Traffic Management Algorithms in Differentiated Services Networks

Kenneth Andersson
and
Jimisola Laursen

Supervisors:
Ulf Ahlfors
SwitchCore, Lund, Sweden
and
Robert Pallbo
Dept. of Computer Science, Lund University, Sweden

September 5, 2000

Abstract

The Differentiated Services (DiffServ) Architecture, a Quality of Service (QoS) solution being worked on by an IETF work group, is aimed to solve the increasing problems with no service guarantees in the current Internet. New services such as video-on-demand and IP-telephony will be unusable without some sort of service guarantees on which to build applications on. A replacement architecture for the Integrated Services (IntServ) Architecture is needed because of its problems with overhead and scalability.
This master thesis studies and evaluates traffic algorithms, specifically scheduling and active queue management algorithms, within the Differentiated Services area using the Network Simulator. The studies investigate Differentiated Services network stability and performance through noise influenced simulations. Results show that against unresponsive users network stability and performance mainly depends on the used scheduling algorithm.

Sammanfattning

Differentiated Services (DiffServ) arkitekturen, en tjänstekvalitetslösning (eng Quality of Service) som utarbetats av en IETF arbetsgrupp, är tänkt att lösa de ökande problemen med saknade tjänstegarantier i dagens Internet. Nya tjänster såsom video-på-begäran (eng video-on-demand) och IP-telefoni kommer att bli oanvändbara utan någon sorts tjänstegaranti att bygga applikationer på. En ersättningsarkitektur för Integrated Services (IntServ) kommer att behövas eftersom den har problem med extra arbete och skalbarhet.
Denna examensarbetesrapport på magisternivå studerar och evaluerar trafik algoritmer, speciellt scheduling och aktiva köhanteringsalgoritmer, inom Differentiated Services området genom att göra simuleringar med Network Simulator. Studierna undersöker Differentiated Services nätverk med avsende på stabilitet och prestanda med störtrafiksimuleringar. Resultaten visar att mot användare som ej är skötsamma så beror stabilitet och prestanda i nätverket i huvudsak på den använda schedulingalgoritmen.

Contents

List of Figures

List of Tables

Preface

We are two undergraduate students at Lund University, Sweden, majoring in Computer Science. This report constitutes a 20 credit master thesis and its target group is computer scientists with knowledge within the area of data communication. The preparations on the master thesis started fall 1999. We were looking for a subject within the area of data communication, real time systems or operating systems. This master thesis was introduced to us by SwitchCore AB (publ) in Lund in December 1999. The subject felt interesting, especially since the area of Differentiated Services is still developing. We started working on our master thesis between Christmas and New Year's Eve 1999.

We would like to thank our supervisors, Ulf Ahlfors at SwitchCore, Lund and Robert Pallbo at the Department of Computer Science, Lund University, Sweden. Besides our supervisors we would like to thank the staff at SwitchCore who has been of great help as well as LaTeX and David E. Knuth for the layout of this report.

1 Introduction

Today's problem with Internet traffic is that it is managed using best-effort (see section 2.3.1). In short it means that all traffic is handled equally, i.e. with no service differentiation at all. A couple of years ago the Internet community realized that there would be an increasing demand of *Quality of Service* (QoS) [19, 30] on the Internet. Today there is such a demand because new services like IP-telephony and video-on-demand need to be integrated with existing services such as email, FTP and World Wide Web (WWW).

1.1 Quality of Service (QoS)

Kilkki [30] writes that the basic attributes of the *Quality of Service* (QoS) concept are robustness, fairness, versatility and cost efficiency.

There are two major QoS services — *Guaranteed Service* and *Controlled-Load Service*. Guaranteed service provides firm, mathematically provable, bounds on end-to-end datagram queueing delays [39]. The controlled-load specification states that "Controlled-load service provides the client data flow with a quality of service closely approximating the QoS that same flow would receive from an unloaded network element, but uses capacity (admission) control to assure that this service is received even when the network element is overloaded." [45].

QoS has been widely used in other customer areas and is now receiving attention within the community of data communication. QoS introduces the possibility for customers to specify their service level requirements in terms of , e.g., bandwidth for which they will of course pay accordingly. One of the main issues with QoS is what to take into consideration when drawing conclusions on the quality level. Within the area of data communication, QoS has a tendency to be used in a limited sense since there is a lot of focus on measurable technical characteristics, hence packet loss ratio and maximum delay, and not customer support etc.

1.2 Present-Day Quality of Service Solutions

One solution to QoS on the Internet was the specification of the *Integrated Services* (IntServ) model [13]. IntServ was intended to be a robust, integrated-service communication infrastructure [30] working on a per-flow basis.

Verma [44] writes that the way the guaranteed and controlled load services are defined within the *IntServ Working Group* is that they do not necessarily need the *Resource Reservation Protocol* (RSVP) [14] or any other signaling protocol for their operation. However, according to Verma it does appear that RSVP will be the predominant — if not the only — means for defining and configuring IntServ support in routers.

Due to problems with scalability and overhead (because of the per-flow management) the IntServ model is not widely used today. It is nevertheless a predecessor to todays *Differentiated Services* (DiffServ) [7, 10] model. Amazingly, the DiffServ model can be the QoS-functionality that opens the door for IntServ as a lower layer [9].

1.3 Our Work

The purpose of our master thesis is to study the area of *Traffic Management Algorithms* within DiffServ networks by comparing and evaluating simulation results using different algorithms and traffic scenarios. This covers, especially, algorithms handling scheduling and queue management. The main goal for *Scheduling Algorithms* [42] is to guarantee some sort of predefined fairness along with starvation avoidance. The main goals for *Active Queue*

Management Algorithms within TCP/IP networks are to maintain a low buffer level and to guarantee some sort of predefined fairness between PHB classes (see section 2.3), while taking the TCP protocol backoff problem [20, 43] into consideration.

Our work has consisted of four major tasks:

- Study available Differentiated Services material
- Implement/bug fix studied algorithms
- Evaluate algorithms in various scenarios
- Write a report about the three former tasks

During task two we bug fixed and modified almost the entire DS-LIB [35] which now (with our additions) consists of almost 240 kb source code. The algorithms we have implemented on our own are a *Three Colored Random Early Detection* (3CRED) (see section 3.2.2) and five different scheduling algorithms (see sections 3.2.6, 3.2.8-3.2.11). During this task we also implemented several simulation scripts (about 5kb each in size) in TCL which were used to test our various implementations. A library with a simplifying interface towards the NS and DS-LIB TCL libraries was also implemented. This library, called ds-include (about 75 kb), contained procedures for making traffic sources, statistics generation etc and can be seen as the linked list library for simulations (for usage see appendix E).

In task three we implemented several Perl scripts to manage all of our simulations. These scripts (about 100kb in total) post processed and calculated average statistical values of several simulations, thereby minimizing the risk of strange results, to get a more firm base for future conclusions. Also during task three we decided on five final simulations (script total sizes about 120 kb):

- Mean Packet Size Influence
- Noise Influence
- PHB Group Combination Noise Influence
- Color Handling Mode Influence
- Scheduler Influence

The above five final simulations were used to evaluate algorithms implemented during task two.

1.4 Reference Awareness

This report is based on of simulation experiments and the analysis thereof, as well as a thorough background study of articles, RFCs and Internet-Drafts within the area of the report. RFC2026 [15] clearly states that: “Under no circumstances should an Internet-Draft be referenced by any paper, report, ...”. We are clearly aware of this and that Internet-Drafts are working documents and therefore do not have any official status — in fact, they can be removed or replaced at any time by newer versions. Unfortunately, many of the most relevant documents in progress are not available in any other form than Internet-Drafts.

1.5 Organization of This Report

About semantics, we have chosen to use *italic* for new expressions and to emphasize in general. **Bold** is seldom — if ever — used. Sectioning is made with at most two levels.

Section 1 contains a general introduction to this master thesis.

Section 2 describes the Differentiated Services model, its different elements, per-hop behaviors and various algorithms used within DiffServ networks.

Section 3 covers our changes and contributions to both the Network Simulator (ns) [6] and the additional DiffServ library [35] for ns by Sean Murphy.

Section 4 contains information about our simulation environment, general information as well as information about simulation setups and discusses our simulation studies and analysis of the same.

Section 5 is a summary of our work including this master thesis.

Appendix A contains a list of acronyms used in this report.

Appendix B contains definitions of the terminology within the field of Differentiated Services.

Appendix C contains simulation reports for our implemented/changed algorithms.

Appendix D contains a short introduction to Pareto Distribution.

Appendix E contains a TCL example script for NS.

Appendix E contains a sample statistics file from one of our simulation.

2 Differentiated Services

2.1 Differentiated Services Architecture Model

The Differentiated Services Architecture Model is aimed at solving the increasing Quality of Service (QoS) demand by differentiating traffic. The *Differentiated Services* (DiffServ) model can be placed somewhere between the current QoS capabilities available in the Internet (i.e. no real guarantees at all) and the capabilities present in the former *Integrated Services* (IntServ) model [13] (i.e. total guarantees).

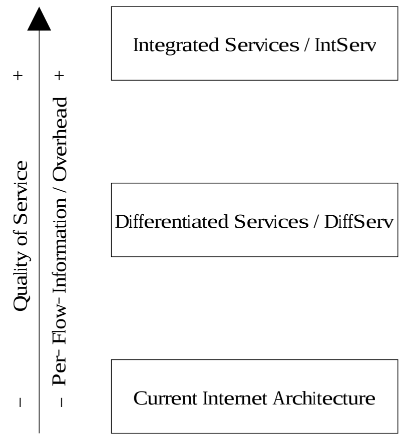

Integrated Services / IntServ
Differentiated Services / DiffServ
Current Internet Architecture
Quality of Service
Per- Flow- Information / Overhead
+
+
–
–

Figure 1: QoS Levels

The IntServ model requires an application to reserve resources for itself along the path to the desired destination. This reservation ensures that the application will be guaranteed to get the desired performance. This use of resources are generally highly inefficient since resources may become idle along a reservation path. This is also one of the main reasons why IntServ is inadequate to be used in any greater extent in the near future. A couple of years ago work started on an alternate QoS scheme which is the model we focus on – Differentiated Services.

The goal of DiffServ is first of all to give good QoS in an internet (an internet being any network using the Internet standards such as TCP/IP etc). Another goal is to be backwards compatible with non-DS domains. Thereby allowing DS domains and non-DS domains to co-exist in the same region as internets become DS-compliant. The solution that is used is to introduce the concept of a *Per-Hop-Behavior* (PHB) (see section 2.3) and *DiffServ CodePoint* (DSCP). The DSCP is a bit-field in the IP header *Type-of-Service* (ToS) Field representing a PHB describing the forwarding treatment the packet should receive (e.g. low delay, low jitter or high bandwidth).

This simple packet marking scheme does not give the same degree of QoS guarantees as IntServ does, since no resources are reserved in advance for every flow. But it is generally believed to be good enough for common use. At times where DiffServ cannot be used, the IntServ or another future model with better QoS could be used.

A more concise description of what was said above would be to state that DiffServ works on a per-hop-level, and the avoidance of per-microflow and per-customer state information within the network reduces overhead which makes it an attractive alternative to current QoS and IntServ.

DiffServ tries to uphold four QoS-attributes [30] which are of main concern for the end-users needs:

1. fairness – from a human viewpoint
2. robustness – network reliability
3. versatility – fulfillment of all user needs
4. cost efficiency – balance of effectively meeting the attributes above

Fairness is an important aspect in any network architecture and is difficult to measure because of its ambiguity. Any network architecture considered unfair by its users is deemed to be unsatisfying. The last three items, robustness, versatility and cost efficiency are more or less measurable in some sense. Robustness refers to *Theft and Denial of Service* and other factors such as network uptime. To fulfill all user needs, some sort of versatility much be achieved. Robustness and versatility affect each other, high versatility decreases robustness of a network. The term cost efficiency relates to the other attributes and the balance between them. A user is usually not willing to pay for a network that is lacking in some aspect. The price must be considered fair by the customer compared to the service received.

2.1.1 DS Domain

A DS domain (see figure 2) consists of a contiguous set of DS nodes (see section 2.1.2). These nodes have to implement the same PHB groups (see section 2.3) as well as share a common service provisioning policy. The service provisioning policy defines how traffic conditioners in DS boundary (see section 2.1.3) nodes are configured and the mapping between behaviors and traffic flows.

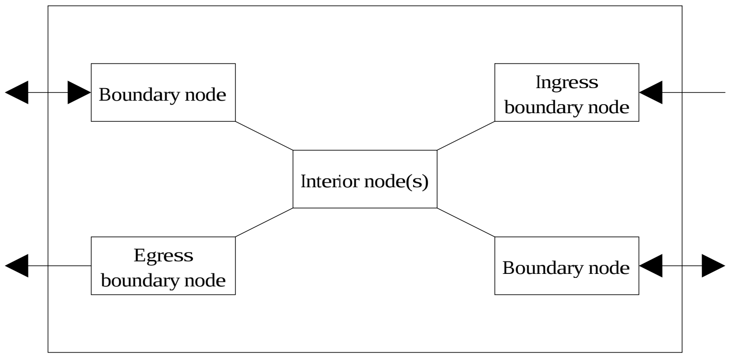


Figure 2: DS Domain

2.1.2 DS Nodes

An internet node is called a DS node if it is DS-compliant (i.e. implements at least one PHB). The default PHB is the *Best Effort PHB* (BE) which really is what most traffic in the current Internet is transmitted as.

2.1.3 DS Boundary Node

A DS node is called DS boundary node if it lies on the edge of a DS domain.

Boundary nodes act as both DS ingress nodes and DS egress nodes for different directions of traffic. An ingress node is the definition of a node which accepts incoming traffic to a domain while an egress node is a node that accepts outgoing traffic. The very important job of a boundary node is to make sure all traffic flows entering or leaving the DS Domain conform with the appropriate profile, i.e. *Service Level Specification* (SLS).

Incoming traffic to a boundary node adjacent to a non-DS domain use the Multi-Field (MF) classifier (see section 2.2.1), or if adjacent to another DS-domain possibly the Behavior Aggregate (BA) classifier (see section 2.2.1). The performance gained is received in the Interior nodes, since when forwarding a packet the classifier only needs to examine the DSCP, whereas the MF classifier needs to examine several other fields as well.

2.1.4 DS Interior Node

A DS node is called DS interior node if it is not a DS boundary node. Usually, it is adjacent to other DS interior nodes.

2.1.5 DS Region

A set of adjacent DS domains is called a DS Region (see figure 3).

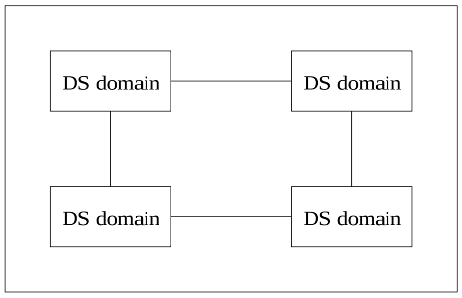


Figure 3: DS Region

2.2 Differentiated Services Elements

This subsection will describe the six major Differentiated Services (DiffServ) elements:

1. classifier
2. meter
3. marker
4. dropper
5. shaper
6. monitor

The first two, the classifier and the meter, are used to determine the appropriate action for packets. The following three, the marker, the dropper and the shaper perform actions, such as policing, on packets. Policing is a general term for the action of preventing a traffic stream from using more than its share of resources in a DiffServ network. Markers police a particular traffic stream by remarking submitted packets. Droppers and shapers do so by limiting the rate at which packets are submitted to a certain traffic stream. Finally, monitors simply measure and account for traffic flows. Actually, it does not need to be a separate mechanism as it can be incorporated in a meter-marker for ease of implementation.

2.2.1 Classifier

Differentiation is the very heart of DiffServ networks and because of that classification has a central role in it. Classifiers [8, 10] use the packet header, or some portion of it, in combination with a field rule for classification. A classifier determines the logical output stream to which the packet should be directed for further processing.

Within DiffServ there are two main types of classifiers — the *Behavior Aggregate* (BA) classifier and the *Multi-Field* (MF) classifier. The BA classifier *only* uses the *Differentiated Service CodePoint* (DSCP) (see section 2.3) and is used mainly by DS interior nodes and possibly by DS boundary nodes adjacent to other DS domains. The *Multi-Field* (MF) classifier on the other hand classifies packets using a combination of one or more header fields, such as DSCP field, source and destination address, source and destination port, IP protocol number and other similar information such as incoming interface. This is mainly used by ingress nodes adjacent to non-DS domains.

2.2.2 Meter

Meters [8, 10] monitor the arrival times of packets on an incoming traffic stream and determines the conformance level for each packet against a parameterized profile (SLS). It then passes the conformance level information onto other DiffServ mechanisms, usually the marker which it is closely bound with. There are different types of meters, e.g. the *Average Rate Meter*, the *Exponential Weighted Moving Average Meter* (EWMA) and the *Token Bucket Meter*.

The Average Rate Meter is very simple, as it only measures the average rate at which packets arrive to it over an averaging time interval. An average rate profile may take the following parameter form: average rate (e.g. in packets per second) and averaging interval (e.g. per second). As long as an arriving packet does not push the continuously maintained average rate of arriving packets it is deemed conforming, otherwise non-conforming. Hence, the average rate meter has two conformance levels.

The EWMA Meter [8] has a formula which is easy to implement in silicon. For every incoming packet it calculates a new average queue size taking into account the current average queue size and the actual queue size. This is done using the following formula:

$$avg_{n+1} = (1 - Gain) * avg_n + Gain * queuesize_{n+1},$$

where the gain controls how much the previous average queue size should impact on the one being calculated. Any incoming packet that causes the average queue size to exceed a predefined average rate is deemed non-conforming, otherwise conforming.

The Token Bucket Meter [8] is a more sophisticated meter. It measures conformance to a token bucket (TB) profile. The TB profile generally have three parameters: an average rate, a peak rate and a burst size. Logically, it consists of a token bucket, where tokens are added with an average rate up to a maximum credit — the burst size. An incoming packet of size S at time T is deemed conforming if there are at least S tokens in the bucket at time T. Packets are allowed to exceed the average rate at bursts of burst size, as long as they do not exceed the peak rate at which the bucket will be drained. Incoming packets are deemed non-conforming if there are insufficient tokens in the bucket.

Figures 4 and 5 display how a simple token bucket work.

There are also more advanced token bucket meters, who often are implemented together with a marker, e.g. the *Multi-Stage Token Bucket Meters*: the *Single Rate Three Color Marker* [23] using three conformance (color) levels and the *Two Rate Three Color Marker* [24] using two burst sizes.

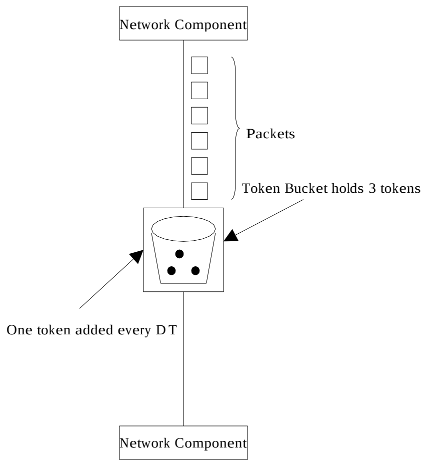


Figure 4: Token Bucket before use

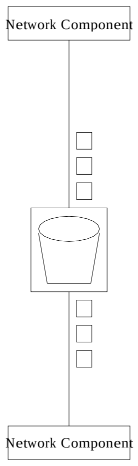


Figure 5: Token Bucket after use

2.2.3 Marker

Incoming packets that are unmarked (i.e. DSCP field of zero) or have been decided to be remarked by previous meter/classifier have their DSCP field set by a marker [8, 10]. Setting the DSCP field subsequently determines a packet's treatment both by the current node and by any other downstream nodes in the DiffServ network. A marker can be parameterized in several ways. Typically, it acts on non-conformant packets and therefore changes the drop precedence of an *Assured Forwarding* (AF) PHB packet (see section 2.3.2), or simply sets the DSCP field to the *Best Effort* (BE) PHB (see section 2.3.1). Incoming unmarked packets get marked according to the *Service Level Specification* (SLS).

2.2.4 Dropper

Droppers [8, 10] drop packets on the incoming traffic stream according to some predefined dropping method for the queue. The traditional non-DiffServ Internet only used the drop-tail method. It got its name from its behavior, i.e. when a packet needs to be dropped it is dropped from the tail of the queue. Other variations exist such as drop-front, which drops packets from the front of the queue and drop-random, which randomly selects a packet to drop when dropping is needed. Drop random is quite expensive when compared with drop-tail or drop-front.

2.2.5 Shaper

A shaper [8, 10] adjusts the downstream rate of non-conforming packets to conform with the downstream profile based on the SLS by delaying some or all of the incoming packets. Shapers usually have a finite-size FIFO buffer. In the AF and EF PHB classes reordering in the shaper is not allowed (see section 2.3.2). In case of overflow, i.e. too large bursts, the shaper may drop packets in some fashion (e.g. drop-tail, drop-front, drop-random).

2.2.6 Monitor

The monitor [8] can be seen as a passive action element making account for the traffic flow. Typically, this can be used for presenting statistics for customer usage-based billing.

2.3 Per-Hop-Behaviors (PHB)

Without the possibility to separate packets from each other we cannot make any guarantees about delay, jitter or bandwidth. In order to separate different flows, the concept of a *Per-Hop-Behavior* (PHB) is introduced in *An architecture for Differentiated Services* [10]. A PHB is a description of an externally observable behavior of a packet. This behavior could be loss, delay and jitter. A simplification of this description would be to say that each PHB will get a minimum predefined amount of resources. These resources are allocated to achieve the desired behavior.

Since the network has to be able to provide this differentiation of packets, one has to be able to perform marking/remarking of packets. This marking will be done by either DS boundary nodes or by the customer. The marking uses six bits of the IPv4 ToS-field or the IPv6 traffic class octet. These six bits are called *Differentiated Services CodePoint* (DSCP) [36]. Each PHB that becomes standardized will be assigned a unique DSCP. The remaining two bits in the octet are the *Currently Unused* (CU) bits. The CU bits are used for experimental purposes such as *Explicit Congestion Notification* (ECN) [38].

The IPv4 ToS-field and the IPv6 traffic class octet can be found in figures 6 and figure 7.

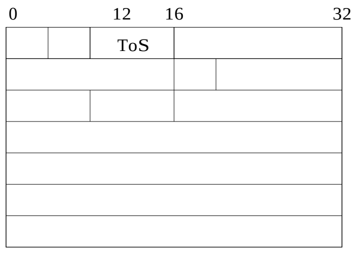


Figure 6: IP Header version 4

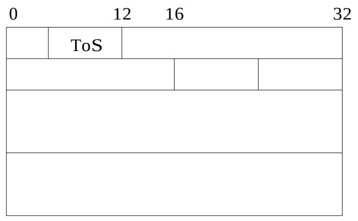


Figure 7: IP Header version 6

A PHB is *Class Selector Compliant* if it conforms with the specifications in RFC2474 [36]. The specification reserves codepoints in the form xxx000 for the use of *CS compliant* PHBs, since the first 3 bits of the old ToS field was used for 8 levels of packet precedence. This backward compatibility will make the network more stable during the transition phase from traditional internets to a DS-capable network.

2.3.1 Best Effort PHB

IP-packets in the current Internet are classified according to the *Best Effort* (BE) PHB. No guarantees about bandwidth, loss or delay is made in this PHB. The BE PHB is the default PHB in the DS architecture.

The codepoint allocated for this PHB is 000000.

2.3.2 Assured Forwarding PHB

The *Assured Forwarding* (AF) PHB [22] is one the two basic PHBs in the current DS architecture.

In the AF PHB specification [22] the current recommendation for the AF PHB is that it should be divided into four classes and within each class the current recommendation are to have three different levels of drop precedence (priority). Resources are allocated separately for the four AF classes. All classes will have different types of resources allocated to it, and provide different services. Therefore a need to set up a contract between the customer and an *Internet Service Provider* (ISP) exist. This contract is called *Service Level Specification* (SLS). In the SLS there exist parameters that limit the traffic rate at which the customer will be allowed to transmit. This could be done in the form of a Token Bucket Meter-Marker (see sections 2.2.2 and 2.2.3, thereby specifying the burst size and at the average rate.

Since the number of classes and drop precedence levels are recommendations only, it should be discussed whether it is enough or to much with four classes and three drop precedence levels. Depending on new standardizations of PHBs and the expansion of more services in the Internet, additional classes/precedence levels might be needed.

In the AF PHB RFC [22] it is said that if there are excess resources not in use by any other PHB, these resources could be allocated in 2 different ways:

- divide all excess resources equally between the AF PHB and BE PHB
- let the AF PHB use as much of the excess resources as they need and then give the remaining resources to the BE PHB

In order to manage the size of the buffer between two nodes, and to minimize the drop ratio, the AF PHB should use active queue management algorithms (see section 2.4) such as *Random Early Detection* (RED), *RED In/Out* (RIO) or some other mechanism.

Reordering of the AF PHB packets within a microflow is not permitted.

We conclude this section with a table of all recommended codepoints for the AF PHB:

	Class 1	Class 2	Class 3	Class 4
Low Drop	001010	010010	011010	100010
Medium Drop	001100	010100	011100	100100
High Drop	001110	010110	011110	100110

Table 1: AF Class Table

2.3.3 Expedited Forwarding PHB

The *Assured Forwarding* PHB group does a decent job when dividing guarantees into different groups, but when one needs absolute guarantees about loss, delay, jitter and bandwidth, the *Expedited Forwarding* (EF) PHB class is what should be used. The EF PHB class is described in *An Expedited Forwarding PHB* [27]. The EF PHB receives its resources independent from other PHBs, just like the different AF classes do in the AF PHB. Also like in the AF PHB an SLS is used as a contract specifying the traffic parameters. In contrast to the AF PHB, the EF PHB specifies that all traffic exceeding the traffic parameters should be dropped immediately.

In the EF PHB, no advanced queue management is needed. This is due to the fact that packets should experience minimal delay, and therefore queues should be nonexistent or at least minimal in length.

The primary use for the Expedited Forwarding PHB is expected to be the virtual leased line [29]. The customer will expect to achieve the same performance that had been achieved with a real dedicated cable.

Reordering of EF PHB packets within a microflow is not permitted.

The DSCP allocated to the EF PHB group is 101110.

2.3.4 Expedited Forwarding with Dropping PHB

Like the *Expedited Forwarding* PHB, the *Expedited Forwarding with Dropping PHB* (EFD) PHB [17] provides low delay but can experience packet loss with a certain probability. The EFD PHB is suited for low delay services in cellular wireless networks, featuring low delay, low jitter, but as mentioned a certain degree of packet loss. The authors claim that neither AF PHB or BE PHB can provide low delay since bursts are allowed, and may have large queues during times of heavy load. The EFD PHB does not steal any resources from any other PHB since it uses resources assigned to the EF PHB group. Therefore performance of the AF and BE PHB groups, in a network also implementing the EFD PHB, will not suffer.

Since the EF PHB and EFD PHB share the same resources it is proposed that they be implemented together as a PHB group. Also since we do not want the EFD PHB to be completely starved the relationship between them is strictly defined in the draft [17].

As to non-wireless networks it is unknown if any advantages with the EFD PHB exist at all.

The EFD PHB has no DSCP allocated to it yet. It is currently in draft stage.

2.3.5 Lower Than Best Effort PHB

The *Lower Than Best Effort PHB* (LBE) [11] is a PHB specification currently in draft stage. The main use for this PHB would be to shield the BE PHB from unfairness, which could be important. The reason for this unfairness is that non-conforming (according to some profile) senders with some PHB currently will have their packets remarked to the BE PHB. *Best Effort* should not suffer because of other misbehaving PHBs. The authors of the LBE PHB propose that demoting non-conforming PHBs to the LBE PHB instead of the BE PHB. This would eliminate the unfairness. The BE PHB and LBE PHB should share the same queue, and use some sort of Queue Management Algorithm like RED, in which the LBE PHB should be dropped harder than the BE PHB.

Other uses for the *Lower Than Best Effort PHB* would be low priority bulk transfer and backup transfers deemed non-important. UDP could also be sent as the LBE PHB.

Even though the name implies that treatment is the worst possible, the LBE PHB must receive a minimal amount of resources to prevent starvation.

If there are excess resources the LBE PHB is allowed to use them. Like in the EF PHB and AF PHB, packets belonging to the same microflow may not be reordered.

The DSCP allocated for the LBE PHB is 101011.

2.3.6 Alternative Best Effort PHB

An interesting suggestion for a new PHB can be found in the draft by Hurley et al. [26]. This PHB, called Alternative Best-Effort (ABE), offers applications a choice between receiving either low end-to-end delay or better throughput. The ABE PHB packets are marked as either green (low end-to-end delay) or blue (better throughput). Since green packets receives lower end-to-end delay it also is restricted to receive less throughput than blue packets. The possibility of having both good throughput and low end-to-end delay is beyond the scope of the ABE PHB, and other services such as AF or EF are better suited for this purpose.

Green packets would ideally be used for transporting real-time traffic such as voice over IP, while blue packets could be used for bulk data transfers.

The ABE PHB service requires no usage control, since it will probably be located at the bottom of the service list in a scheduler, where the current the BE PHB is located. As said, the ABE PHB is not required to police how much green traffic is allowed to be sent. Also the applications using this PHB are required to be TCP-friendly, i.e. a flow does not receive more throughput than a TCP flow would have.

In order to make sure that not all packets are sent as green, the PHB requires that if some sources mark their packets green rather than blue, then the throughput must be at least as good as it was before the change of marking. Therefore green packets should be dropped with a higher probability than blue packets.

The authors say that the ABE PHB can not be achieved by using the AF PHB. This because if blue and green were to be placed in the same AF PHB with green having higher precedence than blue, the requirement that blue should receive at least as much throughput as if it was green would be contradicted. The other alternative would be to map green and blue to different AF classes, but this would contradict the AF specification [22] saying that coordination of packet dropping across two classes is not allowed.

No codepoints are allocated for the ABE PHB, but the recommendation is to map blue packets to the current BE DSCP value, i.e. 000000. The DSCP for green packets would be allocated from the experimental/local pools (xxxx11 or xxxx01).

2.3.7 Interoperability PHB

Kilkki presents a new PHB in his draft, the *Interoperability PHB Group* (IO, called PHB-i by author) [31]. The PHB Group has 2 classes (i.e. low or high urgency) and within each class there is at least 6, preferably 8 levels of drop precedence (i.e. importance).

The main use for this PHB would be to provide interoperation between ISPs. Imagine the scenario where one operator assigns DSCPs to packets based on pricing, and the other operator assigns DSCPs based on application. Since service providers have this freedom when selecting DSCPs for PHBs, The IO PHB is one possible solution to the problems of cooperation. The only question that should be asked is if it is really necessary to use 16 codepoints for this single purpose. 16 codepoints should be enough to map different DSCPs easily between each other. In any case some kind of traffic class interface is needed between service operators in DiffServ networks.

As mentioned above, the IO PHB has two classes with eight codepoints to each class. The recommended codepoints are:

	Urgency	
Importance	0	1
7	111011	111111
6	110011	110111
5	101011	101111
4	100011	100111
3	011011	011111
2	010011	010111
1	001011	001111
0	000011	000111

Table 2: IO PHB Table

Traffic that enters a domain as the IO PHB with high urgency and importance needs to be limited to protect against misbehaving users.

The draft presents three possible applications for the IO PHB:

- IO PHB is used only for interoperability purposes as mentioned above
- IO PHB is used mainly for interoperability purposes, but also to some extent within a domain
- IO PHB is used as the only PHB within a domain

We described only the first of these possible applications in this section, since we feel that it is the one most interesting. A more exhaustive description can be found in the draft by Kilkki [31].

2.4 Active Queue Management Algorithms

In the early days of Internet no congestion control existed. Congestion is the event when the network traffic load is so heavy, that all packets can not be transported to their destination safely. Therefore packets are dropped. In the mid 1980's the Internet collapsed due to the fact that no congestion control existed. The number of users on the Internet was growing faster than the network could handle, and this increased the overall traffic load of the network. This crisis of having no congestion control was solved by V. Jacobson when he introduced his TCP backoff algorithm. This algorithm, or variations of it, is now implemented in all TCP senders by default. The backoff algorithm treats dropped packets as signs of congestion in the network, so once an acknowledgement do not arrive within a specified period of time the TCP sender decreases its output rate and contributes to lowering the traffic load in the network. For this reason TCP flows are called responsive whereas UDP flows are called non-responsive, due to the fact that they do not decrease their sending rate when congestion occurs.

TCP backoff and drop-tail queues (see section 2.2.4) pose a problem. If a number of flows are sending packets to a node, and then suddenly the queue overflows and starts dropping packets from all flows connected to it, then all TCP senders will eventually time out and enter backoff phase, thereby all of them setting their current window to one packet. After a while the queue will start building up and then overflow again. This synchronization problem presents a problem for any network using TCP.

To solve this problem the concept of *Active Queue Management* (AQM) was introduced by Floyd et al. [12], and the main algorithm used is *Random Early Detection* (RED) [20]. Passing a packet through an AQM algorithm before actually placing it in the queue gives the possibility of increased performance and stability. Moreover, the average queue size is kept at reasonable levels, thereby minimizing the queuing delay and offering an additional roof for temporary bursts.

Another way to improve the problem with degraded performance at times of congestion is to improve the TCP senders. Variations of the most basic one TCP/Tahoe exist, and they all work a bit different when entering backoff phase. Examples of variations would be TCP/Reno and TCP/Vegas.

2.4.1 Random Early Detection (RED)

This is the most basic of the active queue management algorithms, and was first described in *Random Early Detection Gateways for Congestion Avoidance* [20]. The algorithm is based on calculating an exponential weighted moving average (see section 2.2.2) of the queue size for every packet arrival. Based on this average queue size the RED gateway drops packets with an certain probability. The gateway drops no packets when the average queue is below the min threshold (min_{th}) parameter. When the average queue size exceeds the max threshold (max_{th}) parameter all incoming packets are dropped or marked, and in between min threshold and max threshold packets are dropped/marked with a probability. Figure 8 displays the relationship between these parameters.

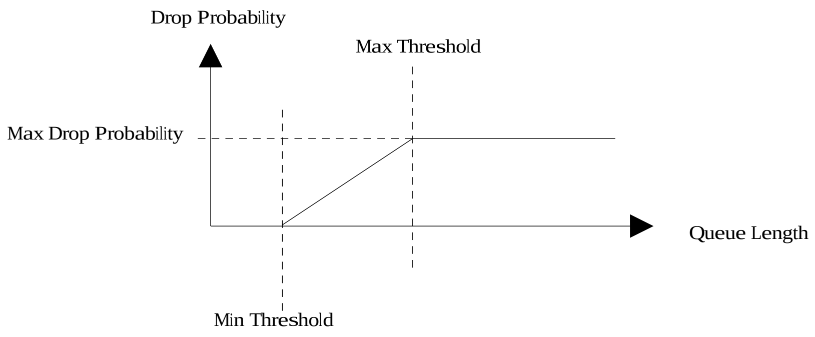


Figure 8: RED Graph

To get a more uniform distribution of the drops a count parameter is used that help even out the drops out over a longer period of time. Figure 9 [20] illustrates the difference between using a count parameter and not using it. The figure displays a drop with a dot over a period of 5000 packets. Flow one is not using count while flow two is using count to even out the drops.

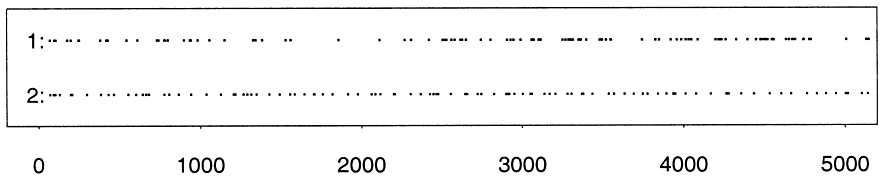


Figure 9: Non-Count(1) vs Count(2)

Since the algorithm is based on an average queue size, the RED gateway allows short bursts of packets to be sent without action, but with longer bursts packets will be lost.

The pseudo code for the RED algorithm [20] is:

```
Initialization:
$avg = 0$
$count = 1$
$q\_time = time$ (added)
for each packet arrival
  calculate the new average queue size $avg$ :
     if the queue is nonempty
         $avg = (1 - w_q)avg + w_q q$
     else
         $m = f(time - q\_time)$
         $avg = (1 - w_q)^m * avg$
  if $min_{th} \leq avg \leq max_{th}$
     increment $count$
     calculate probability $p_a$ :
         $p_b = max_p * (avg - min_{th})/(max_{th} - min_{th})$
         $p_a = p_b/(1 - count * p_b)$
     with probability $p_a$ :
            mark the arriving packet
            $count = 0$
    else if  $max_{th} \leq avg$
        mark the arriving packet
        $count = 0$
    else
        $count = -1$
when queue becomes empty
    $q\_time = time$
```

Saved Variables:
```
    $avg$ : average queue size
    $q\_time$ : start of the queue idle time
    $count$ : packets since last marked packet
```
Fixed Parameters:
```
    $w_q$ : queue weight
    $min_{th}$ : minimum threshold for queue
    $max_{th}$ : maximum threshold for queue
    $max_p$ : maximum value for  $p_b$
```
Other:
```
    $p_a$ : current packet-marking probability
    $q$ : current queue size
    $time$ : current time
    $f(t)$ : a linear function of the time t
```

Simulations using RED can be found in appendix C.1.

There still exist a couple of problems with RED. Firstly, it is sensitive to parameter settings (see section 4.3.2). It is hard to know what to set min and max thresholds to. It is recommended in [20] that max threshold should be set to at least twice the min threshold.

Another problem is how to identify misbehaving flows, for example an UDP flow. Misbehaving flows needs to be separated from behaving flows, otherwise they will take damage. The basic RED gateway does not achieve this very well, but neither did the traditional Internet solution with drop-tail queues.

Since the basic RED version has a number of problems various researchers have tried to make modifications to RED to solve these problems.

In *A Self-Configuring RED Gateway* [18] the *Adaptive RED* (ARED) is presented. ARED tries to solve the problem with sensitive parameter settings. The basic RED algorithm is extended in the following manner.

```
For every avgq update:
      if (min_thresh < avgq < max_thresh)
             status = Between
      if (avgq < min_thresh && status != Below)
             status = Below
             p = p / alpha
      if (avgq > max_thresh && status != Above)
             status = Above
             p = p * beta
```

This code makes RED drop packets less aggressive when avgq were less than min threshold and more aggressive when avgq were greater than max threshold. This makes the RED gateway somewhat automatically adjusted to the current traffic load.

