MASTER IN COMPUTER SCIENCE

Support of fairness and guarantees without per-flow state in routers

Pelsser, Cristel

Award date:
2001

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

Support of fairness and guarantees without per-flow state in routers

Cristel PELSSER

Promoteur : Professeur Olivier Bonaventure

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

Année Académique 2000-2001

Abstract

In this thesis, mechanisms that support minimum throughput guarantees and provide fair allocation of the remaining bandwidth without maintaining per flow state in core routers will be investigated.

Mechanisms have been proposed to support flows with a minimum guaranteed rate and to distribute the remaining bandwidth fairly. These mechanisms usually require that each intermediate network node maintains some state for each flow passing through it and performs per-flow operations. This causes a scalability problem. Other mechanisms, that avoid the use of per-flow information and avoid per-flow processing in the interior of a network domain are now being proposed. Such mechanisms will be exposed and analysed.

Résumé

Différents mécanismes permettant le support de garanties de débit minimum ainsi qu'une attribution équitable de la bande passante restante, sans nécessiter le maintien d'informations concernant chaque flux par les routeurs situés au coeur du réseau, seront étudiés dans ce mémoire.

Plusieurs mécanismes ont été proposés pour permettre le support de garanties de débit minimum et l'attribution équitable de bande passante. Ces mécanismes requièrent que chaque noeud intermédiaire du réseau maintienne un état pour chaque flux le traversant et effectue des opérations pour chacun de ces flux. Ceci entraîne un problème lors de l'utilisation de tels mécanismes sur des réseaux très étendus. D'autres mécanismes, qui permettent d'éviter le maintien d'un état pour chaque flux ainsi que l'exécution d'opérations pour chaque flux, sont proposés actuellement. Certains de ces mécanismes seront exposés et analysés.

Overzicht

In deze thesis worden verschillende mechanismen onderzocht die een minimum bandbreedte kunnen garanderen en de resterende bandbreedte eerlijk verdelen zonder dat een router een toestand moet bijhouden voor elke flow.

Verschillende mechanismen werden reeds voorgesteld die een minimum bandbreedte garanderen met een eerlijke verdeling van het overblijvend gedeelte. Meestal vereisen deze technieken dat elke tussenliggende router een toestand bijhoudt voor elke flow dat door deze router passeert. Dit resulteert in een schaalbaarheidsprobleem wanneer het aantal flows groeit. Ondertussen zijn er ook meer recentere mechanismen die geen toestand en verwerking van elke flow apart vereisen in de kern van het netwerk. Deze thesis zal zulke mechanismen uitgebreid beschrijven en analyseren.

Acknowledgements

First, I would like to thank Professor Olivier Bonaventure for suggesting such a captivating research topic as well as for the preparation and the knowledge he gives through his classes.

I would like to express my profound appreciation to my mentor Stefaan De Cnodder at the Alcatel Network Strategy Group in Antwerp, where I prepared most of this thesis. His advises and his questions stimulated my reflection all along this project.

I would also like to thank Guido Petit for welcoming me in his team, the Alcatel Network Strategy Group, and for providing all the resources necessary to the research, the implementation and the simulations done in the framework of my thesis.

The comments concerning the content and the redaction of the thesis provided by Stefaan De Cnodder, Steve Uhlig and Olivier Bonaventure were helpful and sometimes challenging. I thank them for sharing part of their competence.

I would like to thank Cedric Rosman and Louis Swinnen for their company during the few month spent together in Antwerp. Finally, I would like to express my sincere appreciation to my parents for guiding me all along my studies.

Contents

List of Figures

Acronyms

AF	Assured Forwarding
bps	bits per seconds
CJVC	Core-Jitter Virtual Clock
CSFQ	Core Stateless Fair Queueing
DiffServ	Differentiated Services
DPS	Dynamic Packet State
DSCP	Differentiated Services Code Point
EF	Expedited Forwarding
FADE	Fair Allocation Derivative Estimation
FIFO	First In First Out
GFC	Generic Fairness Configuration
guar.	guarantee
IntServ	Integrated Services
IP	Internet Protocol
ISP	Internet Service Provider
JVC	Jitter Virtual Clock
Kb	Kilo bits

Kbps	Kilo bits per seconds
Mbps	Mega bits per seconds
MC-RED	Multi-Color Marking Scheme
MCSFQ	Modified Core Stateless Fair Queueing
ms	milliseconds
MTU	Maximum Transmission Unit
OPNET	Optimum Network Performance
PH	Per-hop admission control
PHB	Per Hop Behavior
QoS	Quality of Service
RED	Random Early Detection
RSVP	Resource reSerVation Protocol
RTT	Round trip time
SLA	Service Level Agreement
SM	Simple Marking
SPFQ	Stateless Prioritized Fair Queueing
TCP	Transmission Control Protocol
ToS	Type of Service
Tspec	Traffic Specification
UBR	Unit-based reservations
UDP	User Datagram Protocol

VPI Virtual Path Index

WCSFQ Weighted Core Stateless Fair Queueing

WFQ Weighted Fair Queueing

Chapter 1

Quality of Service

Quality of Service (QoS) is a popular concept among the Internet community. But, this concept has various definitions. "To some people it means introducing an element of predictability and consistency into the best-effort network delivery systems. To others, it means obtaining higher transport efficiency from the network and attempting to increase the volume of data delivery while maintaining characteristically consistent behavior. And yet, to others, QoS is simply a means of differentiating classes of data service, offering network resources to higher-precedence service classes at the expense of lower-priority classes. QoS also may mean attempting to match the allocation of network resources to the characteristics of specific data flows. All these definitions fall within the generic area of QoS as we understand it today."[FH98]

The popularity of the QoS concept lies in the possibility of providing more distinguished services than simply getting the packets to their destination. QoS is seen as a way to provide an additional source of revenue and a competitive position to Internet Service Providers. The existence of mechanisms providing differentiated class of services would allow to develop different types of contracts with the Internet users. In these contracts, the service provided would be found as well as the corresponding billing system. Such contracts are called Service Level Agreements. In the corporate, academic and research network areas, there are also compelling reasons to the provision of QoS. Although the competitive and economic reasons may not explicitly apply in these types of networks, providing different levels of services may still be desirable. These levels of services could be important to differentiate the service provided to critical applications to the ones from which benefit accessory applications. A differentiation of service among the users may also be considered.

The attempt to match the allocation of network resources to the requirements of some data flows may be in contradiction to the objective of offering an efficient delivery service supporting many flows each with its proper characteristics. The incapacity to provide such precise services adapted to the flows' needs, in a dynamic environment where the flows are numerous and short-living, is called the scalability problem. It results from the need to maintain a state for each flow, to which a particular service is required, in each router on the flow's path. Such particular service is called a guarantee. And, the role of the flow's state is to store the guarantee of the flow and the current situation concerning this flow. Such information is used by the router to forward the packets according to the service guaranteed to each flow.

By comparison, the provision of Differentiated Services (DiffServ) does not require to maintain state for each flow in the routers. At the border, the entrance of the network, it is determined to which class of service the packets belong. Based on this classification, the packets are marked with a label carrying information on their service class. This label enables routers to determine the forwarding method of packets. Buffer acceptance mechanisms and schedulers are used to implement the different forwarding methods that are supported by the routers. For such differentiated services, no state is required per-flow in the core of the network. Therefore, the provision of such services is more scalable than the provision of guarantees that are adapted to the precise needs of the flows.

The provision of guarantees appropriated to the characteristics of the flows is very interesting for certain flows. The interest lies in the independence of such services to the load of the network and to the properties of the other flows sharing the network. In an Integrated Services (IntServ) architecture the flows are provided with absolute guarantees. To bypass the scalability problem of the IntServ architecture, methods to provide absolute guarantees in a DiffServ architecture are actually the object of research.

The purpose of this thesis is to study scalable methods allowing the provision of absolute guarantees as well as protection of the flows against the other flows. To achieve this goal, a first look is given at some of the guarantees that may be thought of as interesting to provide to the flows. The concept of fairness shows the protection that may be assured to the flows from flows hungry for bandwidth. From these guarantees, Integrated and Differentiated Services may be built. These services providing either absolute or relative guarantees.

1.1 Different kind of guarantees

A flow is a sequence of packets with one common characteristic[Bon00]. This characteristic can be based on any field of the packets. The packets of a layer n flow have a common characteristic in the layer n header. For example, application flows are layer four flows in the TCP/IP suite. The packets of an application flow are used by a specific application. The application is the identifier of the flow. Layer three and two flows may also be considered.

There can be guarantees on the amount of bandwidth a flow is allowed to use, on the maximum delay incurred by the packets of a flow, on the maximum variation of the delay, on the packet loss ratio of a flow, ...

1.1.1 Best effort

Traditionally, the Internet offers only a single Quality of Service (QoS), best effort. With this service, no guarantees in terms of bandwidth, delay and packet loss can be ensured to a flow.

The best effort service is not sufficient for streaming or critical applications. The flows associated to these applications may not be able to use the amount of bandwidth required by the applications, the delay of the packets may be too high or, the lost packets may hinder the good functioning of the applications.

Without providing bandwidth, delay and packet loss guarantees, a best effort network should provide a fair service to all its users. The goal of a best effort network should be to

ensure that there are no flows using all the resources while some other flows are starving. In a best effort network, all packets should receive the same service. There are two manners to define the fairness support, namely fairness on a link and fairness in the network.

Fairness on a link

One way to protect flows from one another is to share the bandwidth between them so that all flows are given the same portion of the bandwidth on each link. The fair share on a link is given by

$$Fair\ share_{on\ a\ link} = \frac{Link\ bandwidth}{Number\ of\ flows} \quad .$$

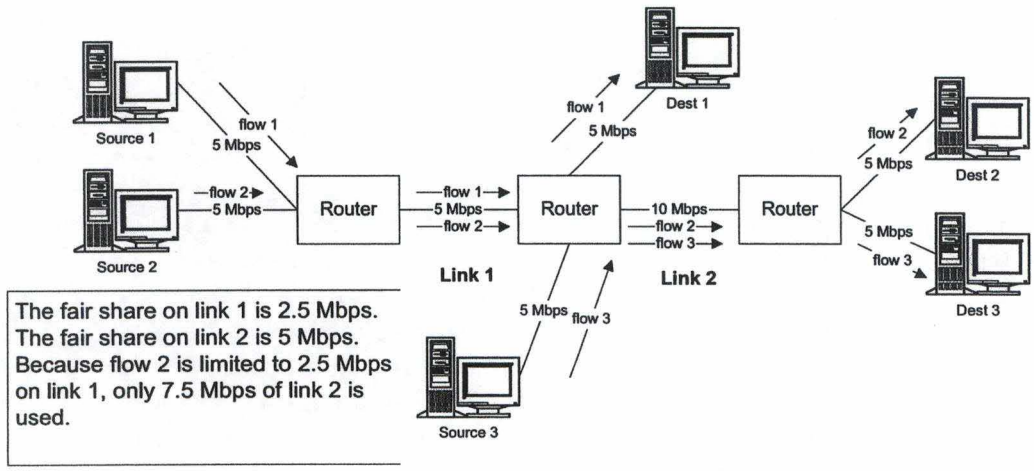

Figure 1.1: Fairness on a link

This solution has a major disadvantage : if the rate of some flows is below the fair share of the link, some bandwidth isn't used. There is a waste of resources. The rate of a flow on a link is below the fair share of the link when the sending rate of the flow is below the fair share of the link or when the flow is bottlenecked before the link. The bandwidth that is not used by such flows cannot be shared between the other flows with this definition of fairness. An example where some resources are unused but the network is congested is shown in figure 1.1. In this situation, it is supposed that each source has a sending rate of 10 Mbps. Flow 1 and flow 2 are congested on the first link. The fair share on link 1 is 2.5 Mbps. Flow 3 is bottlenecked on the second link. The fair share of the second link is 5 Mbps. The rate of flow 2 on link 2 is of 2.5 Mbps because of the fair share on the first link. Flow 3 will be limited to 5 Mbps even though flow 2 doesn't use all its fair share. Only 7.5 Mbps are used on link 2.

Fairness in the network

The *max-min fairness* concept resolves the problems of the definition of fairness on a link. Max-min fairness is applicable to a network while the concept of fairness on a link may introduce a waste of resources when applied to a network.

In a max-min allocation, bandwidth is distributed in a fair manner on a link when all bottlenecked[1] flows have the same share of the bandwidth. This allocation maximizes the attribution of bandwidth to the sources receiving the smallest allocation [BG92].

The allocation of bandwidth to each flow can be determined as follows :

1. Set the bandwidth allocation of each flow to 0 Mbps.
2. Increase all these allocations by 1 Mbps[2].
3. While there aren't any flows bottlenecked in the network, increase the allocation of each flow by 1 Mbps.
4. For the flows that are bottlenecked on a link, set the final allocation of bandwidth to the actual allocation.
5. Do not consider these bottlenecked flows anymore.
6. If the number of flows considered is greater than zero go to number 2.

An example of fair allocation (FA) is shown by figure 1.2. To obtain the fair share of each flow, the algorithm exposed in the previous paragraph is applied. In the figure, FA1 stands for "fair allocation of flow 1 (Fl 1)". At first, the fair allocations of all flows are set to 0 Mbps. Then, they are increased until flow 2 and flow 3 become bottlenecked on link 1. These two flows are taken out of the set of flows considered by the algorithm. The set now contains only flow 1. Its fair allocation is increased until link 2 becomes bottlenecked. Then, the set of flows considered is empty. The algorithm stops. The fair share of each flow is the value of $FA_{flownumber}$ at the time the flow is taken out of the set.

A property of a max-min fair allocation is such that to increase the bandwidth allocated to one source, it is necessary to decrease the bandwidth allocated to another source which already receives a lower allocation [BG92]. For example, as shown by figure 1.2, to increase the bandwidth attributed to source 1 by 1 Mbps, the only possibility is to decrease the amount of bandwidth allocated to source 2 or source 3 or both of them but these sources already have a lower allocation than source 1. On link 2, only the amount of bandwidth attributed to flow 1 can be increased. The other flows are already limited on the first link. On one link, only the flows with the highest bandwidth allocation could benefit of an even higher allocation. The flows that have a lower allocation are either limited by the sending rate of the source or are limited on a previous link in the network. The bandwidth attributed to source 2 can only be increased by decreasing the amount of bandwidth allocated to source 3. Decreasing the bandwidth source 1 gets would have no impact on the bandwidth that can be allocated to source 2 if the bandwidth allocated to source 3 has to be kept constant.

When we talk about fair allocation, we usually consider max-min fairness. Such a fair allocation is hard to determine in practice because flows are established and torn down at any time. Flows might also not be bottlenecked at all in the network leaving more bandwidth for the other flows sharing the network resources.

[1] A flow is said to be bottlenecked on a link when it cannot send at a higher rate or its packets will be dropped at the node transmitting on that link ; some of its packets may already be dropped at that node.

[2] When 1 Mbps is the smallest bandwidth unit considered.

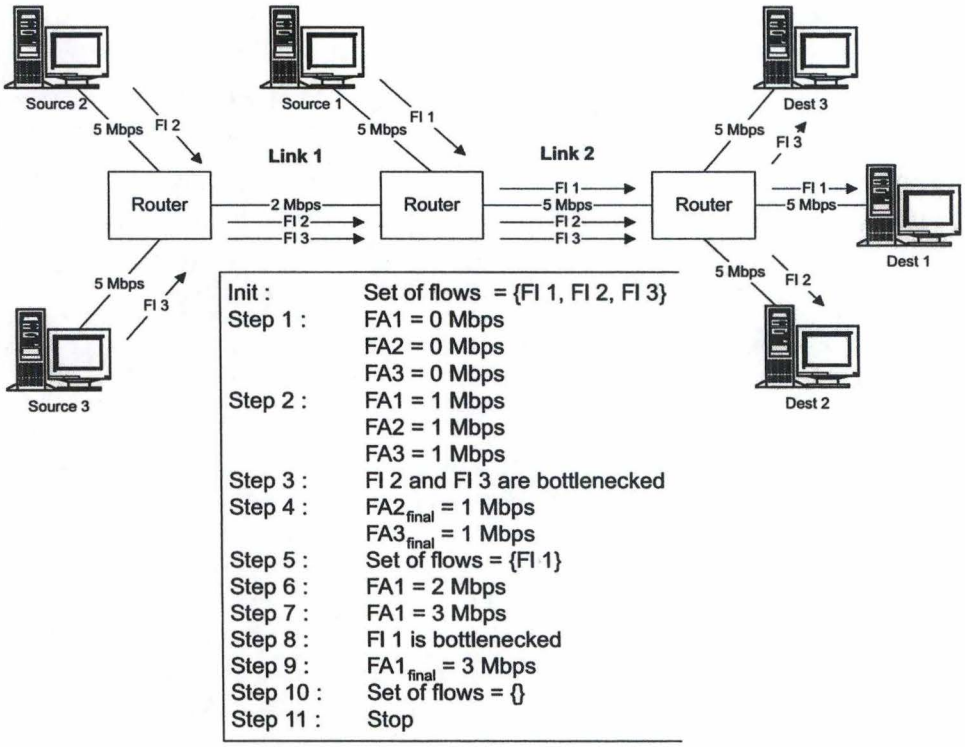

Figure 1.2: Max-min fair allocation

When a weight is associated to the flows, in a fair allocation, all flows bottlenecked on the same link have the same value for the ratio of the outgoing bandwidth used by the flow over the flow's weight [SSZ98a]. Let's consider for example two flows. The first flow (flow 1) has a weight of 2. The second flow (flow 2) has a weight of 1. If these two flows are bottlenecked on the same link, the fair share of flow 1 is twice the fair share of flow 2.

1.1.2 Minimum guarantees

A flow that is the object of a minimum bandwidth guarantee is allowed to use this bandwidth at any time during the reservation. And, if the network is not congested, such flow may use more than its reservation.

This type of guarantees is useful for critical applications as well as for streaming applications that adapt their sending rate to the available bandwidth. These applications are sure to have the resources they need for their functioning if the reservation fits to their critical need. And, when the load of the network is low they may benefit from more bandwidth. This additional bandwidth can be used to provide a better quality to the user of a video conference application, for example.

1.1.3 Maximum guarantees

A maximum throughput guarantee is a guarantee where the flow is allowed to use a certain amount of bandwidth at any time but it is never allowed to use more than its guarantee.

It is useful to provide maximum throughput guarantees to non-adaptive streaming applications. These applications do not need more bandwidth than their maximum sending rate. And, their sending rate is almost constant. The guarantee is then equal to the average sending rate of the flow which is almost equal to the maximum sending rate.

Bandwidth as well as delay, delay jitter and packet loss can be the object of guarantees. The mechanisms that are exposed in chapter 2 either try to provide a fair best effort service to all flows or also support minimum bandwidth guarantees.

Different services aim to provide guarantees to flows. Among these services there are the Integrated Services (IntServ) and the Differentiated Services (DiffServ).

1.2 Integrated Services

IntServ provides Quality of Service (QoS) to layer four flows. These flows inform the network about their QoS requirements. And, the network accepts or rejects the flows based on their requirements and the current state of the network.

An IntServ router has to handle Resource reSerVation Protocol (RSVP) messages and to perform admission control at the control level. RSVP is a protocol used to associate reservations to unidirectional layer four flows. Admission control ensures that if a flow is accepted in the network, there are enough resources to provide its guaranteed requirements.

At the forwarding level, the packets have to be classified into layer four flows. Then, the packets are queued and scheduled according to their classification. The respect of guarantees by the network depends on the shaping, policing, queueing and scheduling algorithms.

There are two types of Integrated Services. The guaranteed service and the controlled load service. These services are introduced in the following subsections.

1.2.1 Guaranteed service

The guaranteed QoS[SPG97] applies to the portion of a flow that conforms to the token bucket Traffic Specification, Tspec, carried in the RSVP messages associated to the flow. The Rspec parameter also carried by RSVP messages defines the actual service rate delivered to the flow by the network as well as some additional terms that bound the actual delay delivered to the flow[McD00]. The guaranteed service should strive to treat packets that fail to conform to the Tspec on a best-effort basis. The guaranteed QoS associated to a flow has to be provided to the portion of the flow that conforms to its Tspec even if the flow in its entirety does not conform to its Tspec. And, flows conforming to their Tspec should receive their guaranteed QoS in spite of the presence of non conforming flows.

In the guaranteed service, the flows benefit from a minimum throughput guarantee and the maximum delay of the guaranteed packets is bounded. There are no guarantees on the minimum or average delay. And, guaranteed packets should not suffer losses.

1.2.2 Controlled-load service

Like the guaranteed service, the controlled-load service applies to the portion of a flow that conforms to the token bucket Traffic Specification, Tspec. The idea of the controlled-load service is to provide the same performance as the best-effort service under unloaded conditions. A very high percentage of transmitted packets will be successfully delivered by the network[Wro97]. And, the transit delay experienced by a very high percentage of the delivered packets will not greatly exceed the minimum transmit delay[Wro97]. Controlled-load service requires that each network element provides the QoS to conforming flows even if some flows do not conform to their Tspec[McD00]. The specification requires that the network attempts to forward nonconforming packets on a best-effort basis[McD00]. The network elements may either degrade the performance of all packets in a nonconforming flow, or only those packets that exceed the Tspec parameters[Wro97].

In the controlled-load service, there are no deterministic guarantees on losses but the amount of lost packets should be almost as low as in the guaranteed service for conforming flows. There are also no guarantees on maximum delay and delay jitter. However, the average queueing delay should be low for conforming packets.

1.3 Differentiated Services

The idea is that DiffServ provides a few classes of traffic. Different QoS commitments, in terms of average delay and average losses, are associated to each class of traffic. The Internet Service Provider (ISP) has to configure its routers to efficiently support the requirements of the traffic classes. The ISP also has to take care of the network provisioning by taking into account the evolution of the load on the links and of the customers' traffic contracts.

To provide these Differentiated Services, a distinction is made between two types of network elements : the interior nodes and the boundary nodes. Interior nodes are simple to operate at high speeds and the more complex functionalities are provided only by the boundary nodes. Among these functionalities, there is the marking of the packets according to the Service Level Agreement (SLA) negotiated with the customer. The type of service is indicated explicitly inside each IP packet in the Differentiated Services Code Point (DSCP). Interior routers rely on the DSCP to provide the different types of services. The ISP associates a Per Hop Behavior (PHB) to each DSCP, inside each router. The PHBs are the building blocks from which an end-to-end service can be constructed[McD00]. The complexity at the interior routers only rely on the number of different services, not on the number of flows.

Among the Differentiated Services there are the best-effort PHB, the Assured Forwarding PHB group and the Expedited Forwarding PHB.

1.3.1 Assured Forwarding

In the Assured Forwarding (AF) PHB group, the traffic is divided into at least four classes. And, inside each class there are at least three drop preferences and at least two drop probabilities. Inside a class, the packets with the highest drop preference will be dropped with a higher probability than the packets with a lower drop preference. Inside each router, some buffer and bandwidth resources are reserved for each class. An additional requirement is that

packets of the same AF class may not be reordered. The Assured Forwarding PHB group is described in [HBWW99].

1.3.2 Expedited Forwarding

A description of the Expedited Forwarding (EF) PHB is given in [JNP99]. The goal of EF is to provide a service with a low packet loss ratio, a low end-to-end delay and a low delay jitter. The ISP configures the amount of bandwidth available to the EF traffic at each node. And, the EF traffic should receive this bandwidth independently of the intensity of any other traffic at the node over any time interval equal to or longer than the time to send a Maximum Transmission Unit (MTU) sized packet at the configured rate.

1.4 Integrated versus Differentiated Services

The major differences between the IntServ architecture and the DiffServ architecture are given in table 1.1 inspired from [DR99]. The major problem of the Integrated Services is the scalability. There is some overhead due to RSVP signalling messages, some state information for each flow is required inside each router and there is heavy per data packet processing (classification, checking of the traffic contract and the reservation).

Additionally, in the DiffServ architecture, SLA only have to be negotiated between pairs of ISPs that are connected. In the IntServ architecture, agreements have to be taken between all the ISPs crossed by the guaranteed flows.

1.5 Scope of the thesis

In this thesis, it is studied how to provide individual flow service differentiation in a DiffServ architecture. The goal is to provide fine granularity service differentiation by avoiding the scalability problems of the IntServ architecture. Also, stateless alternatives to RSVP are given for the admission control. For these protocols, there is no need to maintain per-flow state in the interior routers, only aggregate information is stored. The aim is to obtain a scalable solution where absolute guarantees can be provided to individual flows. To achieve this objective, a global view of the working of core stateless mechanisms is given. A particular mechanism is studied, improved and simulated.

In chapter 2, the basic functioning of scalable methods for the provision of fairness and guarantees is exposed and is illustrated by the presentation of different mechanisms. The features of the services supported by these mechanisms are studied and extensions are proposed. Then, some scalable admission control mechanisms, needed for the provision of absolute guarantees, are discussed.

Among the scalable mechanisms, also called core stateless mechanisms, presented in chapter 2, one mechanism is studied more deeply in the third chapter. This mechanism is called Core Stateless Fair Queueing (CSFQ). Some clarifications about the algorithm are provided and the impacts of the configuration parameters are investigated.

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

Table 1.1: Comparison between Integrated Services and Differentiated Services

Chapter 4 concerns possible improvements of CSFQ and the creation of a new mechanism based these improvements. In this chapter a contribution to CSFQ is brought for the support of minimum throughput guarantees in addition of the initial fair allocation of bandwidth.

The next chapters introduce the multiple scenarios used for the simulations and provide an argumentation of the results obtained from these simulations. The simulation scenarios presented in chapter 5 possess different characteristics concerning the amount of bottlenecked links, the fair share on these links, the Round Trip Times (RTTs) of the flows, ...and are used with various types of traffic.

In chapter 6, the analysis of the simulation results underlines the behavior of CSFQ and its modified version, called Modified Core Stateless Fair Queueing (MCSFQ). Then, the degree of fairness in the bandwidth distribution obtained with CSFQ and MCSFQ in different situations, resulting from the various scenarios, is the center of interest with the capacity to ensure the bandwidth guarantee of the flows. Finally, further improvements to CSFQ are proposed in the further work section of the conclusion.

Chapter 2

Stateless Mechanisms

Stateless mechanisms usually rely on Dynamic Packet State (DPS) to store informations from the state of the flows. They may also rely on feedback mechanisms to communicate informations from the core to the edge nodes. In this chapter, some of the stateless mechanisms presented in the literature are exposed. Then, a comparison of these mechanisms is given in terms of complexity location, guarantees and admission control requirements. Furthermore, we propose adaptations to these mechanisms in order to enlarge their provision of guarantees. In the second part, various stateless admission control mechanisms are exposed. And, finally, an evaluation of these admission control mechanisms is given.

2.1 Dynamic Packet State

Today, to provide guaranteed services, the routers have to implement complicated classification, shaping, policing, buffer management and scheduling mechanisms based on per-flow information. In the Internet nowadays there are many short living flows. They usually don't require a great amount of bandwidth but there are a lot of them. If each router had to maintain some information about each flow passing through it there would be a limitation on the number of these flows.

An other limitation could be the computation speed of the routers. Before, packets were usually buffered waiting for link availability. Now, with the advances made in link technologies, the process could reverse. Routers could become too slow to provide packets quickly enough on their output links.

By avoiding to maintain per-flow state in routers and to provide per-flow treatment, we could gain space in the routers' memory and speed in packet processing and forwarding. The number of flows passing through a router wouldn't be limited anymore. However, to provide guarantees to a flow, the routers have to know how to treat its packets. They therefore need to know the guarantees associated to the flow as well as other information that state the flow's actual situation. For routers to be aware of flows' situations some flow state could be carried by each packet of the guaranteed flows. This state carried in packet headers is called "Dynamic Packet State". The routers don't need to keep per-flow information anymore. They will process packets according to the states included in the packets' headers.

As illustrated by figure 2.1, a distinction is being made between two kind of routers, the ones that are at the border of the network, called edge routers, and the ones that aren't,

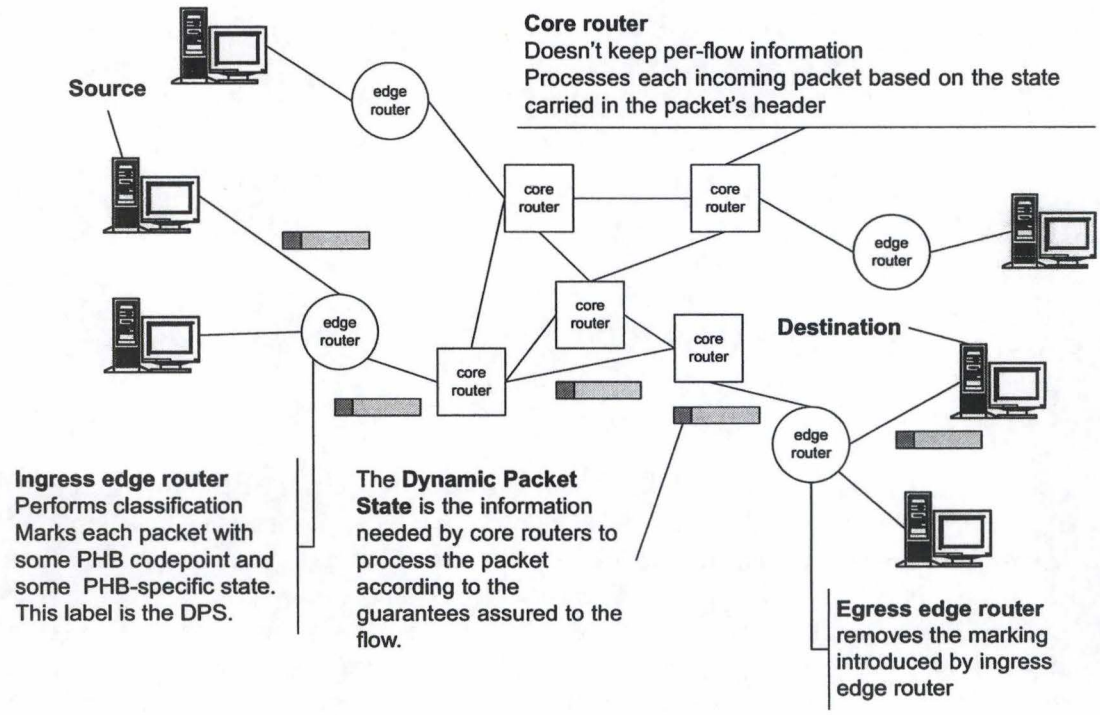

Figure 2.1: Functioning of Dynamic Packet State

which are called core routers. When Dynamic Packet State (DPS) is used, edge routers perform classification of the flows and mark the packets with the per-flow state required by routers to support guarantees. Edge and core routers then process the packets according to the marking. The label determines a Per Hop Behavior (PHB[1]); the marking is also sometimes called Differentiated Services Code Point (DSCP[2]). To respect transparency, when a packet reaches the edge of the network, the state introduced in the packet at the entrance of the network is removed. An introduction to DPS can be found in [SZS+99] as well as in [SZ98].

To support guarantees, all routers have to keep some state for each flow. With Dynamic Packet State, instead of keeping per-flow state in each router, the packets carry the state of the flow. This state can be present in the IP header or an IP option could be defined as well as a hop-by-hop extension header (see figure 2.2).

In the following, we distinguish two types of DPS mechanisms. The first type is composed of the mechanisms that try to allocate bandwidth fairly between the flows sharing the network. The mechanisms of the second type are additionally able to provide minimum throughput guarantees to the flows.

[1]A PHB indicates the way a packet with such PHB will be handled by a hop in the network. It enables the network to provide some guarantees to the packet's flow.

[2]The DSCP indicates the class of services to which a packet belongs in a Differentiated Services network.

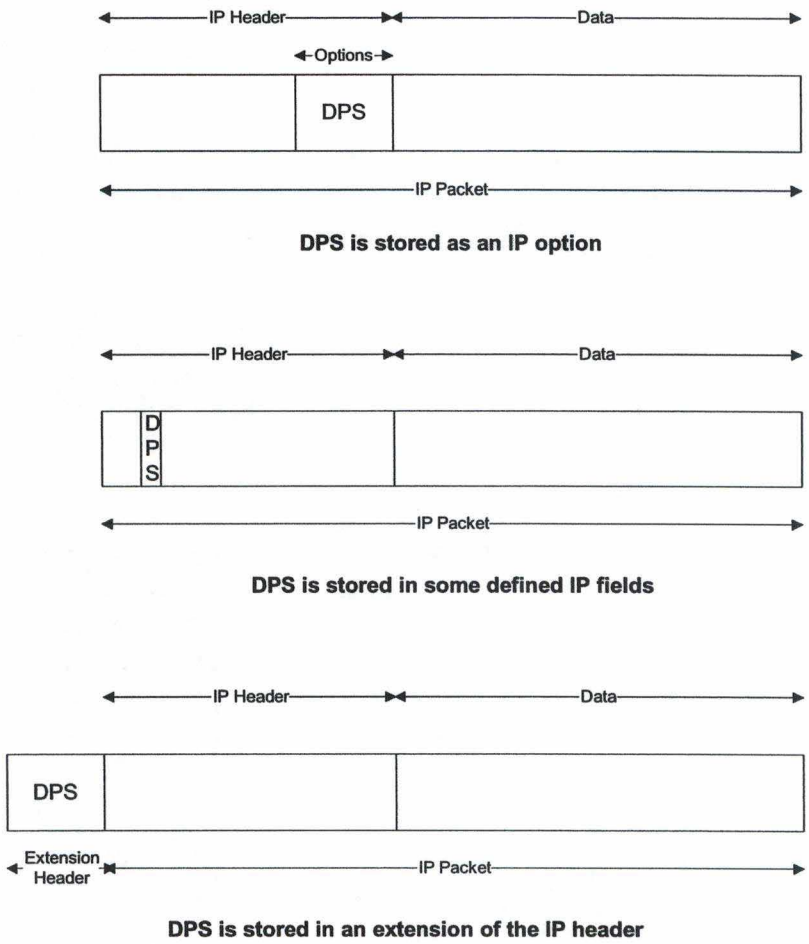

Figure 2.2: Storage possibilities of the DPS

2.2 Fair bandwidth allocation

Almost all Internet traffic is best-effort. When congestion occurs, routers drop packets. Flows react differently to packet drops and congestion depending on the transport protocol they are using. Flows using TCP diminish their sending window, which means that their sending rate decreases. On the other hand, some flows are based on UDP where no congestion mechanism has been implemented[3]. When these two different kind of flows are in competition for bandwidth and no mechanism is used to protect TCP flows, UDP flows monopolize the majority of the bandwidth.

TCP can be configured to modify the default congestion mechanism. It follows that all TCP flows could not react in the same manner to congestion. A protocol could be added to UDP so that UDP flows become sensitive to congestion [Ros01]. For these reasons it isn't enough to make a distinction between TCP and UDP flows and punish UDP without any further knowledge of the flow's behavior. There is a need for a more elaborated mechanism

[3]Some congestion mechanisms have been introduced for UDP but none is widely implemented in practice.

to protect flows from each other and make sure no flow takes away bandwidth from others. Therefore the mechanisms introduced below are trying to provide max-min fairness (see 1.1.1) between the flows sharing the network.

Within the mechanisms that aim to provide the flows with their fair share, there is Core Stateless Fair Queueing (CSFQ). With CSFQ, two flows following the same path and having the same sending rate should get the same throughput. Weighted Core Stateless Fair Queueing (WCSFQ) is a variant of CSFQ. It allows the sharing of the network according to the weight of the flows. These weights introduce relative bandwidth differentiation among the flows. When a weight is associated to the flows, all bottlenecked flows should have the same value for the ratio of the outgoing bandwidth used by the flow over the flows weight in a fair allocation. CSFQ and WCSFQ are presented in [SSZ98b], in [SSZ98a] and in [SZS+99].

2.2.1 Core Stateless Fair Queueing

The goal of Core Stateless Fair Queueing is to approximate the behavior of a network implementing Fair Queueing at each node. In Fair Queueing, there is a packet classification module to determine to which flow a packet belongs. Additionaly, there is a work-conserving scheduling mechanism that computes the time at which the packet should be forwarded and serves packets in order of those times. Each flow should get a portion of the bandwidth according to :

$$rate = \frac{link_{rate}}{nb_{flows}}$$

where nb_{flows} is the number of flows that have at least one packet in the queue. In fair queueing, the value of nb_{flows} is very dynamic. The classification being done by each router, the number of active flows is known at all times by each router.

To provide fair bandwidth sharing, with CSFQ, edge routers estimate the arrival rate of each flow. They insert this estimation in the packets' label. Then, edge and core routers compute the aggregate arrival rate. Based on this approximation, the packet label and the fair share estimation, they calculate a probability to drop the packet. If the dropping probability is greater than zero, the packet is relabeled with the fair share. Then, it is forwarded to the FIFO queue of the output link.Figure 2.3 shows the way edge routers act when a packet arrives. The treatment provided to packets by core routers is illustrated by figure 2.4.

Figure 2.5 displays the pseudo-code of the CSFQ algorithm. The pseudo-code of the fair share estimation is given by figure 2.6. In these figures, i represents the identifier of a flow and p is a packet, AR is the estimation of the arrival rate, FR is the estimation of the portion of the traffic that is forwarded to the queue and FS is the estimation of the fair share. FR is the rate of the packets that are not dropped by CSFQ. This rate may be higher than the rate of the packets that are sent from the queue on the output link because some packets may dropped by the fifo queue.

On packet p arrival, the edge routers perform classification. They determine the flow to which the packet belongs, $flow_i$. The state of this flow is then used to compute the arrival rate of the flow. The arrival rate of a $flow_i$ is determined for any i by

$$AR_{i,new} = (1 - e^{-\frac{T}{K}})\frac{L}{T} + e^{-\frac{T}{K}} AR_{i,old} \tag{2.1}$$

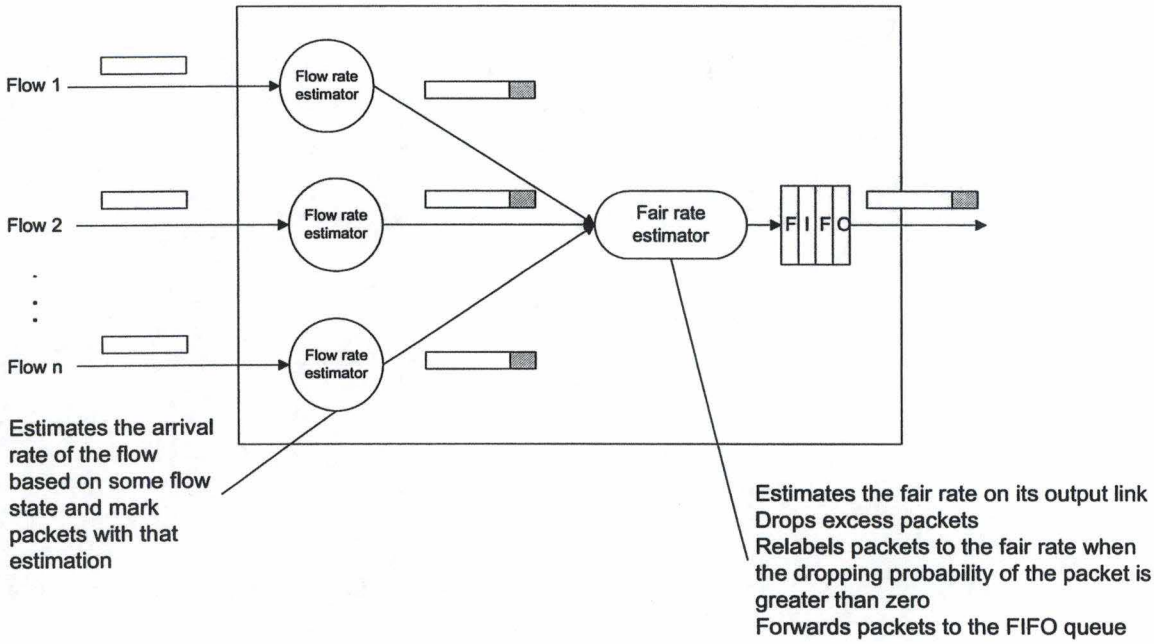

Figure 2.3: Behavior of edge nodes in a CSFQ domain

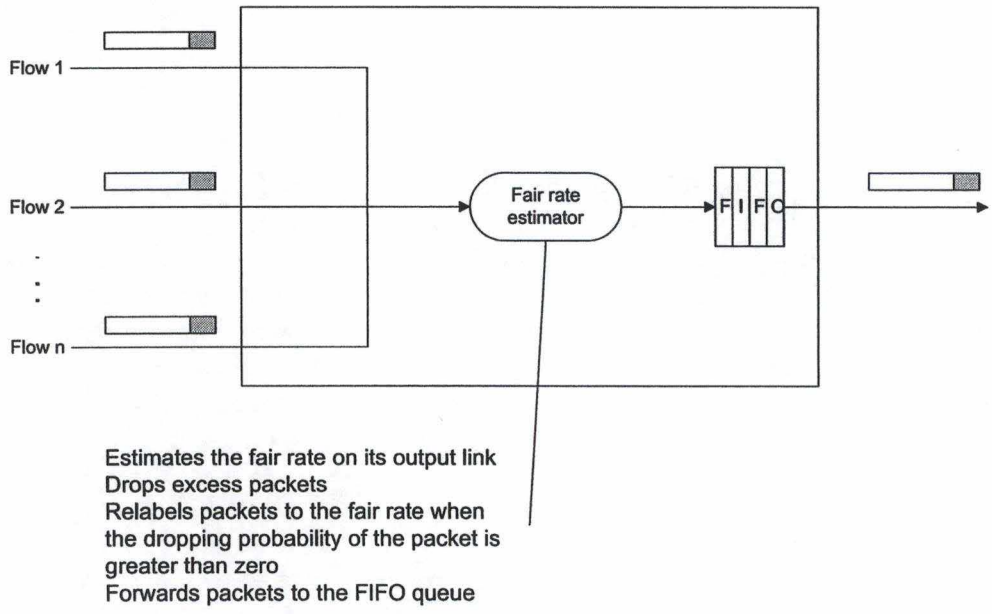

Figure 2.4: Behavior of core nodes in a CSFQ domain

```
On receiving packet p
    if (edge router)
        i = classify(p);
        p.label = estimate_rate(AR_i,p);
    P_drop = max(0,1-FS/p.label);
    if (P_drop > unifrand(0,1))
        FS = estimate_FS(p,1);
        drop(p);
    else
         if (P_drop > 0)
             p.label = FS;  /* relabel p*/
         FS = estimate_FS(p,0);
         enqueue(p);
```

Figure 2.5: CSFQ pseudo-code

where $AR_{i,old}$ is the previous estimation of the arrival rate of $flow_i$ stored in the flow state, K is a constant, L is the length of the current packet p and T is the time elapsed between the arrival of the previous packet of $flow_i$ and the current packet arrival. To obtain T, the arrival time of the last packet of a flow has also to be stored in the flow's state. The new arrival rate estimation of $flow_i$ ($AR_{i,new}$) is used to label the current packet p belonging to $flow_i$. Finally, the flow state is updated. The specific work of the edge routers is then done. After this, the behavior of the edge routers is the same as the one of the core routers.

When a packet p arrives, the core routers compute the drop probability of the packet using equation

$$P_{drop} = \max(0, 1 - \frac{FS}{\text{p.label}}) \tag{2.2}$$

where FS has been estimated on the previous packet arrival and `p.label` is the label of the current packet. Then, a number between 0 and 1 is taken randomly and following a uniform distribution. If the drop probability P_{drop} of the packet is greater than the number taken randomly, then the packet is in excess of the fair share. A new estimation of the fair share is done and the packet is dropped. Otherwise, when P_{drop} is lower than the random number, the packet belongs to the fair allocation. If the drop probability is above zero, the packet is relabeled with the fair share. Then, independently of the value of P_{drop}, the fair share is also estimated. And, after the estimation, the packet is stored in the queue if it is not full.

The estimation of the fair share is done as illustrated by figure 2.6. First of all, the arrival rate of the aggregate traffic at the router and the rate at which the packets are forwarded to the queue are estimated using the exponential averaging, exposed by equation 2.1. Then, it is tested if the arrival rate is bigger than the link rate. If it is the case, the link is supposed to be congested. Otherwise, it is not congested. When the link becomes congested, the variable `congested` indicating the congestion is set and, the beginning of the time window, `start_time`, is set to the current time. If the link was already congested at the previous estimation and the time elapsed since the beginning of the time window is larger than the window size K_c,

```
estimate_FS(p,dropped)
    estimate_rate(AR,p);  /* estimate arrival rate */
    if (dropped == FALSE)
        estimate_rate(FR,p);
    if (AR≥BW)
        if (congested == FALSE)
            congested = TRUE;
            start_time = current_time;
        else
            if (current_time > start_time + K_c);
                FS = FS * BW/FR;
                start_time = current_time;
    else  /* AR < BW */
        if (congested == TRUE)
            congested = FALSE;
            start_time = current_time;
            temp_FS = 0;  /* Used to compute new FS */
        else
            if (current_time < start_time + K_c)
                temp_FS = max(temp_FS,p.label);
            else
                FS = temp_FS;
                start_time = current_time;
                temp_FS = 0;
    return FS;
```

Figure 2.6: Fair share estimation pseudo-code

the fair share is updated according to

$$FS_{new} = FS_{old}\frac{BW}{FR} \quad . \tag{2.3}$$

Then, `start_time` is set to the current time.

