Chapitre 3

Spécification formelle des mécanismes de support des qualités de service dans l'Internet

Guy Leduc et Ludovic Kuty[1]

Résumé : Ce chapitre illustre comment le langage ISO E-LOTOS peut être utilisé pour modéliser les différents mécanismes en cours de normalisation à l'IETF (Internet Engineering Task Force) et visant à assurer une meilleure qualité de service dans les réseaux informatiques basés sur la pile de protocoles TCP/UDP/IP. La richesse de l'architecture Diff-Serv (Services Différenciés) de l'IETF nous permet de montrer de nombreuses facettes du langage E-LOTOS et ses aptitudes à modéliser de façon simple les algorithmes les plus répandus de gestion des files d'attente (RED, RIO, WRED), d'ordonnancement de paquets (WFQ), de régulation et de contrôle de trafic (Token Bucket). Le langage permet aussi d'assembler aisément ces mécanismes de base pour former une architecture cohérente.

Mots-clés : Internet, Qualité de service, Services différenciés, Gestion des files d'attente, Ordonnancement, Régulation de trafic, Contrôle de trafic, E-LOTOS, Spécification.

1. Université de Liège, Institut Montefiore, RUN, B28, B-4000 Liège, Belgique, courriel : leduc@montefiore.ulg.ac.be

3.1. Introduction

L'architecture initiale de l'Internet ne permettait guère de satisfaire les besoins de qualité de service de certaines classes d'applications. Aujourd'hui, des architectures comme les services intégrés et différenciés tentent de rencontrer ces exigences par l'ajout de protocoles et mécanismes spécifiques. Après avoir exposé les concepts centraux liés aux qualités de service dans ces nouvelles architectures, ce chapitre exposera la manière de les décrire avec précision dans le langage de spécification ISO E-LOTOS [ISO]. Des exemples de mécanismes de gestion de files d'attente, d'ordonnancement, de décongestion, de régulation de trafic, de vérification de la conformité du trafic et de contrôle d'admission seront traités. L'approche des algèbres de processus temporisées qui sous-tend E-LOTOS nous permet de factoriser les problèmes et de décrire un système complexe de gestion de qualité de service par la composition de ces mécanismes de base.

Avec la transformation de l'Internet en une infrastructure commerciale, il est de plus en plus évident que plusieurs classes de service soient nécessaires pour satisfaire différents types d'applications. Le simple service du meilleur effort (best-effort) disponible aujourd'hui n'est pas suffisant pour les entreprises qui sont prêtes à payer pour une meilleure qualité de service qui rendrait possible le déploiement d'applications de téléphonie et de visioconférence, ou permettrait déjà d'améliorer les temps d'accès à des serveurs.

Il n'est pas évident a priori que des mécanismes spécifiques soient nécessaires pour améliorer la qualité de service (QdS). Certains soutiennent que l'augmentation des débits des lignes et de la capacité de commutation des routeurs peut suffire. D'autres toutefois pensent que ce type d'augmentation ne résout pas le problème. D'une part, parce que de nouvelles applications viendront probablement consommer ces ressources, et surtout parce que les problèmes de congestion sont par nature indépendants du facteur d'échelle auquel on les considère. En effet, un routeur dont deux lignes d'entrée à 1 Gbps dirigent ponctuellement leur trafic vers une ligne de sortie de 1 Gbps, subit une congestion semblable à un routeur dont deux lignes d'entrée à 1 kbps dirigent leur trafic vers une ligne à 1 kbps.

L'IETF (Internet Engineering Task Force) a proposé plusieurs modèles et architectures répondant au besoin de QdS [XN 99]. Citons notamment les services intégrés (IntServ) [BCS94], les services différenciés (DiffServ) [BLA98], la commutation de labels multiprotocoles (MPLS) [RVC 99, CDF+ 99], l'ingénierie de trafic [ACE+ 00] et le routage sujet aux contraintes [AMA+ 99]. Dans ce chapitre, nous décrirons brièvement IntServ, puis nous nous concentrerons sur DiffServ.

3.2. Deux architectures pour la qualité de service

3.2.1. *Les services intégrés*

Ce modèle, semblable à la philosophie ATM [deP 93], est basé sur la réservation de ressources (CPU et mémoire) dans les routeurs de façon à fournir à chaque flux de données des QdS spécifiques.

