University Of Zimbabwe

Faculty of Science

Department of Computer Science

DESIGNING A QUALITY OF SERVICE FOR MOBILE AD HOC NETWORKS CARRYING MULTIMEDIA TRAFFIC

BY: TAWANDA ARTHUR NYAROTA

SUBERVISED BY MR B NYAMBO

A thesis submitted in partial fulfillment of the requirements of the Master of Science Degree in Computer Science

SEPTEMBER 2010

ABSTRACT

Multimedia transmission over communication networks has been growing rapidly with respect to the number of users and the amount and type of traffic. Wireless communications are a fast emerging technology and have become an essential feature of everyday life. Not only are computer networks becoming mobile, but each device is eventually having one or several wireless interfaces (e.g. laptops, cameras, phones etc.).Simultaneously, multimedia is having an equivalent growth, and is now constituting a large part of the traffic flowing through the wireless network through increased use in both the commercial as well as the social arena. These multimedia applications such as voice over IP, video conferencing, and video-on-demand, generate traffic with characteristics that differ significantly from the traffic generated by data applications, and they have more strict delay and packet loss requirements. Guaranteeing those requirements in wireless environments is very challenging because wireless links have variable characteristics especially through their mobile and ad hoc nature. To deal with this problem, many wireless communication standards have been defined enhancing the Quality of Service (QoS) of the whole system but do not specifically enhance multimedia traffic on its own. In this paper we propose mechanisms for modifying the existing QoS frameworks to include traffic prioritization, with multimedia traffic as the highest priority and to include different dropping probabilities for each class and congestion avoidance. We first study existing QoS frameworks then modify an existing framework (ASAP) to include traffic classification and bandwidth adaptation.

ACKNOWLEDGEMENTS

First and foremost I would like to extend my sincere appreciation to my supervisor Mr B Nyambo for his unwavering support, tolerance and advice, he introduced me to NS2 and MANETS so this research study would not have been a success without his help. Secondly I would like to thank my mother Ireen Nyarota for raising me up and still taking care of me even though Iam now an old man. I would also like to thank Tendai Muza for the love,the whole Computer Science department for welcoming me into this wonderful computer science family. All my friends , students and everyone who helped me in my masters degree.

Above all I thank Jesus Christ my savior, the holy spirit my guardian, and Jehovah the father without you I am nothing at all.

CONTENTS

LIST OF FIGURES

CHAPTER 1 INTRODUCTION

1.1 BACKGROUND

A mobile ad hoc network (MANET)[1] is a decentralized, self-organizing and self configuring wireless network, without any fixed infrastructure. In this network, each mobile node behaves not only as a host, but also as a router which is capable of communicating with other nodes, using either direct wireless links, or multi-hop wireless links. Examples of ad hoc network applications include mobile cellphones,3G/4G internet connections on mobile devices, business associates sharing information during meetings or conferences, soldiers relaying information on a battlefield, and disaster relief personnel coordinating efforts after fires or earthquakes.
With such applications, MANETs are becoming key components in the 4G
Architecture, as they offer multimedia services to mobile users in areas with no preexisting Communications infrastructure. As nodes are mobile in a MANET, links are created and destroyed in an unpredictable way, which makes quite challenging the provision of a good Quality of Service between a pair of nodes that want to communicate with each other. It has been noted that MANETs, though becoming popular, experience serious challenges in imposing any form of reliability and in meeting the demands of throughput-sensitive and delay-sensitive multimedia applications and services. Besides the general challenges that exist for Quality of Service (QoS)[2] in all networks, MANETs also face the following constraints: dynamic topology, limited processing and storing capabilities, limited battery power ,unpredictable link properties, and bandwidth constraints. Research has been and is still being conducted so as to address these challenges in order to provide support for these multimedia applications and services. Multimedia applications (especially real-time) impose heavy requirements on communication parameters, such as bandwidth, throughput, and drop rate which are already constrained by the very nature of MANETS. Current QoS frameworks classify traffic into real-time and Best-effort traffic only, but this is not adequate to provide a satisfactory QoS for multimedia traffic. There is need to prioritize traffic into classes with different QoS provisions on the same channel and to avoid congestion on the channel. The DiffServ[3] architecture provides a way for providing differentiated services for different types of traffic flows on the same channel. The Differentiated Services Random Early Discard (DS Red) queue can be modified to avoid congestion and adopt different dropping probabilities depending on the

bandwidth available. In this research study we aim to propose traffic differentiation through the use of Differentiated Services Code Points (DSCP), congestion avoidance through modification of the DS Red queue, and adaptive dropping probability algorithms for traffic flows in MANETS carrying Multimedia traffic and incorporate them in an existing QoS framework (ASAP)[4].

1.2 PROBLEM DEFINITION

- QoS requirements for multimedia applications are intrinsically different from simple network applications as multimedia services require more bandwidth and lossless delivery. Current QoS frameworks treat all real-time traffic in same manner but these networks must have some mechanism to differentiate traffic into different classes with different QoS provisions.

- Due to the failure of current QoS frameworks to differentiate traffic into classes with different priorities use of default feeding rates on high requirement applications especially Multimedia traffic usually result in congestion and a degradation of the QoS provided on multimedia traffic resulting in dropping of packets and wastage of the scarce battery power in mobile ad hoc networks.
- Most Multimedia traffic is delay sensitive so if on congestion the packets drop at default rates equal to other traffic classes the information successfully sent will not make sense as it would be deformed thus the need to make multimedia traffic drop at the lowest rate possible.

1.3 Research Aim and Objectives

The main aim of this research study is to design a Quality of Service Framework for Mobile ad hoc networks carrying Multimedia traffic.

The objectives of this research include:

- To study and analyse existing Quality of Service frameworks for MANETS.
- To classify network traffic into classes with different priorities.
- To avoid channel congestion

- To be able to assign different dropping probabilities to different traffic classes according to the available bandwidth.

1.4 Justification of the Study

Ad hoc Networks have limited wireless resources; time varying topology and lack of infrastructure. Theses factors make it difficult to provide QoS in MANETs.

The evolution of the multimedia technology and the commercial interest of companies to reach widely civilian applications has made QoS in MANETs an area of great interest. This evolution calls for a complete solution for QoS in MANETs in order to provide effective communication of real-time traffic. With an increase in multimedia applications, there is a need for wireless networks to provide some QoS guarantees for highly sensitive application. An increase in the high requirement traffic load increases the network congestion due to the failure of the traffic differentiation schemes to adapt to new traffic application requirements to the network available bandwidth. This research study seeks to achieve the QoS guarantees through the designing and proposing use of the DS Code Points[5] to differentiate traffic into classes with different priorities so as to give Multimedia traffic superior treatment at the expense of other traffic classes.

The justifications of the study at a glance are:

- ❖ The failure of current QoS frameworks to classify traffic and give Multimedia traffic preferential treatment.
- ❖ The increase in QoS demands of throughput-sensitive and delay-sensitive Multimedia applications and services in MANETS
- ❖ The need to control and manage resources in a way that guarantees high QoS in MANETs carrying Multimedia traffic.

1.5 Overview of the Research Study

The structure of the rest of the thesis will be as follows: In chapter 2 we review relevant literature and provide necessary background information for our project. Chapter 3 describes our approach and relevant tools used for the project. Chapter 4 describes our simulation results for the evaluation of the traffic prioritization, congestion avoidance and the adaptive dropping rates of

the different traffic classes, it also provides our observation and analysis of the results. Chapter 5 concludes the thesis and suggests future work.

Abbreviations

CBR	Constant Bit Rate
DSCP	Differentiated Services Code Points
CIR	Committed Information Rate
PIR	Peak Information Rate
EBS	Excess Burst Size
NS 2	Network Simulator version 2

CHAPTER 2. LITERATURE REVIEW

2.1 INTRODUCTION

This Chapter presents a detailed view of Mobile Ad Hoc Networks and the issues involved in the provision of Quality of Service in these networks. A review of available QoS models for Manets is also presented to highlight the extend to which some researchers have reached in trying to come up with QoS Schemes for Manets.

2.2 Manets and QoS

2.2.1 Manets Features

A Mobile Ad hoc Network (MANET) is an autonomous system of mobile nodes connected by wireless links and communicating over relatively bandwidth-constrained links. Each node operates not only as an end-system, but also as a router to forward packets. The network is decentralized, where all network activity including discovering the topology and delivering messages must be executed by the nodes themselves, i.e., routing functionality will be incorporated into mobile nodes. The nodes are free to move about and organize themselves into a network, thus the network topology may change rapidly and unpredictably over time.

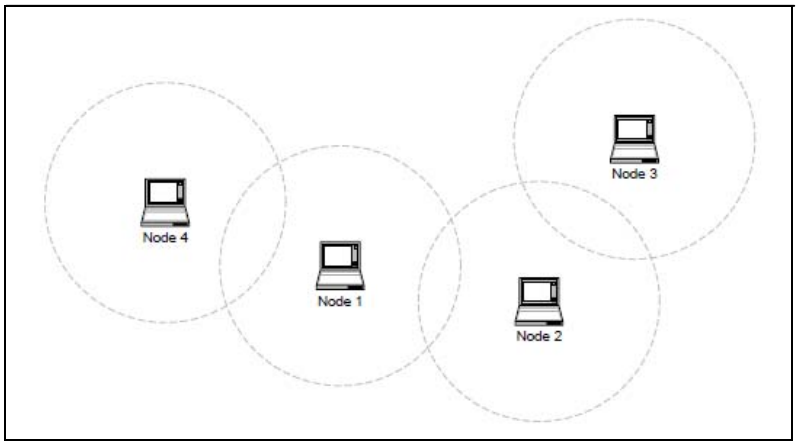


Figure 1: MANET Example

MANETs do not require any fixed infrastructure such as base stations; therefore, they are an attractive networking option for connecting mobile devices quickly and spontaneously especially in the coming generation of wireless communication systems, where there will be a need for the rapid deployment of independent mobile users. Significant examples include establishing survivable, efficient, dynamic communication for emergency/rescue operations, disaster relief efforts, and military networks. Such network scenarios cannot rely on centralized and organized connectivity, and can be conceived as applications of Mobile Ad Hoc Networks.

Thus the main characteristics/features of Manets include the following:

- Multiple node functionalities: Each node in Manets has twofold functions, i.e. host and router/ switch. As a router, a node routes packets for other nodes in the same way as the backbone routers in the Internet.
- Node mobility and non-infrastructure: node mobility is the basic cause of the dynamic network topologies in Manets. The Mac layer allocation of bandwidth to each node changes dynamically according to mobility scenarios. Also, bandwidth is also difficult to control due to the non-infrastructure feature. The roles of nodes as a host or router change too together with node mobility and the dynamic topology.
- Power Constraints: the nodes' processing capability is limited due to the limited battery power. This means there should be low processing overheads of nodes and thus, the control algorithms and QoS algorithms should use bandwidth and energy efficiently.

The set of applications for MANETs is diverse, ranging from small, static networks that are constrained by power sources, to large-scale, mobile, highly dynamic networks. The design of network protocols for these networks is a complex issue.

Regardless of the application, MANETs need efficient distributed algorithms to determine network organization, link scheduling, and routing. However, determining viable routing paths and delivering messages in a decentralized environment where

network topology fluctuates is not a well-defined problem. While the shortest path (based on a given cost function) from a source to a destination in a static network is usually the optimal route, this idea is not easily extended to MANETs. Factors such as variable wireless link quality, propagation path loss, fading, multi-user interference, power expended, and topological changes, become relevant issues. The network should be able to adaptively alter the routing paths to alleviate any of these effects. Moreover, in a military environment, preservation of security, latency, reliability, intentional jamming, and recovery from failure are significant concerns. Military networks are designed to maintain a low probability of intercept and/or a low probability of detection. Hence, nodes prefer to radiate as little power as necessary and transmit as infrequently as possible, thus decreasing the probability of detection or interception. A lapse in any of these requirements may degrade the performance and dependability of the network.

2.3 Quality of Service

QoS is defined as a set of service requirements that needs to be met by the network while transporting a packet stream from a source to its destination This term refers to a measure of performance for a transmission system that reflects its transmission quality and availability of service. In computer networking this term Quality of Service (QoS) refers to the probability of the network meeting a given traffic contract, or in many cases is used informally to refer to the probability of a packet passing between two points in the network.

The drive for QoS has become very strong in recent years because there has been a growth of multimedia traffic such as voice and video, that mixes it with more traditional data traffic such as FTP, Telnet and SMB. Applications such as Voice over IP and Microsoft's Netmeeting have opened up the possibility of more interactive communications between users not just over the LAN but also over the WAN.

2.3.1 Sensitive Traffic

Much data traffic has been very tolerant of delays and packet drops, however voice traffic has different characteristics. Voice is low volume but is intolerant of jitter (delay) and packet loss. Video is also intolerant of jitter and packet loss, plus it has the added complication of being very bursty at times. This convergence of multimedia traffic with traditional data traffic is set to grow

and therefore requires methods and tools to ensure that providers can deliver networks that give the users confidence to use these tools effectively.

2.3.2 Where QoS is Applied

The most significant network bottlenecks exist at the Remote Access points, the WAN access, Internet access and the servers. Once a packet is on the core of the network, it generally deserves to be there, provided that your security is intact. Many of the technologies involved in QoS are to do with how packets are dealt with as they enter and leave a network because merely adding more bandwidth at the edge is only a short term solution that just resolves capacity and perhaps some congestion problems. Adding bandwidth does not resolve jitter or add any traffic prioritisation features.

Problems faced when attempting to provide QoS include:

- A shortage of bandwidth because network links are oversubscribed.
- Packets being lost due to congestion at bursty periods. Modern DSPs can fill in for between 20ms to 50ms of lost voice.
- End-to-End delay made up by a number of components that occur in the following order:
    1. Fixed Switch Delay - as packets are layer 2 switched from the call initiator.
    2. Fixed Encoding Delay
    3. Fixed Voice Activity Detection (VAD**)** - around 5ms.
    4. Fixed Packetisation Delay
    5. Variable Output Queuing Delay - as voice packets enter an output queue and wait for the preceding frame (voice or data) to be played out.
    6. Fixed Serialisation Delay - the time it takes to get the bits on to the circuit. The faster the circuit and the smaller the packet, the less this delay.
    7. Fixed Processing Delay - the time it takes for the packets to be examined, routed, managed etc.
    8. Variable Network Queuing Delay - due to clocking frames in and out of the different network switches.
    9. Fixed Network Propagation Delay - normally assumed to be 6 microseconds/km or 10 microseconds/mile as the packets traverse the medium.

10. Variable Input Queuing Delay - as voice packets enter an input queue and wait for the preceding frame (voice or data) to be played out.
11. Fixed Dejitter Buffer Delay - can be up to 50ms.
12. Fixed Switch Delay - as packets are layer 2 switched to the endpoint.
13. Fixed Decoding Delay - decoding the compressed packets is trivial compared to encoding and so takes less time.

2.3.3 Functions of QoS

QoS needs to enable:

- Predictable Response times
- Management of delay sensitive applications
- Management of jitter sensitive applications
- Control of packet loss when congestion occurs during a burst (note that continuous congestion means that the link is over-subscribed)
- The setting of traffic priorities.
- Dedication of bandwidth on a per application basis.
- The avoidance of congestion
- The management of congestion when it occurs, note that this is different from trying to avoid congestion, sometimes it is not possible to avoid congestion.

QoS can be thought of as operating on one or more of three levels:

- Best Effort
- Differentiated Service - can deal with differing levels of QoS on a packet by packet basis
- Integrated Service - requested level of service by an application to a network. Data is only sent after confirmation that the service level is guaranteed to be available.

The order of events with respect to QoS can be described as follows:

1. Mark and Classify packets according to policies and the behaviour of the traffic. This is carried out with technologies such as IP Precedence and DSCP and is most effective

when carried out as far to the edge of the network as possible, even at the device itself (e.g. a VoIP phone).

2. Congestion management by prioritising traffic based on the marks using queuing technologies that can respond to traffic classes.
3. Avoid congestion by dropping packets that are not a high priority using technologies such as Random Early Detection where low priority packets are weeded out of a queue so as to prevent indescriminant 'Tail Drop' of packets.
4. Police traffic so that bad packets are kept at bay i.e. limit the ingress or egress traffic depending on the class/markings of the packets. Also perform traffic shaping to maximise the use of bandwidth by specifying peak and average traffic rates.
5. Fragment and compress packets to maximise the use of WAN bandwidths.

2.3.4 QoS Concepts

Fundamentally, QoS enables you to provide better service to certain flows. This is done by either raising the priority of a flow or limiting the priority of another flow. When using congestion-management tools, you try to raise the priority of a flow by queuing and servicing queues in different ways. The queue management tool used for congestion avoidance raises priority by dropping lower-priority flows before higher-priority flows. Policing and shaping provide priority to a flow by limiting the throughput of other flows. Link efficiency tools limit large flows to show a preference for small flows.

2.3.5 Basic QoS Architecture

The basic architecture introduces the three fundamental pieces for QoS implementation

- QoS identification and marking techniques for coordinating QoS from end to end between network elements

- QoS within a single network element (for example, queuing, scheduling, and traffic-shaping tools)

• QoS policy, management, and accounting functions to control and administer end-to-end traffic across a network

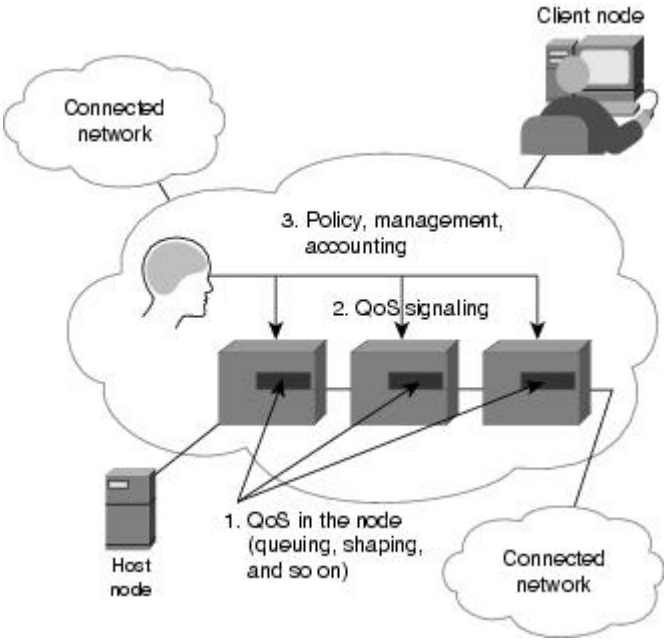


Figure 2 Basic QoS architecture

2.3.6 QoS Identification and Marking

Identification and marking is accomplished through classification and reservation.

2.3.6.1 Classification

To provide preferential service to a type of traffic, it must first be identified. Secondly, the packet may or may not be marked. These two tasks make up classification. When the packet is identified but not marked, classification is said to be on a per-hop basis. This is when the classification pertains only to the device that it is on, not passed to the next router. This happens with priority queuing (PQ) and custom queuing (CQ). When packets are marked for network-wide use, IP precedence bits can be set. Common methods of identifying flows include access control lists (ACLs), policy-based routing, committed access rate (CAR), and network-based application recognition (NBAR).

2.3.7 QoS Within a Single Network Element

Congestion management, queue management, link efficiency, and shaping/policing tools provide QoS within a single network element.

2.3.7.1 Congestion Management

Because of the bursty nature of voice/video/data traffic, sometimes the amount of traffic exceeds the speed of a link. At this point, what will the router do? Will it buffer traffic in a single queue and let the first packet in be the first packet out? Or, will it put packets into different queues and service certain queues more often? Congestion-management tools address these questions. Tools include priority queuing (PQ), custom queuing (CQ), weighted fair queuing (WFQ)[7], and class-based weighted fair queuing (CBWFQ).

2.3.7.2 Queue Management

Because queues are not of infinite size, they can fill and overflow. When a queue is full, any additional packets cannot get into the queue and will be dropped. This is a tail drop. The issue with tail drops is that the router cannot prevent this packet from being dropped (even if it is a high-priority packet). So, a mechanism is necessary to do two things:

1. Try to make sure that the queue does not fill up, so that there is room for high-priority packets

2. Allow some sort of criteria for dropping packets that are of lower priority before dropping higher-priority packets

Weighted early random detect (WRED) provides both of these mechanisms.