Another variant of RED is presented in *RED in a different light* [28]. In this draft the authors claim that sampling on fixed intervals will improve RED performance, in contrast to the original RED which samples every time a packet arrives. They also introduce a way for RED gateway parameters to be calculated by themselves which could be useful. It remains to be seen if this draft will be used in practice or not.

In *Stabilized RED* (SRED) [37] the AQM algorithm keeps statistics about arrived packets in the past, in something called zombies. For every packet arrival the algorithm checks randomly against a zombie list and use the result to determine whether to mark/drop or not. If a match between the arriving packet and the randomly selected packet from the zombie list is found, then the packet is dropped.

In an article named *Dynamics of RED* [33] the authors present another RED variant called *Flow Random Early Drop* (FRED). As the name implies the algorithm keeps information about the active flows going through the gateway. The algorithms maintains drop probability and drop threshold for every flow. By separating the different flows you can identify misbehaving (unresponsive to congestion notification) and penalize them if needed. This is exactly what FRED does. The algorithm also maintains a counter for every active flow called strike. The greater strike, the harder the treatment.

The concept of keeping information about active flows is interesting. However one must be cautious since as the number of flows increase, so does the need for memory. If one refers to flow as a non PHB flow, FRED does not comply with the DiffServ architecture, but if you allow a more general interpretation of flow it could be used to separate different precedence levels within a PHB.

In the article they run a number of different simulations to test FRED. To be mentioned here is a test that verify that the fairness between different flows are maintained. One TCP sender, and a UDP sender is competing for the same 10 Mbps bandwidth. With original RED, the UDP source was in control of 8 Mbps of the link. With FRED, the TCP sender managed to gain it is fair share of 5 Mbps bandwidth

In *Balanced-RED: An algorithm to Achieve Fairness in the Internet* [4] the Balanced RED (BRED) is presented. BRED is like a combination between ARED and FRED. It keeps information about separate flows just like FRED. It has two different levels of drop probability for every flow. They claim to maintain a good balance and improve fairness between flows in the Internet. BRED does not penalize misbehaving flows explicitly like FRED does. This is claimed to be worked on in a future version.

All the above modifications improve original RED in their own way. There will surely exist more variations of RED in the future.

For a more analytical model of RED see *Analytic Evaluation of RED performance* [34].

2.4.2 RED In/Out (RIO)

RED In/Out (RIO) [16] is a more advanced AQM algorithm than RED is. RIO features two levels of drop probability (see figure 10). That is, the algorithm can differentiate between packets and depending on what type it drops with low or high probability. The two types of packets used in RIO is commonly called IN or OUT packets. IN packets are dropped with lower probability than OUT packets. Under light load the RIO gateway receives only IN packets. Since there is no risk of congestion there is no need to start dropping packets as hard as we would have under heavy load.

Because the algorithm is based on two packet types, it needs separate variables for each type as well. RIO uses two sets of thresholds, and two average queue sizes. Whenever a OUT packet arrives only OUT related variables are updated, but when an IN packet arrives both IN and OUT variables are updated.

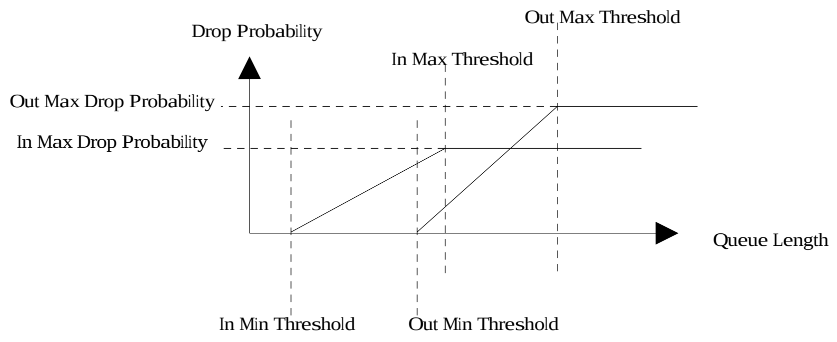


Figure 10: RIO Graph

The pseudo code for the RIO algorithm is:

```
For each packet arrival
    if it is an IN packet
        calculate the average IN queue size  $avg\_in$
    calculate the average queue size  $avg\_total$

If it is an IN packet
    if  $min\_in < avg\_in < max\_in$
        calculate probability  $P_{in}$
        with probability  $P_{in}$  drop this packet
    else if  $max\_in < avg\_in$
        drop this packet
If it is an OUT packet
    if  $min\_out < avg\_total < max\_out$
        calculate probability  $P_{out}$
        with probability  $P_{out}$  drop this packet
    else if  $max\_out < avg\_total$
        drop this packet
```

The separation between IN and OUT packets is usually performed by means of a Token Bucket Meter-Marker (see section 2.2.2). As long as the Token Bucket Meter-Marker profile does not overflow all packets are marked as IN, but all packets that overflow the Token Bucket Meter-Marker profile are marked as OUT.

In conclusion RIO offers better performance during light/medium traffic load than RED does, but during heavy load RED and RIO do not differ much at all (since all packets will overflow the token bucket and be marked as OUT).

2.4.3 RED with 3 Colors (3CRED)

The three color RED extends RIO into three colors. Each color, green, yellow and red, represents a conformance level. Green, representing conforming, can be seen as 'RIO-in'. Yellow, representing partially conforming, and red, representing non-conforming, can be seen as 'RIO-out'. Figure 11 displays the relationship between the colors.

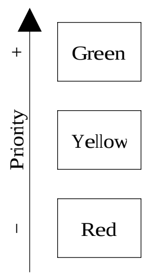


Figure 11: Color Priority Levels

In all color-aware variants of RED, there are a number of parameters and state variables that must be maintained and administrated. This can be done in several ways. The following four color handling modes were discussed in an article on Multicolor-RED by Goyal et al. [21]:

1. Single Accounting, Single Threshold (SAST)
2. Single Accounting, Multiple Threshold (SAMT)
3. Multiple Accounting, Single Threshold (MAST)
4. Multiple Accounting, Multiple Threshold (MAMT)

In the article the definition of the (buffer) accounting term is a little vague. It seems to refer to the accounting as the average queue size, but it could in fact also be the actual queue size or both.

In *Single Accounting* mode only one average queue size is used while in *Multiple Accounting* mode a separate average queue is used for *every* color. The *Single Threshold* mode means that one set of threshold parameters, min_{th} and max_{th}, is used and *Multiple Threshold* mode that there is a set for *each color*. Observe, that the parameter max_p is not included.

The SAST mode is insufficient because of its color-blindness hence it is a simple RED algorithm. The MAST mode is intuitively meaningless since it a trivial case of MAMT. The three modes SAST, SAMT and MAMT will all be omitted in this treatment and focus will be on MAMT which actually includes the other three as special cases.

In the *Multiple Accounting Multiple Threshold* mode the average queue size can be calculated in a number of ways. For simplicity it is our recommendation to divide MAMT into *MAMT-isolated* and *MAMT-higher-priority-inclusive* to differentiate between the two (most likely) calculation options that where mentioned in the same Multi-RED paper by Goyal et al. In the *higher-priority-inclusive* option the average queue size for a color is calculated using the queue size of the same or higher priority colors. The key is that a conformance level (color) should not be lacking in performance because of any other, lower priority, conformance level is draining its own resources. The isolated option is defined, so that the average queue size for *each* color is calculated using the queue size of *only* that color.

An observation by Kim et al. [32] suggests that isolation of thresholds in RIO is not sufficient. Given 3CRED is an extension to RIO, MAMT-hp-inclusive would be the suggested option. We are of the same opinion because of the "non-lacking-performance-reason" above. Also, the Buffer Management Techniques does not clarify how other traffic-dependent variables such as idle, idle time and count, or non-traffic-dependent variables such as max_p should be handled. We are of the opinion that such variables should be treated accordingly based on calculation option (see section 3.2.2).

Finally, conformance level (color) classification is, as in RIO, usually performed by a Token Bucket Meter-Marker (see section 2.2.2).

2.5 Scheduling Algorithms

The scheduler is the instance in a node that decides which of the queues is the next to put a packet on a link. Its job is to isolate these queues from each other so as to prevent that any queue gets completely starved, i.e. the queue do not get the chance to send any packets at all. In addition to prevent against starvation, its job is also to uphold fairness between the queues.

With some type of scheduling algorithms it is possible to give a worst-case estimation of the delay a packet suffer in the queue. This is useful when implementing real-time applications.

2.5.1 Strict Priority (SP)

In *Strict Priority* (SP) [42] scheduling a priority is set on each queue. Then when the scheduler selects which queue to transmit from, it does so on the basis of the priority. As long as there is a non-empty queue with higher priority than some other queue, that queue gets to transmit all its packets before the queues with lower priority will be able to send their packets.

The SP scheduler is very vulnerable to starvation, since as long as there are packets with higher priority, no other packets will be sent. This usually introduces unfairness between the queues as well. Its hard to use this type of scheduler alone in practice, because of this unfairness. The positive side of SP is that it is easy to implement and understand. Its also fast and do not require any computations, just some comparison between the priorities.

There exist a couple of small variations that is possible with SP scheduling. One of these has to do with what to do when a packet with higher priority arrives and the scheduler currently is sending a low-priority packet. Should the scheduler then interrupt the packet with lower priority in favor of the packet with high priority (known as preemptive) or should the scheduler wait to send the high priority packet until the packet that is currently being sent is completed (known as non-preemptive).

2.5.2 Weighted Round Robin (WRR)

Most of the times using just one queue is highly undesirable due to the lack of not being able to differentiate between sending sources. To resolve this issue one could use *Round Robin* (RR). In RR the scheduler maintain a set of queues, and then advance between them in a circular fashion.

Weighted Round Robin (WRR), an extension of RR, keeps a weight on each queue. These weights defines the balance the scheduler should try to maintain between the queues. In this way the bandwidth between the sources can be divided in a configurable way, instead of every source getting the same amount of bandwidth.

WRR maintains moderately good fairness between the queues and prevents the starvation of any queue. The algorithm itself is not very hard to understand nor to implement.

2.5.3 Deficit Round Robin (DRR)

Deficit Round Robin (DRR) [40] is based on the same idea as WRR, that is cycling through the queues using RR. In DRR the weights decide a quantum of how much the queue is maximally allowed to send in a round. If the quantum allocated is sufficient to send a packet, the quantum is decremented and another attempt to send a packet is made. If the quantum is not sufficient to send another packet, the queue waits until the next round start.

The difference between DRR and WRR is that in DRR the proportions in weights is counted also in cases where packet sizes vary.

DRR can operate in a couple of different modes. The three we have studied are:

- Non-Interleaved
- Interleaved
- Deficit Sorted

In Non-Interleaved mode the algorithm first checks if queue one can send, and then allows it to send as many packets as it can according to its credit before it proceeds to the second queue, and so on. This turn-based-behavior results in bursts of many packets from same queue, which is highly undesirable.

In Interleaved mode the goal is to break the bursts apart. Here queue one is allowed to first send one packet if the queue has enough credits. Then the second queue is allowed to send one packet. Then we loop until no queue wants to send or has any credits left. At this point the round is finalized.

In Deficit Sorted mode, as we call it, the idea is to choose the queue that has the most deficit bytes at the moment. Therefore a sorted list (according to deficit bytes) is maintained, and whenever we need to send a packet, we let the queue with the most deficit bytes send. Note that the sorting may be defined by any arbitrary function.

The implementation of DRR is quite easy. The complexity is the same as WRR and DRR maintains at least — if not better — fairness than WRR between the queues.

2.5.4 Worst-Case Fair Weighted Fair Queuing + (WF^2Q+)

The work within the area of Fair Queuing began in 1987 when Nagle proposed a *Fair Queuing* (FQ) [43] Algorithm. The reason behind this algorithm was that Internet used something called *Choke Packets* [43]. *Choke Packets* is basically a request for a source to decrease its sending rate, much like *Explicit Congestion Notification* (see section 2.3). Consider the scenario with 4 sources sending packets to a single node. At some point the node will become swamped and send *Choke Packets* to the sources. One of them cuts back on the sending rate, but the other three maintains its' sending rate. This is why there is a need for some kind of fair queuing in the Internet. FQ maintains a Queue for every source. When the algorithm chooses the next packet, it does so in a *Round Robin* (RR) fashion. The FQ algorithm had some problems though. It gives more bandwidth to large packets than to small packets. Hence Demers et al. suggested in 1990 an improvement in which the algorithm does RR in a byte-by-byte round robin instead of packet-by-packet round robin. It sorts the packet on the basis of the assumed finished time, and then the packets are sent in that order. The major problem with this algorithm is that it gives the same priority to all sources. The extension was called *Weighted Fair Queuing* (WFQ). WFQ maintains weights for all the queues, and the higher the weight the more bandwidth the queue is given at every tick.

WFQ tries to replicate a model called *Generalized Processor Sharing* (GPS). In this model packets are sent in parallel on the link. Since this is not directly applicable, WFQ tries to replicate this behavior as good as possible.

The difference between standard WFQ and *Worst-Case Fair Weighted Fair Queuing* (WF^{2}Q+) [5] is that in WFQ when a queue is to be selected, the Scheduler examines all of the queues. In WF^{2}Q+ only queues that started sending according to the corresponding GPS model is examined. With some other minor changes to the calculations WF^{2}Q+ achieves the complexity of O(log N), whereas ordinary WFQ has the complexity of O(N), where N is the number of queues.

To explain how WF^{2}Q works a little better, suppose we have packets arriving according to figure 12. Packets marked with 1 is smaller than packets marked with 2.

With the packet arrivals in figure 12, GPS would service the packets according to figure 13.

As explained earlier, the main goal for WFQ and WF^{2}Q is to emulate the behavior of GPS. The service order for WFQ can be seen in figure 14.

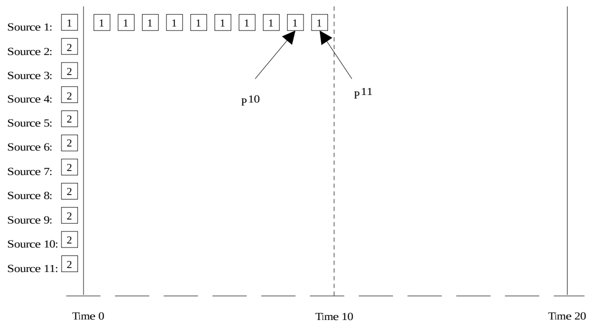


Figure 12: Scheduler Packet Arrivals

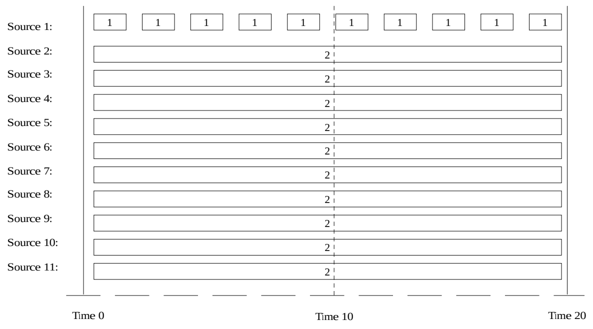


Figure 13: GPS Service Order

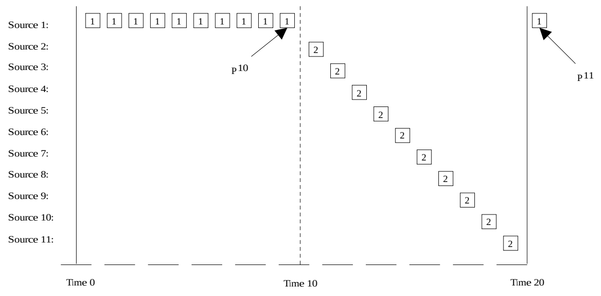


Figure 14: WFQ Packet Service Order

Lastly, figure 15 describes the service order when using a WF^{2}Q scheduler. As can be seen this scheduler does a very good job in emulating GPS.

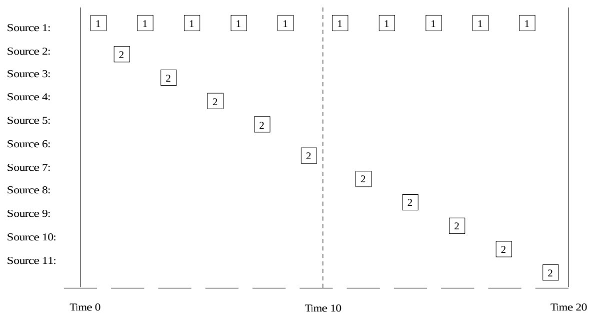


Figure 15: WF^{2}Q Packet Service Order

WF^{2}Q+ prevents starvation of queues, and achieves the best fairness between queues. WF^{2}Q+ is harder to understand and to implement than most algorithms, but has the advantage of being both moderately fast and extremely fair.

3 Our Work - Implementing a DS Simulator Library

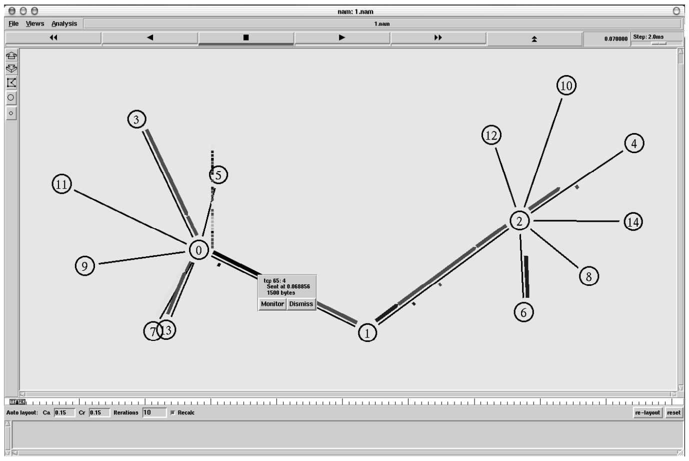
The simulator we have used is the *Network Simulator 2* (*ns-2 2.1b6*) [6] released 18-Jan-2000 (from now on just called NS). The choice of using NS was initially made by SwitchCore, and unless we had any major objections against NS this was the simulator to be used. SwitchCore had made a preliminary study of different simulators available, and had come to the conclusion that NS was the best simulator for this master thesis. The main basis of this conclusion was that Sean Murphy had written an additional Differentiated Services library, DS-LIB [35], to NS. This should save time for us since we did not need to implement this too. Quite soon we found out that DS-LIB was poorly structured and implemented, making our job more time consuming since it needed a lot of bug fixing. DS-LIB has also limited our capabilities to add new features to our modification of Murphy's original implementation.

Another reason why everybody thought NS was a good choice was because it is open source. Open Source is good since it is usually platform independent and frequently updated by its users. NS is also used widely for both institutional and commercial purposes.

NS uses two different languages, C++ and TCL. C++ is used to implement almost all algorithms and other various objects that are CPU sensitive. TCL is used to create objects at a higher level than C++ and is the only language used when setting up a network topology. There is a connection available between the two languages. When creating a object for a particular algorithm with a number of parameters in C++, the need to set these parameters through TCL is crucial. The combination of C++ and TCL makes it possible to set the algorithm parameters (in C++) from the simulation script (in TCL). To make this work a binding list between the variables in TCL and C++ exist. The binding list can be seen as a list of associations between TCL and C++ variables. A simulation script example can be found in appendix E.

NS can be placed on layer three, four and five within the OSI model [43]. This enables NS to run simulations on *Local Area Networks* (LANs) as well as simulations using the *Internet Protocol* (IP) and *Transmission Control Protocol* (TCP). There are several different protocols (IP, TCP etc.) and traffic generators (CBR, VBR etc.) already implemented in NS. NS also features several queue management algorithms other than RED, dynamic routing as well as multicast and unicast routing.

Another tool coupled together with NS is *Network Animator* (NAM). NAM gives the possibility of viewing a simulations in a windowed environment. Within this window you can view queue sizes and packet drops. Its also possible to retrieve some small amount of statistics from NAM. A picture of NAM in action can be seen in figure 16.



tcp 65: 4
Sent at 0.068856
1500 bytes

Figure 16: Network Animator (NAM)

It takes a while to get used to NS (as with any other tool), but when you have this general knowledge it is quite easy to implement your own algorithms and simulate them afterwards.

Using NS and the somewhat faulty DS-LIB our master thesis is to study the area of traffic management algorithms in DiffServ networks. This will be done by implementing various queue management and scheduling algorithms. After implementation and verification is done, the focus will change to evaluating these algorithms in different topologies.

3.1 NS-Library

3.1.1 NS-RED

Random Early Detection (RED) described in section 2.4.1 is implemented in *ns-2 2.1b6*. We call the existing implementation of RED in ns NS-RED, as opposed to our own implementation (DS-RED).

We found a small modification in the NS-RED, compared to the original RED algorithm. The modification was made by letting a parameter, *edv_old*, decide whether the first packet drop after crossing the min threshold should be delayed. Whenever the average queue size crossed the min threshold this parameter was checked if it was zero. If so was the case, the algorithm delayed calculating drop probability, when it really should, and then continued as usual and dropped packets next arrival instead. Because of this the following line:

```
..... }
 else if (edv_.old == 0) {
   delay dropping
} else {
  drop with probability p
....
```

was changed to its new appearance, with it disabled.

```
..... }
 else if (edv_.old < 0) {
   delay dropping (never reached)
} else {
  drop with probability p
....
```

We do not expect that this change in the NS-RED code gave us any bugs. It just changed how the algorithm acted when crossing the threshold, and made it compliant with the original RED algorithm.

3.1.2 Contributed RIO (NS-RIO)

There was a need for an implementation of *Random Early Detection IN/OUT* (RIO), the more advanced version of RED with 2 sets of thresholds coupled with a token bucket meter-marker. There were no implementation of RIO in *ns-2 2.1b6*, but we found a contributed implementation on the ns home page [6]. This contributed implementation was for a snapshot of ns-2 2.1b6 and was provided by *James Scott*. He made the conversion from an older version. Since we were using the final ns-2 2.1b6 we decided to make the conversion to the final ns-2 2.1b6. The code was simply integrated into the changed (compared to the snapshot) architecture of ns. When we started to verify RIO against NS-RED we found that it had a couple of serious bugs in it.

A TCL binding to *out_linterm* did not exist, which represents the maximum dropping probability for OUT packets. *Out_linterm* is originally set from the simulation script, but since this TCL binding was missing, *out_linterm* was assigned a random value. Compare it with declaring an integer and not assigning it a value and then using it in some calculation. Some languages protect/warn programmers about using uninitialized variables, but unfortunately C++ does not. This bug made sure that no packets were dropped randomly in the interval [*out_min_thresh..out_max_thresh*] as they should be.

The second major bug we found was that in the function *run_in_estimator*, which updates the average queue size for IN packets, the local variable *total_f* is never assigned a value in the beginning of the function. Although later in the function is it widely used. This, combined with that at the end of the function the average queue size *avgq* is assigned the value of *total_f*, introduced a serious bug in the code. This affected the drop function as well. Since the average queue size *avgq* is used when comparing against thresholds, a packet was randomly dropped due to no initialization of *total_f* at the beginning of the function *run_in_estimator*.

Verification reports can be found in appendix C.2.

3.2 DS-Library

Since our study is about Differentiated Services, support for this was added through a library [35] that was implemented by Sean Murphy. This library initially had support for basic Differentiated Services components, such as PHBs, profiles, conditioners, schedulers and AQM algorithms. Some of these components remained unused while most of the other parts had to be bug fixed and reimplemented in more efficient ways.

The DS-LIB hierarchy can be seen in figures 17, 18, 19, 20 and 21. Outlined boxes represent classes that we have not implemented or modified in anyway. Some of these outlined boxes are part of NS, and others are part of DS-LIB.

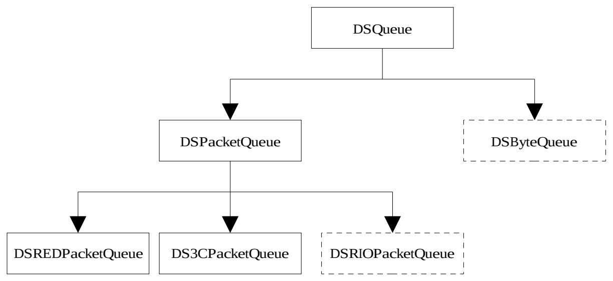


Figure 17: DS-LIB Queue Hierarchy

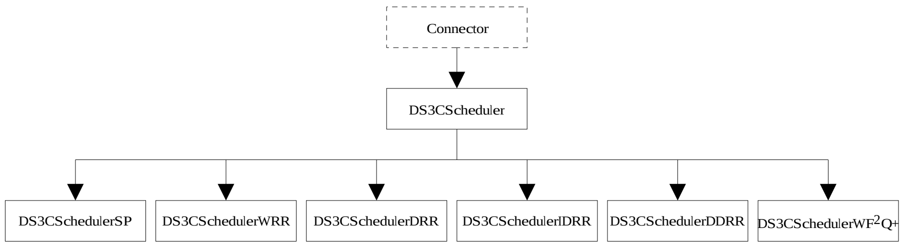


Figure 18: DS-LIB Scheduler Hierarchy

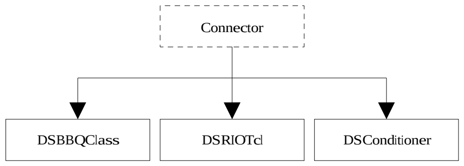


Figure 19: DS-LIB Connector Hierarchy

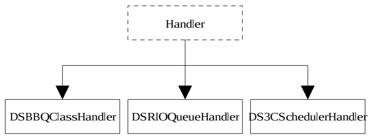


Figure 20: DS-LIB Handler Hierarchy

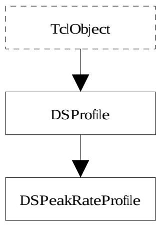


Figure 21: DS-LIB Profile Hierarchy

3.2.1 DS-RED

As mentioned briefly in the NS-RED section above, it was impossible to integrate the already existing implementation of NS-RED into the DS-LIB [35] architecture. Therefore we decided to bug fix Sean Murphy's original implementation of RED and make sure that it would perform in the same way as NS-RED. DS-RED contained several bugs, some which had major implications on the behavior of the algorithm and others that were of minor importance. Some of the changes we had to make was completely due to the fact that we wanted an exact match in performance between NS-RED and DS-RED.

The major structural change we had to make to DS-RED had to do with random number generation. In Murphy's implementation he drew a random number for every packet arrival. In NS-RED a random number was only generated when it was needed. Since without the same random number sequence we would never be able to make an exact match between simulations, we modified DS-RED to only generate a random number when it was needed.

Also, DS-RED originally worked in packet mode, which means that the algorithm's calculations are made on a packet basis, i.e. packet size has no influence. This is highly unrealistic, and therefore support for byte mode was added to the implementation.

Explicit Congestion Notification (ECN) was also added to the implementation, as this already also was a part of NS-RED. Initially, this was something we wanted to evaluate but later lacked time to do.

Another feature that DS-RED was lacking was the variable *count*. The *count* variable helps distribute drops more evenly over an interval as can be seen in pictures in section 2.4.1. This feature was also added to DS-RED.

The major bug we found in DS-RED was that when a packet was dropped and the queue was empty the *idle_* parameter was not set to true as it should have been. The *idle_* and *idle_time_* parameters help the algorithm keep track of how long the queue has been idle, i.e. empty. Based on this time, the scheduler can compute the value of the m variable as can be seen in the pseudo code in section 2.4.1. This bug resulted in long idle times, which in turn resulted in an average queue size that was updated wrong. Verification reports of DS-RED can be found in section C.1.

3.2.2 DS-3CRED

Our implementation of a *Three Colored RED* was implemented into the DS-LIB [35] hierarchy to be used in conjunction with the AF classes instead of Murphy's RIO that did not work properly.

The implementation contains three different color handling modes (see section 2.4.3): SAMT, MAMT-isolated and MAMT-higher-priority-inclusive. They specify how the 3CRED algorithm should share or not share based on packet color the following sharable variables:

- average queue size (based on the EWMA method)
- queue size
- idle
- idle time
- count
- count bytes

There are also a set of variables (e.g. queue weight, service rate and mean packet size) that are global. Moreover, some variables are always color-based, e.g. threshold parameters and statistical variables.

Our implementation of the Three Colored RED incorporates the TRTCM token bucket meter (see section 3.2.3 for implementation details).

3.2.3 Two Rate Three Color Marker (TRTCM)

This marker is an implementation of Juha Heinanen's specification of a *Two Rate Three Color Marker* [24]. We decided to put this directly in our Three Colored RED implementation. Its job is to function as a rate limiter, and whenever the rate is exceeded it remarks packets to a lower precedence (if possible).

This module is also one of the few that have not been verified against a reference model. This is of course due to the fact that no reference model could be found, and that the code needed for its implementation were only a couple of lines. This code was inspected by eye during runtime and was found to have no bugs.

3.2.4 DS-Schedulerlink

This class is implemented entirely in TCL, as its superclass is called *Link*. Its job is to function as a connection between nodes and specify which routes packets may take on its way to the destination. The difference between the superclass *Link* and its child *DS-Schedulerlink* is that *DS-Schedulerlink* has an associated scheduler which all packets must pass through. The scheduler could be implemented with for example any of the scheduling algorithms mentioned in section 2.5. In fact all schedulers mentioned in that section is implemented and presented further down. Since Murphy's original implementation of DS-LIB [35] only supported the use of an *Weighted Round Robin* (WRR) scheduler some changes were necessary to the TCL class in order to make it possible to select some scheduling algorithm other than WRR. Therefore a new parameter was added to the TCL class, that specify which scheduling algorithm to use.

3.2.5 DS-Scheduler

This class is a pure base class implemented in C++ for use by the specific scheduling algorithm implementations. This base class contains declaration of queues and weights on these queues, as well as the interface concerning statistics to the outside world.

A general layout of a scheduler can be seen in figure 22.

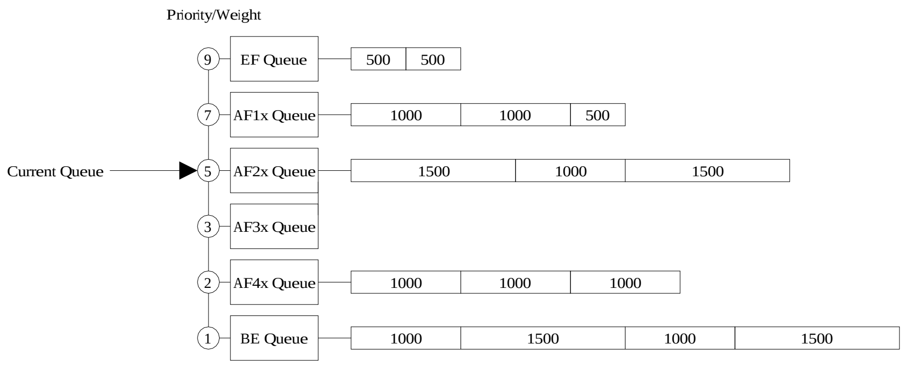


Figure 22: General Scheduler Layout

Murphy's implementation of this base class contained the following declarations:

Queue	Algorithm
BE	RED
All AF classes	RIO
EF	DropTail

Table 3: Murphy's Queue Declarations

As you can see he used one RIO queue for all AF classes. This is a pure waste of AF functionality and the proper decision would be to assign each AF class its own queue. In this context it might seem improper to call RIO a queue, since RIO actually is an AQM algorithm. However both in Murphy's and our implementation the term queue is used to refer to an implementation of the AQM algorithm and the actual queue together.

We declared the BE Queue as using RED and the EF Queue as using 3CRED. However this last declaration of the EF queue as 3CRED is only for simplicity. This queue should be declared as DropTail as Murphy did, but DropTail can be emulated by setting all 3CRED parameters to values so that the 3CRED functionality is disabled. Our declaration of the various queues instead of Murphy's is:

Queue	Algorithm
BE	RED
AF1x	3CRED
AF2x	3CRED
AF3x	3CRED
AF4x	3CRED
EF	3CRED

Table 4: Our Queue Declarations

In a perfect world it would be much better if one could dynamically specify what kind of queue one wants to use, but once again we are limited by Murphy's design.

A lot of statistics functionality was added to the queues managed by the scheduler, so we reimplemented the entire interface to the TCL scripts. Almost nothing was left of the original implementation.

3.2.6 DS-Scheduler-SP

We needed a *Strict Priority* (SP) Scheduler to compare against WRR and WF^2Q+. Implementation of SP is very straightforward, and no problems were encountered.

3.2.7 DS-Scheduler-WRR

Murphy's original implementation of the *Weighted Round Robin* (WRR) scheduler contained a couple of serious bugs that affected the performance of our simulations. Since the DS-3CRED and DS-RED queues both maintain the variable *idle_*, its the schedulers job to make sure that this variable is used. This variable is set to true in the DS-RED and DS-3CRED queues whenever the dequeue operation encounters an empty queue. But since Murphy in the scheduler only performed dequeue when the queue was non-empty, this failed dequeue operation was never executed. This bug was corrected.

This very basic scheduler was not used in any real simulations due to the similar, but more interesting DRR and IDRR implementations.

3.2.8 DS-Scheduler-DRR

Deficit Round Robin was described in section 2.5.3. This implementation of DRR is mostly a port from the already existing NS-DRR. We had to make some small changes to make it better fit our needs. One of the most important changes we did was to make our implementation make use of the weights. In the NS-DRR implementation all queues strangely received the same quantum. Our implementation used the weights on the queues to calculate the real queue quantum for each queue. This implementation is non-interleaved (see section 2.5.3 for explanation of interleaving).

The DRR scheduler also has a mode that does not remember unused quantums. This mode is similar to WRR but not the same since it operates on a byte-level.

3.2.9 DS-Scheduler-IDRR

Interleaved Deficit Round Robin is another mode of DRR and was also described in section 2.5.3. The only difference between the IDRR and the DRR implementations is that IDRR is interleaved.

3.2.10 DS-Scheduler-DDRR

Deficit Driven Round Robin is the last variant of DRR that was mentioned in section 2.5.3. DDRR keeps a list sorted by deficit bytes, and uses it to select which queue gets to send the next packet.

3.2.11 DS-Scheduler-WF^{2}Q+

We needed an more advanced scheduler than the WRR Scheduler for our simulations. On the Internet *Shahzad Ali* [2] had published an implementation of *Worst-case Weighted Fair Queueing* for ns-2 2.1b5. We refitted this code into the DS-Scheduler structure. This scheduler was verified in section C.4.

3.3 Web Traffic Generator

We needed a way to introduce random bursts in our simulations but the in NS existing http agent was to complicated. Our first try was the contributed *Web Traffic Generator* [25], from the NS home page [6], that seemed to suit our needs perfectly. It supports creation of servers and clients and generation of traffic between them. The clients wait a certain amount of time before requesting another WWW page. The servers replies with a certain sized packet to the client. The amount of time or size is taken from a realistic distribution according to the authors and is based on real traffic. The servers and clients both use TCP.

However we found out pretty quickly that this code contained bugs beyond our understanding, so then we turned to a C++ implementation updated by Rogerio Andrade [3] found by searching the NS mailing list. This code was also for an earlier version of NS, namely *ns-2 2.1b5*. We tried to refit this code into *ns-2 2.1b6*, but due to lack of time needed for understanding the TCP implementations in its entirety we failed to make an use of it.

Our last attempt to make a WWW agent that would generate *Pareto Distribution* sized files was a simple TCL implementation. It is believed that the size of files in the actual Internet may be adequately modeled by the Pareto distribution (see appendix D). Since pareto distribution can yield unlimited values a cut-off of the file size is made at one megabyte. The previously mentioned TCL and C++ implementations used Pareto for setting the sizes of the files (actually packets). When calculating the time an WWW agent should sleep we based this on the ad-hoc idea, that the longer file a client requests, the longer time the client needs to consume the information contained. The idle time is calculated according to the formula

$$max(uniform(1, (filesize/200)), thinktime_{max})$$

This gave us a distribution we felt were realistic and it fulfilled our needs. However, we do not claim that this is true WWW traffic, but on the other hand such traffic could only be received from real life traffic traces and that is beyond the scope of this master thesis.

3.4 Shared Memory Issues

In NS [6] all queues in a node allocate buffer independently of all the others. We call this mode *Divided Memory Queue* (DMQ). This mode lacks the realistic demand of an overall buffer limit.

We feel a more realistic mode would be *Shared Memory* (SM) or sometimes also referred to as *Complete Sharing* (CS). In this mode a node has a fixed buffer size allocated for all queues in the node. Each queue also has its own limit on how much of the shared buffer it may use.

When a packet arrives we need to check two things. Firstly, if there is enough overall buffer for this packet and then if this queue is allowed to use the available buffer.

Another reasonable option is to let queues be organized according to incoming and outgoing link modules. Then all queues belonging to a specific link module would share an overall buffer limit. The queues would of course also have their own limit on how much of the overall buffer limit it is allowed to use. We call this mode *Divided Memory Link* (DML).

These 3 options have all been implemented into the DS-LIB (see section 3.2) architecture.

4 Simulations

4.1 Simulation Environment

This section contains information about the simulation environment that we have used in all of our simulations documented in this report. For full information about our own contributions see section 3.

Hardware

All simulations were run on a computer with the following hardware:

- Intel Celeron 466 Mhz
- 128 Mb SDRAM

Operating system

The operating that we have ran the simulator on is Linux (RedHat 6.1, Kernel version 2.2.12-20).

Software

The simulator we have used is *ns-2 2.1b6* [6] released 18-Jan-2000, with additions.

4.2 General Information

This section contains general, relevant, information about our simulations such as definition of most of the expressions used in sections 4.5 through 4.9 when presenting background and results, and making conclusions.

Simulation Topology Scenario	a number of simulations with a common topology and parameters *except* those parameters that we wish to study (e.g. Sim 2x).
Simulation Traffic Scenario	a *Simulation Topology Scenario* instance with *all* parameters fixed *except* the single parameter that we wish to study (e.g. Sim 21).
Simulation Set	a *Simulation Traffic Scenario* instance with *all* parameters fixed *including* the one studied, i.e. it is a number of stand-alone simulations (e.g. Sim 21 with mean packet size set to 50).

