Daniele Ludovici

Performance Analysis of RR and FQ Algorithms in Reconfigurable Routers

Tesi di Laurea Specialistica

Università di Pisa

Anno Accademico 2005/2006

Università di Pisa
Facoltà di Ingegneria

Corso di Laurea Specialistica in Ingegneria Informatica

Performance Analysis of RR and FQ Algorithms in Reconfigurable Routers

Tesi di
Daniele Ludovici

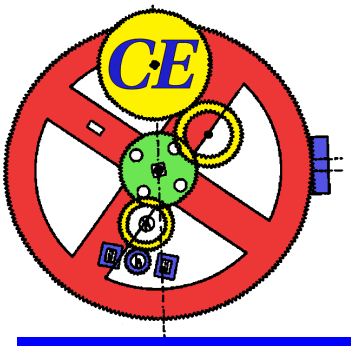
Relatori:

Prof. Luciano Lenzini .....

Prof. Cosimo Antonio Prete .....

Candidato:

Daniele Ludovici .....



Computer Engineering
Mekelweg 4,
2628 CD Delft
The Netherlands
http://ce.et.tudelft.nl/

2006

Performance Analysis of RR and FQ Algorithms in Reconfigurable Routers

Daniele Ludovici

Abstract

Currently, we are witnessing a trend in network routers to include reconfigurable hardware structures to provide flexibility at improved performance levels when compared to software-only implementations. This permits the run-time reconfiguration of the hardware resources, i.e., to change their functionality (for example, from one scheduling algorithm to another), to adapt to changing network scenarios. In particular, different scheduling algorithms are more efficient in handling a specific mix of incoming packet traffic in terms of various criteria (e.g., delay, jitter, throughput, and packet loss). Therefore, reconfigurable hardware is able to provide improved performance levels and to allow more efficient algorithms to be utilized when different incoming packet traffic patterns are encountered. This project investigates the possibilities to improve upon end-to-end delays, jitter, throughput, and packet loss by exploiting the availability of a flexible hardware structure such as an field-programmable gate array (FPGA). The aim of the project is to provide an overview on adaptive scheduling using reconfigurable hardware. Consequently, we investigate different scheduling algorithms that provide QoS provisioning for traffic streams that are sensitive to packet delay and jitter, e.g., mpeg video traffic. The investigation utilizes the NS-2 simulator for which we generate realistic network scenarios. Our approach is based on understanding which kind of traffic is passing in the network, and subsequently change the scheduling algorithm accordingly in the core router to meet specific performance requirements. The investigated scheduling algorithms are taken from two well-known families, i.e., Round Robin (RR) and Fair Queuing (FQ). Our investigation confirmed the idea on the behavior of the two investigated scheduling algorithm: WFQ outperforms WRR in terms of end-to-end delay, jitter and throughput but it is more expensive than it at a computational level. Nonetheless, it is possible to find a tradeoff between the required area in FPGA and the level of performance desired for a kind of stream.

TU Delft

Faculty of Electrical Engineering, Mathematics and Computer Science

to my family, for the infinite love and support they give me every day of my life

Contents

ACRONYMS

AF	Assured Forwarding
ASCII	American Standard Code for Information Interchange
ASIC	Application Specific Integrated Circuit
CBR	Costant Bitrate
CLB	Configurable Logic Block
CPLD	Complex Programmable Logic Device
DRR	Deficit Round Robin
DSP	Digital Signal Processor
Diffserv	Differentieted Services
FCFS	First Come First Served
FPGA	Field Programmable Gate Array
FQ	Fair Queuing
GPS	General Processor Sharing
IntServ	Integrated Services
NAM	Network Animator
NS-2	Network Simulator v2
OO	Object Oriented
QoS	Quality of Service
PGPS	Packet Generalized Processor Sharing
RED	Random Early Detection
RR	Round Robin
SLA	Service Level Agreement
SLS	Service Level Specification
SP	Strict Priority
TES	Transform Expand Sample
W^2FQ+	Worst Case Weighted Fair Queuing
WFQ	Weighted Fair Queuing
WRR	Weighted Round Robin

List of Figures

List of Tables

Acknowledgements

This work represents a very important step for my life and i would like to thank all the people that supported me during this period.
First of all, i would like to thank Stephan Wong for the valuable guidance through my work, for his infinite patience and the advice he suggested to me during the development of this thesis. Moreover, the great friendship he showed me from the beginning.
Thanks to Stamatis Vassiliadis for the opportunity he gave me to study in his group and the great confidence shown in all of our discussions.
I really thank all the people i have met in the CE group that have contributed to make this experience unforgettable.

Daniele Ludovici
Delft, The Netherlands
2006

Ringraziamenti

Quelle che state leggendo sono le ultime righe che scrivo per completare questo lavoro che è sbocciato nella terra dei tulipani, delle canape e delle bitterballen. È stata una grande avventura per la quale devo ringraziare in ordine sparso le persone che mi hanno accompagnato in questi mesi di soggiorno esterofilo: Giacomino, Sergio, Alessandro, Lecorbudar, Mikko, Dimitris, Arcilio, Yizhi, Maristella, Gino, Mattia, Maximilian, Roberto, Lars, Juhan, gli olandesi di cui non ricordo i nomi, la mia cara Bea, Gianni, Lo Maestro, Joaho, Daniel, Andrea, Roberto, Mauro, Christian, tutti i ragazzi del CE-group che mi hanno sempre fatto sentire a casa e tutte le altre centinaia di persone che ho conosciuto e di cui non mi ricordo in questo momento. I sette mesi olandesi mi hanno dato tanto ma forse mi hanno tolto anche qualcosa... qualcosa che mi ha sempre incoraggiato a non mollare mai e a cercare sempre il meglio per me, per questo non potrò che esserti per sempre grato, grazie Pamy. Prima dei tulipani c'è stata la torre pendente, Pisa mi rimarrà per sempre nel cuore grazie agli amici di casa Guelfi che hanno reso questi anni di permanenza un vivere in famiglia. Peppino e le nostre lunghe chiacchierate. Gabriele, tutta la pasta che m'ha fatto mangiare e i gol che per lui ho rifinito. Daniele e le nottate a vedere Smallville e altre serie TV insieme. Grazie a Stefano e Rocco B. fedeli compagni di corso e impagabili amici di strada. Grazie a Rocco V. per i suoi gol mangiati e la tagliata con la rucola. Grazie a Valerio, il suo saper fotografare e il suo passo felpato nel giocare a calcetto. Grazie ad Andrea, la sua pizza al pesto e il suo saper compilare. Grazie a Giuseppe (Re Artù) Arturi. Grazie a tutte quelle della casa "di la": Maddalena, Elisabetta, Giulia, Cristina, Olga, Anna, Marianeve, Sara, M.Elisa e tutte quelle che non hanno passato il casting ma avrebbero voluto. Grazie a Filippo e il suo folle modo di essermi amico. Grazie a Nino e la siora Baglini per tutta la comprensione dimostrataci nel sopportarci come vicini. Grazie a tutti quelli contro i quali ho inveito giocando a calcetto e che mi hanno lasciato il setto nasale al suo posto. Grazie ad Angelo, Antonio e Massimo. Grazie a Mad-house per le cene al crostino, Elisqui e il suo maledetto gatto, Gianfa e la sua logorroica dolcezza, Francesca e il suo sorridere sempre, Valeria che mi assomiglia in quanto essere panda femmina. Grazie a Gianluca che sopporta Elisqui. Grazie agli amici di Perl.it. Non ci sarebbe stata Pisa senza i 747 metri sul livello del mare più belli del mondo, grazie a tutti i miei amici di sempre, Diego e Sandro, Annalisa e Michela, Donato tuttofare, Verdura e Cilento ormai ballerini, Cervo e Cirillo, Zio Lauro, Carlo, Francesca, Tucio, i cugini Mario e Gianluca, Andrea, Simona e ora anche Alessandro :-), Giovanna, Celentano e Borra, McGuiver e Manganozzo, Piero e Ileana, Francesco che non mi è venuto a trovare e Desy, i cuginetti Luca, Marcolino e Marco, tutti gli zii che ho. Grazie a

Mauro per avermi fatto appassionare all'informatica e per esser l'amico unico che è. Grazie a tutti gli insegnanti che ho avuto e in particolare a quelli che mi hanno trasmesso la voglia di viaggiare. Grazie a Zio Spartaco a Zia Graziella. Grazie a Nonna Natalina e Zia Laura per l'immenso amore che avete per me. Un pensiero ai miei nonni che anche da lassù saranno tanto felici.
Grazie a mia madre, mio padre e mio fratello per avermi convinto ad inseguire i miei sogni sempre e comunque.

Daniele Ludovici
Delft, The Netherlands
2006

1 Introduction

Currently, we are witnessing a trend in network routers to include FPGA structures to provide flexibility in scheduler implementations at improved performance levels when compared to software-only implementations. This allows run-time reconfiguration of, e.g., scheduling algorithms utilized, to adapt to changing network scenarios. This thesis describes the project that investigate the possibilities to improve upon end-to-end delays, jitter, throughput, and packet loss by exploiting the availability of a flexible hardware structure such as an FPGA.

This chapter introduces the main argument of this thesis. First of all, we give an overview about the concept of Quality of Service *(QoS), discussing its differences with the* best-effort *approach and justifying the motivations that lead to the development of a new paradigm for the Internet. Subsequently, we introduce the most important metrics for the QoS, i.e., end-to-end delay, jitter, throughput, and packet loss. Finally, we point out the objectives and the motivations of the project described in this thesis.*

1.1 Quality of Service

Internet and computer networks are becoming part of our daily life, a change that affects universities, commercial organizations, and consumers. These networks support a huge amount of different applications like e-mail, web, audio, and video services, including television on demand and file transfer. Furthermore, the requirements for these applications are growing quickly, and probably this fact will continue in the near future. Recently, the provision of service guarantees has become an important issue, which was caused by both the heterogeneity of application requirements and the growing commercialization of the Internet. It is not possible to sell products as television-on-demand or audio streaming with a network infrastructure that does not guarantee on the desired service level.

Currently, the Internet is based on the best-effort model, which means “as much as possible, as soon as possible”. According to this definition, each packet has the same expectation of treatment as it transits a network. Even if this could be enough for traditional Internet applications, such as web, e-mail, news groups, etc., new multimedia applications and services create a new demand to bring different levels of service to packet traffic. The main reason is that, depending on a service, and if possible on the desired level of this service, the traffic characteristics and its requirements for network transport functionality may vary. For instance,

some of these applications are telnet sessions that usually generate low traffic but have high interactivity requirements; some are file transfer protocol (FTP) sessions with bursts of kilobytes or even megabytes, for which the throughput is a fundamental aspect compared with the interactivity; some are hypertext transfer protocol (HTTP) transactions that open a transport connection to transmit a handful of packets but do not need an high bandwidth utilization and some are audio or video streams that require a good bandwidth utilization, a low delay as well as a low jitter. If the Internet handles all data in the same way, then it can result in unacceptable, if not completely unusable, service. For example, if video and file transfer traffic are mixed together in the same part of the network, then the file transfer can experience congestion and packet losses due to the video applications that send data at the constant rate. In turn, the video traffic will not slow down, but will experience losses and poor voice quality due to the presence of the bursty file transfer operations. Therefore, the enhanced Internet architecture should provide different service classes to indicate the treatment of individual packets and flows and to allocate the suitable resources for all of them.

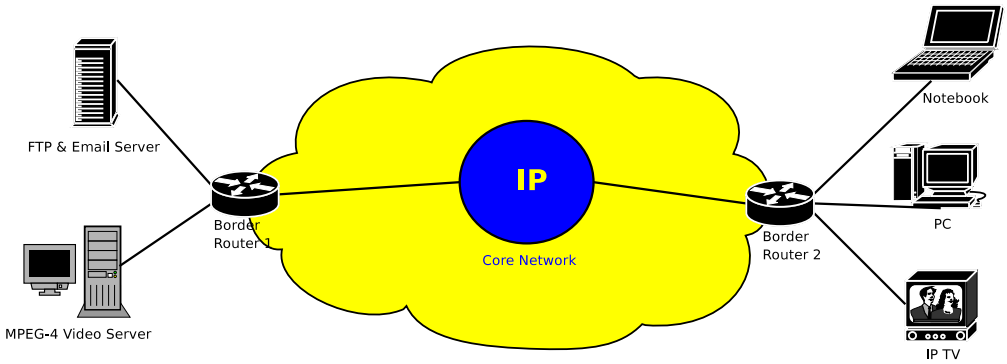


Figure 1.1: A QoS-aware Network Model

The objective of network QoS is to quantify the treatment that a particular packet can expect. A good QoS architecture should provide a way to specify performance objectives for different classes of packets, as well as suggestions on how to meet those performance goals. It has to be clear that QoS cannot create additional bandwidth. When some packets get better treatment, other packets will get worse treatment. Therefore, the objective of QoS is to distribute resources in such a way that all the performance objectives are met.

The QoS stems from the concept that some traffic is more important compared to other and should be treated accordingly. In addition, economical reasons exist such as the Internet has become mission-critical for commercial companies. In an environment, a provider can offer services with specific performance level associating them with a cost consequently, it is possible for the provider to tie the provided expected performance level (or quality) with a price for the service.

From the customer's point of view (either a provider, a user, or another organization), the first step towards the QoS is the Service Level Agreement (SLA), which is negotiated with a provider. The SLA defines the QoS requirements, the anticipated load, actions to take if the load increases the negotiated value for the service level, pricing etc. Since the SLA includes rules and actions in the human readable form, it has to be translated into the machine readable representation. For these purposes, the SLA is partitioned into several documents. The Service Level Objectives (SLO) specifies metrics and operation information to enforce and monitor the SLA. The Service Level Specification (SLS) specifies the handling of a customers traffic by a service provider.

To characterize the QoS requirements and actions in the SLS, a provider and a customer must specify them with a set of well-known parameters, or performance metrics, so that a provider can translate them into the router configuration. The fundamental parameters are throughput, delay, jitter, and packet loss [16].

1.1.1 QoS parameters

In this section, the most important QoS parameters are introduced. A definition is provided with a short description of the meaning of the parameter.

Throughput specifies the amount of bytes (or bits) that an application can send during a time unit without losses. It is one of the most important parameters because most applications include it in the set of their QoS requirements. It should be noted that the throughput stands for the long-term rate of an application. Due to the packet-based nature of most networks, the short-term rate may differ from the long-term value. Therefore, it is usually the case that the throughput refers to the average rate of an application. Consequently, one can use other parameters such as the maximum or the rate and the minimum rate.

Packet delay is a fundamental characteristic of a packet-switched network and it represents the delay required to deliver a packet from a source to destination. It is also called *end-to-end* delay. Each packet inside a network is routed to its destination via a sequence of intermediate nodes. Therefore, the *end-to-end* delay is the sum of the delays experienced at each hop on the way to the destination. It is possible to think about such delay as consisting of two components, a fixed one which includes the transmission delay at a node and propagation delay on the link to the next node, and a variable component which includes the processing and queueing delays at the node.

Jitter specifies the delay variation between two consecutive packets. It is an important parameter for the interactive applications, such as on-line audio and video conversations. Since data exchange between two applications involves sending a significant number of packets, it is often the case that jitter specifies the maximum delay variation observed between the two consecutive packets. However, one can also use a smoothing equation to obtain some mean value over the sequence

of packets. Regardless of the interpretation, ideally the jitter should equal to zero because the bigger its value is, the larger the buffer of a receiving application must be to compensate for the delay variations between the packets.

Packet loss, as its name indicates, characterizes the number of dropped packets during transmission. This parameter is critical for those applications that perform guaranteed data delivery because every time a router drops a packet, a sending application has to retransmit it, which results in ineffective bandwidth utilization. It is also important for some real-time applications since packet drops reduce the quality of transmitted video and/or audio data. Since the number of dropped packets depends on the duration of a session, the packet loss is expressed usually as a ratio of the number of dropped packets to the overall number of packets [16].

Fairness: considering an allocation problem: i users, a vector x where the i-th coordinates represent the allocation for user i, i.e., the rate to which the user i can emit data. A feasible allocation of rates x is "max-min fair" if and only if an increase of any rate within the domain of feasible allocations must be at the cost of a decrease of some already smaller rate. Depending on the problem, a max-min fair allocation may or may not exist. However, if it exists, it is unique. The name "max-min" stems from the idea that it is forbidden to decrease the share of sources that have small values, therefore, in some sense, the flows with small values are privileged.

1.2 Goals & Methodology

How can reconfigurable hardware structures, e.g., FPGAs, be efficiently used in routers to improve their performance in terms of, for example, end-to-end delay, jitter, and throughput for different types of network traffic?

Certainly, knowledge of the traffic characteristics in a certain network scenario may lead to different choices in the system configuration. This knowledge is not always available a priori, therefore the approach is to dynamically understand which kind of traffic is travelling inside the backbone and changing the scheduling algorithm accordingly. This choice can bring a performance improvement in terms of different network metrics. The problem of using an FPGA can be viewed also from the efficiency point of view, that is, using different scheduler algorithms takes more or less space in the hardware device depending on the choice. An investigation on the different possibilities adopting one choice or the other is necessary and also a tradeoff between them can be found. It is possible to think of using this available space on the device with the possibility to handle more stream sessions simultaneously or try to improve the performance of the cheaper algorithm using a different approach in the traffic management.

To reach these goals, this work provides an overview on adaptive scheduling using reconfigurable hardware, giving some guidelines which would help a network

administrator in choosing the configuration which best fits to the requirements of his scenario. Consequently, we investigate different scheduling algorithms that provide QoS guarantees for traffic streams that are sensitive to packet delay and jitter, e.g., MPEG video traffic. The investigation has been carried out with the NS-2 simulator [18] which represents a standard de facto for research in the networking environment. The main steps of this work are:

- add a real-time traffic source in NS-2 to evaluate the performance of our approach in a realistic scenario;
- add an experimental support for the run-time reconfiguration of the scheduling algorithm in the simulator module implementing the scheduler;
- add the Weighted Fair Queuing scheduler to the simulator and test it;
- evaluate the trend of QoS metrics with or without the modification;
- evaluate the benefits that the reconfiguration can give in terms of saved FPGA area.

For the first evaluation, the approach is mainly based on understanding which kind of traffic is passing in the network and change the scheduling algorithm in the core router to meet certain performance requirements. For the WFQ scheduler testing, a test suite is constructed composed of a particular scenario and an ad-hoc script that has been created. To evaluate the benefits in terms of saved FPGA area, a comparison between two schedulers was performed. The investigated scheduling algorithms are taken from two well-known families, i.e., Round Robin (RR) and Fair Queuing (FQ).

1.3 Overview

The remainder of the thesis is organized as follow: Chapter 2 gives an overview about the two big algorithm families: *round-robin* and *fair queuing* providing a classification of scheduling algorithms and giving a description of them, introducing which ones we are going to use in our simulations. Chapter 3 describes the simulation environment as well as the differentiated services architecture for the IP QoS; in this chapter the MPEG4 model is introduced and its support in the simulator described together with a cost analysis of the implementation in FPGA of the chosen algorithms. Chapter 4 presents the *monitor agent* developed to trace the results in the simulator, also the modifications that are needed to support the reconfigurability are described; at the end of the chapter, the WFQ implementation in NS-2, an algorithm that is not available with the standard simulator distribution and that has been added, is presented. In Chapter 5, the methodology to evaluate the results of the experiments is described, and the results of these simulations are

presented and analyzed. Chapter 6 summarizes the work with the last reflections and gives some ideas for future investigations.

2 Scheduling Algorithms

In this chapter, an overview of the scheduling algorithms available in the network simulator will be provided, and the choice of two of these strategies is explained basing on some of their characteristics.

The selection of an appropriate scheduling algorithm is a key point to build a network environment with QoS capabilities. Considering a simple case, a service provider has to share the output bandwidth between all data flows such that each flow obtains a fair portion of bandwidth resources. In this approach, we are assuming that each single flow has the same requirements. However, due to the diversity of the existent applications, a data flow may require a certain minimum amount of the output bandwidth, and this can lead to unequal bandwidth allocation between flows. Therefore, a service provider has to choose the correct disciplines to ensure that the requirements of all the data flows are met.

2.1 Classification of Scheduling Algorithms

A service discipline can be classified as either work-conserving or non-work-conserving [26]. With a *work-conserving* discipline, a scheduler is never idle when there is a packet to send. With a *non-work-conserving* discipline, each packet is assigned an eligible time, this parameter represent the time for the packet departure and if no packet is eligible, none will be transmitted even when a scheduler is idle; with such kind of disciplines, a scheduler can be idle at any time, in an effort to smooth out the traffic pattern [4]. Both these classes have drawbacks and advantages. The work-conserving disciplines are suitable for those environments in which applications can adjust their transmission rates and can react to the packet losses. Indeed, if there are available bandwidth resources, the application can start to send more data achieving the higher throughput and the better network utilization. If later a congestion occurs, some number of packets will be dropped which will signal the sending application to slow down the data transmission. Though the work-conserving disciplines allow to completely utilize the output bandwidth, they alter the traffic profile as packets move from one node to another. Usually the traffic profile represents the temporal properties of a traffic stream and it is represented in terms of rate and burst size. As a result, the provision of the end-to-end guarantees, and especially the delay guarantees, becomes an art that involves many network management solutions. As opposed to this, the non-work-conserving disciplines do not change the traffic profile because the

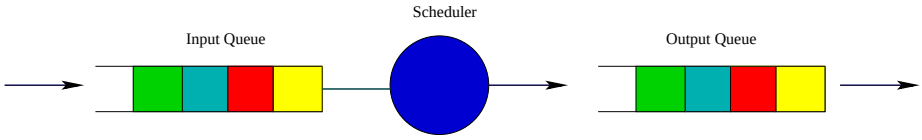


Figure 2.1: FIFO Scheduler

server delays the packet transmission to respect this profile; this behavior simplifies the provision of the end-to-end guarantees. The non-work-conserving disciplines eliminate traffic distortions by delaying packets and, thus, preserving the traffic profile. However, as mentioned above, some bandwidth resources remain unutilized. At the moment, most manufactures of the telecommunication equipment rely upon the work-conserving disciplines. The main reason is that they allow to utilize bandwidth resources more efficiently. Furthermore, two QoS frameworks proposed by the IETF rely upon the work-conserving schedulers. Hence, we will also focus on this class of scheduling disciplines [16].

2.2 FIFO (First In First Out)

This is the classical scheduling algorithm deployed in the best-effort approach in the Internet and it is also know as *FCFS* (First Come First Served, Figure 2.1). With this algorithm, the data are sent in the same order in which they are received. The complexity of this approach is very low and it is also very efficient to implement in hardware. It is a work-conserving algorithm and because its characteristics it has been adopted by a large number of network architectures. Unfortunately, FIFO has several limitations:

- it does not provide fairness;
- the support to control congestion is limited.

This kind of scheduler is not suitable for stream like *multimedia traffic*, because it is not able to isolate real-time sessions from best effort ones. In this case, there is no way to guarantee a specific level of quality to real-time sessions.

2.3 Priority Scheduling

Priority Queuing is one of the first attempts utilized to provide different services and it is able to isolate the sessions among these services.

The concept is as follows: the traffic is inserted into different queues that are served according to their priority. Highest priority queues are served as soon as they have packets. This mechanism can bring to *starvation* if highest priority classes have a large amount of traffic and therefore it is not fair.

This mechanism is work-conserving and it is particularly interesting when the network has a small amount of high priority traffic (for example real-time sessions), because the performance of the best effort traffic is not worsened too much.

The highest priority class has the entire output link bandwidth available. The class that follows has the entire link bandwidth decreased by the amount used by the first class, and so on. Therefore, traffic within each class is influenced only by the traffic sent by the classes that have a higher priority than itself. This algorithm

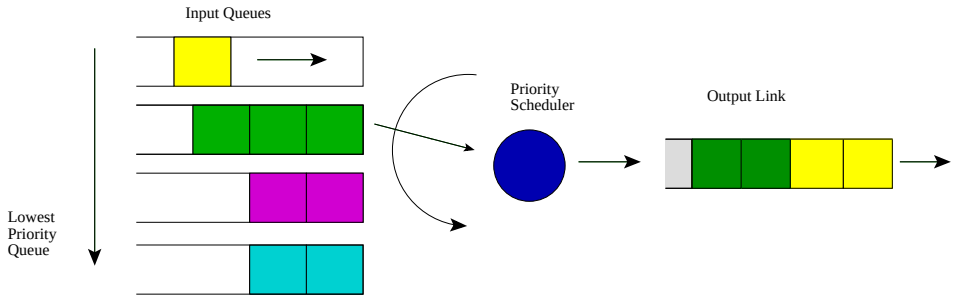


Figure 2.2: Priority Queuing Scheduler

is not very robust because it requires an admission control in order to ensure that the highest priority class does not receive too much bandwidth. Moreover, there are several situations where the service guaranteed by priority queuing is much better than the service required by the sessions. In this case, the priority queuing is not able to delay a high priority packet in order to increase the service pattern for other sessions; for these reasons priority queuing is not a good choice for a network mixing best effort and real-time traffic [1].

