Facultés Universitaires Notre-Dame de la Paix
Institut d'Informatique

Providing Relative Services in IP routers

Louis SWINNEN

Travail de fin d'études réalisé en vue de l'obtention
du titre de Maître en Informatique

Année Académique 2000-2001

Abstract

Because of the development of the Internet and the price descrease of fast connections like ADSL or cable connections, new applications appear on Internet like streaming applications (video on demand for instance). These kinds of application have more requirements and can't work correctly with the default service used today inside the IP routers : first arrived, first served service.

This thesis focus on the relative services and shows how these services can be used to meet all new application requirements. Because application requirements can be usually expressed in terms of loss, delay or bandwidth, it's possible to provide several levels of services (i.e. clasees of service) that allow delay or loss sensitive applications to work correctly. The different levels of services can be achieved by using delay and loss differentiation. Delay differentiation allows to priviledge some packets inside the router (the time that some packets passed inside a queue can be tuned) while loss differentiation allows to determine which packet must be discarded first in case of congestion. Because TCP is mainly used in the Internet, it's also possible to provide bandwidth differentiation by combining the two previous differentiations.

This thesis also contains an important discussion about the implementation of such mechanisms inside a FreeBSD based router using ALTQ and several experiments are done to validate this implementation.

Résumé

Le récent développement d'Internet ainsi que la diminution des prix concernant les connexions rapides comme l'ADSL ou le cable ont fait apparaître de nouvelles applications sur Internet telles que les applications de streaming (vidéo à la demande par exemple). Ces types d'application ont plus d'exigences et ne peuvent fonctionner correctement avec le service habituellement utilisé aujourd'hui dans les routeurs IP : premier arrivé, premier servi.

Ce mémoire se concentre sur les services relatifs et montre comment ces services peuvent être utilisés pour rencontrer les nouvelles exigences des applications. Généralement, ces nouvelles exigences peuvent être exprimées en terme de perte, délai et/ou bande passante, c'est pourquoi il est possible de fournir plusieurs niveaux de services (appelé aussi classes de services) qui permettent à ces nouvelles applications de fonctionner correctement. Les différents niveaux de services sont obtenus en proposant une différenciation sur le délai et sur les pertes. La différenciation sur le délai permet de privilégier certains paquets à l'intérieur du routeur (de sorte que le temps passé par certains paquets dans le routeur peut être réglé) tandis que la différenciation sur la perte permet de déterminer quels paquets doivent d'abord être rejetés en cas de congestion. L'utilisation très généralisée de TCP sur Internet rend possible la différenciation de bande passante simplement par combinaison des deux différenciations précédentes.

Ce mémoire accorde également une part importante à l'implémentation de tels mécanismes à l'intérieur d'un routeur FreeBSD en utilisant ALTQ et propose également quelques expériences afin de valider cette implémentation.

Acknowledgments

I would like to thank all people that helped me during the writing of this thesis. In particular, I thank Prof. Olivier Bonaventure, my promoter, for his help and his precious advice.

Because he was always available for me when it was necessary, during the writing of this thesis, I thank Mr. Stefaan De Cnodder from Alcatel.

For his comments about my work, I also would like to thank Mr. Steve Uhlig from the Infonet group.

Mr. G.H. Petit, Director of the research department, gave me the opportunity to make my training course at the Corporate Research Center of Alcatel Antwerp. There, I collaborated with Mr. Stefaan De Cnodder and Mr. Goeffrey Cristallo on the subject presented here. Thank you for your help during this training course.

Without them, my stay at Antwerp would have been boring, thank you Cédric and Cristel for your good mood.

Finally, I would like to thank my whole family for all the support they always gave me.

Contents

1 Differentiated services **1**
1.1 Introduction . . . 1
1.2 Basic definitions . . . 4
1.3 Quality of Service . . . 5
1.3.1 Integrated Services . . . 5
1.3.2 Differentiated Services . . . 8
1.4 Conclusion . . . 12

2 Relative Services **15**
2.1 Introduction . . . 15
2.2 Relative Differentiation Model . . . 15
2.3 Proportional Differentiation Model . . . 16
2.3.1 Proportional delay differentiation . . . 17
2.3.2 Proportional loss differentiation . . . 17
2.4 Approximation of proportional differentiation model . . . 18
2.4.1 Proportional delay differentiation . . . 18
2.4.2 Proportional loss differentiation . . . 24
2.4.3 Proportional bandwidth differentiation . . . 30
2.5 Conclusion . . . 31

3 Implementation **33**
3.1 Introduction . . . 33
3.2 Overview of FreeBSD . . . 33
3.3 Description of FreeBSD . . . 34
3.3.1 Main features . . . 34
3.3.2 TCP/IP network layer . . . 37
3.4 Description of ALTQ . . . 42
3.4.1 Architecture . . . 42
3.4.2 Working of ALTQ . . . 44
3.5 Extensions to ALTQ . . . 45
3.5.1 Architecture of the solution . . . 45
3.5.2 Implementation of WTP scheduler . . . 46
3.5.3 Implementation of WRED . . . 55
3.5.4 Statistical Variables . . . 58
3.6 Conclusion . . . 60
3.6.1 Limitations of this implementation . . . 60

4 Experiments **63**
4.1 Introduction 63
4.2 Traffic generator 64
4.3 Network configuration 64
4.4 Experiment results 67
4.4.1 Background 67
4.4.2 Heavy-load condition 67
4.4.3 Equal distribution (TCP) 70
4.4.4 Unequal distribution 76
4.5 Conclusion 80

Conclusion **81**

A Implementation **83**
A.1 altq_wtp.h 87
A.2 altq_wtp_opt.h 90
A.3 altq_wtp_util.h 91
A.4 altq_localq.c 92
A.5 altq_wtp_wred.h 111
A.6 altq_wtp_wred.c 114
A.7 wtpd.c 121
A.8 wtpstat.c 132

Chapter 1

Differentiated services

1.1 Introduction

The development of Internet is now a reality. During several years, Internet was used only by the military and universities for research. Today Internet connects corporations, universities, administrations and personal users, and is used to provide more and more services (like e-commerce, e-mail, data exchange, video on demand, ...).

Internet is composed of many local and wide area networks (called also *autonomous systems*) connected together to build a worldwide network. An autonomous system (or *AS*) is a network or a set of networks under the same authority. For instance, Belnet forms an AS composed with all university networks (see figure 1.1).

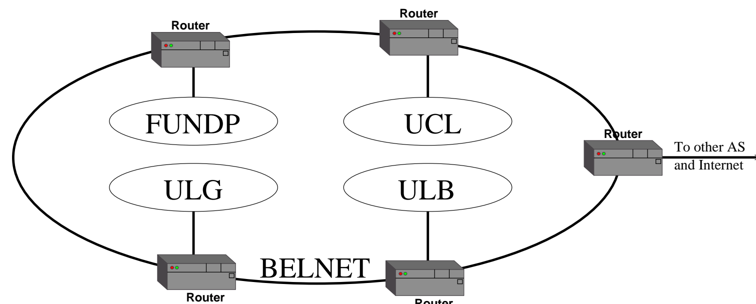


Figure 1.1: Example of autonomous system

The model that is used to allow two computers to communicate together is a modified version of the OSI (Open System Interconnection) model. This theoretical model supposes five layers (as can be seen on figure 1.2) :

Physical layer The physical layer allows to send bits on a given physical media (like twisted pair, coaxial cable or optical fiber) through a given interface (i.e. encodes the bit stream to a form compatible with the selected media and that can be decoded without ambiguity). Each media has a lot of characteristics that will limit the performances of the communication.

Data link layer The data-link layer allows a data exchange between two computers. Based on an address, it's possible to exchange structured information using the first layer. Several types exist, some allow to exchange information in a reliable way (by using acknowledgement when data is received by the destination) while others not. Exchanged information between two data-link layers is called frame.

Network layer The network layer allows to connect more than two computers together. Indeed, many computers with different data link layers can be connected together. Several mechanisms (called routing protocols) exist to find the destination host and to determine the path to transmit the information. This layer relies on the previous layer to provide its services (i.e. an unreliable service is always assumed because of possible differences between successive data link layers). Exchanged information between two network layers is called packet.

Transport layer The transport layer allows two programs on different hosts to communicate together. Mainly, two kinds of transmissions can be offered : reliable and connection oriented or unreliable and connection less service. The first can be compared with a common phone communication : the communication must be established and once the communication is terminated, this connection must be closed. Moreover, in this case, the caller knows if his communication is received. The second can be compared with the post service : a letter is sent to a destination address but the sender has no guarantees that his letter is arrived or not. Exchanged information between two transport layers is called Transport Protocol Data Unit (or *TPDU*).

Application layer This layer allows to provide new network applications to the final user. Many network applications are used today on Internet like e-mail or some others are very popular like WWW, FTP, . . . This layer is used to provide new services and relies on the previous layer to transmit all needed information. Exchanged information between two application layers is called Application Protocol Data Unit (or *APDU*)

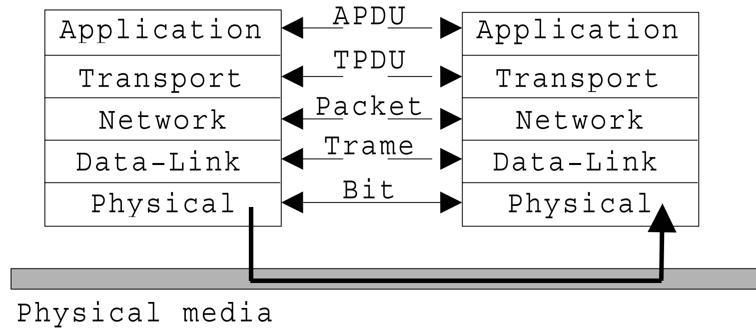


Figure 1.2: A modified version of the OSI model

As part of this thesis, the most important layers are the network, transport and application layers. So when an application sends data to another application on another host, it asks to the underlying layers, especially the transport layer to send its information. The

transport layer splits information into blocks that the network layer can send. If it is a reliable transmission, all lost or wrong packets are sent again. The network layer finds the path to reach the destination and sends packet to the destination (by using the data-link and physical layers). The network layer of the destination host receives from underlying layers several packets and forwards these packets to the transport layer. The transport layer sends the information to the selected application.

The data exchange on Internet works today on the concept *first come, first served* (FCFS) with no distinction between users or applications. Several types of network applications can be observed :

Elastic applications This type of application uses available resources. When resources are not available, this kind of application can wait without serious consequences until needed resources become available. This kind of application can work independently of available resources (the application waits for resources if they are not available). For instance : web traffic, X-Window, NFS, RPC, e-mail, ...

Streaming applications (or real-time applications) This type of application requires a minimum of resources to work. Without this minimum, the application doesn't work. This kind of application can't wait when the resources are not available. For instance : Voice over IP, video on demand, network games, ...

The default service (first come, first served), called *best effort*, is sufficient for elastic applications. Today, more and more providers propose real-time applications and the default service is not adapted to these new application requirements (i.e. minimum of resources). These requirements are usually expressed in terms of delay (i.e. maximum delay), loss rate (i.e. maximum loss rate) and finally bandwidth (i.e. minimum bandwidth).

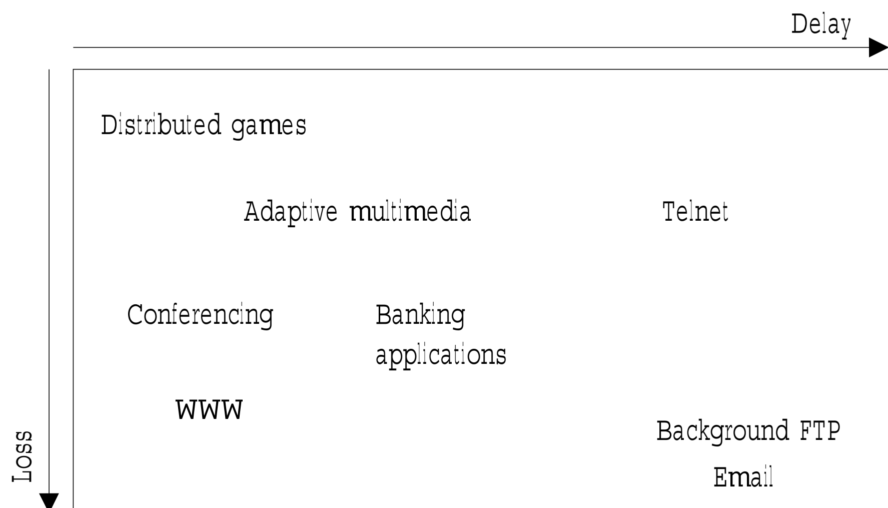


Figure 1.3: Loss and delay requirements for Internet applications [27]

On figure 1.3 (from [27]), a taxonomy of application requirements is presented in terms

of delay and loss : streaming applications (network game, conferencing and adaptive multimedia on the figure) require mainly low delay and loss to work correctly while elastic applications (like e-mail or web traffic) can tolerate high delay.

The idea of providing more services was old because already inside the IP protocol, a field, called *type of service*, was reserved to specify the required service for the information contained inside the packet. But this characteristic was not often used. Today, IETF[1] has two approaches to provide more services : *Integrated Services* and *Differentiated Services*.

This chapter is organized as follows : some basic definitions are given first, a description of integrated service and differentiated services is proposed after.

1.2 Basic definitions

IP Packet A packet is the way used to transport information through an IP network. A packet contains two parts : the header and the data. The header is structured into several fields (see figure 1.4), the most important fields are :

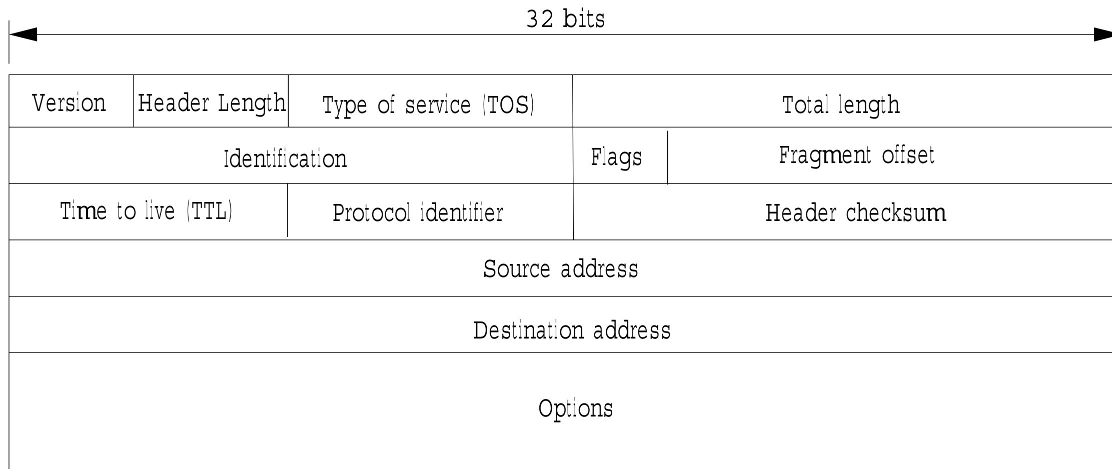


Figure 1.4: IP Header description [17]

- *source and destination addresses :* references the source and destination host of this packet
- *type of service :* indicates the required level of service.
- *protocol number :* informs of the transport protocol (TCP or UDP) used.

The maximum size for packets are 64 kilobytes with minimum 20 bytes for the header.

Protocol A protocol is a description of all valid commands. It is used to describe how computers can communicate together (i.e. all messages exchanged to allow a communication between two computers or more).

[1]Internet Engineering Task Force attempt to standardize all protocols and mechanisms used on the Internet

User Datagram Protocol (UDP) This is a transport protocol (i.e. it works over IP to provide its service) that offers a connectionless and unreliable transmission : it is useful when retransmission of lost packets is too expensive (takes too much time).

Transmission Control Protocol (TCP) This transport protocol is used to provide a connection oriented and reliable transmission (by using retransmission if necessary). It provides complex congestion control algorithm to avoid congestions by reducing its sending rate if packet drop occurs.

Flow of packets A set of IP packets that share a common property like source and destination address and sometimes same protocol identifier and the same port numbers.

1.3 Quality of Service

Quality of Service attempts to provide more architectures that can meet new application requirements. Two different architectures have already been defined : the Integrated services and Differentiated services.

Integrated Services The integrated services [4] [11] (also called *IntServ* or *IS*) attempt to provide strong (i.e. deterministic) guarantees to each individual flow : a flow is defined as a set of packets [4] “that results from a single user activity and requires the same guarantees”. Each flow asks for a level of service (a given amount of resources like buffer space or bandwidth) and the IS router decides whether the requested service can be guaranteed or not. The level of service is usually expressed in terms of maximum end-to-end delay, loss rate or minimum service rate.

Differentiated Services The second architecture is the differentiated services [22] [1]. They are based on the definition of several levels of services (also called *classes of service*). According to the traffic contract (also called *SLA*) signed between an user and the network provider, the incoming traffic will be distributed among the defined classes to obtain the agreed service.

These two architectures will be described in details in the sequel.

1.3.1 Integrated Services

The integrated services focus on the end-to-end guarantees : they attempt to provide guarantees along the path follow by the traffic (from the source to the destination through all routers). That is why IS uses a new protocol to inform the destination host and all routers along the path that a new reservation is requested. This protocol is the Resources reservation protocol (also called *RSVP*). When a new reservation is requested by a host $Source$ (depicted on figure 1.5), a RSVP message $PATH$ is sent such that all routers along the path knows that the host $Source$ would like make a reservation of a given amount of traffic to the $Destination$ host. A $PATH$ message contains information about the amount of traffic that will be send but no reservation is made now : it's only an invitation. This message is also used to establish the route between the source and destination host.

For instance, when the $PATH$ message arrives at the router $R1$ (figure 1.5), it selects the next hop based on its routing information ($R2$ in this case), then it creates a new state for this flow with the following information : identification of this flow, senders of this $PATH$ message and finally, it sends the $PATH$ message to the selected hop $R2$. When the $PATH$ message arrives at the destination host, the destination will accept or reject this new reservation. If it accepts, a message $RESV$ is sent to the sender of the received $PATH$ message ($R3$ in this case). This message contains all information needed and when a router receives a $RESV$ message, it proceeds to make the reservation (or refuses it if not enough resources are available). It is necessary that the $RESV$ messages follow exactly the same route as the $PATH$ messages to confirm the reservation to all routers from the source to the destination (i.e. because of dynamic feature of routing protocol, the route followed by packets can be different according to the packet direction). The routers can find the correct route because they have saved needed information inside the associated flow state. When the source receives the $RESV$ message, it knows that the reservation is done. If a router can't make the reservation, a special message is sent to inform that the reservation failed.

The integrated services architecture has already two services defined : the guaranteed service and controlled load :

Guaranteed Service Attempts to simulate a physical link between the source and destination host. It provides, for conforming flows, firm guarantees (with a bounded end-to-end delay and no losses caused by congestion).

Controlled load Attempts to provide for conforming flows a service similar to the best-effort service when no congestion occurs and the network is lightly loaded (i.e. with low delay and low loss rate).

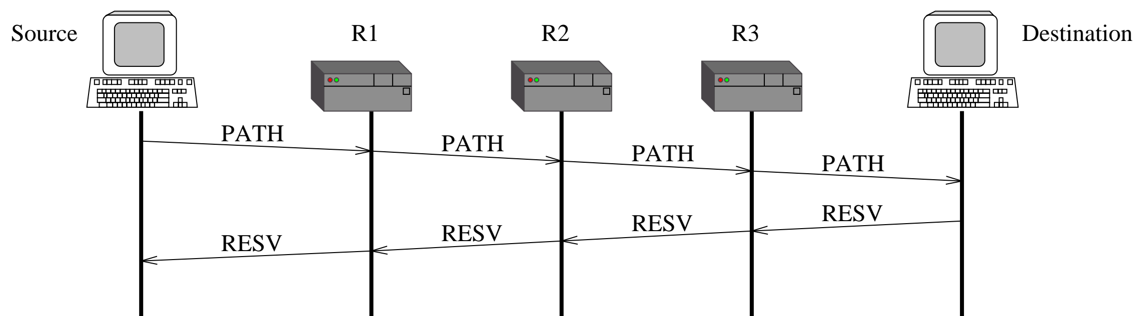


Figure 1.5: Example of resource reservation

The IS architecture relies on special routers (called *IS routers*) composed of many building blocks (as can be seen on figure 1.6). The specific IS blocks are the reservation setup agent, the admission control, the classifier and the packet scheduler.

Routing agent The routing agent implements a common routing protocol (for instance inter-domain BGP routing protocol or intra-domain OSPF routing protocol) and

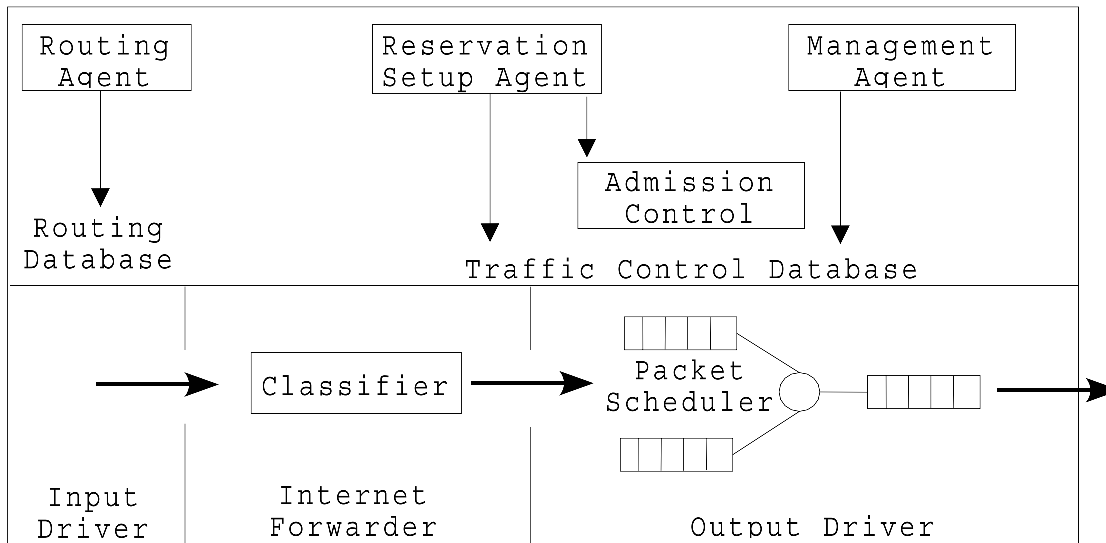


Figure 1.6: Implementation reference model for IS router [4]

constructs a routing database used to select the good path for a packet to reach its destination.

Packet scheduler The packet scheduler uses a set of queues to manage the traffic. According to a specific scheduling algorithm, the packet scheduler selects among queues the packet that has to be sent on the output link.

Classifier Usually based on information included inside the header of the incoming packet, the classifier maps the packet (or flow) to a selected class. All packets from the same class receive the same treatment and a class can hold one or many flows. Finally, the classification is local : inside another router, this packet (or flow) can be classified differently.

Admission control The admission control algorithm must decide if an incoming flow can be guaranteed. Before accepting a new flow, this algorithm checks if enough resources are available to assure that this flow can be accepted without consequences for all guaranteed flows. Moreover, this algorithm has to check if all flows respect their promises : if one flow exceeds its guarantee : other mechanisms are used to enforce the flow to match with its guarantees. These mechanisms : shaping or policing can delay or discard some packets of this flow if necessary.

Reservation Setup agent This agent implements the RSVP protocol described above.

This approach appears to be very interesting to solve the problem of streaming application (i.e. a minimum of forwarding resource must be available otherwise the application doesn't work) but it has the following problem (known as the *scalability problem*) : to accept a new request, each router must maintain a state **for each flow**. Because the resources (memory and CPU power for instance) inside a router are limited and the number

of flows can be very high so the maximum number of flows that a router can support would be an important limitation. A second limitation is the need to support RSVP protocol (all routers on Internet can't be changed at the same time to support this new protocol).

1.3.2 Differentiated Services

The second approach is the differentiated services [22] [1] (also called *DiffServ* or *DS*). DiffServ proposes to *classify* incoming traffic into *a limited number of classes of services* according to the *service level agreement* signed between a client and the service provider. A network that is compliant with the DiffServ model is called a DS-Domain.

The service level agreement (or SLA) is a contract that specifies [1] "the forwarding service (i.e. the class of service) that a customer should receive". A class of service corresponds to a level of service (i.e. a given amount of resources) and each level of service is associated with a given value called *DiffServ CodePoint (DSCP)*. This value indicates the requested service : the expected behaviour of the router (also called Per-Hop Behaviour or PHB). Many PHBs have already been standardized by the IETF. The DSCP value is set by the user or by a special router (edge router). All flows from the incoming traffic requiring the same service are aggregated together into the selected class of service (the class that corresponds with the forwarding requirements).

DiffServ has been developed according two models, the absolute and relative services :

Absolute Services Like IntServ, attempt to provide strong guarantees but without per-flow state and [11] : "with only some semi-static resources reservations (by the use of centralized entity), instead of a dynamic resource reservation protocol". This approach relies on a semi-static resource reservation and requires admission control mechanism.

Relative Services The idea is to have several classes that are ordered such that a class i provides a "better" service than previous classes in terms of local per-hop metrics (from [11]). The service is usually expressed in terms of queueing delay and packet losses. It doesn't require new signaling protocols or complex admission control mechanisms. Finally, no state must be maintained with each individual flow. Better service usually means lower delay, lower loss rate or higher bandwidth.

An important distinction with regard to the Integrated Services is the use of two types of routers (see figure 1.7). DiffServ proposes to have routers located in the core of the domain while others are located at the border of the domain. The idea is to keep all complexity at the border of the DS domain.

Core router High-speed router located in the core of the DS domain and that supports several level of services, provides an adequate behaviour[2] according to the DSCP value.

Edge router An edge router is located at the border of the DS domain, it supports the forwarding mechanisms implemented inside a core router and provides other more complex mechanisms to analyze and performs traffic conditioning if necessary.

[2] also called Per-Hop Behaviour or PHB

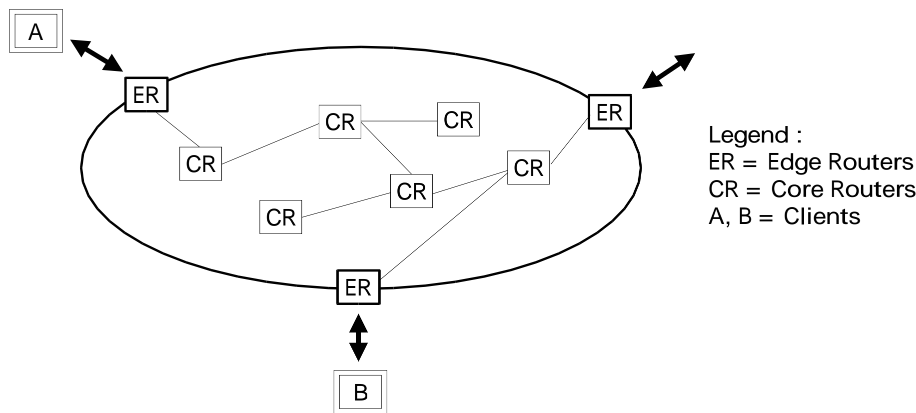


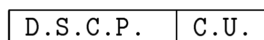
Figure 1.7: Example of DS domain

The traffic conditioning uses some mechanisms to assure that the incoming traffic is conforming to the SLA. Traffic conditioning includes the following mechanisms :

- *Metering :* Measuring several properties of a traffic stream ([1]).
- *Marking :* Setting/Updating the DSCP value (i.e. reduces the requested service). Because the core router only checks this value to decide the good behaviour, the re-marking is sufficient to reduce the provided service.
- *Shaping :* Delaying packets to enforce the traffic stream to conform to the SLA.
- *Policing :* Discarding packets to enforce the traffic stream to conform to the SLA.

To keep the complexity at the border, some mechanisms are only present inside an edge DS router while other are available in all DS router : metering, marking , shaping and policing mechanisms are often present in a DS edge router only.

Because the field type of service was not often used, the IETF proposes to replace the TOS field with the DSCP value. The TOS byte indicated the packet precedence (i.e. the packet priority) and other information that specifies if the packet or corresponding flow requires a low delay, a high throughput or a high reliability ([17]). It was redefined as follows (see [22]) :



D.S.C.P.	C.U.

D.S.C.P. Differentiated Service Code Point (6 bits)

C.U. Currently Unused (but currently assigned to be used for explicit congestion notification). This is not a part of DiffServ (2 bits).

Several arrangements have been made to provide a limited compatibility with the old TOS byte specification.

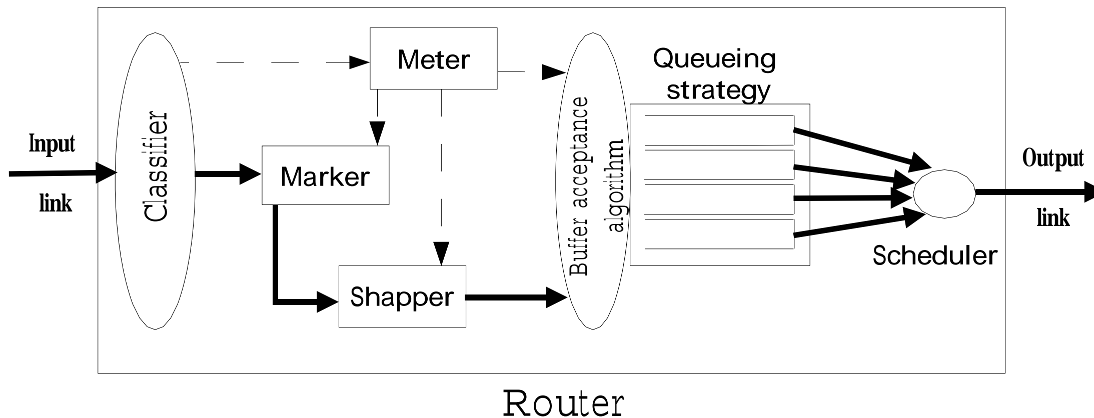


Figure 1.8: Logical view of a router

Description of router mechanisms

As shown on figure 1.8, a DS router has the following functional building blocks (arrows in bold represent the path that packets follow inside the router) :

Classifier Based on some information included inside the packet, the classifier can determine the flows to which the packet belongs. The classifier can work at several levels :

- *Network layer flow :* Two packets belong to the same flow if they have the same source and destination addresses.
- *Transport layer flow :* Two packets belong to the same flow if, in addition to the network layer requirements, they have the same protocol identifier, source and destination port number.
- *Application layer flow :* Two packets belong to the same flow if, on top of transport layer needs, they have same application-dependent information. This type of classifier is very expensive (in terms of required resource) because it must inspect the payload (i.e. the data) of the packet to determine the correct flow.

The classifier can also use other rules to determine the flows.

Meter – Marker – Shaper They implement respectively the metering, marking and shaping function described above.

Buffer acceptance algorithm The buffer acceptance algorithm relies on the meter to decide whether incoming packet must be discarded or can be accepted inside the buffer (policing function). To take a decision, the buffer acceptance algorithm also takes into account the buffer occupancy and the DSCP included in the packet.

Scheduler Like the packet scheduler in the IntServ model, the function of the scheduler is to decide the next packet to transmit on the output link first based on computed

packet priority. The computation of the packet priority will depend on the implementation of the scheduler algorithm.

Queueing strategy It is a part of the scheduler, it describes how packets are organized inside the buffer and what type of queues are used (FIFO, . . .).

As already described above, some blocks are only implemented in an edge router while others are present in all types of routers : an edge router determines the flow of a packet, marks the packet with a good DSCP value and delays, sends or discards the packet if necessary while a core router has a "light-classifier" (that only checks the marking to determine the correct behaviour for this flow) and, usually, no shaping or policing mechanisms.

Per-Hop Behaviour

Two PHBs have already been standardized : the Assured Forwarding PHB [16] and the Expedited Forwarding [18]. The IETF only exposes some guidelines and doesn't propose any implementation of these PHBs.

- *Assured Forwarding PHB (AF)*

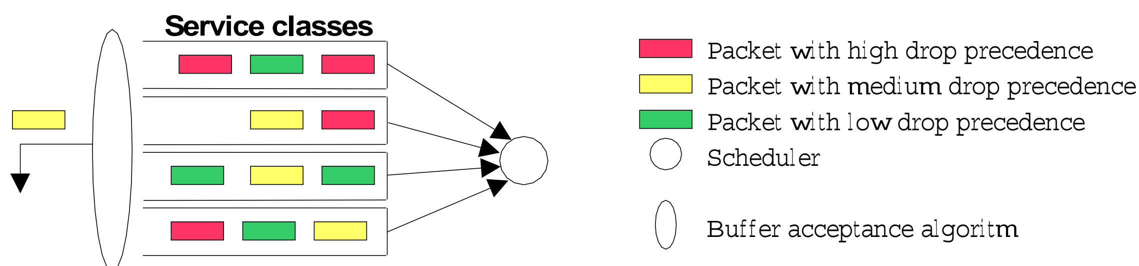


Figure 1.9: Possible implementation of AF PHB

This mechanism aims to [16] assure that IP packets will be forwarded through the network with a high probability as long as the traffic does not exceed the predetermined profile. That's why four classes are defined : these classes provide different levels of service. Inside each class, three drop precedences are defined and packets are discarded according to their drop precedences. To identify the requested level of service (the queue and the drop precedence), each packet is marked (by the source or by an edge router) with a correct DSCP value. Most important rules are :

  - Packets from one class must be forwarded independently from packets in another class i.e. packets from different classes must not be aggregated together. This rule assures the independence between classes : packets are forwarded according to the requested service and the selected drop precedence independently of the load of the other classes.
  - The router associates with each class a defined quantity of resources (buffer space, bandwidth, . . .). Each class should receive the configured service rate over small and large time scales.

  - Packets from the same transport flows must not be reordered.

  A possible implementation of these guidelines is depicted in the figure 1.9. Four classes of service are defined with three drop precedences. The buffer acceptance algorithm discards packet when it is needed while the scheduler is configured to provide the expected level of service.

- *Expedited Forwarding PHB (EF)*

  This mechanism proposes [18] to have one special class that is served such that the resulted loss, latency and jitter are low. To provide such guarantees, this class must experience low queueing delays. Low queueing delay means that :

  - The departure rate of this class must be independent of the load of other classes
  - The arrival rate of this class must be less than its departure rate.

  It's also important to limit the traffic for the EF class to avoid a starvation effect with the other classes. This PHB can be used to emulate a leased line between two hosts through a network.

  For these two PHBs, the IETF has proposed several DSCP values to give the configuration of the router easier : the default configuration of all routers is the same. Many PHBs can be combined together to proposed extended service (i.e. combining of EF and AF for instance).

1.4 Conclusion

This chapter has presented the current working of Internet (first come, first served) and proposed two evolution models : the Integrated Services and Differentiated Services. A summary of the most important differences between IntServ and DiffServ is presented on table 1.1.

The Integrated services proposes deterministic guarantees by requesting resource reservation. This can be achieved by using new mechanisms : the admission control mechanism and the resource reservation protocol. The admission control mechanism is needed to evaluate the impact of new reservations on existing guarantees and available resources while the resource reservation protocol is used to proceed the reservation inside routers along the path. The Differentiated services proposes to classify incoming traffic into several classes of service. This classification can be done by the user or by a special router located at the border of the DS domain. Indeed, two types of router are proposed : core routers and edge routers. Core routers are located in the core of the DS domain and are constructed to be fast while edge router are located at the border of the DS domain and implement more complex mechanisms.

This table shows the main problems of the IntServ architecture : states in IS routers are per-flow while the state in DS routers are per-aggregate (so are limited by the number of classes). This distinction is the cause of the scalability problem and is the main limitation of the IntServ model.

	Integrated Services	**Differentiated Services**
Granularity of service differentiation	Individual flow	Aggregate of flows
State in routers (e.g. scheduling, buffer management)	Per-flow	Per-aggregate
Traffic classification basis	Several header fields	The DS field (6 bits) of the IP header *for core routers* Several header fields *for edge routers*
Type of service differentiation	Deterministic or statistical guarantees	Absolute or relative assurances
Admission control	Required	Required for absolute differentiation only
Signalling protocol	Required (RSVP)	Not required for relative schemes Absolute shemes need semi-static reservations
Coordination for service differentiation	End-to-end	Local (per-hop)
Scalability	Limited by number of flows	Limited by number of classes of service

Table 1.1: A comparison of the IntServ and DiffServ architectures [11]

In the DiffServ architecture, the relative services appears to be less complex than the absolute services because no admission control algorithm and no resource reservation mechanism are required to support it.

That is why IntServ architecture appears not really adapted to Internet because of scalability problem : states inside an IS router are per-flow, the maximum number of flows that an IS router can support is the main limitation. DiffServ architecture seems to be very interesting particularly the relative model (because the absolute service relies on semi-static resources reservation) for incremental deployment. Indeed, because of local coordination, DiffServ can be deployed incrementally and provide (local) differentiation while IntServ must be deployed completely to provide guarantees.

This thesis is structured as follows : the next chapter discuss about the relative services and which mechanisms can be used, the chapter three describes the implementation of selected mechanisms and finally the chapter four proposes experiments to validate and evaluate the proposed implementation. The whole implementation is included in the appendix A.

Chapter 2

Relative Services

2.1 Introduction

The previous chapter has introduced several new architectures to replace the default first come first serve service inside the routers. Architecture like Integrated Services and Differentiated services have been explained and the Differentiated Services appear to be very interesting because they have not the scalability problem (i.e. with the integrated services, a router must maintain a state per flow but the number of flows can be very high and that leads to an important limitation).