Total Bytes Enqueued	is the total number of bytes arriving at the destinations.
Delay	is the time spent in queues along the network path.
Drop Ratio	is the percentage of bytes being dropped, i.e. $Drop\,Ratio = \frac{Total\,Bytes\,Dropped}{Total\,Bytes\,Arrived}$. There is no obvious way to calculate the drop ratio along the network path, therefore we have studied the drop ratio at the first gateway along the path, i.e. Gateway A, see figure 24.
Difference	is when used with the three measures above the largest difference, i.e. between the min value and the max value.
Difference Percentage	$Total\,Bytes\,Enqueued = \frac{Total\,Bytes\,Enqueued_{Difference}}{Total\,Bytes\,Enqueued_{Min}}$ $Delay\,Percentage = \frac{Delay_{Difference}}{Delay_{Min}}$ $Drop\,Ratio = \frac{Drop\,Ratio_{Difference}}{Drop\,Ratio_{Min}}$

Difference between PHB classes and a PHB group

Throughout the rest of this master thesis we have used a convention of our own when we are discussing a several PHB classes at one time. This convention allows us to distinguish between individual colors for a PHB group or *the total* of all colors for a PHB group.

Examples:

PHB class AF11	the AF11 PHB class - nothing else
PHB group AF1	*the total* of PHB classes AF11, AF12 and AF13 on a non-color basis
PHB classes AF1x	all of PHB group AF1, i.e. PHB classes AF11, AF12 and AF13 on a color basis

Note: There are two groups that contain only one PHB class per group, namely EF PHB class and BE PHB class. This even if there exist other PHB classes such as EFD and LBE that could be included as well.

Importance of the Different Simulation Topology Scenarios

Sim 0x was our first simulation topology scenario and is of low interest.

For Sim 1x, after evaluating Sim 0x, we decided to increase the short lived traffic, WWW, and decrease the long lived traffic, FTP. However, in Sim 1x it turned out that our shared buffer restrictions where to low giving it a too big impact on performance. Also, low AQM thresholds gave the AQM algorithms not enough time to react.

So, in Sim2x we increased the shared buffer restrictions to 256k and multiplied all AQM thresholds by three. It is our main simulation topology scenario (the one that should be focused) and of highest interest.

Sim 3x is thought as "a perfect world (no buffer limitations)" reference simulation topology scenario for AQM algorithms. It is the same as Sim 2x, except that it has no shared buffer or queue restrictions. This gave us the chance to study and compare impact of real world buffer restrictions including shared ones.

In Sim 4x, also the same as Sim 2x with some exceptions, is thought as a present-day traffic reference simulation topology scenario. To make it operate as the current Internet, the token bucket was disabled, i.e. there was no color differentiation as all packets are initially sent as green. Also, all scheduler weights were set equal.

As both Sim 8x and Sim 9x are stand-alone simulation topology scenarios they are of high interest.

4.3 Formulating Realistic and Comprehensive Network Studies

During the work with this master thesis we become aware of that it is difficult to create realistic network simulations. Especially if you want them to be comprehensive as well – which one usually wants. We have not studied this area at all – this section is just a *reflection* of our discussions while setting up various simulations with NS.

We have been able to divide this problem into three sub categories:

- Creation of topologies (i.e. nodes and and their direct parameters, e.g. bandwidth and delay)
- Setting algorithm parameters (i.e. for active queue management algorithms and token buckets)
- Collection and analysis of statistics

4.3.1 Creating topologies

There are a number of questions when creating a realistic and comprehensive network topology:

- What size of network — e.g. home-based, medium-sized company, university, corporate company and backbone — are we trying to simulate?
- What are the characteristics in terms of size and structure of such a network?
- What type of network media — e.g. Ethernet, Wireless, Token Ring or Token Bus— are we trying to simulate?

- What type of traffic (FTP, WWW etc) and PHB should be generated to resemble load, burstiness etc?

One should keep in mind that in real life topologies are usually set up without our influence. One's goal should simply be to create parts of a realistic topology so that an algorithm's behavior is easily observable. We agree with K. Nichols[1] that today's problem is that the easily observable topologies, such as the classical bottleneck (see figure 23), tends to be non-realistic.

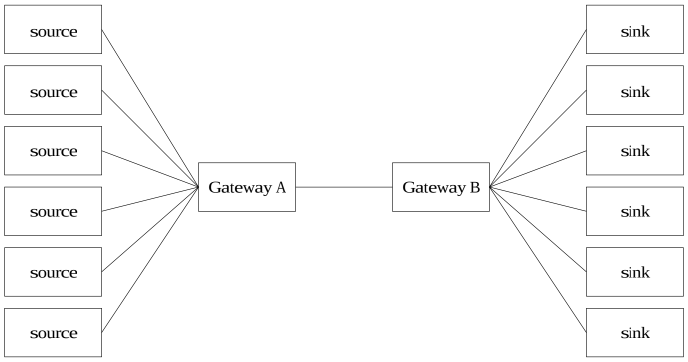


Figure 23: Typical Bottleneck Topology

We have tried to overcome this by using the old-fashion well known bottleneck topology with an extra node in between, to which we added noise (see figure 24). Even though this setup is far from realistic we think, hope, and feel that this is a better representation of a realistic topology than the commonly used bottleneck.

4.3.2 Setting parameters

The second category, which is in the focus in this master thesis – parameter settings of various algorithms such as:

- Meter-marker (token bucket) parameters (i.e. allowance of burstiness, different traffic loads etc)
- Active Queue Management (AQM) parameters (i.e. thresholds, drop probability etc)
- Scheduler parameters (i.e. weight values etc)

There is an interaction between schedulers, meter-markers and AQM algorithms. Assuming a somewhat fair scheduling, i.e. no starvation, we focus our attention on the meter-marker and the downstream AQM algorithm. The meter-marker packet recoloring changes the conditions for the downstream AQM algorithms.

There are three different set of simulation scenarios concerning meter-markers and AQM algorithms. Either you observe their behavior independently (e.g. disabling the meter-marker, i.e. the recoloring, capability, thereby making it easier to observe the AQM algorithm alone) or one wants to observe how they interact together, which we have found is much more difficult. Actually, we decided to keep the meter-marker settings fixed for most of our simulation traffic scenarios. This because the meter-marker setting added another dimension parameter wise and it would have taken to much time to reach any substantial conclusions even with our massive collection of statistics.

[1]stated by K. Nichols in a DiffServ mailing list message available (September 5, 2000) on url `http://www-nrg.ee.lbl.gov/diff-serv-arch/msg03051.html`

4.3.3 Collection and analysis of statistics

When collecting statistical information we have found it very important to keep the number of statistical variables as low as possible. There are several reasons why one should limit the amount of data. A scarce amount of data makes it easier to see patterns, compare and generalize etc. By scarce, we mean both how many statistical variables that are being generated and/or used, as well as the amount of data that is generated for each statistical variable based on the sampling interval.

Simulations in our simulation environment include the possibility to generate a stats file (see appendix F) per scheduler and several xtr files (trace files viewable with xgraph, a Unix graph tool) per scheduler.

The stats file is very handy when getting a quick view of a simulation or simulation set as it contains data (total and for different colors) in both bytes and packets about queue arrivals, departures, drops etc; as well as average queue delay.

The xtr files on the other hand trace a variable, e.g. average queue size or delay, with a chosen time interval. This allows us to visually observe the behavior of an AQM algorithm. Having statistics in both bytes and packets helps understanding, in particular, the influence of WWW traffic as its packet sizes varies. The major problem when generating xtr files is the physical memory that they occupy and the time consumption. Initially, we used a lower sampling interval than the 10 ms we finally decided to use. The accuracy of the trace variable was not satisfying with the lower sampling interval as it missed out on sudden bursts. This caused confusion when the average queue size suddenly increased without the queue size seeming to increase. However, a sampling interval of 10 ms resulted in a file size of 200-300k for each xtr file. We also needed more information for algorithms evaluation causing the number of xtr files to rise to about 29 per scheduler, increasing the real simulation time with a factor of 25 from one minute to approximately twenty-five minutes.

A simulation traffic scenario runs about 8-36 simulation sets, with each set being is at least ten simulations, causing the whole lot to take up about 3.5GB depending on the number of schedulers in the simulation topology. This was of course a prohibitive amount of data, thus the xtr files were only generated when needed.

Analyzing average statistics with a big difference between min and max values increase the risk of making false conclusion because of misleading information. We have no experiences within the area of statistics, but as we have been working on master thesis we have found out that knowing the confidence interval gives a certain depth to ones conclusions.

4.4 Simulation Setups

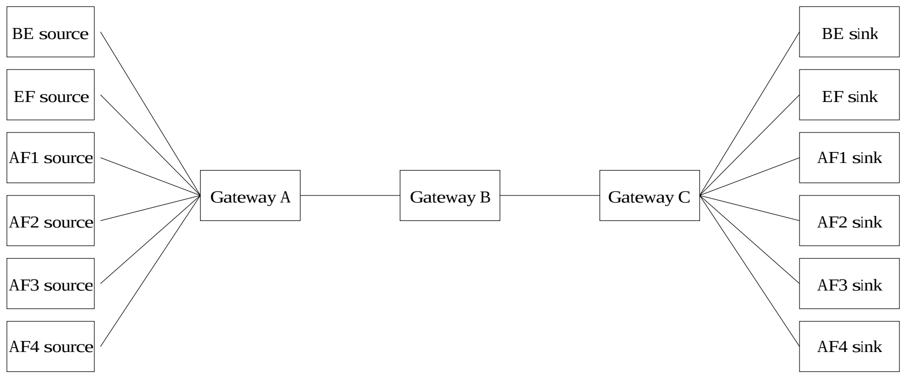


Figure 24: Our Final Simulations Topology

Common parameters	
Start Time	0 s
Stop Time	100 s
Warmup	1 s
Simulation time	99 s

Links between Nodes	Bandwidth	Delay
EF Source ↔ Gateway A	10 Mb	1 ms
AF1 Source ↔ Gateway A	10 Mb	1 ms
AF2 Source ↔ Gateway A	10 Mb	1 ms
AF3 Source ↔ Gateway A	10 Mb	1 ms
AF4 Source ↔ Gateway A	10 Mb	1 ms
BE Source ↔ Gateway A	10 Mb	1 ms
Gateway A ↔ Gateway B	10 Mb	2 ms
Gateway B ↔ Gateway C	10 Mb	2 ms
EF Source ↔ Gateway C	10 Mb	1 ms
AF1 Source ↔ Gateway C	10 Mb	1 ms
AF2 Source ↔ Gateway C	10 Mb	1 ms
AF3 Source ↔ Gateway C	10 Mb	1 ms
AF4 Source ↔ Gateway C	10 Mb	1 ms
BE Source ↔ Gateway C	10 Mb	1 ms

Scheduler Parameters	Byte-Mode	ECN-mode
EF Source	yes	no
AF1 Source	yes	no
AF2 Source	yes	no
AF3 Source	yes	no
AF4 Source	yes	no
BE Source	yes	no

Table 5: Simulation Independent Topology Parameters

All final simulations use the topology found in figure 24.

Table 5 shows parameters that are fixed, i.e. simulation independent, throughout all final simulations made in this master thesis.

We have used seven different parameter and setting setups. Five of them are interconnected. The remaining two have only been used on their own, and not compared against any of the others setups.

4.4.1 Simulation Topology Scenario 0x

The first simulation topology scenario we tested. This simulation topology scenario basically gave us a first insight in what values would be appropriate for several of the parameters. The settings are specified in tables 6, 7 and 8.

Common parameters	
Scheduler used	WF^{2}Q+

FTP Sources	TCP window size/CBR rate	Packet size
EF Source	CBR rate 40 kb/s	500 bytes
AF1 Source	TCP window 64 packets	1500 bytes
AF2 Source	TCP window 64 packets	1500 bytes
AF3 Source	TCP window 64 packets	1500 bytes
AF4 Source	TCP window 64 packets	1500 bytes
BE Source	TCP window 64 packets	1500 bytes

WWW Sources	Number of clients	Max packet size
AF1 Source	20	1500 bytes
AF2 Source	30	1500 bytes
AF3 Source	40	1500 bytes
AF4 Source	50	1500 bytes
BE Source	60	1500 bytes

Noise Source	Noise Destination	CBR Rate	Packet size	PHB Class
Gateway A	Gateway B	varies	1500 bytes	varies
Gateway B	Gateway C	varies	1500 bytes	varies

Table 6: Simulation Dependent Topology Parameters – 0x

Four schedulers are used, one in each direction between gateway A and gateway B followed by another two in each direction between gateway B and gateway C. Each of these schedulers have the same buffer limits and settings.

Scheduler Parameters	
Shared buffer mode	Shared Memory Node
Shared buffer mode size	128 000 bytes
Shared buffer mode ignore	EF Queue
Mean Packet Size (all queues)	varies
Color Handling Mode (all queues)	Higher-Priority-Inclusive

Scheduler Queue	Weight	Limit (bytes)
EF Queue	20	5000
AF1 Queue	10	10000
AF2 Queue	6	20000
AF3 Queue	4	30000
AF4 Queue	2	40000
BE Queue	1	50000

Token Bucket	PBS (bytes)	PIR (bytes)	CBS (bytes)	CIR (bytes)
EF Queue	-	-	-	-
AF1 Queue	200000	150000	150000	100000
AF2 Queue	250000	200000	200000	150000
AF3 Queue	300000	200000	200000	150000
AF4 Queue	300000	200000	200000	150000
BE Queue	-	-	-	-

Table 7: Simulation Dependent Scheduler Parameters – 0x

The default active queue management parameters for simulation topology scenario 0x can be viewed in table 8.

AQM Parameters ALL			
Max drop probability	0.002		

AQM Parameters EF	Min threshold	Max threshold	Linterm
Green	-	-	-
Yellow	-	-	-
Red	-	-	-

AQM Parameters AF1	Min threshold	Max threshold	Linterm
Green	2000 bytes	5000 bytes	50
Yellow	4000 bytes	7000 bytes	40
Red	6000 bytes	9000 bytes	30

AQM Parameters AF2	Min threshold	Max threshold	Linterm
Green	4000 bytes	10000 bytes	50
Yellow	8000 bytes	14000 bytes	40
Red	12000 bytes	18000 bytes	30

AQM Parameters AF3	Min threshold	Max threshold	Linterm
Green	6000 bytes	15000 bytes	50
Yellow	12000 bytes	21000 bytes	40
Red	18000 bytes	27000 bytes	30

AQM Parameters AF4	Min threshold	Max threshold	Linterm
Green	8000 bytes	20000 bytes	50
Yellow	16000 bytes	28000 bytes	40
Red	24000 bytes	36000 bytes	30

AQM Parameters BE	Min threshold	Max threshold	Linterm
	20000 bytes	40000 bytes	50

Table 8: Active Queue Management Simulation Parameters – 0x

4.4.2 Simulation Topology Scenario 1x

Simulation topology scenario 0x gave us some new insight about parameter settings. We noticed that in simulation topology scenario 0x a lot of traffic consisted of much long-lived traffic (FTP sources) and not so much short-lived traffic (WWW sources). We wanted more burstiness to see how the AQM algorithms adapted to changes in traffic. Therefore in simulation topology scenario 1x the window sizes for the FTP sources we decreased substantially. Also some more WWW traffic sources was added. The settings are specified in tables 9, 10 and 11.

Common parameters	
Scheduler used	WF^{2}Q+

FTP Sources	TCP window size/CBR rate	Packet size
EF Source	CBR rate 40 kb/s	500 bytes
AF1 Source	TCP window 4 packets	1500 bytes
AF2 Source	TCP window 8 packets	1500 bytes
AF3 Source	TCP window 16 packets	1500 bytes
AF4 Source	TCP window 16 packets	1500 bytes
BE Source	TCP window 32 packets	1500 bytes

WWW Sources	Number of clients	Max packet size
AF1 Source	30	1500 bytes
AF2 Source	40	1500 bytes
AF3 Source	50	1500 bytes
AF4 Source	60	1500 bytes
BE Source	70	1500 bytes

Noise Source	Noise Destination	CBR Rate	Packet size	PHB Class
Gateway A	Gateway B	varies	1500 bytes	varies
Gateway B	Gateway C	varies	1500 bytes	varies

Table 9: Simulation Dependent Topology Parameters – 1x

Four schedulers are used, one in each direction between gateway A and gateway B followed by another two in each direction between gateway B and gateway C. Each of these schedulers have the same buffer limits and settings.

Scheduler Parameters	
Shared buffer mode	Shared Memory Node
Shared buffer mode size	128 000 bytes
Shared buffer mode ignore	EF Queue
Mean Packet Size (all queues)	varies
Color Handling Mode (all queues)	Higher-Priority-Inclusive

Scheduler Queue	Weight	Limit (bytes)
EF Queue	20	5000
AF1 Queue	10	10000
AF2 Queue	6	20000
AF3 Queue	4	30000
AF4 Queue	2	40000
BE Queue	1	50000

Token Bucket	PBS (bytes)	PIR (bytes)	CBS (bytes)	CIR (bytes)
EF Queue	-	-	-	-
AF1 Queue	200000	150000	150000	100000
AF2 Queue	250000	200000	200000	150000
AF3 Queue	300000	200000	200000	150000
AF4 Queue	300000	200000	200000	150000
BE Queue	-	-	-	-

Table 10: Simulation Dependent Scheduler Parameters – 1x

The default active queue management parameters for simulation topology scenario 1x can be viewed in table 11.

AQM Parameters ALL			
Max drop probability	0.002		

AQM Parameters EF	Min threshold	Max threshold	Linterm
Green	-	-	-
Yellow	-	-	-
Red	-	-	-

AQM Parameters AF1	Min threshold	Max threshold	Linterm
Green	2000 bytes	5000 bytes	50
Yellow	4000 bytes	7000 bytes	40
Red	6000 bytes	9000 bytes	30

AQM Parameters AF2	Min threshold	Max threshold	Linterm
Green	4000 bytes	10000 bytes	50
Yellow	8000 bytes	14000 bytes	40
Red	12000 bytes	18000 bytes	30

AQM Parameters AF3	Min threshold	Max threshold	Linterm
Green	6000 bytes	15000 bytes	50
Yellow	12000 bytes	21000 bytes	40
Red	18000 bytes	27000 bytes	30

AQM Parameters AF4	Min threshold	Max threshold	Linterm
Green	8000 bytes	20000 bytes	50
Yellow	16000 bytes	28000 bytes	40
Red	24000 bytes	36000 bytes	30

AQM Parameters BE	Min threshold	Max threshold	Linterm
	20000 bytes	40000 bytes	50

Table 11: Active Queue Management Simulation Parameters – 1x

4.4.3 Simulation Topology Scenario 2x

As mentioned in section 4.4.2 the simulation topology scenario 0x did not give enough room for the AQM algorithms. The solution used in simulation topology scenario 1x was to decrease the traffic. This was but one of the available options. Another solution providing the AQM algorithms more room is actually to increase the buffer. Therefore, simulation topology scenario 2x keeps the same traffic load as simulation topology scenario 0x, but doubles the buffer space and AQM thresholds. The settings are specified in tables 12, 13 and 14.

Common parameters	
Scheduler used	WF^{2}Q+

FTP Sources	TCP window size/CBR rate	Packet size
EF Source	CBR rate 40 kb/s	500 bytes
AF1 Source	TCP window 64 packets	1500 bytes
AF2 Source	TCP window 64 packets	1500 bytes
AF3 Source	TCP window 64 packets	1500 bytes
AF4 Source	TCP window 64 packets	1500 bytes
BE Source	TCP window 64 packets	1500 bytes

WWW Sources	Number of clients	Max packet size
AF1 Source	20	1500 bytes
AF2 Source	30	1500 bytes
AF3 Source	40	1500 bytes
AF4 Source	50	1500 bytes
BE Source	60	1500 bytes

Noise Source	Noise Destination	CBR Rate	Packet size	PHB Class
Gateway A	Gateway B	varies	1500 bytes	varies
Gateway B	Gateway C	varies	1500 bytes	varies

Table 12: Simulation Dependent Topology Parameters – 2x

Four schedulers are used, one in each direction between gateway A and gateway B followed by another two in each direction between gateway B and gateway C. Each of these schedulers have the same buffer limits and settings.

Scheduler Parameters	
Shared buffer mode	Shared Memory Node
Shared buffer mode size	256 000 bytes
Shared buffer mode ignore	EF Queue
Mean Packet Size (all queues)	varies
Color Handling Mode (all queues)	Higher-Priority-Inclusive

Scheduler Queue	Weight	Limit (bytes)
EF Queue	20	5000
AF1 Queue	10	20000
AF2 Queue	6	40000
AF3 Queue	4	60000
AF4 Queue	2	80000
BE Queue	1	100000

Token Bucket	PBS (bytes)	PIR (bytes)	CBS (bytes)	CIR (bytes)
EF Queue	-	-	-	-
AF1 Queue	200000	150000	150000	100000
AF2 Queue	250000	200000	200000	150000
AF3 Queue	300000	200000	200000	150000
AF4 Queue	300000	200000	200000	150000
BE Queue	-	-	-	-

Table 13: Simulation Dependent Scheduler Parameters – 2x

The default active queue management parameters for simulation topology scenario 2x can be viewed in table 14.

AQM Parameters ALL			
Max drop probability	0.002		

AQM Parameters EF	Min threshold	Max threshold	Linterm
Green	-	-	-
Yellow	-	-	-
Red	-	-	-

AQM Parameters AF1	Min threshold	Max threshold	Linterm
Green	4000 bytes	10000 bytes	50
Yellow	8000 bytes	14000 bytes	40
Red	12000 bytes	18000 bytes	30

AQM Parameters AF2	Min threshold	Max threshold	Linterm
Green	8000 bytes	20000 bytes	50
Yellow	16000 bytes	28000 bytes	40
Red	24000 bytes	36000 bytes	30

AQM Parameters AF3	Min threshold	Max threshold	Linterm
Green	12000 bytes	30000 bytes	50
Yellow	24000 bytes	42000 bytes	40
Red	36000 bytes	54000 bytes	30

AQM Parameters AF4	Min threshold	Max threshold	Linterm
Green	16000 bytes	40000 bytes	50
Yellow	32000 bytes	56000 bytes	40
Red	48000 bytes	72000 bytes	30

AQM Parameters BE	Min threshold	Max threshold	Linterm
	40000 bytes	80000 bytes	50

Table 14: Active Queue Management Simulation Parameters – 2x

4.4.4 Simulation Topology Scenario 3x

Simulation topology scenario 3x is somewhat unrealistic, since it uses unlimited buffer spaces. But the aim of this simulation topology scenario is to really study if active queue management algorithms can control the queue size in a satisfying manner. The only difference between simulation topology scenarios 3x and 2x is that no shared buffer management scheme is used, and that the queues have no buffer limits. The settings are specified in tables 15, 16 and 17.

Common parameters	
Scheduler used	WF^{2}Q+

FTP Sources	TCP window size/CBR rate	Packet size
EF Source	CBR rate 40 kb/s	500 bytes
AF1 Source	TCP window 64 packets	1500 bytes
AF2 Source	TCP window 64 packets	1500 bytes
AF3 Source	TCP window 64 packets	1500 bytes
AF4 Source	TCP window 64 packets	1500 bytes
BE Source	TCP window 64 packets	1500 bytes

WWW Sources	Number of clients	Max packet size
AF1 Source	20	1500 bytes
AF2 Source	30	1500 bytes
AF3 Source	40	1500 bytes
AF4 Source	50	1500 bytes
BE Source	60	1500 bytes

Noise Source	Noise Destination	CBR Rate	Packet size	PHB Class
Gateway A	Gateway B	varies	1500 bytes	varies
Gateway B	Gateway C	varies	1500 bytes	varies

Table 15: Simulation Dependent Topology Parameters – 3x

Four schedulers are used, one in each direction between gateway A and gateway B followed by another two in each direction between gateway B and gateway C. Each of these schedulers have the same buffer limits and settings.

Scheduler Parameters	
Shared buffer mode	Divided Memory Queue
Shared buffer mode size	- bytes
Mean Packet Size (all queues)	varies
Color Handling Mode (all queues)	Higher-Priority-Inclusive

Scheduler Queue	Weight	Limit (bytes)
EF Queue	20	no limit
AF1 Queue	10	no limit
AF2 Queue	6	no limit
AF3 Queue	4	no limit
AF4 Queue	2	no limit
BE Queue	1	no limit

Token Bucket	PBS (bytes)	PIR (bytes)	CBS (bytes)	CIR (bytes)
EF Queue	-	-	-	-
AF1 Queue	200000	150000	150000	100000
AF2 Queue	250000	200000	200000	150000
AF3 Queue	300000	200000	200000	150000
AF4 Queue	300000	200000	200000	150000
BE Queue	-	-	-	-

Table 16: Simulation Dependent Scheduler Parameters – 3x

The default active queue management parameters for simulation topology scenario 3x can be viewed in table 17.

AQM Parameters ALL			
Max drop probability	0.002		

AQM Parameters EF	Min threshold	Max threshold	Linterm
Green	-	-	-
Yellow	-	-	-
Red	-	-	-

AQM Parameters AF1	Min threshold	Max threshold	Linterm
Green	4000 bytes	10000 bytes	50
Yellow	8000 bytes	14000 bytes	40
Red	12000 bytes	18000 bytes	30

AQM Parameters AF2	Min threshold	Max threshold	Linterm
Green	8000 bytes	20000 bytes	50
Yellow	16000 bytes	28000 bytes	40
Red	24000 bytes	36000 bytes	30

AQM Parameters AF3	Min threshold	Max threshold	Linterm
Green	12000 bytes	30000 bytes	50
Yellow	24000 bytes	42000 bytes	40
Red	36000 bytes	54000 bytes	30

AQM Parameters AF4	Min threshold	Max threshold	Linterm
Green	16000 bytes	40000 bytes	50
Yellow	32000 bytes	56000 bytes	40
Red	48000 bytes	72000 bytes	30

AQM Parameters BE	Min threshold	Max threshold	Linterm
	40000 bytes	80000 bytes	50

Table 17: Active Queue Management Simulation Parameters – 3x

4.4.5 Simulation Topology Scenario 4x

In simulation topology scenario 4x all queues have the same weight. Simulation topology scenario 4 is also color-blind, i.e. only one conformance level exist. This simulation topology scenario looks a bit like todays network, with same queue weights and no advanced active queue management algorithms such as 3CRED (see section 2.4.3. The settings are specified in tables 18, 19 and 20.

Common parameters	
Scheduler used	WF^2Q+

FTP Sources	TCP window size/CBR rate	Packet size
EF Source	CBR rate 40 kb/s	500 bytes
AF1 Source	TCP window 64 packets	1500 bytes
AF2 Source	TCP window 64 packets	1500 bytes
AF3 Source	TCP window 64 packets	1500 bytes
AF4 Source	TCP window 64 packets	1500 bytes
BE Source	TCP window 64 packets	1500 bytes

WWW Sources	Number of clients	Max packet size
AF1 Source	20	1500 bytes
AF2 Source	30	1500 bytes
AF3 Source	40	1500 bytes
AF4 Source	50	1500 bytes
BE Source	60	1500 bytes

Noise Source	Noise Destination	CBR Rate	Packet size	PHB Class
Gateway A	Gateway B	varies	1500 bytes	varies
Gateway B	Gateway C	varies	1500 bytes	varies

Table 18: Simulation Dependent Topology Parameters – 4x

Four schedulers are used, one in each direction between gateway A and gateway B followed by another two in each direction between gateway B and gateway C. Each of these schedulers have the same buffer limits and settings.

Scheduler Parameters	
Shared buffer mode	Shared Memory Node
Shared buffer mode size	256 000 bytes
Shared buffer mode ignore	EF Queue
Mean Packet Size (all queues)	varies
Color Handling Mode (all queues)	Higher-Priority-Inclusive

Scheduler Queue	Weight	Limit (bytes)
EF Queue	1	5000
AF1 Queue	1	20000
AF2 Queue	1	40000
AF3 Queue	1	60000
AF4 Queue	1	80000
BE Queue	1	100000

Token Bucket	PBS (bytes)	PIR (bytes)	CBS (bytes)	CIR (bytes)
EF Queue	-	-	-	-
AF1 Queue	-	-	-	-
AF2 Queue	-	-	-	-
AF3 Queue	-	-	-	-
AF4 Queue	-	-	-	-
BE Queue	-	-	-	-

Table 19: Simulation Dependent Scheduler Parameters – 4x

The default active queue management parameters for simulation topology scenario 4x can be viewed in table 20.

AQM Parameters ALL			
Max drop probability	0.002		

AQM Parameters EF	Min threshold	Max threshold	Linterm
Green	-	-	-
Yellow	-	-	-
Red	-	-	-

AQM Parameters AF1	Min threshold	Max threshold	Linterm
Green	8000 bytes	16000 bytes	50
Yellow	-	-	-
Red	-	-	-

AQM Parameters AF2	Min threshold	Max threshold	Linterm
Green	16000 bytes	32000 bytes	50
Yellow	-	-	-
Red	-	-	-

AQM Parameters AF3	Min threshold	Max threshold	Linterm
Green	24000 bytes	48000 bytes	50
Yellow	-	-	-
Red	-	-	-

AQM Parameters AF4	Min threshold	Max threshold	Linterm
Green	32000 bytes	64000 bytes	50
Yellow	-	-	-
Red	-	-	-

AQM Parameters BE	Min threshold	Max threshold	Linterm
	40000 bytes	80000 bytes	50

Table 20: Active Queue Management Simulation Parameters – 4x

4.4.6 Simulation Topology Scenario 8x

Simulation topology scenario 8x is almost the same as simulation topology scenario 2x. The only difference is that in simulation topology scenario 8x we aim to study the color handling mode of the 3CRED algorithm, thus simulation topology scenario 8x uses *Single Accounting Multiple Threshold* mode (see section 2.4.3. The settings are specified in tables 12, 13 and 14 with the only change seen in table 21.

Scheduler Parameters	
Color Handling Mode (all queues)	Single Accounting Multiple Threshold

Table 21: Simulation Dependent Scheduler Parameters – 8x

4.4.7 Simulation Topology Scenario 9x

This simulation is stand alone, and is used to evaluate different schedulers. This simulation topology scenario is the same as simulation topology scenario 2x with varying schedulers. The settings are specified in tables 22, 23 and 24.

Common parameters		
Scheduler used	varies	

FTP Sources	TCP window size/CBR rate	Packet size
EF Source	CBR rate 40 kb/s	500 bytes
AF1 Source	TCP window 64 packets	1500 bytes
AF2 Source	TCP window 64 packets	1500 bytes
AF3 Source	TCP window 64 packets	1500 bytes
AF4 Source	TCP window 64 packets	1500 bytes
BE Source	TCP window 64 packets	1500 bytes

WWW Sources	Number of clients	Max packet size
AF1 Source	20	1500 bytes
AF2 Source	30	1500 bytes
AF3 Source	40	1500 bytes
AF4 Source	50	1500 bytes
BE Source	60	1500 bytes

Noise Source	Noise Destination	CBR Rate	Packet size	PHB Class
Gateway A	Gateway B	varies	1500 bytes	varies
Gateway B	Gateway C	varies	1500 bytes	varies

Table 22: Simulation Dependent Topology Parameters – 9x

Scheduler Parameters	
Shared buffer mode	Shared Memory Node
Shared buffer mode size	256 000 bytes
Shared buffer mode ignore	EF Queue
Mean Packet Size (all queues)	varies
Color Handling Mode (all queues)	Higher-Priority-Inclusive

Scheduler Queue	Weight	Limit (bytes)
EF Queue	20	5000
AF1 Queue	10	20000
AF2 Queue	6	40000
AF3 Queue	4	60000
AF4 Queue	2	80000
BE Queue	1	100000

Token Bucket	PBS (bytes)	PIR (bytes)	CBS (bytes)	CIR (bytes)
EF Queue	-	-	-	-
AF1 Queue	200000	150000	150000	100000
AF2 Queue	250000	200000	200000	150000
AF3 Queue	300000	200000	200000	150000
AF4 Queue	300000	200000	200000	150000
BE Queue	-	-	-	-

Table 23: Simulation Dependent Scheduler Parameters – 9x

Four schedulers are used, one in each direction between gateway A and gateway B followed by another two in each direction between gateway B and gateway C. Each of these schedulers have the same buffer limits and settings (see figure 23).

The default active queue management parameters for simulation topology scenario 9x can be viewed in table 24.

AQM Parameters ALL			
Max drop probability	0.002		

AQM Parameters EF	Min threshold	Max threshold	Linterm
Green	-	-	-
Yellow	-	-	-
Red	-	-	-

AQM Parameters AF1	Min threshold	Max threshold	Linterm
Green	4000 bytes	10000 bytes	50
Yellow	8000 bytes	14000 bytes	40
Red	12000 bytes	18000 bytes	30

AQM Parameters AF2	Min threshold	Max threshold	Linterm
Green	8000 bytes	20000 bytes	50
Yellow	16000 bytes	28000 bytes	40
Red	24000 bytes	36000 bytes	30

AQM Parameters AF3	Min threshold	Max threshold	Linterm
Green	12000 bytes	30000 bytes	50
Yellow	24000 bytes	42000 bytes	40
Red	36000 bytes	54000 bytes	30

AQM Parameters AF4	Min threshold	Max threshold	Linterm
Green	16000 bytes	40000 bytes	50
Yellow	32000 bytes	56000 bytes	40
Red	48000 bytes	72000 bytes	30

AQM Parameters BE	Min threshold	Max threshold	Linterm
	40000 bytes	80000 bytes	50

Table 24: Active Queue Management Simulation Parameters – 9x

4.5 Simulation Study: Mean Packet Size Influence

Description

The *Mean Packet Size* (MPS) parameter, used in the RED algorithm family when calculating the EWMA, determines the decrease rate after the queue has been idle. The MPS parameter has no direct relation to the actual mean packet size of a traffic flow – it could in fact be seen as a negative gain factor used when updating the EWMA variable.

All time saving efforts when running our simulations are welcome. Therefore, limiting the number of parameters is of great interest (see section 4.3.2). We were hoping to determine a common MPS value for all are subsequent simulations. As a result of this we wanted to determine the mean packet size influence on network traffic in our first simulation. Especially, we looked for answers to the following questions:

- In general, all parameters that can be eliminated should be so. Therefore, in particular we wished to study whether the MPS parameter could be eliminated, hence is there a significant difference in network traffic when using different MPS values?
- If the MPS parameter can not be eliminated, is it then possible to manage a historical MPS value in the same manner as e.g. EWMA?

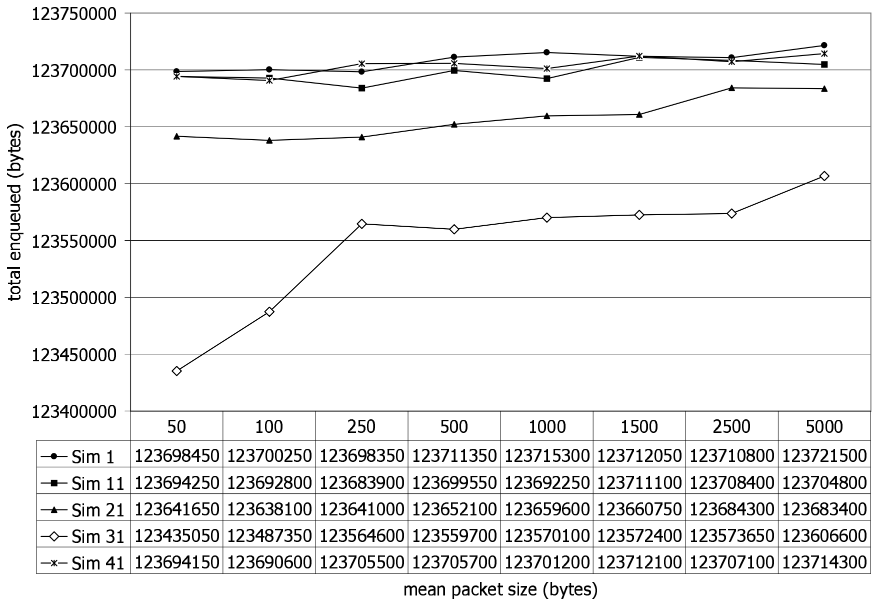
This simulation study is based on five final simulation traffic scenarios: Sim 1, Sim 11, Sim 21, Sim 31 and Sim 41. As mentioned earlier all of them use our extended bottleneck topology (see figure 24). For parameter settings of each different simulation topology scenario see tables in section 4.4 except for the mean packet size parameter. The parameter obviously changes, as it is the one that we wish to study, and it does so in eight discrete steps 50, 100, 250, 500, 1000, 1500, 2500 and 5000.

Results

There was an — in our opinion — insignificant difference in total bytes enqueued with different mean packet sizes, ranging from 50 to 5000, for the various simulation sets (see figure 25 and table 25).

Simulation set	Difference (bytes)	Difference (percentage)
Sim 1	23150	0,019%
Sim 11	27200	0,022%
Sim 21	46200	0,037%
Sim 31	171550	0,139%
Sim 41	23700	0,019%

Table 25: Difference in Total Bytes Enqueued



	50	100	250	500	1000	1500	2500	5000
Sim 1	123698450	123700250	123698350	123711350	123715300	123712050	123710800	123721500
Sim 11	123694250	123692800	123683900	123699550	123692250	123711100	123708400	123704800
Sim 21	123641650	123638100	123641000	123652100	123659600	123660750	123684300	123683400
Sim 31	123435050	123487350	123564600	123559700	123570100	123572400	123573650	123606600
Sim 41	123694150	123690600	123705500	123705700	123701200	123712100	123707100	123714300

Figure 25: Total Bytes Enqueued

The difference in total bytes enqueued within the different PHB groups were quite similar. The largest difference percentage (see table 26) was 4,432%, except for Sim 31's BE PHB class for which it was as high as 13,482%.

Simulation set	EF (%)	AF1 (%)	AF2 (%)	AF3 (%)	AF4 (%)	BE
Sim 1	0	1.195	1.048	3.103	3.045	2.690
Sim 11	0	0.967	1.817	2.168	4.432	1.861
Sim 21	0	1.119	0.698	2.254	2.928	1.203
Sim 31	0	2.276	2.264	1.987	2.811	13.482
Sim 41	0	2.415	0.452	0.528	0.919	1.410

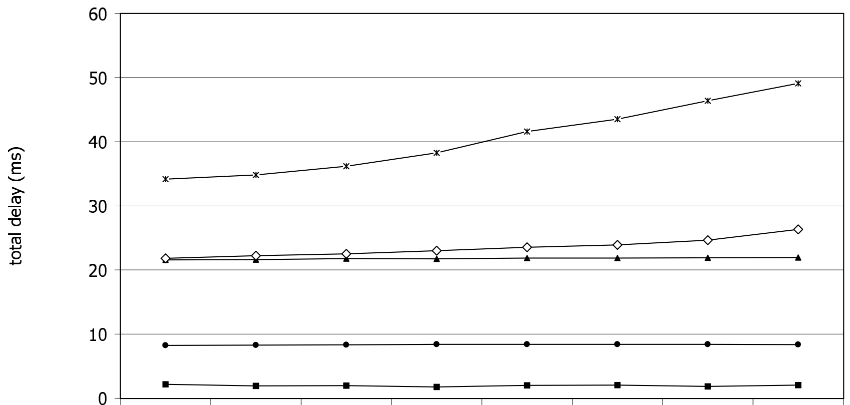
Table 26: Difference Percentage in Total Bytes Enqueued for PHB groups

Total bytes enqueued for Sim31's BE PHB class has an initial tendency to decrease with increased mean packet size (see table 27). Even though there is a large percentage difference, total bytes enqueued is in favor of mean packet size 50 as it has the highest value here.