2.4 Fair Scheduling Algorithms

The Fair Queuing algorithm was originally proposed in [12] by Nagle in order to solve the problem of malicious or erratic TCP implementations. The goal of the algorithm is to guarantee each session with a fair share of network resources, even though some sessions are transmitting at a much higher rate than the allocated one. This class of algorithm is defined fair, because it allows the fair sharing of the bandwidth among all the users [14].

2.4.1 Round Robin

Round Robin (RR) scheduling represents the simplest solution for the controlled bandwidth sharing. Using this scheduling algorithm, a packet is dequeued in a so-called *round-robin manner* and outputs one packet from a queue. Of course, it is not possible to provide any QoS capabilities with such a scheme. Nagle's idea was to maintain, at each node, a separate queue for each traffic sources and to

transmit packets at the head of queues in round-robin order. Empty queues are skipped, so that, given n active sources, the overall effect is to allow each source to send one packet every n packets transmitted on the link because each source has to wait a complete round to be served the next time.

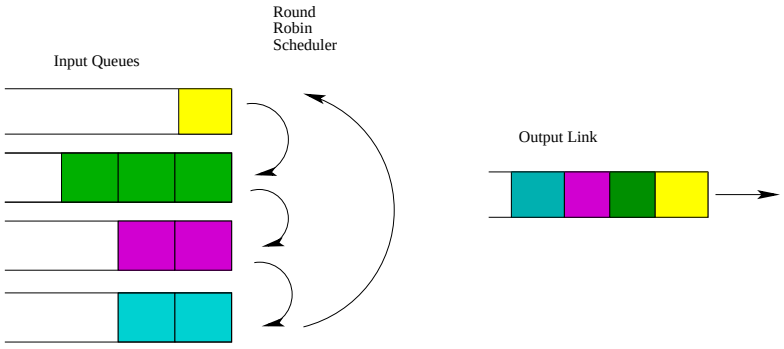


Figure 2.3: Round Robin Scheduler

Having n queues with n users, each of them gets exactly $\frac{1}{n}$ of the total link bandwidth. This approach suffers from several limitations:

- it ignores packet lengths, so it works well (with respect to fairness) just when the average packet size over the flow duration is the same for all flows;
- the performance depends on the arrival pattern; a packet arriving at an empty queue, just after its turn in the round-robin, must wait until the other $(n-1)$ queues have been served;
- it requires a-priori knowledge of the number of sessions (because we put the traffic of each session in a separate queue) but in most cases this is not realistic.

2.4.1.1 WRR - Weighted Round Robin

The WRR scheduler works in a cyclic manner, serving consequently the input queues. The weight is a variable that indicates how many packets have to be sent for each cycle from each queue. If a queue has fewer packets than the value of the weight, the WRR scheduler outputs the existent number of packets and begins to serve the next queue. The WRR scheduler does not take the size of the transmitted packets into account. As a result, it is difficult to predict the actual bandwidth that each queue obtains. In other words, it is difficult to use only the weight values as the means to specify the amount of the output bandwidth. Suppose that w_i is the value of the weight associated with the ith queue. If L_i is the mean packet size of the ith input queue, then $w_i L_i$ bytes of data are sent during each cycle on average. If there are m input queues, then it is easy to show that the average

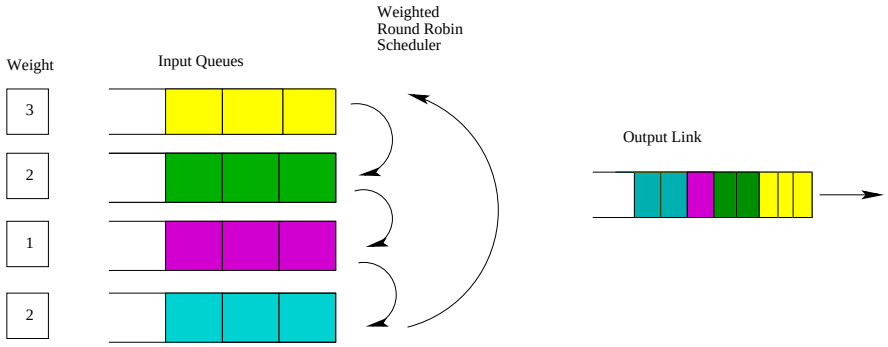


Figure 2.4: WRR Scheduler

amount of data transmitted from all queues during one cycle can be approximated by:

$$\sum_{i=1}^{m} w_i L_i \tag{2.1}$$

Expression 2.1 is referred to as the *frame size*. Taking the mean packet size and weights of all queues into account, it is possible to approximate the output bandwidth for the given kth queue:

$$\frac{w_k L_k}{\sum_i w_i L_i} B, \quad k \in (1, m) \tag{2.2}$$

where B specifies the output bandwidth of an interface, on which a router implements WRR. By approximating the average bandwidth of each queue, it is possible to provide the QoS guarantees [16].

2.4.1.2 DRR - Deficit Round Robin

Deficit Round Robin is one modification of WRR to improve the fairness. DRR can guarantee fairness in terms of throughput. In DRR, the packet scheduler maintains a state variable called *Deficit-Counter* to control the amount of dequeued traffic precisely. A quantity called Quantum is added to *Deficit-Counter* in each round. If the length of the packet at the top of the queue is less than *Deficit-Counter*, the packet scheduler can dequeue the packet and then subtracts the packet length from *Deficit-Counter*. Otherwise, it accesses the next queue.

Although DRR requires only $O(1)$ processing work per packet in order to guarantee bandwidth, it must calculate the length of every packet and the calculation of the packet length takes computational resources of the router [25].

2.4.2 Fair Queuing algorithms

The problem of fair network resource allocation has led to the development of a class of algorithms that provide tight end-to-end delay bounds and efficient re-

source utilization. These algorithms try to approximate the ideal behavior of the Generalized Processor Sharing algorithm (GPS), which is described in the following section.

2.4.2.1 GPS - Generalized Processor Sharing

GPS [13] has been proposed by Parekh and Gallager in 1993, it is based on a fluid model, so it assumes that the input traffic is infinitely divisible and that all sessions can be served at the same time. This is a work-conserving server and it guarantees each session to receive a *service rate* g_i of at least:

$$g_i = \frac{\theta_i}{\sum_{j=1}^{N} \theta_j} r \tag{2.3}$$

where r is the *server rate* and θ_i is the *weight* for the i-th session.

GPS treats each packet as infinitely divisible, so the scheduler picks a small piece of data from each session and transmits it on the output link. Because this assumption, GPS can transmit a *fluid flow* instead of bits. For example, having two sessions m and n, each of which with weight 1, an algorithm like *Bit by Bit Round Robin* transmits one bit of the first session and a bit of the second session. GPS, instead, always uses half bandwidth transmitting session m data and the other half transmitting session n data. An important feature of GPS is that it

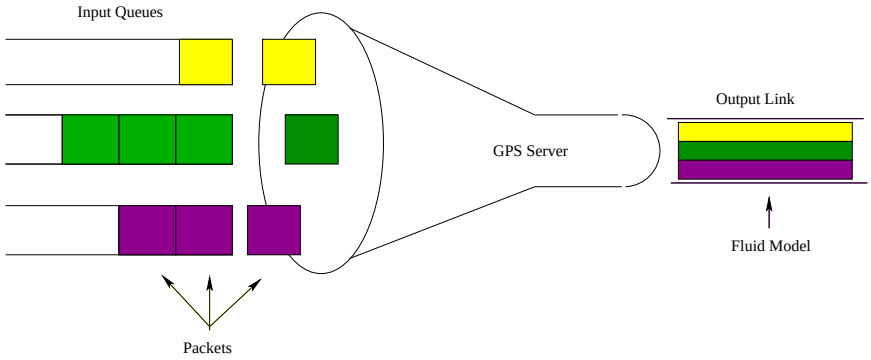


Figure 2.5: Graphical representation of a GPS Server

is possible to compute a worst-case delay bound for leaky bucket[1] constrained sessions. However, GPS is impractical because it assumes the server can serve all the sessions simultaneously and the traffic is infinitely divisible. Therefore, it needs to be approximated by a packet algorithm. In the next section we are going to introduce two of these: WFQ and WF^{2}Q+.

[1]the leaky-bucket implementation is used to control the rate at which traffic is sent to the network. A leaky bucket provides a mechanism by which bursty traffic can be shaped to present a steady stream of traffic to the network, as opposed to traffic with erratic bursts of low-volume and high-volume flows [20].

2.4.2.2 WFQ - Weighted Fair Queuing

One of the first scheduling algorithm proposed to approximate GPS was WFQ, it schedules packets according to their arrival time, size, and the associated weight. Upon the arrival of a new packet, a "virtual finish time" is calculated and the packet is scheduled for departure in the right order with respect to the other packets. This "virtual finish time" represents the time at which the same packet would finish to be served in the GPS system. Then, WFQ outputs packets in the ascending order of the virtual finish time. Such an approach enables the sharing of resources between service classes in a fair and predictable way. Furthermore, it is possible to estimate the bandwidth allocation and the worst-case delay performance, which makes the use of the WFQ discipline very attractive for the provision of QoS, and especially for the provision of the end-to-end guarantees. Suppose that B is the total throughput of an output link on which a router implements WFQ. If all sessions of the WFQ scheduler are active, then each class receives a portion of the total bandwidth, which is determined by its weight w_i and is equal to $w_i B$. B is the available bandwidth. Hence, to simplify the expression, we assume that it holds for all weights w_i that

$$\sum_i w_i = 1, \quad w_i \in (0, 1) \tag{2.4}$$

By knowing the QoS requirements of all data flows, we can find values for w_i such that all the QoS guarantees are ensured.

2.4.2.3 QoS requirements

It is possible to associate each service class with a proper weight w that indicates the allocated bandwidth. Having N_i active flows within the i-th class, then each flow gets a bandwidth that can be approximated by:

$$B_i^f = \frac{w_i B}{N_i} \tag{2.5}$$

B_i^f can be viewed as one of the QoS parameters that specifies the required bandwidth of a flow belonging to the i-th service class. Thus, the minimum value of the weight, which provides the necessary amount of bandwidth for every flow, is given by

$$w_i \geq N_i \frac{B_i^f}{B} \tag{2.6}$$

The inequality states that a provider can allocate more resources than necessary. Therefore, if the network has free bandwidth resources, then a provider can allocate, either explicitly or implicitly, more bandwidth to a service class.

Due to buffering, scheduling, and transmission of packets, the size of a router's queue varies all the time. On the other hand, the length of a queue in a routing

node has an impact on the queuing delay and on the overall end-to-end delay of a packet. It can be shown that under WFQ the worst-case queuing delay is given by the following expression, where L^{max} denotes the maximum packet size:

$$D = \frac{\sigma}{\rho} + \frac{L^{max}}{B} \tag{2.7}$$

In (2.7), it has been assumed that each incoming flow is regulated by the *token bucket* [2] with the bucket depth σ and the token rate ρ. Parameters σ and ρ can be viewed as the maximum burst size and the long term bounding rate respectively.

2.4.2.4 WF^{2}Q+ - Worst Case Weighted Fair Queuing

WF^{2}Q+ is less complex to implement compared with WFQ because it uses a system virtual function that is calculated from the packet system itself and not from GPS, indeed, another advantage is that, there is no need to stamp each packet with its start and finish time, instead it is sufficient to have just one start and finish time for each connection. WF^{2}Q+ approximates GPS more closely, its delay is better than WFQ in a lot of cases, but not always. It has been shown that WF^{2}Q+ does not always outperform WFQ for real-time sources [22]. WFQ presents a better delay than WF^{2}Q+ in situations with bursty traffic with variable packet size, which are typical with real-time sources, of course the augment of end-to-end delay for multimedia traffic, gives better delay guarantees for all other source non-real time traffic in the network. For this reason WFQ has been chosen for our investigation. However, for completeness we introduce the main idea of WF^{2}Q+ in this section.

In WF^{2}Q+ the rule to pick up a packet from a queue is lightly modified by the WFQ rule:

WFQ: the packet having the lower exiting time according to the correspondent GPS system, will be the first one to exit under the WFQ scheduler.

WF^{2}Q+: the packet having the lower exiting time according to the correspondent GPS system, and that has already started its service under the GPS scheduler, will be the first one to exit under WF^{2}Q+ scheduler.

This difference can be understood looking at the Figure 2.6. At time 1 the situation is the following:

- WFQ: the first packet finishing its service under GPS is $Pkt_{1,1}$ and it will be scheduled in WFQ;
- WF^{2}Q+: the packets having already started their service under GPS are $Pkt_{0,2}$, $Pkt_{0,3}$, $Pkt_{0,4}$ and $Pkt_{0,5}$. The first one that will finish is $Pkt_{0,2}$ (but another one should also be scheduled in this example, because they are

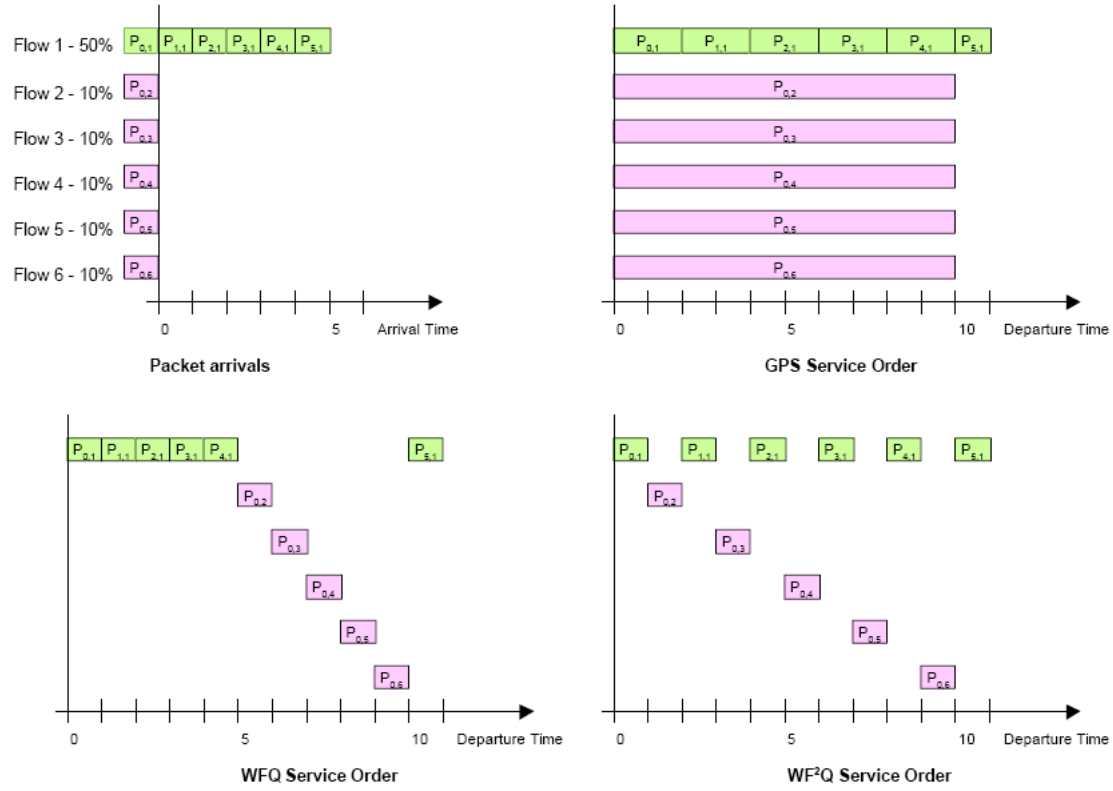


Figure 2.6: Different packet output between WFQ and WF^{2}Q+ (picked from [14])

finishing all together), so this will be the next packet being scheduled by WF^{2}Q+.

WF^{2}Q+ represents an improvement from the implementation point of view compared with WFQ because it exploits a different virtual function to take into account the virtual time, and it is also an improvement for the complexity viewpoint ($O(N)$ against $O(log(N))$).

2.5 Summary

In this chapter, the *round-robin* and the *fair queuing* families of algorithms have been presented. A classification of these scheduling disciplines has been furnished and each algorithm has been described. The *FIFO* and *priority* scheduler are unsuitable to provide fairness as well as QoS guarantees. The priority scheduler can also bring to the starvation if the high priority classes have a large amount of traffic to send. RR, WRR, and DRR have been described as representative of the *round-robin* families. These algorithms are simple to implement but they do not provide QoS guarantees and in certain situation they cannot share correctly the

bandwidth to the competing streams. The *fair queuing* family has been introduced presenting the theoretical model of General Processor Sharing (GPS) from which all the approximations are based on. The most well-known of this algorithm approximation have been presented: weighted fair queueing (WFQ) and worst case weighted fair queuing (WF^2Q+). They are more complex to implement compared with the *round robin* algorithms.

Simulation Setup

3

This chapter introduces the NS-2 Network Simulator; then, we discuss the Diffserv support presenting in the simulator through the Nortel Diffserv module [17]. Another section introduces the MPEG4 model that has been used in the simulations. The last part of the chapter is dedicated to the motivation leading to the use of an FPGA and the cost analysis of WFQ and WRR in hardware.

3.1 NS-2 - The Network Simulator

3.1.1 Overview

NS-2 is an event driven network simulator developed at UC Berkeley that simulates several IP networks. It implements network protocols such as TCP and UDP, traffic source behavior such as FTP, Telnet, Web, CBR and VBR, router queue management mechanism such as Drop Tail, RED and CBQ, routing algorithms such as Dijkstra, and more. NS-2 also implements multicasting and some of the MAC layer protocols for LAN simulations. The NS-2 project is now a part of the VINT project that develops tools for simulation results display, analysis and converters for network topologies. NS-2 is written in C++ and OTcl, Tcl is a script language with object-oriented extensions developed at MIT[1]. The core of the simulator is written in C++ instead the simulation setup is written in Tcl. To add a component in the simulator, as a scheduler or a network object, the C++ code needs to be modified while to setup and run a simulation, a Tcl scenario has to be created.

[1]Massachusetts Institute of Technology

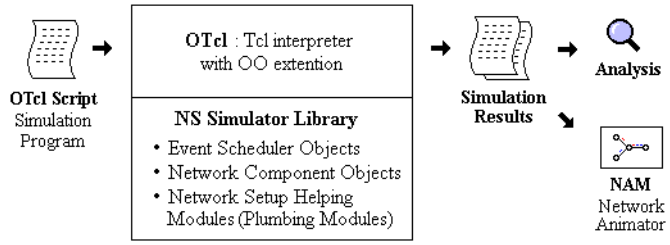


Figure 3.1: Simplified User's View of NS-2

As depicted in Figure 3.1, in a simplified user's view, NS-2 is OTcl script interpreter, that has a set of simulation libraries. Within these, there are network setup (plumbing) module libraries, they are implemented as member functions of the base simulator object. In other words, to use NS-2, a program in OTcl script language has to be written.
To create and run a network simulation:

- write an OTcl script initializing an event scheduler;
- set up the network topology using network objects and library functions;
- define traffic paths:
  - set traffic source and destination;
  - define time to start sending traffic and when stop;
  - define characteristics between links connecting nodes;
- set a method to trace the results;
- start the simulator object.

The term "plumbing" is used for a network setup, because setting up a network can be viewed as plumbing possible data paths among network objects by setting the "neighbor" pointer of an object to the address of an appropriate object. If a new network object has to be written, it has to be created either by writing a new object or by making a compound object from the object library, and plumb the data path through the object. This might be a complicated job, but the plumbing OTcl modules actually make it very easy. The power of NS-2 comes from this plumbing.

Another major component of NS-2 beside network objects is the event scheduler. An event in NS-2 is a packet ID that is unique for a packet with scheduled time and a pointer to an object that handles the event. In NS-2, an event scheduler keeps track of simulation time and fires all the events in the event queue scheduled for the current time by invoking appropriate network components, which usually are the ones who issued the events, and let them do the appropriate action associated with packet pointed by the event. Network components communicate with one another passing packets, however this does not consume actual simulation time. All the network components that need to spend some simulation time handling a packet (i.e., need a delay) use the event scheduler by issuing an event for the packet and waiting for the event to be fired to itself before doing further action handling the packet. For example, a network switch component that simulates a switch with 20 microseconds of switching delay issues an event for a packet to be switched to the scheduler as an event 20 microsecond later. The scheduler after 20

microsecond dequeues the event and fires it to the switch component, which then passes the packet to an appropriate output link component.

Another use of an event scheduler is timer. For example, TCP needs a timer to keep track of a packet transmission time out for retransmission (transmission of a packet with the same TCP packet number but different NS-2 packet ID). Timers use event schedulers in a similar manner that delay does. The only difference is that the timer measures a time value associated with a packet and does an appropriate action related to that packet after a certain time goes by, and does not simulate a delay.

NS-2 is written not only in OTcl but in C++ too. For efficiency reason, NS-2 separates the data path implementation from control path implementations. In order to reduce packet and event processing time (not simulation time), the event scheduler and the basic network component objects in the data path are written and compiled using C++. These compiled objects are made available to the OTcl interpreter through an OTcl linkage that creates a matching OTcl object for each of the C++ objects, and makes the control functions and the configurable variables specified by the C++ object act as member functions and member variables of the corresponding OTcl object. In this way, the controls of the C++ objects are given to OTcl. It is also possible to add member functions and variables to a C++ object. The objects in C++ that do not need to be controlled in a simulation or internally used by another object do not need to be linked to OTcl. Likewise, an object (not in the data path) can be entirely implemented in OTcl. Figure 3.2 shows an object hierarchy example in C++ and OTcl. One thing to note in the figure is that for C++ objects that have an OTcl linkage forming a hierarchy, there is a matching OTcl object hierarchy very similar to that of C++.

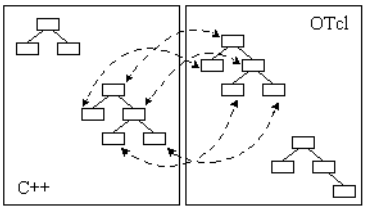


Figure 3.2: C++ and OTcl: The Duality

Figure 3.3 shows the general architecture of NS-2. In this figure a general user (not an NS-2 developer) can be thought of standing at the left bottom corner, designing and running simulations in Tcl using the simulator objects in the OTcl library. The event schedulers and most of the network components are implemented in C++ and available to OTcl through an OTcl linkage that is implemented using tcl. The whole thing together makes NS-2, which is a OO (object-oriented) extended Tcl interpreter with network simulator libraries.

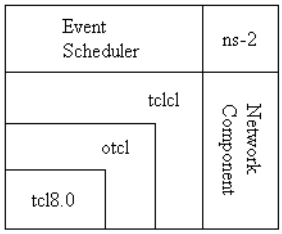


Figure 3.3: Architectural View of NS-2

3.2 Getting results in NS-2

This section explains how to obtain NS-2 simulation results. As shown in Figure 3.1, when a simulation is finished, NS-2 produces one or more text-based output files that contain detailed simulation data, if specified to do so in the input Tcl (or more specifically, OTcl) script. The data can be used for simulation analysis or as an input to a graphical simulation display tool called Network Animator (NAM) that is developed as a part of VINT project [18]. NAM (Figure 3.4) has a nice graphical user interface similar to that of a CD player (play, fast forward, rewind, pause and so on), and also has a display speed controller. Furthermore, it can graphically present information such as throughput and number of packet drops at each link, although the graphical information cannot be used for accurate simulation analysis. As we'll see later in Section 4.1, we have developed a module

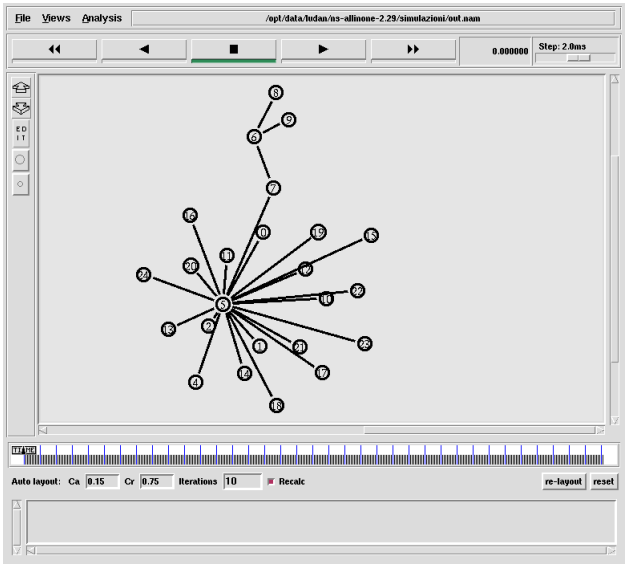


Figure 3.4: NAM: Network Animator

to keep track of some metrics without have to use the NS-2 tracefile, this solution is useful because with a long time simulation these tracefiles would generate a big

huge amount of files with a lot of data that are not interesting for our investigation.