The Differentiated services architecture proposes two models : relative and absolute services. Relative services seem to be easier to implement (they do not require semi-static resources reservation mechanism) : they are based on the idea of classes of services. Indeed, several classes of service are defined and each class is associated to a defined level of service (i.e. usually expressed in terms of delay, loss and bandwidth). Classes are ordered such that the class i provides a better service (i.e. higher level of service) than the previous classes.

This chapter will expose some relative differentiation models first and then, the proportional differentiation model is described. This model was used for delay and loss differentiation. After existing mechanisms that can meet the requirements of the proportional differentiation model are discussed. Finally the combination of these two differentiations to obtain a proportional bandwidth differentiation is exposed.

2.2 Relative Differentiation Model

Several models exist to provide differentiation. A good model must be *predictable* and *controllable* : *predictable* means that the model is consistent (i.e. a high class always receives a better service than a low class) and not-dependent of the variations of the class load; *controllable* means that the network operator can tune the differentiation according to his criteria.

Strict Prioritization This model proposes to serve highest not-empty class first (i.e. for delay differentiation) and when packets must be dropped (i.e. for loss differentiation), it proposes to discard packets from the lowest not-empty class first. This model is not

controllable : the quality spacing between classes cannot be tuned. Moreover, higher classes can be served continuously if no restriction is placed with the consequence that lower classes can never be served (i.e. starvation effect).

Price Differentiation The Paris Metro Pricing [23] proposes that a differentiation can be achieved by using pricing (higher prices lead to lower loads). This model makes the distinction between users that can pay for a high service level and users that can't pay. It is not consistent because lower class can experience a lower load than a higher class if many 'rich' users are active at the same time. The conclusion is that this model is unpredictable : the service level of a given class depend of the current class load.

Capacity Differentiation The capacity differentiation model proposes to allocate a larger amount of forwarding resources to higher classes according to the long-term expected load in each class. This approach has an important problem : sometimes higher classes can obtain a worse service than lower classes because of deviation between the short-term and long-term load. Indeed, the reason is that "forwarding resources allocated to each class do not follow the actual class load variation" (from [11]).

Proportional Differentiation This model proposes to have a proportional differentiation between classes. This differentiation can be tuned by the network operator such that this model is controllable. On the other hand, this model assures that high classes receive a better service than lower classes according to the proportional parameters defined by the operator.

The proportional differentiation models appears to have all required qualities : it is predictable and controllable. It will be described in details in the sequel.

2.3 Proportional Differentiation Model

This model was described by C. Dovrolis first. The main idea is that [11] "certain class performance metrics should be proportional to the *differentiation parameter* that the network operator chooses". Network performance can be measured in terms of queueing delay and loss ratio. In this case, better class performance means lower queueing delays and/or lower loss probabilities.

Assume that $\bar{q}_i(t, t+\tau)$ is the measure of performance for class i during the time interval $[t, t+\tau], \tau > 0$. Classes are ordered such that a class i receives a better service than the previous classes ($< i$). The following relation is defined between the classes :

$$\frac{\bar{q}_i(t, t+\tau)}{\bar{q}_j(t, t+\tau)} = \frac{c_i}{c_j}$$

where $c_1 < c_2 < \ldots < c_N$ (if N classes are defined) are the *Quality Differentiation Parameters (QDPs)*. The relations $\frac{c_i}{c_j}$ between classes are called *quality ratios*. They remain fixed even if the quality level of each class (i.e. the class performance) changes according to the class load. These ratios express the proportional relation between classes. This model does

not propose strong guarantees but only the assurance that a class will receive a service that will be a fraction of the service proposed by another class.

For instance, assume that three classes are defined with the following QDPs : $c_1 = 1$, $c_2 = 2$ and $c_3 = 3$, it appears that :

$$\frac{\bar{q}_1(t,t+\tau)}{\bar{q}_2(t,t+\tau)} = \frac{1}{2} \Leftrightarrow \bar{q}_1(t,t+\tau) = \frac{1}{2}\bar{q}_2(t,t+\tau)$$
$$\frac{\bar{q}_1(t,t+\tau)}{\bar{q}_3(t,t+\tau)} = \frac{1}{3} \Leftrightarrow \bar{q}_1(t,t+\tau) = \frac{1}{3}\bar{q}_3(t,t+\tau)$$
$$\frac{\bar{q}_2(t,t+\tau)}{\bar{q}_3(t,t+\tau)} = \frac{2}{3} \Leftrightarrow \bar{q}_2(t,t+\tau) = \frac{2}{3}\bar{q}_3(t,t+\tau)$$

If class 3 receives a performance of 6 during the time interval $[t, t+\tau]$, class 2 should receive a performance of 4 while the class 1 receives a performance of 2 during the same time interval. In summary, the performance of class 3 will be three times better than the performance of class 1 and the performance of class 2 will be two times better than the performance of class 1.

This model can be used for delay differentiation and loss-rate differentiation.

2.3.1 Proportional delay differentiation

[13] proposes to take into account the average queueing delay metric of a class to provide a proportional delay differentiation. The more queueing delays are high, the more provided service is poor. To use the proportional differentiation model in context of queueing delays, the following relation is set :

$$\bar{q}_i(t,t+\tau) = \frac{1}{\bar{d}_i(t,t+\tau)}$$

where $\bar{d}_i(t,t+\tau)$ is the average queueing delay of the packets belonging to the class i that left during the time period $(t, t+\tau)$. If no packets of class i were served during this interval, $\bar{d}_i(t,t+\tau)$ is not defined. The relation between all pairs (that are defined) of classes is :

$$\frac{\bar{d}_i(t,t+\tau)}{\bar{d}_j(t,t+\tau)} = \frac{\delta_i}{\delta_j}$$

where $\delta_1 > \delta_2 > \ldots > \delta_N > 0$, are the *Delay Differentiation Parameters (DDPs)* (if N classes are defined), ordered such that a class i provides a better service than a previous class ($< i$).

2.3.2 Proportional loss differentiation

This model can also be used to provide a proportional loss-rate differentiation [12]. To make the loss differentiation, the average loss rate of each class will be evaluated : a low loss ratio indicates that the class provides a good service (in terms of loss differentiation only). The following relation is observed :

$$\bar{q}_i(t,t+\tau) = \frac{1}{\bar{l}_i(t,t+\tau)}$$

where $\bar{l}_i(t, t+\tau)$ is [11] "the fraction of class-i packets that were backlogged (i.e. not-empty) at time t or that arrived during the interval $(t, t+\tau)$, and that were dropped in this same time interval". Relations between two classes are defined as follows :

$$\frac{\bar{l}_i(t, t+\tau)}{\bar{l}_j(t, t+\tau)} = \frac{\sigma_i}{\sigma_j}$$

with $\sigma_1 > \sigma_2 > \ldots > \sigma_N$ are the *loss-rate differentiation parameter (LDPs)*, and ordered such that a class i offers a lower loss ratio than the previous classes.

2.4 Approximation of proportional differentiation model

Several mechanisms exist to realize differentiation in terms of delay or loss-rate. These mechanisms will be detailed to show how they approximate the proportional model.

2.4.1 Proportional delay differentiation

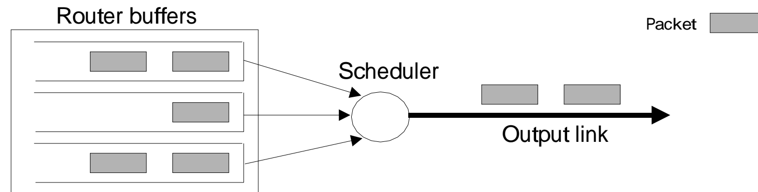


Figure 2.1: Scheduler mechanism

Inside a router, the scheduler is the mechanism (see figure 2.1) that can provide delay differentiation because it selects the next packet that is sent on the output link. By this selection, the scheduler alters the packet queueing delay and can provide a differentiation : the scheduler can privilege delay-sensitive applications like real-time applications if packets from this application are selected before packets from an elastic application.

To achieve the correct differentiation, the DDPs parameters $\delta_1 > \delta_2 > \ldots > \delta_N$ (from the proportional delay differentiation model) will be used. Several scheduler mechanisms will be described now.

Generalized Processor Sharing

Generalized Processor Sharing (GPS) [25] is the first model that provides delay differentiation. The basic idea behind this scheduler is that it considers traffic like a fluid flow. This assumption is very interesting because fluid flows are infinitively divisible. It serves sessions (i.e. classes) at a fixed rate r and can provide delay differentiation between sessions that are active. According to the sequence of positive real numbers $\Phi_1, \ldots, \Phi_N$ (if N sessions are active), GPS is defined as follows (for sessions continuously not-empty during the interval $[\tau, t]$) :

$$\frac{S_i(\tau, t)}{S_j(\tau, t)} \geq \frac{\Phi_i}{\Phi_j} \cdots j = 1, 2, \ldots, N \tag{2.1}$$

where $S_i(\tau, t)$ is the amount of traffic from session i sent during the interval $[\tau, t]$.

The following relation is obtained if all sessions j are summed ($\forall i \in [1, \ldots, N] : \Phi_i, S_i \geq 0$) :

$$S_i(\tau, t) \sum_j \Phi_j \geq (t - \tau) r \Phi_i$$

And the guaranteed rate is the amount of traffic processed during the interval $[\tau, t]$ over the time needed to transmit this traffic :

$$g_i = \frac{\Phi_i}{\sum_j \Phi_j} * r \leq \frac{S_i(\tau, t)}{t - \tau} \quad (2.2)$$

This scheduler has the following advantages :

- Assume that r_i is the average rate of session i. While $r_i \leq g_i$, the throughput will remain independent of the demands of the other sessions (protection between sessions).
- Unlike traditional schedulers (FCFS, LCFS, ...), the delay guarantee of an arriving bit from session i depends only of the queue length of this session.
- The numbers Φ_i allow to tune the behaviour of the scheduler. For instance, if all Φ_i are equal, all sessions will receive the same guarantees.

Because it considers traffic like a fluid flow, this scheduler can't be implemented (it's an ideal scheduler). The traffic from an IP network is packet based : a packet is the lowest information unit that a scheduler can send unlike GPS that considers traffic like a flow (i.e. the transmission can be "stopped" anywhere).

But several schedulers attempts to simulate the GPS behaviour and are packet-based. These schedulers will be studied now.

Weighted Fair Queuing (WFQ)

The first approximation of GPS is the Weighted Fair Queueing scheduler [10] also known as Packet Generalized Processor Sharing [25]. This scheduler computes the time at which the GPS scheduler would have served the packet (called *finish time*). The WFQ scheduler is characterized by $W_1, \ldots, W_N$ the weights associated with the N queues (i.e. sessions). These weights are equivalent to the $\Phi_1, \ldots, \Phi_N$ in the GPS scheduler, and are useful to tune the scheduler behaviour. WFQ guarantees, like GPS, the following rate :

$$g_i = \frac{W_i}{\sum_j W_j} * r \quad (2.3)$$

where : W_i Weight of the queue i
r Rate of the output link

The scheduler selects the packet with the smallest finish time and sends it on the output link first.

The computation of the finish time is the key of this scheduler. The first way to compute the finish time appears to be :

$$F_i^k = a_i^k + \frac{L_i^k}{g_i}$$

where F_i^k , a_i^k and L_i^k are, respectively, the finish time, the arrival time, and the size of packet k in the queue i. In other words, the finish time of a packet is its arrival time plus the time needed to transmit this packet. This approach has the following problem : if the queue holds already several packets, the finish time will not be correct because it doesn't take into account the queueing delay. The finish time is corrected as follows :

$$F_i^k = max(a_i^k, F_i^{k-1}) + \frac{L_i^k}{g_i}$$

This relation takes into account the queueing delay but there exists another problem : when a queue is empty and a packet arrives, its finish time F_i^{k-1} can be less than the finish time of a queue with a higher weight that holds packets. The reason is that the computation of finish time takes the maximum between the arrival time and the last finish time of this queue. But, this last finish time can be zero or less than all current finish time. To correct that, a new function is used to evaluate the arrival time and the finish time is :

$$\begin{aligned} F_i^0 &= 0 \\ F_i^k &= max\Big(v(a_i^k), F_i^{k-1}\Big) + \frac{L_i^k}{g_i} \end{aligned} \tag{2.4}$$

Where : $v(t)$ computes the *virtual finish time*. It assures that an empty queue will not build up credits. It is computed as follows :

$$\begin{aligned} v(0) &= 0 \\ v(t) &= v(l(t)) + (t - l(t)) * \frac{1}{\sum_{i \in B(t)} \frac{W_i}{\sum_{j=1}^{N} W_j}} \end{aligned} \tag{2.5}$$

where : $B(t)$ is the set of not-empty queues at time t

$l(t)$ is the time instant in the past when the set of not-empty queues changed for the last time (on packet arrival and departure) :

$$l(t) = max\{s : B(s) \neq B(t)\}, \quad \forall t : t > l(t) \tag{2.6}$$

But, WFQ has the following problems and drawbacks :

- Last value of $v(l(t))$ and $l(t)$ must be saved when they change (on packet arrival or departure). To avoid the need to check, per packet, the finish time, the scheduler can order packets according to their finish time ;
- This algorithm is static : it doesn't take into account the current queue occupancy;
- Computation of finish time can be complex.

Virtual Spacing

This scheduler is also an approximation of GPS ([15]) : the guaranteed rate g_i is computed in the same way as for GPS and WFQ. This scheduler associates, like WFQ, a finish time

with each packet and serves these packets according to their finish time. It attempts to reduce the complexity of the finish time computation (proposed in WFQ).

The main difference with the previous scheduler is that the last finish time (the finish time of the last transmitted packet) is saved only on packet departure. Assumes that $\hat{V}$ is always this last finish time and the arriving packet k will be stored inside the queue i :

$$\begin{aligned} \hat{F}_i^0 &= 0 \\ \hat{F}_i^k &= \frac{L_i^k}{g_i} + max\Big(\hat{F}_i^{k-1}, \hat{V}\Big) \end{aligned} \tag{2.7}$$

where : $\hat{F}_i^k$ is the finish time of the arriving packet k in queue i
$\hat{F}_i^{k-1}$ is the finish time of last packet stored or sent by the queue i
L_i^k is the size of packet k from queue i
g_i is the guaranteed rate for the queue i

The scenario of this scheduler can be summarized as follows :

1. *(The packet arrives)* The scheduler computes the finish time and the packet is stored in the selected queue (according to the classifier)
2. *(The packet is at the head of the queue)* When it arrives at the head of the queue, the scheduler :
   (a) sends the packet with the lowest finish time (assuming that the selected packet is packet m from the queue h), its associated finish time is $\hat{F}_h^m$
   (b) updates the value $\hat{V} = \hat{F}_h^m$

The computation of the finish time is less complex because the mechanism used to assure that a queue will not build up credits is more simple (with $\hat{V}$).

This scheduler has the following drawbacks :

- The scheduler must save, with each packet, the finish time because of the presence of $\hat{V}$ (the update of this value requires that the finish time must be known for each packet). So the required memory can be large.
- This algorithm is also static (like WFQ).

Dynamic schedulers

An important problem of these schedulers is the static behaviour : once the finish time is computed, it doesn't change anymore and the consequence is that the scheduler selects the next packet according to parameters that doesn't show the current state. Moreover, to be an approximation of the proportional delay differentiation, weights must be computed as a function of the queue length (to take into account the queueing delays). Several extensions will be presented here :

1. Dynamic Weighted Fair Queuing (D_WFQ) [8]:
   The main difference with the WFQ scheduler is that weights are not fixed and are computed as a function of the queue length :

$$W_i(t) = QueueLength_i(t) * \delta_i$$

Where $W_i(t)$, $QueueLength_i(t)$ are, respectively, the weight and the length of the queue i at time t and finally δ_i is the class i's DDP. The finish time is computed like WFQ. D_WFQ uses the same mechanisms to avoid that a queue builds up credits.

The scenario of this scheduler is the following :

(a) *(A packet arrives in the queue i)* The weight W_i is updated and the finish time of this packet is computed.

(b) *(The packet k arrives at the head of its queue i)* The scheduler selects the packet with the lowest finish time among packets at the head of the queues. When the packet is sent, the scheduler computes W_i.

When the queue length is high (at time t), the associated weight will also be high to ensure that the scheduler meet proportional differentiation model requirements (the queueing delays are proportional). Because of dynamic weights, the complexity of the scheduler is higher : these weights must be updated regularly. In fact, they are recomputed when the queue changes (on packet arrival and departure). That is why the complexity increases really.

The D_WFQ scheduler has also the following drawbacks :

- Once the finish time is computed for a queued packet, it doesn't change even if weights change afterwards. So we can have some packets with a wrong finish time because it doesn't correspond to the current queue length.
- Computation of finish time can be complex.

2. Dynamic Virtual Spacing (D_VSPACING) [8] :
This scheduler is a VSPACING variant : the main difference is that the weights are dynamics (recomputed at a given time) according to the current queue length (like D_WFQ). These weights are evaluated just before the finish time computation. Moreover, this scheduler computes only the finish time when a packet arrives at the head of the queue.

The scenario of this scheduler is the following :

(a) *(the packet k arrives in the router).* It is stored in the selected queue i (according to the classifier).

(b) *(packet k is at the head of the queue i).* The weights are updated and the scheduler computes the finish time (for all packets at the head of queues) like VSPACING :

$$\begin{aligned} \hat{F}_i^0 &= 0 \\ \hat{F}_i^k &= \frac{L_i^k}{g_i} + max\Big(\hat{F}_i^{k-1}, \hat{V}\Big) \end{aligned} \tag{2.8}$$

where : $\hat{F}_i^k$ is the finish time of the arriving packet k in queue i
$\hat{F}_i^{k-1}$ is the finish time of last packet stored or sent by the queue i
L_i^k is the size of packet k from queue i
g_i is the guaranteed rate for the queue i

(c) The scheduler selects, among all not-empty queues, the smallest finish time and sends the packet.

(d) The value of $\hat{V}$ is updated to avoid that a queue builds up credit (like in VS-PACING)

This scheduler has the following disadvantages :

- The computation of the finish time can be complex
- Because of weights varying, the behaviour of the scheduler can be strange : when a lot of packets from one flow arrive into a queue : the associated weight decreases such that the queue's rate increases but, when the queue empties, the weight increases and the queue's rate decreases. That's why the last packets from the flow can be served with a low rate. With fixed weights, the scheduler serves all packets at fixed rate.

Waiting Time Priority scheduler (WTP)

This scheduler was described by L. Kleinrock first. A detailed description can be found in [19]. Unlike GPS based schedulers, WTP doesn't compute a complex finish time. The principle behind this scheduler is simple : the scheduler computes a packet priority and this priority increases with the time that the packet has passed inside the buffer (i.e. its queueing delay).

When the packet k arrives, the router puts this packet in the selected queue i and its arrival time a_i^k is saved. When this packet arrives at the head of the queue, its priority is computed as follows :

$$Priority_i^k = (now - a_i^k) * q_i \tag{2.9}$$

where :
now represents the current time
a_i^k arrival time of the packet k at the head of the queue i
$now - a_i^k$ represents queueing delay of the packet k in the queue i
q_i weight associated with the queue i.

Weights $\{q_i\}$ associated with the queues allow the network operator to tune the behaviour of the WTP scheduler (i.e. it defines the differentiation between queues). The scheduler will **transmit the packet with the highest priority** among the packets at the head of the queues first.

This scheduler meets, like GPS based scheduler with dynamic weights, the requirements of the proportional delay differentiation :

$$\begin{aligned} \frac{d_i}{d_j} &= \frac{\delta_i}{\delta_j} \\ \Leftrightarrow \frac{d_i}{\delta_i} &= \frac{d_j}{\delta_j} \end{aligned} \tag{2.10}$$

if the following constraint is verified :

$$q_i = \frac{1}{\delta_i} \tag{2.11}$$

And the proportional delay differentiation equation is :

$$d_i * q_i = d_j * q_j \Leftrightarrow (now - a_i^k) * q_i = (now - a_j^k) * q_j \quad (2.12)$$

The service provided by a class i is better than the service provided by a previous class. The $q_1 < q_2 < \ldots < q_N$ are the (inverse of) DDPs.

For instance, assume that three queues are configured with the following constraints :

- The delay of queue 3 must be the third of the delay of the queue 1
- The delay of queue 3 must be the half of the delay of the queue 2

Weights must be configured as follows : $q_1 = \frac{1}{3}$, $q_2 = \frac{1}{2}$ and $q_3 = 1$

The WTP scheduler has the following advantage and drawback :

- This scheduler is less complex than the previous rate-based scheduler (i.e. no virtual time or other complex operations to compute).
- The WTP scheduler requires that the arrival time of all packets is saved. The amount of required memory can be large : that will depend of the number of packets currently inside the router

Conclusion

Several schedulers have been described and each scheduler has many drawbacks and advantages. A scheduler must not be too complex because if the time needed to select the next packet to transmit is too high, the performance of the scheduler will be not good. Moreover, it is also important to pay attention to the required memory (if too much memory is required, the scheduler can't be deployed on Internet). It appears that the D_VSPACING or WTP schedulers are good trade-off between the complexity and required memory. Moreover, many simulations have been done at Alcatel and are presented in [8] and they show that the WTP scheduler provides the best differentiation on small and large time scales. This scheduler is also proposed in [13] because of very good delay differentiation.

That's why this scheduler will be implemented.

2.4.2 Proportional loss differentiation

The mechanism that provides relative loss differentiation is **the buffer acceptance algorithm** (see figure 2.2). It decides whether the incoming packet can be accepted or must be discarded and it protects the buffer to avoid overflows. This algorithm determines the loss rate.

Several mechanisms are available : the first and simplest is the *tail drop* mechanism : it drops incoming packets when the corresponding queue is full. This mechanism is easy to implement but can't provide any guarantee so it is not useful for loss differentiation.

An other mechanism is *Random Early Detection* [14] (or *RED*). RED attempts to maintain a low buffer occupancy. To realize that, RED drops packets statistically when the buffer occupancy is above a defined threshold. RED algorithm can't do loss differentiation but there are RED extensions [7] which can do this loss differentiation according to the packet drop precedence.

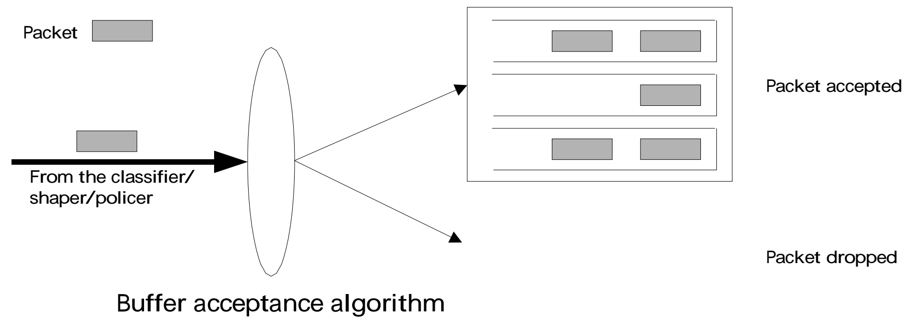


Figure 2.2: Buffer acceptance algorithm

The "simple" RED algorithm (without drop precedence) is explained first and RED variants that supports more drop precedences are described after.

The RED algorithm

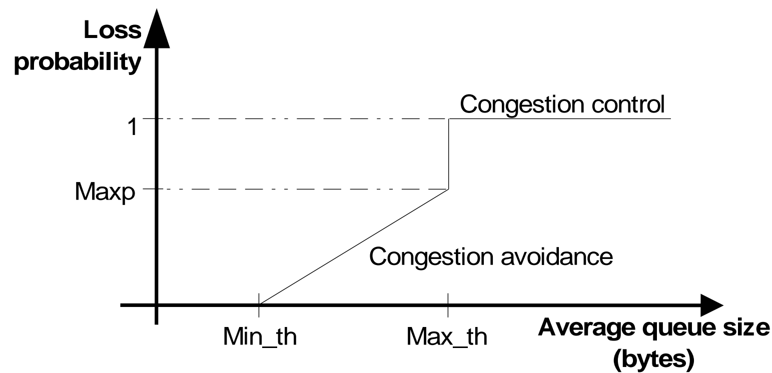


Figure 2.3: RED mechanism

As we can see on the figure 2.3, three parameters must be defined. The *minimum* Min_{th} and *maximum* Max_{th} threshold and *maximum drop probability* $Maxp$. When the average queue size reaches the minimum threshold, it begins to discard packets. Packets are discarded with an increasing probability ; this phase is called *congestion avoidance*. The maximum threshold defines the upper limit, packets are discarded with the defined maximum drop probability when the average queue size is equal to this threshold. Above the maximum threshold, the router is considered as highly congested and all incoming packets are dropped (*congestion control phase*).

This mechanism is friendly with TCP flows because it drops packets statistically. Indeed, when packet drops occurs, TCP decreases automatically its sending rate (congestion control mechanism) because it considers that packet drops occur when a router or network link is congested.

The RED behaviour can be summerazied as follows :

- Below minimum threshold : all packets are accepted in the buffer. The buffer occupancy is not critical.
- Between minimum and maximum threshold : packets are discarded with an increasing probability $p \in [0, \text{Maxp}]$ given by the graph of figure 2.3.
- Above maximum threshold : all incoming packets are discarded.

The use of RED algorithm is recommended in [3] :

> "Internet routers should implement some active queue management mechanism to manage queue lengths, reduce end-to-end latency, reduce packet dropping, and avoid lock-out phenomena within the Internet.
>
> The default mechanism for managing queue lengths to meet these goals in FIFO queues is Random Early Detection (RED). Unless a developer has reasons to provide another equivalent mechanism, we recommend that RED be used."

RED has the following advantages and drawbacks :

- Low buffer occupancy
- Packet drops that are friendly for TCP (avoids to drop a burst of packets)
- Discards packets from flows according to their network usage.

RED algorithm also has important drawbacks : the number of parameters (i.e. min_{th}, max_{th} and max_p). They must be fixed correctly to reach the assumed behaviour. But, it doesn't exist any guidelines to set the optimal values for these thresholds. The minimum threshold defines a lower limit : it defines the occupancy that is accepted inside the buffer (and, into a not-highly loaded[1] network : a lower bound on the queueing delay). The difference $max_{th} - min_{th}$ describes the aggressiveness of the dropper. For a fixed max_p value, if this difference is low : the dropper will be aggressive. Conversely, if this difference is high, the dropper will be less aggressive and that will modify the buffer occupancy ($\rightarrow$ the queueing delay). If this traffic has a congestion control mechanism, it will be preferable to have a non-aggressive dropper. On the contrary, if the traffic has no such mechanism, the dropper must be aggressive.

Weighted RED (WRED)

This variant of RED algorithm can do loss differentiation. The first proposed mechanisms was known as RIO for RED with In/Out [7] bit. As can be seen on figure 2.4, this version supports two drop precedences and assumes that arriving packets are marked with a special bit that indicates if this packet is conforms to the traffic contract (i.e. the SLA). The RIO mechanism can be considered as two RED algorithms that run simultaneously.

When a packet is received, the scheduler checks its drop precedence first. If the packet is marked as in-profile, the scheduler modifies the average of received in-profile packets (*avg_in* on the figure) and total average (i.e. average of all received packets – *avg_total*

[1] we consider here router where the minimal threshold is almost reached

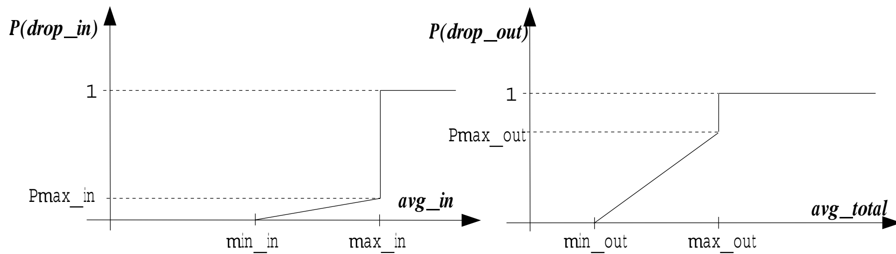


Figure 2.4: RIO mechanism [7]

on the figure). It discards in-packets using the *avg_in* while out-packets are discarded with *avg_out*.

Six parameters must be configured, three for each drop precedence : the minimum and maximum threshold and maximum drop probability. They can be configured independently : the figure 2.4 shows two RED mechanisms configured differently. Indeed, the dropper is more aggressive with out-packets because [7]:

- the maximum drop probability P_{max_out} is higher than the drop probability defined for in-packets.
- the minimum threshold for out-packet *min_out* is smaller : out-packets are dropped earlier than in-packets.
- all out-traffic is discarded before in-traffic because the maximum threshold for in-packets *max_in* is smaller than *max_out*.

The use of total average *avg_total* for out-packets can be justified because out-traffic is considered as excess traffic (traffic that does not conform to the contract) and the network is not sized to be able to forward this traffic.

Versions of RED with more than two drop precedences exist : they are known as Weighted RED (also called *WRED*). WRED works exactly like RIO but supports three or more drop precedences.

A Weighted RED variant will be described now : *Weighted RED in shared buffer mode*. This version supports more than two drop precedences and is adapted to meet the proportional loss differentiation model requirements.

As can be seen on figure 2.5, the value of minimum and maximum thresholds (Min_th and Max_th respectively on the figure 2.5) are the same for all drop precedences and only the maximum drop probability ($Maxp(X)$) must be defined for each case. On the other hand, only the total average is used while separate averages are used in RIO. That assures that the loss differentiation and delay differentiation are not linked together (if separate averages are used, a packet will be accepted only if its corresponding delay queue has not reached the configured threshold : so the low delay classes have also a lower loss rate).

This variant meets the requirements of the proportional loss differentiation : if only one drop precedence is defined, the drop probability can be computed by the straight line defined between the minimum and maximum threshold and maximum drop probability (as shown on figure 2.6).

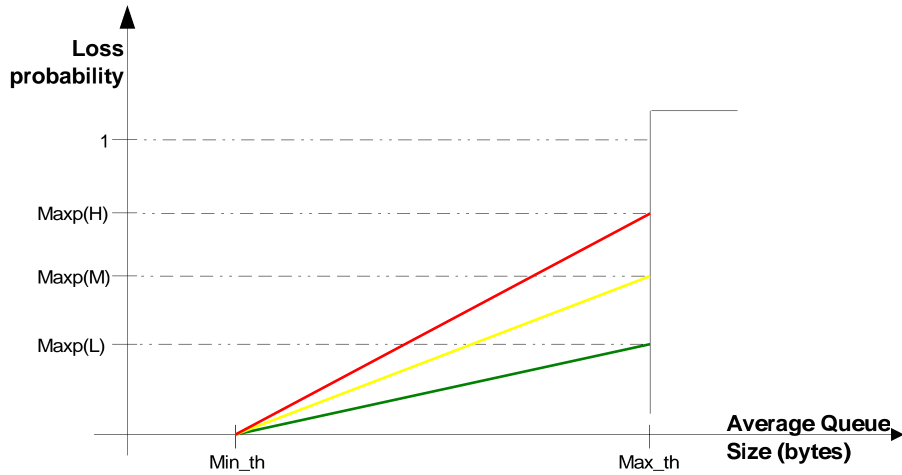


Figure 2.5: Weighted RED in shared buffer mode

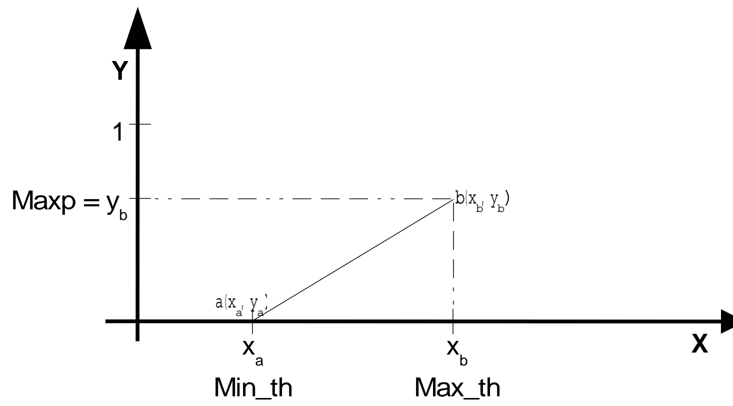


Figure 2.6: Drop probability

If a line is defined by two points (a and b), its equation is obtained as follows :

$$\begin{aligned} (y - y_a) &= m(x - x_a) \\ m &= \frac{y_b - y_a}{x_b - x_a} \end{aligned}$$

The RED algorithm has the following constraints :
$y_a = 0$
$y_b = max_p$
$x_a = min_{th}$
$x_b = max_{th}$
$x = avg$
y is the drop probability (called p_b).

We can compute the drop probability as follows :

$$p_b = \frac{max_p(avg - min_{th})}{max_{th} - min_{th}} \tag{2.13}$$

If this result is extended to more drop precedences, the following relation is obtained :

$$\frac{pb_i}{maxp_i} = \frac{avg - min_{th}}{max_{th} - min_{th}}$$

where pb_i is the computed drop probability of the queue i
$maxp_i$ is the configured loss ratio for the queue i

This last equation conforms to the loss differentiation model :

$$\frac{pb_i}{maxp_i} = \frac{pb_j}{maxp_j} \Leftrightarrow \frac{l_i}{\sigma_i} = \frac{l_j}{\sigma_j}$$

So the $maxp_1 > maxp_2 > \ldots > maxp_N$ are the LDPs (i.e. the $\{\sigma_i\}$ values) and a higher class i will receive a better service (lower loss rate) than previous classes.

The WRED mechanism has exactly same drawbacks than the RED mechanism.

Rate-based RED (RB-RED)

Another RED extension is the rate-based RED. It is *dynamic* and *rate-based* : it attempts to adapt its configuration itself according to network conditions. It is less sensitive to its initial configuration than RED algorithm and only one parameter has to be configured (RB-RED is described in [9]).

Working of RB-RED

Based on an estimation of the arrival rate (for each drop precedence) EAR, it computes the total estimated arrival rate $TEAR(t)$ as follows, at time t :
(Assume that three drop precedences are configured : low, med, high).

$$TEAR(t) = EAR(Packet_{low}, t) + EAR(Packet_{med}, t) + EAR(Packet_{high}, t) \tag{2.14}$$

With :
$EAR(prec, t)$: Estimated arrival rate for the drop precedence $prec$ at time t.
$Packet_{low}$: Packets with a low drop precedence
$Packet_{med}$: Packets with a medium drop precedence
$Packet_{high}$: Packets with a high drop precedence

It computes the drop probability $P_{drop}(t)$ at time t :

$$P_{drop}(t) = max(0, \frac{TEAR(t) - R}{TEAR(t)}) \tag{2.15}$$

where R is the service rate.

Packets have to be dropped **when the total estimated arrival rate is larger than the service rate**. [9] proposes the use of a correction factor to fix the aggressiveness of the dropper as a function of the buffer occupancy. If the buffer occupancy is low, packets will be dropped less aggressively and when it is high, its behaviour can be more aggressive.

Assume now that three drop precedences are defined, four situations are possible :

1. **if the total arrival rate $TEAR(t)$ is below configured service rate R :** no packet is dropped.
2. **if the total arrival rate $TEAR(t)$ is above the service rate R** and :
   (a) $EAR(Packet_{low}) + EAR(Packet_{med}) < R \Rightarrow Packet_{high}$ are discarded statistically, else
   (b) $EAR(Packet_{low}) < R \Rightarrow Packet_{med}$ are discarded statistically and all $packet_{high}$ are dropped, else
   (c) $EAR(Packet_{low}) > R \Rightarrow Packet_{low}$ are discarded statistically, all $Packet_{med}$ and $Packet_{high}$ are dropped.

In RB-RED, only the *aggressiveness parameter* has to be configured (service rate and queue size are not RB-RED specific parameters).

A RB-RED mode for proportional loss differentiation also exists.

The dynamic feature of RB-RED allows to reduce the number of parameters that must be configured. Because of the use of estimations, small inaccuracy in the EAR estimation can lead to strange behaviour (i.e. too many packets can be dropped) and not meet our expectations.

Conclusion

Many dropper have been examined : but which is good for loss differentiation ? Like schedulers, droppers should not be too much complex but must provide loss differentiation : so WRED, WRED in shared buffer mode and RB-RED are mechanisms that can be used. The WRED (extension of RIO) has too many parameters that have to be configured. Moreover, as seen above, the delay and loss differentiation will not be independent such that the dropper will privilege queues with a lower occupancy (and a lower queueing delay leads to higher rate and lower queue occupancy) that is why the WRED dropper is not interesting. The RB-RED mechanism appears to be interesting to implement but more complex (more estimations to compute for instance) than WRED. Finally the WRED in shared buffer mode offers a trade-off between the number of parameters that must be configured and the complexity of the mechanism. That is why this mechanism will be implemented while the RB-RED is also another acceptable solution.

2.4.3 Proportional bandwidth differentiation

It is possible to provide a bandwidth differentiation only by combining delay and loss differentiation. Indeed, some studies have shown that bandwidth for TCP flows can be estimated by the formula from Mathis et al. [20] :

$$throughput \leq \frac{MSS * C}{RTT * \sqrt{l}} \tag{2.16}$$

where MSS is the maximum segment size
C is a constant that depend of the TCP implementation
RTT is the round trip time
l is the loss ratio

This relation shown that delay and loss differentiation can have an important impact on the resulting throughput. Because delay differentiation will privilege some packets, it will affect the RTT value (by reducing the queueing delay). The loss differentiation have also a great impact because it allows to decide whether packets must be discarded and so changes the l loss ratio.