The link is considered uncongested when the arrival rate is below the link rate. In this case, the fair share estimation is done differently than when there is congestion. If the link was congested at the previous estimation but is now uncongested, the variable `congested` is set to false and `temp_FS` is set to zero. The variable `temp_FS` is used to compute the maximum of the labels carried by the packets arriving during an interval of K_c length and starting at `start_time`. If there was already no congestion when the previous estimation of the fair share was done, and a period of a window size has not elapsed since `start_time`, `temp_FS` is set to the maximum of its current value and the label of the packet that has arrived. If the link was already uncongested but K_c seconds have elapsed since `start_time`, then the fair share is set to the maximum of the labels of the packets that crossed the router since a window size time and now, `temp_FS`. The time indicating the beginning of the time window is set to the current time.

It is necessary to update the label in case packets of the flow are dropped because then the arrival rate changes and tends to be equal to the fair share. If this wasn't done, the flow could have packets dropped at the next hop even if the fair share of that node is respected because of the dropping that has already been done (see figure 2.7).

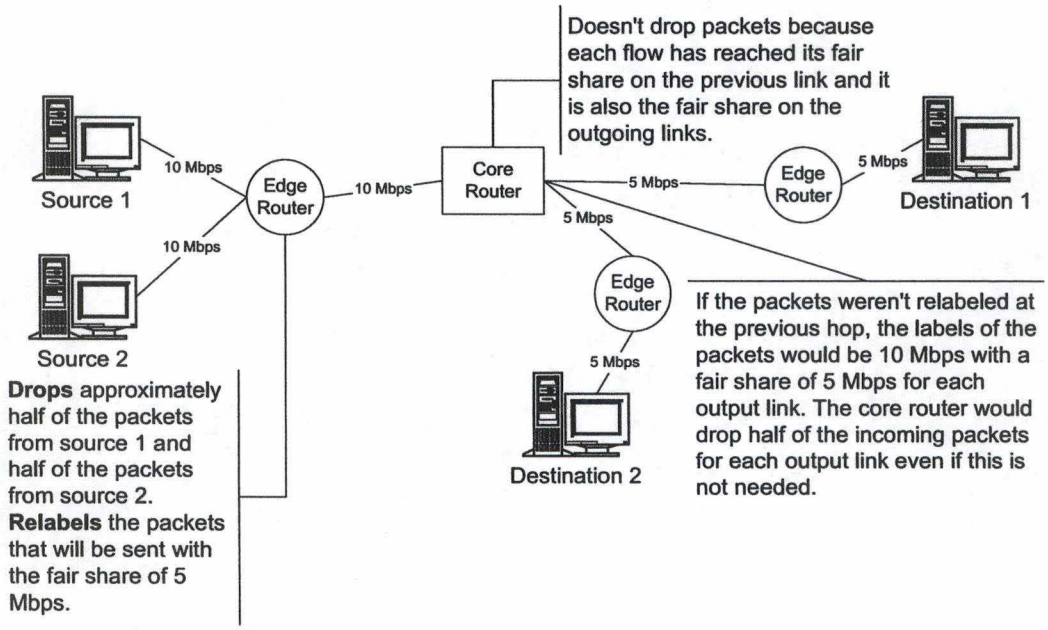

Figure 2.7: Relabeling of packets

The constant K_c is introduced to face the inaccuracies due to exponential averaging. Without this constant, the relationship between the fair share and the forwarding rate has not the time to settle down in the network. The fair share FS is estimated each time the link is congested for at least K_c seconds. It is also computed when the link is uncongested during the last K_c seconds. We can already see that if the link passes from uncongested to congested and vise-versa too quickly (i.e. in a time smaller than K_c) the fair share won't be reestimated.

When the link isn't a bottleneck, the fair share is set to the arrival rate of the most bandwidth taking flow. In this case, there is no need to drop packets from any flow. By setting the fair share to the maximum flow rate, no packets will be dropped because the dropping probability will be zero for each flow according to equation 2.2.

The accepted forwarding rate FR is a function of the estimated fair share FS. We have

$$FR(FS(t)) = \sum_{i=1}^{n} \min(AR_i, FS(t)) \tag{2.4}$$

where n is the number of active flows and AR_i is the rate of flow i. Here, active flows are flows that have a sending rate different from zero. Intuitively, we can see that when the rate of a flow is above the fair allocation, packets are dropped by the bandwidth allocation mechanism to have a rate equal to the fair allocation. But, when a flow sends at a lower rate than the fair share, the forwarding rate for this flow should stay equal to the sending rate.

$FR(.)$ is a continuous function of $FS(t)$. To see that, we have to remember that the sum of continuous functions is also a continuous function. So, we have to show that for any flow i, $\min(AR_i, FS(t))$ is a continuous function of $FS(t)$. For any i, this function can be drawn as

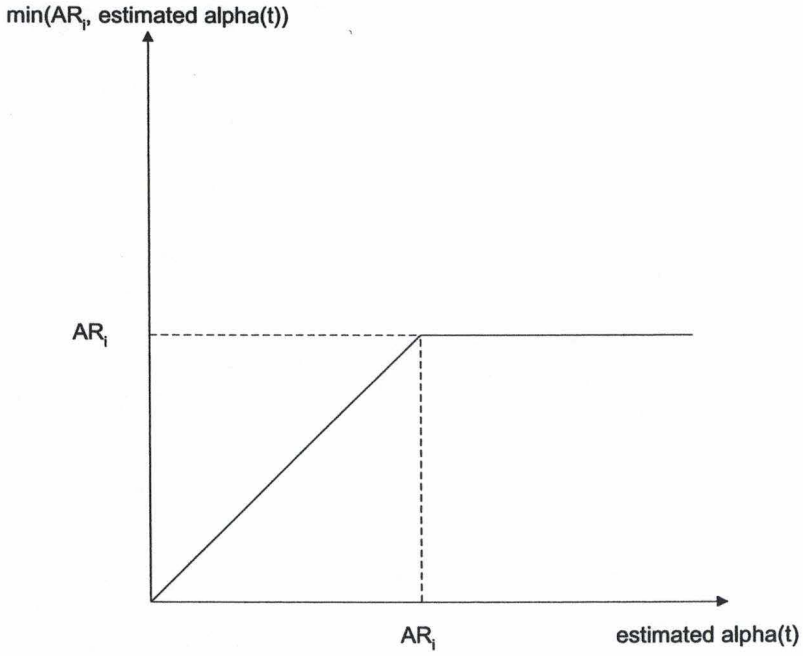

Figure 2.8: Function $\min(AR_i, FS(t))$

shown in figure 2.8. It can then be deduced that $\sum_{i=1}^{n} \min(AR_i, FS(t))$ is also a continuous function (see figure 2.9).

When FS increases, there is less and less flows that reach their fair share. The slope decreases. It follows that FR increases more slowly. FR is a concave function.

FR is piecewise linear. As we can see while FS increases but the same number of flows still have their sending rate above the fair share estimation FS, the slope stays the same. The slope is equal to the number of flows that send at a rate bigger than the fair share estimation FS.

In summary, $FR(.)$ is a continuous, non decreasing, concave and piecewise linear function of $FS(t)$.

In the algorithm, the accepted forwarding rate is approximated by a straight line passing through the origin and with slope $\frac{FR}{FS}$. With what is said before, we can see that at any time the new forwarding rate is always smaller or equal to the one obtained by equation $FR_{new} = \frac{FR}{FS} FS_{new}$. So, FS_{new} will be smaller than the fair share. In that way, overestimation is avoided when there are no changes in traffic. We have to make sure that we don't underestimate the fair share too much either. In fact, the underestimation only takes place when the slope of FR changes. This change is unpredictable because the routers never know the number of flows that are active at some time and their arrival rate. So, we can never predict exactly the forwarding rate and thus the fair share.

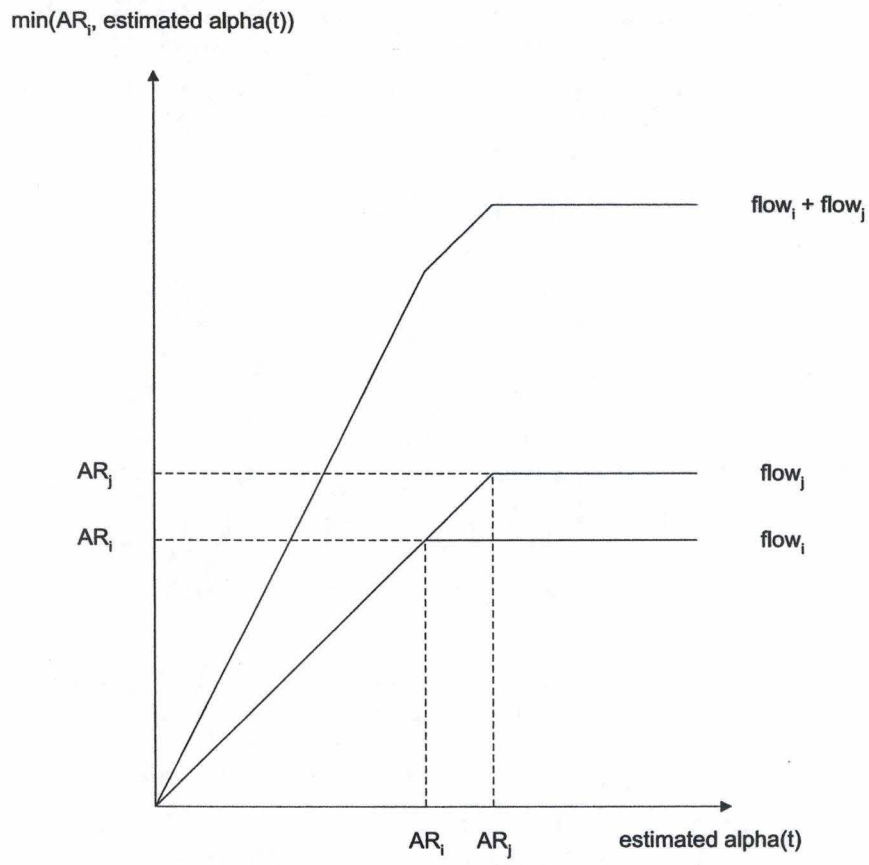

Figure 2.9: Function $\sum_{i=1}^{n} \min(AR_i, FS(t))$

Because this mechanism drops packets probabilistically and because of the various estimations, the amount of data present in the buffer on the output link can fluctuate. The only assurance provided by this mechanism is that each flow will get over a certain time a share of the bandwidth equal to $\min(AR_i, FS)$. It can happen that the buffer overflows. Each time a packet is dropped following a buffer overflow, FS will be decreased by a certain percentage. A limit is fixed to the number of successive times FS can be decreased this way.

In an analogous way, if the link becomes uncongested, that means $AR(t) < BW$, the assumption is being made that the link stays uncongested until some threshold is reached. That allows to store data in the buffer when it is almost empty even if the link becomes congested. In case of rapid load fluctuation, that enables to keep sending at a high rate when the link becomes uncongested for a while without risking buffer overflow when choosing the adequate threshold.

The label stored in each packet represents the arrival rate of the flow to which the packet belongs. It is of type float. This arrival rate estimation is stored in the fragment offset field of the IP header. The fragment offset field may be reused because packets are rarely subject to fragmentation. Fragmentation can be avoided by using a MTU discovery mechanism. The edge node has to detect whether a packet is a fragment or not. When the packet is a fragment,

the fragment offset field is compressed and less room is available for the label[SZS+99]. It is assumed that packets can be fragmented only by egress nodes. Additionally, as exposed in [SZ98], 4 bits of the Type of Service (ToS) field in the IP header, designed for local and experimental use, can also be part of the label.

2.2.2 Weighted Core Stateless Fair Queueing

The functioning of this mechanism is basically the same as that from CSFQ except that now a weight is associated to each flow. This weight indicates the relative importance of the flow as regards other flows. The bandwidth a flow gets should be proportional to its weight. Max-min fairness is achieved when all bottlenecked flows get a share of the outgoing bandwidth AR_i so that the ratios $\frac{AR_i}{w_i}$ are equals (where i indicates the flow).

Each packet of each flow carries a label. The label has value $\frac{AR_i}{w_i}$ for a packet of flow i, where AR_i is the arrival rate and w_i is the weight associated to the flow.

In case of fairness, bottlenecked flows get an amount of the bandwidth that is called the fair share. Note that there are bottlenecked flows only in case of congestion. In that situation, the fair share is the unique solution to the following equation :

$$BW = \sum_{i=1}^{n} w_i \min(\frac{AR_i}{w_i}, FS)$$

In order to achieve fair share per flow, packets are dropped in case the flows transmit at a higher rate than they should. Packets are dropped probabilistically, with a probability

$$P_{drop} = \max(0, 1 - FS\frac{w_i}{AR_i})$$

Table 2.1 shows the similarities and the differences between this weighted mechanism and the previous non-weighted version.

CSFQ	**Weighted-CSFQ**
The label represents the flow arrival rate AR_i, for flow i	The label represents $\frac{AR_i}{w_i}$ where w_i is the weight of flow i
Bottlenecked flows have the same value for AR_i in a fair allocation	Bottlenecked flows have the same value for $\frac{AR_i}{w_i}$ in a fair allocation
$BW = \sum_{i=1}^{n} \min(AR_i, FS)$ when $AR(t) > BW$	$BW = \sum_{i=1}^{n} w_i \min(\frac{AR_i}{w_i}, FS)$ when $AR(t) > BW$
$P_{drop} = \max(0, 1 - \frac{FS}{AR_i})$	$P_{drop} = \max(0, 1 - FS\frac{w_i}{AR_i})$

Table 2.1: Comparison between CSFQ and WCSFQ

The introduction of a weight for each flow enables to set more importance to some flows by comparison to others. It is a relative importance. The bandwidth that a flow will receive will partly depend on the other flows that are present in the network and on their weight.

A flow with a heavy weight that uses a small part of its bandwidth share still allows other flows to take the bandwidth remaining to reach its fair share.

Note that a source should not be able to assign its own weight, unless the price users pay increases with their flows' weights, or else they would choose a weight as heavy as they can in order to get more bandwidth. The introduction of a weight for each flow in Core Stateless Fair Queueing would then be useless. In fact, because edge routers already mark the packets, they should be able to decide of a flow's weight depending on the contract passed with the customer.

There is a limitation to this mechanism. Since a flow's weight is implicitly carried in its packets' labels, the core routers don't know its value. It follows that a flow's weight has to be the same all along an island of routers, that means until an other edge router is met and is enabled to change the flow's relative importance.

2.2.3 Conclusion

Weighted Core Stateless Fair Queueing is an extension of Core Stateless Fair Queueing. When each flow has a unitary weight, Weighted Core Stateless Fair Queueing behaves in exactly the same way as Core Stateless Fair Queueing. We could say that CSFQ is a special case of Weighted-CSFQ.

Weighted Core Stateless Fair Queueing provides a way to share bandwidth in a fair way between different flows that use the network at the same time. The meaning of fairness was exposed. It was also made notice of the fact that fairness implies that the bandwidth allocated to a flow depends on other flows that pass through some of the same ressources in the network at the same time.

This mechanism could be enhanced into Differentiated Services networks. A certain amount of the outgoing bandwidth could be assigned to each class of traffic. For each class, the assigned bandwidth could be shared between the flows of the class using the previously exposed mechanisms. Actually, the arrival rate estimation has been thought to put in the ToS field where the Differentiated Services Code Point associated to a class is stored. When using DiffServ, a new place for the label has to be found.

2.3 Minimum guaranteed bandwidth

A minimum bandwidth guarantee is a guarantee that a flow will get at least a certain amount of the bandwidth independently of the other flows sending properties. A flow for which a minimum guarantee is provided will additionally be allowed to send at a higher rate than the guarantee when the network isn't congested. This enables to use as much resources as possible.

The amount of the bandwidth that is the object of a minimum guarantee is reserved for a flow, no other flow should be allowed to take that bandwidth from it. A back side is that if the flow that made the reservation doesn't use it all, the resource is wasted.

As said before, the minimum quantity of bandwidth attributed to a flow is independent of the other flows' properties and the number of them. It only depends on the reservation that is made for the flow. Minimum bandwidth is for that reason a stronger guarantee than fair

bandwidth allocation where the amount of bandwidth attributed to a flow depends on the network load in terms of the number of flows and eventually of their weight.

Minimum bandwidth guarantees are useful for certain applications' flows that need a certain amount of bandwidth to work correctly. As an example we can take real-time applications. These applications need to send and receive informations at a certain rate so that the receiver can treat the information on time. If the data arrives too late at the destination it is useless. It may also imply certain delay jitter guarantees.

Some applications may be critical to the good functioning of an industry. In that case, it can be interesting to reserve a certain amount of bandwidth for each of these applications to make sure that the rest of the traffic won't jeopardize the industry's activities.

An alternative solution would be to grant high priorities to these flows relative to critical applications. Flows corresponding to non-critical applications would get lower priorities. Priorities are to handle with care because there could be some security problems if applications mark their outgoing packets themselves. Some policing is also important to avoid that high priority applications take all or almost all of the bandwidth from low priority ones.

If some bandwidth isn't the object of any reservation it could be shared between the existing flows. It would be good to distribute this remaining bandwidth in a fair manner. Different mechanisms are trying to enforce that.

2.3.1 Multi-Color Marking Scheme

The Multi-Color Marking Scheme (MC-RED) is composed of a marker and a buffer acceptance algorithm. MC-RED is presented in [PDCE00]. Edge routers use the marker to label the packets and the buffer acceptance algorithm ensures that the packets are forwarded or discarded according to the network load and according to the mark carried by the packets. Core routers only implement the buffer acceptance algorithm.

The marker is used to label the packets of each flow with a drop precedence depending on the rate of the flow. The idea is that flows sending at a low rate will get all of their packets marked with a low drop precedence. On the other hand, flows sending at a high rate will get some of their packets marked with a low dropping priority but the others with a higher dropping priority. In case of congestion, packets with a high drop precedence will be discarded before packets marked with a low drop precedence.

The drop precedence is assigned to a packet using byte based token buckets[4]. A token bucket is characterized by a filling rate R, a size B and the number of tokens currently in the bucket C. At the beginning, the bucket contains B tokens. Each time a packet arrives, if the size of the packet is smaller than C, the packet is transmitted with the marking associated to the bucket and the number of tokens in the bucket (C) is diminished by the packet's size. If the bucket isn't full (i.e. $B > C$), one token is added every $\frac{1}{R}$ seconds. When there aren't enough tokens in the bucket to transmit the packet, the packet can be delayed until enough tokens are available, discarded or passed on to a following token bucket to verify if it does verify its requirements. It depends on the policy choosen. The functioning of a token bucket is illustrated by figure 2.10.

[4]There are also packet based token buckets where each packet smaller than a maximum size consumes one token. In this case, there is an upper bound on the maximum size of the packets.

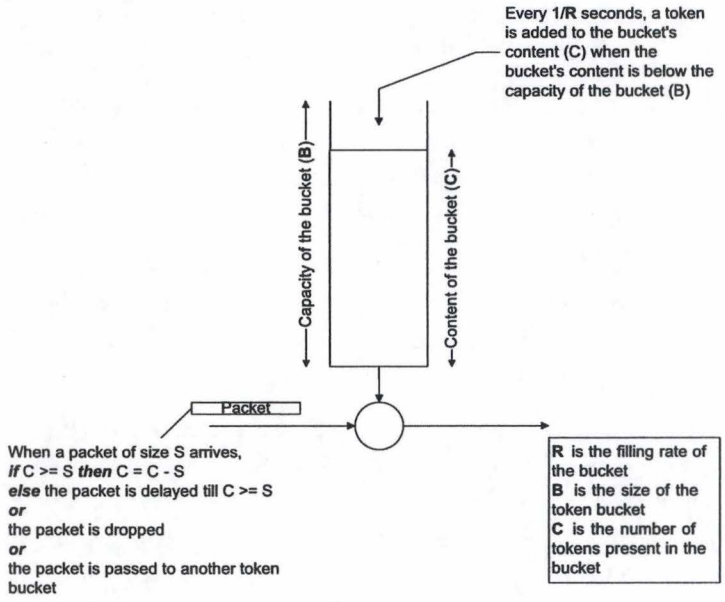

Figure 2.10: Token bucket

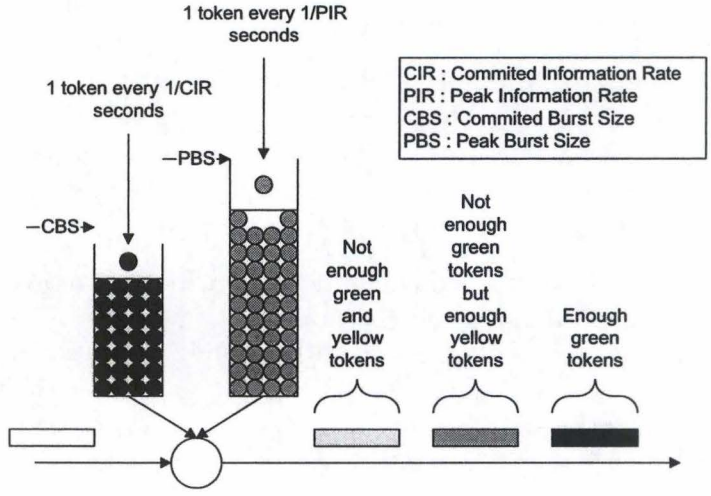

Figure 2.11: Three colors marking scheme

In the mechanism we are interested in, the last option is choosen. There is one token bucket for each color. A color represents a drop precedence. Suppose we have a marker that supports three colors (figure 2.11). When a packet arrives, if there are enough tokens in the first bucket, the packet is marked green, if not, when there are enough tokens in the second bucket, then the packet is marked yellow. In case the number of tokens in the second bucket is less than the packet's size, then the packet is red. When congestion occurs, red packets are discarded first. If the network is still overloaded, yellow packets will start being dropped. Then, at last, green packets will be dismissed. This marker can be generalized to more than

three colors. For a marker of n colors, $n-1$ token buckets are needed.

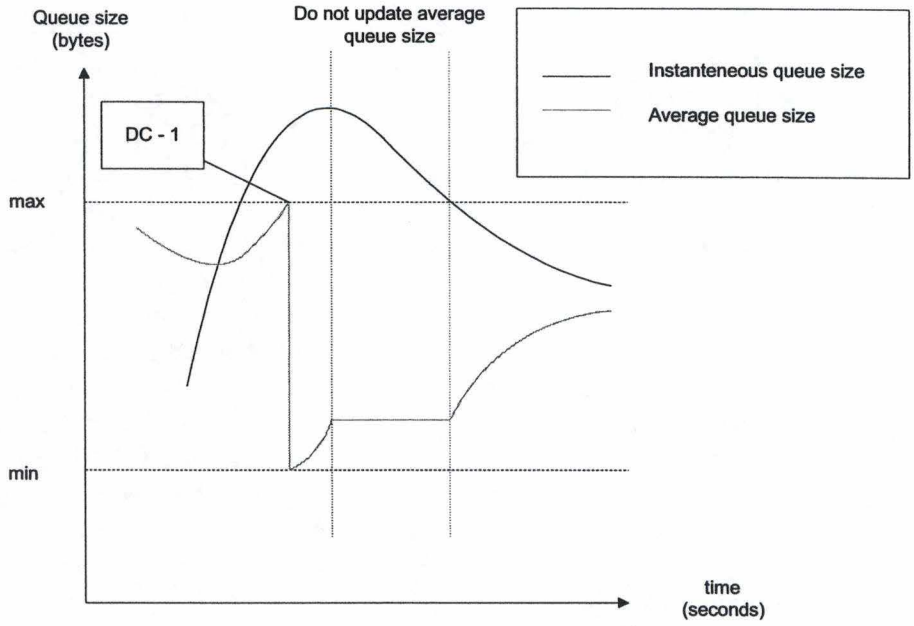

Figure 2.12: DC is decreased

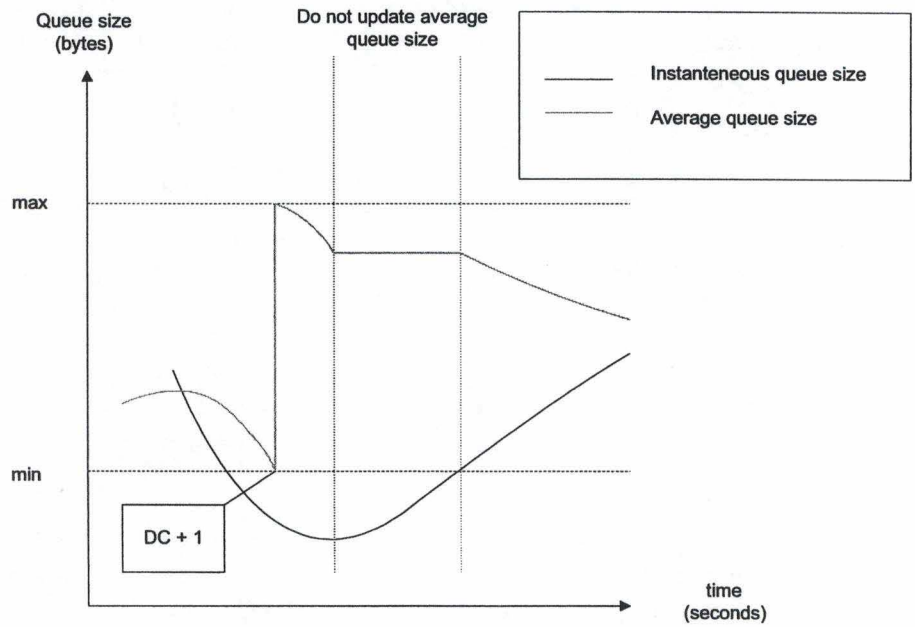

Figure 2.13: DC is increased

The buffer acceptance algorithm determines when packets have to be dropped and which

packets have to be dropped. As a generalization from what was exposed before, packets with color n will be discarded before packets of color $n-1$ where colors are represented by numbers. According to the average buffer occupancy, the algorithm determines a threshold (DC). Packets belonging to layers above DC[5] will be dropped. Packets belonging to layers below DC will be entirely forwarded. Packets from layer DC will be forwarded or dropped based on a Random Early Detection (RED) mechanism. When DC decreases, the layer that was partially discarded is dropped totally and the packets from the layer beneath are dropped with a low probability. If the buffer occupancy starts to grow, the dropping probability will increase for the packets from layer DC. Two threshold are defined : a minimum queue size parameter and a maximum queue size parameter. When the average queue size goes beyond the maximum queue size, DC is decreased. When the average queue size goes below the minimum queue size, DC is increased. If the average buffer occupancy starts to grow, the dropping probability will increase for the packets from layer DC. While passing from DC to $DC-1$, the average queue occupancy is set to the minimum queue size. If the instantaneous queue occupancy decreases, the average queue occupancy is maintained (figure 2.12). It will only start to change when the instantaneous queue occupancy gets equal to the maximum queue size. When the buffer occupancy gets below the minimum queue size, DC is increased so that one more layer is accepted (figure 2.13). The average queue size will be updated to the maximum queue size. If the instantaneous queue size starts to grow, the average queue size is kept constant until the instantaneous occupancy becomes equal to the maximum queue size. This is done to avoid instability of the system. It avoid useless changes of DC due to the time needed for the system to react to the changes in DC.

The multi-color marking scheme provides fair bandwidth allocation to each flow. This is obtained by configuring the marker in the same way for each flow. That means that the thickness of the layers should be the same for each flow. The thickness T of a layer is set to

$$T = \frac{\text{maximum flow rate in the network}}{\text{number of colors}} \quad .$$

The marker and the buffer management algorithm then ensure that bandwidth is distributed in a fair manner. Flows that are bottlenecked at a node in the network get the same amount of the outgoing bandwidth of the node because packets of layer DC are dropped probabilistically using RED.

To support weights for fair allocation of bandwidth, the marker is configured using the weight of the corresponding flow to determine the layers thickness. The thickness of the layers for a flow of weight w is equal to the thickness of the layers when no weights are used times w.

We notice that even if the main objective of multi-color marking scheme is to allocate the bandwidth fairly, this mechanism could be easily adapted to ensure minimum bandwidth guarantees. The only thing that has to be done is to put the thickness of the first layer in the marker relative to a flow equal to the bandwidth guarantee of the flow. The other layers will all have the same thickness for every flow. We have to make sure that green packet won't be dropped. This is made possible by performing admission control at the edge of the network. No ressources should be the object of a reservation if it is not available.

[5] DC stands for Drop Color.

By using multi-color marking scheme, the amount of bandwidth that isn't used can be shared in a fair way between flows sending at a higher rate than the reservation or flows that aren't the object of any reservation. This mechanism allows to use the bandwidth that is the object of a reservation for a flow by an other flow if the first one doesn't use all its reservation.

The number of parameters to configure is rather small. Six parameters are enough to characterize all the functioning : the minimum and maximum queue size, the number of token buckets, the size of the buckets, the filling rate of the buckets and the probability $prob_{max}$ at which packets of the DC layer are dropped when the average queue occupancy is equal to the maximum queue size. When the average queue size equals the minimum queue size, the dropping probability for packets of layer DC is zero. The intermediate dropping probabilities can be obtained considering the fact that the dropping probability evolutes linearly between zero and $prob_{max}$ when the average queue occupancy is comprised in [minimum queue size, maximum queue size].

2.3.2 Core Jitter Virtual Clock

Core Jitter Virtual Clock (CJVC) is introduced in [SZ98] and in [SZS+99]. This mechanism uses a combination of a rate controller and a scheduler. The router attributes an eligible time and a deadline to each packet. Packets stay in the rate controller till their eligible time and then the rate controller passes the packets on to the scheduler. The scheduler transmits the packets in increasing order of their deadline. The role of the shaper is to bound the delay jitter of the packets of a flow and to uniformly distribute the allocation of buffer space inside the network [ZF94], while the scheduler is responsible of the bandwidth allocation. The shaper and scheduler used for this mechanism are the ones from Jitter Virtual Clock (figure 2.14). The computation of the eligible time and the deadline is adapted in Core-Jitter Virtual Clock to eliminate the need for per-flow information in the core routers.

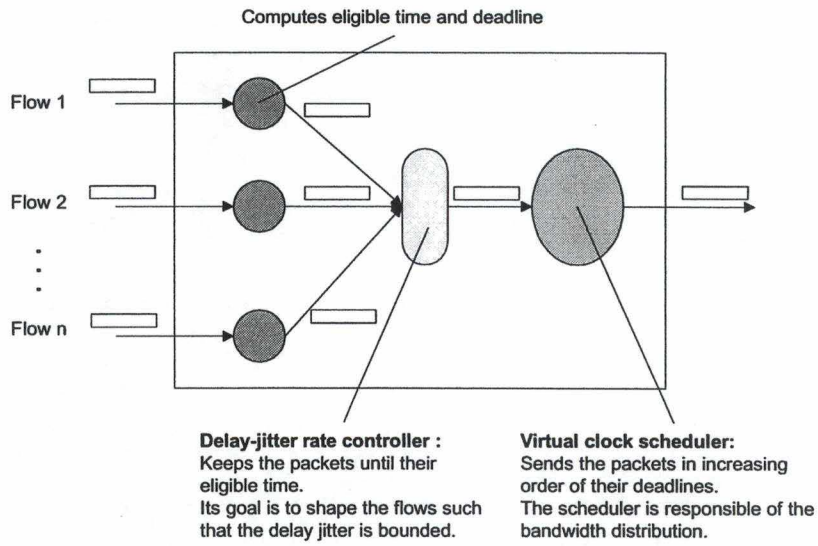

Figure 2.14: Router implementing Jitter Virtual Clock

In Jitter Virtual Clock, the eligible time and the deadline of the packets are computed according to the following equations :

$$e_{i,j}^1 = a_{i,j}^1 \tag{2.5}$$

$$e_{i,j}^k = \max(a_{i,j}^k + g_{i,j-1}^k, d_{i,j}^{k-1}), \quad i,j \geq 1, k > 1 \tag{2.6}$$

$$d_{i,j}^k = e_{i,j}^k + \frac{l_i^k}{r_i}, \quad i,j,k \geq 1 \tag{2.7}$$

where $e_{i,j}^k$ is the eligible time of the k^{th} packet of flow i at node j, $a_{i,j}^k$ is the arrival time at node j of this packet, $d_{i,j}^k$, stamped into the packet header at the previous node, is its deadline, $g_{i,j-1}^k$ is the time ahead of schedule at node $j-1$ for the k^{th} packet of flow i, l_i^k is the length of the packet and r_i is the rate of flow i. The time ahead of schedule $g_{i,j}^k = d_{i,j}^k - s_{i,j}^k$, where $s_{i,j}^k$ is the sending time of the k^{th} packet of flow i at node j, has to be a positive value.

It is easy to see that, instead of keeping a state with the rate of the flow and the deadline of the previous packet at each node for the flow, the rate of the flow can be carried by each packet belonging to the flow. It isn't that obvious for the deadline of the previous packet at the node. One way to avoid keeping the deadline of the last packet of each flow in the core routers would be to compute at the edge of the network the eligible and deadline times of the packet at each node. That way, only edge routers have to keep the deadlines at the core nodes of the last packet of the flows. This is unthinkable in today's internet, where the number of nodes crossed by a packet to reach its destination can be huge. The eligible times and the deadlines of a packet would have to be carried in the packet's header. The size of the header would than be too big in proportion of the data carried by the packet. The overhead would be enormous.

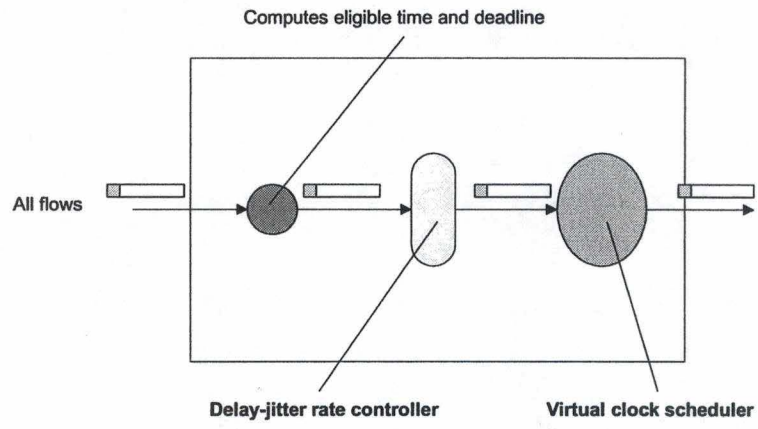

Figure 2.15: Router implementing Core-Jitter Virtual Clock

The deadline of the last packet of a flow is used at a node to compute the eligible time of the next packet of the same flow at that node. This eligible time is computed according to 2.6. In that equation, the eligible time is obtained by taking the maximum from the arrival rate of the packet added to the time ahead of schedule of the packet at the previous node which can also be carried in the packet header and the deadline of the previous packet at this

node. If the deadline of the previous packet is always smaller than the first argument of the maximum than its value would be useless for the computation. In other words, the goal is to find a value δ_i^k as small as possible such that

$$a_{i,j}^k + g_{i,j-1}^k + \delta_i^k \geq d_{i,j}^{k-1}, \quad j > 1.$$

It is shown in [SZ98] that such a value exists and is equal to

$$\begin{cases} \delta_i^1 & = & 0 \\ \delta_i^k & = & \max(0, \delta_i^{k-1} + \frac{l_i^{k-1} - l_i^k}{r_i} - \frac{e_{i,1}^k - e_{i,1}^{k-1} - \frac{l_i^{k-1}}{r_i}}{h-1}), \quad k > 1, h > 1 \end{cases}$$

where the number of hops h on the path of flow i can be computed at admission time.

From equations 2.6 and 2.7, it can be seen that Core Jitter Virtual Clock provides maximum guaranteed bandwidth to the flows. In addition, we can say that this guarantee for flow i is equal to its reserved rate r_i. Intuitively, we can see that the arriving rate of flow i at the scheduler, at all nodes of the network, is below or equal to r_i, because the eligible time of a packet is smaller or equal to the deadline of the previous packet and the deadline of a packet corresponds to the finish time of transmission of this packet on a dedicated link of capacity r_i for packets of flow i. From this observation, we can conclude that a router can only send packets from a flow at a higher rate than its reservation during a limited period of time that depends on the amount of buffers available to this flow.

The mechanism exposed in this section provides guarantees on end-to-end delay. It provides the same guaranteed service as Jitter Virtual Clock and Weighted Fair Queueing in terms of delay[SZ98].

In terms of bandwidth, the guarantees furnished by Core Jitter Virtual Clock are stricter than the ones provided by Weighted Fair Queueing where the bandwidth a flow gets depends on the other active flows' weights. It isn't the case for Core Jitter Virtual Clock where the bandwidth used depends only on the flow's sending properties and the bandwidth reserved.

By comparison to Weighted Fair Queueing, in Jitter Virtual Clock the scheduling of a packet is independent of the other flows. That's why it is easier to make Jitter Virtual Clock stateless in the core of the network than Weighted Fair Queueing. To compute the deadline of the packets on an output link in Weighted Fair Queueing, the router needs to know the weight of all the flows passing through it and leaving on the output link. A flow is not aware of the presence of other flows in the network and of their weight. Therefore, packets from a flow cannot carry in their headers the weight of the active flows so that core routers would be able to compute packets deadlines without keeping per-flow state. Estimations have to be made to approximate networks implementing Weighted Fair Queueing without keeping state for each flow in the core of the network. It's not the case with Jitter Virtual Clock where the deadline of a packet only depends on the flow state and doesn't depend on the other flows.

Maximum delay guarantees are provided even if this mechanism forces all packets, by the way of the eligible time, to incur the maximum delay. This is also a way of avoiding delay jitter. When delay jitter is reduced, the traffic burstiness is limited and, as a result, buffer space requirements and schedulers complexity are reduced. A flow's traffic can become bursty due to load fluctuations in the network.

Note that to implement Jitter Virtual Clock without maintaining per-flow state in the core of the network, slack variables δ_i^k, where i indicates the flow and k indicates a packet from the flow, have been introduced. These variables have a direct influence on delay. So that the delay introduced is as low as possible, these variables should be as near as possible to the solution x_i^k of equation

$$a_{i,j}^k + g_{i,j-1}^k + x_i^k = d_{i,j}^{k-1}, \quad i,j \geq 1, k > 1$$

where x_i^k is a lower bound for δ_i^k.

In this mechanism, it can be seen that the slack variable δ_i^k is independent of the node even if there is one equation 2.6 for each packet k of each flow i at each node j. In fact, the slack variable has to be independent of the node because the slack variable is computed at the border of the network and is carried in each packet's header. Therefore, for one packet of a flow, δ has to be identical for each node on the packet's path. If it were computed at each node, nodes would have to perform per-flow operations, which is what is trying to be avoided in stateless mechanisms. Computing δ for each node at the border and carry the different values in the pachets' header would introduce packets of enormous sizes.

2.3.3 Fair Allocation Derivative Estimation

Fair Allocation Derivative Estimation (FADE) is in fact an estimation algorithm used for fair share estimation of links in the network. The basic idea of this mechanism is that interior nodes estimate the fair share of their input links and transmit these estimations to boundary nodes by a feedback mechanism. Based on the fair share estimations of the links in the domain, edge nodes will compute the fair share of each flow.

In the presentation of FADE [LBL00], a distinction is made between flows that are called high priority flows and the other flows. These high priority flows correspond to flows that are always able to use a certain portion of the bandwidth. Additionally, they are limited at the edge to this amount of bandwidth in order to also allow other flows to use network resources. These flows may correspond to critical applications and they are not allocated bandwidth with FADE. FADE does not take care of these flows. It only estimates the aggregate arrival rate of the high priority flows in order to know the amount of bandwidth that is left to allocate to the lower priority flows.

The bandwidth available on link k at time t that has to be allocated fairly between flows is expressed by

$$B^k(t) = \rho\, BW^k - A_{high}^k(t)$$

where BW^k is the total link bandwidth at link k, ρ is the target utilization of the link and $A_{high}^k(t)$ is the sum of the arrival rates of high priority flows on link k at time t.

The portion of the bandwidth $FS^k(t)$ that each flow bottlenecked at link k at time t should get can be obtained by computing

$$FS^k(t) = \frac{B^k(t) - \hat{A}^k(t) - \mu^k(t)}{w^k(t)}$$

where $\hat{A}^k(t)$ is the sum of the arrival rates of flows passing through link k and bottlenecked elsewhere in the network at time t. $\mu^k(t)$ is the sum of the minimum arrival rates of the flows bottlenecked at link k at time t. The minimum arrival rate of a flow corresponds to its minimum

bandwidth guarantee. But, this type of reservation can cause a waste of resources if the flow sends at a lower rate than its minimum guarantee. Therefore, it would be better to say that the minimum arrival rate of a flow corresponds to min(*minimum guarantee*, *sending rate*). $w^k(t)$ is the sum of the weights of the flows bottlenecked at link k at time t. It is sometimes also referenced as the number of flows bottlenecked at link k at time t.

It can be seen that computing the fair share at a link $FS^k(t)$ requires that interior nodes keep some per-flow information related to the reservation of the flows and the weight of the flows passing through them. Such calculation can therefore not be processed in core nodes. It follows that $FS^k(t)$ has to be estimated.

In case of max-min fairness allocation, the fair share of a flow is the minimum of the fair shares of the links crossed by the flow.

$$FS_i(t) = \min_{k \in \{k:\text{link } k \text{ is crossed by flow } i\}} FS^k(t)$$

When a boundary node knows the links crossed by a flow as well as their fair share, it can compute the flow's fair share $FS_i(t)$. The weighted fair allocation for a flow i is

$$FA_i(t) = \mu_i + w_i \, FS_i(t)$$

in which μ_i is the minimum of the bandwidth guarantee provided to flow i and its sending rate, following the comment made about $\mu^k(t)$ a few paragraph ago. And, w_i is the weight of flow i.

The calculation of the input links' fair shares is done by interior nodes based on a relation found between the arrival rate on a link at time t and the fair share at time t on that link. In the following, $N^k(t)$ represents the set of all flows bottlenecked at link k at time t and $\hat{N}^k(t)$ is the set of flows bottlenecked elsewhere. It can be noticed that the fair share at link k is greater or equal to flows' fair shares of flows bottlenecked elsewhere in the network.

$$FS^k(t) \geq FS^i(t) \quad \forall i \in \hat{N}^k(t)$$

The total arrival rate on link k at time t can be expressed as follows:

$$\begin{aligned}
A^k(t) &= \textstyle\sum_{i=1}^{n} FA_i(t) && \text{It is supposed that each flow sends at its fair rate.} \\
&= \textstyle\sum_{i \in \hat{N}^k(t)} FA_i(t) + \sum_{i \in N^k(t)} FA_i(t) && \text{If a flow doesn't use all its bandwidth, it is lost for other flows.} \\
&= \hat{A}^k(t) + \textstyle\sum_{i \in N^k(t)} FA_i(t) \\
&= \hat{A}^k(t) + \textstyle\sum_{i \in N^k(t)} (\mu_i + w_i FS_i(t)) && \text{Because } FA_i(t) = \mu_i + w_i FS_i(t). \\
&= \hat{A}^k(t) + \textstyle\sum_{i \in N^k(t)} \mu_i + \sum_{i \in N^k(t)} w_i FS_i(t) \\
&= \hat{A}^k(t) + \textstyle\sum_{i \in N^k(t)} \mu_i + FS^k(t) \sum_{i \in N^k(t)} w_i && \text{Because a flow } i \text{ bottlenecked at link } k,\ FS_i(t) = FS^k(t) \\
&= \hat{A}^k(t) + \mu^k(t) + FS^k(t) w^k(t)
\end{aligned}$$

where

$\mu^k(t) = \sum_{i \in N^k(t)} \mu_i$ is the bandwidth that is already allocated to flows bottlenecked at link k at time t and

$w^k(t) = \sum_{i \in N^k(t)} w_i$ is the total weight of the flows that are bottlenecked at link k at time t.

We obtained the equation

$$A^k(t) = \hat{A}^k(t) + \mu^k(t) + FS^k(t)w^k(t)$$

Let's suppose for a moment that $\hat{A}^k(t)$, $\mu^k(t)$ and $w^k(t)$ are constant. This is the case when the same flows stay bottlenecked at link k. It can be seen that, when $FS^k(t)$ changes in an interval such that the other three variables stay constant, $A^k(t)$ remains on the same line segment. When $FS^k(t)$ increases to the point where flows start becoming bottlenecked elsewhere, $w^k(t)$ decreases. $A^k(t)$ then moves to the next linear segment, which has a smaller slope. $A^k(t)$ is expressed as a continuous, non-decreasing, concave and piecewise-linear function of $FS^k(t)$.