The next section presents the module that we have used to differentiate the traffic and an overview of Diffserv Architecture.

3.3 Differentiated Services Architecture for IP QoS

In the traditional IP networks the service offered to the end-user is best-effort ("as much as possible as soon as possible"); in this kind of model, every user packets compete equally for network resources. This model doesn't give support to multimedia traffic and mission-critical applications, because it doesn't permit to differentiate the traffic. Much more attention has been placed on developing IP Quality of Service (QoS), which allows network operators to differentiate the traffic basing on different level of treatment [17].

Differentiated Services, or DiffServ, is an IP QoS architecture based on packet marking that allows packets to be prioritized according to user requirements. A scheme known as Assured Forwarding (AF) has been proposed as a potential user of Diffserv. Assured Forwarding provides differential treatment of traffic by discarding lower priority packets during congestion times. Although the Assured Forwarding mechanism does not explicitly require a particular queue type, it is suited for RED (Random Early Detection) [17].

3.3.1 Queue Management

When the buffer in a queue is becoming full, the *Queue Management* algorithm is used to decide which packet has to be dropped. This algorithm works on a single queue of the network node, if there are more queues on that node, more instances of a queue management are needed. It is also possible to have, within the single node, a queue treated with *RED* management and another one treated with *Drop Tail*.

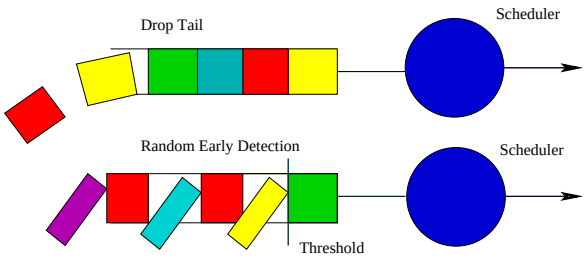


Figure 3.5: Different Queue Management mechanisms

3.3.2 Drop Tail

When there is no more space on the buffer, the last arrived packets are dropped. This is the simplest approach called *Drop Tail*. The available space in the queue depends on the buffering mechanism deployed in that network node: different queue management could be employed for each queue.

This algorithm suffers from limitation, because the packets are discarded when the queue is full and the network utilization varies a lot. Supposing to have a TCP source, experimenting a certain number of losses, the two agents decrease their sending rate. With this behavior, the network oscillates between high traffic and low traffic and it is not able to create an equilibrium. Moreover there can be malicious effects due to the source synchronization, where one source does not experiment any drop and another does [14]. In our simulations, this kind of queue has been utilized on the links connecting the edge routers with the sources and the destinations as shown in Figure 3.6, these links are green; instead the RED queue has been employed in the links between edge and core router and in the figure these are red.

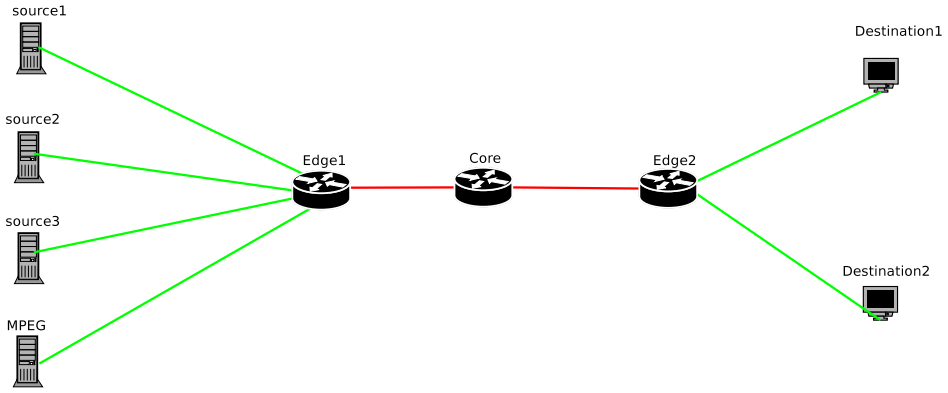


Figure 3.6: Scenario with different kind of queues

3.3.3 An Overview of RED

RED (Random Early Detection) is a congestion avoidance algorithm that can be implemented in routers. The basic queue algorithm for routers is known as Drop Tail. Drop Tail queues simply accept any packet that arrives when there is sufficient buffer space and drop any packet that arrives when there is insufficient buffer space.

RED gateways instead attempt to detect incoming congestion by computing a weighted average queue size, since a sustained long queue is a sign of network congestion. Its behavior is based on a threshold: RED mechanism starts the dropping process when the number of packets stored is larger than this value. The

number of packets dropped (i.e., the probability that an incoming packet has to be dropped) increases as soon as the number of queued packets increases. When the number of packet reaches the maximum queue size, all the incoming packets are dropped (drop probability equal to 1). In this condition the RED algorithm has the same behavior of the Drop Tail one. RED keeps traces of the past status of the queue. The variable (number of packets in queue) that indicate that the algorithm has to start dropping packets does not depend simply on the amount of packets in queue, but it is calculated by means of an exponential moving weighted average function (EMWA) that is able to discriminate a short term burst and a long term congestion. Short-term bursts are quite common in the Internet because of the bursty nature of data traffic. In this case the algorithm does not operate. Vice versa, it must intervene when, for a certain amount of time, the number of packets in queue is above a certain threshold. This means that the network is starting to be overloaded so that it is likely to experiment congestion in a short time interval. RED operates only in presence of a responsible end-to-end adjustment mechanism. RED itself (i.e., dropping packets) is not enough to decrease the amount of traffic into the network. The congestion is avoided thanks to the TCP responsiveness that, in presence of drops, decreases the rate of the incoming traffic because of its typical sawtooth behavior. Clearly this approach does not work with malicious sources. RED cooperates with the enhanced version of TCP (TCP NewReno, TCP SACK) in decreasing the sawtooth behavior that is a peculiar characteristic of this protocol. Drop Tail can discard a lot of packets together so that a TCP session can experiment more than a packet loss per window of data. This triggers a timeout and the throughput of that session decreases. With RED is less likely to experiment more than one loss per window of data: TCP can recover with the Fast Retransmit Algorithm and the session does not experiment any timeouts.

For a RED gateway that drops packets, rather than marking a congestion bit, the following three phases sum up its algorithm:

Phase 1: Normal Operation. If the average queue size is less than the minimum threshold, no packets are dropped.

Phase 2: Congestion Avoidance. If the average queue size is between the minimum and maximum thresholds, packets are dropped with a certain probability. This probability is a function of the average queue size, so that larger queues lead to higher drop probabilities.

Phase 3: Congestion Control. If the average queue size is greater than the maximum threshold, all incoming packets are dropped.

3.3.4 Multiple RED Parameters

The Diffserv architecture provides QoS by dividing traffic into different categories, marking each packet with a code point that indicates its category, and scheduling

packets according to their code points. The Assured Forwarding mechanism is a group of code points that can be used in a Diffserv network to define four classes of traffic, each of which has three drop precedences. Those drop precedences enable differential treatment of traffic within a single class.

Assured Forwarding uses the RED mechanism by enqueuing all packets for a single class into one physical queue that is made up of three virtual queues (one for each drop precedence). Different RED parameters are used for the virtual queues, causing packets from one virtual queue to be dropped more frequently than packets from another. Using a proper traffic shaper on the edge router, a not conform traffic can be moved from a virtual queue with a higher drop preference to another one with lower; then this declassed traffic will be penalized during the congestion phase.

For example, one code point might be used for assured traffic and another for best effort traffic. The assured packet virtual queue will have higher minimum and maximum thresholds than those of best effort queue, meaning that best effort packets will enter the congestion avoidance and congestion control phase prior to assured packets.

3.3.5 Diffserv Architecture

The Diffserv architecture has three major components:

Policy: is specified by network administrator about the level of service a class of traffic should receive in the network.

Edge router: router marks packets with a code point according to the policy specified.

Core router: examines packets' code point marking and forwarding them accordingly.

DiffServ attempts to restrict complexity to only the edge routers.

A policy specifies which traffic receives a particular level of service in the network. Although a policy and resource manager is a necessary component of a Diffserv network that allows an administrator to communicate policies to the edge and core devices, it is not important for the NS-2 Diffserv implementation. Instead, policy information is simply specified for each edge and core device through the Tcl scripts.

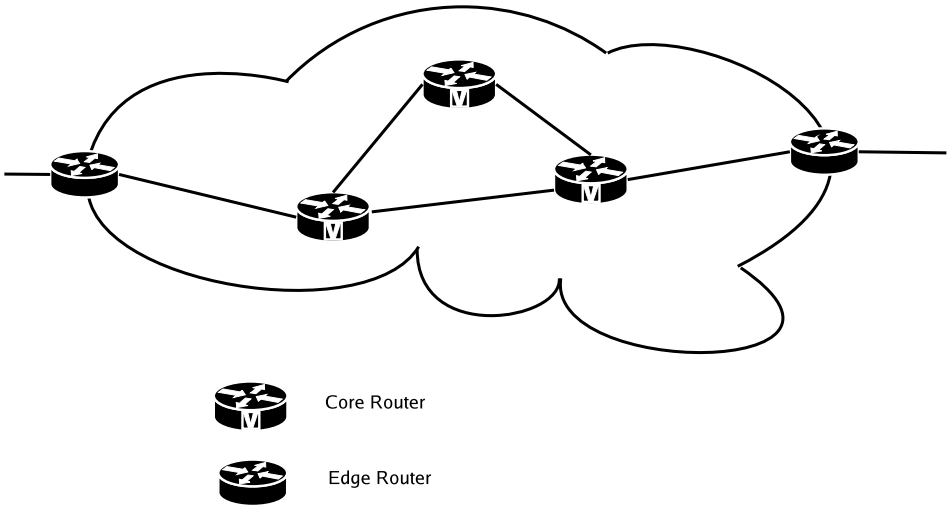


Figure 3.7: Devices in a Diffserv Domain

The notion of edge and core devices, as illustrated in Figure 3.7, is key to the understanding of the NS-2 Diffserv implementation.

Edge Router Responsibilities:

- examining incoming packets and classifying them according to policy specified by the network administrator;
- marking packets with a code point that reflects the desired level of service;
- ensuring that user traffic adheres to its policy specifications, by shaping and policing traffic.

Core Router Responsibilities:

- ensuring that user traffic adheres to its policy specifications, by shaping and policing traffic;
- forwarding incoming packets according to their markings. (Core routers provide a reaction to the marking done by edge routers).

The DiffServ architecture provides QoS by dividing traffic into different categories, marking each packet with a code point that indicates its category, and scheduling packets accordingly. The DiffServ module in NS-2 can support four classes of traffic, each of which has three dropping precedences allowing differential treatment of traffic within a single class. Packets in a single class of traffic are enqueued into one corresponding physical RED queue, which contains three virtual queues (one for each drop precedence).

Different RED parameters can be configured for virtual queues, causing packets from one virtual queue to be dropped more frequently than packets from another. A packet with a lower dropping precedence is given better treatment in times of congestion because it is assigned a code point that corresponds to a virtual queue with relatively lenient RED parameters.

All these features allow to differentiate a multimedia stream by other kinds of traffic, and to think about internal modification of the scheduler, to permit reconfiguration in case of particular condition as the ingress of a particular stream. This simulator module has been exploited for this reason and modified as presented in Chapter 4.

3.4 MPEG4 Model

Our investigation is focused on the study of QoS applied to delay sensitive traffic types, e.g. MPEG4. For this reason, an an MPEG4 traffic source to NS-2 was added. This traffic generator uses the *Transform Expand Sample Methodology*. We have chosen such kind of traffic source because the transfer of digital video will be a crucial component of the design of future home networking applications. This transfer was made feasible by the advancement of digital video encoding techniques that reduced the bandwidth required for this transfer to a practical level. MPEG4 is an encoding technique that is suitable for home networking applications with its low bit rate. It also has the advantage that allows viewers to interact with encoded objects. One of the motivations behind establishing the ISO Moving-Picture-Experts-Group (MPEG) family of standards for digital video encoding is that a lot of researchers and communication experts believe that, sooner or later, all the devices that are part of our daily life will be connected to the Internet.

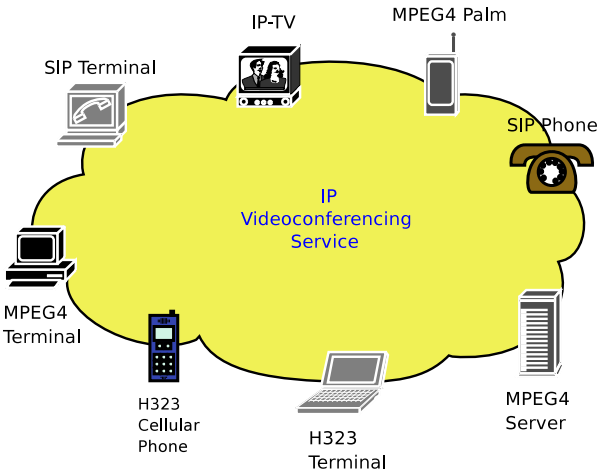


Figure 3.8: Audio and Video in the Internet

Connecting those devices and appliances to the Internet will require the transfer of video or at least will include video transfer as an enhancement. This family of standards includes MPEG1, MPEG2, and MPEG4. MPEG4 is the one that is most suitable for the Internet [5]. The main feature of importance to the network is MPEG4's capability of real-time adaptive encoding. This enhances network utilization and enables MPEG4 senders to be more responsive to changes in network conditions. It generates video in three different frame types (I, P, and B) that serve to encode different portions of the video signal in different levels of quality.

This kind of traffic generator can be used to study MPEG4 behavior and performance through simulation. This is relevant because it permit us to reproduce a more realistic scenario to simulate multimedia transfer on the Internet. Our objective is to study this performance and properties combined in a Diffserv environment using simulation. This traffic generator is able to generate traffic that has almost the same first and second order statistics as an original trace of MPEG4 frames that is generated using an MPEG4 encoder [7]. The source code of this traffic generator is in the Appendix A.

Table 3.1 contains a chunk of TCL code to set up a MPEG4 traffic generator in NS-2.

```
set source [$ns node]
set udp0 [new Agent/UDP]
$ns attach-agent $source $udp0
set vdo [new Application/Traffic/MPEG4]
$vdo set initialSeed_ 0.4
$vdo set rateFactor_ 5
$vdo attach-agent $udp0

$ns at 1.8 "$vdo start"
```

Table 3.1: Part of TCL code using MPEG4 source

The code shows how an application is placed on the top of an agent (UDP in this case). It generates frames every 1/30 seconds. It's also possible to change it in the *C++* source code and recompile. To obtain the correct behavior of this traffic generator, a directory called *video_model* where we are going to run the TCL scripts has been created. This directory contains 6 statistic files provided with the source package. We need also to add two lines in the *tcl/lib/ns-default.tcl* file as shown in Table 3.2.

The *rateFactor* parameter indicates how much it is possible to scale up (or down) the video input while preserving the same sample path and autocorrelation

```
Application/Traffic/MPEG4 set rateFactor_ 1
Application/Traffic/MPEG4 set initialSeed_ 0.5
```

Table 3.2: Configuration lines to add in the NS-2 default file

function for the frame size distribution. The *initialSeed* is used to start generating the first frame in the model. This implementation has been taken from [7].

3.5 Competing Streams

In this section, the flows that will be utilized in the scenarios described in the Chapter 5 are presented and also some excerpts of code are illustrated. These streams compete against the multimedia traffic and every set of simulation is repeated twice. They are basically used to evaluate the performance of the MPEG stream in a real scenario.

3.5.1 Costant Bit Rate

This kind of source generates traffic according to a deterministic rate. Packet size is constant but optionally some randomizing dither can be enabled on the inter-packet departure intervals. In the Table 3.3 a snippet of code follows, in which it is clear how to set up an application sending such kind of traffic. Usually an application is placed on top of an agent like UDP/TCP, and this agent needs a connection with the destination (sink). At the application level it is possible to set some parameters as packet size and sending rate.

3.5.2 Poisson

This traffic application is set up on a UDP agent too. The generator is an Exponential On/Off and it can be configured to behave as a *Poisson process* by setting the variable *burst_time_* to 0 and the variable *rate_* to a very large value. The C++ code guarantees that even if the burst time is zero, at least one packet is sent. Additionally, the next interarrival time is the sum of the assumed packet transmission time (governed by the variable *rate_*) and the random variate corresponding to *idle_time_*. Therefore, to make the first term in the sum very small, make the burst rate very large so that the transmission time is negligible compared to the typical idle times [18]. This generator can be forced to use a random variable as shown in the Table 3.4.

```
set udp1 [new Agent/UDP]
$ns attach-agent $s1 $udp1
$udp1 set fid_ 10
$ns simplex-connect $udp1 $sink1
set cbr1 [new Application/Traffic/CBR]
$cbr1 attach-agent $udp1
#$cbr1 set packet_size_ $packetSize
$cbr1 set packet_size_ 675
$udp1 set packetSize_ $packetSize
$cbr1 set rate_ 3.3Mb
```

Table 3.3: Excerpt of CBR Tcl code

```
set udp0 [new Agent/UDP]
$ns attach-agent $s2 $udp0
$udp0 set fid_ 20
$ns simplex-connect $udp0 $sink1
set rng0 [new RNG]
$rng0 next-substream
set poisson0 [new Application/Traffic/Exponential]
$poisson0 set packetSize_ 675
$poisson0 set burst_time_ 0
$poisson0 set idle_time_ 0.002631
$poisson0 set rate_ 100Gb
$poisson0 attach-agent $udp0
$poisson0 use-rng $rng0
```

Table 3.4: Excerpt of Poisson Tcl code

3.6 Motivation

Recently, reconfigurable hardware has been used to build network solutions. Due to the availability of high bandwidths resulting from high capacity links, the packets can be transmitted through the link at high speeds. Therefore, the router has to be fast enough to be able to switch packets from incoming links into one of the

outgoing link at a speed that matches the available link speed [23]. Reconfigurable hardware is a kind of hardware whose functionality may change in response to the demands placed upon the system while it is running. This gives us both flexibility of software and the performance of hardware. Today's FPGA technology allows reconfigurable hardware to be integrated into standard PC hardware as well as into dedicated router system. With a hardware like this it is possible to offer support for task CPU intensive.

These hardware reconfiguration capabilities can be beneficial from different aspects. Some of these benefits are listed below:

- performance can be gained from limited hardware resources by tuning configuration parameters on a flexible hardware architecture;
- hardwired data structures can be used to accelerate performance by eliminating the drawbacks of traditional memory storage-based data structures;
- the system cost can be reduced by fitting multiple features and applications on a single reconfigurable hardware platform or by partitioning an application into some stages being configured serially on a smaller platform.

Some of currently available field-programmable gate arrays support partial reconfiguration in a few microseconds and full reconfiguration in a few milliseconds. This has made reconfigurable systems more attractive to researchers due to their capability of implementing reconfigurable hardware in real world [3].

The service providers are increasingly confronting the crucial necessity to separate and control their traffic at the service, application, and user level; this motivation brings to employ a lot of commercial solutions using FPGA in the core network. With the current high bandwidths, these requirements translate into the capability of supporting QoS mechanisms in the hardware equipment at various switching points in the network. Thinking about the *flexibility*, it is possible to customize the traffic management through configuration and run-time parameters while for the *scalability*, the FPGA resources can scale according to the number of queues and the channel in the application. Comparing this solution with an ASIC or standard product solution, we are going to incur in a significant amount of risk. Volumes are uncertain, which leads to exorbitant non-recurring engineering (NRE) costs. FPGAs naturally fit for implementing traffic managers because of the limitation of this risk and the ability to differentiate a traffic manager solution. Additionally, it is possible to use a reconfigurable solution to add and support new services in the future.

Our idea start from the concept that we want to utilize the features of reconfigurable hardware to improve the execution of scheduling operation:

- possible to exploit the feature of *reconfigurability*, in other words, the scheduling algorithm can be changed basing on some condition inside the network.

- running this operation in hardware can speed up some particular function that in software are executed slower than in hardware;

The first issue is addressed by our investigation while the second is a challenge for future investigations and could bring another increment at performance level. We address the first issue from the software point of view. Our analysis, and the choice of the scheduling algorithms to evaluate in the simulator, start from a research about the implementation costs of these algorithms in FPGA.

3.7 Cost Analysis

Every server uses a different scheduling algorithm to decide the order in which the requests have to be served. A scheduling discipline should satisfy the following requirements:

1 is easy to be implemented;

2 provide fairly distributed bandwidth to competing streams;

3 guarantees performance bounds for a wide range of traffic types;

4 allows easy admission control decision.

Within the lot of different approaches proposed in literature, Weighted Round Robin (WRR) and Weighted Fair Queuing (WFQ) are perhaps these two most adopted disciplines [19]. Our analysis starts from a literature research on the hardware implementation of the two above mentioned algorithms. Several disciplines have been proposed in literature to provide QoS.

It has been shown that meeting packet-time requirements of multi-gigabit links is difficult using only a software based implementation [24]. To maintain high link utilization, the scheduler algorithm must be able to make a decision on the packet to schedule in a time:

$$\frac{packet\_length}{line\_speed} \tag{3.1}$$

This trend leads to think about architectural framework that can provide scheduling algorithm solutions, balancing performance and constraints and making tradeoffs required in their physical realization [6]. Figure 3.9 (b) shows if the required scheduling rate, can be realized in silicon or reconfigurable logic given the implementation complexity of a given scheduling discipline. For serving a big number of streams, a higher scheduling rate is needed (Figure 3.9 (a).

Devices with FPGA on board represent a good solution to address these problems. We took the information about the implementation in hardware of WRR and WFQ in a Virtex II Device. The typical basic architecture of this device consists of an array of configurable logic blocks (CLBs) and routing channels. A WFQ

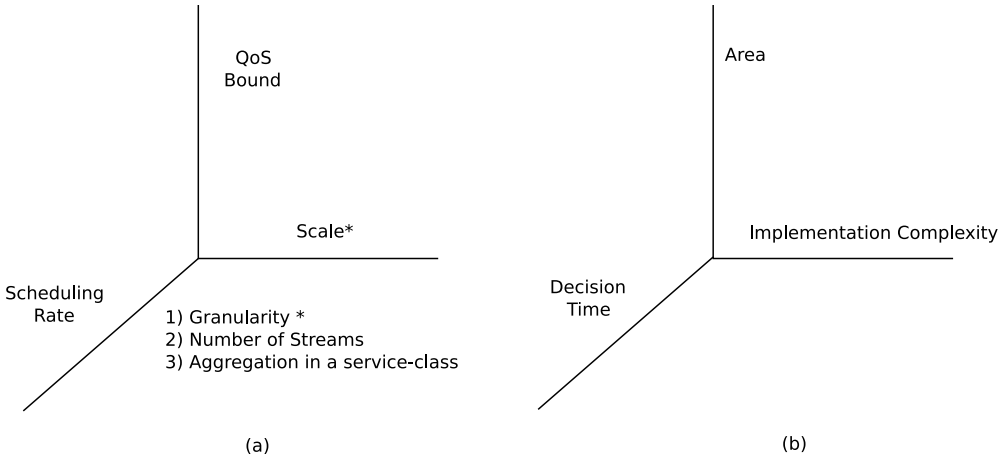


Figure 3.9: Implementation Complexity of Packet Scheduler

implementation in a Virtex II device takes more than 2400 configurable logic block (CLB) [19] and for the same number of streams to handle, WRR takes 1500 CLB [15]. So utilizing all the CLB available on this device it is possible to save almost 70% of area depending on the chosen algorithm. This leads to a big challenge, it is possible to think to a more dynamic handling of the traffic: the unutilized area in the FPGA can save power consumption or using it, it is possible to handle more stream sessions with a different scheduling algorithm. This is what we are going to address in this work.

3.8 Summary

In this chapter, the network simulator and all the environment utilized in the simulations has been introduced. It has been described how to create a simulation scenario and how to get results in NS-2. A description of the differentiated services architecture for IP QoS has been furnished together with the support offered by the simulator through the Nortel Diffserv module. The MPEG4 model has been described and some excerpts of code permitting to set up a multimedia traffic source, in the simulator, has been explained. A short description of the competing streams utilized in the typical network scenario together with the MPEG4 stream has been provided, and also for them some excerpts of code have been explained. The last part of the chapter has illustrated the motivation bringing to the choice of utilize reconfigurable hardware and a cost analysis of the chosen scheduling algorithms in hardware has been presented.

