THE UNIVERSITY OF THE WEST INDIES

ELECTRICAL & COMPUTER ENGINEERING DEPARTMENT

B.Sc. (Engineering)

Department of Electrical and Computer Engineering

ECNG 3020 – Special Project

PROJECT TITLE

FINAL REPORT

Sanjay Ramroop

807004374

September 24th 2010

Project Supervisor: Dr. Richelle Adams

Project Type: III

Statement of Academic Honesty

Rule 32, *The Faculty of Engineering: Undergraduate Regulations 2008-2009*, states:

"Cheating, Plagiarism and Collusion are serious offences under University Regulations.

i. Plagiarism is the unauthorized and/or unacknowledged use of another person's intellectual efforts and creations howsoever recorded, including whether formally published or in manuscript or in typescript or other printed or electronically presented form and includes taking passages, ideas or structures from another work or author without proper and unequivocal attribution of such source(s), using the conventions for attributions or citing used in this University. Plagiarism is a form of cheating.

ii. For the purposes of these Regulations, 'collusion' shall mean the unauthorized or unlawful collaboration or agreement between two or more students in the preparation, writing or production of a course assignment for examination and assessment, to the extent that they have produced the same or substantially the same paper, project report, as the case may be, as if it were their separate and individual efforts, in circumstances where they knew or had reason to know that the assignment or a part thereof was not intended to be a group project, but was rather to be the product of each student's individual efforts.

iii. Where two or more students have produced the same or substantially the same assignment for examination and assessment in circumstances that the assignment was to be the product of each student's individual efforts, they shall receive a failing grade in the course. "

I SANJAY RAMROOP (807004374) declare that the material submitted for assessment in ECNG 3020 is my own work and does not involve plagiarism and collusion.

Student's Signature Date

Abstract

The current Internet provides the Best Effort service. It does not differentiate among packets that belong to different hosts or those that may have different performance requirements. This poses a problem for real time applications that require a minimal amount of resources to operate effectively; such as IP telephony and video conferencing. To ease this problem, the Internet Engineering Task Force (IETF) has proposed the Differentiated Services network architecture that will allow for resource allocation and service differentiation in the Internet. The objective of this project was to evaluate the performance of DiffServ in the transport of multimedia traffic as opposed to without it (i.e. Best Effort service).

The DiffServ architecture was evaluated via simulation using the ns-3 discrete event simulator. The simulator at the time did include a DiffServ implementation; therefore one was designed and then implemented. To generate multimedia traffic, two applications were used; a VoIP application and a video streaming application. Several experiments were carried out to evaluate the architecture. Experiments 1 and 2 involved resource allocation tests via bandwidth allocation and drop priority. Experiment 3 looked at the bandwidth utilization and flow isolation of DiffServ and Best Effort networks. Experiment 4 compared the performance of different metering schemes while Experiment 5 investigated an actual DiffServ deployment and its performance compared to the Best Effort service. The results indicate that the Differentiated Services architecture is able to provide service differentiation and resource allocation in IP networks.

Acknowledgements

This report would not have been possible without the tremendous assistance provided by my project supervisor, Dr. Richelle Adams, whose support and guidance throughout the duration of the project helped me to understand the subject and overcome problems encountered along the way. I am also indebted to my father and mother for providing transportation to and from campus and encouraging me to do my best.

I would also like to thank all those who provided the motivation and inspiration needed for the completion of the project.

Sanjay Ramroop

Table of Contents

List of Abbreviations and Symbols

AF Assured Forwarding

BA Behavior Aggregate

BE Best Effort

DE Default Behavior

DiffServ Differentiated Services

DS Differentiated Services

DSCP Differentiated Services Code Point

EF Expedited Forwarding

FIFO First in First out

IETF Internet Engineering Task Force

IP Internet Protocol

MF Multi Field

PHB Per-Hop Behavior

PQ Priority Queuing

QoS Quality of Service

SLA Service Level Agreement

TCA Traffic Conditioning Agreement

UDP User Datagram Protocol

VoIP Voice over IP

WRED Weighted Random Early Detection

WRR Weighted Round Robin

List of Figures

List of Tables

1 Introduction

1.1 Why is Quality of Service necessary?

The current internet provides a service referred to as the Best Effort service. The Best effort service does not provide any resource assurance to any packets in the network. The network makes its 'best effort' to move packets from sender to receiver as quickly as possible but the user is not given a guarantee that the data is delivered or delivered in a timely manner. All packets sent into the Best Effort network compete equally for network resources such as link bandwidth and buffer space. The network makes no attempt to differentiate between packets sent from different sources. (Wang 2001).

However the Best Effort service is sufficient for some applications. Applications that can tolerate packet losses and large delays; such as email and file transfers are suitable for the Best Effort service. However the Best Effort service does not satisfy the needs of new real time applications that require a minimal amount of resources to operate effectively; such as IP telephony and video conferencing. Another concern is service differentiation. Since the Best effort service does not differentiate between packets; it cannot offer different levels of services. Applications however, require different levels of performance. Requirements also vary among customers. For example, companies that use the internet to perform banking may pay more to have their data have better service. The capability of a network to provide this resource assurance and service differentiation is referred to as Quality of Service (QoS) (Wang 2001).

1.2 Definition of Quality of Service (QoS)

Quality of Service can be defined from two perspectives:

i. The User

ii. The Network

QoS from the perspective of the user is the perception of the quality the end user experiences from the network provider for the service he subscribed to. Quality of Service from the network perspective refers to the capability of the network to provide the QoS perceived by the end user. To provide QoS over a network, the network must have the following abilities (Figure 1):

1. The ability to differentiate between network traffic so that particular classes of traffic can be treated differently than others.

2. The ability to treat the different classes of traffic differently by providing service differentiation and resource assurance in the network

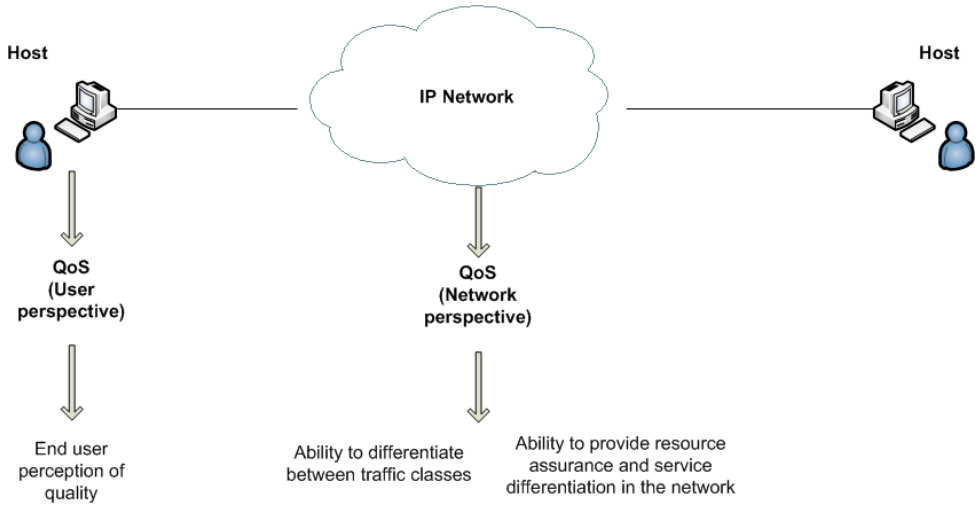

Figure 1: QoS perspectives

Source: (Park 2005)

These two abilities are accomplished by an IP network as shown in Figure 2 below.

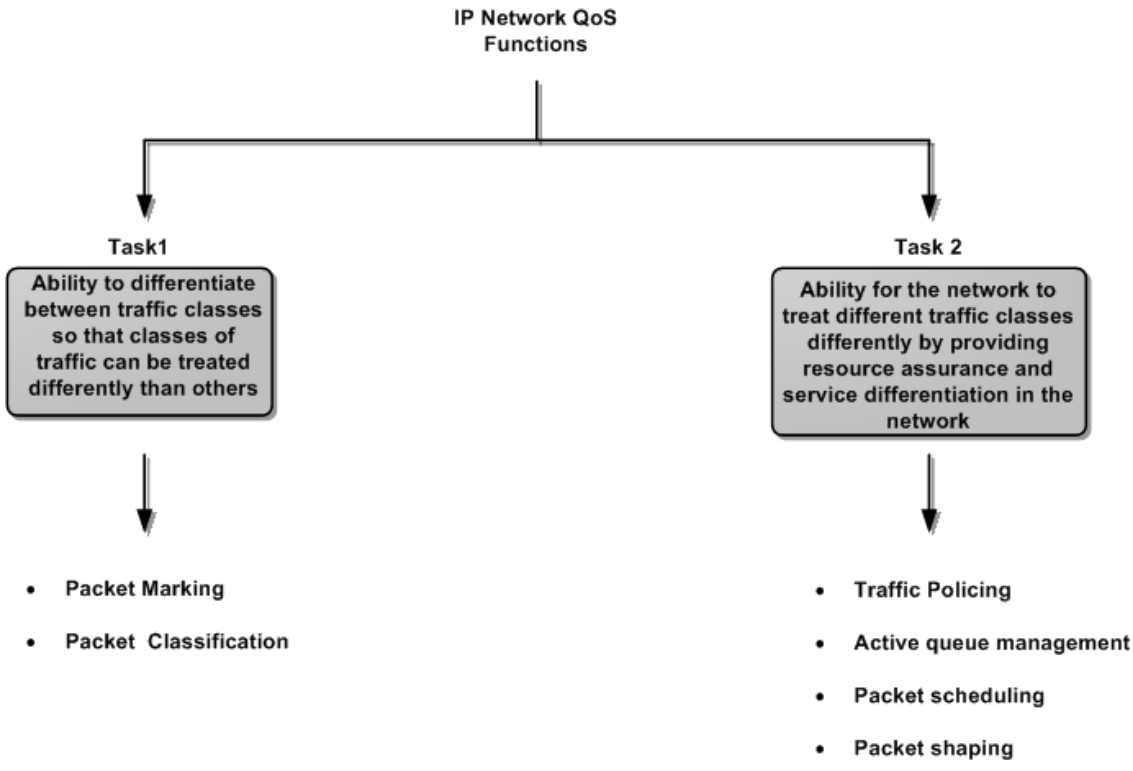

Figure 2: IP QoS requirements

Source: (Park 2005)

1.3 Project Details

A QoS mechanism in the network provides a way for packets to be distinguished and treated differently. The Differentiated Services architecture or DiffServ is a QoS mechanism for IP networks that has been proposed by the Internet Engineering Task Force to allow for resource allocation and service differentiation in the Internet.

The objective of this project was to evaluate the performance benefits of implementing the Differentiated Services architecture as opposed to the current Best Effort service. The architecture was evaluated via simulation using the ns-3 network simulator. Since the simulator at the time did not include a DiffServ implementation, one was designed and then implemented. For the evaluation of the architecture, multimedia applications were modeled within the simulator to generate traffic for the DiffServ enabled network. Experiments were then conducted to compare DiffServ and the Best Effort service. A device for collecting the simulation output was also created to help with the evaluation. Currently in the internet DiffServ is not well deployed. Only a small number of networks are DiffServ enabled. Performing studies such as these may show the performance benefits that DiffServ has over the current Best Effort service and encourage service providers to upgrade their networks to support DiffServ.

2 Literature Review

2.1 Comparison with Past Studies

In recent years there have been a number of studies evaluating the performance of the Differentiated service architecture. Some studies focus on the performance of different scheduling and buffer management schemes for the Assured and Expedited forwarding PHBs. For example both (Jung, Kwak and Byeon 2002) and (Mao, Moh and Wei 2001) investigated the performance of the Priority Round Robin (PRR), Weighted Round Robin (WRR) and a hybrid Priority Queuing Weighted Round Robin (PQWRR) queue scheduling schemes. They both came to the conclusion that PQWRR is able to provide the EF PHB with its jitter and delay requirements while providing better bandwidth allocation among the AF classes according to their weights. (Mao, Moh and Wei 2001) also explained that PQWRR offers the advantage of the PQ and WRR schemes while avoiding their disadvantages and that it is most suitable for supporting DiffServ. Interestingly enough, this same scheduling scheme was designed for this project from a combination of the AF and EF PHB implementations.

Others studies such as (Di and Mouftah 2000) looked at the performance of the Per Hop Behaviors themselves. They came to the conclusion that DiffServ PHB mechanisms are able to provide good service differentiation. However, little work has been done in the comparison of DiffServ and Best Effort networks. This projects aims to evaluate DiffServ in this aspect.

2.2 Comparison with previous Ns-2 DiffServ implementation

The previous ns-2 DiffServ implementation (diffserv Directory Reference n.d.), displays a number of similarities with the ns-3 implementation presented in this report. Ns-2 defines two types of queues; an Edge and Core queue which carry out the specific DiffServ functions of the Edge and Core routers. In comparison to the ns-3 implementation, a single queue is modeled called DiffServQueue but with interchangeable Core and Edge modes. This approach was taken since both Edge and Core routers have a number of common functions and therefore do not require a separate class for their implementation.

The ns-2 scheduling and buffer management within the Edge and Core queues is superior to this implementation. They offer a variable number of internal queues and diverse scheduling schemes as well as configurable DSCP to PHB mappings. However where the ns-3 implementation surpasses ns-2, is at the traffic classification and conditioning stage that occurs at the Edge router.

The ns-2 method of performing traffic classification and conditioning by an Edge router also shows many similarities. They define a data structure called 'PolicyTableEntry' which contains the traffic profile (CIR PIR), meter state parameters (last packet arrival, CBS), meter type (Token bucket, srTCM) and IP source and destination addresses for all traffic between two nodes. Therefore all flows between those two nodes will belong to the same PolicyTableEntry. The ns-3 implementation takes this another step. It identifies unique flows and assigns them to a Service Level Agreement. SLAs are

represented in the ns-3 implementation by the class 'DiffServSla.' What separates these two data types is the level of detail they provide. DiffServSla incorporates the functionality of PolicyTableEntry but is 'attached' to flow objects represented by the class 'DiffServFlow'. In this way, the end nodes or customers of the network can be provided with **flow specific** service differentiation.

Secondly, ns-2 implements metering algorithms, such as the token bucket and the srTCM, in the 'Policy' class or more specifically, sub classes of the Policy class. However the problem lies in the fact that the Policy class and all of its sub classes are contained in a single document. This creates a problem when new metering algorithms are to be implemented since the source code files must change. In the ns-3 implementation, metering algorithms are implemented as sub classes of the class 'DiffServMeter' but are written in their own respective files. To include a new metering algorithm, the user simply creates a new class that inherits from DiffServMeter. This coding method removes the need of altering previous code.

Thirdly, each Policy sub class or meter is associated with a number of metering outcomes. For each of these outcomes there is a user configurable Differentiated Services Code Point (DSCP) that is stored in the data structure 'PolicerTableEntry'. PolicerTableEntry defines an initial code point and two other downgraded code points. Each Policy is mapped to a single PolicerTableEntry data structure so that the metering outcomes produced by the Policy are mapped to a code point contained in the PolicerTableEntry. The disadvantage of this method is that the code point

configurations are static for the particular Policy for the duration of the simulation. That is, all PolicyTableEntrys that use the same Policy will be subjected to the same PolicerTableEntry and hence marked with the same code points.

The presented ns-3 implementation overcomes this by allowing code point decisions to be mapped to a DiffServSla object. Each DiffServSla object contains a data structure called ConformanceSpec which contains the initial code point and two downgraded 'actions'. Therefore even though two DiffServSla objects may use the same metering algorithm, they may have different code point markings for each metering result. Another disadvantage of the ns-2 implementation is that packets can only be downgraded to an inferior point if found non conformant by the metering algorithm. In the ns-3 implementation, packets may be marked with an inferior code point or dropped if the user desires. This provides more options to be performed on packets. Another action that may be performed on packets is packet shaping, however this was not implemented.

2.3 The Differentiated Services Network Architecture

This section presents an explanation of the Differentiated Services architecture to assist the reader in the understanding of the material presented in this report.

2.3.1 Network service and Service Differentiation

A Network Service defines the characteristics of packet transmission in a single direction across one or more paths with a network. The characteristics of packet transmission may be defined in quantitative terms such as throughput, latency, jitter

and loss. It may also be defined in terms of the priority of access to network resources. Service differentiation is intended to satisfy different application requirements, meet different user expectations and accommodate differentiated pricing of services within the Internet (Blake, et al. 1998).

2.3.2 DiffServ Overview

The DiffServ Architecture consists of a number of functional elements that are implemented in network nodes. These include per hop forwarding behaviors and traffic classification and conditioning functions such as metering, marking, shaping and dropping. Network traffic is divided into a small number of forwarding classes and resources are allocated to each class. Traffic first enters at the boundaries of the network, where it is classified, conditioned and assigned into a smaller number of forwarding classes. The forwarding class that a packet belongs to is identified by a unique encoding in the DS field of the IP header, called a Differentiated Services Code Point (DSCP). Each forwarding class represents a predefined forwarding behavior in terms of bandwidth allocation and packet drop priorities. To assign a packet to a particular forwarding class, the DS field of the packet must be encoded with the DSCP of the forwarding class. (Wang 2001)

A collection of packets with the same DSCP is referred to as a Behavior Aggregate (BA). At the interior of the network, each BA and hence DSCP, is associated with a particular forwarding treatment known as a Per-Hop Behavior (PHB). BAs experience their corresponding PHB at every Differentiated Service Compliant node in their path. PHBs

provide a means of allocating forwarding resources (Bandwidth and Buffer) to each BA, and are implemented in interior nodes through various packet scheduling and buffer management mechanisms (Blake, et al. 1998).

Therefore the DiffServ architecture consists of two types of nodes which have different responsibilities. The edge or boundary node is responsible for traffic classification and conditioning and the interior or core node is responsible for PHB based forwarding of packets. Classification and conditioning involves mapping selected packets to a particular forwarding class by marking them with the corresponding DSCP, and also ensuring that the traffic flows from a customer are in conformance with their Service Level Agreements. This involves taking action on non conformant packets; dropping for example. The interior nodes are responsible for identifying the forwarding class from the DSCP of the packet IP header and applying the corresponding PHB (Wang 2001).

It must be noted that boundary nodes must be capable of doing this as well. In this way the DiffServ architecture becomes more scalable as instead of managing the many individual flows, the architecture only manages the fewer, aggregate of the flows. Scalability is also achieved as explained in (Blake, et al. 1998), by only implementing the classifying and conditioning functions at network boundaries and keeping the network interior simple.

An important point to note is that Differentiated Services architecture allocates resources (via PHBs) to the forwarding classes and not to individual flows. The flows that belong to a forwarding class have a degree of performance or resource assurance

that corresponds to the provisioning and prioritization of that class. Therefore DiffServ provides its resource assurance through provisioning and prioritization rather than per flow reservation. By assigning resources to forwarding classes and controlling the amount of traffic for the class; in order words the loading of the class, DiffServ can provide resource assurances and different levels of services. However it cannot provide absolute bandwidth and delay guarantees for individual flows (Wang 2001).

Also explained in (Wang 2001), DiffServ does not define end to end services but only forwarding behaviors or PHBs. Each PHB represents a particular treatment; not a service. The services can be constructed by combining PHBs with other elements. Another point to note is that DiffServ only provides service differentiation in one direction only. Because of this behavior it is called asymmetric (Blake, et al. 1998). Figure 3 below shows a comparison of the current best effort network and the differentiated services network.

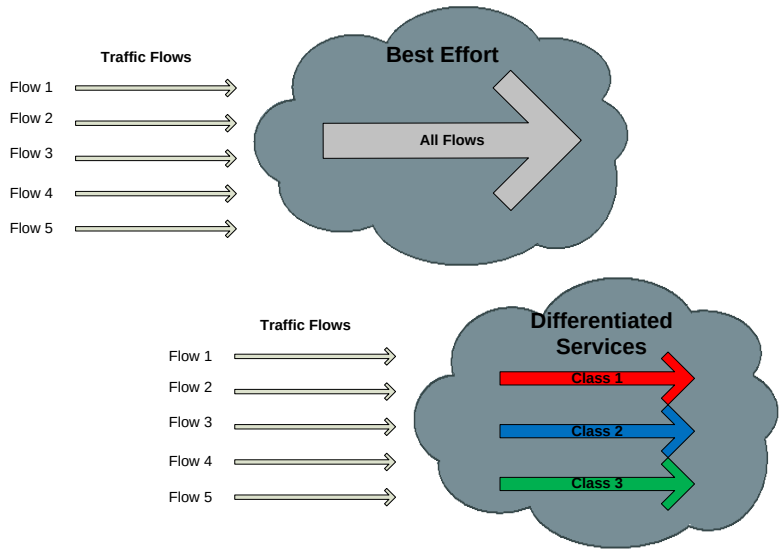

Figure 3: DiffServ vs. Best Effort networks

Source: (Park 2005)

2.3.3 Service Level Agreements and Traffic Conditioning Agreements

Services in the DiffServ architecture are defined between a service provider and a customer in the form of a Service Level Agreement (SLA). The SLA specifies all details of the service to be provided. An important part of the SLA with respect to DiffServ is the Traffic Conditioning Agreement (TCA). The TCA specifies how the classifier and traffic conditioner should operate with respect to a customer's packets. The TCA as defined in (Blake, et al. 1998), is an agreement that specifies:

i. Classifier rules used to select packets

ii. Traffic profiles for each of the forwarding classes

iii. Metering, marking, discarding and/or shaping rules which are to apply to traffic streams selected by the classifier

The Classifier rules are used by a MF classifier to identify the subset of traffic from a customer that requires a differential service. The customer also agrees to transmit packets into a forwarding class at an average rate and burst (may include other parameters such as peak rate). The network will then provide the requested service for this level of traffic from the customer. However the network cannot assume that the customer will uphold his end of the agreement and therefore must monitor and enforce this rate where necessary. The network performs this task through traffic conditioning functions. The actions that can be performed include metering, marking, shaping and dropping. The rate at which the customer agrees to send his packets is called a traffic profile. A traffic profile, as defined in (Blake, et al. 1998), gives a

description of the temporal properties of a packet stream such as rate and burst size. It provides rules to determine if the customer is conformant with his SLA. The marking, dropping and shaping rules of the TCA are the actions taken on conformant and non conformant packets that the customer sends into the network.

As explained in (Wang 2001), the purpose of the DiffServ architecture is to ensure that the SLAs between the service providers and customers are honored. These agreements are enforced by the boundary nodes that connect service providers and customers. The Boundary nodes perform this task through the use of traffic conditioning.

2.3.4 Differentiated Services Domains

A DS domain consists of interconnected DS complaint nodes that implement the same Per Hop Behaviors, and share a common service provisioning policy (Figure 4). A service provisioning policy defines how the traffic conditioners located on boundary nodes are configured as well as how traffic streams are mapped to behavior aggregates. A DS domain normally consists of networks under the same administration such as an ISP or an organization's intranet. The service provider is responsible to ensure that the domain is provisioned with sufficient resources to support the SLAs offered by the domain (Blake, et al. 1998).

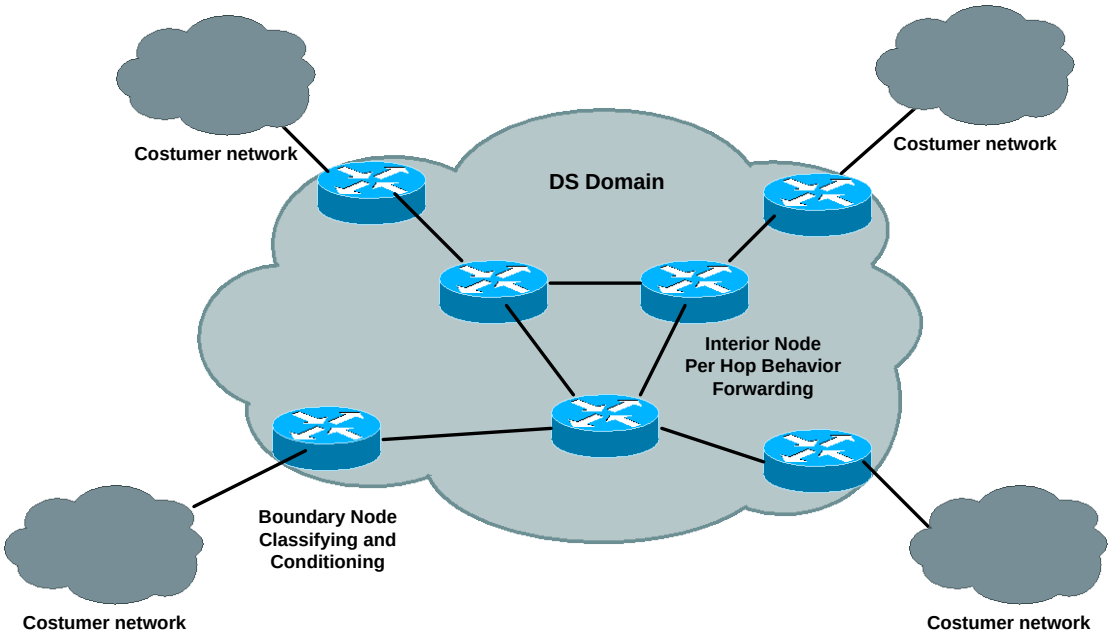

Figure 4: DS Domain

Source: (Wang 2001)

2.3.5 The Network Edge or Boundary nodes

In a DS domain, the boundary node interconnects the current DS domain to other DS or non DS capable domains. It also connects to the interior nodes in the current DS domain. A host in the DS domain may act as a boundary node for the applications running on that host. Otherwise the DS node nearest to that host acts as the DS boundary node for the host's traffic. It is here at the boundary that packets are classified and traffic conditioning is performed (Blake, et al. 1998). The classification and conditioning process of a boundary node in a DS domain is responsible for mapping packets to a forwarding class supported in the network and ensuring that the traffic from a customer conforms to their SLAs (Wang 2001).

2.3.5.1 MF Classification

A packet classifier as selects packets from a traffic flow matching some specified rule, which is based on the content of some portion of the IP packet header. The DiffServ architecture defines two types of classifiers; the Multi field Classifier and the Behavior Aggregate Classifier. The packet classifier in the network edge is the Multi-field (MF) Classifier and selects packets based on the value of a combination of one or more IP header fields such as source IP address, destination IP address, port numbers, DS field etc. MF Classifiers (Figure 5) are used to steer packets that match a specified rule to a logical instance of a traffic conditioner for additional processing (Blake, et al. 1998).

(Blake, et al. 1998) further explains that the classifier gets its classification rules from the TCA. It uses the classification rules specified in the TCA of a customer to select the particular traffic flows that require a certain service.

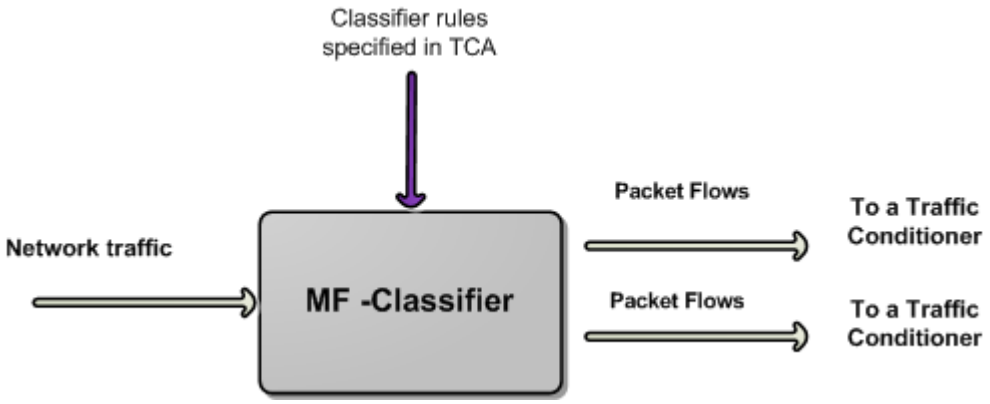

Figure 5: MF Classifier

2.3.5.2 *Traffic Conditioning*

Traffic conditioning as defined in (Blake, et al. 1998), is a set of control functions that is applied to classified packets streams in order to enforce Traffic Conditioning Agreements; which are made between customers and service providers. Traffic conditioning is performed by a Traffic Conditioner and which may contain meters, markers, shapers and droppers as shown in Figure 6 below.

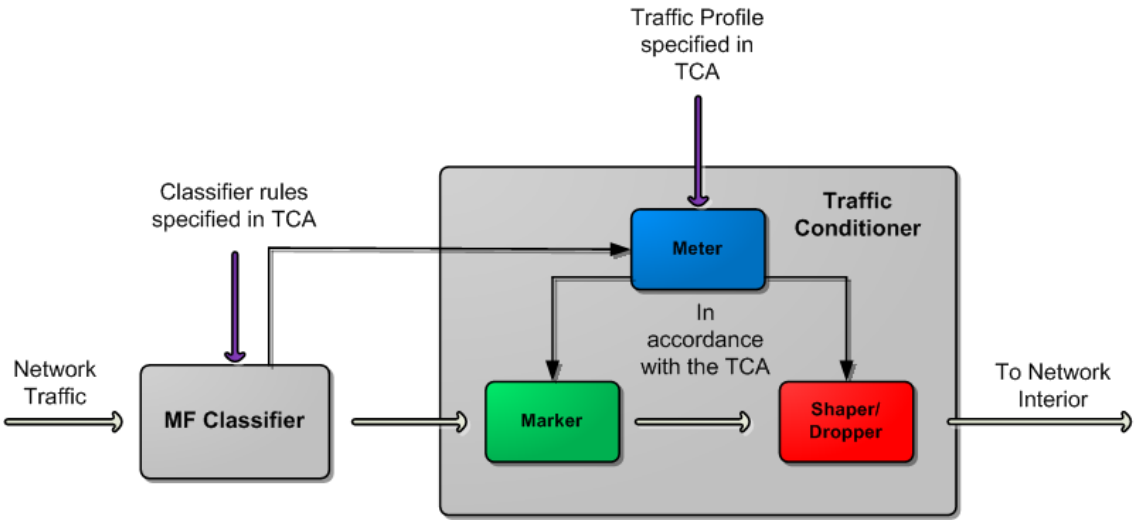

Figure 6: DS domain boundary node - Traffic classification and conditioning

Source: (Blake, et al. 1998)

Traffic conditioners are used to remark the traffic streams selected by the classifier or may shape or discard packets to alter the characteristics of the traffic stream. This is performed for the purpose of bringing the traffic stream in conformance with a traffic profile. A meter is used to measure the classified traffic stream against a traffic profile. The state of the meter (whether the packet is in or out of profile) may then be used to enable a marking, shaping or dropping action (Blake, et al. 1998).The MF classification stage is first used to establish the packet's context or background. The conditioner

then knows which metering profile to apply, what code point the packet should be marked with, and whether to shape or drop packets. These actions as well as the classification rules are specified in the customer's TCA (Armitage 2000).

2.3.5.3 Packet Marking

Marking is the process carried out by a marker to assign a packet to a particular forwarding class. It involves encoding a DSCP into the DS field of the IP packet header (Figure 7). Marking may be carried out on both in and out of profile packets. The DS field replaces the previous IPv4 TOS field and the IPv6 Traffic Class octet. When leaving the traffic conditioner of a boundary node, packets must contain a suitable DSCP (Blake, et al. 1998).

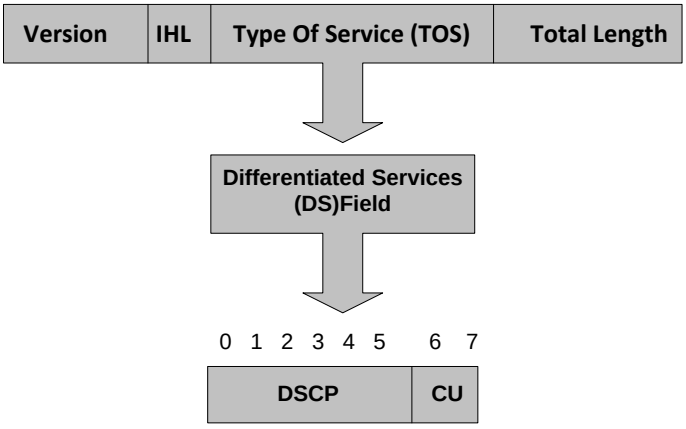

Figure 7: Differentiated Services field and DSCP

Together classifiers and traffic conditioners select which packets are to be added to the different behavior aggregate on entry to a domain. Within the network core, Behavior Aggregates then receive differential treatment.

2.3.6 The Network Interior or Core nodes

(Blake, et al. 1998) explains that interior nodes within a DS domain connect to only other interior and boundary nodes within the same DS domain. Differentiated services are accomplished within the interior node by applying Per Hop Behaviors to each Behavior Aggregate. Each PHB maps to a DSCP and hence Behavior Aggregate. (Blake, et al. 1998) defines a PHB as a description of the particular 'externally observable' forwarding behavior that is applied to a DS Behavior Aggregate at every DS compliant node in their path. A PHB is the manner in which nodes allocate resources to Behavior Aggregates. PHBs can be defined in terms of their resource priority relative to other PHBs such as bandwidth and buffer space, as well as their relative observable traffic characteristics such as packet loss and delay. A PHB may be defined as part of a PHB group. PHBs in a PHB group share a common constraint that applies to all PHBs in the group such as a packet scheduling or buffer management mechanism (Blake, et al. 1998).

The BA to PHB mapping function is performed by a BA classifier (Figure 8). BA classifiers classify incoming packets only by the DSCP code point in the IP packet header. No other field is used (Blake, et al. 1998).

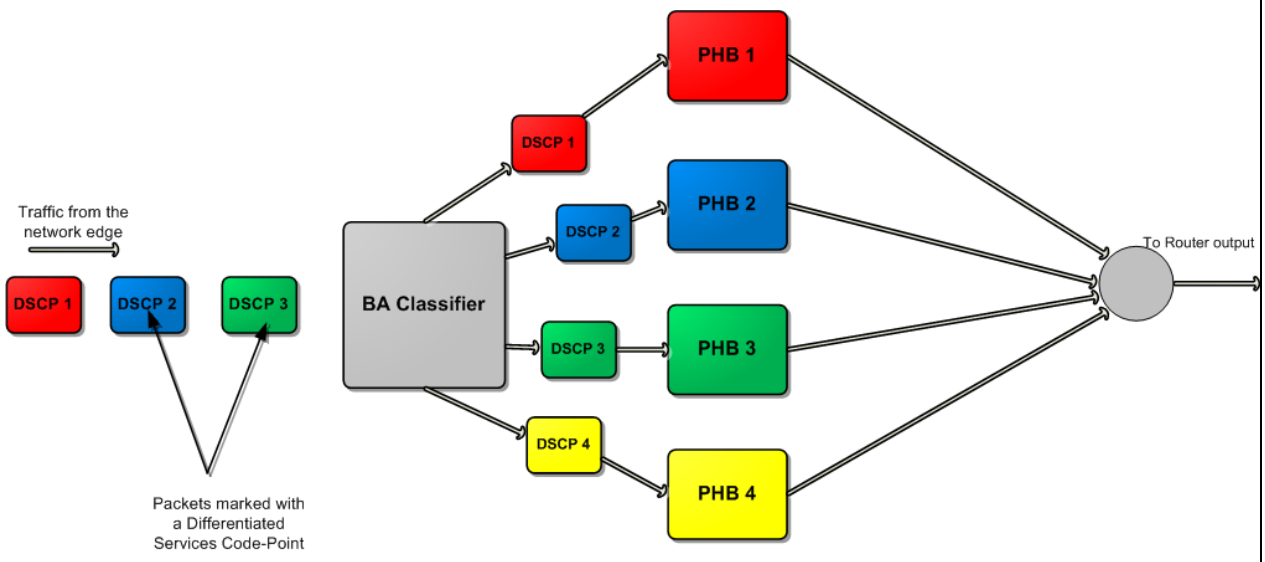

Figure 8: DS domain Interior node – BA to PHB mapping

A particular PHB is then applied to each Behavior Aggregate. PHBs are implemented by means of various buffer management and packet scheduling mechanisms inside of a router. However they are not defined by a particular implementation mechanism. A variety of implementation mechanisms may be capable of implementing a particular PHB. PHBs that have been standardized for the DiffServ architecture are the Assured Forwarding (AF) PHB group and the Expedited Forwarding PHB. (Blake, et al. 1998). A description of these behaviors can be found in Appendix A.

3 Methodology

The methodology chapter comprises of a number of sections. It includes the DiffServ architecture design and implementation, the structure of the ns-3 simulator, performance metrics used for evaluation, statistics collection and the modeling of multimedia applications.

3.1 Designing the DiffServ architecture

The design of the DiffServ architecture consisted of the following three sections:

i. SLA design– How should a customer's flows be mapped to forwarding classes

ii. Core Node Design –What PHBs should be implemented and what mechanisms or algorithms should be used to implement them in the network core routers

iii. Edge Node Design – How should classification and traffic conditioning be performed in the network edge routers

3.1.1 Service Level Agreement design

For the Simulated network, the structure of a customer's SLA must be defined. As stated previously, a customer requests a service from a service provider in the form of a Service Level Agreement. The service provider has a finite number of forwarding classes that the customer may choose from. The customer places his packets into a forwarding class via marking with the appropriate DSCP. The SLA of the customer

contains this code point and the subset of application flows that are to be marked with this code point. However the customer may require more than one forwarding class for different application flows. For simulation purposes, this was not included; the SLA only contains the code point of one forwarding class and for one subset of application flows. If the customer desires to use another forwarding class for a subset of his traffic, another SLA will have to be created.

Now the service provider must limit the amount of traffic the customer can send into each forwarding class through traffic conditioning at the boundary node. The customer and service provider agree on a traffic profile for the customer that specifies the rate at which the customer sends traffic into the forwarding class. Associated with this traffic profile is a particular metering algorithm, which is used to meter the customer's packets against the traffic profile. Since the service provider may have multiple metering methods available, the metering algorithm is specified in the SLA in addition to the agreed traffic profile. The traffic profile for the customer consists of a combination of the following information:

i. Committed Information Rate (CIR)

ii. Committed Burst Size (CBS)

iii. Excess Burst Size (EBS)

iv. Peak Information Rate (PIR)

v. Peak Burst Size (PBS)

Accompanying the traffic profile is the meter's 'state' such as the arrival time of the last packet and the number of tokens currently in the committed bucket. This information is also stored in the SLA for the customer.

If the customer transmits packets within the rate of the agreed traffic profile, the packets are marked with the specified code point. If the customer should exceed this rate, actions are taken on those non-conformant packets. Up to two non-conformant actions are allowed for this implementation and include marking with an inferior code point or dropping. This information is also included in the SLA. Therefore for each forwarding class the customer uses, a SLA is created that includes the following information:

i. the forwarding class the customer's packets belong to (DSCP)

ii. The subset of application flows that belong to the forwarding class

iii. the traffic profile, meter state and meter type

iv. And the two actions that are to be taken for non conformant packets.

The SLAs of all customers of the network are stored in an SLA database for the edge classification and conditioning components to use. Figure 9 summarizes this design.

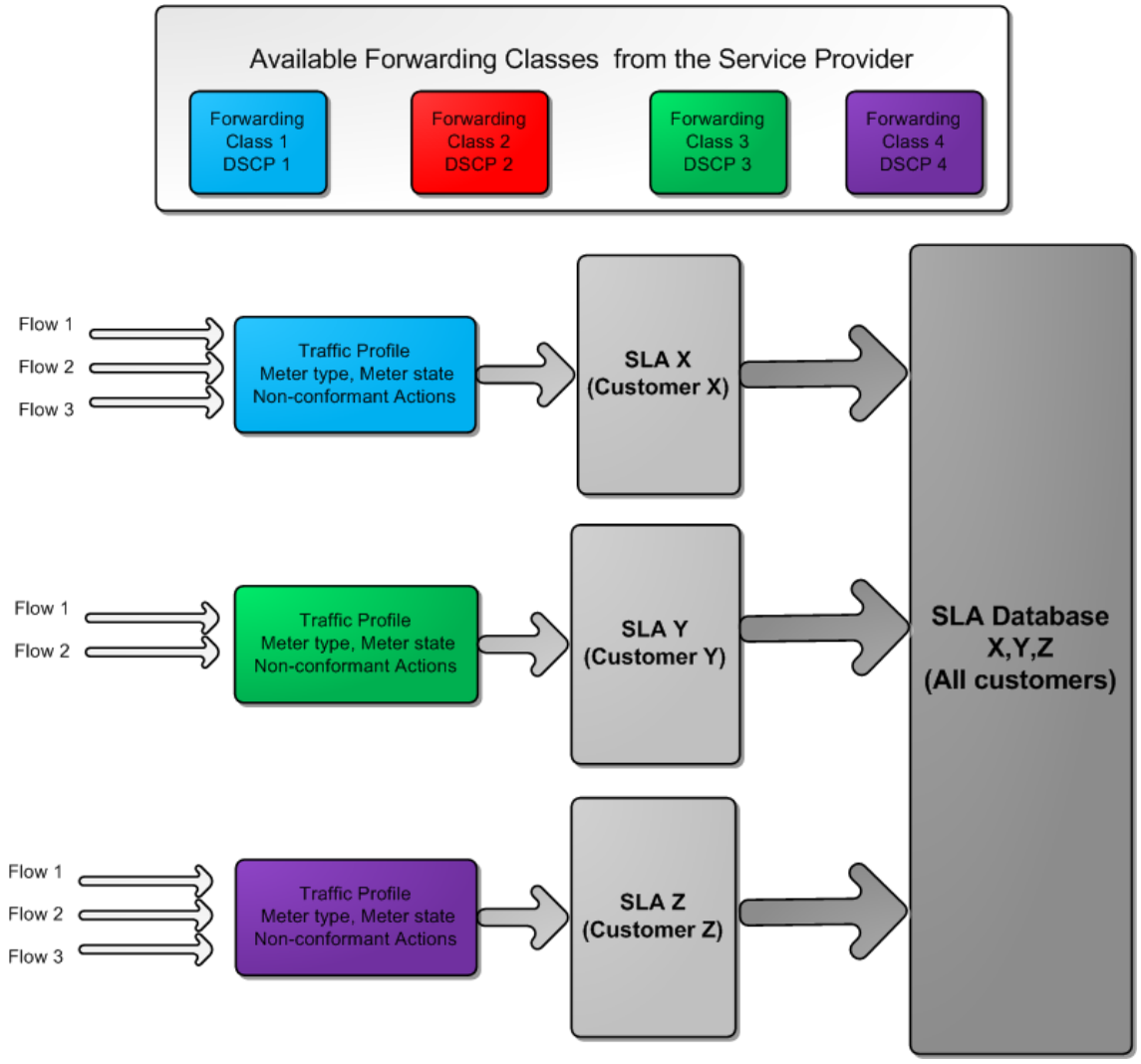

Figure 9: SLA, forwarding class and flow configuration

3.1.2 Core Node Design

The Core Node must include a BA classifier and the PHB implementations. For this DiffServ implementation, the standardized AF PHB group, the EF PHB, and the Default PHB is implemented. Each PHB requires Buffer Management and Packet Scheduling mechanisms for its implementation. However no particular mechanisms for their implementation are specified in the AF and EF PHB standards. This is because the standard only specifies the forwarding behavior and not the implementation. As stated

before, a PHB is a particular 'externally observable' forwarding treatment that a Behavior Aggregate receives at every DS compliant node in their path. Therefore the standard does not specify a particular implementation, but only the 'external behavior' itself. Therefore the decision of the particular implementation mechanisms must be made before attempting to implement the components within the ns3 simulator. An implementation that complies with the AF and EF PHBs standards was created, and is described as follows.

