  hf->ce() = 1;   // mark Congestion Experienced bit
  break;
default:
  break;
}
```

```
}

// Dequing mechanism for both edge and core router.
Packet* dsREDQueue::deque() {
  Packet *p = NULL;
  int queue, prec;
  hdr_ip* iph;
  int fid;
  int dq_id;

  // Select queue to deque under the scheduling scheme specified.
  dq_id = selectQueueToDeque();

  // Dequeue a packet from the underlying queue:
  if (dq_id < numQueues_)
    p = redq_[dq_id].deque();

  if (p) {
    iph= hdr_ip::access(p);
    fid = iph->flowid()/32;
    pktcount[dq_id]+=1;

    // update the average rate for pri-queue when there is a packet dequeued, update
the average rate of each queue ()
```

```
    if (schedMode==schedModePRI)
      for (int i=0;i<numQueues_;i++)
       if (queueMaxRate[i])
         applyTSWMeter(i, (i == dq_id) ? hdr_cmn::access(p)->size() : 0);

    // Get the precedence level (or virtual queue id) for the packet dequeued.
    lookupPHBTable(getCodePt(p), &queue, &prec);

    // decrement virtual queue length
    // Previously in updateREDStateVar,
    //redq_[dq_id].qParam_[prec].qlen--;
    redq_[dq_id].updateVREDLen(prec);
    // update state variables for that "virtual" queue
    redq_[dq_id].updateREDStateVar(prec);
  }

  // Return the dequed packet:
  return(p);
}

//    Extracts the code point marking from packet header.
int dsREDQueue::getCodePt(Packet *p) {
  hdr_ip* iph = hdr_ip::access(p);
  return(iph->prio());
```

```
}

// Reutrn the id of physical queue to be dequeued
int dsREDQueue::selectQueueToDeque() {
  // If the queue to be dequed has no elements,
  // look for the next queue in line
  int i = 0;

  // Round-Robin
  if(schedMode==schedModeRR){

    //printf("RR\n");
    qToDq = ((qToDq + 1) % numQueues_);
    while ((i < numQueues_) && (redq_[qToDq].getRealLength() == 0)) {
      qToDq = ((qToDq + 1) % numQueues_);
      i++;
    }
  }
else if (schedMode==schedModeWRR) { // Weighted Round Robin
    if(wirrTemp[qToDq]<=0){
      qToDq = ((qToDq + 1) % numQueues_);
      wirrTemp[qToDq] = queueWeight[qToDq] - 1;
    } else
        {
```

```
        wirrTemp[qToDq] = wirrTemp[qToDq] -1;
      }

      while ((i < numQueues_) && (redq_[qToDq].getRealLength() == 0)) {
        wirrTemp[qToDq] = 0;
        qToDq = ((qToDq + 1) % numQueues_);
        wirrTemp[qToDq] = queueWeight[qToDq] - 1;
        i++;
      }
    }
      else if (schedMode==schedModeWIRR) {
    qToDq = ((qToDq + 1) % numQueues_);
    while ((i<numQueues_) && ((redq_[qToDq].getRealLength()==0) ||
(wirrqDone[qToDq]))) {
      if (!wirrqDone[qToDq]) {
       queuesDone++;
       wirrqDone[qToDq]=1;
      }
      qToDq = ((qToDq + 1) % numQueues_);
      i++;
    }

    if (wirrTemp[qToDq] == 1) {
      queuesDone +=1;
```

```
        wirrqDone[qToDq]=1;
      }
      wirrTemp[qToDq]-=1;
      if(queuesDone >= numQueues_)
          {
        queuesDone = 0;
        for(i=0;i<numQueues_;i++)
            {
         wirrTemp[i] = queueWeight[i];
         wirrqDone[i]=0;
        }
      }
    } else
        if (schedMode==schedModePRI)
        {
     // Find the queue with highest priority, which satisfies:
      // 1. nozero queue length; and either
      // 2.1. has no MaxRate specified; or
      // 2.2. has MaxRate specified and
      //          its average rate is not beyond that limit.
      i = 0;
      while (i < numQueues_ &&
            (redq_[i].getRealLength() == 0 ||
             (queueMaxRate[i] && queueAvgRate[i]>queueMaxRate[i]))){
```