Interior nodes will compute the fair share of their links based on the previous observations. They estimate $w^k(t)$ by taking into account that $w^k(t)$ is the slope of a linear piece of the expression of $A^k(t)$ in function of $FS^k(t)$.

$$w^k(t) = \frac{A^k(t) - A^k(t-1)}{FS^k(t) - FS^k(t-1)}$$

As the goal is to achieve the target utilization of the bandwidth $B^k(t)$ at all times, the interior node has to find out the fair share relative to this target. Because the target utilization rate changes along with the time depending on the sending rates of high priority flows, an interior router can only know the previous target utilization bandwidth. It will compute the fair share at time $t+1$ by supposing that the target utilization at time $t+1$ will be the same as the one from time t. The fair share at time $t+1$ will be obtained by

$$w^k(t) = \frac{B^k(t) - A^k(t)}{FS^k(t+1) - FS^k(t)}$$

$$FS^k(t+1) - FS^k(t) = \frac{B^k(t) - A^k(t)}{w^k(t)}$$

$$FS^k(t+1) = FS^k(t) + \frac{B^k(t) - A^k(t)}{w^k(t)}$$

For FADE to work correctly, a few assumptions have been made :

- Enough resources have to be available to support the minimum bandwidth reservations.
- A feedback mechanism has to be used to communicate links' fair shares to boundary nodes[6].
- Flows have to react quickly to changes in their fair share and adjust their rate.

FADE helps to allocate in a fair manner the bandwidth that isn't reserved by a flow in the Diffserv framework. Part of the bandwidth is allocated to each class of service. Inside each class of services, bandwidth reservations are made and the remaining has to be shared fairly between the flows belonging to the class. FADE aims to provide fairness between flows

[6]In the following, we will see that this feedback mechanism is not mandatory. The estimation of the fair share performed by the core routers can also be used in CSFQ where no feedback mechanism is used and dropping takes place at all nodes.

from the same aggregate class and that way to avoid that aggressive flows use most of the residual bandwidth. This is done by introducing some dynamic cooperation between interior and boundary nodes.

High priority flows aren't getting more bandwidth than the limit that is allocated to them even if some bandwidth remains unused because otherwise they would steal bandwidth from flows having lower priorities even if they made bandwidth reservation. No protections would be given to those flows. The amount of bandwidth remaining for those flows could never be calculated and there would be no way to be able to know if some reservations can be made.

Because FADE is thought to work in Diffserv networks, it enables a network that support a range of services to provide flow isolation and protection. Flows from the same class of service will be protected from each other and will get a fair share of the bandwidth which depends on the class they belong to.

Nothing in this mechanism, except forwarding the packet, is done on a per-packet basis. The fair share estimation takes place after a fixed amount of time independently of the packets received. The same remark can be done for the communication between interior and boundary nodes. In the previous mechanisms, boundary nodes would label the packets and interior nodes would use this label to forward the packet and provide the guarantees associated to the label. Here, interior nodes make the computation and communicate the results to boundary nodes which use the feedback informations to provide the guarantees to the flows. FADE works in the control plane ; the previous mechanisms work in the data plane. With FADE the forwarding path is kept simple but more control information has to be carried by the network. The packet forwarding algorithm can stay unchanged from what is provided in today's IP networks.

The third assumption that is being made is a limitation to this mechanism. It is said that flows have to react quickly to adjust their rate to their fair share rate. This adjustment is made at the edge of the network. The egress edge router has to perform policing and shaping for the flow to respect it's fair share. The problem is that even if a flow sends at it's fair share, when it is multiplexed with other flows in the network, its rate can change and the flow can become bursty. So it cannot be deduced by the fact that a flow is shaped to a rate that it will stay that way throughout the network. It follows that the assumption may not be true in the interior of the network. Additionally, a buffer acceptance algorithm may have to be used by interior nodes to avoid buffer overflow in case the buffer capacity predicted is insufficient because of traffic bursts.

Concerning the assumption that there is enough ressources to support the minimum bandwidth guarantees of the flows, it can be enforced by performing admission control at creation of the flows.

FADE is a mechanism that estimates the fair share of a flow based on a feedback mechanism. It estimates the fair share of the links and these are communicated to the boundary nodes who compute the fair share of each flow passing through them.

When the fair share at a link increases, flows that were bottlenecked on the link can start to be bottlenecked elsewhere in the network. The sum of the weights of the flows bottlenecked at the link decreases. As a consequence, the aggregated arrival rate is on another line segment with a smaller slope $w^k(t)$ than the previous line segment. It follows that the fair share of a link is never overestimated when the aggregate arrival rate increases. When the aggregate arrival rate decreases, the link fair share may be overestimated but it is never bigger then the

previous fair share so there is enough bandwidth for the same number of flows bottlenecked with this fair share.

It seems that the link fair share estimation is more accurate then the estimation that takes place in Core Stateless Fair Queueing. I would propose to replace the estimation in CSFQ by the estimation made by FADE. This will only make the fair share estimation in CSFQ more accurate. It won't enable CSFQ to support minimum guaranteed flows. The third assumption made for FADE is still valid but this is guaranteed by the probabilistic dropping that takes place in the core for CSFQ. Therefore, in the estimations, the forwarding rate of the output link will be used instead of the arrival rate on the input link (FADE).

2.4 Comparisons

Table 2.2, shows the different properties of the mechanisms exposed previously. In this table, the first two lines give an indication on whether the mechanisms are based on DPS or on a feedback mechanism. The major utilization purposes of these mechanisms can also be seen. And, finally, it is indicated if admission control is needed or not.

	CSFQ	**MC-RED**	**CJVC**	**FADE**
Works in the data plane	Yes	Yes	Yes	No
Works in the control plane	No	No	No	Yes
Fair Allocation of the bandwidth	Yes	Yes	No	No
Supports minimum guaranteed flows	No	Yes	No	Yes
Supports maximum guaranteed flows	No	No	Yes	No
Supports minimum guaranteed flows with fair allocation of the remaining bandwidth	No	Yes	No	Yes
Supports weights for fair allocation	Yes	Yes	No	Yes
Supports high priority flows	No	No	No	Yes
If a flow doesn't use its fair share it can be used by other flows	Yes	Yes	No	No
Admission control has to be performed	No	Yes	Yes	Yes

Table 2.2: Comparison of the exposed mechanisms

2.4.1 CSFQ versus FADE

From the previous presentation of CSFQ and FADE, a lot of similarities, concerning the estimation of the fair share, can be underlined. Therefore, these two estimation algorithms are compared in table 2.3. Then, we show that the estimation performed by FADE should be more accurate than the one performed by CSFQ. A proposal of replacing the current CSFQ fair share estimation by FADE is made to obtain a better fair bandwidth allocation mechanism.

Based on the fact that both functions $F(.)$ and $A(.)$ are concave and that the forwarding rate on an output link is equal to the arrival rate on the input link at the next node along

CSFQ	FADE
A router computes the fair share for its ***output*** links.	A router computes the fair share for its ***input*** links.
The aggregated ***forwarded*** rate F is a continuous, concave, non-decreasing and piecewise-linear function of the estimated fair share.	The aggregated ***arrival*** rate A is a continuous, concave, non-decreasing and piecewise-linear function of the fair share.
F is approximated by a straight line passing through the ***origin*** and of slope $\frac{F(FS(t-1))}{FS(t-1)}$ where t is the actual time and t-1 is the time of the previous estimation.	A is approximated by a straight line passing through ***point*** $(FS^k(t), A^k(t))$ and of slope $w^k(t)$
$F(FS(t))$ is computed by the routers and based on the value that is found, an estimation of $FS(t)$ can be deduced from the approximation of F.	$w^k(t)$ is obtained by approximating $A(.)$ by a straight line passing through $(FS^k(t-1), A^k(t-1))$ and $(FS^k(t), A^k(t))$. $w^k(t)$ is the slope of this straight line. $A^k(t+1)$ is set to $B^k(t)$ where $B^k(t)$ is the bandwidth utilization goal at time t. The fair share $FS^k(t+1)$ is obtained by equation $$FS^k(t+1) = FS^k(t) + \frac{B^k(t) - A^k(t)}{w^k(t)}$$

Table 2.3: Comparison of CSFQ fair share estimation with FADE

the link, I deduce that the estimation of the fair share provided by FADE is better than the one provided by CSFQ. The slope of the straight line used for the approximations in FADE is always smaller or equal to the slope of the straight line in the approximations carried out by CSFQ. Additionally, it has been shown that these two slopes are always greater or equal to the slope of the line segment of the real function (by contradiction to the estimated function). So the slope used in FADE is a better estimation than the slope used in CSFQ.

A proposal to improve the fair share estimation in Core Stateless Fair Queueing would be to use Fair Allocation Derivative Estimation to approximate the fair share on the routers' outgoing links in the network. It would still be based on the computation of the aggregated forwarding rate on the link because the aggregated forwarding rate on a link is equal to the aggregated arrival rate on that link at the next node in the network. The probabilistic dropping of packets for flows sending at a higher rate than their fair share at a node would still take place in the node. It is then ensured that the forwarding rate at a node tends to be equal to the fair share of the outgoing link at that node.

This solution may not be perfect because, in FADE, the estimation of the fair share is made using a straight line passing through the two previous estimations of the couple (fair share, accept rate). And, this straight line is used as an approximation of an increasing (and piecewise linear) function. The slope of such a function is always positive. Therefore, the line used for the approximation should also have a positive slope. But, it is not always true with

the line resulting from the previous estimations because the slope of such a line is obtained by :

$$slope = \frac{AR(t) - AR(t-1)}{FS(t) - FS(t-1)} \quad . \tag{2.8}$$

To have a positive slope, the fair share and the accepted rate have to increase at the same time or to decreases together. That means that when the fair share increases, more bandwidth can be used by the flows and the accepted rate increases. When the fair share decreases, less packets are accepted and therefore the accepted rate decreases. But, this only holds when the network load is constant which is not usually the case in practise where flows are created or torn down all the time. Let's take as a first example the case when the fair share increases. In consequence, less packet should be dropped because less packets of each flow are in excess of the fair share, if the arrival rate stays constant. It follows that the accepted rate should increase. But, if the arrival rate decreases, the increase of the fair share will not cause an increase of the accepted rate. And, the slope of the line used for the estimation becomes negative. Secondly, when the fair share decreases, more packets of the existing flows are dropped. But, if new flows are created, the accepted rate may increase because each flow is allowed to send at the fair share. These two examples lead in situations where the slope of the line used for the estimation is negative. The line is not an approximation of the relation between the fair share and the accepted rate anymore. Because, the hypothesis of a stable arrival rate cannot be made a solution to this problem has to be found. A possibility would be to base the new fair share estimation on the previous slope but this does not seem to allow frequent changes in the network load.

2.4.2 Adaptations to CSFQ

To support minimum guaranteed flows with fair allocation of the remaining bandwidth, a possible approach would be to perform admission control at flow creation to see whether the minimum bandwidth guarantee of the flow can be supported by the network. On packet arrival, edge routers will check if the packet belongs to the guarantee provided to the corresponding flow or not. Guaranteed packets will be attributed a label of value zero indicating that it doesn't use a portion of the bandwidth that has to be allocated fairly. Excess packets of a flow will be labeled with the estimation made by the edge router of the rate of the excess traffic of the packets' flow. No changes are needed in core routers except that they need to know or to estimate the amount of bandwidth that is reserved to figure out the amount of bandwidth that can be shared. They will only drop packets from the excess traffic when its rate, carried in the packets' labels, is higher than the fair share of the outgoing link. Because the packets that are guaranteed are marked by a rate of zero, it never exceeds the fair share of any link and therefore such packets should never be dropped. An exception is made when the buffer is full and no more packets can be accepted. Note that this situation should be avoided because guaranteed packets should not be dropped.

To support high priority flows, almost the same modifications as the ones from previous paragraph can be proposed. Packets from high priority flows are marked with label zero by border routers. The ingress edge routers have to make sure that the bandwidth used by high priority flows is upper bounded. Core routers share the bandwidth that is not attributed to high priority flows between the lower priority ones. The bandwidth available on a link to low priority flows is the total bandwidth minus the upper bound on the bandwidth that

can be used by high priority flows. An other solution would be that core nodes estimate the bandwidths actually used by high priority flows on their outgoing links and deduce from these values and the total link bandwidths the bandwidth available to low priority flows on each of their outgoing links. The packets from high priority flows may be put into a separate queue or into the same queue as the packets from lower priority flows, based on the label. If the high priority packets are put into a different queue, a priority scheduler selects packets from the high priority queue before serving packets from the other queue. This double queue solution may reduce the delay of high priority packets compared to the solution with one queue.

Additionally, we notice that, like FADE, CSFQ may also be used to ensure protection between flows inside the same aggregate class in the Differentiated Services architecture. To do so, a certain amount of bandwidth is allocated to each class. And, the core routers compute a fair share for each class. Packets are dropped according to the fair share relative to the class they belong to. Core routers need to maintain state for each class of service. The edge routers determine the class of service of each packet, compute the sending rate of the portion of the flow that belongs to the class of service of the packets and label the packets with this value.

2.4.3 Adaptations to MC-RED

Minimum guarantees can be provided by setting the first layer thickness to the reserved guarantee of the flow. And, if the remaining bandwidth has to be shared fairly between flows with the same weights, the layers above 0 all have the same thickness. When weights are associated to the flows, the thickness of the layers above 0 may be different for each flow depending on the flows' weights. The support of such guarantees does not require changes in the marking and buffer acceptance algorithm. Only the thickness of the layers have to be configured differently. When minimum guarantees are provided, admission control has to be performed to ensure that green packets are not dropped.

High priority flows will be supported by marking all of their packets green and policing these flows at the edge to make sure that they don't take all the resources away from lower priority flows. An other option is to use the same marker as for the other flows and to set the first layer thickness to the predicted sending rate of the high priority flow. These packets will be marked green and the excess packets will be marked with other colors as the ones from lower priority flows. The network has to ensure the forwarding of green packets. The other packets will be forwarded depending on the network load. Excess traffic of high priority flows could alternatively be marked red in case a lot of bandwidth is already attributed to their conforming packets (green packets).

2.4.4 Adaptations to CJVC

I don't see any possibilities for this mechanism to support minimum guaranteed flows with fair allocation of the remaining bandwidth at this time. This mechanism is too strict to permit easy adaptations.

To support high priority flows, they have to be the object of maximum bandwidth reservations. They cannot send at a higher rate than their reservation because of the scheduler used by this mechanism.

2.4.5 Adaptations to FADE

To support only fair allocation of the bandwidth, the minimum amount of bandwidth guaranteed to a flow has to be set to zero.

The use of the bandwidth allocated fairly to a flow but unused by this flow by other flows is not supported by FADE. A flow is always supposed to be bottlenecked at the link that has the smallest fair share on the flow's path. Even if a flow's sending rate is less than its fair share, it is supposed to use it's total fair share. The bandwidth allocated to this flow all along its path is equal to its fair share even if part of it could be used by other flows that are really bottlenecked on the link where the flow is supposed to be bottlenecked but is not. This follows from what is supposed to be to compute the fair share in interior nodes. In particular, it is a consequence from the shaping and policing that takes place at the edge of the network. One way to avoid this situation would be to eliminate the feedback mechanism used to carry links' fair shares estimations, and to stop shaping or policing low priority flows at the boundaries of the network. That way flows can send as much packets in the network as they can and when these packets arrive on a congested link some of them are dropped so that the flow respects its fair share on the link. Core routers drop packets when the rate of the excess traffic of a flow exceeds the outgoing link's fair share. When a flow is bottlenecked somewhere in the network, it doesn't get more bandwidth then its total fair share. In case it isn't bottlenecked, it allows other flows to use some of its total fair share because flows aren't policed at the edge supposing that all flows use their total fair share and not more than their total fair share. This leads to use FADE to estimate links' fair shares in a CSFQ domain supporting minimum guaranteed flows. The marking and the buffer acceptance algorithms are the ones from CSFQ where the fair share estimation performed in CSFQ is replaced by the estimation performed by the core routers in FADE. We have to notice that the fairness achieved does not quite correspond to the max-min definition of fairness anymore because flows that are bottlenecked somewhere in the network may use more than their max-min fair allocation on the links before their bottleneck, if they are not responsive to congestion. This last remark holds for all the other fair bandwidth allocation mechanisms where the dropping takes place in the core of the network.

2.5 Admission Control

2.5.1 Objectives

Admission control has as objective to control the load of the network. Before a new flow is established, it is checked if there are enough resources in the network to support this flow according to the resources requested by the flow. Admission control enables to ensure that the negotiated guarantees of accepted flows can be provided by a network domain.

Actions taken to attain its goal :

- Measurement of the network load or measurement of the reserved resources.
- Acceptance or rejection of new flows requests' based on the measurements.
- Dropping of admitted flows in case of exceptional events.

To achieve its goal, admission control has to perform some measuring of the actual resource reservations in the network and to accept or reject new flows based on this measurement and the global amount of resources available to flows. Exceptional event may also lead to resource exhaustion in part of the network. Such an exceptional event could be the failure of a link. The traffic that was using that link may be redistributed on other links which can become overloaded. Two options are available. Either the bandwidth guarantees aren't provided to the flows that use the congested link or some of these flows are dropped so that guarantees can still be provided to the remaining flows. In the latest, admission control has to provide the possibility of dropping admitted flows.

2.5.2 Centralized admission control

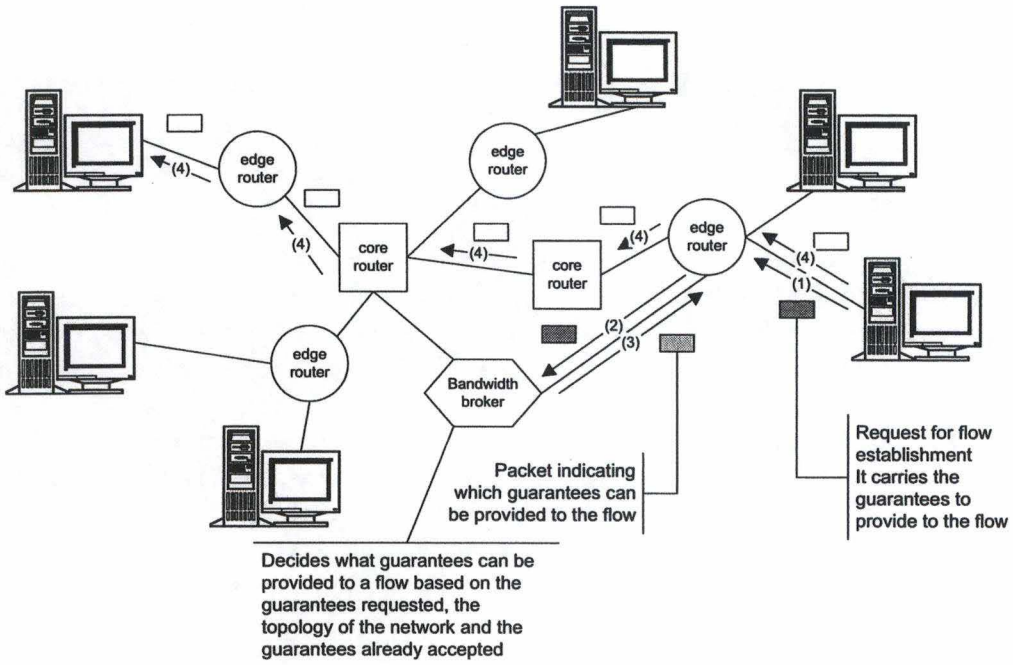

Figure 2.16: Centralized bandwidth broker

For a domain, there is one and only one device that performs admission control (figure 2.16). Each time a new flow is established, a request is sent to this device, called a bandwidth broker. The bandwidth broker then decides, based on the network topology and the reservations already accepted, if the new flow can be supported by the network. The bandwidth broker keeps the topology of the network, information about the guarantees to provide to each supported flow. A signaling protocol between edge routers and the bandwidth broker is also needed to allow communication between these network elements. When the requesting edge router obtains the decision of the bandwidth broker, it accepts the flow or reject it depending on the obtained decision.

2.5.3 Distributed admission control

There are two types of distributed admission control. Either there is more than one bandwidth broker in the domain and each one keep the topology of the domain and the level of resources that are the object of reservations (figure 2.17) or an edge-to-edge signalling protocol is used to check if there are enough free resources on the flow's path to support it (figure 2.18). By opposition to the centralized admission control, here the decision to accept a new flow may not be made only by one device. More than one device for each domain is empowered to take such decision.

Distributed admission control can be performed in two different ways.

- Each device used in the admission control keeps some information for each guaranteed flow supported[7].
- The devices implied in the admission control only memorize information on the aggregate reservation provided.

I will put my interest on mechanisms where the devices only keep information about the total amount of resources reserved.

Simple marking

The basic idea of Simple Marking (SM) (figure 2.19) is that core routers measure the *amount of traffic* passing through them. Edge devices send probe packets as they receive a request for a flow establishment. When core routers encounter near exhaustion of resources they mark the probe packets passing through them. Marked probe packets notify the edge devices of the lack of resources. Edge routers who receive a probe packet send the packet back to its source. The ingress edge router rejects the flow who originated the establishment request if the probe packet is marked, otherwise the establishment of the flow is accepted. To make the scheme robust against packet loss, the initiating edge device may maintain a timer associated with each probe packet. When the timer goes off, a new probe packet is generated and sent.

Unit-based reservations

The general behavior of Unit-based reservations (UBR) is shown by figure 2.20. Ingress edge routers send a probe packet at flow establishment. Core routers estimate the amount of resources reserved and mark probe packets in case there are not enough ressources to support the new flow. When egress edge routers get probe packets, they send them back to their sender. Based on the probe packet that came back, an ingress edge router determines whether the flow corresponding to the probe packet is supported or not. If the flow is not supported, it is rejected. In case there are enough resources to provide the guarantees required by the flow, the flow establishment is accepted.

In addition, ingress edge routers have to regularly mark normal packets to refresh the reservation associated to the packet's flow. Core routers use the number of unmarked probe

[7]ReSource ReserVation Protocol is an end-to-end admission control protocol that requires all routers on a flow's path to keep the level of guarantee associated to the flow.

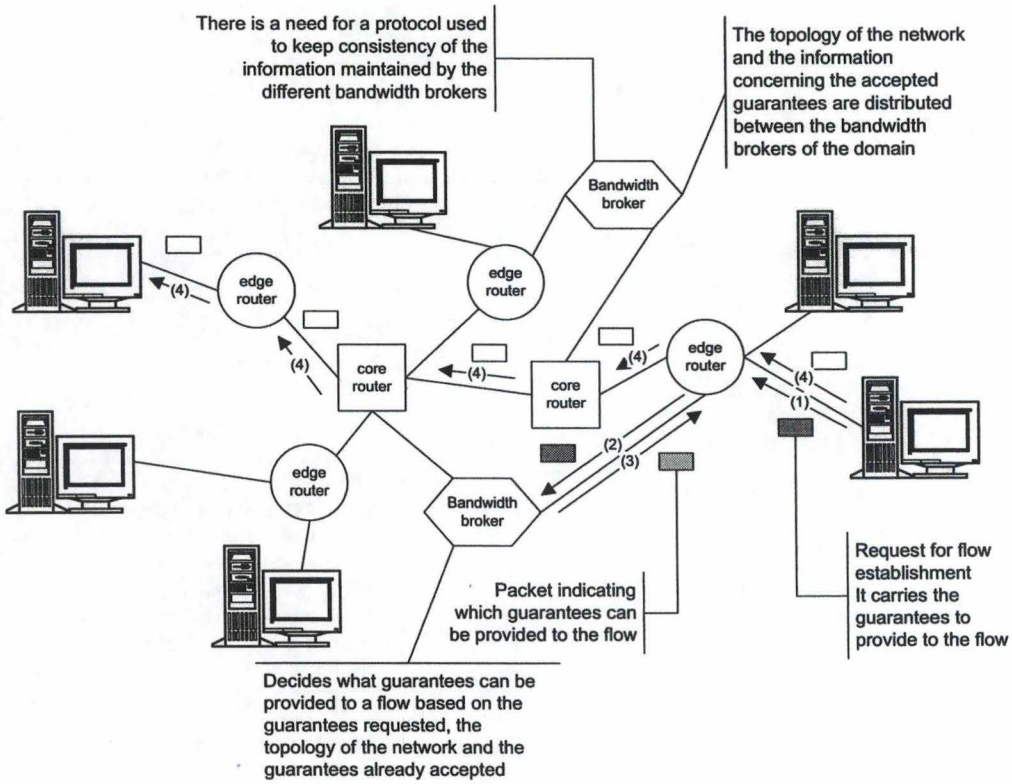

Figure 2.17: Distributed bandwidth brokers

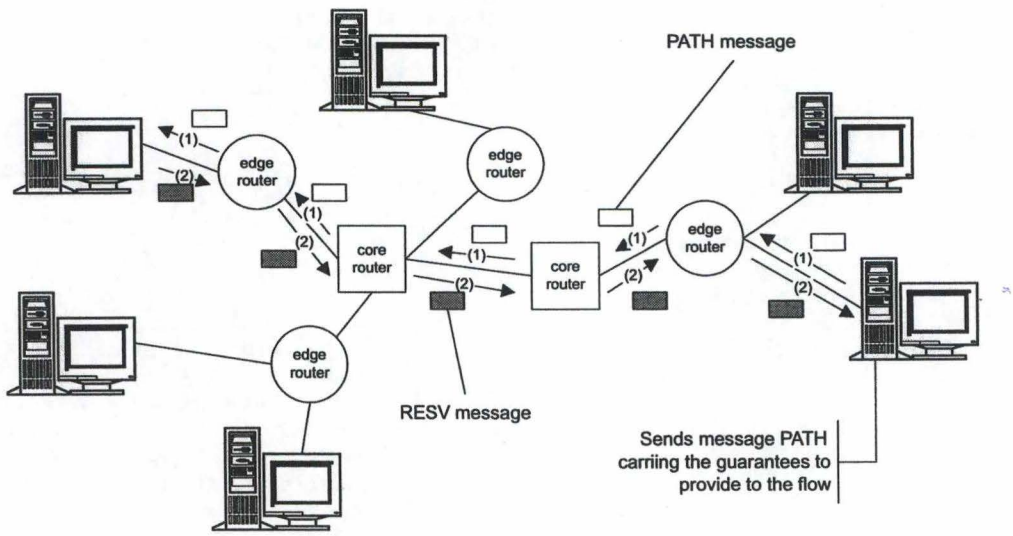

Figure 2.18: Distributed admission control (RSVP)

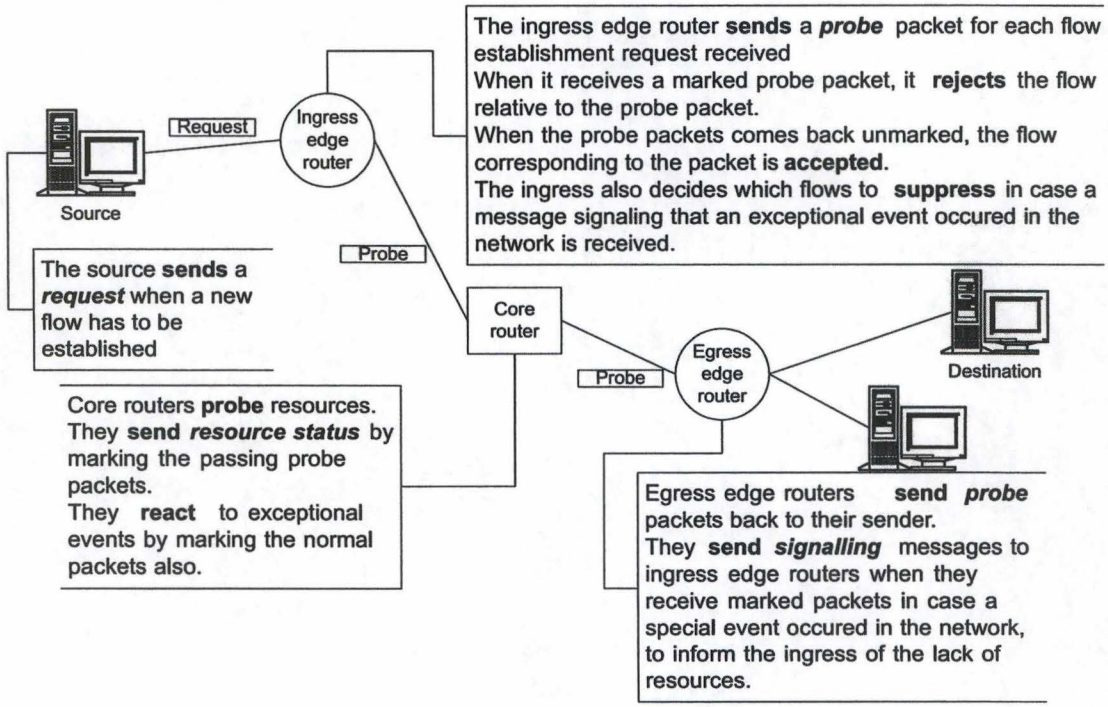

Figure 2.19: Simple marking

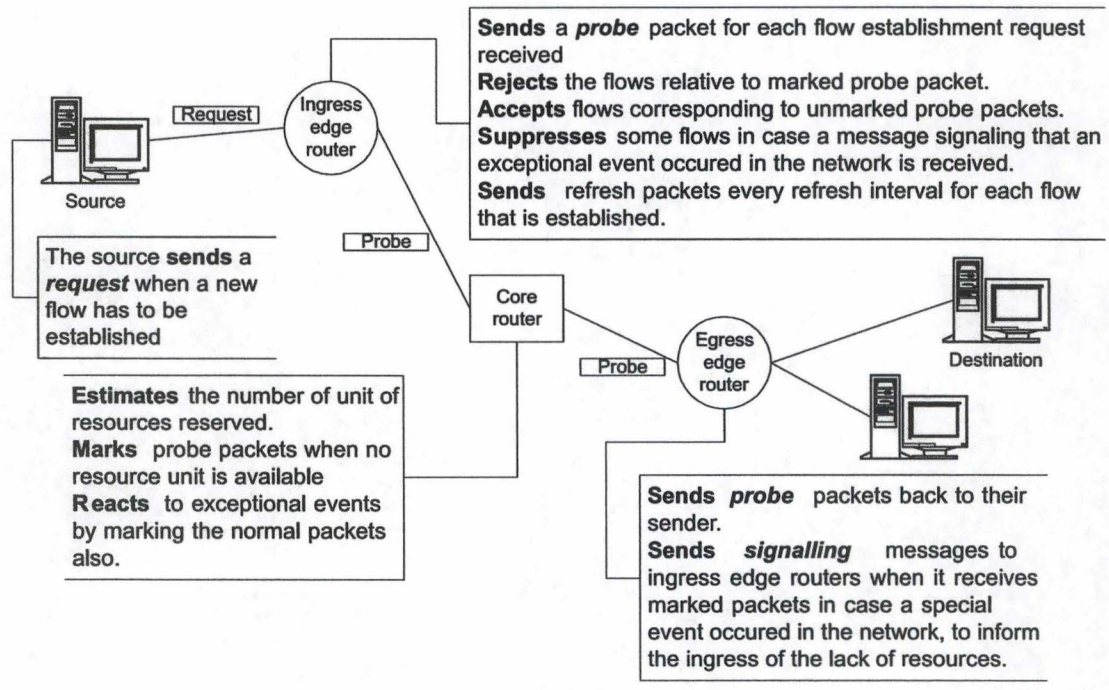

Figure 2.20: Unit-based reservations

packets and refresh packets received during a time interval called the refresh period to determine the amount of resources reserved for the next refresh period. The admission of new flows is made based on that estimation.

When congestion occurs, core routers mark normal packets to inform the egress edge nodes of the lack of available resources. An egress edge node who receives a marked packet sends a signalling message to the ingress edge node of the packet's path to inform it of the congestion. The ingress edge node then reacts to the special event by dropping some admitted flows.

The accuracy of the estimation of the number of allocated units can be increased by generating refresh packets evenly spread in time over a refresh period. This minimizes errors resulting from time alignment differences between routers and edge devices. If refresh packets all are sent in the same portion of the refresh interval, they will all arrive around the same time at one core router. Sometimes the packets are sent on a refresh interval and arrive on the next interval at a core router. The reservation will not be made at that router for the next refresh period (relative to the edge node) but for the interval after (which is not completely disjoint from the previous interval at the edge node). The reservation will be taken into account too late. This can influence the admission of new flows. It can also happen that some of the packets arrive on a refresh interval and that others arrive on the next time interval for reservations that have to hold at the same time.

Per-hop admission control

This admission control mechanism has some similitudes with the Unit-based reservations mechanism. Each core router maintains an upper bound R_{bound} of the aggregate reservations and decides of the marking of probe packets sent by ingress edge routers based on this upper bound. Each time a probe packet is sent unmarked, R_{bound} is increased by the amount of resource needed by the flow and carried by the probe packet. Figure 2.21 shows the operations performed by each network device in the Per-hop admission control mechanism (PH).

No assumption is being made on the reliability of the probe packets initiated by ingress edge routers and forwarded along the data path. If such a packet is lost, it is retransmitted by the ingress edge router after a certain time. It can be deduced that the reservation for a flow can be made twice at some routers. That is a reason why R_{bound} is an upper bound of the aggregate reservation. Sometimes, resources can be reserved at some routers before the probe packets meets a router with not enough resources available for the guarantee carried by the probe. The probe packet will be marked by the router and no more reservations will be made for that flow on the rest of the data path but the reservation already made cannot be undone.

When a flow is torn down, no termination message is sent by the ingress edge nodes. Core routers are not informed of the terminations of flows and R_{bound} is not decreased. R_{bound} stays an upper bound of the aggregate reservation.

If R_{bound} is never decreased resources will soon become unusable because for accepting a new flow, routers test if $R_{bound}+r$, where r is the reservation needed for the new flow and carried by the probe packet, is smaller or equal to the total amount of resources available. When R_{bound} is equal to the level of resources available no more flows can be accepted. Therefore, R_{bound} has to be regularly recalibrated to the amount of resource that is the object of reserva-

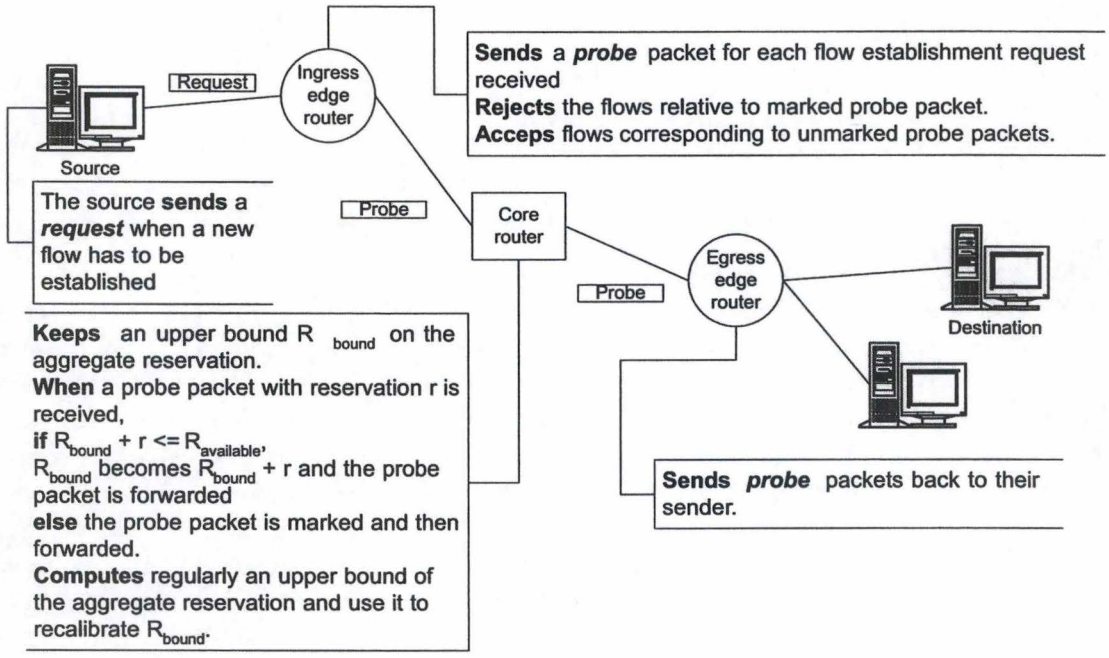

Figure 2.21: Per-hop admission control

tion. To recalibrate R_{bound}, an other upper bound of the aggregate reservation is computed. Then, R_{bound} is set to the minimum of R_{bound} and the calibration upper bound.

Evaluation

This section is dedicated to the comparison of the admission control mechanisms exposed previously. The characteristics of these mechanisms will be exposed as well.

The fundamental difference between Simple marking and Unit-based reservations is the presence of actual reservations in Unit-based reservations. Simple Marking cannot support hard guarantees because it accepts new flows based on the amount of traffic present when the probe packet is sent, independently of the quantity of resources needed by the new flows and the resources reserved by existing flows. More flows can be accepted than by using Unit-based reservations. With Unit-based reservation, if a flow is accepted, we can be sure that there is enough resources to provide its guarantees. These guarantees won't be jeopardized in case a special event occurs in the network. In the Simple marking scheme, there is no strict reservations of the bandwidth. The same guarantees as for Unit-based reservations can therefore not be provided. Simple marking doesn't support minimum bandwidth guarantees.

Unit-based reservations and Per-hop admission control both support hard guarantees. They maintain some state on the aggregate reservation and accept a new flow if there are enough resources to provide the guarantees assured to the already existing flows and to the one that requested its establishment.

When special events happen in the network, core routers do not react in case they perform Per-hop admission control. It is not the case with Simple marking and Unit-based reservations. Per-hop admission control can easily be adapted to also react to special events. Core routers

can measure the amount of traffic passing through them and if they are congested, they mark normal packets. Egress edge routers, when they receive a normal packet that is marked, send a signalling message to the ingress edge router of the packet's path who choses the flows to drop.

In Unit-based reservations, one probe packet is sent for the reservation of one unit of resources (idem for refresh packets). An alternative using only one packet for each flow for any quantity of resources to reserve should be defined. By comparison, in the Per-hop admission control, the amount of resources to be reserved for the flow are specified in the probe packet sent at flow establishment.

In Unit-based reservations, if a probe packet is forwarded unmarked by a core router, but is marked later downstream, that first core router will not be notified and will incorectly maintain the reservation. However, as the flow is rejected, no refresh packets will arrive, and the reservation will time out at the end of the refresh period and will be released. A core router implementing Per-hop admission control also reserves the resources required by a flow if they are available and the probe packet is unmarked. It can happen that not enough resources are available somewhere on the rest of the flow's path and that the probe packet will be marked. Then, the flow is rejected by the ingress node and some resources are already reserved for the flow. The recalibration upper bound is used to lower the aggregate reservation amount at routers by taking the traffic into account.

Table 2.4 summarizes what has been said in this section of evaluation of the different admission control mechanisms that do not maintain per-flow information on the reservations but instead are based on information about the aggregate reservation or the amount of traffic.

	SM	**UBR**	**PH**
Type of guarantees	soft	hard	hard
Reaction to special events	yes	yes	possible
Amount of units of resources reserved by a probe packet	none	one/any	any

Table 2.4: Evaluation of admission control mechanisms

2.6 Conclusion

From the presentation of various core stateless mechanisms, we deduce that many services may be provided without the need to maintain per-flow state in the core routers. With these mechanisms, the complexity is at the edge of the network, which is responsible of the marking, shaping or policing, while the core routers only have to perform aggregate scheduling and active queue management. Therefore, these stateless mechanisms are scalable. Some of these mechanisms allow fair sharing of the bandwidth, others provide maximum guarantees to the flows. Minimum throughput guarantees with fair sharing of the remaining bandwidth may also be provided by some core stateless mechanisms.

As a second step, we have made a comparison of the mechanisms in order to show their context of utilization as a function of the services that are required. By doing this simile,

we have underlined the proximity of the FADE and CSFQ fair share estimation algorithm. While at first sight FADE estimation of the fair share seemed to be more accurate than CSFQ fair share estimation, when looking at FADE estimation more closely we have seen that the hypothesis lying beneath this estimation may be too strong and impossible to meet in practice.

Admission control is required when guaranteed flows need to be supported by networks. It ensures that there are enough resources available to the accepted guaranteed flows. With the RSVP protocol, reservations are made for the guaranteed flows inside each node in the network by keeping some information concerning the guarantee. This solution provided by RSVP is not scalable. Thus, per-flow stateless mechanisms are needed for admission control as well. In the last section, we have seen that there are different per-flow stateless ways to perform admission control depending on the admission control architecture, the type of guarantees to support and the type of reactions required in case special events, like a breakdown of a link, occur.

Chapter 3

CSFQ study

Even though CSFQ seems to be simple from the pseudo code illustrated in figure 2.6. We tempt to show, in this chapter, that it is not obvious that CSFQ allows fair sharing of the bandwidth. At first, it is tried to provide more precision on different possibilities to perform some of the estimations as well as other clarifications concerning the algorithm of CSFQ. Then, a problem in the fair share estimation is underlined. And, finally, the impact on bandwidth allocation of parameters that have to be configured is analysed.

3.1 CSFQ clarifications

In this section, precisions are given concerning the estimation of the forwarding rate. Then, a way to estimate rates when simultaneous packet arrivals occur is proposed. At last, it is tried to determine a correct initial value for the fair share estimation.

3.1.1 Forwarding rate estimation

In CSFQ, two rate estimations are performed by core routers : the arrival rate and the forwarding rate[1] on each link are estimated. These estimations are made using the exponential averaging.

In the exponential averaging, the rate estimation $rate_{new}$ is based on the instantaneous rate and the previous rate estimation. The formula of the exponential averaging is as follows:

$$rate_{new} = (1 - e^{-\frac{T}{K}})\frac{L}{T} + e^{-\frac{T}{K}} rate_{old} \tag{3.1}$$

where T is the time elapsed between the arrival of the current packet and the previous, K is a constant and L is the length of the current packet. T and K determine the relative importance of the instantaneous rate, $\frac{L}{T}$, and the previous estimated rate, $rate_{old}$.

When T is big, the portion of the previous estimation taken into account in the averaging is smaller than when T is small. The bigger T gets, the more importance the instantaneous rate will have in the rate averaging. Taking K small means that the estimated average rate

[1]The forwarding rate is the aggregate rate of the packets sent to the queue that is set on the output link. This is a FIFO queue which may also tail drop packets in case of overflow.

will follow the instantaneous rate closely. On the other hand, a large K means that the rate estimation varies more smoothly in time. This means also that it reacts slowly to traffic bursts.

For the exponential averaging, the time interval T is not of fixed length. It is the time elapsed between the arrival of two successive packets, for an arrival rate estimation. For the forwarding rate estimation, there are two possibilities:

1. The time interval length can be obtained by the difference of the forwarding time of two packets *forwarded* successively.

2. The time interval is the interval between the *arrival* of two successive incoming packets.

With the first option, the rate is only estimated when a new packet is forwarded to the queue. When no packets are forwarded during a long period of time, the rate is not estimated ; it stays constant. But, in that case, the estimated forwarding rate should decrease to indicate that packets are no more forwarded and that the forwarding rate is decreasing. That is why the second solution is introduced. Each time a new packet arrives, the accepted and the forwarding rate are estimated. If the packet that triggers the estimations is forwarded then, like in the first case, where the forwarding rate is estimated on packet forwarding, we use formula 3.1. But, if the packet is dropped, the rate will also be estimated. Then, the time interval T is the time between the arrivals of the two last packets. And, the length of the packet is equal to zero because nothing is forwarded. Equation 3.1 becomes

$$rate_{new} = (1 - e^{-\frac{T}{K}})\frac{0}{T} + e^{-\frac{T}{K}} rate_{old} \tag{3.2}$$

$$rate_{new} = e^{-\frac{T}{K}} rate_{old} \tag{3.3}$$

which is smaller than the previous rate estimation $rate_{old}$ if T is strictly greater than zero. It follows that when packets arrive but are not forwarded the rate estimation is decreased by a small portion.

The second option is useful in the estimation of the fair share performed by CSFQ. In fact, when the output link of a router is congested[2], the fair share is estimated by the formula

$$FS_{new} = FS_{old}\frac{BW}{FR} \tag{3.4}$$

where FS is the fair share, BW is the bandwidth of the link and FR is the forwarding rate. As said above, when no packets are forwarded, i.e. they are dropped because the flow sends at a higher rate than the fair share of the outgoing link, the forwarding rate is decreased according to equation 3.3. When FR decreases, the fair share FS is bigger than if FR would have stayed constant like in the first option. It follows that the drop probability calculated by 2.2 decreases. Less packets will be dropped than previously if the arrival rate stays the same. By comparison, if the first solution is chosen, FR stays constant and, as a consequence, the portion $\frac{BW}{FR}$ is the same as previouly. The fair share FS increases or decreases by the same factor as for the previous forwarded packet. It seems that the second option fits best into CSFQ.