4 Modifications to NS-2

In this chapter, the modifications to the simulator are presented and some excerpts of code are commented. Section 4.1 introduce the implementation of an agent to monitor the results of the simulation; Section 4.2 describe how the support to the reconfigurability has been obtained and Section 4.3 presents the implementation of WFQ scheduler in the simulator.

4.1 Monitor Agent

In order to have output files with data on the simulation (trace files) or files used for visualization (NAM files), we need to tell to the simulator to keep track of traffic in the network with a set of dedicated commands, i.e., *open, trace-all, flush-trace, close*. When tracing into an output ASCII file, the trace is organized in 12 fields as depicted in Figure 4.1. The meaning of these fields is as follows:

Figure 4.1: Trace File structure

Event represents the event type. It is given by one four possible symbols *e, +, -, d* which correspond respectively to receive (at the output of the link), enqueued, dequeued and dropped;

Time gives the time at which the event occurs;

From Node is the input node of the link at which the event occurs;

To node gives the output node of the link at which the event occurs;

Pkt Type gives the packet type (for example, CBR, or TCP. The type corresponds to the name that we gave to those applications);

Ptk Size gives the packet size;

Some flags ;

Fid is the flow id of IPv6 that a user can set for each flow at the input OTcl script. One can further use this field for analysis purpose; it is also used when specifying stream color for the NAM display;

Src Addr is the source address given in the form of "node.port";

Dst Addr is the destination address, given in the same form;

Seq Num is the network layer protocol's packet sequence number. Even though UDP implementation in a real network does not use sequence number, NS-2 keeps track of UDP packet sequence number for analysis purposes;

Pkt ID field shows the unique ID of the packet.

For example, it is possible to take a look to the following lines in Table 5.2 extrapolated from a trace file:

r -t 0.1058 -s 11 -d 5 -p video -e 1000 -c 40 -i 0 -a 40 -x 11.0 9.0 -1 —— null
+ -t 0.1058 -s 5 -d 7 -p video -e 1000 -c 40 -i 0 -a 40 -x 11.0 9.0 -1 —— null
- -t 0.1058 -s 5 -d 7 -p video -e 1000 -c 40 -i 0 -a 40 -x 11.0 9.0 -1 —— null
h -t 0.1058 -s 5 -d 7 -p video -e 1000 -c 40 -i 0 -a 40 -x 11.0 9.0 -1 —— null
r -t 0.1058 -s 12 -d 5 -p video -e 1000 -c 40 -i 2 -a 40 -x 12.0 9.0 -1 —— null
+ -t 0.1058 -s 5 -d 7 -p video -e 1000 -c 40 -i 2 -a 40 -x 12.0 9.0 -1 —— null
r -t 0.1058 -s 13 -d 5 -p video -e 1000 -c 40 -i 4 -a 40 -x 13.0 9.0 -1 —— null
+ -t 0.1058 -s 5 -d 7 -p video -e 1000 -c 40 -i 4 -a 40 -x 13.0 9.0 -1 —— null

Table 4.1: Trace-file Example

Using the trace commands may result in the creation of huge files, about 1 Gigabytes of data for each simulation, this leads to two main problem:

- a lot of disk writes with a considerably slow down simulation time;
- difficult to parse the data file generated.

For these reasons, a new agent has been implemented and it is also possible to monitor only metrics of interest. In the NS-2 architecture, it is possible to put an agent on a node doing what is needed; it is also possible to use predefined agents or implement your own. In the next frames there is a chunk of code implementing this special agent.

Monitor Agent header file

```
1  /*#ifndef ns_udp_h
2  #define ns_udp_h
3  */
4  #include "agent.h"
5  #include "trafgen.h"
6  #include "packet.h"
7  #include <fstream>
8
9  //"rtp timestamp" needs the samplerate
10 #define SAMPLERATE 8000
11 //#define RTP_M 0x0080 // marker for significant events
12
13 class MyAgent : public Agent {
14 public:
15    MyAgent();
16    virtual void recv(Packet* pkt, Handler*);
17 private:
18    double delay;
19 };
```

Monitor Agent source file

```
1  #ifndef lint
2  static const char rcsid[] =
3      "@(#) $Header: /nfs/jade/vint/CVSROOT/ns-2/apps/udp.cc, \
4      v 1.19 2001/11/16 22:29:59 buchheim Exp $ (Xerox)";
5  #endif
6
7  #include "myagent.h"
8  #include "rtp.h"
9  #include "random.h"
10 #include "address.h"
11 #include "ip.h"
12
13 static class MyAgentClass : public TclClass {
14 public:
15    MyAgentClass() : TclClass("Agent/MyAgent") {}
16    TclObject* create(int, const char*const*) {
17       return (new MyAgent());
18    }
19 } class_my_agent;
20
```

```
21 MyAgent::MyAgent() : Agent(PT_UDP) {
22     delay = 0;
23 }
24
25 // Agent/Udp instproc recv {from data} {puts data}
26 // put in timestamp and sequence number,
27 // even though UDP doesn't usually have one.
28
29 void MyAgent::recv(Packet* pkt, Handler*){
30      hdr_cmn* cmn = hdr_cmn::access(pkt);
31      char nome_file[1024];
32      char dim_file[1024];
33      char arrive_file[1024];
34      hdr_ip* hdr_iph = hdr_ip::access(pkt);
35      int flow_id=hdr_iph->flowid();
36      //packet_t mypacket = cmn->ptype();
37      //printf("type: %d - flowid: %d\n",mypacket,flow_id);
38      double now = Scheduler::instance().clock();
39      delay = now - (cmn->timestamp()/SAMPLERATE);
40      sprintf(nome_file,"file_d%d",flow_id);
41      fstream delay_file(nome_file,ios::out|ios::app);
42      delay_file<<delay<<endl;
43      //print each received packet dimension in a file
44      //separately depending on the kind of the packet
45      std::sprintf(dim_file,"file_d%d_dim",flow_id);
46      std::fstream dim(dim_file,std::ios::out|std::ios::app);
47      dim<<cmn->size()<<std::endl;
48
49      std::sprintf(arrive_file,"file_d%d_arrive",flow_id);
50      std::fstream arr(arrive_file,std::ios::out|std::ios::app);
51      arr<<now<<std::endl;
52
53      Packet::free(pkt);
54 }
```

The *recv()* function is called every time a packet is received. So we access to the packet, take its *flow id* to distinguish it from other kinds of traffic, that is, we need to put every delay (or any metric) of a separate stream in a different file. In this way, it will be very simple to evaluate the average *end-to-end* delay, and the *jitter*, processing them with a simple *shell* or Perl script. To calculate the *throughput* we also need of the packet size, so we gather them into another separate file. The file name is built on the base of file ID, in this way different data-stream separation is

obtained automatically. The packet delay is calculated reading the time-stamp in the packet common-header and subtracting it by the current time obtained thanks the clock method of Scheduler instance. To calculate the packet size, it is possible to use the *size()* method of the common-header class.

4.2 Support for Reconfiguration to the Simulator

The work starts from the assumption that reconfigurable hardware is available in the router, so it is possible to change many network parameters in the device, to obtain a better performance for some streams. With an FPGA on board, it is possible to think to change the scheduling algorithm inside the core router, and meet some requirements.

Why do we need to change the scheduling disciplines? As it has been discussed before, different scheduling algorithms take different space in FPGA being implemented, and in the Section 3.7 it has been argued that WFQ implementation takes almost double size in terms of area compared with WRR. This means that it is possible to handle almost double the number of streams with this device [23]. It is also possible to use a cheaper algorithm to save area in FPGA, and save power turning off unitilized area. What we are going to address is the first problem, we want to gain in performance more than saving power. Therefore, thinking about a full utilization of this FPGA, we need to understand *when* and *how* to change the algorithm. In the NS-2 implementation, every kind of traffic is identified by a different packet type. It is possible to access to the common header of the IP packet and understand which kind of packet is going to be processed. During the queuing phase, the packet type is recognized and therefore the scheduling algorithm is changed.

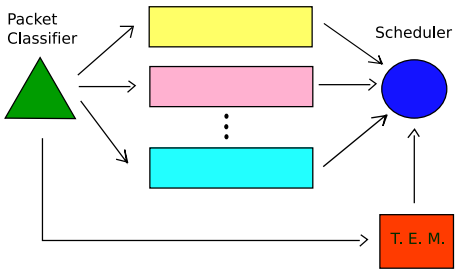


Figure 4.2: Reconfiguration Idea

This operation is performed in the T.E.M (Traffic Evaluation Module) component that we have developed. It is possible to do this mainly in two different ways. First, a static threshold can be set and the scheduling algorithm is changed in hardware when it is exceeded. Second, a ratio between different packet streams, within

a specific time period, can be set to determine whether to switch the scheduling algorithm or not. The second approach is more accurate, but choosing the size of the interval and the desired threshold represents a challenge for further investigations. In this section, a comparison between the two methodologies is presented. To add reconfiguration support to the simulator, the Diffserv module has been modified. In particular the enqueue function has been altered to monitor, keep track of the kind of traffic passing into the network and change the scheduling algorithm consequently. Two main approaches have been adopted:

1. static threshold;
2. average load in a period.

Using the first one, the reconfiguration happens after a fixed number of video packets has been received and forwarded by the core router (dropped packets are not taken in account). This approach lacks in flexibility and it is not suitable for a complex traffic situation, because there could be a long period without multimedia transmission inside the backbone but it may be possible to have a big variation between activity and inactivity interval of the sender. Of course this approach presents a very easy implementation and does not introduce computational overhead for the router. In the next frame, a snippet of code from the enqueue function of the Diffserv module, illustrating this mechanism follows. The major problem is that, if the threshold is passed, the system has no way to recognize whether the MPEG4 traffic is continuing to flow. It would be possible to set a timer and try to understand if such kind of traffic is not passing anymore, but it would be another challenge to choice the value of this deadline.

enqueue function with fixed threshold

```
1  //adaptive change of scheduling algorithm
2
3  //access to packet's common header and check which
4  //kind of packet is it
5  //see packet_t for type of packet in packet.h
6  //UDP = 2, VIDEO = 4, FTP = 28, PARETO = 29, POISSON = 29
7
8  //access to the packet common header
9  hdr_cmn* accesspacket = hdr_cmn::access(pkt);
10 packet_t mypacket = accesspacket->ptype();
11
12 //num_video_packet and num_other_packet are declared
13 //in the header file of diffserv module
14
15 //update the number of packets
16 ( mypacket == 4 ) ? num_video_packet++ : num_other_packet++;
```

```
17
18 //when there are more than N video-packet
19 //in the network, the scheduling algorithm
20 //is changed in WRR
21
22 if(mypacket == 4){
23   if( num_video_packet >= N ){
24       schedMode= schedModeWRR;
25       printf("num of video packet=%d\n",num_video_packet);
26   }
27 }
```

The second one is more accurate: a counter for each kind of packet is kept, every N seconds a check on the ratio between the number of video packet, and the number of the other packet, is calculated and if this ratio goes beyond a certain threshold, then the scheduling algorithm is changed. Dimensioning the threshold is also a challenge, but it is easier than the first approach and it should be possible also to think of a solution with a movable threshold under certain conditions. In the following frame, there is the code implementing the second strategy.

enqueue function with dynamic check

```
1  //adaptive change of scheduling algorithm
2  //access to packet's common header and check which
3  //kind of packet is it
4  //see packet_t for type of packet in packet.h
5  //UDP = 2, VIDEO = 4, FTP = 28, PARETO = 29, POISSON = 29
6  //access to the packet common header
7  hdr_cmn* accesspacket = hdr_cmn::access(pkt);
8  packet_t mypacket = accesspacket->ptype();
9  //get the current time
10 double now = Scheduler::instance().clock();
11 //the variable "double initial" is declared
12 //in the header of this module and it's setted to 0
13 //num_video_packet and num_other_packet are declared
14 //in the header file of diffserv module
15 //update the number of packets
16 ( mypacket == 4 ) ? num_video_packet++ : num_other_packet++;
17 //the check is performed every N seconds
18 if( now - initial > N ){
19    //if the number of video packet is about 5% of total
20    //in this window, the scheduling algorithm is changed
21    if( num_video_packet >=
22            (((num_video_packet+num_other_packet)/100)*5) ){
```

```
23          schedMode = schedModeWRR;
24      }
25      else{
26          schedMode = schedModeRR;
27      }
28      //put at 0 the counter and
29      //update the initial counter for the window
30      num_video_packet = 0;
31      num_other_packet = 0;
32      initial = now;
33  }
```

To validate this modification, here follows Table 4.2 with the results of a set of simulations in which the use of Round Robin, Weighted Round Robin and Round Robin with change in Weighted Round Robin are compared. In the first simulation, every queue has been served by a Round Robin scheduler without the modification. In the second, the modification has been applied and and the simulation run changing the scheduling algorithm from RR to WRR. In the last simulation, the scheduler is set on WRR. For the second and the last one, the queue weights are set as shown in Table 4.2.

Delay (sec)	**RR**	**RR→WRR**	**WRR**	**Weight**
Poisson1	0.188199454862022	0.232979777481014	0.307661438617046	3
Poisson2	0.1885260599879	0.138676982735961	0.0952731051873826	7
MPEG4	0.17178877452345	0.105055283908694	0.0520737071567351	10

Table 4.2: Set of simulations with reconfiguration

Packet Drop	**TotPkts**	**TxPkts**	**link-drops**
All	110451	88655	21796
Poisson1	38174	29749	8425
Poisson2	37809	29746	8063
MPEG4	34468	29160	5308

Table 4.3: Packet drop for Round Robin

In Tables 4.3, 4.4 and 4.5 the drop statistics are gathered for the three queues. Of course, the queue weight does not make any sense for the simulation using RR, because every queue has the same.

Looking at these results is possible to understand how with the reconfiguration from RR to WRR is possible to recover the bad trend for the MPEG stream. Our

Packet Drop	TotPkts	TxPkts	link-drops
All	110451	88779	21672
Poisson1	38174	23435	14739
Poisson2	37809	33468	4341
MPEG4	34468	31876	2592

Table 4.4: Packet drop for RR→WRR

Packet Drop	TotPkts	TxPkts	link-drops
All	110451	88958	21493
Poisson1	38174	17412	20762
Poisson2	37809	37078	731
MPEG4	34468	34468	0

Table 4.5: Packet drop for WRR

idea is to use reconfiguration and give priority to multimedia traffic only when this kind of traffic is present in the network. Doing this, it is possible to save area in FPGA and also handle more streams. This aspects will be described in more detail later in Chapter 5.

4.2.1 Reconfiguration Delay

As observed earlier, NS-2 does not offer any kind of support to the reconfiguration. Another problem to be addressed is the lack of reconfiguration delay. When the FPGA executes a scheduling change, in hardware this operation takes few milliseconds depending on a lot of factors and mainly each device has its own proper delay. The only way to reproduce this factor is to introduce a constant delay between each reconfiguration. How this fixed delay is chosen depends on real simulations on hardware devices. In term of implementation, we will have one delay for each available device. A module to set this factor has been designed and implemented. It is possible to configure this parameter during the scenario setup. As described in Chapter 3, the simulation scenario has to be set in a Tcl description file, so the module maps the implementation code in C++ with the Tcl scenario code. In Table 4.6, the C++ code implementing this features is shown. If no values are

```
void dsREDQueue::command(int argc, const char*const* argv) {
// ..
    if (strcmp(argv[1], "setDevice") == 0) {
        setDevice(argv);
        return(TCL_OK);
    }
// ..
    return(Queue::command(argc, argv));
}

void dsREDQueue::setDevice(const char*const* device_tipe) {
// ..
    if (strcmp(device_type, "XC2V6000") == 0) {
        reconfiguration_delay = DELAY1;
    }
    else if (strcmp(device_type,"XCV1000") == 0) {
        reconfiguration_delay = DELAY2;
    }
// ..
}
```

Table 4.6: Code Mapping Device Delay, C++

specified, the default delay is set to zero, so in this case the reconfiguration does not introduce time lag. It is possible to decide this value in the configuration Tcl file like shown in the code in Table 4.7. This approach permits a more realistic

```
set ns [new Simulator]
set e1 [$ns node]
set e2 [$ns node]
set core [$ns node]
$ns simplex-link $core $e2 6Mb 5ms dsRED/core
set qCE2 [[$ns link $core $e2] queue]
qCE2 setDevice XC2V6000;
```

Table 4.7: Code Mapping Device Delay, Tcl

evaluation of our problem.

4.3 Weighted Fair Queuing Implementation

This section discusses in detail the scheduling algorithm implementation in NS-2. In the following sections, some excerpts of code are commented.

4.3.1 Formulas

WFQ has been introduced in Section 2.4.2.2, anyway the functions to calculate when a packet has to be scheduled are presented here:

$$V(t+\tau) = V(t) + (\frac{\tau}{\sum \phi_{i \in B_j}}) \tag{4.1}$$

$$S_i^k = max\{F_i^{k-1}, V(t+\tau)\} \tag{4.2}$$

$$F_i^k = S_i^k + (\frac{L_i^k}{\phi_i}) \tag{4.3}$$

WFQ schedules packets according to their arrival time, size and associated weight. When a new packet arrives, the virtual time is calculated and the packet is scheduled starting from the packet with the least finish time. In this system an event is defined as the arrival or departure from the GPS scheduler of a packet. For each session, there is a weight associated with it represented by ϕ_i. For an interval τ with a constant set of backlogged flows within any busy period, $V(t+\tau)$, the virtual time in the GPS scheduler when the k^{th} packet from session i arrives or departs, is found from Equation 4.1 [11].

B_j denotes the busy sessions in the interval between two events and L_i^k is the length of the k^{th} packet in session i. For each packet arriving at the GPS scheduler,

a start tag is calculated using 4.2. S_i^k represents the start tag of the k^{th} packet in session i and F_i^{k-1} is the finish tag of the previous packet in session i, this one is calculated by Equation 4.3. $Next(t)$ can be obtained by Equation 4.4, it represents the real time at which the next packet will leave the GPS system after an event at time t, if there are no more arrivals after time t.

$$Next(t) = t + (F_{MIN} - V(t+\tau)) \sum_{i \in B_j} \phi_i \tag{4.4}$$

t is the time elapsed since the scheduler became active, F_{MIN} is the minimum value of finish tag for a packet that has still to depart from the GPS simulation. It is possible to split the operation of a WFQ scheduler in two distinct phases: *packet arrival* and *packet departure*. Let's summarize the steps:

- *Packet Arrival*
  - if system idle before packet arrival, then $V(t)$ is set to zero. Otherwise $V(t)$ is updated with Equation 4.1;
  - calculate start and finish tags for packet using Equations 4.2 and 4.3 respectively;
  - if the session was not backlogged before then it is added to set B_j. Sum of backlogged sessions is then modified;
  - calculate real time of the next packet departure from the GPS system, i.e., $Next(t)$, from Equation 4.4;
- *Packet Departure*
  - update the value of $V(t)$ using Equation 4.1;
  - dequeue the packet with the smallest finish tag, F_{MIN};
  - if a session is no longer backlogged when the packet is de-queued then it is removed from set B_j;
  - calculate $Next(t)$ using Equation 4.4.

4.3.2 The code

All the steps in the *Packet Arrival* phase are performed by the WFQenqueue() function that, is called by the standard enqueue function of the Nortel Diffserv module. This function updates the virtual time variables, inserts control data in GPS List and invokes the scheduling of next departure.

Line 1 is the declaration of the function, the input consisting of a pointer to the packet and its queue id.

Line 3 to 4 access to the packet common header and obtain the packet size.

Line 6 obtain the current simulation time calling a scheduler method.

Line 8 to 16 update the virtual time, if the GPS system is idle, everything is initialized to zero and the last virtual time to the current simulation time, otherwise if the GPS is active all the parameters are update according to Formula 4.1.

Line 18 to 21 calculate the finish time for that packet as in Formula 4.3.

Line 24 to 26 update the list of backlogged session and the sum of the weights for the queues (this function is shown in the next frame).

Line 30 and 31 perform the list insertion in both the PGPG and the GPS list, passing them the queue id of the packet being inserted and also the finish time associated with it.

Line 33 to 38 schedule the next departure in the GPS reference system.

Update Weights Function

```
1  void dsREDQueue::WFQupdateSum () {
2
3   sum = 0;
4
5   for (int i=0; i < numQueues_; i++) {
6       if (B[i]) {
7           sum += queueWeight[i];
8       }
9   }
10 }
```

This function updates the weights for all the backlogged queue (line 5 to 9).

WFQ Enqueue Function

```
1  void dsREDQueue::WFQenqueue(Packet *p, int queueid) {
2
3      hdr_cmn *hdr = hdr_cmn::access(p);
4      int size = hdr->size();
5
6      double now =  Scheduler::instance().clock();
7
8      //virtual time update
9      if(GPS_idle) {
10         last_vt_update=now;
11         virt_time=0;
12         GPS_idle=0;
13     } else {
14         virt_time=virt_time+(now-last_vt_update)/sum;
15         last_vt_update=now;
16     }
17
18     // let's compute finish time
19     finish_t[queueid] = (finish_t[queueid] > virt_time ?
20             finish_t[queueid]:virt_time)
21             +size/(double)queueWeight[queueid]/(bandwidth/8);
22
23         // update sum and B
24         B[queueid]++;
25         WFQupdateSum ();
26     if ( fabs(sum) < safe_limit ) sum=0;
27
28
29     // insertion in the list
30     PGPS_list.insert_order(queueid, finish_t[queueid]);
31     GPS_list.insert_order(queueid, finish_t[queueid]);
32
33     // schedule next departure in the GPS reference system
34     if(wfq_event!=0) {
35         Scheduler::instance().cancel(wfq_event);
36         delete wfq_event;
37     }
38     WFQscheduleGPS();
39 }
```

On the other hand, the steps in the *Packet Departure* phase are executed by the WFQdequeueGPS() function (in the next frame), this one is invoked by the scheduler. It updates the virtual time variables and schedules the next GPS dequeuing event.

WFQ dequeue function

```
1  void dsREDQueue::WFQdequeueGPS(Event *e) {
2
3      double now = Scheduler::instance().clock();
4
5      //update virtual time
6      virt_time=virt_time+(now-last_vt_update)/sum;
7      last_vt_update=now;
8
9          //extract packet in GPS system
10     int queueid=GPS_list.get_data_min();
11     GPS_list.extract();
12
13         //update B and sum
14         B[queueid]--;
15         WFQupdateSum ();
16     if ( fabs(sum) < safe_limit ) sum=0;
17
18     if(sum==0) {
19         GPS_idle=1;
20         for(int i=0;i < MAX_QUEUES;i++) finish_t[i]=0;
21     }
22
23     // if GPS is not idle, schedule next GPS departure
24     delete e;
25     if(!GPS_idle)
26         WFQscheduleGPS();
27     else
28         wfq_event=0;
29 }
```

Line 1 is the function declaration, the input consisting of a pointer to the scheduler *Event*.

Line 3 obtain the current simulation time as usual.

Line 5 to 7 update the virtual time according to Formula 4.1.

Line 9 to 11 extract the queue id of the packet and the packet as well.

Line 13 to 15 update the list of the backlogged queues and also the weight.

Line 17 to 20 if no one queue is backlogged puts the GPS system as idle and reset the finish time for each queue.

Line 22 to 27 otherwise if the GPS system is not idle, a new departure will be scheduled.

The system utilized to keep track of packet in the GPS system (and consequently even in the WFQ) is a list template offering function performing the following operation:

- get_key_min() and get_data_min() return the key and the data of the smallest key element without extracting it from the list;
- extract() extracts the smallest key element from the list;
- insert_order() interface uses flow-id knowledge to allow the implementation of other data structures: Calendar Queues, array of Lists (a list per flow).

This file has been taken from the Alexander Sayenko's implementation [16].

4.3.3 GPS properties and complexity

GPS has two main properties:

- it can guarantee an end-to-end delay to a leaky-bucket constrained queue regardless of the behavior of other queues;
- it can ensure instantaneous fair allocation of bandwidth among all backlogged queues regardless of whether or not their traffic is constrained;

but it is also well known that it is difficult to implement such a scheduler supporting a huge amount of queues with different bandwidth requirements. The main difficulties can be summarized in:

- for each queue, a computation of system virtual time function has to be performed;
- how can the scheduler handle the packet order transmission?