2.3.7.3 Link Efficiency

Many times low-speed links present an issue for smaller packets. For example, the serialization delay of a 1500-byte packet on a 56-kbps link is 214 milliseconds. If a voice packet were to get behind this big packet, the delay budget for voice would be exceeded even before the packet left

the router! Link fragmentation and interleave allow this large packet to be segmented into smaller packets interleaving the voice packet. Interleaving is as important as the fragmentation. There is no reason to fragment the packet and have the voice packet go behind all the fragmented packets.

2.4 Traffic Shaping and Policing

Shaping is used to create a traffic flow that limits the full bandwidth potential of the flow(s). This is used many times to prevent the overflow problem. For instance, many network topologies use Frame Relay in a hub-and-spoke design. In this case, the central site normally has a high-bandwidth link (say, T1), while remote sites have a low-bandwidth link in comparison (say, 384 Kbps). In this case, it is possible for traffic from the central site to overflow the low bandwidth link at the other end. Shaping is a perfect way to pace traffic closer to 384 Kbps to avoid the overflow of the remote link. Traffic above the configured rate is buffered for transmission later to maintain the rate configured.

Policing is similar to shaping, but it differs in one very important way: Traffic that exceeds the configured rate is not buffered (and normally is discarded).

2.4.1 End-to-End QoS Levels

Service levels refer to the actual end-to-end QoS capabilities, meaning the capability of a network to deliver service needed by specific network traffic from end to end or edge to edge. The services differ in their level of *QoS strictness*, which describes how tightly the service can be bound by specific bandwidth, delay, jitter, and loss characteristics.

Three basic levels of end-to-end QoS can be provided across a heterogeneous network, as shown in Figure 3

- **Best-effort service**—Also known as lack of QoS, best-effort service is basic connectivity with no guarantees. This is best characterized by FIFO queues, which have no differentiation between flows.

- **Differentiated service (also called soft QoS)**—Some traffic is treated better than the rest (faster handling, more average bandwidth, and lower average loss rate). This is a statistical preference, not a hard and fast guarantee. This is provided by classification of traffic and the use of QoS tools such as PQ, CQ, WFQ, and WRED (all discussed later in this chapter).

- **Guaranteed service (also called hard QoS)**—This is an absolute reservation of network resources for specific traffic. This is provided through QoS tools RSVP and CBWFQ (discussed later in this chapter).

Deciding which type of service is appropriate to deploy in the network depends on several factors:

- The application or problem that the customer is trying to solve. Each of the three types of service is appropriate for certain applications. This does not imply that a customer must migrate to differentiated and then to guaranteed service (although many probably eventually will). A differentiated service—or even a best-effort service—may be appropriate, depending on the customer application requirements.

- The rate at which customers can realistically upgrade their infrastructures. There is a natural upgrade path from the technology needed to provide differentiated services to that needed to provide guaranteed services, which is a superset of that needed for differentiated services.

- The cost of implementing and deploying guaranteed service is likely to be more than that for a differentiated service.

Figure 3 The Three Levels of End-to-End QoS Are Best-Effort Service, Differentiated Service, and Guaranteed Service

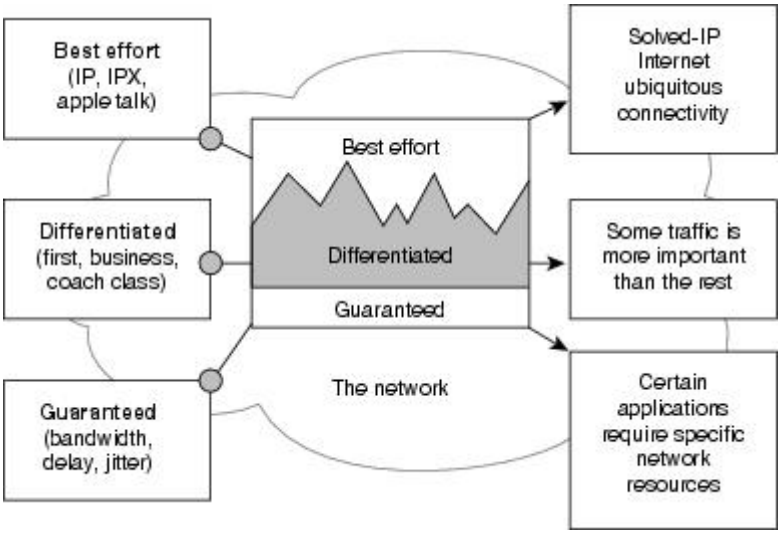


Figure 3 Levels of end to end QoS

2.4.2 Classification—Identifying Flows

To provide priority to certain flows, the flow must first be identified and (if desired) marked. These two tasks are commonly referred to as just *classification*.

Historically, identification was done using access control lists (ACLs). ACLs identify traffic for congestion-management tools such as PQ and CQ. Because PQ and CQ are placed on routers on a hop-by-hop basis (that is, priority settings for QoS pertain only to that router and are not passed to subsequent router hops in the network), identification of the packet is used only within a single router. In some instances, CBWFQ classification is for only a single router. This is contrasted by setting IP precedence bits.

Features such as policy-based routing and committed access rate (CAR) can be used to set precedence based on extended access list classification. This allows considerable flexibility for precedence assignment, including assignment by application or user, by destination and source subnet, and so on. Typically this functionality is deployed as close to the edge of the network (or administrative domain) as possible so that each subsequent network element can provide service based on the determined policy.

Network-based application recognition (NBAR) is used to identify traffic more granularly. For example, URLs in an HTTP packet can be identified. Once the packet has been identified, it can be marked with a precedence setting.

2.4.3 CAR: Setting IP Precedence

Similar in some ways to PBR, the CAR feature enables you to classify traffic on an incoming interface. It also allows specification of policies for handling traffic that exceeds a certain bandwidth allocation. CAR looks at traffic received on an interface, or a subset of that traffic selected by access list criteria, compares its rate to that of a configured token bucket, and then takes action based on the result (for example, drop or rewrite IP precedence).

There is some confusion with using CAR to set IP precedence bits. An attempt to clear up any confusion follows. CAR (as its name describes) is used to police traffic flows to a *committed access rate*. CAR does this with a token bucket. A token bucket is a bucket with tokens in it that represent bytes (1 token = 1 byte). The bucket is filled with tokens at a user-configured rate. As packets arrive to be delivered, the system checks the bucket for tokens. If there are enough tokens in the bucket to match the size of the packet, those tokens are removed and the packet is passed (this packet *conforms*). If there aren't enough tokens, the packet is dropped (this packet *exceeds*).

Committed Rate That Is Decided Upon

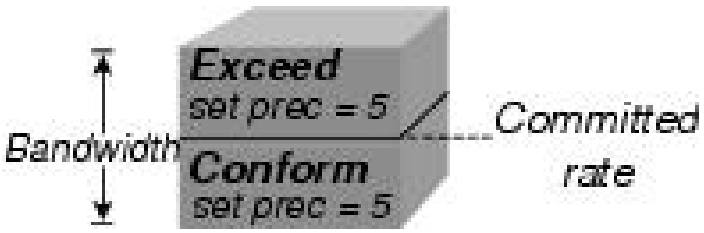


Figure 4 Committed Rate

When IP precedence is set in the host or network client, this setting can be used optionally; however, this can be overridden by policy within the network. IP precedence enables service classes to be established using existing network queuing mechanisms (for example, WFQ or WRED), with no changes to existing applications or complicated network requirements. Note that this same approach is easily extended to IPv6 using its Priority field.

2.44 IP Precedence: Differentiated QoS

IP precedence utilizes the 3 precedence bits in the IPv4 header's Type of Service (ToS) field to specify class of service for each packet, as shown in. You can partition traffic in up to six classes of service using IP precedence (two others are reserved for internal network use). The queuing technologies throughout the network can then use this signal to provide the appropriate expedited handling.

This Diagram Shows the IP Precedence ToS Field in an IP Packet Header

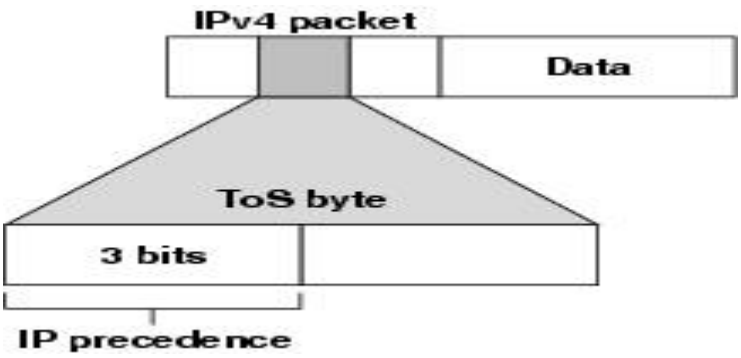


Figure 5 IP Precedence ToS field

The 3 most significant bits (correlating to binary settings 32, 64, and 128) of the Type of Service (ToS) field in the IP header constitute the bits used for IP precedence. These bits are used to provide a priority from 0 to 7 (settings of 6 and 7 are reserved and are *not* to be set by a network administrator) for the IP packet.

Because only 3 bits of the ToS byte are used for IP precedence, you need to differentiate these bits from the rest of the ToS byte. In Figure 49-5, a 1 in the first and third bit positions (viewing from left to right) correlates to an IP precedence setting of 5, but when viewing the ToS byte in a

Traffic that is identified can be marked by setting the IP precedence bits. Thus, it needs to be classified only once. RFC 2475 extends the number of bits used in the ToS byte from 3 to 6. The 6 MSBs will be used for precedence settings (known as DS codepoints), with the 2 least significant bits (the right-most 2 bits) reserved for future use. This specification is commonly referred to as *DiffServ*.

2.5 Congestion-Management Tools

One way network elements handle an overflow of arriving traffic is to use a queuing algorithm to sort the traffic, and then determine some method of prioritizing it onto an output link. Cisco IOS[8] software includes the following queuing tools:

- First-in, first-out (FIFO) queuing
- Priority queuing (PQ)
- Custom queuing (CQ)
- Flow-based weighted fair queuing (WFQ)
- Class-based weighted fair queuing (CBWFQ)

Each queuing algorithm was designed to solve a specific network traffic problem and has a particular effect on network performance, as described in the following sections.

Note Queuing algorithms take effect when congestion is experienced. By definition, if the link is not congested, then there is no need to queue packets. In the absence of congestion, all packets are delivered directly to the interface.

2.5.1 FIFO: Basic Store-and-Forward Capability

In its simplest form, *FIFO* queuing involves storing packets when the network is congested and forwarding them in order of arrival when the network is no longer congested. FIFO is the default queuing algorithm in some instances, thus requiring no configuration, but it has several shortcomings. Most importantly, FIFO queuing makes no decision about packet priority; the

order of arrival determines bandwidth, promptness, and buffer allocation. Nor does it provide protection against ill-behaved applications (sources). Bursty sources can cause long delays in delivering time-sensitive application traffic, and potentially to network control and signaling messages. FIFO queuing was a necessary first step in controlling network traffic, but today's intelligent networks need more sophisticated algorithms. In addition, a full queue causes tail drops. This is undesirable because the packet dropped could have been a high-priority packet. The router couldn't prevent this packet from being dropped because there was no room in the queue for it (in addition to the fact that FIFO cannot tell a high-priority packet from a low-priority packet). Cisco IOS software implements queuing algorithms that avoid the shortcomings of FIFO queuing.

2.5.2 PQ: Prioritizing Traffic

PQ ensures that important traffic gets the fastest handling at each point where it is used.
It was designed to give strict priority to important traffic. Priority queuing can flexibly prioritize according to network protocol (for example IP, IPX, or AppleTalk), incoming interface, packet size, source/destination address, and so on. In PQ, each packet is placed in one of four queues—high, medium, normal, or low—based on an assigned priority. Packets that are not classified by this priority list mechanism fall into the normal queue. During transmission, the algorithm gives higher-priority queues absolute preferential treatment over low-priority queues. PQ is useful for making sure that mission-critical traffic traversing various WAN links gets priority treatment

2.5.3 CQ: Guaranteeing Bandwidth

CQ was designed to allow various applications or organizations to share the network among applications with specific minimum bandwidth or latency requirements. In these environments, bandwidth must be shared proportionally between applications and users. You can use the Cisco CQ feature to provide guaranteed bandwidth at a potential congestion point, ensuring the specified traffic a fixed portion of available bandwidth and leaving the remaining bandwidth to other traffic. Custom queuing handles traffic by assigning a specified amount of queue space to each class of packets and then servicing the queues in a round-robin fashion.

2.5.5 Flow-Based WFQ: Creating Fairness Among Flows

For situations in which it is desirable to provide consistent response time to heavy and light network users alike without adding excessive bandwidth, the solution is flow-based WFQ (commonly referred to as just WFQ). WFQ is one of Cisco's premier queuing techniques. It is a flow-based queuing algorithm that creates bit-wise fairness by allowing each queue to be serviced fairly in terms of byte count. For example, if queue 1 has 100-byte packets and queue 2 has 50-byte packets, the WFQ algorithm will take two packets from queue 2 for every one packet from queue 1. This makes service fair for each queue: 100 bytes each time the queue is serviced.

WFQ ensures that queues do not starve for bandwidth and that traffic gets predictable service. Low-volume traffic streams—which comprise the majority of traffic—receive increased service, transmitting the same number of bytes as high-volume streams. This behavior results in what appears to be preferential treatment for low-volume traffic, when in actuality it is creating fairness. With WFQ, If High-Volume Conversations Are Active, Their Transfer Rates and Interarrival Periods Are Made Much More Predictable

WFQ is designed to minimize configuration effort, and it automatically adapts to changing network traffic conditions. In fact, WFQ does such a good job for most applications that it has been made the default queuing mode on most serial interfaces configured to run at or below E1 speeds (2.048 Mbps).

Flow-based WFQ creates *flows* based on a number of characteristics in a packet. Each flow (also referred to as a *conversation*) is given its own queue for buffering if congestion is experienced. The following descriptions use flow, conversation, and queue interchangeably.

The weighted portion of WFQ comes from the use of IP precedence bits to provide greater service for certain queues. Using settings 0 to 5 (6 and 7 are reserved), WFQ uses its algorithm to determine how much more service to provide to a queue.WFQ is efficient in that it uses whatever bandwidth is available to forward traffic from lower-priority flows if no traffic from higher-priority flows is present. This is different from strict time-division multiplexing (TDM), which simply carves up the bandwidth and lets it go unused if no traffic is present for a particular traffic type. WFQ works with both—IP precedence and Resource Reservation Protocol (RSVP).The

WFQ algorithm also addresses the problem of round-trip delay variability. If multiple high-volume conversations are active, their transfer rates and interarrival periods are made much more predictable. This is created by the bit-wise fairness. If conversations are serviced in a consistent manner with every round-robin approach, delay variation (or jitter) stabilizes. WFQ greatly enhances algorithms such as SNA Logical Link Control (LLC) and the Transmission Control Protocol (TCP) congestion control and slow-start features. The result is more predictable throughput and response time for each active flow.

2.5.6 Cooperation Between WFQ and QoS Signaling Technologies

WFQ is IP precedence-aware; that is, it is capable of detecting higher-priority packets marked with precedence by the IP forwarder and can schedule them faster, providing superior response time for this traffic. This is the weighted portion of WFQ. The IP Precedence field has values between 0 (the default) and 7 (6 and 7 are reserved and normally are not set by network administrators). As the precedence value increases, the algorithm allocates more bandwidth to that conversation to make sure that it is served more quickly when congestion occurs. WFQ assigns a weight to each flow, which determines the transmit order for queued packets. In this scheme, lower weights are provided more service. IP precedence serves as a divisor to this weighting factor. For instance, traffic with an IP Precedence field value of 7 gets a lower weight than traffic with an IP Precedence field value of 3, and thus has priority in the transmit order.

Note A *weight* is a number calculated from the IP precedence setting for a packet in flow. This weight is used in WFQ's algorithm to determine when the packet will be serviced.

Weight = (4096 / (IP precedence + 1)
Weight = (32384 / (IP precedence + 1)

The numerator of the equation changed from 4096 to 32384 in a v12.0 maintenance release. Weight settings can be viewed using the **show queue <interface>** command.

Effect of IP Precedence Settings

The effect of IP precedence settings is described here:
If you have one flow at each precedence level on an interface, each flow will get precedence + 1 parts of the link, as follows:

1 + 2 + 3 + 4 + 5 + 6 + 7 + 8 = 36

The flows will get 8/36, 7/36, 6/36, and 5/36 of the link, and so on. However, if you have 18 precedence—1 flow and 1 of each of the others—the formula looks like this:

1 + 1 8 ¥ 2 + 3 + 4 + 5 + 6 + 7 + 8 = 36 - 2 + 18 ¥ 2 = 70

The flows will get 8/70, 7/70, 6/70, 5/70, 4/70, 3/70, 2/70, and 1/70 of the link, and 18 of the flows will each get approximately 2/70 of the link.

WFQ is also RSVP-aware; RSVP uses WFQ to allocate buffer space and schedule packets, and it guarantees bandwidth for reserved flows. Additionally, in a Frame Relay network, the presence of congestion is flagged by the forward explicit congestion notification (FECN) and backward explicit congestion notification (BECN) bits. WFQ weights are affected by Frame Relay discard eligible (DE), FECN, and BECN bits when the traffic is switched by the Frame Relay switching module. When congestion is flagged, the weights used by the algorithm are altered so that the conversation encountering the congestion transmits less frequently.

2.5.7 Queue Management (Congestion-Avoidance Tools)

Congestion avoidance is a form of queue management. *Congestion-avoidance techniques* monitor network traffic loads in an effort to anticipate and avoid congestion at common network bottlenecks, as opposed to congestion-management techniques that operate to control congestion *after* it occurs

2.5.7.1 WRED: Avoiding Congestion

The *random early detection (RED)*[9] algorithms are designed to avoid congestion in internetworks before it becomes a problem. RED works by monitoring traffic load at points in

the network and stochastically discarding packets if the congestion begins to increase. The result of the drop is that the source detects the dropped traffic and slows its transmission. RED is primarily designed to work with TCP in IP internetwork environments.

2.5.7.2 WRED Cooperation with QoS Signaling Technologies

WRED combines the capabilities of the RED algorithm with IP precedence. This combination provides for preferential traffic handling for higher-priority packets. It can selectively discard lower-priority traffic when the interface starts to get congested and can provide differentiated performance characteristics for different classes of service.Within each queue, a finite number of packets can be housed. A full queue causes tail drops. *Tail drops* are dropped packets that could not fit into the queue because the queue was full. This is undesirable because the packet discarded may have been a high-priority packet and the router did not have a chance to queue it. If the queue is not full, the router can look at the priority of all arriving packets and drop the lower-priority packets, allowing high-priority packets into the queue. Through managing the depth of the queue (the number of packets in the queue) by dropping various packets, the router can do its best to make sure that the queue does not fill and that tail drops are not experienced. This allows the router to make the decision on which packets get dropped when the queue depth increases. WRED also helps prevent overall congestion in an internetwork. WRED uses a minimum threshold for each IP precedence level to determine when a packet can be dropped. (The minimum threshold must be exceeded for WRED to consider a packet as a candidate for being dropped.)

Take a look at this WRED example:

Depth of the queue: 21 packets

Minimum drop threshold for IP precedence = 0: 20

Minimum drop threshold for IP precedence = 1: 22

Because the minimum drop threshold for IP precedence = 0 has been exceeded, packets with an IP precedence = 0 can be dropped. However, the minimum drop threshold for IP precedence = 1

has not been exceeded, so those packets will not be dropped. If the queue depth deepens and exceeds 22, then packets with IP precedence = 1 can be dropped as well. WRED uses an algorithm that raises the probability that a packet can be dropped as the queue depth rises from the minimum drop threshold to the maximum drop threshold. Above the maximum drop threshold, all packets are dropped.

2.5.7.3 Flow RED: RED for Non-TCP-Compliant Flows

WRED is primarily used for TCP flows that will scale back transmission if a packet is dropped. There are non-TCP-compliant flows that do not scale back when packets are dropped. Flow RED is used to deal with such flows. The approach is to increase the probability of dropping a flow if it exceeds a threshold.

Flow-based WRED relies on these two main approaches to remedy the problem of linear packet dumping:

- It classifies incoming traffic into flows based on parameters such as destination and source addresses and ports.

- It maintains state about active flows, which are flows that have packets in the output queues.