[2]This is determined by testing if the estimated arrival rate of the aggregation of flows is greater than the rate of the link.

If the output link is congested, the fair share cannot increase because an increase would trigger more packets to be forwarded and therefore the congestion would be even worst. On the opposite, when the forwarding rate is lower than the link bandwidth, the fair share should increase in order for the router to forward more packets and exploit a bigger portion of the outgoing bandwidth. This is exactly what happens with both solutions proposed. With the second option, when there is no congestion the fair share increases faster than the increase obtained by the first option. The forwarding rate estimation FR is smaller when the second option is used compared to when the first option is used. Additionally, because there is no congestion FR is smaller than BW. It follows that the ratio $\frac{BW}{FR}$, from equation 3.4 is greater with the second option than with the first option. Therefore, the fair share is increased from a bigger portion with the second option, when there is no congestion. In case of congestion, the fair share decreases slower with the second than with the first option. The estimation FR is smaller and the ratio $\frac{BW}{FR}$ is bigger than the one computed with the first option. If not enough packets are dropped by CSFQ, FR is higher than the link rate BW. Then, the fair share decreases more smoothly from one value to the next update. This is useful for TCP flows. They do not react too abruptly to the congestion if their packets are dropped progressively.

According to the previous argumentation, the second option has been implemented and is used in the simulations performed. The forwarding rate is estimated every time a packet *arrives* at the node. It is not estimated on packet forwarding.

3.1.2 Simultaneous packet arrivals

It can happen that packets arrive at the same time. This causes some problems for the estimations of the different rates. The estimations of the arrival rate and the forwarding rate are done using the exponential averaging. To perform this averaging, it is needed to divide by the time elapsed between two packet arrivals, T. When T is equal to zero, equation 3.1 is invalid. To obtain an estimation of the rate when $T = 0$, we will take

$$\begin{aligned} rate_{new} &= \lim_{T \to 0} \left((1 - e^{-\frac{T}{K}}) \frac{L}{T} + e^{-\frac{T}{K}} rate_{old} \right) \\ &= \frac{L}{K} + rate_{old} \quad . \end{aligned}$$

3.1.3 Fair share initialization

Choosing the right initial value for the fair share in each buffer acceptance module is a difficult task. We must first exclude zero in the set of potential values. The reason to reject this value is that, if the network is congested during the first window size of the simulation and stays overloaded, the fair share will keep zero as value. When the network is congested, the fair share is updated according to equation 3.4. If the fair share is zero and the network is congested, the new value for the fair share will also be zero. This will happen as long as the network stays overloaded. Until the aggregate arriving rate at the buffer acceptance module gets below the link rate, the fair share will be zero. That means that all packets arriving at the buffer acceptance module will be dropped. No packets will be transmitted on the output link as long as the fair share remains zero. More important, independently of the initial value, if at one moment the fair share decreases to zero and the network is congested (or starts to

be congested), the same problem happens. The fair share will not increase before the network starts to be uncongested. All packets arriving before that will be discarded.

At initialization, the value of the fair share should not be above the real fair share. The real fair share is the theoretical value the fair share should have, in conformance with the number of flows and their arriving rate (see definition in section 1.1.1). If the fair share is set too high, too many packets will be forwarded to the queue. The queue will be full after some time. The packets will then be dropped without distinction of the flows by the queue. When tail drops occur, the bandwidth is no more distributed in a fair way. A flow with a low sending rate can have packets dropped even if it does not have many packets in the queue. The queue may be full of packets from a small number of flows with a higher rate (that is limited by the fair share). The packets from these flows have more chances to find room in the buffer. More packets of the flows with an arriving rate above the fair share are forwarded to the queue and try to be stored in the buffer. These flows will receive the biggest part of bandwidth on the output link. The streams with a lower rate than the fair share will be disadvantaged. Packets from these flows may be discarded. Even worst, if these flows are TCP flows, when some of their packets are dropped, they reduce their sending window. It follows that TCP flows with a sending rate lower than the fair share may still decrease their sending rate. They will therefore not be allocated a bandwidth equal to the fair share of the path they follow in the network. Smaller packets also have more chances to fit in the queue. Usually, TCP flows have larger packets than the UDP flows. This is one more disadvantage regarding the TCP flows. As a summary, we can say that TCP flows will suffer from tail drops because

- TCP will react to drops and the sources will further decrease their sending window.
- TCP packets are larger than the UDP packets resulting in larger loss.

Figures 3.1 and 3.2 show the way the fair share estimation converges to the real fair share in a congested network. The slope[3] of these figures is derived from equation 2.4. These figures are only valid when the sending rate of the flows as well as the flows sharing the network do not vary in time. This condition ensures that the incoming traffic at each node in the network is stable once CSFQ has converged to the fair share on all links. The slopes that are drawn in figures 3.1 and 3.2 represent the forwarding rate as a function of the fair share at a network node given a link rate and a certain traffic. If the incoming traffic changes, new slopes have to be drawn. For example, if the fair share was overestimated it can become underestimated, if the number of flows suddenly decreases or if many flows do not send at their fair share anymore. The other way around, when the fair share is underestimated, a change in traffic can lead to an overestimation of the fair share. This happens, for example, when the amount of traffic increases by a certain amount making a link pass from uncongested to congested.

Figure 3.1 illustrates the convergence of the fair share to the real fair share when the initial value is an underestimation of the real fair share. The fair share estimation is set to its initial value (FS_0). Then, after a window size[4], the fair share is estimated. Its value FS_1 is obtained on the figure according to the method exposed in what follows. At each packet arrival during the window size period, the forwarding rate is estimated. So, at the end of the first interval of a window size length, we have a point (FS_0, FR_0) that belongs to the slope of FR as a

[3] An explanation of the shape of this slope is given in section 2.2.1.

[4] The window size is the time during which the fair share used for the drop probability computation is constant (section 2.4).

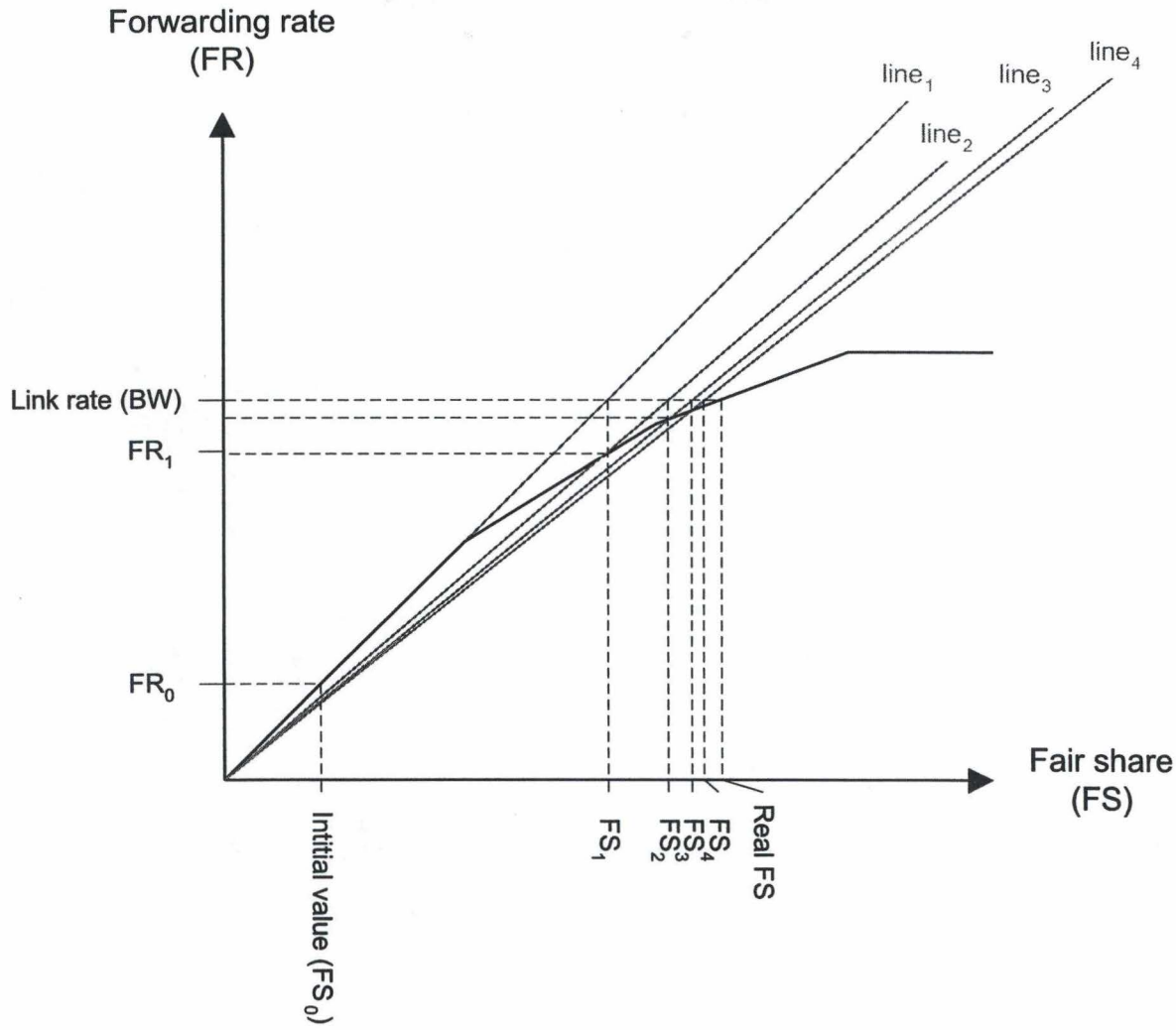

Figure 3.1: Underestimated fair share

function of FS. We draw a line ($line_1$) passing through the origin and (FS_0, FR_0). $line_1$ crosses the horizontal line with ordinate equal to the link bandwidth (BW). The point at the intersection from $line_1$ and this horizontal line gives the new fair share estimation (FS1). To obtain the next estimation, after a window size, a line ($line_2$) is drawn through the origin and (FS_1, FR_1), where (FS_1, FR_1) is on the slope. From the intersection of $line_2$ and the horizontal line at ordinate BW, a new fair share estimation FS_2 is obtained. This mechanism continues to obtain the list of the successive fair share estimations. The farther we go in the list, the more the estimated value is near the real fair share. The fair share valuation increases from one estimation to the next one. But, this increase decreases from one estimation to the following. The fair share never gets above the real fair share if the arriving rates of the flows do not change over the time.

In figure 3.2, the initial value of the fair share is above the real fair share. The same princi-

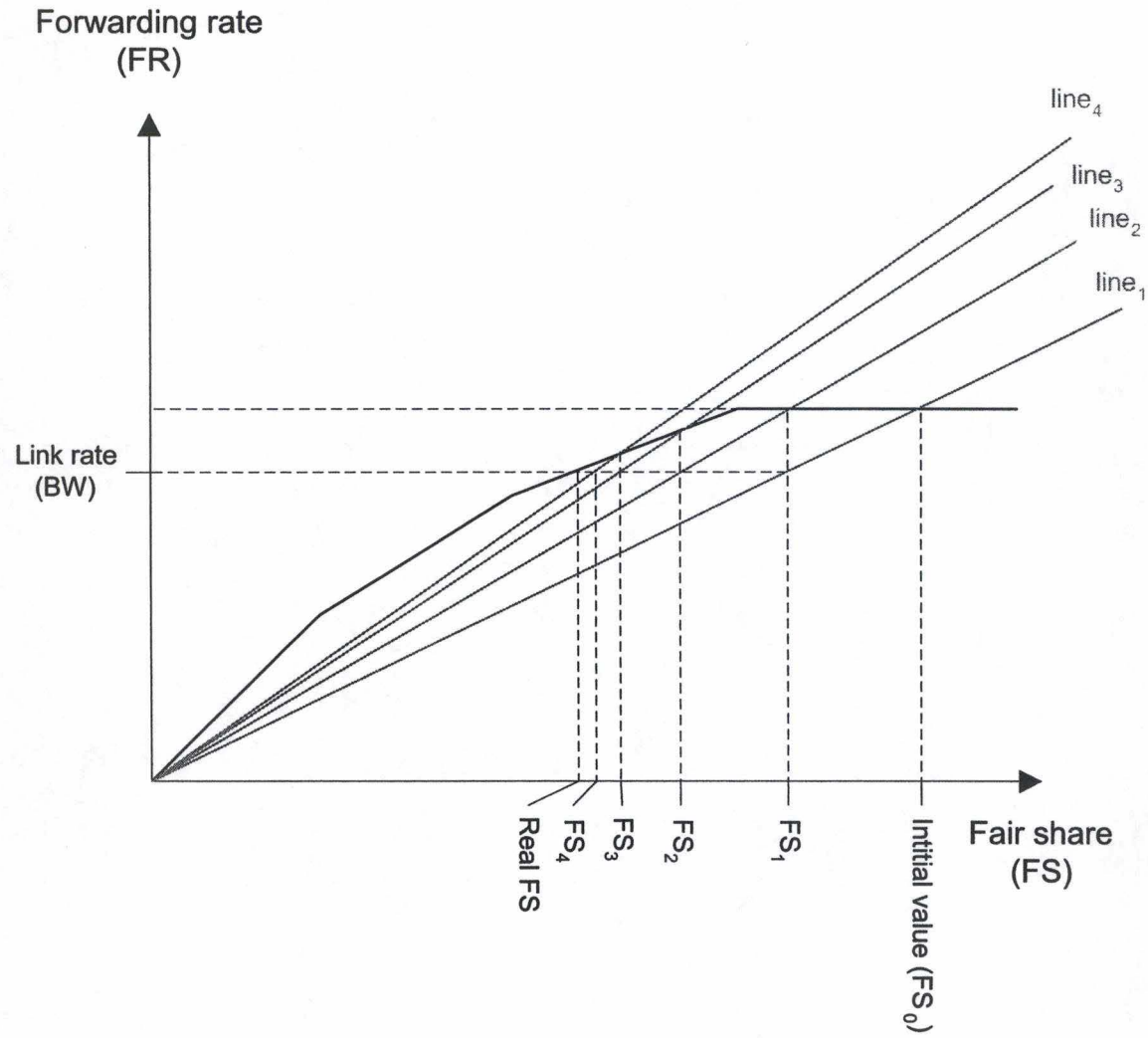

Figure 3.2: Overestimated fair share

ple as the one exposed in the previous paragraph is applied to find the successive estimations of the fair share. From the initial fair share FS_0 and the estimated forwarding rate, we draw a line that crosses the origin. From this line and the link rate, we obtain a new value for the fair share that is smaller than FS_0, and so on. It can be seen that the fair share decreases at each step but the decrease is smaller each time. If the arriving rate of the flows stay constant, the fair share will always be overestimated.

If the initial value of the fair share is below the real fair share (figure 3.1), the estimation will converge to the right value but will always be below it. On the contrary, if the fair share is set to a higher value than the real fair share (figure 3.2), the fair share will also converge to the theoretical value. This time, the fair share will always be above the real value. It means that the queue will build up and tail drops will occur if the network stays congested during a too long period. Because tail drops should be avoided to ensure fair allocation of the bandwidth,

overestimations of the fair share should be prevented. When these overestimations occur, they should be small and brief.

We can conclude that choosing an appropriate initial value for the fair share is not a simple problem if we want to obtain a quick convergence, avoid big underestimations and prevent tail drops.

3.2 Uncongested network problem

It has to be noticed that in the algorithm of CSFQ (figure 2.6) as exposed in [SSZ98a], when there is no congestion, the fair share is set to the maximum of the packets' labels passing through the network node. To obtain this maximum, the fair share is regularly assigned to a temporary variable (temp_FS) whose value is reset to zero after the fair share is updated. This assignation is performed every time the link stays uncongested during a time period determined by the window size. But, during a window size period, sometimes not all flows have packets arriving at a node. Consequently, the fair share might be smaller than the label of certain packets. As a result, the drop probability of certain packets could be higher than zero. Some packets may be dropped even if the link on which they have to be transmitted is not congested. Another problem with the estimation of the fair share as the maximum of the labels, in uncongested mode, is that the flows cannot increase their sending rate. For example, if the maximum arrival rate among the flows at a link increases, and the new arrival rate of the flow is not yet incorporated in the fair share estimation, some packets of the flow are dropped. Again, we observe dropping of packets in an uncongested period. But, in uncongested periods, TCP always tries to increase its sending rate. If packets of the TCP flows are dropped, their sending rate will decrease because dropped packets are interpreted as congested network by TCP. Therefore, this estimation of the fair share done by CSFQ for uncongested links can be seen as TCP-unfriendly. Additionally, forcing the flows to decrease their sending rate leads to underutilization of the resources.

We can also imagine the case where a network node does not get any packets to send on a particular output link. Then, the estimation of the arrival rate should be around 0 bps and the link is considered as uncongested. Because no packets have to be transmitted, and temp_FS is periodically reset to 0, the fair share is estimated to 0 bps. And, when packets will arrive for this link, they will all be dropped. If the link stays uncongested during a window size, the fair share estimation will increase to the maximum of the labels that passed during the last window size period but if the link becomes congested before this update, the fair share estimation will keep zero as value according to equation 3.4.

3.3 Parameters impact

Different parameters need to be configured to determine the behavior of CSFQ. The objective of this section is to determine how these parameters impact on the bandwidth allocation.

3.3.1 Tail drops impact

In [SSZ98a], Stoica proposes two minor amendments to CSFQ. One of them is to decrease the fair share by a small fixed percentage every time a packet is tail dropped. During a window

size period, the drop probability of the arriving packets from whatever flow is computed based on this fair share. If another tail drop occurs, then, the fair share is decreased again, independently of the window. The drop probability of a packet depends on the fair share of the link and the label carried by the packet.

But, there is a question about the percentage by which the fair share has to be decreased in case a tail drop occurs. Also the number of successive decreases allowed has to be determined. When tail drops occur, if the fair share is largely overestimated, the decrease of the fair share may not be enough to avoid future tail drops. An unfair allocation of bandwidth can result from a bad configuration of the decrease percentage and the number of successive decreases that can be performed. Because overestimations of the fair share may happen, when CSFQ has converged, due to variations in traffic, if the load of a network changes very dynamically, it is impossible to choose these two configuration parameters correctly.

3.3.2 Window size impact

The value chosen for the window size has some impact on the way the bandwidth is distributed among the existing flows. During at least a window size, the fair share stays constant. That means that if the load of the traffic to be sent on the output link changes, it will take some time for the fair share to be adapted in consequence. The smaller the window size, the quicker the fair share will be adjusted. But the longer the window size, the more stable the router will be because the forwarding rate estimation is more accurate towards the fair share estimation.

When the output link is not congested, the fair share is computed to be the maximum of the labels from the packets passing during a window size (in the original version of CSFQ). If the widow size is small, the fair share may be smaller than it should be. Not all flows may have packets arriving at the router. The fair share will not take into account the rate of the flows for which no packet has been received during the last window size. This is a problem because TCP sends bursts of packets. TCP sources send packets one after the other. Then, they do not send anything for some time. After a moment, they start sending a burst of packets again, and so on. It may often happen that no packets from a flow are received during a window size if the value chosen for the window is too small.

3.3.3 Threshold impact

The threshold is the size of the queue under which the output link is considered to stay uncongested. When the queue occupancy is below the threshold, the fair share is estimated in the same way as if the output link stayed uncongested even if the aggregate arrival rate becomes greater then the link rate. Once the queue size exceeds the threshold, if the aggregate arrival rate is above the link rate, the output link is considered congested. Then, the fair share is estimated by equation 3.4. Otherwise, the next value of the fair share will be the maximum of the labels of the packets that have to be sent on the output link since the beginning of the window. It can be noticed that if the threshold is set to zero, then it is as if no threshold was used. The output link is considered to be congested when the aggregate arrival rate is above the rate of the output link. It is not congested when the total arrival rate is below the link rate.

From the last assertion, it can be deduced that if the output link is near congestion, the mechanism will be unstable. A link is near congestion when the arrival rate on the link is around the link rate. Sometimes, the arrival rate will become greater than the link rate, then the link will be considered as congested. The time will be restarted. The estimation method of the fair share will change. And, the fair share will be updated only if the link stays congested during a window size. If the link becomes uncongested before the end of the window size, the time is restarted. The fair share should be updated after a window size if the load of the link doesn't exceed the link rate again. It can be observed that the fair share may not be revised for a long time in some situations. With a threshold (greater than zero), the time will be restarted less often (figure 3.3). The output link is considered uncongested for a longer period. The fair share is updated more often. It follows that the bandwidth is distributed according to a more accurate value of the fair share.

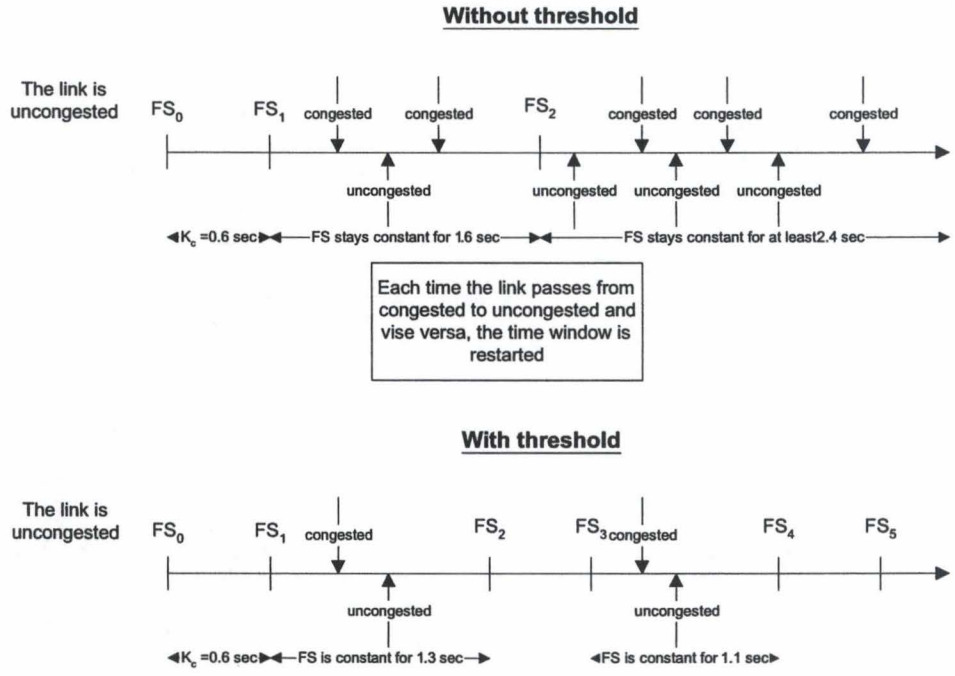

Figure 3.3: Frequency of FS updates

If the output link is not congested and the amount of arriving traffic suddenly increases, the congestion will not be taken into account until the queue occupancy goes beyond the threshold. Between the time when the arrival rate exceeds the link rate and the threshold is outrun, the fair share may still increase. The fair share is yet computed as the maximum of the packets' labels and the previous fair share. But, because of the new congestion, the fair share should not increase. It may stay the same but it should probably decrease. During the interval between the change in traffic and the congestion is noticed, too many packets are accepted. Some of these packets may be dropped by the queue. Then, bandwidth is not distributed in a fair manner anymore. Even if the queue doesn't have time to fill up, unfairness between the flows can occur. When the congestion is noticed, the fair share may decrease a lot. Many packets are suddenly dropped from all bottlenecked flows. This abrupt dropping

is not beneficial to the TCP flows. When they will detect the congestion, they will decrease their sending rate. But, the UDP flows will not perceive and react to the congestion. If the sending rate of the TCP flows decreases too much, it will take some time before the sending rate equals the fair share again. During that time the bandwidth allocation is unfair.

An important problem with the threshold is to find the right value for it. The value chosen should be such that we can benefit from the advantage presented two paragraphs above. That means that the threshold has to be set such that the fair share can be estimated regularly in situations were the output link is near congestion. It has to be decided until when a link can be considered as not congested. The threshold may not be too high because if the load of the arriving traffic occurs a significant increase, the fair share estimation should change to take the congestion into account. The implementation of a threshold is the second amendment that Stoica suggests in [SSZ98a].

3.4 Conclusion

The pseudo code presented in figure 2.6 is a simplification of the algorithm of CSFQ. It has been seen that there are different possibilities to implement the estimation of the forwarding rate. And, the solution chosen corresponds to the second proposal. Some special cases like the way division by zero is avoided when packets arrive at the same time are not treated in the pseudo code but they have to be handled in an implementation. The value used for the fair share initialization is also not given. We have underlined that such value has to be determined carefully and we have eliminated zero right away. For more information on the implementation we refer to appendix C.

As a second step we tried to evaluate CSFQ from a theoretical point of view. First, we have demonstrated that problems in fair bandwidth allocation can occur in uncongested networks. Then, the impact of tail drops on the allocation of bandwidth is mentionned. It is showed that the fair share estimation depends on the window size and the constant used in the rate estimations. The fair share is also influenced by the threshold that can be implemented to obtain a more stable behavior but sometimes postpones estimations when an earlier estimation would have been wiser.

Chapter 4

CSFQ ameliorations

Some weaknesses of the original algorithm have been discovered in the previous chapter. We have seen that the degree of fairness depends on the setting of some parameters and we also found out that the fair share estimation may not be adequate when there is no congestion. To solve this problem, an amelioration to CSFQ is proposed. This solution is implemented in a new mechanism called Modified Core Stateless Fair Queueing (MCSFQ).

The original CSFQ mechanism does not support flows with minimum throughput guarantees. Our second objective in this chapter is to propose a way to enhance CSFQ by enabling the provision of minimum throughput guarantees in addition to the fair sharing of the bandwidth that is not the object of guarantees.

4.1 Fair bandwidth sharing

To avoid the dropping of packets when there is no congestion (section 3.2), it is chosen to initialize the temporary variable temp_fs to the initial value of the fair share. The temporary variable will only be reset when the link passes from congested to not congested and it will be given the actual value of the fair share. When the link stays uncongested, the temporary variable is equal to the maximum of the packets' labels and the value of the fair share at initialization, if the link was never congested. If the link was once congested but is not actually a bottleneck, the temporary variable is the maximum of the packets' labels and the fair share of the link when it was last congested.

Figure 4.1, shows the pseudo code of fair share estimation in the Modified version of Core Stateless Fair Queueing. To obtain MCSFQ, changes are only made in the fair share estimation. More precisely, the fair share estimation is only altered in the non congested case. By comparison to the fair share estimation algorithm of CSFQ, the variable `temp_FS` is not set to zero when the link starts to be uncongested. Instead, it is set to the actual estimation of the fair share. Additionally, after a window size period of uncongestion, `temp_FS` is not reset; it keeps its current value.

The solution proposed is interesting because, when the link becomes uncongested, it can be considered that the right fair share has been found. The new fair share should not be lower than the fair share used when there was congestion because the arrival rate decreased, so, less packets have to be dropped. It is logical that the fair share should be the maximum of the

```
estimate_FS(p,dropped)
    estimate_rate(AR,p);  /* estimate arrival rate */
    if (dropped == FALSE)
        estimate_rate(FR,p);
    if (AR≥BW)
        if (congested == FALSE)
            congested = TRUE;
            start_time = current_time;
        else
            if (current_time > start_time + K_c);
                FS = FS * BW/FR;
                start_time = current_time;
    else  /* AR < BW */
        if (congested == TRUE)
            congested = FALSE;
            start_time = current_time;
            temp_FS = FS;  /* Used to compute new FS */
        else
            if (current_time < start_time + K_c)
                temp_FS = max(temp_FS,p.label);
            else
                FS = temp_FS;
                start_time = current_time;
    return FS;
```

Figure 4.1: MCSFQ fair share estimation pseudo-code

fair share at the last congestion update and the packet's labels that passed. When the link stays uncongested, the temporary variable is not reset because all packets can be forwarded. When reseting it, like in the proposal made in [SSZ98b], if the labels of the packets that pass during a period of a window size length are strictly lower than the maximum of the packets' labels passed during the previous period, the fair share decreases even if there is no congestion. There is a chance that the flow with the high sending rate is still active and if its packets arrive some of them might be dropped because their label is above the fair share. But, packets should not be dropped when there is no congestion because either each flow has already attain its fair share or the flows are not using all network resources. Here, there is no need to limit the use of resources of certain flows. Some bandwidth is still available for new flows or flows increasing their need for bandwidth.

It can be noticed that, when the link becomes congested for a window size period, the fair share will be updated according to

$$FS_{new} = FS_{old}\frac{BW}{FR} \quad .$$

If the fair share is too high when the link becomes congested, the forwarding rate measured during the first window period will be higher than the link rate and therefore, the next estimation of the fair share will be smaller.

In the next chapter, some results from simulations using CSFQ and Modified CSFQ (MCSFQ) are exposed. By CSFQ, we refer to the mechanism presented in [SSZ98a]. The estimation of the forwarding rate is performed on packet arrivals. The fair share is initialized to zero when this value is not a problem otherwise 1 bps is chosen. And, the temporary variable `temp_FS` is reset to zero every time the output link is not congested during a window size. In MCSFQ, the estimation of the forwarding rate is also carried out on packet arrivals. The fair share is initialized to 1 bps usually. But, most important, `temp_FS` is not reset to zero after a window size of non congestion. When the link starts to be congested as well as when the link is uncongested since a window size, `temp_FS` is set to the actual estimation of fair share, not to zero like it is done in CSFQ.

4.2 Minimum guarantees support

In this section, a way to enable CSFQ to provide minimum throughput guarantees to some flows is exposed. It is showed that a small change in the labelling performed by the edge routers is enough to achieve the support of minimum throughput guarantees. Some modifications may be done to the buffer acceptance module also. These alterations are not mandatory to obtain satisfying results.

To be able to support minimum guaranteed flows, an admission control mechanism should be implemented. The objective of admission control is to ensure that there are enough ressources to provide the guarantees associated to the flows accepted in the network. In the simulations that have been run to test if an adaptation of CSFQ has a good behavior in supporting minimum guarantees with fair sharing of the remaining bandwidth, no admission control mechanism is used. The amount of bandwidth reserved for each flow is static. It is made sure in advance, that the amount of the aggregated bandwidth reserved on a link does not exceed the link rate.

4.2.1 General principle

To enable a mechanism like CSFQ to support minimum guarantees, some changes have to be made. First of all, while labelling packets, edge routers have to be able to determine which packets are part of the guarantee associated to the packet's flow and which packets will be treated as best-effort (section 4.2.2). Bandwidth that is not the object of any reservation has to be shared fairly between the flows sending packets in excess of their traffic contract. Therefore, CSFQ will only apply to excess packets. Edge routers will for that purpose mark all guaranteed packets with a label of zero. This indicates that the packet doesn't use any portion of the bandwidth that has to be allocated fairly. But, in excess packets will be marked with a label equal to the excess rate of the flow [NSY00]. Such principle is also used in [VMSB00] for the provision of minimum throughput guarantees by Stateless Prioritized Fair Queueing (SPFQ).

When a packet arrives at a router, it is checked if the label of the packet is higher than the fair share of the output link on which it has to be transmitted. When the packet is guaranteed, the fair share will never be smaller than the label because the label has a value of zero. But, this might happen for packets sent in excess of the guarantee. Because the drop probability

of a packet is determined by the equation

$$P = \max(0, 1 - \frac{FS}{label}) \quad , \tag{4.1}$$

it can be deduced that, if $label > FS$ than $P > 0$. And, when $label \leq FS$, we have $\frac{FS}{label} \geq 1$. It follows that $P = 0$.

In case the label has a value of zero, equation 4.1 is invalid. Because the fair share should never be zero[1], or, even worst, negative, we are in the case where the label is smaller than the fair share so, the packet should not be dropped. If guaranteed packets are marked with a label of zero, they should not be dropped by CSFQ. When a packet with label zero is received the drop probability is set equal to zero and the packet is not discarded, except if the queue is full and tail drops occur.

```
On receiving packet p
    if (edge router)
        i = classify(p);
        AR_i = estimate_rate(AR_i,p);
        /* Determine of the packet is in or out of the guarantees */
        P_out = max(0,1 - guar_rate_i / AR_i);
        /* Mark the packet */
        if (P_out <= unifrand(0,1))
            p.label = 0;
        else
            p.label = estimate_rate(ER_i,p);
    /* Edge and core routers */
    if (p.label = 0)
        /* Packet in the guarantee */
        P_drop = 0;
    else
        /* Packet out of the guarantee */
        P_drop = max(0,1-FS/p.label);
    if (P_drop > unifrand(0,1))
        FS = estimate_FS(p,1);
        drop(p);
    else
         if (P_drop > 0)
             p.label = FS;  /* relabel p*/
         FS = estimate_FS(p,0);
         enqueue(p);
```

Figure 4.2: support of minimum guarantees : pseudo-code

To provide flows with minimum throughput guarantees no changes are required in the fair share estimation. The only changes required are shown in figure 4.2. In the pseudo code given,

[1]The fair share should never be zero because this means than no excess packets will be transmitted. There is no bandwidth available for such packets. In fact, it could happen if the sum of all guarantees was equal to the link rate but, normally, admission control will reject new flows when 90% of the bandwidth is reserved.

the marking is performed based on a probabilistic determination of packets in and out of the guarantees. In the following subsection, a deterministic method is also proposed. Moreover, to obtain a quicker convergence of the fair share estimation, we suggest in subsection 4.2.3 to estimate the amount of aggregate guarantee. This modification requires modifications the the fair share estimation function. We suggest to refer to the implementation in appendix C for informations concerning the implementation of this estimation as well as the deterministic marking.

4.2.2 Probabilistic versus deterministic marking

In our implementation, two different ways to determine the packets that have to be marked out of the traffic contract or inside the contract are available. In the case treated, the contract corresponds to the amount of minimum throughput that is guaranteed to a flow. Packets can be marked in a probabilistic or in a deterministic way.

Probabilistic method

Depending on the arrival rate of the flow and the guarantee associated to the flow, a probability is determined

$$Prob_{out} = 1 - \frac{\text{Guarantee}}{\text{Arrival rate}} \quad .$$

Packets from the flow will be marked as not belonging to the minimum guaranteed portion of the flow with that probability. And, packets will be marked as belonging to the guarantee with a probability of $1 - Prob_{out}$.

With this marking method, a source cannot know in advance which packets will not be guaranteed and the ones that will be guaranteed. It may be important for certain kind of traffic that the source be able to make such distinction. A source could then send important information in guaranteed packets and accessory information in best-effort packets. The source is then assured that the information that is crucial to its functioning will be received by the destination, because of the guarantee. Multimedia applications could take advantage from this depending on the encoding method used.

An advantage of this method is the possibility for successive packets not to be marked in excess even if the flow sends at a higher rate than its guarantee. This is interesting because excess packets will be dropped before guaranteed packets in the network. When packets of a burst are not all marked in excess, some of the packets of the burst are assured to arrive at the destination. The burst will not be totally dropped because guaranteed packets should not be dropped in the network. This is good for the TCP flows. They detect the congestion faster if some of their packets arrive at the destination. Sometimes, they may be able to do fast retransmit and recover quickly from the lost of packets.

To compute the arrival rate of a flow with the exponential averaging, a window size has to be chosen. Because such window is also used by core routers, the same value can be used by the marking performed at the edges. No configuration difficulty is added to CSFQ when packets are marked probabilistically.

Deterministic method

A token bucket can be used to determine if a packet will be guaranteed or whether it is out of the traffic contract (section 2.3.1). There is one token bucket for each flow. The filling rate of the token bucket is the amount of bandwidth reserved for the flow. The size of the token bucket determines the size of the bursts of traffic that are considered as belonging to the minimum throughput guarantee.

When a token bucket is used by edge routers to determine the guaranteed packets, the source can also determine the guaranteed packets by using a token bucket configured in the same way as the one from the edge router. This works only when a flow is composed of packets from one and only one source. When the guarantee concerns an aggregation of flows, a source may not be able to determine if its packets are respecting the contract since it may also depend on other sources.

In the simulations, where some flows are guaranteed, presented in chapter 6, the marking is done using the probabilistic algorithm even though the deterministic method has also been implemented. The reason for this is that a good size for the token buckets has to be chosen such that burst of packets can belong to the guarantee if they are not too long. But, the rate of the guaranteed packets should not exceed the guaranteed throughput over a time interval of a certain length. It is difficult to find an adequate value for the size of the token bucket. A solution for this problem could be a rate adaptive shaper as described in [BDC00].

4.2.3 Estimation of the aggregate guarantee

It has been said previously that it is possible to support minimum throughput guarantees by only making changes in the arrival rate estimator module. In this section it will be shown that by bringing small changes to the buffer acceptance module, the bandwidth that is not the object of any reservations on a link can be better shared between the flows.

When we try to plot on a graph the forwarding rate as a function of the fair share for a link, we obtain the graph in figure 4.3. The function doesn't cross the X axis in a positive abscissa. The ordinate axis is crossed at a value equal to the amount of the aggregate bandwidth reserved, if the flows send at least at their guaranteed throughputs. Otherwise, the Y axis is crossed at the sum of the sending rates of the flows sending below their guarantee and the guarantees of the other flows. When the fair share is zero, the forwarding rate on the link is equal to the aggregate rate of the guaranteed packets that arrive. No packets in excess are transmitted. The forwarding rate is greater or equal to zero. Then, in the positive abscissas, the shape of the function is the same than when no guarantees are provided. The function is only translated upwards along the ordinate axis.

When there is no estimation of the amount of reserved traffic transmitted on the link, the estimation of the fair share converges less quickly to the right value than when the guaranteed traffic rate is estimated. This comparison can be done from figure 4.3. In this figure, FS is used for fair share and FR denotes the forwarding rate. A slope is drawn from the point with coordinate (Current FS estimation, Current FR estimation) and the origin. Then, the new fair share without estimation of the guaranteed rate is the point on the slope with ordinate equal to the link rate (BW). The value obtained for the fair share without the estimation of the aggregate guarantee is smaller than the new value obtained with estimation of the amount

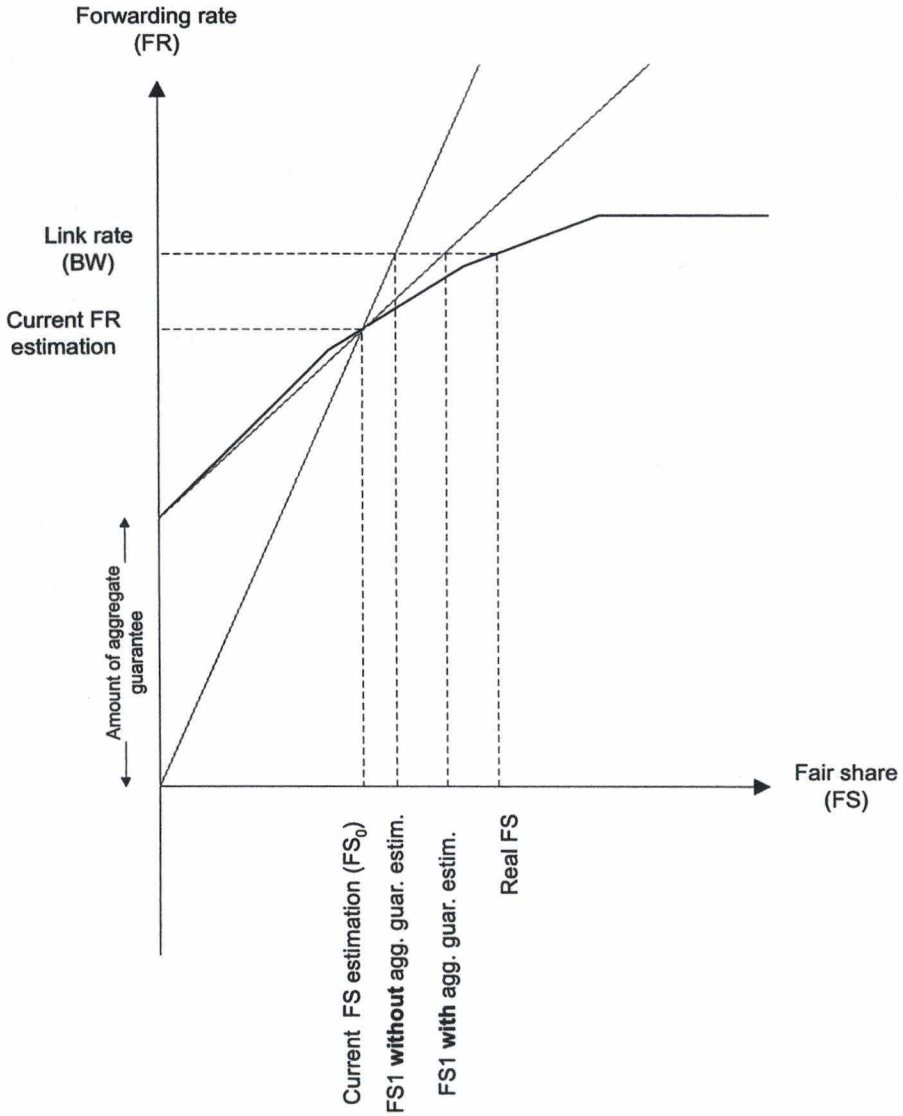

Figure 4.3: Fair share estimation without/with estimation of the aggregate guarantee

of reserved bandwidth. When the fair share is overestimated, an analogous conclusion can be drawn : the estimation converges less quickly to the fair share when the quantity of guaranteed traffic is not estimated.

4.3 Conclusion

In this chapter, a solution has been proposed to the problem of CSFQ that occurs when links are uncongested. This solution is implemented in MCSFQ. The pseudo code of the ameliorated fair share estimation is given with the same simplifications concerning the estimation of the forwarding rate, the treatment of special cases, ... as in the original pseudo code (figure

2.6).

For both CSFQ and MCSFQ, it is possible to provide minimum throughput guarantees. This enhancement is done to the labelling process also called the arrival rate estimator (appendix B). For an optimisation, the fair share estimation function may also incur small changes by estimating the amount of aggregate guarantees. In the simulations that are exposed in chapter 6, both CSFQ and MCSFQ are tested with and without the presence of guaranteed flows.

Chapter 5

Simulation scenarios

This chapter introduces the scenarios that are used for the simulations of CSFQ. In each of these scenarios, there is one flow per source. These flows may be an aggregate of TCP or UDP flows. The data packets are sent from source to destination. The data packets all go in the same direction, from the left to the right of the networks. And, the acknowledgments follow the reverse path. Two way traffic, i.e. traffic with a mix of acknowledgments and data packets, are not considered. Because the acknowledgements are much smaller and less numerous than the data packets, there is never congestion in the reverse path. In the simulations, CSFQ is not used on the path of the acknowledgments. There is no need for CSFQ on this way of transmission because the links crossed by the acknowledgements are not congested. Only the throughput of the data flows is influenced by CSFQ. The acknowledgements are transmitted in a best-effort network. The bandwidths that are indicated on the figures of this chapter are the bandwidths for one way of transmission, the data path. For the other direction, the links have an almost infinite bandwidth. In the following sections, the routers and the links will be numbered from the left to the right.

Three scenarios are used for the simulations : the single bottleneck, the multiple bottleneck and the Generic Fairness Configuration (GFC) scenario. Each one of these scenarios is used with different configurations of CSFQ. The results from these various configurations are compared in chapter 6.

5.1 Single bottleneck scenario

In the single bottleneck scenario (figure 5.1), there are four sources, an ingress edge router, a core router and four destinations. A flow is associated to each source. All packets generated by one source belong to the same flow. There are four sources, so there are four flows in total. The first flow goes from source 1 to destination 1, the second goes from source 2 to destination 2, and so on. All four flows cross the two routers of the network. They first pass through the edge router where they are multiplexed on a single link. Then, the flows cross the core router. Finally they are distributed on the links leading to their destinations.

In this scenario, the links from the users to the routers have a rate of 150 Mbps with a fixed propagation delay of 2.5 ms. The TCP flows are shaped for a link of 100 Mbps. The link joining the two routers is of 3.75 Mbps with a fixed propagation delay of 10 ms in each direction.

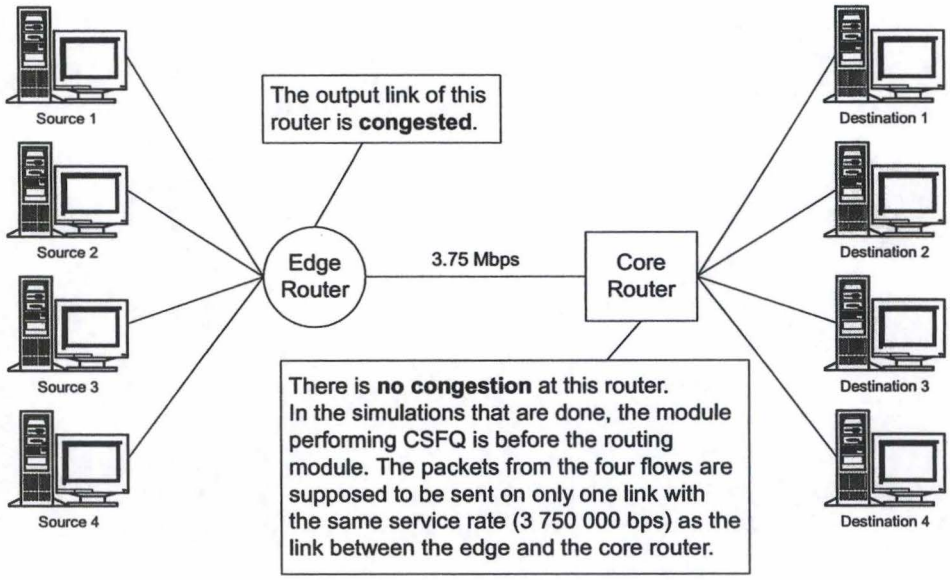

Figure 5.1: Single bottleneck scenario

The edge router performs the labelling of the packets. Then, it computes the drop probability of each packet based on the estimation of the fair share. The packets that are not discarded according to this probability, are stored in the queue of the outgoing link. The core router only computes the drop probability of the packets and then drops or accepts the packets. It does not maintain per-flow informations. The packets arriving at this router are already marked.