3.1.2.1 The Assured Forwarding PHB group implementation

In this implementation each AF class is represented by its own queue (Figure 10). This allows packets of an AF class to be independently forwarded from traffic from other AF classes and also gives each AF class its own amount of buffer space as required by the AF standard. The standard also requires that each AF class be provided with a minimum, configurable amount of bandwidth. To accomplish this, the AF queues are serviced under a Weighted Round Robin scheduler. The configurable weight that is given to each AF queue determines the share of bandwidth they receive. The scheduler also uses a work conserving discipline. Therefore if the current queue to be serviced is empty, the scheduler moves on to the next queue, so that the link is always utilized.

Internally each AF queue is serviced using a FIFO mechanism to prevent reordering of packets of the same AF class (another requirement). All queuing and discarding requirements of the AF PHB are fulfilled using the Weighted Random Early Detection

AQM mechanism (WRED). The AF standard requires that packets of a lower drop precedence level have a higher probability of being forwarded than a packet of higher drop precedence level. WRED accomplishes this requirement by using different RED profiles for different packets within a queue. For higher drop precedence levels, a more aggressive RED profile can be used. A more detailed explanation of these mechanisms can be found in Appendix B.

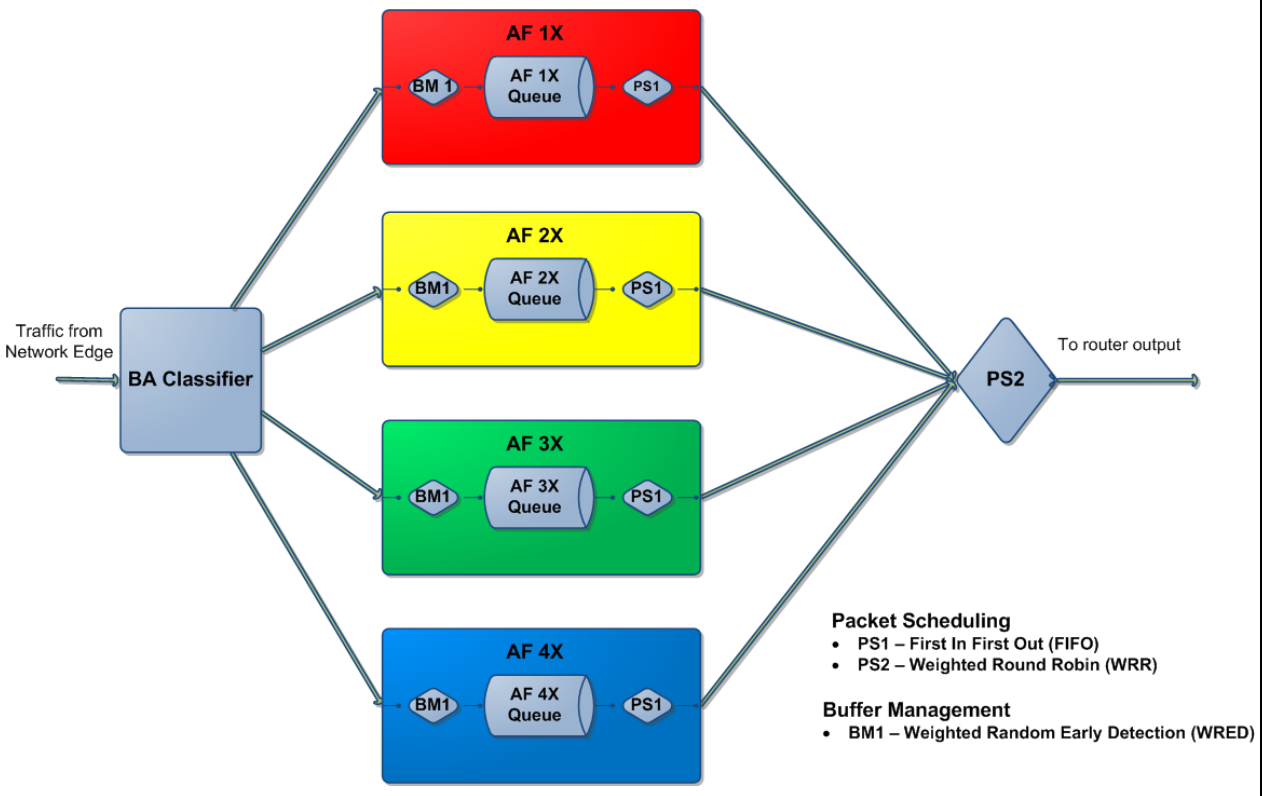

Figure 10: AF PHB implementation

3.1.2.2 The EF PHB implementation

In this implementation the EF PHB is represented by its own queue (Figure 11). The EF standard requires that the departure rate of EF packets from a DS node should equal or exceed a configurable rate and that the departure rate be independent of the

current traffic load of the DS node. This is accomplished using Priority Queue scheduling to service the EF queue and by giving the EF queue the highest priority of all the queues in the router. Internally the EF queue is serviced using a FIFO mechanism and Drop Tail buffer management. According to the standard, if the EF PHB is implemented using mechanism that allows the unlimited preemption of other traffic such as a priority queue, a rate limiter must be implemented to ensure the EF queue does not starve other queues in the router. This rate limiter is implemented as a token bucket dropper. Traffic that exceeds this rate is discarded as required by the standard. The token bucket also allows the departure rate to be configurable. A more detailed explanation of these mechanisms can be found in Appendix B.

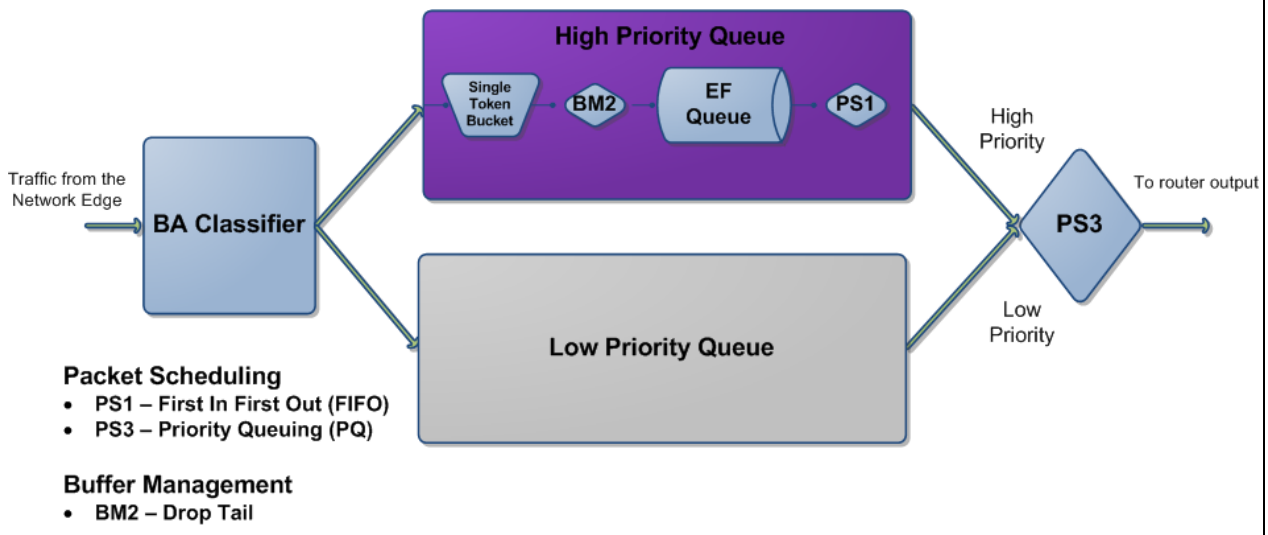

Figure 11: EF PHB implementation

3.1.2.3 The Default PHB implementation

The Default PHB is also implemented using a separate queue. The queue is serviced under the same WWR scheduler as the AF queues. Internally the queue is serviced using a FIFO mechanism and Drop Tail buffer management.

3.1.2.4 Combining PHB implementations

Combining the AF, EF and the Default PHB into one system produces the following setup in the Core Node as shown in the Figure 12 below. Note that the packet scheduler is a two level scheduling scheme. The first scheduling level is the AF WRR scheduler which services the AF and default queues. The second level is the Priority Queue scheduler which services the EF queue and the output of the WRR scheduler. From the perspective of the Priority Queue scheduler, the AF and default queues are seen as the low priority queue while the EF queue is seen as the high priority queue. This two level scheduling scheme ensures that EF packets have higher priority over AF and Default packets and provides EF packets with low delay and jitter. The scheduler performs this task while still allowing varying bandwidth allocations among AF queues.

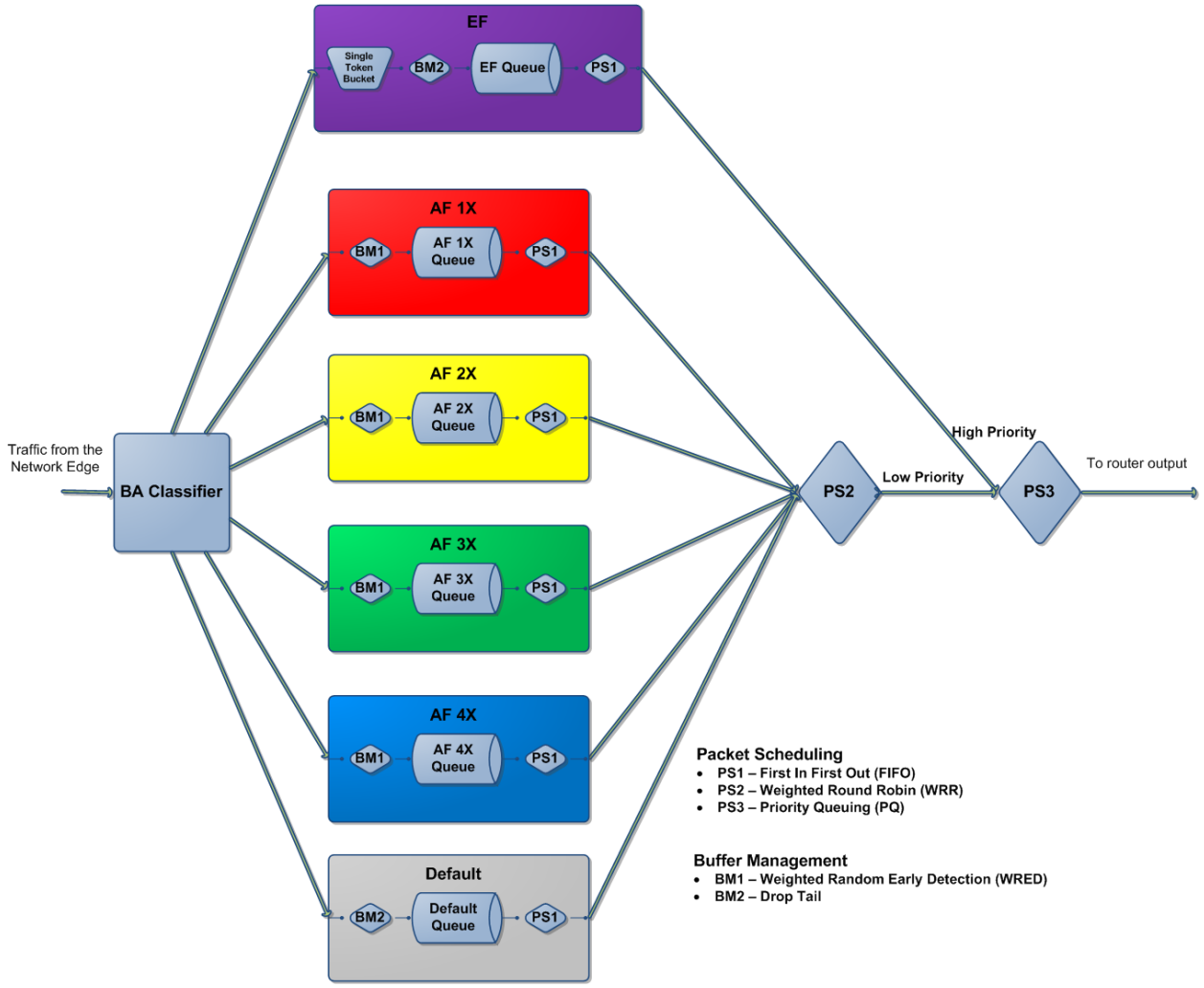

Figure 12: Core Node Design

3.1.3 Edge Node Design

The Edge Node in the network contains the functionality of the Core Node as well as traffic classification and conditioning functions. Traffic classification and conditioning is carried out by the following three components:

1. MF Classifier - SLA identification
2. Meter - conformance evaluation
3. Enforcer – SLA enforcement

3.1.3.1 MF classifier

When a packet enters the MF Classifier (Figure 13), it looks at the packet's IP header fields and selects the SLA that the packet belongs to from the SLA database. Recall that each SLA is associated with a specific and unique subset of flows. Therefore the classifier can use the IP header fields to identify the SLA the packet belongs to. The classifier then forwards the packet and the identified SLA to the Meter. If the packet does not belong to a SLA, it is not conditioned. It remains in the default forwarding class and is forwarded to the BA classifier to be enqueued onto the default queue.

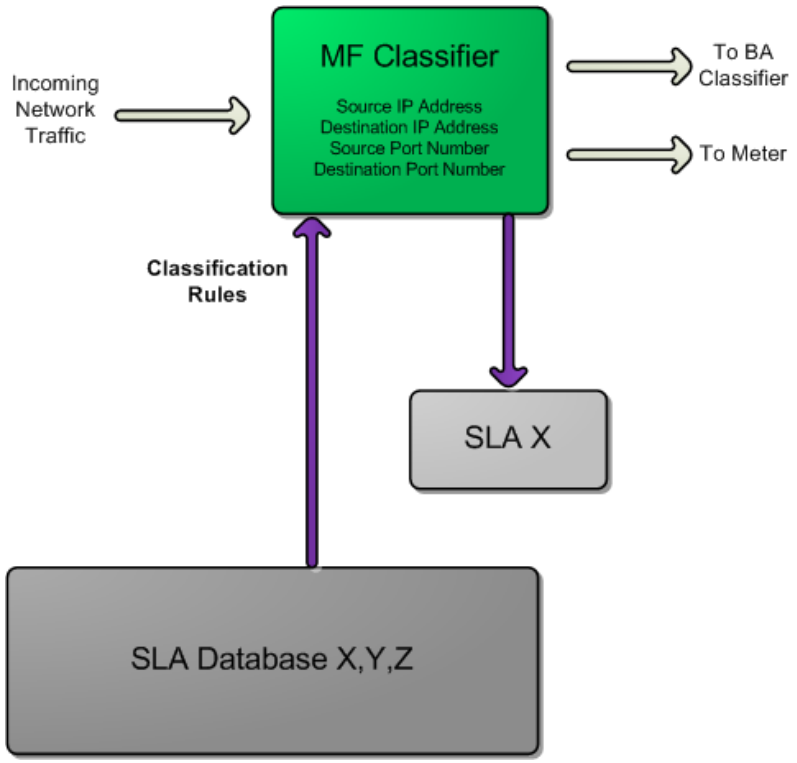

Figure 13: Edge Node –MF Classifier

3.1.3.2 Meter

When a packet leaves the MF Classifier and enters the Meter, the particular meter type specified in the SLA of the packet is chosen from a Metering algorithm database and is used to operate on the packet. The traffic profile that the meter uses to operate on the packet is also retrieved from the SLA of the packet as well as the current meter state. Metering is performed **per** SLA. That is, the subset application flows that belong to a particular SLA are metered together using the same traffic profile. After Metering is performed the packet is associated with a particular conformance level. Up to three conformance levels are defined for this implementation. The Meter also updates the new meter state with the packet's SLA.

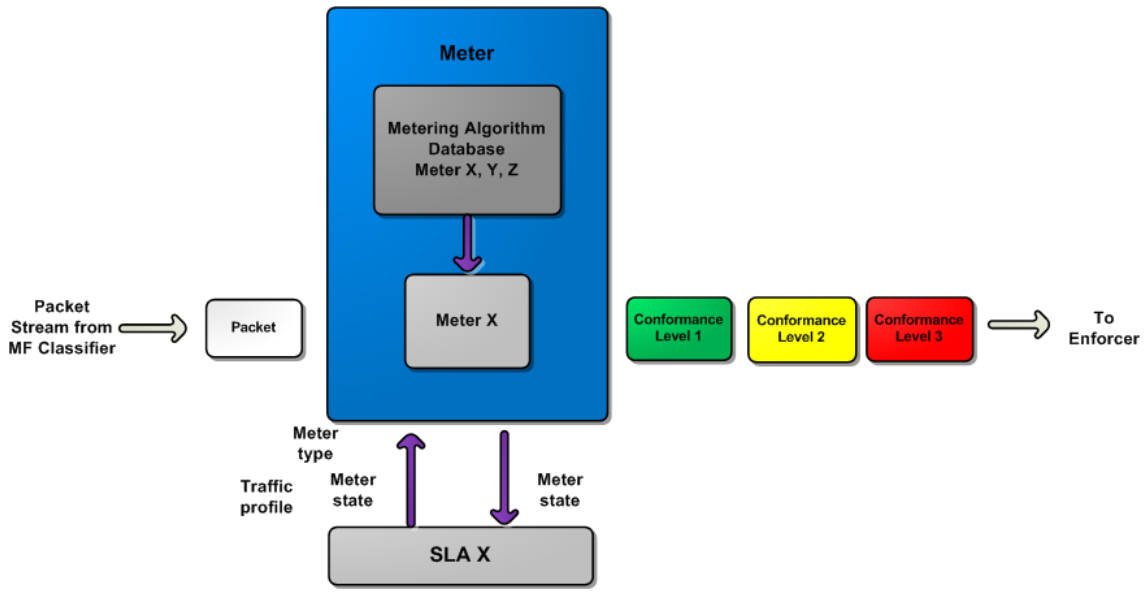

Figure 14: Edge Node – Meter

For this DiffServ implementation the following three metering algorithms will be implemented:

i. Token Bucket meter

ii. Single Rate Three Color Marker (srTCM) meter

iii. Two Rate Three Color Marker (trTCM) meter

The Token Bucket meter uses a traffic profile consisting of a CIR and a CBS and produces one of two conformance levels; Conformant and Non Conformant. A packet is conformant if it within the subscribed profile and non-conformant is it exceeds the subscribed profile.

The srTCM meter uses a traffic profile consisting of a CIR, CBS and EBS and produces one of three conformance levels; Conformant-Green, Non Conformant-Yellow and Non Conformant-Red. The trTCM meter uses a traffic profile consisting of a CIR, CBS, PIR,

and PBS and produces the same conformance levels as the srTCM meter. Conformant-Green is used for packets that are within their subscribed profile i.e. conformant. Non Conformant-Yellow is used for packets that are outside the subscribed profile up to a certain point i.e. non-conformant up to a point. Non Conformant-Red is used for packets that have crossed the point of Non Conformant-Yellow. A more detailed description of the operation of these meters can be found in the Appendix C.

3.1.3.3 Enforcer –Conformant and Non Conformant actions

For each conformance level the meter produces, a specific action is taken. These actions are carried out by the Enforcer (Figure 15) and are specified in the packet's SLA. If the packet is identified as Conformance Level 1, the packet is marked with the code point of the forwarding class specified in its SLA. If the packet is identified as Conformance Level 2, the corresponding non conformant action specified in the SLA for this conformance level is performed. Similarly if the packet is identified as Conformance Level 3, the corresponding non-conformant action specified in the packet's SLA is performed. For this DiffServ implementation, non conforming actions include; marking the packet with an inferior code point and dropping the packet. An example of such a configuration is shown in Table 1 below.

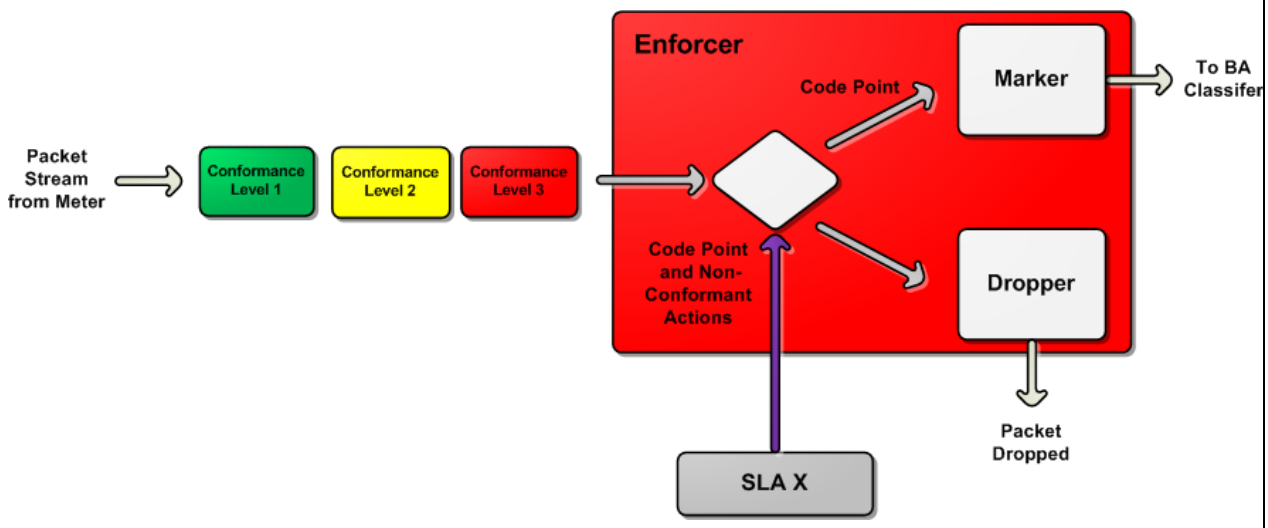

Figure 15: Edge Node – Enforcer

Table 1: Meter Conformance levels and corresponding Enforcer actions example

Meter Conformance Level	Enforcer action specified in SLA
1	mark packet with code point X
2	mark packet with code point Y
3	drop packet

Combining the MF classifier, the Meter and Enforcer components into one system produces the following setup in the Edge node (Figure 16).

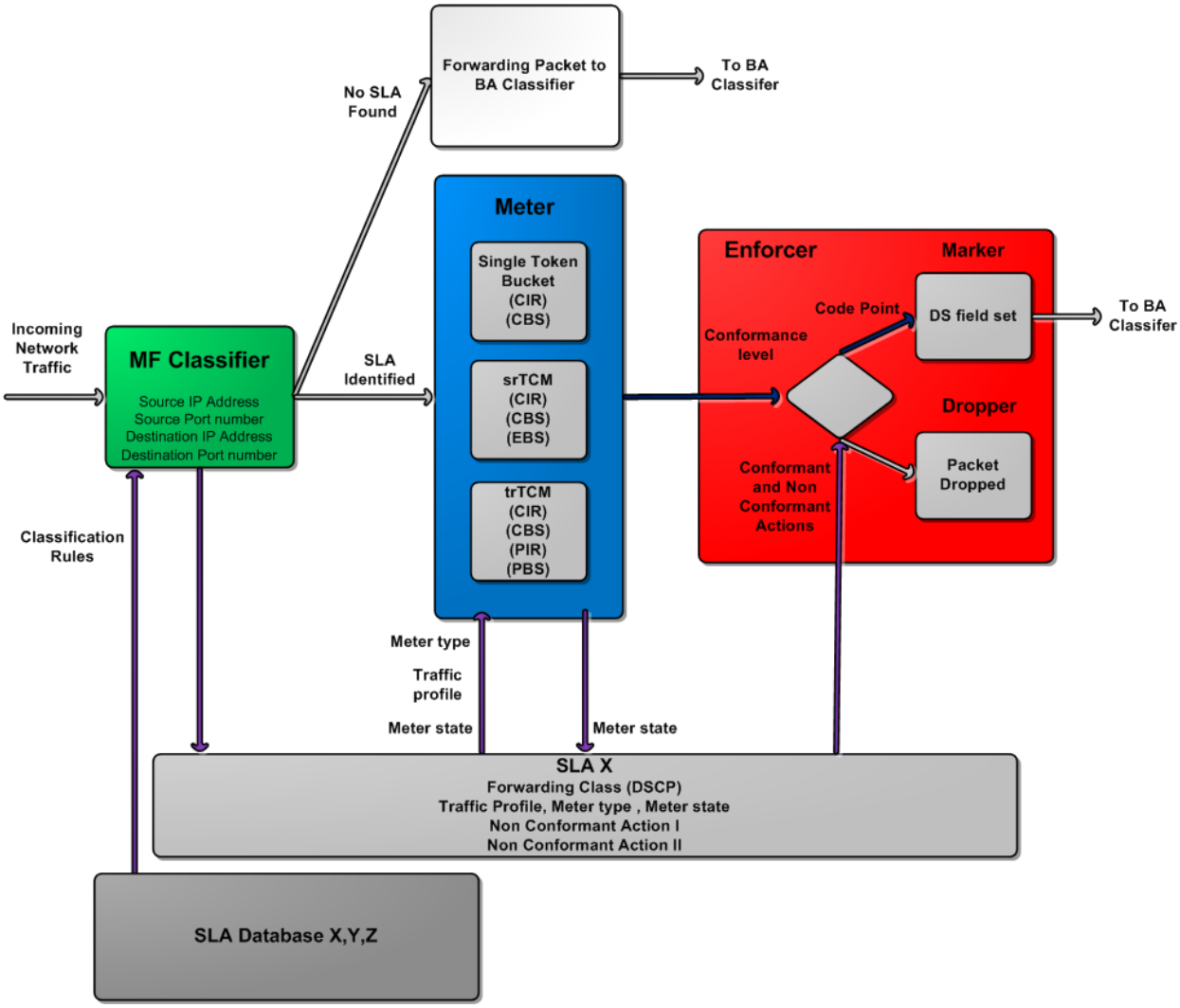

Figure 16: Edge Node Design

3.2 Network simulation and the structure of the ns-3 simulator

3.2.1 What is Simulation?

Simulation is the process of imitating the operation a real world system over time. A simulation model is developed to study the behavior of a system over time and is based on a set of assumptions about the system. The assumptions about the system are expressed in terms of the relationship between the objects of interest of the system. The simulation model can be used to investigate a real system or a system in the design stage, before it is implemented. Therefore simulation can be used as both an analysis or design tool. Systems may be categorized as discrete or continuous. Discrete systems are those in which the state variables change at discrete points in time, while continuous systems have state variables that change continuously.

Therefore discrete event simulation is the modeling of systems in which the state variables change at discrete points in time. The simulation models are analyzed numerically rather than analytically. Analytical methods attempt to solve the model using mathematics where as numerical methods use computational procedures. Models that are analyzed using numerical methods are run rather than solved. Observations are collected from the simulation and are analyzed to approximate the behavior of the system. Such simulations are usually done through the use of computers. (Banks, CarsonII and Nelson 2010)

3.2.2 The ns-3 simulator

The ns-3 simulator is such a discrete event simulator that models IP networks. The simulator is open source and is written and scripted in the C++ programming language. Although it was required that ns-3 be used as the simulator for this project, it does offer a number of new features. It is able to generate Pcap trace files for each node in the simulated network and support multiple levels of logging. Also new features can easily be added which makes it suitable for the purpose of this project. Ns-3 is also one of the fastest and efficient network simulators available. Within ns-3, the simulation time discretely moves from one event to another. The simulator keeps a list of events, from which it chooses and executes. Events are scheduled at a specific simulation time where it will wait until that time to be executed. The method 'Simulation::Run' gets the first event started. To perform a simulation, the network topology is first setup and configured with the desired attributes such as link bandwidths, propagation delays and traffic sources. After which the simulation is run and the results are analyzed.

3.2.3 NS-3 Main Components

An ns-3 simulation consists of the following main components:

3.2.3.1 Node

In an IP network, the end systems or hosts transmit and receive data in the form of packets. In the core of the network, network routers forward packets from the source to destination. In the ns3 simulator, both end systems and network routers are

represented by the Node class. The Node class represents a computing device to which functionality is then added, such as applications, protocol stacks and network devices.

3.2.3.2 Applications

Network applications generate the packets that end systems transmit and receive. These applications are represented in ns3 by the Applications class. All specialized versions of applications, such as ns-3 on-off and UDP echo applications, are sub classes of this class.

3.2.3.3 Protocols

Network protocols define the type and order of the messages exchanged between the network nodes. They also define the actions that occur when a message is received. These protocols are implemented in ns-3 using their own class such as UdpL4Protocol and TcpL4Protocol. Protocols in ns-3 are usually implemented in a stack. Packets that are generated by an ns-3 application will interact with ns-3 protocols as it moves down the protocol stack.

3.2.3.4 Channel

Network nodes connect to each other via communication links. Ns-3 Nodes communicate with each other via the Channel class. All specialized versions of channels such as point to point, Wi-Fi and CSMA are sub classes of this class.

3.2.3.5 Net-device

To allow end systems to connect to the network, a peripheral card known as a Network interface card (NIC) is needed. The ns-3 NetDevice class represents this piece

of hardware as well as the software drivers used to control it. A NetDevice is installed on a Node to allow the Node to communicate with other Nodes in the network. All specialized versions of NetDevices such as point to point, Wi-Fi and CSMA are sub classes of this class. Note that a specialized version of a NetDevice can only work with its corresponding specialized channel. For example a point to point NetDevice can only work with a point to point channel. Also note that a single Node may contain multiple NetDevices that connect to multiple channels

3.2.3.6 Queue

In an IP network, queues are memory locations within a node that a packet waits until transmitted. Queues have a maximum size which if exceeded will cause incoming packets to be dropped (drop-tail queue). Queues are represented in ns-3 by the Queue class. All specialized versions of queues are sub classes of this class. Every ns-3 NetDevice owns an ns-3 Queue. The structure of the ns-3 Queue will be discussed in the following section.

3.2.3.7 Packet

Data is transferred throughout the network in the form of packets. The ns-3 Packet class represents network packets in the simulation. Each packet object is associated with a set of protocol headers, byte tags, packet tags, and a byte buffer. The tags can be used to store extra information associated with the packet such as a flow identifier. Packets are created by ns-3 applications, move down the protocol stack where they receive protocol headers and passed to the node's NetDevice. The packet is stored within the Net Device's transmit queue where it waits to be transmitted onto the

outgoing channel. The nodes within the network then route the packet to the destination node. Figure 17 below shows the how these main components are organized within a node.

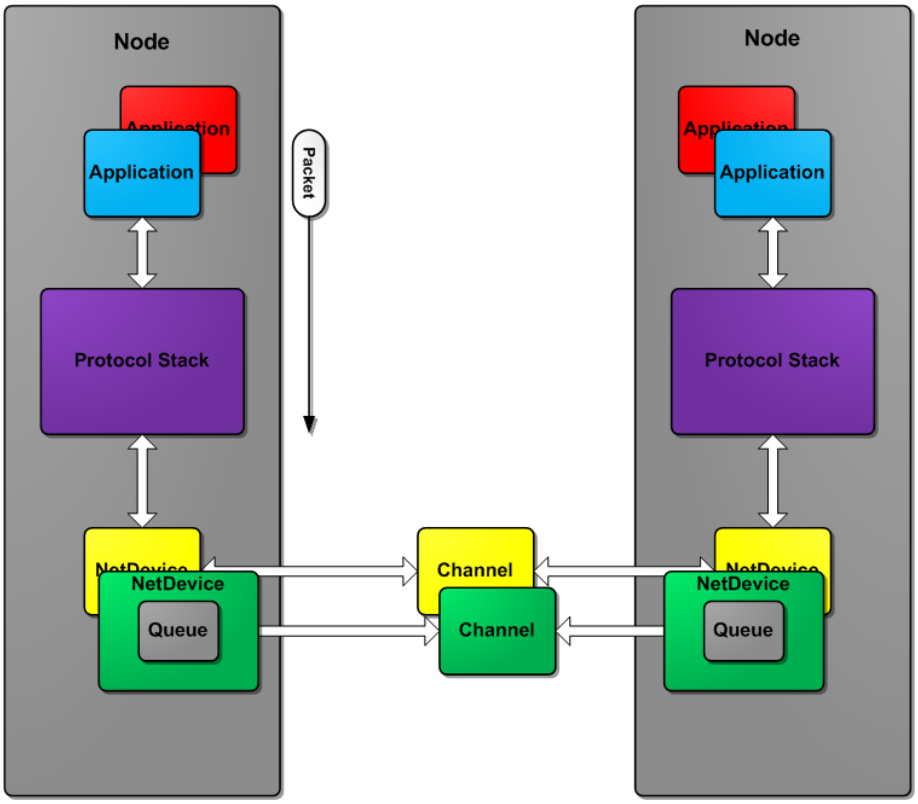

Figure 17: ns-3 main components

Source: (Riley 2010)

3.2.4 Choosing a location to implement the DiffServ architecture

When deciding on a location to implement the DiffServ components, a number of factors were considered. When choosing a location it was desirable to:

i. To not change the original ns3 source code but to add functionality to it instead

ii. To implement the DiffServ architecture without affecting how the original system operates

iii. To implement the components in a location that allows them to carry out their functions as simple as possible

iv. To choose a location that allows for the individual components to be easily installed on a node and installed on any node the user desires.

Considering these factors, the location deemed most suitable was the transmit queues of the Network Devices. A transmit queue is a place where all packets in a network must transverse which makes it suitable for this application. Different queues can be installed on any node and NetDevice as the user desires and does not change the original operation of the simulation. Queues allow for operations such as packet dropping without affecting the system since these operations typically occur in a queue. Lastly by making new a queue class the original ns-3 source code can remain unchanged while the DiffServ architecture is implemented.

3.2.5 The structure of the ns-3 Queue

The ns-3 Queue class has two main virtual methods; DoEnqueue and DoDequeue (Figure 18). The DoEnqueue method is called by the interface to place a packet onto the queue for transmission. The method returns 'true' if the enqueue was successful and false if unsuccessful (the packet was dropped). The DoDequeue method is called to remove a packet from the queue and transmit it on the channel. The method returns the packet pointer if successful and '0' if unsuccessful (the queue was empty).

From the perspective of the simulator, the DoEnqueue method only returns true and false to indicate packet enqueue and packet drop. The rest of the simulator is not aware of the actions that take place inside of the queue itself. Therefore as long as the queue performs the required methods, it is possible to implement a wide variety of components within the queue such as classifiers, active queue management and scheduling schemes.

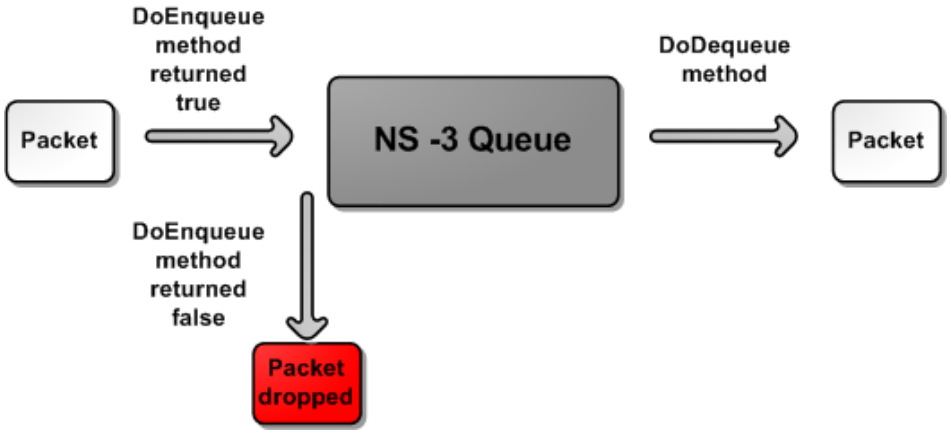

Figure 18: NS-3 Queue

The designed DiffServ architecture was implemented as a sub class of the ns-3 Queue class called DiffServQueue. The DiffServQueue implements the required DoEnqueue and DoDequeue functions so that it interacts with the interface. However within these methods the DiffServ components were implemented. Therefore when a packet is enqueued and dequeued onto and from a DiffServQueue, the designed DiffServ processes will be internally carried out. The following implementation section explains this in detail.

3.3 Implementation of the architecture into the ns-3 simulator

For the implementation of the designed architecture, the following new classes were implemented. These classes were included into the simulator's source files to create the DiffServ enabled network.

i. DiffServQueue (Main Class)

ii. DiffServFlow

iii. DiffServSla

iv. DiffServMeter (Abstract Class)

v. TokenBucket (DiffServMeter sub class)

vi. srTCM (DiffServMeter sub class)

vii. trTCM (DiffServMeter sub class)

viii. DiffServAQM (Abstract Class)

ix. WRED (DiffServAQM sub class)

3.3.1 Class DiffServFlow and DiffServSla

These two classes comprise the SLA database. They contain all the information for a customer; the flows that belong to a customer, the traffic profile for the meter, the metering algorithm that should be used and the conformant and non conformant actions that should be performed on packets. These classes interact with the Edge DiffServQueue by providing the Edge DiffServQueue with the information it needs in order to classify and condition traffic entering the network.

Class DiffServFlow contains the information required to identify a flow; this includes source IP address, destination IP address, source port number and destination port number. It also contains a flow identifier to distinguish it from other flows. Since an application flow is associated with a customer SLA, the DiffServFlow class has a DiffServSla pointer as a data member as well. This allows a flow to be 'tied' to a particular SLA. Also since many flows may be under the same SLA; many different flow objects may contain the same DiffServSla pointer data member. In this way, all the flows that belong to a customer will be associated to the customer's SLA.

Class DiffServSla contains two data structures, ConformanceSpec and MeterSpec (Figure 19) MeterSpec contains the traffic profile information such as CIR and PIR that is used by the Meter. It also keeps the meter's state variables such as the current size of the committed bucket and the arrival time of the last packet. The MeterSpec also contains a meter identifier that is used to select a meter from the metering algorithm database. ConformanceSpec contains the forwarding class DSCP and non conformant actions that are used by the Enforcer; labeled as initial code point, nonConformantAction I and II respectively.

```
struct ConformanceSpec{
int initialCodePoint;
int nonConformantActionI;
int nonConformantActionII;
 };

struct MeterSpec{
string meterID;
int   cIR;
int   cBS;
int   eBS;
int   pIR;
int   pBS;
float lastPacketArrivalTime;
float committedBucketSize;
float extraBucketSize;
float peakBucketSize;
};
```

Figure 19: DiffServSla MeterSpec and ConformanceSpec

Both the DiffServFlow and DiffServSla classes are equipped with Get and Set functions to enable the user to specify the parameters and to enable the traffic conditioning components to retrieve and re-write the information stored (Figure 20).