In literature, it is possible to find several proposed WFQ implementations, the difference between them is mainly in the tradeoff between complexity and accuracy in the computation of the system virtual time function [10]. We have chosen the Packet Generalized Processor Sharing (PGPS): it uses the virtual time function defined by the GPS system whose worst case complexity is $O(N)$.

4.4 Summary

In this chapter, the *monitor agent* implemented to trace the simulations results was presented. This approach permits to save a lot of disk writes without incur in a considerable slow down of the simulations. It also permits to avoid the tedious parsing of the results data file generated by the simulator. The module to support the reconfiguration in the simulator has been illustrated describing the main reconfiguration idea and the approach utilized. Some excepts of code have been commented and the results of the utilization of this modification has been presented. The module developed to add the possibility to simulate the reconfiguration delay (typical of an hardware device during the change of the scheduling algorithm) has been described. At the end of the chapter, the weighted fair queuing implementation has been presented starting from a theoretical definition arriving to the code realization.

5 Simulations

This chapter presents the work of simulation performed to compare weighted round robin and weighted fair queuing. The first and second sections are dedicated to the tools utilized in the simulations to obtain reliable values. The first one presents a script calculating "confidence interval", which permits a realistic evaluation of the results, the other one introduces a class available from the simulator, permitting to use a random generator to vary the behavior of the traffic sources. Another section presents the fairness evaluation for which a test program has been written. The other sections discuss the simulations, and the last one points out the results of this work.

5.1 Confidence Intervals

A *confidence interval* gives an estimated range of values which is likely to include an unknown population parameter, the estimated range being calculated from a given set of sample data.

If independent samples are taken repeatedly from the same population, and a confidence interval calculated for each sample, then a certain percentage (confidence level) of the intervals will include the unknown population parameter. Confidence intervals are usually calculated so that this percentage is 95%, but we can produce 90%, 99%, 99.9%, confidence intervals for the unknown parameter. The width of the confidence interval gives us some idea about how uncertain we are about the unknown parameter. A very wide interval may indicate that more data should be collected before anything definitive can be said about the parameter; to address this problem, another set of simulations can be performed changing the seed of the random generator obtaining more value to estimate . Confidence intervals are more informative than the simple results of hypothesis tests since they provide a range of plausible values for the unknown parameter.

The *confidence level* is the probability value $1 - \alpha$ associated to a confidence interval. It is often expressed as a percentage. For example, say $\alpha = 0.05 = 5\%$, then confidence $level = (1 - 0.05) = 0.95$, that is, a 95% confidence interval level

For each simulation set, the confidence interval has been calculated with a confidence level of 95%. To easily calculate this interval, a Perl script has been developed. Each simulation task collects the results in a separate directory named with an incremental number to separate them. The Perl script walks through these directories and calculates the mean delay (Formula 5.1) for each traffic flow

in each simulation, then the average mean is calculated and it is used to obtain the variance (Formula 5.2). The last step consists in the the standard error computing (Formula 5.3), from which the beam for the confidence interval can be derived taking the value from the T-student Table 5.1 and multiplying this value for the standard error. The choice of the value from the T-Student table depends on the number of simulations performed and the confidence level desired.

$$\overline{x} = \frac{1}{n}\sum_{i=1}^{n} x_i \tag{5.1}$$

$$\sigma^2 = \frac{1}{n-1}\sum_{i=1}^{n}(x_i - \overline{x})^2 = \frac{n}{n-1}\left(\frac{1}{n}\sum_{i=1}^{n} x_i^2 - \overline{x}^2\right) \tag{5.2}$$

$$standard\_error = \sqrt{\frac{\sigma^2}{n}} \tag{5.3}$$

df/p	**0.40**	**0.25**	**0.10**	**0.05**	**0.025**	**0.01**	**0.005**	**0.0005**
1	0.324	1.000	3.077	6.313	12.706	31.820	63.656	636.619
2	0.288	0.816	1.885	2.919	4.302	6.964	9.924	31.599
3	0.276	0.764	1.637	2.353	3.182	4.540	5.840	12.924
4	0.270	0.740	1.533	2.131	2.776	3.746	4.604	8.610
5	0.267	0.726	1.475	2.015	2.570	3.364	4.032	6.868
6	0.264	0.717	1.439	1.943	2.446	3.142	3.707	5.958
7	0.263	0.711	1.414	1.894	2.364	2.997	3.499	5.407
8	0.261	0.706	1.396	1.859	2.306	2.896	3.355	5.041
9	0.260	0.702	1.383	1.833	2.262	2.821	3.249	4.780
10	0.260	0.699	1.372	1.812	2.228	2.763	3.169	4.586

Table 5.1: T-Student

These steps are summarized in the following frame.

Confidence Interval Calculation

```
#!/usr/bin/perl -w
use strict;
use List::Util qw( sum );

#calculate the mean
my @directory = 2 .. 9;
my @medie_campionarie;
my $coeff_conf = 2.36462; #for 8 simulations, n-1 = 7

foreach my $num (@directory){
    #build the file name for each directory
    my $filename = $num."/$ARGV[0]";
    open my $file, '<', $filename;
    my @samples;
    my $values_sum;

    while(my $value = <$file>){
        push @samples, $value;
        $values_sum += $value;
    }
    my $mean = $values_sum / scalar @samples;
    push @medie_campionarie, $mean;
    close $file;
}
#average of means
my $avg_mean =
     sum(@medie_campionarie) / scalar @medie_campionarie;
#calculate variance
my $summa;
foreach my $elem (@medie_campionarie){
    $summa+=($elem-$avg_mean)**2;
}
my $variance = (1/((scalar @medie_campionarie) - 1))*$summa;
print "average of the means: $avg_mean\n";

#calculate standard error
my $error = sqrt($variance/(scalar @medie_campionarie));
my $beam = $coeff_conf*$error;
print"interval [",$avg_mean-$beam,",",$avg_mean + $beam,"]";
```

5.2 Random Variables

In NS-2, it is possible to use a class containing an implementation of the combined multiple recursive generator MRG32k3a [8]. The concept utilized in this generator is the following: 1.8 x 10^{19} independent streams of random numbers are provided, each of which consists of 2.3 x 10^{15} sub-streams. A *period* is defined as the number of random numbers before an overlap occur, and each sub-stream has a period of 7.6 x 10^{22}. In Figure 5.1 a graphical idea of how the streams and sub-streams fit together is provided.

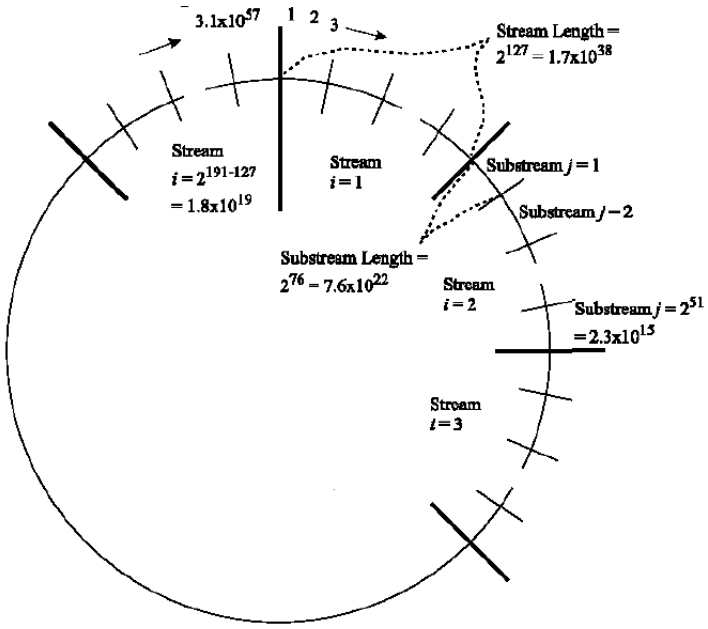


Figure 5.1: Overall arrangement of streams and sub-streams [9]

It is possible to use a default random number generator (defaultRNG) created at simulator initialization time. Each random variable in a simulation should use a separate RNG object. As soon as a new RNG object is created, it is automatically seeded to the beginning of the next independent stream of random numbers.

In our investigation we have to perform a statistical analysis on some metrics, it is made using independent replications of a simulation. For each simulation the seed of the generator is changed to ensure that the random numbers will be independent. It is also possible to force a traffic source to use this random variable being sure about traffic randomization.

5.3 Fairness Evaluation

NS-2 gives the possibility to create a particular traffic source using a *TrafficTrace* file. In such file, each record consists of two 32-bits fields in network (big-endian) byte order. The first one contains a delta δ expressed in microseconds, and it represents the time until the next packet is generated. The second contains the length in bytes of the next packet. We used this feature to create a Variable Bitrate Traffic Source and to test the fairness of WRR and WFQ. In the next frame, a snippet of Perl code follows. This script generates a random sequence of packets with variable size, included within a certain interval, that can be specified during the generation of the trace file.

VBR Traffic Generator

```
1  #!/usr/bin/perl -w
2  use strict;
3  use Tie::File;
4  my $CONST = 1_000_000; #number of desired record;
5  my $MICRO  = 1_000_000; #microseconds constant value
6
7  if (@ARGV != 4) {
8    print "use: perl $0 [wait_time] [min] [max] [filename]\n";
9    exit;
10 }
11
12 my ($wait_time, $min, $max, $filename) = @ARGV;
13 open my $file, '>', $filename;
14 binmode $file;
15
16 for my $step (1 .. $CONST) {
17     my $size = $min + rand ($max - $min);
18     my $record =  pack 'N*', $wait_time * $MICRO, $size;
19     print $file $record;
20 }
21
22 close $filename;
```

This script fetches command line interfaces parameters and it generates the number of packets desired. It is possible to specify the maximum and minimum packet size, the program will generate packets within this interval. Once generated, the record will be written in the *Tracefile* requested format. The datafile generated by this script is utilizable by the traffic source in the Tcl source file as in Table 5.2

As introduced in Section 1.1.1 the *fairness* concept gives us a measure of how much a scheduling algorithm can "well-distribute" the bandwidth to competitive

```
set tfile [new Tracefile]
$tfile filename mytracefile

set udp [new Agent/UDP]
$ns attach-agent $source $udp
$udp set fid_ 10
$ns simplex-connect $udp $sink1
set t1 [new Application/Traffic/Trace]
$t1 attach-agent $udp
$t1 attach-tracefile $tfile
```

Table 5.2: Tracefile setup

flows. Table 5.4 shows the results for the simulation performed with the scenario in Figure 5.2. Each simulation is done replacing the generic source with

- CBR: constant bitrate
- Poisson
- Our Variable Bitrate Traffic Source

The MPEG4 source sends data with a variable packet size while CBR and Poisson sources have a predefined 1000 bytes packet size. Every source sends its traffic to Destionation1 and the MPEG4 server sends to Destination2. The transmission rate, the weight of all the queues and also the transmission delay is the same for every source. The queues between the edges and the core router use *DsRed* queues (as described in 3.3.3), all the other queues use *DropTail* mechanism. Simulation length is 10.000 seconds.

The following results in Table 5.4 and 5.3 show that, using WFQ, it is possible to obtain a share of the bandwidth more accurate. In this case the outgoing link of the core router is fully loaded and represents a bottleneck for the network. These results are only for a fully loaded (125% of load on Core-Edge2 link) network but the simulations have been done also with different loads 50(%), 75(%), 100(%) and the bandwidth share in these cases is almost the same between the two algorithms. To obtain a fair behavior with WRR, we need to know a priori the average packet size of the variable bitrate flow, it represents a big problem. It should be noted that it would be possible to use another algorithm than WRR to attack the fairness problem, this algorithm is DRR (Section 2.4.1.2) and it presents a little

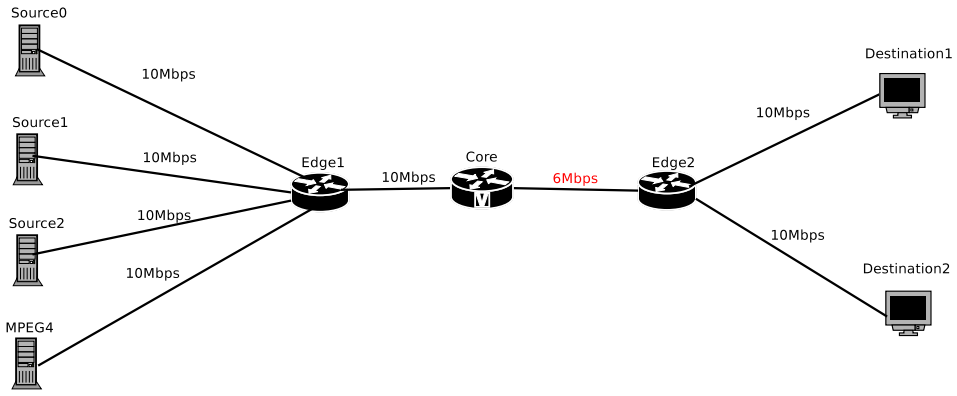


Figure 5.2: Network Scenario for Fairness Evaluation

modification in terms software implementation but it wastes area in FPGA and it does not guarantees a tight delay bound for multimedia traffic. This test has been carried out to make sure about the weighted fair queueing implementation in the simulator, in fact from the theory we know that WFQ is more fair than WRR and these results confirm the hypothesis [4].

Throughput (Mbit/s)	**WFQ**	**WRR**
MPEG4	2.860974929	2.311915367
CBR	2.861114501	3.410140991

Table 5.3: Fairness evaluation with MPEG4 and CBR

Throughput (Mbit/s)	**WFQ**	**WRR**
MPEG4	2.860978874	2.370964892
Poisson	2.861091613	3.334030151

Table 5.4: Fairness evaluation with MPEG4 and Poisson

Throughput (Mbit/s)	**WFQ**	**WRR**
MPEG4	2.960876420	2.460978874
VBR	2.971314601	3.530543925

Table 5.5: Fairness evaluation with MPEG4 and our VBR

Another test has been carried to make a comparison between WRR and WFQ. In these simulations, the sources in the scenario are a CBR and a VBR. The avail-

able bandwidth on the bottleneck link is $4.5 Megabit/s$. The scenario is presented in Figure 5.3.

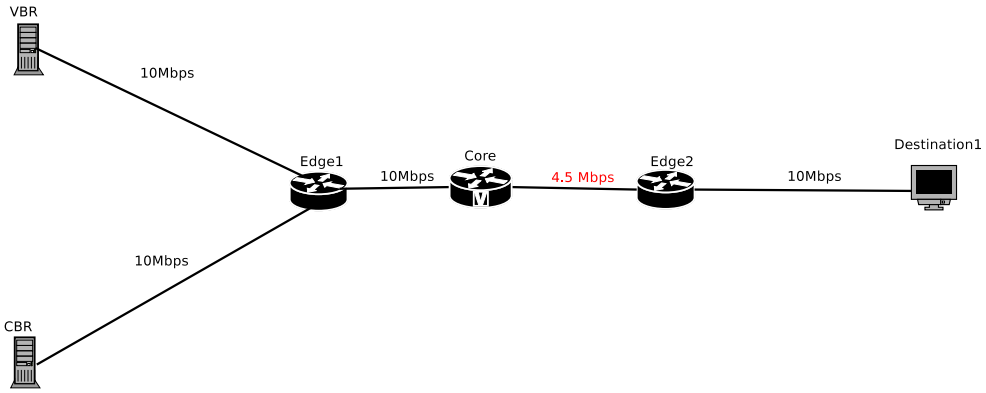


Figure 5.3: Fairness evaluation with VBR and CBR sources

In the previous tests, the priority of the queues was the same, then the scheduler didn't give privileges to a kind of traffic or the other one. Now the queue weight is set up giving priority to the VBR traffic and the result about the bandwidth utilization is shown in Table 5.6.

Throughput (Mbit/s)	**WFQ**	**WRR**	**Queue Weight**
CBR	1.430978874	1.548876420	2
VBR	2.861543925	2.744314601	4

Table 5.6: Fairness evaluation with CBR and our VBR

It is possible to note that using WFQ, the ratio between the bandwidth utilization is kept better than using WRR. We have the same behavior for the end-to-end delay as shown in Table 5.7.

Delay (sec)	**WFQ**	**WRR**	**Queue Weight**
CBR	0.248	0.230	2
VBR	0.128	0.126	4

Table 5.7: Delay evaluation with CBR and our VBR

5.4 Simulation Scenario

Two main simulation sessions have been set up to study the behavior of the two schedulers and how some factors, like traffic load and queue weight, influence their performance:

Variable Traffic Load: the effect of changing the traffic load in the link between the core router and the second edge has been studied. The impact of this change on the metrics of interest has been evaluated.

Variable Queue Weight: the effect of changing the queue size of the MPEG4 class has been studied. The impact of this change on the metrics has been evaluated as in the previous case.

In both these sets, the simulations are carried several times changing the random seed in the simulator each time. The average value for every metric has been calculated for every run. Later for these means, the mean over all the runs has been calculated and its 95% confidence interval too. Table 5.8 and 5.9 summarizes the simulation plan.

Variable Traffic Load				
Traffic	MPEG4	CBR	Poisson	
Netload	76.5%	86.5%	100%	110%

Table 5.8: Simulation Plan 1

Variable Class Weight	
MPEG4	weight variation
CBR	fixed
Poisson	fixed

Table 5.9: Simulation Plan 2

5.5 Traffic Load Variation

A set of simulations to test the performance of WFQ and WRR has been carried out. These simulations compare the two scheduling algorithm varying the traffic load on the link of the core router. The scenario with two sending nodes is the following: a MPEG4 source and another source, actually we alternate it with a Constant Bit Rate source and a Poisson source. There are also a couple of edge router and a core router in the middle. Each link has the same capacity (10 Mbps)

and the outgoing link for the core router has been set up at 6 Mbps creating a bottleneck for the transfer. The transmission delay is 5ms. Each source starts to send at the same time and they send the traffic to a unique sink. Figure 5.4 illustrates the network scenario. This simulation runs in nine main sessions, each of which consists of eight simulations to calculate confidence interval; in every one of the eight main sessions the sending rate of the MPEG4 node has been changed to vary the load on the core router link. The duration is 1000 seconds for each simulation.

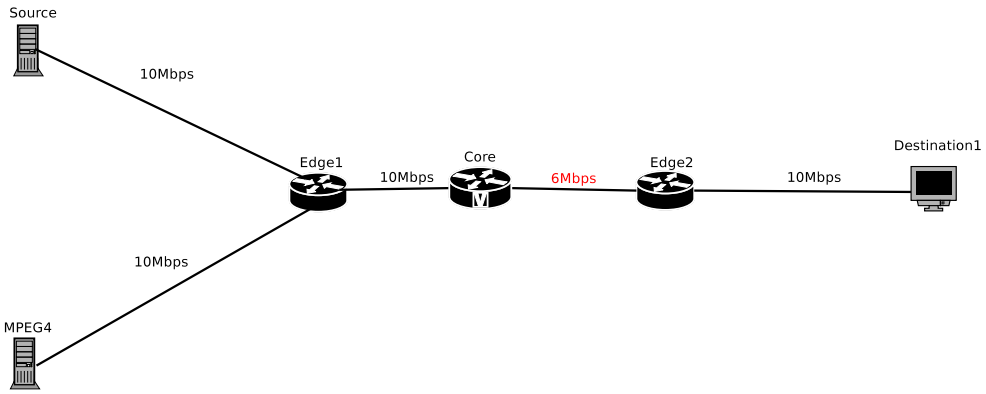


Figure 5.4: Network Scenario with load variation

5.5.1 Delay

Figure 5.5 shows the average delay experimented by the MPEG4 traffic in different load condition in the network. The comparison is made using one of the two scheduler, and later the other one. It should be noted that the WFQ scheduler treats this stream better when the traffic load reaches a fully loaded point. The other kind of traffic in this scenario is a constant bitrate. The advantage using sorted priority algorithms like WFQ instead WRR is that, in the former, the delay (maximum but also the average) is proportional to the allocated rate, in the latter, the proportionality is not so clear, because in the delay computation there a fix term (the round period) dominating the term depending by the rate.

The same set of simulations has been repeated changing the other kind of traffic, in this set, Poisson source traffic has been used. The result is shown in Figure 5.6.

The trend of delay is almost the same and it confirms the hypothesis also using a poissonian traffic source competing for the resource allocation with the multimedia stream.

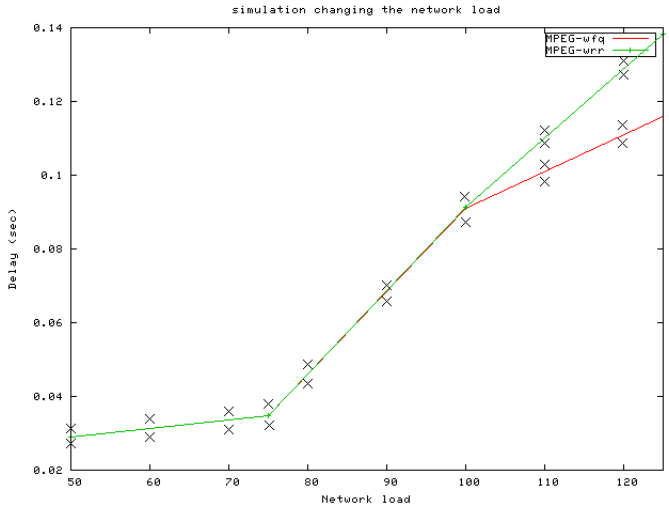


Figure 5.5: MPEG4 delay variation for different traffic load with CBR

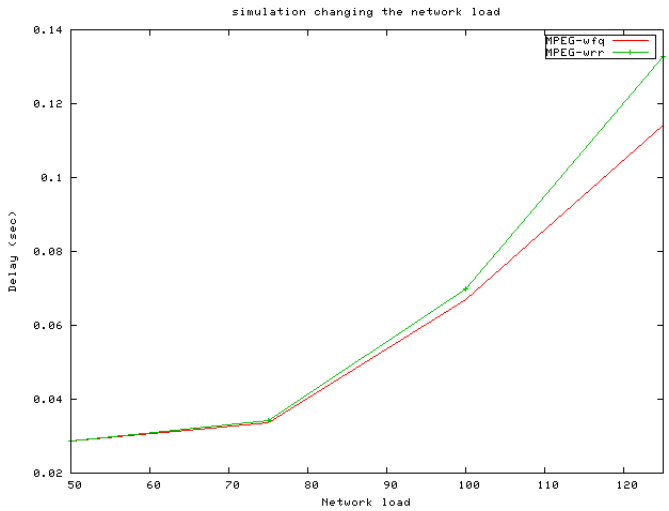


Figure 5.6: MPEG4 delay variation for different traffic load with Poisson

5.5.2 Jitter

In the same set of simulations, the jitter has been evaluated too. Figure 5.7 and 5.8 confirm how after the link saturation, Weighted Fair Queueing gives a better jitter to the multimedia traffic compared with the Weighted Round Robin one.

This parameter is very important for streams like the MPEG4, because in a video transmission, experimenting a big difference between the frame arrival frequency is could be unacceptable.

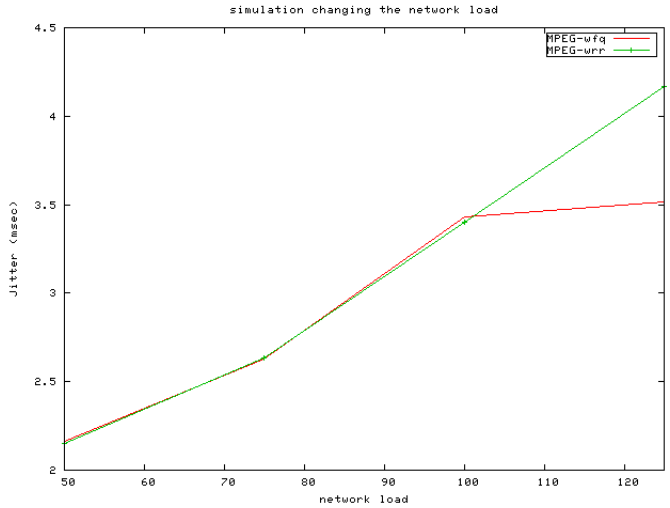


Figure 5.7: MPEG4 jitter variation for different traffic load with CBR

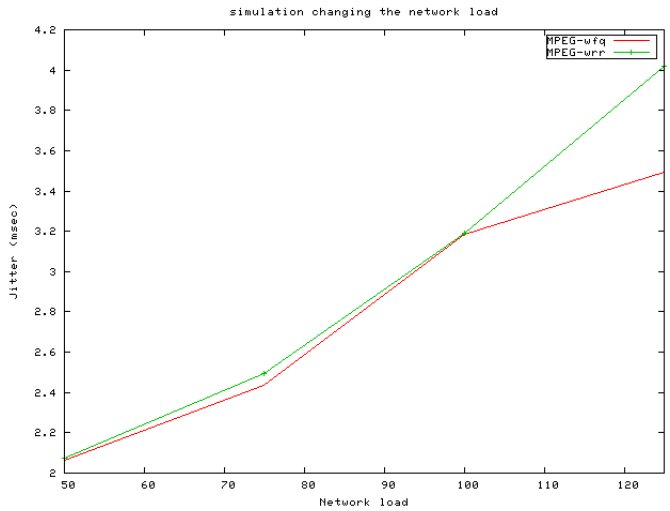


Figure 5.8: MPEG4 jitter variation for different traffic load with Poisson

5.6 Queue Weight Variation

An alternative simulation scenario has been created to evaluate the impact of changing the queue size of the multimedia stream, keeping the weights for the other queues fixed. This scenario is presented in Figure 5.9 and the simulations have been repeated with two different kind of traffics, CBR and Poisson and of course the multimedia stream. There is a bottleneck on the link between the core router and the second edge node, due to the fact that the sending rate of all the sources is calculated to fully utilize that link.