```
        i++;
      }
      qToDq = i;

      // If no queue satisfies the condition above,
      // find the Queue with highest priority,
      // which has packet to dequeue.
      // NOTE: the high priority queue can still have its packet dequeued
      //       even if its average rate has beyond the MAX rate specified!
      //       Ideally, a NO_PACKET_TO_DEQUEUE should be returned.
      if (i == numQueues_)
        {
        i = qToDq = 0;
        while ((i < numQueues_) && (redq_[qToDq].getRealLength() == 0)) {
         qToDq = ((qToDq + 1) % numQueues_);
         i++;
        }
      }
    }
    return(qToDq);
  }
```

```
/*-----------------------------------------------------------------------------
void lookupPHBTable(int codePt, int* queue, int* prec)
    Assigns the queue and prec parameters values corresponding to a given code
point.  The code point is assumed to be present in the PHB table.  If it is
not, an error message is outputted and queue and prec are undefined.
-----------------------------------------------------------------------------*/

void dsREDQueue::lookupPHBTable(int codePt, int* queue, int* prec) {
  for (int i = 0; i < phbEntries; i++) {
    if (phb_[i].codePt_ == codePt) {
      *queue = phb_[i].queue_;
      *prec = phb_[i].prec_;
      return;
    }
  }
  // quiet the compiler
  *queue = 0;
  *prec = 0;
  printf("ERROR: No match found for code point %d in PHB Table.\n", codePt);
  assert (false);
}
```

```
/*-----------------------------------------------------------------------------

void addPHBEntry(int codePt, int queue, int prec)

Add a PHB table entry. (Each entry maps a code point to a queue-precedence pair.)

-----------------------------------------------------------------------------*/

void dsREDQueue::addPHBEntry(int codePt, int queue, int prec) {

  if (phbEntries == MAX_CP) {

    printf("ERROR: PHB Table size limit exceeded.\n");

  }

    else

     {

    phb_[phbEntries].codePt_ = codePt;

    phb_[phbEntries].queue_ = queue;

    phb_[phbEntries].prec_ = prec;

    stats.valid_CP[codePt] = 1;

    phbEntries++;

  }

}


/*-----------------------------------------------------------------------------

void addPHBEntry(int codePt, int queue, int prec)

    Add a PHB table entry.   (Each entry maps a code point to a queue-precedence
pair.)

-----------------------------------------------------------------------------*/
```

```
double dsREDQueue::getStat(int argc, const char*const* argv) {
       if (argc == 3) {
            if (strcmp(argv[2], "drops") == 0)
         return (stats.drops*1.0);
            if (strcmp(argv[2], "edrops") == 0)
         return (stats.edrops*1.0);
            if (strcmp(argv[2], "pkts") == 0)
         return (stats.pkts*1.0);
           }
           if (argc == 4) {
            if (strcmp(argv[2], "drops") == 0)
               return (stats.drops_CP[atoi(argv[3])]*1.0);
            if (strcmp(argv[2], "edrops") == 0)
               return (stats.edrops_CP[atoi(argv[3])]*1.0);
            if (strcmp(argv[2], "pkts") == 0)
               return (stats.pkts_CP[atoi(argv[3])]*1.0);
       }
       return -1.0;
}
/*-----------------------------------------------------------------------------
void setNumPrec(int prec)
   Sets the current number of drop precendences.  The number of precedences is the
number of virtual queues per physical queue.
-----------------------------------------------------------------------------*/
```

```
void dsREDQueue::setNumPrec(int prec) {
        int i;
        if (prec > MAX_PREC)
        {
        printf("ERROR: Cannot declare more than %d prcedence levels (as defined by
                MAX_PREC)\n",MAX_PREC);
        } else
              {
               numPrec = prec;

               for (i = 0; i < MAX_QUEUES; i++)
                     redq_[i].numPrec = numPrec;
        }
}