Flow-based WRED uses this classification and state information to ensure that each flow does not consume more than its permitted share of the output buffer resources. Flow-based WRED determines which flows monopolize resources, and it more heavily penalizes these flows.

This is how flow-based WRED ensures fairness among flows: It maintains a count of the number of active flows that exist through an output interface. Given the number of active flows and the output queue size, flow-based WRED determines the number of buffers available per flow.

To allow for some burstiness, flow-based WRED scales the number of buffers available per flow by a configured factor and allows each active flow to have a certain number of packets in the output queue. This scaling factor is common to all flows. The outcome of the scaled number of buffers becomes the per-flow limit. When a flow exceeds the per-flow limit, the probability that a packet from that flow will be dropped increases.

Active Queue Management (AQM) can improve the performance of TCP, and has been recommended by the Internet Engineering Task Force (IETF) for use in the routers of the next generation Internet . The goals of AQM is three folds. First, to improve *throughput* by reducing the number of packets dropped. This is achieved by keeping the average queue length small in order to absorb naturally occurring bursts without dropping packets. Second, AQM provides low *delay* to interactive services by maintaining a small average queue length. Third, AQM avoids the *lock out* phenomenon arising from tail drop. Lock out phenomena results in a single connection (or a few flows) monopolizing the buffer space which, in turn, results in packets from other connections being dropped and causing unfairness in bandwidth sharing among connections. Random Early Detection (RED) an AQM scheme, was originally proposed to solve the global synchronization problem in TCP/IP based networks. It uses a *single linear drop function* to calculate the drop probability of a packet, and has four parameters and an average queue size to regulate its performance. The four parameters are *Minth* and *Maxth* which represent buffer thresholds for packet drop at the queue, *Maxdrop* represents the maximum drop probability at *Maxth*, and w is a weight parameter to calculate the average queue size.

The idea behind RED is to provide, as soon as is possible, a feedback to responsive flows (like TCP) before the queue overflows in an effort to indicate that congestion is inminent, instead of waiting until the congestion has become excessive. Also, packet drops are distributed more fairly across all flows.

The RED gateway calculates the average queue size, using a low-pass filter with an exponential weighted moving average. The average queue size is compared to two thresholds, a minimum threshold and a maximum threshold. When the average queue size is less than the minimum threshold, no packets are marked. When the average queue size is greater than the maximum threshold, every arriving packet is marked. If marked packets are in fact dropped, or if all source nodes are cooperative, this ensures that the average queue size does not significantly exceed the maximum threshold. When the average queue size is between the minimum and the maximum threshold, each arriving packet is marked with probability pa, where pa is a function of the average queue size avg. Each time that a packet is marked, the probability that a packet is marked from a particular connection is roughly proportional to that connection's share of the bandwidth at the gateway. The general RED gateway algorithm is given in Figure 6

Figure 6 RED Algorithm

```
Initialization
    avg = 0
    count = 1
end
for each packet arrival
    calculate the new average queue size `avg'
        if the queue is nonempty
            avg = (1 - wq) * avg + wq * q  // this is the low-pass filter EWMA
        else
            m = f(time - q_time)
            avg = (1 - wq)^m * avg
        end
        if minth <= avg < maxth
            increment count
            calculate probability `pa'
                pb = maxp * (avg - minth) / (maxth - minth)
                pa = pb / (1 - count * pb)
            end
            with probability `pa'
                mark the arriving packet
                count = 0
            end
        end
        else if maxth <= avg
            mark the arriving packet
            count = 0
        end
        else
            count = -1
        end
    end
end

when queue becomes empty
    q_time = time
end

Saved Variables:
       avg : average queue size
       q_time : start of the queue idle time
       count : packets since last marked packet
Fixed parameters:
       wq : queue weight
       minth : minimum threshold for queue
       maxth : maximum threshold for queue
       maxp : maximum value for pb
Other:
       pa : current packet-marking probability
       q : current queue size
       time : current time
       f(t) : a linear function of the time t
```

Figure 2.6.1

Thus the RED gateway has two separate algorithms. The algorithm for computing the average queue size determines the degree of burstiness that will be allowed in the gateway queue. The algorithm for calculating the packet-marking probability determines how frequently the gateway marks packets, given the current level of congestion. The goal is for the gateway to mark packets at fairly evenly-spaced intervals, in order to avoid biases and to avoid global synchronization, and to mark packets sufficiently frequently to control the average queue size.

Figure 7 RED dropping probabilities

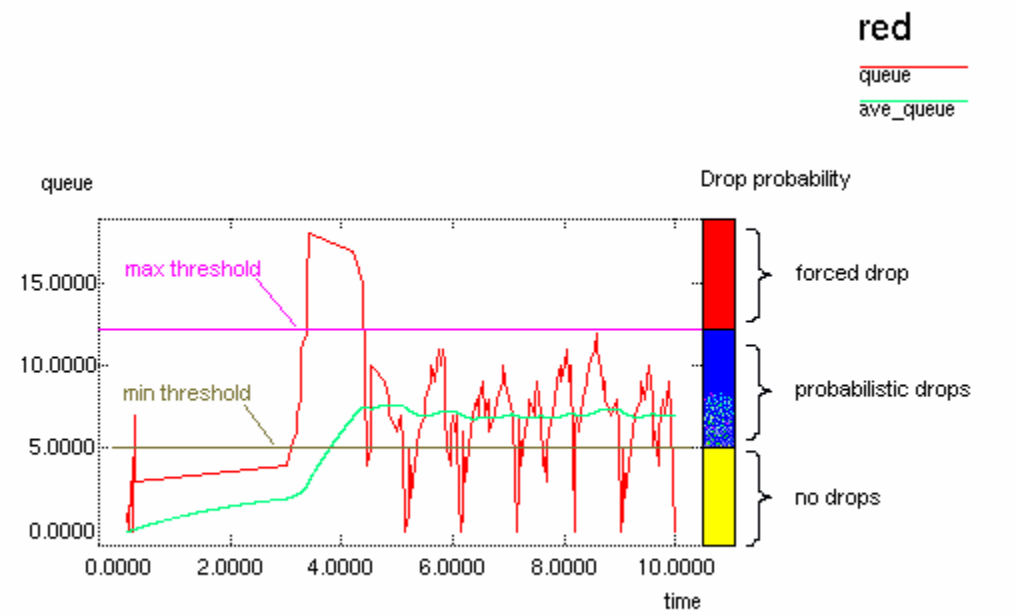


Figure 2.6.2

RED's packet dropping decisions are mode-based[9]. Figure illustrates the ideas behind the RED algorithm. This figure shows the instantaneous (red color) and weighted average (green color) queue size (in packets) over time. The current mode, indicated on the right hand side, is determined by the relation between the average queue size and the minimum and maximum thresholds. When the average queue size is less than the minimum threshold this indicates no congestion, so no action is taken. This is no drop mode (yellow band on the figure) and the probability that an arriving packet will be dropped is zero. In this mode arriving packets are always enqueued. At the other extreme, when the average queue size is greater than the maximum threshold, or if the queue is full, the algorithm infers persistent and severe congestion. All arriving packets are dropped. The probability an arriving packet will be dropped is one. This mode is referred to as forced drop mode (red band on the figure). Finally, when the average queue size is between the two thresholds the algorithm operates in a probabilistic (i.e., random)

drop mode. In this mode, arriving packets are dropped at random. The probability that a given packet will be dropped ranges between zero and one as a function of three parameters: maxp, the current average queue size avg, and count. The input parameter, maxp, determines the maximum probability that two consecutive packets will be dropped while in probabilistic drop mode. The variable, count, tracks how many packets have been enqueued since the last packet was dropped.

2.6 QoS Metrics

QoS metrics are base parameters of quality for a network. In order for a network to be able to provide a specific QoS it depends upon the inherent properties of the network itself [2] which span over all the elements in the network. For the transmission link QoS parameters include link delay, throughput, loss rate, error rate etc, whilst for the nodes the parameters include hardware capability like processing speed and memory size.

Throughput, delay , jitter and loss are amongst the most essential QoS metrics for multimedia applications.

The metrics are defined in detail below:

- Throughput - In communication networks, this is the amount of digital data per time unit that is delivered to a certain terminal in a network, from a network node, or from one node to another, for example via a communication link. The throughput is usually measured in bit per second (bit/s or bps).

- . Delay – this is in two parts i.e.

    - End-to-end delay - refers to the time taken for a packet to be transmitted across a network from source to destination i.e the time between the arrival of a packet and its successful delivery to the receiver.

    - Access delay - the time between packet arrival and packet transmission by the sender.

- Jitter - variation of delay and in electronics and telecommunications, it is an abrupt and unwanted variation of one or more signal characteristics, such as the interval between

successive pulses, the amplitude of successive cycles, or the frequency or phase of successive cycles. Jitter is a significant factor in multimedia applications.

- ❖ Bandwidth - the measure of data transmission rates/capacity when communicating over certain media or devices and it influences throughput.

2.7 OVERVIEW OF EXISTING QOS FRAMEWORKS

2.6.1 INSIGNIA

INSIGNIA[10] is one of the noteworthy QoS frameworks for MANETs. The main goal of INSIGNIA is to provide adaptive QoS guarantees for real-time traffic. It adopts in-band signaling to piggyback control information into the IP header of traffic so that resource reservation and QoS treatment can be provided along the flow, without the need of a pre-established flow path. INSIGNIA considers two optional QoS levels: base QoS and enhanced QoS. Flow traffic carries MIN/MAX bandwidth requests in the packet headers. In each hop, the flow reserves bandwidth to meet MIN/MAX request. At a bottleneck, a hop where only MIN or best effort QoS can be supported, all the hops preceding the bottleneck will adjust their reservation to no more than the bottleneck's QoS. Finally, base QoS or enhanced QoS traffic will be sent by the sender upon receiving a QoS report from the receiver indicating the total bandwidth reserved along the path. Useful as it is, INSIGNIA however is still not flexible enough and does not cover all aspects of the QoS problems in MANETs

- First, the QoS framework should be able to support any level of throughput within the range of [Min, Max] bandwidth requirements. For instance, multimedia applications support fine-grained throughput levels with adaptive codecs that can be dynamically configured at run-time. Real-time data streams, such as stock prices, can be displayed at any speed by adjusting the sampling rate according to the available throughput of network. This is difficult to do with INSIGNIA's two-level QoS support.
- Second, excess reservation should be avoided or minimized. In a MANET, invalid and unused reservation on nodes runs the risk of being propagated across the network as nodes move. To collect these extra reservations as soon as possible is critical

to maintain adequate throughput levels. Most QoS frameworks focus on fast reservations but neglect the problem of excessive reservations.

- Also signaling and processing overhead should be reduced. Complex signaling systems consume energy, reduce the memory available to the application, and degrade the overall processing capacity.
- Last but most importantly INSIGNIA does not differentiate between traffic classes but considers only two optional flows: base and enhanced QoS.

2.7.2 SWAN (Seamless Wireless Asynchronous Transfer Mode (ATM) Network)

SWAN[11] is an alternative to INSIGNIA with improved scalability properties. SWAN is a stateless network scheme specifically designed for wireless ad hoc networks employing a best effort distributed MAC. Intermediate nodes do not keep any per-flow information and thus avoid complex signaling and state control mechanisms and make the system more simple and scalable. SWAN distinguishes between two traffic classes: real time and best effort. When best effort packets arrive at a node, they enter a leaky bucket traffic shaper that has a previously calculated rate, derived from an AIMD (Additive Increase Multiplicative Decrease) rate control algorithm. Every node measures the MAC delays continuously and this information is used as feedback for the rate controller. Every T seconds, each device increases its transmission rate gradually until the packet delays at the MAC layer become excessive. As soon as the rate controller detects excessive delays, it reduces the rate of the shaper with a decrement rate. Rate control restricts the bandwidth of best effort traffic so that real time applications can use the required bandwidth. For the real time traffic, SWAN uses sender based admission control. This mechanism works by sending an end-to-end request/response probe along the constructed route to estimate the bandwidth availability at each node and then determines whether a new real time session should be admitted or not.

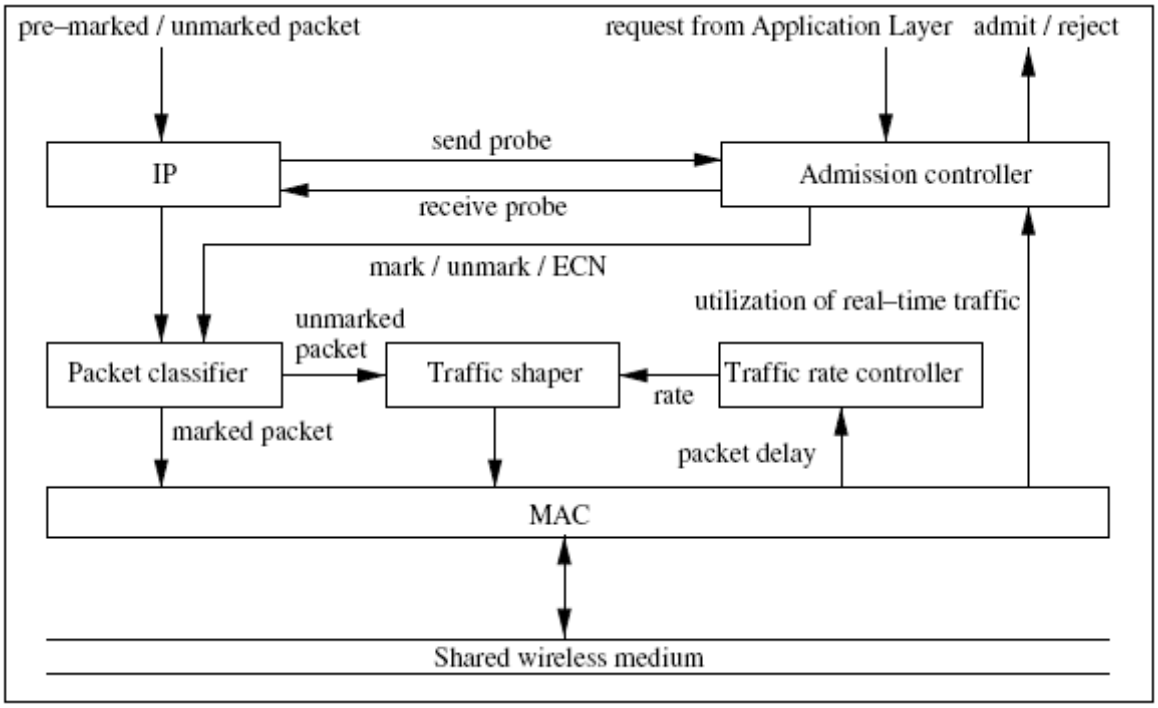


Figure 8 SWAN Architecture

Upon receiving a packet from the IP layer, the packet classifier module checks whether it is marked (i.e., real-time packet) or not (i.e., best-effort packet). If it is a best-effort packet, it is forwarded to the traffic shaper for regulation. If it is a real-time packet, the module directly forwards it to the MAC layer bypassing the traffic shaper. The traffic shaper represents a simple leaky bucket traffic policy. The traffic shaper delays best-effort packets in conformance with the rate calculated by the traffic rate controller. The call admission controller module is responsible for admitting or rejecting new real time sessions. The decision on whether to admit or reject a real-time session is taken solely by the source node based on the result of an end-to-end request/response probe.

in this scheme congested or overbooked nodes randomly select a congestion set of real-time sessions and only mark packets associated with that set. A congested node marks the congested set for a time period of T seconds and then calculates a new congested set. Hence some intelligence is required at the intermediate nodes,the nodes stop marking packets as congested when the measured rate of real-time traffic reaches below the admission control rate.

Advantages and disadvantages

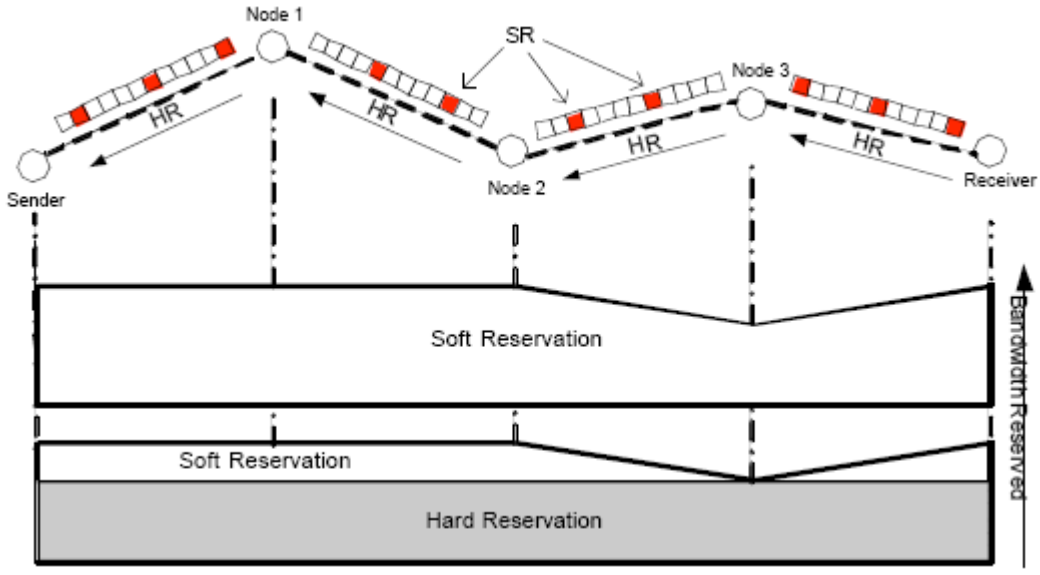
SWAN gives a framework for supporting real time applications by assuming a best-effort MAC protocol and not making any resource reservation .It uses feedback based control mechanisms to regulate real-time traffic at the time of congestion in the network.

As best-effort traffic serves as a buffer zone for real-time traffic, this model does not work well in scenarios where most of the traffic is real time in nature. Even though this model is scalable (because the intermediate nodes do not maintain any per flow or aggregate state information), it cannot provide hard QoS guarantees due to lack of resource reservation at the intermediate nodes. An admitted real-time flow may encounter periodic violations in its bandwidth requirements. In the worst case, it may have to be dropped or be made to live with downgraded best-effort service. Hence, the local rate control of best-effort traffic mechanism alone may not be sufficient to fully support real-time traffic.

Most importantly SWAN does not prioritize traffic classes so cannot be used to efficiently transmit Multimedia traffic.

ASAP (Adaptive ReServation And Pre-allocation Protocol)

Figure 9 ASAP



ASAP is an adaptive reservation QoS protocol. By adopting a simple signaling system and a two-phase reservation mechanism, ASAP provides adaptive QoS support, fast flow path setup and local repairing, as well as processing optimization A reservation in ASAP occurs in two phases: soft reservation and hard reservation. When a bandwidth request arrives, the soft reservation checks the available bandwidth and reserves the resource for future use. The soft reserved resource (SR) can be temporarily used by other traffic (both best-effort and QoS traffic) but cannot be reserved by other real-time flows. After the soft reservation, a hard reservation procedure cleans up the soft reservation by forcing out the traffic which temporarily occupies the soft-reserved resources and admits the real-time flow that made the reservation ASAP defines two signaling messages: *SR* and *HR*. SRs are periodically inserted in the flow to setup and repair paths, and collect QoS information. Whenever there are changes in a QoS availability through the flow path, an HR message is sent in the opposite direction of the flow to update the reservation state and inform the sender of the QoS adaptation The structure of the two messages is similar: *request indicator* (QoS to be reserved) and *reservation indicator* (QoS reserved). SR contains four fields: *MinBW*, *MaxBW*, *SoftBW* and *HardBW*. MinBW and MaxBW indicate the minimum and maximum bandwidth to be reserved. SoftBW and HardBW show how much bandwidth has been soft-reserved and hard-reserved on the flow path. HRo nly contains three fields: *SetBW*, *SoftBW* and *HardBW*. SetBW shows the amount of bandwidth the host should hard reserve. SoftBW and HardBW have the same meaning as in an SR message.
SR messages are in-band signaling. Senders of real-time flows periodically insert SRs into the IP option header of the traffic packets sent to the receiver. The QoS monitoring and reservation is done while SR travels. The receiver then replies to the sender with a HR message (out-of-band) for QoS reporting and adaptation. The reason of combining soft and hard reservation is to avoid the waste of extra reservations in hosts other than the bottleneck, a problem that occurs in INSIGNIA .After a flow path is established, SR messages are periodically inserted into the traffic flow. As each SR message collects QoS information along the flow path, the receiver can keep track of the latest QoS situation and give the sender feedback using HR messages, so that real-time applications can adapt their QoS profile accordingly. Similarly, the sender can scale down the real-time flow transmission in case the flow path changes and less guaranteed bandwidth can be maintained. Intermediate hosts release extra reservations upon receiving the HR message and the sender adapts its flow speed accordingly. ASAP has a simple but efficient

path local repairing mechanism. When a host receives an SR that it has not seen before, and the SR's SoftBW and HardBW are not zero, it realizes that the flow is away from the original path, and it triggers local path reparation. The host will create a new flow entry in the flow table, and make a soft reservation according to MinBW/MaxBW in the SR. This procedure is the same as in the flow setup stage. Then, the host will switch its soft reservation to hard reservation by as much as it is indicated in the HardBW of the SR, and keep the rest as soft reservation.
After that, the SR is modified and relayed to the next hop. When SR arrives at the receiver, the broken path has been repaired.
The total of the QoS packets that violate Guaranteed QoS include two parts: packets treated as best effort and packets that are simply dropped. As the flow bandwidth is quite small comparing to the link throughput, the dropped QoS packets are within a small percentage regardless of mobility. But the amount of packets treated as best effort rather than dropped increases with increasing mobility. This is because the high movement speed leads to a high handover rate that makes the restoration of QoS levels a continuous effort.