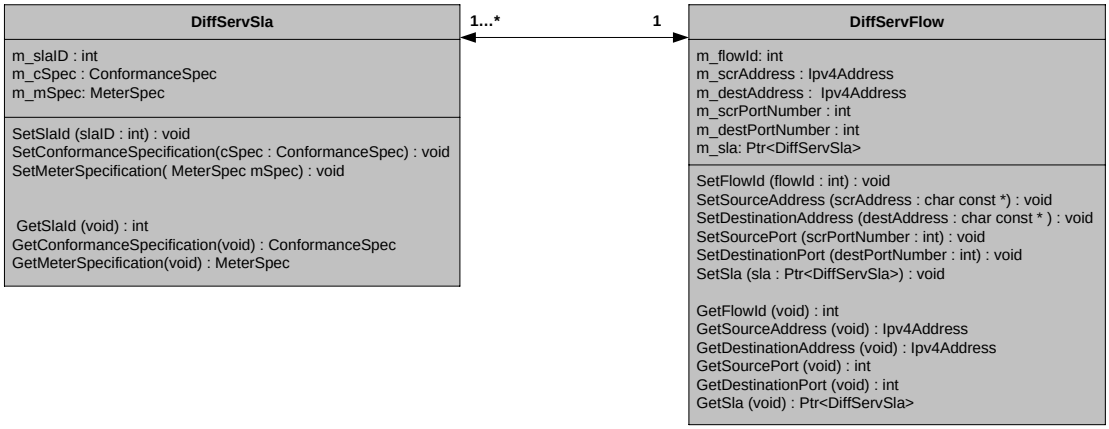

Figure 20: DiffServFlow and DiffServSla class diagram

3.3.2 DiffServSla and DiffServFlow setup – creating the SLA database

To create the SLA database, DiffServFlow and DiffServSla objects and their pointers are first created and configured with the desired information. DiffServFlow objects are also configured with their respective DiffServSla pointer. All DiffServFlow pointers are then inserted onto a vector of type DiffServFlow. The vector is then linked to all DiffServQueues via a static DiffServQueue method called 'SetDiffServFlows' (Figure 21). To make the process of setting the data members easier, the constructors of both the DiffServFlow and DiffServSla classes were designed to take the values of each data member as a parameter. Note that this method of setting the data members is optional and the user may call on each individual 'Set' member function instead.

```
//Flow Setup

Ptr<DiffServFlow> flow1Ptr = CreateObject<DiffServFlow>

(001,"10.0.0.1","192.168.2.2",0,5111);

//Conformance and Meter Specs
ConformanceSpec cSpec1;
cSpec1.initialCodePoint = AF11;
cSpec1.nonConformantActionI = AF12;
cSpec1.nonConformantActionII = drop;

MeterSpec mSpec1;
mSpec1.meterID = "SRTCM";
mSpec1.cIR = 10273437;
mSpec1.cBS = 1500;
mSpec1.eBS = 0;
mSpec1.pIR = 1500000;
mSpec1.pBS = 1500;

//SLA Setup

Ptr<DiffServSla> sla1Ptr = CreateObject<DiffServSla>(001,cSpec1,mSpec1);
flow1Ptr->SetSla(sla1Ptr);

vector< Ptr<DiffServFlow> > myFlowVector;
myFlowVector.push_back(flow1Ptr);

DiffServQueue::SetDiffServFlows(myFlowVector);
```

Figure 21: DiffServSla and DiffServFlow Setup for creating the SLA database

3.3.3 Class DiffServMeter

Class DiffServMeter is an abstract class that metering algorithms inherit from and comprises the metering algorithm database. Classes TokenBucket, SRTCM and TRTCM are all sub classes of DiffServMeter (Figure 22). DiffServMeter interacts with Edge DiffServQueues to condition packets that are about to enter the network. It provides the Edge DiffServQueue with the conformance level of a packet according to a specified profile in a DiffServSla.

DiffServMeter contains a single virtual function called 'MeterPacket' which takes a packet and DiffServSla pointer as parameters. This provides the meter sub classes with the required traffic profile and current meter state to operate on the packet. All DiffServMeter sub classes contain a unique identifier. Each DiffServMeter sub class returns an integer representing the conformance level of the packet. Integers 1, 2 and 3 are used to represent conformance levels 1, 2, and 3 respectively. The conformance level returned to the DiffServQueue is then used with the DiffServSla's ConformanceSpec to decide on what action to take. Meters must be available for the Edge DiffServQueues to use otherwise no metering can be performed and the metering stage will be skipped. Should this occur, all packets will be considered conformant and will be marked with their specified code point.

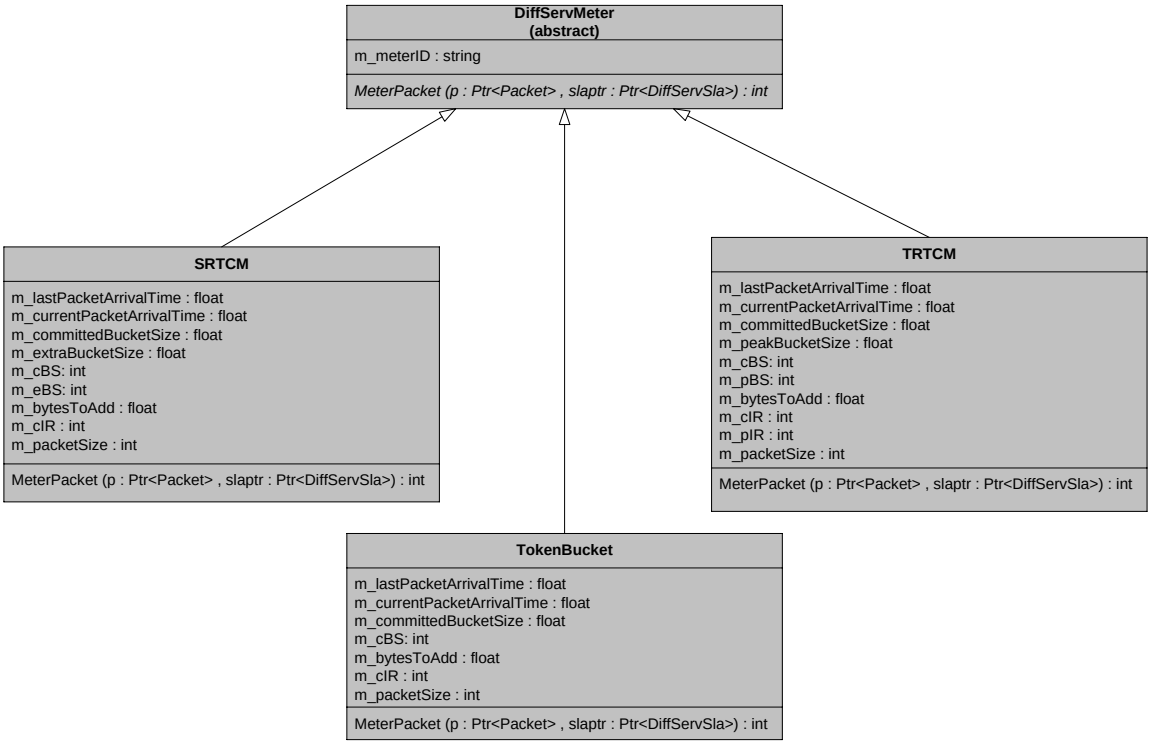

Figure 22: DiffServMeter inheritance diagram

3.3.4 DiffServMeter setup

To create the metering algorithm database, the meters and their pointers are first created. Next, the pointers are then pushed onto a vector of type DiffServMeter. The vector is then linked to all DiffServQueues via a static DiffServQueue method called 'SetDiffServMeters' (Figure 23). Class DiffServQueue contains a static vector of DiffServMeter pointers that is configured through this method. To choose a meter for a customer's SLA, the SLA's MeterSpec data member, meterID, must be set to the corresponding meterID of the desired meter (e.g. token bucket). Figure 21 above illustrates this.

```
//Meters setup

Ptr<DiffServMeter> STB = Create<TokenBucket>();
Ptr<DiffServMeter> srTCM = Create<SRTCM>();
Ptr<DiffServMeter> trTCM = Create<TRTCM>();

vector< Ptr<DiffServMeter> > myMeterVector;
myMeterVector.push_back(STB);
myMeterVector.push_back(srTCM);
myMeterVector.push_back(trTCM);

DiffServQueue::SetDiffServMeters(myMeterVector);
```

Figure 23: DiffServMeter Setup

3.3.5 Class DiffServAQM

Class DiffServAQM is the abstract class for Active Queue Management algorithms. Class WRED (Weighted Random Early Detection) is a sub class of class DiffServAQM (Figure 24). Class DiffServAQM interacts with both Edge and Core DiffServQueues. DiffServAQM sub classes internally carry out their respective AQM algorithms and inform the DiffServQueue when packet drops should occur. Each DiffServAQM sub class internally services all four AF queues of the DiffServQueue. Each DiffServAQM sub class also contains a unique identifier similar to DiffServMeter.

Class DiffServAQM contains a single virtual function called 'DoAQM'. This function takes two parameters; the current queue size of an AF queue and the DS field of the packet. The function returns a Boolean value (i.e. Enqueue or Drop). Class WRED implements this function using the WRED algorithm.

Each DiffServQueue owns a vector of DiffServAQM pointers that is configured using the DiffServQueue method 'SetDiffservAQM' which takes a vector of DiffServAQM pointers as a parameter. A vector of DiffServAQM objects is used, as opposed to a

single DiffServAQM object, so that the user has the option of choosing different DiffServAQM mechanisms to be applied to each AF queue. The user specifies, using the DiffServAQM identifier, which AQM mechanism is to be applied to each AF queue through the 'SetDiffservAQM' method as well.

This vector, unlike the SLA and metering databases, is not static. Each DiffServQueue is associated with its own vector of DiffServAQM pointers. The DiffServAQM vector cannot be static since DiffServAQM objects keep state variables such as average queue size and therefore belongs to a particular DiffServQueue exclusively. Additionally, if the user does not configure any DiffServAQM objects for a DiffServQueue via this method, then the DiffServQueue will apply drop tail buffer management by default.

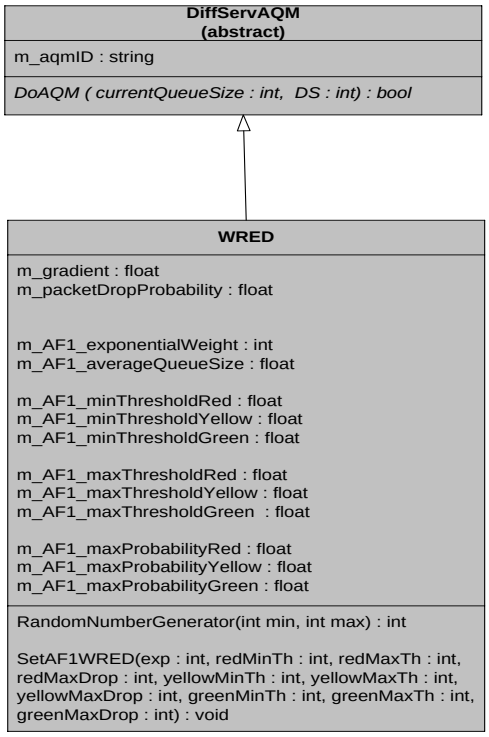

Figure 24: DiffServAQM inheritance diagram

3.3.6 DiffServAQM setup

The AQM objects are first created by the user, configured, pushed onto a vector of type DiffServAQM, and then set inside the DiffServQueue using the 'SetDiffServAQM' method. The user can also specify using the DiffServAQM identifier which DiffServAQM mechanism contained in the vector should be applied to each AF queue via the 'SetDiffServAQM' (Figure 25).

```
//AQM setup

Ptr<WRED> AQM1Ptr = Create<WRED>();

vector< Ptr<DiffServAQM> > myAQMVector;
myAQMVector.push_back(AQM1Ptr);

q1->SetDiffServAQM (myAQMVector,"WRED","WRED","WRED","WRED");

AQM1Ptr->SetAF1WRED(7, 10,50,100,  10,100,60,  10,100,50);
AQM1Ptr->SetAF2WRED(7, 10,50,100,  10,100,60,  10,100,50);
AQM1Ptr->SetAF3WRED(7, 10,50,100,  10,100,60,  10,100,50);
AQM1Ptr->SetAF4WRED(7, 10,50,100,  10,100,60,  10,100,50);
```

Figure 25: DiffServAQM setup

3.3.7 Class DiffServQueue

Class DiffServQueue is the main class of the architecture implementation. It inherits from ns-3 class Queue and therefore implements the required virtual functions of class Queue; DoEnqueue and DoDequeue. DiffServQueue queue performs the PHB based forwarding of the designed Core Node as well as the traffic classification and conditioning functions of the Edge Node. Therefore DiffServQueue operates in two modes; Edge DiffServQueue and Core DiffServQueue. By default, the DiffServQueue is in Core mode. To switch to Edge mode and hence enable traffic classification and conditioning, the DiffServQueue method 'SetQueueMode' must be called. Figure 26

below shows how the components within the DiffServQueue interact with each other and describe the order of operations that are carried out for both Edge and Core modes.

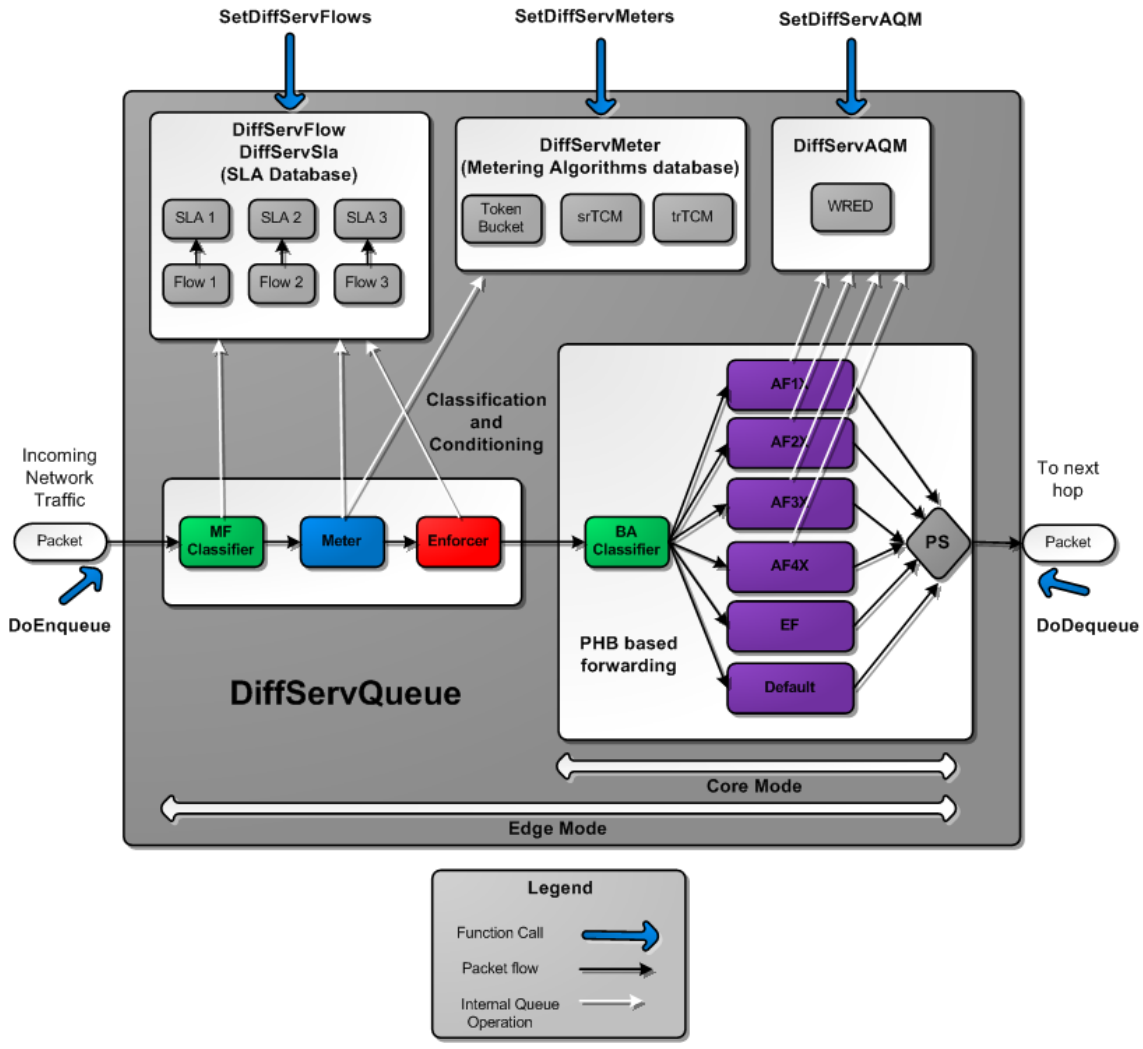

Figure 26: DiffServQueue internal operation

3.3.7.1 Edge Mode

As explained in the Edge Node design, the traffic classification and conditioning process has three main stages; MF-classification, Metering and Enforcing. These tasks are carried out internally by the DiffServQueue method 'ClassifyAndCondition'. This

function interacts with the SLA database and metering algorithm database that were configured for DiffServQueues when the network topology was set up. 'ClassifyAndCondition' is called at the start of the virtual DoEnqueue method when a packet arrives at the queue. Therefore all classification, metering, marking and dropping is performed before a packet is enqueued. As previously explained, all Edge DiffServQueues interact with the same SLA database and metering algorithms since they are static. After a packet has been conditioned, it must now be enqueued. The DiffServQueue contains a queue for each forwarding class. The EF class, each AF class, and the default class have their own queue; a total of six individual queues. Theses queues are implemented internally using standard C++ queues.

When the virtual DoEnqueue function is called by the interface it must access one of the internal queues implemented and enqueue a packet. Similarly when the DoDequeue function is called it must access one of the internal queues, remove a packet and transmit it on the channel. To accomplish this, six new Enqueue and Dequeue functions were created, that accesses each individual internal queue of the DiffServQueue. The Enqueue functions are called from within the virtual DoEnqueue function. Similarly the individual Dequeue functions are called from within the DoDequeue function.

The particular internal queue Enqueue function that is called in the virtual DoEnqueue function will depend on the DSCP of the packet. The choosing mechanism in the DoEnqueue function is implemented in the BA classifier function. Recall that the

purpose of the BA classifier is to identify the DSCP of the packet and enqueue the packet onto the queue that corresponds to the identified DSCP. Therefore the BA classifier function is called in the DoEnqueue function to identify the DSCP of the packet and call the appropriate Enqueue function. The internal queue Enqueue function called would then enqueue the packet onto its queue. The BA Classifier is called in the DoEnqueue function after a packet has been conditioned. Within each Enqueue function of the four AF queues, a WRED object is called by the DiffServQueue to determine if the packet should be enqueued.

The particular Dequeue function that is called when the interface calls the DoDequeue function will be decided using the priority queue and WRR scheduling algorithms which are implemented within the DoDequeue function.

Class DiffServQueue also has a number of data members that may be configured by the user. These include the Weighted Round Robin queue weights, queue sizes and EF rate limiter profile. These are configured through 'Set' member functions of the DiffServQueue class that the user calls in the main script. DiffServQueues must be configured with the SLA and metering algorithm databases and its own AQM algorithms. The following methods of Class DiffServQueue perform these tasks:

1. Static DiffServQueue :: SetDiffServFlows
2. Static DiffServQueue::SetDiffServMeters
3. DiffServQueue ::SetDiffServAQM

An example of a network topology configuration is shown in Figure 27 below. In the DiffServ enabled network, a DiffServQueue should be installed on all nodes that belong to the service provider. DiffServQueues' at the network boundary should be configured into Edge Mode and interior DiffServQueues' should be configured into Core Mode.

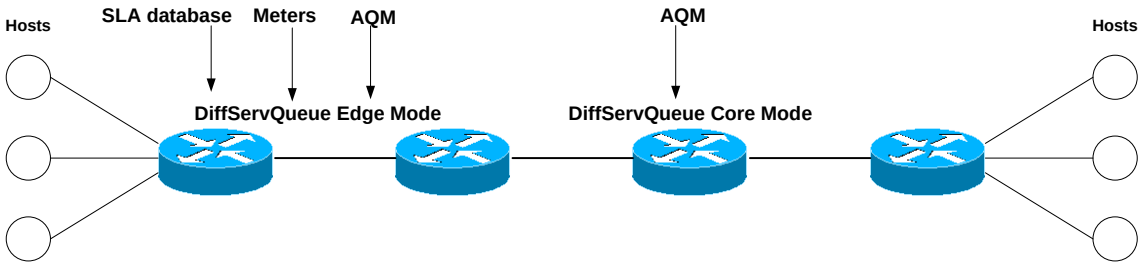

Figure 27: DiffServ enabled network topology configuration

The association diagram of the implemented classes is shown in Figure 28 below.

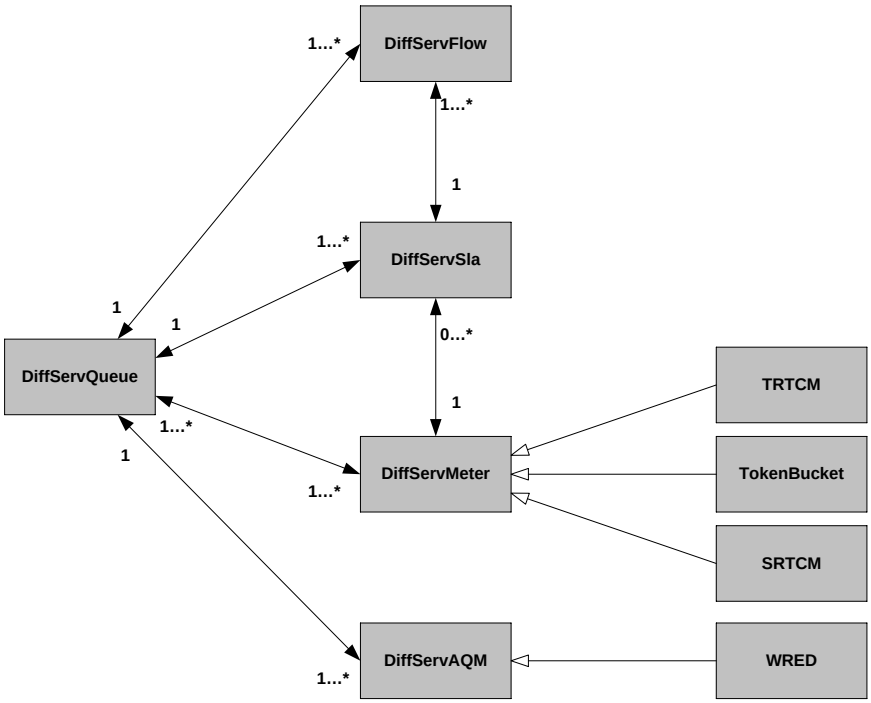

Figure 28: Implemented classes – association diagram

3.3.7.2 Core Mode

Core Mode is identical to Edge Mode but without the traffic classification and conditioning functionality. DiffServQueue skips the traffic classification and conditioning stage and goes directly to the BA classifier to enqueue the packet into a queue as shown in Figure 26 above.

3.3.8 Edge DiffServQueue Operation

In this section, the operations associated with each step of the Edge DiffServQueue will be explained with some sample code provided. The Core DiffServQueue operation is a subset of the Edge and hence need not be explained. The DiffServQueue has two main functions as all ns-3 queues; the DoEnqueue and DoDequeue virtual functions that are called by the interface.

3.3.8.1 DiffServQueue - DoEnqueue function

The DoEnqueue function contains two function calls; the ClassifyAndCondition and BA Classifier function. These are explained in the following sections.

```
DiffServQueue::DoEnqueue (Ptr<Packet> p)
{
if( (m_queueMode == "Edge") && (m_enable
{
 ClassifyAndCondition(p);

if (m_drop == true)
    {
      Drop (p);
      return false;
    }
}
  return BA_Classifier(p);
```

Figure 29: DiffServQueue – DoEnqueue

3.3.8.2 ClassifyAndCondition function

The Edge DiffServQueue begins at the DoEnqueue method where the ClassifyAndCondition function is called. The ClassifyAndCondition function consists of the following three steps:

i. MF Classification

ii. Metering

iii. SLA Enforcement

3.3.8.2.1 Step 1 -MF Classification

The MF classifier algorithm first retrieves four packet header fields from the packet; IP source and destination address and source and destination port numbers. The IP

addresses are located in the IP header and the port numbers are located in the transport protocol header. Both these headers must be obtained from the packet.

This is accomplished by first creating a copy of the packet and then removing the IP and transport headers from the packet copy. The IP header is represented in ns-3 by class IPv4Header and the UDP and TCP headers are represented by classes UdpHeader and TcpHeader respectively. The IPv4Header and transport header classes are equipped with Get and Set functions, which enable the simulator to set and retrieve each header field. The Get functions are used in this case to retrieve the information on the four fields stated above. The values of the fields are stored in four local variables.

```
Ipv4Address scrAddressPacket = ipv4Header.GetSource();
Ipv4Address destAddressPacket = ipv4Header.GetDestination();
int protocolNumber = ipv4Header.GetProtocol();

if (protocolNumber == 17)
{
 q->RemoveHeader (udpHeader);
 scrPortPacket = udpHeader.GetSourcePort();
 destPortPacket = udpHeader.GetDestinationPort();
}
```

Figure 30: DiffServQueue-MF classifier step 1

The MF classifier then accesses the vector of DiffServFlow object pointers (SLA database) that is stored in within the DiffServQueue; data member m_flowVector. It goes through each DiffServFlow object pointer with the use of a 'for' loop control structure; looking for a matching flow. This is performed by first using the Get

functions of the DiffServFlow class to retrieve the contents of a DiffServFlow object. The four fields retrieved from the DiffServFlow object are also stored in local variables.

After this initial step, the classifier does a comparison on the two sets of header fields obtained; header fields taken from the packet and those that were taken from the DiffServFlow object. If all fields match up, then a classification match is found and the packet's SLA is identified. The 'for' loop performs this process a number of times equal to the size of the vector; the number of DiffServFlow object pointers stored in the vector.

If a classification match is found, the classifier stores the pointer of the matching flow object so that it can be used later and moves on to the metering step. If no match is found, all traffic conditioning is skipped and control is returned to the DoEnqueue function to Enqueue the packet. Since the packet did not belong to a SLA, the packet remains in the default forwarding class

```
for (uint32_t i = 0; i < m_flowVector.size();i++)
{
Ipv4Address scrAddressFlow = (m_flowVector.at(i))->GetSourceAddress();
Ipv4Address destAddressFlow = (m_flowVector.at(i))->GetDestinationAddress();
int scrPortFlow = (m_flowVector.at(i))->GetSourcePort();
int destPortFlow = (m_flowVector.at(i))->GetDestinationPort();

if ((scrAddressPacket == scrAddressFlow) && (destAddressPacket == destAddres
&& (scrPortPacket == scrPortFlow) && (destPortPacket == destPortFlow))
  {
  m_flow =  m_flowVector.at(i);
  break;
```

Figure 31: DiffServQueue-MF classifier step 2

3.3.8.2.2 Step 2 – Metering

The Meter first checks the MeterSpec from the SLA that the packet belongs to and retrieves the meterID. It does this via the DiffServSla pointer data member within the identified DiffServFlow object. First a Get function of the DiffServFlow class is used to get the DiffServSla pointer. The DiffServSla pointer can then be used to get the MeterSpec data member.

The Meter then goes through the vector of metering algorithms that were set when the network was set up and retrieves the meterID of each meter in the vector. If the meterID of the SLA matches the meterID of a meter in the vector, then that meter algorithm is used.

The identified meter object is then called to meter the packet via the virtual function 'MeterPacket'. The meter takes the packet pointer and the DiffServSla pointer as parameters. The DiffServSla object contains the traffic profile and bucket state that the meter requires to operate on the packet. The meter then returns one of three integers, each representing a conformance level and proceeds to the next step. If no metering match was found, then the packet cannot be metered and is treated as conformant.

```
for (uint32_t j = 0; j < m_meterVector.size();j++)
{
meterID_MeterVector = ( m_meterVector.at(j) )->m_meterID;
if ( meterID_SLA == meterID_MeterVector)
  {
    m_meter = m_meterVector.at(j);
    m_meterIdentified = true;
    break;
  }
}//end loop
//If meter identified --Calling meter object to meter packet
 if ( m_meterIdentified == true)
 {m_conformance =  m_meter->MeterPacket(p, m_flow->GetSla() );}
```

Figure 32: DiffServQueue – Metering

3.3.8.2.3 Step 3 SLA Enforcement

After the packet has been metered, the actions specified in the SLA must now be enforced. The meter result is used together with the packet's DiffServSla ConformanceSpec to carry out a marking or dropping action (Figure 33). Each meter result or outcome maps to a Conformance spec data member (Table 2). If a marking action is performed, the DSCP is specified. The ConformanceSpec is retrieved in the same way as the MeterSpec.

Table 2 : Meter result and ConformanceSpec mappings

Meter Result (conformance level)	ConformanceSpec mapping
1	Initial code point (Marking)
2	Non conformant action I (Marking or Dropping)
3	Non conformant action II (Marking or Dropping)

```
if ( m_conformance == conformanceLevel_1)
{
 Marker(p, ( (m_flow->GetSla() )->m_cSpec).initialCodePoint );
}

if (m_conformance == conformanceLevel_2)
 {
  if ( ( (m_flow->GetSla() )->m_cSpec).nonConformantActionI == 256)
     {
       m_drop = true;
       return;
```

Figure 33: DiffServQueue Enforcer

3.3.8.2.4 Marker

The Marker function sets the DS fields of the packet to a particular value. The function takes two parameters; the packet pointer and the desired code point. Class IPv4Header represents the TOS field using an 'int' size variable. Therefore when encoding the DS field with an appropriate code-point, the DSCP is specified in decimal format.

```
//Marking the Tos field
ipv4HeaderNew.SetTos(DS);
```

Figure 34: DiffServQueue Marking

3.3.8.2.5 Dropper

The dropper is executed when the ClassifyAndCondition function ends and control returns to the DoEnqueue function (Figure 29). The dropper ensures that the queue's data member m_drop was set to 'true' by the Enforcer (Figure 33). The dropper returns the DoEnqueue function as 'false' to indicate the packet was dropped. The 'Drop' function that is seen before the 'return' is called does not drop packets. It is only used by the system to indicate packet drops.

3.3.8.3 BA Classifier function

The BA classifier function is now called after the ClassifyAndCondition function ends in the DoEnqueue function (Figure 29). It is called to choose one of the internal queue Enqueue functions. It first retrieves the DS field of the packet. It achieves this by first copying the packet and removing the IPv4 header from the packet copy. Class IPv4Header 'GetTos ()' function is used to retrieve the value of the DS field. The BA classifier then calls the appropriate internal queue Enqueue function corresponding to the value of the DS field (Figure 35). For example if the code point retrieved is AF11, AF12 or AF13, then the AF1 Enqueue function is called. If the code point does not map to an existing internal queue (e.g. DSCP = 100), the packet is enqueued onto the default queue.

```
  m_DS = ipv4Header.GetTos();

if ((m_DS == 40 ) || (m_DS == 48) || (m_DS== 56)   )
{
 return AF1_DoEnqueue (p);
}
```

Figure 35: DiffServQueue BA Classifier

3.3.8.4 AF Active Queue Management

Within each AF queue Enqueue function, a DiffServAQM object is called to determine if the packet should be enqueued. The particular DiffServAQM object would have been chosen when the network topology was first setup. The DiffServAQM object is called to carry out its virtual function 'DoAQM' to determine the action for the packet. If the

value returned is '0', then the packet is dropped. Otherwise the queue will 'attempt' to enqueue the packet. That is, if the queue is not full, the packet will be enqueued. This process is carried out for all AF enqueue functions. The non AF queues simply do a queue size check to determine if the packet should be dropped or enqueued (Drop tail buffer management).

```
if(m_AF1_AQMId != "DropTail")
{
 m_aqmOutput = m_AF1_AQMPtr->DoAQM(m_AF1queue.size (), m_DS);

 if(m_aqmOutput == 0)
 {
  Drop (p);
  return false;
  }
    else
    {
     if(m_AF1queue.size () >= m_AF1_maxPackets)
       { Drop (p);
        return false;
       }
       m_AF1queue.push(p);
       return true;
     }
```

Figure 36: DiffServQueue AF enqueue

3.3.8.5 DiffServQueue – DoDequeue function

The DoDequeue function is called by the interface when a packet needs to be dequeued. It is here within this function that the Weighted Round Robin and Priority Queue scheduling algorithms are implemented (Figure 37). These algorithms decide the order in which packets are dequeued from the internal queues. For the implementation of the WWR mechanism, each queue is given a specific weight (an integer). Each queue can only dequeue a number of times equal to its weight. The queue's weight is decremented by one when it dequeues a packet. When the queue's

weight is equal to zero, the queue can no longer dequeue any packets and must wait for its queue weight to be reset. When the queue weights are reset, they are set to their initial value. This value is configured by the user for each queue serviced by the WRR scheduler.

For the implementation of the priority queue for the EF PHB, a queue was simply configured with a weight that is not decremented, thereby allowing it to dequeue a number of times equal to the number of packets it contains. Therefore as long as the EF queue contains packets, it would always be able to dequeue a packet before the other queues.

```
if(!m_EFqueue.empty())
    {
     return EF_DoDequeue ();
    }

if(!m_AF1queue.empty())
    {
       if( m_AF1_weightTemp > 0)
         {
           m_AF1_weightTemp--;
           return AF1_DoDequeue ();
```

Figure 37: DiffServQueue -WRR and PQ scheduler

3.3.8.6 Resetting queue weights

The queue weights are rest when one of the following events occurs:

1. Each queue has been serviced a number of times equal to its weight; that is a queue has a packet to transmit and all queue weights are zero
2. A queue has packets to transmit, its weight is zero and no other queues contain any packets

```
if(!m_AF4queue.empty() || !m_AF3queue.empty() || !m_AF2queue.empty()

|| !m_AF1queue.empty()    || !m_BestEffortqueue.empty())

 { // NS_LOG_INFO ("RESET WEIGHTS");
   m_AF1_weightTemp = m_AF1_weight ;
   m_AF2_weightTemp = m_AF2_weight ;
   m_AF3_weightTemp = m_AF3_weight ;
   m_AF4_weightTemp =  m_AF4_weight ;
   m_BestEffort_weightTemp = m_BestEffort_weight ;
   goto loop;
```

Figure 38: DiffServQueue scheduler - resetting queue weights

3.4 IP Performance Metrics

The customer flows that transit the DiffServ network do so with varying levels of performance. This performance must be measurable when deciding how effective DiffServ is in applying Quality of Service to application flows. RFC 2680 (A One Way Packet Loss Metric), RFC 2679 (A One Way Delay Metric) and RFC 3393(IP Packet delay Variation Metric) were used for this purpose.

RFC 2680 proposes a singleton loss metric called 'Type-P-One-way-Packet-Loss' that is used to measure a single packet loss or transmission from host to destination. This metric is useful since if the end to end loss between hosts is large enough, some applications will become ineffective. RFC 2679 proposes a singleton delay metric called 'Type-P-One-way-Delay' that is used to measure a single end to end Delay for a packet from host to destination. The minimum value this metric produces gives an indication of transmission and propagation delays experienced in the network. RFC 3393 proposes a singleton delay metric called 'Type-P-One-way-ipdv' (IP Packet Delay Variation) that is used to define a single instance of an ipdv measurement. For a stream of packets between two measurement points, ipdv is defined of a pair of packets within the stream. The ipdv is the difference between the Type-P-One-way-Delay of each packet of the pair.

3.4.1 Metric Units

Type-P-One-way-Packet-Loss is either a '0' or a '1'. Type-P-One-way-Delay is either a real number (positive) or an undefined number of seconds. Type-P-One-way-ipdv is either a real number (positive or negative) or an undefined number of seconds

3.4.2 Methodology

1. At the source host a time stamp is placed in the packet and sent towards the destination
2. If the packet arrives at the destination host within a certain time threshold, a new time stamp is taken as soon as possible. By subtracting the two time stamps an estimate of One-way delay is calculated. The One-way packet-loss is taken to be zero
3. If the packet fails to arrive at the destination host or arrive within the time threshold, One-way packet-loss is taken to be one and One-way –delay is taken to be undefined. The specific time threshold is a parameter of the methodology
4. For each **pair** of packets, the Type-P-One-way-ipdv metric can be calculated by subtracting the One-way-Delay measurement of the second packet from the first.
5. If one or both One-way-Delay measurements are undefined for the packet pair, then the Type-P-One-way-ipdv measurement for that packet pair will be undefined.

3.4.3 Intended use for this project

For each application flow, the Type-P-One-way-Packet-Loss and Type-P-One-way-Delay metric will be determined for each packet the application sends. Type-P-One-way-ipdv will be determined for every pair of packets. This data in whole will form a sample from which statistics can be derived.

3.4.4 Statistics

Using the sample collected for an application flow, statistics can now be derived from this sample. RFC 2680, 2679 and 3393 proposes the following statistics:

3.4.4.1 Type-P-One-way-Packet-Loss-Average

Using a sample, the Type-P-One-way-Packet-Loss-Average is the average of all Loss values in the sample. For example, given a sample (0, 0, 0, 0, 1), the average loss would be 0.2.

3.4.4.2 Type-P-One-way-Delay-Percentile

Using a sample and a percent 'X' (between 0 and 100), Type-P-One-way-Delay-Percentile is the X^{th} percentile of all the values in the sample. RFC 2330 defines percentile as follows.

The 'empirical distribution function' is a function F(x) of a set of scalar measurements. For any 'x' value, the function gives the fraction of the total number of measurements that are less than or equal to 'x'. If x is greater than the maximum value, then the F(x) = 1. If x is smaller than the minimum value, then the F(x) = 0.

Percentile is now defined as the smallest value of 'x' for which F(x) is greater than or equal to a given percentage.

Example:

Given the following sample (-2, 7, 4, 7, 18, -5)

F (-5) = 1/6

F (7) = 5/6

F (200) = 1

F (4) = 3/6

Since F (4) = 0.5, then the measurement '4' is the 50^{th} percentile.

3.4.4.3 Type-P-One-way-ipdv-percentile

Using a sample and a percent 'X' (between 0 and 100), Type-P-One-way-ipdv-Percentile is the X^{th} percentile of all the values in the sample. Percentile is calculated as described previously.

3.5 Statistics Collection – Class StatCollector

Now that the metrics for evaluating the performance of application flows have been defined, the actual system that will perform the previously stated metric methodologies must be implemented. This system was implemented in class StatCollector. Class StatCollector collects the loss, delay and delay variation metrics for each application flow. In addition to collecting end to end statistics, the class also stores information regarding each internal queue of each DiffServQueue created. For each internal queue, the packet enqueue/dequeue times, loss, queuing delay and average queue size is computed. Class StatCollector is a static class; all data members and member functions are static. The class needs to be static so that all other objects 'report' to a single entity.

3.5.1 End to End statistics – loss, delay, and delay variation

To perform this task, StatCollector first classifies each new application flow and assigns it a unique flow identifier. When a new flow is detected, the information related to that flow; the IP addresses and port numbers are stored and associated with a flow identifier. This information is stored in the 'FourTuple' data structure shown below (Figure 39).

```
struct FourTuple{

Ipv4Address scrAddress;
Ipv4Address destAddress;
int scrPort;
int destPort;
};
```

Figure 39: StatCollector data structure – FourTuple

The number of packets each flow sends is also stored (the packet sequence). Class StatCollector performs end to end statistics collection by 'connecting' itself to the transport layer protocols. Therefore when a packet is generated by an application and moves down the protocol stack, StatCollector is made aware of the packet and its header fields. Similarly when a packet is received at the destination node and moves up the protocol stack. The processes that occur at both sending and receiving sides of the transport layer protocols are described as follows.

3.5.1.1 Sending End

On the sending end of the transport protocol, StatCollector will receive the packet pointer, IP addresses and port numbers. This information will be used to classify the packet. The flow the packet belongs to is either unclassified and therefore given a flow identifier or was already classified and the flow identifier is known. When a packet is transferred, the packet carries with it certain information that will be needed by the receiving side to identify the packet. This information is the flow and sequence identifier. To 'attach' this information to the packet, 'Tags' are used. Associated with each packet object is a set of 'Packet Tags'. These tags do not increase the size of the packet but are only used to add extra information. The Tags used to store the flow and sequence identifiers are classes FlowIdTag and SeqIdTag respectively. Class FlowIdTag belongs to ns-3. SeqIdTag was created using FlowIdTag as a reference. At the time of packet transmission, flow and sequence identifier tags are created, configured, and attached to the packet. The snippet of code below (Figure 40) shows how this is performed.

```
FlowIdTag flowTag;
flowTag.SetFlowId(m_flowId);
packet->AddPacketTag (flowTag);
NS_LOG_INFO("Tx FlowIdTag: " << flowTag.GetFlowId());

m_seqId = m_sequenceVector.at(m_flowId);
m_seqId = m_seqId +1 ;
m_sequenceVector.at(m_flowId) = m_seqId;


SeqIdTag seqTag;
seqTag.SetSeqId(m_seqId);
packet->AddPacketTag (seqTag);
NS_LOG_INFO("Tx SeqIdTag: " << seqTag.GetSeqId());
```

Figure 40: StatCollector – creating, configuring, and attaching packet tags

For each packet sent, StatCollector creates a container that stores data associated with that packet. This container is implemented in the data structure 'PacketStatistics' and is shown in the figure below (Figure 41).

```
struct PacketStatistics
{
int sequenceNumber;
float txTime;
float rxTime;
float delay;
float delayVariation;
bool  loss;
float packetSize;
};
```

Figure 41: StatCollector data structure – PacketStatistics

On the sending side, StatCollector takes the time of packet transmission, the packet sequence number in the flow, and packet size and stores it in a PacketStatistics object. Each flow StatCollector classifies owns a vector of PacketStatistics objects. It is in this vector that the PacketStatistics object is inserted. The delay and delay variation fields are set to 'un-defined' (a large integer) and the loss field to 1. That is, all sent packets are considered lost until found.

3.5.1.2 Receiving End

The receiving end must first remove the packet tags from the packet in order to identify the packet. Once the flow and sequence identifiers are known, the StatCollector brings up the information stored for the packet. It first uses the flow identifier to select the vector of PacketStatistics objects that belong to the flow. Next, using the sequence identifier it selects the PacketStatistics object from the vector. It checks the current simulation time and subtracts the transmission time in order to find the end to end delay. To find the delay variation, the end to end delay of the previous packet is subtracted from the current packet. It then sets these values into the same PacketStatistics object. The loss field is set to '0' to indicate the packet was received. Therefore the packets that were lost in the network will keep the loss value of '1'.

3.5.2 Internal Queue Statistics

Similar to the manner in which the StatCollector is connected to the transport layer protocols for collecting end to end statistics, the StatCollector is also connected to the DiffServQueues of the network, or more specifically the queues internal to the DiffServQueue. For each of these internal queues, the following information is desired:

i. the total packets 'to be' enqueued

ii. the total number of packets dropped

iii. Enqueue, dequeue and packet drop times

iv. the queuing delay for each packet

v. The current queue size for each packet enqueue

vi. The average queue size

vii. The size of the packets to be enqueued

When a DiffServQueue is first created, the DiffServQueue constructor function calls the StatCollector to assign it a unique DiffServQueue identifier. This identifier is stored within the DiffServQueue itself. Whenever a DiffServQueue of the network reports to the StatCollector, it gives its DiffServQueue identifier. Recall that each DiffServQueue contains 6 internal queues. The internal queues are numbered 0-5. Each internal queue reports to the StatCollector whenever it enqueues, dequeues or drops a packet. These processes are described as follows.