The edge router is the only bottleneck. The core router performs CSFQ also, but the traffic in excess of each flow should already have been dropped by the first router. The core router is only there to analyse the behavior of CSFQ on a link that is near congestion but not congested. At the core router, it is supposed that all traffic is sent on one link with the same service rate as the link connecting the two routers (figure 5.2). The service rate of the buffer acceptance module[1] is set to 3.75 Mbps. The traffic is distributed to the different output links *after* CSFQ has been performed.

Although the single bottleneck scenario is simple, it already gives a good indication of the behavior of CSFQ. It shows the influence of certain configuration parameters. It indicates also whether or not CSFQ is able to distribute the bandwidth in a fair way in simple networks.

5.1.1 First utilisation

The single bottleneck scenario will be used with different types of traffic. At first, all sources will send UDP packets except the first source that will establish one TCP connection. The three UDP sources will be sending at a rate of 500 Kbps. In this simulation, the network will be lightly congested. The total rate of the UDP flows will be 1.5 Mbps. That means that 2.25 Mbps are left to the TCP flow. The UDP flows are not bottlenecked because the

[1]The roles of the buffer acceptance module and the arrival rate estimator are presented in appendix B

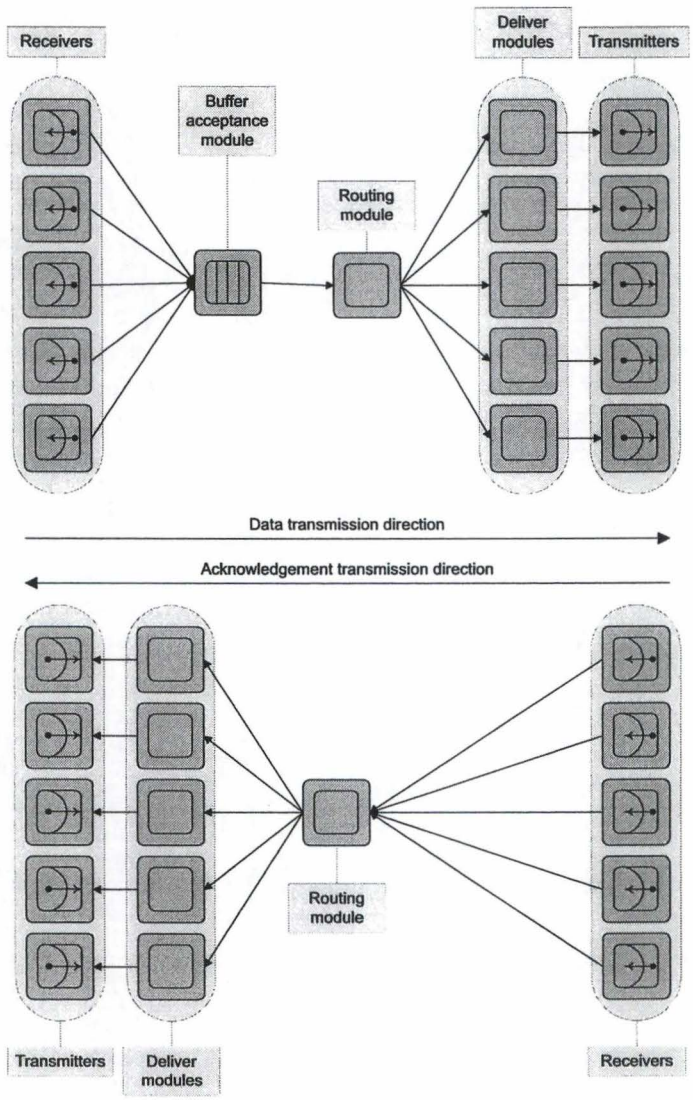

Figure 5.2: Core router module composition

fair shares of the links are above the sending rate of the UDP sources. The delays occurring to the TCP packets have been increased by 32 ms compared to the delays mentioned in the previous paragraph. This increase in delays has been distributed over the access links of the TCP source and destination.

5.1.2 Second utilisation

In the other simulations, there will only be two UDP sources. The other two sources will establish TCP connections. Each TCP source establishes 5 connections. It follows that each TCP flow will be an aggregation of 5 TCP microflows. The UDP sources have, in this scenario, a sending rate of 2 Mbps. The total sending rate of the UDP flows will be equal to 4 Mbps. This rate is above the link rates at the routers on the data path. In this scenario, the network is congested. At least it is congested at the first router. After the first router the traffic is adapted to the link of 3.75 Mbps. In theory, no drops should occur at the second router.

The flows generated by the four sources can also be the object of some guarantees. Some simulations will be made where the flows do not use any guarantees. Then, some simulations will be made where the sum of the guarantees on the flows is equal to half the rate of the link binding the two routers. Each flow profits from the same guarantee. The goal is to show, with the results of these simulations, that CSFQ is able to guarantee an amount of bandwidth to each flow and, additionally, to distribute the remaining bandwidth in a fair way between the flows. Finally, the single bottleneck scenario will be used to simulate CSFQ in a network where 90% of the bandwidth is guaranteed. All flows will also get the same amount of guaranteed bandwidth. Here we aim to see if CSFQ is able to provide each flow with its guarantee when the total guaranteed bandwidth is high. There is almost no remaining bandwidth to distribute fairly between the flows.

	Simulations			
	Single 1	**Single 2**		
		without guar.	50% guar.	90% guar.
Number of TCP flows	1	2	2	2
Number of UDP flows	3	2	2	2
Number of TCP connections per flow	1	5	5	5
Sending rate of UDP flows (Mbps)	0.5	2	2	2
Amount of guar. for each flow (Kbps)	0	0	468.75	843.75

Table 5.1: Single bottleneck scenarios

The different configurations used for the simulations of the single bottleneck scenario are summarized in table 5.1. There are two utilisations with different types traffics. In the second utilisation, no guarantees are provided, at first. Then, 50% of the 3.75 Mbps is guaranteed to the flows. And, at last, 90% of the 3.75 Mbps of the link between the two routers is reserved.

5.2 Multiple bottleneck scenario

There are six sources and six destinations in this scenario. As for all scenarios considered, there is one flow corresponding to each source. Therefore, there are six flows in this scenarios (figure 5.3). Besides, these flows go by pairs. There are always two flows following the same path in the network. Flow 1 and 2 cross the edge router, then the core router and finally the egress router. Flows 3 and 4 pass through the edge and the core routers. The remaining two flows, 5 and 6, traverse the core and the egress routers.

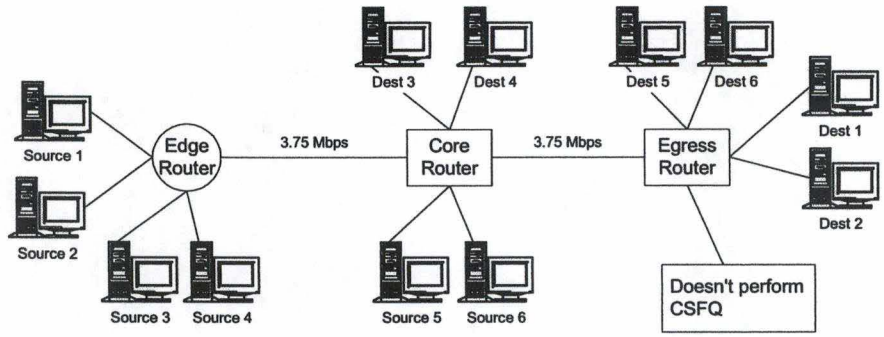

Figure 5.3: Multiple bottleneck scenario

The edge router performs the labelling of the packets from sources 1, 2, 3 and 4. Then, it computes the drop probability. It decides to accept or to drop the packets according to the drop probability. The accepted packets are sent on the output link. The core router performs the labelling of the packets from source 5 and from source 6, which enter the network. The drop probability for each packet that is routed to the link toward the egress router is computed. The packets are transmitted to the egress edge router if they are not dropped by CSFQ. The packets that are routed to their destination do not occur drops. The direct output links toward the destinations are no bottlenecks. The last router, the egress edge router, does not perform CSFQ (figure 5.4). Each incoming packet is routed to the correct output link and then transmitted to its destination. In this scenario, there are only two queues with CSFQ, namely, the queue on the link from the edge router to the core router and, from the core router towards the egress router.

On figure 5.3, it can be seen that the flows from source 1 and 2 cross all the network. They pass through three routers and thus they encounter two bottlenecks. The other four flows only cross two routers, which means that they only pass through one bottleneck. The round trip times (RTT) of flow 1 and flow 2 are higher than the RTT of the other flows because they cover a longer path. For the TCP flows, the RTT has an influence on the throughput. The throughput of the TCP flows can be approximated by equation [MSMO97]

$$\begin{aligned} Throughput &= \frac{Data\ Per\ Cycle}{Time\ Per\ Cycle} \\ &\approx \frac{Maximum\ Segment\ Size}{Round\text{-}Trip\ Time} \frac{Constant}{\sqrt{Packet\ loss\ ratio}} \end{aligned} .$$

The throughput is inversely proportional to the RTT. The maximum segment size and the constant are the same for all flows. If the packet loss ratio also has the same value for all the

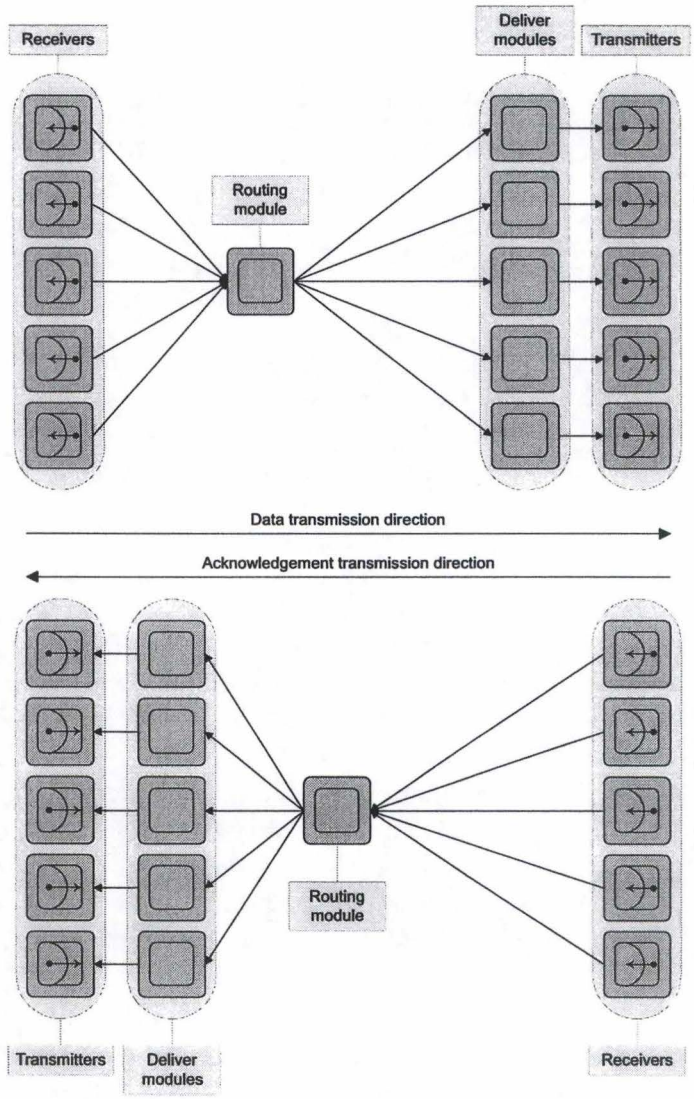

Figure 5.4: Egress router module composition

flows, the throughput of flow 1 and flow 2 will be smaller than the throughput of the other flows because flow 1 and 2 have a larger RTT. In order to have a fair allocation, all flows should have the same throughput, in this scenario. Therefore, the packet loss ratio has to be lower for flow 1 and flow 2 than for the flows from 3 to 6. A flow with a low RTT has a higher sending rate compared to a flow with a bigger RTT. When a flow has a higher sending rate than the other flows bottlenecked on the same link, more packets have to be dropped to achieve the fair share. Therefore, the packet loss ratio of this flow is higher than the loss ratio of the other flows on the link. The multiple bottleneck scenario is used here to see if similar TCP sources with different RTT can share the network in a fair way when the routers implement CSFQ.

In this scenario, the propagation delays are fixed. They are set to 2.5 ms for the access links and 10 ms for the two links connecting the routers with each other. The propagation delays are alike on all the access links. The rates of the links from users to routers and between routers are also the same as in the previous scenario (section 5.1). The size of the queues at the edge router and at the core routers are 750 Kb. It follows that the maximum delay incurred by a packet at these routers is 200 ms. From these delays as well as from the propagation delays, it can be deduced that flows 1 and 2 will have a maximum RTT of 450 ms while the maximum RTT of the other flows is 230 ms. Because the queues have a high occupancy once CSFQ has converged, the average RTTs of the flows are not much lower than their maximum RTT.

5.2.1 First utilisation

The multiple bottleneck topology can be used with different types of sources. At first, simulations will be done with TCP sources only. Each source establishes 5 TCP connections. All the flows are therefore an aggregation of 5 TCP microflows. The network is congested. As underlined previously, the objective of these simulations is to show that bandwidth can be distributed in a fair way between TCP flows that have different RTTs.

5.2.2 Second utilisation

In this second scenario, half of the sources send UDP packets and the rest are TCP sources. The sources with an odd number establish 5 TCP connections each. Even numbers are used for the UDP sources, destinations and flows. On figure 5.3, it can be seen that flows always go by pairs. That means that there are always two flows crossing exactly the same routers along the network. In each pair, there is one TCP and one UDP flow.

The two flows going from one end to the other end of the network (flow 1 and flow 2) will be guaranteed. Flow 1 have a guaranteed throughput of 0.75 Mbps. The amount of reserved bandwidth for flow 2 is 1 Mbps. The objective here is to show that guaranteed flows are approximately able to receive their fair share in addition to the guarantee they reserve. This is possible in congested networks where all flows are not bottlenecked on the same link. It will also be shown that the flows that are not the object of any reservation also receive their fair share of the bandwidth.

Table 5.2 shows the different configurations of the multiple bottleneck scenario that are used for the simulations. Two types of traffic are involved. When TCP and UDP flows share

	Simulations	
	without guar.	with guar.
Number of TCP flows	6	3
Number of UDP flows	0	3
Number of TCP connections per flow	5	5
Sending rate of UDP flows (Mbps)	0	2
Amount of guar. for flow 1(Mbps)	0	0.75
Amount of guar. for flow 2(Mbps)	0	1
Amount of guar. for the other flow (Mbps)	0	0

Table 5.2: Multiple bottleneck scenarios

the network, simulations are done with some flows having a minimum throughput that is guaranteed.

	without guar.	with guar.
Flow 1	0.94	1.25
Flow 2	0.94	1.5
Flow 3	0.94	0.5
Flow 4	0.94	0.5
Flow 5	0.94	0.5
Flow 6	0.94	0.5

Table 5.3: Throughput the different flows should get in Mbps

From the link rates, the paths of the flows and the guarantees associated to these flows, the throughput of the flows should be equal to the values in table 5.3 if the mechanism implemented is able to provide the guarantees and allocate the remaining bandwidth fairly among the flows.

5.3 Generic Fairness Configuration scenario

There are 10 sources and 10 destinations in this scenario (figure 5.5). It follows that 10 flows are considered. These are TCP flows composed of 15 TCP connections each. The flows initiated by the B and X sources are congested on the first link. The flows starting from the C sources are bottlenecked on the second link. And, the flows from the A sources as well as from the Y sources are bottlenecked on the third link. The B sources are bottlenecked *before* the central link while the A sources are bottlenecked *after* the central link and the C sources are bottlenecked *on* the central link (i.e. the second link). Sources X and Y act as background sources to create congestion at the edge router and at the second core router respectively.

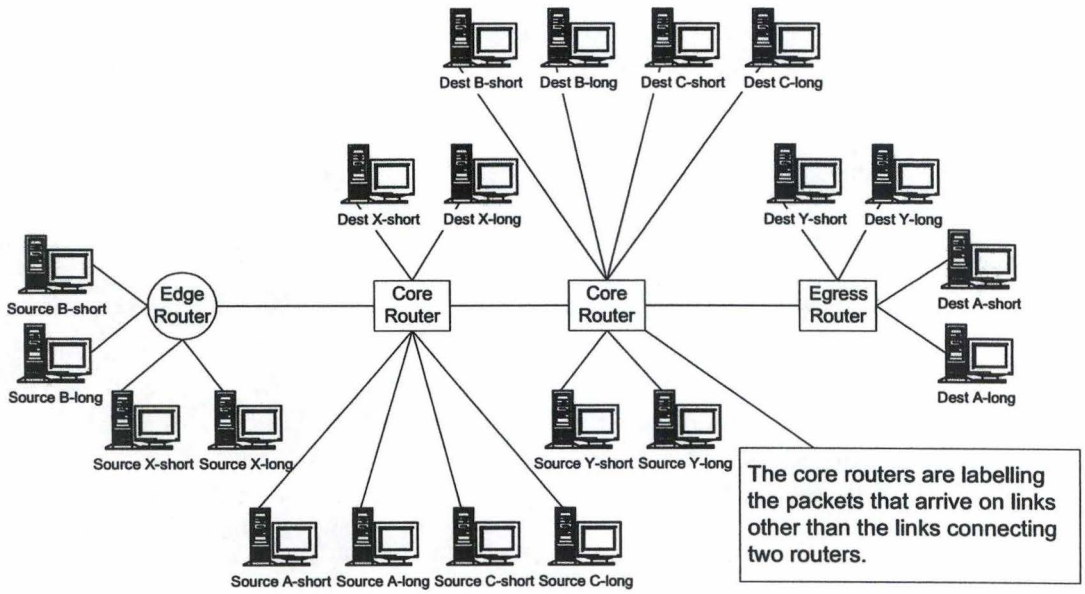

Figure 5.5: Generic Fairness Configuration scenario

In the Generic Fairness Configuration (GFC) scenario, the first router is an edge router. It behaves like the edge routers described in sections 5.1 and 5.2. The second and third routers are classical core routers. They label the packets coming from the sources directly connected to the router. Then, they route the packets to the output links. Finally, the packets are dropped or accepted according to CSFQ, if the router is not connected directly to the packets' destination. The last router is an egress edge router. It does not perform CSFQ. The arriving packets are routed to an output link. Then, they are transmitted on the link towards their destination.

The propagation delays on the different links are fixed. The delays of propagation on the links between two routers are 10 ms. Flows A-short, B-short, C-short, X-short and Y-short have a slightly smaller RTT than flows A-long, B-long, C-long, X-long and Y-long. The propagation delay on the links between a source, or destination, of a flow with a small RTT, and a router is 2.5 ms. For flows A-long, B-long, C-long, X-long and Y-long the total delay occurred on the access links has been increased by 32 ms by comparison to the flows with a smaller RTT. This increase is distributed over the two access links crossed by the flows.

5.3.1 First utilisation

In the first utilisation of the GFC scenario, there will be no guaranteed flows. The A flows will use 30% of link 2. The B flows should use 10% of the second link and the C flows should use the remaining 60%. The rate of link 2 will be set to 100 Mbps. From this rate, the rates of the two other links can be deduced. There are four flows on link 1 and two of them pass on link 2. These two flows should get 10% of link 2, i.e. 10 Mbps. Because the bandwidth on link 1 is shared between four flows, the rate of this link is set to 20 Mbps. Following the same reasoning, the rate of link 3 is set to 60 Mbps.

The goal of these simulations is to show that bandwidth can be distributed fairly between flows bottlenecked on different links in the network. These flows do not have the same fair share by opposition to the multiple bottleneck scenario.

5.3.2 Second utilisation

The amount of reserved bandwidth on the second link is 50% of the link rate, in these simulations. Each flow profits from the same throughput guarantee of 3 Mbps. The rate of link 2 is set to 36 Mbps. 50% of these 36 Mbps is reserved by the flows. There are still 18 Mbps left to share between the A, B and C flows. The excess of the B flows should get 10% of these remaining 18 Mbps. That means that the B flows should be able to use 1.8 Mbps of link 2 in excess of their guarantee. Because there are four flows sharing link 1, the total excess rate on this link should be 1.8 Mbps $\times$ 2 = 3.6 Mbps. The rate of link 1 will be set to 4×3 Mbps + 3.6 Mbps = 15.6 Mbps. The excess of the B flows should therefore, in average obtain, 1.8 Mbps on link 1. The B flows are not bottlenecked on the second link. It follows that no packet from these flows should be dropped on link 2. The B flows should be allowed to use 10% of link 2 in average. An analog reasoning leads to set the rate of the third link to 22.8 Mbps. The deduction is based on the fact that the A flows should have 30% of the bandwidth of link 2 in addition of their guarantees.

This utilisation of the GFC scenario, is made to show that flows can benefit from their guarantee and their fair share altogether in more complex networks where the fair share or the RTT varies from one flow to another.

	Simulations	
	without guar.	50% guar.
Number of TCP connections per flow	15	15
Amount of guar. for each flow (Mbps)	0	3
Rate link 1 (Mbps)	20	15.6
Rate link 2 (Mbps)	100	36
Rate link 3 (Mbps)	60	22.8

Table 5.4: GFC scenarios

Table 5.4 gives a summary of the configurations for the GFC scenario. This scenario is used without providing any guarantees to the flows. Then, the flows all have the same guaranteed throughput such that the sum of the guaranteed rates on each link is equal to half the link rate.

It is important to see the throughput that the different flows should obtain if the mechanism tested works correctly. These throughputs are presented in table 5.5.

5.4 Conclusion

In this chapter, three simulation scenarios have been presented. First, the single bottleneck scenario has been exposed as well as different utilizations of this scenario. This scenario is

	without guar.	with guar.
Flow B-short	5	3.9
Flow B-long	5	3.9
Flow X-short	5	3.9
Flow X-long	5	3.9
Flow A-short	15	5.7
Flow A-long	15	5.7
Flow Y-short	15	5.7
Flow Y-long	15	5.7
Flow C-short	30	8.4
Flow C-long	30	8.4

Table 5.5: Throughput the different flows should get in Mbps

simple but already gives an idea of the bandwidth sharing in a network where at most one link is congested.

In the multiple bottleneck scenario, there are more than one congested link and with the suggested link rates, all the bottlenecked links have the same fair share. This scenario is primarily used to show the impact of the RTT on the bandwidth allocated to TCP flows. It is also useful to check whether drops occur only at the first router on the flows' path or not, to have an indication concerning the correctness of the fair share estimation.

The congested links do not have the same fair share in the GFC scenario. The GFC scenario is the most complex scenario that has been simulated with the implemented mechanism. Its objective is to look at the accuracy of the fair share estimations through the amount bandwidth allocated to the flows.

These three scenarios are used with different types of traffic. Sometimes only TCP flows share the network. Other times TCP flows are mixed with UDP flows. In some of the proposed utilizations of these scenarios, all flows possess a certain amount of guaranteed bandwidth. In others, only certain flows are guaranteed. All these utilizations have been proposed in this chapter in order to study in chapter 6 the capability of CSFQ and MCSFQ to ensure flows with their guaranteed throughput and a fair allocation of the remaining bandwidth.

Chapter 6

Simulations

The way the bandwidth is shared by the flows with the implemented mechanism is illustrated by simulations in this chapter. First, it is studied how different parameters impact on the distribution of bandwidth and on the provision of guarantees. Then, it is showed that guarantees and approximate fair bandwidth allocation can be provided in different network topologies with various traffics.

In the fulfilled simulations, the value used for the constant in the exponential averaging equation is equal to the value chosen for the window size. Additionally, when a threshold is used, it is set to half the maximum queue size as suggested in [SSZ98a].

6.1 Fairness indicators

To evaluate the fairness of a mechanism, different indicators may be used. The goodput or throughput of the flows may be compared to the ideal goodput or throughput of these flow, respectively. The objective is to obtain the smallest difference between these two values. An other possibility is to define the deviation of the throughput or goodput of a flow from the ideal throughput or goodput of this flow. Then, the average from these deviations may be taken to evaluate the global fairness of the mechanism. It should also be checked whether these deviations are in the same range for all flows.

The deviation of the throughput from the fair share is computed as follows :

$$Deviation = 100\frac{|Fair\ share - Throughput|}{Fair\ share} \quad .$$

The average deviation is simply the algebraic average of the deviations of the throughput from the fair share for each flow :

$$\text{Average deviation} = \sum_{i=1}^{\text{nb flows}} \frac{\text{Deviation}_i}{\text{nb flows}} \quad . \tag{6.1}$$

6.2 Behavior of the mechanism

The impact of the window size, of the fair share initialization and of tail drops on the bandwidth distribution is studied in this section. A problem about the fair share estimation

in CSFQ in uncongested networks is illustrated as well.

6.2.1 Window size

The scenario used, for the simulations described in this subsection, is the single bottleneck scenario. The configuration adopted is named as the second utilisation (subsection 5.1.2) of this scenario. No guarantees are provided to the flows. A summary of the traffic characterization for this configuration is given in the second column of table 5.1.

The fair share is initialized to 1 bps for all simulations exposed in this subsection. After the first five seconds of the simulations, the statistics are reset such that the network has the time to stabilize. The behavior of the network before stabilization is put aside. Only the behavior after the first five seconds of simulations is analysed.

The constants used for the rate estimations at the edge and at the core as well as the window size, are set to the same value. The impact of this value is analyzed in the following paragraphs. At first, these constants are set to 0.1 seconds. Then, it is increased by 0.1 seconds after each simulation until the value of these constants reaches 0.9 seconds. Two sets of simulations are performed : one with CSFQ as the bandwidth allocation mechanism and the other with MCSFQ. In these simulations, each flow should have a throughput of $\frac{3.75}{4} = 0.938$ Mbps in an ideal fair allocation. In such case, the goodput of the aggregation of the two TCP flows should be near $\frac{3.75}{2} * 0.965 = 1.809$ Mbps. The factor 0.965 is obtained by dividing the amount of data in a TCP packet by the total size of the TCP packets. And, 1.809 Mbps is the maximum total goodput of the TCP flows because retransmissions are not part of the goodput. So in the best case, when no TCP packets are retransmitted, the goodput of TCP should be equal to 1.809 Mbps in a fair allocation.

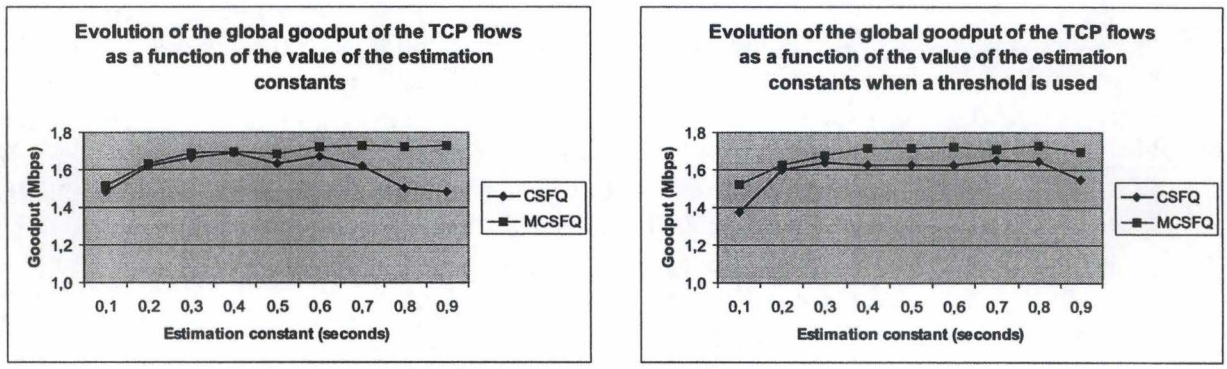

Figure 6.1: Goodput of the TCP flows

Figure 6.1 shows the goodput of the aggregation of the two TCP flows as a function of the value chosen for the constants. The graph on the left is obtained from simulations performed without a threshold while to obtain the graph on the right a threshold of half the link rate, as suggested in [SSZ98a], is used. It can be seen that with the original CSFQ, the global goodput is far from half the link capacity when the constants are set to 0.1 seconds. But when this value increases, the goodput increases quickly. When the constants are set to a value near 1 second, the goodput starts to decrease again. By comparison, when the modified version

of CSFQ is used and the value of the constants is small, we observe the same behavior as with CSFQ: the goodput is low. When the value increases, the goodput increases quickly. But, with MCSFQ, the goodput is always above the goodput of TCP in the simulations using CSFQ. When MCSFQ is used and the value of the constants increases, the goodput keeps increasing to converge near the fair allocation the TCP flows should get.

The decrease of the goodput, with CSFQ, when the constants have a high value, can be explained by the fact that when the window size increases, the fair share changes less often. The fair share is kept constant during at least a period of a window size. It can be deduced that when the window size is big and the fair share is underestimated, packets of flows sending over the computed fair share will be dropped during a long period of time even if some of the packets could have been transmitted. As a response to packet drops, TCP decreases it's sending window and therefore the goodput decreases also. If the window size is small, the impact on the goodput of an underestimation of the fair share is less important.

The goodput of the flows, when the modified version of CSFQ is used, is above the values obtained for CSFQ. This is due to the fact that the fair share computed by MCSFQ is above or equal to the value obtained by CSFQ (section 4.1). As a consequence, less packets are dropped by the fair allocation mechanism and, when the fifo queue isn't full, the goodput is greater. Because MCSFQ doesn't underestimate the fair share as much as CSFQ, when the window size increases the same behavior as with CSFQ isn't observed. Here, the goodput keeps growing slowly and remains constant for large values, close to 1 second. This means that with MCSFQ, not only the fairness is increased but, also, the performance is less sensitive to the value of the constants resulting in an easier dimensioning of the constants.

It can be seen that there is not much difference between the results obtained when no threshold is used and when a threshold is used. For CSFQ, when a threshold is used, the goodput of TCP seems more stable for a value between 0.2 seconds and 0.8 seconds than without the threshold.

The throughput of each flow in the different sets of simulations is shown in figure 6.2 as a function of the value chosen for the constants. In these simulations, each flow should have a throughput of $\frac{3.75}{4} = 0.938$ Mbps in an ideal fair allocation.

When the value of the constants is small and increases, the difference between the throughput of the flows decreases. This observation can be related to what has been said about figure 6.1. When the goodput of the TCP flows increases, the throughput of these flows increases and the throughput of the UDP flows decreases because more bandwidth is used by TCP packets. For CSFQ, when the value of the constants increases too much, the difference of the throughput of the flows increases again. The fairness of CSFQ decreases when the constants have a high value. It is not the case for the modified version of CSFQ. The difference of throughput between the different flows is smaller with MCSFQ than this difference in the simulations using CSFQ. Therefore, MCSFQ is fairer than CSFQ.

Again we can see in figure 6.2 that there is not much difference between the results obtained without and with a threshold. The global tendency of the bandwidth distribution is independent of the use of the threshold for both bandwidth allocation mechanisms, CSFQ and MCSFQ.

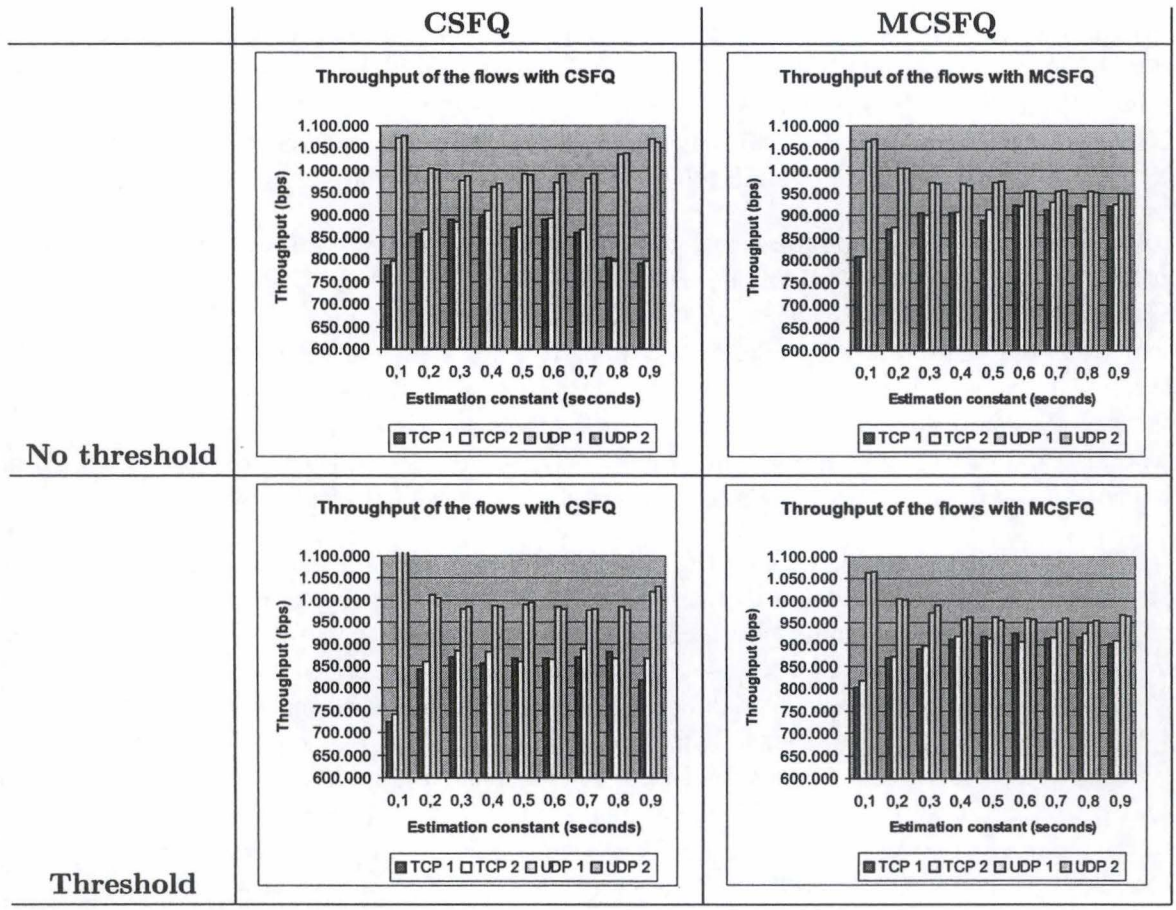

Figure 6.2: Impact of the window size on the throughput

6.2.2 Fair share initialization

In this subsection the behaviors of CSFQ and MCSFQ for an initialization of the fair share estimation to zero are compared to their behaviors with an other initialization value. The configuration of the single bottleneck scenario used for the simulations is presented in subsection 5.1.2 and it is summarized in table 5.1. The window sizes are set to 0.6 seconds which seems to be a good value for CSFQ and MCSFQ. Each flow benefits from a guaranteed throughput of 469 Kbps. That means that 50% of the bandwidth of the link joining the two routers is reserved. All flows should get the same throughput. They have the same guarantee and the remaining bandwidth should be divided between the four flows. For CSFQ, the fair share is initialized to 0 Kbps. But, the fair share is initialized to half the link rate (1.875 Mbps) in the simulations where MCSFQ distributes the bandwidth.

The results are deduced from the statistics collected at the edge router. At the edge router the output link for the data transmission is congested. Therefore, the behaviors of MCSFQ and CSFQ are identical at the edge router when used with the same configuration. As a consequence, the conclusions drawn for CSFQ with the initialization of the fair share to zero are also applicable to MCSFQ with the same fair share initialization. And, the same results are obtained with CSFQ and MCSFQ for a fair share initialization to half the link rate.

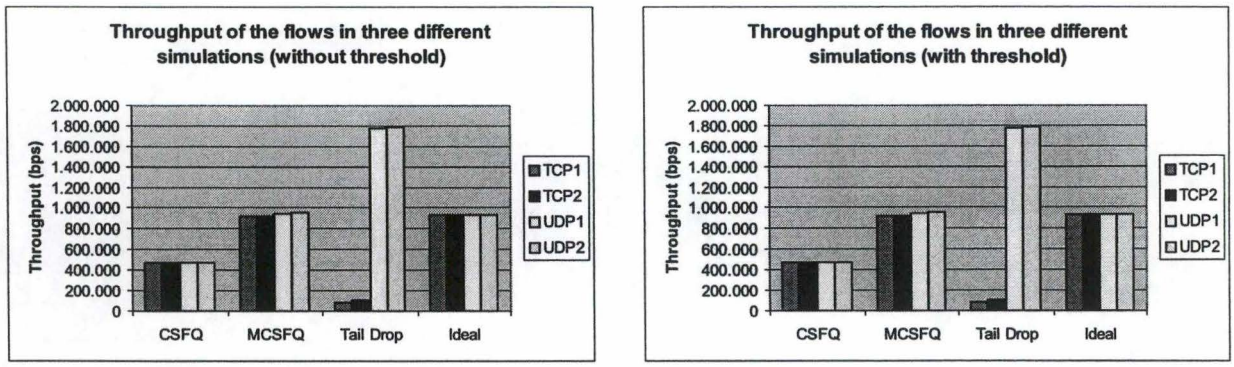

Figure 6.3: Comparison of fair share initializations

Figure 6.3 shows the throughput of the four flows with CSFQ, MCSFQ and tail drop. It can be seen that the two graphs of figure 6.3 are the same. There is always congestion at the edge node. Therefore, the threshold has no impact on the behavior of the bandwidth distribution mechanisms considered.

It can be seen that with the initialization of the fair share estimation to zero, only the guaranteed packets are transmitted to the next router. In these simulations, the edge router becomes congested already during the first window size period of the simulations. And, when there is congestion, the fair share estimation is updated according to equation 3.4. The new fair share estimation is equal to the last fair share estimation times a factor. But, at initialization, the fair share is set to zero. Then, the first update of the fair share estimation will also be zero because of equation 3.4. If the link stays congested all the time, the fair share estimation will stay equal to zero. It follows that all best-effort packets are dropped. Only the guaranteed packets, marked with a label of zero, are stored in the queue and then transmitted on the output link toward the next router.

When the initialisation of the fair share is set to half the link rate, each flow benefits from its guarantee and the remaining bandwidth is distributed approximately fairly. The flows almost all have the same throughput. The throughput of the TCP flows is slightly lower because there are some tail drops occurring at the beginning of the simulation. At the start, the fair share is overestimated and too many packets are forwarded to the queue. When the queue is full it tail drops arriving packets leading to unfairness in the bandwidth distribution (subsection 3 on page 49). Additionally, each time a packet of a TCP flow is dropped the sending rate rate of the flow decreases. The throughput of this flow is then lower until the sending window reaches a certain size again. The impact of the decrease in the sending window size is reduced with the use of an aggregation of 5 TCP micro-flows in one TCP flow.

Now, let's compare the bandwidth distribution of CSFQ and MCSFQ with initialisation of the fair share to half the link rate to the bandwidth distribution obtained with tail drop as the only buffer acceptance mechanism. With tail drop, nothing enables to provide throughput guarantees. Even worst, the bandwidth is not distributed fairly. The UDP flows take almost all the link bandwidth away from the TCP flows. This happens because TCP reacts to the congestion while the UDP flows keep sending at the same rate.

The fair share estimation could be set to 1 bps at initialization. This value avoids the

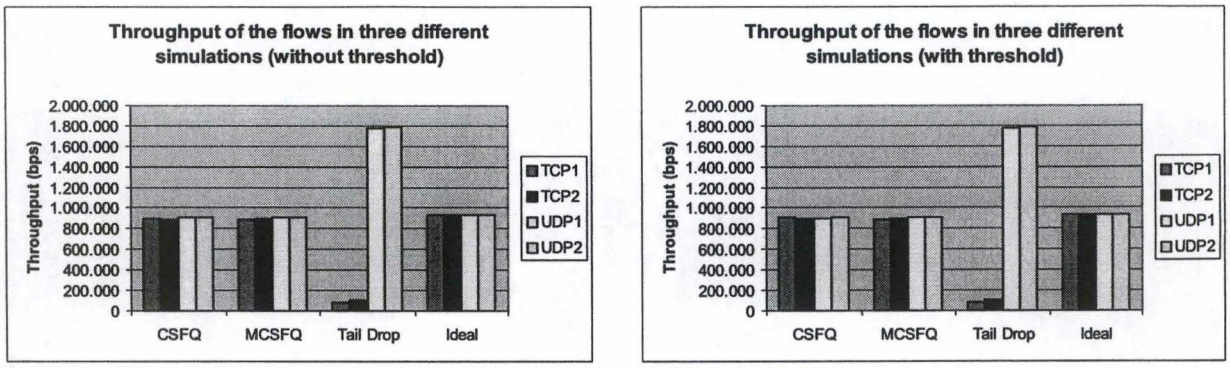

Figure 6.4: Fair share initialized to 1 bps

problem of the initialization to zero. It also prevents overestimations of the fair share to occur right after initialization. To obtain the results in figure 6.4, the fair share estimation is initialized to 1 bps for both CSFQ and MCSFQ. As said at the opening of this subsection, the results obtained for CSFQ and MCSFQ are alike and the threshold does not have an impact on a highly congested link.

The bandwidth is distributed according to the guaranteed throughputs and the fair share (for CSFQ and MCSFQ). All flows receive around the same amount of bandwidth. The throughput of all four flows is near the sum of their guaranteed throughput and the fair share of the bandwidth that is not the object of reservations. CSFQ and MCSFQ are able to achieve fair sharing of the remaining bandwidth while it is not the case with tail drop.

The fact that CSFQ and MCSFQ are able to provide minimum throughput guarantees to the flows is shown by the results obtained with CSFQ and illustrated in figure 6.3. The throughput of the flows is around their guaranteed throughput. The guaranteed packets are not dropped because a fair share of zero is not above the label of these packets. It also shows that the probabilistic marking does a good job to determine the amount of packets that should be guaranteed and the ones that should be considered as best effort, according to a contract.

In the other simulations, the fair share is usually initialized to 1 bps. We saw in this subsection that 0 bps was really a bad choice for the initialization. Also, half the link rate is not really good and it gets worst the more this value overestimates the real fair share. When the real fair share is above the initial value but different from zero, then the initial value is adequate. When the initial value is bounded by the real fair share, the higher the initialization value, the quicker the convergence will be. But, in real networks, the traffic can usually not be predicted. The real fair share is not known in advance. By choosing a higher value than 1 bps, it is never sure that there will be no overestimation.

6.2.3 Uncongested network problem

In this subsection, the results, obtained from simulations where the network is uncongested or only congested at times, are presented. The results are obtained from statistics taken at the edge router. The network used is the single bottleneck scenario, where only one source establishes TCP connections. The other three sources send UDP packets at a lower rate than

one fourth of the link bandwidth. They are not responsible of a possible congestion. The properties of the scenario used for these simulations are exposed in subsection 5.1.1 and are also summarized in table 5.1. The fair share estimation is initialized to 1 bps and the window sizes are set to 0.6 seconds.

If all four sources were sending above the fair share, the fair share of the link should be equal to $\frac{3.75}{4} = 0.938Mbps$. But in the simulations of this subsection, the UDP sources are sending at $0.5Mbps$. They leave $2.25Mbps$ to the TCP flow. In this situation, the fair share estimation should be around $2.25Mbps$ to allow the TCP flow to make use of the bandwidth that is unused by the UDP flows.