Proportional loss differentiation combined with proportional delay differentiation leads to proportional bandwidth differentiation.

2.5 Conclusion

This chapter has described the proportional differentiation model first and then how this model can be achieved by using existing mechanisms. Many mechanisms have been discussed and two mechanisms have been selected : the WTP scheduler to provide the proportional delay differentiation and the WRED (in shared buffer mode) dropper to provide the proportional loss rate differentiation. These two mechanisms have several characteristics :

- They are not too complex : networks become more and more fast so that the router must react quickly.
- The required memory is not excessive : but today, it is less important than the complexity because the price of memory chips is low.

This chapter has explained also how a proportional bandwidth differentiation can be achieved only by combining proportional delay and loss differentiation.

Chapter 3

Implementation

3.1 Introduction

In the previous chapter, the proportional model was presented and was used to provide a proportional delay and loss differentiation. The way to provide bandwidth was also described. Several mechanisms have been proposed to achieve the delay, loss and bandwidth differentiation : the waiting time priority scheduler for the delay differentiation, the weighted RED dropper for the loss differentiation while the bandwidth differentiation for TCP micro-flows can be achieved by combining the delay and loss differentiation using studies done by Mathis et al. and presented in [20].

This chapter aims to show how relative services can be implemented into a FreeBSD based router using the ALTQ architecture. This chapter is organized as follows : first, FreeBSD and its kernel will be described in details; after, the ALTQ architecture will be presented and finally the proposed solution will be discussed.

3.2 Overview of FreeBSD

FreeBSD is a free UNIX system. It works on several architectures like Intel x86 based processors, DEC/COMPAQ Alpha, etc. FreeBSD is already used by large companies or organizations on the Internet.

FreeBSD operating system has the following advantages : *preemptive multitasking, TCP/IP based, multi user, 32-bits, documented, open-source,. . . .* It is also an excellent development platform because it provides the most popular compilers for free. It is based on 4.4BSD-Lite from Berkeley (University of California). FreeBSD is mainly used in the context of networking because "FreeBSD Project has put in many thousands of hours in fine tuning the system for maximum performance and reliability in real-life load situation" [26].

FreeBSD allows us to "transform" a simple PC with several Ethernet interfaces into a router with complex packet filtering and forwarding mechanisms. That is why we will use FreeBSD to implement mechanisms as described in the previous chapter. Several stable versions of FreeBSD are available, we will use FreeBSD 2.2.8. New versions are available but that doesn't change anything for this implementation (more information about FreeBSD can be found on `http://www.freebsd.org`).

3.3 Description of FreeBSD

3.3.1 Main features

The kernel can be seen like a server that provides some functionalities to user applications. So an application can send a request (called *system call*) to the kernel to obtain a service in connection with its functionalities. The kernel proposes the following functionalities :

- process management
- memory management service
- time service
- user and group identifier
- resource management service
- starting of the system
- communication

After the description of these functionalities, we will see how the kernel can answer to a system call.

Process management

A process is a task that the system must execute. With each process, the system associates information. The most important is :

- *Process identifier (PID)* : A number given by the kernel to reference this process. This PID is very useful referencing process in a system call.
- *Parent process identifier (PPID)* : A reference to the process that asked the creation of the current process. An initial process is created during the initialization phase so that all processes have a parent process. We must remember here that a process can't create by itself a new process, only the kernel can do that because it manages all resources needed by a process. A process must ask through a system call the new process allocation.
- *State of the process* : To allow multitasking and process multiplexing, the kernel associatesa state with each process. This state describes the activity (or the life) of the process : under creation, can be executed (if all required resources are available), waiting for a given event, stopped or terminated (just before its death).

The kernel proposes system call to allow process to request the creation of another process (i.e. *fork* system call). When a process is created, the system sets its PID, PPID and state. The kernel must also support process destruction : all allocated resources must be freed (memory, file, lock, etc.). A process termination can be intentional or not (in case of errors). Indeed, it's possible that a system call can't answer to the request for several reasons (access not allowed, not enough memory, hardware problem).

The system provides signal management. A signal is an event (hardware and software event like illegal instruction, floating point exception, bus error, etc.) that a user or the system can sends to a process. A process can catch selected signals and provide actions. In this way, the system can report some hardware problems to a process.

Memory management

The kernel must also manage all kinds of memory available on the system. Memories can be classified according to their access time and size : some are fast (i.e. less than 10 ns) like the main memory but are volatile and have limited size (i.e. less than 1 Gb) while others are less fast (i.e. 10 ms) like secondary memory but are permanent and have great size (i.e. several Gb). The kernel must put the information in the appropriate memory type. These memory types have different usages : fast memory allows to execute a process, it contains all instructions of this process and all variables while slower memory is used to hold user data and programs.

The kernel supports also the swapping : this functionality allows the multitasking and is used when the kernel decides to change the current process. Indeed, the state (all registers values, program counter and data values) from the process must be saved usually on a slower memory to allow another process to start. When the process must be restored, its state is restored into the main memory and its execution can continue.

Time service

The kernel maintains the current time. Usually, time is updated when the system receives a clock interrupt. At each clock interrupt, the system increments time with a given number of microseconds (that corresponds with one clock tick). The internal representation of time (granularity of a tick) is a trade-off between the cost of clock interrupt management and clock precision. But today with fast processor, the clock precision can be high.

The external representation is always the same. It is the number of seconds and microseconds elapsed since the Epoch (January 1, 1970 00:00 UTC). This time can be obtained by a system call *gettimeofday()*.

User and group identifier

Users and user groups must be identified by the kernel to decide whether a given action can be authorized. All users and groups are represented by a number given by the system administrator (User IDentification and Group IDentification). A user belongs to a primary group and to a number of other groups.

The system administrator is represented by the identifier zero (UID = 0) and can do anything on the system. Users are structured in groups so that it's possible to give several rights to a set of users. These identifiers are useful to check if a user can access to a given resource (like a file) and determine the kind of access (read, write, execute).

Resource management service

Because resources are limited, a resource management must be present. Indeed, the system has to limit the quantity of resource a user can allocate (for instance, it is not reasonable

that a “regular” user can allocate all available memory). That's why some limitations on the resource utilization are fixed by the administrator.

The main resource to control is the processor. Because of the multitasking feature of this operating system, the processor must be shared in a fair way among all processes. The kernel orders processes according to their priorities. The processor is granted to a process during a defined time period. Once the time period has expired, the state of current process is saved and the next process is selected to start or continue its processing (the system restores the state of the process). To allow a user to tune the process priority, a special variable `nice` is defined and used in the priority calculation. Naturally, a user can only tune the priority of his processes.

Other resources must be controlled :

memory It's important to bound the amount of memory that a process can use.

disk space With quota system, the kernel can check if a user exceeds the allowed space.

The administrator sets for each resource the maximum limit and the kernel controls if the current utilization does not exceed these maximums for each resource.

Starting of the system

The kernel must also ensure the bootstrapping of the system. During this startup phase, the kernel must analyze the configuration of the system, initialize all hardware devices, launch all required software processes to allow users to use the system.

Communication

The kernel supports also two kinds of communication : communication between processes on the same host and communication through a network.

Communication between two processes This communication can be done by using some mechanisms like tube, shared memory, semaphores or signals. A tube is memory space that allows two processes to exchange bytes of data in a reliable way. The two processes must be executed on the same host. Shared memory is another way to exchange data : two processes have access (can read and write) to the same memory buffer. Semaphores and signals are mechanisms supported by the system that allow synchronization between two processes (i.e. a process can wait after a given event generated by another process).

Communication through a network Today the most used mechanism to exchange data between two processes through a network is the socket. A socket is the end point of a bidirectional communication. It must be allocated, then connected to a host (by using the host address) and then data can be exchanged using input and output streams. Once the communication is finished, the socket must be closed such that the communication can be interrupted and all allocated resources can be freed.

But today this distinction isn't made : communication between processes on the same hosts can be done using socket also : the destination and source hosts are the same in this case.

System call processing

When the kernel receives a system call, the processing changes from current process to the kernel : the state of the process is saved and the action corresponding to this system call is executed. Before, the kernel must check if the parameters of the system call are valid. Special system calls are often used : the `ioctl` system call. "The `ioctl` system call supports a generic command interface used by a process to access features of a device that aren't supported by the standard system calls" (from [26])

The system call succeeds or fails. Once it is finished, the kernel must return the result value (of the error code) to the process and resumes it. To do that, the kernel will restore the saved state and the process will be reactivated.

3.3.2 TCP/IP network layer

Description of Memory Buffer

The network protocols need a good memory management to have good performance : they must manipulate data with varying size, add or remove headers, send data to another layer. The kernel proposes a special structure, the memory buffer (called *mbuf*), to store information for the network protocol. In fact, a memory buffer is a memory area of a fixed size, usually 128 bytes (this fixed size assures that mbufs are aligned by the kernel's memory allocator with 128 bytes blocks of memory). According to the mbuf type, between 100 and 108 bytes of (user) data can be stored inside.

Four types of mbufs are possible depending on the mbuf type and flags :

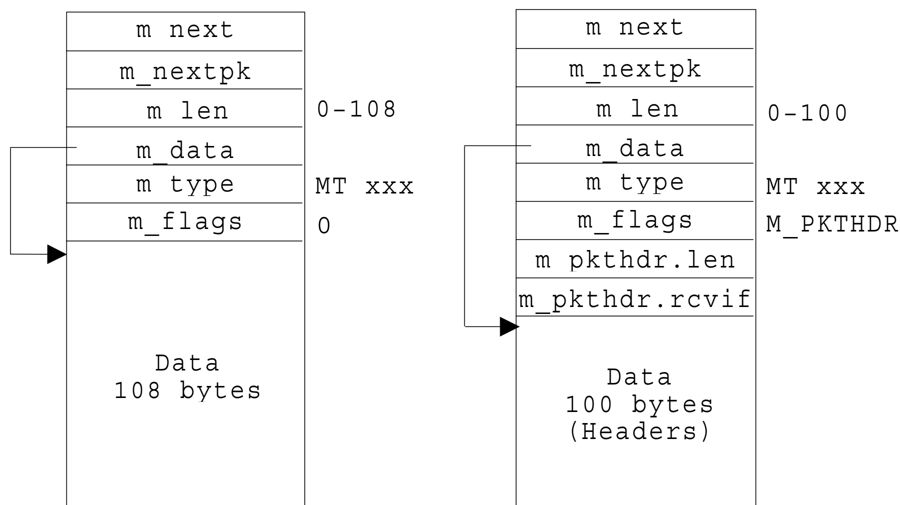


Figure 3.1: *Left:* Mbuf with data [29]. *Right:* Mbuf with header [29]

- As can be seen on figure 3.1, the first type of mbuf has a type `m_type = MT_xxx` and flags `m_flags = 0`. This mbuf contains only data.

  m_data It is a pointer to the first byte of data in the buffer.

m_len It indicates the data length (in byte) inside the buffer.

A description for `m_next, m_nextpk` will be given later. These six fields are always present in all mbuf types.

- Figure 3.1 also shows the second type of mbuf. The value of `m_type` is unchanged but the `m_flags` value is now `M_PKTHDR`. This indicates that the mbuf holds a packet header. Only 100 bytes are available because two new fields are defined (`m_pkthdr.len` and `m_pkthdr.rcvif`). The first will be described later. The second indicates the interface from which the packet comes (a given Ethernet interface or PPP interface, etc.). Consequently, a packet that the kernel sends (output packet) has the `m_pkthdr.rcvif` field sets to `NULL`.

- This third type of mbuf (figure 3.2) is used when more than 208 bytes of data must be stored in a mbuf. The `M_EXT` value indicates that data is saved inside an external buffer (called *cluster*) of 2048 bytes (cluster size can also be 1024 bytes). Three extra fields are defined into mbuf to manage this cluster (`m_ext.ext_buf, m_ext.ext_free` and `m_ext.ext_size`). The first is a pointer to the cluster, the second is currently unused, the third is the cluster size. The space for the packet header information (i.e. fields `m_pkthdr.len` and `m_pkthdr.rcvif`) is reserved but left blank. All data is saved in the cluster so space remains free and unused inside the mbuf (i.e. bytes beyond the `m_ext.ext_size` field).

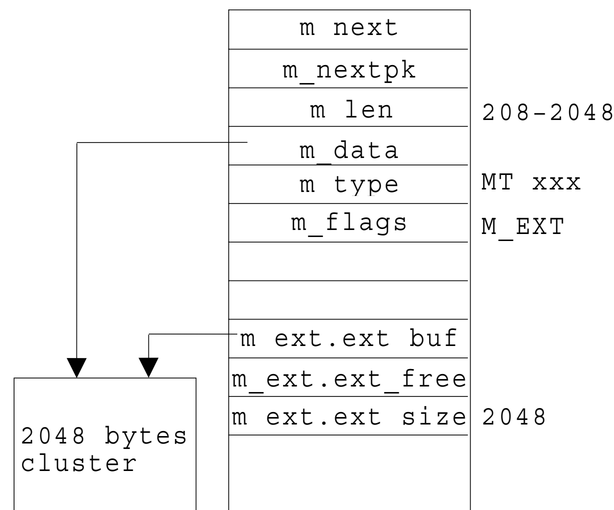


Figure 3.2: Mbuf with cluster [29]

- The last mbuf type (depicted on figure 3.3) shows that cluster and packet header can be combined. Indeed, the cluster contains packet headers with more that 208 bytes of data. Contrary to the previous case, all fields are filled in this case.

When the router receives a packet, this packet is stored inside a mbuf chain. Headers inside the packet are extracted to be stored inside the first mbuf of the chain. This mbuf

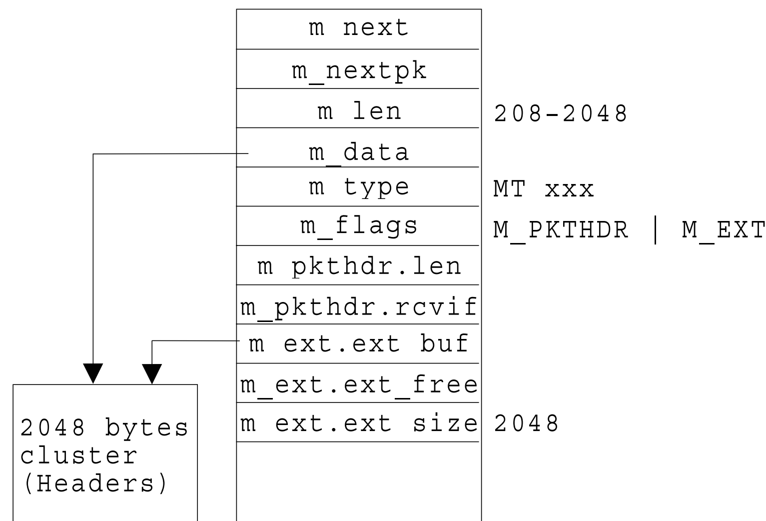


Figure 3.3: Mbuf with header and cluster [29]

with the flag set to M_PKTHDR, contains packet headers (IP and TCP for instance). The total chain size is specified in the field m_pkthdr.len (m_pkthdr.len $= \sum$ m_len from all mbufs in this chain). That also corresponds to the size of the whole packet. The following mbufs contain data from the packet.

The mbufs can be linked together with the fields m_next and m_nextpk (see figure 3.4). All mbufs from the same packet are linked with the field m_next. All mbufs that contain packet header (from different packets) are linked together with m_nextpk (so all packets are linked together). To avoid a long chain, when the packet size exceeds 208 bytes, the kernel allocates a cluster to store information from the packet. So packets are structured in a queue (because all packet headers are linked together with the m_nextpk field). The queuing strategy provided by the kernel is *first in first out* (FIFO). Moreover, all mbufs from one packet are linked together into a chain (by the m_next field).

Figure 3.4 shows an example with a UDP packet (i.e. the first packet that contains UDP and IP headers) of 192 bytes and a TCP packet of 1514 bytes (i.e. it contains a TCP and IP headers). These two packet types have different m_type value.

Several functions have been defined to support mbufs : mbufs allocation and destruction, some functions to duplicate mbufs and to destroy a packet (all mbuf from the same chain).

Packet Handling

This section will describe how the FreeBSD kernel processes packets. Figure 3.5 shows the main functions used to process IP packets.

A packet arrives from the network interface, is queued inside the IP input queue associated with this interface (ipintrq: on figure 3.5). The ipintr function contains all actions for input processing. If the packet reaches its destination (the destination address matches one network interface address), it is transmitted to the transport layer for processing. On

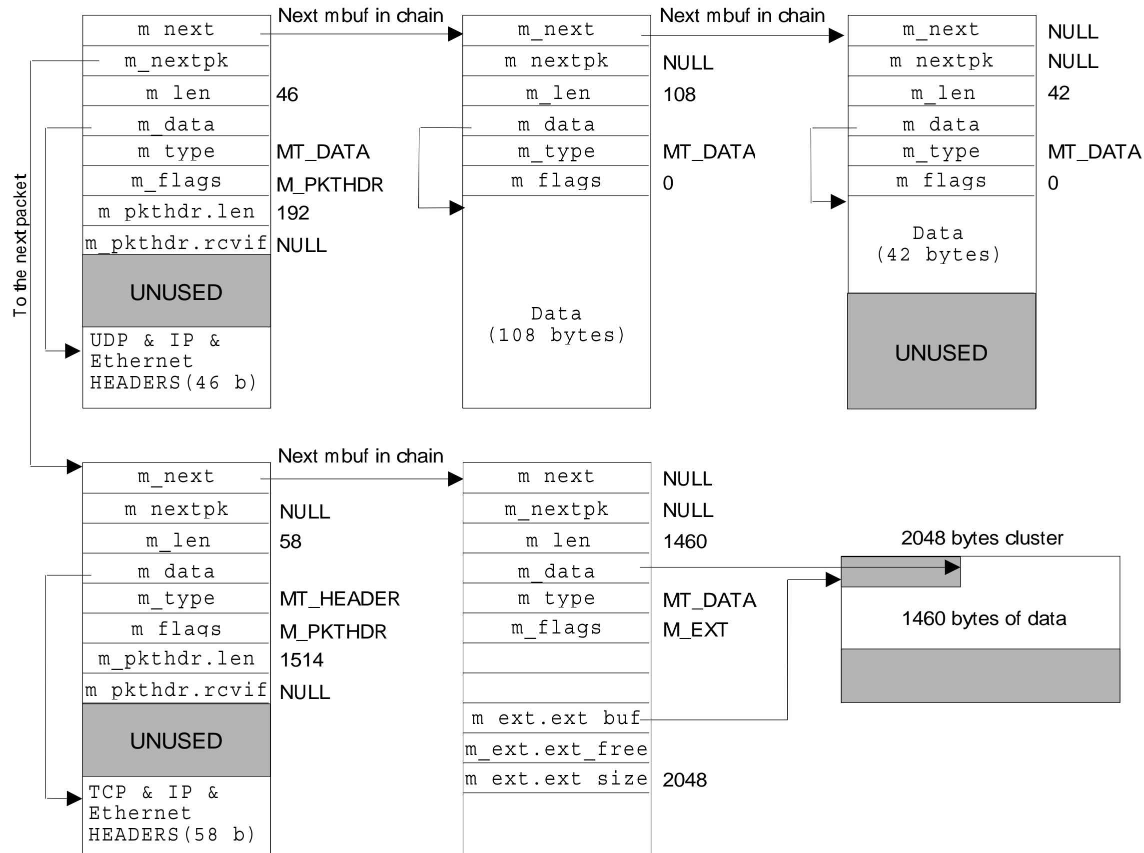


Figure 3.4: Queue with two packets [29]

the contrary, if this router is not the destination and if it can be forwarded (this decision depends of the `ip_forward` function), the packet is sent (usually through another interface) by the output processing (`ip_output` function). The transport layer can also generate a packet and this packet is also sent by the output processing.

The input processing, forwarding mechanism and output processing will be described in the following paragraphs.

Input processing When a packet arrives from a network interface, the following tasks must be done :

Verification The packet is dequeued (from *ipintrq*) and its content is checked to detect errors in the packet. The kernel looks the IP version encapsulated inside the IP packet (current IP version is 4), if the IP version isn't valid, the packet is discarded. A checksum is included inside an IP packet to protect the packet header, it must be verified. The packet length is also checked to assure that all data is present and no extra data is added.

Option processing and forwarding If this packet has not reached its destination (its

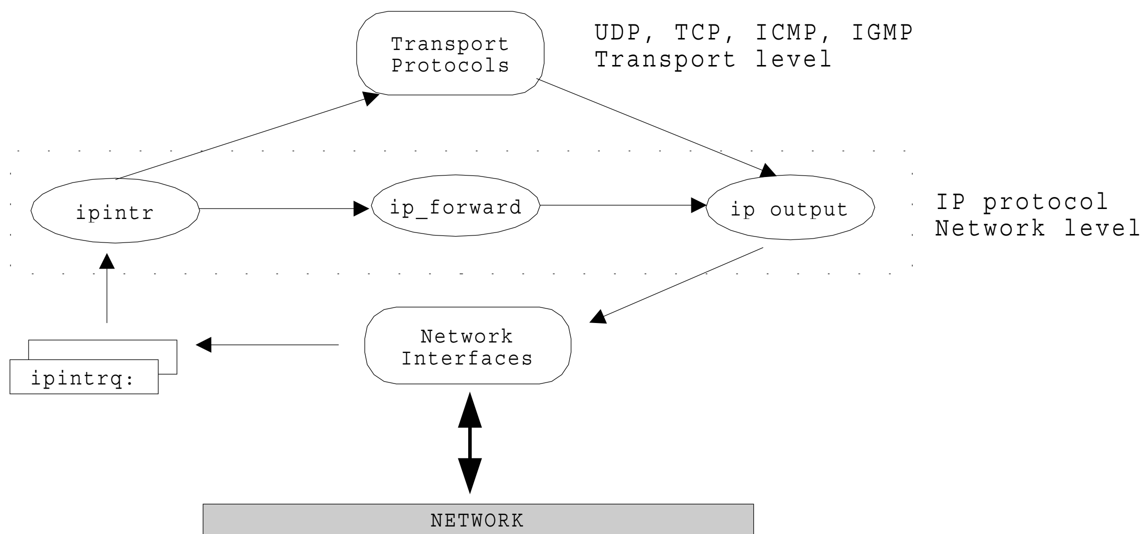


Figure 3.5: IP layer processing [29]

destination does not correspond to an address of this host), the packet can be forwarded (according to the result of the `ipforwarding` function). Some IP options can be included inside the packet, these options are processed by this function.

Packet reassembly An IP packet can be fragmented in several small fragments. This function reconstructs the whole IP packet from all fragments.

Demultiplexing The information from the packet is passed to the transport level layer for processing if this router is the destination, after, the kernel has removed the IP header before.

Forwarding To allow packet forwarding, the kernel must be configured as a router. In this case, when a packet arrives and doesn't reach its final destination, it is forwarded through a network interface to the next hop (by using the output processing). At each hop, the time to live (TTL) field is decremented. When TTL is equal to zero, the packet is dropped. It is also possible that the kernel does not know a valid route to reach the destination, an error message (ICMP message) is, in this case, sent to the sender.

Output processing Packets can come from the forwarding mechanism or from the transport layer. The output processing has mainly three functions :

Header initialization If the packet comes from the transport layer, the kernel must construct a new IP header with the necessary information. Some information is set by the upper layer (destination address, data length, TTL, protocol identifier, ...) while several other fields of the IP header are filled by the output processing (like IP version, ...). On the contrary, if packet comes from the forwarding mechanism, a header already exists and must be preserved.

Route selection A route can be provided by the transport layer (called *cached route*) for optimization propose. If no route is provided, the kernel has to determine the next hop according to the routing table entries and destination address.

Source address selection and fragmentation Because a router has many network interfaces with various IP addresses, a valid source address must be included inside the IP header. This address will depend of the interface used to send the packet. Fragmentation is only used if the selected network interface has a maximum transfer unit (MTU) smaller than the current packet length.

Before sending the packet to the selected network interface, the kernel has to compute the checksum.

3.4 Description of ALTQ

Alternate Queuing (or ATLQ) is a framework created by K. Cho from Sony Corp. to build new behaviours into FreeBSD based routers [5] [6]. ALTQ is free and already provides several mechanisms : WFQ, RED, RIO, ... are implemented inside ALTQ architecture.

ALTQ is presented as a patch for the FreeBSD kernel. This patch updates the kernel source code to add new behaviours. As we have seen, FreeBSD provides only a FIFO queue (for input and output packets) and ALTQ permits to use more complex algorithms.

3.4.1 Architecture

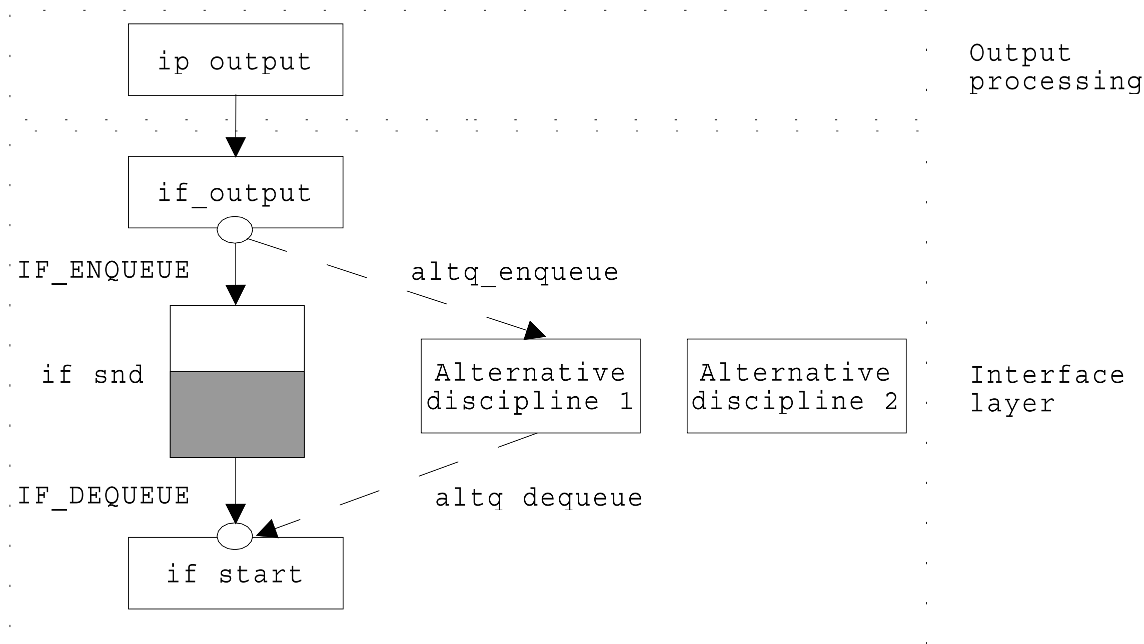


Figure 3.6: Alternate Queueing Architecture [5]

Figure 3.6 shows the ALTQ architecture. The `ip_output` implements the output processing mechanism already described before. The interface layer contains all hardware devices and software drivers that allow the kernel to send information through a particular network. Interface layer provides, in particular, a standard queueing method for outgoing packets ([29]). An alternative discipline is a new behaviour implemented inside the kernel : mechanisms described in the previous chapter are good example of new behaviours. In figure 3.6, the main functions are also showed (from [29]):

Function name	Description
`if_output`	queues outgoing packet for transmission
`if_snd`	the queue of outgoing packets for a given interface
`IF_ENQUEUE`	adds a packet to the end of the queue (for a given interface)
`IF_DEQUEUE`	takes the first packet of the selected queue
`if_start`	begin transmission

To help to develop new alternative discipline, ALTQ proposes the queueing architecture described in figure 3.7.

This architecture follows the IP layer processing (see figure 3.5) with the distinction be-

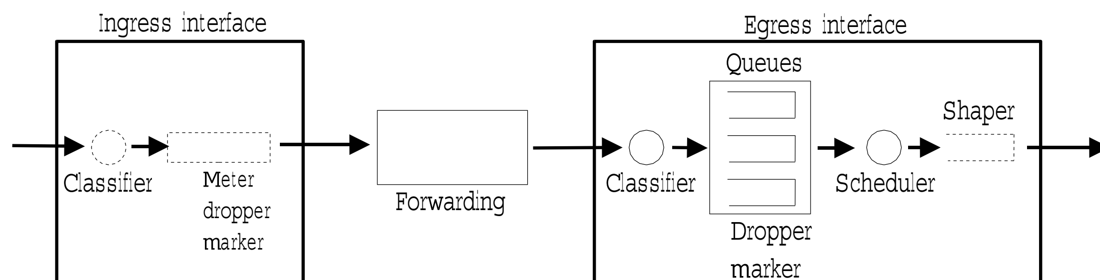


Figure 3.7: Queueing Architecture [5]

tween the input, forwarding and output processing. At the ingress interface (see figure 3.7), ALTQ allows to develop classifier, meter, dropper or marker. At the egress interface, usually, a classifier, queues with dropping mechanism and a scheduler are implemented; a shaper can be also implemented before the transmission of the packet. The following function blocks are defined :

Classifier The classifier identifies the flow and puts the packet in the correct queue based on the content of several fields of the packet header. This classification is done to ensure that the packet will receive the required service.

Meter The meter will measure some traffic characteristic (like bandwidth, packet count) to adapt the behaviour of the router if necessary.

Marker The marker will put an acceptable DSCP value in the IP header. This marking can be static (always the same value) or dynamic (the value will depend on the metering for instance). This value can be used by the classifier to identify the flow to which a packet belongs.

Dropper The dropper will decide if a given packet must be discarded or not. It will attempt to limit the queue occupancy in order to avoid congestion.

Queues The queues are finite buffers and will contain packets.

Scheduler The scheduler will select the next packet to be transmitted on the output link.

Shaper The shaper will delay some packets to ensure that the outgoing flow conforms to the configured value (from the SLA).

All function blocks are not always present. Especially, at the ingress interface because it is used for more complex algorithms like mechanisms implemented inside an edge router : a router can mark packet according to the metering block. For high speed core network routers, only mechanisms at the egress interface are usually implemented.

3.4.2 Working of ALTQ

Assume the host running ALTQ is configured as a router (packet forwarding is allowed with `ip_forward`) so all incoming packets are received by the input processing `ipintr` (see figure 3.5), forwarded by the forwarding mechanism `ip_forward` and finally sent to the network by the output processing `ip_output` through another interface.

- If ALTQ isn't enabled and a packet arrives at the router :
    1. The kernel checks the queue occupancy to know if enough space is available to hold this packet (`if_output` on figure 3.6).
        - The queue is (almost) full and can't hold this packet : the packet is dropped, i.e. tail drop. The processing of this packet stops here.
        - If queue can hold the packet, it is stored at the end of the queue (by using `IF_ENQUEUE`). The interface driver is called if it isn't already active (`if_start`).
    2. The interface driver is used to send all queued packets to the network. Packets will be dequeued (using `IF_DEQUEUE`) and send. The driver will remain active as long the queue isn't empty.
- If ALTQ is enabled and a packet arrives :
    1. The kernel (i.e. the `if_output`) calls the new queueing function (`altq_enqueue`) and bypasses the default queue occupancy check. This function calls the user enqueue method : this method usually encapsulates the classifier and the dropper (i.e. the buffer acceptance algorithm) :
        - The dropper decides to discard this packet : the processing of this packet stops, or
        - The packet is accepted inside the buffer : it must be stored inside the appropriate queue (i.e. the queue that matches with the requested level of service). The interface driver is called at that time.

2. The interface driver has been modified to use the new dequeue function (`altq_dequeue`). This function calls the user dequeue method : it contains the scheduling mechanism that must choose the packet that will be send. Like in the default behaviour, the driver remains active until the queue is empty. The driver doesn't know that separate queues are used (several delay queues), it continues to work while the dequeue function returns packets (i.e. all queues are empty).

3.5 Extensions to ALTQ

This implementation will propose new extensions to support WTP and WRED inside ALTQ. These extensions will be described in this section.

3.5.1 Architecture of the solution

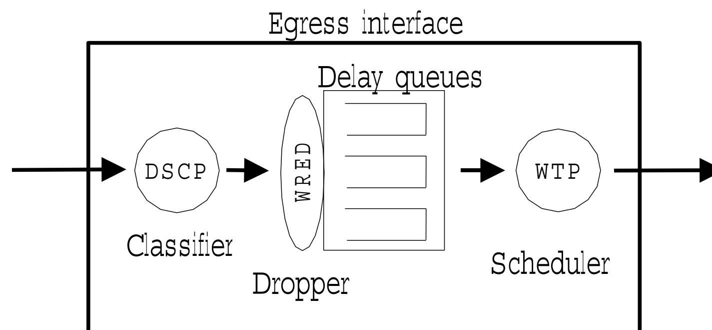


Figure 3.8: Architecture of the solution

As it can be seen on figure 3.8, the following function blocks has to be implemented :

DSCP classifier : this classifier checks whether or not the marking is valid. If it is valid, the classifier attempts (because of the WRED mechanisms) to put the packet in the right delay queue (according to the DSCP value). If the marking isn't correct, the packet must be placed in the default queue. Here we assume that packets are already marked when they arrive at this router. Packet metering mechanisms used to check if the SLA is respected are complex and are outside the scope of this thesis.

WRED dropper : the WRED must decide whether the incoming packet is queued or discarded according to the queue occupancy.

Delay queues : delay queues are used to store packets. The number of delay queues depends on the number of valid DSCP values because packets requiring different service are stored in a different queue.

WTP scheduler : the WTP scheduler provides the delay differentiation. It computes the queueing delay, selects the next packet and sends it.

The most important functions are `wtp_ifenqueue` and `wtp_ifdequeue`, which are invoked by ALTQ when a packet, respectively, arrives or leaves the router. The first function contains a part of the WTP scheduler (the computation of packet priorities and the packet selection) while the second function contains the classifier, the WRED dropper and the first part of the WTP scheduler (about the saving of the packet arrival times).

3.5.2 Implementation of WTP scheduler

Here the implementation of delay differentiation will be described (see 2.4.1 for a theoretical description of the WTP scheduler and the WTP implementation is included in the appendix A). Problems to solve are :

1. To find where to store the arrival time.

2. To determine how to store the arrival time.

Where to store the arrival time ?

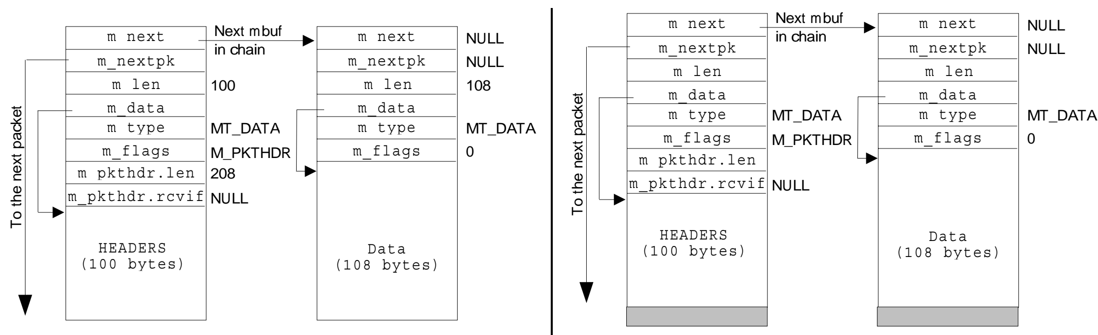


Figure 3.9: *Left :* No free space inside mbuf *Right :* Mbuf extended with a new field

The main issue in the WTP implementation is where to store the arrival time. Several solutions are possible :

- Use free space inside the mbuf structure. According to the mbuf type, it is often possible to find free space inside the mbuf structure. But, this method is not deterministic and in several cases, all mbufs of the chain can be fully filled. For instance, if the first packet contains 100 bytes with headers and the following packet holds 108 bytes of data (depicted on figure 3.9).

- Extend the mbuf structure and add a new field to store the arrival time (as can be seen on figure 3.9). This is not a good idea because :

  1. it assumes that inside the kernel a mbuf structure is $MSIZE$ bytes (fixed).
  2. With this method, a field will be reserved in each mbuf among the chain so space inside mbuf will be reserved but never used. Moreover, it seems that extending the data area to use this field when no arrival time will be stored is a complex

task because the code of the kernel memory allocator must be reviewed to take into account this modification : the size of the mbuf is used to determine the mbuf type and to decide when a cluster must be allocated. Finally, side effects can't be excluded : because of optimization, the kernel takes a lot of assumptions that can be modified easily.

- Create a new structure to save this arrival time. Two linked chains will be handled at the same time : the first with packets and the second with arrival times. This solution is depicted on figure 3.10.

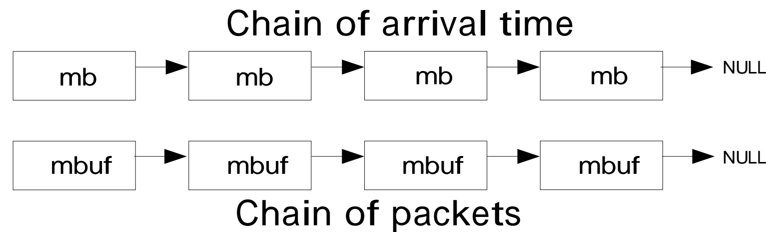


Figure 3.10: mbuf structure

When a packet arrives, its arrival time will be stored inside the `mb` structure. When the packet priority must be computed (when this packet is at the head of the queue), the saved arrival time will be used. If this packet is dequeued, the associated `mb` structure is destroyed. Because of linked chains, a pointer to the previous member must be reserved. It's not necessary to have a connection between a packet (contains inside the mbuf chains) and its arrival time because packets shouldn't be moved inside the router (queues are handled by our implementation so it's possible to enforce this constraint).