WEAKNESSES OF ASAP

Although ASAP is a QoS framework that provides high service quality while minimizing wasted reservations and it does so by using a soft/hard reservation concept and a flexible signaling schema, it needs to be streamlined to be efficient by exploiting and adapting the MAC layer to improve QoS signaling.
ASAP also differentiates traffic using the criteria of traffic being either real time or best effort (elastic). Therefore ASAP cannot be used to efficiently transmit Multimedia traffic.

2.8 Network Simulator (NS-2)

Network Simulator 2 (NS-2) is a simulation tool originated from Lawrence Berkeley National Laboratory. It is targeted at networking research and based on discrete events simulations.

The simulator is written in C++ and it uses OTcl as a command and configuration interface. This means that OTcl scripts are used to set up simulation scenarios in the simulator. One great benefit of this is that there is no need to recompile the simulator between different simulations since you are able to set up topology, link bandwidth, traffic sources etc, from the OTcl scripts.

So, once you have implemented the basic functionality within the simulator (C++ code) you only have to change the OTcl scripts to run various simulations.

NS-2 provides substantial support for simulation of routing and multicast protocols over wired and wireless networks. NS-2 is a popular tool among the researchers in both the wired and wireless world has been used to obtain vast amount of simulation results used in various papers.

The idea of a discrete event scheduler is that actions may only be started as a result of an event. In NS-2 this is taken care of by a scheduler and a scheduling list. Events are inserted into scheduling list upon request, together with their expiration time. The scheduler is responsible to go through the list and perform the necessary actions.

Simulations in NS-2 can be logged to trace files, which include detailed information about received and transmitted packets and allow for post-run processing with some analysis tools. shows a simplified view of NS-2

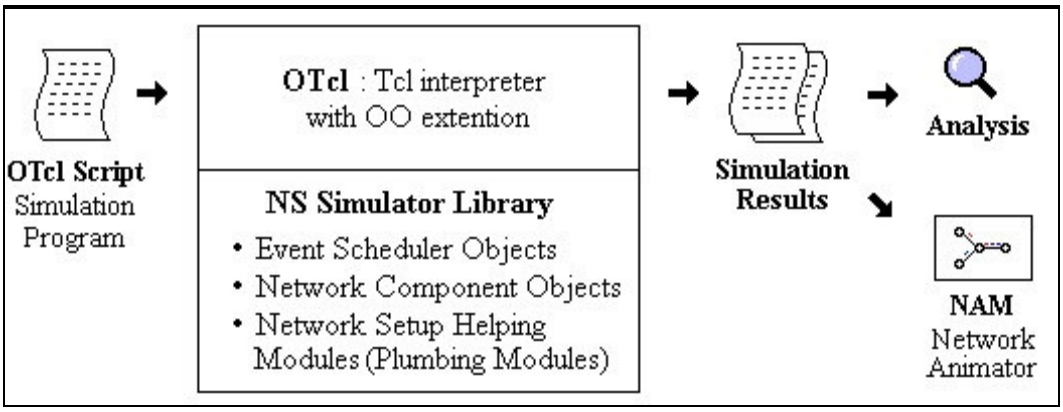


Figure 10 Simplified User's View of NS-2

2.9 CONCLUSION

QoS provisioning in MANETs is very challenging due to the network's highly time-variant characteristics. Based on existing work like the INSIGNIA protocol, ASAP has been

designed to further explore QoS provisioning mechanisms and improve existing frameworks. By adopting a Soft/Hard reservation mechanism, ASAP provides very flexible and adaptive QoS support on any QoS level, and maintains reservation efficiency (20-30% less bandwidth reserved comparing to protocols without Soft/Hard mechanism). ASAP has only two signaling messages and involves only simple decision-making. This feature makes ASAP's signaling system elegant and less process-intensive. Furthermore, the local repair mechanism in ASAP facilitates stable and continuous QoS provisioning in case of broken paths ASAP needs to be streamlined to be efficient by exploiting and adapting the MAC layer to improve QoS signaling.

All in all the current QoS frameworks are not ideal for efficiently transmitting Multimedia traffic especially due to their inherent differentiation of traffic into real time and best effort traffic because there is need to prioritize Multimedia traffic so that it can be granted superior QoS at the expense of all other traffic classes.

Based on the previous work especially on the problems inherent in service differentiation, and congestion control this research study seeks to come up with algorithms and codes to incorporate traffic differentiation, avoid congestion and assign different dropping probabilities to the different traffic classes and incorporate these into an existing QoS framework namely ASAP.

CHAPTER 3 :METHODOLOGY

In this chapter, we are discussing our approach to the problems described in the problem statement. In Section 3.1 we present the introduction. Section 3.2 describes the algorithms I modified and implemented.

3.1 1NTRODUCTION

After an extensive study of the existing QoS frameworks ,as stated in the last sections of the previous chapter the biggest shortcoming of the frameworks is their failure to differentiate traffic into separate classes either by the traffic content (data, voice, video) or any other criteria rather than by either being real time or elastic (best effort) traffic.

In this research study I will tackle traffic prioritisation ,enforcing different dropping probabilities per traffic class and also congestion avoidance. I will eventually try to incorporate these changes into the ASAP framework. I will then compare the proposed scheme to ASAP by making simulations with NS 2 simulator.

3.2 TRAFFIC DIFFERENTIATION

To differentiate traffic classes we made use of Differentiated Service Code Points (DSCP). Network routers can apply differentiated grades of service to packet streams based on a Diffserv Code Point (DSCP) marker in an IP packet header. DiffServ is concerned with classifying packets as they enter the local network. This classification then applies to a flow of traffic where a flow is defined by 5 elements:

- Source IP address
- Destination IP
- Source port
- Destination port
- Transport protocol.

A flow that has been classified or marked can be acted upon by other QoS mechanisms. Multiple flows can therefore be dealt with in a multitude of ways depending on the requirements of each flow,voice this way and data that way, etc DiffServ uses a Code Point to determine the per hop behavior of the flow.We made use of the Assured Forwarded where there are three drop probabilities. The DSCP determines the Per-Hop Behavior (PHB)[3] of a flow, packets are first classified according to their current DSCP. Then they are separated into queues where one queue may be routed via a marking mechanism and another queue may be examined more closely. After further examination additional packets may be sent for marking or be sent direct to the shaping/dropping mechanisms where all packets end up before leaving the interface. Assured Forwarding (AF): This is like riding in coach with extra leg room for business class. AF has four classes and three drop precedence within each class (so a total of twelve codepoints). Excess AF traffic is not delivered with as high probability as the traffic "within profile," which means it may be demoted but not necessarily dropped.

3.2.1 DSCP Precedence Purpose

0 0 Best effort

8 1 Class 1

16 2 Class 2

24 3 Class 3

32 4 Class 4

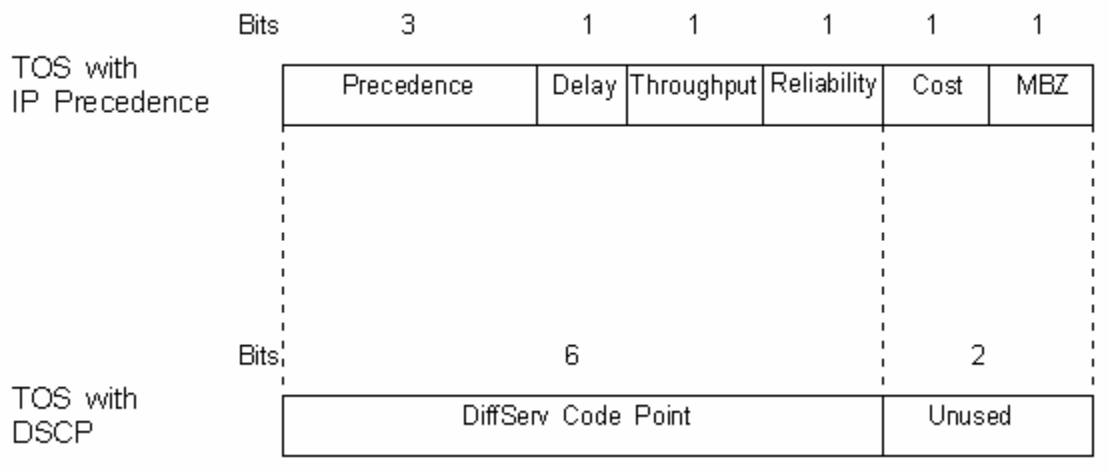
40 5 Express forwarding

48 6 Control

56 7 Control

DiffServ Code Point (DSCP) uses the first 6 bits of the TOS field thereby giving 2^6 = **64** different values

Figure 11 DiffServ Code Points



A flow, or flows, of packets marked with a particular DSCP in the DS field will be subject to a particular method of forwarding and rules as encapsulated in the **Behaviour Aggregate**. This Aggregate has three elements to it (or three colours) which determine whether the router interface 1) Drops the datagram, 2) Sends the datagram or 3) reclassifies it. For instance 5 flows can be treated as a Behaviour Aggregate so they are treated similarly as a group in most respects. Each flow is then distinguished by an additional **Drop Probability** and **Forwarding Behaviour**. Be aware that as the Drop Preference value increases, so the probability of being dropped increases!

The following table illustrates the DSCP values:
Figure 12 DSCP Values

Per Hop Behaviour (PHB)			**DiffServ Code Point (DSCP)**		**IP Precedence**
Default					0
		000000			
Assured Forwarding		**Low Drop Probability**	**Medium Drop Probability**	**High Drop Probability**	
	Class 1	AF11	AF12	AF13	1
		001010	**001**100	**001**110	
	Class 2	AF21	AF22	AF23	2
		010010	**010**100	**010**110	
	Class 3	AF31	AF32	AF33	3
		011010	**011**100	**011**110	
	Class 4	AF41	AF42	AF43	4
		100010	**100**100	**100**110	
Expedited Forwarding		EF			5
		101110			

The three Most Significant Bits (MSB) determine the Class and map directly to the IP Precedence bits. The **Class Selector Code Points** all have the form **xxx000**. The three Least Significant Bits (SLB) determine the Drop Probabilities and are ignored by IP Precedence-only

devices, the LSB is always '0'. A packet with a DSCP not mapped to one of the above PHB i.e. different from the recommendations, will have its DSCP mapped to the Default PHB of 000000.

3.2.2 APPROACH TAKEN

We divided the traffic into three different classes:

Class 1: Audio and video traffic

Class 2:Real time and Interactive data

Class 3: Elastic data (best effort)

We modified ip.cc so as to recognize the names of these classes as follows

```
static class IPHeaderClass : public PacketHeaderClass {
public:
        IPHeaderClass() : PacketHeaderClass("PacketHeader/IP",
                                  sizeof(hdr_ip)) {
            bind_offset(&hdr_ip::offset_);
      }
      void export_offsets() {
            field_offset("src_", OFFSET(hdr_ip, src_));
            field_offset("dst_", OFFSET(hdr_ip, dst_));
            field_offset("ttl_", OFFSET(hdr_ip, ttl_));
            field_offset("fid_", OFFSET(hdr_ip, fid_));
            field_offset("prio_", OFFSET(hdr_ip, prio_));
      }
} class_iphdr;

DSCPType determine_codepoint(const char*string)
{
      char charbuf[20];
      char* to = charbuf;
      while ((*to++ = tolower(*string++)))
            /* NOTHING */;
      *to = '\0';
      to=charbuf;

      if (strcmp(to, "be" ) == 0)
            return BE ;
      if (strcmp(to, "Voice_IP" ) == 0)
            return Class_1 ;
      if (strcmp(to, "Video_IP" ) == 0)
            return Class_1 ;
      if (strcmp(to, "IP_Radio" ) == 0)
            return Class_1 ;
      if (strcmp(to, "Real_time" ) == 0)
```

```
            return Class_2 ;
      if (strcmp(to, "Email" ) == 0)
            return Class_3 ;
            return INVALID_CP;
}
```

To add the code point aliases we included the code-point aliases statement at the [edit class-of-service] hierarchy level:
[edit class-of-service]

code-point-aliases {

dscp {

Class_1 110001;

Class_2 101110;

Class_3 110000;

be 000001;

}

}

The sample configuration produces this mapping:

```
user@host>show class-of-service code-point-aliases dscp
Code point type: dscp
Alias              Bit pattern
    ef/Class_2             101110
    af11                   001010
    af12                   001100
    af13                   001110
    af21                   010010
    af22                   010100
    af23                   010110
    af31                   011010
    af32                   011100
    af33                   011110
    af41                   100010
    af42                   100100
    af43                   100110
    be                     000001
    cs1                    001000
    cs2                    010000
    cs3                    011000
    cs4                    100000
    cs5                    101000
    Class_3                110000
    nc2                    111000
    Class_1                110001
```

Explanation:

- `Class_1 110001`:

- 110001 was not mapped to anything before, and `Class_1` is a new alias.
- Nothing in the default mapping table is changed by this statement.

- `Class_2 101110`:

- 101110 is now mapped to `Class_2` as well as `ef`.

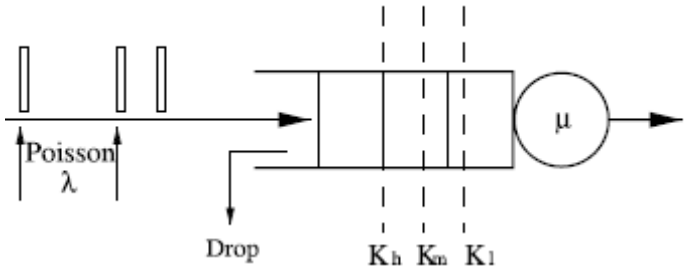
- `be 000001`:

- `be` is now mapped to 000001.
- The old value of `be`, 000000, is not associated with any alias. Packets with this DSCP value are now classified to the default forwarding class.

3.3 Congestion Control

To avoid congestion we made use of dsRed queue.

Figure 13 DSRed Queue



DSRED is based on dividing the buffer segment between *Kl* and *Kh* into two sub-segments separated at *Km*. The drop function from *Kl* to *Kh* is described by two linear segments with slopes α and β respectively.

The slopes of these two linear segments are complementary, and are adjusted by a mode selector, γ. Although, *Km* will be configured by the gateway administrator,

The drop function, *pd* (*a*vg), of DSRED can then be expressed as:

pd (*a*vg) =
0 *a*vg < *Kl*
α (*a*vg − *Kl*) *Kl* ≤ *a*vg < *Km*
1 − γ + β (*a*vg − *Km*) *Km* ≤ *a*vg < *Kh*
1 *Kh* ≤ *a*vg ≤ *N* (1)

where, α, β and *a*vg are given by:

$$\alpha = 2(1 - \gamma)/Kh - Kl \quad (2)$$

$$\beta = 2\gamma/(Kh - Kl) \quad (3)$$

$$a\text{vg} = (1 - \text{w})a\text{vg} + \text{w}q \quad (4)$$

The above equations, which govern packet drops in DSRED, translate to the following rules:

· No packet is dropped when the average queue length, *a*vg, is less than *Kl*;

· When the average queue length, *a*vg, is between *Kl* and *Km*, packets are dropped according to the drop function with slope α;

· When the average queue length, *a*vg, is between *Km* and *Kh*, packets are dropped according to the drop function with slope β;

· When the average queue length, *a*vg, is *Kh* or higher, packets are dropped with a probability of one.

Where:

N: Buffer size at the gateway in units of packets.

· *Kl:* Threshold (in packet) for average queue length to start active packet dropping at the buffer; this is the same as in [3]

· *Kh:* Threshold (in packet) for average queue length to start active packet dropping at the buffer with probability of one; this is the same as in [3]

· *Km:* Threshold (in packet) for average queue length to change the drop function slope;

· α*:* Slope of the drop function for the first linear segment between *Kl* and *Km*. Since the congestion is not severe in this segment, we need to drop as few packets as possible. α is, therefore, chosen to have a small value;

· β*:* Slope of the drop function for the second linear segment between *Km* and *Kh*. In this segment, the packet drop probability should be high to warn senders of congestion. Therefore, β should be larger than α;

· γ*:* Mode selector for adjusting the slopes of the drop functions;

· λ*:* Rate of traffic arrival at the buffer in packets/second;

· μ*:* Packet processing rate at the buffer in packets/second;

· ρ*:* Offered load factor defined as λμ ;

· *q:* Instantaneous queue length in packets;

· *a*vg*:* Average queue length

· w*:* Weight parameter as defined in [3] to calculate *a*vg;

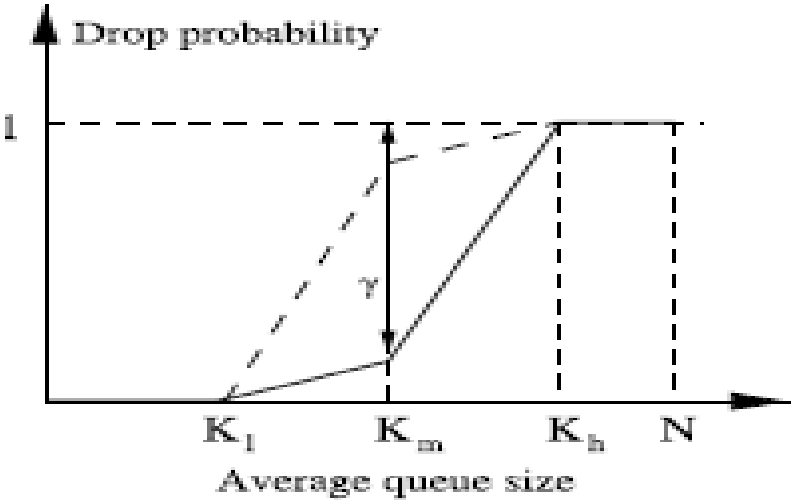
· *V*(*i*)*:* Steady state probability of *a*vg = *i*;

· *pa*(*i*)*:* Packet acceptance probability at State *i*;

· *pd*(*i*)*:* Packet dropping probability at State *i*; this is probability that the buffer drops an arriving packet when *a*vg = *i*. Note that *pd*(*i*) = 1 − *pa*(*i*);

· λ*i:* Effective arrival rate at the buffer at State *i*. This is defined as the rate at which packets are queued and is given by λ*pa*(*i*);

Figure 14 Varying drop probabilities



the buffer is modeled by a FIFO queue with a processing rate of μ. An arriving packet is dropped with a probability based on the average queue length. Therefore by implementing the dsRED queue the queue will never be congested since it will drop packets before it is full.

3.4 Varying Drop Probabilities

		Drop precedence		
		Low	Medium	High
Class	AF1	001010	001100	001110
	AF2	010010	010100	010110
	AF3	011010	011100	100100
	AF4	100010	100100	100110

To make the three classes have different drop probabilities, we modified the Time Slide Window 3 Color Marker (TSW3CM) Policy.The TSW3CM meters a traffic stream according to two

conditioning parameters i.e. Committed Information Rate (CIR) and the Peak Information Rate (PIR) , with CIR< PIR .

Committed Information Rate or CIR in a Frame relay network is the average bandwidth for a virtual circuit guaranteed by an ISP to work under normal conditions. At any given time, the bandwidth should not fall below this committed figure. The bandwidth is usually expressed in kilobits per second (kbit/s).