/*-----------------------------------------------------------------------------
void setMREDMode(const char* mode)
   sets up the average queue accounting mode.
-----------------------------------------------------------------------------*/
void dsREDQueue::setMREDMode(const char* mode, const char* queue) {
  int i;
  mredModeType tempMode;

  if (strcmp(mode, "RIO-C") == 0)
```

```
    tempMode = rio_c;
  else if (strcmp(mode, "RIO-D") == 0)
    tempMode = rio_d;
  else if (strcmp(mode, "WRED") == 0)
    tempMode = wred;
  else if (strcmp(mode, "DROP") == 0)
    tempMode = dropTail;
  else {
    printf("Error: MRED mode %s does not exist\n",mode);
    return;
  }

  if (!queue)
    for (i = 0; i < MAX_QUEUES; i++)
      redq_[i].mredMode = tempMode;
  else
    redq_[atoi(queue)].mredMode = tempMode;
}
/*-----------------------------------------------------------------------
void printPHBTable()
    Prints the PHB Table, with one entry per line.
-----------------------------------------------------------------------*/
void dsREDQueue::printPHBTable() {
   printf("PHB Table:\n");
```

```
   for (int i = 0; i < phbEntries; i++)
      printf("Code Point %d is associated with Queue %d, Precedence %d\n",
phb_[i].codePt_, phb_[i].queue_, phb_[i].prec_);
   printf("\n");
}
/*----------------------------------------------------------------------------
void printStats()
    An output method that may be altered to assist debugging.
----------------------------------------------------------------------------*/
void dsREDQueue::printStats() {
      printf("\nPackets Statistics\n");
      printf("=====================================\n");
      printf(" CP  TotPkts   TxPkts   ldrops   edrops\n");
      printf(" --  -------   ------   ------   ------\n");
      printf("All   %8ld   %8ld   %8ld   %8ld\n", stats.pkts, stats.pkts-stats.drops-
stats.edrops, stats.drops, stats.edrops);
      for (int i = 0; i < MAX_CP; i++)
            if (stats.pkts_CP[i] != 0)
                  printf("%3d              %8ld              %8ld              %8ld
%8ld\n", i, stats.pkts_CP[i], stats.pkts_CP[i]-stats.drops_CP[i]-
stats.edrops_CP[i], stats.drops_CP[i], stats.edrops_CP[i]);
}
void dsREDQueue::printWRRcount() {
   int i;
   for (i = 0; i < numQueues_; i++){
```

```
        printf("%d: %d %d %d.\n", i, slicecount[i],pktcount[i],queueWeight[i]);
    }
}

/*----------------------------------------------------------------------------
void setSchedularMode(int schedtype)
   sets up the schedular mode.
----------------------------------------------------------------------------*/
void dsREDQueue::setSchedularMode(const char* schedtype) {
        if (strcmp(schedtype, "RR") == 0)
        schedMode = schedModeRR;
        else if (strcmp(schedtype, "WRR") == 0)
                schedMode = schedModeWRR;
        else if (strcmp(schedtype, "WIRR") == 0)
                schedMode = schedModeWIRR;
        else if (strcmp(schedtype, "PRI") == 0)
                schedMode = schedModePRI;
        else
                printf("Error: Scheduler type %s does not exist\n",schedtype);
}
/*----------------------------------------------------------------------------
void addQueueWeights(int queueNum, int weight)
   An input method to set the individual Queue Weights.
----------------------------------------------------------------------------*/
```

```
void dsREDQueue::addQueueWeights(int queueNum, int weight) {
  if(queueNum < MAX_QUEUES){
    queueWeight[queueNum]=weight;
  } else {
    printf("The queue number is out of range.\n");
  }
}
//Set the individual Queue Max Rates for Priority Queueing.
void dsREDQueue::addQueueRate(int queueNum, int rate) {
  if(queueNum < MAX_QUEUES){
    // Convert to BYTE/SECOND
    queueMaxRate[queueNum]=(double)rate/8.0;
  } else {
    printf("The queue number is out of range.\n");
  }
}
/*-----------------------------------------------------------------------
int command(int argc, const char*const* argv)
    Commands from the ns file are interpreted through this interface.
-----------------------------------------------------------------------*/
int dsREDQueue::command(int argc, const char*const* argv) {
  if (strcmp(argv[1], "configQ") == 0) {
    // modification to set the parameter q_w by Thilo
    redq_[atoi(argv[2])].config(atoi(argv[3]), argc, argv);
```