How to store the arrival time

In a UNIX based environment, time can be stored inside a special structure called `timeval`. Recall that time is represented by the number of seconds and microseconds since the Epoch. That's why the `timeval` structure is defined as follows :

```
struct timeval {
   long tv_sec;  /* seconds */
   long tv_usec; /* and microseconds */
};
```

Inside the kernel, the current time can be obtained by the `microtime` function. The structure definition shows that to store the arrival time, 8 bytes are needed (a `long` value takes 4 bytes). It's possible to reduce that by using a "compressed" time. To keep a data-type that can be processed easily, a 32 bits integer is used (this solution is shown on figure 3.11).

For microseconds, an upper bound exists : 999 999 μs (because $10^6\mu s = 1s$). So maximum 20 bits are needed to store microseconds and second can be encoded onto 12

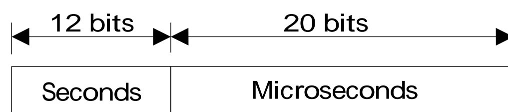


Figure 3.11: Proposed compressed time

bits. That's not a problem : this assumption enforces that packets leave the router after a maximum of 4096 seconds (2^{12}), but that is always true. This time compression reduces the space required by a factor of two (only four bytes are needed to store the arrival time).

Conversion to compressed time An important point for the proposed compressed time is that conversion between the two representations (original `timeval` format and its compressed version) must not be complex to compute. This is possible by using fast shift operations.

Assume that `time` is a `timeval structure` and `comptime` is the associated compressed time.

```
comptime = (time.tv_sec << 20) | (time.tv_usec & 0x000FFFFF);
```

In addition the uncompressed time can be obtained as follows :

```
time.tv_sec = comptime >> 20;
time.tv_usec = comptime & 0x000FFFFF;
```

Implementation details

We define the `mb` structure :

```
struct mb
{
  unsigned long arr_time;  /* Arrival time [COMPRESSED TIME] */
  struct mb* next;         /* pointer to next mb structure */
};
```

An arrival time must be associated with **each packet** : 8 bytes per IP packet (32 bits for arrival time and 32 bits for the "next" pointer. The compressed time enforces a limit of 4096 seconds : an overflow appears if the router works more than 4096 seconds. To avoid it, the arrival time can be encoded as the time elapsed since the previous packet arrival time in the same queue. To allow the computation of this arrival time, it's necessary to keep the last arrival time per queue. Another problem has to be solved : only the time between two packets is saved so it's not possible to convert the compressed arrival time into a UNIX compatible time (i.e. the original `timeval` format) without a reference time per queue. When the first packet arrives in the queue, the reference time is assigned to the current time. The first arrival time is zero (i.e. the arrival time of a packet arriving at an empty queue) while others are encoded as the time elapsed since the previous packet. The reference time is used to compute the packet arrival time :

$$PacketArrivalTime_q^k = ReferenceTime_q + PacketCompressedTime_q^k$$

where $PacketArrivalTime_q^k$: is the arrival time (in UNIX compatible format) of the packet k at the head of the queue q
$ReferenceTime_q$: is the reference time (in UNIX compatible format) of the queue q
$PacketCompressedTime_q^k$: is the compressed time of the packet k in the queue q

The queueing delay is computed as follows :

$$QueueingDelay_q^k = CurrentTime - PacketArrivalTime_q^k$$

where $QueueingDelay_q^k$: is the queueing delay (in μs) of the packet k at the head of the queue q
$CurrentTime$: is the current time of the system

The reference time of the queue q has to be updated when the packet k leaves the router :

$$ReferenceTime_q^{new} = ReferenceTime_q^{old} + PacketCompressedTime_q^k$$

For instance, assume that the router receives 3 packets. The first packet arrives when the queue is empty so its arrival time is zero and the reference time is set to the current time. The second packet arrives after 10 microseconds so its compressed arrival time is 10 and finally, the last packet arrives 15 microseconds after the first packet. Its compressed arrival time is 5 $(15 - 10)$. This situation is depicted in figure 3.12.

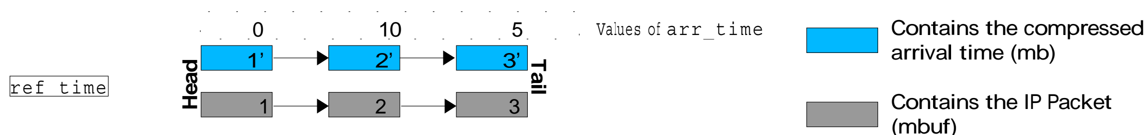


Figure 3.12: Example of `arr_time`

When this queue is selected by the scheduler, the packet arrival time is computed : $ReferenceTime+0$, the packet is sent and the queue reference time is updated : $ReferenceTime = ReferenceTime + 0$. When the second packet has to be sent, its arrival time is calculated : $ReferenceTime + 10$. When it is sent, the reference time is modified : $ReferenceTime = ReferenceTime + 10$. Finally, when the last packet is transmitted, its arrival time will be $ReferenceTime + 5$ and the reference time will be updated.

A queue (or class) for WTP is defined as follows :

```
typedef struct wtp_queue
{
  struct mb *head_at, *tail_at;   /* arrival time chain */
  struct mbuf* head, *tail;       /* packets in this queue */
  int size;                       /* Bytes in this queue (for the dropper) */
  int weight;                     /* Weight of this queue (integer) */
  short int dscp;                 /* ID/DSCP of this queue (for the classifier) */
  struct timeval ref_time;        /* Reference time [COMP_TIME] */
  struct timeval last_time;       /* Arrival time of last packet */
```

```
    /* Statistical variables */
         ...

} wtp;
```

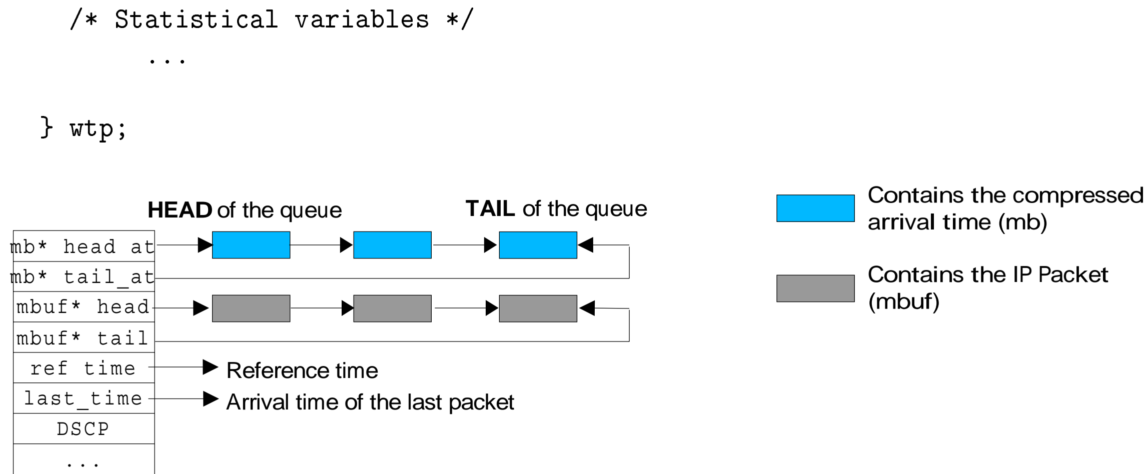


Figure 3.13: main queue field

As it can be seen on figure 3.13, two linked chains are used. The first contains the packet arrival times (`mb` structure) and the second contains the packets stored in this queue (`mbuf` structure). The `size` of each queue is also saved : useful for a tail drop mechanism. To avoid that a queue overflows, all queues are limited by a tail drop mechanism.

The `weight` parameter is used for delay differentiation between the classes while the `dscp` field indicates the DSCP of this queue. Finally, the reference time and the last arrival time are also saved. We also have some variables to maintain statistics.

The two main functions (i.e. wtp_ifenqueue and wtp_ifdequeue) will be described now.

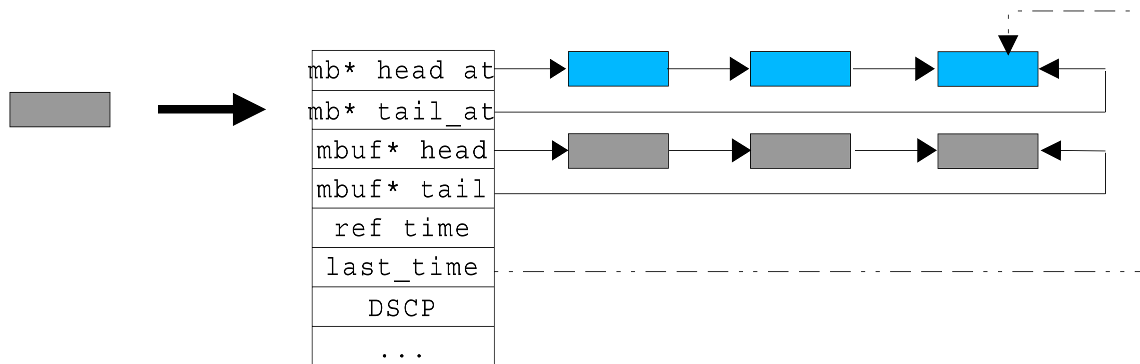


Figure 3.14: A packet arrives

wtp_ifenqueue When a packet arrives (see figure 3.14), we must :

1. extract its DSCP value. To extract packet information and identify the flow of a packet, ALTQ provides a special function for doing this.

2. check if this packet must be discarded by the dropper (described in the section 3.5.3).
3. place the packet in the right queue.

This classifier is not complex : only the DSCP is needed because it indicates the required level of service.

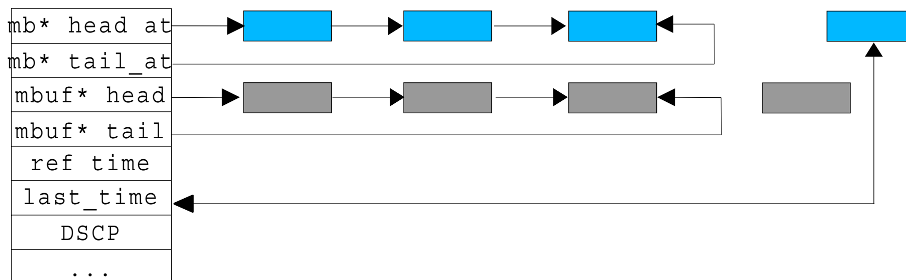


Figure 3.15: `mb` allocation

If the packet is accepted by the buffer acceptance mechanism, a new `mb` structure is allocated (see figure 3.15) and associated with this packet. If this structure can't be allocated (not enough free memory for instance), an error is reported to the kernel and to the user. If the `mb` structure is allocated without problem, the arrival time has to be saved. The arrival time will depend on the queue occupancy. The pseudo-code is following :

```
if(QueueEmpty())
{
  /* insert packet at the head of the linked chain */
  queue->head_at = mb; /* the mb is put at the head of the mb chain */
  mb->next = NULL; /* no mb follows this */
  update_time(queue->ref_time); /* put the current time in ref time */
  queue->last_time = queue->ref_time;
  mb->arr_time = 0; /* First packet: compressed arrival time = 0 */
  queue->head = mbuf; /* the packet is placed at the head of the queue */
}else
{
  tmp = queue->last_time;
  mb->next = NULL;
  queue->tail_at->next = mb; /* mb stored at the tail of the queue */
  tmp = now - queue->last_time;
  /* tmp is the time since the last packet arrival */
  queue->last_time = now;
  mb->arr_time = (tmp.tv_sec << 20) | (tmp.tv_usec & 0x000FFFFF);
  /* mb->arr_time is the compressed version of tmp */
  queue->tail->m_nextpkt = mbuf; /* mbuf stored at the tail of the queue */
}
```

If the chosen queue is empty, the `mb` (with arrival time) and `mbuf` structure (with incoming packet) must be saved at the head of the chains. The arrival time `arr_time` will

be zero and it is important to update the reference time `queue->ref_time` and the last arrival time `queue->last_time` with the current time (if other packets arrive). On the contrary, if the queue is not empty :

1. The arrival time is not zero anymore but it is equal to the time elapsed since the last packet arrival and must be computed. Because the last packet arrival time is saved, it's easy to compute the new packet arrival time. Once computed, the last arrival time must be updated.

2. The compressed arrival time is computed and stored inside the `mb` structure.

3. All chains are updated to hold the new packet and its associated arrival time.

Figure 3.16 shows the state of the two chains when the packet and its associated arrival time are queued. The two structures have been added at the end of the chains and the last arrival time is updated.

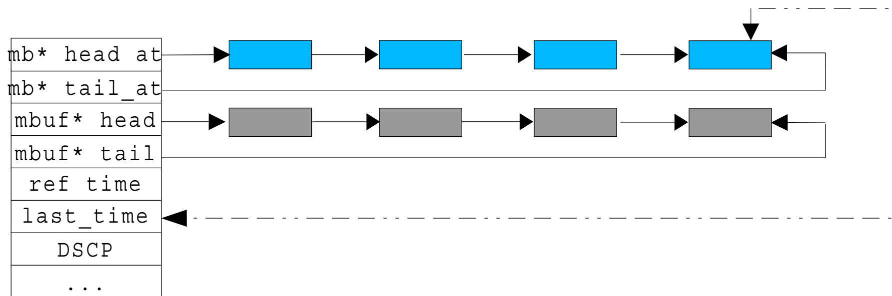


Figure 3.16: Enqueue a packet

Finally, the tail pointers and statistic (this is done in all cases : if queue is empty or not) are updated :

```
queue->tail_at = mb;
queue->tail = mp;
byte = mp->m_pkthdr.len;
queue->size += byte;
```

If the incoming packet is not accepted by the buffer acceptance mechanism, the packet is destroyed. The kernel and user are informed that a packet has been discarded by returning a special value `ENOBUFS` :

```
m_freem(mbuf); /* free the memory reserved for the packet */
error = ENOBUFS; /* inform of this drop */
```

wtp_ifdequeue

Until queues are empty, the kernel calls this method to obtain the next packet to send. That is why the first thing to do is to check if all queues are empty or not. If all queues are

empty, this method will return a `NULL` pointer to inform the kernel (especially the interface driver) that queues are empty and no packet has to be sent. Otherwise, this method must return the pointer of the packet that will be sent. This function implements the WTP scheduler, it has to :

1. Check if queues are empty
2. Compute queue priorities
3. Select queue with highest priority
4. Dequeue and send the packet at the head of the queue

First, check if queues are empty. Because the size of each queue is maintained, it is easy to verify that :

```
i=0;
while(i < number_queue and queue[i].size == 0)
   i = i + 1;
if( i == number_queue)
   return NULL; /* no packet in the queues */
```

In fact, it's possible to implement this in a more efficient way : if the kernel maintains a global queue size, this checking can be done with a simple instruction. For this implementation, this optimization is used.

Then, queues priorities (i.e. the priorities of packets at the head of the queues) are computed and the scheduler selects the queue with the highest priority :

```
for(i=0; i<number_queue;++i)
{
  long prio = 0;
  struct timeval delay;
  if(isNotEmpty(queue[i])) /* if this queue isn't empty */
  {
    delay = now - refTime(queue[i]); /* delay = queueing delay */
    prio = convertMicroSec(delay) * (long) queue[i].weight;
    queue[i].head_at->arr_time = 0;


    if(prio >= maxprio) /* if the priority of this packet is */
    {                   /* >= than the current maximum priority */
        maxprio = prio;
        queueMAXPRIO = queue[i];
        SAVEDdelay = delay;
    }
  }
}
```

To obtain the queueing delay, a new `refTime` function is used. This function returns the arrival time of the packet at the head of a given queue by using the reference and compressed arrival times stored inside the `mb` structure.

```
static struct timeval refTime(wtp* queue)
/* this function updates the ref_time inside the given WTP queue and returns it */
{
   struct timeval tmp;
   unsigned long deltaTime = queue->head_at->arr_time;
   tmp.tv_usec = deltaTime & 0x000FFFFF;
   tmp.tv_sec = deltaTime >> 20;
   queue->ref_time.tv_sec += tmp.tv_sec;
   queue->ref_time.tv_usec += tmp.tv_usec;
   if(queue->ref_time.tv_usec >= 1000000)
   {
      queue->ref_time.tv_usec -= 1000000;
      queue->ref_time.tv_sec +=1;
   }

   return(queue->ref_time);
}
```

The reference time will be updated with the packet at the head of the queue. The packet arrival time is equal to this reference time but when the reference time is computed, it can't be sure that this queue will be selected by the WTP scheduler because another queue can contain a packet with a higher priority so it is important to assign zero to the compressed arrival time because the reference time has been updated. This ensures that the reference time will be modified only one time for a given packet.

Once the scheduler has selected the queue with the highest priority, the packet from the selected queue is dequeued :

```
/* queueMAXPRIO is the selected queue by the scheduler */
mbuf = queueMAXPRIO->head;
```

The packet at the queue's head is selected. This packet must be returned and the associated `mb` structure must be destroyed. Finally, several statistics are updated.

```
int byte = mbuf->m_pkthdr.len; /* packet size in byte */
if(!(queueMAXPRIO->head = mbuf->m_nextpkt)) /* if this queue doesn't */
   queueMAXPRIO->tail = NULL;               /* holds more packet */
mbuf->m_nextpkt = NULL;
queueMAXPRIO->size -= byte;
...    /* update some statistics */

if(!(svg = queueMAXPRIO->head_at->next)) /* if this queue doesn't */
   queueMAXPRIO->tail_at = NULL;         /* holds more mb */
```

```
queueMAXPRIO->head_at->next = NULL;
FREE(queueMAXPRIO->head_at, M_DEVBUF);  /* Free the mb structure */
queueMAXPRIO->head_at = svg;
queueMAXPRIO->mbuf_bytes = byte;
...    /* WRED dependent code */
return(mbuf);
```

3.5.3 Implementation of WRED

Here the implementation of the loss differentiation (by using WRED) will be described. WRED was presented in the section 2.4.2. As can be seen on the architecture (figure 3.17), the WRED mechanism must be called just after the classification (in the enqueue method). The WRED implementation is included in the appendix A.

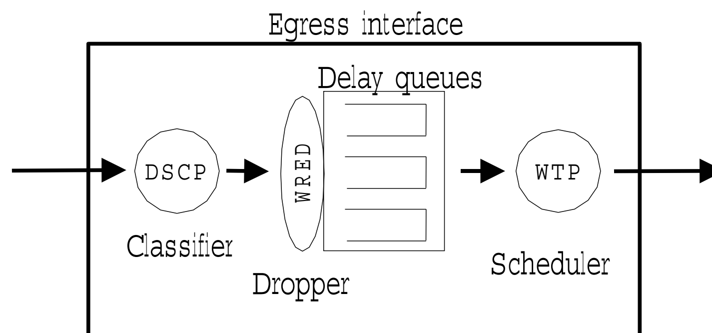


Figure 3.17: Architecture of the solution

Here we show the RED pseudo-code (from [14]) :

```
for each packet arrival
   calculate the average queue size ‘avg’
   if min_th <= avg < max_th
      calculate probability ‘pa’
      with probability ‘pa’ :
          mark/drop the arriving packet
   else if max_th <= avg
          mark/drop the arriving packet
```

The implementation of (W)RED can be splited into small sub-problems :

1. Computes the average queue size
2. Calculates the packet drop probability
3. Drops incoming packet if necessary

RED is already implemented inside ALTQ. But it is only available with other mechanism. So I must rewrite it to assure that WTP and (W)RED can work together (at the same time). WRED algorithm is only a small extension to the RED algorithm to support several drop precedences (see chapter 2).

We use WRED with shared buffer. It means that, for WRED, all packets are stored into the same buffer (even if we have several delay classes) to assure that the drop probability is calculated with an average queue size independent of the delay classes.

Kernel programming issues

Programming at kernel level enforces some limitations. For instance, floating point number can't be used inside a kernel because it's too difficult to support this data-type. Some operations like divide operation must be used only when it's necessary because they take a lot of time.

Sometimes decimal expression have to be computed. In this case, a fixed point representation will be used : with a 32 bits integer, a decimal point will always be assumed after the 20th bit. Indeed, in fixed point representation (also called *FXP*), the first 20 bits are the integer fraction of the number while the last 12 bits are the decimal fraction of this number. For instance, if the following number is considered (in binary) :

`1000 1101 1000 1010 1100 1100 0101 1001` i.e. `8D8ACC59` in hexadecimal

It will be interpreted (in FXP) as :

`1000 1101 1000 1010 1100 . 1100 0101 1001` i.e. `8D8AC.C59` in hexadecimal

Finally, in decimal representation : $2^{19} + 2^{15} + 2^{14} + 2^{12} + 2^{11} + 2^{7} + 2^{5} + 2^{3} + 2^{2} + \frac{1}{2^1} + \frac{1}{2^2} + \frac{1}{2^6} + \frac{1}{2^8} + \frac{1}{2^9} + \frac{1}{2^{12}} = 579756.771728515625$

Average Queue Size

The average queue size will be estimated with an Exponential Weighted Moving Average (EWMA) :

$$\begin{aligned} avg &= (1 - w_q) * avg + w_q * q \\ avg &= avg + w_q * (q - avg) \end{aligned}$$

Where w_q is the weight for EWMA average ($w_q \in$[0,1])
q is the current queue size (instantaneous)

When the parameter w_q is large (close to 1), the average queue size would be near to the instantaneous queue size. Conversely, if w_q is too low (close to 0), average queue size would be too far from the queue size. [14] suggests to use $w_q = 0.002$.

The average queue size can be updated only when a packet arrives. A problem occurs when a packet arrives and the buffer is empty (also reported in [14]), the average can't be correct because its computation will take into account an old *avg* average as first value. That's why when a packet arrives and buffers are empty, the average will be modified to take into account the number of packets that the router would have transmitted during

the inactive time (time interval where the buffer is empty). So the following relation will be used in this case :

$$avg = (1 - w_q)^{(time - q_time)/s} * avg$$

Where $time - q_time$ is the inactive time in μsec
s is the typical transmission time (e.g. 800 μsec)

Implementation details To compute the *avg* value, the fixed point representation will be used. The pseudo-code of the implementation is as follows :

```
if(isEmpty())
{
  t = time - q_time;
  if(t > 60 seconds )
      avg = 0;
  else
  {
      t = t / s;
      if(t >= 0)
          avg = avg * pow_w(t);
  }
}
avg = avg + ((q - avg) >> inv_wq);
```

Where `time` : is the current time
`pow_w(t)` : is an estimation of the value of $(1 - w_q)^t$ in fixed point representation.
`inv_wq` : is the inverse of w_q value. It must be a power of two.

The value of $(1 - w_q)^t$ is estimated with a table lookup. At initialization phase, a table w_tab with the values of `w_tab[t] =` $(1 - w_q)^{2^t}$ is built. With this table it is possible to compute the result of the function `pow_w(t)`. For instance, if $t = 53$, the value $(1 - w_q)^{53}$ is computed as follows :

$$(1 - w_q)^{53} = (1 - w_q)^{32} * (1 - w_q)^{16} * (1 - w_q)^4 * (1 - w_q)$$

As can be seen from the pseudo-code, when a packet arrives and the buffers are empty, the average queue size will be updated two times. The first time, the router evaluates the number of packets that it would have sent during the inactive period and then, it updates the average queue size using the EWMA algorithm and the new instantaneous queue size.

Packet Drop probability

In chapter 2, the relation to compute the packet drop probability has been established :

$$p_b = \frac{max_p(avg - min_{th})}{max_{th} - min_{th}} \tag{3.1}$$

In [14], the author states that the drop probability is $p_a = \frac{p_b}{1 - count * p_b}$. It assures that the drop probability p_a is a uniform random variate ($count$ is the number of received packets since last packet drop).

$$
\begin{aligned}
p_a &= \frac{max_p * avg - max_p * min_{th}}{max_{th} - min_{th}} * \frac{1}{1 - count * \frac{max_p * avg - max_p * min_{th}}{max_{th} - min_{th}}} \\
p_a &= \frac{max_p * avg - max_p * min_{th}}{max_{th} - min_{th}} * \frac{1}{\frac{max_{th} - min_{th} - count * (max_p * avg - max_p * min_{th})}{max_{th} - min_{th}}} \\
p_a &= \frac{max_p * avg - max_p * min_{th}}{max_{th} - min_{th} - count * (max_p * avg - max_p * min_{th})} * \frac{\frac{1}{max_p}}{\frac{1}{max_p}} \\
p_a &= \frac{avg - min_{th}}{\frac{max_{th} - min_{th}}{max_p} - count * (avg - min_{th})} \qquad (3.2)
\end{aligned}
$$

Implementation details A description of how the kernel computes the drop probability is given here. To compute this probability, three parameters are needed : `fp_len`, `fp_probd` and `count`. $fp_len = avg - min_{th}$ and $fp_probd = \frac{max_{th} - min_{th}}{max_p}$.

```
int drop(int fp_len, int fp_probd, int count)
{
  d = fp_probd - count * fp_len; /* denominator of drop-probability */
  if(d <= 0)
    return(1);   /* count exceeds the hard limit : drop */

  if(random() % d) < fp_len)
  {
    return(1);   /* drop */
  }
  return(0);   /* no drop */
}
```

To simulate the random behaviour of the dropper, the kernel uses a random number given by the `random()` function (it generates numbers in the interval $[0, 2^{31} - 1]$). The use of modulo operator (`%` symbol) ensures that the random number is always in the interval $[0, d - 1]$ like the numerator. This interval is split into two sub-intervals : $[0, fp_len[$ and the interval $[fp_len, d - 1]$. If the random number x belongs to the first sub-interval (i.e. $x < fp_len$), a small value of `fp_len` results in a small drop probability.

3.5.4 Statistical Variables

To check and validate this implementation, some experiments have to be done. During these experiments, several statistical variables maintained by the router will be collected to evaluate its performance. Here the focus is on the data that the router must collect to provide enough information to validate this implementation. Information that is collected :

queue[i].bytes	amount of bytes currently in the queue i
queue[i].nbpacket	number of packets currently in the queue i
queue[i].last_del_pack	queueing delay of the last transmitted packet
queue[i].sent_packets	number of packets sent by this queue
queue[i].drop_packets	number of packets dropped by this queue
queue[i].sent_bytes	amount of bytes already sent by this queue
queue[i].drop_bytes	amount of bytes already dropped by this queue

Additional information that collected :

AvgQS	value of the current average queue size
tot_nbpacket	current queue size (instantaneous)

These variables are updated :

- *(when a packet arrives).* The `wtp_enqueue` function is called. The `AvgQS` is updated.
  - *(if the packet is accepted).* The router will determine the destination queue i and update the following values : `queue[i].bytes`, `queue[i].nbpacket` and `tot_nbpacket`.
  - *(if the packet is not accepted).* The router will also determine the queue i to which the packet would belong and update the following values : `queue[i].drop_packets` and `queue[i].drop_bytes`
- *(when a packet leaves).* The `wtp_dequeue` function is called. If the packet belongs to the queue i, the following values are updated : `queue[i].bytes`, `queue[i].nbpacket`, `queue[i].last_del_pack`, `queue[i].sent_packets` and `queue[i].sent_bytes`.

The differentiation can be checked by inspecting these variables :

Delay differentiation If the value of `last_del_pack` is monitored for all queues, the queueing delay will be shown and it is possible to check if the expected differentiation is reached based on the configured weight for each queue.

Loss differentiation To check the loss differentiation, the number of dropped packets and the total number of packets is monitored to evaluate the drop ratio. Based on the configured WRED parameters, it can be determined whether the differentiation is correct or not.

Bandwidth differentiation *(for TCP traffic only).* The bandwidth used by a queue can easily be known if the amount of bits b send during a given time t is also known. Indeed, $\frac{b}{t}$ represents this bandwidth. Based on the formula from Mathis et al. ([20]), the bandwidth differentiation can be approximated and it's possible to check whether the differentiation is good or not. The bandwidth that a queue receives is given by :

$$throughput \leq \frac{MSS * C}{RTT * \sqrt{l}}$$

The WRED dropper will affect the loss ratio l while the WTP scheduler will affect the RTT value.

More statistics can be imagined and implemented inside the kernel but it's important to pay attention to the required resources (in terms of memory and CPU) for maintaining these variables. That's why an external program will be used to do complex operations. In that way, the kernel has to maintain simple counters.

Statistical program

The ALTQ package contains a lot of implementation and some statistical programs. These can only be used with the associated mechanisms but these are good examples that can help to construct new statistical program in connection with this implementation. This program must ask to the kernel the value of variables. To ask something to the kernel, a system call is used. This implementation supports the `ioctl` system call to collect information. So, with an external program, it's possible to monitor the behaviour of the implemented router.

Description of statistical program Only a high-level description of the statistical program will be given here. This program will ask to the kernel every t seconds, the value of variables. This program can compute the real throughput per queue :

$$\frac{\textit{amount of bytes sent during the period } t_i - t_{i-1}}{t_i - t_{i-1}}$$

All collected data are written into files. These are used to produce several graphics to display the results.

Impact on kernel behavior Because of interruptions caused by the statistical program, the behaviour of the kernel is modified. Indeed, the kernel answers to the submitted requests (i.e. from the statistical program) and returns the value of several variables instead of normal processing. It is difficult to measure the real impact of these requests because the router is only a part of the kernel. It is also difficult to avoid this because the evaluation of this implementation can only be done by using these variables.

3.6 Conclusion

This chapter has discussed the main features of the FreeBSD operating system. The packet handling is also described in details and a description of ALTQ has shown the real flexibility of a FreeBSD based router. Finally, the proposed implementation of delay and loss differentiation was examined. Now, it's important to discuss about several limitations of this implementation.

3.6.1 Limitations of this implementation

Some limitations are FreeBSD dependent while others are dependent of implemented mechanism.

time granularity Because this implementation is done at kernel level, it is dependent on the time granularity provided by the FreeBSD kernel. The time is represented

as the number of seconds and microseconds elapsed since January 1, 1970 00:00:00 (UTC). It's not possible to be more accurate : if two packets arrive during the same microsecond, it's difficult to do a correct delay differentiation. The the maximum throughput that this implementation can handle is, in theory, 500 Gbps only because of the timer granularity.

memory occupancy This implementation requires 8 bytes (64 bits) per packet. To avoid an overflow, all queues are limited by a tail drop mechanism. If all packets have the same size L and all the maximum queue size is q : this queue can hold $\frac{L+8}{q}$ packets at the same time. If L is small, the 8 bytes overhead can be significant.

Chapter 4

Experiments

4.1 Introduction

This chapter describes experiments done to validate the proposed implementation. These results are useful to determine :

- "Quality" of this implementation : the implementation should be stable (i.e. no bugs) and provide good performance;
- "Correctness" of the implementation : it should match with the model and the results should match with our expectations.

Experiments vs Simulations

While several simulations are proposed (see [27]) about the studied mechanisms, it is important to show the interests of these experiments.

Experiments They are done using a real environment : a lot of computers with network equipment (i.e network interfaces, cables, hubs, ...) are needed. The traffic is real : packets are sent by sources throughout the network and are received by the destination.

Simulations They are done with a special program : the simulator. It allows the user to build his network topology and to set all network parameters (like MSS, RTT, link rate, ...). The network topology usually includes source and destination hosts, core and edge routers. All traffic is simulated : the simulator generates all events (i.e. packet arrival, packet departure, packet sent, packet received, ...) in connection with the simulation and computes all needed statistics.

The following table shows a summary of the differences between the experiments and simulations :

Description	Experiments	Simulations
Limitations	Limited by the available hardware	Limited by the simulator features and qualities
Traffic generators	Any network program can be used to generate traffic (Web, FTP, ...)	Only traffic generators provided by the simulator can be used.
Performance measures	Done by an external program	Can be integrated inside the simulator
Implementation sensitivity	High, all optimizations have an important impact on the performance	Low. The network is simulated.
Usage	Used to validate simulations and make mechanisms available	Used to validate theoretical model (i.e. new mechanisms).

Both simulations and experiments are useful because the usage is different. This chapter will present several experiments and present results from real environment in order to validate this implementation. Unlike simulations, if this implementation provides good results, it can be used inside IP routers to provide delay and loss differentiation.

This chapter is organized as follows : the first part shortly depicts traffic generators because they are used to generate random packets, the performance measure are done (to evaluate the quality of this implementation) and finally, the network configuration is presented and the results are discussed (exactness of the implementation).

4.2 Traffic generator

The experiments are done using traffic generators. Two traffic generators have been used : Netperf for TCP and UDP traffic. Traffic generators are useful to generate random traffic and evaluate the behaviour of implemented mechanisms.

Traffic generators contains usually two programs : a server and a client. The server runs on the destination host and listens on a selected port while the client runs on the sender hosts and send traffic to the receiver (using the selected port).

The quality of a traffic generator can be evaluated if it can reproduce the exact behaviour of TCP sources. Indeed, the Internet traffic has some observed characteristics (like TCP flows are usually less than 6 kb) that must be integrated inside the generator to provide realistic traffic (and not ideal traffic).

4.3 Network configuration

The network topology is depicted on figure 4.1. Five hosts are used : three sources (A, B, C), one router (D) and one destination (E).

Experiments have the following characteristics :

Hosts All hosts are COMPAQ Pentium 200 Mhz running FreeBSD 2.2.8. Five computers are used :

- Three computers (A, B, C) are used as source.
- One computer (D) is configured as a router. This router implements WRED and WTP mechanisms.

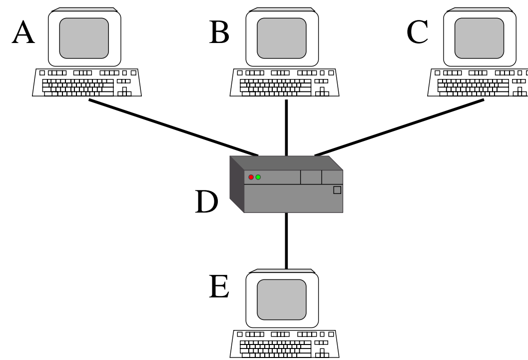


Figure 4.1: Network topology for implementation validation

- One computer (E) is the receiver.

Network link All network links are fast Ethernet(100 Mbit/s)

Marking All packets from one source can only be marked at the outgoing network interface (because of external marking program) : so three different DSCP values can be used (they will be specified before the experiment results).

Many flows will share the link between the router and the receiver (link D–E). This link will be congested and the router will react according to the configuration parameters.

The following parameters are used for all experiments :

- Number of drop precedences = 3
- Number of delay queues = 3
- Link rate = fast Ethernet (100 Mbits / s)
- Packet size = 1500 bytes
- Maximum queue size = 512 kilobytes
- WTP parameters :

Queue	q_i
$Queue_{low}$	1
$Queue_{med}$	$\frac{1}{2}$
$Queue_{high}$	$\frac{1}{3}$

  $Queue_{low}$ means the queue with the largest delay while $Queue_{high}$ means that the queue is served with the smallest delay.

- WRED parameters :

Description	value
Min_{th}	300 packets
Max_{th}	1000 packets
$Drop_{high}$	$\frac{1}{10}$
$Drop_{med}$	$\frac{1}{20}$
$Drop_{low}$	$\frac{1}{40}$

Recall that $Drop$ is the drop ratio when the average queue size reaches the Max_{th}. The more this value is high the low it is good (i.e. the loss rate is high). In this case, $Drop_{high}$ experiences the largest drop rate while $Drop_{low}$ experiences the smallest loss rate. The fixed thresholds (Min_{th} and Max_{th}) can appear high but the considered queue size is the total average queue size (not per queue). Moreover, when the minimum threshold is high, that means that the dropper is not aggressive : that's good for TCP like flows (they reduce their sending rate when packet drop occurs). So, packet drop occurs when the total average queue size reaches 450 kb[1] and it discards all incoming packets when this average queue size is more than 1,5 Mb (remark that this maximum threshold is reach when all queues have reach their limits of 512 kb). This loss differentiation is depicted on figure 4.2.

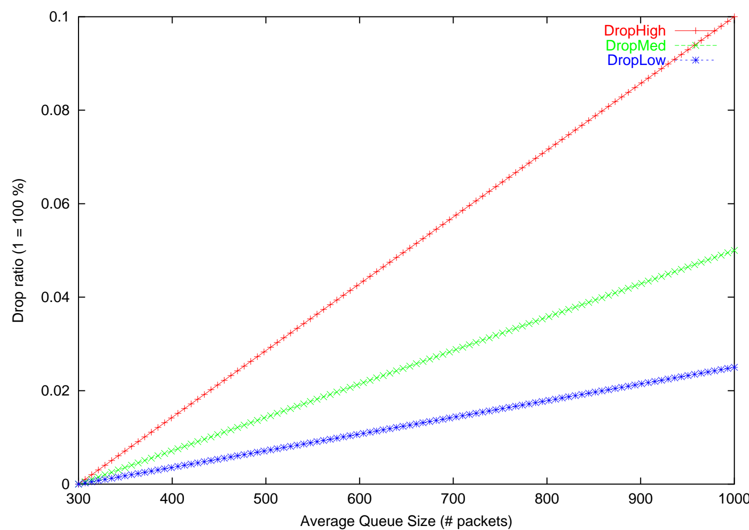


Figure 4.2: Expected drop ratio in the interval [300,1000]

[1]if 1 kb = 1 000 bytes

4.4 Experiment results

4.4.1 Background

Before exposing results, it is important to recall that the throughput for TCP traffic is estimated using the formula from Mathis et al. (see [20]) :

$$Throughput \leq \frac{MSS * C}{RTT * \sqrt{l}}$$

Where :		
	MSS	is the maximum segment size
	C	is a constant dependent on the TCP implementation
	RTT	is the round-trip-time
	l	is the loss ratio

Recall also that queueing delays have impacts on round-trip-time (RTT) parameter while drop precedence have impacts on loss ratio l. The following formula take into account the number of TCP connections (because this number has a effect on the throughput for TCP micro-flows) :

$$Throughput \leq \frac{MSS * C}{RTT * \sqrt{l}} * NumberOfTCPconnections \tag{4.1}$$

4.4.2 Heavy-load condition

Parameters

This first experiment shows the behaviour of this mechanisms in case of heavy load condition : it is important to see if this mechanism can support a highly loaded network because this situation can appear. The parameters for this experiment are the default parameter, but instead UDP traffic is used : the main consequence is that no congestion control algorithm is implemented inside UDP protocol (unlike TCP, UDP doesn't reduce its sending rate when packet drop occurs) so it is easier to obtain heavy load condition with UDP traffic. In this case, it is difficult to foresee the behaviour of the mechanism because :

- UDP traffic is used : the throughput can't be estimated with the formula 4.1.
- The queue size will contains always 1000 packets or more and the loss differentiation will not be good (all incoming packets will be dropped because the maximum threshold is always reached).