Mean packet size	Total Bytes Enqueued
50	6396150
100	6300450
250	5866500
500	5717050
1000	5651850
1500	5685650
2500	5636250
5000	5667300
Difference	759900

Table 27: Total Bytes Enqueued for Sim 31's BE PHB class

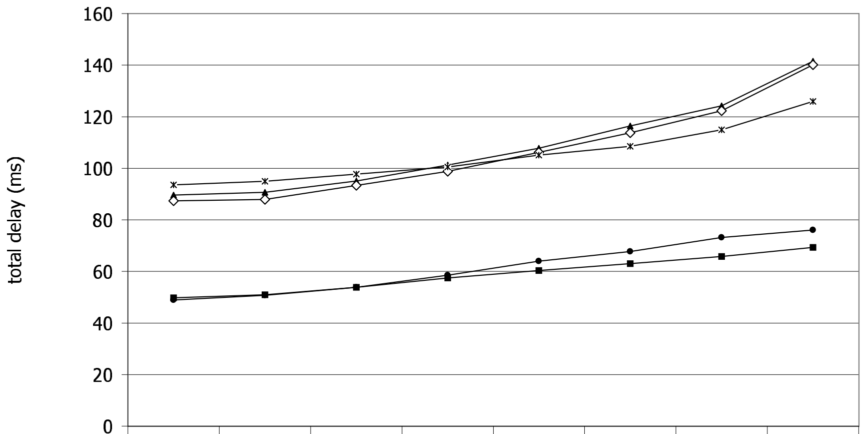
The network delay increases, except for some rare cases, with increased mean packet size (see figures 26 and 27). This behavior is more obvious as the PHB group receives a lower priority treatment by the scheduler.



	50	100	250	500	1000	1500	2500	5000
—●— Sim 1	8,26076	8,30202	8,31737	8,39669	8,3951	8,39821	8,39573	8,38466
—■— Sim 11	2,15544	1,91856	1,97365	1,76555	1,99921	2,04819	1,85139	2,04388
—▲— Sim 21	21,55771	21,61005	21,77161	21,73859	21,85477	21,87411	21,88727	21,95147
—◇— Sim 31	21,83636	22,23903	22,53297	23,02618	23,55706	23,91052	24,65973	26,32672
—×— Sim 41	34,15848	34,81011	36,1905	38,2623	41,56589	43,50773	46,36737	49,07816

mean packet size (bytes)

Figure 26: AF1 Network Delay



	50	100	250	500	1000	1500	2500	5000
—●— Sim 1	48,95944	50,75551	53,86851	58,5189	64,00973	67,74333	73,12108	76,07164
—■— Sim 11	49,75014	50,91131	53,85282	57,44111	60,37684	62,97152	65,83653	69,31607
—▲— Sim 21	89,57476	90,6702	95,10704	101,2046	107,8086	116,3809	124,2164	141,5005
—◇— Sim 31	87,34754	87,88504	93,34523	98,78449	106,2137	113,7502	122,3118	140,1173
—×— Sim 41	93,60873	94,97532	97,70302	100,503	105,0502	108,5577	114,9652	125,9689

mean packet size (bytes)

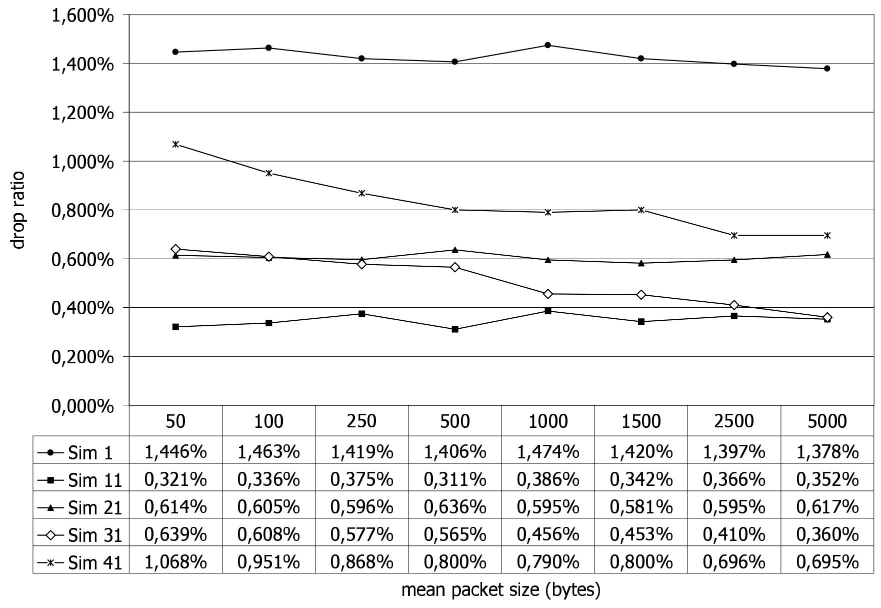
Figure 27: AF3 Network Delay

Typical simulation delay values, min, max and their difference for PHB groups, are shown in table 28. Min and max values for a PHB class are most often MPS values 50 respectively 5000, but not always,

PHB Group	Min Delay (ms)	Max Delay (ms)	Delay Difference (ms)
EF	0,54550	0,55337	0,00787
AF1	21,55771	21,95147	0,39376
AF2	62,73776	75,67076	12,93300
AF3	89,57476	141,50054	51,92578
AF4	150,84879	288,66259	137,81380
BE	771,45928	1328,82979	557,37051

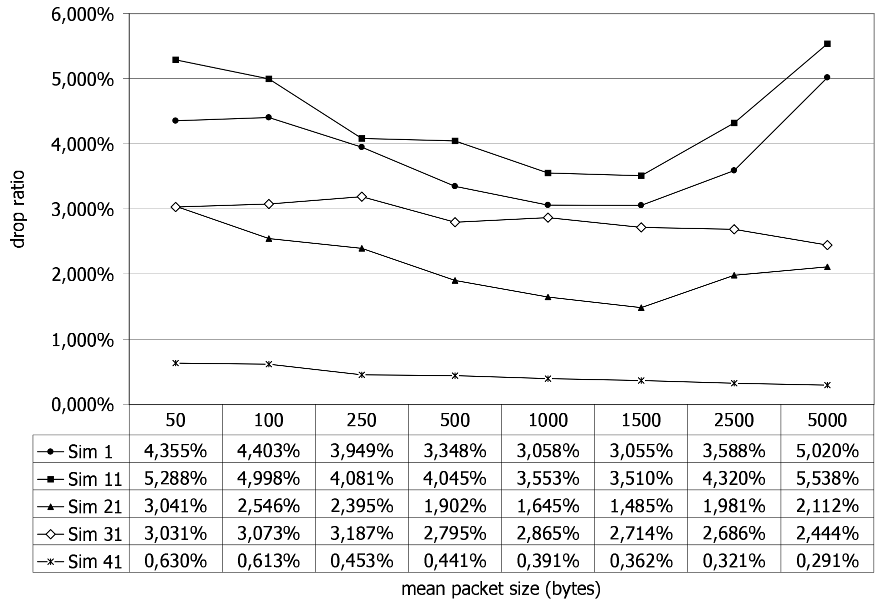
Table 28: Network Delay for Sim 21's PHB groups

Drop ratio values for a simulation set are of, or are combinations of, the tendency types: stable, decreasing and convex. The EF PHB class is stable where MPS 5000 is the most favorable. The PHB group AF1 tends to be stable. For most simulation sets the AF2 PHB group tends to be stable, however some are decreasing (see figure 28). The PHB group AF3 is also decreasing. All three PHB group, AF1, AF2 and AF3 has the the most favorable drop ratio at MPS 5000. The PHB group AF4 is convex (see figure 29) with MPS 1500 as the drop ratio minimum. BE PHB class also tends convex even if it is a little more unstable than AF4 PHB group. MPS 500 gives the most favorable drop ratio.



	50	100	250	500	1000	1500	2500	5000
Sim 1	1,446%	1,463%	1,419%	1,406%	1,474%	1,420%	1,397%	1,378%
Sim 11	0,321%	0,336%	0,375%	0,311%	0,386%	0,342%	0,366%	0,352%
Sim 21	0,614%	0,605%	0,596%	0,636%	0,595%	0,581%	0,595%	0,617%
Sim 31	0,639%	0,608%	0,577%	0,565%	0,456%	0,453%	0,410%	0,360%
Sim 41	1,068%	0,951%	0,868%	0,800%	0,790%	0,800%	0,696%	0,695%

Figure 28: AF2 Drop Ratio



	50	100	250	500	1000	1500	2500	5000
Sim 1	4,355%	4,403%	3,949%	3,348%	3,058%	3,055%	3,588%	5,020%
Sim 11	5,288%	4,998%	4,081%	4,045%	3,553%	3,510%	4,320%	5,538%
Sim 21	3,041%	2,546%	2,395%	1,902%	1,645%	1,485%	1,981%	2,112%
Sim 31	3,031%	3,073%	3,187%	2,795%	2,865%	2,714%	2,686%	2,444%
Sim 41	0,630%	0,613%	0,453%	0,441%	0,391%	0,362%	0,321%	0,291%

Figure 29: AF4 Drop Ratio

Typical simulation drop ratio values, min, max and their difference for PHB groups, are shown in table 29. Min and max values for a PHB group are most often MPS values 50 respectively 5000, but not always.

PHB Group	Min Drop Ratio	Max Drop Ratio	Drop Ratio Difference
EF	0.000%	0.000%	0.000%
AF1	0.936%	0.981%	0.046%
AF2	0.581%	0.636%	0.055%
AF3	0.497%	0.915%	0.418%
AF4	1.485%	3.041%	1.556%
BE	4.308%	7.418%	3.110%

Table 29: Drop Ratio for Sim 21's PHB classes

Conclusion

It is our conclusion to use 50 as the mean packet size for our subsequent simulations. The reason for this is that the mean packet size seem to have an insignificant impact on total bytes enqueued, both in total and for different PHB groups, and even if the drop ratio tends to increase with the mean packet size it has a low effect on total bytes enqueued. Our conclusion is based on the fact that a low delay is desirable and the lowest delay was with 50 as the mean packet size.

4.6 Simulation Study: Noise Influence

Description

Noise traffic is used to emulate traffic from parts of the network left out, due to comprehensiveness, in our simulations. Its influence on network stability and performance can be seen as the networks background load sensitivity.

This simulations intention was to study how much noise traffic a DiffServ network can manage without it affecting the other classes to some great extent.

The major questions to whom we looked for answers were:

- Is there a limit of how much noise traffic a differentiated services network can manage without affecting other services and PHB classes? If so, what is this limit? Is there a crucial limit where the network breaks down and cease to function? What classes are affected by this supposedly breakdown (lower, higher or all classes)?
- In what way does noise from *Gateway A* to *Gateway B* affect the traffic from *Gateway B* to *Gateway C*?
- In what way does noise from *Gateway B* to *Gateway C* affect the traffic from *Gateway A* to *Gateway B*?
- How much did the choice to use AF41 as the noise PHB class for all simulations affect the results?

This simulation uses simulation topology scenarios 0x, 1x, 2x, 3x and 4x (parameters can be found in tables in sections 4.4.1 to 4.4.5). As mentioned earlier all uses our extended bottleneck topology (see figure 24) and mean packet size 50.

This simulation contains three sub-simulations. In the first sub-simulation we only transmit noise between *Gateway A* and *Gateway B*. In the second sub-simulation noise is only transmitted between *Gateway B* and *Gateway C*. In the last and third sub-simulation noise is transmitted between all gateways. This study sends noise using CBR and the AF41 PHB class on both of the noise paths, *Gateway A* to *Gateway B* and *Gateway B* to *Gateway C*. The noise load is increased in eleven discrete steps of 50 kb/s, ranging from 0 kb/s to 500 kb/s.

The convention AFx-A or AFx-B will be used to represent the drop ratio between *Gateway A* and *Gateway B* respectively between *Gateway B* and *Gateway C*.

Results

Since this simulation has three sub-simulations we will study them in turn and use all three to try to get a common conclusion.

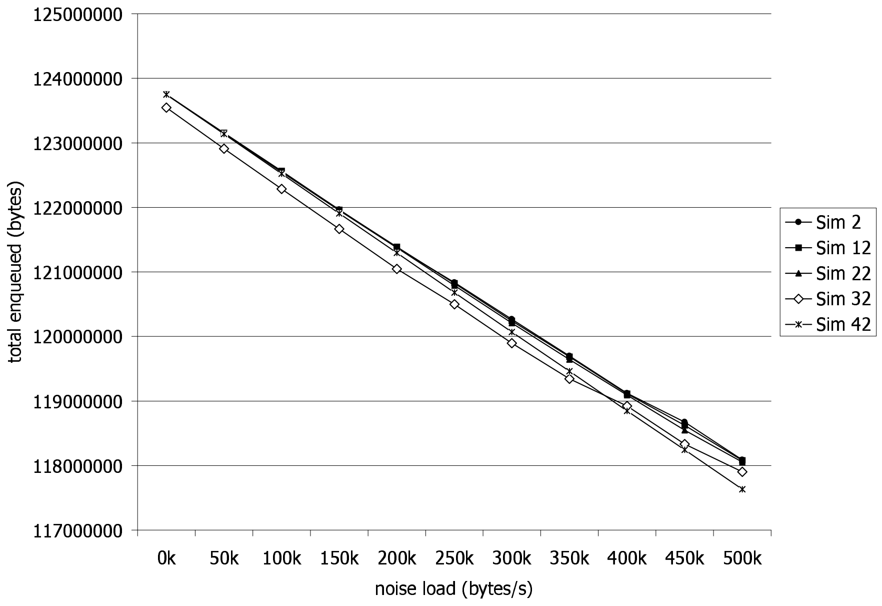


Figure 30: Total Bytes Enqueued with noise A → B

For the first sub-simulation the total number of bytes enqueued (see figure 30) decrease as the noise load increases. This behavior was expected since the more interrupted the TCP sources get, the more seldom the TCP sources will transmit at full speed due to the sudden decreases in window sizes. However, there seems to be no crucial limit where the network breaks down and throughput becomes unusually low. In figure 31 it can be seen that the AF1 PHB group do not follow the total bytes enqueued graph. The AF1 PHB group graph is stable and does not seem to be affected by the increasing noise load at all. The same behavior is displayed for all PHB groups except AF4 which is also the noise PHB. Total Bytes Enqueued for the AF4 PHB group (see figure 32) decreases when the noise increases. The reason the AF4 PHB group total bytes enqueued decrease is since more noise increases the chance of a drop. When a drop is made the TCP window decreases and henceforth the TCP senders cannot transmit at full speed. Note that there are both a AF41 PHB class CBR noise traffic sender and TCP/FTP (long-lived) traffic sources, which gets interrupted by the noise source.

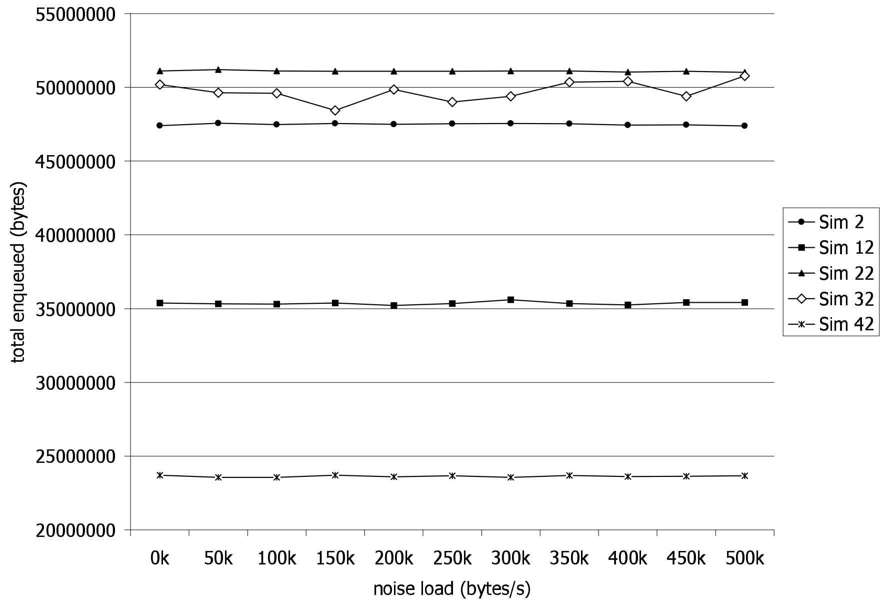


Figure 31: AF1 Total Bytes Enqueued with noise A → B

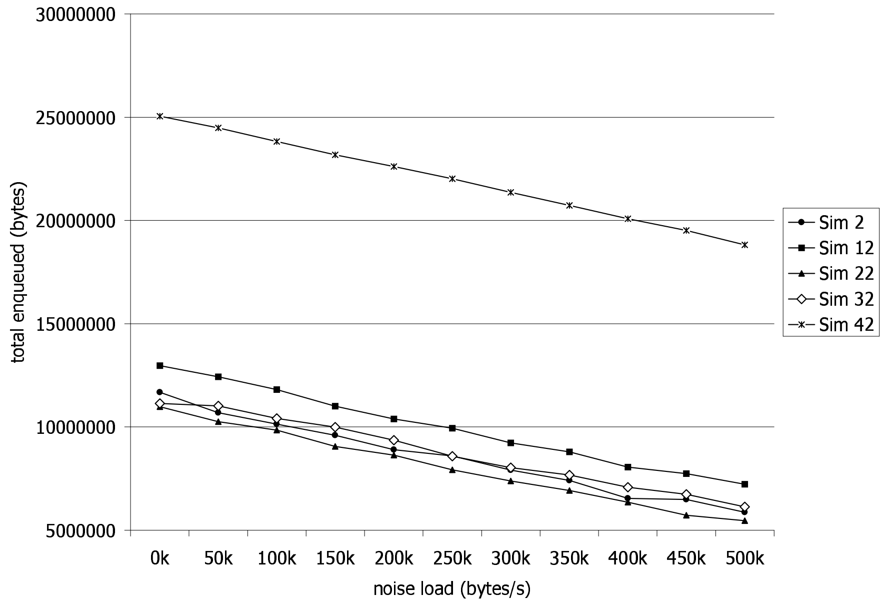


Figure 32: AF4 Total Bytes Enqueued with noise A → B

The same behavior that was displayed for total bytes enqueued can also be seen with network delay. The delay for all PHB classes except for PHB group AF4, is stable and seems not to be affected by increasing the noise rate. The delay for PHB group AF4 increases steadily increasing as the noise increases and at noise load 150 kb/s and 300 kb/s there are a more noticeable increases in delay. This behavior for the AF4 PHB group can be seen in figure 34 while the AF1 PHB group behavior can be seen in figure 33.

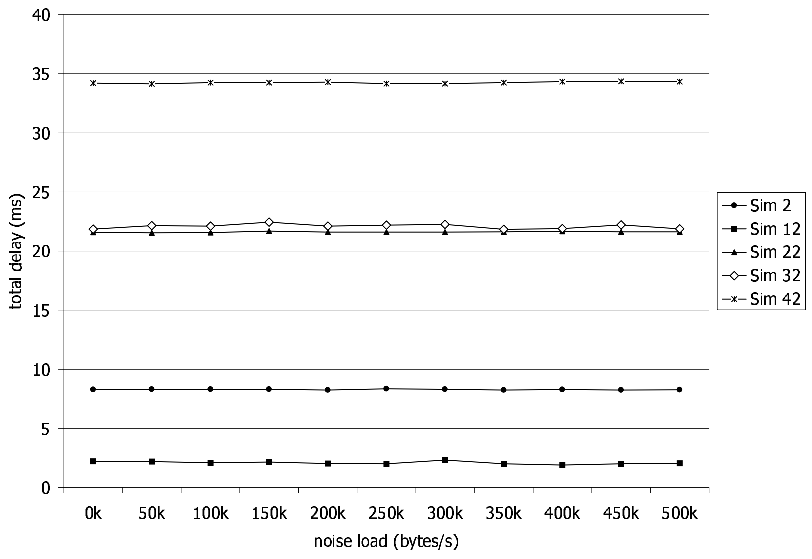


Figure 33: AF1 Network Delay with noise A → B

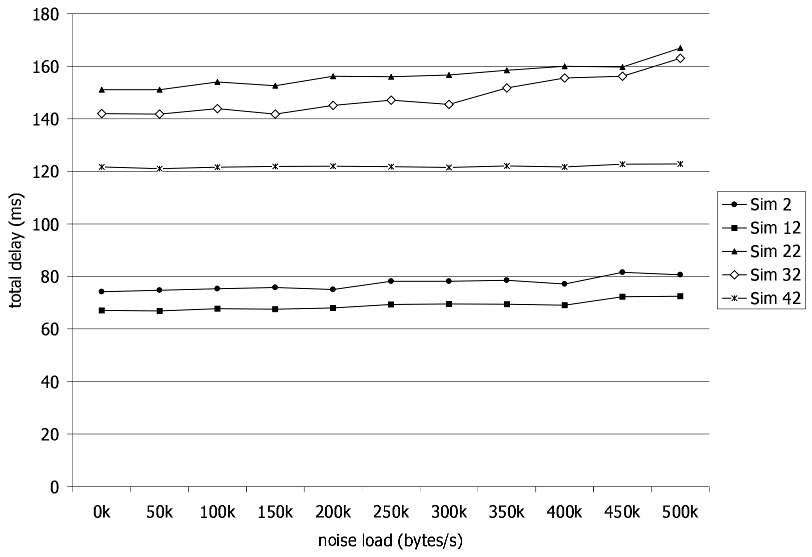


Figure 34: AF4 Network Delay with noise A → B

Because noise is only sent between the first two gateways, there is no need to study the drop ratio between the last two gateways. The drop ratio for all PHB groups (see example in figure 35) tends to be stable except for PHB group AF4.

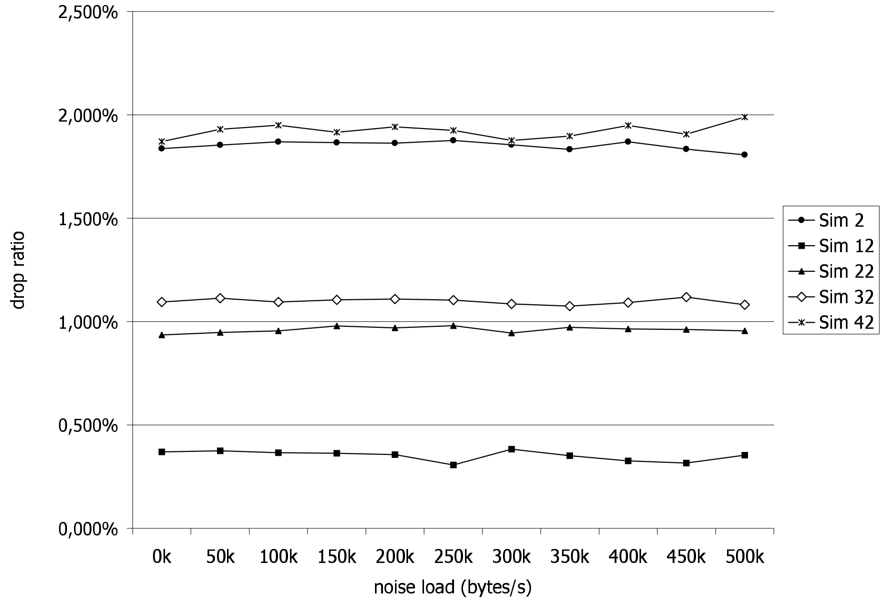


Figure 35: AF1-A Drop Ratio with noise A → B

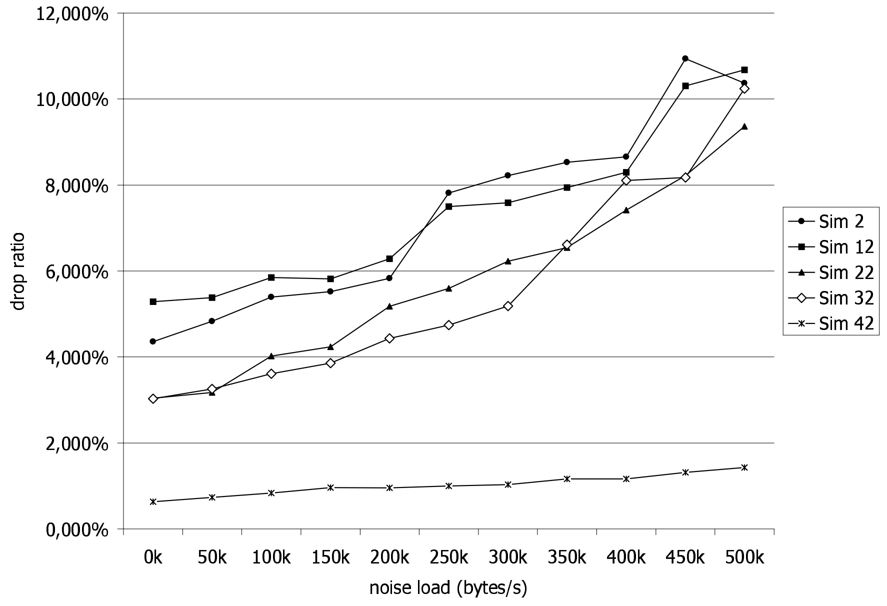


Figure 36: AF4-A Drop Ratio with noise A → B

The drop ratio for the AF4 PHB group (see figure 36) increases for every increase in noise load, and at noise load 500 kb/s it stops with a drop ratio around 10 percent for all simulation topology scenarios except 4x. Normally this would be unacceptable. The reason for this high drop ratio is because the scheduler protects the other flows from this unresponsive AF4 PHB group traffic flow. Only a certain amount (decided by the scheduler weight) of AF4 PHB group traffic is accepted. All traffic exceeding this amount cannot be handled and is therefore dropped. The reason simulation topology scenario 4x does not follow the other drop ratios is that simulation topology scenario 4x is color-blind, and that all PHB groups has the same scheduler weights.

The second sub-simulation sends only noise between the last gateways. Even though this sub-simulation might seem very similar to the first sub-simulation there is a significant difference. In the first sub-simulation both the normal and the noise traffic flow gets shaped at the first gateway, but in the second sub-simulation the normal traffic gets shaped at the first gateway and the noise traffic gets shaped at the second gateway.

As can be seen in figure 37 there is no significant variation in the total bytes enqueued for this sub-simulation except for the simulation topology scenario 3x. Even the difference of ≈ 270 kb in total bytes enqueued experienced for simulation topology scenario 3x is insignificant when compared to the total bytes enqueued value. There also seem to be no crucial limit where the network breaks down.

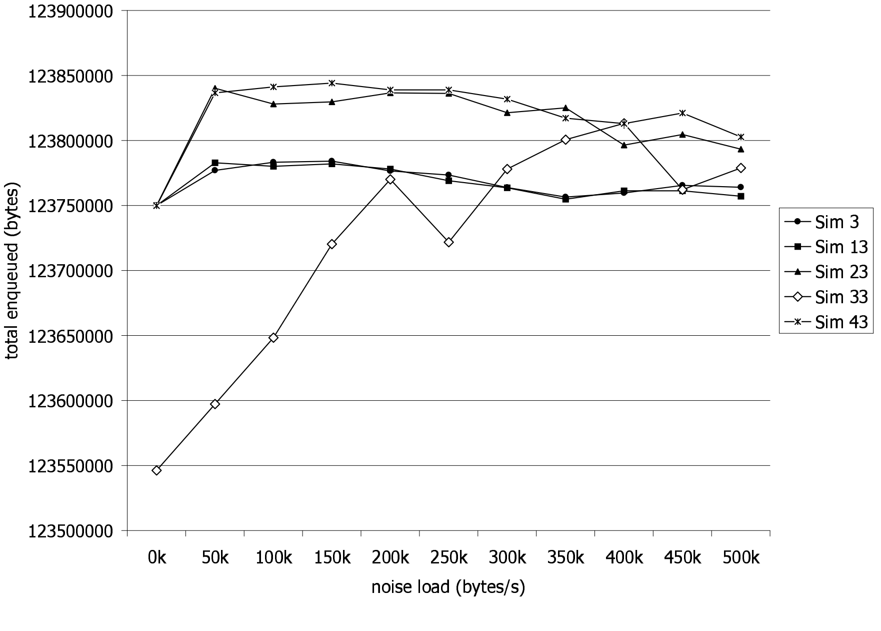


Figure 37: Total Bytes Enqueued with noise B $\rightarrow$ C

The best throughput is received at either noise load between 50 kb/s and 150 kb/s depending on the simulation topology scenario. Maybe this limit could be the value where the network can manage most noise. Additional noise causes performance degradation while a lower noise load results in non-utilized resources.

The individual PHB groups total bytes enqueued are stable and follow the behavior of the total bytes enqueued for all PHB groups (see figure 37).

All PHB groups, except for the EF PHB group that have a low delay, exhibits the same appearance for the network delay (see figure 38) in this sub-simulation.

When noise load is increased to 50kb/s all PHB groups experienced a increase in delay (see example in figure 38). After this sudden increase the curve evens out as more noise is added.

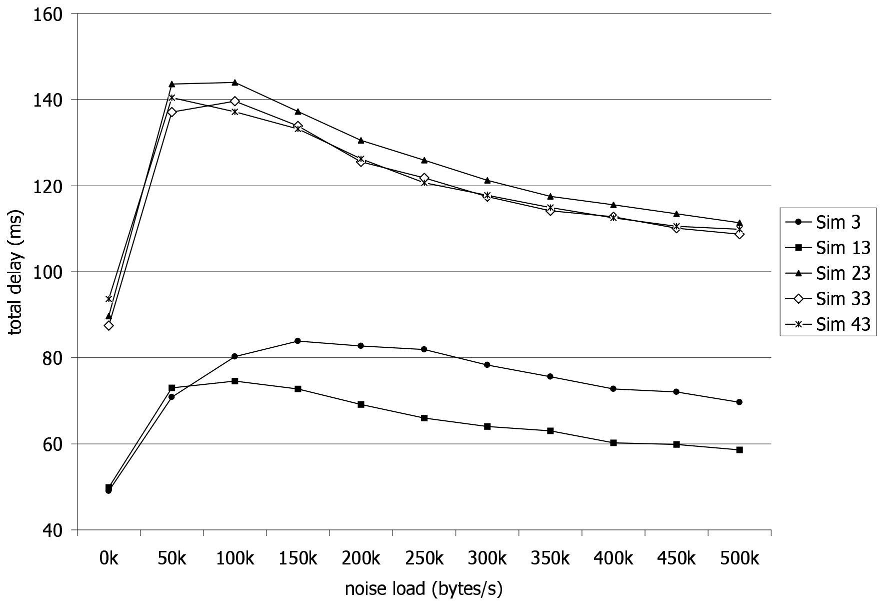


Figure 38: AF3 Network Delay with noise B → C

The drop ratio has to be studied on two locations in this simulation. Firstly between the two first gateways and then between the two last.

The appearance of the drop ratio between the first gateways are common for all PHB groups, except EF that does not drop packets. The higher the noise load, the lower drop ratio the PHB groups experience. This is most likely because the noise travels between the last two gateways, and when the noise increases the TCP sources get disturbed resulting in a lower pressure on the first gateways. The AF1 PHB group drop ratio is displayed in figure 39.

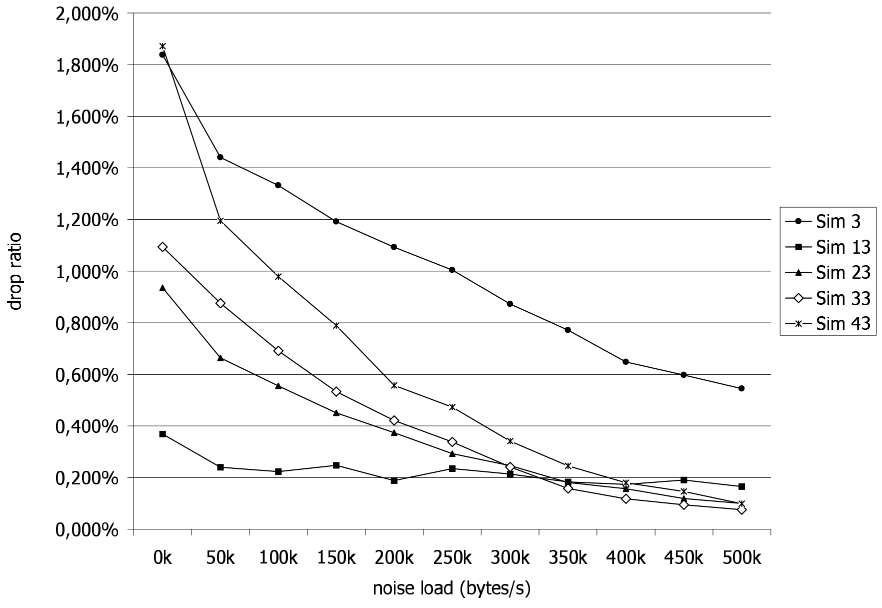


Figure 39: AF1-A Drop ratio with noise B → C

The AF1 PHB group drop ratio between Gateway B and Gateway C (see figure 40) increases with the noise load. This behavior applies to all PHB groups, except the EF PHB group that have no packets dropped. This is not so strange because the network simply cannot handle all packets.

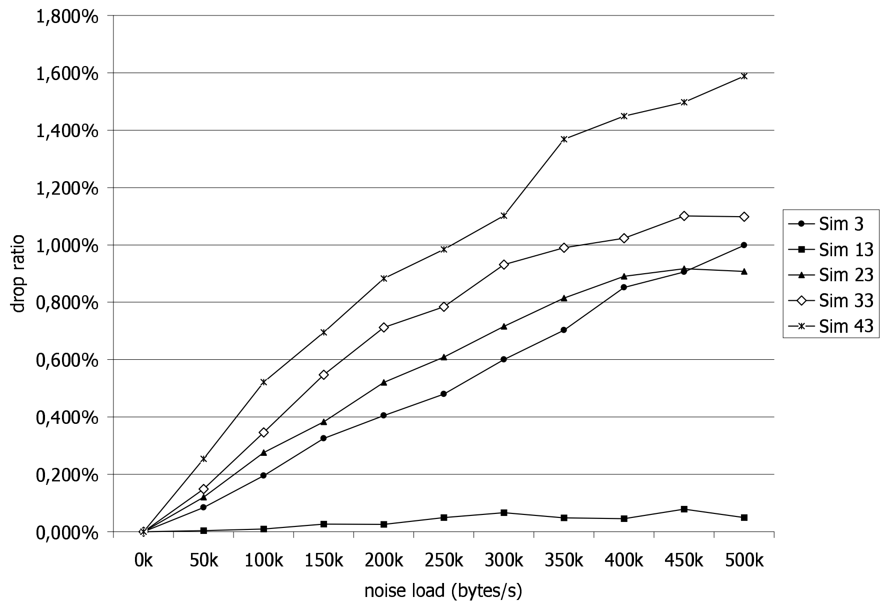


Figure 40: AF1-B Drop ratio with noise B $\rightarrow$ C

The third and final sub-simulation in this noise influence study has noise between Gateway A and Gateway B as well as between Gateway B and Gateway C. This is the most realistic sub-simulation of the three as there will always exist unresponsive noise sources at all points of a network.

As can be seen in figure 41 the total bytes enqueued for all simulation topology scenarios except x3 tends to be stable. No matter how much the noise load increases, the same number of bytes gets enqueued. Simulation topology scenario 3x is as mentioned different from the others and have maximums in noise loads 200 kb/s and 500 kb/s.

The individual PHB groups are no different, i.e. they are very stable and tends to be unaffected as the noise load increases.

The delay for PHB groups AF1, AF2 and AF3 all have marginal increases in delay when the noise load also increases. The PHB group AF4 has a major increase in delay between different noise loads (about twice when comparing 0 kb/s and 500 kb/s). The delay for the AF1 PHB group can be seen in figure 42 and the delay for the AF4 PHB group can be seen in figure 43.

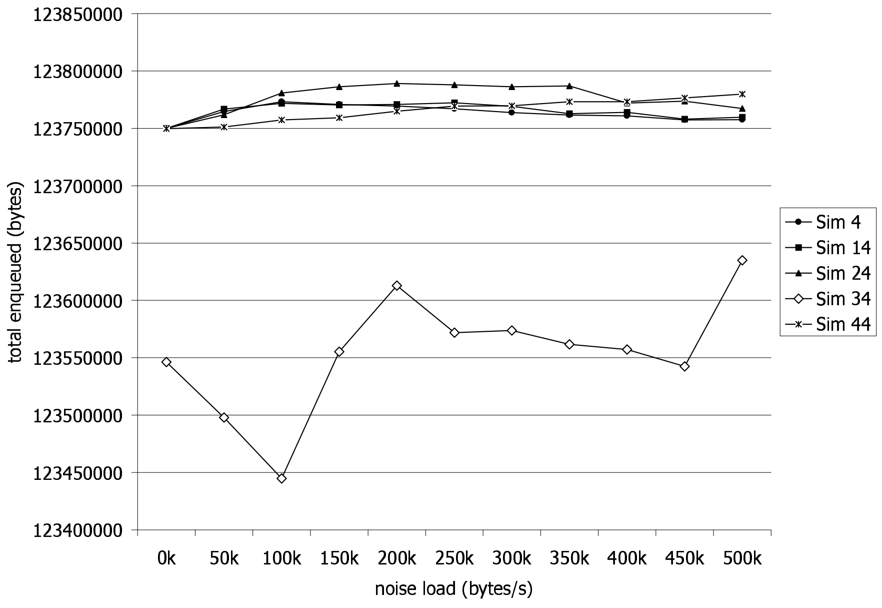


Figure 41: Total Bytes Enqueued with noise A → B, B → C

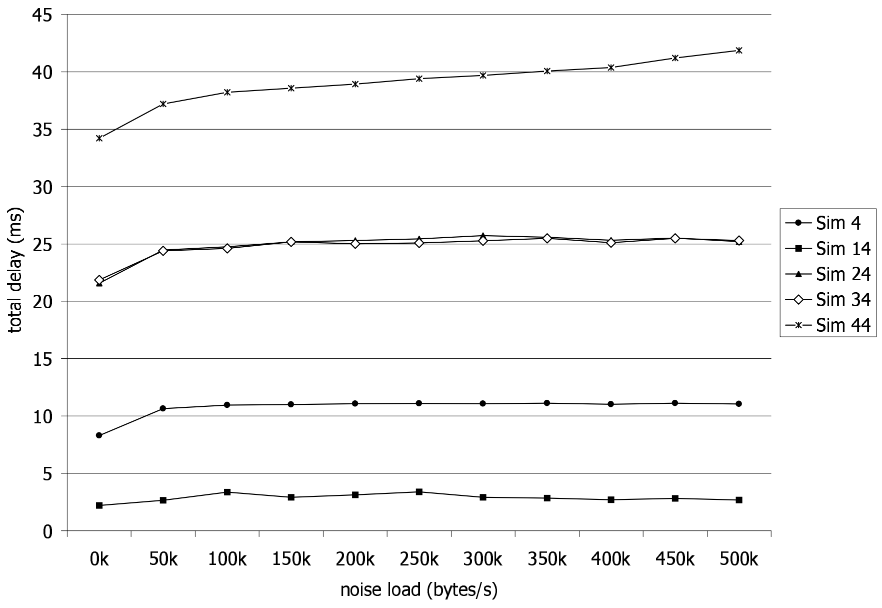


Figure 42: AF1 Network Delay with noise A → B, B → C

Because noise is sent between all gateways we need to study the drop ratio both between Gateway A and Gateway B, as well as between Gateway B and Gateway C.