3.5.2.1 Packet Enqueue or Drop

The internal queue passes its DiffServQueue queue identifier, its internal queue number, the packet pointer, and whether an enqueue or drop action is performed (bool). Additionally the current queue size is also passed. Similar to the end to end statistics collection, a data structure called PacketStatisticsQueue was implemented that is used to store information associated with one packet (Figure 42).

```
struct PacketStatisticsQueue
{
 int sequenceNumber;
 float enqueueTime;
 float dequeueTime;
 float queuedTime;
 bool  drop;
 int currentQueueSize;
float packetSize;
};
```

Figure 42: StatCollector data structure – PacketStatisticsQueue

The current simulation time, the sequence number of the packet for the queue, whether the packet is enqueued or dropped, the packet size and the current queue size is stored in a PacketStatisticsQueue object. The 'dequeue' and 'queued' times are configured when the packet leaves the queue. Each internal queue of all DiffServQueues in the network owns a vector of PacketStatisticsQueue objects similar to the end to end statistics collection. The PacketStatisticsQueue object configured for the packet is inserted onto this vector. If the packet is to be enqueued, the packet is given a packet tag that contains its sequence number. The sequence number will be used to identify the packet when the packet is dequeued. The sequence tag is called SeqIdQueueTag which mimics the SeqIdTag.

3.5.2.2 Packet Dequeue

The internal queue passes its DiffServQueue queue identifier, its internal queue number, the current queue size, and the packet pointer to the StatCollector. Using the packet pointer it can retrieve the SeqIdQueueTag and hence the sequence number of the packet. With this set of information, the StatCollector can bring up the stored PacketStatisticsQueue object of the packet. It takes the current simulation time and subtracts the enqueue time of the packet to find the queuing delay. The dequeue time and queued time data members can now be set in the PacketStatisticsQueue object.

3.5.2.3 Average Queue Length

For each internal queue, StatCollector calculates the average queue length for the duration of the simulation. This is performed by finding the area under the graph of current queue size vs. time and then dividing by the simulation time (Figure 43).

$$Average\ Queue\ length = \frac{Area\ under\ Current\ queue\ size\ vs.\ time}{Duration\ of\ Simulation}$$

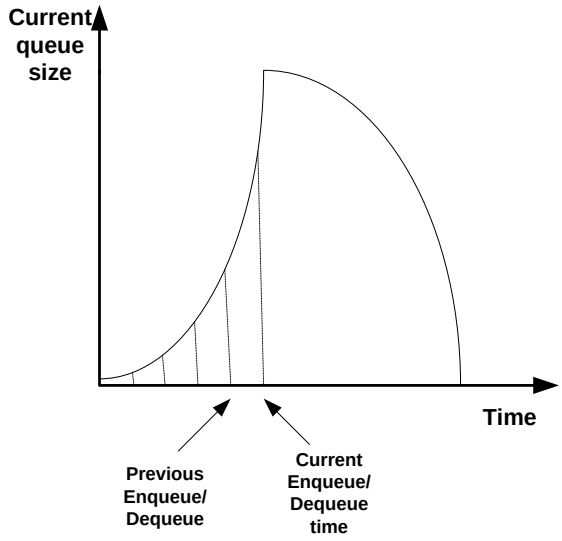

Figure 43: Average queue length calculation

To find this area, the current queue size and simulation time is taken at every enqueue **and** dequeue operation. The following calculation is then performed.

$$Area = Area + [Current\ queue\ size\ \times (current\ enqueue\ or\ dequeue\ time - last\ enqueue\ ordequeue\ time)]$$

3.5.3 StatCollector Output

Now that the simulation is over the user may want to observe the results of the simulation that the StatCollector has collected. To display this information, StatCollector creates a text file and 'writes' to it. It uses nested 'for' loops to iterate through all vectors and displays the information for each packet (PacketStatistics and PacketStatisticsQueue). This is performed for both end to end and internal statistics collection in two separate text files. The figures shown below are samples taken from

the text files for both end to end (Figure 44) and internal queue statistics collection (Figure 45). To make processing results easier for the user, StatCollector includes a Shell sorting algorithm that was obtained from (Roberts and Roberts n.d.) and a created 50th percentile function. With these functions, StatCollector can compute the 50th percentile for delay and delay variation for application flows and queuing delay for internal queues. The packet loss percentage for each application flow and internal queue is also obtained as well as the average queue length of each internal queue.

```
                                                    ****STATISTICS COLLECTOR****

Total number of flows: 1
--------------------------------------------------------------------------------------------------------------------

Flow number: 0
Source IP Address:       10.0.4.1
Destination IP Address:  10.0.13.2
Source Port Number:      49153
Destination Port Number: 5443

Flow Summary:
Number of packets sent:     2929
Number of packets dropped:  0
Percentage of packets lost: 0%
Delay 50th percentile:      0.036499
DelayVar 50th percentile:   0

Sequence Numbers     Tx Time             Rx Time             Delay               Delay Variation      Packet Lo

1                    0.4096              0.446093            0.036493            999                  0
2                    0.8192              0.855693            0.036493            2.98023e-08          0
3                    1.2288              1.26529             0.0364929           5.96046e-08          0
4                    1.6384              1.67489             0.0364929           0                    0
5                    2.048               2.08449             0.0364928           1.19209e-07          0
6                    2.4576              2.49409             0.0364928           0                    0
7                    2.8672              2.90369             0.0364931           2.38419e-07          0
```

Figure 44: StatCollector- end to end

```
                                                    ****STATISTICS COLLECTOR****

Total number of DiffServQueues: 9
*************************************************************************************************************

DiffServQueue number: 2

PHB Queue number: 0--AF1 Queue

PHB Queue Summary:
Total number of packets to be enqueued: 2929
Total number of packets dropped:        0
Percentage of packets dropped:          0%
Queueing Delay 50th percentile:         0
Average Queue Length          :         0

Sequence Numbers     Enqueue Time         Dequeue Time         Queued Time          Packet drop          Cur

1                    0.414077             0.414077             0                    0                    0
2                    0.823677             0.823677             0                    0                    0
3                    1.23328              1.23328              0                    0                    0
4                    1.64288              1.64288              0                    0                    0
5                    2.05248              2.05248              0                    0                    0
6                    2.46208              2.46208              0                    0                    0
7                    2.87168              2.87168              0                    0                    0
```

Figure 45: StatCollector-internal queues

3.6 Multimedia Application Modeling

For the simulated DiffServ network, the hosts will generate traffic that the network will attempt to deliver. The DiffServ architecture will be evaluated on its ability to provide QoS to application flows and therefore the network traffic should consist of multiple media types with different performance requirements. Two media types were considered; voice and video. An application of each media type was modeled within the simulator.

3.6.1 Voice media

For voice media, a VoIP application was modeled within ns-3 using the model proposed by (HASSAN, GARCIA and BRUN n.d.). The model states that voice traffic at the source is characterized by two periods; an active or ON period and an inactive or OFF period (Figure 46). The ON period is time the user spends talking, during which constant size packets are transmitted at regular intervals of length T (packetization time). The OFF period is the time the user stops talking and no packets are transmitted. The ON and OFF periods are estimated by the following exponential distributions. Note that the VoIP application only represents one side of a two way voice call (i.e. a single flow).

$$\text{ON Period } F(T) = \alpha e^{-\alpha T}$$

$$\text{OFF Period } F(T) = \beta e^{-\beta T}$$

The mean duration of the ON period $T_{ON} = \frac{1}{\alpha}$

The mean duration of the OFF period $T_{OFF} = \frac{1}{\beta}$

The mean duration of ON and OFF periods are given as 0.352 and 0.650 seconds respectively.

The constant packet rate during the ON period $\lambda = \frac{1}{T}$

The average packet rate for the ON-OFF process = $\frac{T_{ON}\ (\lambda)}{(T_{ON}+T_{OFF})}$

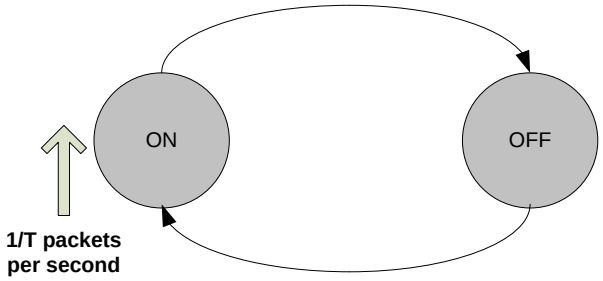

Figure 46: voice on-off model

Source: (HASSAN, GARCIA and BRUN n.d.)

The actual bit rate depends on the codec used and the packetization time. The codec used for this project is the G.711. It performs no compression thus ensuring the best voice quality. However as a result, its bandwidth requirements will be higher. The data rate produced by this codec is 64Kbps (at the application layer). The packetization time chosen was a tradeoff between end to end delay and bandwidth requirement. The larger the packetization time, the more voice samples are put in a packet; thus requiring fewer packets to be transmitted. However, the end to end delay becomes longer. For shorter packetization times, packets are transmitted more frequently, thus reducing end to end delay. However since packets are transmitted more frequently,

each with their own overhead (RTP, UDP, IP protocol headers), more bandwidth is required. For small end to end delay and bandwidth requirement a packetization delay of 20ms was chosen.

3.6.1.1 VoIP call bandwidth calculation

For the conducted evaluation, it was necessary to create a degree of congestion in the network. This requires knowledge of the average data rate the application achieves.

Codec specification

Codec G.711 output = 64kbps

For a packetization time of 20ms per packet, a packet has 160 bytes for its payload.

Overhead calculation:

The protocols that are used to transport voice data are UDP (8 bytes), RTP (12 bytes) and IP (20 bytes).

Bandwidth per voice call = (160 bytes) packet payload + (8+12+20 bytes) transport and IP header + layer 2 header (bytes) × 50 packets per second = bandwidth in bytes per second**.**

At IP layer the bandwidth required = 80 Kbps per call.

Using the On-OFF process model, the **average** bandwidth required becomes $\frac{0.352\ (\ 80)}{(1.002)}$ = 28.1 kbps

3.6.1.2 NS-3 implementation

The VoIP application was modeled using an ns-3 On/Off application (Figure 47). The On/Off application takes the following parameters:

1. ON/OFF times
2. Packet size (payload)
3. Data rate (bps)

The resulting inter-arrival times are calculated internally by the application. The ON and OFF times were produced using two ns-3 exponential random number generators with means of 0.352 and 0.65 seconds respectively. Since ns-3 did not have a RTP implementation, the size of the RTP header was added to the voice payload. The payload at the application layer no becomes 172 bytes (160 byte payload + 12 RTP header bytes). The data rate also now becomes 68,800 bps (172 bytes per packet × 50 packets per second). The resulting inter-arrival time becomes 20ms.

```
onoffhelper1.SetAttribute("PacketSize", UintegerValue (172));
onoffhelper1.SetAttribute("DataRate", DataRateValue (68800));
onoffhelper1.SetAttribute("OnTime", RandomVariableValue (onTime));
onoffhelper1.SetAttribute("OffTime", RandomVariableValue (offTime));
```

Figure 47: VoIP application - ns-3 implementation

3.6.2 Video media

A streaming video application was used to represent this media type. The application mimics the data sent from a video streaming web site to a single host (i.e. a single flow in one direction). The application was modeled within the simulator using the ns-3 'UDP-trace client' application. The application sends UDP packets based on a trace file

of a MPEG4 stream. The trace file is specified by the user and the application produces packets with sizes and arrival times as specified in the trace file. A movie's trace file obtained from (Telecommunication Networks Group n.d.) was used for this purpose. Trace files must have four columns specifying the frame index, the frame type, the time the frame was generated by the encoder and the frame size in bytes (Figure 48).

The MPEG codec defines three frame types; I, P and B frames. 'I' frames are frames that are compressed using only the information contained within the frame. 'P' frames are compressed using the frame itself and the closest preceding 'I' or 'P' frame. 'B' frames are compressed using data from the closest preceding and following 'I' or 'P' frame. As a result, the order in which the frames are generated by the encoder are different from the frame transmission order (Simpson 2008).

The sample trace file shown below (Figure 48) displays the frame **transmission** order. The frames that have been reordered are the type 'B' frames. The frames are held back until a following 'P' or 'I' frame is generated. The 'B' frames are then compressed and transmitted. Therefore in the sample trace below, the two 'B' frames (frames 2 and 3), will be transmitted at the same time as the preceding 'P' frame at 120ms.

Frame No.	Frametype	Time[ms]	Length [byte]
1	I	0	402
2	P	120	490
3	B	40	142
4	B	80	567
5	P	240	539
6	B	160	142
7	B	200	159
8	P	360	76
9	B	280	38
10	B	320	33

Figure 48: Video streaming application - sample trace file

Source: (Telecommunication Networks Group n.d.)

4 Results

This chapter comprises the experiments conducted for the performance evaluation of the architecture. Due to the large size of the testing section, it was placed in the Appendices (Appendix F). The results of the following experiments can be found in Appendix G.

4.1 Experiments conducted for the evaluation of DiffServ

A single DS domain was used for the evaluation of DiffServ (Figure 49). This network topology was chosen as it mimics a real network scenario. The first level of congestion is at the customers' boundary nodes (Edge 0 and Edge 1). The customers that are connected to each boundary node compete for the available resources. The second level of congestion is at the bottleneck link labeled Core 2. All application flows of each customer must transverse this point in order to reach their destinations. By providing multiple congestion points, the DiffServ network can be tested on an edge to edge basis for Quality of Service. Because of the asymmetric QoS that DiffServ provides, data is only sent from hosts 1-4 to hosts 5-8 for all subsequent experiments (i.e. one direction)

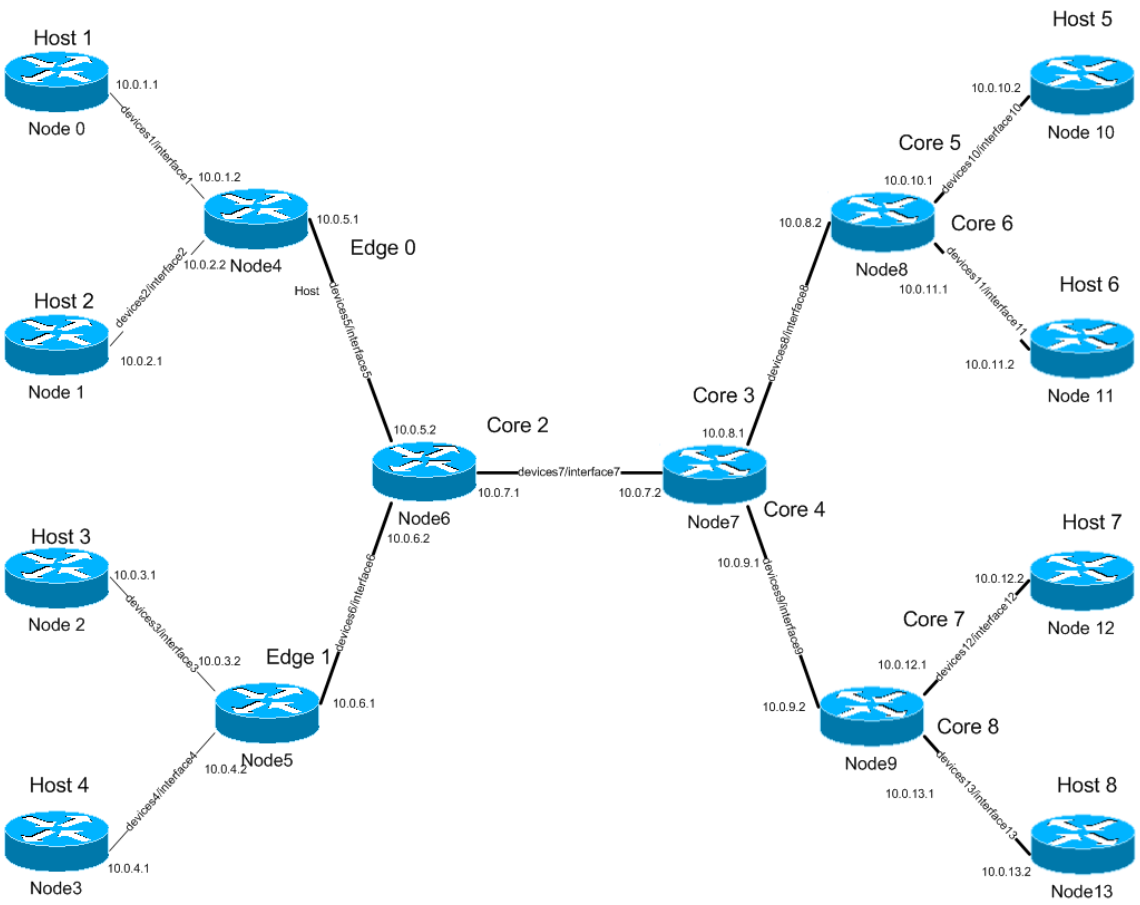

Figure 49: Experiments- Network topology

Table 3 below displays the network configuration for each experiment. All channels had a propagation delay of 2ms. All internal queues were of size 100 packets unless specified otherwise.

Table 3: Experiments- Network topology configuration

Queue	Queue Type	NetDevice Data Rate (Mbps)					
		Exp 1	Exp 2	Exp 3	Exp 4	Exp 5	Exp 6
Host 1	Drop Tail	100	100	100	n/a	100	100
Host 2	Drop Tail	100	100	100	n/a	100	100
Host 3	Drop Tail	100	100	100	n/a	100	100
Host 4	Drop Tail	100	100	100	n/a	100	100
Edge 0	DiffServQueue(Edge)	0.5	10	0.35	n/a	0.6	0.4
Edge 1	DiffServQueue(Edge)	0.5	10	0.35	n/a	0.6	0.4

Core 2	DiffServQueue(Core)	0.5	0.5	0.35	n/a	0.6	0.4
Core 3	DiffServQueue(Core)	0.5	0.5	0.35	n/a	0.6	0.4
Core 4	DiffServQueue(Core)	0.5	0.5	0.35	n/a	0.6	0.4
Core 5	DiffServQueue(Core)	0.5	0.5	0.35	n/a	0.6	0.4
Core 6	DiffServQueue(Core)	0.5	0.5	0.35	n/a	0.6	0.4
Core 7	DiffServQueue(Core)	0.5	0.5	0.35	n/a	0.6	0.4
Core 8	DiffServQueue(Core)	0.5	0.5	0.35	n/a	0.6	0.4

4.1.1 Application Configuration for all experiments

Voice – The VoIP application was run for 1200 seconds using an ns-3 exponential random variable for it's on and off times. The average rate obtained from Wire Shark was approximately 0.028Mbps. Voice applications produced approximately 20,000 packets for this duration.

Video – The video streaming application was run for 1200 seconds using a trace file with the ns-3 UDP trace client application. The average rate obtained from Wire Shark was approximately 0.1Mbps. Video applications produced approximately 35,000 packets for this duration.

4.1.2 Experiment 1 – Testing the forwarding classes for resource allocation

The objective of this experiment was to observe the ability of the architecture to allocate resources in terms of priority and bandwidth allocation to different forwarding classes as compared to the Best Effort service. In the Best Effort network, there is only a single class of traffic where all flows experience similar service. In the DiffServ network there are multiple forwarding classes which have different bandwidth allocations and priority.

Hosts 1 to 4 transmitted both voice and video traffic into a particular forwarding class for subsequent experiments (Exp 1.1 to 1.6) (Table 5). A single SLA was created for each host that includes both the voice and video applications flows. Each SLA was configured to place the Hosts' applications into the specified forwarding class. No metering and AQM was performed; all data sent form hosts were marked with the specified code point. The Weighted Round Robin queue weights for all DiffServQueues were varied for subsequent experiments. In Exp 1.7 the Best Effort service was simulated. To simulate the Best Effort network, all packets were unmarked and entered the default forwarding class queue. The default forwarding class queue size was set to 400 packets for this experiment to retain the amount of forwarding resources from the previous experiments. This was done so a fair comparison could be made across experiments.

Table 4: Experiment 1 – host configuration

Host #	SLA #	Flow type
1	1	Voice, video
2	2	Voice, video

3	3	Voice, video
4	4	Voice, video

Table 5: Experiment 1 – experiment configuration

Experiment#	Queue Weights (AF1, AF2, AF3,AF4)	SLA to forwarding class mappings (SLA 1, SLA 2, SLA 3, SLA 4)
1.1	10,9,8,7	AF1,AF2,AF3,AF4
1.2	9,8,7,6	Same as above
1.3	8,7,6,5	Same as above
1.4	7,6,5,4	Same as above
1.5	1,1,1,1	Same as above
1.6	1,1,1,n/a	AF1,AF2,AF3,EF
1.7	n/a	Default (queue size = 400)

4.1.3 Experiment 2 – Testing an AF forwarding class drop precedence for resource allocation

The objective of this experiment was to observe the resource allocation of DiffServ drop precedence levels on increasing traffic loads as compared to increasing traffic loads using the Best effort service. When congestion occurs in the Best effort network, packets are dropped when queues overflow. However no preference is given to any flow when packet drops occur. The DiffServ network however allows packets to have different levels of drop precedence that can give a packet a higher priority to buffer space than another.

For this experiment, hosts 1 to 3 transmitted voice and video traffic into the AF1 forwarding class at different drop precedence levels; host 1 (AF11), host 2 (AF12) and

host 3 (AF13). Host 4 transmitted CBR traffic at a particular rate and AF1 drop precedence level to increase the congestion in the network.

Host 4 performed this for the three AF1 drop precedence levels and for 10 different CBR rates (Exp 2.1 to 2.3) (Table 8). Metering was disabled; packets were marked with their respective code points. WRED was enabled in all DiffServQueues using the parameters shown in Table 7. The experiment was then re- performed using the Best Effort service (Exp 2.4). This was performed with Marking and WRED disabled. All packets entered the default forwarding class.

Table 6: Experiment 2 – host configuration

Host #	SLA #	Flow type
1	1	Voice, video
2	2	Voice, video
3	3	Voice, video
4	4	CBR traffic

Table 7: Experiment 2 – AF1 WRED profile

Packet Color	MinTH	MaxTH	Max Pr
Red	10	40	100
Yellow	30	70	70
Green	60	100	40

Table 8: Experiment 2 – experiment configuration

Experiment#	SLA to forwarding class mappings (SLA 1, SLA 2, SLA 3, SLA 4)	Host 4 CBR rate
2.1	AF11,AF12,AF13,AF11	10kbs to 100kbs in 10kbps increments Packet size of 512 bytes
2.2	AF11,AF12,AF13,AF12	Same as above

2.3	AF11,AF12,AF13,AF13	Same as above
2.4	Default, Default, Default, Default	Same as above

4.1.4 Experiment 3 – Comparing Bandwidth utilization and flow isolation of DiffServ and Best Effort networks

The objective of this experiment was two compare the DiffServ and Best Effort networks in terms of bandwidth utilization and flow isolation. In the Best effort network, traffic from subscribers that exceed their subscribed rate is dropped to protect the rest of the network. This is performed regardless of the congestion level of the network. However in the DiffServ network, subscribers can exceed their subscribed profile while the rest of the network is still protected. This is accomplished by marking those non conformant packets with an inferior code point. The advantage of this method is that DiffServ networks can utilize the network resources more efficiently.

For this experiment (Exp 3.1), hosts 1 and 3 transmitted voice and video traffic into the AF1 forwarding class (SLA 1 and SLA 3). Each SLA comprised both applications. For each experiment, host 1 also generated CBR traffic at a particular rate into the AF1 class under the same SLA (SLA 1). This is performed so that Host 1 exceeds his subscribed rate defined in SLA 1. Host 1 performed this for 5 different CBR rates for subsequent experiments. Metering was performed using the token bucket meter at the traffic profile shown in Table 9. Conformant packets were marked with the AF11 code point (green). Non conformant packets were marked with the AF12 code point (yellow).

WRED was enabled with the profile shown below. The experiment was repeated using the Best effort service by using the default forwarding class (Exp 3.2). The flows of each SLA were still metered using the same traffic profile and token bucket meter, however no marking was performed. Instead, conformant packets were allowed into the network while non conformant packets were dropped at the edge. WRED was also disabled for all DiffServQueues. Exp 3.2 is then repeated with metering disabled so that all packets are allowed into the network.

Table 9: Experiment 3 – experiment configuration

Experiment #	SLA –forwarding class mapping (SLA 1 and SLA 3)	Host 1 CBR	Metering Profile	AF1 WRED profile (MinTH, MaxTH, Max Pr)
3.1	AF1	10kbs to 50kbs in 10kbps increments Packet size of 512 bytes	CIR = 150,000 bps CBS = 30,000 bytes	Green (n/A, n/A,0) Yellow(10, 50, 100)
3.2	Default	Same as above	Same as above	n/A
3.3	Default	Same as above	none	n/A

4.1.5 Experiment 4 – Comparison of Metering schemes

For this experiment the token bucket, the srTCM and the trTCM meters were compared using CBR traffic. The CBR traffic data rate was varied between 100kbps to 250kbps in 10kbps increments for each meter. For each subsequent experiment, a different meter was used. Table 10 below shows the actions that were taken for the

conformant and non conformant packets. The objective was to compare the performance of the different metering schemes under different data rates.

Table 10: Experiment 4 – experiment configuration

Experiment #	Meter Type	CBR traffic	Traffic profile	Metering actions
Exp 4.1	Token bucket	100kbps to 250kbps in 10kbps increments Packet size of 512 bytes	CIR = 100,000 bps CBS = 1000 bytes	Conformant – AF11 Non conformant - AF13
Exp 4.2	srTCM	Same as above	CIR = 100,000 bps CBS = 1000 bytes EBS = 100,000 bytes	Conformant green – AF11 Non conformant yellow- AF12 Non conformant red – AF13
Exp 4.2	trTCM	Same as above	CIR = 100,000 bps CBS = 1000 bytes PIR = 130,000 bps PBS = 1000 bytes	Same as above

4.1.6 Experiment 5 – Comparison of a DiffServ deployment and Best Effort network

This experiment aimed to compare a DiffServ deployed network and a Best effort network. In the Best effort network only a single class of service is provided. The DiffServ network however can provide different levels of service to customers so that their application flows meet their performance requirements.

The first experiment (Exp 5.1) used the DiffServ network where hosts 1 to 4 transmitted both voice and video traffic in the forwarding classes shown in Table 11.

The network provided three services for the customers' flows, the Premium service created from the EF class, the Gold service from the AF1 class and the Silver service from the AF2 class. The AF1 class was better provisioned than the AF2 class such that the Gold service provided better performance than the Silver service. The voice applications of all hosts used the Premium service (EF). Hosts' 1 and 3 video applications used the Gold service while hosts' 2 and 4 video applications used the Silver. This was performed using two SLAs per host; one for the video and one for the voice application. Table 11 displays the configuration. The experiment was then repeated with the Best Effort service (Exp 5.2). The Best Effort network was simulated using the default forwarding class as done in previous experiments. The default queue size was changed to 200 packets to keep the amount of buffer space consistent.

Table 11: Experiment 5 – experiment configuration

Experiment #	Host	SLA #	Flows	Forwarding class	WRR weights
5.1	1	1.1	Voice	EF	n/a
		1.2	Video	AF1	4
	2	2.1	Voice	EF	n/a
		2.2	Video	AF2	3
	3	3.1	Voice	EF	n/a
		3.2	Video	AF1	4
	4	4.1	Voice	EF	n/a
		4.2	Video	AF2	3
5.2	1	1.1	Voice	default	n/a
		1.2	Video	default	n/a
	2	2.1	Voice	default	n/a
		2.2	Video	default	n/a
	3	3.1	Voice	default	n/a
		3.2	Video	default	n/a
	4	4.1	Voice	default	n/a
		4.2	Video	default	n/a

5 Discussion

5.1 Experiment 1

The objective of this experiment was to investigate the ability of the architecture to allocate resources in terms of priority and bandwidth allocation to different forwarding classes as compared to the Best Effort service. For Exp 1.1 to 1.4, different WRR weights were assigned to AF queues which produced different bandwidth allocations among them. Exp 1.5 used equal weights among the AF queues and supposedly equal bandwidth allocations. In Exp 1.6, SLA 4 used the EF forwarding class instead of AF4. Lastly Exp 1.7 used the Best Effort network where all traffic sent was not marked with any code point. All traffic entered a single queue with a buffer size equal to the combined size of the AF queues of the previous experiments (400 packets). This was done to ensure that the forwarding resources did not change by using a single queue and to maintain fairness across experiments.

5.1.1 Packet Loss (Figure 85 and Figure 86)

Exp 1.1 to 1.4 shows increasing levels of packet loss among the four SLAs for both voice and video traffic. This is expected, since for each experiment, the weighted round robin weights were assigned among the AF queues in an ascending order and each host used a different AF class. Additionally, for each subsequent experiment for Exp 1.1 to Exp 1.4, the loss percentage experienced by each SLA becomes increasingly diverse. This is also expected since from one experiment to another (Exp 1.1 to Exp

1.4) the weights were made more concentrated resulting in more diverse bandwidth allocations among the AF queues. For example, the weights were moved from (10, 9, 8, 7) to (9, 8, 7, 6) to (8, 7, 6, 5). It is also observed that resources tend to move away from SLAs 3 and 4 and distributed among SLAs 1 and 2 for both voice and video. It is also observed that the voice traffic suffered significantly more loss than the video traffic for all experiments. The larger volume of traffic produced by the video applications may be responsible for this.

Exp 1.5 used equal WRR weights for all AF queues and as such displays little variance in the amount of loss experienced by the four SLAs for both voice and video traffic. This is expected since each AF queue was given an equal share of the available bandwidth.

In Exp 1.6, SLA 4 used the EF forwarding class instead of the AF4 class and as such displays no loss for both voice and video traffic. This is expected since the priority queue gives preference to EF traffic before the AF forwarding classes. SLA 4 received the full available bandwidth of the outgoing link and therefore received no loss. SLAs 1 to 3 show very little variance in the amount of loss experienced during this experiment for both voice and video. This occurs due to the equal WRR weights that were set for the remaining AF1, AF2 and AF3 queues.

The final experiment, Exp 1.7, was performed using the Best Effort service; the Default forwarding class was used for all SLAs. Voice traffic for Exp 1.7 shows relatively similar loss across SLAs as opposed to video traffic which shows a significant variance in the amount of loss experienced by all SLAs. This may be due to the proposed 'single queue

lock out effect' as a result of the traffic characteristics of the video applications (proven to be true in Experiment 6). This effect is explained as follows.

All video applications use the same trace file, and therefore have identical transmission times and packet sizes. Since this experiment used the Best Effort service, all video flows enter the same queue. A point in time arrives where the queue becomes full and incoming packets are dropped. However since the video packets arrive at the queue at relatively similar times, the packets of some flows enter the queue while others are dropped, thus causing some flows to be 'locked out' of the queue.

However if this was the case for this experiment, it happened on small scale. As expected, the voice traffic does not show this behavior due to their random on and off times which also supports the occurrence of this effect for video traffic.

In comparing DiffServ and the Best Effort service, Exp 1.7 is compared to the previous experiments. Exp 1.7 compared to Exp 1.5 shows a greater overall loss for video traffic but a lower amount of loss for voice traffic. The lower loss for voice traffic can be explained by the fact that the queue size was four times greater and hence more readily to accept packet bursts, thus causing fewer packets to be dropped. When Exp 1.7 is compared to Exp 1.1 to Exp 1.6, it is clear that the Best Effort network does not achieve the diverse performance of the DiffServ network.

5.1.2 End to End Delay (Figure 87 and Figure 88)

Similar to packet loss, Exp 1.1 to 1.4 shows varying levels of delay among the four SLAs for both voice and video traffic. This is expected since varying bandwidth allocations to each AF queue causes varying levels of queuing, and hence greater queuing delay in AF queues with less bandwidth. Similar to Packet loss, the delay increases in an ascending order from SLA 1 to 4 for video traffic. This is expected since the WRR weights and hence bandwidth allocations were distributed in an ascending order.

For voice traffic however, the delay increases in an ascending order for SLA 1 to SLA 3. SLA 4 seems to have a lower delay than SLA 3. At first glance this seems to be contradictory to the bandwidth allocations. However upon closer inspection at Exp 1.5 and Exp 1.7 (the equal weights and best effort experiments respectively) SLA 3 has a significantly higher delay than SLA 4 under these conditions. These conditions which supposedly showed no bias to a particular AF queue shows that SLA 3 had greater levels of delay than SLA 4. This can provide the assumption that the increased delay of SLA 3 is due to the voice application itself. Recall that the voice applications used random variables to generate their on and off times. If SLA 3‘s voice application had a less distributed on and off periods than other voice applications, meaning that it was more bursty over a long period of time, then one can expect that it will cause more congestion for the AF3 queue and hence increased end to end delay. Another possibility is that the combined voice and video traffic for SLA 3 caused the increase in delay for the voice traffic.

Nevertheless, when the bandwidth allocations were made more concentrated for each subsequent experiment during Exp 1.1 to 1.4, it is observed that the delay difference between SLA 3 and SLA 4 decreases with each experiment until finally at Exp 1.4 SLA 3's delay becomes lower than that of SLA 4. This supports the previously made assumption about the application burstiness. Note that if the experiment trend was continued, that is if further experiments were performed with increasingly concentrated WRR weights (e.g. 4,3,2,1), it is expected that the delay of SLA 3 would become increasing lower than that of SLA 4.

Another point to note is that the delay of voice traffic is significantly less than that of video traffic for all experiments. This is most likely due to the larger packet sizes produced by the video applications. The voice applications on the other hand produced a constant packet size of 200 bytes. A Larger packet size in the network causes a longer serialization time, which in turn increases end to end delay. The number of nodes a packet has to transverse is also a factor. The network topology consisted of 6 'hops' from source to destination. At each of these points each packet needed to be serialized and hence added an extra delay.

During Exp 1.5 it is observed that the voice applications show diverse delay. As explained previously, this is mostly likely due to nature of the applications themselves and not the network. This assumption is further supported by the fact that the video applications which used the same trace file to determine transmission times show relatively similar delay.

For Exp 1.6, SLA 4's voice and video application shows very low delay (36ms and 82 ms respectively) with the use of the EF forwarding class. The video application's delay is greater due to the serialization delay explained above. This is expected since the use of the priority queue gave SLA 4 unlimited preemption of other traffic at the node and hence was able to utilize the full link bandwidth, thus resulting in very low end to end delay.

The last experiment Exp1.7, the best effort service was used. Once again the voice applications show diverse delay while the delay of the video applications is relatively similar for the reasons explained above. Nevertheless, the Best Effort service does not provide the differential treatment that DiffServ is able to as displayed by Exp 1.1 to Ex 1.6.

5.1.3 Delay Variation (Figure 89 and Figure 90)

For all experiments, the voice traffic for all SLAs shows little change in delay variation. One explanation for this behavior is the constant packet sizes of the voice traffic. Constant packets sizes result in constant serialization delays for all voice packets throughout the network. In Exp 1.6, SLA 4 used the EF queue but its voice application's delay variation did not decrease significantly from previous experiments. This indicates that the delay variation experienced by all voice applications is already at a minimum. When comparing the Best Effort service of Exp 1.7 to the previous experiments, there is no significant change in delay variation.

The video traffic however, shows an increasing delay variation among SLAs 1 to 4 for Exp 1.1 to 1.4. This is expected since the WRR scheduler will service a queue with a greater weight for a longer time than a queue with a smaller weight. A queue that dequeues more packets than another queue will have less delay variation among the packets transmitted. The queue with the shorter servicing time will have to wait a longer period to be serviced again, thus causing more delay variation.

Also for video applications, a slight increase in delay variation for all SLAs is observed between subsequent experiments for Exp 1.1 to 1.4 (not very noticeable from Figure 90) .This behavior at first glance seems unusual and puzzling, however there is an explanation. For these four experiments the WRR weights assigned were made more concentrated in each subsequent experiment (i.e. (10, 9, 8, 7) to (9, 8, 7, 6) to (8, 7, 6, 5)). As a result, for each subsequent experiment, each queue was serviced for shorter and shorter times before moving on to another queue. This behavior is what is suspected to cause the increase in delay variation observed. If the experiment trend was continued, this behavior is likely to be seen more clearly. However, Exp 1.5 and 1.6 clearly shows this behavior. In Exp 1.5, the WRR weights of (1,1,1,1) were used and as a result the delay variation among each video application is significantly higher than Exp 1.1 to Exp 1.4. This occurs because when the WRR scheduler services one packet from a queue, that same queue has to wait until one packet each has been removed from the remaining three queues until it is serviced again. This directly increases the delay variation for the flows that use that queue. Since Exp 1.6 used equal WRR

weights (1,1,1) for the remaining AF queues as well, similar high values of delay variation was experienced.

The Best Effort network of Exp 1.7 also shows high delay variation for all SLA video applications. This can be explained by the fact that all video flows use the same queue and therefore the traffic from all flows is mixed together. This results in packets from one flow being separated by packets from other flows and hence causes high delay variation values for that flow.

For the comparison between the Best Effort and DiffServ networks, the DiffServ network achieves roughly 50% of the delay variation experienced in the Best Effort network through the use of separate queues and the WRR scheduler.

5.1.4 DiffServQueue statistics (Figure 91,Figure 92 and Figure 93)

The DiffServQueue labeled as CoreQueue 2 was analyzed for each experiment using the StatCollector. Packet loss percentage, queuing delay 50^{th} percentile and average queue length were computed for each internal queue. CoreQueue 2 was chosen since it is the most congested queue in the topology where differential service is mostly likely to be seen.

The packet loss graph displays the loss experience by each internal queue. Since each SLA used a different internal queue, the internal queues should reflect the performance of the SLAs. From the loss graph, this seems to be true. From Exp1.1 to 1.4, the loss experienced by each AF queue is distributed in an ascending order. Also similar to the SLAs, the loss experienced by each internal queue seems to become

increasingly diverse from experiment to experiment for Exp1.1 to Exp 1.4 as expected. Also expected, a similar trend occurs for the queuing delay and average queue length graphs.

Exp 1.5 shows the equal weights experiment. It is observed that the loss of each internal queue for this experiment is lower than that of Exp 1.6 where the EF queue is used. This increase in loss for Exp 1.6 is a direct result of the priority queue scheduler. The priority queue scheduler gives the EF queue full use of the available link bandwidth, resulting in less bandwidth and hence increased loss for the AF remaining queues. The EF queue however experiences no loss. The queuing delay among the internal queues for Exp 1.5 is relatively similar as expected, except for the AF3 queue which is significantly higher than the others. This observation supports the previous argument that Host 3/SLA 3‘s voice application which uses the AF3 class is significantly bustier than the other voice flows. However the AF3 average queue length for this experiment shows no significant difference between the other internal queues.

The EF queue for Exp 1.6 shows no loss and a very small queuing delay as expected, compared to the others, as low as 6.5ms (too small to be seen on graph). It can be assumed that this delay value would be even lower if only voice traffic used the EF queue. However since both voice and video traffic were using the EF queue, the video traffic larger packet size and hence larger serialization delay would have an effect on the queuing delays experienced by all packets of the EF queue. The average queue length for this experiment is also very low as expected (0.22 packets)

5.2 Experiment 2 (Figure 94 to Figure 105)

The objective of this experiment was to observe the resource allocation of DiffServ drop precedence levels on increasing traffic loads, as compared to increasing traffic loads using the Best effort service.

Experiments 2.1 to 2.3 show that green traffic has the least loss (< 1%) among the three drop precedence levels, followed by yellow traffic (2-5%) and then red traffic with (22-31%) dropped packets for each of the three experiments. This behavior is expected since the red traffic was configured with a more aggressive RED (random early detection) dropping profile followed by yellow and green traffic. It is also observed that the video traffic had a slightly greater amount of packet loss than the voice traffic. This may be explained by the larger volume of video traffic compared to voice traffic in the network.

In comparison to the Best effort network, the voice traffic sent by SLAs 1 to 3 shows similar loss for on increasing traffic loads (Figure 100). The video traffic of SLAs 1 to 3 however, had significant difference in the levels of packet loss, on increasing traffic loads (Figure 101). The previously explained 'lock out effect' that occurs for video traffic may be responsible for this behavior. (Experiment 6 confirms this effect). However the loss differences among the SLAs are nowhere as diverse as those experienced in the DiffServ network.

Next, the comparison between experiments 2.1 – 2.3 is made. It is observed from the comparison graphs (Figure 102, Figure 103, and Figure 104), that increasing green

traffic causes the greatest loss on all packet colors but mostly on red followed by yellow. Increasing yellow traffic causes less loss on all packet colors than increasing green traffic followed by increasing red traffic. Interestingly enough, increasing yellow traffic seems to have a similar effect on red traffic as when the green traffic was increased. This may be due to the low maximum threshold set for red traffic. The maximum threshold, when exceeded would cause all incoming red packets to be dropped. Increasing red traffic seemed to causes the lowest loss on all packets colors, especially on green traffic where the effect seems nonexistent.

These behaviors may be explained by the average queue size and how it varies. Figure 105 shows how the AF1 average queue length varied for Exp 2.1 to 2.3. It is observed that increasing green traffic caused the average queue length to increase the most, followed by increasing yellow and red. This is expected since green packets are more readily accepted by the queue and therefore increases the average queue length the most. The increase in average queue length then reacts more aggressively on the yellow and red traffic that attempts to be enqueued thus causing more loss as the traffic load is increased.

In conclusion, the experiment clearly indicates that the DiffServ network is able to allocate buffer space to different classes of traffic during congestion. Usually, AF drop precedence levels are assigned to packets based on their level of conformance. However this is not the only way drop precedence can be used. As shown in this experiment they can be assigned to a customer for all his traffic where it will receive

preferential treatment over other traffic. Note that the RED profiles assigned in this experiment represent an extreme case and that an actual deployment using this scheme will have more similar profiles for green, yellow and red packets. For example RED profiles with max probability of 10, 15 and 20 percent for green, yellow and red respectively can be used to achieve this service.