Results

The average queue size (see figure 4.3) reaches the maximum threshold : 1000 packets and the router is always congested as can be expected : the heavy load conditions are verified. The real queue size often exceeds the maximum threshold (it is normal because the dropper only takes into account the average queue size and not the real queue size) and should cause an overflow : that's why all queues are protected by a tail drop mechanism that avoid overflows : if the queue size is 512 kb, all incoming packet will be rejected.

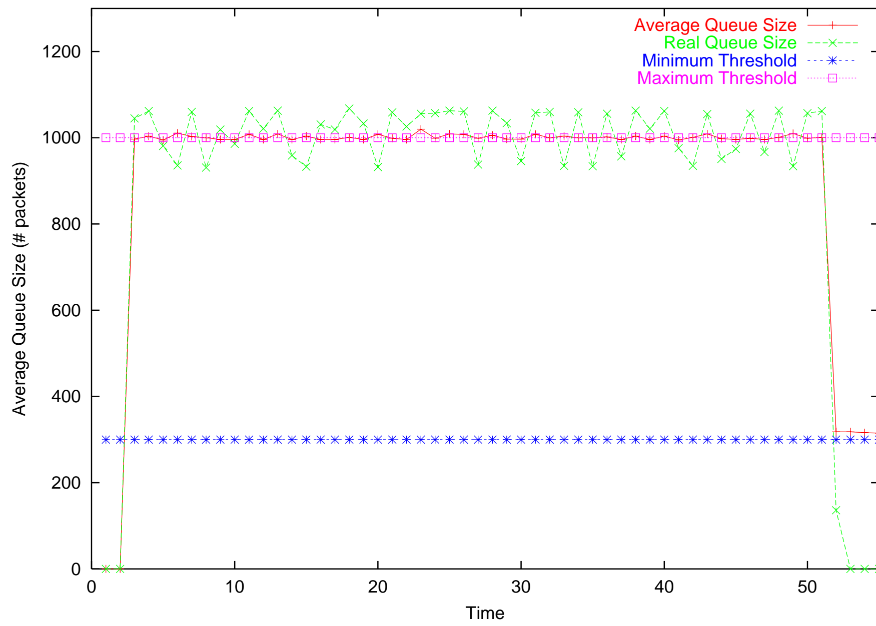


Figure 4.3: Average queue size

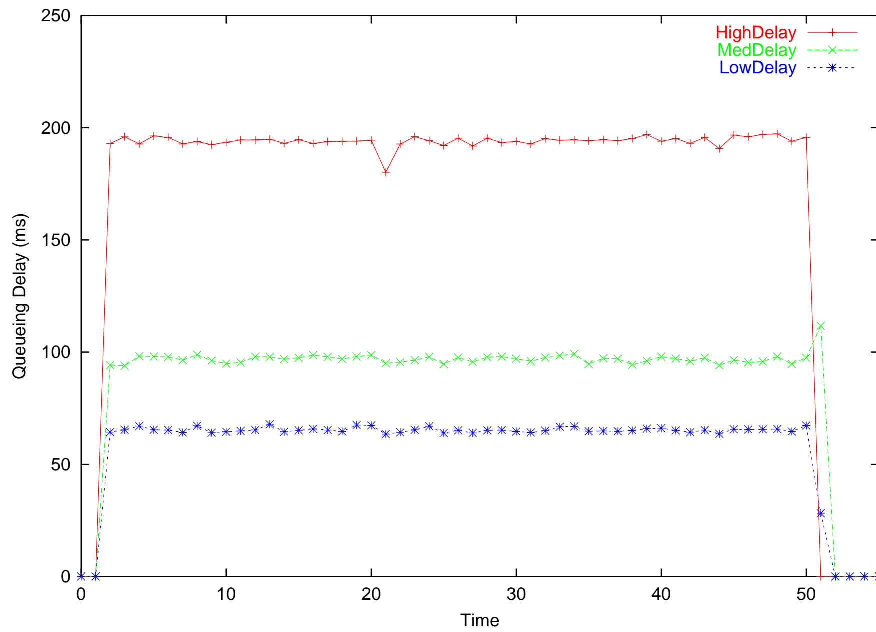


Figure 4.4: Queueing delay

Delay differentiation works quite well (see figure 4.4) according to the configured pa-

rameter :

$$QueueingDelay_{LowDelay} = 3 * QueueingDelay_{HighDelay}$$
$$QueueingDelay_{MedDelay} = 2 * QueueingDelay_{MedDelay}$$

Because the delay differentiation is completely independent of the queue load, the expected delay differentiation is reached.

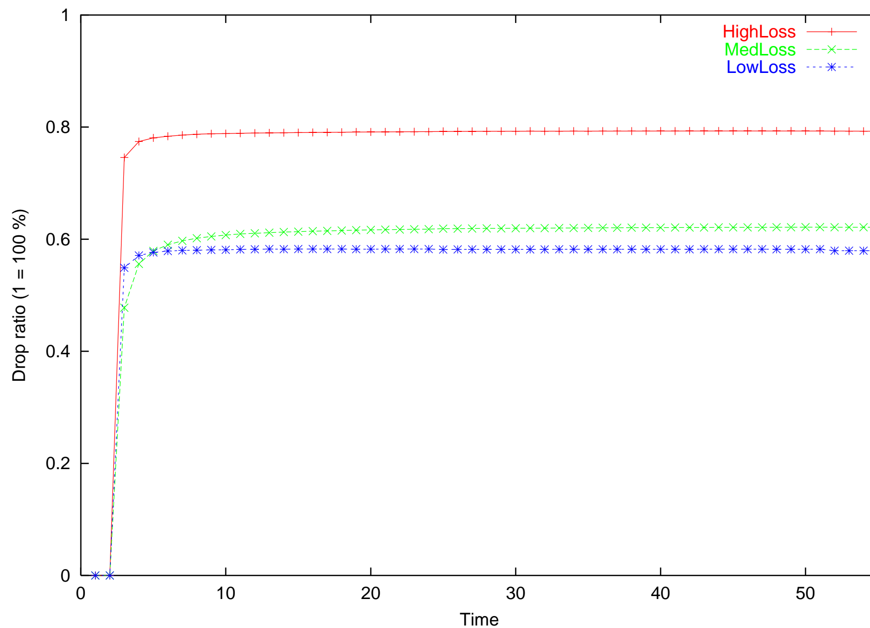


Figure 4.5: Drop per queue

As expected the loss differentiation (figure 4.5) doesn't match with the configuration, the reasons are :

- deterministic drop behaviour of the dropper when the router reaches the maximum threshold (i.e. all incoming packets are discarded) ;
- some packets are dropped by the tail drop mechanism.

This experiment shows that this implementation supports heavy load condition even if the loss differentiation is not good, the router reacts according to the implemented mechanisms. The use of UDP traffic instead of TCP traffic has only one consequence, the throughput differentiation can't be checked because of no congestion control mechanism in UDP protocol. Mechanisms for queue protection (like tail-drop) are indispensable to assure the stability of the router. This point is very important : an unstable router can't be tolerated.

4.4.3 Equal distribution (TCP)

Parameters

In addition to the default parameters described above, the following marking parameters are used :

Packets from source	requested delay	requested drop	# TCP connections	Queue number
A	$Queue_{low}$	$Drop_{high}$	30	0
B	$Queue_{med}$	$Drop_{med}$	30	1
C	$Queue_{high}$	$Drop_{low}$	30	2

In this case, packets from source A should experience the largest queueing delay (among the three defined queues) and the highest loss-rate while the queueing delay for the source B should be half of the source A's queueing delay and the half of the loss-rate. The last source C should have a queueing delay that is the third and a loss-rate that is the fourth of those of source A. All host have 30 TCP connections to the receiver.

Theoretical results

This measure should show that the implementation meets our expectations when the traffic is distributed equally (all queues have the same number of TCP connections). This case is not realistic (because the traffic distribution is rarely equally distributed) but it allows to show and explain easier the expected values and deviations.

Based on the configuration, the following bandwidth differentiation can be expected ($throughput_0$ means throughput of $Queue_0$ from source A i.e. queue with largest queueing delay and loss rate) :

$$throughput_0 \leq \frac{MSS * C}{RTT_0 * \sqrt{l_0}}$$

The medium queue $Queue_1$ (i.e. the queue with medium delay and medium loss rate) is served with (remark that the fixed delay like processing or propagation delay is assumed to be zero) :

$$throughput_1 \leq \frac{MSS * C}{RTT_1 * \sqrt{l_1}} \Leftrightarrow throughput_1 \leq \frac{MSS * C}{\frac{RTT_0}{2} * \sqrt{\frac{l_0}{2}}}$$

And finally the high quality queue $Queue_2$ (i.e. lowest delay and loss rate) have the following throughput :

$$throughput_2 \leq \frac{MSS * C}{RTT_2 * \sqrt{l_2}} \Leftrightarrow throughput_2 \leq \frac{MSS * C}{\frac{RTT_0}{3} * \sqrt{\frac{l_0}{4}}}$$

The following bandwidth proportional differentiation is obtained :

$$throughput_1 \leq 2 * \frac{1}{\sqrt{\frac{1}{2}}} * throughput_0 \Leftrightarrow throughput_1 \leq 2.82 * throughput_0 \quad (4.2)$$

$$throughput_2 \leq 3 * \frac{1}{\frac{1}{2}} * throughput_0 \Leftrightarrow throughput_2 \leq 6 * throughput_0 \quad (4.3)$$

Because the maximum link rate is 100 Mbit/s, the following throughput can be expected for each queue :

$$\begin{aligned}100\ Mbps &= throughput_0 + throughput_1 + throughput_2\\ \Leftrightarrow 100 &= throughput_0 + 2.82 * throughput_0 + 6 * throughput_0\\ \Leftrightarrow 100 &= throughput_0 * (1 + 2.82 + 6)\\ \Leftrightarrow \frac{100}{9.82} &= throughput_0\end{aligned}$$

Finally, the following throughput is obtained (these are depicted on figure 4.6) :

$$\begin{aligned}throughput_0 &= 10.183299\ Mbps\\ throughput_1 &= 2.82 * throughput_0 = 28.716904\ Mbps\\ throughput_2 &= 6 * throughput_0 = 61.099796\ Mbps\end{aligned}$$

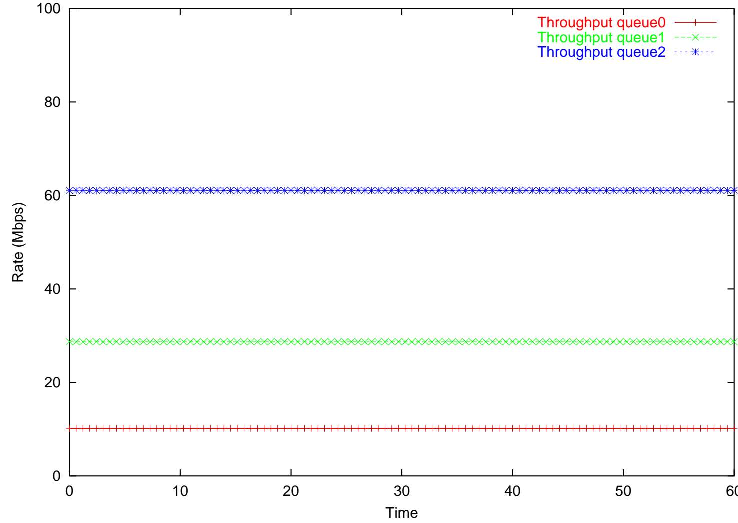


Figure 4.6: Expected Throughput per queue

Because the distribution of the traffic is equal among the queues, the following throughput per TCP micro-flows can be expected :

$$\begin{aligned}throughputFlowsQueue_0 &= \frac{10.183299Mbps}{30} = 0.33944331Mbps\\ throughputFlowsQueue_1 &= \frac{28.716904Mbps}{30} = 0.95723014Mbps\\ throughputFlowsQueue_2 &= \frac{61.099796Mbps}{30} = 2.0366599Mbps\end{aligned}$$

The throughput per TCP micro-flows is very important for the client because it shows exactly the service rate that he will receive.

It's difficult to have an approximation of the queueing delay because it depends on the queue size and it can be evaluated in an easy way. But the delay differentiation must be conform with the configuration :

$$QueueingDelay_{Queue_2} = \frac{QueueingDelay_{Queue_0}}{3}$$
$$QueueingDelay_{Queue_1} = \frac{QueueingDelay_{Queue_0}}{2}$$

For the loss differentiation, the figure 4.2 shows the differentiation as a function of the total average queue size. For instance, if the current average queue size is 500 packets, the following drop probability are obtained :

$$Packet_{HighDrop} = 0.028$$
$$Packet_{MedDrop} = 0.014$$
$$Packet_{LowDrop} = 0.007$$

Results

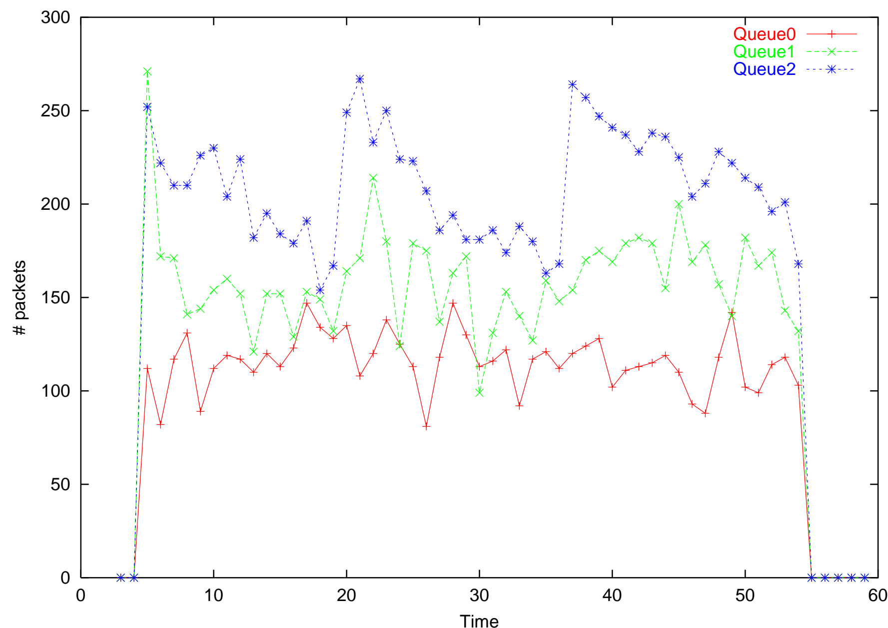


Figure 4.7: Queue occupancy

Figure 4.7 shows the queue occupancy : it is not stable because of congestion control mechanism of TCP sources. Indeed, TCP uses an AIMD (additive increase, multiplicative decrease) algorithm. When no packet drop occurs, TCP increase its sending rate with 1 packet (i.e. additive increase) but when packet drop occurs, TCP reduces its sending rate with a factor of two. It's also important to remark that the statistical program makes its requests every two seconds (that can explain why the decreasing of the queue occupancy

appears smooth). The queue occupancy will be used to check if the queueing delay is good and if the drop ratio is acceptable. The following average can be observed from figure 4.7 :

$$\begin{aligned} AverageQueueSize_0 &= 116\ packets \\ AverageQueueSize_1 &= 164\ packets \\ AverageQueueSize_2 &= 211\ packets \\ AverageQueueSize_{global} &= 491\ packets \end{aligned}$$

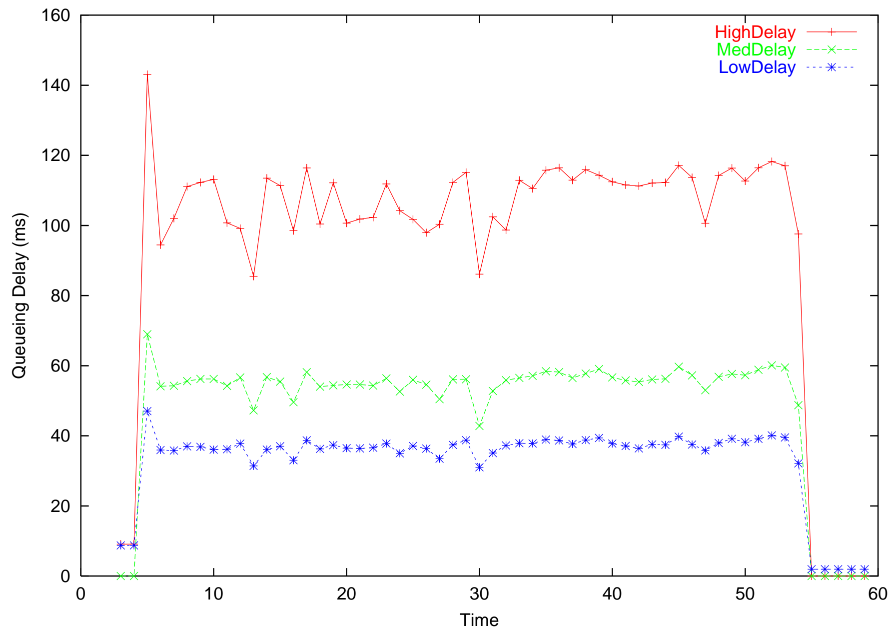


Figure 4.8: Queueing delay

Based on the computed average queue occupancy, the average queueing delay can be computed as follows :

$$QueueingDelay = \frac{NumberOfPacket * PacketSize * 8}{Rate_{queue}}$$

where $NumberOfPacket$ is the average queue size (in packets)
$PacketSize$ is the size of a packet in bytes (i.e. 1500 bytes)
$Rate_{queue}$ is the rate at which the queue is served in bps

Because the size of the queue must be expressed in bits, the numerator is multiplied by 8.

The following average queueing delay can be expected (according to the rate of each queue) :

$$QueueingDelay_0 = \frac{116 * 1500 * 8}{10183299} = 0.13669441\ s = 136.69\ ms$$

$$QueueingDelay_1 = \frac{164 * 1500 * 8}{28716904} = 0.068531064\ s = 68.53\ ms$$
$$QueueingDelay_2 = \frac{211 * 1500 * 8}{61099796} = 0.0414404\ s = 41.44\ ms$$

As can be seen on figure 4.8, a small deviation is observed, the queueing delay is less than the expected value. Observed average queueing delays are :

$$HighDelay = QueueingDelay_0 = 109\ ms$$
$$MedDelay = QueueingDelay_1 = 56\ ms$$
$$LowDelay = QueueingDelay_2 = 38\ ms$$

This small deviation can be explained by the use of an “expected” throughput and this expected throughput can be different from the real throughput (the throughput differentiation will be analyzed in the sequel). But, the delay differentiation is quite good : the queueing delay of the queue $LowDelay$ is one third of the queueing delay of the queue $HighDelay$ while the queueing delay of the queue $MedDelay$ is half the one of $HighDelay$. This corresponds to the configured parameters.

The warm up period must be ignored, it corresponds to the first seconds : this period is the time needed by the mechanism to reach a stable behaviour. This period must be as small as possible and is the consequence of the scenario : the statistical program is started first (so empty statistics are collected because no sources have been started) then the sources. In this case, the warm up period is the time needed by the sources to reach their sending rate.

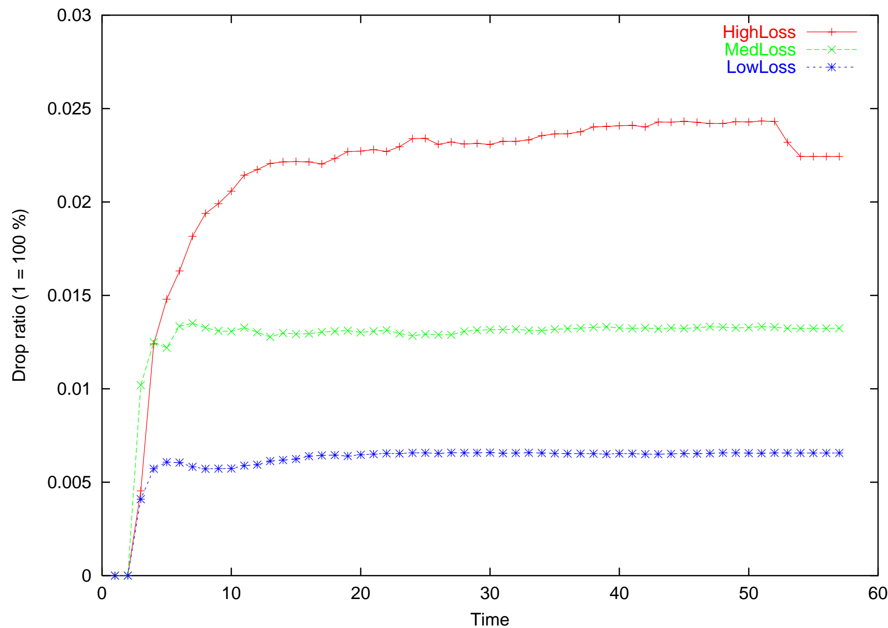


Figure 4.9: The dropped packet ratio

Figure 4.9 shows the dropped packet ratio. Based on the theoretical values (see figure 4.2), it appears that, with an average queue size of 491 packets, the following drop ratios can be expected :

$$HighLoss = DropQueue_0 = 0.027$$
$$MedLoss = DropQueue_1 = 0.013$$
$$LowLoss = DropQueue_2 = 0.006$$

And a small deviation can be observed (see figure 4.9) here especially for the $DropQueue_0$. but the loss differentiation is good too : the experiment parameters are configured such that the difference in loss ratio between two consecutive classes is a factor of two. The dropped packet ratio of the $MedLoss$ class is almost half the dropped packet ratio of the $HighLoss$ while the loss of $LowClass$ is half the dropped packet ratio's $MedLoss$ (the warm up period must be ignored).

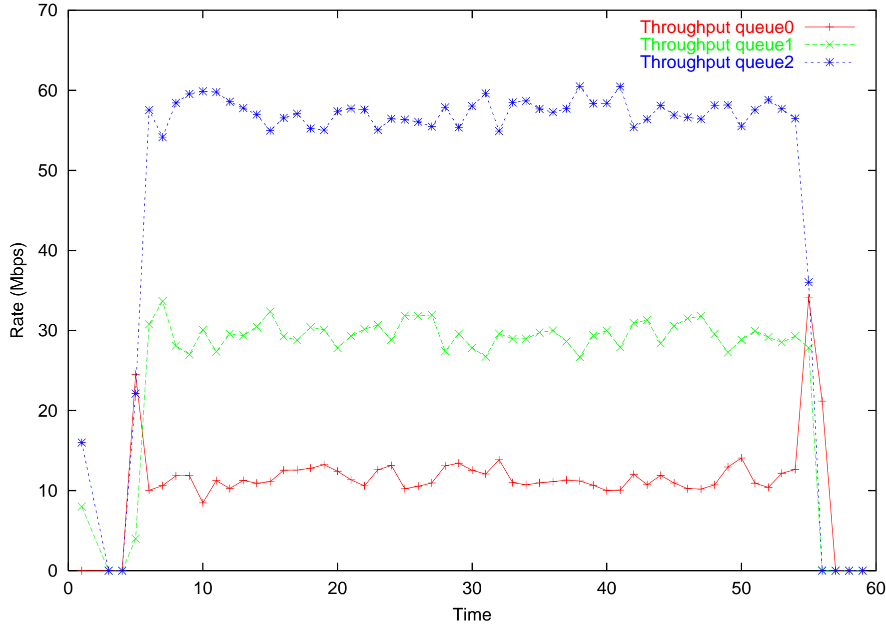


Figure 4.10: Throughput per queue

Figure 4.6 showed the expected throughput per queue while figure 4.10 shows the real throughput per queue. The real throughput matches almost with the expected throughput : small deviations can be observed but the throughput differentiation is almost good : as can be seen, the $throughputQueue_2$ is less than our expectations :

$$Average\ ThroughputQueue_0 = 12.357885\ Mbps$$
$$Average\ ThroughputQueue_1 = 28.464231\ Mbps$$
$$Average\ ThroughputQueue_2 = 55.135192\ Mbps$$

The throughput per queue and throughput per TCP micro-flows present the same differentiation (and same small deviation in comparison with the theoretical values) : that's

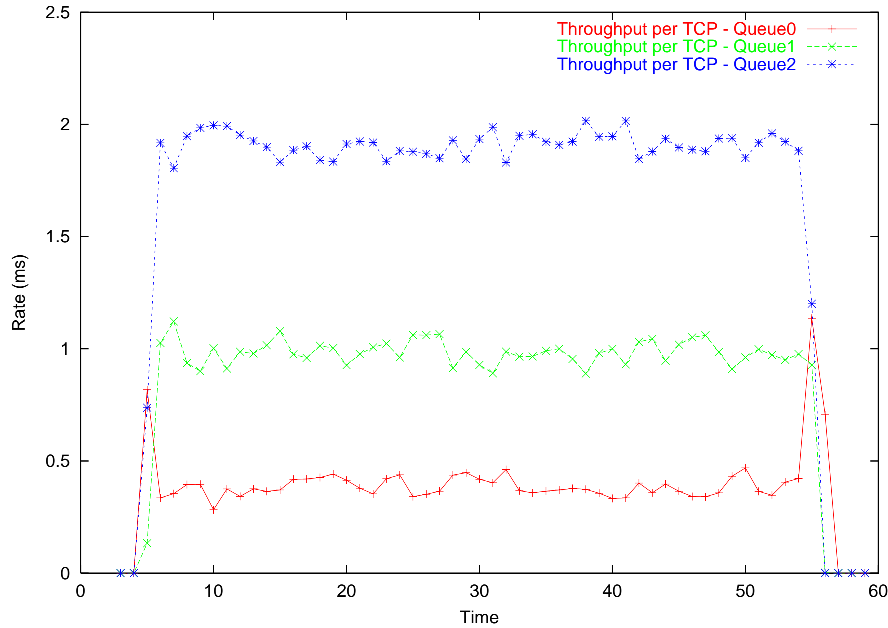


Figure 4.11: Throughput per TCP micro-flows

normal because the number of TCP connections are the same for all senders (an experiment with different numbers of TCP connections will be shown in the sequel). The bandwidth differentiation is good according to the configured parameters.

4.4.4 Unequal distribution

Parameters

Five hosts are needed : three sources (A, B, C on the figure 4.1), one router (D on the same figure) and one destination (E on the figure). The marking is configured as follows (all other parameters are the same as previous experiment) :

Packets from source	requested delay	requested drop	# TCP connections	Queue number
A	$Queue_{low}$	$Drop_{high}$	60	0
B	$Queue_{med}$	$Drop_{med}$	20	1
C	$Queue_{high}$	$Drop_{low}$	10	2

Theoretical results

In the previous experiments, the stable behaviour have already been checked, the loss, delay and bandwidth differentiation have been verified. This experiment will focus on the throughput per TCP micro-flows : it will attempt to check if this throughput is good when the number of TCP connections are not the same among sources. It's important to make this experiments because an unequal distribution is common on the Internet.

Because the $Queue_0$ (i.e. worst queue) has 60 TCP connections, it receives almost the same TCP throughput as the other queues. Indeed, the following relations are obtained :

$$\begin{aligned} Q_0 &= \frac{MSS * C}{RTT_0 * \sqrt{l_0}} \\ throughput_0 &\leq Q_0 * 60 \\ throughput_1 &\leq Q_0 * 20 * 2.82 \\ throughput_2 &\leq Q_0 * 10 * 6 \end{aligned}$$

The maximum link rate is the same as previous 100 Mbps, so the following throughput can be expected for each queue :

$$\begin{aligned} 100\ Mbps &= throughput_0 + throughput_1 + throughput_2 \\ \Leftrightarrow 100 &= Q_0 * 60 + Q_0 * 56.4 + 60 * Q_0 \\ \Leftrightarrow 100 &= Q_0 * (60 + 60 + 56.4) \\ \Leftrightarrow \frac{100}{176.4} &= Q_0 \end{aligned}$$

Finally, the following throughput is obtained (these are depicted on figure 4.6) :

$$\begin{aligned} throughput_0 &= \frac{100 * 60}{176.4} = 34.013605\ Mbps \\ throughput_1 &= \frac{100 * 56.4}{176.4} = 31.972789\ Mbps \\ throughput_2 &= \frac{100 * 60}{176.4} = 34.013605\ Mbps \end{aligned}$$

Remark that the $throughput_0$ and $throughput_2$ values are the same (that is why these two curves are mixed together on figure 4.6).

The following throughput per TCP micro-flows can be expected (depicted on figure 4.13) :

$$\begin{aligned} throughputFlowsQueue_0 &= \frac{34.013605\ Mbps}{60} = 0.56689342\ Mbps \\ throughputFlowsQueue_1 &= \frac{31.972789\ Mbps}{20} = 1.5986394\ Mbps \\ throughputFlowsQueue_2 &= \frac{34.013605\ Mbps}{10} = 3.4013605\ Mbps \end{aligned}$$

Results

As can be expected and shown on figure 4.14, the bandwidth differentiation is not so clear than the previous simulation. The reason is that the number of TCP connections is not the same as before.

Because the $Queue_0$ (i.e. worst queue) has 60 TCP connections, it receives almost the same TCP throughput as other queues. Deviations can be observed from the theoretical

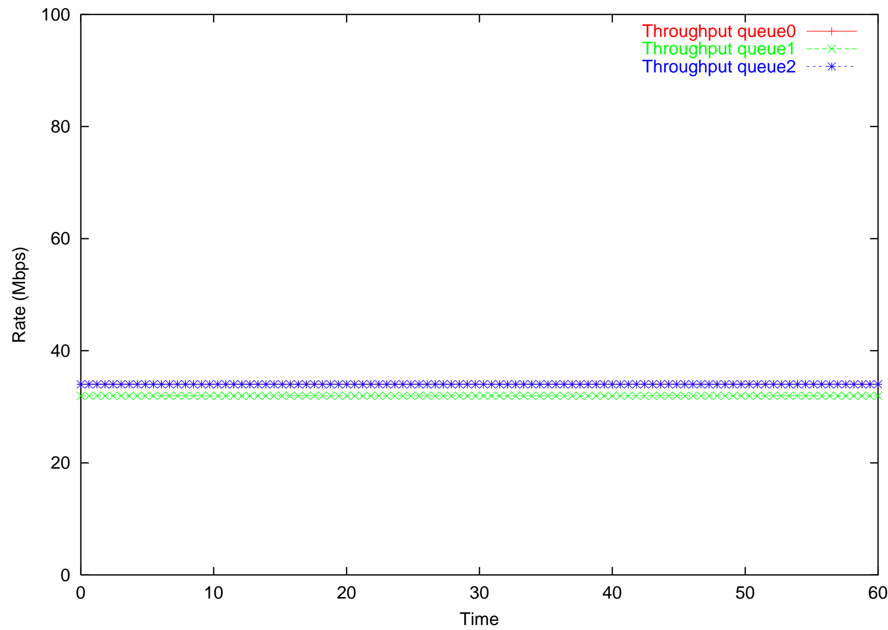


Figure 4.12: Expected Throughput per queue

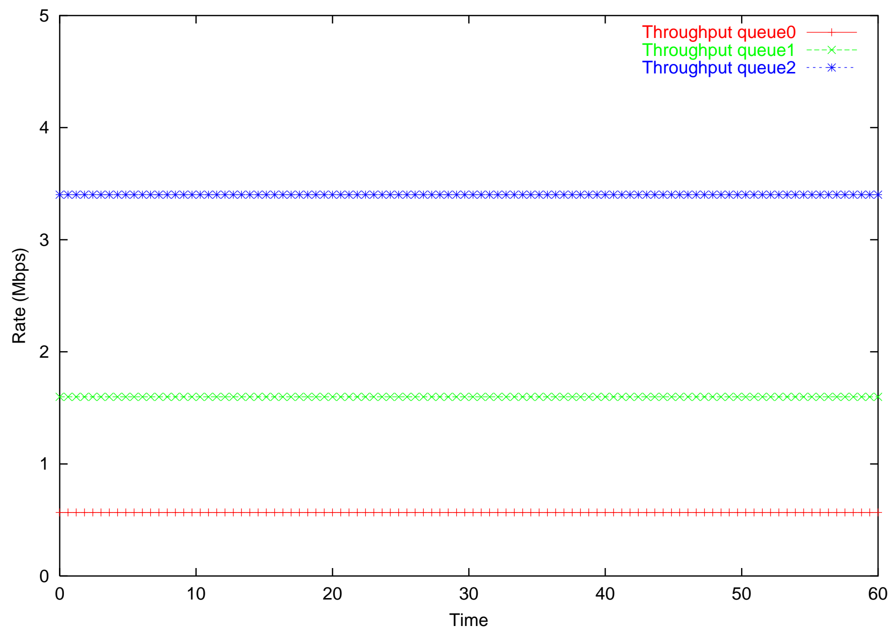


Figure 4.13: Expected Throughput per TCP micro-flows

results (figure 4.12), a light "starvation effect" appears from the queues with a higher number of TCP connections. Indeed, the $Queue_0$ receives a higher throughput than expected,

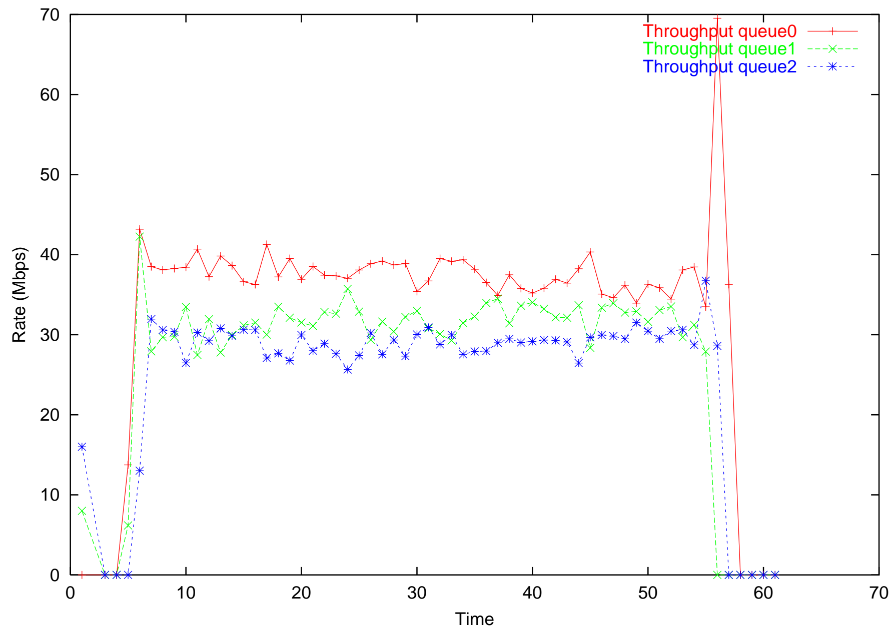


Figure 4.14: Throughput per queue

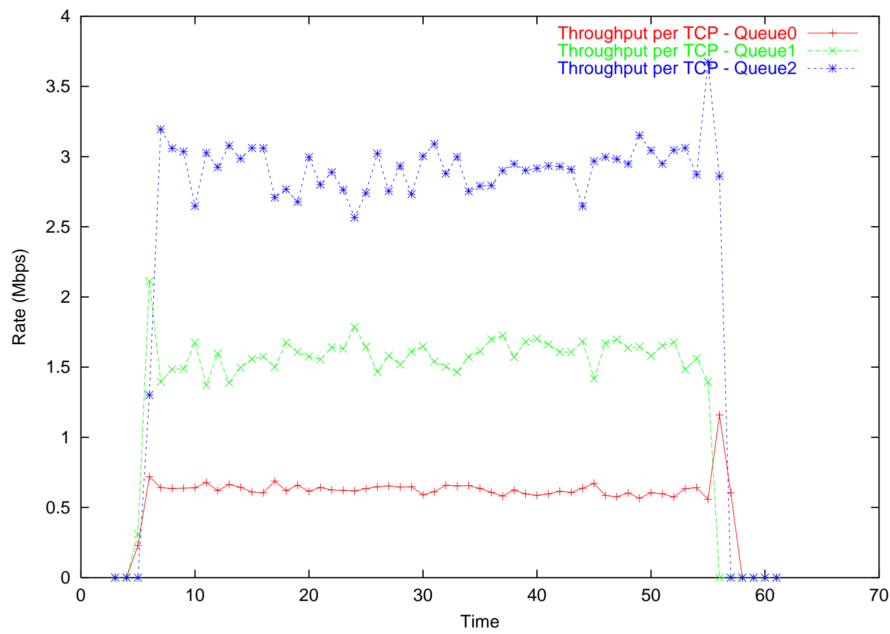


Figure 4.15: Throughput per TCP micro-flows

and the $Queue_2$ (with only 10 TCP connections) has a lower throughput than expected.