The drop ratio between Gateway A and Gateway B for the AF1, AF2 and AF3 PHB groups (see example in figure 44) decreases slightly. Between Gateway A and Gateway B the drop ratio for the AF4 PHB group (see figure 45) increases steadily as the noise increases. This is obviously because the network cannot handle all traffic being sent. The drop ratio for the AF4 PHB group between Gateway B and Gateway C (see figure 46) increases significantly as well.

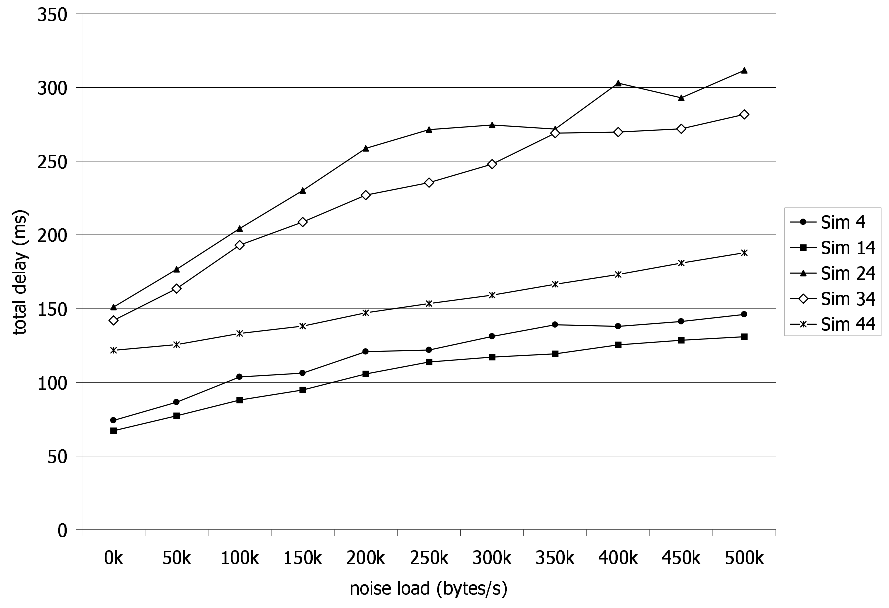


Figure 43: AF4 Network Delay with noise A → B, B → C

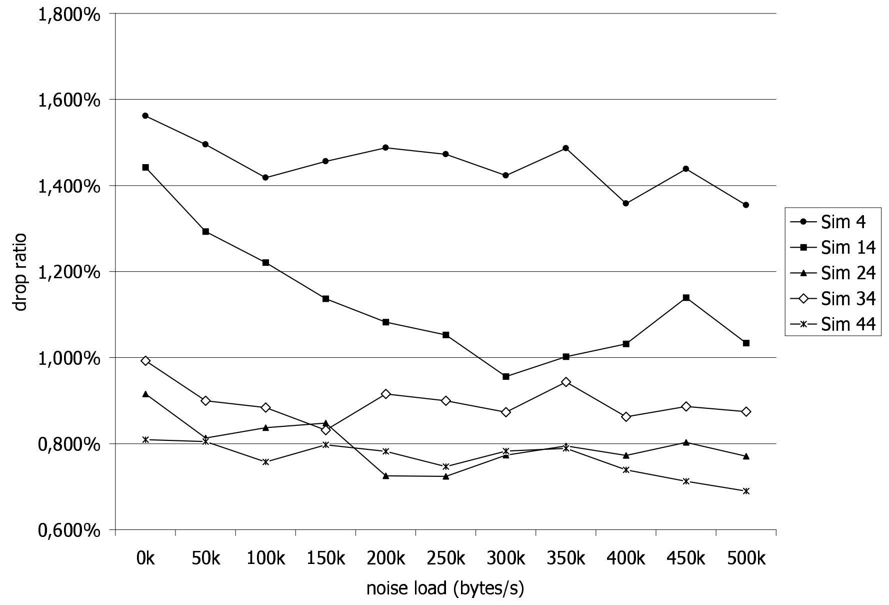


Figure 44: AF3-A Drop Ratio with noise A → B, B → C

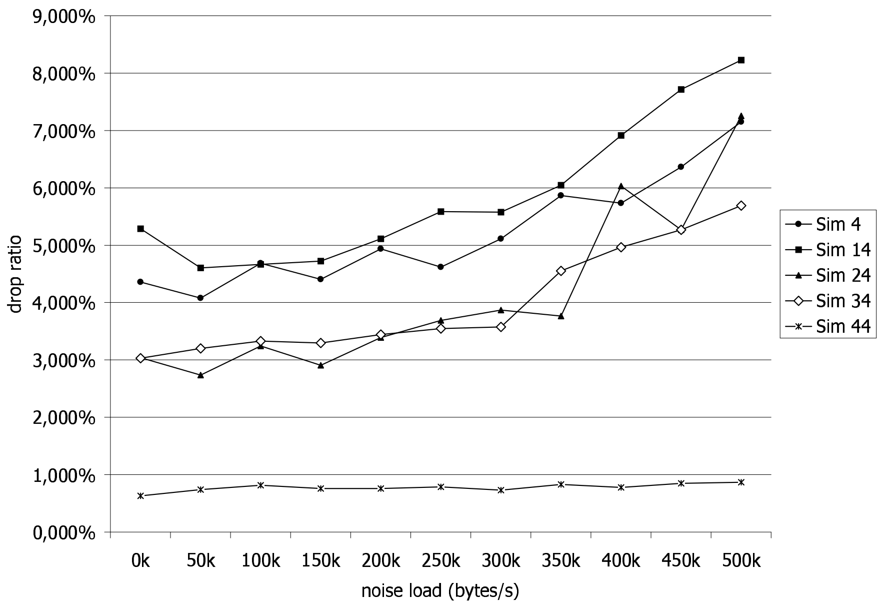


Figure 45: AF4-A Drop Ratio with noise A → B,B → C

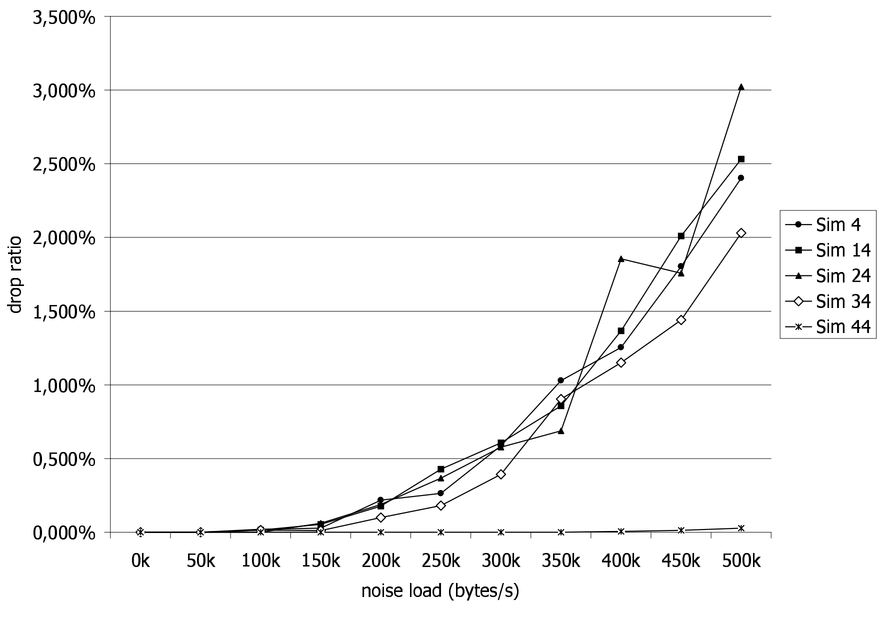


Figure 46: AF4-B Drop Ratio with noise A → B, B → C

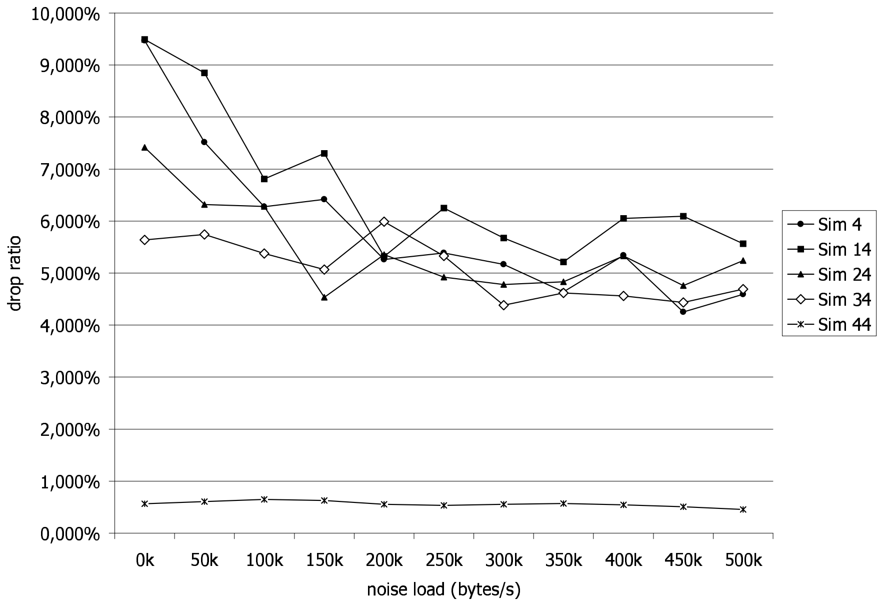


Figure 47: BE-A Drop Ratio with noise A $\rightarrow$ B, B $\rightarrow$ C

As the noise increases the BE PHB group drop ratio (see figure 47) between Gateway A and Gateway B decreases. The drop ratio between Gateway B and Gateway C for the AF1, AF2, AF3, BE and EF PHB groups is very small, making them insignificant to the results. This is once again mainly since we send the noise traffic as the AF4 PHB group.

Conclusion

For the first sub-simulation it has been shown that the only PHB group that gets affected is the noise PHB group itself. The total bytes enqueued decreased only for the AF4 PHB group. It was also only in the AF4 PHB group an increase in delay and drop ratio was found. The conclusion that can be drawn from this is that the WF^{2}Q+ scheduler only allows a group to send a predetermined (by the scheduler weight) amount of bytes each round and is henceforth very good at its job in this aspect.

With the single exception of simulation topology 3x the PHB groups and total bytes enqueued was also stable for the second sub-simulation. In this sub-simulation there was however no clear increase or decrease regarding the delay. A maximum delay was found with a noise load between 50 kb/s and 150 kb/s. The drop ratio for all PHB groups decreased as the noise load between Gateway A and Gateway B increased, while the drop ratio increased as the noise load between Gateway B and Gateway C increased. The reason the drop ratio looks like it does is as mentioned previously that the TCP senders seldom send packets at full speed due to the window decreasing. Same setup at Gateway A and Gateway B causes shaped traffic at Gateway A to result in a scarce amount of excess resources at Gateway B. This amount is however utilized best by the noise source at loads between 50 kb/s and 150 kb/s.

In the third and most realistic sub-simulation it is our opinion once again that the total bytes enqueued were stable. Delay increased for all PHB groups as expected since the scheduler had more packets to handle. The delay increased the most for the AF4 PHB group, because noise is sent as this PHB.

To answer the initial questions, we found that there is no clear limit on the noise load a network can manage. This limit would be very simulation dependent which in this simulation seemed to be between noise loads 50 kb/s and 150 kb/s where the best throughput was received. No crucial limit was found except for the 10% drop ratio for the AF4 PHB group where the network got congested resulting in a breakdown. But the 10% drop ratio only affects traffic in the actual noise PHB class arriving at Gateway A. Usually, TCP sender backoff results in a lower drop ratio while a misbehaving CBR source does not.

Noise between Gateway A and Gateway B did not affect the traffic between Gateway B and Gateway C. The only thing that got affected was the noise PHB group itself. In a future simulation about noise influence it would be preferable to be able to separate the noise traffic from the PHB groups. Maybe by sending noise as a own temporary PHB class or having separate statistics for noise traffic.

It is our opinion that noise between Gateway B and Gateway C did affect the traffic between Gateway A and Gateway B. This because a higher traffic load between Gateway B and Gateway C results in more packet drops which in turn causes the TCP sources to backoff (lowering their TCP window sizes).

In this section we have chosen to use AF41 as the PHB class noise, while letting all load parameters change continuously. In the simulations in section 4.7 we evaluate a similar study with three fixed rates while the noise PHB class is allowed to vary.

4.7 Simulation Study: PHB Group Combination Noise Influence

Description

In this simulation study we wanted to study the impact on network stability and performance of noise being sent as different PHB groups. We focused on typical behaviors/characteristics for each simulation topology scenario and not actual statistical values for total bytes enqueued, delay and drop ratio.

We looked for answers to the following questions:

- What happens when noise is sent as a higher priority PHB group/class compared to the AF41 PHB class which is used by previous simulations studies including noise?
- Are there any significant differences in performance when sending noise as two priority adjacent PHB classes, e.g. AF3 or AF4?

This simulation study is based on six simulation traffic scenarios: Sim 25 to Sim 27 and Sim 44 to Sim 47. As mentioned earlier all of them use our extended bottleneck topology (see figure 24) and mean packet size 50.

Parameter settings except noise parameters (load and PHB group combinations) for each different simulation topology scenario is shown in tables in sections 4.4.3 and 4.4.5. This study sends noise using combinations of all six PHB groups (EF, AF1-AF4 and BE) on both of the noise paths, *Gateway A* to *Gateway B* and *Gateway B* to *Gateway C*. The noise load in doubled in three discrete steps: 75 kb/s (Sim x5), 150 kb/s (Sim x6) and 300 kb/s (Sim x7).

Results

Before bringing up the results we would like to point out that total bytes enqueued, both total and for PHB groups, are a little misleading. Total bytes enqueued at Gateway C (see figure 24) does not include the noise sent from Gateway A to Gateway B. The correct solution would have been to use a new topology (see figure 48) where an additional node, *Gateway D*, is inserted directly after Gateway C to get statistics without the the noise traffic included in statistics (only its influence). Results for which the PHB group is the same as the noise PHB group sent between Gateway B and Gateway C therefore differs from the rest.

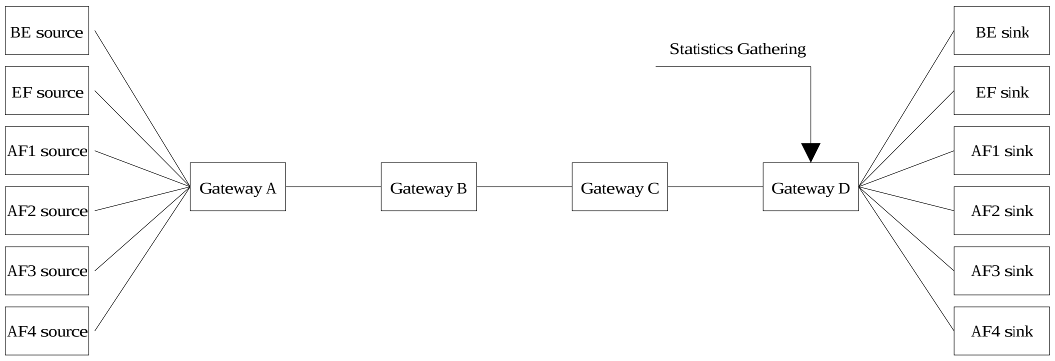


Figure 48: Proposed Final Simulations Topology

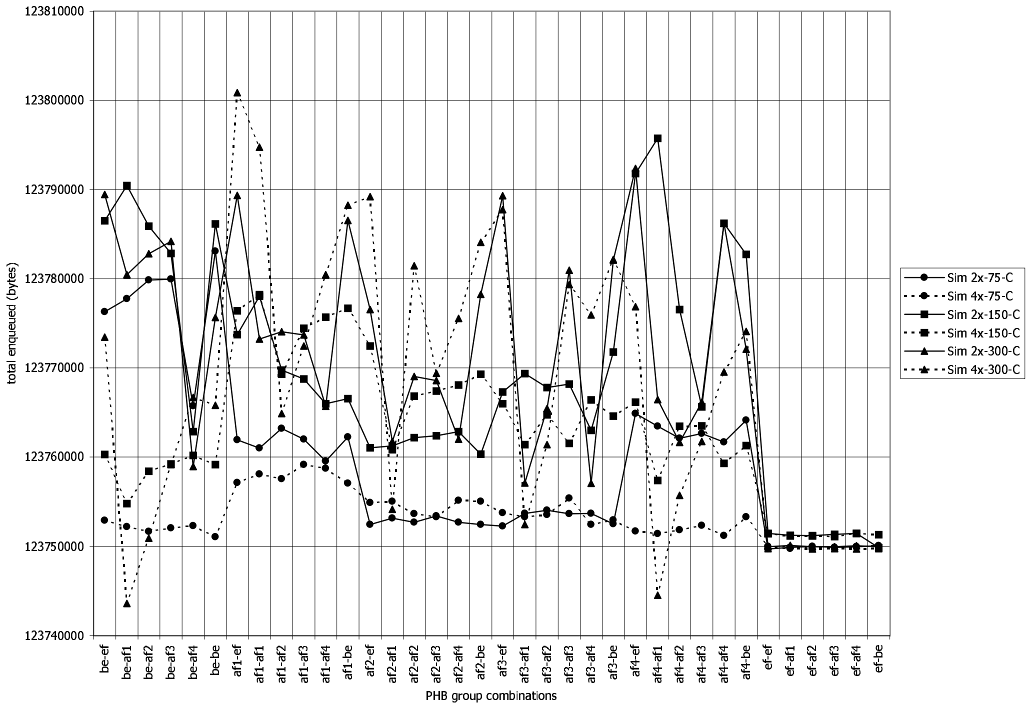


Figure 49: Total Bytes Enqueued

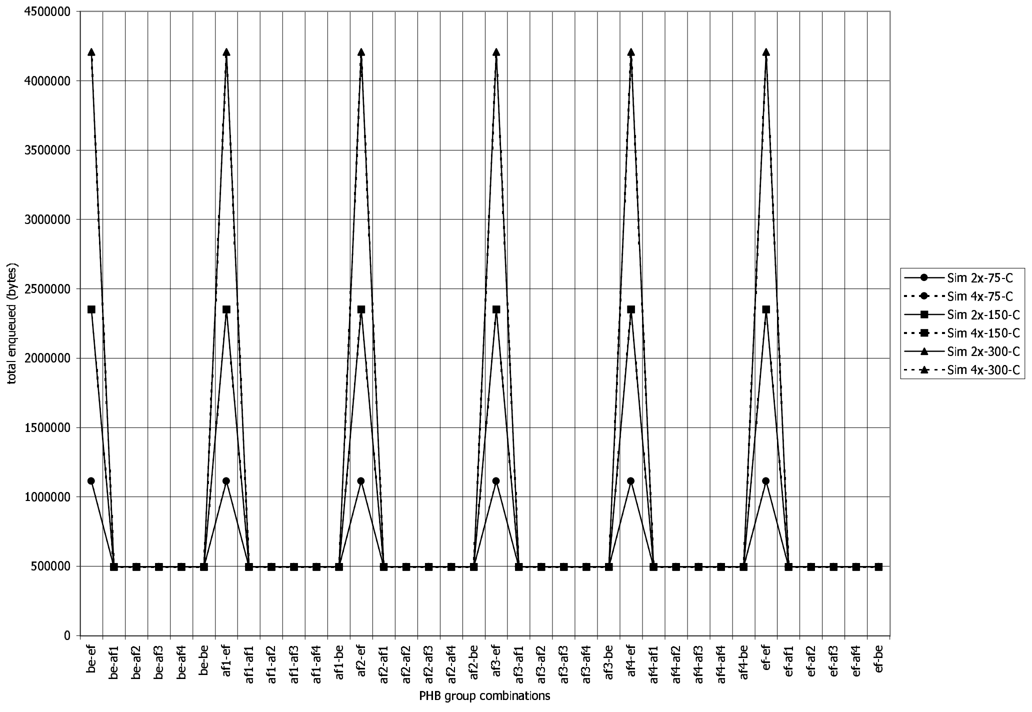


Figure 50: EF Total Bytes Enqueued

Total bytes enqueued (see figure 49) is unstable for all PHB combinations except the EF PHB class when sending noise as a EF PHB class from Gateway A. However, the difference is very small for non-EF PHB group traffic as well, with at most ≈36 kb for Sim 2x and ≈456 kb for 4x (largest difference for the EF PHB class when sending noise from Gateway A with the same PHB class is 1750 bytes for both Sim 2x and Sim 4x). The reason for the stableness of the EF PHB class is that it has a higher priority in the scheduler and is not disturbed by other traffic and backoffs.

The EF PHB class (see figure 50) would have been very stable in total bytes enqueued for different PHB group combinations if total bytes enqueued did not incorporate noise traffic. All other PHB groups, AF1, AF2, AF3, AF4 and BE, experience the same behavior as total bytes enqueued (see figure 51):

- *decreases* for the PHB group when sending noise with that same PHB group from Gateway A
- *increases* for the PHB group when sending noise with that same PHB group from Gateway B

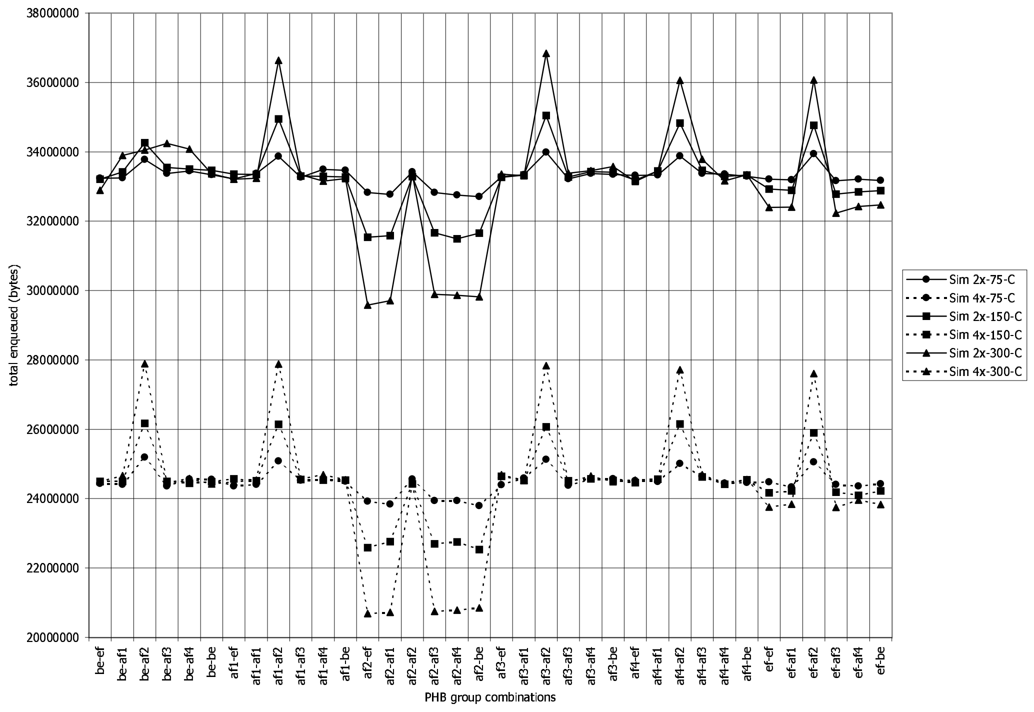


Figure 51: AF2 Total Bytes Enqueued

The delay for the EF (see figure 52) and BE PHB classes are stable for all PHB combinations except the increase for those PHB group combinations that send noise as the EF PHB group from Gateway B. When noise is sent as EF from Gateway A there exists a general increase in delay for all PHB group combinations.

All the AF PHB groups (see figure 53) had an unstable delay when noise is sent as the BE PHB class from Gateway A and a stable delay when noise is sent as EF PHB class from Gateway A. Delay for all AF PHBs groups, when noise is sent as an AF PHB class from Gateway A, tends to be stable with increase in delay for the PHB group of the same PHB group as the noise sent from Gateway B.

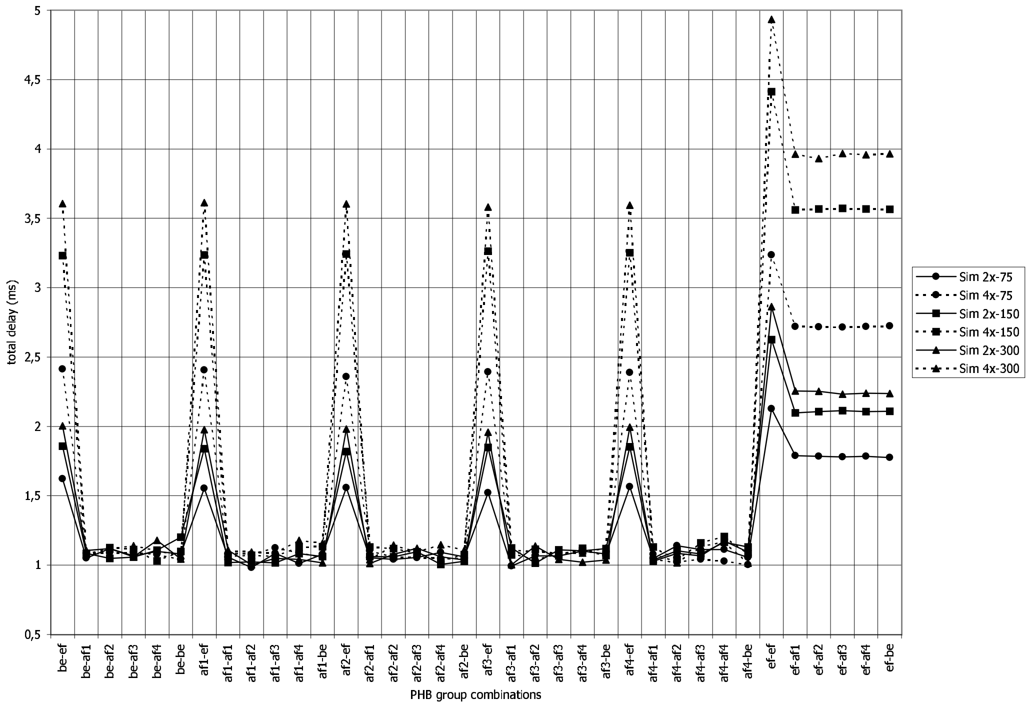


Figure 52: EF Network Delay

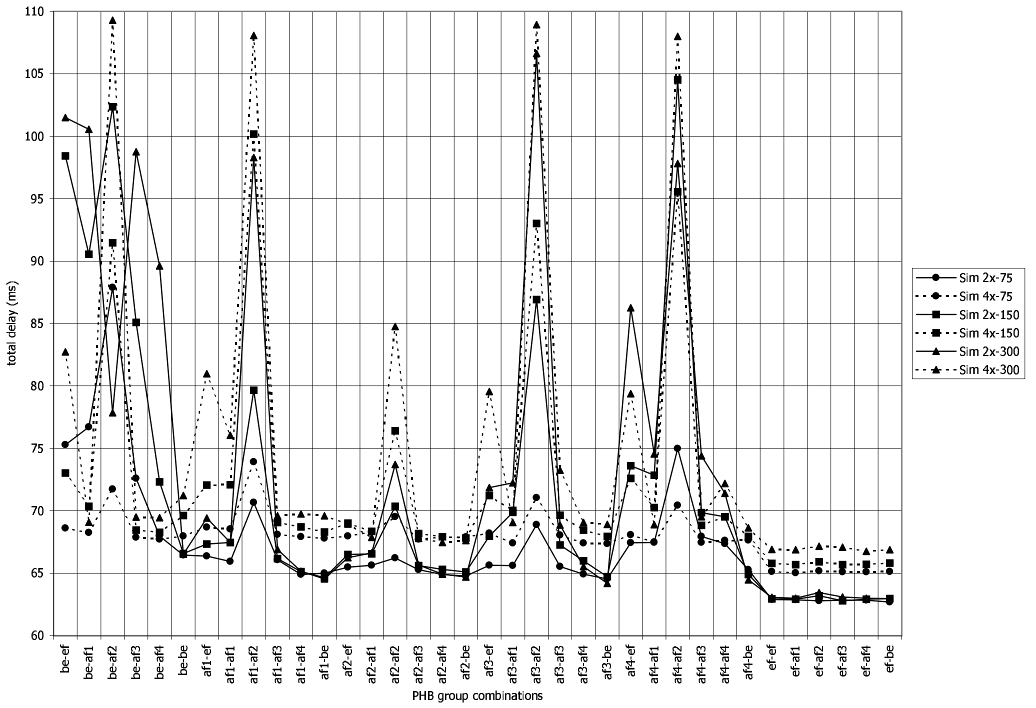


Figure 53: AF2 Network Delay

Drop ratio on both Gateway A and Gateway B for the EF PHB class is always zero. For all AF PHB groups drop ratio in Gateway A is stable for all PHB group combinations except for the decrease (minimum) in the drop ratio at Gateway A for a PHB when sending noise from Gateway B as the same PHB group. Also drop ratio generally increase at Gateway A when sending noise with the same PHB group from Gateway A. Drop ratio for BE-A PHB class (BE PHB class on Gateway A) is stable with a higher drop ratio level (up to ≈20%) when sending noise from Gateway A as the BE PHB class. Sending noise from Gateway A as the BE PHB class results in a higher drop ratio level for all AF PHB groups. Otherwise, it is stable with increased drop ratio for the PHB group which noise is sent from Gateway B as except when also sending noise as the same PHB group from Gateway A (does not apply to PHB group AF4). Drop ratio for the BE-B PHB class is generally stable at zero. However at three PHB group combinations, BE-BE, AF4-EF, AF4-BE, there exist local maximums (global is BE-BE with ≈7.4%).

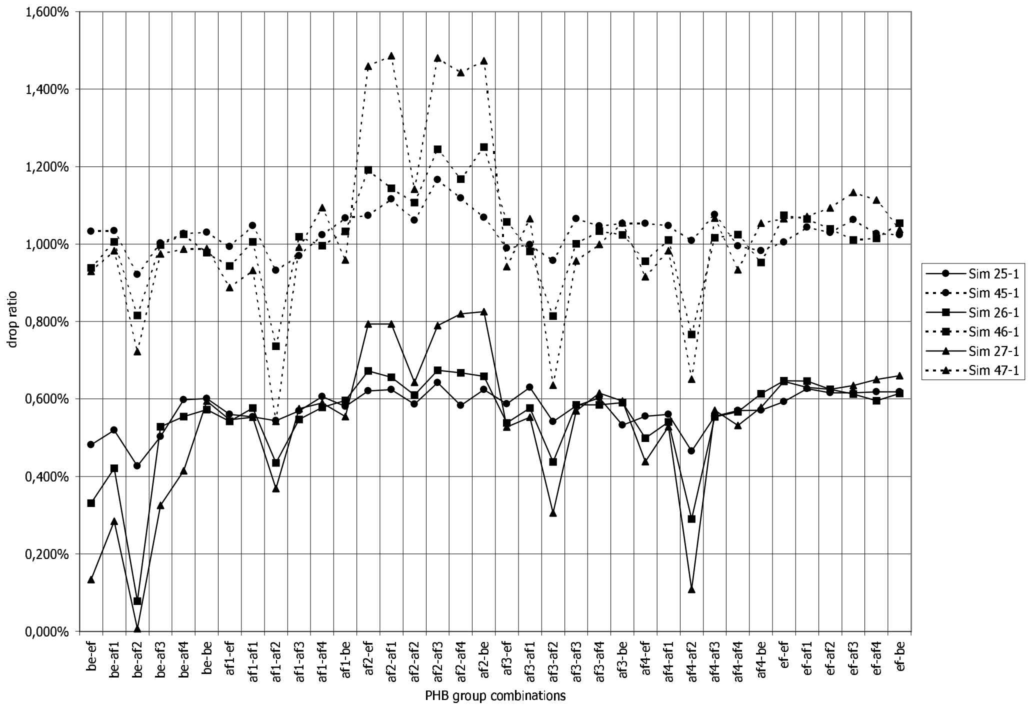


Figure 54: AF2-A Drop Ratio

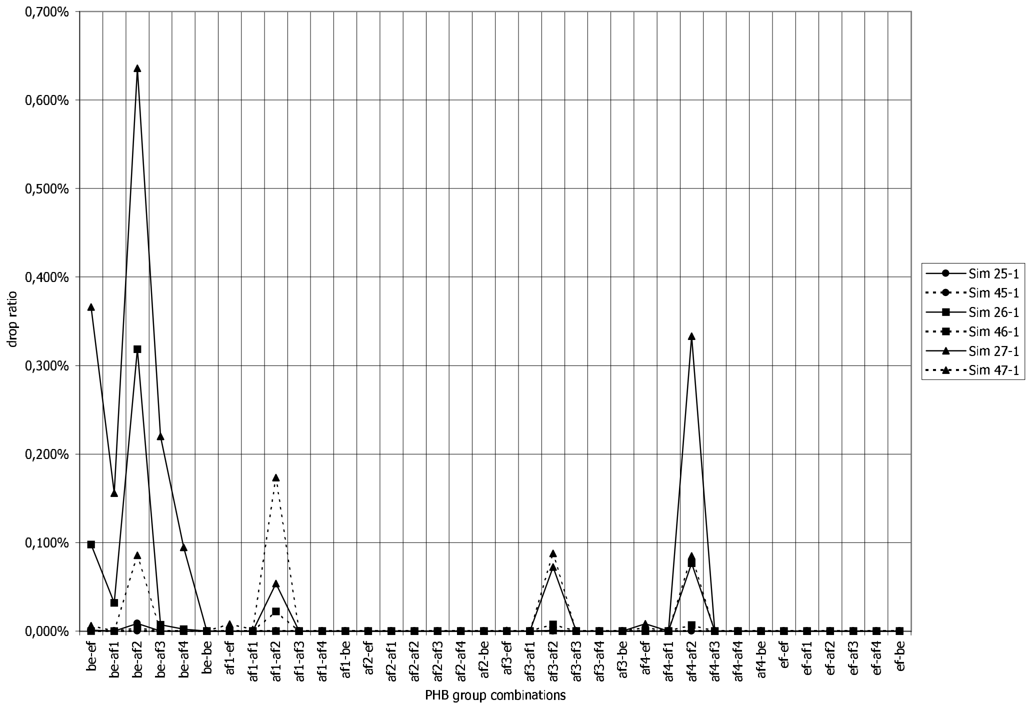


Figure 55: AF2-B Drop Ratio

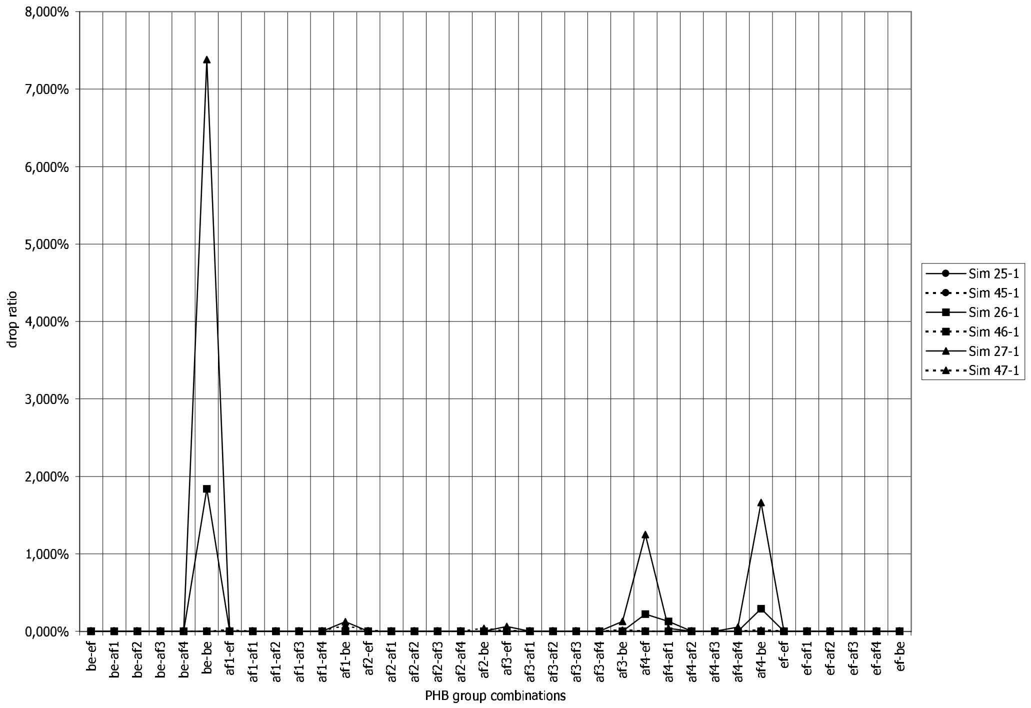


Figure 56: BE-B Drop Ratio

Conclusion

Noise being sent as a higher priority PHB group compared to the AF41 PHB class (used in previous simulations) had no severe impact on PHB groups other than the PHB group in which the noise was sent. These experiments were performed for the WF^{2}Q+ scheduler and it would have been of interest to study if this apply with other scheduling algorithms such as IDRR and DDRR.