This set of simulations compares the impact of the choice of the scheduler on

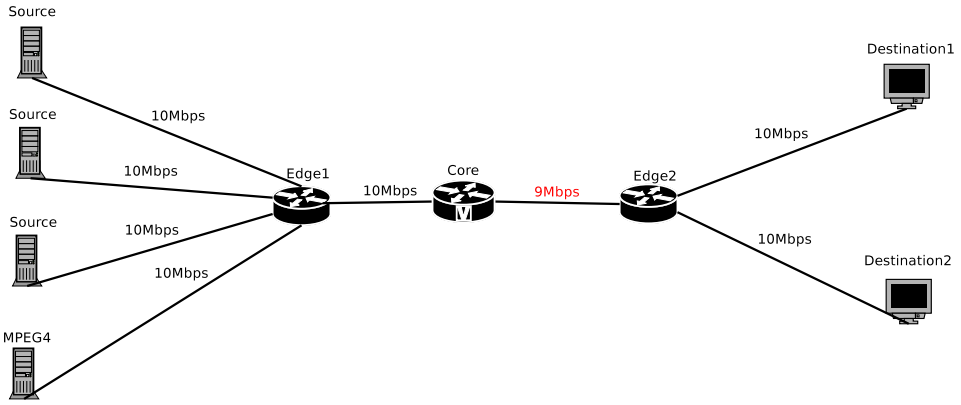


Figure 5.9: Simulation Scenario with Queue Weight Variation

the performance of the MPEG4 stream, in fact the same analysis is carried out with WFQ and WRR. The queue weight of the multimedia traffic has been varied starting from a condition of disadvantage, with respect to the other 3 streams, arriving to a condition of very big vantage. A queue weight represents the class that a flow belongs to. For the traffic source competing against the multimedia node, this class is fixed and set up to the value as in Table 5.10, instead for the MPEG4, it has been varied starting from 1 to 12.

	Weight											
Source 1	4											
Source 2	6											
Source 3	8											
MPEG4	1	2	3	4	5	6	7	8	9	10	11	12

Table 5.10: Queue Weights

5.6.1 Delay

The result of this set of simulations has been plotted in Figure 5.10, in this case the competing flows are CBR while in Figure 5.11 the competing flows are poissonian. These figures plot the trend of the delay for the multimedia stream each time that its class is changed. At a qualitative level, the trend between the two simulations is the same apart the region in which the weight's value is 1 and 4. This probably happens because the drop percentage is very high when the MPEG4 class is low so the delay behavior could be very random.

This figures also show that the zone in which the performance gain is maximum is [4 - 8].

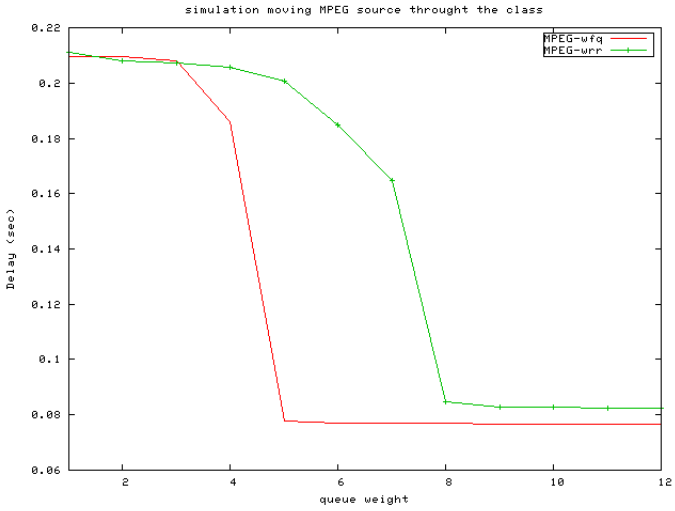


Figure 5.10: MPEG4 delay variation for different queue weight with CBR

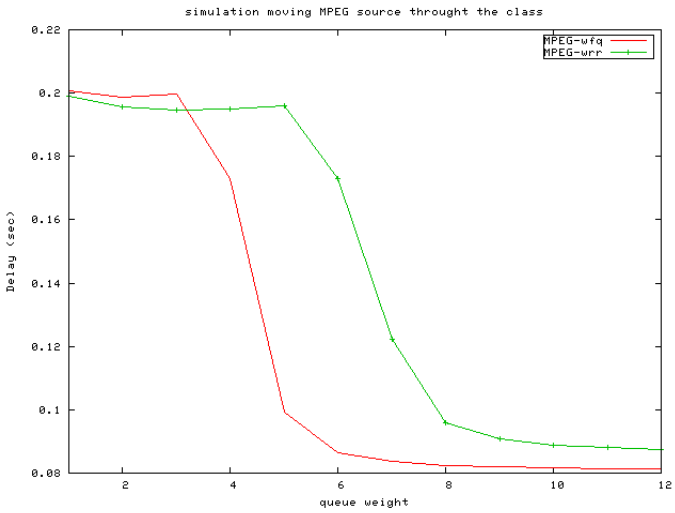


Figure 5.11: MPEG4 delay variation for different queue weight with Poisson

This investigation can help to understand some criteria to address the problem of the correct choice of the reconfiguration time. This means that the area of reconfigurability in terms of queue weights can be characterized. Having the necessity of to handle more streams, the scheduler algorithm could be switched from WFQ to WRR when the multimedia stream does not pose too strict delay requirements. For example, unless the choice of the weight (for MPEG4) is within the area [4 - 8], could be more suitable to utilize WRR than WRR because the gain in terms of delay is not so high compared to the number of streams that would be possible to handle adopting the other solution.

5.6.2 Jitter

For the jitter, the considerations are analogous, in the range [4 - 8], WFQ outperforms WRR and it should be also noted that assigning low weight to the MPEG4 stream, as it is possible to see from Figure 5.12 and 5.13, could result in an unstable behavior of the arrive frequency on the destination.

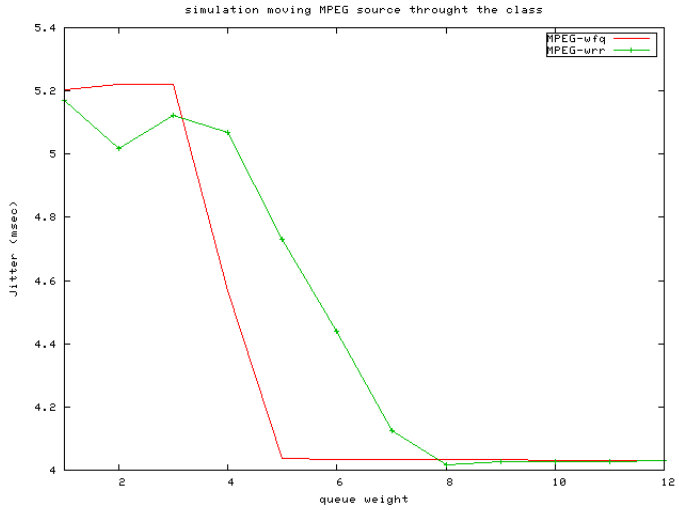


Figure 5.12: MPEG4 jitter variation for different queue weight with CBR

Starting from a certain threshold (queue weight = 8) the jitter experimented by the flow using WFQ or WRR tends to be the same, as shown in Figure 5.12. The same trend is experimented by the multimedia traffic in case the competing flows are poissonian as Figure 5.13 points out.

The surprising trend of the curve between [1 - 4] could be imputable to the difference between the drop percentage experimented using one scheduler or the other. With WFQ scheduler, in that range, the drop numbers are less than using WRR, this because WFQ share better the bandwidth, and consequently the number of packets that can be sent is greater because the average packet size for the multimedia traffic is less than that of CBR and Poisson (that are fixed and greater). This factor may influence the average delay, resulting in an ambiguous trend in that area. To address this problem a greater queue size (in terms of number of packets that can be enqueued in the buffer) could be used, depending on the implementation; this solution can use area or memory in the device and this has to be taken into account.

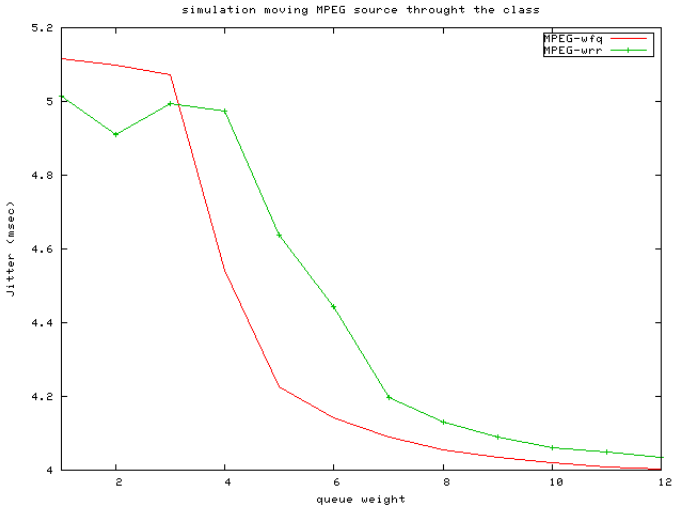


Figure 5.13: MPEG4 jitter variation for different queue weight with Poisson

5.7 Splitting Traffic

This investigation is finalized to understand and find a tradeoff between the use of FPGA space and performance requirements to meet having the possibility of change the scheduler. As we have shown in the previous sections, using a fair queueing algorithm brings more benefits than a round robin in term of delay, jitter and throughput. These benefits are paid from the fact that it is possible to handle fewer number of streams compared with the use of a round robin scheduler, but we also know that using such kind of scheduler we can dispose of more area in FPGA. What we can do with this available area? As we said before, it is possible to think to handle a number of streams greater without gain in performance, but it is also possible to think to split the traffic present in a queue in more queues, because a queue in a router has usually a limited buffer to store the packets. This could bring a several effects like:

- reduction of packet drop;
- performance gain in term of delay.

The following table shows the results of a set of simulations made running WRR as scheduler. There are 12 MPEG4 sources sending traffic to a destination, there are also other three constant bit rate sources. These simulations have been repeated splitting the multimedia traffic in two queues and not splitting it; also the queue weight for this kind of traffic has been changed consistently: in case of splitting, each queue has received the half value compared with the case of not split. The queue weights for the CBR sources have been kept fixed and they are set up to 4, 6 and 8.

Delay (sec)		
Queue Weight	**Not Splitted**	**Splitted**
4	0.109042510	0.136568848
8	0.055353869	0.043299841
12	0.054353227	0.042517102

Table 5.11: Delay splitting the traffic or not

Also the packet drop is improved because when the queue weight for MPEG4 is set up to 4, the number of drop in the set of simulations without splitting are more then the number of drop with split. This effect happens because having the same size for the buffer in a queue, if we split the traffic in two queues is like if we were using a queue with a double size in term of number of packet that can be stored. These results lead to another challenge, is it possible to use more queues to handle a particular stream when there is available space in the FPGA. This fact should be investigated and the problem addressed on the hardware side.

5.8 Final Reflections

These simulations compared two possible scheduling algorithm to be used in a router: Weighted Round Robin (WRR) and Weighted Fair Queuing (WFQ). The first solution provides, having the same available area in FPGA, the possibility to handle more streams than the second one or to save power switching off some unutilized area. This approach lacks of support to QoS that can be obtained changing it with WFQ. The second configuration (WFQ) permits a more accurate provisioning of the required performance, paid with a bigger area utilization. WFQ combined with a stream constrained by a leaky bucket can offer the possibility to easily calculate a tight bound on the end-to-end delay.

Simulations results do not show unexpected behavior, nonetheless they permit to draw attention to the different properties of the two solutions. These allow to point out some guidelines which would give an help to a network administrator in choosing the configuration which best fits to the requirements of his scenario.

end-to-end delay The two approaches fit differently the delay requirements. Using WFQ with a leaky bucket constrained stream, it is possible to calculate a bound for this metric and provide it. WRR can serve a greater number of streams despite of worse performance. In situation of reconfiguration, changing from WRR to WFQ can recover and improve the trend for the end-to-end delay but attention should be posed on the problem of how to handle the number of flows that the scheduler would leave out using WFQ after the reconfiguration.

jitter If a multimedia traffic has to be served, the choice of WFQ is mandatory in situation of high traffic load. Using the reconfiguration feature, the choice of the disciplines has to be taken case by case because depending on the level of privilege desired, also WRR could give good performance and permitting to handle a greater number of streams.

throughput WRR suffers variable bit rate streams because it does not take into account the packet size, if the average packet size is not known a priori, it is very difficult to obtain an accurate share of the bandwidth. This problem can be solved using a fair queuing algorithm in which this problem is automatically considered by the virtual finish function.

5.9 Summary

In this chapter the methodology utilized to evaluate the simulations and obtain realistic results has been explained. A section has been dedicated to the fairness evaluation illustrating the traffic source that has been developed to test the scheduling algorithms. Furthermore, the simulation plans have been presented describing the two approach utilized: *traffic load variation* and *queue weight variation*, in the former the load of the link between the core and the edge router has been varied starting from a low level arriving to a fully utilization; in the latter the queue weight for the multimedia stream has been modified simulation by simulation keeping fixed the weights of the competing streams. For each set of simulations, the end-to-end delay and the jitter has been evaluated and the results discussed. The results confirm the hypothesis on the behavior of the two investigated scheduling algorithm. The last section of the chapter has introduced the idea of *splitting traffic*, some simulations have been carried out to understand the response of the traffic to this idea and it should be investigated in the next investigations.

Conclusions and Future Research

6

This thesis aimed to study and compare different scheduling algorithms for the choice of the best disciplines in a QoS aware network router using an FPGA.

6.1 Summary

In Chapter 2, the *round-robin* and the *fair queuing* families of algorithms were presented. A classification of these scheduling disciplines has been provided and each algorithm has been described. The *FIFO* and *priority* scheduler are unsuitable to provide fairness or QoS guarantees. The priority scheduler can also lead to starvation if the high priority classes have a large amount of traffic to send. RR, WRR, and DRR have been described as representative of the *round-robin* families. These algorithms are simple to implement but they do not provide QoS guarantees and in certain situations they cannot share correctly the bandwidth to the competing streams. The *fair queuing* family was introduced by presenting the theoretical model of General Processor Sharing (GPS) from which all the approximations are based on. The most well-known of this algorithm approximation have been presented: weighted fair queueing (WFQ) and worst case weighted fair queuing (WF^2Q+). They are more complex to implement compared with the *round robin* algorithms.

In Chapter 3, the network simulator and all the environment utilized in the simulations has been introduced. It has been described how to create a simulation scenario and how to get results in NS-2. A description of the differentiated services architecture for IP QoS has been furnished together with the support offered by the simulator through the Nortel Diffserv module. The MPEG4 model has been described and some excerpts of code permitting to set up a multimedia traffic source, in the simulator, was presented. A short description of the competing streams utilized in the typical network scenario together with the MPEG4 stream has been provided, and also for them some excerpts of code have been explained. The last part of the chapter has illustrated the motivation bringing to the choice of utilize reconfigurable hardware and a cost analysis of the chosen scheduling algorithms in hardware has been presented.

In Chapter 4, the *monitor agent* implemented to trace the simulations results has been presented. This approach permits to save a lot of disk writes without incurring in a considerable slow down of the simulations. It also permits to avoid the tedious parsing of the results data file generated by the simulator. The module

to support the reconfiguration in the simulator has been illustrated describing the main reconfiguration idea and the approach utilized. Some excepts of code have been commented and the results of the utilization of this modification has been presented. The module developed to add the possibility to simulate the reconfiguration delay (typical of an hardware device during the change of the scheduling algorithm) has been described. At the end of the chapter, the weighted fair queuing implementation has been presented starting from a theoretical definition arriving to the code realization.

In Chapter 5, the methodology utilized to evaluate the simulations and obtain realistic results has been explained. A section has been dedicated to the fairness evaluation illustrating the traffic source that has been developed to test the scheduling algorithms. Furthermore, the simulation plans have been presented describing the two approaches utilized: *traffic load variation* and *queue weight variation*, in the former the load of the link between the core and the edge router has been varied starting from a low level arriving to a fully utilization; in the latter the queue weight for the multimedia stream has been modified simulation by simulation keeping fixed the weights of the competing streams. For each set of simulations, the end-to-end delay and the jitter has been evaluated and the results discussed. The results confirm the hypothesis on the behavior of the two investigated scheduling algorithm: WFQ outperforms WRR in terms of end to end delay, jitter and throughput but it is more expensive than it at a computational level. The last section of the chapter has introduced the idea of *splitting traffic*, some simulations have been carried out to understand the response of the traffic to this idea and it should be investigated in future investigations.

6.2 Main contributions

A theoretical analysis on several algorithms has been provided. This work would provide a network operator with some guidelines helping him on the choice of the scheduling discipline which best fit the needs of his environment. The support for the scheduler reconfiguration has been added in the simulator as well as the support for the reconfiguration delay of a real device. The *splitting traffic* idea has been introduced and the first investigation has been carried on it. The tradeoff in the choice of WFQ or WRR has been evaluated and a test suite to automatize simulations and collect results has been created.

6.3 Future Research Directions

Some suggestions can be pointed out to inspire future investigations:

- the next research step should study the problem of reconfiguration in hardware implementing this mechanism in a real device;

- a set of measures on the reconfiguration delay needs to be done;
- the possibility of a queue splitting is another important challenge to investigate both software and hardware side.

Bibliography

[1] Chipalkatti, J. Kurose, and Townsley, *"Scheduling policies for Real-Ttime and non Real-Ttime Traffic in Statical Multiplexer"*, Proceedings of IEEE INFOCOM (1989), 774–783.

[2] R. Cruz, *"A Calculus for Network Delay, part i:network elements in isolation."*, IEEE Transaction on Information Theory **1** (1991), 114–131.

[3] Hamid Fadishei, Morteza Saheb Zamani, and Masoud Sabaei, *"A Novel Reconfigurable Hardware Architecture for IP Address Lookup"*, ANCS (Symposium on Architecture for Networking and Communications Systems) (2005), 81–90.

[4] Kevin Fall, *"Scheduling Best-Effor and Guaranteed Connections"*, 1999, URL reference: `http://www.cs.berkeley.edu/~kfall/EE122/lec27/`.

[5] R. Koenen, *"MPEG-4 Overview"*, IEEE SPECTRUM **36** (1999), no. 2.

[6] R. Krishnamurthy, S. Yalamanchili, K. Schwan, and R. West, *"Share Streams: a scalable architecture and hardware support for high-speed qos packet schedulers"*, Field-Programmable Custom Computing Machines, 2004. FCCM 2004. 12th Annual IEEE Symposium **20**.

[7] A. Matrawy L., I. Lambadaris, and C. Huang, *"MPEG-4 Traffic Modeling Using the Transform Expand Sample Methodology"*, Networked Appliances, 2002. Gaithersburg.Proceedings. IEEE 4th International Workshop on (2002), 249–256.

[8] Pierre L'Ecuyer, *"Good parameters and implementations for combined multiple recursive random number generators"*, Operations Research (2001).

[9] Pierre L'Ecuyer, Richard Simard, E. Jack Chen, and W. David Kelton, *"An object-oriented random number package with many long streams and substreams"*, Operations Research (2001).

[10] J. Lu and R. Robotham, *"On the implementation of weighted fair queuing in high speed networks"*, Electrical and Computer Engineering, 2004. Canadian Conference **2** (2004), 809–813.

[11] C. McKillen and S. Sezer, *"A weighted fair queuing finishing tag computation architecture and implementation"*, SOC Conference, 2004. Proceedings. IEEE International (2004), 270–273.

[12] J. Nagle, *"On Packet Switches with Infinite Storage"*, IEEE Transaction on Communications **35** (1987), 435–438.

[13] Parekh and Gallager, *"A Generalized Processor Sharing Approach to Flow Control in Integrated Services Networks: The Singlenode Case"*, ACM/IEEE Transactions on Networking (1993), 344–357.

[14] Fulvio Risso, *Quality of Service on Packet Switched Networks*, Ph.D. thesis, Politecnico di Torino, January 2000.

[15] L. Rohan and D. Taube, *"Weighted Round Robin Scheduling Module"*, 2001, URL reference: `http://www.arl.wustl.edu/~lockwood/class/cs535/project/fairqueue/index.html`

[16] A. Sayenko, *"Adaptive Scheduling for the QoS Supported Networks"*, Master's thesis, University of Jyvaskyla, 2005.

[17] F. Shallwani, J. Ethridge, P. Pieda, and M. Baines, *"A Network Simulator Differentiated Services Implementation"*, Open IP, Nortel Networks (2000).

[18] UCB/LBNL/VINT Network simulator NS-2 http://www.isi.edu/nsnam/ns/.

[19] Meina Song, Junde Song, and Hongwen Li, *"Implementing a High Performance Scheduling Discipline WF2Q+ in FPGA"*, IEEE CCECE Canadian Conference **1** (2003), 187–190.

[20] Wikipedia `http://en.wikipedia.org/wiki/`.

[21] Tom Murphy `http://www.cs.cmu.edu/~tom7/`.

[22] P. Vellore and R. Venkatesan, *"Scheduling Disciplines in Packet Switched Networks"*, IEEE Newfoundland Electrical and Computer Engineering Conference.

[23] ———, *"Performance Analysis of Scheduling Disciplines in Hardware"*, Electrical and Computer Engineering, 2004. Canadian Conference on **2** (2005), 715–718.

[24] R. West, K. Schwan, and C. Poellabauer, *"Scalable scheduling support for loss and delay constrained media streams"*, IEEE Real Time Technology and Application Symposium (1999).

[25] Ito Y., Tasaka S., and Ishibashi Y., *"Variably Weighted Round Robin Rueueing for Core IP Routers"*, Performance, Computing, and Communications Conference, 2002. 21st IEEE International (2002), 159–166.

[26] Sungwon Yi, Xidong Deng, G. Kesidis, and C. R. Das, *"Providing Fairness in Diffserv Architecture"*, IEEE Global Telecommunications Conference **2** (2002), 1435–1439.