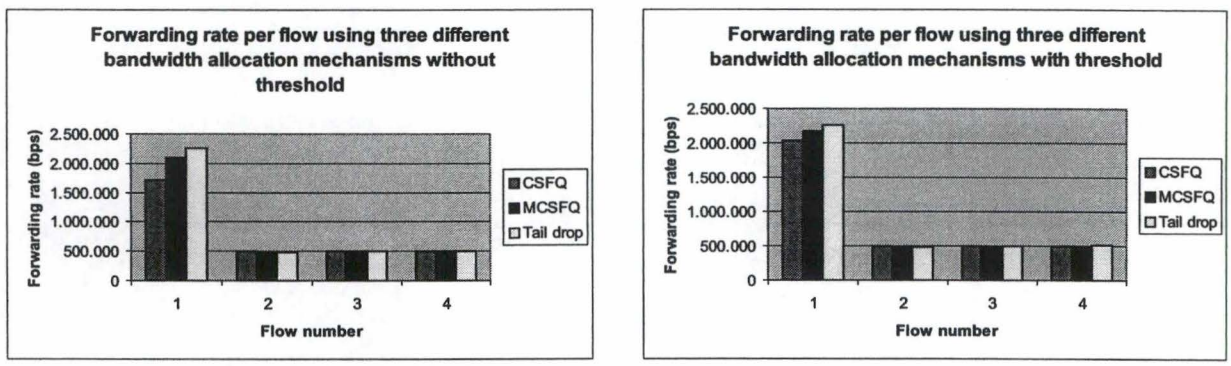

Figure 6.5: MCSFQ versus CSFQ : forwarding rate

On figure 6.5, it can be seen that the throughput of the UDP flows is around 500 Kbps independently of the bandwidth allocation mechanism. On the other hand, the throughput of the TCP flow varies depending on the bandwidth distribution mechanism used. It is at its peak with tail drop, it is slightly lower with MCSFQ and it is the smallest with CSFQ. This means that the bandwidth resources are better used with tail drop than with MCSFQ. And, even less resources are used with CSFQ.

The only congestion in these simulations is generated by the TCP flow. The congestion happens when the sending window, and therefore the sending rate, of the TCP flow are too high. With tail drop, packets are dropped only if the queue is full. Because TCP has the highest sending rate, at least one of its packets will be dropped and the window size of the flow will be decreased, reducing the congestion. When tail drop is used, UDP packets may be dropped as well as TCP packets (figure 6.6). With MCSFQ and CSFQ, packets may be dropped even if the queue is not full. The dropping is based on the fair share estimation. This estimation should allow the TCP flow to use the remaining bandwidth. We can see that the throughput of the TCP flow is higher with MCSFQ than with CSFQ. This is because the estimated fair share is higher with MCSFQ than with CSFQ (subsection 3 on page 53). In uncongested periods, the fair share estimation differs from one mechanism to the other leading to a higher estimation in MCSFQ than CSFQ. And, the network is not congested most of the time in these simulations. With MCSFQ and CSFQ, the fair share estimation is at least equal to the rate of the UDP flows. It follows that no UDP packets are dropped (figure 6.6). Only TCP packets are dropped. Each time a TCP packet is dropped the sending window is reduced and the load of the network decreases. The more TCP packets are dropped the lower the

rate of TCP will be and the less resources are used. The sending window of the TCP flow with CSFQ is usually lower than with MCSFQ. And, this sending window is lower when the bandwidth is allocated by MCSFQ than by tail drop.

It can be seen that the use of the threshold especially helps CSFQ and, to a smaller degree, MCSFQ to use more bandwidth. As said in subsection 3 on page 54, the link is considered as uncongested during a longer period when a threshold is used. This enables CSFQ to update its fair share estimation more often. An estimation of the fair share is used during a smaller period to drop the packets. This avoids small fair share estimations to be used to drop packets during long periods when the congestion is really light, like here. There is only one flow at the source of the congestion and it is a TCP flow. Once a packet from this flow is discarded, the flow's rate is reduced and the congestion is resolved.

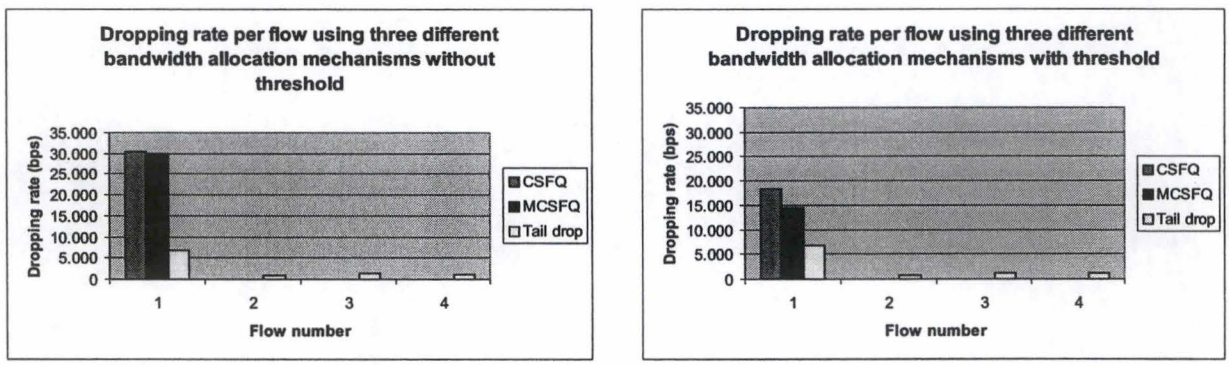

Figure 6.6: MCSFQ versus CSFQ : dropping rate

Less packets are dropped for CSFQ and MCSFQ when they use a threshold (figure 6.6). As a consequence, the TCP flow has a higher goodput with the threshold.

When the same simulations are performed with the window sizes set to 0.1 seconds instead of 0.6 seconds, the TCP flow is only able to use the same portion of bandwidth as the one used by each UDP flow (figure 6.7), if CSFQ is used. As explained in subsection 3 on page 53, with CSFQ, when the link is not congested, the fair share is set to the maximum of the labels from the packets that crossed the router during the last window. Here, the window size is rather small. The smaller the window size is, the less chances there are that at least one packet from all flows crosses the router during a window size period. The flows for which no packets have been received by the router will not influence the fair share estimation. The rate of these flows may be higher than the fair share estimation. Because TCP is bursty, the rate of the TCP flow may be high even if no packets are send during certain periods. Its packets carry labels that are above the fair share estimation. Some of its packets might be dropped even if there is no congestion on the link. On figure 6.8, it can be seen that even UDP packets can be dropped that way.

TCP tries to increase its sending rate when it does not detect congestion. But, the fair share is based on the maximum of the flow rates from the last window. Packets carrying a higher label than the rates present during the previous window will have a drop probability higher than zero. Some of these packets may be dropped. If these are TCP packets, TCP will

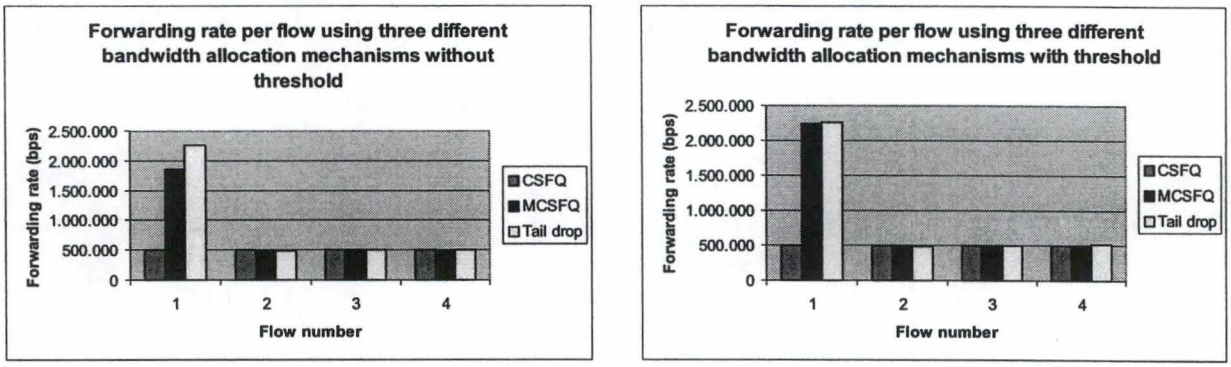

Figure 6.7: MCSFQ versus CSFQ : forwarding rate

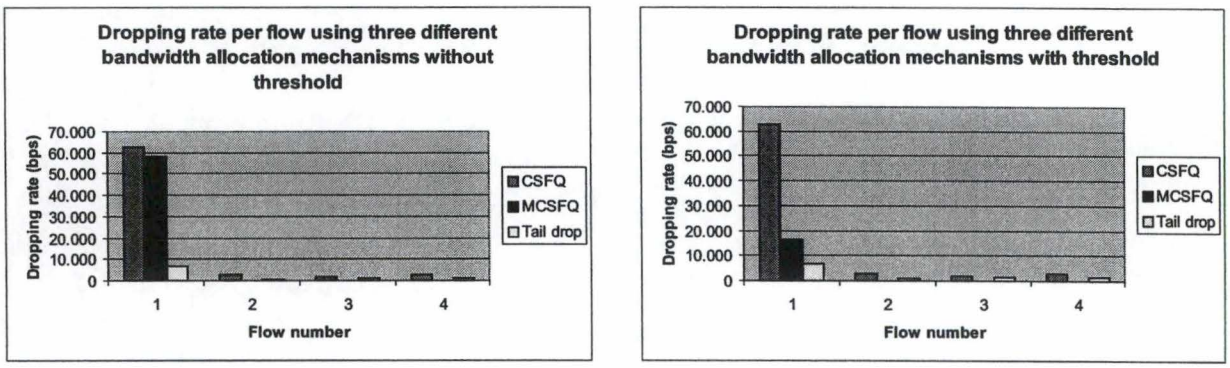

Figure 6.8: MCSFQ versus CSFQ : dropping rate

interpret the dropping as the presence of congestion and will reduce its sending window. The fair share estimation will then again be equal to the rate of the UDP flows. When the sending window of the TCP flow has increased enough again, a packet is dropped. The TCP flow is not able to send above the throughput of the UDP flows with CSFQ when the window size is small. With a higher window size, the fair share estimation is higher and there are more chances for the TCP flow to be able to increase its sending rate.

With CSFQ, the use of the threshold is not salutary. The network is always considered as not congested because the total arriving rate of the flows is around 2 Mbps while the output link rates of the routers is 3.75 Mbps. TCP is not able to create congestion. There is no oscillation of the links from congested to not congested and vice versa, which is what the threshold tries to solve.

The MCSFQ bandwidth allocation mechanism is not subject to such an important impact of the window size on the throughput of the flows in uncongested networks. The fair share estimation, in uncongested periods, is the maximum of the packets' labels, received since the end of the congestion and of the last fair share update before the link uncongestion. It follows that the fair share estimation is higher with MCSFQ than with CSFQ, allowing to forward packets at a higher rate. The fair share estimation can only stay constant or increase, with MCSFQ, in period of non congestion. It does not decrease. When TCP increases its sending

rate, the first packets with a high label may be dropped but their label is taken into account in the next fair share update. Then, the TCP flow will be able to send at a higher rate than before. When TCP's sending window will have increased enough, new packets will be dropped but the fair share estimation will increase again until it reaches 2.25 Mbps.

For the simulation performed with MCSFQ, the threshold has a positive impact (figure 6.7) on the throughput. The throughput of the TCP flow is the same as when tail drop is the only bandwidth distribution mechanism used. Here, the TCP flow is able to create congestion once the fair share has reached 2.25 Mbps. But, the congestion happens during a short time. There is congestion between the time the sending rate of the TCP flow gets above 2.25 Mbps and the time the congestion is noticed by the flow when it discovers that at least one packet has been dropped. This time is very short. If the queue does not exceed the threshold during that time, the fair share estimation method stays the same. Otherwise, the fair share estimation may decrease if the link stays congested for 0.1 seconds, leading to a lower throughput of the TCP flow. But, this happens rarely with a threshold of a certain size. If there was no threshold, the link would be considered as congested quicker and the fair share would be decreased sooner. This has as consequence that the throughput of the TCP flow is smaller than its ideal fair share for a certain time before it grows again once the fair share has increased.

On figure 6.8, it can be seen that packets from the UDP flows are dropped with tail drop and CSFQ. We cannot say that this is a fair behavior because these flows do not create the congestion. They are not bottlenecked. On the contrary, no UDP packet is dropped by MCSFQ. The throughput of TCP with MCSFQ is slightly smaller than with tail drop when no threshold is used. With a threshold, the throughput of TCP is the same with MCSFQ and tail drop. In this utilization of the single bottleneck scenario, we can say that MCSFQ is the fairer bandwidth allocation mechanism. Additionally, it allows maximum bandwidth utilisation when used with a threshold. Finally, we notice that MCSFQ is again less sensitive to the window size than CSFQ. If the window size is small the bandwidth is not shared fairly with CSFQ as seen in figure 6.7. And, a high window size does also not provide fairness (section 6.2.1).

6.3 Distribution of bandwidth

Different network scenarios with various types of traffic are used to illustrate the behavior of the implemented bandwidth distribution mechanism. This mechanism is employed to provide fair bandwidth sharing sometimes in addition of minimum throughput guarantees.

6.3.1 Single bottleneck scenario

The single bottleneck scenario is introduced in section 5.1. It is the simplest scenario used for the simulations of the implemented mechanism. Only one router is bottlenecked.

Without guarantees

In this subsection, the results from simulations where one source starts to send packets at the middle of the simulations are exposed. The single bottleneck scenario is used with the

configuration summarized in the second data column of table 5.1. The total simulation length is 100 seconds and one of the UDP sources starts to send at 50 seconds after the beginning of the simulations. The other sources start sending right at the beginning of the simulations. The statistics are reset after the first 5 seconds. The fair share estimation is initialized to 1 bps. The window size is set at all routers to 0.6 seconds. No guarantees are provided to the flows.

The goal of these simulations is to see if the fair share estimation reacts quickly to changes in the traffic. At the middle of the simulations the load is increased by 2 Mbps at the border of the network.

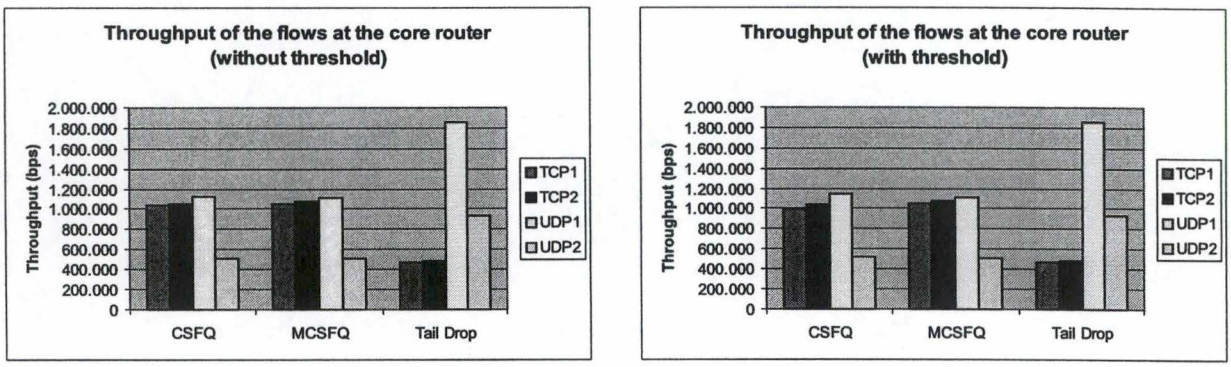

Figure 6.9: Throughput with a latecomer source

The throughput in figure 6.9 is computed as the number of bits sent during the simulation divided by the simulation length. Because the second UDP source starts to send at the middle of the simulation, the throughput of this flow should be equal to half the throughput of the other flows, in a fair bandwidth allocation. It can be seen on figure 6.9, that the bandwidth is allocated in an approximate fair way for CSFQ and MCSFQ with and without threshold.

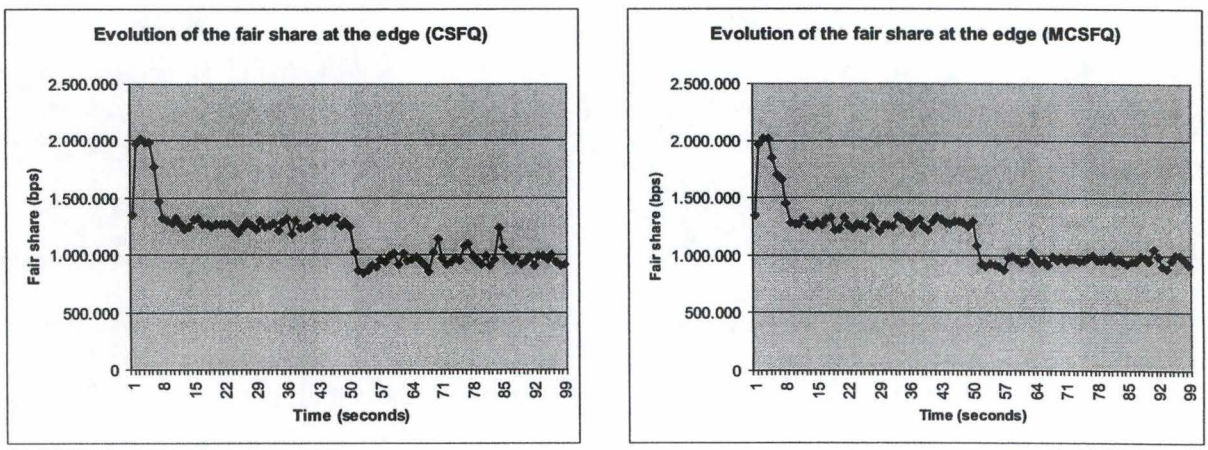

Figure 6.10: Fair share estimation evolution at the edge

At the beginning of the simulations, the fair share takes a few seconds before it converges to the real fair share. Then, when the second UDP source starts to send packets, the fair share estimation in adapted to the new traffic in at most two seconds. This is true for the simulations done with CSFQ as well as with MCSFQ (figure 6.10). We also see that the fair share estimation at the edge is more stable with MCSFQ than with CSFQ.

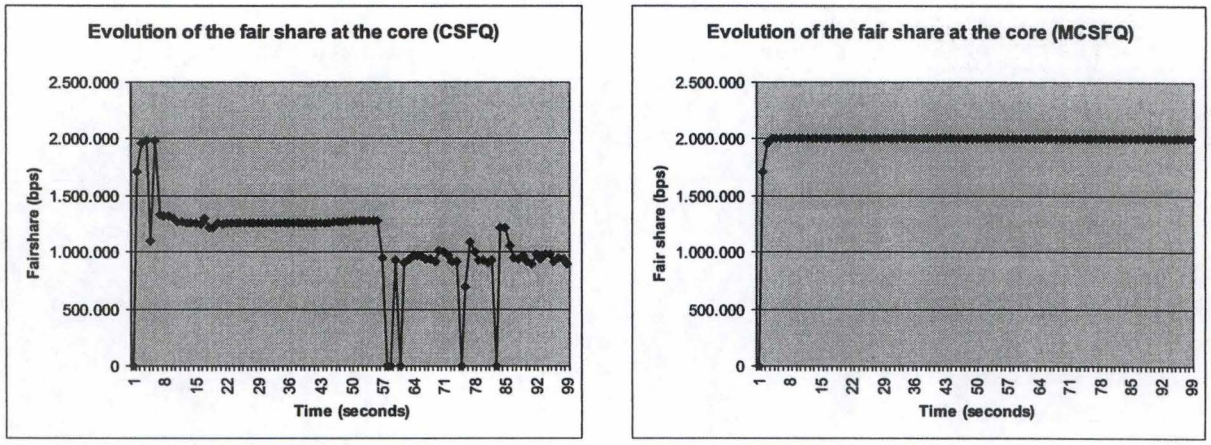

Figure 6.11: Fair share estimation evolution at the core

The core router is near congestion but it is not congested. For CSFQ, the fair share is estimated as the maximum of the labels passing during a window size period. On figure 6.11, we notice that the fair share estimation has zero as value at times. This means that during a window size period, the maximum of the labels of the packets received is zero. Because there are no guaranteed packets, it follows that no packets are received during such periods.

With MCSFQ, the fair share is set to the maximum label that passed through the router since the end of the congestion and the fair share estimation at the beginning of the uncongested period (1 bps in this case). At the beginning of the simulation, the fair share estimation increases rapidly. Then, it becomes stable.

Because the fair share estimation fluctuates at the core node with CSFQ, packets are being dropped even if there is no congestion while all packets arriving at the core node in the MCSFQ simulation are transmitted on its output links. In total, 1.6 Mb are dropped by CSFQ during the simulation of 100 seconds. For both mechanisms, the queue does not need to tail drop packets.

With 50% guarantees

The results presented by figure 6.12 are extracted from simulations done with the single bottleneck scenario. The configuration used stands in the third data column of table 5.1. The fair share estimation is set to 1 bps at the beginning of the simulation. The window size is set to 0.6 seconds. The statistics are reset after 15 seconds of simulations. All flows have the same guaranteed throughput and have a high sending rate. They should all get the same throughput in a fair bandwidth allocation. The total amount of reserved bandwidth is equal to half the rate of the link binding the two routers.

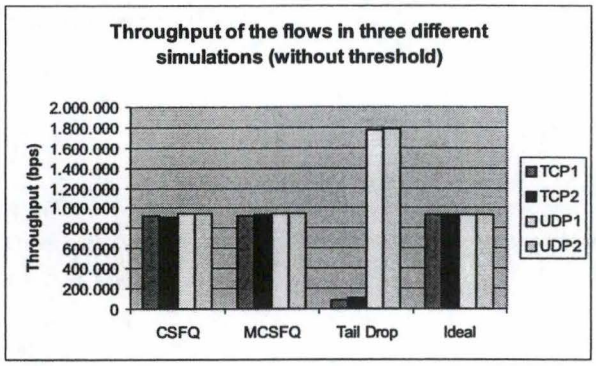

Figure 6.12: Throughput with 50% guaranteed throughput

The results on figure 6.12 have been obtained in simulation done without threshold. The simulations using a threshold give the same results for each bandwidth distribution mechanism respectively. It can be seen that all flows have almost the same throughput with CSFQ and MCSFQ. Each flow benefits from its guarantee and its fair share of the remaining bandwidth. In a best-effort network (tail drop), no guarantees are provided and the UDP flows get the majority of the available bandwidth.

With 90% guarantees

The simulations leading to the results from figure 6.13 are done with the single bottleneck scenario configured as exposed in the last column of table 5.1. The flows are each guaranteed for a throughput of 843.75 Kbps. As in the previous subsection, the fair share is set to 1 bps and the window size is set to 0.6 seconds at the initialization. The results illustrated in the figure do not include the statistics collected during the first 15 seconds of the simulation.

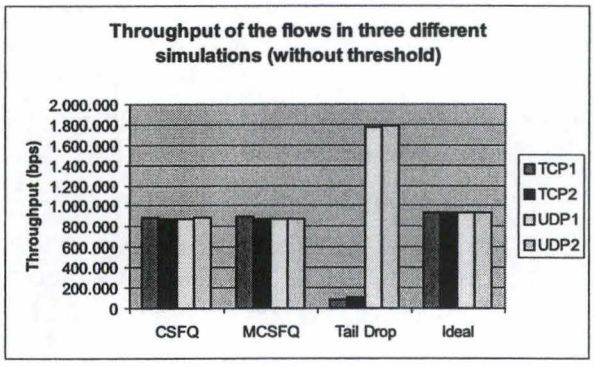

Figure 6.13: Throughput with 90% guaranteed throughput

Here, there is almost no bandwidth left to distribute between the flows once the guarantees are achieved. The goal is to see if guarantees can still be provided even if the total amount of guaranteed bandwidth is near the link rate. On figure 6.13, we can effectively see that each

flow benefits from its guarantee with CSFQ and MCSFQ. Again, the same results are obtained for the simulations done with a threshold.

6.3.2 Multiple bottleneck scenario

The multiple bottleneck scenario exposed in section 5.2 is used for the simulations whose results are presented in this subsection. More than one router is bottlenecked in the multiple bottleneck scenario. But, the fair share is the same on all links on which flows are multiplexed.

Without guarantees

The configuration of the multiple bottleneck scenario for the following simulations is summarized in the first data column of table 5.2. The fair share is initialized to 1 bps and the window size is set to 0.6 seconds at all routers. All sources are TCP sources. The first two sources have a larger RTT and encounter more bottlenecks due to the topology of the multiple bottleneck scenario.

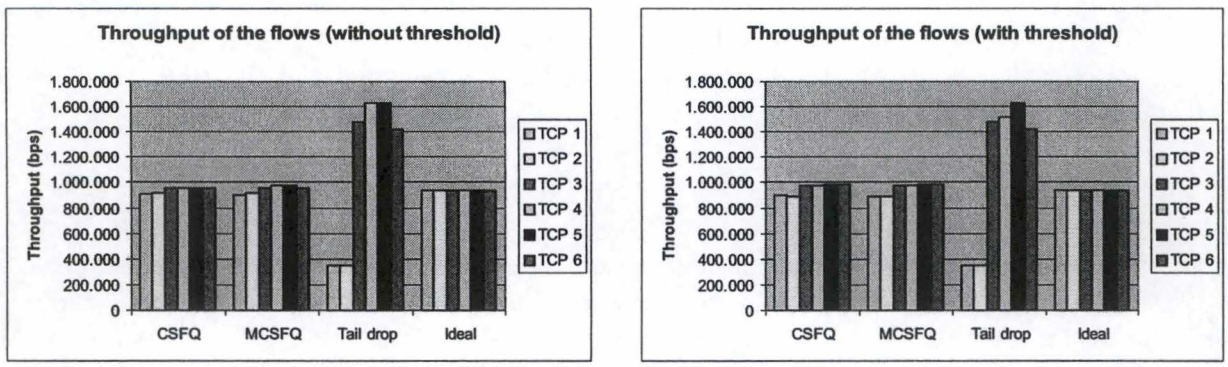

Figure 6.14: Throughput without guarantees

There are big differences between the throughput of the first two flows and the others when tail drop is the only bandwidth distribution mechanism (figure 6.14). This follows from the fact that flows 1 and 2 have a larger RTT and a larger loss ratio. Therefore, they have a smaller sending rate.

In the simulations performed with CSFQ and MCSFQ, we can see that this difference is strongly reduced. The fairness between the TCP flows is almost achieved. Additionally, we notice that the throughput difference is slightly bigger with the use of a threshold than without.

In addition to the bigger RTT, the lower throughput of the two first flows can also be explained by the fact that these two flows may have packets dropped at two routers while the other flows only have drops at one router. Once the packets of the first two flows have passed the edge router, they should not incur drops at the second router because the output link of the second router has the same real fair share as the output link of the first router. But, if the fair share estimation at the second router is smaller than the estimation of the fair share at the edge router, some packets from the first two flows are dropped. Therefore, the throughput

of flows 1 and 2 will be smaller than the throughput of flows 3 and 4. Because flows 1 and 2 have a higher RTT than flows 5 and 6, when the fair share estimation will increase, the two first flows will increase their sending window slower than flows 5 and 6 leading to a higher throughput of these two last flows.

With guarantees

The following simulations use the configuration described in the second data column of table 5.2. Additionally, the window size has been set to 0.6 seconds at all routers and the fair share estimation is set to 1 bps at the beginning of the simulations.

In these simulations, flows TCP 1 and UDP 1 are the object of guarantees. Flow TCP 1 has a minimum throughput guarantee of 0.75 Mbps and flow UDP 1 is guaranteed for 1 Mbps. That leaves 2 Mbps on each link joining two routers, on the data path, to share between four flows. Therefore, the real fair share of these link is 500 Kbps when all flows send over this fair share.

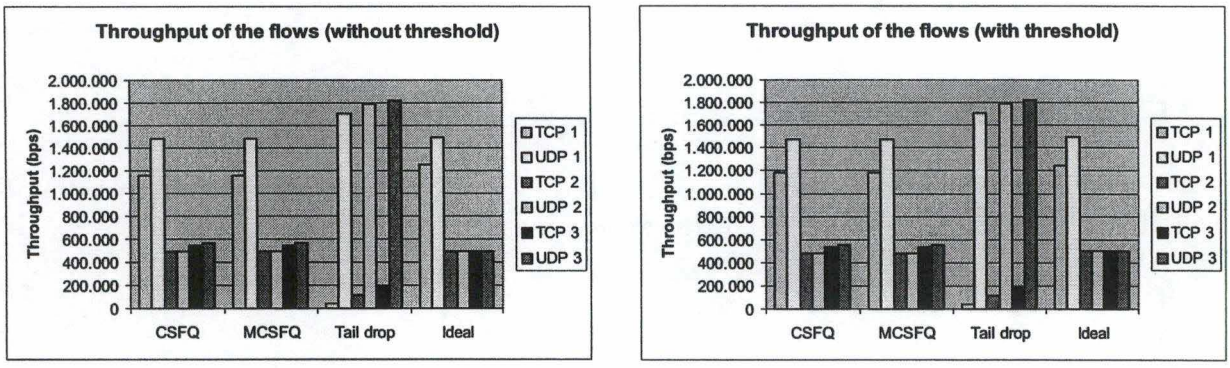

Figure 6.15: Throughput with guarantees

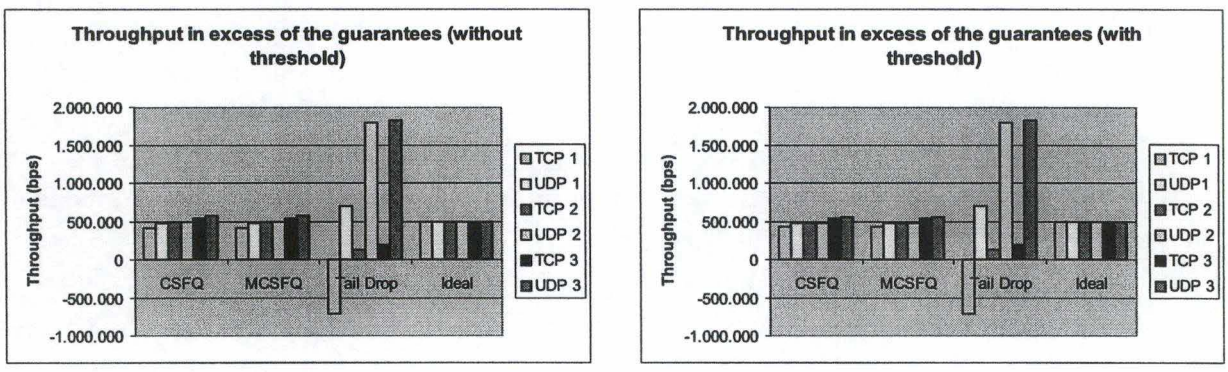

Figure 6.16: Excess throughput

When tail drop is the only buffer acceptance mechanism, the TCP flows have a lower throughput than the UDP flows. Bandwidth is stolen from them by the UDP flows (figure

6.15). Additionally, the more the identifier of the TCP flow increases the larger is the throughput. This is also true for the UDP flows. Flow TCP 1 and UDP 1 may have packets dropped at the first and the second router while flows TCP 2 and UDP 2 only have drops at the first router. The rate of flows TCP 1 and UDP 1 is higher on the output link of the first router than on the output link of the second router. This leaves more bandwidth for flows TCP 3 and UDP 3 than for flows TCP 2 and UDP 2.

With CSFQ as well as with MCSFQ, the guaranteed flows are given their minimum throughput guarantees (figure 6.15). The throughput of the guaranteed flows is above their guarantee. On figure 6.16, we notice that in excess of the guarantee, these flows almost get the same throughput as the non guaranteed flows. We can say that the bandwidth that is not the object of guarantees is distributed almost fairly. It can be noticed that the guaranteed TCP flow has a lower portion of best-effort throughput compared to the guaranteed UDP flow. Additionally, flow TCP 3 and flow UDP 3 have a higher best-effort throughput than the other flows. These tendencies are also established with tail drop. But, with CSFQ and MCSFQ, they are smoothed whereas for tail drop the excess bandwidth is not distributed fairly at all. Also, it can be observed that the fairness is increased with the use of the threshold in this case.

6.3.3 Generic Fairness Configuration scenario

The GFC scenario presented in section 5.3 is used to obtain the results introduced in this subsection. Like in the multiple bottleneck scenario, there is more than one bottlenecked link. Additionally, the fair share varies from one link to another.

Without guarantees

The results exposed in this subsection are obtained with the GFC scenario configured as in the first data column of table 5.4. As usual, the window size has a value of 0.6 seconds and the fair share is set to 1 bps at initialization. The throughput each flow should have in an ideal fair allocation is given in the first column of table 5.5. In the following, the goodput of the flows is considered, not the throughput. The goodput of a TCP flow is at most 96.5% of its throughput. As a consequence the maximal ideal goodput of a flow is equal to 96.5% of its ideal throughput.

In the simulations done with the GFC scenario, the same results are obtained for CSFQ and MCSFQ. This comes from the fact that all links are congested. In a congested mode, the behavior of these two mechanisms is the same.

Figure 6.18 compares the goodput of the flows for the simulations done with CSFQ and MCSFQ to the maximal ideal goodput of these flows. It can be seen that the real goodputs are near the ideal goodputs. We can say that fairness is achieved in these simulations.

The deviation from the fair goodput for each flow never exceeds 5%. And, the average of these deviations is less than 2%.

By comparison to the simulations done with CSFQ and MCSFQ, with only tail drop, the distribution of the bandwidth is not fair. In the GFC scenario, there are always two flows following the same path in the core of the network. We say that these two flows form a pair.

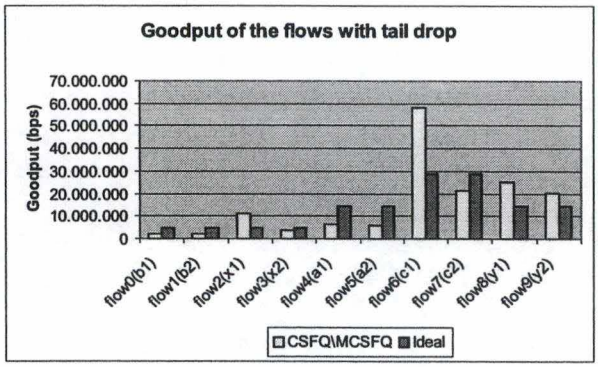

Figure 6.17: Goodput of the flows with tail drop

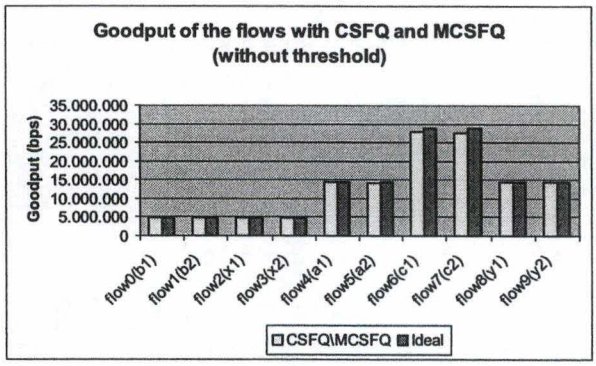

Figure 6.18: Goodput of the flows with CSFQ and MCSFQ

Among each pair of flows, the flow with the larger RTT has a lower goodput. And some pairs of flows have less bandwidth than other pairs that are bottlenecked on the same link. For example the A flows and the Y flows are bottlenecked on the link between the third and the last router. But, the goodput of the A flows is smaller than the goodput of the Y flows. The average of the deviations from the ideal goodputs, for the simulation performed with tail drop, is around 62%.

With 50% guarantees

The configuration of the GFC scenario used to obtain the results of this subsection is exposed in table 5.4, in the second column of data. Here, the flows have a certain amount of guaranteed throughput. The fair share and the window size are initialized to 1 bps and 0.6 seconds, respectively. The excess goodput of the flows is considered in figures 6.19 and 6.20.

As for the simulations with the previous configuration of the GFC scenario, in this configuration, the flows get the same goodput with CSFQ and MCSFQ.

Figure 6.19 indicates that the flows get their minimum goodput guarantees because their excess goodput is above zero. Additionally, the unreserved bandwidth is distributed approximately fairly among the different flows.

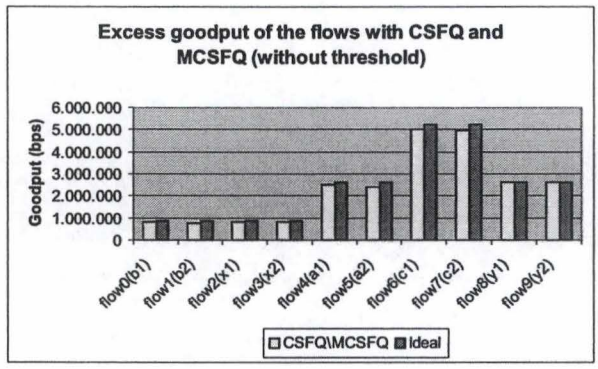

Figure 6.19: Excess goodput of the flows with CSFQ and MCSFQ

The deviations from the fair share are higher for some flows than with the previous configuration of the GFC network but they are still in an acceptable range. The bandwidth is still distributed in an approximately fair way. The biggest deviation is above 10% of the fair share and the average deviation is around 5%.

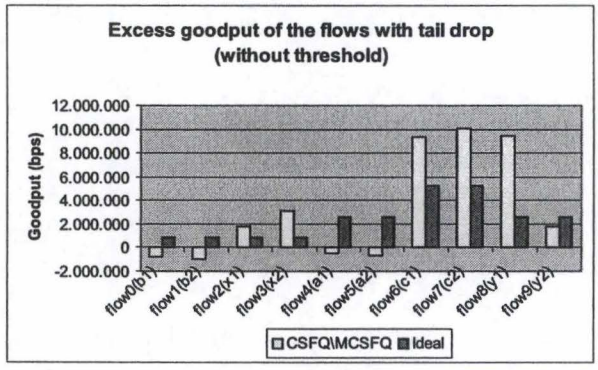

Figure 6.20: Excess goodput of the flows with tail drop

When CSFQ (or MCSFQ) is not used (figure 6.20), the flows are not guaranteed to obtain a minimum goodput. Fairness is also not achieved by comparison to the distribution of bandwidth in the GFC network using CSFQ or MCSFQ.

6.4 Conclusion

Some conclusions concerning the implemented bandwidth allocation mechanism can be drawn from the simulations that have been analyzed in this chapter. A basic deduction from this analysis is that tail drop does not allocate bandwidth fairly in many cases. Therefore, other mechanisms are needed when fairness is required. Consequently, the study of the simulation results concerns mostly CSFQ and the proposed MCSFQ.

First of all, we can say that CSFQ is more sensitive to the window size than MCSFQ.

A small as well as a large value lead to a depreciation in the fairness with CSFQ. Secondly, it has been underlined that the fair share is less underestimated by MCSFQ than by CSFQ allowing better use of the resources. We have also noticed that the behavior of CSFQ and MCSFQ is analogous in congested networks. And, in such networks, the use of the threshold has no impact on the fair share estimation and therefore on the bandwidth allocation.

Concerning the initialisation of the fair share, it has been shown that 0 bps is a bad choice for CSFQ as well as for MCSFQ. And, we proposed to use 1 bps instead to avoid overestimations of the fair share at the initialisation. It is important to avoid overestimation because they may lead to tail drops and consequently to unfairness.

In uncongested networks, it may happen that bandwidth utilization is not maximized with CSFQ. This underutilisation can be partly solved by increasing the window size. But, a more stable and efficient solution is provided by MCSFQ which provides a higher and more appropriate estimation of the fair share than CSFQ.

From multiple simulation scenarios we have shown that CSFQ and MCSFQ are able to distribute bandwidth in an approximately fair way in congested networks. The unfairness among TCP flows that happens due to variations in RTTs is reduced and TCP is protected from aggressive UDP flows. Also, the adaptation of the fair share estimation when new sources start to transmit is done in an honest time interval. Finally, small as well as large throughput guarantees can be provided to flows by these two mechanisms as long as there are enough resources to support these guarantees in the network.

Conclusion and further work

Actually, Quality of Service (QoS) is a hot topic. Its objective is to enable ISPs to propose a diversity of attractive services to their customers. This goal is praiseworthy unless it jeopardizes the performance of the actual networks. Two architectures, called Integrated Services and Differentiated Services, have been proposed for the support of QoS at the IETF. The provision of guarantees is conceptually different in these two architectures. Integrated Services enable the provision of absolute guarantees while the second architecture mostly provides relative services. The complexity of an Integrated Services network depends on the number of flows supported while the number of services supported by a Differentiated Services network determines its complexity. Therefore the Integrated Services architecture provides attractive services but is not scalable. The objective of this thesis has been to show that absolute guarantees may be provided in a Differentiated Services architecture that is more scalable and easier applicable to current networks.

Conclusion

In this thesis, we have started by introducing the concept of flow and the different guarantees that may be provided by networks to the flows. We have also underlined the importance of the provision of fairness among the flows to ensure the protection of these flows against resource starvation. And, the max-min definition of fairness has been given. Then, the Integrated Services and the Differentiated services architectures are presented by means of a comparison of their features.

In the second chapter, the general idea behind the working of core stateless mechanisms is presented. These stateless mechanisms, as we call them, are either based on a label carried by the packets (DPS) or on a feedback mechanism that carries information from the core back to the edge nodes. In both cases, the complexity is moved to the edge of the networks leading to more scalable mechanisms. Then, some of these scalable mechanisms are exposed and compared. From this study, we deduce that many types of guarantees as well as max-min fairness may be supported by core stateless mechanisms. Additionally, some of these mechanisms may be included in the Differentiated Services framework to ensure protection and guarantees to flows within a class of service clearly allowing stronger guarantees than the ones supplied by Differentiated Services networks. For certain guarantees, admission control need to be performed before ensuring that the guarantee can be provided to a flow. In the second part of this chapter, it is shown that scalable admission control, that implies to avoid the storage of per-flow state in core nodes, may be achieved. Depending on the guarantees supported, the most appropriate scalable admission control has to be determined.

Then, a particular core stateless mechanism, called CSFQ, is taken under the microscope.

CSFQ initially allows fair sharing of the bandwidth by estimating the rate of the flows at the edge, by estimating the fair share of the links at the core routers and by dropping packets based on a comparison of their labels and the current fair share estimation. In this third chapter, we have given some clarifications concerning the different estimations performed by CSFQ and others related to the initialization of CSFQ. We have shown that these clarifications are important to the bandwidth distribution resulting from CSFQ. Secondly, a case where unfairness happens with CSFQ has been raised and an explanation of this behavior has been provided. Finally, we have seen that fairness in the distribution of bandwidth depends on the accuracy of the fair share estimation, inaccuracies leading either to underutilization of bandwidth or to tail drops, these tail drops being the root of unfair bandwidth allocation. The accuracy of the fair share estimation is impacted by two configurable constants called the window size and the threshold.

The fourth chapter concerns the personal contribution brought to CSFQ. The first amelioration is proposed as a solution to the unfairness, that results from a bad fair share estimation, raised in the previous chapter. CSFQ modified as suggested is called Modified Core Stateless Fair Queueing (MCSFQ). The second amelioration suggested touch to the support of minimum throughput guarantees. The changes required by this fonctionallity to CSFQ and MCSFQ are done to the labelling process and subordinately to the buffer acceptance module, to obtain better performances.

The following chapter introduces the scenarios used for the simulations of CSFQ and its modified version MCSFQ in order to distribute the bandwidth in a fair way between the flows or to, additionally, provide minimum throughput guarantees. Then, in the last chapter, results obtained from these simulations are analysed. From these simulations, it is shown that MCSFQ has a better behavior than CSFQ in uncongested networks as well as in networks that are near congestion. And, in such conditions, MCSFQ distributes bandwidth in a fair way while the network is under utilized with CSFQ. In congested conditions, the bandwidth distribution of CSFQ and MCSFQ is alike and the threshold has no impact. The bandwidth is distributed approximately fairly and minimum guarantees are ensured to guaranteed flows in congested networks possessing one or multiple bottlenecked links having the same or various fair shares.

Further work

Because QoS is a current concern and scalability in the provision of QoS is crucial, many core stateless mechanisms have been and surely will be proposed in the future. Therefore, it would be interesting to lead a broader study of these mechanisms in order to associate to each of these mechanisms the panel of the guarantees they propose. The ultimate objective of this study would be to find or create a mechanism enabling the provision of a wide varieties of guarantees to the flows. The mechanisms handled in this thesis concern mostly bandwidth guarantees. Therefore, a special attention would be given to delay guarantees, delay jitter guarantees and prioritized dropping for flows with intra-flow priorities.

As regards the functionality extension suggested for the support of minimum throughput guarantees, results from simulations using the estimation of the aggregate guaranteed rate should be analyzed in depth to confirm its impact of bandwidth utilization.

The amelioration of CSFQ, MCSFQ, may be further fine tuned. In uncongested periods the fair share estimation should at least be bounded by the link rate and methods to allow a quicker convergence of the fair share when congestion starts should be implemented. Among such methods, we can propose the progressive decrease of the fair share in uncongested periods or to decrease the fair share estimation only when the congestion is noticed. Such proposals should be studied among others to provide a better fairness in the distribution of bandwidth by diminishing the amount of tail drops.

With the same objective of avoiding tail drops, instead of decreasing the fair share estimation by a percentage when a packet is tail dropped [SSZ98a], we could think of implementing a correction factor as suggested in [DCPE00]. The new probability to drop a packet equals to the old packet drop probability times a factor that takes into account the average queue size and a target level of the queue. Adaptations to this proposal are needed to avoid interferences of this factor in the estimation of the fair share. Therefore, the following proposal may be better suited. Instead of multiplying the drop probability by a correction factor, the estimation of the fair share could be multiplied by a correction factor to obtain a next estimation of the fair share, this correction factor having the same consistency as the one mentioned earlier in this paragraph. The goal is find a method that predicts and avoid tail drops while the solution proposed in [SSZ98a] reacts to tail drops.