The most important curve in this experiment is the throughput per TCP micro-flows

(see figure 4.15) : the examination of throughput per TCP micro-flows (obtained if the $throughput_{queue}$ is divided by the $NumberOfTCPconnections$) shows that the differentiation is good (small deviations can be observed between real and theoretical results like in the throughput per queue figure), and $Queue_2$ (i.e. the best queue) receives a higher throughput than the previous classes. Like in the throughput per queue, the $Queue_0$ receives a throughput higher than expected but the deviation is small (i.e. 0.63 Mbps instead of 0.56 Mbps) while the $Queue_1$ obtains a throughput conform with our expectation and the $Queue_2$ has a small deviation (2.9 Mbps instead of 3.4 Mbps).

So this mechanisms works also when the traffic is not equally distributed.

4.5 Conclusion

This chapter has shown that the proposed implementation is stable and matches with the model through several experiments :

- The heavy load experiment has shown that this implementation is very stable even if the network is highly loaded. Although the differentiations can't be verified in this case, the results match the expected behaviour.
- The equal distribution experiment has shown that the studied proportional model is verified and the delay and loss differentiation are good according to the configured parameters.
- The unequal distribution experiment has shown that the throughput per TCP micro-flows (i.e. the most important throughput for the client) meets all expectations because it was correct even if the distribution of the traffic is not the same among the different classes.

More experiments can be done to evaluate this implementation, especially performance measures : because of short time, performance measures was not done This kind of measure does not require complex programs or new hardware devices, these measures propose to compare the performance provide by the default FCFS service with the new implemented mechanism. The results indicate the complexity of this new implementation.

Conclusion

This thesis described alternative architectures that can be used to provide more services : the Integrated Services and Differentiated Services architecture. The first attempts to provide strong and deterministic end-to-end guarantees (i.e. from the source to the destination hosts). The second, simpler, proposes several levels of services : according to the traffic contract, the incoming traffic is distributed among all classes to achieved the requested service. Integrated Services do not appear very interesting because of the scalability problem : the required resources to support this architecture are excessive and this model cannot be deployed at the scale of the Internet.

The thesis also discussed relative services : they propose to have different classes of service (each class is associated with a level of service that is measured in terms of local queueing delays and packet losses). The proportional differentiation model appears to be adapted because it is controllable (i.e. can be tuned by the network operator) and predictable. It was used to provide proportional delay, loss and bandwidth differentiation (only for TCP flows). Various mechanisms have been described and two mechanisms were selected to provide the proportional delay and loss differentiation : the WTP scheduler and the WRED dropper in shared buffer mode. The bandwidth differentiation is achieved by combining these two differentiations.

The thesis also showed how these mechanisms have been implemented inside a FreeBSD based router using ALTQ. Some limitations have been described like the time granularity that depends of the time service provided by the FreeBSD kernel or the memory occupancy, the WTP scheduler need memory to save the arrival time of each packet.

Finally, this thesis also proposed experiments to validate the implementation. Three experiments have been carried : heavy load condition, equal traffic distribution and unequal traffic distribution. The first demonstrated the stability of this implementation (this router can support an overloaded network). The second showed that the delay and loss differentiation work quite good according to the configured parameters. It also showed that the bandwidth differentiation per queue is also quite good. The third experiment showed that these mechanisms work also when the load is not equally distributed between classes of services.

The proposed implementation is, in fact, a prototype that shows how such mechanisms can be implemented. This kind of prototype can be used to evaluate the feasibility of an implementation and to assess the performance of new mechanisms. Indeed, the implementation phase is important because it shows all practical problems of a theoretical solution. In our case, the implementation of WRED and WTP has shown that these two

mechanisms can be implemented and can work together and that some technical problems like the unavailability of floating point data type can be bypassed quite easily. The second interest is to measure the performance of a new mechanism. This aspect is not really taken into account when using a simulator. The implemented mechanisms must be optimized unless the resulted performance will be less good. Great improvements can be achieved, for instance, by using faster operations (in WRED dropper) like the shift operation instead of the multiplication or divide operations.

More development can be done in connection with this work :

- It is important to measure the real performance of this mechanism : measure the overhead of this mechanism in comparison with the default FIFO.
- This prototype can be extended by implementing rate-base RED instead of WRED. Because RB-RED is a new mechanism, its implementation will allow its study into a real environment.
- Performing the user experiments : because the mechanisms are implemented, it is now possible to make new experiments with applications that are more relevant for the client. Indeed, all network applications can be used as traffic generators.

Appendix A

Implementation

This section contains the source code of the WTP scheduler and WRED dropper. It is written in C language and is composed of many files : some header files (.h extention) and others C source files (.c). A brief description of each source file can be found in the sequel.

This implementation was done at Alcatel Telecom's Corporate Research Center of Antwerp.

The included version is the 0.6 BETA. But, when it is compiled without the option `EXT_STAT`, the version can be considered as 0.5 STABLE.

The following files are included :

altq_wtp.h This files contains many basic definitions used by the WTP scheduler and WRED dropper. This file defines the DSCP values used, all data structures used to store arrival time (`mb` structure), queue description and additional data types used for statistic.

altq_wtp_opt.h This header file contains *all available compilation options*. It allows to select mechanisms and features to compile. Five options are allowed :

- **COMP_TIME** When this option is present, the implementation will use a compressed time instead of standard UNIX time `timeval` format.
- **Q_DELAY** This option allows to collect the queueing delay from an external program : useful if statistics are needed.
- **WTP_WRED** This option enables the WRED dropper with the WTP scheduler. By default, the WTP scheduler is associated with a tail-drop mechanism.
- **WRED_STAT** When this option is present : the kernel will provide statistic about the WRED dropper. Useful for statistic.
- **EXT_STAT** This option allows to collect more statistic (extended statistic). This option is experimental and must be used with caution.

altq_wtp_util.c This file contains some common functions. Indeed, it contains two functions, the first `elapsedFromNow` computes the time elapsed between two timestamps. The second `convertusec` converts a UNIX compatible time in microseconds (no test is done to avoid overflows so it must be used with caution).

altq_localq.c This is the main file in this implementation, it contains the WTP scheduler and the functions that are called when a packet arrives or leaves the router (`wtp_enqueue` and `wtp_dequeue` respectively). Functions included inside this file are described here :

- **refTime** This function updates the queue reference time with the arrival time of the packet at the head of the queue and returns it.
- **localopen** Function called to initialize all implemented mechanisms.
- **wtp_flush** This function allows to flush all delay queues attached to a given interface.
- **wtp_setenable** It allows to enable or disable the implemented mechanisms on a given interface.
- **wtp_ifattach** It allows to initialize all implemented mechanisms on a selected interface. This function reserves all required memory to store queues, collects statistics and configures all mechanisms with default values (3 queues in addition to the default queue, queue limit = 512 kb, WRED with 3 drop precedences).
- **wtp_ifdequeue** It contains all actions done when a packet leaves the router. The WTP scheduler is implemented in this function : it returns the packet that the router must transmit first. This function maintains also some statistics.
- **wtp_config** It allows to configure the WTP with other values than the default values.
- **classifypacket** This function provides the queue that corresponds with a given DSCP value on a given interface. If the DSCP value is not valid, the default queue is returned.
- **wtp_enqueue** It contains all actions done when a packet arrives at the router : it relies on several functions to make the classification and to discard the packet if necessary (WRED).
- **wtp_ifdetach** Remove the WTP/WRED mechanisms on a selected interface : all queues are flushed (all packets are discarded).
- **wtp_getstats** This function is used when a request about statistic information (on WTP/WRED) is received by the kernel.
- **wtp_setweight** This function is called when the weights for the WTP scheduler must be set or updated.
- **localqclose** This function is called when the mechanisms must be desallocated.
- **stat_reset** This function resets all statistic variables.
- **localqioctl** This is the main ioctl function for WTP/WRED. It is the interface between the kernel and all requests made by a user or on external program. Allowed operations are : enable or disable (with `wtp_setenable` function), attach on a given interface (`wtp_ifattach` is called) or detach (by using `wtp_ifdetach` function), setting the weights of WTP scheduler (with `wtp_setweight` function), configure WTP (with `wtp_config` function), obtain all current statistic values (by using `wtp_getstats`) or reset these (with `stat_reset`). It allows

also to configure the WRED mechanism (by using the `wred_conf` fonction included in another source file) or obtain statistic values (with `wred_getinfo` or `wred_getstats` functions).

altq_wtp_wred.h This file contains all data type descriptions usefull for the WRED mechanism.

altq_wtp_wred.c It includes all functions for WRED mechanism. Most important are :

mainwtab_alloc This function is used to compute the value of $(1 - Wq)^t$. It allocates the lookup table that contains all $(1 - Wq)^{2^t}$ values.

wred_pow_n This function computes $(1 - Wq)^t$ values by using the previous function.

wred_drop_early According to its drop precedence, this function decides if the packet has to be dropped.

allocWRED It allocates all needed variables to configure the WRED according to the given parameters. The configurable parameters are : the thresholds, number of drop precedences and associated maximum drop probabilities and the weight for the EWMA.

allocWRED_default This function allocates a new WRED with default values : minimum threshold = 200, maximum threshold = 1000, number of drop precedences is 3 and the associated drop probabilities are 0.2, 0.1 and 0.05. The default value for the EWMA weight is 512.

acceptPacket This function is the heart of the WRED mechanism, it updates the average queue size on packet arrival and it decides whether a packet must be discarded (by using wred_drop_early finction) according to the extracted drop precedence

wred_conf This function allows to reconfigure the WRED mechanism.

wred_getinfo This function returns the current configuration of the WRED mechanism.

wred_getstats This function provides all available statistics on the WRED mechanism.

wtpd.c This file includes the (interactive) configuration program. It allows to enable, attach and configure WTP / WRED on a given interface.

wtpstat.c This file includes the statistical program. This program can work interactively or in background. In interactive mode, this program asks to the kernel all statistic variables at given time (i.e. every x seconds and x can be configured) and displaies all values for WRED and WTP. This program also allows to know the current configuration of all mechanisms and the following statistical information :

- WTP scheduler :
  - instantaneous queueing delay
  - instantaneous size of all queues

  - instantaneous number of packets or bytes sent / dropped by queues
  - current throughput of all queues
- WRED dropper :
  - number of packets and bytes sent / dropped for all drop precedences
  - instantaneous average queue size
  - current queue size

In background mode, the statistical program saves all values into CSV files such that all values can be used easily by various program.

A.1 altq_wtp.h

The first section contains all definitions used for WTP implementation.

```
/*
 * WTP : Waiting Time Priority scheduler implementation for ALTQ 1.2
 * by Louis SWINNEN - FUNDP & ALCATEL CRC ANTWERPEN
 *
 * Version 0.6 BETA
 */

/*
 * Classifier method: we will use DSCP (DiffServ CodePoint) field to put
 * a packet in the good class.
 *
 * Values:
 * 000000 default class
 * 001000 class 1 lower delay than default class
 * 010000 class 2 lower delay than previous class
 * 011000 class 3 lower delay than previous class
 * 100000 class 4 lower delay than previous class
 * 101000 class 5 lower delay than previous class
 * 110000 class 6 lower delay than previous class
 * 111000 class 7 lowest delay class
 *
 * these DSCP values are fixed automaticaly !
 *
 * Maximum 7 delay classes are allowed (+ default class)
 */
#ifndef _NETINET_ALTQ_WTP_H
#define _NETINET_ALTQ_WTP_H

#ifdef WTP_WRED
#include<netinet/altq_wtp_wred2.h>
#endif /* WTP_WRED */

#include<netinet/in.h>

#define MIN_QNUMS 2    /* Minimum : 2 queues (1 default and 1 delay) */
#define DEFAULT_QNUMS 3 /* by default, 3 queues (2 delay queue and 1 default queue) are created */
#define MAX_QNUMS 8 /* currently, maximum 8 queues (1 default and 7 delay queues) are allowed */
#define DEFAULT_QLIMIT (512 * 1024) /* by default, queue size is 512 K */
#define MIN_QLIMIT 1500 /* the minimum queue size is 1500 bytes */
#define MAX_QLIMIT (1024 * 1024)
#define DSCP(i) ((i)==0?0:(i)==1?0x08:(i)==2?0x10:(i)==3?0x18:(i)==4?0x20:(i)==5?0x28:(i)==6?0x30:(i)==7?0x38:0)
#define DSCP2int(i) ((i)==0?0:(i)==0x08?1:(i)==0x10?2:(i)==0x18?3:(i)==0x20?4:(i)==0x28?5:(i)==0x30?6:(i)==0x38?7:-1)

/* #if defined(KERNEL) || defined (_KERNEL) */

#define ENABLE            0
#define DISABLE            1
#define WTP_ENABLE        _IOW('Q', 1, struct wtp_interface)
#define WTP_DISABLE        _IOW('Q', 2, struct wtp_interface)
#define WTP_IF_ATTACH    _IOW('Q', 3, struct wtp_interface)
#define WTP_IF_DETACH    _IOW('Q', 4, struct wtp_interface)
```

```
#define WTP_SET_WEIGHT    _IOWR('Q', 5, struct wtp_setweight)
#define WTP_CONFIG        _IOWR('Q', 6, struct wtp_conf)
#define WTP_GET_STATS    _IOWR('Q', 7, struct wtp_getstats)
#define WTP_STAT_RESET    _IOW('Q', 8, struct wtp_interface)
#ifdef WTP_WRED
#define WRED_CONFIG        _IOWR('Q', 9, struct wred_config)
#define WRED_GETINFO    _IOWR('Q', 10, struct wred_info)
#ifdef WRED_STAT
#define WRED_GETSTAT    _IOWR('Q', 11, struct wred_stat)
#endif /* WRED_STAT */
#endif /* WTP_WRED */
#ifdef EXT_STAT
#define GET_EXT_STATS    _IOWR('Q', 12, struct ext_stat)
#endif /* EXT_STAT */

#ifndef IFNAMSIZ
#define IFNAMSIZ    16
#endif

/* I must store some information (like the arrival time) with each packet */
struct mb
{
#ifdef COMP_TIME
    unsigned long arr_time;
#else
    struct timeval arr_time;
#endif /* COMP_TIME */
    struct mb* prev;        /* pointer to previous mb structure */
#ifdef EXT_STAT
    char dscp;
#endif /* EXT_STAT */
};

typedef struct wtp_queue
{
    struct mb *head_at, *tail_at;    /* arrival time */
    struct mbuf* head, *tail;        /* packet in this queue */
    int size;                        /* Bytes in this queue (needed for the dropper) */
    int nbpacket;                    /* Number of packet in this queue */
    int weight;                      /* Weight of this queue (integer) */
    short int dscp;                  /* ID/DSCP of this queue (needed for the classifier) */
#ifdef COMP_TIME
    struct timeval ref_time;         /* Reference time [COMP_TIME] */
    struct timeval last_time;        /* time of last packet */
#endif /* COMP_TIME */

    /* Statistic variables */
#ifdef Q_DELAY
    int mbuf_bytes;
    struct timeval last_del_pack;    /* Queueing Delay */
#endif /* Q_DELAY */
    u_int sent_packets;              /* packets sent in this queue */
    u_int drop_packets;              /* dropped packets */
    u_quad_t sent_bytes;             /* bytes sent in this queue */
    u_quad_t drop_bytes;             /* bytes dropped in this queue */
#ifdef EXT_STAT
```

```
    u_quad_t *dscp_sent_bytes;       /* For statistics only, bytes sent (with a given DSCP) */
    u_quad_t *dscp_drop_bytes;       /* For statistics only, bytes dropped (with a given DSCP) */
#endif /* EXT_STAT */
} wtp;

typedef struct wtpstate
{
    struct wtpstate *next;           /* for wtpstate list */
    struct ifnet *ifp;               /* interface */
    int nums;                        /* number of queues */
    int bytes;                       /* total bytes in all the queues */
    int qlimit;                      /* maximum bytes per queue */
    wtp *queue;                      /* pointer to the queue list */
#ifdef WTP_WRED
    struct wred *wred;               /* WRED informations */
#endif /* WTP_WRED */
} wtp_state_t;

/* #endif KERNEL */


typedef struct wtp_interface
{
    char wtp_ifacename[IFNAMSIZ];
    u_int wtp_ifacelen;
} wtp_iface_t;

struct wtp_setweight
{
    wtp_iface_t iface;
    int dscp; /* ID of the queue */
    int weight;
};

struct wtp_conf
{
    wtp_iface_t    iface;
    int nqueues;        /* number of queues */
    int qlimit;            /* queue size in bytes */
};

typedef struct one_queue_stats
{
    int bytes;        /* bytes currently in this queue */
    int nbpacket;    /* nuber of packet currently in the queue */
    int weight;        /* queue weight */
#ifdef Q_DELAY
/*     struct timeval tot_delay; */
    struct timeval last_del_pack;
#endif /* Q_DELAY */
    u_int sent_packets;    /* number of packets sent in this queue */
    u_int drop_packets; /* number of packets drop in this queue */
    u_quad_t sent_bytes; /* bytes sent in this queue */
    u_quad_t drop_bytes; /* bytes dropped in this queue */
    int mbuf_bytes;
```

```
} queue_stats;


struct wtp_getstats
{
    wtp_iface_t iface;
    int dscp;
    queue_stats stats;
};


#ifdef EXT_STAT
struct ext_stat
{
    wtp_iface_t iface;
    int dscp;
    u_quad_t dscp_sent_bytes[WRED_MAXDROP];
    u_quad_t dscp_drop_bytes[WRED_MAXDROP];
};
#endif /* EXT_STAT */

#endif /* _NETINET_ALTQ_WTP_H*/
```

A.2 altq_wtp_opt.h

This file contains the options that can be selected for the compilation.

```
#ifndef _WTP_COMPIL_OPTIONS
#define _WTP_COMPIL_OPTIONS

/* OPTION DOR WTP/WRED implementation */
/* by Louis SWINNEN */

/* ************************************** COMPILE OPTIONS ***************************************** */
/* TIME COMPRESSING (COMP_TIME)
 * ----------------
 *
 * With this option, this implementation will use a compressed version of time using the following description :
 *
 * 32 bits are used (instead of 2*32 bits)
 *
 * 20 bits for coding ţsec
 * 12 bits for coding seconds
 */

#define COMP_TIME
/* Q_DELAY
 * -------
 * This option allow to collect Queueing delay from external program (for statistic)
 */
#define Q_DELAY

/* WTP_WRED
```

```
 * --------
 * This option activate the WRED dropper over the WTP scheduler. Default dropper mechanism : tail-drop
 */
#define WTP_WRED

/* WRED_STAT
 * ----------
 * This option activate all statistics inside WRED dropper
 */
#define WRED_STAT

/* EXT_STAT (experimental - should not be used for stable version)
 * --------
 * This function enable extended statistics
 */
/* #define EXT_STAT */
/* ************************************************************************************************** */

#endif /* _WTP_COMPIL_OPTIONS */
```

A.3 altq_wtp_util.h

```
/* Definition of util functions
 *
 * ALTQ/WTP & WRED implementation
 *
 * Louis SWINNEN : FUNDP & CRC Alcatel ANTWERPEN
 *
 * version 0.6 BETA
 */

#ifndef _ALTQ_WTP_UTIL_H
#define _ALTQ_WTP_UTIL_H
struct timeval elapsedFromNow(struct timeval, struct timeval*);
long convertusec(struct timeval);

struct timeval
elapsedFromNow(tv, gNow)                /* compute the time elapsed between 'tv' and now */
    struct timeval tv;
    struct timeval* gNow;
{
    struct timeval * now, nowTime, res;

    if(!gNow)
        now = &nowTime;
    else
        now = gNow;

    microtime(now);

    res.tv_sec = now->tv_sec - tv.tv_sec;
    res.tv_usec = now->tv_usec - tv.tv_usec;
    /*****
     * Caution !!!    "now" is 'greater' than "tv". So, i have :
     * (now.tv_sec > tv.tv_sec) || (now.tv_sec = tv.tv_sec && now.tv_usec > tv.tv_usec)
```

```
     *
     *****/
    if(res.tv_sec > 0)
    {
        if(res.tv_usec < 0)
        {
            res.tv_sec--;
            res.tv_usec = 1000000 + res.tv_usec;
        }
    } /* if res.tv_sec = 0, we can return res because res.tv_usec is greater than zero  (res.tv_sec < 0 isn't possible) */

    return res;
}

long
convertusec(tv)
    struct timeval tv;
{
    return(tv.tv_sec*1000000 + tv.tv_usec);
}

#endif /* _ALTQ_WTP_UTIL_H */
```