5.3 Experiment 3 (Figure 106, Figure 107 and Figure 108)

The objective of this experiment was to compare DiffServ and Best effort networks in terms of bandwidth utilization and flow isolation. Bandwidth utilization meaning - how efficient the network is in using the available bandwidth. Flow isolation meaning – how well the network the network prevents misbehaving flows from affecting other flows in the network.

Exp 3.1 first tested the DiffServ enabled network. Hosts 1 and 3 (SLA 1 and SLA 3) transmitted both voice and video traffic into the AF 1 forwarding class. They both used the token bucket meter, where conformant packets are marked green (AF11) and non conformant packets are marked yellow (AF12). Host 1 also transmitted CBR traffic at a particular rate so that the traffic profile of SLA 1 was exceeded. This was done for five CBR rates (10kbs to 50kbs, in 10kb increments). The network used WRED to determine which packets are to be dropped and which are to be enqueued. Green packets were configured not to be dropped while yellow packets had an aggressive dropping RED profile. Exp 3.2 re-performed Exp 3.1 under the Best Effort service. The only changes

made were that conformant packets were sent into the network and non conformant packets were dropped at the edges. The AQM mechanism was also disabled.

From the packet loss graph (Figure 106) it can be seen that the DiffServ network is able to provide flow isolation between SLA 1 and SLA 3. SLA 1 was transmitting above its subscribed profile due to the extra CBR traffic but this extra traffic does not seem to significantly affect the other flows in the network. What the extra traffic does have an effect on is SLA 1's voice and video application by causing an increase in the amount of loss they experience. This is expected since the combined CBR, voice and video traffic sent will be outside SLA 1's traffic profile and therefore some packets from each flow will be marked yellow (AF12). The extra traffic causes SLA 1's packets to have a greater probability of being dropped in the network. However, since SLA 3 is mostly inside its subscribed profile, its packets will be marked mostly green (AF11) and therefore have a lower probability of being dropped. 'Mostly' is used since SLA 3 was not completely transmitting inside its subscribed profile and therefore a small portion of yellow packets were generated.

It is also observed that SLA 1's video traffic seemed to have greater loss than its voice traffic. Once again, this may be due to the larger amount of traffic the video application produces compared to the voice application. Nevertheless, it can be seen that DiffServ markings can provide a degree of flow isolation in the network. In other words DiffServ can protect customers that transmit within their subscribed rate

against those that exceed their subscribed rate by mostly dropping packets from the non conformant flows.

In Exp 3.2 the Best effort network was used. The loss graph (Figure 107) indicate that the increased CBR traffic of SLA 1 does not have an effect on the voice and video of SLA 3 but rather on the voice and video of SLA 1. This behavior is also expected. Since SLA 1 is transmitting at a rate outside its subscribed profile, the excess non conformant packets are dropped at the edge. The benefit of this method is that SLA 1's excess traffic does not affect other network flows as was previously seen in the experiment with the DiffServ network. Also seen in Exp 3.1 with the DiffServ network, SLA 1's video traffic has significantly more loss that its voice traffic. Once again this is probably due to the larger volume of traffic produced by the video application.

By comparing the loss graphs of Exp 3.1 and 3.2 it can be seen that the DiffServ network has significantly lower levels of loss than the Best effort network. Therefore, although the Best effort network provides a degree of flow isolation similar to the DiffServ network, it does not achieve its bandwidth utilization. DiffServ accomplishes this by 'marking down' non conformant packets at the edge and only dropping them at the core if congestion occurs. The Best effort network however, drops non conformant packets at the edge, regardless of the state of the network, thus it does not use the available bandwidth efficiently.

Now, one would ask the question, why the Best Effort network cannot let non conformant packets into the network instead of dropping them. Surely this would

greatly utilize the network's available bandwidth. This is the case of Exp 3.3. In Exp 3.3 the Best Effort network does not drop non conformant packets but allows both into the network. The loss graph of Exp 3.3 (Figure 108), shows that both voice and video applications of both SLAs achieve lower loss than Exp 3.2 as expected. However upon closer inspection it is observed that the flow isolation is no longer achieved. With each increment of CBR traffic, the flows of both SLA 1 and SLA 3 experience increased loss even though only SLA 1 is above its subscribed profile. Even contradictory, SLA 3 seems to experience more loss than SLA 1 for both voice and video traffic. Therefore this experiment shows that while the bandwidth it greatly utilized, the isolation among flows is no longer achieved.

Therefore in conclusion the results indicate that the DiffServ enabled network can achieve greater bandwidth utilization while still providing flow isolation compared to the Best Effort network. It does this via down marking (marking with an inferior code point) non conformant packets and only dropping them during high levels of congestion.

5.4 Experiment 4 (Figure 109, Figure 110, Figure 111)

The objective of this experiment was to compare the performance of the token bucket, srTCM and trTCM metering schemes. The PBS and/or CBS of the three meters were configured to be a small value of 1000 bytes so that only the information rates (CIR, PIR) are considered. The meters were tested on increasing CBR data rates for subsequent experiments (100kbps to 250 kbps with 10kbps increments, packet size of

512 bytes) and the number of packets at each conformance level was recorded. For each experiment only 1000 packets were sent. Note that the CBR rates are at the application layer. The resultant CBR rate at the IP layer was found by adding the UDP and IP header sizes to the packet sizes. These rates were calculated and are shown on each meter graph.

For the token bucket experiment (Figure 109), increasing the CBR rate decreases the number of green (conformant) packets and increases the number of red (non conformant) packets. This increase and decrease seems to occur symmetrically between the green and red packets. Note that when the CIR is at 200kbps, the two lines intersect resulting equal amounts of red and green packets. The CBR rate at which this occurs is double the token bucket's CIR. Further increases in the CBR result in more red than green packets.

The srTCM experiment (Figure 110) shows very similar characteristics to the token bucket meter. When the CBR rate increases, the number of green packets decreases while the number of red packets increases, similar to the token bucket. However the increase-decrease behavior is not symmetric. This occurs due to the extra bucket of the srTCM. Recall that when the data rate of incoming packets exceeds the srTCM's CIR, it will look in the extra bucket and use those tokens. These packets are then marked yellow. When the extra bucket is empty, the packets are marked red. The srTCM graph displays this behavior as red packets are only produced when the number of yellow packets reaches a maximum (indicating that the extra bucket has been

depleted). The extra bucket is also responsible for the lack of symmetry between the number of red and green packets; it acts like an offset. It also causes the srTCM red and green graphs to intersect at a higher CBR rate. The token bucket 'red packet graph' was super imposed onto the srTCM graph to display this.

The trTCM experiment (Figure 111) displays very different results from the previous two meters. It is observed that for all CBR rates before the peak rate (130kbps) there are only green and yellow packets, where the number of green and yellow packets are decreasing and increasing respectively. As soon as the trTCM's PIR is exceeded (130 kbps), red packets are produced. For further increases in CBR, the number of green packets as well as yellow packets (which were previously increasing) decreases while the number of red packets increases.

The results of these tests indicate that both the token bucket and srTCM can police a traffic stream using a CIR and a CBS. The srTCM however, is more lenient and offers a threshold (extra bucket) that when exceeded, produces non conformant red packets. The token bucket however considers all packets that exceed the CBS as non conformant. The concept of the srTCM is that the host will eventually transmit at a lower rate and allow the extra bucket to be replenished. The results of the trTCM experiment indicate that it can police a traffic stream using a CIR as well as a PIR. The trTCM however, is not as lenient as the srTCM; all packets that exceed its PIR are marked as red. In terms of policing strictness, the trTCM appears to be the strictest followed by the token bucket and then the srTCM.

5.5 Experiment 5

This experiment simulated a scenario where DiffServ is deployed in a real network. The performance of the application flows in DiffServ network can then be compared to the performance in the Best Effort network. The DiffServ network provided three services; the Premium service, the Gold service and the Silver service. The Premium service was created from the EF forwarding class and offers low loss, delay and delay variation. The Gold and Silver services were created from the AF1 and AF2 forwarding classes. The Gold service was better provisioned than the Silver service, such that the flows using the Gold service experience better performance. Hosts 1-4 all transmit voice and video traffic into the network. Each host has two SLAs; one for each application. All voice applications used the Premium service while the video applications used either the Gold or Silver services. Hosts 1 and 3 video applications used the Gold service while Hosts 2 and 4 used the Silver service. The experiment is then repeated with the Best Effort network.

5.5.1 Loss (Figure 112)

From the packet loss graph it can be observed that the DiffServ network's Premium, Gold and Silver services do indeed offer differentiated service (Exp 5.1). The video applications that used the Gold service (Hosts 1 and 3) experience roughly three times less loss than those that used the Silver (Hosts 2 and 4). Also the voice applications that used the Premium service experience no loss as expected. In comparison to Exp 5.2, the Best effort network, the video applications experience varying levels of loss as a result of the 'single queue lock out' effect (proven to be true in experiment 6).

Nevertheless, the video applications in the Best Effort network do not achieve the diverse performance that the DiffServ network provides. When comparing the performance of the voice applications, it can be seen that the DiffServ network's Premium service provides no loss for all voice applications, while under the Best Effort network, a loss of 7-8%.

5.5.2 Delay (Figure 113)

Similar to packet loss, the video applications that used the Gold service have approximately one third the delay experienced by the applications that use the Silver service. In comparison to the Best effort network (Exp 5.2), the delay experienced among the video applications shows little variance as expected.

For voice applications, the DiffServ network's Premium service provides 4-5 times less delay than the Best Effort network as shown in Figure 113. The minimum delay a voice packet can experience in this network was calculated and found to be 16.7ms. The calculation includes serialization and propagation delays of a single voice packet through the network. The voice applications in the DiffServ network all experience roughly 27ms. While queuing delay does contribute to end to end delay, it is most likely not significant in this case since the EF queue never grows large as indicated by the Average queue length of 0.09 packets and queuing delay of 2.6ms (Figure 116 and Figure 117). What is suspected as the cause of this increased delay is the large video packets present in the network. For example, the situation where a large video packet is being serviced by the scheduler (i.e. serialized to the output link) and a voice packet

arrives. The voice packet will have to wait until the video packet completely serialized before it can be serviced, even though it has higher priority. For example, a typical video packet of 1000 bytes has a serialization delay of 13.3ms on the 0.6Mbps link. This can add additional delay to the voice packet.

5.5.3 Delay Variation (Figure 114)

The delay variation graphs show that the video applications that used the Gold service experience lower delay variation than those that used the Silver as expected. In comparison to the Best Effort network, the delay variation varies among the video applications but vary in no organized way. The delay variation of the voice applications are less than 50% smaller in the DiffServ network than the Best Effort network. In both the DiffServ and Best Effort networks, the delay variation across the voice applications is very similar. This is expected since the Premium service provides low delay variation.

5.5.4 Queue statistics (Figure 115, Figure 116, Figure 117)

Once again CoreQueue 2 is analyzed for the DiffServ network. The queue statistics also support the end to end behavior; showing the EF queue with no loss and the AF1 (Gold) and AF2 (Silver) queues with 3.3% and 11%loss respectively. It also shows that the EF queue has the lowest queuing delay and average queue length followed by the AF1 queue and then the AF2 queue as expected.

5.5.5 Other observations

An important observation to note is that the 'average' performance of the video applications in the Best effort network is greater than the DiffServ network. This

occurs since more resources are distributed among the voice applications than the video. In other words, the video applications in the DiffServ network experience greater loss so that the voice applications experience none. This situation clearly demonstrates the concept of quality of service and it implications. That is, QoS does not created additional bandwidth for a network but instead allocates the available bandwidth and/or priority so that some flows get more and some flows get less.

It can also be observed that in terms of loss, delay and delay variation, of the Gold service performs better than the Best Effort service while the Silver service performs worst than the Best Effort service. This also demonstrates the concept of Quality of Service.

Another observation to note is that the voice applications all experience very similar loss, delay and delay variation since no differentiation is made within the Premium service.

The DiffServ network also clearly shows that the Premium service is able to support the performance requirements of the voice traffic by offering a low loss, delay and delay variation service. On the other hand, the Best Effort network fails to this by providing no service Differentiation to the voice traffic. (Szigeti and Hattingh 2004) explain that voice traffic should have no more than 1% packet loss and a one way delay no more than 150ms to maintain voice quality. The loss and one way delay experienced in the Best Effort network is roughly 8% and 120ms. In the DiffServ network, a loss and delay of 0% and 27ms is experienced which is clearly superior.

This experiment also confirms the DiffServ theory that - the flows that belong to a forwarding class have a degree of performance that corresponds to the provisioning and prioritization of that class, and - that no absolute performance guarantees can be made for a flow since per flow reservations are not made.

5.6 Experiment 6 (Figure 118)

This experiment aimed to verify the 'single queue lock out' effect (suggested in experiments 1, 2 and 5) that occurs for video applications in the Best Effort network. Exp 6.1 uses four video applications to send video traffic into the Best Effort network. Exp 6.2 then uses the DiffServ network, where each application is put into a different AF forwarding class and hence uses a different queue. The results of Exp 6.1 clearly show that the video applications do experience diverse loss between them and they do so in structured way. The results of Exp 6.2 show that when the video applications use different queues, they experience similar loss. However these results are not enough to confirm the lock out effect. Therefore a final experiment, Exp 6.3, was performed where Exp 6.1 was repeated (Best Effort) but the application start times were varied such that one application starts a few seconds after another. The results show that the applications experience similar loss as displayed by Exp 6.2 (DiffServ). This confirms the occurrence of the proposed single queue lock out effect. The effect occurs when flows of similar traffic characteristics enter a single queue. As the queue becomes full due to congestion, incoming packets will be dropped. However the

dropping is not equal among the flows but instead seems to favor some flows than others due to their 'timing' of entering the queue. What is suspected to occur is that, the queue will dequeue a number of packets and have extra space to accept more packets. Some flows will then enter the queue and take the available space and thus 'locking' out other flows. This process is then repeated resulting in some flows having significantly higher or lower loss than others as a result of their timing of entering the queue and similar traffic characteristics.

6 Conclusion

This project aimed to investigate the performance benefits of implementing the Differentiated Services architecture as opposed to the Best Effort service. The results of Experiment 1 and 2 conclude that DiffServ is able to provide resource allocation to forwarding classes in terms of bandwidth allocation, priority to bandwidth and drop priority. Experiment 3 proves that DiffServ is able to achieve greater bandwidth utilization than the Best Effort network, while still providing flow isolation to subscribers that are within their traffic profile. Experiment 5 proves that DiffServ is able to provide different levels of service to customers' applications and meet their QoS performance requirements. The results of the experiments conclude that DiffServ is able to provide not exactly Quality of Service but a Class of Service. That is, DiffServ is able to provide better performance for certain subsets or classes of traffic but not to individual flows. The flows that use a forwarding class do not have absolute performance guarantees. DiffServ can provide better performance for a service class but the performance of the flows that use that forwarding class is dependent on the provisioning, prioritization and traffic load of the forwarding class. The experiments conducted also demonstrate the concept of Quality of Service in that some traffic can receive better treatment at the cost of other traffic receiving worst.

In Future work, it is desired to make a number of upgrades to the ns-3 DiffServ implementation. The current implementation only supports six internal queues. The

code points that represent these queues are also hard coded into the DiffServQueue class. This poses a problem for new per hop behaviors. To solve this problem, a vector of queues with a user configurable DSCP to Queue mapping table can be implemented. Another issue is the packet scheduling. Currently the scheduler is hard coded within the class and therefore does not allow for different scheduling schemes. This can be solved by implementing the scheduler in an abstract class and having different scheduling schemes inherit it. Also the current DiffServ implementation only considers a single DS domain. It is desired to upgrade this implementation to support DS regions as well. This can be accomplished by modifying the DiffServFlow class to accept DS code points.

7 References

Almes, G., S. Kalidindi, and M. Zekauskas. "A One-way Delay Metric for IPPM RFC 2697." *Internet Engineering Task Force.* September 1999. http://datatracker.ietf.org/doc/rfc2679/ (accessed January 1, 2011).

—. "A One-way Packet Loss Metric for IPPM RFC 2680." *Internet Engineering Task Force.* September 1999. http://datatracker.ietf.org/doc/rfc2680/ (accessed January 1, 2011).

Armitage, Grenville. *Quality of Service in IP networks.* Lucent Technologies, 2000.

Banks, Jerry, John CarsonII, and Barry Nelson. *Discrete Event System Simulation.* New Jersey: Pearson Education Inc., 2010.

Blake, S., D. Black, M. Carlson, E. Davies, Z. Wang, and W. Weiss. "An Architecture for Differentiated Service RFC 2475." *The Internet Engineering Task Force (IETF).* December 1998. http://datatracker.ietf.org/doc/rfc2475/ (accessed September 2010).

Chao, H. Jonathan, and Xiaolei Guo. *Quality of Service Control in High-Speed Networks.* New York: John Wiley & Sons, 2002.

Choi, Hyoung-Kee, and John O. Limb. *A Behavioral Model of Web Traffic.* Georgia , 1999.

Cisco. *Weighted Random Early Detection (WRED).* http://www.cisco.com/en/US/docs/ios/11_2/feature/guide/wred_gs.html#wp6486 (accessed September 20, 2010).

Davie, B., A. Charny, and J.C.R. Bennett. "An Expedited Forwarding PHB (Per-Hop Behavior) RFC 3246." *The Internet Engineering Task Force (IETF).* March 2002. http://datatracker.ietf.org/doc/rfc3246/ (accessed September 2010).

Demichelis, C., and P. Chimento. "IP Packet Delay Variation Metric for IP Performance Metrics (IPPM) RFC 3393." *Internet Engineering Task Force.* November 2002. http://datatracker.ietf.org/doc/rfc3393/ (accessed January 1, 2011).

Di, Zesong, and H. T. Mouftah. *Performance Evaluation of Per-Hop Forwarding Behaviors in the Diffserv Internet.* Ontario, 2000.

diffserv Directory Reference. http://www.auto-nomos.de/ns2doku/dir_ee9a1c83055b828988a3931fc2ebb9d5.html (accessed January 2011).

Ellis, Juanita, Charles Pursell, and Joy Rahman. *The Convergence of Voice,Video & Network Data.* USA: Elsevier Science, 2003.

Garcia, Leon, and Widjaja. *Communications Networks - Fundamental Concepts.* New York: McGraw-Hill Education, 2001.

HASSAN, Hassan, Jean-Marie GARCIA, and Olivier BRUN. *GENERIC MODELING OF MULTIMEDIA TRAFFIC SOURCES.* France.

Heinanen, J., F. Baker, W. Weiss, and J. Wroclawski. "Assured Forwarding PHB Group RFC 2597." *The Internet Engineering Task Force (IETF).* June 1999. http://datatracker.ietf.org/doc/rfc2597/ (accessed September 2010).

Jacobson, V., K. Nichols, Cisco, and K. Poduri. "An Expedited Forwarding PHB." *The Internet Engineering Task Force (IETF).* 1999. http://tools.ietf.org/html/rfc2598 (accessed September 2010).

Jung, Sangkil, Jaiseung Kwak, and Okhwan Byeon. *Performance Analysis of Queue Scheduling Mechanisms for EF PHB and AF PHB in DiffServ Networks.* Korea, 2002.

Kurose, James, and Keith Ross. *Computer Networking A Top-Down Approach Featuring the Internet.* Addison-Wesley, 2000.

Makkar, Rupinder, Ioannis Lambadaris, Jamal Hadi Salim, Nabil Seddigh, Biswajit Nandy, and Jozef Babiarz. "Empirical Study of Buffer Management Scheme for Diffserv Assured Forwarding PHB." 2000.

Mao, Jianmin, W. Melody Moh, and Belle Wei. *PQWRR Scheduling Algorithm in Supporting of DiffServ.* San Jose, 2001.

Nichols, K., S. Blake, F. Baker, and D. Black. "Definition of the Differentiated Services Field (DS Field) in the IPv4 and IPv6 Headers RFC 2474." *The Internet Engineering Task Force (IETF).* December 1998. http://datatracker.ietf.org/doc/rfc2474/ (accessed September 2010).

Park, Kun I. *QoS in Packet Networks.* Boston: Springer Science + Bussiness Media, Inc., 2005.

Riley, George. *Network Simulation with ns-3.* Georgia , April 12, 2010.

Roberts, Frederick, and Donna Roberts. *Computer Science 2 Topic : Arrays -Shell Sort.* http://mathbits.com/mathbits/compsci/arrays/Shell.htm (accessed March 15, 2011).

Simpson, Wes. *Video Over IP.* Burlington: Elsevier Inc., 2008.

Szigeti, Tim, and Christina Hattingh. *Quality of Service Design Overview.* December 17, 2004. http://www.ciscopress.com/articles/article.asp?p=357102 (accessed March 20th, 2011).

Telecommunication Networks Group. *Trace Files.* http://www.tkn.tu-berlin.de/research/trace/ltvt.html (accessed February 18, 2011).

Vegesna, Srinivas. *IP Quality of Service.* Indianapolis: Cisco Press, 2001.

Wang, Zheng. *Internet QoS: architectures and mechanisms for quality of service.* San Francisco: Morgan Kaufmann Publishers, 2001.

8 Appendices

The Appendix chapter consists of the following sub sections:

Appendix A –Standardized PHBs

Appendix B - Packet Scheduling and Buffer Management algorithms

Appendix C- Metering Algorithms

Appendix D – Token Bucket algorithm code walkthrough

Appendix E- WRED algorithm code walkthrough

Appendix F- Testing of implemented components and multimedia applications

Appendix G - Experiment results

8.1 Appendix A - Standardized Per Hop Behaviors (PHBs)

There are two PHBs that have been standardized for the DiffServ architecture. These are the Assured Forwarding (AF) PHB group and the Expedited Forwarding PHB.

8.1.1 The Assured Forwarding (AF) PHB Group

The Assured Forwarding PHB group allows a DS domain service provider to offer different levels of forwarding assurance to his subscribers. The AF PHB group defines a set of PHBs, each with their individual mapping to a particular DSCP. The AF PHB group is first separated into classes. Then within each AF class there are different drop precedence levels. Each AF class is awarded a specific amount of network forwarding resources (buffer and bandwidth). The drop precedence level within a class determines the level of importance of the packet within the class. Therefore when congestion occurs at a router, the packets of an AF class with lower drop precedence level are less likely to be dropped than one with higher precedence. The forwarding assurance that a packet receives in a DS node (DS compliant node) depends on three factors:

1. The AF class the packet belongs to; how much forwarding resources has been allocated to the class.
2. The current traffic-load the class experiences.
3. The drop precedence level the packet belongs to within the AF class, if congestion should occur

   (Heinanen, et al. 1999)

The AF PHB group consists of N different classes containing M different drop precedence levels. A packet of Class x and drop precedence y is given the code point AF xy.

$$\text{Where } 1 \leq x \leq N \text{ and } 1 \leq y \leq M$$

Currently in general use N = 4 and M = 3. The table below shows the derived recommended code points for this range.

Table 12: AF Recommended Code Points

AF Class	Code point name	Code point value (DSCP)	Ds field value(binary)	Ds field value(Decimal)	Drop precedence level
1X	AF 11	001010	001010 00	40	Low
	AF 12	001100	001100 00	48	Medium
	AF 13	001110	001110 00	52	High
2X	AF 21	010010	010010 00	72	Low
	AF 22	010100	010100 00	80	Medium
	AF 23	010110	010110 00	88	High
3X	AF 31	011010	011010 00	104	Low
	AF 32	011100	011100 00	112	Medium
	AF 33	011110	011110 00	120	High
4X	AF 41	100010	100010 00	136	Low
	AF 42	100100	100100 00	144	Medium
	AF 43	100110	100110 00	152	High

An implementation Assured Forwarding PHB group must satisfy the following requirements:

8.1.1.1 Forwarding Behavior

1. Packets from an AF class must be forwarded independently from packets from other AF classes
2. A minimum amount of Bandwidth and buffer space (forwarding resources) must be allocated to an AF class
3. Packets with lower drop precedence levels must have a higher probability of being forwarded than a packet of a higher drop precedence level.
4. A DS node must accept the three drop precedence levels for each AF class and must provide at least two different drop probabilities.
5. Packets of the same micro-flow and AF class must not be reordered no matter what their drop precedence level may be.

8.1.1.2 Queuing and Discarding Behavior

1. Within each implemented AF class long term congestion must be minimized while allowing for short term congestion as a result of bursts of traffic. For this behavior the use of an Active Queue Management (AQM) algorithm is needed.
2. The response to long term congestion must be to drop packets, while the response to short term congestion is to queue packets.
3. Packet flows with identical long term packet rates but different short term bursts shapes should have the same probability of a packet being dropped. This can be achieved by using randomness in the AQM algorithm.
4. The AQM algorithm selected must provide equal treatment to all packets of the same AF class and drop precedence level

5. For each AF class and drop precedence level, the AQM parameters must be independently configurable

8.1.1.3 Traffic conditioning behavior

1. Any Traffic conditioning actions performed on AF traffic must not cause packets of the same micro-flow to be reordered. (Heinanen, et al. 1999)

8.1.2 The Expedited Forwarding PHB

The intention of the Expedited Forwarding Per-Hop Behavior is to build a low loss, low latency, low jitter, assured bandwidth service for subscribers from end to end. This service is also called the Premium service, and would appear to its subscribers as a virtual leased line (VLL). The queuing delay in routers is a partly responsible for the amount of loss, latency, and jitter traffic experiences from the network. Hence to provide such a service described above would require a minimum queuing delay. Queues are formed when the rate at which packets arrive at an output link is greater than their departure rate. Therefore to minimize queuing, the maximum arrival rate of packets should always be less than the minimum departure rate.

8.1.2.1 Forwarding Behavior

1. The departure rate of packets from a DS node should equal or exceed a configurable rate.
2. The departure rate of EF packets should be independent of current load of the node.

3. The minimum configured departure rate must be settable by the network administrator.
4. If the mechanism chosen to implement the EF PHB preempts other traffic unlimitedly, then the implementation must include a rate limiter to minimize the damage EF packets could have on other traffic.
5. EF packets that exceed the configured rate must be discarded

The recommended code point for the Expedited Forwarding PHB is 101110.(Jacobson, et al. 1999)

8.1.3 The Default PHB

The Default PHB is the standard best effort service provided by the network and must be available in DS- complaint nodes. Packets belong to this aggregate when no agreements are in effect. Packets of this PHB are sent into the network without any conditioning and the network will try its best to deliver as many of these packets as it can. The Recommended code point for this PHB is '000000'. (Nichols, et al. 1998)

8.2 Appendix B- Scheduling and Buffer management algorithms

8.2.1 Packet scheduling algorithm 1 - First in – First out (FIFO)

The FIFO algorithm selects packets from the queue to transmit on the outgoing link, based on the same order they arrived in the queue.

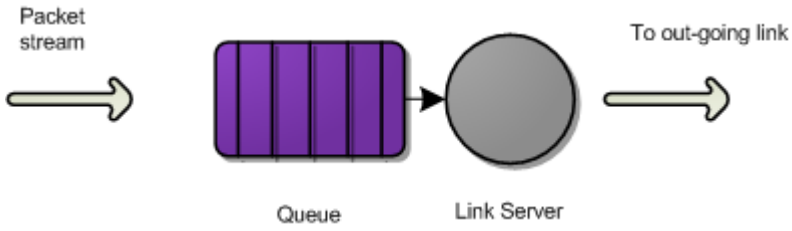

Figure 50: FIFO scheduling

Source: (Kurose and Ross 2000)

8.2.2 Packet Scheduling algorithm 2 – Weighted Round Robin (WRR)

The WWR algorithm selects packets from multiple queues to transmit on the outgoing link. The algorithm starts servicing packets from the first queue, then moves on to the second queue and then the third and so forth, and then repeats the process. However the amount of time the algorithm spends on a certain queue depends on that queue's 'weight'. A queue with a larger weight than another queue will have a longer servicing time than the other and hence receive a larger share of the link's bandwidth. Also if the algorithm follows a 'work conserving' discipline, it never allows the link to remain idle when there is a queue containing packets for transmission. That is, if it comes across an empty queue, it immediately services the next queue in the sequence. (Kurose and Ross 2000)

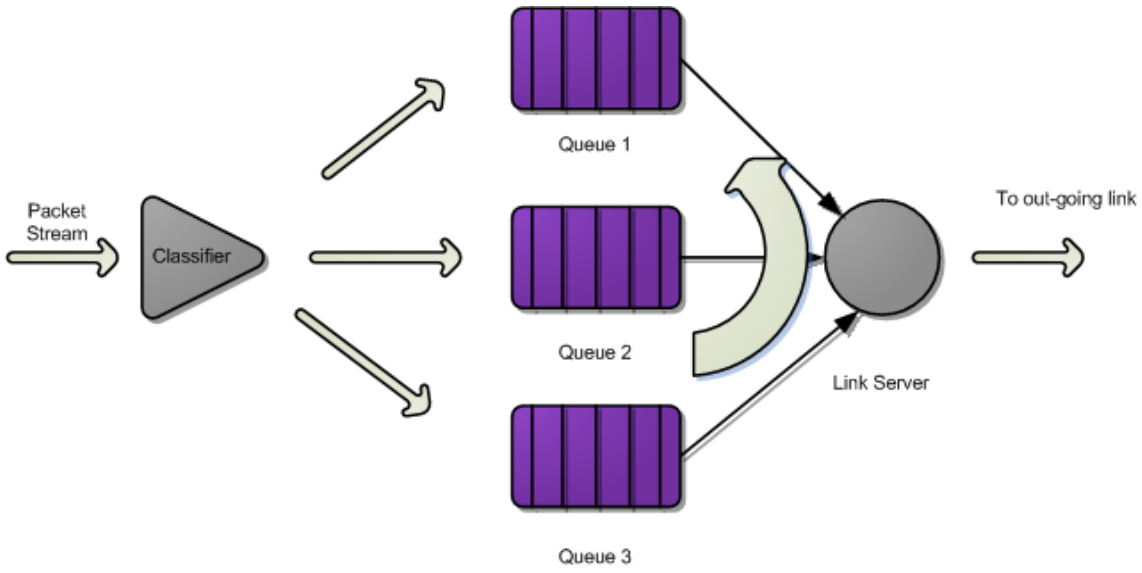

Figure 51: WWR scheduling

Source: (Kurose and Ross 2000)

8.2.3 Packet Scheduling algorithm 3 – Priority Queue (PQ)

The Priority Queue scheduling algorithm also selects packets from multiple queues to transmit on the outgoing link. Each queue that the scheduler would service has a priority that is relative to the other queues. A queue of higher priority will always be serviced before a queue of lower priority. Therefore when choosing a packet to transmit, the scheduler will service the queue of highest priority that has a non empty queue. (Kurose and Ross 2000)

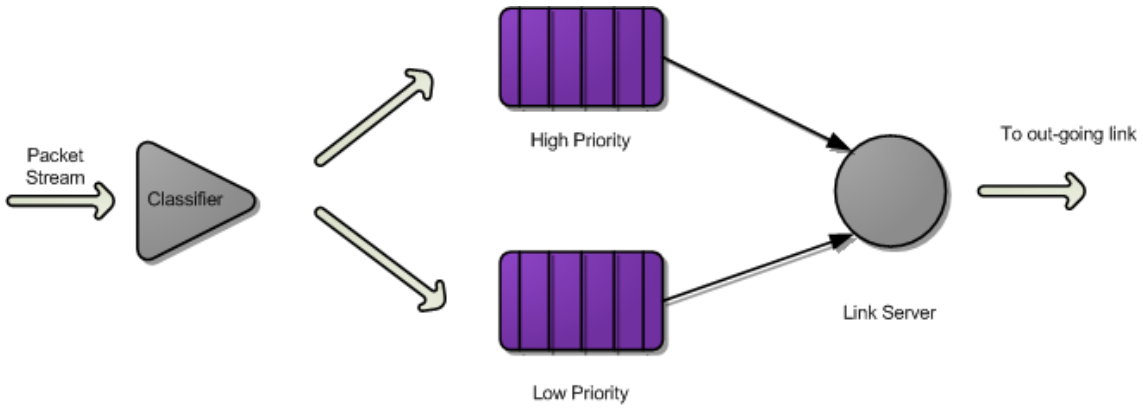

Figure 52: PQ scheduling

Source: (Kurose and Ross 2000)

8.2.4 Buffer Management mechanism 2 –Drop Tail

The Drop Tail (also called Tail Drop) buffer management mechanism drops all incoming packets when the queue is full. Only when the queue has 'de-queued' some packets and has the capacity to accept more, are packets allowed into the queue. A queue using this algorithm does not differentiate among the packets it drops.

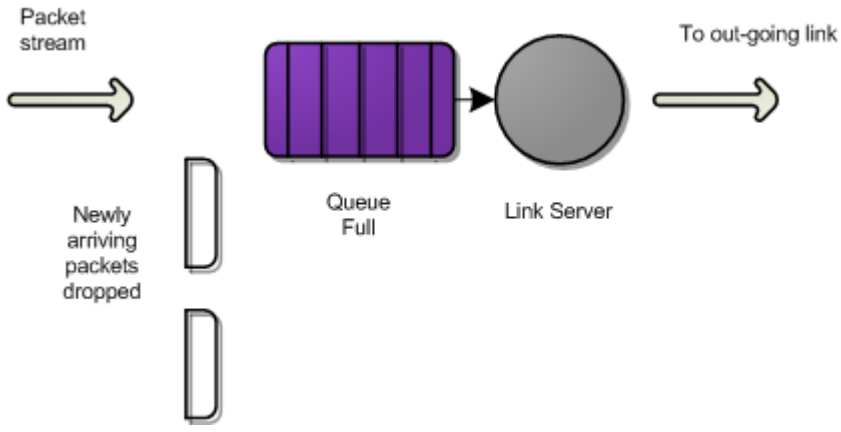

Figure 53: Buffer Management Drop Tail

8.2.5 Buffer Management mechanism 1 - The Weighted Random Early Detection algorithm (WRED)

The Random Early Detection algorithm includes the following parameters:

i. The Average Queue Length (packets)

ii. A Minimum Threshold (packets)

iii. A Maximum Threshold (packets)

iv. Exponential Weighing Constant

v. The Maximum Probability of packet drop

The RED algorithm is described as follows. When a packet arrives at the queue, the average queue length is calculated. If the average queue length is lower than the minimum threshold, the packet is not dropped. If the average queue length is between the minimum and maximum threshold, the packets is dropped with a linearly rising probability. If the average queue length is at the maximum threshold then the incoming packet is dropped. (Makkar, et al. 2000)

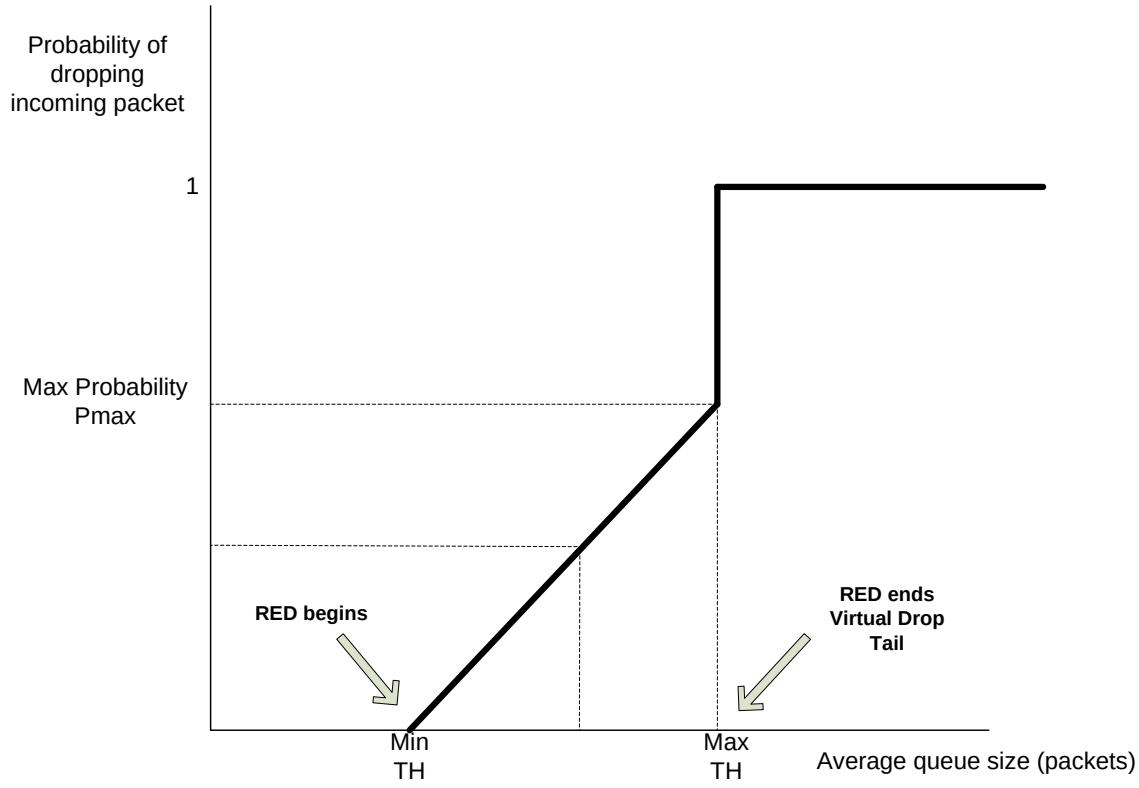

Figure 54: Random early detection AQM mechanism

The WRED algorithm is an extension of RED that allows for different RED parameters or profiles to be applied to different packets in the same queue. Hence different drop probabilities are calculated for packets of different levels of drop precedence. However the WRED algorithm calculates a single Average Queue Length for all drop precedence levels (Makkar, et al. 2000).

In the WRED implementation shown below, each packet color has a Minimum and Maximum threshold and a Maximum probability of packet drop. The packet colors represent the three drop precedence (DP) levels. Green, Yellow, and Red represents DP0, DP1 and DP2 respectively. Lower drop precedence levels are assigned a lower

Maximum Probability of packet drop. Hence DP0 has the lowest Maximum Probability of being dropped while DP2 has the highest. In other words, a lower drop precedence level has a less aggressive RED profile than a higher drop precedence level.

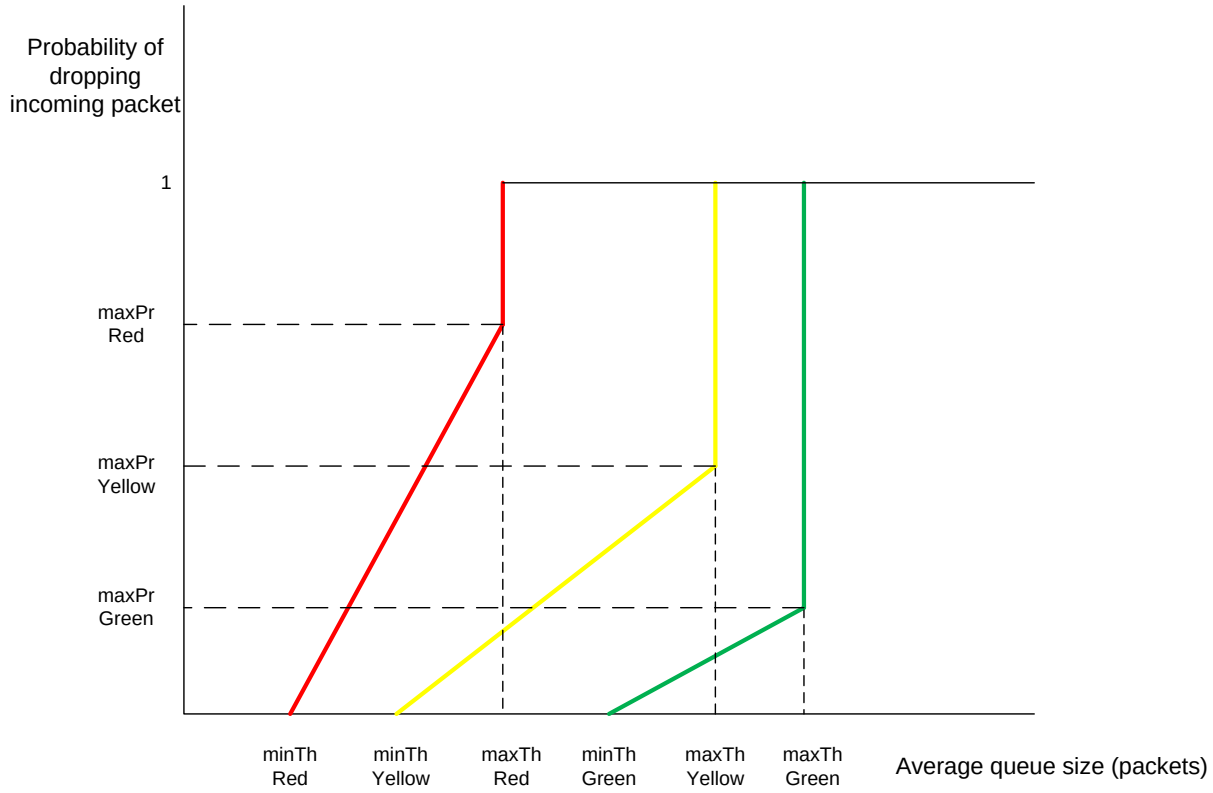

Figure 55: WRED Buffer management mechanism (partially overlapped)

Source: (Makkar, et al. 2000)

8.2.5.1 Calculating the Average Queue Length

The average queue length is calculated using an exponential weighted average of the current queue length.

$$\text{Average Queue Length} = \left[\text{Old AverageQueueLength} \times \left(1 - \frac{1}{2}^{n}\right)\right] + \left[\text{CurrentQueueLength} \times \left(\frac{1}{2}^{n}\right)\right]$$

Where n is the exponential weight and can be configured by the user. The greater the value of the n, the more important the previous average and the less important the current queue length becomes. However if n is chosen too high, packets will be dropped or en-queued as if WRED was not implemented. If chosen too low WRED will over-react to bursts of traffic and may unnecessarily drop packets (Cisco n.d.).