An estimation of the upper bound of the forwarding rate may be used to estimate the forwarding rate on packet drops. This method is proposed in [DCPE00] and it has been implemented in our mechanism but simulations should be run to test its efficiency.

In CSFQ and MCSFQ, the estimations of the different rates are based on the exponential averaging. A question about the adequacy of such estimations to the Internet traffic can be formulated. More realistic simulations should be done with various traffic including on-off sources, bursty flows, short as well as long living flows, ... The accuracy of the estimation would be tested with the ability to provide fairness and minimum throughput guarantees.

Plenty of time may be spent in the subject treated in this thesis in the future. Many research possibilities are still available on this broad subject. Existing core stateless mechanisms may be further fine tuned or extended to the provision of a wider range of services. And, we cannot exclude the elaboration of revolutionary new core stateless mechanisms.

Bibliography

[BDC00] Olivier Bonaventure and Stefaan De Cnodder. A rate adaptive shaper for differentiated services. Internet Engineering Task Force, RFC 2963, October 2000.

[BG92] D. Bertsekas and R. Gallager. *Data Networks.* 1992.

[Bon00] Olivier Bonaventure. *Téléinformatique et réseaux : matières approfondies*, 1999-2000. Teaching reference.

[CWZ00] Z. Cao, Z. Wang, and E. Zegura. Rainbow fair queueing: Fair bandwidth sharing without per-flow state. In *Proceedings INFOCOM '00*, March 2000.

[DCPE00] Stefaan De Cnodder, Kenny Pauwels, and Omar Elloumi. A rate based RED mechanism. In *Proc. of the 10th International Workshop on Network and Operating System Support for Digital Audio and Video*, NOSSDAV 2000, Chapel Hill, NC, 26-28 June 2000.

[DR99] Constantinos Dovrolis and Parameswaran Ramanathan. A case for relative differentiated services and the proportional differentiation model. *IEEE Network*, 13(5):26–34, September/October 1999.

[FH98] Paul Ferguson and Geoff Huston. *Quality of Service Delivering QoS in the Internet and in Corporate Networks.* John Wiley & Sons, Inc., 1998.

[FSN00] W. Fang, N. Seddigh, and B. Nandy. A time sliding window three colour marker (TSWTCM). Internet Engineering Task Force, RFC 2859, June 2000.

[HBWW99] J. Heinanen, F. Baker, W. Weiss, and J. Wroclawski. Assured forwarding PHB group. Internet Engineering Task Force, RFC 2597, June 1999.

[HG99a] J. Heinanen and R. Guerin. A single rate three color marker. Internet Engineering Task Force, RFC 2697, September 1999.

[HG99b] J. Heinanen and R. Guerin. A two rate three color marker. Internet Engineering Task Force, RFC 2698, September 1999.

[JNP99] V. Jacobson, K. Nichols, and K. Poduri. An expedited forwarding PHB. Internet Engineering Task Force, RFC 2598, June 1999.

[LBL00] Na Li, Marissa Borrego, and San-qi Li. Achieving per-flow fair rate allocation within diffserv. In *Proceedings of the fifth IEEE Symposium on Computers and Communications*, ISCC 2000, Antibes, France, 3-6 July 2000.

[McD00] David McDysan. *QoS & Traffic Management in* IP *&* ATM *Networks.* McGraw-Hill, 2000.

[MIL] MIL 3, Inc., Washington, D.C. OPNET *Modeler.* Online documentation, Release 7.0.B.

[MSMO97] Matthew Mathis, Jeffrey Semke, Jamshid Mahdavi, and Teunis Ott. The macroscopic behavior of the TCP congestion avoidance algorithm. In ACM SIGCOMM, editor, *Computer Communication Review*, volume 27. July 1997.

[NSY00] M. Nabeshima, T. Shimizu, and I. Yamasaki. Fair queueing with in/out bit in core stateless networks. In *Proc. of the 8th International Workshop on Quality of Service*, IWQoS 2000, Pittsburgh, PA, 5-7 June 2000.

[PDCE00] Kenny Pauwels, Stefaan De Cnodder, and Omar Elloumi. A multi-color marking scheme to achieve fair bandwidth allocation. In *Proc. of the 1st International Workshop on Quality of future Internet Services*, QofIS 2000, Berlin, Germany, 25-26 September 2000.

[Ros01] Cedric Rosman. TCP-friendly congestion control for multimedia applications. Master's thesis, Facultés Universitaires Notre-Dame de la Paix, Namur, Belgium, June 2001.

[SPG97] S. Shenker, C. Partridge, and R. Guerin. Specification of guaranteed quality of service. Internet Engineering Task Force, RFC 2212, September 1997.

[SSZ98a] Ion Stoica, Scott Shenker, and Hui Zhang. Core-stateless fair queueing : Achieving approximately fair bandwidth allocatons in high speed networks. Technical report, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA 15213, August 1998.

[SSZ98b] Ion Stoica, Scott Shenker, and Hui Zhang. Core-stateless fair queueing: Achieving approximately fair bandwidth allocations in high speed networks. Technical Report CMU-CS-98-136, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA 15213, June 1998.

[SZ98] Ion Stoica and Hui Zhang. Providing guaranteed services without per flow management. Technical Report CMU-CS-99-133, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA 15213, May 1998.

[SZS+99] I. Stoica, H. Zhang, S. Shenker, R. Yavatkar, D. Stephens, A. Malis, Y. Bernet, Z. Wang, F. Baker, J. Wroclawski, C. Song, and R. Wilder. Per hop behaviors based on dynamic packet states. Internet Engineering Task Force, Work in Progress, draft-stoica-diffserv-dps-00.txt, February 1999.

[VMSB00] N. Venkitaraman, J. Mysore, R. Srikant, and R. Barnes. Stateless prioritized fair queueing. Internet Engineering Task Force, Work in Progress, draft-venkitaraman-spfq-00.txt, July 2000.

[Wro97] J. Wroclawski. Specification of the controlled-load network element service. Internet Engineering Task Force, RFC 2211, September 1997.

[WTP00] L. Westberg, Z. R. Turanyi, and D. Partain. Load control of real-time traffic a two-bit resource allocation scheme. Internet Engineering Task Force, Work in Progress, draft-westberg-loadcntr-03.txt, March 2000.

[ZF94] H. Zhang and D. Ferrari. Rate-controlled service disciplines. *In Journal of High Speed Networks*, 3(4):389–412, 1994.

Appendix A

Overview of OPNET

The description of OPNET is largely inspired from [MIL]. Only concepts essential to the understanding of the code from the implementation are exposed. OPNET is object-oriented. The modelling of networks in OPNET is done using three levels of abstraction (figure A.1). The role of these three layers is exposed in the following subsections.

A.1 Network model

The network model defines the topology of a communication network. The communicating entities are called nodes and the specific capabilities of each node are defined in their corresponding node model. Inside a network model there may be many node instances, with identical functionalities, that are based on the same node model. The node instances are connected by links that may also be parameterized.

A.2 Node model

The node model provides for the emulation of communication devices that can be deployed and interconnected at the network level. These devices are called nodes. They may correspond to various types of computing and communicating equipment in the real world.

The node models consist of modules and connections. Some modules offer a capability that is predefined and can be configured through a set of parameters. Transmitters and receivers belong to these predefined modules. They allow a node to be attached to communication links in the network models. Other modules can be programmed. The behavior of the programmable modules is prescribed by an assigned process model and its parameters. Among the types of connections OPNET provides between the modules from a node model, only packet streams have been used in the implementation of CSFQ. Packet streams allow packets to flow from one module to another.

A.3 Process model

Process models define the behavior of programmable modules used at the node layer. Process models are expressed in a language called Proto-C. Proto-C is based on a combination

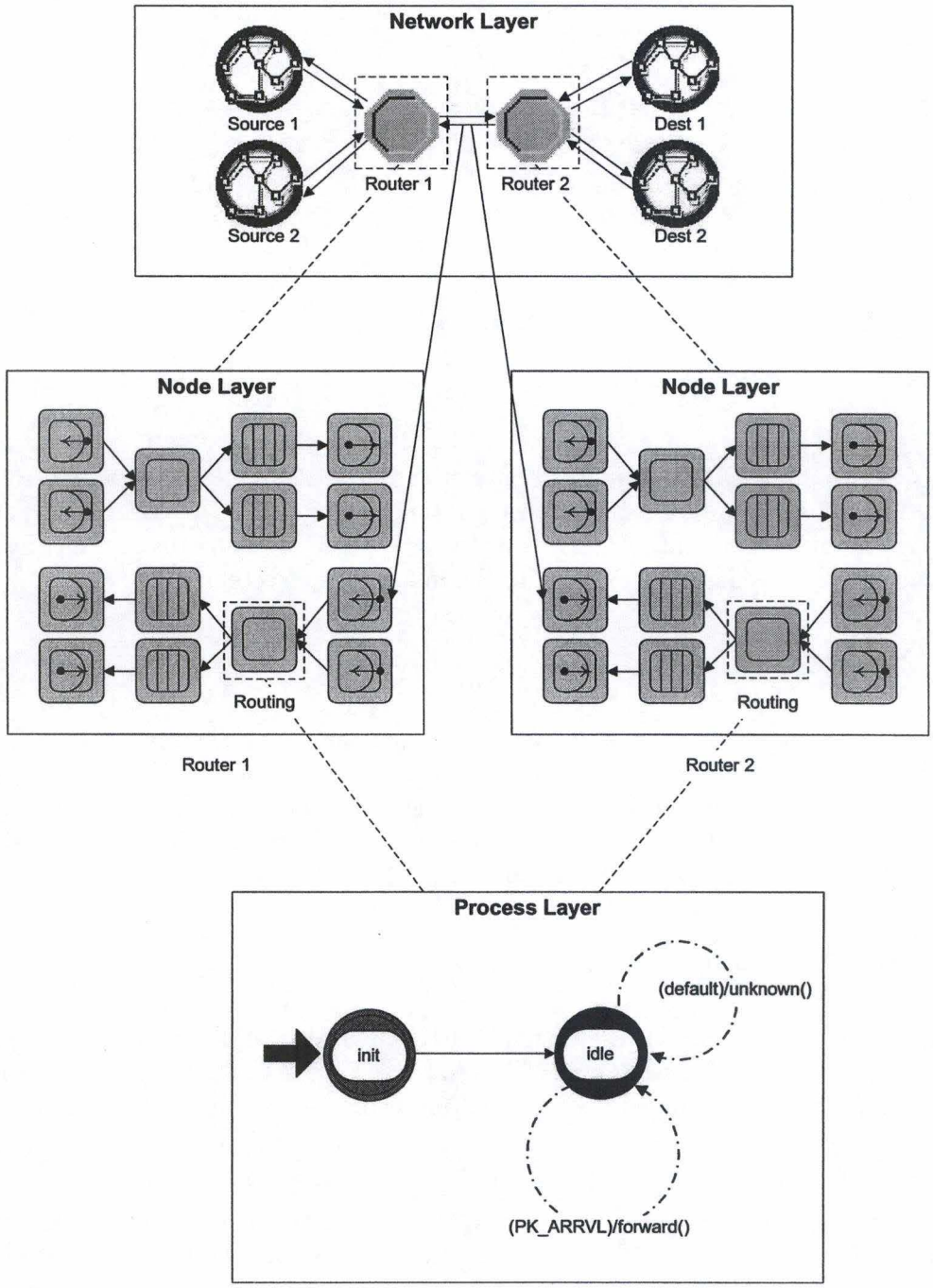

Figure A.1: The three layers of OPNET models

of state transition diagrams, a library of high level commands known as Kernel Procedures, and the general facilities of the C. Processes respond to interrupts, which indicate that events of interest have occurred such as the arrival of a message or the expiration of a timer. When a process is interrupted, it takes actions in response and then blocks, awaiting a new interrupt. A process makes the transition from one state to another when a certain condition is fulfilled.

This condition may be the emergence of an interruption when the process is in an unforced state[1]. The transition conditions can also be bound to the value of some variables. When the process is in one state and needs the occurrence of an interruption to pass into another state, it is blocked. Once the awaited interruption occurs, the process makes the transition to the next state. The actions, described in C and by means of the Kernel procedures, corresponding to the state are taken. Then, if the state is forced[2], a transition is directly made to another state of the process without being interrupted. Then, the code of the new state is executed. When a state is forced, it means that, when the module is in that state, it executes the code of the state. Then, it makes the transition to the next state without letting itself be interrupted. If an interruption occurs, it will be stored in a list of interruptions to handle and it will be taken care of after the transition. Otherwise, the state is unforced. In this case, the process may be blocked while another process runs or it is blocked until the expected interruption occurs. When the process is activated again, the process jumps to another state and executes the actions relative to the code of the new state.

A process model may define parameters, called attributes, that are used once it is instantiated as a process to customize aspects of its behavior. For example, the size of a queue and the service rate of the queue can be set through an attribute. It is possible to create as many attributes as needed. Depending on the value given to the attributes at simulation time the behavior of the process can change. In the implementation, attributes have been defined to enable the choice of using CSFQ or just accept packets in the queue in a tail-drop way.

There are many components that contribute to the definition of a process model. Among these components there are the temporary variables block, the state variables block, the header block and the function block. The goals of these components are presented in the following paragraphs.

The temporary variables of a process are defined in its temporary variables block while its state variables are declared in its state variables block. Temporary and state variables do not have the same range of application. The range of temporary variables is within the execution of the code from a state. Between one invocation of the process model and the end of the previous invocation execution, the values from the temporary variables may have changed. Temporary variables are used in order to store information that does not require persistence. They are not guaranteed to have any set of values when the process model is invoked. By opposition, the range of the state variables is the process model. The values of the state variables of a process "are frozen" when it blocks. They are found unchanged by the process when it resumes execution at the next invocation. State variables are persistent. From the perspective of each process, they retain their value over time. Changes made to state variables of one process do not affect the values of variables held by another.

The header block is an area for C language code, similar to the top portion of a C file. The constants and the data structures are defined in the header block. Among the constants defined in the header block there are the transition conditions. For example, in figure A.1, PK_ARRVL is a transition condition that is defined in the header block of the routing process. External definition files may also be included through the header block. The inclusion of header files is useful when multiple process models share a consistent set of definitions.

[1]In OPNET, unforced states in a state and transition diagram are represented by red (dark) circles.
[2]Forced states are represented by green (light) circles, in OPNET.

When functions are used in the process execution, they may be defined in a file included through the header block or they may be defined in the function block.

Table A.1 summarizes the roles of the different modeling levels in OPNET.

Modeling level	**Modeling Focus**
Network	Network topology described in terms of subnetworks, nodes and links.
Node	Node internal architecture described in terms of functional elements and data flow between them.
Process	Behavior of processes (protocols, algorithms, applications), specified using finite state machines and extended high-level language.

Table A.1: OPNET modeling levels

Appendix B

Implementation architecture

CSFQ and MCSFQ have been implemented in OPNET to provide flows with minimum throughput guarantees and fair sharing of the remaining bandwidth. Additionally, MCSFQ provides some corrections to the CSFQ algorithm. In this chapter, the basic architecture of the implementation is introduced. First, the different processes that are the building blocks of these two mechanisms are presented. Then, the constitution of an edge and a core node supporting these two mechanisms is introduced.

B.1 Process models

CSFQ and MCSFQ have been implemented by creating two modules with different objectives. The first module can be called the "arrival rate estimator". The other module implements the buffer acceptance algorithm.

B.1.1 Arrival rate estimator

The goal of the arrival rate estimator is to approximate, every time a packet arrives, the arrival rate of the flow to which the packet belongs and to mark the packet with this estimation.

The process model of the arrival rate estimator module is implemented in OPNET by the state and transition diagram shown in figure B.1. The labels on the arrows indicate the conditions under which the transactions occurs. For example, PK_ARRVL is true when the interruption handled is caused by a packet arrival. The value of EOS is true when the interruption relative to the end of simulation has arisen. When the end of simulation interruption is handled, the function labelling_eos() is called. The default condition is used when an other interruption than a packet arrival or the end of the simulation has to be treated.

The state and transition diagram indicates that at the beginning the arrival rate estimator is in the initialization state where the variables used by the estimator are set to their initial value.

Then, the module goes into the idle state and waits for packet arrivals. If the module receives an end of simulation interruption, it will handle it when it is in the idle state because the other states are forced. When a packet arrives, an interruption occurs. The interruption

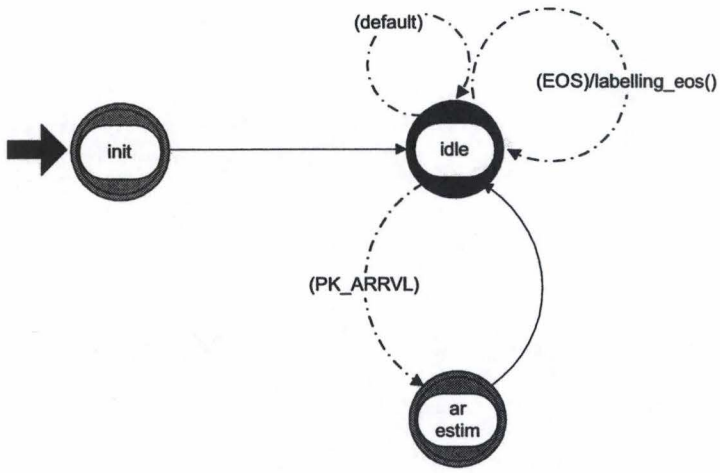

Figure B.1: Arrival rate estimator : process model

will be handled by the idle state for the same reason. A transition is then made from the idle to the ar_estimation state.

The arrival rate estimator moves into the ar_estimation state when it receives a packet. In this state, the flow to which the incoming packet belongs is identified. It has been considered here that all packets generated by the same source belong to the same flow. It follows that packets are classified according to the address of the source carried in the header. The arrival rate estimator maintains some state for each flow passing through it. This state is composed of the last arrival rate estimation from the flow and the time of the reception of the last packet belonging to the flow. Based on the state of the flow carried by the received packet, the packet's length and arrival time, the arrival rate of the flow is estimated . Once the arrival rate is obtained, the packet is marked with this value. Then, it is sent to the buffer acceptance module.

B.1.2 Buffer acceptance module

It is supposed that when a packet arrives at the buffer acceptance module, it is marked with the rate of the packet's flow. On the output link of the module, the bandwidth is distributed approximately fairly between the different flows sharing the link.

Figure B.2 shows the state and transition diagram of the buffer acceptance module. It can be seen that at the beginning, the module is in the init state where the variables used by the module are initialized.

When the execution of the init state is finished, the module goes into the idle state. It then waits for packet arrivals or for packet service completions.

When the module is in the idle state and a packet arrives, a transition is made to the forward state. In this state, either the packet is discarded or it is decided to store it in the buffer based on the previous estimation of the fair share and the label carried by the packet.

When packets of the flow, to which the current packet belongs, are dropped and the current packet is not discarded, the packet is relabelled with the fair share. The relabelling of the

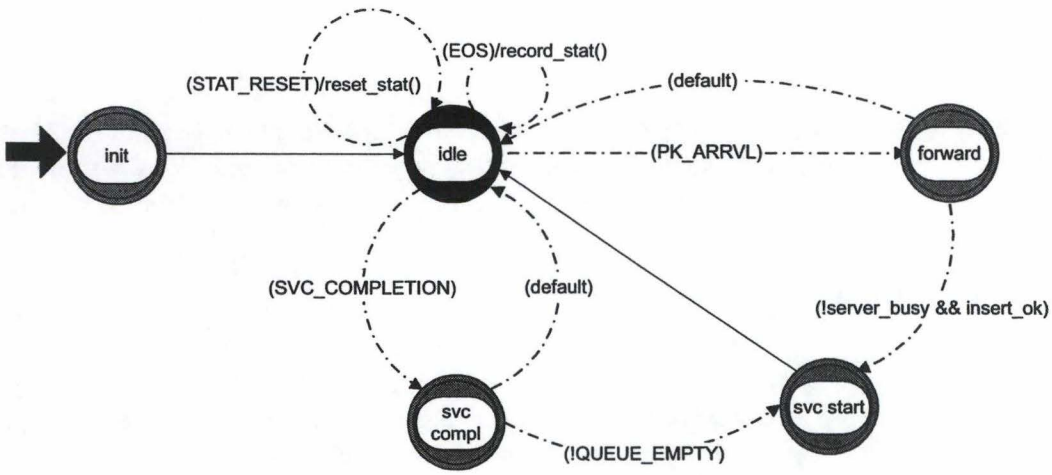

Figure B.2: Buffer acceptance module : process model

packet has to be performed to meet the precondition of the buffer acceptance module at the next node. It is required that the label of the packets arriving at the next network node should represent the estimation of the rate of their flow, at the next node. When packets from the flow are dropped, the rate of the flow decreases. It can be approximated by the fair share of the output link because the drop probability is computed such that all packets in excess of the fair share should be dropped. Therefore, the packets can be relabelled with the fair share of the link when the actual label is above the fair share.

After the decision to drop or to put the packet in the queue, the fair share is reestimated. This value of the fair share will be used at the next packet arrival.

If the packet is not in excess of the fair share, that means that it has not been decided to drop it, the packet has to be stored in the queue. But, the packet can still be discarded if the buffer happens to be full when the insertion in the queue is tried.

If the packet has been inserted in the queue and it is the first packet in that queue, the module goes into the svc_start state. The time at which the packet will be transmitted is then computed and an interruption is scheduled for that time.

When a scheduled interruption occurs, the module is in the idle state because all the other states in the diagram are forced. A interruption scheduled by the buffer acceptance module indicates that the first packet in the queue has to be transmitted. The module makes a transition to the svc_complete state. The packet at the head of the queue is sent.

When the module is in the svc_complete state and the packet associated to the interruption, that led to this state, has been sent, a transition is made to the svc_start state, if the queue is not empty. In the svc_start state, the next service completion is scheduled. But, if the queue is empty, the module goes directly into the idle state.

B.2 Nodes constitution

The utilization of the arrival rate estimator and the buffer acceptance module is illustrated by the module composition of edge and core nodes used for the simulations. Both modules are found in an edge router while only the buffer acceptance module is found in core nodes.

B.2.1 Edge node composition

In this section, an example of the module composition from a CSFQ (or MCSFQ) edge router is given. Many variations are possible. It has been chosen here to present the structure from the edge routers used in the simulations introduced in chapter 5.

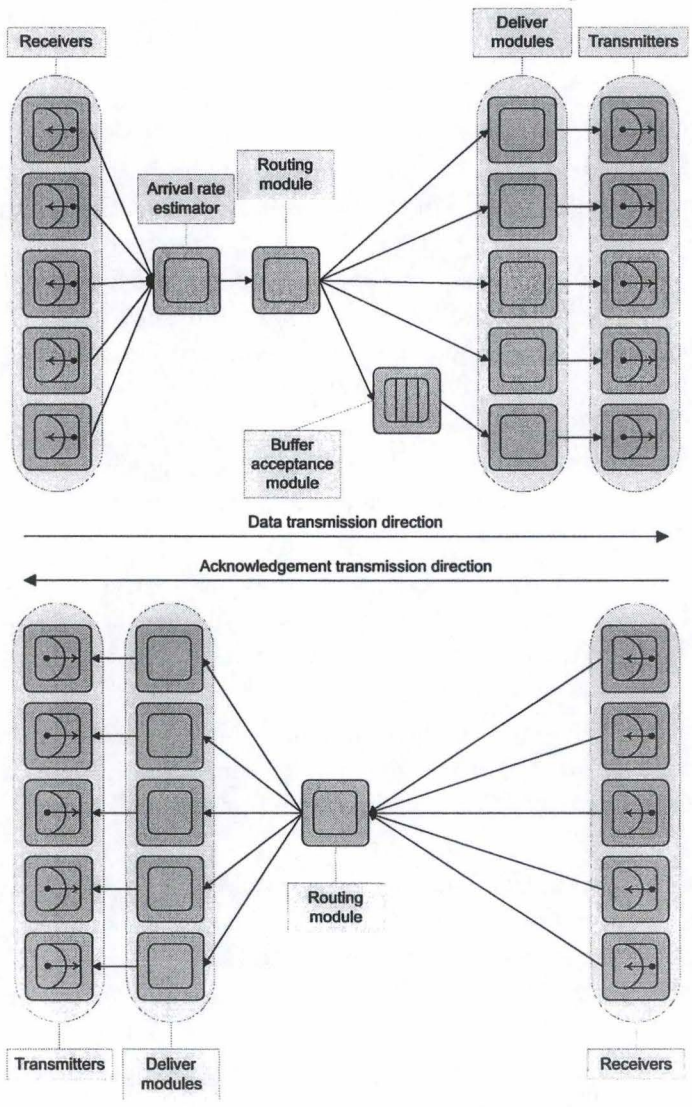

Figure B.3: Edge node

Figure B.3 shows the module composition of an edge router with 5 input and 5 output full-duplex links. In the first half of the figure, the data transmission direction is represented. Packets arriving from any of the 5 input links in the data path, go through the arrival rate estimator where they are marked with the estimated arrival rate of their flow. Then, they are sent to the routing module. The output link on which each packet has to be transmitted is determined. Once the output link toward the destination is obtained, the packets are sent on the packet streams that enable them to reach the output link. These packet streams either lead to the deliver modules of the output links or to the buffer acceptance module. The buffer acceptance module decides according to the label of the packets and the estimation of the fair share of the output link which packets are stored in the queue. When the packets arrive at the head of the queue they are sent to the deliver module of the output link. The deliver modules keep the packets for a certain time, set as a parameter to each deliver module. These times are the fixed propagation delays of the different links. Then, the packets are sent to the next nodes connected at network layer to the transmitters that stand at the end of the output streams of the deliver modules.

Here, only one buffer acceptance module is used in the data path of the edge routers. In the simulations that are performed, there is only one link where different flows are multiplexed. The other output links are supposed to be used to send data directly to final destinations. There is only one flow passing on each of these links. Because the objective of the simulations is to provide fairness between different flows, the links connected to the destinations are not bottlenecked. CSFQ (or MCSFQ) does not have to be performed on these links. But, on the links where different flows are multiplexed to be transmitted to another router CSFQ (or MCSFQ) is needed to attribute the bandwidth fairly among the flows that share the link.

In the other half of figure B.3, the dealing of the acknowledgment packets by the edge nodes is represented. Data packets are supposed to flow only in one direction. They go from the left to the right in figure B.3 as well as in the network scenarios exposed in chapter 5. The acknowledgements of the TCP flows go in the opposite direction (from the right to the left). When acknowledgment packets arrive at an edge router, they are received by the receiver modules. Then, they are transmitted to the routing module which determines the packet streams on which to send each packet. The packets arrive at the deliver modules and are delayed. Finally they are sent to the next network nodes on their path. These nodes are connected to a transmitter, reachable by the output link of a deliver module, at network layer.

In the transmission direction of the acknowledgments, the networks used for the simulations are not supposed to be congested. The amount of acknowledgments is small compared to the amount of data packets that are transmitted. CSFQ (or MCSFQ) is not performed on the acknowledgment packets.

B.2.2 Core node composition

The module composition of the core nodes (figure B.4) of the networks used in the simulations is similar to the composition of the edge nodes. At the core nodes, the data packets arriving on certain links are already marked. The labelling has not to be done to the packets that came over these links. Only the packets arriving on links connected to their source have to be marked. In the scenarios of the simulations presented in chapter 5, the packets that are marked all come via the same link. Therefore, there is only one receiver that is not connected

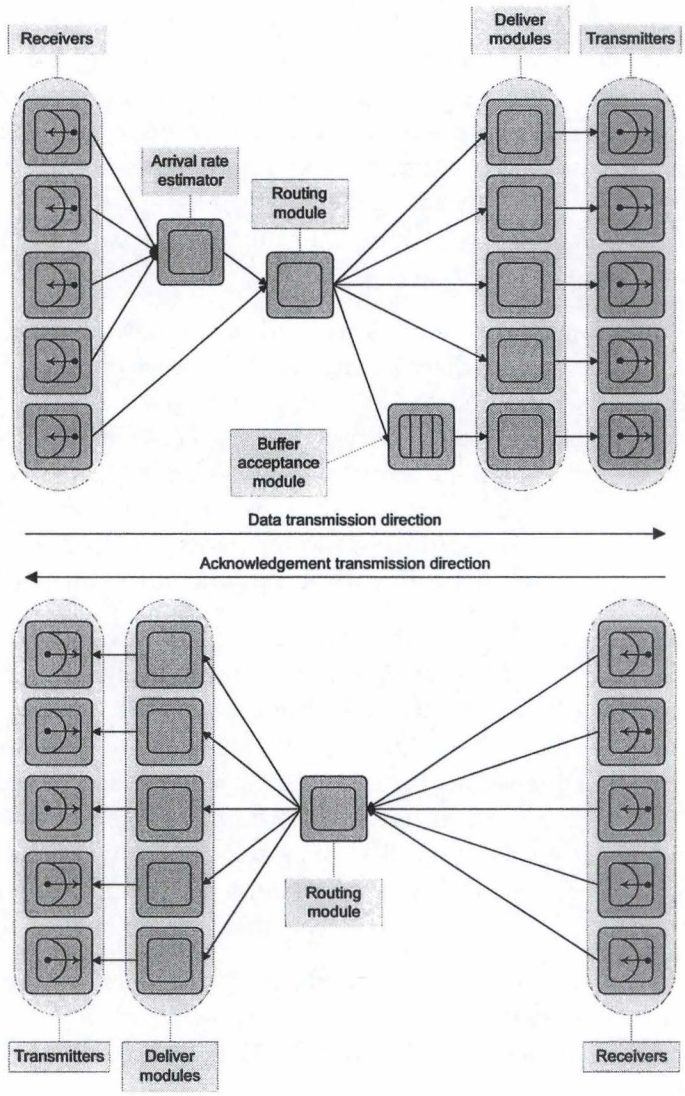

Figure B.4: Core node

to the labelling module. Packets received by this receiver are directly sent to the routing module. Then, the packets from the flows that are multiplexed at this router are sent to the buffer acceptance module. Finally, they are transmitted to the deliver and transmitter modules if they are not discarded. The packets from the flows arriving at destination at the next nodes do not go through the buffer acceptance module because the links toward the destinations are not overloaded. These packets are sent to the deliver modules and finally they are transmitted on the links connected to their destination.

For the acknowledgement packets, the mechanism is the same as at the edge routers. There is no congestion on this way of transmission. The packets arrive at the receivers and they are routed to the correct output streams. They incur a fixed delay at the deliver modules. And, finally, they are sent on the output links connected to the transmitters.

Appendix C

Implementation code

The code from the implementation of the processes that compose the edge and core nodes used in the simulations is given in this chapter. Four process models have been developed : the deliver process, the routing process, the labelling process and the core process. The last two processes correspond to the arrival rate estimator and the buffer acceptance module (annexe B), respectively. The code from the deliver and the routing processes is presented first. The deliver and the routing processes are not proper to CSFQ and MCSFQ. After giving the processes that compose the framework of any router composition, the code from the modules implementing CSFQ and MCSFQ is submitted.

The coding of the process models is listed for each process model with the declaration of the temporary variables, the declaration of the state variables, the header block, the function block and the code executed in each state of the finite state diagram.

C.1 Deliver process

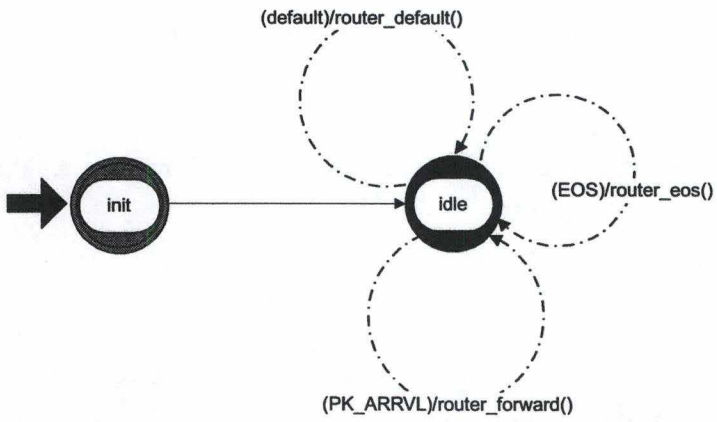

Figure C.1: Deliver process

All packets arriving on the input stream of the deliver process are transmitted to the next network node bound to the deliver module with constant delay given as a parameter to the

process at simulation time. The delay introduced by a deliver process corresponds to the fixed propagation delay on a link. The deliver process is used to avoid the need to configure the link rates[1] and delays. When the deliver process is used, there is no problem about the reliability of the links binding the different nodes. The state and transition diagram of the deliver process is given in figure C.1.

C.1.1 state variables

```
Objid   \routerdest;
double  \routerdelay;
```

C.1.2 temporary variables

No temporary variables

C.1.3 header block

```
#define PK_ARRVL (op_intrpt_type()==OPC_INTRPT_STRM)
#define EOS (op_intrpt_type()==OPC_INTRPT_ENDSIM)
```

C.1.4 function block

```
void router_forward(void){
    Packet* pk;

    pk=op_pk_get (op_intrpt_strm());
    op_pk_deliver_delayed (pk, routerdest, 0, routerdelay);
}

void router_eos(void){
    printf("The interruption eos occured\n");
}

void router_default(void){
    printf("The interruption that occured isn't a strm_intrpt\n");
}
```

C.1.5 init state

In the init state, the delay of the packets, for this node, is set using the value of an attribute. And, the destination process of the packets is determined by searching the first process or queue of the node connected though a transmitter to the deliver process instance.

```
Objid  my_self;
Objid txid,lkid,rxid;
```

[1] In our models, the link rates are determined by the service rates of the queues implemented in the core process.

```
my_self=op_id_self();

op_ima_obj_attr_get(my_self,"routerdelay",&routerdelay);

/* Objid of the transmitter connected to the deliver process */
txid = op_topo_assoc(my_self,OPC_TOPO_ASSOC_OUT,OPC_OBJTYPE_PTTX,0);
if (op_topo_assoc_count(txid,OPC_TOPO_ASSOC_OUT,OPC_OBJTYPE_LKSIMP)==1){
    /* Objid of the link connected to the transmitter */
    lkid = op_topo_assoc(txid,OPC_TOPO_ASSOC_OUT,OPC_OBJTYPE_LKSIMP,0);
    /* Objid of the receiver connected to the link */
    rxid = op_topo_assoc(lkid,OPC_TOPO_ASSOC_OUT,OPC_OBJTYPE_PTRX,0);
    if (op_topo_assoc_count(rxid,OPC_TOPO_ASSOC_OUT,OPC_OBJTYPE_QUEUE) == 1){
        /* Objid of the queue connected to the receiver */
        routerdest = op_topo_assoc(rxid,OPC_TOPO_ASSOC_OUT,OPC_OBJTYPE_QUEUE,0);
    }
    else{
        /* Objid of the process connected to the receiver */
        routerdest = op_topo_assoc(rxid,OPC_TOPO_ASSOC_OUT,OPC_OBJTYPE_PROC,0);
    }
}
```

C.2 Routing process

The objective of the routing process is to send the incoming packets on the correct output stream. The routing is done based on a table that enables the association of each virtual path index (VPI) which is embedded in the packet header with the number of an output stream. The state and transition diagram of the routing process is given in figure C.2.

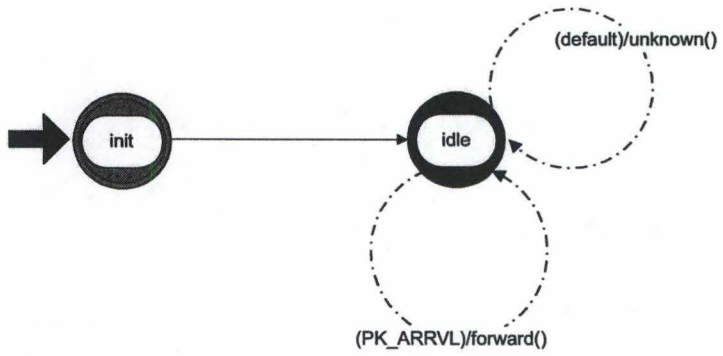

Figure C.2: Routing process

C.2.1 state variables

```
/* used to route the packets to their destination */
RoutingTable    \routingtable;
```

C.2.2 temporary variables

No temporary variables

C.2.3 header block

```
#include </home/users/cnodderst/diffserv/routing/routing.h>
#define PK_ARRVL (op_intrpt_type() == OPC_INTRPT_STRM)
```

C.2.4 function block

```
void forward(void){

    Packet *pk;
    int stream_index;
    int vpi;

    stream_index = op_intrpt_strm();
    pk = op_pk_get(stream_index);
    /* take the value of the VPI field of the packet */
    op_pk_nfd_get(pk, "VPI", &vpi);
    /* determine the packet stream corresponding to the vpi
    and send it on this packet stream */
    op_pk_send(pk,routingtable[vpi]);
}

void unknown(void){
    printf("Unknown event has occured\n");
}
```

C.2.5 init state

```
/* initialisation of the routing table */
init_routetable(routingtable);
```

C.3 Labelling process

This process is used by edge nodes to label the packets that arrive with the estimation of the arrival rate of the packet's flow. It keeps some informations for each flow in the fs array. On packet arrival, the process estimates the packet's flow arrival rate and updates the flow's informations. It then marks the packet and forwards it.

C.3.1 state variables

```
/* Array where each element is the state of a flow */
flowstate   \fs[MAX_FLOWS];

/* Constant used in the estimation of the arrival rate of the user */
```

```
/* flow.                                                              */
double  \K;

/* file descriptor used to store the statistics */
FILE *  \fptr;

/* indicates if the support of minimum guarantees is wanted */
int \min_support;

/* indicates if the marking of the traffic is made using the token */
/* bucket                                                          */
int \bucket_mark;

/* Indicates the size of the token bucket */
double  \bk_size;

/* Array used to produce the averages of the statistics */
flowstatistics  \fs_stat[MAX_FLOWS];

/* indicates the length of the intervals on which the stat averages */
/* are performed                                                    */
double  \time_average;

/* indicates the start of the current stat averaging interval */
double  \start_averg_time;

/* indicates if the statistics are of type average or not */
int \average;
```

C.3.2 temporary variables

```
Packet *pk;
double pk_arrvl_time;

int flow_nb; /* identifier of the flow used to access its state */
double arrvl_rate;

int i; /* counter */
int strm_index;

char fmt[20];/* to store the format name */

double p_out;/* probability by which the packet is marked
             to be out of the guarantee */
```

C.3.3 header block

```
#include <math.h>
#include </home/users/cnodderst/diffserv/routing/routing.h>
#include </home/users/pelsserc/op_models/flowclassify.h>

/* definition of the conditions */
#define PK_ARRVL (op_intrpt_type() == OPC_INTRPT_STRM)
#define EOS (op_intrpt_type()==OPC_INTRPT_ENDSIM)

/* maximum number of flows */
#define MAX_FLOWS (MAX_ROUTING_VP/2)

/* define the structure of the state of a flow */
typedef struct{
    double guar_rate; /* reservation of the flow */
    double estim_rate; /* current flow's estimated rate */
    double prev_time; /* arrival time of the previous
                      packet of the flow */
    double estim_exc_rate; /* current flow's estimated
                           excess rate */
    double exc_prev_time; /* arrival time of the previous
                      excess packet of the flow */
    double bk_content; /* number of tokens in the bucket */
}flowstate;

typedef struct{
    double avrg_arrvl;
    double avrg_exc;

    double sum_arrvl;
    double sum_exc;

    int nb_val_arrvl;
    int nb_val_exc;
}flowstatistics;
```

C.3.4 function block

```
/* estimation of a rate using the exponential averaging */
double estim_flow_rate (int flowid, Packet *pkptr,
    double pk_arrvl_time){

    int pk_size; /* size of the packet */

    double rate; /* estimation of the rate */
    double prev_arrvl_time; /* arrival time of the
```

```
                                previous packet */

    double pk_interval; /* time between the last and the
                           current packet */

    /* affectation of the packet's size to pk_size*/
    pk_size = op_pk_bulk_size_get(pkptr);

    rate = fs[flowid].estim_rate;
    prev_arrvl_time = fs[flowid].prev_time;
    fs[flowid].prev_time = pk_arrvl_time;

    pk_interval = pk_arrvl_time - prev_arrvl_time;

    if (pk_interval != 0.0){
        fs[flowid].estim_rate = (1.- exp(- pk_interval / K))
                * ((double)pk_size/pk_interval)
            + exp(- pk_interval / K) * rate;
    }
    else{
        fs[flowid].estim_rate = ((double)pk_size/K) + rate;
    }

    return(fs[flowid].estim_rate);
}

/* estimation of the excess rate of a flow using
the exponential averaging */
double estim_exc_flow_rate (int flowid, Packet *pkptr,
    double pk_arrvl_time){

    int pk_size; /* size of the packet */

    double rate; /* estimation of the rate */
    double prev_arrvl_time; /* arrival time of the
                               previous packet */

    double pk_interval; /* time between the last and the
                           current packet */

    /* affectation of the packet's size to pk_size*/
    pk_size = op_pk_bulk_size_get(pkptr);

    rate = fs[flowid].estim_exc_rate;
    prev_arrvl_time = fs[flowid].exc_prev_time;
    fs[flowid].exc_prev_time = pk_arrvl_time;
```

```
    pk_interval = pk_arrvl_time - prev_arrvl_time;

    if (pk_interval != 0.0){
        fs[flowid].estim_exc_rate = (1.- exp(- pk_interval / K))
                * ((double)pk_size/pk_interval)
            + exp(- pk_interval / K) * rate;
    }
    else{
        fs[flowid].estim_exc_rate = ((double)pk_size/K) + rate;
    }

    return(fs[flowid].estim_exc_rate);
}

/* the last statistics are written to the stat file at the end
of the simulation */
void labelling_eos(void){
    int i;
    double crt_time = op_sim_time();
    if (average){
        for (i = 0; i < MAX_FLOWS; i++){
            if (fs_stat[i].nb_val_arrvl != 0){
                fs_stat[i].avrg_arrvl = fs_stat[i].sum_arrvl /
                    fs_stat[i].nb_val_arrvl;
                fprintf(fptr,"arrvl_rate;%d;%f;%f\n",i,
                    start_averg_time,
                    fs_stat[i].avrg_arrvl);
            }
            if (fs_stat[i].nb_val_exc != 0){
                fs_stat[i].avrg_exc = fs_stat[i].sum_exc /
                    fs_stat[i].nb_val_exc;
                fprintf(fptr,"excess_rate;%d;%f;%f\n",i,
                    start_averg_time,
                    fs_stat[i].avrg_exc);
            }
        }
    }
    else{
        for (i = 0; i < MAX_FLOWS; i++){
            if (crt_time > start_averg_time + time_average){
                if ((fs[i].prev_time > 0.0)
                || (fs[i].estim_rate > 0.0)
                || (fs[i].exc_prev_time > 0.0)
                || (fs[i].estim_exc_rate > 0.0)){
                    fprintf(fptr,"arrvl_rate;%d;%f;%f\n",i,
                        crt_time,
                        fs[i].estim_rate);
```

```
                    fprintf(fptr,"excess_rate;%d;%f;%f\n",i,
                        crt_time,
                        fs[i].estim_exc_rate);
                }
            }
        }
    }
    fprintf(fptr,"end of sim\n");
    fclose(fptr);
}

/* increment the number of tokens in a bucket on packet arrival */
void incr_bk(int flow_id, double pk_arrvl_time){
    double t,token;

    t = pk_arrvl_time - fs[flow_id].prev_time;
    token = t * fs[flow_id].guar_rate;
    if (bk_size >= fs[flow_id].bk_content + token){
        fs[flow_id].bk_content = fs[flow_id].bk_content + token;
    }
    else{
        fs[flow_id].bk_content = bk_size;
    }
}
```