Above the CIR, an allowance of burstable bandwidth is often given, known as the Excess Information Rate (EIR) or Peak Information Rate (PIR). The provider guarantees that the connection will always support the CIR rate, and sometimes the EIR rate provided that there is adequate bandwidth. The CIR plus excess burst rate (EIR) is either equal to or less than the speed of the access port into the network. Frame relay carriers define and package CIRs differently, and CIRs are adjusted with experience.

CIR is derived from the term Committed Data Rate or CDR, and is used in similar fashion, but refers also to voice and non-data packets and not only to data packets as in CDR. The **Peak Information Rate** is a burstable rate set on routers and/or switches that allows throughput overhead. Related to Committed Information Rate which is a committed rate speed guaranteed/capped. For example, a CIR of 10 Mbit/s PIR of 12 Mbit/s allows you access to 10 Mbit/s minimum speed with burst/spike control that allows a throttle of an additional 2 Mbit/s. (This allows for data transmission to "settle" into a flow.)

The marker marks a packet as green, yellow, or red. The TSW3CM is composed of two parts: a rate estimator and a packet marker.The rate estimator provides an estimate of the stream’s arrival rate.The window length/size is set by the user and it corresponds to the maximum size of

Initially:

Win_length = a constant
Avg-rate =CIR
t-front = 0
Upon each packet arrival the rate estimator updates its variables:

Bytes-in-win = avg-rate * Win-length

New-bytes = Bytes-in-win + pkt-size

Avg-rate = New-byte/ (now –t-front + win-length)

t-front = now

Where:

Now = the time of the current packet arrival

Pkt-size = the packet size in bytes of the arriving packet

Avg-rate = measured arrival rate of the traffic stream

Win-length = the window over which history is kept

Let avg be the estimated average rate of the traffic stream, also Po = (avg-CIR)/ avg, and P1 = (avg-PIR), and P 2 = (PIR – CIR)/ avg = Po – P1

Therefore :

If avg < CIR the packet is marked green.

If CIR < avg < PIR the packet is marked yellow with probability Po and green with probability 1- Po.

If PIR < avg then

With probability P1 the packet is marked red

With probability P2 the packet is marked yellow, and

With probability 1- (P1 + P2) the packet is marked green

So choosing CIR relatively large compared to avg , then a large number of packets will go through as green. If PIR is chosen as relatively small compared to avg then there will be many red packets.

Approach

I modified the TSW3CM dropping probabilities in the dsPolicy.cc as follows:

```
void TSW3CMPolicy::applyMeter(policyTableEntry *policy, Packet *pkt) {
  double now, bytesInTSW, newBytes;
  hdr_cmn* hdr = hdr_cmn::access(pkt);

  bytesInTSW = policy->avgRate * policy->winLen;
  newBytes = bytesInTSW + (double) hdr->size();
  now = Scheduler::instance().clock();
  policy->avgRate = newBytes / (now - policy->arrivalTime + policy-
>winLen);
  policy->arrivalTime = now;
}
```

```
/*-------------------------------------------------------------------
---------
int TSW3CMPolicy::applyPolicer(policyTableEntry *policy, int
initialCodePt, Packet *pkt)
Pre: policy points to a policytableEntry that is using the TSW3CM
policer and
  whose state variables (avgRate, cir, and pir) are up to date.
Post: Sets code points with the following probabilities when rate >
PIR:

          red:    (rate - PIR) / rate
          yellow: (PIR - CIR) / rate
          green:  CIR / rate

and with the following code points when CIR < rate <= PIR:

          red:    0
          yellow: (rate - CIR) / rate
          green:  CIR / rate

    When rate is under CIR, a packet is always marked green.
Returns: A code point to apply to the current packet.
Uses: Methods downgradeOne() and downgradeTwo().
-------------------------------------------------------------------
-------*/
int TSW3CMPolicy::applyPolicer(policyTableEntry *policy,
policerTableEntry *policer, Packet *pkt) {
  double rand = policy->avgRate * (1.0 - Random::uniform(0.2, 0.8));

  if (rand > policy->pir)
    return (policer->downgrade2);
  else if (rand > policy->cir)
    return(policer->downgrade1);
  else
    return(policer->initialCodePt);
}
```

CHAPTER 4: RESULTS AND FINDINGS

4.1 INTRODUCTION

This chapter provides a description of the proposed traffic management scheme implementation, which was presented in Chapter 3. The Chapter includes an outline of the simulation framework together with the results of several test cases performed on the data.

4.2 Simulation Environment

In this Section we describe the environment and the scenarios used in our simulations.

First Scenario

The topology used in the first experiment consists of two stations transmitting to a fourth station via node 0 which is acting like a router.This node 0 is the buffer as it is implemented on top of the dsRED queue and will be the one dropping packets to avoid congestion.The uplink bandwidth is set at 2.112Mb whilst the downlink bandwidth is set at 38.016Mb. The queue limit was set at 1500 packets.TCP packets size was set to 1412 whilst UDP packets were set to 65536 whilst that of CBR was set to 1452 as shown below;

```
global opt
set opt(chan)           Channel/Sat
set opt(bw_up)                  2.112Mb       ;# uplink bandwidth
set opt(bw_down)       38.016Mb       ;# downlink bandwidth
set opt(mac)            Mac
set opt(ifq)           Queue/DropTail
set opt(qlim)              1500          ;# queue size (pkts)
set opt(ll)            LL/Sat
set opt(wiredRouting)   OFF
```

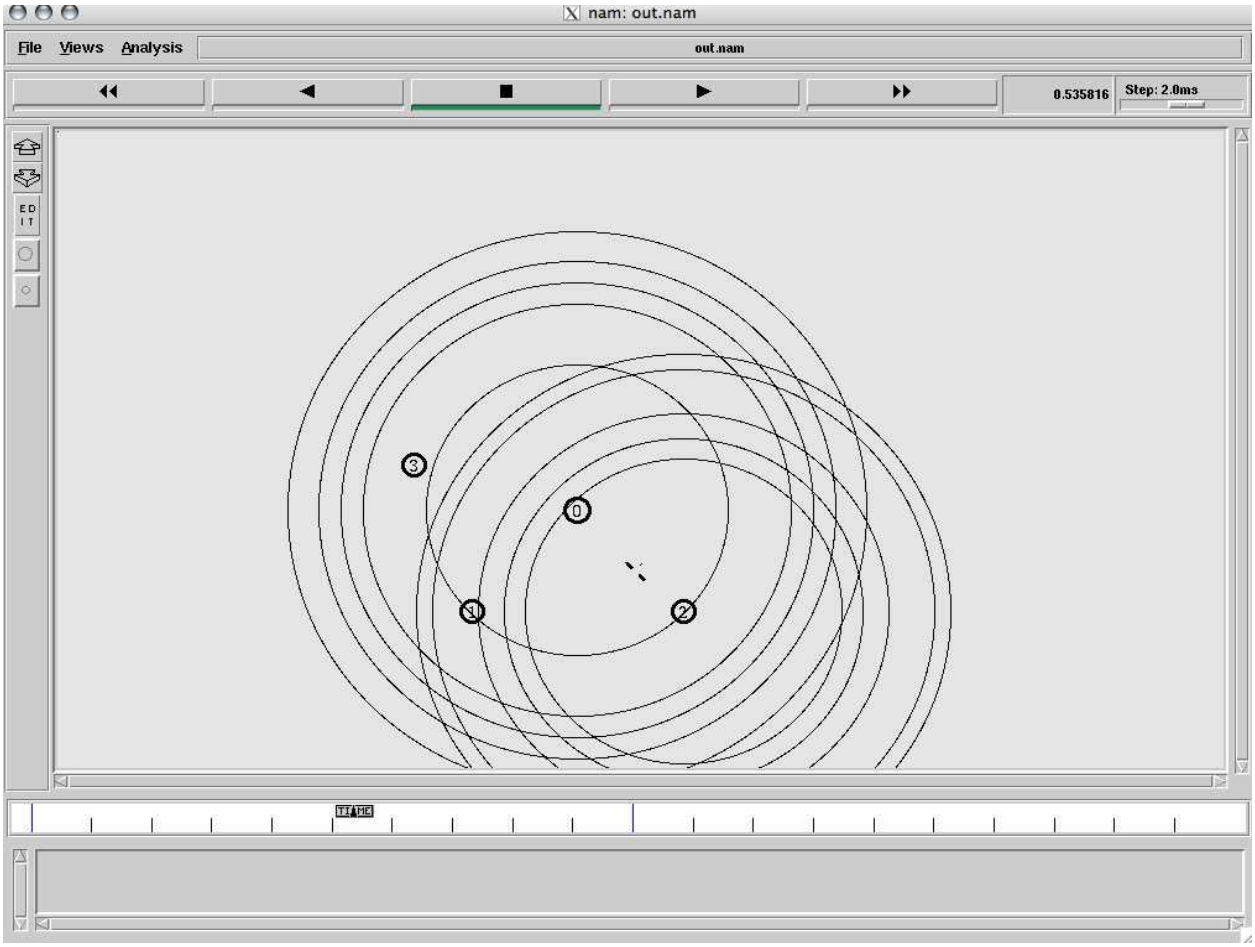
Agent/TCP set window_ 100000 ;# arbitrarily high adv. wind

Agent/TCP set packetSize_ 1412 ;# TCP payload (for full packet +92B)

Agent/UDP set packetSize_ 65536

Application/Traffic/CBR set packetSize_ 1452

Figure 15 Simulation topology

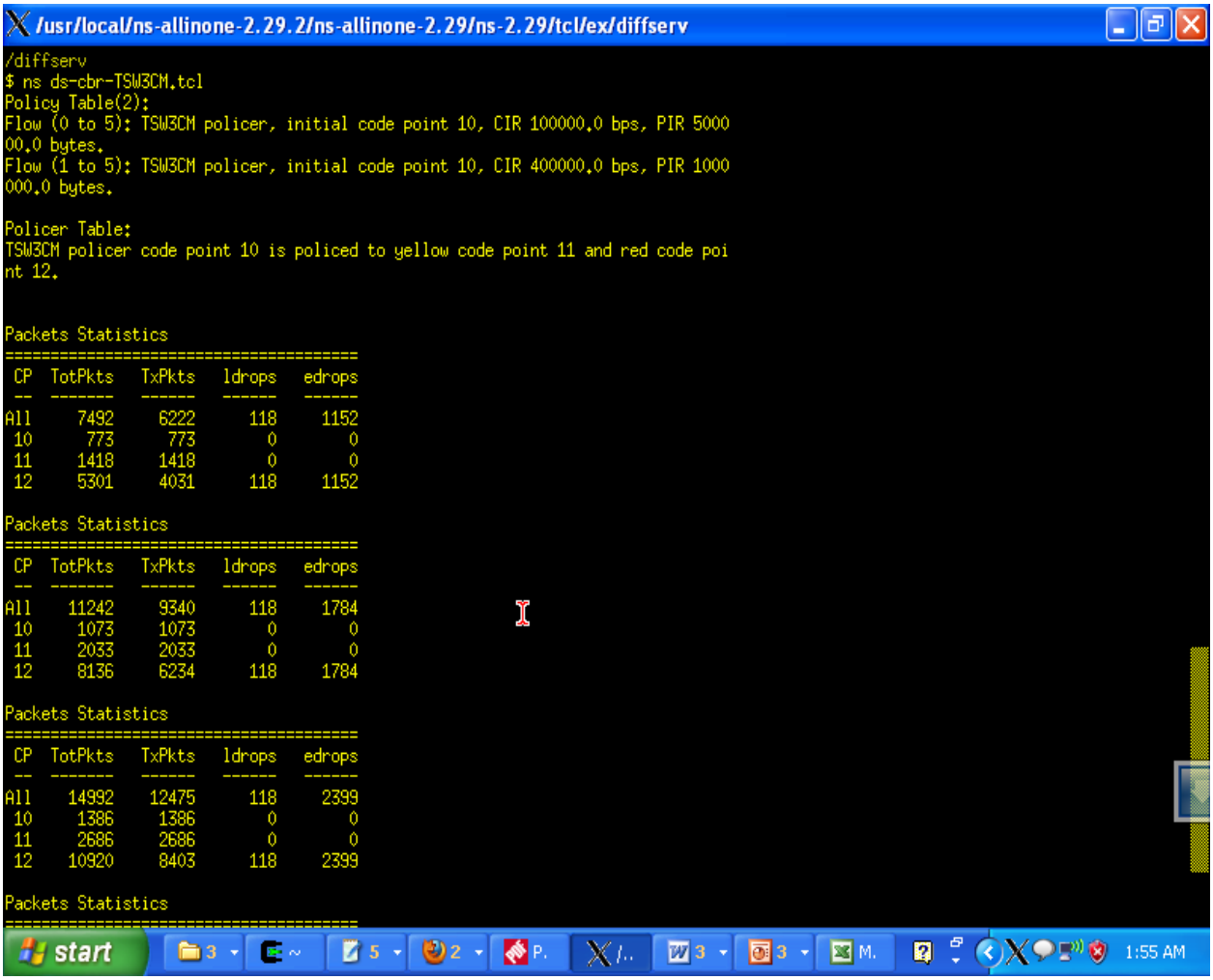


Screenshot of the simulation topology

4.3 Traffic prioritization

Using the DSCP codepoints we were able to classify the packets into three different classes with varying QoS.On running the simulation the screenshot shown below was captured which is a report on the packet statistics. It shows that there are three different traffic classes after a set time the number of packets sent for each class is printed and also the number of dropped packets is also printed. We also used trace files to get results of the simulation and plotted graphs using Microsoft Excel.

Figure 16 Traffic prioritization screenshot



```
+ 1.5524 3 4 cbr 1000 ------- 0 0.0 5.0 770 1156
- 1.5524 3 4 cbr 1000 ------- 0 1.0 5.1 385 1155
r 1.5526 3 4 cbr 1000 ------- 0 0.0 5.0 767 1151
+ 1.5526 4 5 cbr 1000 ------- 0 0.0 5.0 767 1151
- 1.5526 4 5 cbr 1000 ------- 0 0.0 5.0 767 1151
r 1.5536 4 5 cbr 1000 ------- 0 0.0 5.0 764 1147
r 1.5536 2 3 cbr 1000 ------- 0 0.0 5.0 771 1157
+ 1.5536 3 4 cbr 1000 ------- 0 0.0 5.0 771 1157
r 1.5538 1 2 cbr 1000 ------- 0 1.0 5.1 387 1161
+ 1.5538 2 3 cbr 1000 ------- 0 1.0 5.1 387 1161
- 1.5538 2 3 cbr 1000 ------- 0 1.0 5.1 387 1161
r 1.5538 0 2 cbr 1000 ------- 0 0.0 5.0 774 1162
+ 1.5538 2 3 cbr 1000 ------- 0 0.0 5.0 774 1162
+ 1.554 0 2 cbr 1000 ------- 0 0.0 5.0 777 1166
- 1.554 0 2 cbr 1000 ------- 0 0.0 5.0 777 1166
- 1.554 3 4 cbr 1000 ------- 0 0.0 5.0 770 1156
r 1.5542 3 4 cbr 1000 ------- 0 1.0 5.1 384 1152
+ 1.5542 4 5 cbr 1000 ------- 0 1.0 5.1 384 1152
- 1.5542 4 5 cbr 1000 ------- 0 1.0 5.1 384 1152
- 1.5546 2 3 cbr 1000 ------- 0 0.0 5.0 774 1162
r 1.5552 4 5 cbr 1000 ------- 0 0.0 5.0 765 1148
r 1.5556 2 3 cbr 1000 ------- 0 1.0 5.1 386 1158
+ 1.5556 3 4 cbr 1000 ------- 0 1.0 5.1 386 1158
- 1.5556 3 4 cbr 1000 ------- 0 0.0 5.0 771 1157
r 1.5558 0 2 cbr 1000 ------- 0 0.0 5.0 775 1163
+ 1.5558 2 3 cbr 1000 ------- 0 0.0 5.0 775 1163
- 1.5558 2 3 cbr 1000 ------- 0 0.0 5.0 775 1163
r 1.5558 3 4 cbr 1000 ------- 0 0.0 5.0 768 1153
+ 1.5558 4 5 cbr 1000 ------- 0 0.0 5.0
```

At the early stages only a few packets from the lowest class are dropped as there would not be much congestion and no packets are dropped from the other classes. As more packets continue to be sent more packets are dropped from class 3 and some packets from class 2 will start to be dropped too. Finally as we get closer to congestion some packets from class 3 will also be dropped but at the lowest rate for the three classes as depicted in the graph.

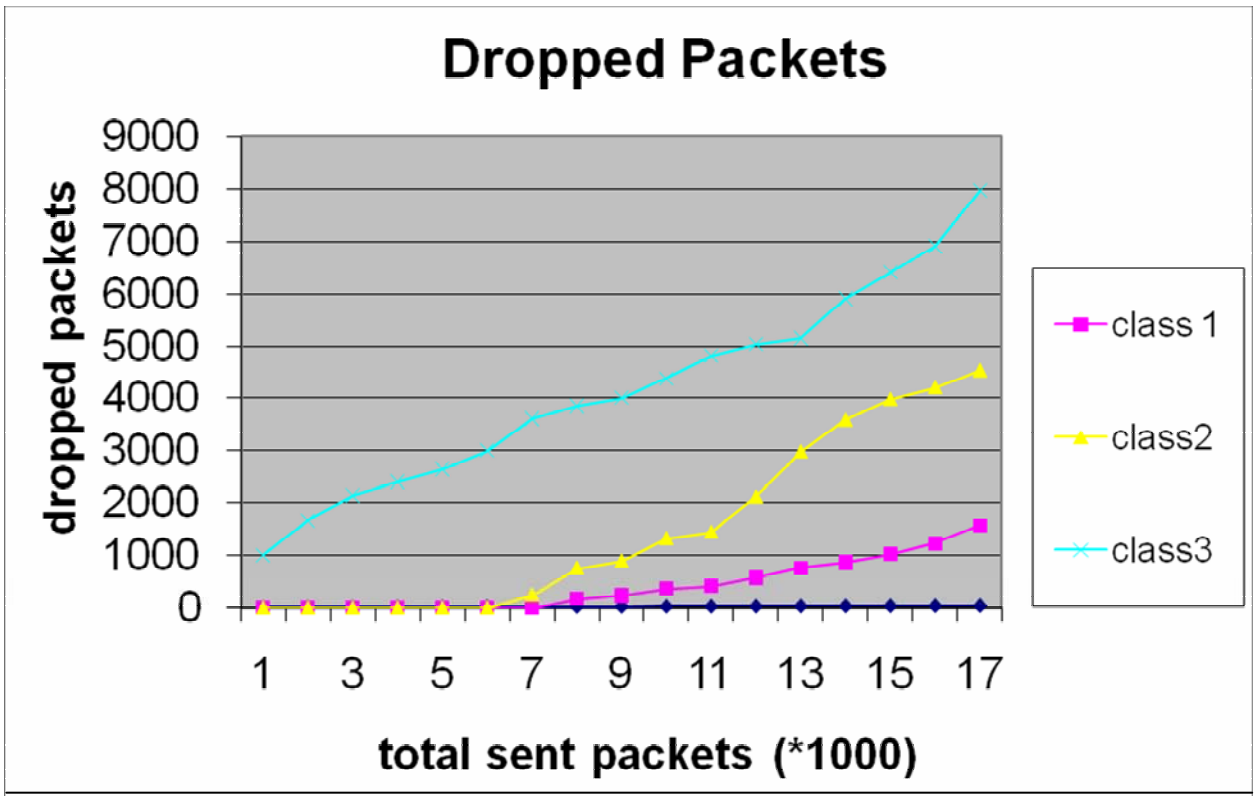


Figure 17 Dropped packets

Thus we were able to implement traffic classification.

The three classes receive different QoS as shown by following graph which is shows results of the throughput for each class.