8.3 Appendix C - Metering algorithms

8.3.1 Single token bucket algorithm

The Single Token Bucket specifies two parameters:

i. The Committed Information Rate (CIR) – The rate at which tokens are added to the bucket

ii. The Committed Burst Size (CBS) – The number of tokens the bucket can hold

Where tokens are defined in bytes

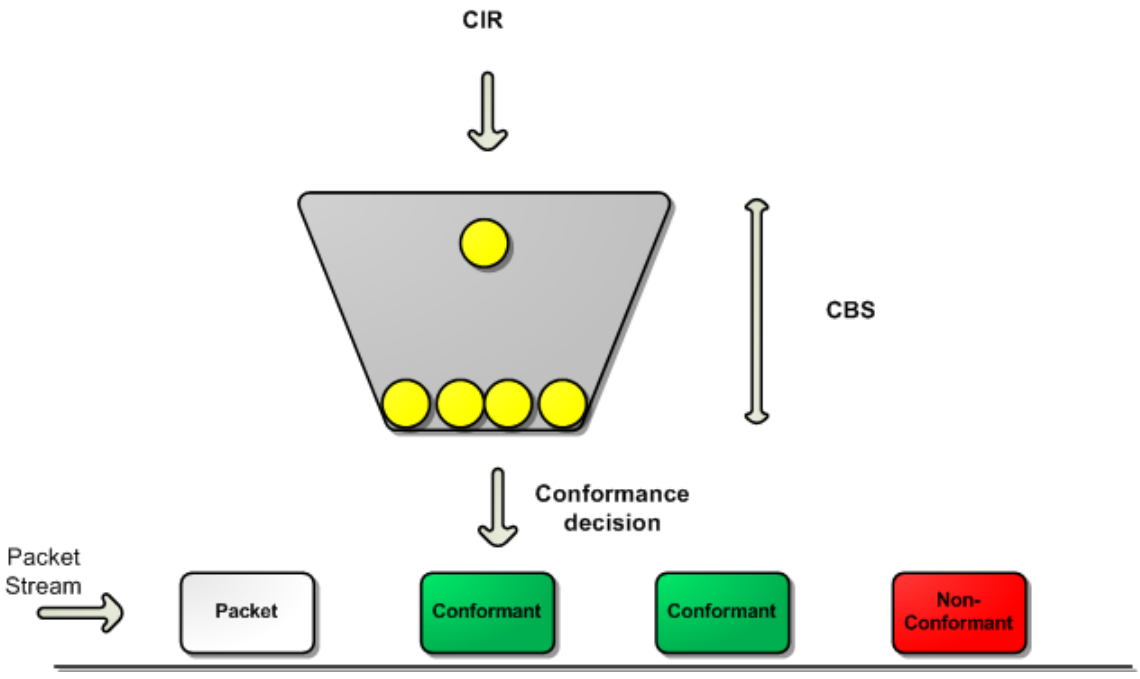

Figure 56: Single Token Bucket algorithm

- Tokens are generated at the Committed Information Rate and stored in a bucket of size CBS

- When the bucket is full, newly generated tokens are discarded
- When a packet arrives, a comparison is done on the size of the packet and the number of tokens available in the bucket
- If the bucket has enough tokens corresponding to the size of the packet, the packet is deemed as conformant. Token are removed from the bucket of equivalent amount
- If the bucket is empty or insufficient tokens are available corresponding to the size of the packet, the packet is deemed as non-conformant and no tokens are removed from the bucket.

8.3.2 Single Rate Three-color Marker metering algorithm

The srTCM specifies three parameters:

i. The Committed Information Rate (CIR) – The rate at which tokens are added to the committed bucket
ii. The Committed Burst Size (CBS) – The number of tokens the committed bucket can hold
iii. The Excess Burst Size (EBS) – The number of tokens the excess bucket can hold

Where tokens are defined in bytes

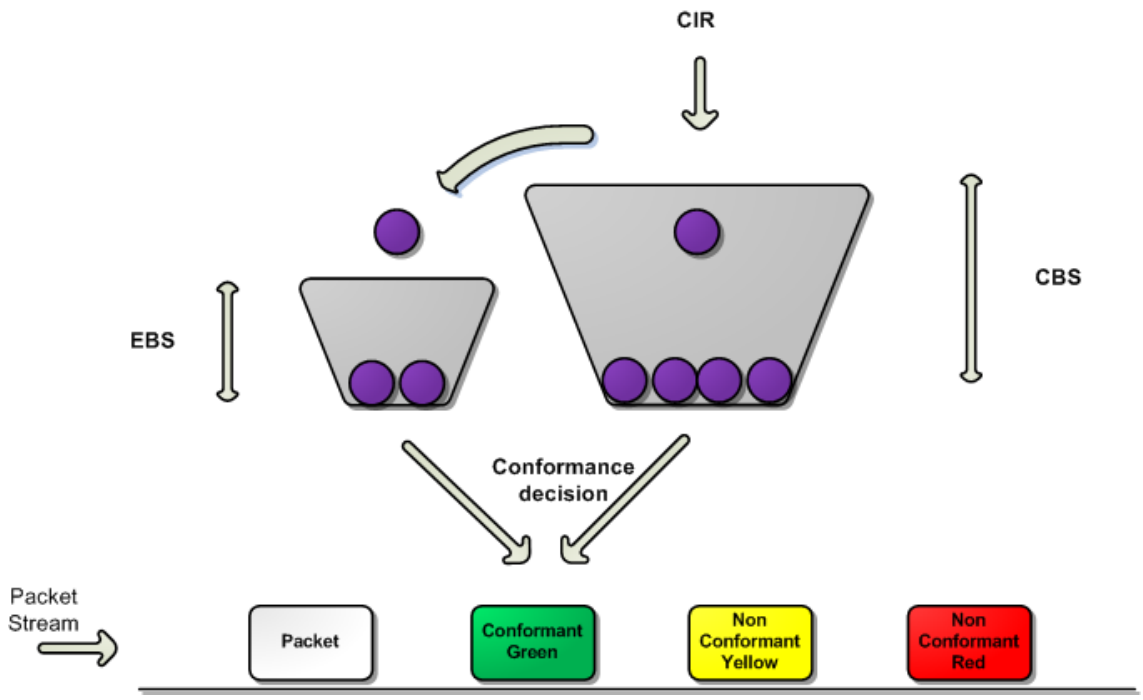

Figure 57: srTCM algorithm

The srTCM meter is able to operate in one of two modes; color-aware and color-blind. In the color-aware mode the meter assumes that some entity has pre-colored the incoming packets and takes the color of the packet as a factor in determining the new color of the packet. In color-blind mode, the meter considers all incoming packets to be un-colored. Since for this simulation we assume that all incoming packets to the meter will be un-colored, we choose for the srTCM to operate in the color-blind mode. The srTCM meters a packet stream and produces three conformance levels; conformant-green, non conformant-yellow and non conformant-red.

- Tokens are generated at the Committed Information Rate and stored in the committed bucket of size CBS

- When the committed bucket is full, newly generated tokens spill over into the excess bucket
- When a packet arrives, a comparison is done on the size of the packet and the number of tokens available in the buckets
- If the committed bucket has enough tokens corresponding to the size of the packet, the packet is deemed as conformant green. Token are removed from the committed bucket of equivalent amount
- If the committed bucket is empty or insufficient tokens are available corresponding to the size of the packet and the excess bucket has sufficient tokens, the packet is deemed as non-conformant yellow and tokens are removed from the excess bucket corresponding to the size of the packet
- If the committed bucket is empty or insufficient tokens are available corresponding to the size of the packet and the excess bucket has insufficient tokens, the packet is deemed as non-conformant red and no tokens are removed from the committed and excess buckets.

8.3.3 Two Rate Three-color Marker metering algorithm

The trTCM specifies three parameters:

i. The Committed Information Rate (CIR) – The rate at which tokens are added to the committed bucket

ii. The Committed Burst Size (CBS) – The number of tokens the committed bucket can hold

iii. The Peak Information Rate (PIR) – The rate at which tokens are added to the peak bucket

iv. The Peak Burst Size (EBS) – The number of tokens the peak bucket can hold

Where tokens are defined in bytes

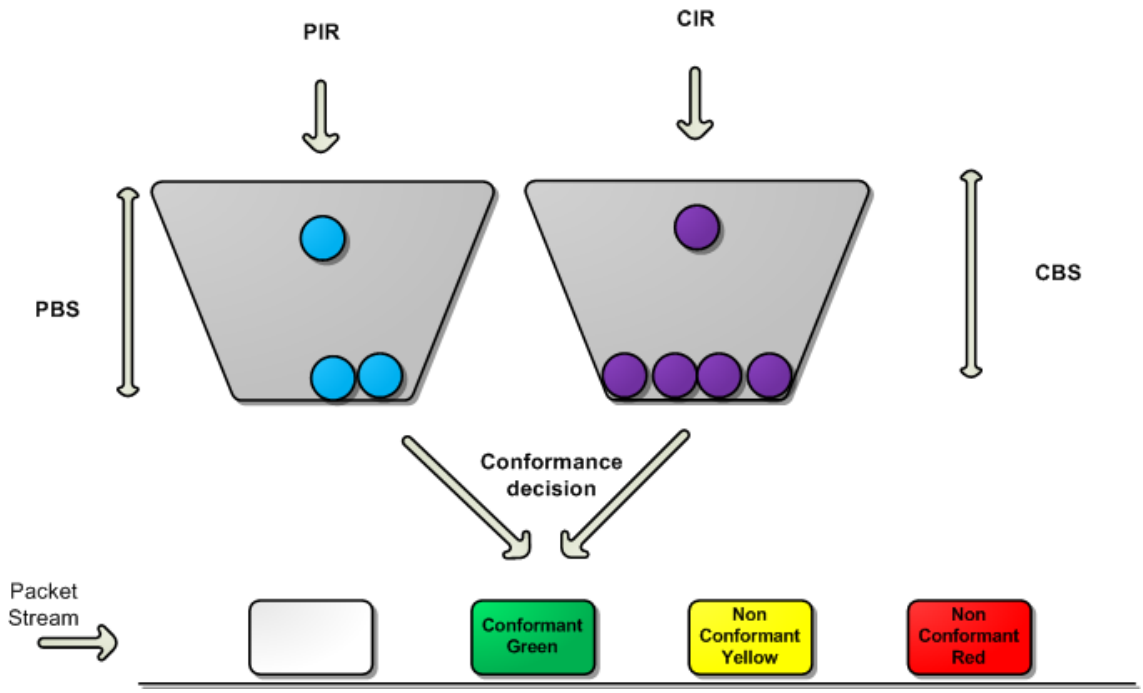

Figure 58: trTCM metering

The trTCM meter is able to operate in one of two modes; color-aware and color-blind. In the color-aware mode the meter assumes that some entity has pre-colored the incoming packets and takes the color of the packet as a factor in determining the new color of the packet. In color-blind mode, the meter considers all incoming packets to be un-colored. Since for this simulation we assume that all incoming packets to the

meter will be un-colored, we choose for the trTCM to operate in the color-blind mode. The trTCM meters a packet stream and produces three conformance levels; conformant-green, non conformant-yellow and non conformant-red.

- Tokens are generated at the Committed Information Rate and stored in the committed bucket of size CBS
- Tokens are also generated at the Peak Information Rate and stored in the peak bucket of size PBS
- When a packet arrives, a comparison is done on the size of the packet and the number of tokens available in the buckets
- If the peak bucket does not have enough tokens corresponding to the size of the packet, the packet is deemed non- conformant red and no tokens are removed from either bucket
- If the peak bucket has sufficient tokens corresponding to the size of the packet and the committed bucket does not, the packet is deemed non- conformant yellow and tokens are removed from the peak bucket of equivalent amount
- If the peak bucket and committed bucket has sufficient tokens corresponding to the size of the packet, the packet is deemed conformant green and tokens are removed from the peak bucket and committed bucket of equivalent amount

8.4 Appendix D - Token Bucket Algorithm Code Walkthrough

The Token Bucket meter first accesses the MeterSpec from the SLA the DiffServSla the packet belongs to. It retrieves the traffic profile and the token bucket state variables such as the committed information rate and the current state of the committed bucket. It then executes the token bucket algorithm.

```
int TokenBucket :: MeterPacket(Ptr<Packet> p, Ptr<DiffServSla> SLA)
{
//Getting info from SLA
   m_lastPacketArrivalTime = (SLA->m_mSpec).lastPacketArrivalTime;
   m_committedBucketSize  = (SLA->m_mSpec).committedBucketSize;
   m_cBS = (SLA->m_mSpec).cBS;
   m_cIR = (SLA->m_mSpec).cIR;
```

Figure 59: Token Bucket - retrieving traffic profile and meter state

The token bucket algorithm can be implemented in one of two ways; you could continuously update the token bucket at regular intervals or you could only update the token bucket when a packet arrives. The second method was used since it is much more efficient than the first.

The token bucket algorithm can be separated into three stages;

1. token calculation
2. updating the bucket size
3. conformance testing

1- Token calculation

The token bucket algorithm retrieves the current simulation time and compares it with the last time a packet arrived that belonged to this same SLA. The Simulator class

keeps track of the time since the simulation began and can be accessed using the member function; 'Simulator::Now ()'

The time difference is calculated by subtracting the two time values. The meter uses this time difference together with the CIR to calculate how many bytes to add to the bucket. The CIR is divided by 8 to get the figure in bytes, since it was intended that the user would specify the CIR in bits per second. The current packet arrival time is now set as the last packet arrival time and with the DiffServSla object the packet belongs to.

```
// Calculating bytes to add to bucket
m_currentPacketArrivalTime = (Simulator::Now()).GetSeconds();

m_bytesToAdd = (m_currentPacketArrivalTime - m_lastPacketArrivalTime)
* (m_cIR/8);

m_lastPacketArrivalTime = m_currentPacketArrivalTime;

(SLA->m_mSpec).lastPacketArrivalTime = m_lastPacketArrivalTime;
```

Figure 60: Token Bucket - step 1 – Token calculation

2- *Updating the bucket size*

The number of bytes that were calculated is then added to the bucket; however the bucket size must not exceed the Committed Burst size. If this should occur, the extra bytes are discarded. The new bucket size is updated with the DiffServSla object the packet belongs to.

```
//Adding bytes to bucket
if (m_committedBucketSize + m_bytesToAdd >= m_cBS)
    {
       m_committedBucketSize = m_cBS; }
else {
       m_committedBucketSize = m_committedBucketSize + m_bytesToAdd;}
(SLA->m_mSpec).committedBucketSize = m_committedBucketSize;
```

Figure 61: Token Bucket - step 2 - updating the bucket size

3- *Conformance testing*

The size of the packet is found using a function of the Packet class: 'Packet::GetSize', and compared with the number of bytes in the bucket. If the bucket has sufficient bytes, the packet is conformant and a number of bytes equal to the size of the packet are removed from the bucket Otherwise the packet is non conformant and no bytes are removed.

If the packet is conformant the meter returns '1'. If the packet is non conformant the meter returns '2'. The new committed bucket size is updated with the DiffServSla object the packet belongs to.

```
//Getting the size of the packet in bytes
PppHeader pppHeader;
Ptr<Packet> q = p->Copy ();
q->RemoveHeader (pppHeader);
 m_packetSize = q->GetSize();

//Testing conformance
if ( m_committedBucketSize - m_packetSize >= 0)
    {
      m_committedBucketSize = m_committedBucketSize - m_packetSize;
      (SLA->m_mSpec).committedBucketSize = m_committedBucketSize;
     return conformanceLevel_1;}
else
    {
       return conformanceLevel 2;}
```

Figure 62: Token Bucket - step 3 – conformance testing

8.5 Appendix E - (WRED) algorithm Walkthrough

The WRED algorithm was implemented in class WRED that inherits from DiffServAQM. Its 'DoAQM' function is called in each AF class Enqueue function. It returns either 1 or 0 indicating whether the packet should be enqueued or not. The WRED algorithm consists of the following steps:

1- Calculating the new average queue size of the queue the packet belongs to

The average queue size of the queue the packet belongs to is updated; AF1, AF2, AF3 or AF4. The exponential constant is a user configurable value for each queue.

```
bool WRED:: DoAQM (int currentQueueSize, int DS)
{

NS_LOG_INFO("WRED:");

//Updating Average Queue size

//AF1

if( (DS == 40) || (DS == 48) || (DS == 56) )
{

m_AF1_averageQueueSize = m_AF1_averageQueueSize

* ( 1-(pow(0.5,m_AF1_exponentialWeight) ) )+

(currentQueueSize *  pow(0.5,m_AF1_exponentialWeight) );
```

Figure 63: WRED - step 1 - Average queue size calculation

2- Determining the packet drop probability of the current packet

The drop probability of the packet (for 0 ≤ drop probability ≤ 100) is calculated using the min and max thresholds and max drop probability of the packet color, of the AF queue the packet belongs to. The thresholds and drop probabilities for each packet color for each AF queue is configured by the user.

```
switch(m_DS)
{
//AF1

case 40:

NS_LOG_INFO("AF1 GREEN packet");

if (m_AF1_averageQueueSize <= m_AF1_minThresholdGreen)
  {
    m_packetDropProbability = 0;
  }

if (m_AF1_averageQueueSize >= m_AF1_maxThresholdGreen)
  {
    m_packetDropProbability = 100;
  }

if((m_AF1_minThresholdGreen< m_AF1_averageQueueSize)&&

   (m_AF1_averageQueueSize < m_AF1_maxThresholdGreen))
  {
m_gradient = (m_AF1_maxThresholdGreen - m_AF1_minThresholdGreen)/

m_AF1_maxProbabilityGreen;

m_packetDropProbability = (m_AF1_averageQueueSize - m_AF1_minThresholdGreen)/

m_gradient;

   NS_LOG_INFO("AF1 Green gradient :" <<m_gradient);
  }
break;
```

Figure 64: WRED - step 2 - Drop probability calculation

3- Determining if the packet should be enqueued or dropped

Now that the packet drop probability has been calculated it can now be used to determine if the packet should be dropped. First a random number generator generates a random number from 1 to 100. The probability that the calculated packet drop probability is less than or equal to the random number is equal to the packet drop probability. For example if the packet drop probability is 25, then the probability that the random number will be less than equal to 25 is 25%. If the random number is less than equal to 25 then the function returns 'true' to the Enqueue function for the packet to be dropped. If the random number is greater than the packet drop probability then the function returns 'false' for the packet to be enqueued.

```
int randomInt = RandomNumberGenerator(1,100);

NS_LOG_INFO("Random number: "<< randomInt);


if(randomInt <=  m_packetDropProbability)
  {
  NS_LOG_INFO("Packet to be dropped--returning false");
  return false;
  }

else
 {
 NS_LOG_INFO("Packet to be enqueued--returning true");
 return true;
 }
```

Figure 65: WRED - step 3- Determining enqueue or drop

8.6 Appendix F- Testing Implemented Components and Applications

This section involves the testing of all implemented components as well as the modeled applications.

8.6.1 Recording results

To ensure that the implemented components function as desired; the simulation output must be recorded and verified. To display the output of the simulation, the following two methods were available by the simulator:

i. The NS-3 logging module

ii. Pcap traces

8.6.2 Ns-3 Logging module

This method simply prints text to the screen similar to using the 'cout' operator in C++. However the ns-3 logging has a few advantages over using cout. The user is able to turn on and off the output messages directly from the main script without changing any of the source code. With cout messages, the messages would always print to the screen even if the information was not desired. The logging module takes another step further and adds different levels of logging messages. Therefore the user is able to have greater control over what messages actually appear.

The logging messages can be used to indicate when a particular path of code is being executed, and to display the value of a variable at a point in time.

8.6.3 Pcap traces

Ns-3 device helpers allow the user to create packet trace files in the Pcap format. These files can then be viewed with a network packet analyzer program such as Wire-Shark. A trace file can be created for every network device on every node in the network topology. In this way the packets sent by applications can be tracked throughout the network.

Pcap traces can be used to identify the nodes where packets were dropped from each flow. They can also be used to ensure that the network topology was setup correctly. Using Wire-Shark, packet filters can be created to observe specific flows that transverse a node. Traces can also show the contents of the packets themselves; which would be useful in identifying if a packet had been marked.

8.6.4 Test Network Topology

For testing of the components, a network topology needed to be constructed. The network is made relatively simple since only testing is being performed (Figure 66). Node 0 is used as a host node and generates traffic. Node 1 is used as an Edge node where an Edge DiffServQueue will be installed. Node 2 is the receiving Host node that receives all traffic sent by Node 0.

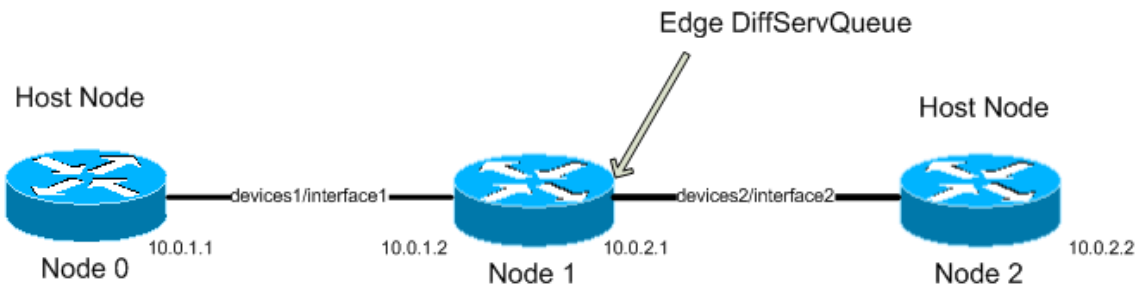

Figure 66: Test Network Topology

The DiffServ components that will be tested include:

1. The MF Classifier and SLA database
2. The Token Bucket meter
3. The srTCM meter
4. The trTCM meter
5. The Enforcer (Marker and Dropper)
6. The Weighted Round Robin and Priority Queue Scheduling mechanisms
7. The Weighted Random Early Drop AQM mechanism

Additional:

1. StatCollector

The Applications to be tested include:

1. The VoIP application
2. The video streaming application

8.6.5 Testing the MF classifier and the SLA database

Four on/off applications were created on Node 0 and were configured to transmit to Node 2. Therefore four applications flows were setup, each with a unique source and destination address and source and destination port numbers shown in the table below.

Table 13: Application configuration for MF classifier test

On/off Application #	Source IP address	Destination IP address	Source port	Destination port
1	10.0.1.1	10.0.2.2	N/A	5111
2	10.0.1.1	10.0.2.2	N/A	5222
3	10.0.1.1	10.0.2.2	N/A	5333
4	10.0.1.1	10.0.2.2	N/A	5444

For this test the data rate and other application specifications are irrelevant; only the MF classifier's ability to identify SLAs is being tested. A SLA was constructed for three of the four application flows. The application source port number was unable to be configured therefore this value was not used for classification. The diagram below shows how the SLA database is configured.

```
//Flow Setup

Ptr<DiffServFlow> flow1Ptr = CreateObject<DiffServFlow> (001,"10.0.1.1","10.0.2.2",0,5111);
Ptr<DiffServFlow> flow2Ptr = CreateObject<DiffServFlow> (002,"10.0.1.1","10.0.2.2",0,5222);
Ptr<DiffServFlow> flow3Ptr = CreateObject<DiffServFlow> (003,"10.0.1.1","10.0.2.2",0,5333);
Ptr<DiffServFlow> flow4Ptr = CreateObject<DiffServFlow> (004,"10.0.1.1","10.0.2.2",0,5444);

//SLA Setup

Ptr<DiffServSla> sla1Ptr = CreateObject<DiffServSla>(001,cSpec1,mSpec2);
Ptr<DiffServSla> sla2Ptr = CreateObject<DiffServSla>(002,cSpec2,mSpec2);
Ptr<DiffServSla> sla3Ptr = CreateObject<DiffServSla>(003,cSpec3,mSpec2);
Ptr<DiffServSla> sla4Ptr = CreateObject<DiffServSla>(004,cSpec4,mSpec2);


flow1Ptr->SetSla(sla1Ptr);
flow2Ptr->SetSla(sla2Ptr);
flow3Ptr->SetSla(sla3Ptr);
flow4Ptr->SetSla(sla4Ptr);

vector< Ptr<DiffServFlow> > myFlowVector;
myFlowVector.push_back(flow1Ptr);
myFlowVector.push_back(flow2Ptr);
myFlowVector.push_back(flow3Ptr);
//myFlowVector.push_back(flow4Ptr);

DiffServQueue::SetDiffServFlows(myFlowVector);
q1->SetQueueMode("Edge");
```

Figure 67: SLA configuration for MF classifier test

The logging module was used to display the results of this test. When a packet enters the Edge DiffServQueue it will be identified as belonging to a SLA or not. For a packet from each application, the output printed to the screen using the logging module is shown in the diagram below. The highlighted segments are the destination port number and the response of the MF Classifier.

```
DiffServQueue_Enqueue  QueueId: 0 Queue Mode: Edge
ns3::PppHeader (Point-to-Point Protocol: IP (0x0021)) ns3::Ipv4Header (tos
0x0 ttl 63 id 0 protocol 17 offset 0 flags [none] length: 1052 10.0.1.1 > 1
0.0.2.2) ns3::UdpHeader (length: 1032 49153 > 5111) Payload (size=1024)
ClassificationAndConditionining:
STEP 1: MF Classifier:
flowId: 1
slaId: 1
DiffServQueue_Enqueue  QueueId: 0 Queue Mode: Edge
ns3::PppHeader (Point-to-Point Protocol: IP (0x0021)) ns3::Ipv4Header (tos
0x0 ttl 63 id 1 protocol 17 offset 0 flags [none] length: 1052 10.0.1.1 > 1
0.0.2.2) ns3::UdpHeader (length: 1032 49154 > 5222) Payload (size=1024)
ClassificationAndConditionining:
STEP 1: MF Classifier:
flowId: 2
slaId: 2
DiffServQueue_Enqueue  QueueId: 0 Queue Mode: Edge
ns3::PppHeader (Point-to-Point Protocol: IP (0x0021)) ns3::Ipv4Header (tos
0x0 ttl 63 id 2 protocol 17 offset 0 flags [none] length: 1052 10.0.1.1 > 1
0.0.2.2) ns3::UdpHeader (length: 1032 49155 > 5333) Payload (size=1024)
ClassificationAndConditionining:
STEP 1: MF Classifier:
flowId: 3
slaId: 3
DiffServQueue_Enqueue  QueueId: 0 Queue Mode: Edge
ns3::PppHeader (Point-to-Point Protocol: IP (0x0021)) ns3::Ipv4Header (tos
0x0 ttl 63 id 3 protocol 17 offset 0 flags [none] length: 1052 10.0.1.1 > 1
0.0.2.2) ns3::UdpHeader (length: 1032 49156 > 5444) Payload (size=1024)
ClassificationAndConditionining:
STEP 1: MF Classifier:
No classification found--packet does not belong to a SLA
```

Figure 68: Simulation output for MF classifier test

As can be seen in the diagram above, the MF Classifier identified the SLA of the first three applications. The fourth application did not belong to a SLA and therefore the MF Classifier could not identify a SLA for its packets.

8.6.6 Testing the Metering algorithms

An on/off application was created on node 0 and was configured to transmit to node 2. The application was configured as shown in the table below. Note that the application was always on and acted like a CBR source (off time = 0). The application was configured to only send 10 packets.

Table 14: Application configuration for meter tests

Node	On/off Application #	Transport Protocol	Source IP address	Destination IP address	Source port	Destination port	Application CBR Mbps	Packet size (bytes)
0	1	UDP	10.0.1.1	10.0.2.2	N/A	5111	1	1024

8.6.6.1 Packet inter arrival time calculation

To compare hand calculated values with the simulation output, the packet inter-arrival time must be known. For an application that generates CBR data at 1 Mbps and at a packet size of 1024 bytes, the inter arrival time was calculated to be 0.008192 seconds.

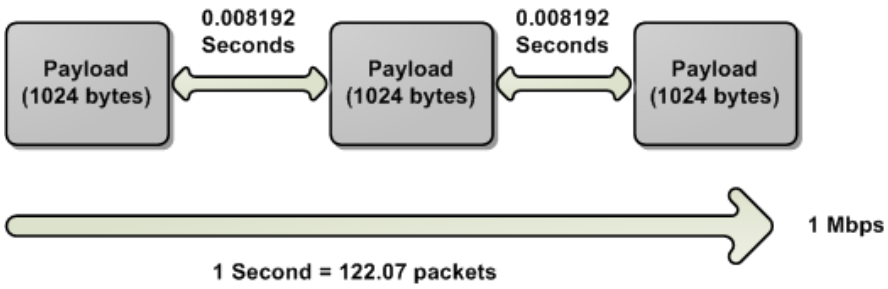

Figure 69: On/Off application data rate without IP and UDP headers

8.6.6.2 Application data rate at the IP layer

However the on/off application generates this data at the application layer as shown in the diagram above. Therefore this data is the packet payload. When a packet leaves the host node, the packet will have a transport layer header and an IP header. The application used the UDP transport protocol, which added an extra 8 bytes per packet. The IP header added an extra 20 bytes. Therefore the resultant data rate was calculated to be 128,417.6 bytes per second. Therefore the data rate became 128,417.6 bytes per second or 1,027,341.1 bps or 1.02734 Mbps. The actual data rate of the application at the Network layer was 1.02734 Mbps. Therefore the protocol layers added an overhead of 0.02734 Mbps.

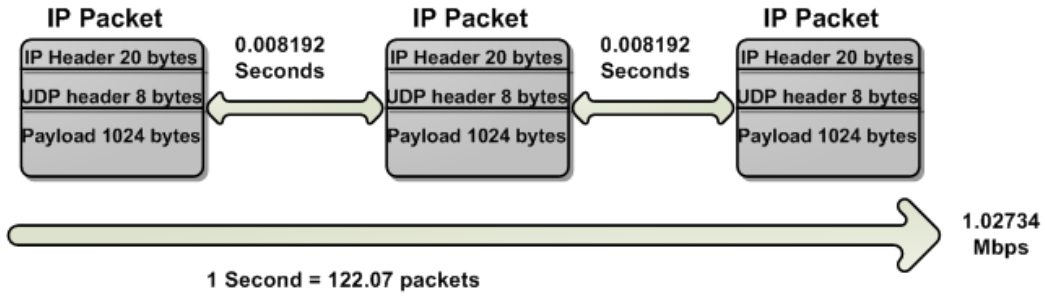

Figure 70: On/Off application data rate with IP and UDP headers

The packet size at the IP layer was 1052 bytes and the inter-arrival time was 0.008192 seconds. The application generated packets at this constant bit rate (CBR). Using this application the metering algorithms were tested; the Token Bucket, the srTCM and the trTCM. The application sent only 10 packets and for these 10 packets the bucket sizes, token calculations and conformance levels were recorded from the simulation using

the logging module. These values were then compared with 'hand calculated' values. For each test, a traffic profile was provided for the metering algorithm to use.

To perform the hand calculations, the traffic profile given to the meter was used. Using the CIR or PIR etc specified in the traffic profile and the inter-arrival time previously calculated, the amount of bytes/token to be added was calculated for each incoming packet. Similarly, using the previously calculated packet size, the number of tokens to be removed from the bucket was known. This process was carried out for all ten packets and for all three metering algorithms.

8.6.6.3 Testing the Token Bucket Meter

A SLA was constructed for this application flow as shown in the diagram below. The SLA traffic profile was set to a CIR of 0.5 Mbps and a CBS of 2000 bytes.

```
MeterSpec mSpec1;
mSpec1.meterID = "TokenBucket";
mSpec1.cIR = 500000;
mSpec1.cBS = 2000;
mSpec1.eBS = 0;
mSpec1.pIR = 0;
mSpec1.pBS = 0;

//SLA Setup

Ptr<DiffServSla> sla1Ptr = CreateObject<DiffServSla>(001,cSpec1,mSpec1);
```

Figure 71: SLA configuration for the Token Bucket test

The table below shows the calculated results that the simulation results were compared to. The results obtained from the logging module were exactly as calculated.

T able 15: Expected results for the Token Bucket test

Packet no	Bucket size (bytes)	Bucket size after adding bytes	Conformance Level	Bucket size after removing bytes (bytes)
Initial	2000	N/A	N/A	N/A
1	2000	2000	Conformant	948
2	948	1460	Conformant	408
3	408	920	Non-Conformant	920
4	920	1432	Conformant	380
5	380	892	Non-Conformant	892
6	892	1404	Conformant	352
7	352	864	Non-Conformant	864
8	864	1376	Conformant	324
9	324	836	Non-Conformant	836
10	836	1348	Conformant	296

8.6.6.4 Testing the srTCM Meter

A SLA was constructed for this application flow. The SLA traffic profile was set to a CIR of 0.5Mbps, a CBS of 2000 bytes and an EBS of 2000 bytes.

```
MeterSpec mSpec1;
mSpec1.meterID = "SRTCM";
mSpec1.cIR = 500000;
mSpec1.cBS = 2000;
mSpec1.eBS = 2000;
mSpec1.pIR = 0;
mSpec1.pBS = 0;

//SLA Setup

Ptr<DiffServSla> sla1Ptr = CreateObject<DiffServSla>(001,cSpec1,mSpec1);
```

Figure 72: SLA configuration for the srTCM test

The table below shows the calculated results that the simulation results were compared to. The results obtained from the logging module were exactly as calculated

Table 16: Expected results for the srTCM test

Packet no	CB Bucket (bytes)	EB Bucket (bytes)	CB Bucket after adding bytes	EB Bucket after adding bytes	Conformance Level	CB Bucket after removing bytes (bytes)	EB Bucket after removing bytes (bytes)
Initial	2000	2000	N/A	N/A	N/A	N/A	N/A
1	2000	2000	2000	2000	Green	948	2000
2	948	2000	1460	2000	Green	408	2000
3	408	2000	920	2000	Yellow	920	948
4	920	948	1432	948	Green	380	948
5	380	948	892	948	Red	892	948
6	892	948	1404	948	Green	352	948
7	352	948	864	948	Red	864	948
8	864	948	1376	948	Green	324	948
9	324	948	836	948	Red	836	948
10	836	948	1348	948	Green	296	948

8.6.6.5 Testing the trTCM Meter

A SLA was constructed for this application flow. The SLA traffic profile was set to a CIR of 0.5Mbps, a PIR of 1Mbps, a CBS of 2000 bytes and a PBS of 2000 bytes.

```
MeterSpec mSpec1;
mSpec1.meterID = "TRTCM";
mSpec1.cIR = 500000;
mSpec1.cBS = 2000;
mSpec1.eBS = 0;
mSpec1.pIR = 1000000;
mSpec1.pBS = 2000;

//SLA Setup

Ptr<DiffServSla> sla1Ptr = CreateObject<DiffServSla>(001,cSpec1,mSpec1);
```

Figure 73: SLA configuration for the trTCM test

The table below shows the calculated results that the simulation results were compared to. The results obtained from the logging module were exactly as calculated

Table 17: Expected results for the trTCM test

Packet no	CB Bucket size (bytes)	PB Bucket Size (bytes)	Bytes to add to PB Bucket	CB Bucket size after adding bytes	Peak Bucket size after adding bytes	Conformance Level	CB Bucket size after removing bytes (bytes)	PB Bucket size after removing bytes (bytes)
Initial	2000	2000	N/A	N/A	N/A	N/A	N/A	N/A
1	2000	2000	N/A	2000	2000	Green	948	948
2	948	948	1024	1460	1972	Green	408	920
3	408	920	1024	920	1944	Yellow	920	892
4	920	892	1024	1432	1916	Green	380	864
5	380	864	1024	892	1888	Yellow	892	836
6	892	836	1024	1404	1860	Green	352	808
7	352	808	1024	864	1832	Yellow	864	780
8	864	780	1024	1376	1804	Green	324	752
9	324	752	1024	836	1776	Yellow	836	724
10	836	724	1024	1348	1748	Green	296	696

8.6.7 Testing the Enforcer (Marker and Dropper)

This test was carried out using the same application configuration used for testing the meters. A ConformanceSpec was configured for this application flow and is shown in Figure 74 below. This test was composed of three sub-tests. Each sub tests used the output of one the three meter tests previously performed.

The conformant-green packets were configured to be marked with the AF11 code point, non conformant-yellow packets were to be marked with the AF12 code point and non conformant-red packets are to be dropped.

```
//Conformance and Meter Specs

ConformanceSpec cSpec1;
cSpec1.initialCodePoint = AF11;
cSpec1.nonConformantActionI = AF12;
cSpec1.nonConformantActionII = drop;
```

Figure 74: SLA configuration for Enforcer testing

The expected results are shown in the table below. Each meter result instigated a particular action as specified in the SLA. On the receiving node a packet trace file was created and viewed on Wire-Shark. The results were as expected

Table 18: Enforcer test - expected results

Meter Type	No. of AF11 Packets	No. of AF12 Packets	No. of dropped Packets
Token Bucket	6	4	0
srTCM	6	1	3
trTCM	6	4	0

.

8.6.8 Testing the Weighted Round Robin and Priority Queue Schedulers

Four on/off applications were created on host Node 0 and were configured to transmit to Node 2. The application specification is given in the table below.

Table 19: Weighted Round Robin application configuration

On/off Application #	Source IP address	Destination IP address	Source port	Destination port
1	10.0.1.1	10.0.2.2	N/A	5111
2	10.0.1.1	10.0.2.2	N/A	5222
3	10.0.1.1	10.0.2.2	N/A	5333
4	10.0.1.1	10.0.2.2	N/A	5444

The NetDevice of the Edge DiffServQueue installed on Node 1 was configured to transmit at 10Mbps. Each PHB queue was given a maximum size of 2 packets. This was done to ensure that the forwarding classes were given the specified amount of bandwidth by limiting the forwarding resources to bandwidth only.

All other NetDevices transmitted at 100Mbps and had a queue size of 100 packets. Therefore the only congestion point was at the Edge DiffServQueue on Node 1. WRED was disabled for this test. Metering was also disabled; packets receive the code point specified in the SLA. No dropping by the Edge DiffServQueue was performed; only marking to the specified forwarding class. The aim of this test was to create bandwidth constraints and measure packet loss. For this test, the number of lost packets for a flow was proportional to the bandwidth it received.

8.6.8.1 Testing Weighted Round Robin Scheduler

For this test each class was allocated a portion of the link bandwidth according to its weight. Each application transmitted into a different forwarding class at a particular constant bit rate. Each application sent a maximum of 1000 packets. The number of packets dropped by the Edge DiffServQueue for each application was found using Pcap traces from the receiving node. Note that:

i. The number of packets dropped from each application is directly proportional to the bandwidth given to its forwarding class.(since the queue sizes have been reduced to 2 packets)

ii. When the applications transmit at different CBRs, the application with the highest rate will finish transmitting first (since only 100 packets are sent from each application); leaving extra bandwidth for the other application flows. Similarly for the application with the second highest CBR and so on.

Two sub-tests were performed: Constant and Varying Weights.

8.6.8.1.1 Constant Weights

For this test the WRR weights were held constant while the application CBRs were varied for several values. It was expected that by awarding equal weights to each queue, it would distribute the available bandwidth equally among the queues. By varying the application CBRs, this functionality could be tested. That is, the amount of loss an application experiences would be proportional to the amount by which its CBR exceeded its allocated bandwidth. The simulation and expected results are shown

below. The results were as expected. Note that the CBR rates would be slightly higher due to the protocol overhead and therefore the minimum packets expected would be higher than the actual number of received packets.

Table 20: WRR Test – constant weights– simulation and expected results

Application#	Constant Bit Rate (Mbps)	Forwarding class	Queue Weight	Bandwidth that should be given (Mbps)	Packets received(from 1000 packets sent)	Minimum Packets Expected
Test 1						
1	2.5	AF1	1	2.5	973	1000
2	2.5	AF2	1	2.5	973	1000
3	2.5	AF3	1	2.5	973	1000
4	2.5	AF4	1	2.5	972	1000
Test 2						
1	20	AF1	1	2.5	124	125
2	20	AF2	1	2.5	124	125
3	20	AF3	1	2.5	124	125
4	2.5	AF4	1	2.5	997	1000
Test 3						
1	8	AF1	1	2.5	306	312.5
2	6	AF2	1	2.5	454	416.6
3	4	AF3	1	2.5	787	626
4	2.5	AF4	1	2.5	993	1000
Test 4						
1	10	AF1	1	2.5	245	250
2	10	AF2	1	2.5	245	250
3	10	AF3	1	2.5	245	250
4	10	AF4	1	2.5	244	250

8.6.8.1.2 Varying Weights

For this test the WRR weights were varied while the application CBRs were held constant for several values. It was expected that by awarding unequal weights to each queue, it would distribute the available bandwidth in a manner that complies with the awarded weights. By varying the WRR weights, this functionality could be tested. Therefore the amount of loss an application experiences would be proportional to the

amount by which its CBR exceeds its allocated bandwidth. The simulation and expected results are shown below. The results were as expected.