```
    return(TCL_OK);
  }
  if (strcmp(argv[1], "addPHBEntry") == 0) {
    addPHBEntry(atoi(argv[2]), atoi(argv[3]), atoi(argv[4]));
    return (TCL_OK);
  }
  if (strcmp(argv[1], "meanPktSize") == 0) {
    for (int i = 0; i < MAX_QUEUES; i++)
      redq_[i].setMPS(atoi(argv[2]));
    return(TCL_OK);
  }

  if (strcmp(argv[1], "setNumPrec") == 0) {
    setNumPrec(atoi(argv[2]));
    return(TCL_OK);
  }

  if (strcmp(argv[1], "getAverage") == 0) {
    Tcl& tcl = Tcl::instance();
    tcl.resultf("%f", redq_[atoi(argv[2])].getWeightedLength());
    return(TCL_OK);
  }

  if (strcmp(argv[1], "getStat") == 0) {
```

```
    Tcl& tcl = Tcl::instance();
    tcl.resultf("%f", getStat(argc,argv));
    return(TCL_OK);
  }

  if (strcmp(argv[1], "getCurrent") == 0) {
    Tcl& tcl = Tcl::instance();
    tcl.resultf("%f", redq_[atoi(argv[2])].getRealLength()*1.0);
    return(TCL_OK);
  }

  if (strcmp(argv[1], "printStats") == 0) {
    printStats();
    return (TCL_OK);
  }

  if (strcmp(argv[1], "printWRRcount") == 0) {
    printWRRcount();
    return (TCL_OK);
  }

  if (strcmp(argv[1], "printPHBTable") == 0) {
    printPHBTable();
    return (TCL_OK);
```

```
}
if (strcmp(argv[1], "link") == 0) {
  Tcl& tcl = Tcl::instance();
  LinkDelay* del = (LinkDelay*) TclObject::lookup(argv[2]);

  if (del == 0) {
    tcl.resultf("RED: no LinkDelay object %s",
            argv[2]);
    return(TCL_ERROR);
  }

  link_ = del;
  return (TCL_OK);
}

if (strcmp(argv[1], "early-drop-target") == 0) {
  Tcl& tcl = Tcl::instance();
  NsObject* p = (NsObject*)TclObject::lookup(argv[2]);

  if (p == 0) {
    tcl.resultf("no object %s", argv[2]);
    return (TCL_ERROR);
  }
  de_drop_ = p;
```

```
    return (TCL_OK);
  }
  if (strcmp(argv[1], "setSchedularMode") == 0) {
    setSchedularMode(argv[2]);
    return(TCL_OK);
  }

  if (strcmp(argv[1], "setMREDMode") == 0) {
    if (argc == 3)
      setMREDMode(argv[2],0);
    else
      setMREDMode(argv[2],argv[3]);
    return(TCL_OK);
  }

  if (strcmp(argv[1], "addQueueWeights") == 0) {
    addQueueWeights(atoi(argv[2]), atoi(argv[3]));
    return(TCL_OK);
  }

  if (strcmp(argv[1], "addQueueRate") == 0) {
    addQueueRate(atoi(argv[2]), atoi(argv[3]));
    return(TCL_OK);
  }
```

```
    // Returns the weighted RED queue length for one virtual queue in packets
    if (strcmp(argv[1], "getAverageV") == 0) {
      Tcl& tcl = Tcl::instance();
      tcl.resultf("%f",redq_[atoi(argv[2])].getWeightedLength_v(atoi(argv[3])));
      return(TCL_OK);
    }

    // Returns the length of one virtual queue, in packets
    if (strcmp(argv[1], "getCurrentV") == 0) {
      Tcl& tcl = Tcl::instance();
      tcl.resultf("%f",redq_[atoi(argv[2])].getRealLength_v(atoi(argv[3]))*1.0);
      return(TCL_OK);
    }
    return(Queue::command(argc, argv));
  }
```

APPENDIX D: Differentiated Services (red.h)

```
#ifndef dsred_h

#define dsred_h

#include "red.h"    // need RED class specs (edp definition, for example)