The results clearly show that the AF PHB groups (AF1, AF2, AF3 and AF4) all have similar appearance, while EF as well as BE respond differently to noise.

Noise load generally only impacts on total bytes enqueued, delay and drop ratio for the PHB group that is sent as noise from Gateway B. In these cases a higher noise load results in lower performance. It would have been interesting to increase the noise load (say up to 1 Mb/s) for a high priority PHB group, such as EF or AF1. This because the high priority PHB groups usually do not use their resources reserved by the scheduler to full extent, thus allowing lower priority PHB groups to use these excess resources.

Our final question can not be answered because we made a mistake in the statistics gathering as was mentioned earlier in the beginning of this section. Our results indicate, however, an increase in delay for all PHB groups except the EF PHB group as noise was sent with a lower priority PHB group, e.g. the AF3 PHB group has a lower delay when sending noise as AF3 PHB group than when noise is sent as the AF4 PHB class.

4.8 Simulation Study: Color Handling Mode Influence

Description

The key of a DiffServ network is differentiation, but how detailed must the AQM algorithms differentiate? Algorithm complexity goes hand in hand with the level of differentiation. Therefore, in this simulation study we compared two different color handling mode (*CHM*) implementations for 3CRED, namely SAMT and MAMT-higher-priority-inclusive (see section 2.4.3).

The main question to ask is — are the differences when comparing SAMT and MAMT-hp-inclusive of such significance that they make it worth the overhead of MAMT-higher-priority-inclusive compared to SAMT?

In this simulation study two final simulation traffic scenarios, Sim 24 (MAMT-higher-priority-inclusive, MAMT-hpi) and Sim 84 (SAMT), has been used. As mentioned earlier both of them use our extended bottleneck topology (see figure 24) and mean packet size 50.

For parameter settings see tables in section 4.4.3 for Sim 24 and 4.4.6 for Sim 84. This study sends noise using AF41 PHB class on both of the noise paths, *Gateway A* to *Gateway B* and *Gateway B* to *Gateway C*. The noise load is increased in eleven discrete steps of 50 kb/s, ranging from 0 kb/s to 500 kb/s.

Results

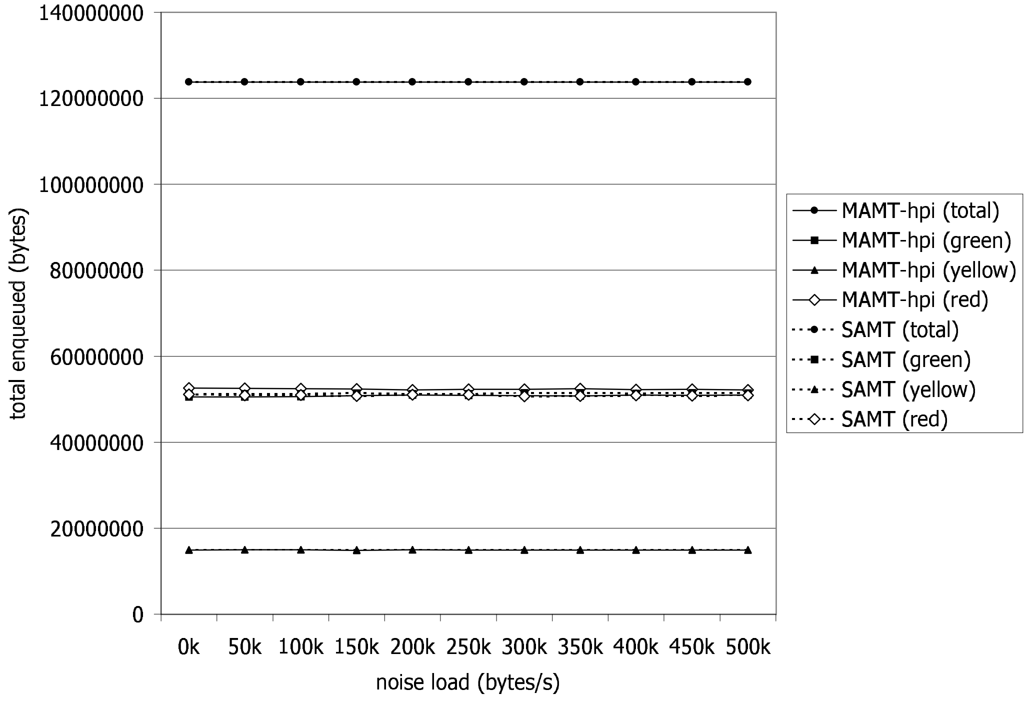


Figure 57: Total Bytes Enqueued

Total bytes enqueued for the two CHMs, MAMT-higher-priority-inclusive and SAMT, can be seen in figure 57. The difference in total bytes enqueued, for noise loads in the interval 0-500 kb/s, was insignificant as it was almost identical for the two, MAMT-hpi and SAMT.

There is no difference in the EF PHB, as usual, between MAMT-hpi and SAMT. It will therefore be omitted from further studying.

On a PHB classes level MAMT-hpi has more bytes enqueued for the AF1x PHB classes, at most ≈4.34 Mb. Summing total bytes enqueued for all AF2x PHB classes MAMT-hpi and SAMT are quite similar with a difference of ≈275 kb at most. SAMT enqueued more bytes for PHB classes AF3x, AF4x and BE namely ≈2.14 Mb, ≈1.62 Mb and ≈863 kb.

The largest difference in total bytes enqueued was 8000 bytes based on all colors. SAMT enqueued more green- and yellow-wise, largest difference was ≈749 kb respectively ≈120 kb. But, MAMT-hpi enqueued most redwise, at most ≈1.57 Mb.

Studying green traffic shows that MAMT-hpi clearly had more bytes enqueued for the AF11 PHB class as the largest difference is ≈518 kb. For the AF21 PHB class the two CHM implementations were quite similar (largest difference ≈310 kb). SAMT was in favor enqueueing PHB classes AF31 and AF41, at most ≈205 kb more than MAMT-hpi for the AF31 PHB class and ≈1.62 Mb more for the AF41 PHB class.

Because no AF41 PHB class traffic is recolored as it aggregates through the network, i.e. arrived bytes for PHB classes AF42 and AF43 are zero, the two will not be studied any further. Total yellow bytes enqueued for all AF PHB classes are quite similar, i.e. nearly the same, between the two. The largest difference for the four yellow AF PHB classes are: ≈29 kb for AF12, ≈38 kb for AF22, ≈125 kb for AF32 and 0 bytes for AF42.

Then finally red traffic, where MAMT-hpi enqueued at most ≈3.89 Mb more of the AF13 PHB class more than SAMT. For PHB classes AF23 and AF33 SAMT enqueued most, at most ≈540 kb respectively ≈2.05 Mb.

Delay for the different PHB classes tends to be stable for the EF PHB class. While the delay for the other PHB classes increases and goes from stable to more increasing as the PHB class receives a lower priority treatment by the scheduler, i.e. from PHB classes AF1x, ..., AF4x to BE PHB class.

The difference ($[MAMT\text{-}hpi] - [SAMT]$) in delay between the two is shown in table 30. It is clear, without doubt, that the delay is lower and better for SAMT than MAMT-hpi for all PHB classes except the "up-and-down" EF PHB class (see figure 58) for which MAMT-hpi is better for noise load 50 kb/s and tends to better from 350 kb/s and up. Delays for the four other PHB classes, AF2x and BE, are shown in figures 59 and 60.

PHB class(es)	Min Delay Difference (ms)	Max Delay Difference (ms)
EF	-0.9068	0,06196
AF1x	12,92121	14,38584
AF2x	38,73975	43,63282
AF3x	39,07374	47,10059
AF4x	17,15455	53,33044
BE	89,22092	2229,95879

Table 30: Network Delay Difference for PHB classes

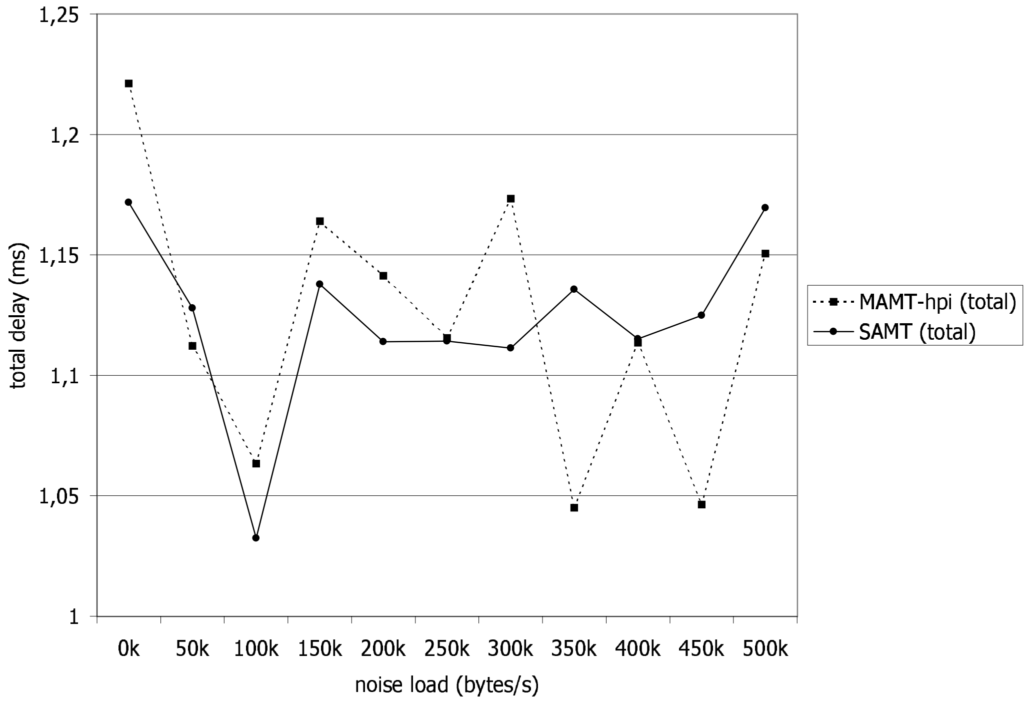


Figure 58: EF Network Delay

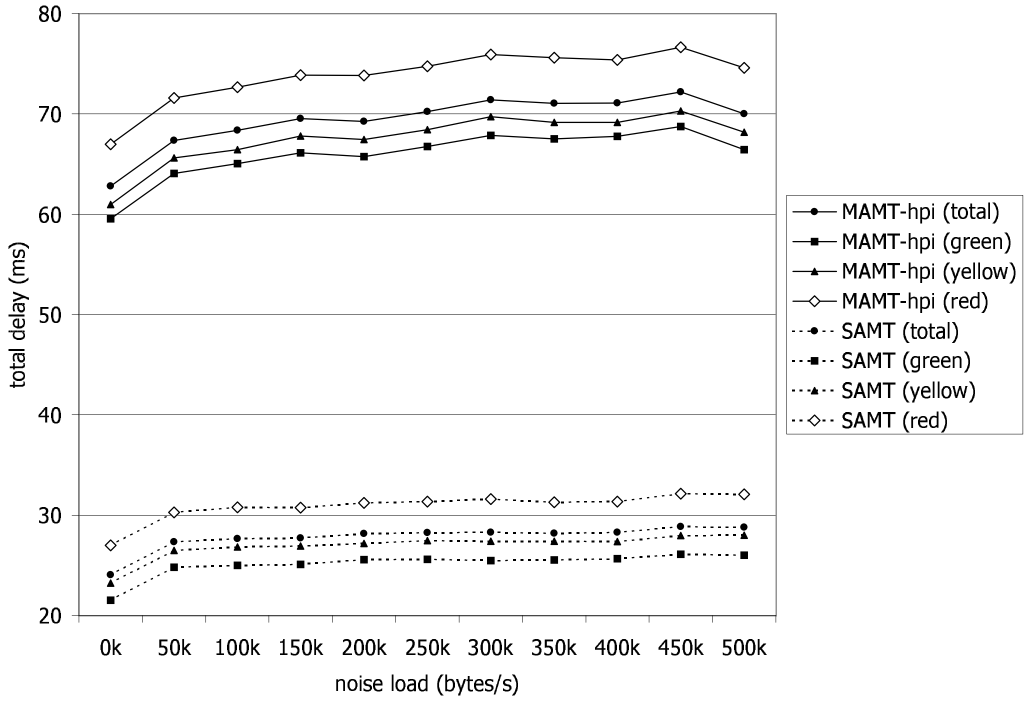


Figure 59: AF2x Network Delay

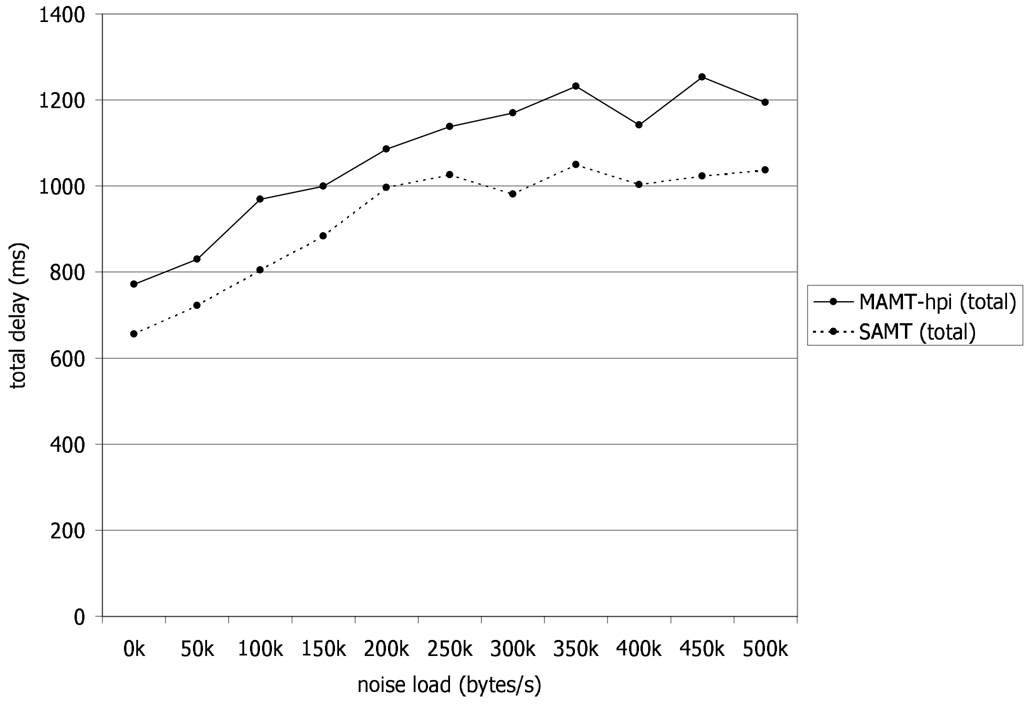


Figure 60: BE Network Delay

As there is no simple way to describe the different drop ratios we will focus completely on the difference between MAMT-hpi and SAMT.

Drop ratio for EF PHB class is zero as all arrived packets where enqueued. PHB classes AF1x, AF2x and AF3x tends to be stable on Gateway A and, green and yellow traffic, experience a lower drop ratio for MAMT-hpi than SAMT. While SAMT has a lower drop ratio than MAMT-hpi for red traffic. On Gateway B the difference in drop ratio for PHB classes AF1x, AF2x and AF3x is no higher than 0,015% for MAMT-hpi and SAMT and therefore insignificant. MAMT-hpi has a higher drop ratio for PHB class AF4x on both Gateway A and B from noise load 350 kb/s and up (similar up to 350 kb/s). Finally BE PHB class, where MAMT has the higher drop ratio on Gateway A and vice versa for Gateway B.

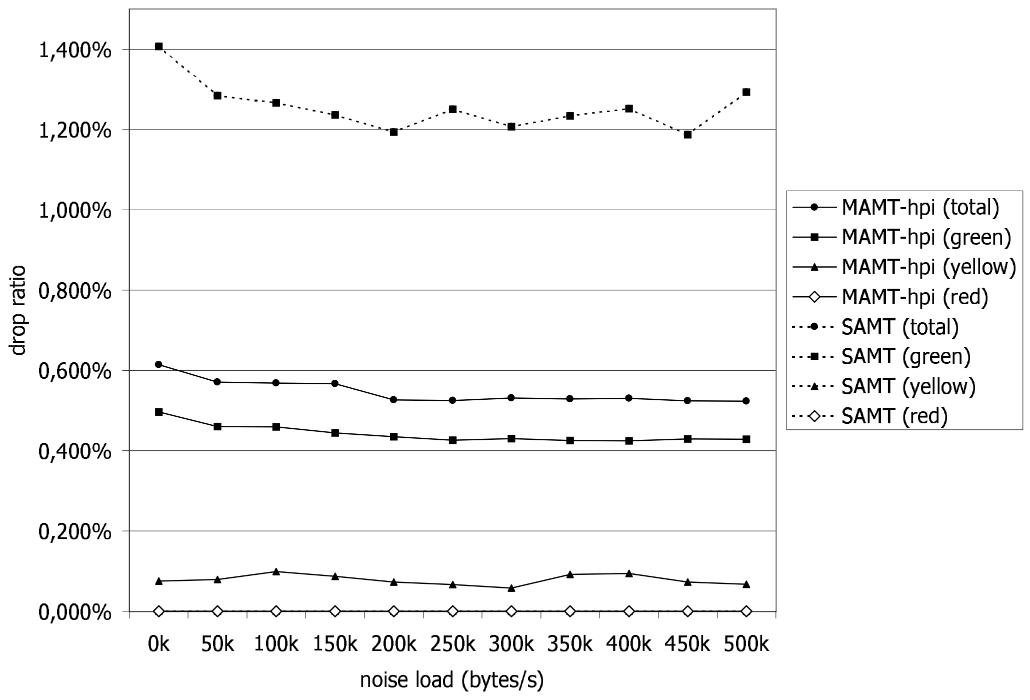


Figure 61: AF2x-A Drop Ratio

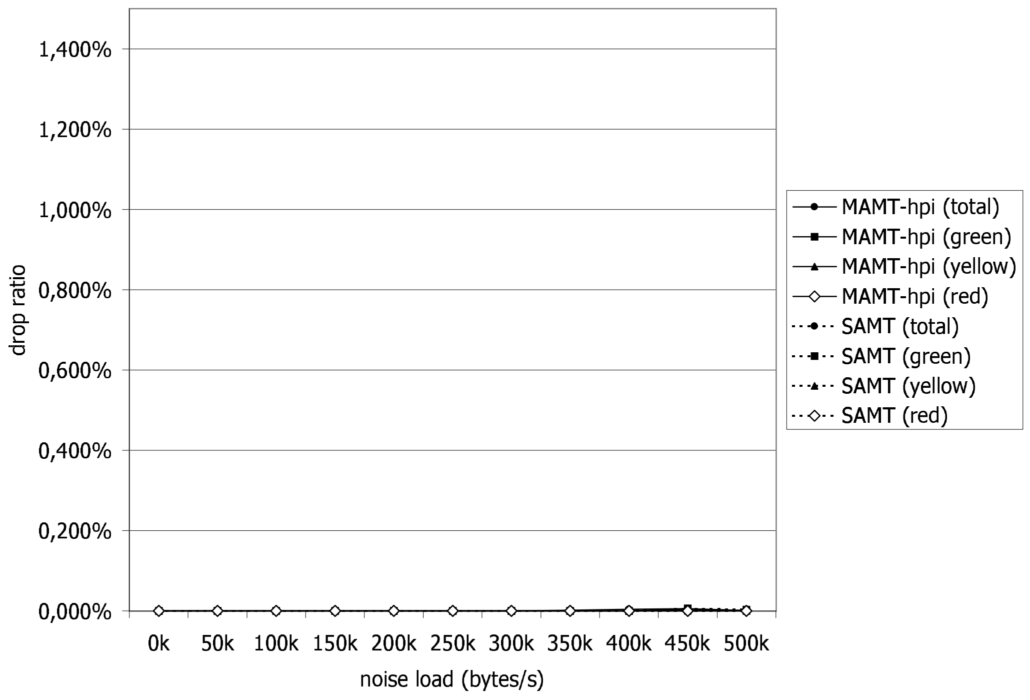


Figure 62: AF2x-B Drop Ratio

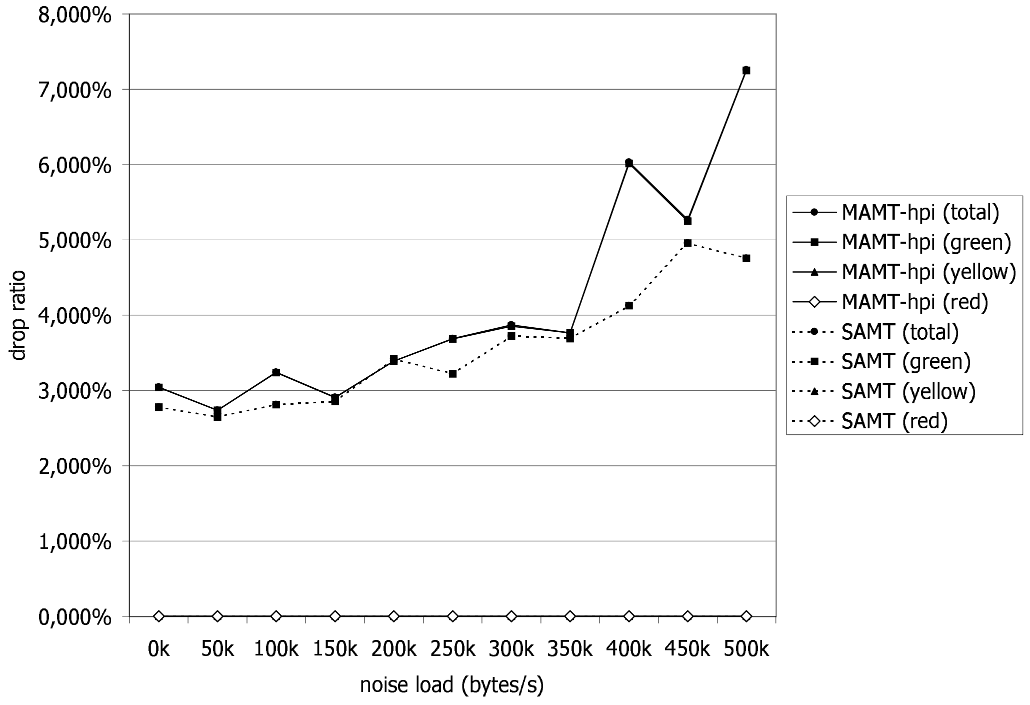


Figure 63: AF4x-A Drop Ratio

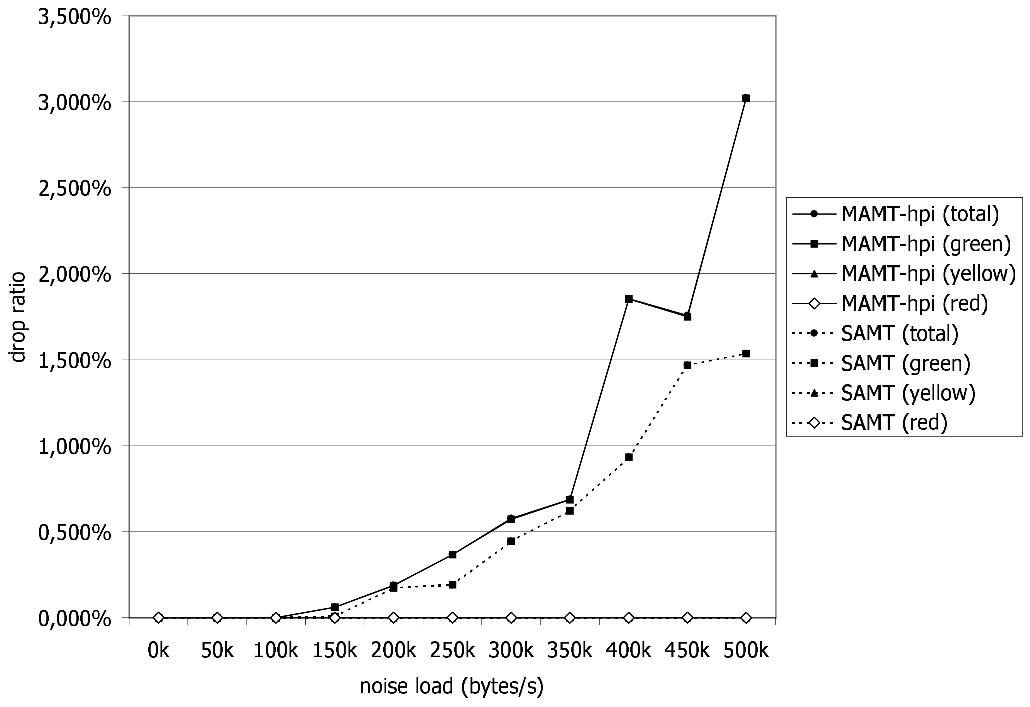


Figure 64: AF4x-B Drop Ratio

Conclusion

MAMT-hpi and SAMT both enqueue about the same if we do not pay attention to PHB groups or colors. MAMT-hpi favours high priority traffic as it enques mostly AF1x PHB classes traffic, even though much of it is red (i.e. PHB class AF13). While SAMT spreads resources over PHB classes in a greater extent than MAMT-hpi. This is also verified by drop ratio results since MAMT-hpi had a lower drop ratio for the all green AF PHB classes except PHB class AF41.

For reasons w do not know delay is much better for SAMT than for MAMT-hpi. It would have been interesting to study the cause of the big difference in delay, but also here we were under time constraints, and had to leave the issue as a subject for further study.

We do not have a “MAMT-hpi is (always) better than SAMT” conclusion or vice versa. The inital question that sought an answer was if the performance of MAMT-hpi compared to SAMT made up for the overhead of MAMT-hpi. The overhead is most likely one of the reasons for MAMT-hpi’s high delay.

However, for us to answer the question is totally dependent of whether the delay is of high importance — if so, use SAMT. Otherwise, if it depends more on other AQM algorithm characteristics, e.g. total bytes enqueued and fairness between groups and/or classes, use the one that is most suitable based on SLS, network load, recoloring etc.

4.9 Simulation Study: Scheduler Influence

Description

A scheduler is a major component of a network router and since they all operate a bit differently, they may cause rather different behaviors, but some attributes are more important than others. It is the difference in these attributes, mainly flow fairness at optimal efficiency and starvation prevention, that we wish to study. The schedulers that will be evaluated here are DRR, DRR/WRR, IDRR, DDRR, WF^{2}Q+. The main questions which we sought answers to were:

- How does various scheduling algorithms affect total enqueued bytes, drop ratio and queueing delay both totally and individually for each PHB class?
- Do schedulers maintain fairness even during times of heavy network congestion?
- Do all schedulers protect flows from starvation?
- If traffic already is evenly distributed when arriving at the scheduler, is it sufficient to use a *Strict Priority* (SP) scheduler?

This simulation uses our extended bottleneck topology (see figure 24) with the simulation dependent parameters in tables 22, 23 and 24. Mean packet size is set to 50.

Results

Before the simulation we predicted that SP would give results that had to be viewed on their own. It is simply not realistic to compare SP against any other scheduler when the traffic is distributed as it is in this simulation. However, if a classifier divides traffic after importance to priority levels, the SP scheduler might still be useful.

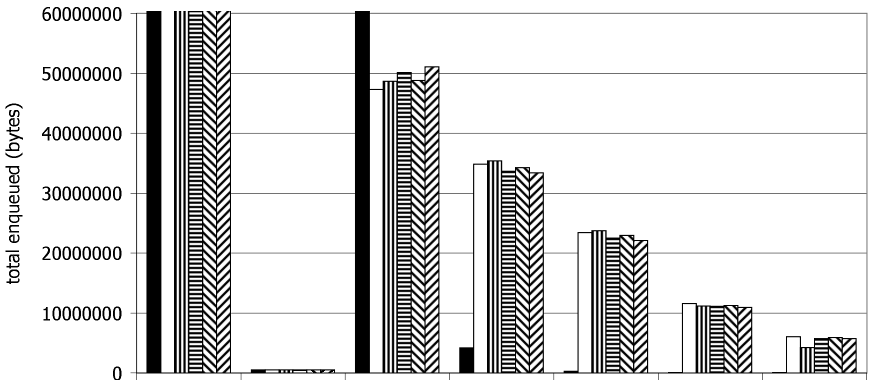
To begin with, figure 65 shows the total bytes enqueued for the PHB classes and the summation of them.

If we disregard the SP scheduler statistics and compare the total bytes enqueued for the other schedulers we can see in table 31 that the maximum difference is approximately 100 kb, which is insignificant when compared to the total bytes enqueued. The differences found between the PHB classes are larger, but still not large enough to conclude that any scheduler is better than another.

Scheduler	Total Bytes Enqueued
DRR	123636450
DRR/WRR	123652800
IDRR	123637700
DDRR	123637250
WF2Q+	123637650
Difference	95450

Table 31: Difference in Total Bytes Enqueued

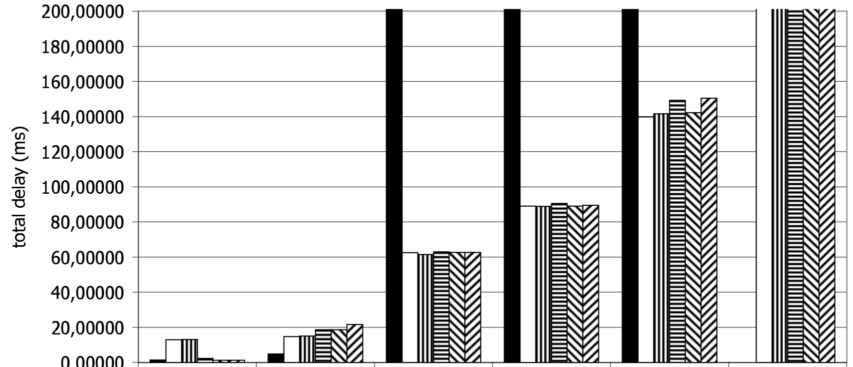
Instead we study the average queueing delay, i.e. the time spent in queues along the network path. As can be seen in figure 66 the delay for the high priority classes with SP is minimal, but already for the AF2 PHB group the delay has grown huge when compared against the other schedulers. It is hard to judge how efficient a classifier would be in distributing the traffic since we do not have a classifier implementation. This would be a useful study to make in an extension of this master thesis.



	Total	EF	AF1	AF2	AF3	AF4	BE
■ SP	123731900	495000	118759200	4164850	263250	22800	26800
□ DRR	123636450	495000	47288350	34851250	23407400	11581550	6012900
▥ DRR/WRR	123652800	495000	48670600	35393000	23704850	11152600	4236750
▤ IDRR	123637700	495000	50128800	33672700	22493350	11111700	5736150
▧ DDRR	123637250	495000	48783950	34256150	22949900	11251150	5901100
▨ WF2Q+	123637650	495000	51075550	33362750	22088550	10922750	5693050

PHB Class

Figure 65: Total Bytes Enqueued



	EF	AF1	AF2	AF3	AF4	BE
■ SP	1,29524	4,72154	475,31458	11436,90748	15568,66055	0,00000
□ DRR	12,83713	14,67068	62,46966	88,95798	139,83802	718,47967
▥ DRR/WRR	12,96271	15,00480	61,35403	88,87684	141,59351	1089,90518
▤ IDRR	2,28166	18,51990	62,97366	90,46408	149,16409	761,41629
▧ DDRR	1,18259	18,61225	62,61366	89,05200	142,24659	720,77533
▨ WF2Q+	1,18728	21,58654	62,69292	89,38990	150,44957	762,40395

PHB class

Figure 66: Network Delay

When comparing DRR/WRR delay with the other schedulers we observed that it has about 350ms higher delay for the BE PHB class. Due to time constraints we have not been able to study this phenomenon. But since DRR/WRR mode is very similar to the DRR scheduler, they should exhibit the same characteristics.

The delay for the EF PHB class for the DRR and DRR/WRR schedulers part from the other schedulers. This is because both these implementations are non-interleaved (see section 2.5.3) causing packets from PHB classes to be sent in groups. This results in a high delay for the high priority EF PHB class which relates to fairness as interleaving makes a scheduler more fair seen from a short perspective.

Since the DRR and DRR/WRR schedulers cause this high delay on the important EF PHB class, we only study the IDRR, DDRR and WF^{2}Q+ schedulers in more detail.

In figure 66 it appears as if the DDRR scheduler gives slightly better overall delay performance, except for the EF PHB class, than the WF^{2}Q+ scheduling algorithm. Also the IDRR scheduler gives better performance than WF^{2}Q+ in some PHB groups such as AF1 and BE.

The delay oscillation (stability) is of concern when studying scheduling algorithms. In figure 67 and 68 a comparison of the various packet delays seen in a shorter perspective (from time 20-22) for the AF4 PHB can be observed for the IDRR, DDRR and WF^{2}Q schedulers. In figure 67 it can be seen that the WF^{2}Q+ delay for the AF3 PHB group is very stable compared to the IDRR and DDRR delays. The AF3x PHB classes is an exception. In the delay graphs for all other AF classes and BE the delay is not as stable as for the AF3x PHB classes. An example of the delay for the AF4 PHB group can be seen in figure 68.

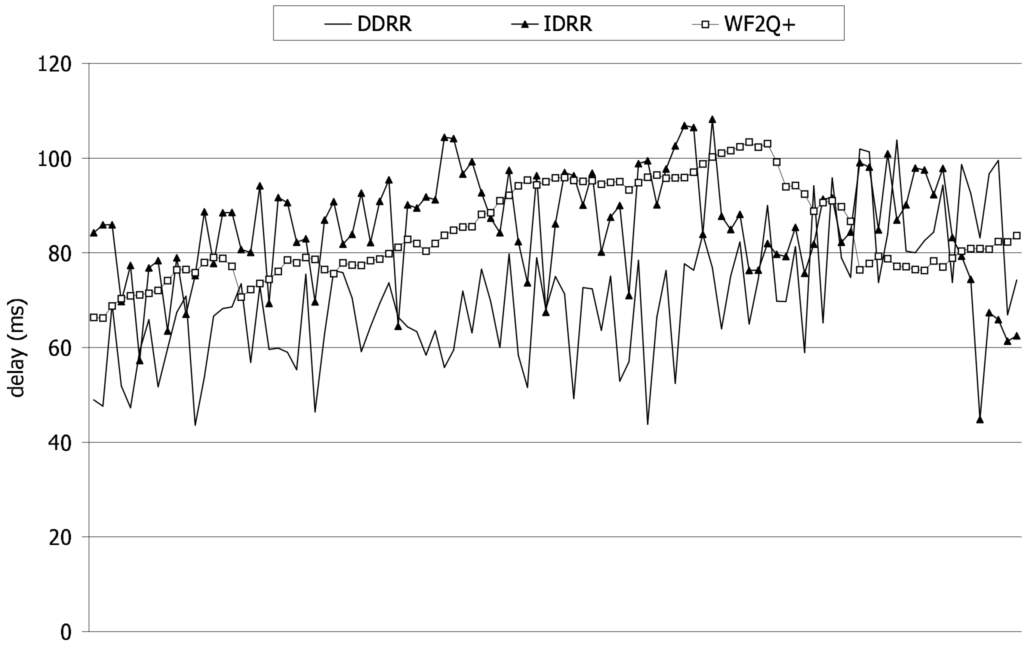


Figure 67: AF3 Network Delays

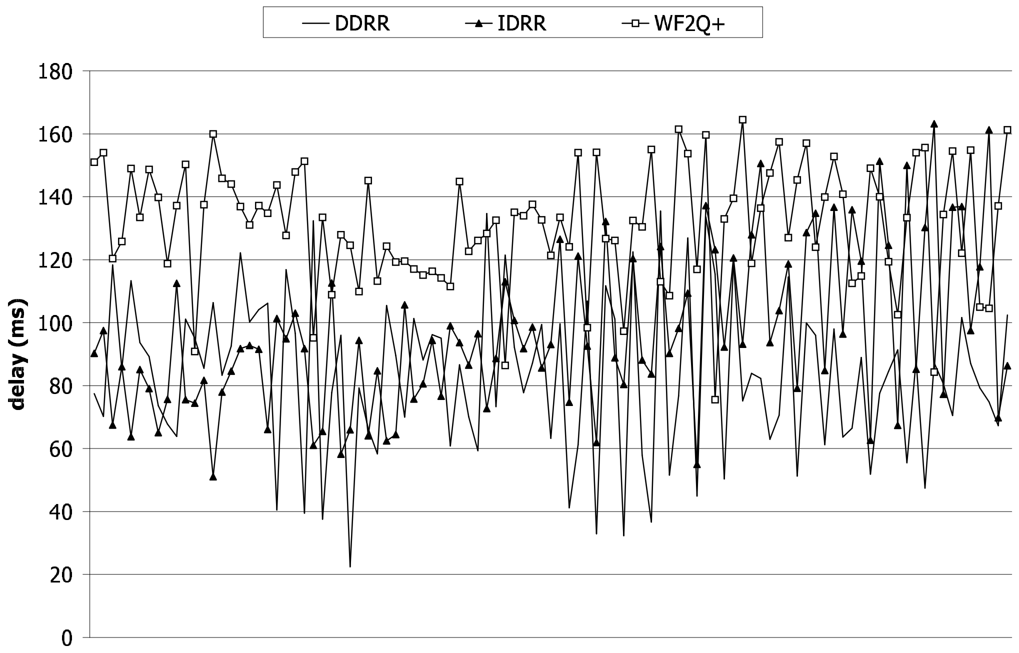


Figure 68: AF4 Network Delays

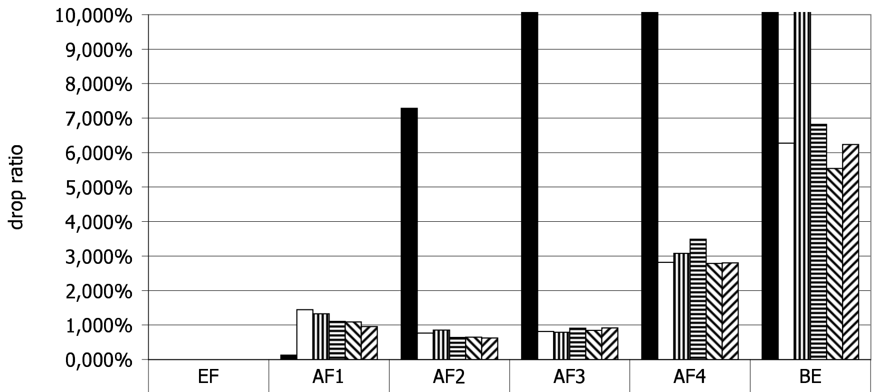
The following table displays the difference between the max and min delays for the different schedulers and PHBs. In this aspect the WF^{2}Q+ scheduler is superior towards IDRR and DDRR except for the AF2 PHB group.