MPEG-4 Source Code

A

MPEG-4 Source Code

```
/* -*- Mode:C++; c-basic-offset:8; tab-width:8;
 * indent-tabs-mode:t -*- *t
 * Copyright(c) Xerox Corporation 1997. All rights reserved.
 *
 * License is granted to copy, to use, and to make and to
 * use derivative works for research and evaluation purposes,
 * provided that Xerox is acknowledged in all documentation
 * pertaining to any such copy or derivative work. Xerox
 * grants  no other licenses expressed or implied. The Xerox
 * trade name should not be used in any advertising without
 * its written permission.
 *
 * XEROX CORPORATION MAKES NO REPRESENTATIONS CONCERNING
 * EITHER THE MERCHANTABILITY OF THIS SOFTWARE OR THE
 * SUITABILITY OF THIS SOFTWARE FOR ANY PARTICULAR PURPOSE.
 * The software is provided "as is" without express or
 * implied warranty of any kind.
 *
 * These notices must be retained in any copies of any part
 * of this software.
 */

/*
 * A source for MPEG4 traffic.
 * Procduces traffic that has the same marginal
 * distribution and autocorrelation function of a specified
 * mpeg4 trace.
 * A TES model is developed first to get the statistics and
 * then the statistics are used to generate the traffic
 */

#include <stdlib.h>
#include "random.h"
#include "trafgen.h"
```

```
#include "ranvar.h"

class Frame {
    public:
        Frame(const char*, const char*);
        //get the next frame size
        void Get_Frame(double &,int &);
    protected:
        RNG* rng_;
        //number of entries in the innovation CDF table
        int numEntry_inv_;
        //number of entries in the frame size CDF table
        int numEntry_hist_;
        //size of the CDF table (mem allocation)
        int maxEntry_;
        //CDF table of frame sizes (val_, cdf_)
        CDFentry* table_hist_;
        //CDF table of innovation (val_, cdf_)
        CDFentry* table_inv_;
        double value(double, CDFentry*, int );
        double interpolate(double x, double x1, double y1,
                double x2, double y2);
        int lookup(double u,CDFentry*,int);
        void loadCDF(const char* filename, CDFentry*& ,
                int &);
};

Frame::Frame(const char* file_1, const char* file_2) :
    maxEntry_(120),table_inv_(0), table_hist_(0){
    rng_ = RNG::defaultrng();
    loadCDF (file_1,table_hist_,numEntry_hist_);
    loadCDF (file_2,table_inv_,numEntry_inv_);
}

void Frame::Get_Frame (double &U, int &size){
    size = int(value(U, table_hist_,numEntry_hist_));
    double rnd = rng_->uniform(0,1);
    double w = value (rnd,table_inv_,numEntry_inv_);
    U = U + w;
    U = U - floor(U);
    //smoothing using eata = 0.5
    if (U<0.5) U = U/0.5;
```

```
    else U = (1-U)/0.5;
}

double Frame::value(double u, CDFentry *table_,
        int numEntry_){
    if (numEntry_ <= 0) return 0;
    int mid = lookup(u,table_,numEntry_);
    return interpolate(u, table_[mid-1].cdf_,
            table_[mid-1].val_, table_[mid].cdf_,
            table_[mid].val_);
}

double Frame::interpolate(double x, double x1, double y1,
        double x2, double y2){
    return (y1 + (x - x1) * (y2 - y1) / (x2 - x1));
}

int Frame::lookup(double u, CDFentry* table_,
        int numEntry_){
    // always return an index whose value is >= u
    int lo, hi, mid;
    if (u <= table_[0].cdf_)
        return 0;
    for (lo=1, hi=numEntry_-1; lo < hi; ) {
        mid = (lo + hi) / 2;
        if (u > table_[mid].cdf_)
            lo = mid + 1;
        else hi = mid;
    }
    return lo;
}

void Frame::loadCDF(const char* filename, CDFentry*& table_,
        int & numEntry_){
    FILE* fp;
    char line[256];
    CDFentry* e;
    fp = fopen(filename, "r");
    if (fp == 0)
        return ;
    if (table_ == 0)
        table_ = new CDFentry[maxEntry_];
```

```
    for (numEntry_=0; fgets(line, 256, fp); numEntry_++) {
        e = &table_[numEntry_];
        sscanf(line, "%lf %lf", &e->val_, &e->cdf_);
    }
    return;
}
////////////////////////////////////////////////////////////
class VIDEO_Traffic : public TrafficGenerator {
    public:
        VIDEO_Traffic();
        virtual double next_interval(int&);
        double u0;
        double rateFactor_;
    protected:
        RNG* rng_;
        const char* I_File_1;
        const char* I_File_2;
        const char* P_File_1;
        const char* P_File_2;
        const char* B_File_1;
        const char* B_File_2;
        char prev_frame_type, next_frame_type;
        double inter_frame_interval_ ;
        int GOP_count;
        virtual void start();
        virtual void timeout();
        //random seeds for the three frame type s
        double Ui, Up, Ub;
        Frame *i;
        Frame *p;
        Frame *b;
};

static class VIDEOTrafficClass : public TclClass {
    public:
        VIDEOTrafficClass() :
            TclClass("Application/Traffic/MPEG4") {}
        TclObject* create(int, const char*const*) {
            return (new VIDEO_Traffic());
        }
} class_video_traffic;
```

```
VIDEO_Traffic::VIDEO_Traffic(){
    bind("initialSeed_",&u0);
    bind("rateFactor_",&rateFactor_);
    I_File_1 = "./video_model/Imodel_hist_1";
    I_File_2 = "./video_model/Imodel_inv_1";
    P_File_1 = "./video_model/Pmodel_hist_1";
    P_File_2 = "./video_model/Pmodel_inv_1";
    B_File_1 = "./video_model/Bmodel_hist_1";
    B_File_2 = "./video_model/Bmodel_inv_1";
    rng_ = RNG::defaultrng();
}

void VIDEO_Traffic::start(){
    Ui = u0;
    size_ = 0;
    next_frame_type = 'I';
    prev_frame_type = ' ';
    //first GOP is 10 frames only
    GOP_count = 3;
    //30 frame/sec
    inter_frame_interval_ = 1.0/30.0;
    i = new Frame (I_File_1,I_File_2);
    p = new Frame (P_File_1,P_File_2);
    b = new Frame (B_File_1,B_File_2);
    if (agent_) agent_->set_pkttype(PT_VIDEO);
    running_ = 1;
    timeout();
}

void VIDEO_Traffic::timeout(){
    double frame_in_bytes;
    if (! running_)
        return;
    /* figure out when to send the next one */
    nextPkttime_ = next_interval(size_);
    /* send a packet */
    frame_in_bytes = rateFactor_*size_/8 ;
    agent_->sendmsg(frame_in_bytes);
    /* schedule it */
    if (nextPkttime_ > 0)
        timer_.resched(nextPkttime_);
    else
```

```
        running_ = 0;
}

double VIDEO_Traffic::next_interval(int& size){
    if (next_frame_type == 'I'){
        i->Get_Frame(Ui, size);
        Up = Ui;
        Ub = Ui;
        if (prev_frame_type == 'B')
            next_frame_type = 'B';
        else next_frame_type = 'P';
        prev_frame_type = 'I';
    } // end if == I
    else if (next_frame_type == 'P'){
        p->Get_Frame (Up, size);
        next_frame_type = 'B';
        prev_frame_type = 'P';
    } // end if == P
    else if (next_frame_type == 'B'){
        b->Get_Frame (Ub, size);
        if (prev_frame_type != 'B') next_frame_type = 'B';
        else if (GOP_count == 12){
            next_frame_type = 'I'; GOP_count = 0;}
        else next_frame_type = 'P';
            prev_frame_type = 'B';
    } // end if == B
    GOP_count++ ;
    return(inter_frame_interval_);
}
```

Diffserv Module

B

Diffserv Module

```
/*
 * Copyright (c) 2000 Nortel Networks
 * All rights reserved.
 *
 * Redistribution and use in source and binary forms, with
 * or without modification, are permitted provided that the
 * following conditions are met:
 * 1. Redistributions of source code must retain the above
 *    copyright notice, this list of conditions and the
 *    following disclaimer.
 * 2. Redistributions in binary form must reproduce the
 *    above copyright notice, this list of conditions and
 *    the following disclaimer in the documentation and/or
 *    other materials provided with the distribution.
 * 3. All advertising materials mentioning features or use
 *    of this software must display the following
 *    acknowledgement:
 *    This product includes software developed by Nortel
 *    Networks.
 * 4. The name of the Nortel Networks may not be used
 *    to endorse or promote products derived from this
 *    software without specific prior written permission.
 *
 * THIS SOFTWARE IS PROVIDED BY NORTEL AND CONTRIBUTORS
 * ``AS IS'' AND ANY EXPRESS OR IMPLIED WARRANTIES,
 * INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES
 * OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
 * ARE DISCLAIMED.  IN NO EVENT SHALL NORTEL OR
 * CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT,
 * INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL
 * DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF
 * SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR
 * PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND
 * ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT
```

```
 * LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE)
 * ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN
 * IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
 *
 * Developed by: Farhan Shallwani, Jeremy Ethridge
 *               Peter Pieda, and Mandeep Baines
 * Maintainer: Peter Pieda <ppieda@nortelnetworks.com>
 */

/*
 * dsred.h
 *
 * The Positions of dsREDQueue, edgeQueue, and coreQueue in
 * the Object Hierarchy.
 *
 * This class, i.e. "dsREDQueue", is positioned in the class
 * hierarchy as follows:
 *
 *              Queue
 *                |
 *            dsREDQueue
 *
 *
 *   This class stands for "Differentiated Services RED
 *   Queue". Since the original RED does not support
 *   multiple parameters, and other functionality
 *   needed by a RED gateway in a Diffserv architecture,
 *   this class was created to support the desired
 *   functionality. This class is then inherited by two
 *   more classes, moulding the old hierarchy as follows:
 *
 *
 *              Queue
 *                |
 *            dsREDQueue
 *            |         |
 *     edgeQueue     coreQueue
 *
 *
 * These child classes correspond to the "edge" and "core"
 * routers in a Diffserv architecture.
 *
```

```
 */
//Modification for the support to Weighted Fair Queueing is
//taken from the work of Alexander Sayenko, see the
//Bibliografy for more information


#ifndef dsred_h
#define dsred_h
// need RED class specs (edp definition, for example)
#include "red.h"
// need Queue class specs
#include "queue.h"
#include "dsredq.h"

/***************** WFQ addition ***********************/
#include "wfq-list.h"
/***************** WFQ addition ***********************/


/* The dsRED class supports the creation of up to MAX_QUEUES
 * physical queues at each network device, with up to
 * MAX_PREC virtual queues in each queue. */

// maximum number of physical RED queues
#define MAX_QUEUES 8
// maximum number of virtual RED queues in one physical
// queue
#define MAX_PREC 3
// maximum number of code points in a simulation
#define MAX_CP 40
// default mean packet size, in bytes, needed for RED
// calculations
#define MEAN_PKT_SIZE 1000

//enum schedModeType {schedModeRR, schedModeWRR,
//  schedModeWIRR, schedModePRI};
/***************** WFQ addition ***********************/
enum schedModeType {schedModeRR, schedModeWRR,
    schedModeWIRR, schedModePRI,schedModeWFQ};
/***************** WFQ addition ***********************/

#define PKT_MARKED 3
```

```
#define PKT_EDROPPED 2
#define PKT_ENQUEUED 1
#define PKT_DROPPED 0


/*-----------------------------------------------------------
struct phbParam
    This struct is used to maintain entries for the PHB
    parameter table, used to map a code point to a physical
    queue-virtual queue pair.
-----------------------------------------------------------*/
struct phbParam {
   int codePt_;
   int queue_;  // physical queue
   int prec_;   // virtual queue (drop precedence)
};

struct statType {
        long drops;        // per queue stats
        long edrops;
        long pkts;
        long valid_CP[MAX_CP];  // per CP stats
        long drops_CP[MAX_CP];
        long edrops_CP[MAX_CP];
        long pkts_CP[MAX_CP];
};


/***************** WFQ addition ***********************/
class dsREDQueue;

class WFQdsHandler : public Handler
{
 public:
   WFQdsHandler (dsREDQueue* queue) : q(queue) {};
   void handle (Event* e);

 protected:
   dsREDQueue* q;
};
/***************** WFQ addition ***********************/
```

```
/*-----------------------------------------------------------------
class dsREDQueue
    This class specifies the characteristics for a Diffserv
    RED router.
-----------------------------------------------------------------*/
class dsREDQueue : public Queue {
 public:

/***************** WFQ addition ***********************/
  void WFQdequeueGPS (Event*);
/***************** WFQ addition ***********************/

  dsREDQueue();
  // interface to ns scripts
  int command(int argc, const char*const* argv);

 protected:

/***************** WFQ addition ***********************/
  void WFQenqueue (Packet*, int);
  void WFQscheduleGPS ();
  void WFQdequeuePGPS ();
  void WFQupdateSum ();

  Event*          wfq_event;
  WFQdsHandler    wfq_hand;

  int GPS_idle;
  double virt_time;
  double last_vt_update;
  double sum;
  double safe_limit;

  double finish_t [MAX_QUEUES];
  unsigned int B [MAX_QUEUES];

  List<int> GPS_list;
  List<int> PGPS_list;

  double bandwidth;
/***************** WFQ addition ***********************/
  // the physical queues at the router
```

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

//DAN-s
//count the number of video packet (id=4)
//in the core router
int num_video_packet; //number of video packet in the net
int num_other_packet; //number of other packet in the net
//last time that we have seeing a video packet
double last_seen_vp;
```

```
  double initial;
  //DAN-e

  void reset();
  // used so flowmonitor can monitor early drops
  void edrop(Packet* p);
  // enques a packet
  void enque(Packet *pkt);
  // deques a packet
  Packet *deque(void);
  // given a packet, extract the code point marking from
  // its header field
  int getCodePt(Packet *p);
  // round robin scheduling dequing algorithm
  int selectQueueToDeque();
  // looks up queue and prec numbers corresponding
  // to a code point
  void lookupPHBTable(int codePt, int* queue, int* prec);
  // edits phb entry in the table
  void addPHBEntry(int codePt, int queue, int prec);
  void setNumPrec(int curPrec);
  void setMREDMode(const char* mode, const char* queue);
  void printStats(); // print various stats
  double getStat(int argc, const char*const* argv);
  void printPHBTable();  // print the PHB table
  // Sets the schedular mode
  void setSchedularMode(const char* schedtype);

  // Add a weigth to a WRR or WIRR queue
  void addQueueWeights(int queueNum, int weight);
  // Add a maxRate to a PRI queue
  void addQueueRate(int queueNum, int rate);

  void printWRRcount();          // print various stats

  // apply meter to calculate average rate of a PRI queue
  // Modified by xuanc(xuanc@isi.edu) Oct 18, 2001,
  // referring to the patch contributed by
  // Sergio Andreozzi <sergio.andreozzi@lut.fi>
  void applyTSWMeter(int q_id, int pkt_size);
};
```

```
#endif
```

Diffserv Module

```
/*
 * Copyright (c) 2000 Nortel Networks
 * All rights reserved.
 *
 * Redistribution and use in source and binary forms, with
 * or without modification, are permitted provided that the
 * following conditions are met:
 * 1. Redistributions of source code must retain the above
 *    copyright notice, this list of conditions and the
 *    following disclaimer.
 * 2. Redistributions in binary form must reproduce the
 *    above copyright notice, this list of conditions and
 *    the following disclaimer in the documentation and/or
 *    other materials provided with the distribution.
 * 3. All advertising materials mentioning features or use
 *    of this software must display the following
 *    acknowledgement:
 *    This product includes software developed by Nortel
 *    Networks.
 * 4. The name of the Nortel Networks may not be used
 *    to endorse or promote products derived from this
 *    software without specific prior written permission.
 *
 * THIS SOFTWARE IS PROVIDED BY NORTEL AND CONTRIBUTORS
 * ``AS IS'' AND ANY EXPRESS OR IMPLIED WARRANTIES,
 * INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES
 * OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
 * ARE DISCLAIMED.  IN NO EVENT SHALL NORTEL OR
 * CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT,
 * INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL
 * DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF
 * SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR
 * PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND
 * ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT
 * LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE)
 * ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN
 * IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
 *
 * Developed by: Farhan Shallwani, Jeremy Ethridge
```

```
 *                Peter Pieda, and Mandeep Baines
 * Maintainer: Peter Pieda <ppieda@nortelnetworks.com>
 */
//Modification for the support to Weighted Fair Queueing
//is taken from the work of Alexander Sayenko, see the
//Bibliografy for more information

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

/***************** WFQ addition ***********************/
#include <math.h>
/***************** WFQ addition ***********************/


/*-----------------------------------------------------------
dsREDClass declaration.
    Links the new class in the TCL heirarchy.  See "Notes
    And Documentation for NS-2."
-----------------------------------------------------------*/
static class dsREDClass : public TclClass {
public:
        dsREDClass() : TclClass("Queue/dsRED") {}
        TclObject* create(int, const char*const*) {
                return (new dsREDQueue);
        }
} class_dsred;

/***************** WFQ addition ***********************/
inline void WFQdsHandler::handle (Event* e) {
    q->WFQdequeueGPS(e);
}
/***************** WFQ addition ***********************/

/*-----------------------------------------------------------
dsREDQueue() Constructor.
```

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

/***************** WFQ addition ***********************/
dsREDQueue::dsREDQueue() : de_drop_(NULL), link_(NULL),
    wfq_hand (this) {

  wfq_event = 0;

  virt_time = last_vt_update = sum = 0;
  GPS_idle = 1;
  safe_limit = 0.000001;
/***************** WFQ addition ***********************/

  bind("numQueues_", &numQueues_);
  bind_bool("ecn_", &ecn_);
  int i;

  //DAN-s
  //init the control paramters
  num_video_packet = 0;
  num_other_packet = 0;
  last_seen_vp = 0;
  initial = 0;
  //DAN-e

  numPrec = MAX_PREC;
  schedMode = schedModeRR;

  for(i=0;i<MAX_QUEUES;i++){
    queueMaxRate[i] = 0;
    queueWeight[i]=1;

/***************** WFQ addition ***********************/
    finish_t[i] = 0;
    B[i] = 0;
/***************** WFQ addition ***********************/
```

```
  }

  queuesDone = MAX_QUEUES;
  // Number of entries in PHB table
  phbEntries = 0;

  reset();

}

/***************** WFQ addition ***********************/

void dsREDQueue::WFQupdateSum ()
{
 sum = 0;

 for (int i=0; i < numQueues_; i++)
   if (B[i])
     sum += queueWeight[i];
}

/*-----------------------------------------------------------
  | Function    : WFQenqueue()
  | Description : updates virtual time variables, inserts
  |               control data in
  |               GPS and PGPS Lists, invokes the
  |               scheduling of next departure
  | Changes     :
  | Input       : Packet *p, int queueid
  | Output      :
  -----------------------------------------------------------*/

void dsREDQueue:: WFQenqueue(Packet *p, int queueid) {


        hdr_cmn *hdr = hdr_cmn::access(p);
        int size = hdr->size();

        double now =  Scheduler::instance().clock();

        // virtual time update
        // formula 10 in "virtual time implementation"
```

```
        // paragraph
        if(GPS_idle) {
                last_vt_update=now;
                virt_time=0;
                GPS_idle=0;
        } else {
                virt_time=virt_time+(now-last_vt_update)/sum;
                last_vt_update=now;
        }

        // let's compute finish time
        // implements formula 11
        finish_t[queueid] = (finish_t[queueid] > virt_time ?
                finish_t[queueid]:virt_time)
            +size/(double)queueWeight[queueid]/(bandwidth/8);

        // update sum and B
        B[queueid]++;
        WFQupdateSum ();
        if ( fabs(sum) < safe_limit ) sum=0;


        // insertion in both lists
        PGPS_list.insert_order(queueid, finish_t[queueid]);
        GPS_list.insert_order(queueid, finish_t[queueid]);

        // schedule next departure in the GPS
        // reference system

        if(wfq_event!=0) {
                Scheduler::instance().cancel(wfq_event);
                delete wfq_event;
        }
        WFQscheduleGPS();
}



/*-----------------------------------------------------------
  | Function    : WFQscheduleGPS()
  | Description : schedules the next event in the GPS
```

```
  | Changes      :
  | Input        :
  | Output       :
  ------------------------------------------------------------*/

void dsREDQueue::  WFQscheduleGPS() {

        wfq_event=new Event();

        // implements last unnumbered formula in
        // "Virtual Time Implemetation" paragraph
        // "GPS Approach to flow...:single node case"
    // Parekh e Gallager
        double tmp=(GPS_list.get_key_min()-virt_time)*sum;

        // following line is there to recover errors due to
    // finite precision
        if (tmp<0) tmp=0;
        Scheduler::instance().schedule((Handler *)&wfq_hand,
            wfq_event,tmp);
}


/*------------------------------------------------------------
  | Function     : WFQdequeueGPS()
  | Description  : is invoked by scheduler, dequeues GPS
  |                (but doesn't
  |                dequeue PGPS), updates virtual time
  |                variables and schedules the next GPS
  |                dequeuing event
  | Changes      :
  | Input        : Event *e
  | Output       :
  ------------------------------------------------------------*/

void dsREDQueue::  WFQdequeueGPS(Event *e) {

        double now = Scheduler::instance().clock();

        //update virtual time
        virt_time=virt_time+(now-last_vt_update)/sum;
        last_vt_update=now;
```

```
        //extract packet in GPS system
        int queueid=GPS_list.get_data_min();
        GPS_list.extract();

        //update B and sum
        B[queueid]--;
        WFQupdateSum ();
        if ( fabs(sum) < safe_limit ) sum=0;
                if(sum==0) {
                GPS_idle=1;
                for(int i=0;i < MAX_QUEUES;i++)
            finish_t[i]=0;
        }

        // if GPS is not idle, schedule next GPS departure
        delete e;
        if(!GPS_idle)
                WFQscheduleGPS();
        else
                wfq_event=0;
}

/*-----------------------------------------------------------------
  | Function    : WFQdequeuePGPS()
  | Description : is invoked by selectQueueToDequeue and
  |               this by dequeue which is invoked by
  |               simulation. Determines from wich physical
  |               queue a packet should be dequeued and
  |               extracts the control data for that packet
  |               in the PGPS List.
  |
  | Changes     :
  | Input       :
  | Output      :
  -----------------------------------------------------------------*/

void dsREDQueue::WFQdequeuePGPS() {

  qToDq = PGPS_list.get_data_min();

  PGPS_list.extract();
```

```
}

/***************** WFQ addition ***********************/

// RED queues initilization
void dsREDQueue::reset() {

  int i;

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

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

/***************** WFQ addition ***********************/
    finish_t[i] = 0;
    B[i] = 0;
  }

  GPS_idle = 1;
  virt_time = last_vt_update = sum = 0;
/***************** WFQ addition ***********************/

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

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

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

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

     bandwidth = link_->bandwidth ();
    }
/***************** WFQ addition ***********************/

  Queue::reset();

}


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

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

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

```
Post: Adjusts policy's TSW state variables avgRate and
arrivalTime
  (also called tFront) according to the bytes sent.
Note: See the paper "Explicit Allocation of Best effor
Delivery Service" (David
  Clark and Wenjia Fang), Section 3.3, for a description
  of the TSW Tagger.
-------------------------------------------------------*/
void dsREDQueue::applyTSWMeter(int q_id, int pkt_size) {
  double now, bytesInTSW, newBytes;
  double winLen = 1.0;

  bytesInTSW = queueAvgRate[q_id] * winLen;

  // Modified by xuanc(xuanc@isi.edu) Oct 18, 2001,
  // referring to the patch contributed by
  // Sergio Andreozzi <sergio.andreozzi@lut.fi>
  newBytes = bytesInTSW + pkt_size;

  // Calculate the average rate (SW)
  now = Scheduler::instance().clock();
  queueAvgRate[q_id] = newBytes / (now -
          queueArrTime[q_id] + winLen);
  queueArrTime[q_id] = now;
}


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


  //DAN-s
  //double now;
  //adaptive change of scheduling algorithm
```

```
  //access to packet's common header and check which
  //kind of packet is it
  //see packet_t for kind of packet
  //UDP = 2, VIDEO = 4, FTP = 28, PARETO = 29, POISSON = 29
/*  hdr_cmn* accesspacket = hdr_cmn::access(pkt);
  packet_t mypacket = accesspacket->ptype();

  if (mypacket == 2) {
       int dim = accesspacket->size();
       printf("size of udp packet %d\n", dim);
  }
  else if (mypacket == 4) {
       int dim = accesspacket->size();
       printf("size of video packet %d\n", dim);
  }
  else if (mypacket == 29) {
       int dim = accesspacket->size();
       printf("packet type: %d\n", mypacket);
       printf("size of pareto packet: %d\n", dim);
  }
*/


//update the number of packets
//(mypacket == 4)?num_video_packet++:num_other_packet++;
//printf("video=%d other=%d schedMode=%d\n",
//num_video_packet, num_other_packet, schedMode);
//get the current time

//now = Scheduler::instance().clock();
//  if(now > 50){
//     schedMode = schedModeWRR;
//  }

 // printf("now = %f, initial = %f\n", now, initial);

  //check every time window 1
/*  if(now - initial > 0.5){
     //if the number of video packet is about 5% of tot
     //in that window, we change
     if(num_video_packet >=
```

```
            (((num_video_packet+num_other_packet)/100)*5)){
          schedMode = schedModeWRR;
      }
      else{
          schedMode = schedModeRR;
      }
      //put at 0 the counter and update the initial counter
      //for the window
      num_video_packet = 0;
      num_other_packet = 0;
      initial = now;
//      printf("simulation time = %f\n", now);
  }
  */
/*
  if(now - last_seen_vp >= 5){
    printf("sono 5 secondi che non vedo un pacchetto video:
        %f\n",now - last_seen_vp);
    schedMode = schedModeRR;
  }
*/

//  printf("tipopacchetto: %d\n",mypacket);

  //when we have videopacket in the net, we change
  //the scheduling algorithm in WRR
/*
   if(mypacket == 4){
     if(num_video_packet >= 1000){
        //printf("oltre 1000 pkts\n",num_video_packet);
        schedMode= schedModeWRR;
//        printf("num video packet=%d\n",num_video_packet);
     }

     last_seen_vp = now;
  }
*/




//  printf("schedMode = %d\n",schedMode);
```

```
  //DAN-e

  //DAN-s
//   printf("prio = %d\n",codePt);
  //DAN-e

  //extracting the marking done by the edge router
  codePt = iph->prio();
  int ecn = 0;