C.3.5 initialization state

```
/* initialisation of the rate estimator and packet
labelling process */

char stat_file[20];
/* next six variables are used to get the guarantees of the flows */
Objid my_self,table,line;
int nb_lines;
int vpi;
double g_rate;
char mark_type[20];
char stat_type[20];

if (op_ima_obj_attr_get(op_id_self(),
    "estim_const",&K) == OPC_COMPCODE_FAILURE)
    printf("error getting attribute's value\n");
if (op_ima_obj_attr_get(op_id_self(),
    "stat file",stat_file) == OPC_COMPCODE_FAILURE)
    printf("error getting attribute's value\n");

if (op_ima_obj_attr_get(op_id_self(),
```

```
    "min_guar_support",&min_support) == OPC_COMPCODE_FAILURE)
    printf("error getting attribute's value\n");

if (op_ima_obj_attr_get(op_id_self(),
    "exc_mark",mark_type) == OPC_COMPCODE_FAILURE)
    printf("error getting attribute's value\n");

if (strncmp(mark_type,"probabilistic",13) == 0){
    /* probabilistic decision of in contract and in excess packets */
    bucket_mark = 0;
}
else{
    if (strncmp(mark_type,"deterministic",13) == 0)
        /* determination of in contract and out of contract packets
        by means of token buckets */
        bucket_mark = 1;
    if (op_ima_obj_attr_get(op_id_self(),
        "bucket_size",&bk_size) == OPC_COMPCODE_FAILURE)
        printf("error getting attribute's value\n");
}

if (op_ima_obj_attr_get(op_id_self(),
    "stat type",stat_type) == OPC_COMPCODE_FAILURE)
    printf("error getting attribute's value\n");

if (strncmp(stat_type,"sample",6) == 0){
    average = 0;
}
else{
    if (strncmp(stat_type,"average",7) == 0)
        average = 1;
}

/* Initialisation of the arrays used to store the state and the
statistics of the flows */
for (i = 0; i < MAX_FLOWS; i++){
    fs[i].guar_rate = 0.0;
    fs[i].estim_rate = 0.0;
    fs[i].prev_time = 0.0;
    fs[i].estim_exc_rate = 0.0;
    fs[i].exc_prev_time = 0.0;
    if (bucket_mark){
        fs[i].bk_content = 0.0;
    }
    if (average){
        fs_stat[i].avrg_arrvl = 0.0;
        fs_stat[i].avrg_exc = 0.0;
```

```
        fs_stat[i].sum_arrvl = 0.0;
        fs_stat[i].sum_exc = 0.0;

        fs_stat[i].nb_val_arrvl = 0;
        fs_stat[i].nb_val_exc = 0;
    }
}
if (op_ima_obj_attr_get(op_id_self(),
    "stat time average",&time_average) == OPC_COMPCODE_FAILURE)
    printf("error getting attribute's value\n");
start_averg_time = 0.0;

/* stat file opened in append mode */
fptr = fopen(stat_file,"a");

/* when minimum guarantees are supported, initialisation of these
guarantees */
if (min_support){
    my_self = op_id_self();
    /* Objid of the attribute table where the guarantees are given */
    table = op_id_from_name(my_self,OPC_OBJTYPE_COMP,
        "bw_guarantees");
    nb_lines = op_topo_child_count(table,OPC_OBJTYPE_GENERIC);
    for (i=0;i<nb_lines;i++){
        line = op_topo_child(table,OPC_OBJTYPE_GENERIC,i);
        op_ima_obj_attr_get(line,"VPI",&vpi);
        op_ima_obj_attr_get(line,"rate",&g_rate);
        /* flow_vpi2id is a function that takes a vpi and translate
        it into the corresponding flow identifier */
        fs[flow_vpi2id(vpi)].guar_rate = g_rate;
        if (bucket_mark){
            fs[flow_vpi2id(vpi)].bk_content = bk_size;
        }
        fprintf(fptr,"guarantees;%d;%f\n",flow_vpi2id(vpi),
            g_rate);
    }
}

/*Writing of the process attributes to the stat file*/
fprintf(fptr,"stat type;%d\n",average);
fprintf(fptr,"stat time average;%f\n",time_average);
fprintf(fptr,"estim_const;%f\n",K);
fprintf(fptr,"min_support;%d\n",min_support);
fprintf(fptr,"mark_type;%d\n",bucket_mark);
if (bucket_mark){
    fprintf(fptr,"bucket_size;%f\n",bk_size);
```

```
}
```

C.3.6 ar_estimation state

```
/* Determine the flow to which the packet belongs,
estimate the rate of the flow,
mark the packet with that estimation. */

/* determination of the packet arrival time*/
pk_arrvl_time = op_sim_time();

strm_index = op_intrpt_strm();
pk = op_pk_get(strm_index); /* takes the incoming
                        packet on the stream */

/* packet classification */

/* flow_id is a function that determines the identifier
of a flow by taking in one of its packets */
flow_nb = flow_id (pk);
if (flow_nb>=MAX_FLOWS){
    printf("flow identifier too high\n");
    printf("this problem might be caused by a wrong assignation of vpi\n");
}

if (pk_arrvl_time > start_averg_time + time_average){
    /* computation and writing of the statistics to the stat file */
    for (i = 0; i < MAX_FLOWS; i++){
        if (average){
            if (fs_stat[i].nb_val_arrvl != 0){
                fs_stat[i].avrg_arrvl = fs_stat[i].sum_arrvl /
                    fs_stat[i].nb_val_arrvl;
                fprintf(fptr,"arrvl_rate;%d;%f;%f\n",i,
                    start_averg_time,
                    fs_stat[i].avrg_arrvl);
            }
            if (fs_stat[i].nb_val_exc != 0){
                fs_stat[i].avrg_exc = fs_stat[i].sum_exc /
                    fs_stat[i].nb_val_exc;
                fprintf(fptr,"excess_rate;%d;%f;%f\n",i,
                    start_averg_time,
                    fs_stat[i].avrg_exc);
            }
            fs_stat[i].sum_arrvl = 0.0;
            fs_stat[i].sum_exc = 0.0;

            fs_stat[i].nb_val_arrvl = 0;
```

```
            fs_stat[i].nb_val_exc = 0;
        }
        else{
            if ((fs[i].prev_time > 0.0)
                || (fs[i].estim_rate > 0.0)
                || (fs[i].exc_prev_time > 0.0)
                || (fs[i].estim_exc_rate > 0.0)){
                fprintf(fptr,"arrvl_rate;%d;%f;%f\n",i,
                    pk_arrvl_time,
                    fs[i].estim_rate);
                fprintf(fptr,"excess_rate;%d;%f;%f\n",i,
                    pk_arrvl_time,
                    fs[i].estim_exc_rate);
            }
        }
    }
    start_averg_time += time_average;
}

/* incrementation of the content of the bucket associated to the flow
if bucket defined */
if (bucket_mark){
    incr_bk(flow_nb,pk_arrvl_time);
}
/* estimation of the arrival rate of the flow */
arrvl_rate = estim_flow_rate (flow_nb, pk, pk_arrvl_time);
if (average){
    fs_stat[flow_nb].sum_arrvl += arrvl_rate;
    fs_stat[flow_nb].nb_val_arrvl += 1;
}

/* determination of the marking that has to be done to the packet */
if (!bucket_mark){/* probabilistic marking */
    if (1.0-fs[flow_nb].guar_rate/arrvl_rate > 0.0){
        p_out = 1.0-fs[flow_nb].guar_rate/arrvl_rate;
    }
    else{
        p_out = 0.0;
    }
    if (p_out <= op_dist_uniform(1.0)){
        /* the packet is in the traffic contract */
        arrvl_rate = -10.0;
        /* in theory, arrvl_rate should be set to zero but this
        could lead to some errors due to approximations */
    }
    else{
        /* the packet is in excess */
```

```
        /* the packet has to be marked with the excess rate */
        arrvl_rate = estim_exc_flow_rate (flow_nb, pk, pk_arrvl_time);
        if (average){
            fs_stat[flow_nb].sum_exc += arrvl_rate;
            fs_stat[flow_nb].nb_val_exc += 1;
        }
    }
}
else{/* deterministic marking */
    if (op_pk_bulk_size_get(pk) <= fs[flow_nb].bk_content){
        /* the packet is in the traffic contract */
        fs[flow_nb].bk_content = fs[flow_nb].bk_content -
            op_pk_bulk_size_get(pk);
        arrvl_rate = -10.0;
    }
    else{
        /* the packet is in excess */
        /* the packet has to be marked with the excess rate */
        arrvl_rate = estim_exc_flow_rate (flow_nb, pk, pk_arrvl_time);
        if (average){
            fs_stat[flow_nb].sum_exc += arrvl_rate;
            fs_stat[flow_nb].nb_val_exc += 1;
        }
    }
}

/* marking of the packet */

/* look at the format of the packet to know where to store
the label */
op_pk_format(pk, fmt);
if (strncmp(fmt,"cp_udp_ip",9) == 0){
    if (op_pk_nfd_set(pk, "TOS", arrvl_rate)==
        OPC_COMPCODE_FAILURE)
        printf ("error while labelling packet udp\n");
    }
else{
    if (op_pk_nfd_set(pk, "Infod", arrvl_rate)==
        OPC_COMPCODE_FAILURE)
        printf ("error while labelling packet tcp\n");
    }

op_pk_send(pk, 0);
```

C.4 Core process

The core process is the process used by routers to determine the fair share on one of its outputs links and the dropping probability of each packet. Based on this probability, the packets is dropped or stored in the FIFO queue. The implementation of the queue in this process is inspired from the acb_fifo queue provided by Opnet. The queue service rate is set to the link rate.

C.4.1 state variables

```
/* rate of the output link */
double  \link_rate;

/* estimation of the aggregate arrival rate */
double  \tot_arrvl_rate;

/* fair share of the flows (it is FS in the algorithm) */
double  \fair_share;

/* It is the rate with which the algorithm accepts packets. */
/* accept_rate = som_i (min(r_i,FS)) at some time t         */
double  \accept_rate;

/* arrival time of the last packet */
double  \last_arrvl;

/* indicates if there is congestion or not on the output link */
int \congested;

/* time when the link passes from congested to uncongested  */
/* or time when a new interval is started if the link stays */
/* congested or uncongested longer than K_c seconds         */
double  \start_time;

/* used to compute the new fair share */
double  \temp_fair_share;

/* indicates that the queue is serving a packet */
int \server_busy;

/* indicates whether the packet has been inserted in the queue */
int \insert_ok;

/* Used for the statistics to count the number of packets that are */
/* dropped because flows send at more than their fair share        */
int \excs_drp_cnt;
```

```
/* Amount of bits received since the beginning of the simulation */
int \bits_rcvd;

/* Amount of bits forwarded to the queue since the beginning of the */
/* simulation                                                       */
int \bits_fwrd_qu;

/* Amount of bits dropped because flow sends over fair share, since */
/* the beginning of the simulation                                  */
int \excs_bits_drp;

/* descriptor from the file where the statistics are written */
FILE *  \fptr;

/* window size used for computing the time when the fair share */
/* has to be updated                                          */
double  \K_c;

/* used to compute the estimation of the accepted rate and the */
/* total arrival rate                                          */
double  \K_alpha;

/* Indicates the lenght of the FIFO queue */
int \max_queue_size;

/* Indicates the actual queue size */
int \queue_size;

/* number of packets dropped because the queue overflowed */
int \qu_drop_count;

/* Stores the number of bits dropped because of queue overflow. */
int \qu_bdrp_count;

/* used to store the number of packets in excess dropped for each */
/* flow                                                           */
int \exc_fl_drop[MAX_FLOWS];

/* used to store the number of packets forwarded to the queue */
/* for each flow                                              */
int \fl_fwrd[MAX_FLOWS];

/* used to store the number of bits dropped for each flow */
int \exc_fl_bdrp[MAX_FLOWS];

/* used to store the number of bits forwarded for each flow */
int \fl_bfwrd[MAX_FLOWS];
```

```
/* indicates the amount of bandwidth that is the object of */
/* reservations                                              */
double  \agg_guar;

/* determines the size of the queue under which the link is */
/* considered to be uncongested                              */
int \threshold;

/* used to store the method of the fair share estimation */
/* possible values are :                                 */
/* csfq                                                  */
/* cp_csfq (corresponds to MCSFQ)                        */
/* tail drop                                             */
char    \fair_share_estim[10];

/* indicates if best-effort is used instead of a mechanism that */
/* tries to allocates bandwidth fairly                          */
int \tail_drop;

/* indicates the number of packets dropped by the queue for each */
/* flow                                                          */
int \fl_qdrop[MAX_FLOWS];

/* indicates the number of bits dropped by the queue for each flow */
int \fl_bqdrop[MAX_FLOWS];

/* It is the time on which an average of the statistics will be */
/* performed.                                                   */
double  \time_average;

/* Sum of the fair share values since the start of new interval of */
/* length time_average                                             */
double  \sum_fs;

/* Sum of the accepted rate values since the start of new interval of */
/* length time_average                                                */
double  \sum_ar;

/* Sum of the total arrival rate values since the start of new */
/* interval of length time_average                             */
double  \sum_tar;

/* number of fair share values used in the average computation */
int \nb_val_fs;

/* number of accepted rate values used in the average computation */
```

```
int \nb_val_ar;

/* number of total arrival rate values used in the average */
/* computation                                              */
int \nb_val_tar;

/* average of the fair share on the last time_average seconds */
double  \average_fs;

/* average of the arrival rate on the last time_average seconds */
double  \average_ar;

/* average of the total arrival rate on the last time_average seconds */
double  \average_tar;

/* time at wich this average interval has started */
double  \start_averg_time;

/* indicates if the amount of guaranteed bandwidth has to be */
/* estimated                                                 */
int \agg_estim;

/* indicates the arrival of the previous guaranteed packet */
double  \last_guar_arrvl;

/* Sum of the aggregate guaranteed rate values since the start */
/* of new interval of length time_average                      */
double  \sum_ag;

/* number of aggregate rate values used in the average */
/* computation                                         */
int \nb_val_ag;

/* average of the aggregate guaranteed rate on the */
/* last time_average seconds                       */
double  \average_ag;

/* indicates if the statistic type is average or not. */
int \average;

/* indicates if an upper bound is used to estimate the forwarding */
/* rate when packets are dropped (not by the queue)               */
int \fwrd_upper_bound;

/* it is the upper bound of the accepted rate in case */
/* fwrd_upper_bound is true                           */
double  \accept_upper;
```

```
/* It is the maximum size of the packets passing through the router */
/* This is used when fwrd_upper_bound is true                       */
int \max_pk_size;

/* it is the forwarding rate. When stoica's estimation is used, it  */
/* is the same as the accept_rate. But when the fwrd_upper_bound is */
/* on, it is the minimum of the accept_rate and accept_upper        */
double  \fwrd_rate;

/* indicates the time at the beginning of the simulation after when */
/* the statistics are reset                                         */
double  \warm_up_time;
```

C.4.2 temporary variables

```
double pk_arrvl_time;
double drop_prob;
double label;
Packet *pk;
char fmt[20];
int strm_index;

int pk_len;
double pk_svc_time; /* time at which the packet at the head of the
                       queue is transmitted to the next process model */

int flow_nb; /* used for the statistics only */
```

C.4.3 header block

```
/* To be able to use the exponential function*/
#include <math.h>

#include </home/users/cnodderst/diffserv/routing/routing.h>
#include </home/users/pelsserc/op_models/flowclassify.h>

#define MAX_FLOWS (MAX_ROUTING_VP/2)

/* Define the conditions*/
#define PK_ARRVL (op_intrpt_type() == OPC_INTRPT_STRM)
#define EOS (op_intrpt_type()==OPC_INTRPT_ENDSIM)
#define SVC_COMPLETION ((op_intrpt_type() == OPC_INTRPT_SELF)
    && (op_intrpt_code() == 0))
#define QUEUE_EMPTY (op_q_empty())
#define STAT_RESET ((op_intrpt_type() == OPC_INTRPT_SELF)
    && (op_intrpt_code() == 101277))
```

C.4.4 function block

```
/* gives the maximum of two doubles */
double maximum(double x, double y){
    if (x > y)
        return x;
    else
        return y;
}

/* estimates the rate of a flow but does not set prev_time to
arr_time at the end */
void estim_rate(double *rate, Packet *pk,
    double *prev_time, double arr_time){
    int size;
    double T;
    double expon;
    size = op_pk_bulk_size_get(pk);
    T = arr_time - (*prev_time);
    expon = exp(-(T/(K_alpha)));
    if (T > 0.0){
        (*rate) = (1.0-expon)*((double)size)/T
        + expon*(*rate);
    }
    else{
        (*rate) = ((double)size)/K_alpha + (*rate);
    }
}

/* estimation of the fair share */
void estim_fair_rate(Packet *pk, double arrvl_time,
    int dropped){
    double label;
    char fmt[20];

    if (!agg_estim){
        /* no estimation of the amount of guaranteed bandwidth */
        /* when such estimation is performed, the statistics
        are computed before calling this function */
        if (arrvl_time > start_averg_time + time_average){
            /* computation and writing of the statistics */
            if (average){
                if (nb_val_fs){
                    average_fs = sum_fs / nb_val_fs;
                    fprintf(fptr,"fair_share;%f;%f\n",
                        start_averg_time,
                        average_fs);
```

```
            }
            if (nb_val_ar){
                average_ar = sum_ar / nb_val_ar;
                fprintf(fptr,"accept_rate;%f;%f\n",
                    start_averg_time,
                    average_ar);
            }
            if (nb_val_tar){
                average_tar = sum_tar / nb_val_tar;
                fprintf(fptr,"tot_arrvl_rate;%f;%f\n",
                    start_averg_time,
                    average_tar);
            }
            sum_fs = 0.0;
            sum_ar = 0.0;
            sum_tar = 0.0;

            nb_val_fs = 0;
            nb_val_ar = 0;
            nb_val_tar = 0;
        }
        else{
            fprintf(fptr,"fair_share;%f;%f\n",arrvl_time,fair_share);
            fprintf(fptr,"accept_rate;%f;%f\n",arrvl_time,fwrd_rate);
            fprintf(fptr,"tot_arrvl_rate;%f;%f\n",arrvl_time,tot_arrvl_rate);
        }
        start_averg_time += time_average;
    }
}

/* estimation of the total arrival rate */
estim_rate(&tot_arrvl_rate,pk,&last_arrvl,arrvl_time);
if (average){
    sum_tar += tot_arrvl_rate;
    nb_val_tar += 1;
}

/* estimation of the accepted rate */
if (!dropped){
    estim_rate(&accept_rate,pk,&last_arrvl,arrvl_time);
    if (average){
        sum_ar += accept_rate;
        nb_val_ar += 1;
    }
    fwrd_rate = accept_rate;

    last_arrvl = arrvl_time;
```

```
}
else{
    /* estimation of the accepted rate when the packet is dropped */
    if (!fwrd_upper_bound){
        if (arrvl_time - last_arrvl > 0.0){
            accept_rate = exp(-((arrvl_time-last_arrvl)/(K_alpha)))
                * accept_rate;
            /* because (1.0-exp(-((arrvl_time-last_arrvl)/(K_alpha))))
                * ((double)0)/T = 0*/
            if (average){
                sum_ar += accept_rate;
                nb_val_ar += 1;
            }
        }
        else{
            /* the accept_rate stays the same because accept_rate
            = exp(0) * accept_rate */
            if (average){
                sum_ar += accept_rate;
                nb_val_ar += 1;
            }
        }
        fwrd_rate = accept_rate;

        last_arrvl = arrvl_time;
    }
    else{
        /* fwrd_upper_bound == 1 */
        if (arrvl_time - last_arrvl > 0.0){
            accept_upper = (1.0-exp(-((arrvl_time-last_arrvl)/(K_alpha))))*
                ((double)max_pk_size)/(arrvl_time - last_arrvl) +
                exp(-((arrvl_time-last_arrvl)/(K_alpha))) * accept_rate;
        }
        else{
            accept_upper = ((double)max_pk_size)/(K_alpha) + accept_rate;
        }
        if (accept_upper < accept_rate){
            fwrd_rate = accept_upper;
        }
        else{
            fwrd_rate = accept_rate;
        }
        if (average){
            sum_ar += fwrd_rate;
            nb_val_ar += 1;
        }
        /* note that we do not update last_arrvl in this case */
```

```
    }
}

/* estimation of the fair share */
if (tot_arrvl_rate >= link_rate){
    /* there is congestion */
    if ((!congested) && (queue_size >= threshold)){
        congested = 1;
        start_time = arrvl_time;
    }
    else{
        /* until the queue_size does not overlap the threshold, the fair
        share estimation is done like if there was no congestion */
        if((!congested) && (queue_size < threshold)){
            if (arrvl_time < start_time + K_c){
                op_pk_format(pk,fmt);
                if (strncmp(fmt,"stcp_ip",7)==0){
                    op_pk_nfd_get(pk,"Infod",&label);
                }
                else{
                    op_pk_nfd_get(pk,"TOS",&label);
                }
                temp_fair_share = maximum(temp_fair_share,
                    label);
            }
            else{
                /* the fair share estimation is updated */
                fair_share = temp_fair_share;
                if (average){
                    sum_fs += fair_share;
                    nb_val_fs += 1;
                }

                start_time = arrvl_time;

                if (strncmp(fair_share_estim,"csfq",4) == 0){
                    temp_fair_share = 0.0;
                /* temp_fair_share is not reset for MCSFQ */
                }
            }
        }
        else{
            /* congested == 1 */
            if (arrvl_time > start_time + K_c){
                start_time = arrvl_time;
                if (!agg_estim){
                    if (fwrd_rate != 0.0){
```

```
                        fair_share = fair_share*(link_rate)/fwrd_rate;
                    }
                    else{
                        /* fair_share is set to the available link rate */
                        /* because previous formula would give infinity */

                        fair_share = link_rate;
                    }
                }
                else{
                    if (fwrd_rate - agg_guar > 0.0){
                        fair_share = fair_share*(link_rate - agg_guar) /
                            (fwrd_rate - agg_guar);
                    }
                    else{
                        /* fair_share is set to the available link rate */
                        /* because previous formula would give infinity */

                        fair_share = link_rate - agg_guar;
                    }
                }
                if (!agg_estim){
                    if (fair_share > link_rate){
                        fair_share = link_rate;
                    }
                }
                else{
                    if (fair_share > link_rate - agg_guar){
                        fair_share = link_rate - agg_guar;
                    }
                }
                if (average){
                    sum_fs += fair_share;
                    nb_val_fs += 1;
                }
            }
        }
    }
}
else{ /* tot_arrvl_rate < link_rate */
    /* there is no congestion */
    if (congested){
        congested = 0;
        start_time = arrvl_time;
        if (strncmp(fair_share_estim,"cp_csfq",7) == 0){
            /* MCSFQ */
            temp_fair_share = fair_share; /* used to compute the
```

```
                    new fair share */
            }
            else{
                if (strncmp(fair_share_estim,"csfq",4) == 0){
                    /* CSFQ */
                    temp_fair_share = 0.0;
                }
            }
        }
        else{
            if (arrvl_time < start_time + K_c){
                op_pk_format(pk,fmt);
                if (strncmp(fmt,"stcp_ip",7)==0){
                    op_pk_nfd_get(pk,"Infod",&label);
                }
                else{
                    op_pk_nfd_get(pk,"TOS",&label);
                }
                temp_fair_share = maximum(temp_fair_share,
                    label);
            }
            else{
                fair_share = temp_fair_share;
                if (average){
                    sum_fs += fair_share;
                    nb_val_fs += 1;
                }

                start_time = arrvl_time;

                if (strncmp(fair_share_estim,"csfq",4) == 0){
                    /* temp_fair_share is reset for MCSFQ */
                    temp_fair_share = 0.0;
                }
            }
        }
    }
}

/* performs the writing of the results at the end of the
simulation */
void recordstat(void){
    int i; /* counter */
    double crt_time = op_sim_time();

    if (average){
        if (nb_val_fs){
```

```
        average_fs = sum_fs / nb_val_fs;
        fprintf(fptr,"fair_share;%f;%f\n",start_averg_time,average_fs);
    }
    if (nb_val_ar){
        average_ar = sum_ar / nb_val_ar;
        fprintf(fptr,"accept_rate;%f;%f\n",start_averg_time,average_ar);
    }
    if (nb_val_tar){
        average_tar = sum_tar / nb_val_tar;
        fprintf(fptr,"tot_arrvl_rate;%f;%f\n",start_averg_time,average_tar);
    }
    if (nb_val_ag && agg_estim){
        average_ag = sum_ag / nb_val_ag;
        fprintf(fptr,"agg_guar;%f;%f\n",start_averg_time,agg_guar);
    }
}
else{
    if (crt_time > start_averg_time + time_average){
        fprintf(fptr,"fair_share;%f;%f\n",crt_time,fair_share);
        fprintf(fptr,"accept_rate;%f;%f\n",crt_time,fwrd_rate);
        fprintf(fptr,"tot_arrvl_rate;%f;%f\n",crt_time,tot_arrvl_rate);
        fprintf(fptr,"agg_guar;%f;%f\n",crt_time,agg_guar);
    }
}

/* local statistics for the aggregation of flows */
fprintf(fptr,"nb of excess packets dropped;%d\n",
    excs_drp_cnt);
fprintf(fptr,"nb of bits received;%d\n",bits_rcvd);
fprintf(fptr,"nb of bits forwarded to the queue;%d\n",
    bits_fwrd_qu);
fprintf(fptr,"nb of bits in excess dropped;%d\n",
    excs_bits_drp);
fprintf(fptr,"nb of packets dropped by queue;%d\n",
    qu_drop_count);
fprintf(fptr,"nb of bits dropped by queue;%d\n",
    qu_bdrp_count);
/* per-flow statistics */
for (i=0;i<MAX_FLOWS;i++){
    if ((exc_fl_drop[i]!=0) || (fl_fwrd[i]!=0) || (fl_qdrop[i]!=0)){
        fprintf(fptr,
            "nb of excess packets dropped per flow;%d;%d\n",
            i,
            exc_fl_drop[i]);
        fprintf(fptr,
            "nb of packets forwarded per flow;%d;%d\n",
            i,
```

```
                fl_fwrd[i]);
            fprintf(fptr,
                "nb of packets dropped by queue per flow;%d;%d\n",
                i,
                fl_qdrop[i]);
        }
        if ((exc_fl_bdrp[i]!=0) || (fl_bfwrd[i]!=0) || (fl_bqdrop[i]!=0)){
            fprintf(fptr,
                "nb of excess bits dropped per flow;%d;%d\n",
                i,
                exc_fl_bdrp[i]);
            fprintf(fptr,
                "nb of bits forwarded per flow;%d;%d\n",
                i,
                fl_bfwrd[i]);
            fprintf(fptr,
                "nb of bits dropped by queue per flow;%d;%d\n",
                i,
                fl_bqdrop[i]);
        }
    }
    fprintf(fptr,"end of sim\n");
    if (fclose(fptr) == EOF)
        printf("error occured when closing stat file\n");
}

/* called when the warm-up time is over to reset the statistics */
void reset_stat(){

    int i;

    excs_drp_cnt = 0;
    bits_rcvd = 0;
    bits_fwrd_qu = 0;
    excs_bits_drp = 0;
    qu_drop_count = 0;
    qu_bdrp_count = 0;
    for (i=0;i<MAX_FLOWS;i++){
        exc_fl_drop[i] = 0;
        fl_fwrd[i] = 0;
        fl_qdrop[i] = 0;
        exc_fl_bdrp[i] = 0;
        fl_bfwrd[i] = 0;
        fl_bqdrop[i] = 0;
    }
    printf("Stats have been reset\n");
}
```

C.4.5 init state

```
/* initialisation state */
int i; /* counter */

char stat_file[20]; /* name of the file where the stats are written */
char proc_model[25]; /* name of this process model */
char stat_type[20]; /* type of the stats : average or sample */
char fwrd_estim[20]; /* method used to compute the forwarding rate */

if (op_ima_obj_attr_get(op_id_self(),
        "fair share estimation",
        fair_share_estim) == OPC_COMPCODE_FAILURE)
    printf("error getting attribute's value\n");
if (strncmp(fair_share_estim,"tail drop",9) == 0){
    tail_drop = 1;
}
else{
    tail_drop = 0;
}

if (op_ima_obj_attr_get(op_id_self(),
        "aggr guarantee estim",
        &agg_estim) == OPC_COMPCODE_FAILURE)
    printf("error getting attribute's value\n");

if (op_ima_obj_attr_get(op_id_self(),
        "forwarding rate estim",
        fwrd_estim) == OPC_COMPCODE_FAILURE)
    printf("error getting attribute's value\n");
if (strncmp(fwrd_estim,"upper bound",11) == 0){
    fwrd_upper_bound = 1;
}
else{
    if (strncmp(fwrd_estim,"stoica",6) == 0){
        fwrd_upper_bound = 0;
    }
}

if (op_ima_obj_attr_get(op_id_self(),
        "max pk size",
        &max_pk_size) == OPC_COMPCODE_FAILURE)
    printf("error getting attribute's value\n");

if (op_ima_obj_attr_get(op_id_self(),
        "warm up time",
        &warm_up_time) == OPC_COMPCODE_FAILURE)
```

```
    printf("error getting attribute's value\n");

/* schedules an interruption at the warm-up time to enable
the calling of the reset_stat function */
op_intrpt_schedule_self(warm_up_time,101277);

tot_arrvl_rate = 0.0;
accept_rate = 0.0;
last_arrvl = 0.0;
congested = 0;
start_time = 0.0;
if (op_ima_obj_attr_get(op_id_self(),
        "service rate",
        &link_rate) == OPC_COMPCODE_FAILURE)
    printf("error getting attribute's value\n");
if (op_ima_obj_attr_get(op_id_self(),
        "queue size",
        &max_queue_size) == OPC_COMPCODE_FAILURE)
    printf("error getting attribute's value\n");
if (!tail_drop){
    if (op_ima_obj_attr_get(op_id_self(),
            "window_size",
            &K_c) == OPC_COMPCODE_FAILURE)
        printf("error getting attribute's value\n");
    if (op_ima_obj_attr_get(op_id_self(),
            "estim_const",
            &K_alpha) == OPC_COMPCODE_FAILURE)
        printf("error getting attribute's value\n");
    if (op_ima_obj_attr_get(op_id_self(),
            "queue threshold",
            &threshold) == OPC_COMPCODE_FAILURE)
        printf("error getting attribute's value\n");
}
if (op_ima_obj_attr_get(op_id_self(),
        "process model",
        proc_model) == OPC_COMPCODE_FAILURE)
    printf("error getting attribute's value\n");

if (strncmp(fair_share_estim,"csfq",4) == 0){
    fair_share = 1.0;
}
else{
    if (strncmp(fair_share_estim,"cp_csfq",7) == 0){
        /*fair_share = link_rate/2.0;*/
        fair_share = 1.0;
    }
}
```

```
temp_fair_share = fair_share;
agg_guar = 0.0;
last_guar_arrvl = 0.0;

/* initialization of the variables used
to make averages on the statistics*/
if (op_ima_obj_attr_get(op_id_self(),
            "stat type",
            stat_type) == OPC_COMPCODE_FAILURE)
        printf("error getting attribute's value\n");
if (strncmp(stat_type,"sample",6) == 0){
        average = 0;
    }
else{
    if (strncmp(stat_type,"average",7) == 0){
        average = 1;
    }
}

if (average){
    sum_fs = 0.0;
    sum_ar = 0.0;
    sum_tar = 0.0;
    sum_ag = 0.0;

    nb_val_fs = 0;
    nb_val_ar = 0;
    nb_val_tar = 0;
    nb_val_ag = 0;

    average_fs = 0.0;
    average_ar = 0.0;
    average_tar = 0.0;
    average_ag = 0.0;
}

start_averg_time = 0.0;

if (op_ima_obj_attr_get(op_id_self(),
            "stat time average",
            &time_average) == OPC_COMPCODE_FAILURE)
        printf("error getting attribute's value\n");
/* initially the server is idle*/
server_busy = 0;

/* initially the queue is empty*/
```

```
queue_size = 0;

/* local statistics */
excs_drp_cnt = 0;
bits_rcvd = 0;
bits_fwrd_qu = 0;
excs_bits_drp = 0;
qu_drop_count = 0;
qu_bdrp_count = 0;
for (i=0;i<MAX_FLOWS;i++){
    exc_fl_drop[i] = 0;
    fl_fwrd[i] = 0;
    fl_qdrop[i] = 0;
    exc_fl_bdrp[i] = 0;
    fl_bfwrd[i] = 0;
    fl_bqdrop[i] = 0;
}

/* file descriptor to collect the statistics */
if (op_ima_obj_attr_get(op_id_self(),
        "stat file",
        stat_file) == OPC_COMPCODE_FAILURE)
    printf("error getting attribute's value\n");

/* stat file opened in append mode */
fptr = fopen(stat_file,"a");
/*Writing of the process attributes to the stat file*/
fprintf(fptr,"process_model;%s\n",proc_model);
fprintf(fptr,"fair_share_estim;%s\n",fair_share_estim);
fprintf(fptr,"aggr_guarantee_estim;%d\n",agg_estim);
fprintf(fptr,"service rate;%f\n",link_rate);
fprintf(fptr,"max queue size;%d\n",max_queue_size);
fprintf(fptr,"stat time average;%f\n",time_average);
fprintf(fptr,"stat type;%d\n",average);
fprintf(fptr,"fwrd rate estim;%s\n",fwrd_estim);
fprintf(fptr,"max pk size;%d\n",max_pk_size);
fprintf(fptr,"warm up time;%f\n",warm_up_time);
if (!tail_drop){
    fprintf(fptr,"queue_threshold;%d\n",threshold);
    fprintf(fptr,"window_size;%f\n",K_c);
    fprintf(fptr,"estim_const;%f\n",K_alpha);
}

/*Writing of the initial values to stat file*/
fprintf(fptr,"init fair_share;%f\n",fair_share);
fprintf(fptr,"init tot_arrvl_rate;%f\n",tot_arrvl_rate);
fprintf(fptr,"init accept_rate;%f\n",accept_rate);
```

```
fprintf(fptr,"init last_arrvl;%f\n",last_arrvl);
fprintf(fptr,"init congested;%f\n",congested);
fprintf(fptr,"init start_time;%f\n",start_time);
fprintf(fptr,"init temp_fair_share;%f\n",temp_fair_share);
fprintf(fptr,"init agg_guar;%f\n",agg_guar);
fprintf(fptr,"init last_guar_arrvl;%f\n",last_guar_arrvl);
```

C.4.6 forward state

```
/* Estimate the aggregate arrival rate,
compute drop probability,
decide to drop the packet or to forward it,
store the packet in the queue,
transmit the packets to the next process model */

pk_arrvl_time = op_sim_time();
strm_index = op_intrpt_strm();
pk = op_pk_get(strm_index);

/*used for the statistics only*/
flow_nb = flow_id(pk);
if (flow_nb >= MAX_FLOWS){
    printf("flow identifier too big\n");
}

bits_rcvd = bits_rcvd + op_pk_bulk_size_get(pk);

if (!tail_drop){
    op_pk_format(pk, fmt);
    if (strncmp(fmt,"stcp_ip",7)==0){
        op_pk_nfd_get(pk,"Infod",&label);
    }
    else{
        op_pk_nfd_get(pk,"TOS",&label);
    }
    if (label > 0.0){
        /* packet in excess */
        drop_prob = (maximum(0,1-(fair_share/label)));
    }
    else{
        /* guaranteed packet */
        drop_prob = -10.0;
    }
    if (drop_prob > op_dist_uniform(1.0)){
        /* packet is discarded */
        /* statistics gathering*/
        exc_fl_drop[flow_nb] = exc_fl_drop[flow_nb] + 1;
```

```
exc_fl_bdrp[flow_nb] = exc_fl_bdrp[flow_nb]
    + op_pk_bulk_size_get(pk);

excs_drp_cnt  = excs_drp_cnt + 1;

excs_bits_drp = excs_bits_drp + op_pk_bulk_size_get(pk);

if (agg_estim){
    /*dealing of the statistics*/
    if (pk_arrvl_time > start_averg_time + time_average){
        if (average){
            if (nb_val_fs){
                average_fs = sum_fs / nb_val_fs;
                fprintf(fptr,
                    "fair_share;%f;%f\n",
                    start_averg_time,
                    average_fs);
            }
            if (nb_val_ar){
                average_ar = sum_ar / nb_val_ar;
                fprintf(fptr,
                    "accept_rate;%f;%f\n",
                    start_averg_time,
                    average_ar);
            }
            if (nb_val_tar){
                average_tar = sum_tar / nb_val_tar;
                fprintf(fptr,
                    "tot_arrvl_rate;%f;%f\n",
                    start_averg_time,
                    average_tar);
            }
            if (nb_val_ag){
                average_ag = sum_ag / nb_val_ag;
                    fprintf(fptr,
                    "agg_guar;%f;%f\n",
                    start_averg_time,
                    average_ag);
            }
            sum_fs = 0.0;
            sum_ar = 0.0;
            sum_tar = 0.0;
            sum_ag = 0.0;

            nb_val_fs = 0;
            nb_val_ar = 0;
            nb_val_tar = 0;
```

```
                nb_val_ag = 0;
            }
            else{
                fprintf(fptr,
                    "fair_share;%f;%f\n",
                    pk_arrvl_time,
                    fair_share);
                fprintf(fptr,
                    "accept_rate;%f;%f\n",
                    pk_arrvl_time,
                    fwrd_rate);
                fprintf(fptr,
                    "tot_arrvl_rate;%f;%f\n",
                    pk_arrvl_time,
                    tot_arrvl_rate);
                fprintf(fptr,
                    "agg_guar;%f;%f\n",
                    pk_arrvl_time,
                    agg_guar);
            }
            start_averg_time += time_average;
        }
        agg_guar =
            exp(-((pk_arrvl_time-last_guar_arrvl)/(K_alpha)))
            * agg_guar;
        /* because the packet is not guaranteed and therefore
            (1.0-exp(-((pk_arrvl_time-last_guar_arrvl)/(K_alpha))))
            * ((double)0)/T
            = 0*/
        last_guar_arrvl = pk_arrvl_time;
        if (average){
            sum_ag += agg_guar;
            nb_val_ag += 1;
        }
    }
    estim_fair_rate(pk,pk_arrvl_time,1);

    if (strncmp(fmt,"stcp_ip",7)==0){
        op_stcp_discard_packet(pk);
    }
    else {
        op_pk_destroy(pk);
    }
    insert_ok = 0;
}
else{
    /* packet is forwarded */
```

```
fl_fwrd[flow_nb] = fl_fwrd[flow_nb] + 1;
fl_bfwrd[flow_nb] = fl_bfwrd[flow_nb] + op_pk_bulk_size_get(pk);

bits_fwrd_qu = bits_fwrd_qu + op_pk_bulk_size_get(pk);

/* relabel the packet if needed */
if (drop_prob > 0.0){
    if (strncmp(fmt,"stcp_ip",7)==0){
        op_pk_nfd_set(pk,"Infod",&fair_share);
    }
    else{
        op_pk_nfd_set(pk,"TOS",&fair_share);
    }
}

/* estimate the fair share */
if (agg_estim){
    /*dealing of the statistics*/
    if (pk_arrvl_time > start_averg_time + time_average){
        if (average){
            if (nb_val_fs){
                average_fs = sum_fs / nb_val_fs;
                fprintf(fptr,
                    "fair_share;%f;%f\n",
                    start_averg_time,
                    average_fs);
            }
            if (nb_val_ar){
                average_ar = sum_ar / nb_val_ar;
                fprintf(fptr,
                    "accept_rate;%f;%f\n",
                    start_averg_time,
                    average_ar);
            }
            if (nb_val_tar){
                average_tar = sum_tar / nb_val_tar;
                fprintf(fptr,
                    "tot_arrvl_rate;%f;%f\n",
                    start_averg_time,
                    average_tar);
            }
            if (nb_val_ag){
                average_ag = sum_ag / nb_val_ag;
                fprintf(fptr,
                    "agg_guar;%f;%f\n",
                    start_averg_time,
```

```
                average_ag);
        }
        sum_fs = 0.0;
        sum_ar = 0.0;
        sum_tar = 0.0;
        sum_ag = 0.0;

        nb_val_fs = 0;
        nb_val_ar = 0;
        nb_val_tar = 0;
        nb_val_ag = 0;
    }
    else{
        fprintf(fptr,
            "fair_share;%f;%f\n",
            pk_arrvl_time,
            fair_share);
        fprintf(fptr,
            "accept_rate;%f;%f\n",
            pk_arrvl_time,
            fwrd_rate);
        fprintf(fptr,
            "tot_arrvl_rate;%f;%f\n",
            pk_arrvl_time,
            tot_arrvl_rate);
        fprintf(fptr,
            "agg_guar;%f;%f\n",
            pk_arrvl_time,
            agg_guar);
    }

    start_averg_time += time_average;
}

if (label > 0.0){
    /* packet in excess */
    agg_guar =
        exp(-((pk_arrvl_time-last_guar_arrvl)/(K_alpha)))
        * agg_guar;
    /* because
        (1.0-exp(-((pk_arrvl_time-last_guar_arrvl)/(K_alpha))))
        * ((double)0)/T
        = 0*/
    if (average){
        sum_ag += agg_guar;
        nb_val_ag += 1;
    }
```

```
        estim_fair_rate(pk,pk_arrvl_time,0);
    }
    else{
        /*estimate the aggregate guarantee*/
        if (pk_arrvl_time-last_guar_arrvl > 0.0){
            agg_guar =
                exp(-((pk_arrvl_time-last_guar_arrvl)/(K_alpha)))
                    * agg_guar
                + (1.0-exp(-((pk_arrvl_time-last_guar_arrvl)
                        / (K_alpha))))
                    * ((double)op_pk_bulk_size_get(pk))
                    / (pk_arrvl_time-last_guar_arrvl);
            if (average){
                sum_ag += agg_guar;
                nb_val_ag += 1;
            }
        }
        else{
            agg_guar = ((double)op_pk_bulk_size_get(pk))/K_alpha
                + agg_guar;
            if (average){
                sum_ag += agg_guar;
                nb_val_ag += 1;
            }
        }
        estim_fair_rate(pk,pk_arrvl_time,0);
    }
    last_guar_arrvl = pk_arrvl_time;
}
else{
    estim_fair_rate(pk,pk_arrvl_time,0);
}

/* attempt to enqueue the packet at tail    */
/* of subqueue 0*/
if (queue_size + op_pk_bulk_size_get(pk) <= max_queue_size){
    /* packet is inserted into the queue */
    op_subq_pk_insert (0, pk, OPC_QPOS_TAIL);
    /* insertion was successful             */
    insert_ok = 1;
    /* the queue is increased by the packet length */
    queue_size = queue_size + op_pk_bulk_size_get(pk);
}
else{
    /* the queue is full
    the packet is discarded
    statistic is updated*/
```

```
            qu_drop_count = qu_drop_count + 1;
            qu_bdrp_count = qu_bdrp_count + op_pk_bulk_size_get(pk);

            fl_qdrop[flow_nb] = fl_qdrop[flow_nb]+1;
            fl_bqdrop[flow_nb] = fl_bqdrop[flow_nb]+op_pk_bulk_size_get(pk);

            if (strncmp(fmt,"stcp_ip",7)==0){
                op_stcp_discard_packet(pk);
            }
            else{
                op_pk_destroy (pk);
            }
            /* set flag indicating insertion fail   */
            /* this flag is used to determine       */
            /* transition out of this state         */
            insert_ok = 0;
            /* packet is dropped by queue */
        }
    }
}
else{
    /* tail drop is the only buffer acceptance mechanism */
    fl_fwrd[flow_nb] = fl_fwrd[flow_nb]+1;
    fl_bfwrd[flow_nb] = fl_bfwrd[flow_nb]+op_pk_bulk_size_get(pk);

    bits_fwrd_qu = bits_fwrd_qu + op_pk_bulk_size_get(pk);

    /* attempt to enqueue the packet at tail    */
    /* of subqueue 0*/
    if (queue_size + op_pk_bulk_size_get(pk) <= max_queue_size){
        /* packet is inserted into the queue */
        op_subq_pk_insert (0, pk, OPC_QPOS_TAIL);
        /* insertion was successful             */
        insert_ok = 1;
        /* the queue is increased by the packet length */
        queue_size = queue_size + op_pk_bulk_size_get(pk);
    }
    else{
        /* the queue is full
        the packet is discarded
        statistic is updated*/
        qu_drop_count = qu_drop_count + 1;
        qu_bdrp_count = qu_bdrp_count + op_pk_bulk_size_get(pk);
        fl_qdrop[flow_nb] = fl_qdrop[flow_nb]+1;
        fl_bqdrop[flow_nb] = fl_bqdrop[flow_nb]+op_pk_bulk_size_get(pk);

        if (strncmp(fmt,"stcp_ip",7)==0){
```

```
            op_stcp_discard_packet(pk);
        }
        else{
            op_pk_destroy (pk);
        }
        /* set flag indicating insertion fail   */
        /* this flag is used to determine       */
        /* transition out of this state         */
        insert_ok = 0;
        /* packet dropped by queue */
    }
}
```

C.4.7 svc_start state

```
/* get a handle on packet at head of subqueue 0 */
/* (this does not remove the packet)            */
pk = op_subq_pk_access (0, OPC_QPOS_HEAD);

/* determine the packets length (in bits)       */
pk_len = op_pk_bulk_size_get (pk);

/* determine the time required to complete      */
/* service of the packet                        */
pk_svc_time = (double)(pk_len) / link_rate;

/* schedule an interrupt for this process       */
/* at the time where service ends.              */
op_intrpt_schedule_self (op_sim_time () + pk_svc_time, 0);

/* the server is now busy.                      */
server_busy = 1;
```

C.4.8 svc_complete state

```
/* extract packet at head of queue; this    */
/* is the packet just finishing service     */
pk = op_subq_pk_remove (0, OPC_QPOS_HEAD);

queue_size = queue_size - op_pk_bulk_size_get(pk);
/* forward the packet on stream 0, causing  */
/* an immediate interrupt at destination.   */
op_pk_send_forced(pk, 0);

/* server is idle again.                    */
server_busy = 0;
```