PHB Group	IDRR	DDRR	WF^{2}Q+
AF1	15,055 ms	13,887 ms	10,397 ms
AF2	27,864 ms	31,847 ms	37,680 ms
AF3	64,492 ms	63,462 ms	47,365 ms
AF4	138,177 ms	118,596 ms	98,116 ms
BE	602,335 ms	548,341 ms	511,830 ms

Table 32: Difference in Network Delay

The question that remains to be asked regarding delay, is if it is worth using WF^{2}Q+ with the added complexity it gives. The delay graph may be more even for the AF3 PHB class, but it costs more for each packet to be processed in the scheduler. It is hard for us to judge the actual costs and benefits using WF^{2}Q+ against any other scheduling algorithm, but we feel that in this simulation environment and topology DDRR and IDRR gives better performance.

Figure 69 shows the drop ratio percentage for the different PHB classes. The DRR/WRR scheduler exhibits the same characteristics for the BE PHB delay. This drop ratio is much larger than for the other PHBs, but besides from that everything is very even.



	EF	AF1	AF2	AF3	AF4	BE
■ SP	0,000%	0,128%	7,282%	44,794%	90,396%	89,692%
□ DRR	0,000%	1,446%	0,767%	0,812%	2,816%	6,276%
▥ DRR/WRR	0,000%	1,324%	0,852%	0,793%	3,082%	10,625%
▤ IDRR	0,000%	1,106%	0,641%	0,909%	3,490%	6,822%
▧ DDRR	0,000%	1,091%	0,644%	0,847%	2,784%	5,544%
▨ WF2Q+	0,000%	0,954%	0,622%	0,918%	2,804%	6,236%

PHB class

Figure 69: Network Drop Ratio

Conclusion

There are so many factors to take into consideration before selecting a scheduler that stands out from the rest. A conclusion that can be drawn however is that interleaved schedulers are more fair than schedulers that are non-interleaved. This mainly because the delay caused to the EF PHB class is simply unacceptable for a high priority class such as EF.

Before the simulations were made, we thought that the WF^{2}Q+ delay graphs would tend to be even and not oscillate very much. This was indeed wrong with the exception for the AF3 PHB group. In this topology and its settings DDRR performed generally as good as WF^{2}Q+ did. Other topologies and configurations may have given us other results, but for this simulation we judge that DDRR would be a better choice than WF^{2}Q+ because WF^{2}Q+ has a greater time complexity than DDRR.

The comparison of delay between IDRR and DDRR was almost the same as well, but DDRR gave slightly better performance than IDRR.

As mentioned previously, a study that would be interesting to do, is to try out a more intelligent classifier/dropper block in combination with a simple SP scheduler, but this have not been done in this master thesis.

5 Summary

The dream QoS architecture would be something like Integrated Services but with all overhead excluded. The Differentiated Services Architecture is good, but we feel it is merely a temporary solution to Quality of Service in the Internet. There will surely be other solutions in the future improving Quality of Service while minimizing overhead. Since it is expensive to globally change the network architecture, we feel it is important that prior to its global deployment test its performance using smaller networks and simulations.

Our work was originally supposed to study newer scheduling and active queue management algorithms more in depth, but implementation additions and especially bug fixes of Sean Murphy's DS-LIB (see section 3.2) took more time than we expected. Also, there was also a lot of "side work" involved in writing Perl scripts that could handle simulations and their data.

There was, and still is, a major uncertainty regarding creation of topologies and network settings (see section 4.3). This due to the large amount of parameters, the interaction between them and the need for them to be realistic.

Nevertheless, we feel that this master thesis has inspired future studies in this area. As can be seen from the conclusions stated in the previous chapter, there are several issues that deserve futher research. Some possible extensions of our simulation studies are:

- Would our results have changed much with other (larger) topologies?
- Would separating the noise traffic from other traffic, i.e. it having its own PHB class, had given us significantly different results?
- Why does MAMT-hpi have a much higher delay than SAMT?
- Do scheduling algorithms other than WF^{2}Q+ handle noise equally well?

Finally, this master thesis was not primarily intended to make any simulator environment recommendations for SwitchCore. However, since we at present-day are the two persons at SwitchCore with most knowledge about the NS environment, we have some comments concerning it. Firstly, NS is operated on per-packet level and not chip/cell level which SwitchCore usually run their simulations on. Thus, there is a discrepancy in the level of details that should be addressed. The major advantage of the NS environment is that it is open source and therefore is continuously updated (both bug fixes and algorithm additions). The disadvantage, however, is that one must master the huge NS library, TCL and C++, and the connection between the two programming languages.

Appendices

A Acronyms

3CRED	Three Colored Random Early Detection
AF	Assured Forwarding PHB
ARED	Adaptive Random Early Detection
BA	Behavior Aggregate
BE	Best Effort PHB
BRED	Balanced Random Early Detection
CHM	Color Handling Mode
CS	Class Selector PHB *or* Complete Sharing
CU	Currently Unused bits in the IP header
DDRR	Deficit Driven Round Robin
DiffServ	Differentiated Services
DML	Divided Memory Link
DMQ	Divided Memory Queue
DRR	Deficit Round Robin
DS	Differentiated Services
DS-3CRED	Implementation of 3CRED in DS-LIB
DS-RED	Implementation of RED in DS-LIB
DS-LIB	Differentiated Services Library additon to NS (originally by Sean Murphy)
DS3CRED	Implementation of 3CRED in DS-LIB
DSCP	Differentiated Services CodePoint
EF	Expedited Forwarding PHB
EFD	Expedited Forwarding with Drop PHB
EWMA	Exponential Weighted Moving Average
FIFO	First In First Out
FQ	Fair Queueing
FRED	Flow Random Early Detection
FTP	File Transfer Protocol
GPS	Generalized Processor Sharing
IDRR	Interleaved Deficit Round Robin
IntServ	Integrated Services
IO	Interoperability PHB
IP	Internet Protocol
IPv4	Internet Protocol Version 4
IPv6	Internet Protocol Version 6
ISP	Internet Service Provider
LAN	Local Area Network
LBE	Lower than Best Effort PHB
MAMT	Multiple Accounting, Multiple Threshold
MAST	Multiple Accounting, Single Threshold
MF	Multi-Field Classifier
MPS	Mean Packet Size
NAM	Network Animator
NS	Network Simulator
NS-RED	Implementation of RED in NS

PHB	Per-Hop-Behavior
RED	Random Early Detection
RFC	Request For Comments
RIO	RED In/Out
RR	Round Robin
QoS	Quality of Service
SAMT	Single Accounting, Multiple Threshold
SAST	Single Accounting, Single Threshold
SLS	Service Level Specification
SM	Shared Memory
SP	Strict Priority
SRED	Stabilized Random Early Detection
SRTCM	Single Rate Three Color Marker
TB	Token Bucket
TCP	Transmission Control Protocol
TCP/IP	Common combination of TCP/IP
ToS	Type of Service, IP header field
TRTCM	Two Rate Three Color Marker
UDP	User Data Protocol
WF2Q+	Worst Case Weighted Fair Queueing+
WFQ	Weighted Fair Queueing
WRR	Weighted Round Robin
WWW	World Wide Web

B Terminology

Parts of this terminology section was taken from the RFC document *An architecture for Differentiated Services* [10].

Behavior Aggregate (BA)	a DS behavior aggregate.
BA classifier	a classifier that selects packets based only on the contents of the DS field.
Boundary link	a link connecting the edge nodes of two domains.
Classifier	an entity which selects packets based on the content of packet headers according to defined rules.
DS behavior aggregate	a collection of packets with the same DS codepoint crossing a link in a particular direction.
DS boundary node	a DS node that connects one DS domain to a node either in another DS domain or in a domain that is not DS-capable.
DS-capable	capable of implementing differentiated services as described in this architecture; usually used in reference to a domain consisting of DS-compliant nodes.
DS codepoint	a specific value of the DSCP portion of the DS field, used to select a PHB.
DS-compliant	enabled to support differentiated services functions and behaviors as defined in [36], this document, and other differentiated services documents; usually used in reference to a node or device.
DS domain	a DS-capable domain; a contiguous set of nodes which operate with a common set of service provisioning policies and PHB definitions.
DS egress node	a DS boundary node in its role in handling traffic as it leaves a DS domain.
DS ingress node	a DS boundary node in its role in handling traffic as it enters a DS domain.
DS interior node	a DS node that is not a DS boundary node.

DS field	the IPv4 header ToS octet or the IPv6 Traffic Class octet when interpreted in conformance with the definition given in [36]. The bits of the DSCP field encode the DS codepoint, while the remaining bits are currently unused.
DS node	a DS-compliant node.
DS region	a set of contiguous DS domains which can offer differentiated services over paths across those DS domains.
Downstream DS domain	the DS domain downstream of traffic flow on a boundary link.
Dropper	a device that performs dropping.
Dropping	the process of discarding packets based on specified rules; policing.
Marker	a device that performs marking.
Marking	the process of setting the DS codepoint in a packet based on defined rules; premarking, remarking.
Mechanism	a specific algorithm or operation (e.g., queuing discipline) that is implemented in a node to realize a set of one or more per-hop behaviors.
Meter	a device that performs metering.
Metering	the process of measuring the temporal properties (e.g., rate) of a traffic stream selected by a classifier. The instantaneous state of this process may be used to affect the operation of a marker, shaper, or dropper, and/or may be used for accounting and measurement purposes.
Micro flow	a single instance of an application to application flow of packets which is identified by source address, source port, destination address, destination port and protocol id.
MF Classifier	a multi-field (MF) classifier which selects packets based on the content of some arbitrary number of header fields; typically some combination of source address, destination address, DS field, protocol ID, source port and destination port.
Per-Hop-Behavior	the externally observable forwarding behavior

(PHB)	applied at a DS-compliant node to a DS behavior aggregate.
PHB group	a set of one or more PHBs that can only be meaningfully specified and implemented simultaneously, due to a common constraint applying to all PHBs in the set such as a queue servicing or queue management policy. A PHB group provides a service building block that allows a set of related forwarding behaviors to be specified together (e.g., four dropping priorities). A single PHB is a special case of a PHB group.
Policing	the process of discarding packets (by a dropper) within a traffic stream in accordance with the state of a corresponding meter enforcing a traffic profile.
Premark	to set the DS codepoint of a packet prior to entry into a downstream DS domain.
Provider DS domain	the DS-capable provider of services to a source domain.
Remark	to change the DS codepoint of a packet, usually performed by a marker in accordance with a TCA.
Service	the overall treatment of a defined subset of a customer's traffic within a DS domain or end-to-end.
Service Level Specification (SLS)	a set of parameters and their values which together define the service offered to a traffic stream by a DS domain.
Service Provisioning Policy	a policy which defines how traffic conditioners are configured on DS boundary nodes and how traffic streams are mapped to DS behavior aggregates to achieve a range of services.
Shaper	a device that performs shaping.
Shaping	the process of delaying packets within a traffic stream to cause it to conform to some defined traffic profile.
Source domain	a domain which contains the node(s) originating the traffic receiving a particular service.

Traffic conditioner	an entity which performs traffic conditioning functions and which may contain meters, markers, droppers, and shapers. Traffic conditioners are typically deployed in DS boundary nodes only. A traffic conditioner may remark a traffic stream or may discard or shape packets to alter the temporal characteristics of the stream and bring it into compliance with a traffic profile.
Traffic conditioning	control functions performed to enforce rules specified in a TCA, including metering, marking, shaping, and policing.
Traffic Conditioning Agreement (TCA)	an agreement specifying classifier rules and any corresponding traffic profiles and meter ing, marking, discarding and/or shaping rules which are to apply to the traffic streams selected by the classifier. A TCA encompasses all of the traffic conditioning rules explicitly specified within a SLA along with all of the rules implicit from the relevant service requirements and/or from a DS domain's service provisioning policy.
Traffic profile	a description of the temporal properties of a traffic stream such as rate and burst size.
Traffic stream	an administratively significant set of one or more micro flows which traverse a path segment. A traffic stream may consist of the set of active micro flows which are selected by a particular classifier.
Upstream DS domain	the DS domain upstream of traffic flow on a boundary link.

C Simulation Results

C.1 Random Early Detection (RED)

C.1.1 Simulation 1

Type: Verification

Common parameters		
Simulation time	1 s	
Scheduler used	WRR	
Packet Size	1000 B	

TCP Source	Window size	Start Time
Node 1	33	0
Node 2	67	0.2
Node 3	112	0.4
Node 4	78	0.6

Links between Nodes	Bandwidth	Delay
Node 1 ↔ Gateway	100 Mb	1 ms
Node 2 ↔ Gateway	100 Mb	4 ms
Node 3 ↔ Gateway	100 Mb	8 ms
Node 4 ↔ Gateway	100 Mb	5 ms
Gateway ↔ Sink	45 Mb	2 ms

RED Parameters on Link Gateway ↔ Sink	
Queue Buffer Limit	Unlimited
Queue Weight	0.002
Max Drop Prob	50
Min Thresh	5
Max Thresh	15
NS-RED wait_	false

Table 33: RED Simulation 1 Parameters

Topology

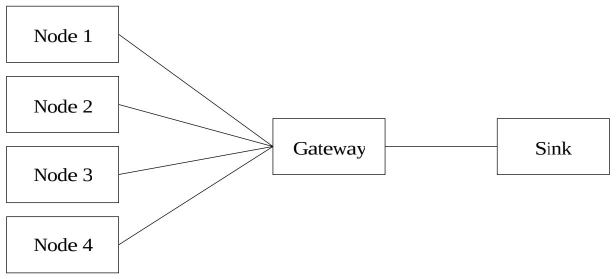


Figure 70: Verification Simulations Topology

The simulation topology consists of 4 TCP/Tahoe senders which always (whenever the TCP window allows them too) sends maximal sized FTP packets to a gateway. The Gateway is equipped with RED functionality, and forwards the packets to the destination sink.

Simulation Variants:

1. Packet mode Active & ECN Inactive

2. Byte mode Active & ECN Inactive

3. Byte mode Active & ECN Active

Description

The purpose of this simulation was to verify that the implementation of Sean Murphy's RED-algorithm (called DS-RED from now on) was correct. There is already an implementation of RED in ns (from now on we will call it NS-RED). In NS-RED there exist a parameter called *edv_.old*. This parameter effects when a drop should be made. It specifically has to do with when we should drop when crossing a threshold. Since we did not want this feature in DS-RED, we disabled this in the NS-RED version.

We did 500 simulations on each of the simulation variants mentioned above with different fixed seeds. The output from the simulator were in the form of trace-files, describing every event that happened during the simulation. We compared the output from the DS-RED simulation and the NS-RED simulation hoping that they would be exact matches. In all but two of the simulations the trace files were matches. The difference in the trace-files were in the form

DS-RED		NS-RED	
r	packet information	r	packet information
+	packet information		
-	packet information		

Table 34: NS-RED and DS-RED trace file differences

Above *r* denote a receive packet event, *+* an enqueue event, and - a dequeue event.

Conclusion

Even though we found differences in 2 of the 500 simulations we conclude that DS-RED is verified, and is implemented correctly. We had to do several major bug fixes to Murphy's implementation to get it verified. More about these bug fixes and implementation specifics can be found in section 3.2.

The differences we found in 2 of the 500 simulations can be explained by the use of different schedulers, where one of them finds time to issue another packet.

C.1.2 Simulation 2

Type: Verification

Topology

The simulation uses the same topology as previous simulations (see figure 70). The only thing that has changed it the window sizes for the TCP sources and the bandwidth/delay on the bottleneck-link between the gateway and the sink.

Common parameters		
Simulation time	1 s	
Scheduler used	WRR	
Packet Size	1000 B	
TCP Source	**Window size**	**Start Time**
Node 1	31	0
Node 2	50	0.2
Node 3	75	0.4
Node 4	56	0.6
Links between Nodes	**Bandwidth**	**Delay**
Node 1 $\leftrightarrow$ Gateway	100 Mb	1 ms
Node 2 $\leftrightarrow$ Gateway	100 Mb	4 ms
Node 3 $\leftrightarrow$ Gateway	100 Mb	8 ms
Node 4 $\leftrightarrow$ Gateway	100 Mb	5 ms
Gateway $\leftrightarrow$ Sink	25 Mb	4 ms
RED Parameters on Link Gateway $\leftrightarrow$ Sink		
Queue Buffer Limit	Unlimited	
Queue Weight	0.002	
Max Drop Prob	50	
Min Thresh	5	
Max Thresh	15	
NS-RED wait_	false	

Table 35: RED Simulation 2 Parameters

Simulation Variants:

1. **Packet mode Active & ECN Inactive**

2. **Byte mode Active & ECN Inactive**

3. **Byte mode Active & ECN Active**

Description

Here we did 50 simulations on each of the simulation variants mentioned above with different fixed seeds. Trace-files from NS-RED and DS-RED were identical in all different cases.

Conclusion

Since all trace-files were exact matches we conclude this verification successful.

C.1.3 Simulation 3

Type: Verification

Common parameters		
Simulation time	1 s	
Scheduler used	WRR	
Packet Size	1000 B	

TCP Source	Window size	Start Time
Node 1	31	0
Node 2	50	0.2
Node 3	75	0.4
Node 4	56	0.6

Links between Nodes	Bandwidth	Delay
Node 1 ↔ Gateway	100 Mb	1 ms
Node 2 ↔ Gateway	100 Mb	4 ms
Node 3 ↔ Gateway	100 Mb	8 ms
Node 4 ↔ Gateway	100 Mb	5 ms
Gateway ↔ Sink	25 Mb	4 ms

RED Parameters on Link Gateway ↔ Sink		
Queue Buffer Limit	20	
Queue Weight	0.002	
Max Drop Prob	50	
Min Thresh	5	
Max Thresh	15	
NS-RED wait_	false	

Table 36: RED Simulation 3 Parameters

Topology

The simulation uses the same topology as in *Simulation 2* (figure 70). The only thing that has changed from *Simulation 2* is that we now have limited buffer space in the RED gateway.

Simulation Variants:

1. Packet mode Active & ECN Inactive

2. Byte mode Active & ECN Inactive

3. Byte mode Active & ECN Active

Description

We did 50 simulations on each of the simulation variants mentioned above with different fixed seeds. All trace-files matched completely.

Conclusion

Our implementation of DS-RED works under more realistic conditions, such as limited buffer space. This simulation concludes our verification of DS-RED.

C.2 Random Early Detection In/Out (RIO)

C.2.1 Simulation 1

Type: Verification

Common parameters		
Simulation time	1 s	
Scheduler used	WRR	
Packet Size	1000 B	

TCP Source	Window size	Start Time
Node 1	33	0
Node 2	67	0.2
Node 3	112	0.4
Node 4	78	0.6

Links between Nodes	Bandwidth	Delay
Node 1 ↔ Gateway	100 Mb	1 ms
Node 2 ↔ Gateway	100 Mb	4 ms
Node 3 ↔ Gateway	100 Mb	8 ms
Node 4 ↔ Gateway	100 Mb	5 ms
Gateway ↔ Sink	45 Mb	2 ms

RED Parameters on Link Gateway ↔ Sink		
Queue Buffer Limit	Specified Later	
Queue Weight	0.002	
Max Drop Prob	50	
Min Thresh	5	
Max Thresh	15	
wait_	false	

RIO Parameters on Link Gateway ↔ Sink		
Queue Buffer Limit	Specified Later	
Queue Weight	0.002	
In Max Drop Prob	50	
In Min Thresh	5	
In Max Thresh	15	
Out Max Drop Prob	50	
Out Min Thresh	5	
Out Max Thresh	15	
wait_	false	

Table 37: RIO Simulation 1 Parameters

Topology

The simulation uses the same topology as previous simulations (see figure 70).

Simulation Variants:

1. Packet mode Active & ECN Inactive

Queue Buffer Limit	Unlimited

2. Byte mode Active & ECN Inactive

Queue Buffer Limit	Unlimited

3. Byte mode Active & ECN Active

Queue Buffer Limit	Unlimited

4. Packet mode Active & ECN Inactive

Queue Buffer Limit	20

Description

The purpose of this simulation was to verify that a contributed RIO, found on the ns-page [6]. Since we do not have an implemented reference model of RIO, we use NS-RED (once again) to verify that the RIO do not mix the colors up, i.e we run RIO with in/out thresholds set to the same as the RED thresholds. We also only send either IN or OUT packets in this simulation. The token bucket was also disabled in this simulation, therefore packets that are out of profile will not be remarked.

The *edv_.old* parameter mentioned in section C.1 also exist in this contributed implementation of RIO. This parameter was disabled in both NS-RED and the contributed RIO (from now on called NS-RIO since the structure of the two implementations look alike).

We did 25 simulations on each of the simulation variants mentioned above with different fixed seeds sending IN packets only. When comparing the NS-RIO traces with the NS-RED traces they were exact matches.

After this we did the same sets of simulations but replacing IN packets with OUT packets. This did not give us satisfying results. After a while we found the reason for our problem. A TCL binding to *out_max_p_inv* did not exist, and therefore this value got random values to whatever the C++ compiler initialized them too. This "bug" disabled the whole idea with RIO (and RED), i.e dropping packets randomly to prevent synchronization problems with TCP senders. This "bug" will have had serious implications on any studies using this contributed piece of RIO code. More about bug fixes can be found in section 3.2.

When the bug fix was done the same simulations were ran again, and this time the NS-RED and NS-RIO traces were exact matches.

Conclusion

The bug we found in the NS-RIO code is what we consider serious. If a programmer test the code to the best of his abilities this bug should be found. If the programmer do not have a reference implementation, one could at least verify that the parameters sent to the algorithm are correct. This was clearly not done in this case. But once the bug was eliminated NS-RIO verified against NS-RED. We consider the modified NS-RIO correctly implemented and can now rely on its results in the future, although one should keep in mind that we have not verified it using both IN and OUT packets.

C.3 3 Colored Random Early Detection (3CRED)

C.3.1 Simulation 1

Type: Verification

Common parameters			
3CRED Queue Mode	Common		
Simulation time	1 s		
Scheduler used	WRR		
Packet Size	1000 B		

TCP Source	Window size	Start Time	AF Class
Node 1	33	0	AF1x
Node 2	67	0.2	AF1x
Node 3	112	0.4	AF1x
Node 4	78	0.6	AF1x

Links between Nodes	Bandwidth	Delay
Node 1 $\leftrightarrow$ Gateway	100 Mb	1 ms
Node 2 $\leftrightarrow$ Gateway	100 Mb	4 ms
Node 3 $\leftrightarrow$ Gateway	100 Mb	8 ms
Node 4 $\leftrightarrow$ Gateway	100 Mb	5 ms
Gateway $\leftrightarrow$ Sink	45 Mb	2 ms

RED Gateway $\leftrightarrow$ Sink	
Queue Buffer Limit	Unlimited
Queue Weight	0.002
Min Thresh	5
Max Thresh	15
wait_	false

3CRED Gateway $\leftrightarrow$ Sink	
Queue Buffer Limit	Unlimited
Queue Weight	0.002
Green Max Drop Prob	50
Green Min Thresh	5
Green Max Thresh	15
Yellow Max Drop Prob	50
Yellow Min Thresh	5
Yellow Max Thresh	15
Red Max Drop Prob	50
Red Min Thresh	5
Red Max Thresh	15

Table 38: 3CRED Simulation 1 Parameters

Topology

The simulation uses the same topology as previous simulations (see figure 70).

Simulation Variants:

1. Packet mode Active & ECN Inactive

2. Byte mode Active & ECN Inactive

3. Byte mode Active & ECN Active

Description

The purpose of this simulation was to verify that our own 3CRED, an extension of RED and RIO, works correctly. A description of 3CRED can be found in section 2.4.3. Since 3CRED have 3 different modes, that differ in the treatment of sharable parameters such as the *Exponential Weighted Moving Average*, simulations to verify that all different modes are needed. In this simulation we send only green packets, and the token bucket is currently disabled so that no packets are ever remarked. In this simulation we verify 3CRED against DS-RED.

We did 50 simulations on each of the simulation variants mentioned above with different fixed seeds sending green packets only. When comparing the 3CRED traces with the DS-RED traces they were exact matches.

After this we did the same sets of simulations but replacing green packets with yellow packets. This gave us the same result and all traces were matches. Finally yellow packets were replaced with red packets and once again they matched completely.

Conclusion

Although we do not expect to use common mode of the 3CRED queue, this mode needed verification. We now only need to test that using different classes work or limited queue does not give us wrong results.

C.3.2 Simulation 2

Type: Verification

Topology

The simulation uses the same topology as previous simulations (see figure 70).

Simulation Variants:

1. Packet mode Active & ECN Inactive

Sending AF Class	AF13

Common parameters			
3CRED Queue Mode	Common		
Simulation time	1 s		
Scheduler used	WRR		
Packet Size	1000 B		

TCP Source	Window size	Start Time	AF Class
Node 1	33	0	AF1x
Node 2	67	0.2	AF1x
Node 3	112	0.4	AF1x
Node 4	78	0.6	AF1x

Links between Nodes	Bandwidth	Delay
Node 1 ↔ Gateway	100 Mb	1 ms
Node 2 ↔ Gateway	100 Mb	4 ms
Node 3 ↔ Gateway	100 Mb	8 ms
Node 4 ↔ Gateway	100 Mb	5 ms
Gateway ↔ Sink	45 Mb	2 ms

RED Gateway ↔ Sink	
Queue Buffer Limit	20
Queue Weight	0.002
Min Thresh	5
Max Thresh	15
wait_	false

3CRED Gateway ↔ Sink	
Queue Buffer Limit	20
Queue Weight	0.002
Green Max Drop Prob	50
Green Min Thresh	5
Green Max Thresh	15
Yellow Max Drop Prob	50
Yellow Min Thresh	5
Yellow Max Thresh	15
Red Max Drop Prob	50
Red Min Thresh	5
Red Max Thresh	15

Table 39: 3CRED Simulation 2 Parameters

2. Byte mode Active & ECN Inactive

Sending AF Class	AF11

3. Byte mode Active & ECN Active

Sending AF Class	AF12

Description

The purpose of this simulation was to verify that our own 3 Colored RED, an extension of RED and RIO, works correctly when using limited queue. A description of 3CRED can be found in section 2.4.3.

We did 50 simulations on each of the simulation variants mentioned above with different fixed seeds. When comparing the 3CRED traces with the DS-RED traces they were exact matches.

Conclusion

This simulation tested the 3CRED algorithm in common mode, and our requirements were met accordingly.

C.3.3 Simulation 3

Type: Verification

Topology

The simulation uses the same topology as previous simulations (see figure 70).

Simulation Variants:

1. Packet mode Active & ECN Inactive

Sending AF Class	AF21
Queue Buffer Limit	20

2. Byte mode Active & ECN Inactive

Sending AF Class	AF32
Queue Buffer Limit	Unlimited

3. Byte mode Active & ECN Active

Sending AF Class	AF43
Queue Buffer Limit	20

Description

The purpose of this simulation was to verify that our own 3 Colored RED, an extension of RED and RIO, works correctly when using different AF classes. A description of 3CRED can be found in section 2.4.3.

We did 50 simulations on each of the simulation variants mentioned above with different fixed seeds. When comparing the 3CRED traces with the DS-RED traces they were exact matches.

Common parameters			
3CRED Queue Mode	Common		
Simulation time	1 s		
Scheduler used	WRR		
Packet Size	1000 B		
TCP Source	**Window size**	**Start Time**	**AF Class**
Node 1	33	0	-
Node 2	67	0.2	-
Node 3	112	0.4	-
Node 4	78	0.6	-
Links between Nodes	**Bandwidth**	**Delay**	
Node 1 $\leftrightarrow$ Gateway	100 Mb	1 ms	
Node 2 $\leftrightarrow$ Gateway	100 Mb	4 ms	
Node 3 $\leftrightarrow$ Gateway	100 Mb	8 ms	
Node 4 $\leftrightarrow$ Gateway	100 Mb	5 ms	
Gateway $\leftrightarrow$ Sink	45 Mb	2 ms	
RED Gateway $\leftrightarrow$ Sink			
Queue Buffer Limit	Specified Later		
Queue Weight	0.002		
Min Thresh	5		
Max Thresh	15		
wait_	false		
3CRED Gateway $\leftrightarrow$ Sink			
Queue Buffer Limit	Specified Later		
Queue Weight	0.002		
Green Max Drop Prob	50		
Green Min Thresh	5		
Green Max Thresh	15		
Yellow Max Drop Prob	50		
Yellow Min Thresh	5		
Yellow Max Thresh	15		
Red Max Drop Prob	50		
Red Min Thresh	5		
Red Max Thresh	15		

Table 40: 3CRED Simulation 3 Parameters

Conclusion

This simulation tested the 3CRED algorithm in common mode with different AF classes, and our requirements were met accordingly.

C.3.4 Simulation 4

Type: Verification

Common parameters		
3CRED Queue Mode	Higher Priority Inclusive	
Simulation time	1 s	
Scheduler used	WRR	
Packet Size	1000 B	

TCP Source	Window size	Start Time
Node 1	33	0
Node 2	67	0.2
Node 3	112	0.4
Node 4	78	0.6

Links between Nodes	Bandwidth	Delay
Node 1 ↔ Gateway	100 Mb	1 ms
Node 2 ↔ Gateway	100 Mb	4 ms
Node 3 ↔ Gateway	100 Mb	8 ms
Node 4 ↔ Gateway	100 Mb	5 ms
Gateway ↔ Sink	45 Mb	2 ms

RIO Gateway ↔ Sink	
Queue Weight	0.002
In Max Drop Prob	50
In Min Thresh	5
In Max Thresh	10
Out Max Drop Prob	50
Out Min Thresh	10
Out Max Thresh	15
wait_	false

3CRED Gateway ↔ Sink	
Queue Weight	0.002
Green Max Drop Prob	50
Green Min Thresh	5
Green Max Thresh	10
Yellow Max Drop Prob	50
Red Max Drop Prob	50
Red Min Thresh	10
Red Max Thresh	15

Table 41: 3CRED Simulation 4 Parameters

Topology

The simulation uses the same topology as previous simulations (see figure 70).

Simulation Variants:

1. Byte mode Active & ECN Inactive

Queue Buffer Limit	30
TCP Source	AF Class/RIO Priority
Node 1	AF13/OUT
Node 2	AF11/IN
Node 3	AF13/OUT
Node 4	AF11/IN
3CRED Gateway ↔ Sink	
Yellow Min Thresh	Unspecified
Yellow Max Thresh	Unspecified

2. Packet mode Active & ECN Active

Queue Buffer Limit	Unlimited
TCP Source	AF Class/RIO Priority
Node 1	AF13/OUT
Node 2	AF11/IN
Node 3	AF13/OUT
Node 4	AF11/IN
3CRED Gateway ↔ Sink	
Yellow Min Thresh	Unspecified
Yellow Max Thresh	Unspecified

3. Byte mode Active & ECN Active

Queue Buffer Limit	Unlimited
TCP Source	AF Class/RIO Priority
Node 1	AF12/OUT
Node 2	AF11/IN
Node 3	AF12/OUT
Node 4	AF11/IN
3CRED Gateway ↔ Sink	
Yellow Min Thresh	10
Yellow Max Thresh	15

4. Packet mode Active & ECN Inactive

Queue Buffer Limit	25
TCP Source	AF Class/RIO Priority
Node 1	AF12/OUT
Node 2	AF11/IN
Node 3	AF12/OUT
Node 4	AF11/IN
3CRED Gateway ↔ Sink	
Yellow Min Thresh	10
Yellow Max Thresh	15

5. Byte mode Active & ECN Inactive

Queue Buffer Limit	40
TCP Source	AF Class/RIO Priority
Node 1	AF13/OUT
Node 2	AF12/IN
Node 3	AF13/OUT
Node 4	AF12/IN
3CRED Gateway ↔ Sink	
Yellow Min Thresh	5
Yellow Max Thresh	10

6. Packet mode Active & ECN Inactive

Queue Buffer Limit	Unlimited
TCP Source	AF Class/RIO Priority
Node 1	AF13/OUT
Node 2	AF12/IN
Node 3	AF13/OUT
Node 4	AF12/IN
3CRED Gateway ↔ Sink	
Yellow Min Thresh	5
Yellow Max Thresh	10

Description

Higher Priority Inclusive mode works a bit different than common mode. Higher Priority Inclusive mode is described in 2.4.3. We verify 3CRED against a bug fixed contributed RIO. What bug fixes that were needed can be found in section 4.1.

We did 50 simulations on each of the simulation variants mentioned above with different fixed seeds. When comparing the 3CRED traces with the contributed RIO traces they were all exact matches.

Conclusion

3CRED works in Higher Priority Inclusive mode, all that remains is to test other classes such as AF2, AF3 and AF4.

C.3.5 Simulation 5

Type: Verification

Topology

The simulation uses the same topology as previous simulations (see figure 70).

Common parameters		
3CRED Queue Mode	Higher Priority Inclusive	
Simulation time	1 s	
Scheduler used	WRR	
Packet Size	1000 B	
TCP Source	**Window size**	**Start Time**
Node 1	15	0
Node 2	22	0.2
Node 3	32	0.4
Node 4	25	0.6
Links between Nodes	**Bandwidth**	**Delay**
Node 1 ↔ Gateway	100 Mb	1 ms
Node 2 ↔ Gateway	100 Mb	4 ms
Node 3 ↔ Gateway	100 Mb	8 ms
Node 4 ↔ Gateway	100 Mb	5 ms
Gateway ↔ Sink	10 Mb	5 ms
RIO Gateway ↔ Sink		
Queue Weight	0.002	
Queue Buffer Limit	30	
In Max Drop Prob	50	
In Min Thresh	5	
In Max Thresh	10	
Out Max Drop Prob	50	
Out Min Thresh	10	
Out Max Thresh	15	
wait_	false	
3CRED Gateway ↔ Sink		
Queue Buffer Limit	30	
Queue Weight	0.002	
Green Max Drop Prob	50	
Green Min Thresh	5	
Green Max Thresh	10	
Yellow Max Drop Prob	50	
Red Max Drop Prob	50	
Red Min Thresh	10	
Red Max Thresh	15	

Table 42: 3CRED Simulation 5 Parameters

Simulation Variants:

1. Packet mode Active & ECN Inactive

TCP Source	AF Class/RIO Priority
Node 1	AF11/IN
Node 2	AF13/OUT
Node 3	AF11/IN
Node 4	AF13/OUT
3CRED Gateway ↔ Sink	
Yellow Min Thresh	Unspecified
Yellow Max Thresh	Unspecified

2. Byte mode Active & ECN Inactive

TCP Source	AF Class/RIO Priority
Node 1	AF11/IN
Node 2	AF12/OUT
Node 3	AF11/IN
Node 4	AF12/OUT
3CRED Gateway ↔ Sink	
Yellow Min Thresh	10
Yellow Max Thresh	15

3. Byte mode Active & ECN Active

TCP Source	AF Class/RIO Priority
Node 1	AF12/IN
Node 2	AF13/OUT
Node 3	AF12/IN
Node 4	AF13/OUT
3CRED Gateway ↔ Sink	
Yellow Min Thresh	5
Yellow Max Thresh	10

Description

We wanted to test 3CRED with different classes and also change the load in the network. Therefore delay and bandwidth was changed on the bottleneck link between the gateway and the sink.

We did 50 simulations on each of the simulation variants mentioned above with different fixed seeds. When comparing the 3CRED traces with the contributed RIO traces they were all exact matches.

Conclusion

3CRED works in Higher Priority Inclusive mode in these simulations as well, and this concludes our 3CRED verification.

C.4 Worst-case Fair Weighted Fair Queuing+ (WF2Q+)

C.4.1 Simulation 1

Type: Verification

Common parameters		
Simulation time	1 s	
Scheduler used	WF^2Q	
Packet Size	1000 B	

TCP Source	Window size	Start Time
Node 1	33	0
Node 2	67	0.2
Node 3	112	0.4
Node 4	78	0.6

Links between Nodes	Bandwidth	Delay
Node 1 ↔ Gateway	100 Mb	1 ms
Node 2 ↔ Gateway	100 Mb	4 ms
Node 3 ↔ Gateway	100 Mb	8 ms
Node 4 ↔ Gateway	100 Mb	5 ms
Gateway ↔ Sink	45 Mb	2 ms

RED Parameters on Link Gateway ↔ Sink	
Queue Buffer Limit	Unlimited
Queue Weight	0.002
Max Drop Prob	50
Min Thresh	Unlimited
Max Thresh	Unlimited

Table 43: WF^{2}Q+ Simulation 1 Parameters

Topology

The simulation uses the same topology as previous simulations (see figure 70).

Simulation Variants:

1. All sources same flow

AF Class/Flow id	WF^2Q Flow weight
AF11 / 0	1

TCP Source	AF Class/Flow id
Node 1	AF11 / 0
Node 2	AF11 / 0
Node 3	AF11 / 0
Node 4	AF11 / 0

2. Different flows and same weights

AF Class/Flow id	WF^2Q Flow weight
AF11 / 0	1
AF21 / 1	1
AF31 / 2	1
AF41 / 3	1
TCP Source	AF Class/Flow id
Node 1	AF11 / 0
Node 2	AF21 / 1
Node 3	AF31 / 2
Node 4	AF41 / 3

3. Different flows and different weights

AF Class/Flow id	WF^2Q Flow weight
AF11 / 0	4
AF21 / 1	3
AF31 / 2	2
AF41 / 3	1
TCP Source	AF Class/Flow id
Node 1	AF11 / 0
Node 2	AF21 / 1
Node 3	AF31 / 2
Node 4	AF41 / 3

Description

We needed to test that the WF^2Q+ Scheduler was correctly implemented. We decided to test it against the contributed WF^2Q Scheduler. Comparing these to schedulers (with disabled RED functionality in DS) we ran 50 simulations on each of the simulation variants found above.

Conclusion

All trace files matched with diff (Unix file comparison tool), and this verification was successful.

C.4.2 Simulation 2

Type: Verification

Common parameters		
Simulation time	1 s	
Scheduler used	WF^2Q/WRR	
Packet Size	1000 B	

TCP Source	Window size	Start Time
Node 1	33	0
Node 2	67	0.2
Node 3	112	0.4
Node 4	78	0.6

Links between Nodes	Bandwidth	Delay
Node 1 ↔ Gateway	100 Mb	1 ms
Node 2 ↔ Gateway	100 Mb	4 ms
Node 3 ↔ Gateway	100 Mb	8 ms
Node 4 ↔ Gateway	100 Mb	5 ms
Gateway ↔ Sink	45 Mb	2 ms

RED Parameters on Link Gateway ↔ Sink	
Queue Buffer Limit	50
Queue Weight	0.002
Max Drop Prob	50
Min Thresh	5
Max Thresh	15

Table 44: WF^{2}Q+ Simulation 2 Parameters

Topology

The simulation uses the same topology as previous simulations (see figure 70).