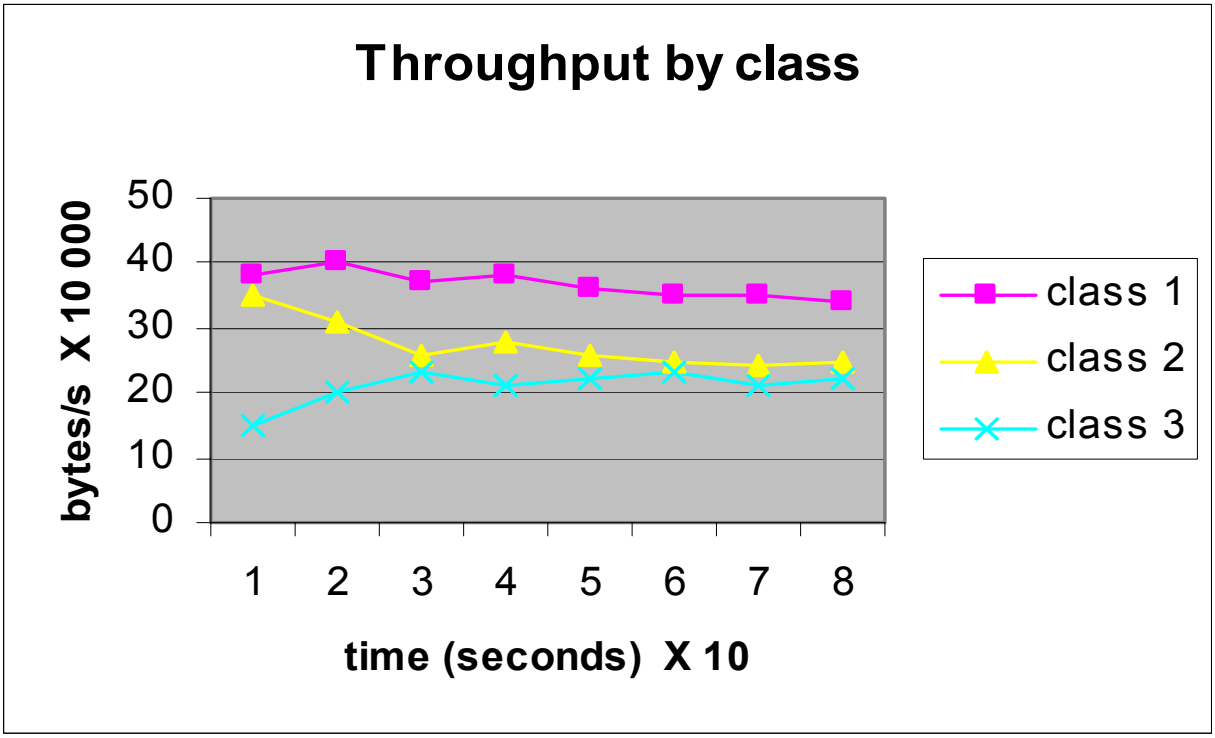


Figure 18 Throughput by class

So the different classes receive different QoS with class 1 having the best throughput followed by class 2 then lastly class 3. As more packets are introduced the throughput for all classes will slightly drop because of the increase in the number of packets dropping, but at the end the throughput will become fairly constant since the dsRED queue will avoid congestion.

4.4 Congestion Avoidance

From the previous graph it is shown that as time passes the throughput decreases but ends up a bit constant. This is because as more packets are introduced into the channel more packets will be dropped so as to avoid congestion. When the throughput starts getting a bit constant it is when the channel is near congestion that is when the buffer limit has been reached. So there will never be a situation where there is congestion since the dsRED queue will discard the packets before congestion is reached.

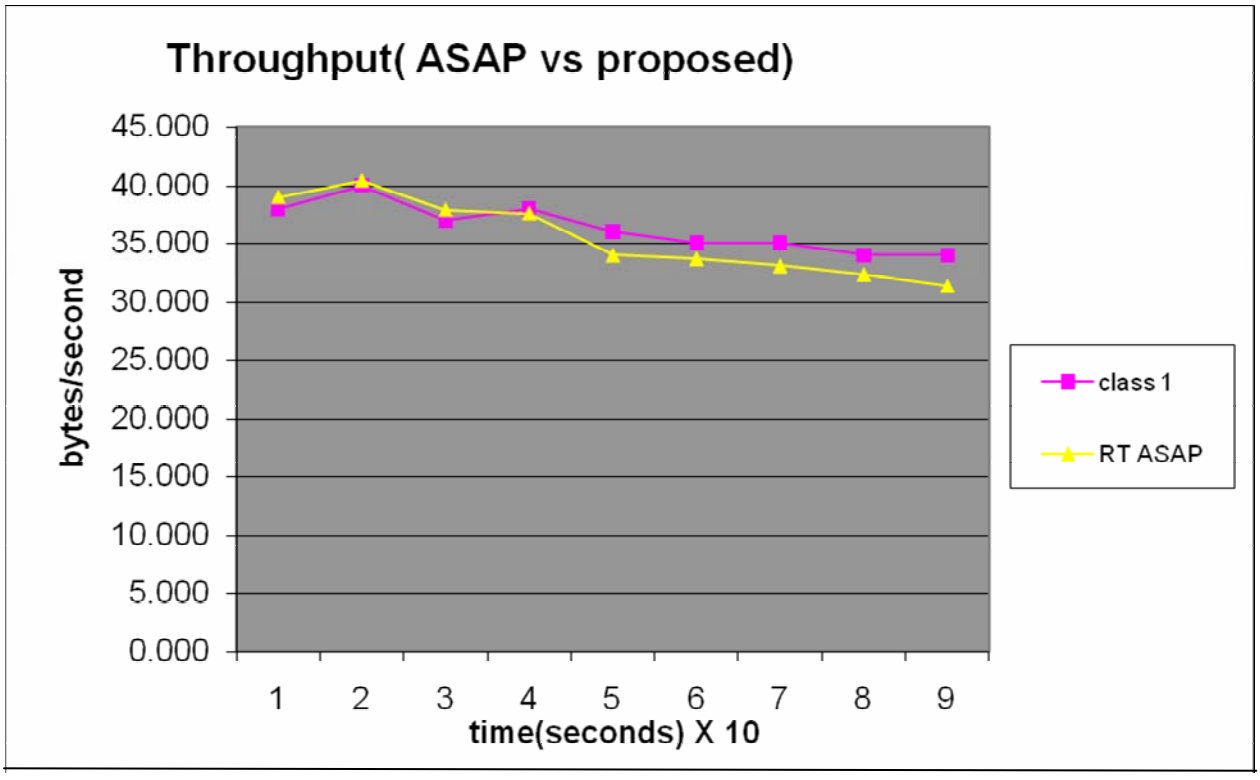
4.5 Varying dropping probabilities

Through the use of the DS code points and the modified TSW3CM[3] policy we were able to assign different dropping probabilities for each class. This is shown in the previous two tables which clearly show that class 1 has the lowest dropping probability followed by class 2 then lastly class 3.

4.6 Comparison with existing QoS frameworks

From chapter two it was clear that ASAP was the best QoS framework, so simulations were made to compare the performance of the proposed framework with ASAP.The simulation parameters were as described earlier in this chapter. In the first case we were comparing the throughput for class 1 of the proposed scheme with real time class for ASAP since it is real time traffic which receives superior service.

Figure 19 comparison of throughput



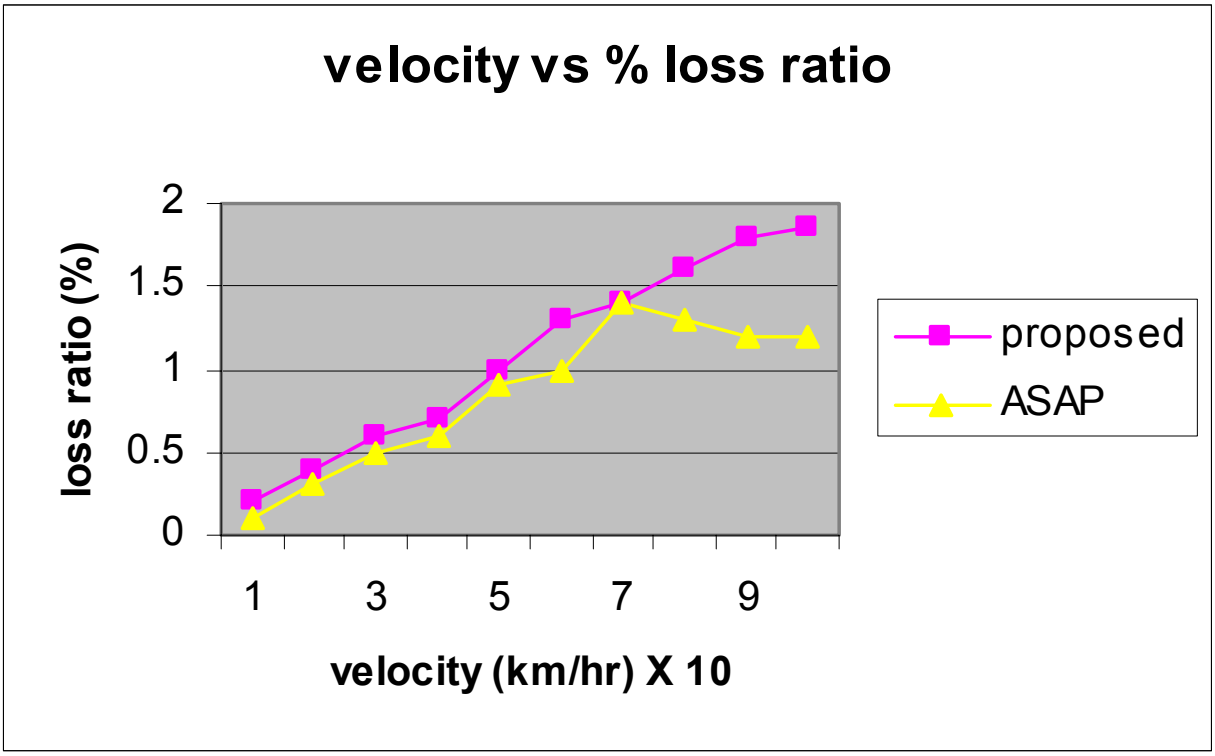
Initially ASAP had better throughput than proposed because of the reservation method used by ASAP. As more packets are introduced more packets from ASAP real time class are dropped

compared to class 1 of the proposed scheme. So eventually the throughput of the proposed scheme will be higher than that of ASAP. This is due to the fact that the dsRED queue avoids congestion and also because of the very low dropping probability set for class 1 of the proposed framework, whilst for ASAP once a flow is admitted there is not an efficient way to avoid congestion and on congestion many packets from the real time class will be dropped compared to class 1 of the proposed scheme.

Velocity vs. Packet loss ratio

This simulation shows the performance in different mobility circumstances: from a speed of 10km/h (human movement) to110km/h (car or train movement). We used 8 nodes randomly moving within a 600m*600m area with the specified maximum speed. 6 flows with 16kbps bandwidth are generated by 6 different sender nodes. Each flow has a randomly selected receiver node. Background traffic is randomly generated on 6 randomly selected nodes. The sending interval is 1 second. The radio coverage range is 250 meters. As ASAP works regardless of MANET routing protocols, AODV is used to for this simulation.. As the flow bandwidth is quite small comparing to the link throughput, the dropped QoS packets are within a small percentage regardless of mobility. But the amount of packets treated as best effort rather than dropped increases with increasing mobility. This is because the high movement speed leads to a high handover rate that makes the restoration of QoS levels a continuous effort.

Figure 20 velocity vs % loss ratio



From the results it was shown that at speeds above 70km/hr ASAP dropped more packets than the proposed scheme though the proposed scheme dropped more packets than ASAP when the speed was below 70km/hr. Overally both schemes performed well in this regard as even at speeds around 110km/hr the dropped packets for both schemes was less than 2%.

Conclusion

The proposed scheme works comparatively well and has other features like traffic classification and varying drop probabilities which are not in ASAP or current QoS frameworks. The main shortcoming was that we were not able to integrate the proposed scheme into ASAP or any other established QoS framework so that we could compare the modified QoS with established ones.

CHAPTER 5 CONCLUSION AND FUTURE WORK

In this thesis we designed and modified algorithms for traffic classification and traffic prioritization, and implemented DSRED to avoid congestion.We also modified the TSW3CM policy so as to implement different dropping probabilities for each class, with the first class having the lowest dropping probability and the third class with the highest dropping probability.

The proposed scheme performed comparatively well with an established QoS scheme namely ASAP and even outperformed it in certain regards. The proposed scheme dropped fewer packets than ASAP at high speeds of above 70km/hr and was also able to classify traffic into the following three classes;

Class 1: Audio and video traffic
Class 2:Real time and Interactive data
Class 3: Elastic data (best effort)
These classes have different dropping probabilities which depend on the CIR and PIR at any given moment.
The proposed scheme was also able to avoid congestion by dropping packets randomly with the probability of that class when we were about to have congestion.
The proposed scheme is extremely useful for the transportation of Multimedia traffic since it can provide superior QoS for multimedia traffic at the expense of other traffic classes. Since this research study mainly aimed at improving the transmission of multimedia traffic as it is the type of traffic which is getting more popular by the second and also it is the type of traffic that needs very stringent measures to transmit successfully.
Our main shortcoming was not being able to integrate these algorithms into an established QoS framework especially ASAP especially due to time constraints.

5.2 FUTURE WORK

It is vital that these algorithms be incorporated or integrated into an existing QoS framework especially ASAP so that the full effect of the modifications can be noted and also comparisons with other existing QoS frameworks can be done.

REFFERENCES

[1] Corson, S., and Macker, "Mobile Ad-hoc Networks: Performance Issues and Evaluation Considerations", IETF RFC-2501, January 1999

[2] S. Chakrabarti, "Qos issues in ad hoc wireless networks, "IEEE Communications Magazine, pp. 142-148, 2001.

[3] Differentiated Services (DiffServ) IETF Working Group.

[4] ASAP Adaptive QoS Support with Reduced Reservation Overhead in MANETs Patrick Stuedi, Jianbo Xue, Gustavo Alonso

[5] Different Service Codepoints IETF Working Group.

[6] Voice Activation Delay (VAD): IETF Working Group.

[7] Weighted Fair Queue(WFQ): IETF Working Group.

[8] CISCO IOS: CISCO IOS Journal 2002-4.

[9] "Red Early Discard (RED) Queuing." K Shamatra.

[10] INSIGNIA: An IP-Based Quality of Service Framework for Mobile ad Hoc Networks

Seoung-Bum Lee, Gahng-Seop Ahn, Xiaowei Zhang, and Andrew T. Campbell

[11] SWAN: Service Differentiation in Stateless Wireless Ad Hoc Networks Gahng-Seop Ahn, Andrew T. Campbell,