A.4 altq_localq.c

```
/*****
 * ALTQ LOCALQ/WTP & WRED implementation
 * by Louis SWINNEN FUNDP & Alcatel CRC ANTWERPEN
 * Version 0.5
 *****/

#ifndef _NO_OPT_ALTQ_H_
#include "opt_altq.h"
#if !defined(__FreeBSD__) || (__FreeBSD__ > 2)
#include "opt_inet.h"
#endif
#endif /* !_NO_OPT_ALTQ_H_ */
#ifdef LOCALQ    /* localq is enabled by LOCALQ option in opt_altq.h */



#include <sys/param.h>
#include <sys/malloc.h>
#include <sys/mbuf.h>
#include <sys/uio.h>
#include <sys/socket.h>
#include <sys/systm.h>
#include <sys/proc.h>
#include <sys/errno.h>
#include <sys/time.h>
#include <sys/kernel.h>

#include <net/if.h>
#include <net/if_types.h>
```

```
#include <net/altq_conf.h>
#include <netinet/in.h>
#include <netinet/in_systm.h>
#include <netinet/ip.h>

#include <netinet/altq_wtp_opt.h>
#include <netinet/altq.h>
#include <netinet/altq_wtp_util.h>

#ifdef WTP_WRED
#include <netinet/altq_wtp_wred2.c>
#endif /* WTP_RED */

#include <netinet/altq_wtp.h>

/* Debug options */
#define WTP_DEBUG
/* #define WTP_DEBUG_TRACING
/* #define WTP_DBGLGHT */



/*
 * localq device interface
 */

altqdev_decl(localq);

static wtp_state_t *wtp_list = NULL;

/* Function prototype */
static int wtp_flush(struct ifnet*);
static int wtp_setenable(struct wtp_interface*, int);
static int wtp_ifattach(struct wtp_interface*);
static struct mbuf* wtp_ifdequeue(struct ifnet*, int);
static int wtp_config(struct wtp_conf*);
static wtp* classifypacket(int, wtp_state_t*);
static int wtp_ifenqueue(struct ifnet*, struct mbuf*, struct pr_hdr*, int);
static int wtp_ifdetach(struct wtp_interface*);
static int wtp_getstats(struct wtp_getstats*);
static int wtp_setweight(struct wtp_setweight*);
static int stat_reset(struct wtp_interface*);
#ifdef EXT_STAT
static int get_extstat(struct ext_stat *);
#endif /* EXT_STAT */
#ifdef COMP_TIME
static struct timeval refTime(wtp*);

static struct timeval
refTime(wtp* queue)
{
    struct timeval tmp;
    unsigned long deltaTime = queue->head_at->arr_time;
    tmp.tv_usec = deltaTime & 0x000FFFF;
    tmp.tv_sec = deltaTime >> 20;
    queue->ref_time.tv_sec += tmp.tv_sec;
```

```
    queue->ref_time.tv_usec += tmp.tv_usec;
    if(queue->ref_time.tv_usec >= 1000000)
    {
        queue->ref_time.tv_usec -= 1000000;
        queue->ref_time.tv_sec +=1;
    }


    return(queue->ref_time);
}

#endif /* COMP_TIME */



int
localqopen(dev, flag, fmt, p)
    dev_t dev;
    int flag, fmt;
    struct proc *p;
{
    /* everything will be done when the queueing scheme is attached. */
#ifdef COMP_TIME
    printf("Compressed Time enabled\n");
#else
    printf("Compressed Time disabled\n");
#endif /* COMP_TIME */
#ifdef WTP_WRED
    printf("Weighted RED enabled/Tail Drop disabled\n");
#else
    printf("Tail Drop enabled/Weighted RED disabled\n");
#endif /* WTP_WRED */

    return 0;
}

static int
wtp_flush(ifp)                     /* flush all WTP queues attached on this interface */
    struct ifnet *ifp;
{
    struct mbuf *mp;

    while(mp = wtp_ifdequeue(ifp, ALTDQ_DEQUEUE)) /* != NULL*/
        m_freem(mp);

    return 0;
}


static int
wtp_setenable(ifacep, flag)        /*    Enable / Disable ALTQ/WTP on this interface */
    struct wtp_interface* ifacep;
    int flag;
{
    wtp_state_t *wtps;
    int error = 0;
```

```
    if(!(wtps = altq_lookup(ifacep->wtp_ifacename, ALTQT_LOCALQ)))
#ifdef WTP_DEBUG
    {
        printf("Interface not found (RETURN) \n");
#endif /* WTP_DEBUG */
        return(EBADF);

#ifdef WTP_DEBUG
    }
#endif /* WTP_DEBUG */

    switch(flag)
    {
        case ENABLE:
            error = if_altqenable(wtps->ifp);
            break;
        case DISABLE:
            error = if_altqdisable(wtps->ifp);
            break;
    }

    return error;

}

static int
wtp_ifattach(ifacep)            /* Initialize WTP on this interface */
    struct wtp_interface *ifacep;
{
    int error = 0, i,j;
    struct ifnet *ifp;
    struct wtpstate *new_wtps;
    wtp *queue;

    if((ifp = ifunit(ifacep->wtp_ifacename)) == NULL)
    {
#ifdef WTP_DEBUG
        printf("wtp_ifattach() ... no ifp found\n");
#endif /* WTP_DEBUG */
        return(ENXIO);
    }

    if(!ALTQ_IS_READY(ifp))
    {
#ifdef WTP_DEBUG
        printf("wtp_ifattach() ... altq is not ready\n");
#endif /* WTP_DEBUG */
        return(ENXIO);
    }

    /* allocate and initialize wtp_state_t */
    MALLOC(new_wtps, wtp_state_t *, sizeof(wtp_state_t), M_DEVBUF, M_WAITOK);
    if(!new_wtps)
#ifdef WTP_DEBUG
    {
```

```
        printf("wtp_ifattach() ... not enough memory\n");
#endif /* WTP_DEBUG */
        return(ENOMEM); /* new_wtps == NULL */
#ifdef WTP_DEBUG
    }
#endif /* WTP_DEBUG */

    bzero(new_wtps, sizeof(wtp_state_t));
    MALLOC(queue, wtp*, sizeof(wtp) * DEFAULT_QNUMS, M_DEVBUF, M_WAITOK);

    if(!queue)
    {    /* queue == NULL */
#ifdef WTP_DEBUG
        printf("wtp_ifattach() ... not enough memory (2)\n");
#endif /* WTP_DEBUG */

        FREE(new_wtps, M_DEVBUF);
        return(ENOMEM);
    }
    bzero(queue, sizeof(wtp) * DEFAULT_QNUMS);
    queue[0].dscp = DSCP(0); /* default class */
    queue[1].dscp = DSCP(1); /* 0x08; class 1 : (00)001000 */
    queue[2].dscp = DSCP(2); /* 0x10; class 2 : (00)010000 */
    queue[3].dscp = DSCP(3); /* 0x18; class 3 : (00)011000 */

    /* keep the ifp */
    new_wtps->ifp = ifp;
    new_wtps->nums = DEFAULT_QNUMS; /* 3 */
    new_wtps->bytes = 0;
    new_wtps->queue = queue;
    new_wtps->qlimit = DEFAULT_QLIMIT;

#ifdef EXT_STAT
    for(i=0;i<new_wtps->nums;++i)
    {
        MALLOC(queue[i].dscp_sent_bytes, u_quad_t*, sizeof(u_quad_t) * WRED_MAXDROP, M_DEVBUF, M_WAITOK);
        MALLOC(queue[i].dscp_drop_bytes, u_quad_t*, sizeof(u_quad_t) * WRED_MAXDROP, M_DEVBUF, M_WAITOK);
    }
#endif /* EXT_STAT */


#ifdef WTP_WRED
    if(!(new_wtps->wred = allocWRED_default(new_wtps)))
    {
        printf("if_altqattach() ... WRED initialisation error\n");
        FREE(new_wtps->queue, M_DEVBUF);
        FREE(new_wtps, M_DEVBUF);
        return(ENOMEM);
    }
#endif /* WTP_WRED */

#ifdef WTP_DEBUG
    printf("%d queues are created with maximum %d bytes inside.\n", new_wtps->nums, new_wtps->qlimit);
    for(i=0;i<new_wtps->nums;++i)
    {
        printf("Queue[%d] with DSCP=%d/%x\n",i,queue[i].dscp, queue[i].dscp);
```

```
    }
#endif /* WTP_DEBUG*/

    for(i=0;i<new_wtps->nums;++i, ++queue)
    {
        /* queue->next = queue->prev = NULL; */
        queue->head = queue->tail = NULL;
        queue->head_at = queue->tail_at = NULL;  /* We will allocate a struct mb when we will receive a packet */
        queue->size = 0;
        queue->nbpacket = 0;
        queue->weight = 1;

#ifdef EXT_STAT
        for(j=0;j<WRED_MAXDROP;++j)
        {
            queue->dscp_sent_bytes[j] = (i==0?1:0);
            queue->dscp_drop_bytes[j] = 0;
        }

  #ifdef WTP_DEBUG
        for(j=0;j<WRED_MAXDROP;++j)
        {
            printf("Valeur de queue->dscp_sent_bytes[%d] = %u\n", j, queue->dscp_sent_bytes[j]);
        }

  #endif /* WTP_DEBUG */
#endif /* EXT_STAT */
    }

    /*
     * set WTP to this ifnet structure
     */

    if(error = if_altqattach(ifp, new_wtps, wtp_ifenqueue, wtp_ifdequeue, ALTQT_LOCALQ))
    {    /* error != 0*/
        FREE(new_wtps->queue, M_DEVBUF);
        FREE(new_wtps, M_DEVBUF);
#ifdef WTP_DEBUG
        printf("wtp_ifattach() ... error while if_altqattach(error = %d)\n", error);
#endif /* WTP_DEBUG */

        return(error);
    }

    new_wtps->next = wtp_list;
    wtp_list= new_wtps;
    return(error);
}

static struct mbuf*                 /* provide the next packet choosen by the WTP scheduler */
wtp_ifdequeue(ifp, mode)
    struct ifnet *ifp;
    int mode; /* DEQUEUE / FLUSH */
{
    wtp_state_t *wtps;
    wtp *queueMAXPRIO=NULL;
```

```
    int byte, i;
    unsigned long maxprio = 0;
    struct mbuf *mbuf;
#ifdef Q_DELAY
        struct timeval SAVEDdelay;
#endif /* Q_DELAY */

    wtps = (wtp_state_t *)ifp->if_altqp;

    if(mode == ALTDQ_FLUSH)
    {
        wtp_flush(ifp);
        return NULL;
    }
    if(mode != ALTDQ_DEQUEUE)
            printf("!! No Dequeue !!\n");

    if(wtps->bytes == 0)
        return NULL; /* no packet in the queues or no active queues */

    for(i=0; i<wtps->nums;++i)
    {
        long prio = -1;
#ifdef Q_DELAY
        struct timeval delay;
#endif /* Q_DELAY */
        if(wtps->queue[i].tail) /* if this queue isn't empty */
        {
#ifdef WTP_DEBUG
            if(!wtps->queue[i].head || !wtps->queue[i].head_at || !wtps->queue[i].tail_at)
                printf("BUG LIST\n");
#endif /* WTP_DEBUG */
#ifdef COMP_TIME
  #ifdef Q_DELAY
              delay = elapsedFromNow(refTime(&wtps->queue[i]), NULL);
            wtps->queue[i].last_del_pack = delay;
            prio = convertusec(delay) * (long) wtps->queue[i].weight;
  #else /* Q_DELAY */
              prio = convertusec(elapsedFromNow(refTime(&wtps->queue[i]), NULL)) * (long) wtps->queue[i].weight;
  #endif /* Q_DELAY */
            wtps->queue[i].head_at->arr_time = 0;
#else /* COMP_TIME */
  #ifdef Q_DELAY
              delay = elapsedFromNow(wtps->queue[i].head_at->arr_time, NULL);
            wtps->queue[i].last_del_pack = delay;
            prio = convertusec(delay) * (long)(wtps->queue[i].weight);
  #else /* Q_DELAY */
            prio = convertusec(elapsedFromNow(wtps->queue[i].head_at->arr_time, NULL)) * (long)(wtps->queue[i].weight);
  #endif /* Q_DELAY */
#endif /* COMP_TIME */

#ifdef WTP_DEBUG
            if(prio < 0)
                printf("*** BUG PRIO ***\n");
#endif /* WTP_DEBUG */
```

```
            if(prio >= maxprio)
            {
                    maxprio = prio;
                    queueMAXPRIO = &wtps->queue[i];
#ifdef Q_DELAY
                    SAVEDdelay = delay;
#endif /* Q_DELAY */
            }

        }
    }
#ifdef WTP_DEBUG
    if(!queueMAXPRIO)
        printf("*** BUG queueMAXPRIO NULL\n");
#endif /* WTP_DEBUG */

    mbuf = queueMAXPRIO->head; /* queueMAXPRIO is the selected queue by the scheduler */

    if(mode == ALTDQ_DEQUEUE) /* dequeue the selected packet */
    {
        struct mb* svg;
#ifdef EXT_STAT
        int dscp = (int)queueMAXPRIO->head_at->dscp;
#endif /* EXT_STAT */
        int byte = mbuf->m_pkthdr.len;
        if(!(queueMAXPRIO->head = mbuf->m_nextpkt))
            queueMAXPRIO->tail = NULL; /* mbuf->m_nextpkt == NULL */
        mbuf->m_nextpkt = NULL;
        wtps->bytes -= byte;
        queueMAXPRIO->size -= byte;
        queueMAXPRIO->sent_packets++;
        queueMAXPRIO->sent_bytes += byte;
#ifdef EXT_STAT
/*        printf("packet dscp = %x\n", dscp); */
        queueMAXPRIO->dscp_sent_bytes[DROPPREC(dscp & 0x0007)]+= byte;
#endif /* EXT_STAT */

        if(!(svg = queueMAXPRIO->head_at->prev))
            queueMAXPRIO->tail_at = NULL; /* queueMAXPRIO->head_at->prev == NULL */

#ifdef WTP_DEBUG
        if(queueMAXPRIO->head !=NULL && svg == NULL || queueMAXPRIO->head == NULL && svg != NULL)
            printf("BUG *ifdequeue()* svg and queueMAXPRIO->head doesn't have same size\n");
#endif /* WTP_DEBUG */
        queueMAXPRIO->head_at->prev = NULL;
        FREE(queueMAXPRIO->head_at, M_DEVBUF);  /* Free the mb structure */
        queueMAXPRIO->head_at = svg;
#ifdef Q_DELAY
/*         queueMAXPRIO->q_delay.tv_sec += SAVEDdelay.tv_sec;
        queueMAXPRIO->q_delay.tv_usec += SAVEDdelay.tv_usec;*/
/*        while(queueMAXPRIO->q_delay.tv_usec >= 1000000)
        {
            queueMAXPRIO->q_delay.tv_usec -= 1000000;
            queueMAXPRIO->q_delay.tv_sec++;
        } */
        queueMAXPRIO->mbuf_bytes = byte;
```

```
#endif /* Q_DELAY */
#ifdef WTP_WRED
        wtps->wred->nbpacket--;
        queueMAXPRIO->nbpacket--;
        if(!wtps->wred->nbpacket)
        {
            wtps->wred->idle = 0;
            microtime(&wtps->wred->last_time);
        }
#endif /* WTP_WRED */
    }
    return(mbuf);
}

static int
wtp_config(cf)                    /* configure WTP scheduler in terms of queues and queue limit */
    struct wtp_conf *cf;
{
    wtp_state_t    *wtps;
    wtp    *queue;
    int i,j;

    if((wtps=altq_lookup(cf->iface.wtp_ifacename, ALTQT_LOCALQ)) == NULL)
#ifdef WTP_DEBUG
    {
        printf("wtp_config() ... bad interface\n");
#endif /* WTP_DEBUG */
        return(EBADF);
#ifdef WTP_DEBUG
    }
#endif /* WTP_DEBUG */


    if(cf->nqueues < MIN_QNUMS || MAX_QNUMS < cf->nqueues)
        cf->nqueues = DEFAULT_QNUMS; /* CAUTION ! cf is modified if 'nqueues' is bad */

    if(cf->qlimit < MIN_QLIMIT || cf->qlimit > MAX_QLIMIT)
        cf->qlimit = DEFAULT_QLIMIT; /* CAUTION ! cf is modified if 'qlimit' is bad */

    if((cf->nqueues != wtps->nums) || (cf->qlimit != wtps->qlimit))
    {
        /* free queued mbuf */
        wtp_flush(wtps->ifp);
#ifdef EXT_STAT
        for(i=0;i<wtps->nums;++i)
        {
            FREE(wtps->queue[i].dscp_sent_bytes, M_DEVBUF);
            FREE(wtps->queue[i].dscp_drop_bytes, M_DEVBUF);
        }
#endif /* EXT_STAT */

        FREE(wtps->queue, M_DEVBUF);

        MALLOC(queue, wtp *, sizeof(wtp) * cf->nqueues, M_DEVBUF, M_WAITOK);

        if(queue == NULL)
```

```
#ifdef WTP_DEBUG
        {
            printf("wtp_config() ... not enough memory\n");
#endif /* WTP_DEBUG*/
            return(ENOMEM);
#ifdef WTP_DEBUG
        }
#endif /* WTP_DEBUG*/
        bzero(queue, sizeof(wtp) * cf->nqueues);
        wtps->qlimit=cf->qlimit;
        wtps->nums = cf->nqueues;
        wtps->bytes = 0;
        wtps->queue = queue;
#ifdef WTP_DEBUG
        printf("%d queues are created with maximum %d bytes inside\n", wtps->nums, wtps->qlimit);
#endif /* WTP_DEBUG */
#ifdef EXT_STAT
        for(i=0;i<wtps->nums;++i)
        {
            MALLOC(queue[i].dscp_sent_bytes, u_quad_t*, sizeof(u_quad_t) * WRED_MAXDROP, M_DEVBUF, M_WAITOK);
            MALLOC(queue[i].dscp_drop_bytes, u_quad_t*, sizeof(u_quad_t) * WRED_MAXDROP, M_DEVBUF, M_WAITOK);
            if(queue[i].dscp_sent_bytes == NULL)
            {
                for(j=0; j < i;++j)
                {
                    FREE(queue[j].dscp_sent_bytes, M_DEVBUF);
                    FREE(queue[j].dscp_drop_bytes, M_DEVBUF);
                }
                FREE(queue, M_DEVBUF);
                return(ENOMEM);
            }else
                if(queue[i].dscp_drop_bytes == NULL)
                {
                    for(j=0; j < i;++j)
                    {
                        FREE(queue[j].dscp_sent_bytes, M_DEVBUF);
                        FREE(queue[j].dscp_drop_bytes, M_DEVBUF);
                    }
                    FREE(queue[i].dscp_sent_bytes, M_DEVBUF);
                    FREE(queue, M_DEVBUF);
                    return(ENOMEM);
                }
        }
#endif /* EXT_STAT */
        for(i=0;i<wtps->nums;++i, ++queue)
        {
            queue->head = queue->tail = NULL;
            queue->head_at = queue->tail_at = NULL;
            queue->size = 0;
            queue->nbpacket = 0;
            queue->weight = 1;
            queue->dscp = DSCP(i);
#ifdef EXT_STAT
            for(j=0;j<WRED_MAXDROP;++j)
            {
                queue->dscp_sent_bytes[j] = (i==0?1:0);
```

```
                queue->dscp_drop_bytes[j] = 0;
            }
#endif /* EXT_STAT */
#ifdef WTP_DEBUG
            printf("Queue[%d] DSCP =%d/%x, weight = 1\n", i, queue->dscp, queue->dscp);
#endif /* WTP_DEBUG */
        }

    }

    if(cf->qlimit > 0)
        wtps->qlimit = cf->qlimit;
    return 0;
}


static wtp*
classifypacket(dscp, wtps)        /* Determine the right queue for the given flow */
    int dscp;
    wtp_state_t *wtps;
{
    int i;
    wtp* default_queue, *queue;

    dscp &= 0xF8;

    default_queue = queue = wtps->queue;
    for(i = 0; i<wtps->nums; ++i, ++queue)
    {
        if(queue->dscp == dscp)
            break;
    }

    return(i==wtps->nums?default_queue:queue);
/*  i == wtps->nums --> DSCP is invalid !
 *  i <  wtps->nums --> a queue with the given DSCP was found
 */
}


static int
wtp_ifenqueue(ifp, mp, pr_hdr, mode)    /* Try to enqueue the given packet */
    struct ifnet *ifp;
    struct mbuf *mp;
    struct pr_hdr *pr_hdr;
    int mode;
{
    wtp_state_t *wtps;
    struct flowinfo flow;
    struct mb *mb;
    int dscp;
    wtp *queue;
    int byte, error = 0;

    if(mode != ALTEQ_NORMAL)
        return 0;
```

```
    wtps = (wtp_state_t *) ifp->if_altqp;
    mp->m_nextpkt = NULL;

    altq_extractflow(mp, pr_hdr, &flow, 0);
    dscp = (((struct flowinfo_in*) &flow)->fi_tos) >> 2;

    queue = classifypacket(dscp, wtps);



#ifdef WTP_WRED
    if((queue->size < wtps->qlimit) && acceptPacket(DROPPREC(dscp & 0x0007), wtps->wred) )
#else
    if(queue->size < wtps->qlimit)
#endif
    {
        MALLOC(mb, struct mb*, sizeof(mb), M_DEVBUF, M_WAITOK);     /* Allocate the mb structure to store the packet arrival time */
        if(!mb)
        {
            printf("Error during malloc for mb structure\n");
            return(ENOMEM);
        }
#ifdef EXT_STAT
        mb->dscp = dscp;
#endif /* EXT_STAT */
        if(!queue->tail)
        {
#ifdef WTP_DEBUG
            if(queue->head)
                printf("BUG *enqueue* tail = NULL && head <>  NULL\n");
            if(queue->tail_at)
                printf("BUG *enqueue* synchro mb <-> mbuf\n");
#endif /* WTP_DEBUG */
            /* queue->tail == NULL --> this queue was empty */
            queue->head_at = mb;
            mb->prev = NULL;
#ifdef COMP_TIME
            microtime(&queue->ref_time);
            queue->last_time = queue->ref_time;
            mb->arr_time = 0;
#else
            microtime(&mb->arr_time);
#endif /* COMP_TIME */
            queue->head = mp;
        }else
        {
#ifdef COMP_TIME
            struct timeval tmp = queue->last_time;
#endif /* COMP_TIME */
#ifdef WTP_DEBUG
            if(!queue->head)
                printf("BUG *enqueue* head = NULL && tail <> NULL !\n");
            if(!queue->tail_at)
                printf("BUG *enqueue* synchro mb <-> mbuf (2)\n");
#endif /* WTP_DEBUG */
```

```
            /* queue->tail != NULL --> this queue isn't empty -> add the packet at the tail of the queue */
            mb->prev = NULL;
            queue->tail_at->prev = mb;
#ifdef COMP_TIME
            tmp = elapsedFromNow(tmp, &queue->last_time);
            mb->arr_time = (tmp.tv_sec << 20) | (tmp.tv_usec & 0x000FFFFF);
#else
            microtime(&mb->arr_time);
#endif /* COMP_TIME */
            queue->tail->m_nextpkt = mp;
        }
        queue->tail_at = mb;
        queue->tail=mp;
        byte = mp->m_pkthdr.len;
        queue->size += byte;
        wtps->bytes += byte;
        queue->nbpacket++;
#ifdef WTP_WRED
        wtps->wred->nbpacket++;
#endif
    /*
     * call the driver's start routine.
     */

        ifp = wtps->ifp;
        if(ifp->if_start && (ifp->if_flags & IFF_OACTIVE) == 0 )
            (*ifp->if_start)(ifp);
    }else
    {
        queue->drop_packets +=1;
        queue->drop_bytes += mp->m_pkthdr.len;
#ifdef EXT_STAT
        queue->dscp_drop_bytes[DROPPREC(dscp & 0x0007)] += mp->m_pkthdr.len;
#endif /* EXT_STAT */
        m_freem(mp);
        error = ENOBUFS;
    }
    return error;
}

static int
wtp_ifdetach(ifacep)                 /* Remove WTP from this interface */
    struct wtp_interface *ifacep;
{
    int error =0;
#ifdef EXT_STAT
    int i;
#endif /* EXT_STAT */
    wtp_state_t *wtps;


    if(!(wtps = altq_lookup(ifacep->wtp_ifacename, ALTQT_LOCALQ)))
#ifdef WTP_DEBUG
    {
        printf("wtp_ifdetach() ... bad interface \n");
#endif /* WTP_DEBUG */
```

```
        return(EBADF);     /* Bad interface */
#ifdef WTP_DEBUG
    }
#endif /* WTP_DEBUG */

    /* free queued mb */
    wtp_flush(wtps->ifp);

    /* Remove WTP/LOCALQ from the ifnet structure. */
    (void)if_altqdisable(wtps->ifp);
    (void)if_altqdetach(wtps->ifp);

    /* Remove from the wtpstate list */
    if(wtp_list == wtps)
        wtp_list = wtps->next;
    else
    {
        wtp_state_t *wt = wtp_list;
        do
        {
            if(wt->next == wtps)
            {
                wt->next = wtps->next;
                break;
            }
        } while(wt = wt->next); /* wp != NULL */
    }
    /* deallocate wtp_state_t */
#ifdef WTP_WRED
    flushWRED(wtps->wred);
#endif
#ifdef EXT_STAT
    for(i=0; i< wtps->nums; ++i)
    {
        FREE(wtps->queue[i].dscp_sent_bytes, M_DEVBUF);
        FREE(wtps->queue[i].dscp_drop_bytes, M_DEVBUF);
    }
#endif /* EXT_STAT */

    FREE(wtps->queue, M_DEVBUF);
    FREE(wtps, M_DEVBUF);
    return error;
}

static int
wtp_getstats(gs)                    /* Provide some statistics for the WTP scheduler */
    struct wtp_getstats *gs;
{
    wtp_state_t *wtps;
    wtp *queue;
    queue_stats *stats;
    int i;

    if((wtps = altq_lookup(gs->iface.wtp_ifacename, ALTQT_LOCALQ)) == NULL)
#ifdef WTP_DEBUG
    {
```

```
        printf("wtp_getstats() ... bad interface \n");
#endif /* WTP_DEBUG */
        return(EBADF); /* Bad interface */
#ifdef WTP_DEBUG
    }
#endif /* WTP_DEBUG */

    for(i=0, queue=wtps->queue;i<wtps->nums;++i, ++queue) /* locate the queue with the given DSCP */
    {
        if(gs->dscp == queue->dscp)
            break;
    }
    if(i == wtps->nums)
#ifdef WTP_DEBUG
    {
        printf("wtp_getstats() ... invalid DSCP \n");
#endif /* WTP_DEBUG */
        return(EINVAL); /* invalid DSCP */
#ifdef WTP_DEBUG
    }
#endif /* WTP_DEBUG */

    stats = &gs->stats;

    /* copy queue statistics to gsp structure */
    stats->bytes = queue->size;
    stats->nbpacket = queue->nbpacket;
    stats->weight = queue->weight;
    stats->sent_packets = queue->sent_packets;
    stats->sent_bytes = queue->sent_bytes;
    stats->drop_packets = queue->drop_packets;
    stats->drop_bytes = queue->drop_bytes;
#ifdef Q_DELAY
    stats->mbuf_bytes = queue->mbuf_bytes;
    stats->last_del_pack = queue->last_del_pack;
#endif /* Q_DELAY */
    return 0;

}

#ifdef EXT_STAT
static int
get_extstat(es)
    struct ext_stat *es;
{
    wtp_state_t *wtps;
    int i, dscp;
    wtp *queue;

    if((wtps = altq_lookup(es->iface.wtp_ifacename, ALTQT_LOCALQ)) == NULL)
#ifdef WTP_DEBUG
    {
        printf("get_extstat() ... bad interface \n");
#endif /* WTP_DEBUG */
        return(EBADF); /* Bad interface */
#ifdef WTP_DEBUG
```

```
    }

#endif /* WTP_DEBUG */
    dscp = es->dscp & 0x00F8;
    /* printf("get_extstat() ... DSCP2int = %d, queue->dscp_sent_byte[0] = %qu\n", DSCP2int(dscp), wtps->queue[DSCP2int(dscp)].dscp_sent_bytes); */

    if(i = DSCP2int(dscp) < 0)
#ifdef WTP_DEBUG
    {
        printf("get_extstat() ... invalid DSCP \n");
#endif /* WTP_DEBUG */
        return(EINVAL); /* invalid DSCP */
#ifdef WTP_DEBUG
    }
#endif /* WTP_DEBUG */
    queue = &(wtps->queue[i]);

    for(i=0;i<WRED_MAXDROP;++i)
    {
        es->dscp_sent_bytes[i] = queue->dscp_sent_bytes[i];
        es->dscp_drop_bytes[i] = queue->dscp_drop_bytes[i];
    }
    return 0;
}
#endif /* EXT_STAT */

static int
wtp_setweight(sw)                /* set weight of a queue */
    struct wtp_setweight *sw;
{
    wtp_state_t    *wtps;
    wtp *queue;
    int i;

    if(sw->weight < 0)
#ifdef WTP_DEBUG
    {
        printf("wtp_setweight ... set weight in natural number\n");
#endif /* WTP_DEBUG */
        return(EINVAL); /* wrong number format */
#ifdef WTP_DEBUG
    }
#endif /* WTP_DEBUG */


    if((wtps = altq_lookup(sw->iface.wtp_ifacename, ALTQT_LOCALQ)) == NULL)
#ifdef WTP_DEBUG
    {
        printf("wtp_setweight() ... bad interface \n");
#endif /* WTP_DEBUG */
        return(EBADF); /* Bad interface */
#ifdef WTP_DEBUG
    }
#endif /* WTP_DEBUG */

    queue = wtps->queue;
```

```
    for(i=0;i<wtps->nums;++i, ++queue)
    {
        if(queue->dscp == sw->dscp)
            break;
    }

    if(i == wtps->nums)
#ifdef WTP_DEBUG
    {
        printf("wtp_setweight() ... invalid DSCP \n");
#endif /* WTP_DEBUG */
        return(EINVAL); /* Invalid DSCP */
#ifdef WTP_DEBUG
    }
#endif /* WTP_DEBUG */
    queue->weight = sw->weight;

    return 0;

}


int
localqclose(dev, flag, fmt, p)            /* close LOCALQ/WTP on the device 'dev' */
    dev_t dev;
    int flag, fmt;
    struct proc *p;
{
    struct ifnet *ifp;
    struct wtp_interface iface;
    wtp_state_t *wtps;
    int s;


    s=splimp();

    while(wtps = wtp_list) /* wtps != NULL */
    {
        ifp = wtps->ifp;
#ifdef __NetBSD__
        sprintf(iface.wtp_ifacename, "%s", ifp->if_xname);
#else
        sprintf(iface.wtp_ifacename, "%s%d", ifp->if_name, ifp->if_unit);
#endif
        iface.wtp_ifacelen = strlen(iface.wtp_ifacename);
        wtp_ifdetach(&iface);

    }
    splx(s);
    return 0;
}

static int
stat_reset(ifacep)                /* Clear all statistics variables */
    struct wtp_interface *ifacep;
```

```
{
    wtp_state_t *wtps;
    int i,j;

    if(!(wtps = altq_lookup(ifacep->wtp_ifacename, ALTQT_LOCALQ)))
        return(EBADF); /* Interface not found */

    for(i=0;i<wtps->nums;++i)
    {
        wtps->queue[i].sent_packets = 0;
        wtps->queue[i].drop_packets = 0;
        wtps->queue[i].sent_bytes = 0;
        wtps->queue[i].drop_bytes = 0;
#ifdef Q_DELAY
        wtps->queue[i].last_del_pack.tv_sec = 0;
        wtps->queue[i].last_del_pack.tv_usec = 0;
/*        wtps->queue[i].q_delay.tv_sec = 0;
        wtps->queue[i].q_delay.tv_usec = 0; */
#endif /* Q_DELAY */
#ifdef EXT_STAT
        for(j=0; j<WRED_MAXDROP;++j)
        {
            wtps->queue[i].dscp_sent_bytes[j] = 0;
            wtps->queue[i].dscp_drop_bytes[j] = 0;
        }
#endif /* EXT_STAT */
    }
#ifdef WRED_STAT
    for(i=0;i<WRED_MAXDROP;++i)
    {
        wtps->wred->sent_packets[i]=0;
        wtps->wred->drop_packets[i]=0;
    }
#endif /* WRED_STAT */

    return(0);
}


int
localqioctl(dev, cmd, addr, flag, p) /* main ioctl function for LOCALQ/WTP. It executes the cmd command on the given device (dev) */
    dev_t dev;
    ioctlcmd_t cmd;
    caddr_t addr;
    int flag;
    struct proc *p;
{
    int error = 0;
    int s;

    /* check cmd for superuser only */
    switch(cmd)
    {
        case WTP_GET_STATS:
            break;
#ifdef WTP_WRED
```

```
        case WRED_GETINFO:
            break;
    #ifdef WRED_STAT
        case WRED_GETSTAT:
            break;
    #endif /* WRED_STAT */
#endif /* WTP_WRED */
#ifdef EXT_STAT
        case GET_EXT_STATS:
            break;
#endif /* EXT_STAT */
        default:
            if ((error = suser(p->p_ucred, &p->p_acflag) ) !=0)
            {
                return(error);
            }
            break;
    }

    s=splimp();

    switch(cmd)
    {
        case WTP_ENABLE:
            error = wtp_setenable((struct wtp_interface *) addr, ENABLE);
            break;

        case WTP_DISABLE:
            error=wtp_setenable((struct wtp_interface *) addr, DISABLE);
            break;


        case WTP_IF_ATTACH:
            error = wtp_ifattach((struct wtp_interface *) addr);
            break;

        case WTP_IF_DETACH:
            error = wtp_ifdetach((struct wtp_interface *) addr);
            break;

        case WTP_SET_WEIGHT:
            error = wtp_setweight((struct wtp_setweight *) addr);
            break;

        case WTP_CONFIG:
            error = wtp_config((struct wtp_conf *) addr);
            break;

        case WTP_GET_STATS:
            error = wtp_getstats((struct wtp_getstats *) addr);
            break;
        case WTP_STAT_RESET:
            error = stat_reset((struct wtp_interface *) addr);
            break;

#ifdef WTP_WRED
```

```
        case WRED_CONFIG:
            error = wred_conf((struct wred_config *) addr);
            break;
        case WRED_GETINFO:
            error = wred_getinfo((struct wred_info *) addr);
            break;


    #ifdef WRED_STAT
        case WRED_GETSTAT:
            error = wred_getstats((struct wred_stat *) addr);
            break;
    #endif /* WRED_STAT */

#endif /* WTP_WRED */
#ifdef EXT_STAT
        case GET_EXT_STATS:
            error = get_extstat((struct ext_stat *) addr);
            break;
#endif /* EXT_STAT */

        default:
            error = EINVAL;
            break;
    }

    splx(s);

    return error;
}

#ifdef KLD_MODULE

#include <net/altq_conf.h>

static struct altqsw localq_sw =
    {"localq", localqopen, localqclose, localqioctl};

ALTQ_MODULE(altq_localq, ALTQT_LOCALQ, &localq_sw);

#endif /* KLD_MODULE */

#endif /* LOCALQ */
```

A.5 altq_wtp_wred.h

```
/* altq_wtp_wred.h
 *
 * ALTQ/WTP & WRED Implementation
 *
 * by Louis SWINNEN : FUNDP & CRC Alcatel Antwerpen
 *
 * version 0.6 BETA
 *
```

```
 */
#ifndef _NETINET_ALTQ_WTP_WRED_H
#define _NETINET_ALTQ_WTP_WRED_H


#define WRED_MAXDROP 3  /* Maximum 3 drop precedence */
#define FP_SHIFT 12
#include<netinet/altq_wtp.h>
/* #include<netinet/altq_wtp_opt.h> */
/* DEFAULT VALUES */
#define DEF_MINTH         200
#define DEF_MAXTH         1000
#define DEF_NUMDROP       3
#define DEF_WEIGHT        512
#define DEF_MAXP(i)       (i)==0?20:(i)==1?10:5
#define DEF_TTT           800
#define DROPPREC(i)       ((i)==2?0:(i)==4?1:(i)==6?2:0)


/* *************** Weighted Random Early Detection ************************
 *  min_th = minimum threshold : we begin to drop when the average queue size exceed this threshold
 *  max_th = maximum threshold : All incoming packets are discarded if the Avg Queue Size is above this threshold
 *  maxp = maximum drop probability
 *         ! maxp/(max_th - min_th) must be a power of two !
 *  C1 = maxp/(max_th - min_th)  used for EWMA filter
 *  C2 = (maxp * min_th)/(max_th - min_th)  = C1 * min_th
 *
 *
 * DROP PROBABILITY :
 *
 * The first drop probability 'pb' is given by the formula :
 * Pb = C1 * Avg - C2 Avg = Average Queue Size
 *
 * The final drop probablity is :
 * Pa = Pb/(1- (count * Pb))  [Uniform Random Variable]
 * Count = # received packets since the last dropped packet.
 *
 *
 *
 *
 * ESTIMATED AVERAGE QUEUE SIZE
 *
 * We will estimate the average queue size with a EWMA filter
 *
 * Wq = 1/weight                    ! Wq must be a (negative) power of two
 * Avg = (1-Wq) Avg + Wq * q        q = actual queue size
 *     = Avg + Wq*(q-Avg)
 *
 * Because we adjust only the average queue size when a packet arrives, we must take into account the
 * time during queue is empty and estimate the number of packet that the queue would be send during this
 * inactive time.
 * For an empty queue, we use the following formula :
 *
 *  m = (time - q_time) /s
 *  Avg = Avg * (1-Wq)^m
 *
```

```
 * time - q_time = time elapsed since the last packet arrival.
 * s = typical transmission time
 *
 *
 *
 * WARNING : To avoid the use of *, /, ^ operators, we will estimate a lot of parameter because
 *           these operators require a lot of time.
 *           In the kernel, we can't use floating point variable, so when it is needed, we will
 *           use a fixed point representation.
 *
 * FIXED-POINT REPRESENTATION
 *
 *    31                  11          0
 *    +-------------------+-----------+
 *    |                   |           |
 *    +-------------------+-----------+
 *
 *
 * The bit 12 last bit are considered as decimal part and the 20 other bits are considered as integer part
 *
 */



struct wred
{
    unsigned short int dropNum;     /* number of used drop precedence */
    unsigned int inv_maxp[WRED_MAXDROP];
    unsigned int weight_power;      /* Wq = 1/(2^weight_expo). For instance, 9 for 1/512 */
    unsigned int ttt;                 /* Typical Transmission Time */
    int count;                /* number of received packet since last drop */
    unsigned int nbpacket;            /* number of packet in buffers */
    int32_t min_th, max_th, fp_probd[WRED_MAXDROP];
    int32_t min_th_s, max_th_s;
    int32_t AvgQS;                     /* Average Queue size in FXP notation */
    struct timeval last_time;         /* time instant of last packet sent */
    int idle;
    int old;
    wtp_state_t wtps;                /* WTP information */
#ifdef WRED_STAT
    u_int drop_packets[WRED_MAXDROP];
    u_int sent_packets[WRED_MAXDROP];
#endif /* WRED_STAT */
};


struct wtab
{
    int w_weight;
    int w_param_max;
    int32_t w_tab[32];
} * main_wtab;

struct wred_config
{
    wtp_iface_t    iface;
```

```
    int minTH, maxTH;            /* Min & Max Threshold */
    unsigned short int NumDrop; /* Number of dropPrec */
    unsigned int * maxp;        /* table maxp^-1 value */
    unsigned int weight;        /* weight used for EWMA filter  !! must be a power of two !! */
    unsigned int ttt;            /* typical transmission time */
};

struct wred_info
{
    wtp_iface_t    iface;
    int32_t minTH, maxTH, AvgQS;
    unsigned int weightPW;
    unsigned int nbpacket;
    int count;
    u_int *drop_packets, *sent_packets;
};

#ifdef WRED_STAT
struct wred_stat
{
    wtp_iface_t iface;
    u_int drop_packets[WRED_MAXDROP], sent_packets[WRED_MAXDROP];
};
#endif /* WRED_STAT */



#endif /*  _NETINET_ALTQ_WTP_WRED_H */
```

A.6 altq_wtp_wred.c

```
/* ALTQ/WTP & WRED implementation
 * Louis SWINNEN : FUNDP & CRC Alcatel Antwerpen
 *
 * WRED version 0.6 BETA
 */
#ifndef _NETINET_ALTQ_WTP_WRED_C
#define _NETINET_ALTQ_WTP_WRED_C

#include <netinet/altq_wtp_util.h>
#include <netinet/altq_wtp_wred2.h>
#define int2FXP(i)    (i << FP_SHIFT)
#define isEmpty(wred) wred->nbpacket == 0

#define WRED_DEBUG

/* Prototype */
void mainwtab_alloc(int weight);
int mainwtab_destroy(void);
int32_t wred_pow_w(int n);
int wred_drop_early(int fp_len, int fp_probd, int count);
```

```
struct wred* allocWRED(int min_th, int max_th, int numDrop, int maxp[WRED_MAXDROP], int weight, int ttt, wtp_state_t *wtps);
struct wred* allocWRED_default(wtp_state_t *wtps);
void updateAVG(struct wred* wred);
int acceptPacket(int drop, struct wred* wred);
void flushWRED(struct wred* wred);
int wred_getinfo(struct wred_info* info);
#ifdef WRED_STAT
int wred_getstats(struct wred_stat* stat);
#endif /* WRED_STAT */
/* The following code is from altq_red.c */


/*
 * helper routine to calibrate avg during idle.
 * pow_w(wtab, n) returns (1-Wq)^n in fixed-point
 * here Wq = 1/weight and the code assumes Wq is close to zero.
 *
 * w_tab[n] holds ((1 - Wq)^(2^n)) in fixed-point.
 *
 */


void
mainwtab_alloc(weight)
    int weight;
{
    int i;

    MALLOC(main_wtab, struct wtab *, sizeof(struct wtab), M_DEVBUF, M_WAITOK);
    if (main_wtab == NULL)
        panic("wtab_alloc: malloc failed" );
    bzero(main_wtab, sizeof(struct wtab));
    main_wtab->w_weight = weight;

    /* Initialize the weight table */
    main_wtab->w_tab[0] = ((weight - 1) << FP_SHIFT) / weight;
    for(i = 1; i< 32; ++i)
    {
        main_wtab->w_tab[i] = (main_wtab->w_tab[i-1] * main_wtab->w_tab[i-1]) >> FP_SHIFT;
        if(main_wtab->w_tab[i] == 0 && main_wtab->w_param_max == 0)
            main_wtab->w_param_max = 1 << i;
    }
}

int
mainwtab_destroy(void)
{

    FREE(main_wtab, M_DEVBUF);
    return(0);
}

int32_t
wred_pow_w(n)
    int n;
{
```

```
    int i, bit;
    int32_t val;

    if(n >= main_wtab->w_param_max)
        return(0);

    val = 1 << FP_SHIFT;
    if(n <= 0)
        return(val);

    bit = 1;
    i = 0;
    while(n)
    {
        if(n & bit)
        {
            val = (val * main_wtab->w_tab[i]) >> FP_SHIFT;
            n &= ~bit;
        }
        ++i;
        bit <<= 1;
    }
    return(val);

}

/*
 * early drop probability is calculated as follows :
 *   prob = p_max * (avg - th_min) / (th_max - th_min)
 *   prob_a = prob / (1-count*prob)
 *          = (avg-th_min) / ((th_max-th_min)*inv_p_max - count *(avg-th_min))
 * here prob_a increases as successive undrop count increases.
 *
 */

int wred_drop_early(fp_len, fp_probd, count)
    int fp_len;        /* (avg - TH_MIN) in FXP */
    int fp_probd;    /* (TH_MAX-TH_MIN) / pmax in FXP */
    int count;        /* how many successive undropped packets */
{
    int d;            /* denominator of drop-probability */

    d = fp_probd - count * fp_len;

    if(d <= 0)
        /* count exceeds the hard limit: drop or mark */
        return(1);

    if((random() % d) < fp_len)
    {
        /* drop or mark */
        return(1);
    }

    /* no drop/mark */
    return(0);
```

```
}


/* ********************************** end of altq_red.c code *************************************** */

struct wred* allocWRED(int min_th, int max_th, int numDrop, int inv_maxp[WRED_MAXDROP], int weight, int ttt, wtp_state_t *wtps)
{
    int i = 0, w=weight;
    int32_t base;
    struct wred* wred;

    MALLOC(wred, struct wred*, sizeof(struct wred), M_DEVBUF, M_WAITOK);
    if(!wred)
        return(NULL);
    bzero(wred, sizeof(struct wred));

    /* Compute weight expo (must be a power of two) */
    for(i=0; w>1; ++i)
        w >>= 1;

    wred->weight_power = i;

    wred->min_th = min_th;
    wred->max_th = max_th;
    wred->min_th_s = min_th << (wred->weight_power + FP_SHIFT);
    wred->max_th_s = max_th << (wred->weight_power + FP_SHIFT);

    if(numDrop < WRED_MAXDROP)
        wred->dropNum = numDrop;
    else
        wred->dropNum = WRED_MAXDROP;

    wred->ttt = ttt;
    wred->count = 0;

    for(i=0;i<numDrop; ++i)
    {
        wred->inv_maxp[i] = inv_maxp[i];
        wred->fp_probd[i] = (1* (max_th - min_th) * inv_maxp[i]) << FP_SHIFT;
    }
    mainwtab_alloc(weight);

    microtime(&wred->last_time);

    wred->AvgQS = 0;
    wred->idle = 1;

#ifdef WRED_DEBUG
    printf("Weighted RED initialized ...\n");
    printf("... min threshold = %d (in FXP)\n",wred->min_th);
    printf("... max threshold = %d (in FXP)\n", wred->max_th);
    printf("... number of drop precedence = %d\n", wred->dropNum);
    printf("... weight power for EWMA filter = %d\n", wred->weight_power);
    printf("... typical transmission time = %d\n", wred->ttt);
    for(i=0;i<numDrop;++i)
    {
```

```
        printf("... fp_probd[%d] = FXP(%d), maxp[%d] = %d/FXP(%d)\n",i,wred->fp_probd[i],i,inv_maxp[i], int2FXP(inv_maxp[i]));
    }
#endif

    wtps->wred = wred;
    return(wred);

}

struct wred* allocWRED_default(wtp_state_t *wtps)
{
    int dropPrec[DEF_NUMDROP],i;
    for(i = 0; i<DEF_NUMDROP; ++i)
    {
        dropPrec[i] = DEF_MAXP(i);
    }
    return(allocWRED(DEF_MINTH, DEF_MAXTH, DEF_NUMDROP, dropPrec, DEF_WEIGHT, DEF_TTT, wtps));
}

void flushWRED(struct wred* wred)
{
    FREE(wred, M_DEVBUF);
    mainwtab_destroy();
}

/* return 0 if packet is dropped and 1 if packet is accepted */
int acceptPacket(int drop, struct wred* wred)
{
    int avg = wred->AvgQS;
    int droptype;

    if(wred->idle)
    {
        struct timeval now;
        int t, n;

        wred->idle = 0;
        microtime(&now);
        t = (now.tv_sec - wred->last_time.tv_sec);
        if(t > 60)
        {
            avg = 0;
        }
        else
        {
            t = t * 1000000 + (now.tv_usec - wred->last_time.tv_usec);
            n = t / wred->ttt - 1;
            if(n > 0)
                avg = (avg >> FP_SHIFT) * wred_pow_w(n);
        }
    }

    avg += (wred->nbpacket << FP_SHIFT) - (avg >> wred->weight_power);
    wred->AvgQS = avg;
    /* wred->count++; */
```

```
    droptype = 0; /* NO DROP */

    if(avg >= wred->min_th_s && wred->nbpacket > 1)
    {
        if(avg >= wred->max_th_s)
        {
            droptype = 1;
        }else
            if(wred->old == 0)
            {
                wred->count = 1;
                wred->old = 1;
            }else
                if(wred_drop_early((avg - wred->min_th_s) >> wred->weight_power, wred->fp_probd[drop], 0 /* wred->count*/))
                {
                    droptype = 1;
/* #ifdef WRED_STAT
                    wred->drop_packets[drop]++;
#endif */ /* WRED_STAT */
                }else
                    wred->count++;
    }else
    {
        wred->old = 0;
    }

#ifdef WRED_STAT
    if(!droptype)
        wred->sent_packets[drop]++;
    else
        wred->drop_packets[drop]++;
#endif /* WRED_STAT */

    return(!droptype);
}


static int
wred_conf(struct wred_config* cf)
{
        int i=0, weight = cf->weight;
        wtp_state_t *wtps;
        struct wred* wred;

        if((wtps= altq_lookup(cf->iface.wtp_ifacename, ALTQT_LOCALQ)) == NULL)
            return(EBADF);

        wred = wtps->wred;

        mainwtab_destroy();

        if(cf->NumDrop > WRED_MAXDROP)
            cf->NumDrop = WRED_MAXDROP;

        if(weight == 0)
            weight = DEF_WEIGHT;
```

```
        if(cf->ttt == 0)
            wred->ttt = DEF_TTT;
        else
            wred->ttt = cf->ttt;

        for(i=0; weight>1; ++i)
            weight >>= 1;

        wred->weight_power = i;


        wred->min_th = cf->minTH;
        wred->max_th = cf->maxTH;
        wred->min_th_s = cf->minTH << (wred->weight_power + FP_SHIFT);
        wred->max_th_s = cf->maxTH << (wred->weight_power + FP_SHIFT);

        wred->AvgQS = 0;
        wred->idle = 1;


        for(i=0; i<WRED_MAXDROP; ++i)
        {
            if(i < cf->NumDrop)
            {
                wred->inv_maxp[i] = cf->maxp[i];
                wred->fp_probd[i] = ( 1* (wred->max_th - wred->min_th) * wred->inv_maxp[i]) << FP_SHIFT;
            }
            else
                wred->inv_maxp[i] = 0;
        }

        mainwtab_alloc(cf->weight);

#ifdef WRED_DEBUG
        printf("Weighted RED initialized ...\n");
        printf("... min threshold = %d (in FXP)\n",wred->min_th);
        printf("... max threshold = %d (in FXP)\n", wred->max_th);
        printf("... number of drop precedence = %d\n", wred->dropNum);
        printf("... weight power for EWMA filter = %d\n", wred->weight_power);
        printf("... typical transmission time = %d\n", wred->ttt);
        for(i=0;i<cf->NumDrop;++i)
        {
            printf("... fp_probd[%d] = FXP(%d), maxp[%d] = %d/FXP(%d)\n",i,wred->fp_probd[i],i,cf->maxp[i], int2FXP(wred->inv_maxp[i]));
        }

#endif     /* WRED_DEBUG */

        return(0);
}

int wred_getinfo(struct wred_info* info)
{
    wtp_state_t *wtps;
    struct wred* wred;
```

```
    if((wtps= altq_lookup(info->iface.wtp_ifacename, ALTQT_LOCALQ)) == NULL)
        return(EBADF);

    wred = wtps->wred;
    info->minTH = wred->min_th;
    info->maxTH = wred->max_th;
    info->weightPW = wred->weight_power;
    info->AvgQS = wred->AvgQS;
    info->nbpacket = wred->nbpacket;
    info->count = wred->count;
    return(0);
}

#ifdef WRED_STAT

int wred_getstats(struct wred_stat* stat)
{
    int i;
    wtp_state_t *wtps;
    struct wred* wred;

    if((wtps= altq_lookup(stat->iface.wtp_ifacename, ALTQT_LOCALQ)) == NULL)
        return(EBADF);

    wred = wtps->wred;

    for(i=0;i<WRED_MAXDROP;++i)
    {
        stat->drop_packets[i] = wred->drop_packets[i];
        stat->sent_packets[i] = wred->sent_packets[i];
    }
    return(0);
}

#endif /* WRED_STAT */

#endif /* _NETINET_ALTQ_WTP_WRED_C */
```