Simulation Variants:

1. All sources same flow

PHB Class/Flow id	WF^2Q/WRR Flow weight
BE / 0	1
TCP Source	PHB Class/Flow id
Node 1	BE / 0
Node 2	BE / 0
Node 3	BE / 0
Node 4	BE / 0

Description

We needed to test that the WF^2Q+ Scheduler was correctly implemented when compared with a single flow against another scheduler. Since the only other scheduler we could compare against was the WRR Scheduler, a single simulation was done. We compare the WRR

Scheduler against the WF^2Q Scheduler with a single BE flow. We ran 50 simulations on each of the simulation variants found above.

Conclusion

All traces files matched with diff (Unix file comparison tool), and this concludes our WF^2Q verification.

D Pareto Distribution

The *Pareto Distribution* function is parameterized by two values, scale and shape.

$$P(scale, shape) = scale * \frac{1.0}{pow(uniform(), 1.0/shape)}$$

$$where\ P(scale, shape) \geq scale, scale > 0, shape > 0$$

The *Pareto Distribution* is a heavy-tailed distribution. Thus, the mean, variance, and other moments are finite only if the shape parameter is sufficiently large. *Pareto Distribution* will give values in the range $[scale, \infty]$. Therefore there is often a need to truncate the values, e.g. when used as a file size distribution.

A view of pareto distribution for two different shape values can be seen in figure 71.

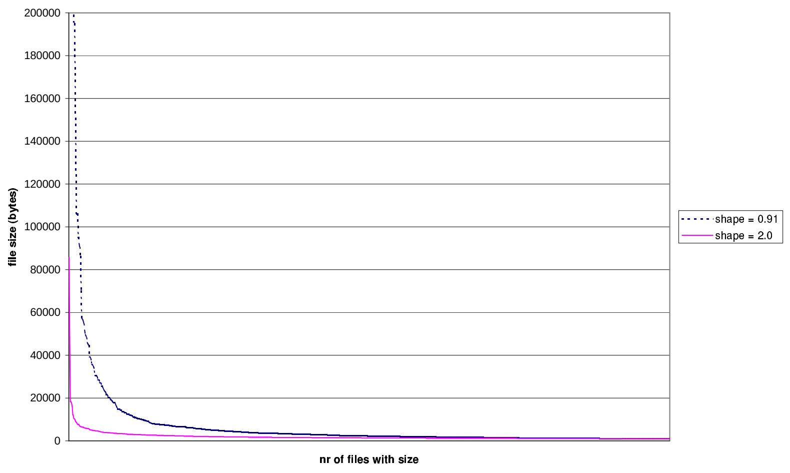


Figure 71: Pareto Distribution(scale = 1000)

More information about Pareto Distribution can be found on the Internet [1, 41].

E TCL Example Script

Example script using a TCL subroutine library we have implemented ourselves. Please note that it has been adjusted to fit the page properly.

```
source "../../ds_include.tcl"

proc create_topology {} {
    global colors scheduler trace_filename nam_filename start
           stop warmup param max_think

    set ns [Simulator instance]
    set fid 0

#    start_namoutput $nam_filename
#    start_trace_output Recv $trace_filename "" ""

    puts "Creating topology..."
    puts "\tnodes."

    # create nodes
    set gateway1 [$ns node]
    set gateway2 [$ns node]
    set gateway3   [$ns node]

    for {set i 0} {$i < 6} { incr i} {
        set nodesrc($i) [$ns node]
        set nodedst($i) [$ns node]
    }

    puts "\tlinks."

    # create links (and monitors)
    $ns DS3CSchedulerLink-duplex-link DS3CScheduler/WF2Q $gateway1 $gateway2
        10Mb 2ms
    create_monitor $gateway1 $gateway2 0.5
    $ns DS3CSchedulerLink-duplex-link DS3CScheduler/WF2Q $gateway2 $gateway3
        10Mb 2ms

    # create src links
    for {set i 0} {$i < 6} { incr i} {
        $ns duplex-link $gateway1 $nodesrc($i) 10Mb 1ms DropTail
        $ns duplex-link $gateway3 $nodedst($i) 10Mb 1ms DropTail
    }

    ### SENDING ###
    # NODESRC(0)
    puts "\tsending for nodesrc(0)."
    # ftp sources
    for {set i 0} {$i < 1} {incr i} {
        set ns0_tcp($i) [create_agent Agent/TCP AF11 $nodesrc(0)]
        $ns0_tcp($i) set window_ 64
        $ns0_tcp($i) set ecn_ 0
        $ns0_tcp($i) set fid_ [incr fid]
        $ns0_tcp($i) set packetSize_ 1500
        $ns color 0 $colors(0)
        set ns0_ftp($i) [new Application/FTP]
        $ns0_ftp($i) attach-agent $ns0_tcp($i)
        $ns at 0.0 "$ns0_ftp($i) start"
    }

    # NODESRC(1)
    # ftp sources
    puts "\tsending for nodesrc(1)."
    for {set i 0} {$i < 1} {incr i} {
        set ns1_tcp($i) [create_agent Agent/TCP AF21 $nodesrc(1)]
        $ns1_tcp($i) set window_ 64
        $ns1_tcp($i) set ecn_ 0
        $ns1_tcp($i) set fid_ [incr fid]
        $ns1_tcp($i) set packetSize_ 1500
        $ns color 1 $colors(1)
        set ns1_ftp($i) [new Application/FTP]
        $ns1_ftp($i) attach-agent $ns1_tcp($i)
        $ns at 0.0 "$ns1_ftp($i) start"
    }

    # NODESRC(2)
    # ftp sources
    puts "\tsending for nodesrc(2)."
    for {set i 0} {$i < 1} {incr i} {
        set ns2_tcp($i) [create_agent Agent/TCP AF31 $nodesrc(2)]
        $ns2_tcp($i) set window_ 64
        $ns2_tcp($i) set ecn_ 0
        $ns2_tcp($i) set fid_ [incr fid]
        $ns2_tcp($i) set packetSize_ 1500
        $ns color 2 $colors(2)
        set ns2_ftp($i) [new Application/FTP]
        $ns2_ftp($i) attach-agent $ns2_tcp($i)
        $ns at 0.0 "$ns2_ftp($i) start"
    }

    # NODESRC(3)
    # ftp sources
    puts "\tsending for nodesrc(3)."
    for {set i 0} {$i < 1} {incr i} {
        set ns3_tcp($i) [create_agent Agent/TCP AF41 $nodesrc(3)]
        $ns3_tcp($i) set window_ 64
        $ns3_tcp($i) set ecn_ 0
        $ns3_tcp($i) set fid_ [incr fid]
        $ns3_tcp($i) set packetSize_ 1500
        $ns color 3 $colors(3)
        set ns3_ftp($i) [new Application/FTP]
        $ns3_ftp($i) attach-agent $ns3_tcp($i)
        $ns at 0.0 "$ns3_ftp($i) start"
    }

    # cbr
    puts "\tsending for nodesrc(4)."
    set ns4_src_udp0 [create_agent Agent/UDP EF $nodesrc(4)]
    set ns4_cbr0 [new Application/Traffic/CBR]
    $ns4_cbr0 set rate_ 40kb
    $ns4_cbr0 set packetSize_ 500
    $ns4_cbr0 attach-agent $ns4_src_udp0
    $ns at 0.0 "$ns4_cbr0 start"


    # NODESRC(5)
    # ftp sources
    puts "\tsending for nodesrc(5)."
    for {set i 0} {$i < 1} {incr i} {
        set ns5_tcp($i) [create_agent Agent/TCP BE $nodesrc(5)]
        $ns5_tcp($i) set window_ 64
        $ns5_tcp($i) set ecn_ 0
        $ns5_tcp($i) set fid_ [incr fid]
        $ns5_tcp($i) set packetSize_ 1500
        $ns color 5 $colors(5)
        set ns5_ftp($i) [new Application/FTP]
        $ns5_ftp($i) attach-agent $ns5_tcp($i)
        $ns at 0.0 "$ns5_ftp($i) start"
    }

    # NOISE 0 CBR
    set nsrc_udp0 [create_agent Agent/UDP $param(3) $gateway1]
    set nsrc_cbr0 [new Application/Traffic/CBR]
    $nsrc_cbr0 set rate_ $param(1)kb
```

```
$nsrc_cbr0 set packetSize_ 1500
$nsrc_cbr0 attach-agent $nsrc_udp0
$ns at 0.0 "$nsrc_cbr0 start"

# NOISE 1 CBR
set nsrc_udp1 [create_agent Agent/UDP $param(4) $gateway2]
set nsrc_cbr1 [new Application/Traffic/CBR]
$nsrc_cbr1 set rate_ $param(2)kb
$nsrc_cbr1 set packetSize_ 1500
$nsrc_cbr1 attach-agent $nsrc_udp1
$ns at 0.0 "$nsrc_cbr1 start"


### RECEIVING ###


# NODEDST(0)
# ftp sources
puts "\treceiving for nodedst(0)."
for {set i 0} {$i < 1} {incr i} {
    set nd0_sink($i) [create_agent Agent/TCPSink AF11 $nodedst(0)]
    $nd0_sink($i) set fid_ 100
    $ns color 100 $colors(10)
    $ns connect $ns0_tcp($i) $nd0_sink($i)
}

# NODEDST(1)
# ftp sources
puts "\treceiving for nodedst(1)."
for {set i 0} {$i <1} {incr i} {
    set nd1_sink($i) [create_agent Agent/TCPSink AF21 $nodedst(1)]
    $nd1_sink($i) set fid_ 100
    $ns color 100 $colors(10)
    $ns connect $ns1_tcp($i) $nd1_sink($i)
}

# NODEDST(2)
# ftp sources
puts "\treceiving for nodedst(2)."
for {set i 0} {$i < 1} {incr i} {
    set nd2_sink($i) [create_agent Agent/TCPSink AF31 $nodedst(2)]
    $nd2_sink($i) set fid_ 100
    $ns color 100 $colors(10)
    $ns connect $ns2_tcp($i) $nd2_sink($i)
}

# NODEDST(3)
# ftp sources
puts "\treceiving for nodedst(3)."
for {set i 0} {$i < 1} {incr i} {
    set nd3_sink($i) [create_agent Agent/TCPSink AF41 $nodedst(3)]
    $nd3_sink($i) set fid_ 100
    $ns color 100 $colors(10)
    $ns connect $ns3_tcp($i) $nd3_sink($i)
}

puts "\treceiving for nodedst(4)."

# sink for cbr
set nd4_null0 [create_agent Agent/Null EF $nodedst(4)]
$ns connect $ns4_src_udp0 $nd4_null0

# NODEDST(5)
# ftp sources
puts "\treceiving for nodedst(5)."
for {set i 0} {$i < 1} {incr i} {
    set nd5_sink($i) [create_agent Agent/TCPSink BE $nodedst(5)]
    $nd5_sink($i) set fid_ 100
    $ns color 100 $colors(10)
    $ns connect $ns5_tcp($i) $nd5_sink($i)
}

puts "\temulating www traffic."
#www
for {set i 0} {$i<20} {incr i} {
    send_www $nodesrc(0) $nodedst(0) AF11 1000 $max_think
 [uniform 0.0 $warmup]
}

#www
for {set i 0} {$i<30} {incr i} {
    send_www $nodesrc(1) $nodedst(1) AF21 1000 $max_think
         [uniform 0.0 $warmup]
}

#www
for {set i 0} {$i<40} {incr i} {
    send_www $nodesrc(2) $nodedst(2) AF31 1000 $max_think
         [uniform 0.0 $warmup]
}

#www
for {set i 0} {$i<50} {incr i} {
    send_www $nodesrc(3) $nodedst(3) AF41 1000 $max_think
         [uniform 0.0 $warmup]
}

#www
for {set i 0} {$i<60} {incr i} {
    send_www $nodesrc(5) $nodedst(5) BE 1000 $max_think
         [uniform 0.0 $warmup]
}

# NOISE 0 CBR SINK
set ndst_null0 [create_agent Agent/Null $param(3) $gateway2]
$ns connect $nsrc_udp0 $ndst_null0
# NOISE 1 CBR SINK
set ndst_null1 [create_agent Agent/Null $param(4) $gateway3]
$ns connect $nsrc_udp1 $ndst_null1


# SCHEDULERS
puts "\tschedulers."
# create 4 DS-schedulers
set scheduler(0) [create_scheduler $gateway1 $gateway2]
set scheduler(1) [create_scheduler $gateway2 $gateway3]
set scheduler(2) [create_scheduler $gateway3 $gateway2]
set scheduler(3) [create_scheduler $gateway2 $gateway1]
# red/3cred shared-buffer-mode
$scheduler(0) set-shared-buffer-mode 1
$scheduler(0) set-ignore-sbm-mode 1 EF
$scheduler(0) set-shared-buffer-size [$gateway1 id]
    [$gateway2 id] 128000
$scheduler(1) set-shared-buffer-size [$gateway2 id]
    [$gateway3 id] 128000
$scheduler(2) set-shared-buffer-size [$gateway3 id]
    [$gateway2 id] 128000
$scheduler(3) set-shared-buffer-size [$gateway2 id]
    [$gateway3 id] 128000

# set scheduling params (queue weights, color-handling-mode,
      byte-mode, ecn, tb etc)
for {set i 0} {$i < 4} {incr i} {
    # $scheduler($i) set-quantum 1000

    # phb scheduling weights
```

```
        $scheduler($i) set ef_queue_weight_ 20
        $scheduler($i) set af1_queue_weight_ 10
        $scheduler($i) set af2_queue_weight_ 6
        $scheduler($i) set af3_queue_weight_ 4
        $scheduler($i) set af4_queue_weight_ 2
        $scheduler($i) set be_queue_weight_ 1

        # 3cred color-handling-mode
        $scheduler($i) set-colorhandling-mode 1 EF
        $scheduler($i) set-colorhandling-mode 1 AF1
        $scheduler($i) set-colorhandling-mode 1 AF2
        $scheduler($i) set-colorhandling-mode 1 AF3
        $scheduler($i) set-colorhandling-mode 1 AF4

        # red/3cred byte-mode
        $scheduler($i) set-byte-mode 1 EF
        $scheduler($i) set-byte-mode 1 AF1
        $scheduler($i) set-byte-mode 1 AF2
        $scheduler($i) set-byte-mode 1 AF3
        $scheduler($i) set-byte-mode 1 AF4
        $scheduler($i) set-byte-mode 1 BE

        # red/3cred ecn-mode
        $scheduler($i) set-ecn-mode 0 EF
        $scheduler($i) set-ecn-mode 0 AF1
        $scheduler($i) set-ecn-mode 0 AF2
        $scheduler($i) set-ecn-mode 0 AF3
        $scheduler($i) set-ecn-mode 0 AF4
        $scheduler($i) set-ecn-mode 0 BE

        # red/3cred queue lengths
        $scheduler($i) set-queue-length 5000  EF
        $scheduler($i) set-queue-length 10000 AF1
        $scheduler($i) set-queue-length 20000 AF2
        $scheduler($i) set-queue-length 30000 AF3
        $scheduler($i) set-queue-length 40000 AF4
        $scheduler($i) set-queue-length 50000 BE

        # red/3cred queue lengths
        $scheduler($i) set mean_ef_packet_size_ $param(0)
        $scheduler($i) set mean_af1_packet_size_ $param(0)
        $scheduler($i) set mean_af2_packet_size_ $param(0)
        $scheduler($i) set mean_af3_packet_size_ $param(0)
        $scheduler($i) set mean_af4_packet_size_ $param(0)
        $scheduler($i) set mean_be_packet_size_ $param(0)

# red/3cred thresholds and linterms
        $scheduler($i) queue-red3-params 0.002 6000 6000  50 6000
               6000  40 6000  6000  30 EF
        $scheduler($i) queue-red3-params 0.002 2000 5000  50 4000
                7000  40 6000  9000  30 AF1
        $scheduler($i) queue-red3-params 0.002 4000 10000 50 8000
                14000 40 12000 18000 30 AF2
        $scheduler($i) queue-red3-params 0.002 6000 15000 50 12000
                21000 40 18000 27000 30 AF3
        $scheduler($i) queue-red3-params 0.002 8000 20000 50 16000
                28000 40 24000 36000 30 AF4
        $scheduler($i) be-queue-red-params 0.002 20000 40000 50

        # disable token bucket
        $scheduler($i) set-use-tb 0 EF
        $scheduler($i) set-use-tb 1 AF1
        $scheduler($i) set-use-tb 1 AF2
        $scheduler($i) set-use-tb 1 AF3
        $scheduler($i) set-use-tb 1 AF4

        # 3cred token bucket
        $scheduler($i) set-tokenbucket-params      0      0      0
```

```
                0  EF
        $scheduler($i) set-tokenbucket-params 200000 150000 150000
                100000 AF1
        $scheduler($i) set-tokenbucket-params 250000 200000 200000
                150000 AF2
        $scheduler($i) set-tokenbucket-params 300000 200000 200000
                150000 AF3
        $scheduler($i) set-tokenbucket-params 300000 200000 200000
                150000 AF4

        $ns at $warmup "$scheduler($i) reset-stats"

    }
    puts "\tdone."
}

proc final_section { } {
    # each line MUST start with a # (hash) char
    return "#<no data>\n#"
}

proc times {} {
    set ns [Simulator instance]
    puts "Simulation Time is [$ns now]"
    $ns at [expr 1+[$ns now]] "times"
}

proc end {} {
    global nam_filename final_section xgraph_filenames scheduler
           trace_filename stats_filename simId

    set ns [Simulator instance]

#    stop_trace_output $trace_filename
#    stop_namoutput $nam_filename

    for {set i 0} {$i<4} {incr i} {
        set stats_filename "$simId-$i.stats"
        set stats [open $stats_filename w]

        print_scheduler_stats $scheduler($i) $stats
        puts $stats "<EOF>"

        close $stats
    }

#    stop_all_graphs $scheduler(0) 0 $simId
#    stop_all_graphs $scheduler(1) 1 $simId


    exit 0
}

# initialisation

# parse command line parameters

set param_def(0) 1000
set param(0) ""
set param(1) ""
set param(2) ""
set param(3) ""
set param(4) ""
set param(5) ""

if {$argc >= 3} {
    set simId [lindex $argv 0]
    set seed [lindex $argv 1]
    for {set i 0} {$i<$argc-2} {incr i} {
set param($i) [lindex $argv [expr $i+2]]
    }
} elseif {$argc == 2} {
    set simId [lindex $argv 0]
    set seed [lindex $argv 1]
    set param(0) $param_def(0)
} elseif {$argc == 1} {
    set simId [lindex $argv 0]
    set seed 0
    set param(0) $param_def(0)
} elseif {$argc == 0} {
    set simId 1
    set seed 0
    set param(0) $param_def(0)
} else {
    exit 0
}

puts "Sim #$simId running with seed $seed with option params
          $param(0) $param(1) $param(2) $param(3) $param(4) $param(5)"

set start 0.0
set warmup 1.0
set stop 100

set max_think [expr ($stop-$start)/2]

set nam_filename "$simId.nam"
set trace_filename "$simId.tr"
set xgraph_filenames "$simId-1.tr"

set ns [new Simulator]
ds_incl_init
ns-random $seed

create_topology

#make_all_graphs $scheduler(0) 0 $simId 0.01 50 100
#make_all_graphs $scheduler(1) 1 $simId 0.01 50 100

$ns at $start "times"
$ns at $stop "end"

# run

puts "-- Starting Simulation --"
$ns run
```

F Statistics File Example

The example below is the average over ten Sim 21 simulations with mean packet size 50. Please note that it has been adjusted to fit the page properly.

```
-----------------------------------------------
AF1 Arrived Packets    : 35047.9 // 35047.9 // 0.0 //  0.00
AF2 Arrived Packets    : 23591.7 // 23591.7 // 0.0 //  0.00
AF3 Arrived Packets    : 16345.7 // 16345.7 // 0.0 //  0.00
AF4 Arrived Packets    : 9337.4 // 9337.4 // 0.0 //  0.00
EF  Arrived Packets    : 990.0 // 990.0 // 0.0 //  0.00
BE  Arrived Packets    : 5946.2 // 0.0 // 0.0 // 0.00
Total Arrived Packets  : 91258.9 // 85312.7 // 0.0 // 0.00
-----------------------------------------------
AF1 Arrived Packets %  : 100.0  // 100.0  // 0.0  // 0.00
AF2 Arrived Packets %  : 100.0  // 100.0  // 0.0  // 0.00
AF3 Arrived Packets %  : 100.0  // 100.0  // 0.0  // 0.00
AF4 Arrived Packets %  : 100.0  // 100.0  // 0.0  // 0.00
EF  Arrived Packets %  : 100.0  // 100.0  // 0.0  // 0.00
BE  Arrived Packets %  : 100.0  // 0.0  // 0.0  // 0.0
-----------------------------------------------
AF1 Arrived Bytes      : 51588200.0 // 51588200.0 // 0.0 //  0.0
AF2 Arrived Bytes      : 33634000.0 // 33634000.0 // 0.0 //  0.0
AF3 Arrived Bytes      : 22201350.0 // 22201350.0 // 0.0 //  0.0
AF4 Arrived Bytes      : 11265100.0 // 11265100.0 // 0.0 //  0.0
EF  Arrived Bytes      : 495000.0 // 495000.0 // 0.0 //  0.0
BE  Arrived Bytes      : 6149350.0 // 0.0 // 0.0 // 0.0
Total Arrived Bytes    : 125333000.0 // 119183650.0 // 0.0 // 0.0
-----------------------------------------------
AF1 Arrived Bytes %    : 100.0  // 100.0  // 0.0  // 0.0
AF2 Arrived Bytes %    : 100.0  // 100.0  // 0.0  // 0.0
AF3 Arrived Bytes %    : 100.0  // 100.0  // 0.0  // 0.0
AF4 Arrived Bytes %    : 100.0  // 100.0  // 0.0  // 0.0
EF  Arrived Bytes %    : 100.0  // 100.0  // 0.0  // 0.0
BE  Arrived Bytes %    : 100.0  // 0.0  // 0.0  // 0.0
-----------------------------------------------
AF1 Enqueued Packets   : 34711.1 // 6728.0 // 3338.2 // 24644.9
AF2 Enqueued Packets   : 23439.2 // 10361.6 // 3472.3 // 9605.3
AF3 Enqueued Packets   : 16188.6 // 10789.3 // 3597.4 // 1801.9
AF4 Enqueued Packets   : 9045.0 // 9045.0 // 0.0 // 0.0
EF  Enqueued Packets   : 990.0 // 990.0 // 0.0 // 0.0
BE  Enqueued Packets   : 5514.1 // 0.0 // 0.0 // 0.0
Total Enqueued Packets : 89888.0 // 37913.9 // 10407.9 // 36052.1
-----------------------------------------------
AF1 Enqueued Packets % : 100.0  // 19.384  // 9.618  // 70.999
AF2 Enqueued Packets % : 100.0  // 44.211  // 14.816  // 40.974
AF3 Enqueued Packets % : 100.0  // 66.662  // 22.222  // 11.117
AF4 Enqueued Packets % : 100.0  // 100.0  // 0.0  // 0.0
EF  Enqueued Packets % : 100.0  // 100.0  // 0.0  // 0.0
BE  Enqueued Packets % : 100.0  // 0.0  // 0.0  // 0.0
-----------------------------------------------
AF1 Enqueued Bytes     : 51105400.0 // 9816750.0 // 4922200.0 // 36366450.0
AF2 Enqueued Bytes     : 33427450.0 // 14666350.0 // 4979800.0 // 13781300.0
AF3 Enqueued Bytes     : 21998100.0 // 14546600.0 // 4970200.0 // 2481300.0
AF4 Enqueued Bytes     : 10922500.0 // 10922500.0 // 0.0 // 0.0
EF  Enqueued Bytes     : 495000.0 // 495000.0 // 0.0 // 0.0
BE  Enqueued Bytes     : 5693200.0 // 0.0 // 0.0 // 0.0
Total Enqueued Bytes   : 123641650.0 // 50447200.0 // 14872200.0 // 52629050.0
```

```
-----------------------------------------------
AF1 Enqueued Bytes %   : 100.0  // 19.21  // 9.633  // 71.158
AF2 Enqueued Bytes %   : 100.0  // 43.878  // 14.898  // 41.224
AF3 Enqueued Bytes %   : 100.0  // 66.136  // 22.592  // 11.273
AF4 Enqueued Bytes %   : 100.0  // 100.0  // 0.0  // 0.0
EF  Enqueued Bytes %   : 100.0  // 100.0  // 0.0  // 0.0
BE  Enqueued Bytes %   : 100.0  // 0.0  // 0.0  // 0.0
-----------------------------------------------
AF1 Dropped Packets    : 336.8 // 40.4 // 19.3 // 277.1
AF2 Dropped Packets    : 152.5 // 120.2 // 3.0 // 29.3
AF3 Dropped Packets    : 157.1 // 157.1 // 0.0 // 0.0
AF4 Dropped Packets    : 292.4 // 292.4 // 0.0 // 0.0
EF  Dropped Packets    : 0.0 // 0.0 // 0.0 // 0.0
BE  Dropped Packets    : 432.1 // 0.0 // 0.0 // 0.0
Total Dropped Packets  : 1370.9 // 610.1 // 22.3 // 306.4
-----------------------------------------------
AF1 Dropped Packets %  : 100.0  // 12.0  // 5.734  // 82.268
AF2 Dropped Packets %  : 100.0 % // 79.332  // 1.908  // 18.759
AF3 Dropped Packets %   : 100.0  // 100.0  // 0.0  // 0.0
AF4 Dropped Packets %   : 100.0  // 100.0  // 0.0  // 0.0
EF  Dropped Packets %   : 100.0  // 0.0  // 0.0  // 0.0
BE  Dropped Packets %   : 100.0  // 0.0  // 0.0  // 0.0
-----------------------------------------------
AF1 Dropped Bytes       : 482800.0 // 56800.0 // 27850.0 // 398150.0
AF2 Dropped Bytes       : 206550.0 // 166750.0 // 3750.0 // 36050.0
AF3 Dropped Bytes       : 203250.0 // 203250.0 // 0.0 // 0.0
AF4 Dropped Bytes       : 342600.0 // 342600.0 // 0.0 // 0.0
EF  Dropped Bytes       : 0.0 // 0.0 // 0.0 // 0.0
BE  Dropped Bytes       : 456150.0 // 0.0 // 0.0 // 0.0
Total Dropped Bytes     : 1691350.0 // 769400.0 // 31600.0 // 434200.0
-----------------------------------------------
AF1 Dropped Bytes %     : 100.0  // 11.773  // 5.773  // 82.457
AF2 Dropped Bytes %     : 100.0  // 81.12  // 1.766  // 17.112
AF3 Dropped Bytes %     : 100.0  // 100.0  // 0.0  // 0.0
AF4 Dropped Bytes %     : 100.0  // 100.0  // 0.0  // 0.0
EF  Dropped Bytes %     : 100.0  // 0.0  // 0.0  // 0.0
BE  Dropped Bytes %     : 100.0  // 0.0  // 0.0  // 0.0
-----------------------------------------------
AF1 Recolored Packets   : 3357.5 // 24922.0 // 0.0
AF2 Recolored Packets   : 3475.3 // 9634.6 // 0.0
AF3 Recolored Packets   : 3597.4 // 1801.9 // 0.0
AF4 Recolored Packets   : 0.0 // 0.0 // 0.0
EF  Recolored Packets   : 0.0 // 0.0 // 0.0
BE  Recolored Packets   : 0.0 // 0.0 // 0.0
Total Recolored Packets : 10430.2 // 36358.5 // 0.0
-----------------------------------------------
AF1 Recolored Packets % : 9.579  // 71.108  // 0.0
AF2 Recolored Packets % : 14.732  // 40.834  // 0.0
AF3 Recolored Packets % : 22.007  // 11.01  // 0.0
AF4 Recolored Packets % : 0.0  // 0.0  // 0.0
EF  Recolored Packets % : 0.0  // 0.0  // 0.0
BE  Recolored Packets % : 0.0  // 0.0  // 0.0
```

```
-----------------------------------------------
AF1 Recolored Bytes     : 4950050.0 // 36764600.0 // 0.0
AF2 Recolored Bytes     : 4983550.0 // 13817350.0 // 0.0
AF3 Recolored Bytes     : 4970200.0 // 2481300.0 // 0.0
AF4 Recolored Bytes     : 0.0 // 0.0 // 0.0
EF  Recolored Bytes     : 0.0 // 0.0 // 0.0
BE  Recolored Bytes     : 0.0 // 0.0 // 0.0
Total Recolored Bytes   : 14903800.0 // 53063250.0 // 0.0
-----------------------------------------------
AF1 Recolored Bytes %   : 9.595  // 71.264  // 0.0
AF2 Recolored Bytes %   : 14.818  // 41.077  // 0.0
AF3 Recolored Bytes %   : 22.384  // 11.168  // 0.0
AF4 Recolored Bytes %   : 0.0  // 0.0  // 0.0
EF  Recolored Bytes %   : 0.0  // 0.0  // 0.0
BE  Recolored Bytes %   : 0.0  // 0.0  // 0.0
-----------------------------------------------
AF1 Drop-ratio          : 0.0097 //  0.0011 //  0.00563 //  0.0
AF2 Drop-ratio          : 0.0065 //  0.00496 //  0.00075 //  0.0
AF3 Drop-ratio          : 0.00968 //  0.00915 // 0.0 // 0.0
AF4 Drop-ratio          : 0.03225 // 0.03039 // 0.0 // 0.0
EF  Drop-ratio          : 0.0 // 0.0 // 0.0 // 0.0
BE  Drop Ratio          : 0.0722 // 0.0 // 0.0 // 0.0
-----------------------------------------------
AF1 Average Delay (ms)  : 21.55771 // 19.46322 // 19.53005 // 22.40421
AF2 Average Delay (ms)  : 62.73776 // 59.46574 // 60.90943 // 66.9386
AF3 Average Delay (ms)  : 89.57476 // 85.24647 // 92.9102 // 109.0107
AF4 Average Delay (ms)  : 150.84879 // 150.84879 // 0.0 // 0.0
EF  Average Delay (ms)  : 0.5478 // 0.5478 // 0.0 // 0.0
BE  Average Delay (ms)  : 771.45928 // 0.0 // 0.0 // 0.0
-----------------------------------------------
<EOF>
```

Internet Drafts and Request for Comments documents can be downloaded from the IETF homepage, namely `http://www.ietf.org/`. When searching for references we can also recommend searching on the Network Bibliography (`http://www.cs.columbia.edu/~hgs/netbib/`).

References

[1] Sigeo Aki. *Figures of Probability Density Functions in Statistics.* `http://www.sigmath.es.osaka-u.ac.jp/~aki/pdf/pdf.htm`.

[2] Shahzad Ali. *WF2Q implementation.* `http://www.cs.cmu.edu/~cheeko/wf2q+/`, 1998.

[3] Rogerio Andrade. *WWW C++ Generator.* `http://www.di.ufpe.br/~rca3/sources/ns2-rca.tgz`, 1999.

[4] Farooq M. Anjum and Leandros Tassiulas. Balanced-RED: An Algorithm to Achieve Fairness in the Internet. Technical Report CSHCN T.R. 99-9, Center For Satellite and Hybrid Communiation Networks, 1999.

[5] Jon C. R. Bennett and Hui Zhang. Hierarchical Packet Fair Queueing Algorithms. *IEEE/ACM Transactions on Networking*, 5(5):675–689, October 1997.

[6] UC Berkely, USC/ISI, LBL, and Xerox PARC. *UCB/LBNL/VINT Network Simulator - ns (version 2).* `http://www-mash.cs.berkeley.edu/ns/`, 1999.

[7] Y. Bernet, J. Binder, S. Blake, M. Carlson, B. Carpenter, S. Keshav, E. Davies, B. Ohlman, D. Verma, Z. Wang, and W. Weiss. A Framework for Differentiated Services. *Internet Draft*, Internet Engineering Task Force, March 1999. Work in progress.

[8] Y. Bernet, A. Smith, S. Blake, and D. Grossman. A Conceptual Model for Diffserv Routers. *Internet Draft*, Internet Engineering Task Force, March 2000. Work in progress.

[9] Y. Bernet, R. Yavatkar, F. Baker, B. Davie, P. Ford, B. Braden, J. Wroclawski, M. Speer, E. Felstaine, and L. Zhang. A Framework for Integrated Services Operation over Diffserv Networks. *Internet Draft*, Internet Engineering Task Force, March 2000. Work in progress.

[10] S. Blake, D. Black, M. Carlson, E. Davies, Z. Wang, and W. Weiss. *RFC 2475*: An Architecture for Differentiated Services, December 1998. Status: PROPOSED STANDARD.

[11] R. Bless and K. Wehrle. A Lower than Best-Effort Per-Hop Behavior. *Internet Draft*, Internet Engineering Task Force, September 1999.

[12] B. Braden, D. Clark, J. Crowcroft, B. Davie, S. Deering, D. Estrin, S. Floyd, V. Jacobson, G. Minshall, C. Partridge, L. Peterson, K. Ramakrishnan, S. Shenker, J. Wroclawski, and L. Zhang. *RFC 2309*: Recommendations on Queue Management and Congestion Avoidance in the Internet, April 1998. Status: INFORMATIONAL.

[13] R. Braden, D. Clark, and S. Shenker. *RFC 1633*: Integrated Services in the Internet Architecture: an Overview, June 1994. Status: INFORMATIONAL.

[14] R. Braden, Ed., L. Zhang, S. Berson, S. Herzog, and S. Jamin. *RFC 2205*: Resource ReSerVation Protocol (RSVP) — version 1 functional specification, September 1997. Status: PROPOSED STANDARD.

[15] S. Bradner. *RFC 2026*: The Internet Standards Process – Revision 3, October 1996.

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

[17] J. Diederich and M. Zitterbart. An expedited forwarding with dropping phb. *Internet Draft*, Internet Engineering Task Force, October 1999.

[18] Wu-chang Feng, Dilip Kandlur, Debanjan Saha, and Kang Shin. A Self-Configuring RED Gateway. In *Conference on Computer Communications (IEEE Infocom)*, New York, March 1999.

[19] Paul Ferguson and Geoff Huston. *Quality of Service.* Wiley Computer Publishing, 1998.

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

[21] Mukul Goyal, Arian Durresi, Raj Jain, and Chunlei Liu. Performance Analysis of Assured Forwarding. *Internet Draft*, Internet Engineering Task Force, February 2000.

[22] J. Heinanen, F. Baker, W. Weiss, and J. Wroclawski. *RFC 2597*: Assured Forwarding PHB Group, June 1999. Status: STANDARDS TRACK.

[23] J. Heinanen and R. Guerin. *RFC 2697*: A Single Rate Three Color Marker, September 1999. Status: INFORMATIONAL.

[24] J. Heinanen and R. Guerin. *RFC 2698*: A Two Rate Three Color Marker, September 1999. Status: INFORMATIONAL.

[25] Tom Henderson and Emile Sahouria. *Web Traffic Generator.* `ftp://daedalus.cs.berkeley.edu/pub/ns/httptrafficgen.tar`, 1998.

[26] Paul Hurley and Jean-Yves Le Boudec. The Alternative Best-Effort Service. *Internet Draft*, Internet Engineering Task Force, June 2000.

[27] V. Jacobson, K. Nichols, and K. Poduri. *RFC 2598*: An Expedited Forwarding PHB, June 1999. Status: STANDARDS TRACK.

[28] V. Jacobson, K. Nichols, and K. Poduri. RED in a Different Light. *Internet Draft*, Internet Engineering Task Force, September 1999.

[29] V. Jacobson, K. Nichols, and K. Poduri. The 'Virtual Wire' Behavior Aggregate. *Internet Draft*, Internet Engineering Task Force, March 2000. Work in progress.

[30] Kalevi Kilkki. *Differentiated Services for the Internet.* MacMillan Technical Publishing, 1999.

[31] Kalevi Kilkki. Interoperability PHB Group. *Internet Draft*, Internet Engineering Task Force, October 1999.

[32] Hyogon Kim, Will E. Leland, and Susan E. Thomson. Evaluation of Bandwidth Assurance Service using RED for Internet Service Differentiation. 1998.

[33] D. Lin and R. Morris. Dynamics of Random Early Dectection. *ACM Computer Communication Review*, 27(4):127–136, October 1997. ACM SIGCOMM'97, Sept. 1997.

[34] Martin May, Thomas Bonald, and Jean Bolot. Analytic Evaluation of RED Performance. In *Conference on Computer Communications (IEEE Infocom)*, Tel Aviv, Israel, March 2000.

[35] Sean Murphy. *ns-2 Differentiated Services Library Addition.* `http://www.teltec.dcu.ie/~murphys/ns-work/diffserv/index.html`, 1999.

[36] K. Nichols, S. Blake, F. Baker, and D. Black. *RFC 2474*: Definition of the Differentiated Services Field (DS Field) in the IPv4 and IPv6 Headers, December 1998.

[37] Teunis J. Ott, T. V. Lakshman, and Larry H. Wong. SRED: Stabilized RED. In *Conference on Computer Communications (IEEE Infocom)*, New York, March 1999.

[38] K. Ramakrishnan and S. Floyd. RFC 2481: A proposal to add Explicit Congestion Notification (ECN) to IP, January 1999.

[39] S. Shenker, C. Partridge, and R. Guerin. *RFC 2212*: Specification of Guaranteed Quality of Service, September 1997. Status: PROPOSED STANDARD.

[40] M. Shreedhar and G. Varghese. Efficient Fair Queueing using Deficit Round Robin. *ACM Computer Communication Review*, 25(4):231–242, October 1995.

[41] Kyle Siegrist. *The Pareto Distribution*. `http://www.math.uah.edu/stat/special/special12.html`.

[42] Galvin Silberschatz. *Operating System Concepts*. Addison-Wesley Publishing Company, 1998.

[43] Andrew S. Tanenbaum. *Computer Networks*. Prentice Hall International Editions, 1996.

[44] D. Verma. *Supporting Service Level Agreements on IP Networks*. MacMillan Technology Series, 1999.

[45] J. Wroclawski. *RFC 2211*: Specification of the Controlled-Load Network Element Service, September 1997. Status: PROPOSED STANDARD.