[12] Network Simulator 2 (NS-2) Homepage. <URL:http://www.isi.edu/nsnam/ns/

APPENDICES

APPENDICE A: MODIFIED TSW3CM policy

```
#include "dsPolicy.h"
#include "ew.h"
#include "dewp.h"
#include "packet.h"
#include "tcp.h"
#include "random.h"

// The definition of class PolicyClassifier.
//Constructor.
PolicyClassifier::PolicyClassifier() {
  int i;

  policyTableSize = 0;
  policerTableSize = 0;

  for (i = 0; i < MAX_POLICIES; i++)
    policy_pool[i] = NULL;
}

/*-------------------------------------------------------------------
---------
void addPolicyEntry()
    Adds an entry to policyTable according to the arguments in argv.
A source
and destination node ID must be specified in argv, followed by a
policy type
and policy-specific parameters.  Supported policies and their
parameters
are:

Null          InitialCodePoint
TSW2CM        InitialCodePoint  CIR
TSW3CM        InitialCodePoint  CIR  PIR
TokenBucket   InitialCodePoint  CIR  CBS
srTCM         InitialCodePoint  CIR  CBS  EBS
trTCM         InitialCodePoint  CIR  CBS  PIR  PBS
nyarota       initialcodepoint  CIR   PIR
    No error-checking is performed on the parameters.  CIR and PIR
should be
specified in bits per second; CBS, EBS, and PBS should be specified in
bytes.

    If the Policy Table is full, this method prints an error message.
-------------------------------------------------------------------
-------*/
void PolicyClassifier::addPolicyEntry(int argc, const char*const*
argv) {
```

```
  if (policyTableSize == MAX_POLICIES)
    printf("ERROR: Policy Table size limit exceeded.\n");
  else {
    policyTable[policyTableSize].sourceNode = atoi(argv[2]);
    policyTable[policyTableSize].destNode = atoi(argv[3]);
    policyTable[policyTableSize].codePt = atoi(argv[5]);
    policyTable[policyTableSize].arrivalTime = 0;
    policyTable[policyTableSize].winLen = 1.0;

    if ((strcmp(argv[4], "Dumb") == 0) || (strcmp(argv[4],"Null") ==
0)) {
      if(!policy_pool[Null])
     policy_pool[Null] = new NullPolicy;
      policyTable[policyTableSize].policy_index = Null;
      policyTable[policyTableSize].policer = nullPolicer;
      policyTable[policyTableSize].meter = nullMeter;
    } else if (strcmp(argv[4], "TSW2CM") == 0) {
      if(!policy_pool[TSW2CM])
     policy_pool[TSW2CM] = new TSW2CMPolicy;
      policyTable[policyTableSize].policy_index = TSW2CM;
      policyTable[policyTableSize].policer = TSW2CMPolicer;
      policyTable[policyTableSize].meter = tswTagger;

      policyTable[policyTableSize].cir =
     policyTable[policyTableSize].avgRate = (double) atof(argv[6]) /
8.0;
      if (argc == 8) policyTable[policyTableSize].winLen = (double)
atof(argv[7]);/* mb */
    } else if (strcmp(argv[4], "TSW3CM") == 0) {
      if(!policy_pool[TSW3CM])
     policy_pool[TSW3CM] = new TSW3CMPolicy;
      policyTable[policyTableSize].policy_index = TSW3CM;
      policyTable[policyTableSize].policer = TSW3CMPolicer;
      policyTable[policyTableSize].meter = tswTagger;

      policyTable[policyTableSize].cir =
     policyTable[policyTableSize].avgRate = (double) atof(argv[6]) /
8.0;
      policyTable[policyTableSize].pir = (double) atof(argv[7]) / 8.0;
    } else if (strcmp(argv[4], "TokenBucket") == 0) {
      if(!policy_pool[TB])
     policy_pool[TB] = (Policy *) new TBPolicy;
      policyTable[policyTableSize].policy_index = TB;
      policyTable[policyTableSize].policer = tokenBucketPolicer;
      policyTable[policyTableSize].meter = tokenBucketMeter;

      policyTable[policyTableSize].cir =
     policyTable[policyTableSize].avgRate = (double) atof(argv[6]) /
8.0;
      policyTable[policyTableSize].cbs =
     policyTable[policyTableSize].cBucket = (double) atof(argv[7]);
    } else if (strcmp(argv[4], "srTCM") == 0) {
```

```
      if(!policy_pool[SRTCM])
      policy_pool[SRTCM] = new SRTCMPolicy;
      policyTable[policyTableSize].policy_index = SRTCM;
      policyTable[policyTableSize].policer = srTCMPolicer;
      policyTable[policyTableSize].meter = srTCMMeter;

      policyTable[policyTableSize].cir =
      policyTable[policyTableSize].avgRate = (double) atof(argv[6]) /
8.0;
      policyTable[policyTableSize].cbs =
      policyTable[policyTableSize].cBucket = (double) atof(argv[7]);
      policyTable[policyTableSize].ebs =
      policyTable[policyTableSize].eBucket = (double) atof(argv[8]);
    } else if (strcmp(argv[4], "trTCM") == 0) {
      if(!policy_pool[TRTCM])
      policy_pool[TRTCM] = new TRTCMPolicy;
      policyTable[policyTableSize].policy_index = TRTCM;
      policyTable[policyTableSize].policer = trTCMPolicer;
      policyTable[policyTableSize].meter = trTCMMeter;

      policyTable[policyTableSize].cir =
      policyTable[policyTableSize].avgRate = (double) atof(argv[6]) /
8.0;
      policyTable[policyTableSize].cbs =
      policyTable[policyTableSize].cBucket = (double) atof(argv[7]);
      policyTable[policyTableSize].pir = (double) atof(argv[8]) / 8.0;
      policyTable[policyTableSize].pbs =
      policyTable[policyTableSize].pBucket = (double) atof(argv[9]);
    } else if (strcmp(argv[4], "nyarota") == 0) {
      if(!policy_pool[nyarota])
      policy_pool[nyarota] = new nyarotaPolicy;
      policyTable[policyTableSize].policy_index = nyarota;
      policyTable[policyTableSize].policer = nyarotaPolicer;
      policyTable[policyTableSize].meter = nyarotaMeter;

      policyTable[policyTableSize].cir =
      policyTable[policyTableSize].avgRate = (double) atof(argv[6]) /
8.0;
      policyTable[policyTableSize].cbs =
      policyTable[policyTableSize].cBucket = (double) atof(argv[7]);
      policyTable[policyTableSize].ebs =
      policyTable[policyTableSize].eBucket = (double) atof(argv[8]);
    } else if (strcmp(argv[4], "SFD") == 0) {
      if(!policy_pool[SFD])
      policy_pool[SFD] = new SFDPolicy;
      policyTable[policyTableSize].policy_index = SFD;
      policyTable[policyTableSize].policer = SFDPolicer;
      policyTable[policyTableSize].meter = sfdTagger;

      // Use cir as the transmission size threshold for the moment.
      policyTable[policyTableSize].cir = atoi(argv[6]);
    } else if (strcmp(argv[4], "EW") == 0) {
```

```
      if(!policy_pool[EW])
      policy_pool[EW] = new EWPolicy();

      ((EWPolicy *)policy_pool[EW])->
      init(atoi(argv[6]), atoi(argv[7]), atoi(argv[8]));

      policyTable[policyTableSize].policy_index = EW;
      policyTable[policyTableSize].policer = EWPolicer;
      policyTable[policyTableSize].meter = ewTagger;
  } else if (strcmp(argv[4], "DEWP") == 0) {
    if(!policy_pool[DEWP])
      policy_pool[DEWP] = new DEWPPolicy;

    ((DEWPPolicy *)policy_pool[DEWP])->
      init(atof(argv[6]));

    policyTable[policyTableSize].policy_index = DEWP;
    policyTable[policyTableSize].policer = DEWPPolicer;
    policyTable[policyTableSize].meter = dewpTagger;
  } else {
      printf("No applicable policy specified, exit!!!\n");
      exit(-1);
    }
    policyTableSize++;
  }
}

/*-------------------------------------------------------------------
---------
policyTableEntry* PolicyClassifier::getPolicyTableEntry(long source,
long dest)
Pre: policyTable holds exactly one entry for the specified source-dest
pair.
Post: Finds the policyTable array that matches the specified source-
dest pair.
Returns: On success, returns a pointer to the corresponding
policyTableEntry;
  on failure, returns NULL.
Note: the source-destination pair could be one-any or any-any (xuanc)
-------------------------------------------------------------------
-------*/
policyTableEntry* PolicyClassifier::getPolicyTableEntry(nsaddr_t
source, nsaddr_t dest) {
  for (int i = 0; i <= policyTableSize; i++) {
    if ((policyTable[i].sourceNode == source) ||
(policyTable[i].sourceNode == ANY_HOST)) {
      if ((policyTable[i].destNode == dest) ||
(policyTable[i].destNode == ANY_HOST))
     return(&policyTable[i]);
    }
  }
```

```
  // !!! Could make a default code point for undefined flows:
  printf("ERROR: No Policy Table entry found for Source %d-Destination
%d.\n", source, dest);
  printPolicyTable();
  return(NULL);
}

/*-----------------------------------------------------------------------
---------
void addPolicerEntry(int argc, const char*const* argv)
Pre: argv contains a valid command line for adding a policer entry.
Post: Adds an entry to policerTable according to the arguments in
argv.  No
  error-checking is done on the arguments.  A policer type should be
specified,
  consisting of one of the names {Null, TSW2CM, TSW3CM, TokenBucket,
  srTCM, trTCM}, followed by an initial code point.  Next should be an
  out-of-profile code point for policers with two-rate markers; or a
yellow and
  a red code point for policers with three drop precedences.
      If policerTable is full, an error message is printed.
-----------------------------------------------------------------------
-------*/
void PolicyClassifier::addPolicerEntry(int argc, const char*const*
argv) {
  //int cur_policy;


  if (policerTableSize == MAX_CP)
    printf("ERROR: Policer Table size limit exceeded.\n");
  else {
    if ((strcmp(argv[2], "Dumb") == 0) || (strcmp(argv[2],"Null") ==
0)) {
      if(!policy_pool[Null])
     policy_pool[Null] = new NullPolicy;
      policerTable[policerTableSize].policer = nullPolicer;
      policerTable[policerTableSize].policy_index = Null;
    } else if (strcmp(argv[2], "TSW2CM") == 0) {
      if(!policy_pool[TSW2CM])
     policy_pool[TSW2CM] = new TSW2CMPolicy;
      policerTable[policerTableSize].policer = TSW2CMPolicer;
      policerTable[policerTableSize].policy_index = TSW2CM;
    } else if (strcmp(argv[2], "TSW3CM") == 0) {
      if(!policy_pool[TSW3CM])
     policy_pool[TSW3CM] = new TSW3CMPolicy;
      policerTable[policerTableSize].policer = TSW3CMPolicer;
      policerTable[policerTableSize].policy_index = TSW3CM;
    } else if (strcmp(argv[2], "TokenBucket") == 0) {
      if(!policy_pool[TB])
     policy_pool[TB] = new TBPolicy;
      policerTable[policerTableSize].policer = tokenBucketPolicer;
      policerTable[policerTableSize].policy_index = TB;
```

```
    } else if (strcmp(argv[2], "srTCM") == 0) {
      if(!policy_pool[SRTCM])
      policy_pool[SRTCM] = new SRTCMPolicy;
      policerTable[policerTableSize].policer = srTCMPolicer;
      policerTable[policerTableSize].policy_index = SRTCM;
    } else if (strcmp(argv[2], "nyarota") == 0) {
      if(!policy_pool[nyarota])
      policy_pool[nyarota] = new nyarotaPolicy;
      policerTable[policerTableSize].policer = nyarotaPolicer;
      policerTable[policerTableSize].policy_index = nyarota;
    } else if (strcmp(argv[2], "trTCM") == 0){
      if(!policy_pool[TRTCM])
      policy_pool[TRTCM] = new TRTCMPolicy;
      policerTable[policerTableSize].policer = trTCMPolicer;
      policerTable[policerTableSize].policy_index = TRTCM;
    } else if (strcmp(argv[2], "SFD") == 0) {
      if(!policy_pool[SFD])
      policy_pool[SFD] = new SFDPolicy;
      policerTable[policerTableSize].policer = SFDPolicer;
      policerTable[policerTableSize].policy_index = SFD;
    } else if (strcmp(argv[2], "EW") == 0) {
      if(!policy_pool[EW])
      policy_pool[EW] = new EWPolicy;
      policerTable[policerTableSize].policer = EWPolicer;
      policerTable[policerTableSize].policy_index = EW;
    } else if (strcmp(argv[2], "DEWP") == 0) {
      if(!policy_pool[DEWP])
      policy_pool[DEWP] = new DEWPPolicy;
      policerTable[policerTableSize].policer = DEWPPolicer;
      policerTable[policerTableSize].policy_index = DEWP;
    } else {
      printf("No applicable policer specified, exit!!!\n");
      exit(-1);
    }
  };

  policerTable[policerTableSize].initialCodePt = atoi(argv[3]);
  if (policerTable[policerTableSize].policer == nullPolicer)
      policerTable[policerTableSize].downgrade1 = atoi(argv[3]);
    else
      policerTable[policerTableSize].downgrade1 = atoi(argv[4]);
  if (argc == 6)
    policerTable[policerTableSize].downgrade2 = atoi(argv[5]);
  policerTableSize++;
}

// Return the entry of Policer table with policerType and
initCodePoint matched
policerTableEntry* PolicyClassifier::getPolicerTableEntry(int
policy_index, int oldCodePt) {
  for (int i = 0; i < policerTableSize; i++)
    if ((policerTable[i].policy_index == policy_index) &&
```

```
      (policerTable[i].initialCodePt == oldCodePt))
       return(&policerTable[i]);

  printf("ERROR: No Policer Table entry found for initial code point
%d.\n", oldCodePt);
  //printPolicerTable();
  return(NULL);
}

/*-----------------------------------------------------------------------
---------
int mark(Packet *pkt, double minRTT)
Pre: The source-destination pair taken from pkt matches a valid entry
in
  policyTable.
Post: pkt is marked with an appropriate code point.
-----------------------------------------------------------------------
-------*/
int PolicyClassifier::mark(Packet *pkt) {
  policyTableEntry *policy;
  policerTableEntry *policer;
  int policy_index;
  int codePt;
  hdr_ip* iph;
  int fid;

  iph = hdr_ip::access(pkt);
  fid = iph->flowid();
  policy = getPolicyTableEntry(iph->saddr(), iph->daddr());
  if (policy) {
    codePt = policy->codePt;
    policy_index = policy->policy_index;
    policer = getPolicerTableEntry(policy_index, codePt);

    // bug pointed by Jason Kenney <jason@linear.engmath.dal.ca>
    if (policy_pool[policy_index]) {
      policy_pool[policy_index]->applyMeter(policy, pkt);
      codePt = policy_pool[policy_index]->applyPolicer(policy,
policer, pkt);
    }
  } else {
    printf("The policy object doesn't exist, ERROR!!!\n");
    exit(-1);
  }

  iph->prio_ = codePt;
  return(codePt);
}

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

```
Pre: The command line specifies a source and destination node for
which an
  RTT-Aware policy exists and a current RTT value for that policy.
Post: The aggRTT field of the appropriate policy is updated to a
weighted
  average of the previous value and the new RTT value specified in the
command
  line.  If no matching policy is found, an error message is printed.
------------------------------------------------------------------------
-------*/
void PolicyClassifier::updatePolicyRTT(int argc, const char*const*
argv) {
  policyTableEntry *policy;

  policy = getPolicyTableEntry(atoi(argv[2]), atoi(argv[3]));
  if (policy == NULL)
    printf("ERROR: cannot update RTT; no existing policy found for
Source %d-Desination %d.\n",
        atoi(argv[2]), atoi(argv[3]));
  else {
    policy->winLen = (double) atof(argv[4]);
  }
}

/*-----------------------------------------------------------------------
---------
Pre: The command line specifies a source and destination node for
which a
  policy exists that uses a cBucket value.  That policy's cBucket
parameter is
  currently valid.
Post: The policy's cBucket value is found and returned.
Returns: The value cBucket on success; or -1 on an error.
------------------------------------------------------------------------
-------*/
double PolicyClassifier::getCBucket(const char*const* argv) {
  policyTableEntry *policy;

  policy = getPolicyTableEntry(atoi(argv[2]), atoi(argv[3]));
  if (policy == NULL) {
    printf("ERROR: cannot get bucket size; no existing policy found
for Source %d-Desination %d.\n",
        atoi(argv[2]), atoi(argv[3]));
    return(-1);
  }
  else {
    if ((policy->policer == tokenBucketPolicer) || (policy->policer ==
srTCMPolicer) || (policy->policer == trTCMPolicer))
      return(policy->cBucket);
    else {
```

```
      printf("ERROR: cannot get bucket size; the Source %d-Desination
%d Policy does not include a Committed Bucket.\n", atoi(argv[2]),
atoi(argv[3]));
      return(-1);
    }
  }
}

//    Prints the policyTable, one entry per line.
void PolicyClassifier::printPolicyTable() {
  printf("Policy Table(%d):\n",policyTableSize);
  for (int i = 0; i < policyTableSize; i++)
    {
      switch (policyTable[i].policer) {
      case nullPolicer:
      printf("Flow (%d to %d): Null policer, ",
               policyTable[i].sourceNode,policyTable[i].destNode);
      printf("initial code point %d\n", policyTable[i].codePt);
      break;
       case TSW2CMPolicer:
      printf("Flow (%d to %d): TSW2CM policer, ",
               policyTable[i].sourceNode,policyTable[i].destNode);
      printf("initial code point %d, CIR %.1f bps.\n",
               policyTable[i].codePt, policyTable[i].cir * 8);
      break;
       case TSW3CMPolicer:
      printf("Flow (%d to %d): TSW3CM policer, initial code ",
               policyTable[i].sourceNode, policyTable[i].destNode);
      printf("point %d, CIR %.1f bps, PIR %.1f bytes.\n",
               policyTable[i].codePt, policyTable[i].cir * 8,
               policyTable[i].pir * 8);
      break;
      case nyarotaPolicer:
      printf("Flow (%d to %d): nyarota policer, initial code ",
               policyTable[i].sourceNode, policyTable[i].destNode);
      printf("point %d, CIR %.1f bps, EBS %.1f bytes.\n",
               policyTable[i].codePt, policyTable[i].cir * 8,
               policyTable[i].ebs * 8);
      break;
       case tokenBucketPolicer:
      printf("Flow (%d to %d): Token Bucket policer, ",
               policyTable[i].sourceNode,policyTable[i].destNode);
      printf("initial code  point %d, CIR %.1f bps, CBS %.1f bytes.\n",
               policyTable[i].codePt, policyTable[i].cir * 8,
               policyTable[i].cbs);
      break;
       case srTCMPolicer:
      printf("Flow (%d to %d): srTCM policer, initial code ",
               policyTable[i].sourceNode, policyTable[i].destNode);
      printf("point %d, CIR %.1f bps, CBS %.1f bytes, EBS %.1f
bytes.\n",
               policyTable[i].codePt, policyTable[i].cir * 8,
```

```
                policyTable[i].cbs, policyTable[i].ebs);
      break;
       case trTCMPolicer:
      printf("Flow (%d to %d): trTCM policer, initial code ",
                policyTable[i].destNode, policyTable[i].sourceNode);
      printf("point %d, CIR %.1f bps, CBS %.1f bytes, PIR %.1f bps, ",
             policyTable[i].codePt, policyTable[i].cir * 8,
                policyTable[i].cbs, policyTable[i].pir * 8);
      printf("PBS %.1f bytes.\n", policyTable[i].pbs);
      break;
       case SFDPolicer:
      printf("Flow (%d to %d): SFD policer, ",
             policyTable[i].sourceNode,policyTable[i].destNode);
      printf("initial code point %d, TH %d bytes.\n",
             policyTable[i].codePt, (int)policyTable[i].cir);
      break;
       case EWPolicer:
      printf("Flow (%d to %d): EW policer, ",
             policyTable[i].sourceNode,policyTable[i].destNode);
      printf("initial code point %d.\n", policyTable[i].codePt);
      break;
       case DEWPPolicer:
      printf("Flow (%d to %d): DEWP policer, ",
             policyTable[i].sourceNode,policyTable[i].destNode);
      printf("initial code point %d.\n", policyTable[i].codePt);
      break;
       default:
      printf("ERROR: Unknown policer type in Policy Table.\n");
       }
    }
  printf("\n");
}

// Prints the policerTable, one entry per line.
void PolicyClassifier::printPolicerTable() {
  bool threeColor;

  printf("Policer Table:\n");
  for (int i = 0; i < policerTableSize; i++) {
    threeColor = false;
    switch (policerTable[i].policer) {
    case nullPolicer:
      printf("Null ");
      break;
    case TSW2CMPolicer:
      printf("TSW2CM ");
      break;
    case TSW3CMPolicer:
      printf("TSW3CM ");
      threeColor = true;
      break;
    case tokenBucketPolicer:
```

```
      printf("Token Bucket ");
      break;
    case srTCMPolicer:
      printf("srTCM ");
      threeColor = true;
      break;
       case nyarotaPolicer:
      printf("nyarota ");
      threeColor = true;
      break;
    case trTCMPolicer:
      printf("trTCM ");
      threeColor = true;
      break;
    case SFDPolicer:
      printf("SFD ");
      //printFlowTable();
      break;
    case EWPolicer:
      printf("EW ");
      break;
    case DEWPPolicer:
      printf("DEWP ");
      break;
    default:
      printf("ERROR: Unknown policer type in Policer Table.");
    }

    if (threeColor) {
      printf("policer code point %d is policed to yellow ",
          policerTable[i].initialCodePt);
      printf("code point %d and red code point %d.\n",
          policerTable[i].downgrade1,
          policerTable[i].downgrade2);
    } else
      printf("policer code point %d is policed to code point %d.\n",
          policerTable[i].initialCodePt,
          policerTable[i].downgrade1);
  }
  printf("\n");
}

// The beginning of the definition of the NullPolicy
// NullPolicy will do nothing, but is also a good example to show how
to add
// new policy.

/*-----------------------------------------------------------------
---------
void NullPolicy::applyMeter(policyTableEntry *policy, Packet *pkt)
Do nothing
```

```
-------------------------------------------------------------------------
-------*/
void NullPolicy::applyMeter(policyTableEntry *policy, Packet *pkt) {
  policy->arrivalTime = Scheduler::instance().clock();
}

/*-----------------------------------------------------------------------
---------
int DumbPolicy::applyPolicer(policyTableEntry *policy, int
initialCodePt, Packet *pkt)
Always return the initial codepoint.
-------------------------------------------------------------------------
-------*/
int NullPolicy::applyPolicer(policyTableEntry *policy,
policerTableEntry *policer, Packet *pkt) {
  return(policer->initialCodePt);
}

// The end of NullPolicy

// The beginning of the definition of TSW2CM
/*-----------------------------------------------------------------------
---------
void TSW2CMPolicy::applyMeter(policyTableEntry *policy, Packet *pkt)
Pre: policy's variables avgRate, arrivalTime, and winLen hold valid
values; and
  pkt points to a newly-arrived packet.
Post: Adjusts policy's TSW state variables avgRate and arrivalTime
(also called
  tFront) according to the specified packet.
Note: See the paper "Explicit Allocation of Best effor Delivery
Service" (David
  Clark and Wenjia Fang), Section 3.3, for a description of the TSW
Tagger.
-------------------------------------------------------------------------
-------*/
void TSW2CMPolicy::applyMeter(policyTableEntry *policy, Packet *pkt) {
  double now, bytesInTSW, newBytes;
  hdr_cmn* hdr = hdr_cmn::access(pkt);

  bytesInTSW = policy->avgRate * policy->winLen;
  newBytes = bytesInTSW + (double) hdr->size();
  now = Scheduler::instance().clock();
  policy->avgRate = newBytes / (now - policy->arrivalTime + policy-
>winLen);
  policy->arrivalTime = now;
}

/*-----------------------------------------------------------------------
---------
int TSW2CMPolicy::applyPolicer(policyTableEntry *policy, int
initialCodePt, Packet *pkt)
```

```
Pre: policy points to a policytableEntry that is using the TSW2CM
policer and
  whose state variables (avgRate and cir) are up to date.
Post: If policy's avgRate exceeds its CIR, this method returns an out-
of-profile
  code point with a probability of ((rate - CIR) / rate).  If it does
not
  downgrade the code point, this method simply returns the initial
code point.
Returns: A code point to apply to the current packet.
Uses: Method downgradeOne().
------------------------------------------------------------------------
-------*/
int TSW2CMPolicy::applyPolicer(policyTableEntry *policy,
policerTableEntry *policer, Packet *pkt) {
  if ((policy->avgRate > policy->cir)
      && (Random::uniform(0.0, 1.0) <= (1-(policy->cir/policy-
>avgRate)))) {
    return(policer->downgrade1);
  }
  else {
    return(policer->initialCodePt);
  }
}

// The end of TSW2CM

// The Beginning of TSW3CM
/*----------------------------------------------------------------------
---------
void TSW3CMPolicy::applyMeter(policyTableEntry *policy, Packet *pkt)
Pre: policy's variables avgRate, arrivalTime, and winLen hold valid
values; and
  pkt points to a newly-arrived packet.
Post: Adjusts policy's TSW state variables avgRate and arrivalTime
(also called
  tFront) according to the specified packet.
Note: See the paper "Explicit Allocation of Best effor Delivery
Service" (David
  Clark and Wenjia Fang), Section 3.3, for a description of the TSW
Tagger.
------------------------------------------------------------------------
-------*/
void TSW3CMPolicy::applyMeter(policyTableEntry *policy, Packet *pkt) {
  double now, bytesInTSW, newBytes;
  hdr_cmn* hdr = hdr_cmn::access(pkt);

  bytesInTSW = policy->avgRate * policy->winLen;
  newBytes = bytesInTSW + (double) hdr->size();
  now = Scheduler::instance().clock();
  policy->avgRate = newBytes / (now - policy->arrivalTime + policy-
>winLen);
```