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

  // code added for ECN support
  //hdr_flags* hf = (hdr_flags*)(pkt->access(off_flags_));
  // Changed for the latest version instead of 2.1b6
  hdr_flags* hf = hdr_flags::access(pkt);

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

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

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

```
/***************** WFQ addition ***********************/
    if (schedMode == schedModeWFQ)
      WFQenqueue (pkt, eq_id);
/***************** WFQ addition ***********************/
    break;
  default:
    break;
  }
}

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

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

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

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

    // update the average rate for pri-queue
    // Modified by xuanc(xuanc@isi.edu) Oct 18, 2001,
    // referring to the patch contributed by
    // Sergio Andreozzi <sergio.andreozzi@lut.fi>
    // When there is a packet dequeued,
    // update the average rate of each queue ()
    if (schedMode==schedModePRI)
      for (int i=0;i<numQueues_;i++)
        if (queueMaxRate[i])
          applyTSWMeter(i, (i == dq_id) ?
              hdr_cmn::access(p)->size() : 0);
```

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

    // decrement virtual queue length
    // Previously in updateREDStateVar,
    // moved by xuanc (12/03/01)
    //redq_[dq_id].qParam_[prec].qlen--;
    redq_[dq_id].updateVREDLen(prec);
    // update state variables for that "virtual" queue
    redq_[dq_id].updateREDStateVar(prec);
  }

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

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

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

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

/***************** WFQ addition ***********************/
  } else if (schedMode == schedModeWFQ) {
```

```
    WFQdequeuePGPS ();
/***************** WFQ addition ***********************/
    // Weighted Round Robin
  } else if (schedMode==schedModeWRR) {
    if(wirrTemp[qToDq]<=0){
      qToDq = ((qToDq + 1) % numQueues_);
      wirrTemp[qToDq] = queueWeight[qToDq] - 1;
    } else {
      wirrTemp[qToDq] = wirrTemp[qToDq] -1;
    }
    while ((i < numQueues_) &&
           (redq_[qToDq].getRealLength() == 0)) {
      wirrTemp[qToDq] = 0;
      qToDq = ((qToDq + 1) % numQueues_);
      wirrTemp[qToDq] = queueWeight[qToDq] - 1;
      i++;
    }
  } else if (schedMode==schedModeWIRR) {
    qToDq = ((qToDq + 1) % numQueues_);
    while ((i<numQueues_) &&
           ((redq_[qToDq].getRealLength()==0) ||
            (wirrqDone[qToDq]))) {
      if (!wirrqDone[qToDq]) {
        queuesDone++;
        wirrqDone[qToDq]=1;
      }
      qToDq = ((qToDq + 1) % numQueues_);
      i++;
    }

    if (wirrTemp[qToDq] == 1) {
      queuesDone +=1;
      wirrqDone[qToDq]=1;
    }
    wirrTemp[qToDq]-=1;
    if(queuesDone >= numQueues_) {
      queuesDone = 0;
      for(i=0;i<numQueues_;i++) {
        wirrTemp[i] = queueWeight[i];
        wirrqDone[i]=0;
      }
    }
```

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

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

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

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

/*---------------------------------------------------------------------
void addPHBEntry(int codePt, int queue, int prec)
    Add a PHB table entry. (Each entry maps a code point
    to a queue-precedence pair.)
---------------------------------------------------------------------*/
void dsREDQueue::addPHBEntry(int codePt, int queue,
        int prec) {
  if (phbEntries == MAX_CP) {
    printf("ERROR: PHB Table size limit exceeded.\n");
  } else {
    phb_[phbEntries].codePt_ = codePt;
    phb_[phbEntries].queue_ = queue;
    phb_[phbEntries].prec_ = prec;
    stats.valid_CP[codePt] = 1;
    phbEntries++;
  }
}

/*---------------------------------------------------------------------
void addPHBEntry(int codePt, int queue, int prec)
    Add a PHB table entry.  (Each entry maps a code point
    to a queue-precedence pair.)
---------------------------------------------------------------------*/
double dsREDQueue::getStat(int argc,
```

```
        const char*const* argv) {

        if (argc == 3) {
                if (strcmp(argv[2], "drops") == 0)
         return (stats.drops*1.0);
                if (strcmp(argv[2], "edrops") == 0)
         return (stats.edrops*1.0);
                if (strcmp(argv[2], "pkts") == 0)
         return (stats.pkts*1.0);
   }
        if (argc == 4) {
                if (strcmp(argv[2], "drops") == 0)
         return (stats.drops_CP[atoi(argv[3])]*1.0);
                if (strcmp(argv[2], "edrops") == 0)
         return (stats.edrops_CP[atoi(argv[3])]*1.0);
                if (strcmp(argv[2], "pkts") == 0)
         return (stats.pkts_CP[atoi(argv[3])]*1.0);
        }
        return -1.0;
}


/*-----------------------------------------------------------
void setNumPrec(int prec)
    Sets the current number of drop precendences.
    The number of precedences is the number of virtual
    queues per physical queue.
-----------------------------------------------------------*/
void dsREDQueue::setNumPrec(int prec) {
        int i;

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

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

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

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

  if (!queue)
    for (i = 0; i < MAX_QUEUES; i++)
      redq_[i].mredMode = tempMode;
  else
    redq_[atoi(queue)].mredMode = tempMode;
}


/*-----------------------------------------------------------------
void printPHBTable()
    Prints the PHB Table, with one entry per line.
-----------------------------------------------------------------*/
void dsREDQueue::printPHBTable() {
   printf("PHB Table:\n");
   for (int i = 0; i < phbEntries; i++)
      printf("Code Point %d is associated with Queue %d,
              Precedence %d\n", phb_[i].codePt_,
              phb_[i].queue_, phb_[i].prec_);
   printf("\n");
```

```
}


/*-----------------------------------------------------------------
void printStats()
    An output method that may be altered to assist
    debugging.
-----------------------------------------------------------------*/
void dsREDQueue::printStats() {
        printf("\nPackets Statistics\n");
        printf("==========================================\n");
        printf(" CP  TotPkts    TxPkts    ldrops
edrops\n");
        printf(" --  -------    ------    ------
-------\n");
        printf("All %8ld %8ld %8ld %8ld\n",
            stats.pkts,stats.pkts-stats.drops-stats.edrops,
            stats.drops,stats.edrops);
        for (int i = 0; i < MAX_CP; i++)
            if (stats.pkts_CP[i] != 0)
                printf("%3d %8ld %8ld %8ld %8ld\n",i,
            stats.pkts_CP[i],
            stats.pkts_CP[i]-stats.drops_CP[i]-stats.edrops_CP[i],
            stats.drops_CP[i],stats.edrops_CP[i]);
}


void dsREDQueue::printWRRcount() {
   int i;
   for (i = 0; i < numQueues_; i++){
      printf("%d: %d %d %d.\n", i, slicecount[i],
              pktcount[i],queueWeight[i]);
   }
}


/*-----------------------------------------------------------------
void setSchedularMode(int schedtype)
   sets up the schedular mode.
-----------------------------------------------------------------*/
void dsREDQueue::setSchedularMode(const char* schedtype) {
        if (strcmp(schedtype, "RR") == 0)
```

```
        schedMode = schedModeRR;
        else if (strcmp(schedtype, "WRR") == 0)
                schedMode = schedModeWRR;
        else if (strcmp(schedtype, "WIRR") == 0)
                schedMode = schedModeWIRR;
        else if (strcmp(schedtype, "PRI") == 0)
                schedMode = schedModePRI;
/***************** WFQ addition ***********************/
   else if (strcmp(schedtype,"WFQ") == 0)
      schedMode = schedModeWFQ;
/***************** WFQ addition ***********************/
        else
                printf("Error: Scheduler type %s does not
                exist\n",schedtype);
}


/*-----------------------------------------------------------
void addQueueWeights(int queueNum, int weight)
   An input method to set the individual Queue Weights.
-----------------------------------------------------------*/
void dsREDQueue::addQueueWeights(int queueNum,
        int weight) {
  if(queueNum < MAX_QUEUES){
    queueWeight[queueNum]=weight;
  } else {
    printf("The queue number is out of range.\n");
  }
}

//Set the individual Queue Max Rates for Priority Queueing
//void dsREDQueue::addQueueRate(int queueNum, int rate) {
  if(queueNum < MAX_QUEUES){
    // Convert to BYTE/SECOND
    queueMaxRate[queueNum]=(double)rate/8.0;
  } else {
    printf("The queue number is out of range.\n");
  }
}

/*-----------------------------------------------------------
int command(int argc, const char*const* argv)
```

```
    Commands from the ns file are interpreted through this
    interface.
-------------------------------------------------------------*/
int dsREDQueue::command(int argc, const char*const* argv) {
  if (strcmp(argv[1], "configQ") == 0) {
    // modification to set the parameter q_w by Thilo
    redq_[atoi(argv[2])].config(atoi(argv[3]), argc, argv);
    return(TCL_OK);
  }
  if (strcmp(argv[1], "addPHBEntry") == 0) {
    addPHBEntry(atoi(argv[2]), atoi(argv[3]),
            atoi(argv[4]));
    return (TCL_OK);
  }
  if (strcmp(argv[1], "meanPktSize") == 0) {
    for (int i = 0; i < MAX_QUEUES; i++)
      redq_[i].setMPS(atoi(argv[2]));
    return(TCL_OK);
  }
  if (strcmp(argv[1], "setNumPrec") == 0) {
    setNumPrec(atoi(argv[2]));
    return(TCL_OK);
  }
  if (strcmp(argv[1], "getAverage") == 0) {
    Tcl& tcl = Tcl::instance();
    tcl.resultf("%f",
            redq_[atoi(argv[2])].getWeightedLength());
    return(TCL_OK);
  }
  if (strcmp(argv[1], "getStat") == 0) {
    Tcl& tcl = Tcl::instance();
    tcl.resultf("%f", getStat(argc,argv));
    return(TCL_OK);
  }
  if (strcmp(argv[1], "getCurrent") == 0) {
    Tcl& tcl = Tcl::instance();
    tcl.resultf("%f",
            redq_[atoi(argv[2])].getRealLength()*1.0);
    return(TCL_OK);
  }
  if (strcmp(argv[1], "printStats") == 0) {
    printStats();
```

```
    return (TCL_OK);
  }
  if (strcmp(argv[1], "printWRRcount") == 0) {
    printWRRcount();
    return (TCL_OK);
  }
  if (strcmp(argv[1], "printPHBTable") == 0) {
    printPHBTable();
    return (TCL_OK);
  }
  if (strcmp(argv[1], "link") == 0) {
    Tcl& tcl = Tcl::instance();
    LinkDelay* del =
        (LinkDelay*) TclObject::lookup(argv[2]);
    if (del == 0) {
      tcl.resultf("RED: no LinkDelay object %s",
                  argv[2]);
      return(TCL_ERROR);
    }
    link_ = del;

/***************** WFQ addition ***********************/
    if (schedMode == schedModeWFQ)
      bind_bw("bandwidth_",&bandwidth);
/***************** WFQ addition ***********************/

    return (TCL_OK);
  }
  if (strcmp(argv[1], "early-drop-target") == 0) {
    Tcl& tcl = Tcl::instance();
    NsObject* p = (NsObject*)TclObject::lookup(argv[2]);
    if (p == 0) {
      tcl.resultf("no object %s", argv[2]);
      return (TCL_ERROR);
    }
    de_drop_ = p;
    return (TCL_OK);
  }
  if (strcmp(argv[1], "setSchedularMode") == 0) {
    setSchedularMode(argv[2]);
    return(TCL_OK);
  }
```

```
  if (strcmp(argv[1], "setMREDMode") == 0) {
    if (argc == 3)
      setMREDMode(argv[2],0);
    else
      setMREDMode(argv[2],argv[3]);
    return(TCL_OK);
  }
  if (strcmp(argv[1], "addQueueWeights") == 0) {
    addQueueWeights(atoi(argv[2]), atoi(argv[3]));
    return(TCL_OK);
  }
  if (strcmp(argv[1], "addQueueRate") == 0) {
    addQueueRate(atoi(argv[2]), atoi(argv[3]));
    return(TCL_OK);
  }
  // Returns the weighted RED queue length for one virtual
  // queue in packets
  // Added by Thilo
  if (strcmp(argv[1], "getAverageV") == 0) {
    Tcl& tcl = Tcl::instance();
   tcl.resultf("%f",
   redq_[atoi(argv[2])].getWeightedLength_v(atoi(argv[3])));
    return(TCL_OK);
  }
  // Returns the length of one virtual queue, in packets
  // Added by Thilo
  if (strcmp(argv[1], "getCurrentV") == 0) {
    Tcl& tcl = Tcl::instance();
    tcl.resultf("%f",
    redq_[atoi(argv[2])].getRealLength_v(atoi(argv[3]))*1.0);
    return(TCL_OK);
  }

  return(Queue::command(argc, argv));
}
```

C Overview on FPGAs

C.1 What is a FPGA

A field programmable gate array (FPGA) is a semiconductor device containing programmable logic components and programmable interconnects. The programmable logic components can be programmed to duplicate the functionality of basic logic gates such as AND, OR, XOR, NOT or more complex combinational functions such as decoders or simple math functions. In most FPGAs, these programmable logic components (or logic blocks, in FPGA parlance) also include memory elements, which may be simple flip-flops or more complete blocks of memories.

A hierarchy of programmable interconnects allows the logic blocks of an FPGA to be interconnected as needed by the system designer, somewhat like a one-chip programmable breadboard. These logic blocks and interconnects can be programmed after the manufacturing process by the customer/designer (hence the term "field programmable") so that the FPGA can perform whatever logical function is needed.

FPGAs are generally slower than their application-specific integrated circuit (ASIC) counterparts, can't handle as complex a design, and draw more power. However, they have several advantages such as a shorter time to market, ability to re-program in the field to fix bugs, and lower non-recurring engineering costs. Vendors can sell cheaper, less flexible versions of their FPGAs which cannot be modified after the design is committed. The development of these designs

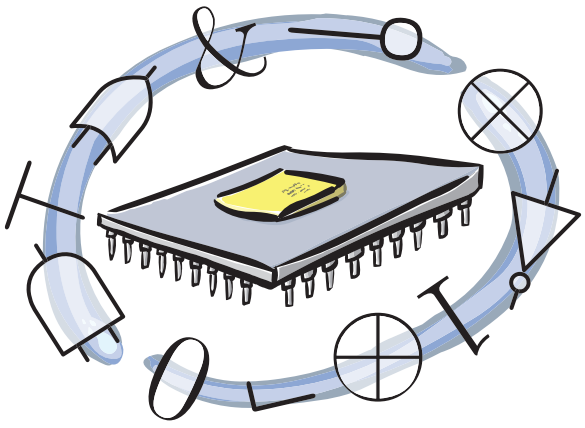


Figure C.1: FPGA [21]

is made on regular FPGAs and then migrated into a fixed version that more resembles an ASIC. Complex programmable logic devices, or CPLDs, are another alternative.[20]

C.1.1 Historical Roots

The historical roots of FPGAs are in complex programmable logic devices (CPLDs) of the early to mid 1980s. CPLDs and FPGAs include a relatively large number of programmable logic elements. CPLD logic gate densities range from the equivalent of several thousand to tens of thousands of logic gates, while FPGAs typically range from tens of thousands to several million.

The primary differences between CPLDs and FPGAs are architectural. A CPLD has a somewhat restrictive structure consisting of one or more programmable sum-of-products logic arrays feeding a relatively small number of clocked registers. The result of this is less flexibility, with the advantage of more predictable timing delays and a higher logic-to-interconnect ratio. The FPGA architectures, on the other hand, are dominated by interconnect. This makes them far more flexible (in terms of the range of designs that are practical for implementation within them) but also far more complex to design for.

Another notable difference between CPLDs and FPGAs is the presence in most FPGAs of higher-level embedded functions (such as adders and multipliers) and embedded memories. A related, important difference is that many modern FPGAs support full or partial in-system reconfiguration, allowing their designs to be changed "on the fly" either for system upgrades or for dynamic reconfiguration as a normal part of system operation. Some FPGAs have the capability of partial re-configuration that lets one portion of the device be re-programmed while other portions continue running.

A recent trend has been to take the coarse-grained architectural approach a step further by combining the logic blocks and interconnects of traditional FPGAs with embedded microprocessors and related peripherals to form a complete "system on a programmable chip". Examples of such hybrid technologies can be found in the Xilinx Virtex-II PRO and Virtex-4 devices, which include one or more PowerPC processors embedded within the FPGA's logic fabric. The Atmel FPSLIC is another such device, which uses an AVR processor in combination with Atmel's programmable logic architecture. An alternate approach is to make use of "soft" processor cores that are implemented within the FPGA logic. These cores include the Xilinx MicroBlaze and PicoBlaze, and the Altera Nios and Nios II processors, as well as third-party (either commercial or free) processor cores.

As previously mentioned, many modern FPGAs have the ability to be reprogrammed at "run time," and this is leading to the idea of reconfigurable computing or reconfigurable systems CPUs that reconfigure themselves to suit the task at hand. Current FPGA tools, however, do not fully support this methodology.

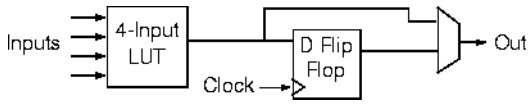


Figure C.2: Typical FPGA logic block

It should be noted here that new, non-FPGA architectures are beginning to emerge. Software-configurable microprocessors such as the Stretch S5000 adopt a hybrid approach by providing an array of processor cores and FPGA-like programmable cores on the same chip. Other devices (such as Mathstar's Field Programmable Object Array, or FPOA) provide arrays of higher-level programmable objects that lie somewhere between an FPGA's logic block and a more complex processor.[20]

C.1.2 Architecture

The typical basic architecture consists of an array of configurable logic blocks (CLBs) and routing channels. Multiple I/O pads may fit into the height of one row or the width of one column. Generally, all the routing channels have the same width (number of wires). An application circuit must be mapped into an FPGA with adequate resources.

The typical FPGA logic block consists of a 4-input lookup table (LUT), and a flip-flop, as shown in C.2. There is only one output, which can be either the registered or the unregistered LUT output. The logic block has four inputs for the LUT and a clock input. Since clock signals (and often other high-fanout signals) are normally routed via special-purpose dedicated routing networks in commercial FPGAs, they are accounted for separately from other signals.

For this example architecture, the locations of the FPGA logic block pins are shown in C.3.

Each input is accessible from one side of the logic block, while the output pin can connect to routing wires in both the channel to the right and the channel below the logic block. Each logic block output pin can connect to any of the wiring segments in the channels adjacent to it. Similarly, an I/O pad can connect to any one of the wiring segments in the channel adjacent to it. For example, an I/O pad

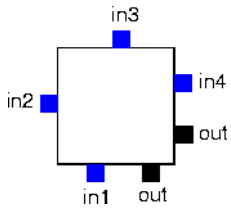


Figure C.3: Locations of the FPGA logic block pins

at the top of the chip can connect to any of the W wires (where W is the channel width) in the horizontal channel immediately below it. Generally, the FPGA routing is unsegmented. That is, each wiring segment spans only one logic block before it terminates in a switch box. By turning on some of the programmable switches within a switch box, longer paths can be constructed. For higher speed interconnect, some FPGA architectures use longer routing lines that span multiple logic blocks. Whenever a vertical and a horizontal channel intersect there is a switch box. In this architecture, when a wire enters a switch box, there are three programmable switches that allow it to connect to three other wires in adjacent channel segments. The pattern, or topology, of switches used in this architecture is the planar or domain-based switch box topology. In this switch box topology, a wire in track number one connects only to wires in track number one in adjacent channel segments, wires in track number 2 connect only to other wires in track number 2 and so on. The figure C.4 illustrates the connections in a switch box. Modern FPGA families expand upon the above capabilities to include higher level functionality fixed into the silicon. Having these common functions embedded into the silicon reduces the area required and gives those functions increased speed compared to building them from primitives. Examples of these include multipliers, generic digital signal processor (DSP) blocks, embedded processors, high speed IO logic and embedded memories. FPGAs are also widely used for systems validation including pre-silicon validation, post-silicon validation, and firmware development. This allows chip companies to validate their design before the chip is produced in the factory, reducing the time to market.[20]

C.1.3 FPGA design and programming

To define the behavior of the FPGA the user provides a hardware description language (HDL) or a schematic design. Common HDLs are VHDL and Verilog. Then, using an electronic design automation tool, a technology-mapped netlist is

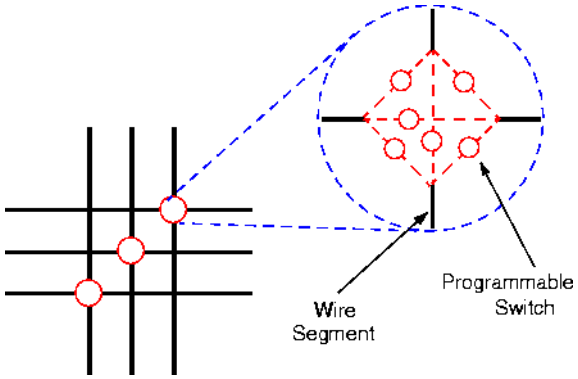


Figure C.4: Switch Box Topology

generated. The netlist can then be fitted to the actual FPGA architecture using a process called place-and-route, usually performed by the FPGA company's proprietary place-and-route software. The user will validate the map, place and route results via timing analysis, simulation, and other verification methodologies. Once the design and validation process is complete, the binary file generated (also using the FPGA company's proprietary software) is used to (re)configure the FPGA device.

In an attempt to reduce the complexity of designing in HDLs, which have been compared to the equivalent of assembly languages, there are moves to raise the abstraction level of the design. Companies such as Cadence, Synopsys and Celoxica are promoting SystemC as a way to combine high level languages with concurrency models to allow faster design cycles for FPGAs than is possible using traditional HDLs. Approaches based on standard C or C++ (with libraries or other extensions allowing parallel programming) are found in the Catapult C tools from Mentor Graphics, and in the Impulse C tools from Impulse Accelerated Technologies. Annapolis Micro Systems, Inc.'s CoreFire Design Suite provides a graphical dataflow approach to high-level design entry. Languages such as SystemVerilog, SystemVHDL, and Handel-C (from Celoxica) seek to accomplish the same goal, but are aimed at making existing hardware engineers more productive versus making FPGAs more accessible to existing software engineers.

To simplify the design of complex systems in FPGAs, there exist libraries of predefined complex functions and circuits that have been tested and optimized to speed up the design process. These predefined circuits are commonly called IP cores, and are available from FPGA vendors and third-party IP suppliers (rarely free, and typically released under proprietary licenses). Other predefined circuits are available from developer communities such as OpenCores.org (typically "free", and released under the GPL, BSD or similar license), and other sources.

In a typical design flow, an FPGA application developer will simulate the design at multiple stages throughout the design process. Initially the RTL description in VHDL or Verilog is simulated by creating test benches to stimulate the system and observe results. Then, after the synthesis engine has mapped the design to a netlist, the netlist is translated to a gate level description where simulation is repeated to confirm the synthesis proceeded without errors. Finally the design is laid out in the FPGA at which point propagation delays can be added and the simulation run again with these values back-annotated onto the netlist.[20]

C.1.4 FPGA with Central Processing Unit Core

Some engineering applications have used a single FPGA device to replace the function of a simple embedded-microcontroller. More recently, a complete 32-bit CPU (Central Processing Unit) core can be implemented through the programmable logic of a high-capacity FPGA. Such CPU cores are called soft CPU core. Be-

yond this, some FPGA devices contain dedicated hardware CPU core(s). Selected Virtex parts from Xilinx contain 1 or more IBM PowerPC 405 CPU embedded cores, in addition to the FPGA's own programmable logic. For a given CPU architecture, a hard (embedded) CPU core will outperform a soft-core CPU (i.e., a programmable-logic implementation of the CPU.) The embedded CPU contains exactly the logic and only the logic structures needed for the CPU's function, and the embedded CPU's logic is task-specific optimized, whereas a softcore CPU must live within the FPGA's general-purpose logic fabric. Embedded CPUs can be also easier to integrate into a FPGA-based application because the fixed-nature of the embedded CPU possesses predictable timing characteristics, and the complexity of an equivalent programmable-logic CPU consumes much more of the FPGA's scarce programmable-logic resources, complicating the placement & routing of the design's remaining non-CPU components.[20]

Figure C.5: Xilinx FPGA