Table 21: WRR Test – varying weights – simulation and expected results

Application#	Constant Bit Rate (Mbps)	Forwarding class	Queue Weight	Bandwidth that should be given (Mbps)	Packets received(from 1000 packets sent)	Minimum Packets Expected
Test 1						
1	10	AF1	4	4	391	400
2	10	AF2	3	3	293	300
3	10	AF3	2	2	196	200
4	10	AF4	1	1	99	100
Test 2						
1	10	AF1	1	1	100	100
2	10	AF2	2	2	196	200
3	10	AF3	3	3	293	300
4	10	AF4	4	4	390	400
Test 3						
1	10	AF1	4	5.714	558	571.4
2	10	AF2	1	1.428	141	142.8
3	10	AF3	1	1.428	140	142.8
4	10	AF4	1	1.428	140	142.8

8.6.8.2 Testing the Priority Queue Scheduler

For this test application used the EF class instead of the AF1 class since the EF forwarding class uses the priority queue scheduler. Each application transmitted into a different forwarding class at a particular constant bit rate. Each application sent a maximum of 1000 packets. The number of packets dropped by the Edge DiffServQueue for each application was found using Pcap traces from the receiving node. It was expected that the priority queue will preempt other traffic from accessing the link bandwidth and give the full use of the link to its packets (i.e. application 1). Therefore application 1 would have full use of the link bandwidth and preempt the other

applications, which used the other forwarding classes, from using the link. The expected and simulation results are shown in the table below. The results were as expected.

Table 22: Priority queuing test – simulation and expected results

Application#	Constant Bit Rate (Mbps)	Forwarding class	Queue Weight	Bandwidth that should be given (Mbps)	Packets received(from 1000 packets sent)	Minimum Packets Expected
1	10	EF	n/a	High priority	973	1000
2	10	AF1	1	Low priority	2	2
3	10	AF2	1	Low priority	2	2
4	10	AF3	1	Low priority	2	2

The two packets that are received for the applications using the AF forwarding classes are due to the queue size of 2 packets.

8.6.9 Testing the Weighted Random Early Detection (WRED) AQM mechanism

An on/off application was created on node 0 and was configured to transmit to node 2. The application was configured as shown in the table below. Note that the application was always on (off time = 0). The application was configured to only send 100 packets.

Table 23: WRED test - application configuration

Node	On/off Application #	Transport Protocol	Source IP address	Destination IP address	Source port	Destination port	Application CBR Mbps	Packet size (bytes)
0	1	UDP	10.0.1.1	10.0.2.2	N/A	5111	50Mbps	1024

An SLA was created for the application flow and was configured to be marked with the AF11 code point. Metering was not performed; all packets were marked with the AF11 code point. The application only sent 100 packets at the CBR specified. The NetDevice data rate on Node 1 was set to 0.1Mbps to create congestion in the network. All queue sizes were set at 100 packets. With this configuration, the packets that the application sends would be forced to be queued. This test consisted of two parts:

1. Test and verify the average queue size calculation
2. Test and verify the drop probability calculation that occurs for the average queue size values obtained

8.6.9.1 Testing the average queue size calculation

For each packet to be enqueued, measurements were taken using the logging module, such as the new average and current queue size at each packet enqueue. Using the list of current queue size values, the corresponding average queue size values were 'hand' calculated. The average queue size for each packet enqueue was compared to calculated values. This was done for multiple exponential constants. Note that packet dropping by WRED was disabled so that the average queue size kept growing and was not affected by packet drops. The resulting packet drop probabilities for each average queue size value were compared with calculated values.

The diagram below shows the queue size data obtained from the simulation and those that were calculated. Notice that the current queue size grows linearly from zero. Using the current queue size data, the average queue sizes were calculated and

compared to the values the simulator calculated. This was done for exponential constants 5, 6, and 7. The results were identical to calculated values as shown in the diagram below.

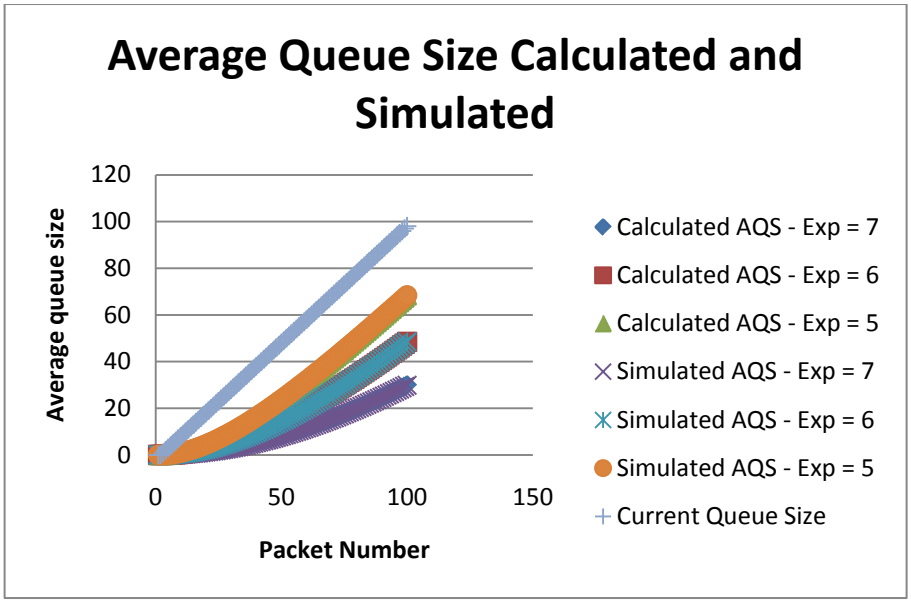

Figure 75: WRED Test 1 – testing the average queue size calculation

8.6.9.2 Testing the drop probability calculation

For this test the AF11 WRED profile was configured with a MinTH, MaxTH and Max Probability of 10, 20 and 100 respectively. The application and SLA configuration from the previous test was used. The calculated average queue sizes values for Exp = 7 were used together with the above WRED profile to calculate the corresponding drop probabilities. The drop probabilities were also obtained from the simulation using the logging module. Note that these were **drop probabilities** and not a dropping action. Dropping was disabled to allow the average queue size to grow. The result is shown in the diagram below.

Notice how the graph matches with the specified profile in the table above. That is, the drop probability remains at zero before an average queue size of 10 (min threshold) and remains at a 100 after the average queue size has exceeded 20 (the max threshold). The calculated probabilities are also shown on the graph. The calculated probabilities also match with the simulated probabilities.

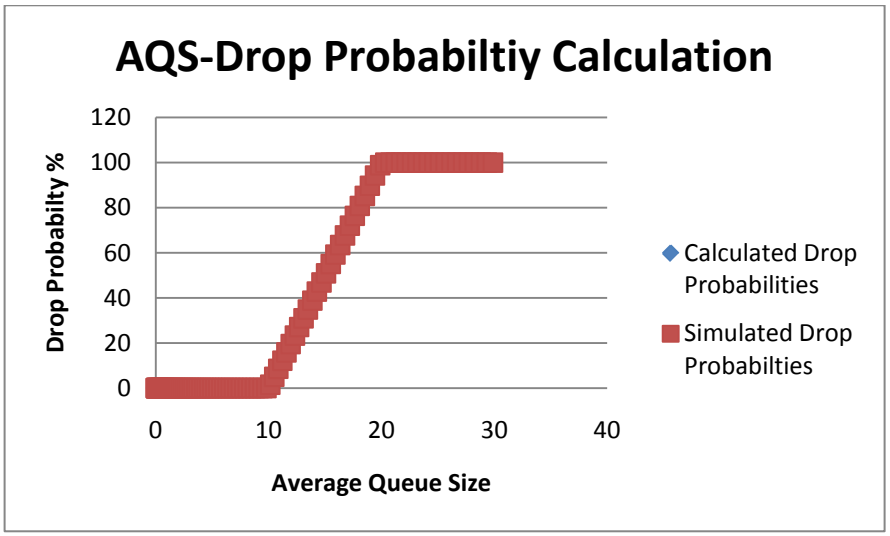

Figure 76: WRED test 2 – Testing the drop probability calculation

8.6.10 Testing the Stat Collector

While the StatCollector is not part of the DiffServ architecture, it was still tested since it provides the simulation output for the evaluation. Testing the StatCollector consisted of the following two tests:

1. Testing the ability to dynamically classify flows in real time
2. Testing the Tx and Rx times and the resulting delay and delay variation and metric computations
3. Testing the packet loss metric

8.6.10.1 Test 1 –Flow classification

The application configuration used for the testing of the MF classifier is used. It is expected that the StatCollector will classify each new flow and assign it a flow identifier. The figure below shows the text file created from the simulation. The results are as expected.

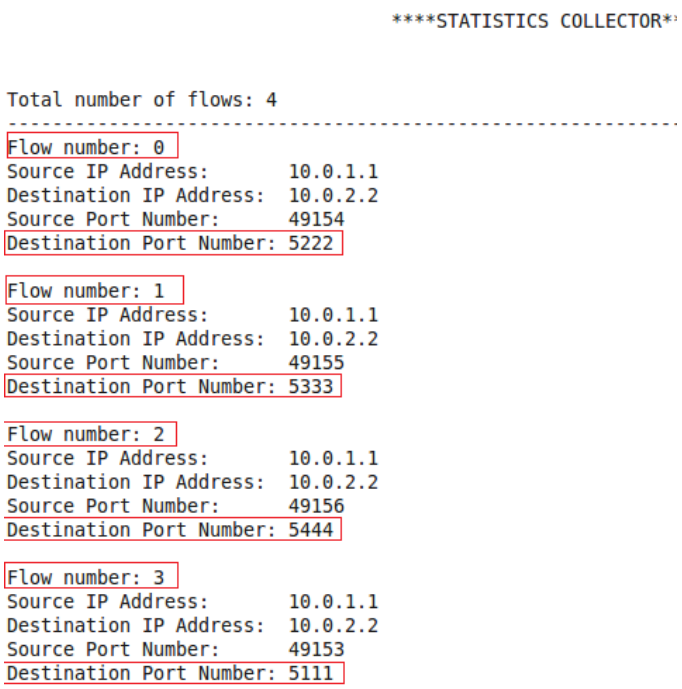

Figure 77: StatCollector test 1 - classification

8.6.10.2 Test 2 – Delay and Delay Variation metrics

The application configuration used for testing the meters was used; the application transmission times should follow an inter-arrival time of 0.008192 seconds. The application only sends 10 packets. The packet transmission times can be calculated since the inter arrival times are known. The packet reception times can be calculated using the configured propagation and resultant serialization delays and be compared

to the times recorded by the StatCollector. From these values the end to end delay and delay variation can be calculated.

The test network has 2 packet serialization points; node 0 and node 1 before packets arrive at the destination node2. The NetDevices at nodes 0 and 1 are configured to 100 and 10 Mbps respectively. There are two links; between nodes 0 and 1 and between nodes 1 and 2. The n0-n1 and n1-n2 link are configured with a propagation delay of 2 and 0ms. Note that this test does not involve queuing delays since no congestion is created. If congestion was created, packets would be forced onto a queue and queuing delays would result. However these queuing delays involve complex calculations and hence were omitted from this test. The end to end delay is calculated as follows:

Packet size = 1024 bytes + UDP (8 bytes) + IP (20 bytes) + PPP (2 bytes) = 1054 bytes

Serialization delay at Node 0 = (1054*8)/100Mbps =0.0000844 seconds

Serialization delay at Node 1 = (1054*8)/10Mbps =0.000844 seconds

Total end to end delay for a packet = 0.0000844 + 0.000844 + 0.002 + 0 = 0.002928 seconds

The expected reception times can be calculated by adding the calculated end to end delay to the known transmission times. The delay variation is expected to be zero.

Table 24: StatCollector test 2 – expected results

Tx Time	Rx Time	Delay	Delay Variation
0.008192	0.01112	0.002928	Undefined
0.016384	0.019312	0.002928	0
0.024576	0.027504	0.002928	0
0.032768	0.035696	0.002928	0
0.04096	0.043888	0.002928	0
0.049152	0.05208	0.002928	0
0.057344	0.060272	0.002928	0
0.065536	0.068464	0.002928	0
0.073728	0.076656	0.002928	0
0.08192	0.084848	0.002928	0

The figure below shows part of the output text file produced by the StatCollector for the simulation run. They are exactly as calculated

Tx Time	Rx Time	Delay	Delay Var
0.008192	0.0111195	0.00292752	999
0.016384	0.0193115	0.00292752	0
0.024576	0.0275035	0.00292752	0
0.032768	0.0356955	0.00292752	0
0.04096	0.0438875	0.00292752	0
0.049152	0.0520795	0.00292752	0
0.057344	0.0602715	0.00292752	0
0.065536	0.0684635	0.00292752	0
0.073728	0.0766555	0.00292752	0
0.08192	0.0848475	0.00292752	0

Figure 78: StatCollector test 2 –simulation results

8.6.10.3 Test 3 – Packet loss metric

The application configuration used in Test 2 is used with a few changes. The application now sends a maximum of 1000 packets at 20Mbps. Queue sizes are at 100 packets. The objective of this test is to cause congestion and hence packet drops. The packet drop information provided by the StatCollector can be confirmed with the Pcap file created for the receiving NetDevice on node 2. The figures below show the

simulation results from the StatCollector output and the Pcap trace file. Both the StatCollector and Pcap file display the same number of packet drops.

```
Total number of flows: 1
--------------------------------

Flow number: 0
Source IP Address:       10.0.1.1
Destination IP Address:  10.0.2.2
Source Port Number:      49153
Destination Port Number: 5111

Flow Summary:
Number of packets sent:     1000
Number of packets dropped:  414
Percentage of packets lost: 41.4%
```

Figure 79: StatCollector test3 – simulation results from StatCollector

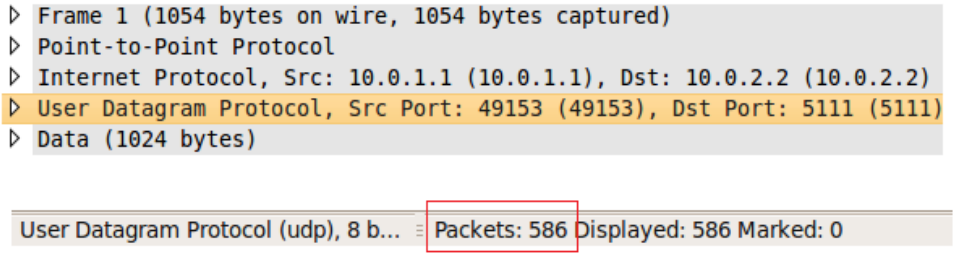

Figure 80: StatCollector test3 - simulation results from Pcap file on receiving node

8.6.11 Application Testing

In this section, the output of the applications modeled will be examined to verify their functionality. The applications examined are the VoIP and video streaming applications.

8.6.11.1 VoIP application

First the exponential random variables used by the VoIP application for it's on and off times will be used to generate a number of random numbers as shown in the table below. The application will then generate packets using the exponential variables. The output Pcap file created by the simulator was then viewed on Wire-Shark to verify the application operation. The Pcap file shows that the packets were transmitted during specified times.

Table 25: Consecutive On/Off values produced by the two random variables

Consecutive On and Off times
0.131748 (off)
0.0634106
0.324662
0.226033
0.456324
0.31282
0.626211
1.44938
0.647896
0.288759

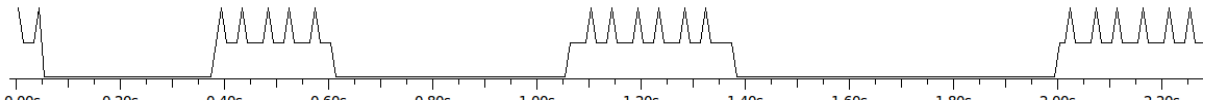

Figure 81: IO graph taken from wire shark showing on and off durations

The random variables were then tested to confirm that they produce the required probability distribution functions. This was performed for both the on and off time exponential variables. The exponential variables were first used to generate 1000 random numbers. From these values the discrete PDF could be created. The theoretical continuous PDFs were then super imposed onto the discrete distributions as shown below. The graphs show that the random variables do indeed produce, to a sufficient level, the required probability distribution functions.

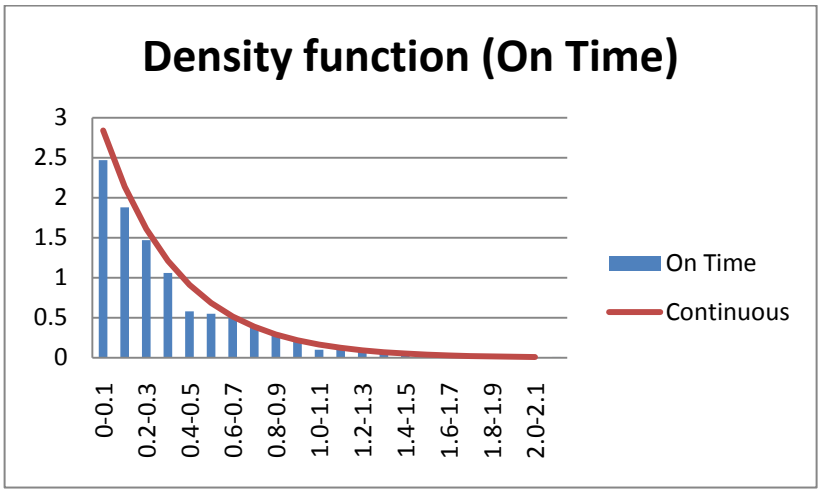

Figure 82: random variable (on time) PDF

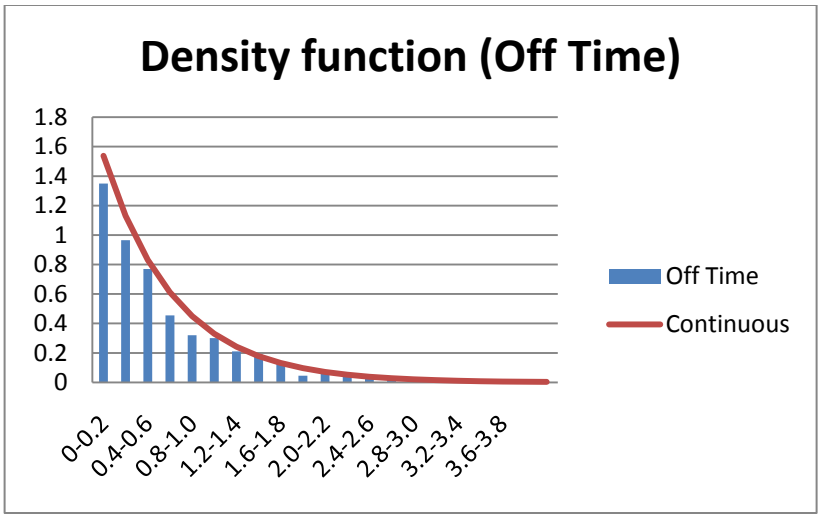

Figure 83: random variable (off time) PDF

8.6.11.2 Video Streaming application

For this test, the application will transmit a number of packets from a trace file. The output Pcap file created will then be viewed on Wire-Shark to verify the application operation. The table below shows the first five packets from the trace file used.

Table 26: Video streaming application test – first five packets of the trace file

Frame Number	Frame type	Time(ms)	Length
1	I	0	402
2	P	120	490
3	B	40	142
4	B	80	567
5	P	240	539

The diagram below displays the Pcap file viewed on Wire-Shark. Observe that each consecutive frame size matches those stated in the trace file. The transmission times also match the times specified in the trace file.

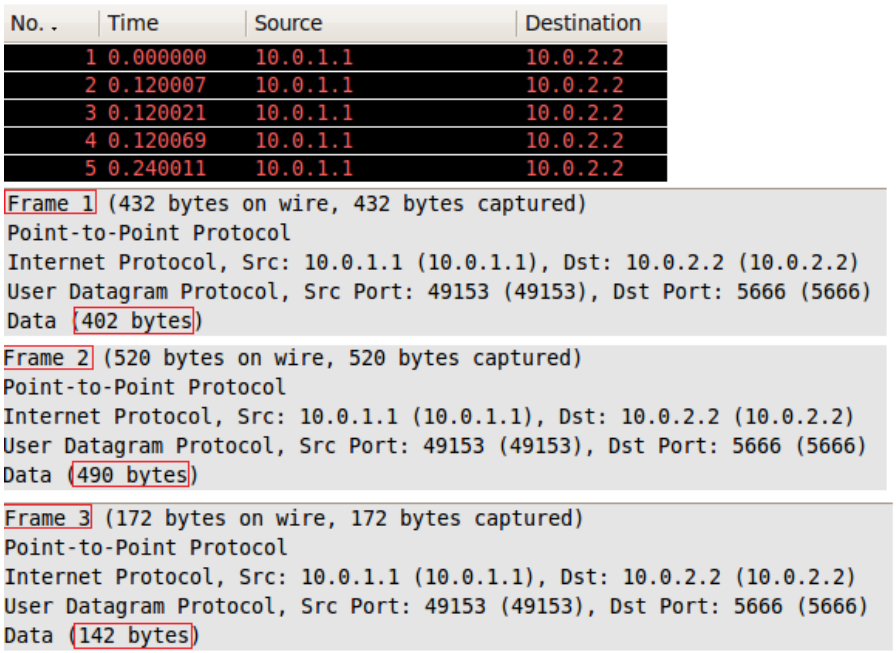

Figure 84: Video streaming application test – video streaming application output viewed on Wire-Shark