A.7 wtpd.c

```
/*
 * WTPd : Waiting Time Priority Scheduler
 * for ALTQ/1.2 by Louis SWINNEN FUNDP & Alcatel CRC
 * Version 0.6 BETA
 */

#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <errno.h>
#include <sys/time.h>
#include <err.h>
```

```
#include <ctype.h>
#include <signal.h>
#include <sys/socket.h>
#include <sys/ioctl.h>
#include <sys/fcntl.h>
#include <netinet/in.h>
#include <netinet/altq.h>
#include<netinet/altq_wtp_opt.h>
#ifdef WTP_WRED
#define WRED_CONF    "wredconf"
#include<netinet/altq_wtp_wred2.h>
#else
#include <netinet/altq_wtp.h>
#endif /* WTP_WRED */
#include <arpa/inet.h>
#undef ENABLE
#undef DISABLE


#define WTP_DEV        "/dev/localq"
#ifndef IFNAMSIZ
#define IFNAMSIZ    16
#endif
#define PROMPT    "wtpd >"
#define QUIT    "quit"
#define ENABLE    "enable"
#define DISABLE    "disable"
#define GETSTATS    "getstats"
#define SETWEIGHT    "setweight"
#define RESETSTAT    "resetstat"

#define IFLIST_SIZE    16

#define WTPD_DEBUG

int debug=0;
int interactive = 0;
int nqueue = 0;
int qlimit = 0;
int niface = 0;
char* if_names[IFLIST_SIZE];

/*
 * function prototype
 */
static void usage(void);
static void sigint_handler(int sig);
int get_ifname(char* ifname, char *p);
int do_command(int fd, char *buf);


static void
usage(void)
{
    fprintf(stderr, "Usage: wtpd [-di] [-n nqueues] [-l limit] [if_name...]\n");
    fprintf(stderr, "    -d: debug (do not daemonize)\n");
```

```
    fprintf(stderr, "    -i: interactive\n");
    fprintf(stderr, "    -n: number of queues (Min: 2, Max : %d\n", MAX_QNUMS);
    fprintf(stderr, "    -l: queue size limit in bytes\n");
    exit(1);
}

static void
sigint_handler(int sig)
{
    /* do nothing */
}

void main(int argc, char* argv[])
{
    int fd, ch, i, if_idx;
    char *cp, buf[256];

    while((ch = getopt(argc, argv, "dil:n:")) != EOF)
    {
        switch(ch)
        {
            case 'd': /* debug mode */
                debug = 1;
                break;
            case 'i': /* interactive mode */
                interactive = 1;
                break;
            case 'l': /* queue size limit (in bytes) */
                cp = NULL;
                qlimit = strtol(optarg, &cp, 0);
                if(cp != NULL)
                {
                    if(*cp == 'K' || *cp == 'k')
                        qlimit *=1024;
                    else
                        if(*cp == 'M' || *cp == 'm')
                            qlimit *= 1024*1024;
                }
                if(qlimit < MIN_QLIMIT)
                {
                    fprintf(stderr, "qlimit: %d bytes to small (min: %d) !\n", qlimit, MIN_QLIMIT);
                    usage();
                }
                break;
            case 'n': /* number of queues */
                nqueue = atoi(optarg);
                if(nqueue < MIN_QNUMS)
                {
                    fprintf(stderr, "nqueues: minimum number of queues : %d\n", MIN_QNUMS);
                    usage();
                }
                break;
        }
    }

#ifdef WTPD_DEBUG
```

```
    printf("[DEBUG] Options : debug = %d, interactive = %d, qlimit = %d\n", debug, interactive, qlimit);
#endif /* WTPD_DEBUG */
    argc -= optind;
    argv += optind;

    if(argc > 0)
    {
        for(if_idx = 0; if_idx < IFLIST_SIZE && argc > 0; if_idx ++)
        {
            if_names[if_idx] = argv[0];
            argc--;
            argv++;
        }
        if(if_idx == IFLIST_SIZE)
            errx(1, "wtpd: too many interfaces specified !");

    }
    else     if(!interactive)
            {
                fprintf(stderr, "missing interface name ! \n");
                usage();
            }
    signal(SIGINT, sigint_handler);
    signal(SIGTERM, sigint_handler);

    /*
     * open file and get a file descriptor
     */

#ifdef WTPD_DEBUG
    printf("[DEBUG] Attempt to open the device %s\n", WTP_DEV);
#endif /* WTPD_DEBUG */

    if ((fd = open(WTP_DEV, O_RDONLY)) <0)
        err(1, "Can't open wtp device file");



    if(interactive)
    {
        while(1)
        {
            printf("%s", PROMPT);
            fgets(buf, sizeof(buf), stdin);

            /*
             * execute command
             */
            if(do_command(fd, buf) < 0)
                break;
        }
    }
    else
    {
        for(i=0;i<if_idx;++i)
        {
```

```
            sprintf(buf, "%s %s %d", if_names[i], ENABLE, nqueue);
            do_command(fd, buf);
        }

        if(!debug)
            daemon(0,0);

        /* wait for signals and terminate */
        pause();

        if(debug)
            printf("Got signal . Exiting . . . \n");
    }

    while(niface > 0)
    {
#ifdef WTPD_DEBUG
        printf("[DEBUG] Attempt to disable the interface %s\n", if_names[0]);
#endif /* WTPD_DEBUG */

        sprintf(buf, "%s %s", if_names[0], DISABLE);
        do_command(fd, buf);
    }

#ifdef WTPD_DEBUG
    printf("[DEBUG] Attempt to close the device %s\n", WTP_DEV);
#endif /* WTPD_DEBUG */

    close(fd);
}

int get_ifname(char* ifname, char *p)
{
    int i;

    for(i=0; i<IFNAMSIZ -1 && *p != ' ' && *p != '\t' && *p !='\n'; ++i)
        ifname[i] = *p++;
    ifname[i] = '\0';

    return i;
}

void del_iface(int pos)
{
    int i;
    for(i=pos; i<niface -1 ; ++i)
        if_names[i]=if_names[i+1];
    niface--;
}

int exist_iface(char* ifname_)
{
    int i;
    for(i=0;i<niface;++i)
        if(strlen(if_names[i]) == strlen(ifname_) && !strcmp(if_names[i], ifname_))
            return i;
```

```
    return -1;
}

int do_command(int fd, char *buf)
{
    struct wtp_interface iface;
    int len;
    char *p, ifname[IFNAMSIZ];

    for(p = buf; *p== ' ' || *p == '\t'; ++p)
        ;

    if(*p=='\n')
        return 0;

    if(!strncmp(QUIT, p, strlen(QUIT)))
        return -1;

    /*
     * get interface name
     */
    if((len = get_ifname(ifname,p)) == IFNAMSIZ - 1)
    {
        fprintf(stderr, "too long interface name : %s\n", ifname);
        return 0;
    }
    strcpy(iface.wtp_ifacename, ifname);
    iface.wtp_ifacelen = len;

    p += len;
    while(*p == ' ' || *p == '\t')
        p++;
    if(!strncmp(ENABLE, p, strlen(ENABLE)))
    {
        int n, q;

#ifdef WTPD_DEBUG
        printf("[DEBUG] ENABLE Identified \n");
#endif /* WTPD_DEBUG */

        p += strlen(ENABLE);

        while(*p == ' ' || *p == '\t')
            ++p;
        for(n = 0; '0' <= *p && *p <= '9'; ++p)
            n= n*10 + *p - '0';

        /* if(n != 0) */
        nqueue = n;


        while(*p == ' ' || *p == '\t')
            ++p;

        for(q = 0; '0' <= *p && *p <= '9'; ++p)
            q= q*10 + *p - '0';
```

```
        /* if(q != 0) */
        qlimit = q;

#ifdef WTPD_DEBUG
    printf("[DEBUG] Attempt to ATTACH ...\n");
#endif /* WTPD_DEBUG */

        if(ioctl(fd, WTP_IF_ATTACH, &iface) < 0)
            err(1, "ioctl WTP_IF_ATTACH");

        if(nqueue != 0 || qlimit != 0)
        {
            /* configure the interface */
            struct wtp_conf conf;

            conf.iface = iface;
            conf.nqueues = nqueue;
            conf.qlimit = qlimit;

#ifdef WTPD_DEBUG
            printf("[DEBUG] Attempt to configure interface with nqueue = %d, qlimit = %d\n", nqueue, qlimit);
#endif /* WTPD_DEBUG */

            if(ioctl(fd, WTP_CONFIG, &conf) < 0)
                err(1, "ioctl WTP_CONF");
        }

#ifdef WTPD_DEBUG
    printf("[DEBUG] Attempt to ENABLE\n");
#endif /* WTPD_DEBUG */
        if(ioctl(fd, WTP_ENABLE, &iface) < 0)
            err(1, "ioctl WTP_ENABLE");

        if_names[niface] = (char*)malloc(IFNAMSIZ);
        strncpy(if_names[niface++], ifname, len);
    }
    else
    {
        if(!strncmp(DISABLE, p, strlen(DISABLE)))
        {
            int i = exist_iface(ifname);
#ifdef WTPD_DEBUG
            printf("[DEBUG] DISABLE identified ! \n");
#endif /* WTPD_DEBUG */


            if(i >=0 )
            {

#ifdef WTPD_DEBUG
                printf("[DEBUG] Attempt WTP_DISABLE ! \n");
#endif /* WTPD_DEBUG */

                if(ioctl(fd, WTP_DISABLE, &iface) < 0)
                    err(1, "ioctl WTP_DISABLE");
```

```
#ifdef WTPD_DEBUG
                printf("[DEBUG] Attempt WTP_IF_DETACH ... \n");
#endif /* WTPD_DEBUG */

                if(ioctl(fd, WTP_IF_DETACH, &iface) < 0)
                    err(1, "ioctl WTP_IF_DETACH");

                free(if_names[i]);
                del_iface(i);
            }else
                printf("WTPD error : This interface is not enabled.\n");

#ifdef WTPD_DEBUG
            printf("[DEBUG] DISABLE end ! \n");
#endif /* WTPD_DEBUG */

        }else
        {
            if(!strncmp(SETWEIGHT, p, strlen(SETWEIGHT)))
            {
                struct wtp_setweight sw;
                sw.iface = iface;
                p += strlen(SETWEIGHT);


#ifdef WTPD_DEBUG
                printf("[DEBUG] SETWEIGHT identified ! \n");
#endif /* WTPD_DEBUG */

                while(*p== ' ' || *p == '\t')
                    ++p;
                for(sw.dscp = 0; '0' <= *p && *p <='9'; ++p)
                    sw.dscp = sw.dscp * 10 + *p - '0';

                while(*p ==' ' || *p =='\t')
                    p++;

                for(sw.weight = 0; '0' <= *p && *p <= '9'; ++p)
                    sw.weight = sw.weight * 10 + *p - '0';

                if(exist_iface(ifname) >= 0)
                {
#ifdef WTPD_DEBUG
                    printf("[DEBUG] Attempt WTP_SET_WEIGHT with DSCP = %d/%x, WEIGHT = %d...\n", sw.dscp, sw.dscp, sw.weight);
                    printf("... cmd value = %d\n", WTP_SET_WEIGHT);
#endif /* WTPD_DEBUG */

                    if(ioctl(fd, WTP_SET_WEIGHT, &sw) < 0)
                        err(1, "ioctl WTP_SETWEIGHT");
                }else
                    printf("WTPd error : interface not enabled\n");
#ifdef WTPD_DEBUG
                printf("[DEBUG] SETWEIGHT end! \n");
#endif /* WTPD_DEBUG */
```

```
        }else
        {
            if(!strncmp(GETSTATS, p, strlen(GETSTATS)))
            {
                struct wtp_getstats gs;
                gs.iface = iface;
#ifdef WTPD_DEBUG
                printf("[DEBUG] GETSTATS identified ! \n");
#endif /* WTPD_DEBUG */


                p += strlen(GETSTATS);
                while(*p==' ' || *p == '\t')
                    p++;

                for(gs.dscp = 0; '0' <= *p && *p<= '9'; ++p)
                    gs.dscp = gs.dscp * 10 + *p - '0';


#ifdef WTPD_DEBUG
                printf("[DEBUG] Attempt WTP_GET_STATS with dscp = %d/%x\n", gs.dscp, gs.dscp);
#endif /* WTPD_DEBUG */

                if(ioctl(fd, WTP_GET_STATS, &gs) < 0)
                    err(1, "ioctl WTP_GET_STATS");

                printf("Bytes = %d\n", gs.stats.bytes);
                printf("Weight = %d\n", gs.stats.weight);
                printf("Sent : packets = %u, bytes = %qu\n", gs.stats.sent_packets, gs.stats.sent_bytes);
                printf("Drop : packets = %u, bytes = %qu\n", gs.stats.drop_packets, gs.stats.drop_bytes);

#ifdef WTPD_DEBUG
                printf("[DEBUG] GETSTATS end ! \n");
#endif /* WTPD_DEBUG */
            }
            else
            {
                if(!strncmp(RESETSTAT, p, strlen(RESETSTAT)))
                {
                            int i = exist_iface(ifname);
#ifdef WTPD_DEBUG
                            printf("[DEBUG] RESETSTAT identified ! \n");
#endif /* WTPD_DEBUG */


                            if(i >=0)
                            {

#ifdef WTPD_DEBUG
                                printf("[DEBUG] Attempt WTP_STAT_RESET ! \n");
#endif /* WTPD_DEBUG */

                                if(ioctl(fd, WTP_STAT_RESET, &iface) < 0)
                                    err(1, "ioctl WTP_STAT_RESET");

                            }
```

```
        }
        else
#ifdef WTP_WRED
        {
            if(!strncmp(WRED_CONF, p, strlen(WRED_CONF)))
            {
                /*
                    struct wred_config
                    {
                        wtp_iface_t    iface;
                        int minTH, maxTH;                Min & Max Threshold
                        unsigned short int NumDrop;     Number of dropPrec
                        unsigned int * maxp;            table maxp^-1 value
                        unsigned int weight;            weight used for EWMA filter  !! must be a power of two !!
                        unsigned int ttt;                typical transmission time
                    }; */
                struct wred_config conf;
                int i = exist_iface(ifname);



                if(i >=0)
                {
                    int drop;

                    conf.iface = iface;
                    p += strlen(WRED_CONF);

                    while(*p==' ' || *p == '\t')
                        p++;

                    for(conf.minTH = 0; '0' <= *p && *p<= '9'; ++p)
                        conf.minTH = conf.minTH * 10 + *p - '0';

                    while(*p==' ' || *p == '\t')
                        p++;

                    for(conf.maxTH = 0; '0' <= *p && *p<= '9'; ++p)
                        conf.maxTH = conf.maxTH * 10 + *p - '0';

                    while(*p==' ' || *p == '\t')
                        p++;

                    for(conf.NumDrop = 0; '0' <= *p && *p<= '9'; ++p)
                        conf.NumDrop = conf.NumDrop * 10 + *p - '0';


                    if(conf.NumDrop <= WRED_MAXDROP)
                    {
                        int wred_init[WRED_MAXDROP];
                        for(drop = 0; drop < conf.NumDrop; ++drop)
                        {
                            while(*p==' ' || *p == '\t')
                                p++;

                            for(wred_init[drop] = 0; '0' <= *p && *p<= '9'; ++p)
```

```
                                           wred_init[drop] = wred_init[drop] * 10 + *p - '0';
                                  }

                                  conf.maxp = wred_init;

                                  while(*p==' ' || *p == '\t')
                                      p++;

                                  for(conf.weight = 0; '0' <= *p && *p<= '9'; ++p)
                                      conf.weight = conf.weight * 10 + *p - '0';

                                  while(*p==' ' || *p == '\t')
                                      p++;

                                  for(conf.ttt = 0; '0' <= *p && *p<= '9'; ++p)
                                      conf.ttt = conf.ttt * 10 + *p - '0';

                                  if(ioctl(fd, WRED_CONFIG, &conf) < 0)
                                      err(1, "ioctl WRED_CONFIG");

                              }else
                              {
                                  printf("The given number of drop precedence is too large (MAX: %d)\n", WRED_MAXDROP);
                              }

                          }else
                          {
                              printf("This interface isn't initialised\n");
                          }


                      }else
#endif
                  {
                      fprintf(stderr, "Usage : interface %s [nqueues [qlimit]]\n", ENABLE);
                      fprintf(stderr, "        interface %s\n", DISABLE);
                      fprintf(stderr, "        interface %s dscp\n", GETSTATS);
                      fprintf(stderr, "        interface %s dscp weight \n", SETWEIGHT);
                      fprintf(stderr, "        interface %s\n", RESETSTAT);
#ifdef WTP_WRED
                      fprintf(stderr, "        interface %s minTH maxTX numDrop {maxp} weight ttt\n", WRED_CONF); }
#endif /* WTP_WRED */
                  }

              }
          }

      }
    }
    return 0;
}
```

A.8 wtpstat.c

```
/*
WTPStat.c
by Louis SWINNEN - FUNDP & Alcatel CRC ANTWERPEN
Version 0.6 BETA
*/

/* Compiler Option */
#include <netinet/altq_wtp_opt.h>


#define int2FXP(i)    ((i) << FP_SHIFT)
#define FXP2int(i)  ((i) >> FP_SHIFT)
#define FLOAT(i)     ((float) (i))

#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <signal.h>
#include <errno.h>
#include <err.h>
#ifndef NO_CURSES
#include <curses.h>
#endif /* NO_CURSES */

#include<math.h>
#include <sys/param.h>
#include <sys/ioctl.h>
#include <sys/fcntl.h>
#include <sys/time.h>

#include <sys/socket.h>
#include <net/if.h>

#include <netinet/altq.h>


#define WTP_DEVICE    "/dev/localq"
#define NTOP    10

#ifdef WTP_WRED
#include<netinet/altq_wtp_wred2.h>
#else
#include <netinet/altq_wtp.h>
#endif /* WTP_WRED */

/* #define WTPSTAT_DEBUG */

int file=0;
u_quad_t *oldSentByte, *dscpSentByte;
FILE* fiSt[7], *fiWRED;

float convertmsec(struct timeval tv)
{
```

```
    return(((float)tv.tv_sec*1000.0) + ((float)tv.tv_usec/1000.0));
}

struct wtpinfo
{
    int dscp;
    queue_stats stats;
    u_quad_t last_bytes;
    double mbps;
};

int fd = -1;
char * if_name = NULL;
int interval = 5;
int ntop = NTOP;

static void
usage (void)
{
    fprintf(stderr, "usage: wtpstat [-i interval] [-n ntop] [-f base name of] interface\n");
    exit(1);
}

static void
sigint_handler(int sig)
{
    int i;
#ifndef NO_CURSES
    if(!file)
        endwin();
#endif /* NO_CURSES */

    fprintf(stderr, "Exiting on signal %d\n", sig);

    close(fd);
    free(oldSentByte);
    free(dscpSentByte);


    for(i=0; i< file; ++i)
    {
        fclose(fiSt[i]);
#ifdef WTP_WRED
        fclose(fiWRED);
#endif /* WTP_WRED */
    }

    exit(0);
}

int main(int argc, char **argv)
{
    struct wtp_getstats wtp_stats;
#ifdef WRED_STAT
    struct wred_stat wred_stats;
#endif /* WRED_STAT */
```

```
    struct timeval cur_time, last_time;
    u_quad_t xmit_bytes, last_bytes;
    int msecG, i, j, k, nqueue;
    char* if_name;
    char fn[35], ffn[255];
    int ch;
    struct wtpinfo *qinfo, **top;
#ifdef WTP_WRED
    struct wred_info info;
#endif /* WTP_WRED */
#ifdef EXT_STAT
    struct ext_stat es;
#endif /* EXT_STAT */

    for(i=0;i<7;++i)
    {
        fiSt[7]=NULL;
    }

    while((ch=getopt(argc, argv, "i:n:f:")) != EOF)
    {
        switch(ch)
        {
            case 'i':
                interval = atoi(optarg);
                break;
            case 'n' :
                ntop = atoi(optarg);
                break;

            case 'f' :
                sprintf(fn, "%s", optarg);
                printf("[DEBUG] using OUTPUT FILE %s\n",fn);
                file = 1;
                break;
        }
    }
    argc -= optind;
    argv +=optind;
    if(argc > 0)
    {
        if_name = argv[0];
    }
    else
    {
        if(argc==0)
            fprintf(stderr,"missing interface name\n");
        usage();
    }

    signal(SIGINT, sigint_handler);
    signal(SIGTERM, sigint_handler);

    if((fd = open(WTP_DEVICE, O_RDONLY)) < 0)
        err(1, "wtp open");
```

```
    if(strlen(if_name) > IFNAMSIZ)
        err(1, "interface name %s too long!\n", if_name);

    wtp_stats.iface.wtp_ifacename[IFNAMSIZ-1] = '\0';
    strncpy(wtp_stats.iface.wtp_ifacename, if_name, IFNAMSIZ-1);
    wtp_stats.iface.wtp_ifacelen = strlen(wtp_stats.iface.wtp_ifacename);

#ifdef WTP_WRED
    info.iface.wtp_ifacename[IFNAMSIZ-1] = '\0';
    strncpy(info.iface.wtp_ifacename, if_name, IFNAMSIZ-1);
    info.iface.wtp_ifacelen = strlen(info.iface.wtp_ifacename);
#endif /* WTP_WRED */

#ifdef EXT_STAT
    es.iface.wtp_ifacename[IFNAMSIZ-1] = '\0';
    strncpy(es.iface.wtp_ifacename, if_name, IFNAMSIZ-1);
    es.iface.wtp_ifacelen = strlen(es.iface.wtp_ifacename);
#endif /* EXT_STAT */

#ifdef WRED_STAT
    wred_stats.iface.wtp_ifacename[IFNAMSIZ-1] = '\0';
    strncpy(wred_stats.iface.wtp_ifacename, if_name, IFNAMSIZ-1);
    wred_stats.iface.wtp_ifacelen = strlen(wred_stats.iface.wtp_ifacename);
#endif /* WTP_STAT */


    /*
     * first, find out how many queues are available
     */
    for (i=0; i<MAX_QNUMS; ++i)
    {
        wtp_stats.dscp = DSCP(i);

        if(ioctl(fd, WTP_GET_STATS, &wtp_stats) < 0)
            break;

    }
    if(file)
        file = nqueue = i;
    else
        nqueue = i;

    printf("wtp on %s: %d queues are used\n", if_name, nqueue);


    qinfo = malloc(nqueue * sizeof(struct wtpinfo));
    top = malloc(ntop * sizeof(struct wtpinfo*));
    if(!qinfo || !top)
        err(1, "malloc failed");
    if(!file)
    {
#ifndef NO_CURSES
        sleep(2);     /* wait a bit before clearing the screen */
        initscr();
#endif /* NO_CURSES */
    }
```

```
    gettimeofday(&last_time, NULL);
    last_time.tv_sec -= interval;
    last_bytes = 0;


    if(file)
    {
        for(i=0;i<nqueue;++i)
        {
            sprintf(ffn,"%s.queue%i.csv", fn,i);
            fiSt[i] = fopen(ffn, "wt");
            if(!fiSt[i])
            {
                printf("Can't open %s", ffn);
                exit(255);
            }

            fprintf(fiSt[i], "WTPStat file, Louis SWINNEN\n");
            fprintf(fiSt[i], "Queue :, %d, DSCP = %x, %d\n",i, DSCP(i), DSCP(i));
#ifdef Q_DELAY
    #ifdef EXT_STAT
             fprintf(fiSt[i],"Weight,QS # Packet,QS(KB),# packets sent, # KB sents, # pkts dropped, # KB dropped, \\
             Throughput (mbps),Throughput DROP_0,Throughput DROP_1,Throughtput DROP_2, Avg Queueing Delay, Queueing Delay\n");
    #else
             fprintf(fiSt[i],"Weight,QS # Packet,QS(KB),# packets sent, # KB sents, # pkts dropped, # KB dropped, \\
             Mbps, Avg Queueing Delay, Queueing Delay\n");
    #endif /* EXT_STAT */
#else
    #ifdef EXT_STAT
            fprintf(fiSt[i],"Weight,QS # Packet,QS(KB),# packets sent, # KB sents, # pkts dropped, # KB dropped, \\
            Throughput (mbps), Throughput DROP_0, Throughput DROP_1, Throughtput DROP_2\n");
    #else
            fprintf(fiSt[i],"Weight,QS # Packet,QS(KB),# packets sent, # KB sents, # pkts dropped, # KB dropped, Mbps\n");
    #endif /* EXT_STAT */
#endif /* Q_DELAY */


        }
#ifdef WTP_WRED
        sprintf(ffn,"%s.wred.csv", fn);
        fiWRED = fopen(ffn,"wt");
        if(!fiWRED)
        {
            printf("Can't open %s", ffn);
            exit(255);
        }
        fprintf(fiWRED, "MinTH,MaxTH,AvgQS,QS,Count\n");

#endif /*WTP_WRED */
    }

#ifdef EXT_STAT
        oldSentByte = malloc(sizeof(u_quad_t)*nqueue*WRED_MAXDROP);
        dscpSentByte = malloc(sizeof(u_quad_t)*nqueue*WRED_MAXDROP);
```

```
        for(i=0;i<nqueue;++i)
            for(j=0;j<WRED_MAXDROP;++j)
            {
                /* printf("i= %d, j= %d, val = %d\n", i, j, i*WRED_MAXDROP + j); */
                dscpSentByte[i*WRED_MAXDROP+j]=0;
                oldSentByte[i*WRED_MAXDROP+j]=0;
            }
#endif /* EXT_STAT */

    while(1)
    {

        for(j=0; j< ntop; ++j)
            top[j] = NULL;

        for(i=0; i<nqueue; ++i)
        {
            wtp_stats.dscp = DSCP(i);

            if (ioctl (fd, WTP_GET_STATS, &wtp_stats) < 0)
                err(1, "ioctl WTP_GET_STATS");

            qinfo[i].dscp = DSCP(i);
            qinfo[i].stats = wtp_stats.stats;

        }


        gettimeofday(&cur_time, NULL);
        msecG = (cur_time.tv_sec - last_time.tv_sec) * 1000 + (cur_time.tv_usec - last_time.tv_usec) / 1000;
        last_time = cur_time;

#ifdef WTP_WRED
        if (ioctl (fd, WRED_GETINFO, &info) < 0)
            err(1, "ioctl WRED_GETINFO");

#ifdef WRED_STAT
        if (ioctl (fd, WRED_GETSTAT, &wred_stats) < 0)
            err(1, "ioctl WRED_GETSTAT");
#endif /* WRED_STAT */
#endif /* WTP_WRED */

        /*
         * calculate the throughput of each queue
         */

        for(i=0; i<nqueue; ++i)
        {
#ifdef EXT_STAT
            es.dscp = DSCP(i);
            /* printf("************** DSCP = %x *******************\n", es.dscp); */
            if (ioctl (fd,GET_EXT_STATS, &es) < 0)
                err(1, "ioctl GET_EXT_STATS");

            for(j=0;j<WRED_MAXDROP;++j)
            {
```

```
                oldSentByte[i*WRED_MAXDROP + j] = dscpSentByte[i*WRED_MAXDROP + j];
                dscpSentByte[i*WRED_MAXDROP + j] = es.dscp_sent_bytes[j];
            }
#endif /* EXT_STAT */
            xmit_bytes = qinfo[i].stats.sent_bytes - qinfo[i].last_bytes;
            qinfo[i].mbps = (double)xmit_bytes * 8.0 / (double)msecG * 1000.0 / 1000.0 / 1000.0;
            qinfo[i].last_bytes = qinfo[i].stats.sent_bytes;


            for(j=0;j<ntop;++j)
            {
                if(top[j]==NULL)
                {
                    top[j] = &qinfo[i];
                    break;
                }
                if (top[j]->mbps < qinfo[i].mbps || (top[j]->mbps == qinfo[i].mbps && top[j]->stats.sent_packets < qinfo[i].stats.sent_packets))
                {
                    for(k=ntop-1;k>j; --k)
                        top[k] = top[k-1];
                    top[j] = &qinfo[i];
                    break;
                }
            }
        }

        /*
         * display top
         */



         if(!file)
         {
#ifdef Q_DELAY
            printf("[ID]  W.  QS    (KB) SENT(pkts)       (KB)         DROP(pkts)        (KB)        Mbps  Delay(msec)\n");
#else
            printf("[ID]  W.  QS    (KB) SENT(pkts)       (KB)         DROP(pkts)        (KB)        Mbps\n");
#endif /* Q_DELAY */
            for(j=0;j<ntop;++j)
            {
                if(top[j] != NULL)
#ifdef Q_DELAY
                {
                    /* float msec = convertmsec(top[j]->stats.tot_delay);
                    float spack = (float) (top[j]->stats.sent_packets); */
                    printf("[%2d]  %1d  %2d  %4d  %10u %14qu %10u %14qu %8.2f %1d \t\t%d\n",
#else
                    printf("[%2d]  %1d  %2d  %4d  %10u %14qu %10u %14qu %8.2f\n",
#endif /* Q_DELAY */
                        top[j]->dscp,
                        top[j]->stats.weight,
                        top[j]->stats.nbpacket,
                        top[j]->stats.bytes / 1024,
                        top[j]->stats.sent_packets,
                        top[j]->stats.sent_bytes / 1024,
```

```
                        top[j]->stats.drop_packets,
                        top[j]->stats.drop_bytes / 1024,
#ifdef Q_DELAY
                        top[j]->mbps,
                        /* spack>0?(msec / spack):0,*/
                        top[j]->stats.last_del_pack.tv_usec,
                        top[j]->stats.mbuf_bytes);
                }
#else
                        top[j]->mbps);
#endif /* Q_DELAY */
                else
                    printf("\n");
            }
#ifdef WTP_WRED
            printf("\n\nWRED Informations :\n");
            printf("MinTH (FXP) : %d(%d), MaxTH (FXP) : %d(%d)                \n", info.minTH, int2FXP(info.minTH), info.maxTH, int2FXP(info.maxTH));
            printf("MinTH_S     : %d, MaxTH_S : %d                              \n", (info.minTH << (info.weightPW + FP_SHIFT) ), \\
               (info.maxTH << (info.weightPW + FP_SHIFT) ));
            printf("AvgQS(FXP)  : %d(%d)                                        \n", (FXP2int(info.AvgQS) >> info.weightPW), info.AvgQS);
            printf("QS(FXP)     : %d(%d)                                        \n", info.nbpacket, int2FXP(info.nbpacket));
            printf("Count : %d\n", info.count);
  #ifdef WRED_STAT
            for(j=0;j<WRED_MAXDROP;++j)
            {
                printf("# drop[%d]=%d\t",j, wred_stats.drop_packets[j]);
                printf("# sent[%d]=%d\t(%4.2f)\n",j, wred_stats.sent_packets[j], wred_stats.sent_packets[j]>0 ? \\
                   FLOAT(wred_stats.drop_packets[j]) / FLOAT(wred_stats.sent_packets[j] + wred_stats.drop_packets[j]) : 0);
            }
  #endif /* WRED_STAT */
#endif /* WTP_WRED */

#ifndef NO_CURSES
            refresh();
            mvcur(ntop+1,0,0,0);
#endif
        }else
        { /* !!!!!!!!!!!!!!!!!!!!!!!!!!!  FILE */


            for(j=0;j<ntop;++j)
            {
                if(top[j] != NULL)
                {
#ifdef Q_DELAY
  #ifdef EXT_STAT
                    fprintf(fiSt[DSCP2int(top[j]->dscp)],"%d,%d,%d,%u,%qu,%u,%qu,%f, %f, %f ,%f,%ld\n",
  #else
                    fprintf(fiSt[DSCP2int(top[j]->dscp)],"%d,%d,%d,%u,%qu,%u,%qu,%f,%ld\n",
  #endif /* EXT_STAT */
#else /* Q_DELAY */
  #ifdef EXT_STAT
                    fprintf(fiSt[DSCP2int(top[j]->dscp)],"%d,%d,%d,%u,%qu,%u,%qu,%f, %f, %f\n",
  #else
                    fprintf(fiSt[DSCP2int(top[j]->dscp)],"%d,%d,%d,%u,%qu,%u,%qu,%f\n",
  #endif /* EXT_STAT */
```

```
#endif /* Q_DELAY */
                        top[j]->stats.weight,
                        top[j]->stats.nbpacket,
                        top[j]->stats.bytes / 1024,
                        top[j]->stats.sent_packets,
                        top[j]->stats.sent_bytes / 1024,
                        top[j]->stats.drop_packets,
                        top[j]->stats.drop_bytes / 1024,
#ifdef Q_DELAY
                        top[j]->mbps,
#ifdef EXT_STAT
                        ((double)(dscpSentByte[DSCP2int(top[j]->dscp) * WRED_MAXDROP + 0]-oldSentByte[DSCP2int(top[j]->dscp) * \\
                            WRED_MAXDROP + 0])) * 8.0 / (double)msecG * 1000.0 / 1000.0 / 1000.0,
                        ((double)(dscpSentByte[DSCP2int(top[j]->dscp) * WRED_MAXDROP + 1]-oldSentByte[DSCP2int(top[j]->dscp) * \\
                            WRED_MAXDROP + 1])) * 8.0 / (double)msecG * 1000.0 / 1000.0 / 1000.0,
                        ((double)(dscpSentByte[DSCP2int(top[j]->dscp) * WRED_MAXDROP + 2]-oldSentByte[DSCP2int(top[j]->dscp) * \\
                            WRED_MAXDROP + 2])) * 8.0 / (double)msecG * 1000.0 / 1000.0 / 1000.0,
#endif /* EXT_STAT */
                        top[j]->stats.last_del_pack.tv_usec);
#else
                        top[j]->mbps
#ifdef EXT_STAT
                        ,((double)(dscpSentByte[DSCP2int(top[j]->dscp) * WRED_MAXDROP + 0]-oldSentByte[DSCP2int(top[j]->dscp)* \\
                            WRED_MAXDROP + 0])) * 8.0 / (double)msecG * 1000.0 / 1000.0 / 1000.0,
                        ((double)(dscpSentByte[DSCP2int(top[j]->dscp) * WRED_MAXDROP + 1]-oldSentByte[DSCP2int(top[j]->dscp) * \\
                            WRED_MAXDROP + 1])) * 8.0 / (double)msecG * 1000.0 / 1000.0 / 1000.0,
                        ((double)(dscpSentByte[DSCP2int(top[j]->dscp) * WRED_MAXDROP + 2]-oldSentByte[DSCP2int(top[j]->dscp * \\
                            WRED_MAXDROP + 2])) * 8.0 / (double)msecG * 1000.0 / 1000.0 / 1000.0,
#endif /* EXT_STAT */
                        );
#endif /* Q_DELAY */

                }
            }

#ifdef WTP_WRED
    #ifdef WRED_STAT
            fprintf(fiWRED,"%d,%d,%d,%d,%d,",
    #else /* WRED_STAT */
            fprintf(fiWRED,"%d,%d,%d,%d,%d\n",
    #endif /* WRED_STAT */
                    info.minTH,
                    info.maxTH,
                    (FXP2int(info.AvgQS) >> info.weightPW),
                    info.nbpacket,
                    info.count);

 #ifdef WRED_STAT
            for(j=0;j<WRED_MAXDROP;++j)
            {
                fprintf(fiWRED, "# drop[%d],%d,",j, wred_stats.drop_packets[j]);
                fprintf(fiWRED, "# sent[%d],%d,",j, wred_stats.sent_packets[j]);
            }
            fprintf(fiWRED,"\n");
  #endif /* WRED_STAT */
```

```
#endif /* WTP_WRED */
        }
        sleep(interval);
    }
    return(0);
}
```

Bibliography

[1] S. Blake, D. Blake, M. Carlson, E. Davies, Z. Wang, and W. Weiss. An Architecture for Differentiated Services. RFC 2475, December 1998.

[2] J-Y. Le Boudec and F. Farkas. A Delay Bound for a Network with Aggregate Scheduling, May 1998.

[3] R. Braden and al. Recommendations on Queue Management and Congestion Avoidance in the Internet. RFC 2309, April 1998.

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

[5] K. Cho. A Framework for Alternate Queueing: Towards Traffic Management by PC-UNIX Based Routers. In *USENIX 1998 Annual Technical Conference*, pages 247–258, June 1998.

[6] K. Cho. Managing Traffic with ALTQ. In *USENIX 1999 Annual Technical Conference: FREENIX Track*, June 1999. `http://www.usenix.org/events/usenix99/technical_freenix.html`.

[7] D. D. Clark and W. Fang. Explicit Allocation of Best Effort Packet Delivery Service. *IEEE/ACM Transactions on Networking*, 6:362–373, August 1998.

[8] S. De Cnodder and K. Pauwels. Relative Delay Priorities in a Differentiated Services Network Architecture. Deliverable, Alcatel Corporate Research Center, 1999.

[9] S. De Cnodder, K. Pauwels, and O. Elloumi. A Rate Based RED Mechanism. In *10th International Workshop on Network and Operating System Support Digital Audio and Video – NOSSDAV 2000*, Chapel Hill, North Carolina, USA, June 2000.

[10] A. Demers, S. Keshav, and S. Shenker. Analysis and Simulation of a Fair Queuing Algorithm. *Internetworking : Research and Experience*, 1(1):3–26, 1990.

[11] C. Dovrolis and P. Ramanathan. A Case for Relative Differentiated Services and the Proportional Differentiation Model. In *IEEE Network Magazine*, September 1999.

[12] C. Dovrolis and P. Ramanathan. Proportional Differentiated Services, Part II : Loss Rate Differentiation and Packet Dropping. In *ACM SIGCOMM*, June 2000.

[13] C. Dovrolis, D. Stiliadis, and P. Ramanathan. Proportional Differentiated Services : Delay Differentiation and Packet Scheduling. In *ACM SIGCOMM*, September 1999.

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

[15] S. Golestani. A Self-Clocked Fair Queueing Scheme for Broadband Applications. In *IEEE INFOCOM '94*, pages 636–646, April 1994.

[16] J. Heinanen, F. Baker, W. Weiss, and J. Wroclawski. Assured Forwarding PHB Group. RFC 2597, June 1999.

[17] Information Sciences Institute. Internet Protocol - DARPA Internet Program Protocol Specification. RFC 0791, University of Southern California, September 1981.

[18] V. Jacobson, K. Nichols, and K. Poduri. En Expedited Forwarding PHB. RFC 2598, June 1999.

[19] G. Jennes, G. Leduc, and M. Tufail. *A Scheduler for Relative Delay Service Differentiation*, 2000.

[20] M. Mathis, J. Semke, J. Mahdavi, and T. Ott. The Macroscopic Behavior of the TCP Congestion Avoidance Algorithm. *Computer Communication Review*, 27(3):67–82, July 1997.

[21] K. McKusick, K. Bostic, M. Karels, and J. Quaterman. The Design and Implementation of the 4.4BSD Operating System. Addison – Wesley, 1996.

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

[23] A. M. Odlyzko. Paris Metros Pricing : The Minimalist Differentiated Services Solution. In *IEEE/IFIP International Workshop on Quality of Service*, June 1999.

[24] J. Padhye, V. Firoiy, D. Towsley, and J. Kurose. Modeling TCP Throughput: A Simple Model and its Empirical Validation, May 1998.

[25] A. K. Parekh and R. G. Gallager. A Generalized Processor Sharing Approach to Flow Control in Integrated Services Networks : The Single-Node Case. *IEEE/ACM Transaction on Networking*, 1(3):344–357, June 1993.

[26] The FreeBSD Documentation Project. FreeBSD HandBook, February 1999. `ftp://ftp.FreeBSD.org/pub/FreeBSD/doc`.

[27] T. Soetens, S. De Cnodder, and O. Elloumi. A Relative Bandwidth Differentiated Service for TCP Micro-flows. In *Proc. of Internet QoS IQ 2001*, Brisbane, Australia, May 2001.

[28] A. Tanenbaum. *Computer networks*. Prentice Hall, 1997.

[29] G. R. Wright and W. R. Stevens. TCP/IP Illustrated, volume 2 of *Professional Computing Series*. Addison – Wesley, 1995.