```
  policy->arrivalTime = now;
}

/*-----------------------------------------------------------------
---------
int TSW3CMPolicy::applyPolicer(policyTableEntry *policy, int
initialCodePt, Packet *pkt)
Pre: policy points to a policytableEntry that is using the TSW3CM
policer and
  whose state variables (avgRate, cir, and pir) are up to date.
Post: Sets code points with the following probabilities when rate >
PIR:

          red:    (rate - PIR) / rate
          yellow: (PIR - CIR) / rate
          green:  CIR / rate

and with the following code points when CIR < rate <= PIR:

          red:    0
          yellow: (rate - CIR) / rate
          green:  CIR / rate

    When rate is under CIR, a packet is always marked green.
Returns: A code point to apply to the current packet.
Uses: Methods downgradeOne() and downgradeTwo().
-----------------------------------------------------------------
-------*/
int TSW3CMPolicy::applyPolicer(policyTableEntry *policy,
policerTableEntry *policer, Packet *pkt) {
  double rand = policy->avgRate * (1.0 - Random::uniform(0.2, 0.8));

  if (rand > policy->pir)
    return (policer->downgrade2);
  else if (rand > policy->cir)
    return(policer->downgrade1);
  else
    return(policer->initialCodePt);
}

// End of TSW3CM
// The Beginning of nyarota
/*-----------------------------------------------------------------
---------
void TSW3CMPolicy::applyMeter(policyTableEntry *policy, Packet *pkt)
Pre: policy's variables avgRate, arrivalTime, and winLen hold valid
values; and
  pkt points to a newly-arrived packet.
Post: Adjusts policy's TSW state variables avgRate and arrivalTime
(also called
  tFront) according to the specified packet.
```

```
Note: See the paper "Explicit Allocation of Best effor Delivery
Service" (David
  Clark and Wenjia Fang), Section 3.3, for a description of the TSW
Tagger.
-------------------------------------------------------------------
-------*/
void TSW3CMPolicy::applyMeter(policyTableEntry *policy, Packet *pkt) {
  double now, bytesInTSW, newBytes;
  hdr_cmn* hdr = hdr_cmn::access(pkt);

  bytesInTSW = policy->avgRate * policy->winLen;
  newBytes = bytesInTSW + (double) hdr->size();
  now = Scheduler::instance().clock();
  policy->avgRate = newBytes / (now - policy->arrivalTime + policy-
>winLen);
  policy->arrivalTime = now;
}

/*-------------------------------------------------------------------
---------
int TSW3CMPolicy::applyPolicer(policyTableEntry *policy, int
initialCodePt, Packet *pkt)
Pre: policy points to a policytableEntry that is using the TSW3CM
policer and
  whose state variables (avgRate, cir, and pir) are up to date.
Post: Sets code points with the following probabilities when rate >
PIR:

          red:    (rate - PIR) / rate
          yellow: (PIR - CIR) / rate
          green:  CIR / rate

and with the following code points when CIR < rate <= ebs:

          red:    0
          yellow: (rate - CIR) / rate
          green:  CIR / rate

    When rate is under CIR, a packet is always marked green.
Returns: A code point to apply to the current packet.
Uses: Methods downgradeOne() and downgradeTwo().
-------------------------------------------------------------------
-------*/
int TSW3CMPolicy::applyPolicer(policyTableEntry *policy,
policerTableEntry *policer, Packet *pkt) {
  double rand = policy->avgRate * (1.0 - Random::uniform(0.0, 1.0));

  if (rand > policy->ebs)
    return (policer->downgrade2);
  else if (rand > policy->cir)
    return(policer->downgrade1);
  else
```

```
    return(policer->initialCodePt);
}

// End of nyarota



// Begin of Token Bucket.
/*-------------------------------------------------------------------
---------
void TBPolicy::applyMeter(policyTableEntry *policy, Packet *pkt)
Pre: policy's variables cBucket, cir, cbs, and arrivalTime hold valid
values.
Post: Increments policy's Token Bucket state variable cBucket
according to the
  elapsed time since the last packet arrival.  cBucket is filled at a
rate equal  to CIR, capped at an upper bound of CBS.
  This method also sets arrivalTime equal to the current simulator
time.
-------------------------------------------------------------------
-------*/
void TBPolicy::applyMeter(policyTableEntry *policy, Packet *pkt) {
  double now = Scheduler::instance().clock();
  double tokenBytes;

  tokenBytes = (double) policy->cir * (now - policy->arrivalTime);
  if (policy->cBucket + tokenBytes <= policy->cbs)
   policy->cBucket += tokenBytes;
  else
   policy->cBucket = policy->cbs;
  policy->arrivalTime = now;
}

/*-------------------------------------------------------------------
--------
int TBPolicy::applyPolicer(policyTableEntry *policy, int
initialCodePt,
        Packet* pkt)
Pre: policy points to a policytableEntry that is using the Token
Bucket policer
and whose state variable (cBucket) is up to date.  pkt points to a
newly-arrived packet.
Post: If policy's cBucket is at least as large as pkt's size, cBucket
is
decremented by that size and the initial code point is retained.
Otherwise,
the code point is downgraded.
Returns: A code point to apply to the current packet.
Uses: Method downgradeOne().
-------------------------------------------------------------------
-------*/
```

```
int TBPolicy::applyPolicer(policyTableEntry *policy, policerTableEntry
*policer, Packet* pkt) {
  hdr_cmn* hdr = hdr_cmn::access(pkt);

  double size = (double) hdr->size();

  if ((policy->cBucket - size) >= 0) {
    policy->cBucket -= size;
    return(policer->initialCodePt);
  } else{
    return(policer->downgrade1);
  }
}

// End of Tocken Bucket.

// Begining of SRTCM
/*-------------------------------------------------------------------
---------
void SRTCMPolicy::applyMeter(policyTableEntry *policy)
Pre: policy's variables cBucket, eBucket, cir, cbs, ebs, and
arrivalTime hold
  valid values.
Post: Increments policy's srTCM state variables cBucket and eBucket
according
  to the elapsed time since the last packet arrival.  cBucket is
filled at a
  rate equal to CIR, capped at an upper bound of CBS.  When cBucket is
full
  (equal to CBS), eBucket is filled at a rate equal to CIR, capped at
an upper
  bound of EBS.
      This method also sets arrivalTime equal to the current
  simulator time.
Note: See the Internet Draft, "A Single Rate Three Color Marker"
(Heinanen et
  al; May, 1999) for a description of the srTCM.
-------------------------------------------------------------------
-------*/
void SRTCMPolicy::applyMeter(policyTableEntry *policy, Packet *pkt) {
  double now = Scheduler::instance().clock();
  double tokenBytes;

  tokenBytes = (double) policy->cir * (now - policy->arrivalTime);
  if (policy->cBucket + tokenBytes <= policy->cbs)
    policy->cBucket += tokenBytes;
  else {
    tokenBytes = tokenBytes - (policy->cbs - policy->cBucket);

    policy->cBucket = policy->cbs;
    if (policy->eBucket + tokenBytes <= policy->ebs)
      policy->eBucket += tokenBytes;
```

```
    else
      policy->eBucket = policy->ebs;
  }
  policy->arrivalTime = now;
}

/*-----------------------------------------------------------------
---------
int SRTCMPolicy::applyPolicer(policyTableEntry *policy, int
initialCodePt, Packet* pkt)
Pre: policy points to a policyTableEntry that is using the srTCM
policer and
  whose state variables (cBucket and eBucket) are up to date.  pkt
points to a
  newly-arrived packet.
Post: If policy's cBucket is at least as large as pkt's size, cBucket
is
  decremented by that size and the initial code point is retained.
Otherwise,
  if eBucket is at least as large as the packet, eBucket is
decremented and the
  yellow code point is returned.  Otherwise, the red code point is
returned.
Returns: A code point to apply to the current packet.
Uses: Method downgradeOne() and downgradeTwo().
Note: See the Internet Draft, "A Single Rate Three Color Marker"
(Heinanen et
  al; May, 1999) for a description of the srTCM.
-----------------------------------------------------------------
-------*/
int SRTCMPolicy::applyPolicer(policyTableEntry *policy,
policerTableEntry *policer, Packet* pkt) {

  hdr_cmn* hdr = hdr_cmn::access(pkt);
  double size = (double) hdr->size();

  if ((policy->cBucket - size) >= 0) {
    policy->cBucket -= size;
    return(policer->initialCodePt);
  } else {
    if ((policy->eBucket - size) >= 0) {
      policy->eBucket -= size;
      return(policer->downgrade1);
    } else
      return(policer->downgrade2);
  }
}

// End of SRTCM

// Beginning of TRTCM
```

```
/*-----------------------------------------------------------------------
--------
void TRTCMPolicy::applyMeter(policyTableEntry *policy, Packet *pkt)
Pre: policy's variables cBucket, pBucket, cir, pir, cbs, pbs, and
arrivalTime
  hold valid values.
Post: Increments policy's trTCM state variables cBucket and pBucket
according
  to the elapsed time since the last packet arrival.  cBucket is
filled at a
  rate equal to CIR, capped at an upper bound of CBS.  pBucket is
filled at a
  rate equal to PIR, capped at an upper bound of PBS.
      This method also sets arrivalTime equal to the current simulator
time.
Note: See the Internet Draft, "A Two Rate Three Color Marker"
(Heinanen et al;
  May, 1999) for a description of the srTCM.
-----------------------------------------------------------------------
-----*/
void TRTCMPolicy::applyMeter(policyTableEntry *policy, Packet *pkt) {
  double now = Scheduler::instance().clock();
  double tokenBytes;
  tokenBytes = (double) policy->cir * (now - policy->arrivalTime);
  if (policy->cBucket + tokenBytes <= policy->cbs)
    policy->cBucket += tokenBytes;
  else
    policy->cBucket = policy->cbs;

  tokenBytes = (double) policy->pir * (now - policy->arrivalTime);
  if (policy->pBucket + tokenBytes <= policy->pbs)
    policy->pBucket += tokenBytes;
  else
    policy->pBucket = policy->pbs;

  policy->arrivalTime = now;
}

/*-----------------------------------------------------------------------
--------
int TRTCMPolicy::applyPolicer(policyTableEntry *policy, int
initialCodePt, Packet* pkt)
Pre: policy points to a policyTableEntry that is using the trTCM
policer and
  whose state variables (cBucket and pBucket) are up to date.  pkt
points to a
  newly-arrived packet.
Post: If policy's pBucket is smaller than pkt's size, the red code
point is
  retained.  Otherwise, if cBucket is smaller than the packet size,
the yellow
```

```
  code point is returned and pBucket is decremented.  Otherwise, the
packet
  remains green and both buckets are decremented.
Returns: A code point to apply to the current packet.
Uses: Method downgradeOne() and downgradeTwo().
Note: See the Internet Draft, "A Two Rate Three Color Marker"
(Heinanen et al;
  May, 1999) for a description of the srTCM.
-------------------------------------------------------------------
-------*/
int TRTCMPolicy::applyPolicer(policyTableEntry *policy,
policerTableEntry *policer, Packet* pkt) {
  hdr_cmn* hdr = hdr_cmn::access(pkt);
  double size = (double) hdr->size();

  if ((policy->pBucket - size) < 0)
    return(policer->downgrade2);
  else {
    if ((policy->cBucket - size) < 0) {
      policy->pBucket -= size;
      return(policer->downgrade1);
    } else {
      policy->cBucket -= size;
      policy->pBucket -= size;
      return(policer->initialCodePt);
    }
  }
}
// End of TRTCM

// Beginning of SFD
//Constructor.
SFDPolicy::SFDPolicy() : Policy() {
  flow_table.head = NULL;
  flow_table.tail = NULL;
}

//Deconstructor.
SFDPolicy::~SFDPolicy(){
  struct flow_entry *p, *q;
  p = q = flow_table.head;
  while (p) {
    printf("free flow: %d\n", p->fid);
    q = p;
    p = p->next;
    free(q);
  }

  p = q = NULL;
  flow_table.head = flow_table.tail = NULL;
}
```

```
/*-------------------------------------------------------------------
---------
 void SFDPolicy::applyMeter(policyTableEntry *policy, Packet *pkt)
 Flow states are kept in a linked list.
 Record how many bytes has been sent per flow and check if there is
any flow
 timeout.
-------------------------------------------------------------------
-------*/
void SFDPolicy::applyMeter(policyTableEntry *policy, Packet *pkt) {
  int fid, src_id, dst_id;
  struct flow_entry *p, *q, *new_entry;

  double now = Scheduler::instance().clock();
  hdr_cmn* hdr = hdr_cmn::access(pkt);
  hdr_ip* iph = hdr_ip::access(pkt);
  fid = iph->flowid();
  dst_id = iph->daddr();
  src_id = iph->saddr();

  //  printf("enter applyMeter\n");
  //  printFlowTable();

  p = q = flow_table.head;
  while (p) {
    // Check if the flow has been recorded before.
    if (p->fid == fid) {
    //if (p->src_id == src_id && p->dst_id == dst_id ) {
      p->last_update = now;
      p->bytes_sent += hdr->size();
      return;
    } else if (p->last_update + FLOW_TIME_OUT < now){
      // The coresponding flow is expired.
      if (p == flow_table.head){
     if (p == flow_table.tail) {
       flow_table.head = flow_table.tail = NULL;
       free(p);
       p = q = NULL;
     } else {
       flow_table.head = p->next;
       free(p);
       p = q = flow_table.head;
     }
      } else {
     q->next = p->next;
     if (p == flow_table.tail)
       flow_table.tail = q;
     free(p);
     p = q->next;
      }
    } else {
      q = p;
```

```
      p = q->next;
    }
  }

  // This is the firt time the flow shown up
  if (!p) {
    new_entry = new flow_entry;
    new_entry->fid = fid;
    new_entry->src_id = src_id;
    new_entry->dst_id = dst_id;
    new_entry->last_update = now;
    new_entry->bytes_sent = hdr->size();
    new_entry->count = 0;
    new_entry->next = NULL;

    // always insert the new entry to the tail.
    if (flow_table.tail)
      flow_table.tail->next = new_entry;
    else
      flow_table.head = new_entry;
    flow_table.tail = new_entry;
  }

  //  printf("leave applyMeter\n");
  return;
}

/*-----------------------------------------------------------------------
---------
void SFDPolicy::applyPolicer(policyTableEntry *policy, int
initialCodePt, Packet *pkt)
    Prints the policyTable, one entry per line.
-----------------------------------------------------------------------
-------*/
int SFDPolicy::applyPolicer(policyTableEntry *policy,
policerTableEntry *policer, Packet *pkt) {
  int fid, src_id, dst_id;
  struct flow_entry *p;

  hdr_ip* iph = hdr_ip::access(pkt);
  fid = iph->flowid();
  dst_id = iph->daddr();
  src_id = iph->saddr();

  //  printf("enter applyPolicer\n");
  //printFlowTable();

  p = flow_table.head;
  while (p) {
    // Check if the flow has been recorded before.
    if (p->fid == iph->flowid()) {
      //if (p->src_id == src_id && p->dst_id == dst_id) {
```

```
      if (p->bytes_sent > policy->cir) {
     // Use downgrade2 code to judge how to penalize out-profile
packets.
     if (policer->downgrade2 == 0) {
       // Penalize every packet beyond th.
       //printf("leave applyPolicer  %d, every downgrade\n", p->fid);
       return(policer->downgrade1);
     } else if (policer->downgrade2 == 1) {
       // Randomized penalization.
       if (Random::uniform(0.0, 1.0) > (1 - (policy->cir/p-
>bytes_sent))) {
         //printf("leave applyPolicer %d, random initial.\n", p->fid);
         return(policer->initialCodePt);
       } else {
         //printf("leave applyPolicer %d, random, downgrade\n", p-
>fid);
         return(policer->downgrade1);
       }
     } else {
       // Simple scheduling on penalization.
       if (p->count == 5) {
         // Penalize 4 out of every 5 packets.
         p->count = 0;
         //printf("leave applyPolicer %d, initial, %d\n", p->fid, p-
>count);
         return(policer->initialCodePt);
       } else {
         p->count++;
         //printf("leave applyPolicer %d, downgrade, %d\n", p->fid, p-
>count);
         return(policer->downgrade1);
       }
     }
      } else {
     //   printf("leave applyPolicer, initial\n");
     return(policer->initialCodePt);
      }
    }
    p = p->next;
  }

  // Can't find the record for this flow.
  if (!p) {
    printf ("MISS: no flow %d (%d, %d) in the table\n", fid, src_id,
dst_id);
    printFlowTable();
};

  //  printf("leave applyPolicer, init but problem...\n");
  return(policer->initialCodePt);
}
```

```
//    Prints the flowTable, one entry per line.
void SFDPolicy::printFlowTable() {
  struct flow_entry *p;
  printf("Flow table:\n");

  p = flow_table.head;
  while (p) {
    printf("flow id: %d [%d %d], bytesSent: %d, last_update: %f\n",
        p->fid, p->src_id, p->dst_id, p->bytes_sent, p->last_update);
    p = p-> next;
  }
  p = NULL;
  printf("\n");
}
// End of SFD
```

APPENDICE B: TRAFFIC CLASSIFICATION

```
#ifndef ns_ip_h
#define ns_ip_h

#include "config.h"
#include "packet.h"


#define IP_HDR_LEN      20
#define IP_DEF_TTL      32
#define IP_BROADCAST  ((u_int32_t) 0xffffffff)

enum DSCPType {Voice_IP, Video_IP, IP_Radio, IP_TV, IP_GAME,
Video_Streaming,AUDIO_IP,INTERACTIVE,REAL TIME, BE, INVALID_CP};

DSCPType determine_codepoint (const char *string) ;

struct hdr_ip {
     /* common to IPv{4,6} */
     ns_addr_t  src_;
     ns_addr_t  dst_;
     int        ttl_;

     /* Monarch extn */
//   u_int16_t  sport_;
//   u_int16_t  dport_;

     /* IPv6 */
     int        fid_; /* flow id */
     int        prio_;

     /* diffserv */
     DSCPType   DSCP_;
```

```
      static int offset_;
      inline static int& offset() { return offset_; }
      inline static hdr_ip* access(Packet* p) {
            return (hdr_ip*) p->access(offset_);
      }

      /* per-field member acces functions */
      ns_addr_t& src() { return (src_); }
      nsaddr_t& saddr() { return (src_.addr_); }
        int32_t& sport() { return src_.port_;}

      ns_addr_t& dst() { return (dst_); }
      nsaddr_t& daddr() { return (dst_.addr_); }
        int32_t& dport() { return dst_.port_;}
      int& ttl() { return (ttl_); }
      /* ipv6 fields */
      int& flowid() { return (fid_); }
      int& prio() { return (prio_); }

      /* diffserv field */
      DSCPType& DSCP() { return DSCP_; }

};

#endif
```

APPENDICE B1

```
#ifndef lint
static const char rcsid[] =
    "@(#) $Header: /usr/src/mash/repository/vint/ns-2/ip.cc,v 1.8
1998/08/12 23:41:05 gnguyen Exp $";
#endif

#include <ctype.h>
#include "packet.h"
#include "ip.h"

int hdr_ip::offset_;


static class IPHeaderClass : public PacketHeaderClass {
public:
        IPHeaderClass() : PacketHeaderClass("PacketHeader/IP",
                                  sizeof(hdr_ip)) {
            bind_offset(&hdr_ip::offset_);
      }
      void export_offsets() {
            field_offset("src_", OFFSET(hdr_ip, src_));
            field_offset("dst_", OFFSET(hdr_ip, dst_));
```

```
            field_offset("ttl_", OFFSET(hdr_ip, ttl_));
            field_offset("fid_", OFFSET(hdr_ip, fid_));
            field_offset("prio_", OFFSET(hdr_ip, prio_));
      }
} class_iphdr;

DSCPType determine_codepoint(const char*string)
{
      char charbuf[20];
      char* to = charbuf;
      while ((*to++ = tolower(*string++)))
            /* NOTHING */;
      *to = '\0';
      to=charbuf;

      if (strcmp(to, "be" ) == 0)
            return BE ;
      if (strcmp(to, "Voice_IP" ) == 0)
            return Voice_IP ;
      if (strcmp(to, "Video_IP" ) == 0)
            return Video_IP ;
      if (strcmp(to, "IP_Radio" ) == 0)
            return IP_Radio ;
      if (strcmp(to, "IP_TV" ) == 0)
            return IP_TV ;
      if (strcmp(to, "IP_GAME" ) == 0)
            return IP_GAME ;
      if (strcmp(to, "Video_Streaming" ) == 0)
            return Video_Streaming ;
      return INVALID_CP;
      if (strcmp(to, "INTERACTIVE" ) == 0)
            return IP_GAME ;
if (strcmp(to, "REALTIME" ) == 0)
            return IP_GAME ;

}
```