8.7 Appendix G - Experiment Results

8.7.1 Experiment 1

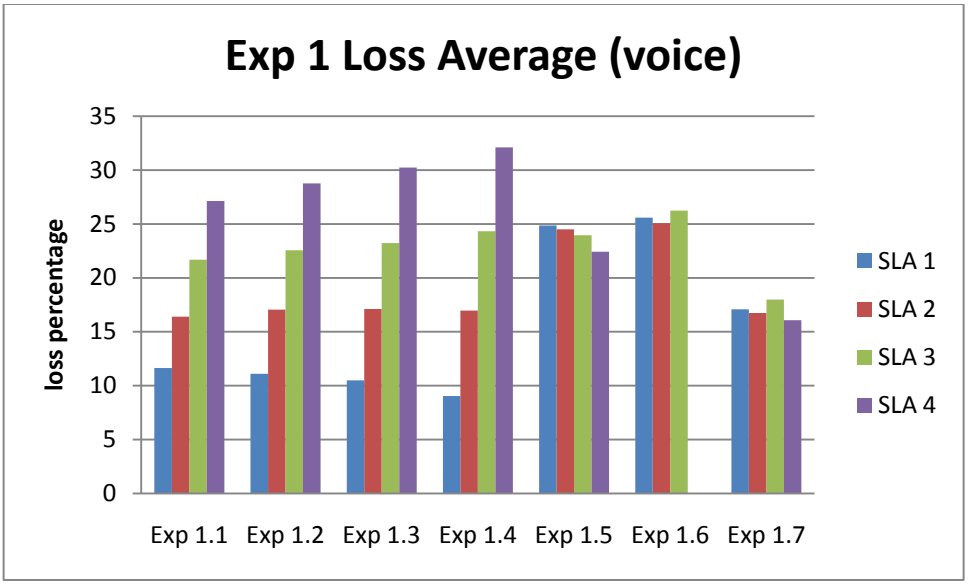

Figure 85: Experiment 1 - Packet loss among SLAs – voice

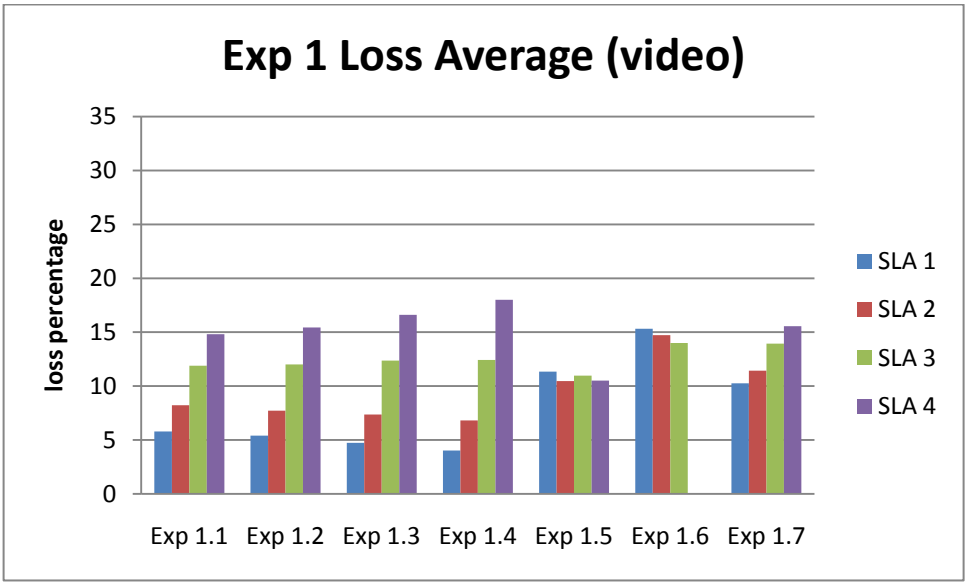

Figure 86: Experiment 1 - Packet loss among SLAs - video

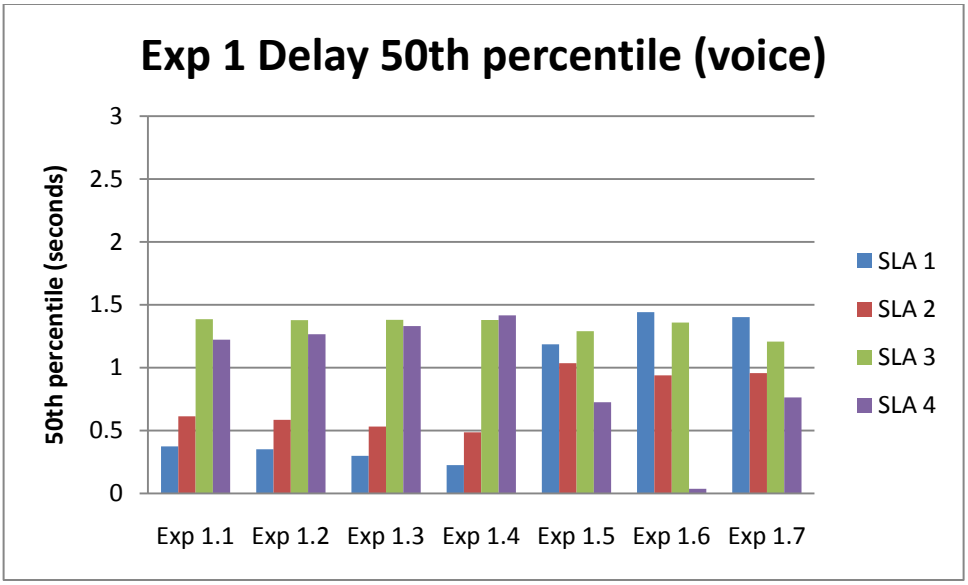

Figure 87: Experiment 1 - Delay among SLAs – voice

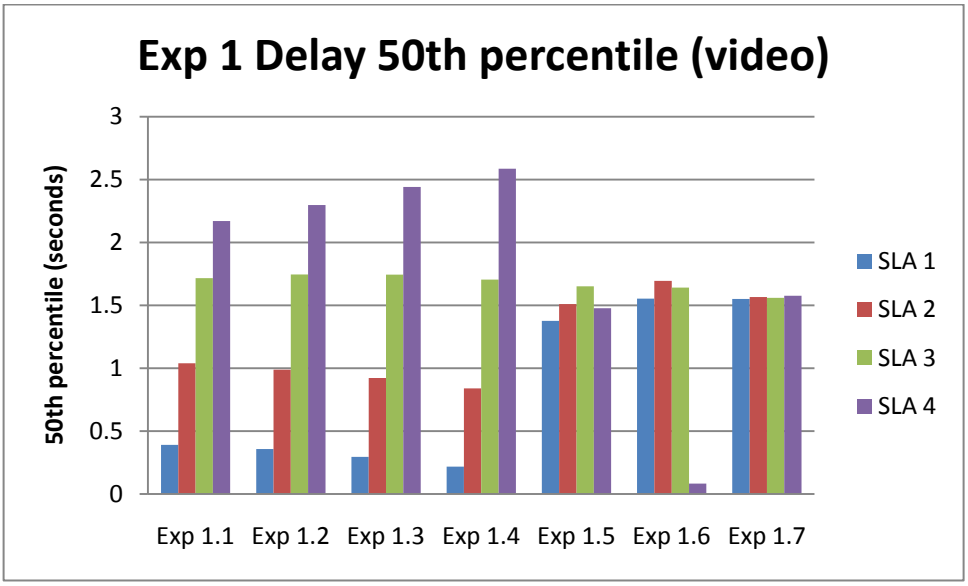

Figure 88: Experiment 1 - Delay among SLAs - video

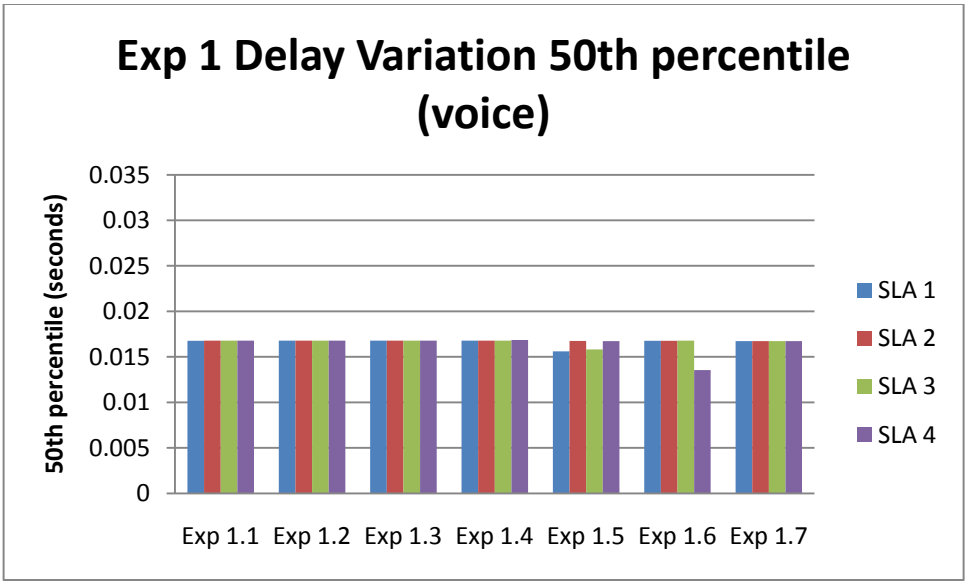

Figure 89: Experiment 1 - Delay variation among SLAs – voice

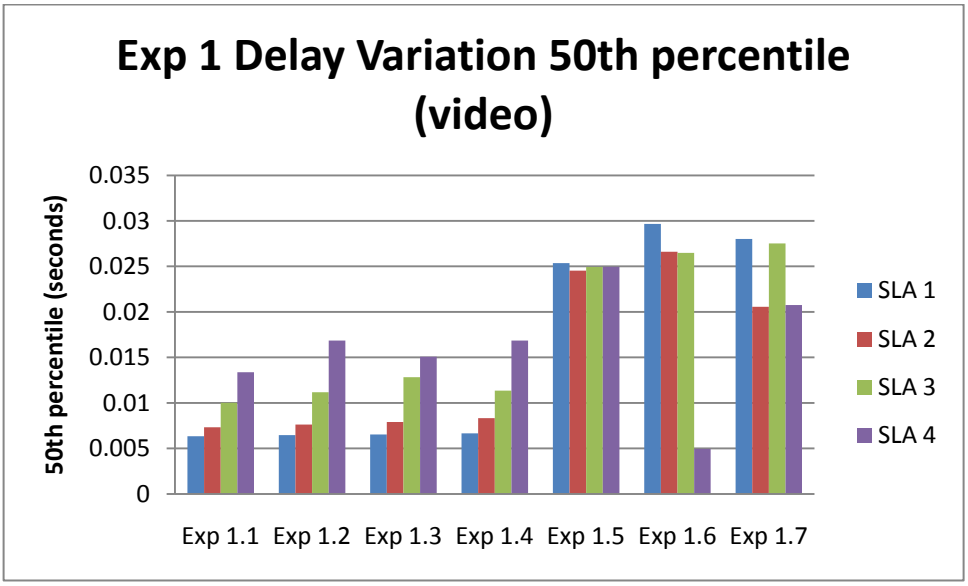

Figure 90: Experiment 1 - Delay variation among SLAs - video

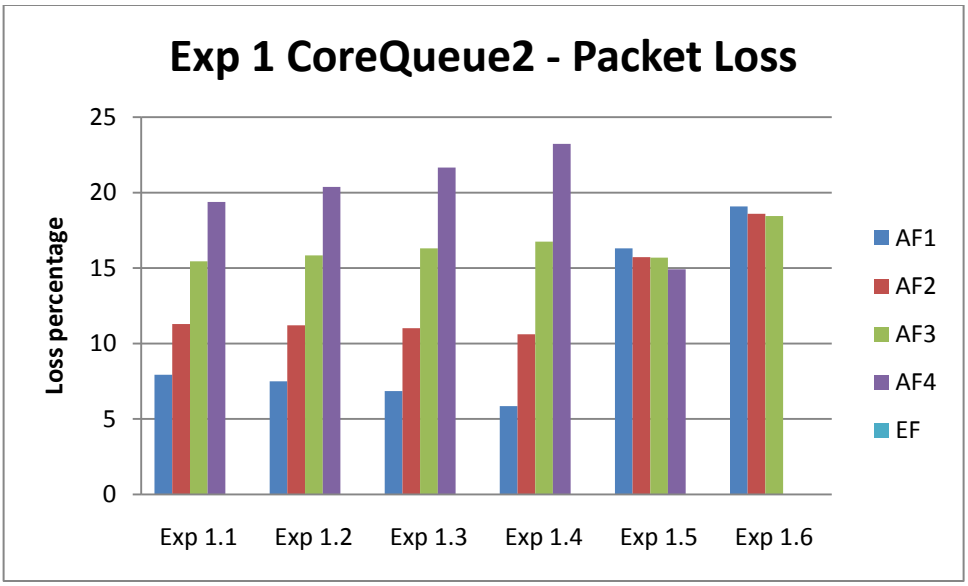

Figure 91: Experiment 1 - CoreQueue 2 - Packet loss

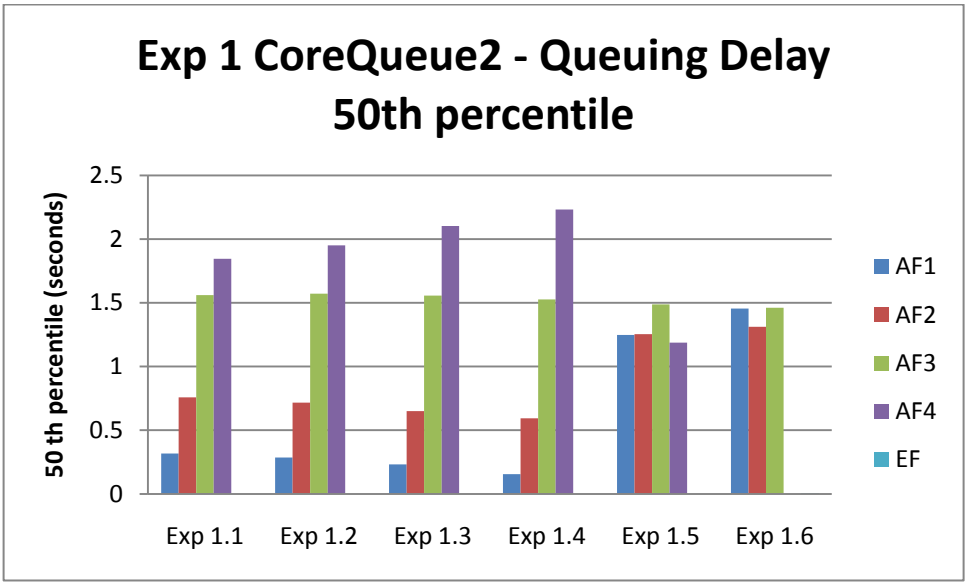

Figure 92: Experiment 1 - CoreQueue 2 - Queuing delay

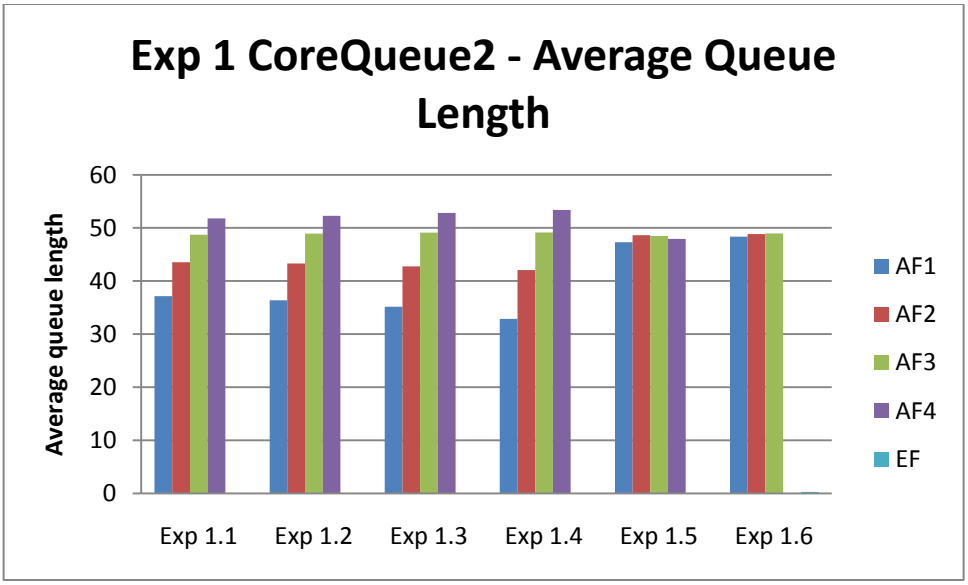

Figure 93: Experiment 1 - CoreQueue 2 - Average queue length

8.7.2 Experiment 2

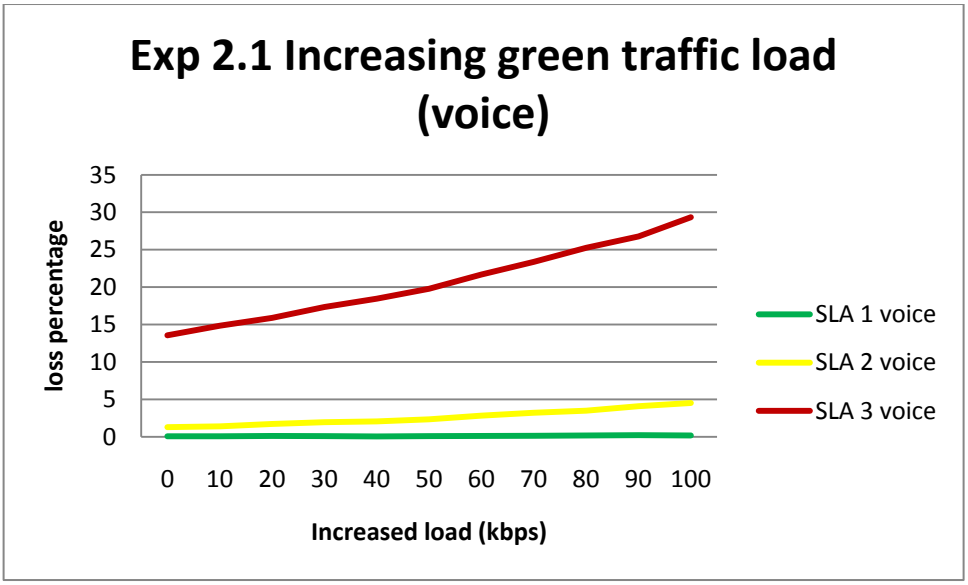

Figure 94: Experiment 2.1 - voice - Packet loss

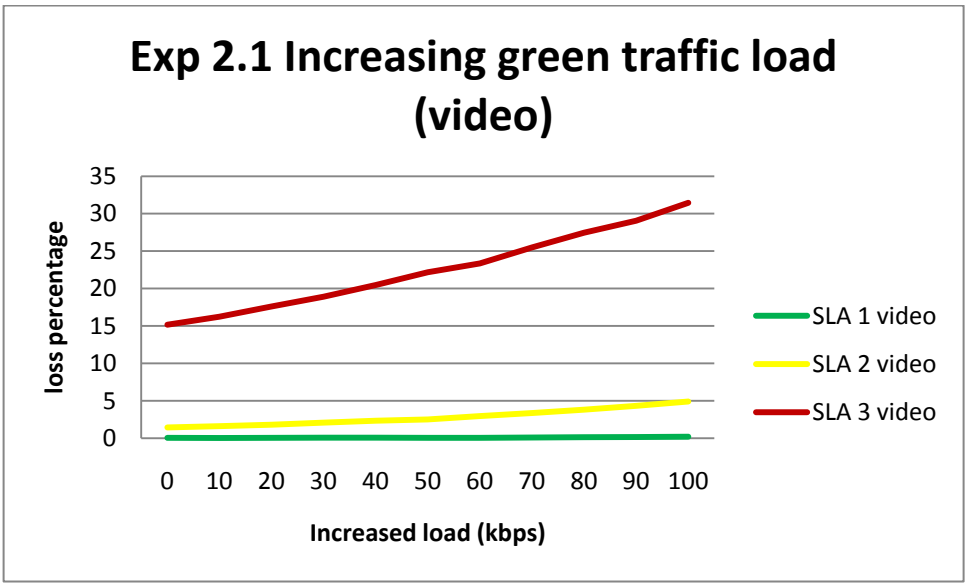

Figure 95: Experiment 2.1 - video - Packet loss

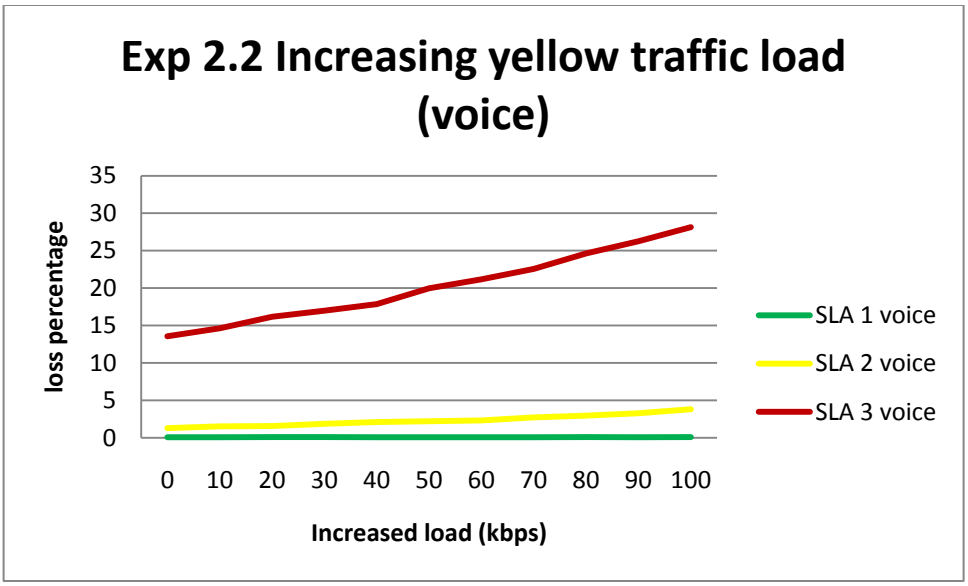

Figure 96: Experiment 2.2 - voice - Packet loss

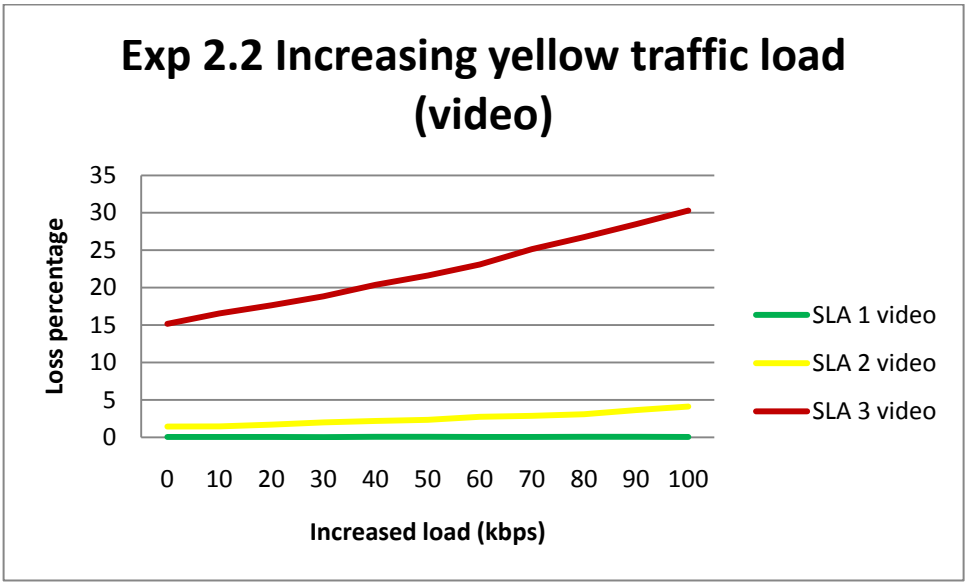

Figure 97: Experiment 2.2 - video - Packet loss

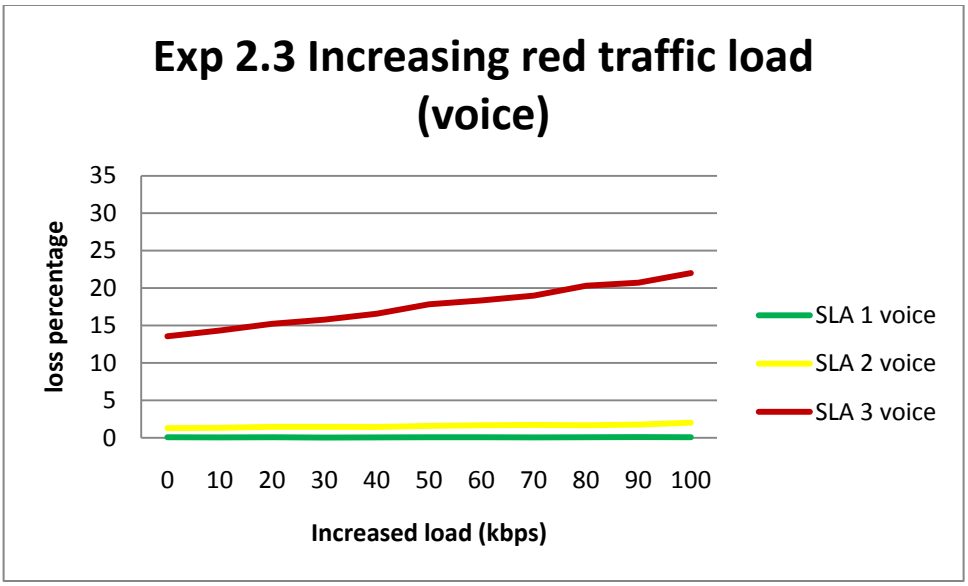

Figure 98: Experiment 2.3 - voice - Packet loss

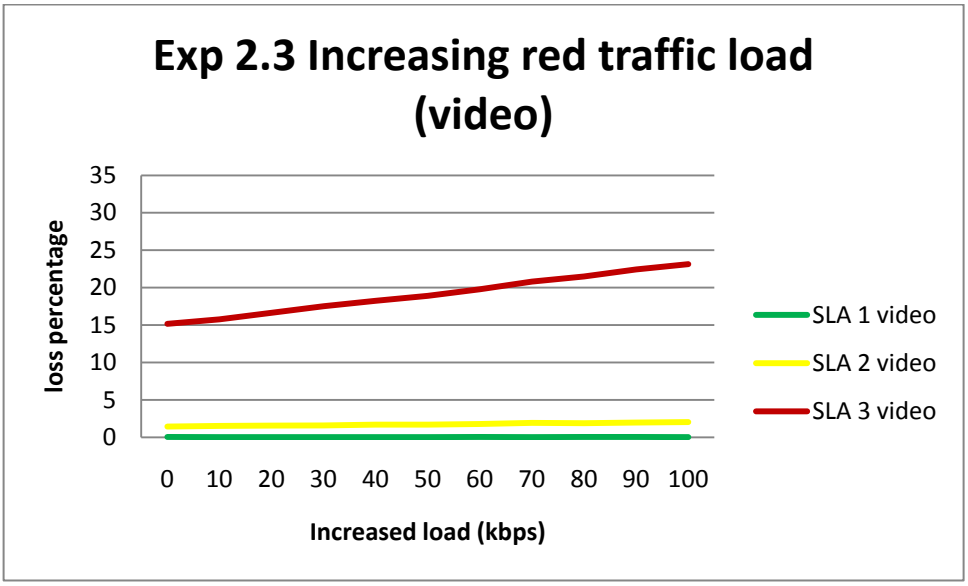

Figure 99: Experiment 2.3 - video - Packet loss

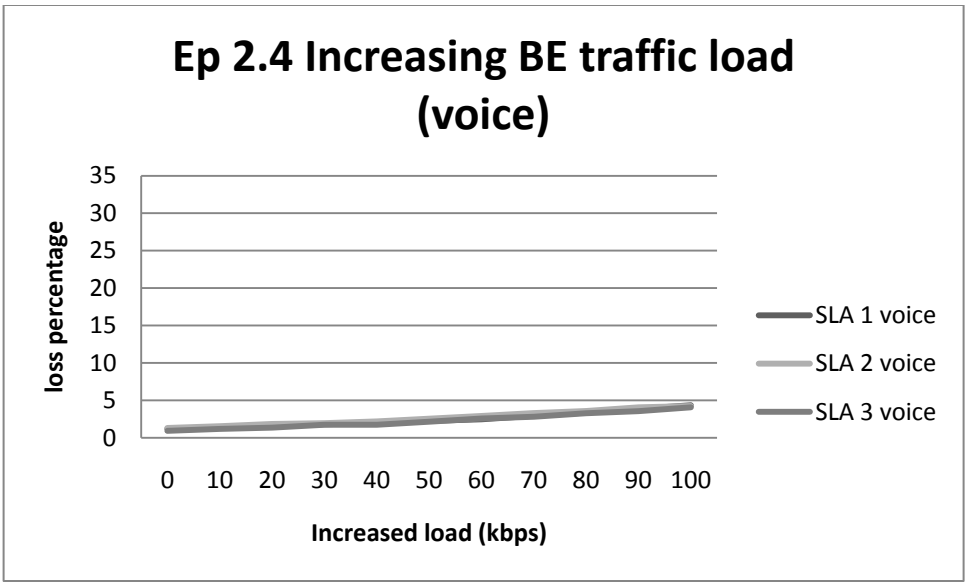

Figure 100: Experiment 2.4 - voice - Packet loss

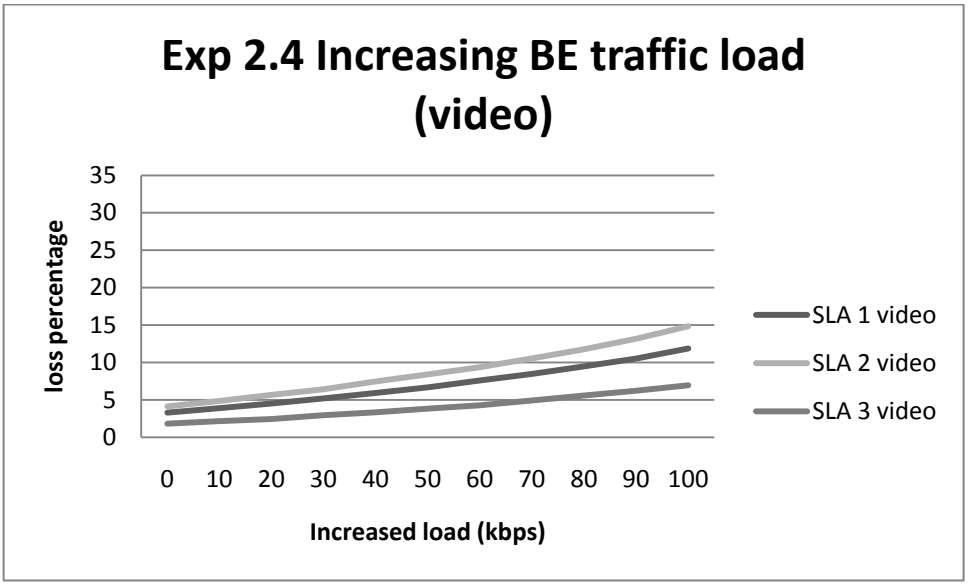

Figure 101: Experiment 2.4 - video - Packet loss

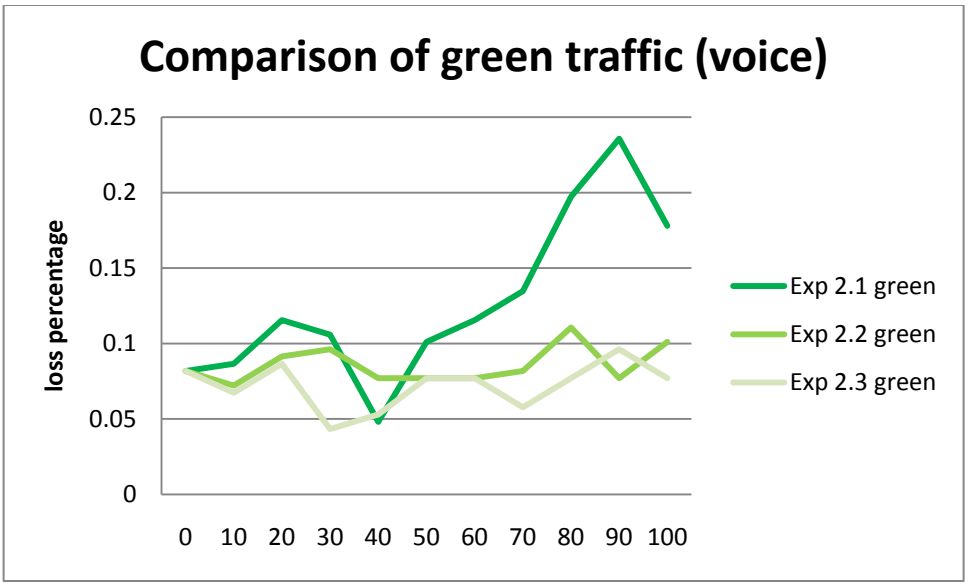

Figure 102: Experiment 2 - comparison of green traffic

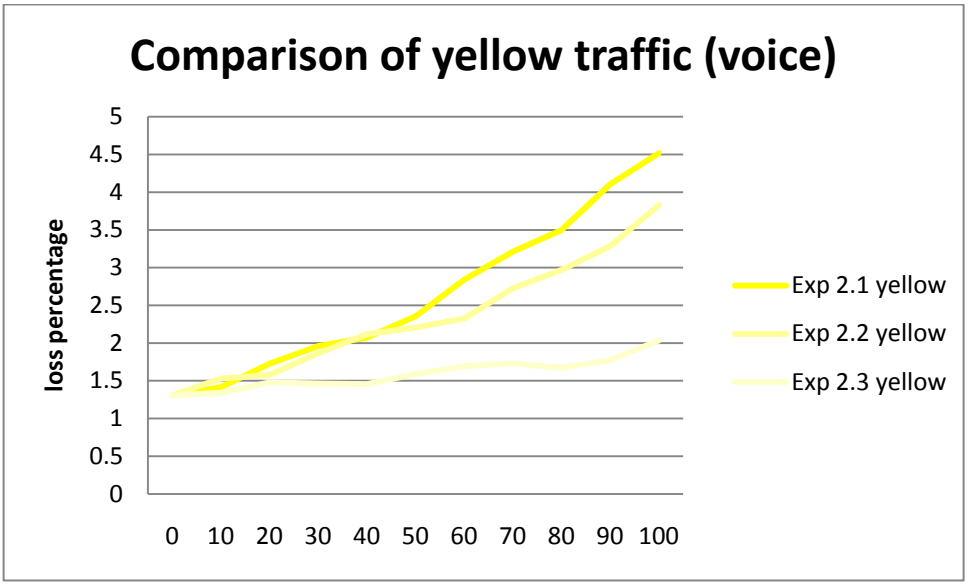

Figure 103: Experiment 2 - comparison of yellow traffic

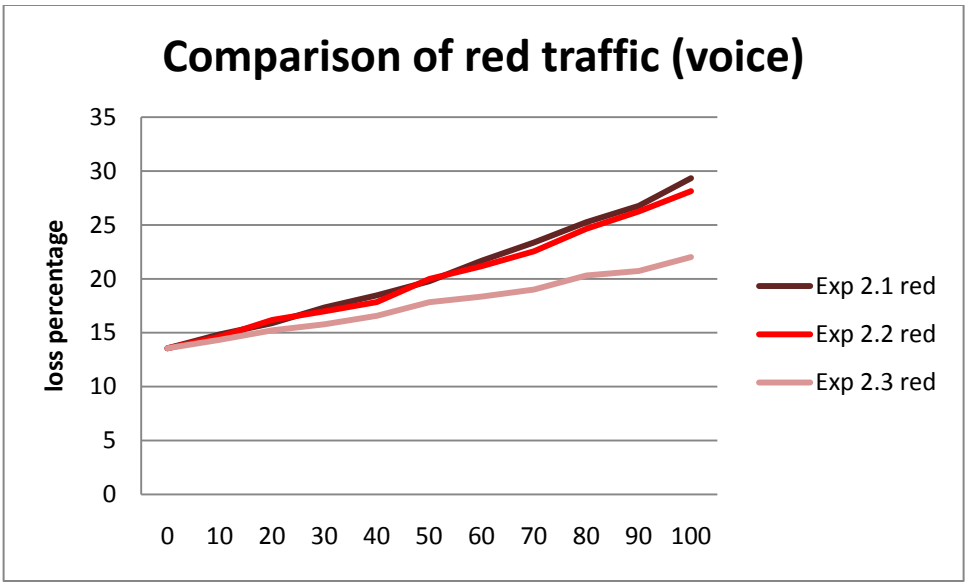

Figure 104: Experiment 2 - comparison of red traffic

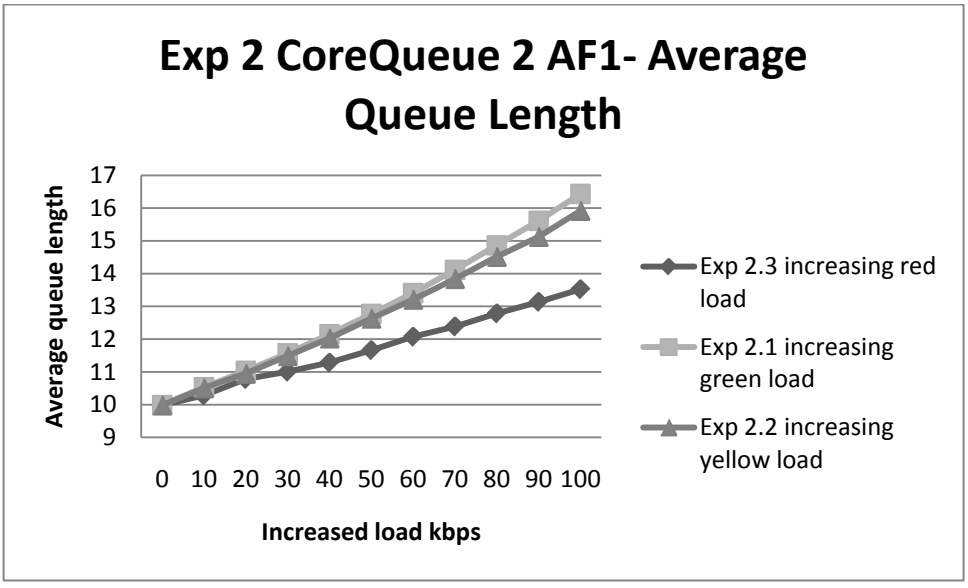

Figure 105: Experiment 2 - CoreQueue2 - AF1 Average queue length

8.7.3 Experiment 3

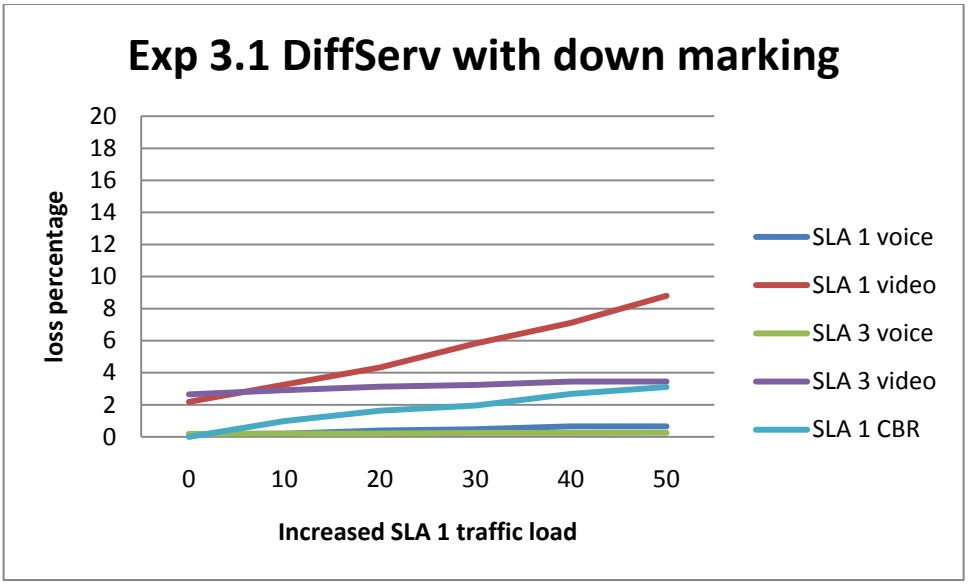

Figure 106: Experiment 3.1 - DiffServ network with down marking – Packet loss

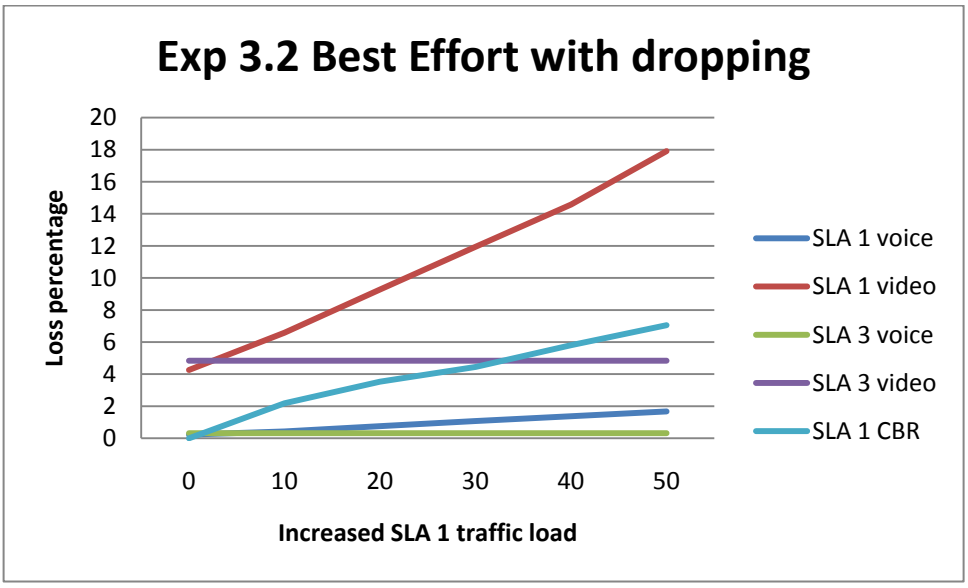

Figure 107: Experiment 3.2 - Best Effort network with dropping – Packet loss

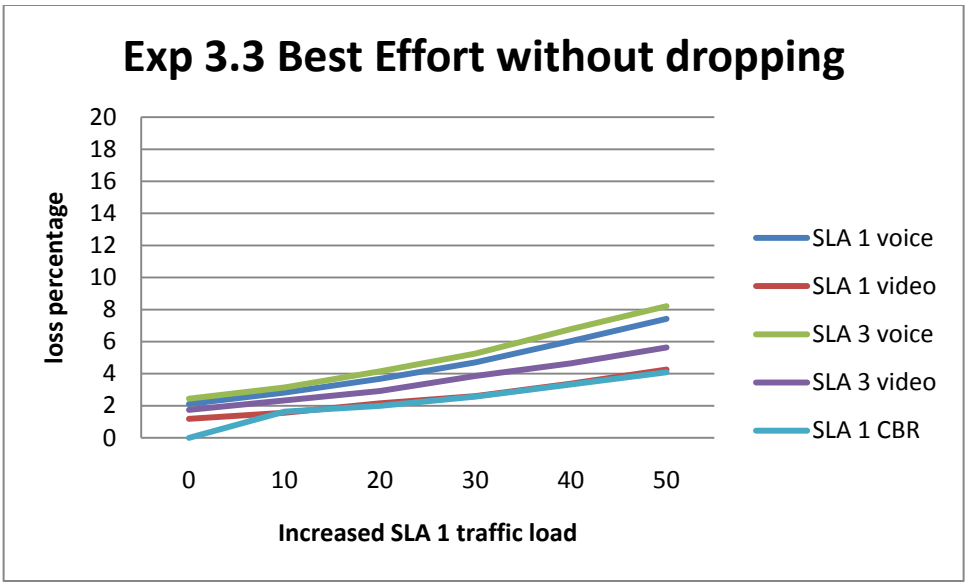

Figure 108: Experiment 3.3 - Best Effort network without dropping – Packet loss

8.7.4 Experiment 4

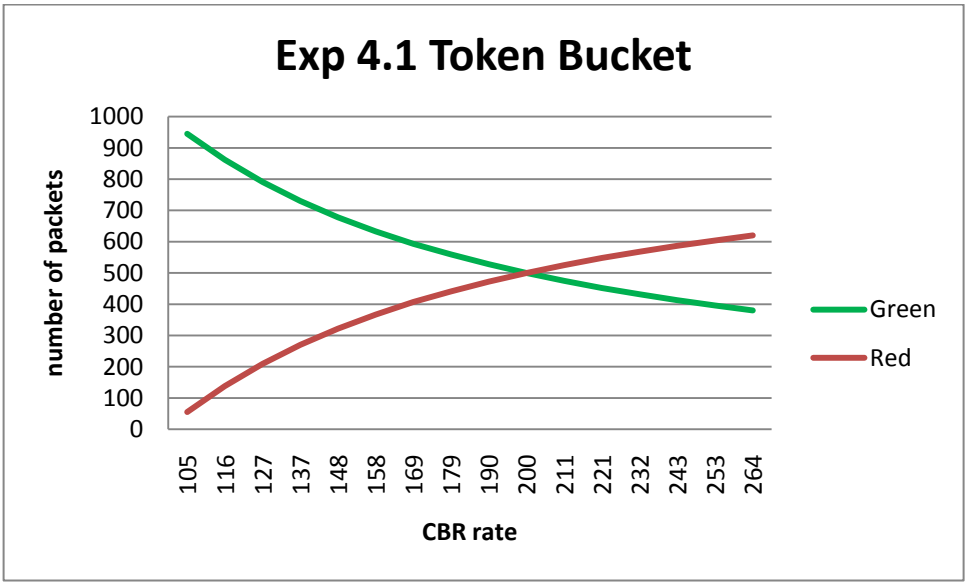

Figure 109: Experiment 4 -Token bucket

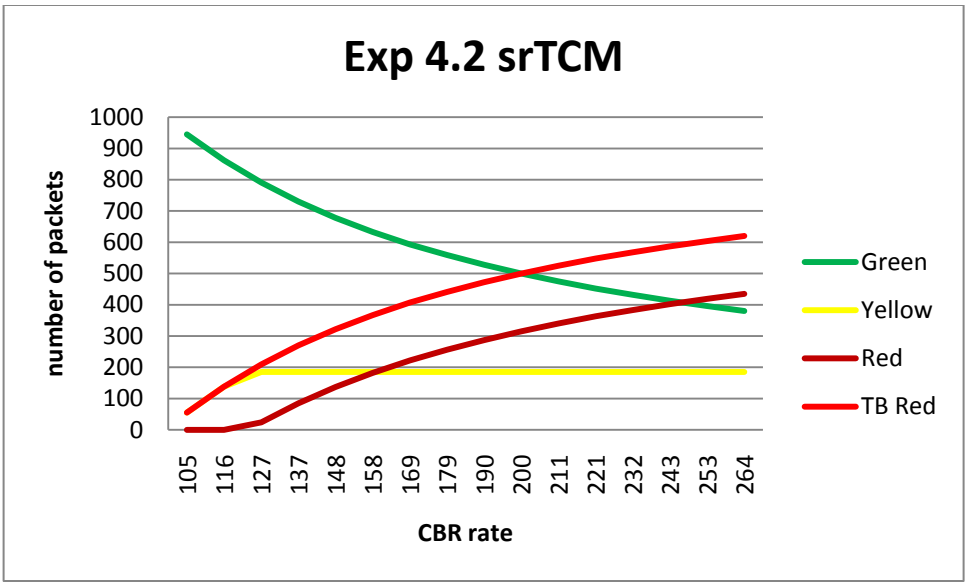

Figure 110: Experiment 4 –srTCM

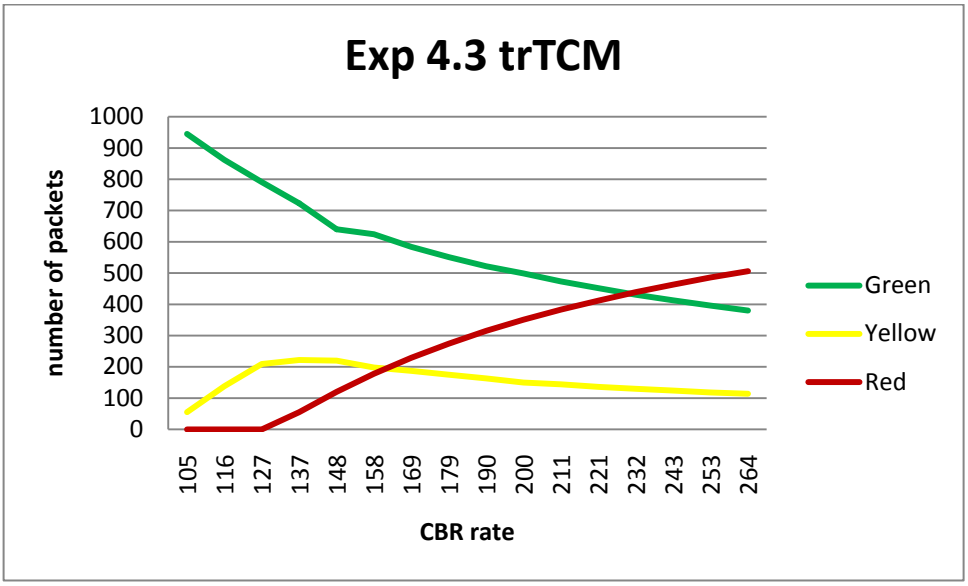

Figure 111: Experiment 4 -trTCM

8.7.5 Experiment 5

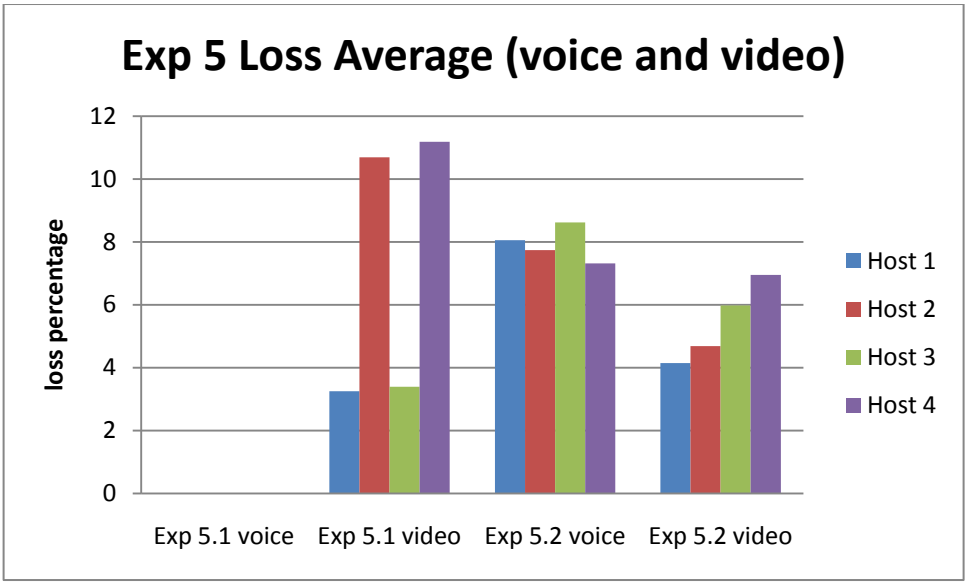

Figure 112: Experiment 5 - Packet loss among SLAs - voice and video

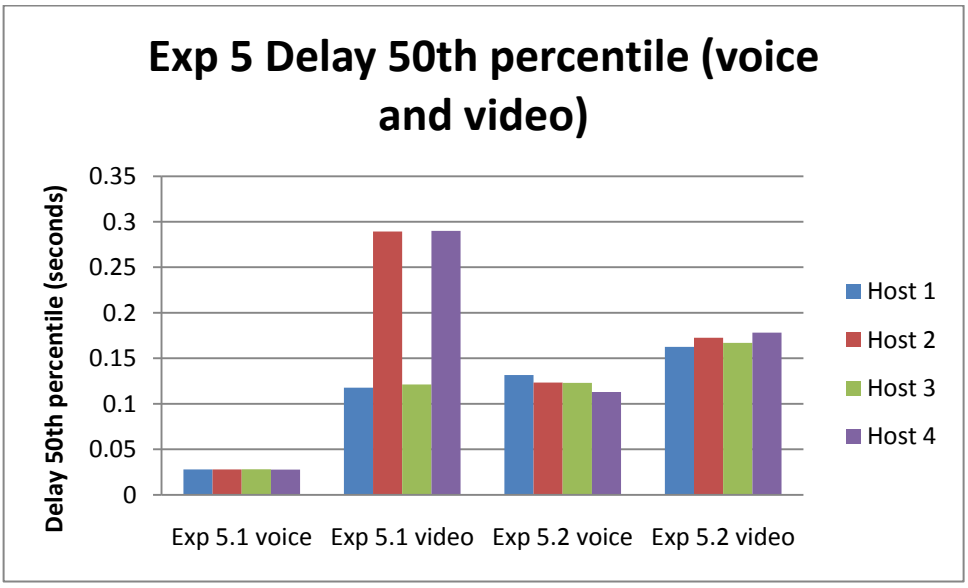

Figure 113: Experiment 5 - Delay among SLAs - voice and video

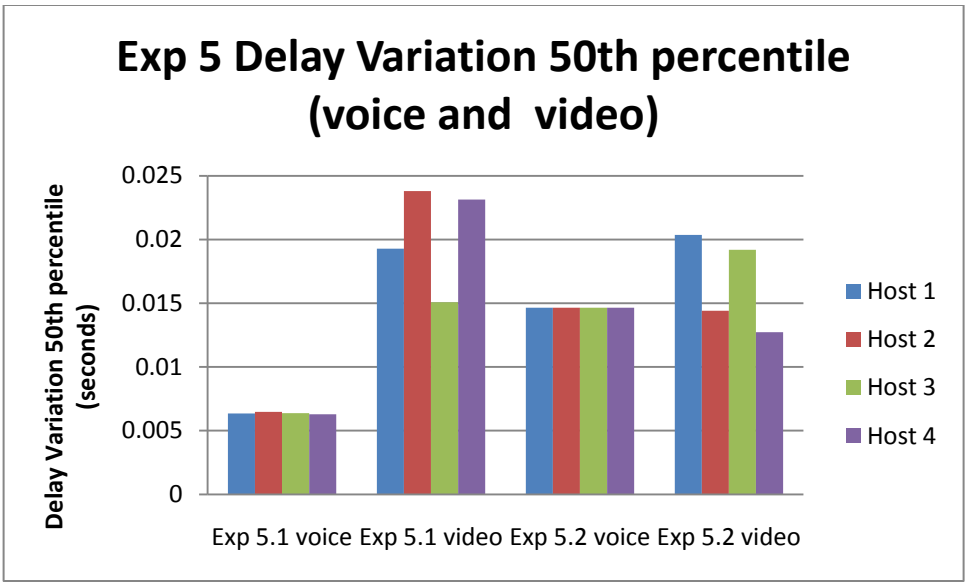

Figure 114: Experiment 5 - Delay variation among SLAs - voice and video

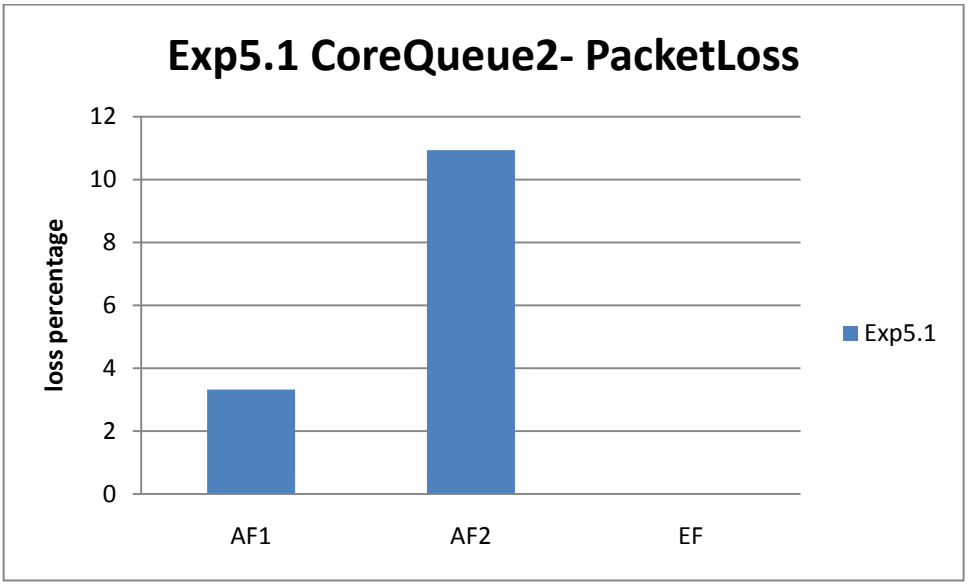

Figure 115: Experiment 5 - CoreQueue2 - Packet loss

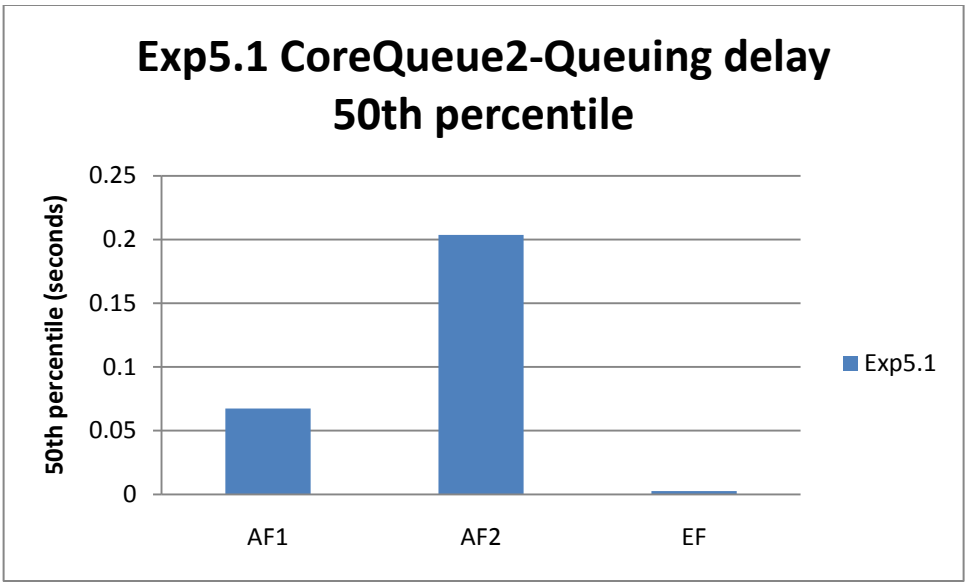

Figure 116: Experiment 5 - CoreQueue2 - Queuing delay

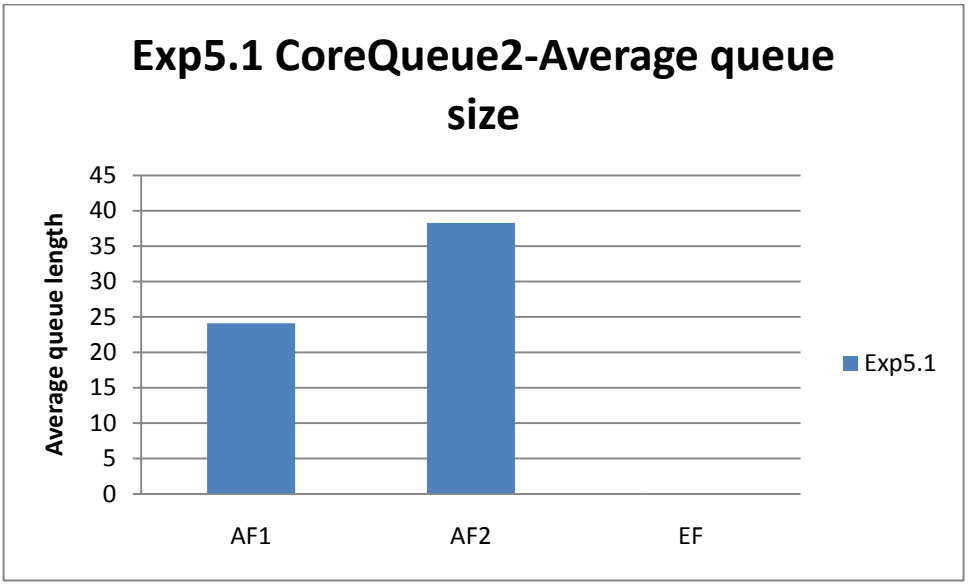

Figure 117: Experiment 5 - CoreQueue2 - Average queue length

8.7.6 Experiment 6

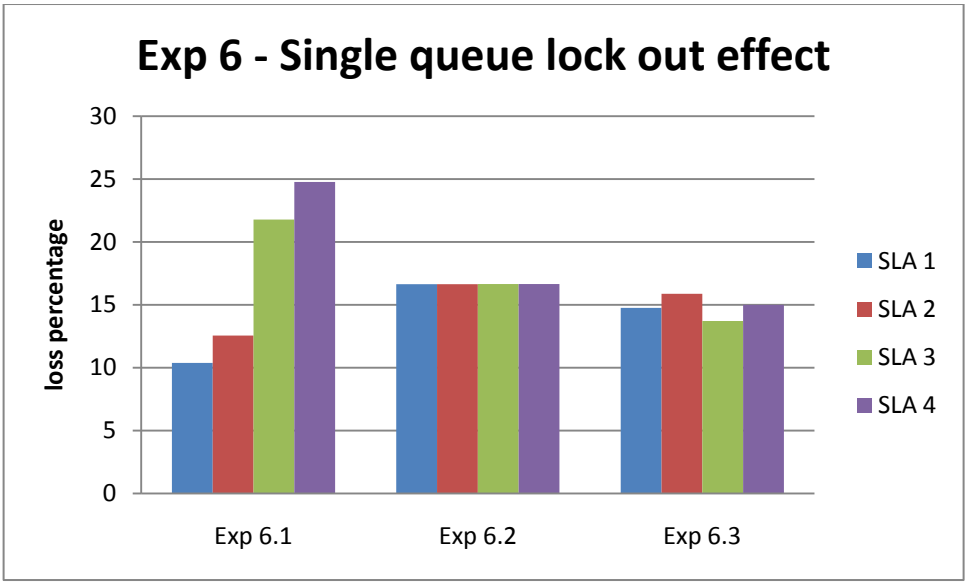

Figure 118: Experiment 6 - Single queue lock out effect