#include "queue.h"  // need Queue class specs

#include "dsredq.h"

/* The dsRED class supports the creation of up to MAX_QUEUES physical queues at

each network device, with up to MAX_PREC virtual queues in each queue. */

#define MAX_QUEUES 8// maximum number of physical RED queues

#define MAX_PREC 3  // maximum number of virtual RED queues in one physical queue

#define MAX_CP 40   // maximum number of code points in a simulation

#define MEAN_PKT_SIZE 1000       // default mean packet size, in bytes, needed for
RED calculations

enum schedModeType {schedModeRR, schedModeWRR, schedModeWIRR, schedModePRI};

#define PKT_MARKED 3

#define PKT_EDROPPED 2

#define PKT_ENQUEUED 1

#define PKT_DROPPED 0


/*-----------------------------------------------------------------------------

struct phbParam

    This struct is used to maintain entries for the PHB parameter table, used to map
a code point to a physical queue-virtual queue pair.

-----------------------------------------------------------------------------*/
```

```
struct phbParam {
   int codePt_;
   int queue_;      // physical queue
   int prec_;// virtual queue (drop precedence)
};
struct statType {
       long drops;       // per queue stats
       long edrops;
       long pkts;
       long valid_CP[MAX_CP];  // per CP stats
       long drops_CP[MAX_CP];
       long edrops_CP[MAX_CP];
       long pkts_CP[MAX_CP];
};
/*-----------------------------------------------------------------------
class dsREDQueue
    This class specifies the characteristics for a Diffserv RED router.
-----------------------------------------------------------------------*/
class dsREDQueue : public Queue {
 public:
  dsREDQueue();
  int command(int argc, const char*const* argv);     // interface to ns scripts
   protected:
  redQueue redq_[MAX_QUEUES];    // the physical queues at the router
```

```
  NsObject* de_drop_;            // drop_early target
  statType stats; // used for statistics gatherings
  int qToDq;              // current queue to be dequeued in a round robin manner
  int numQueues_;         // the number of physical queues at the router
  int numPrec;                    // the number of virtual queues in each physical
queue
  phbParam phb_[MAX_CP];         // PHB table
  int phbEntries;                // the current number of entries in the PHB table
  int ecn_;               // used for ECN (Explicit Congestion Notification)
  LinkDelay* link_;       // outgoing link
  int schedMode;                  // the Queue Scheduling mode
  int queueWeight[MAX_QUEUES];    // A queue weight per queue
  double queueMaxRate[MAX_QUEUES];   // Maximum Rate for Priority Queueing
  double queueAvgRate[MAX_QUEUES];   // Average Rate for Priority Queueing
  double queueArrTime[MAX_QUEUES];       // Arrival Time for Priority Queueing
  int slicecount[MAX_QUEUES];
  int pktcount[MAX_QUEUES];
  int wirrTemp[MAX_QUEUES];
  unsigned char wirrqDone[MAX_QUEUES];
  int queuesDone;
    void reset();
  void edrop(Packet* p); // used so flowmonitor can monitor early drops
  void enque(Packet *pkt); // enques a packet
  Packet *deque(void);    // deques a packet
```

```
  int getCodePt(Packet *p); // given a packet, extract the code point marking from
its header field

  int selectQueueToDeque();      // round robin scheduling dequing algorithm

  void lookupPHBTable(int codePt, int* queue, int* prec); // looks up queue and prec
numbers corresponding to a code point

  void addPHBEntry(int codePt, int queue, int prec); // edits phb entry in the table

  void setNumPrec(int curPrec);

  void setMREDMode(const char* mode, const char* queue);

  void printStats(); // print various stats

  double getStat(int argc, const char*const* argv);

  void printPHBTable();  // print the PHB table

  void setSchedularMode(const char* schedtype); // Sets the schedular mode

  // Add a weigth to a WRR or WIRR queue

  void addQueueWeights(int queueNum, int weight);

  // Add a maxRate to a PRI queue

  void addQueueRate(int queueNum, int rate);

  void printWRRcount();          // print various stats

  // apply meter to calculate average rate of a PRI queue

   void applyTSWMeter(int q_id, int pkt_size);

};

#endif
```