Plusieurs mécanismes doivent être mis en œuvre dans IntServ :

- Le protocole RSVP (Resource ReSerVation Protocol) [BRAD 97] est le protocole de signalisation qui permet aux applications de réserver dans les routeurs des ressources pour chaque flux (point-à-point ou multipoints). On réserve essentiellement du débit pour le flux, en configurant les ordonnanceurs des routeurs. Le débit réservé peut aussi permettre de garantir un délai maximum à condition que la source respecte une certaine enveloppe de trafic (Cf. le régulateur ci-dessous) [PG 93, PG 94].
- Le contrôle d'admission décide ou non d'accepter une demande de ressource pour un flux en fonction des ressources disponibles.
- La classification multichamps des paquets reconnaît le flux à partir d'un ou plusieurs champs de ceux-ci (p.ex. adresse IP, numéro de port, identifiant du protocole, …), afin de lui appliquer le traitement adéquat via l'ordonnanceur, notamment en le plaçant dans la file d'attente de ce flux.
- Un ordonnanceur (par ligne de sortie du routeur) décide de l'ordre de transmission des paquets en attente vers cette sortie en fonction des QdS demandées.
- Un régulateur de trafic (*shaper* en anglais) permet à une source de respecter une certaine enveloppe de trafic. Celle-ci est définie principalement par son débit de pointe, son débit moyen et la taille de sa plus grande rafale au débit de pointe. Un régulateur peut aussi être utilisé à la sortie d'un routeur pour remettre en forme le trafic avant que celui-ci ne transite dans un autre domaine.
- Un contrôleur de trafic (*policer* en anglais) permet de vérifier que l'enveloppe de trafic est respectée à l'entrée d'un domaine.

Le problème principal d'IntServ est son passage à l'échelle lorsque les flux sont très nombreux. Ceci est dû au fait que chaque routeur doit tenir à jour de l'information pour chaque flux qui le traverse et doit en principe allouer une file d'attente par flux, sans parler de l'ordonnanceur qui doit gérer toutes ces files. De plus, chaque routeur, y compris au cœur du réseau, doit au moins disposer des quatre premiers mécanismes décrits ci-dessus. Enfin, le déploiement d'IntServ n'est possible que si tous les routeurs disposent des mécanismes ad hoc, sans quoi la QdS

n'est pas offerte de bout-en-bout. Pour ces raisons, l'IETF a ensuite proposé l'architecture DiffServ.

3.2.2. Les services différenciés

La meilleure façon de résoudre le problème du passage à l'échelle rencontré par IntServ consiste à ne plus traiter les flux de données individuellement, mais de les regrouper en classes qui aggrégeront un ensemble de flux demandant le même type de QdS. Dans ce modèle, la source, ou plus logiquement son routeur local comme nous le verrons par la suite, devra estampiller ses paquets avec l'identifiant de la classe de service souhaitée. Six bits du champ TOS (Type of Service), rebaptisé DS (DiffServ), sont prévus à cet effet dans l'en-tête des paquets IPv4; ce qui permet potentiellement d'identifier 64 services différents. Ces six bits constituent le DSCP (DiffServ Code Point) [NIC 98]. L'en-tête des paquets IPv6 prévoit également 6 bits à cet effet.

Le modèle DiffServ correspond aussi très bien au modèle commercial actuel où une entreprise qui désire recevoir des services différenciés de son fournisseur de service Internet (ISP) négociera avec celui-ci un contrat (SLA = Service Level Agreement) portant globalement sur la quantité de trafic qui pourra être échangée dans chaque classe de service disponible chez cet ISP. Il est en effet classique qu'une organisation paie pour un débit global à l'accès, plutôt que pour chaque flux de données. Cela facilite aussi grandement la facturation. Ce mode de fonctionnement est caractérisé par des SLA statiques négociés sur une base mensuelle ou annuelle. Il est toutefois également concevable de rendre ces SLA dynamiques à condition d'utiliser un protocole de signalisation (p.ex. RSVP) pour obtenir des QdS à la demande. Dans notre contexte, la partie la plus importante du SLA est le SLS (Service Level Specification) qui définit le trafic échangé à l'interface avec l'ISP. Il comprend typiquement une description de l'enveloppe de trafic admise (débit de pointe, débit moyen, plus grande rafale, …). C'est sur cette base que l'ISP vérifiera si le client respecte son contrat et, le cas échéant, détruira ou dégradera les paquets non conformes.

Le modèle DiffServ définit trois types de services principaux : (a) le service Premium, (b) le service assuré, et bien sûr (c) le simple service d'interconnectivité (best-effort). Le service Premium est prévu pour les applications temps réel qui requièrent un délai faible et une gigue de délai faible. Citons par exemple la téléphonie ou encore la visioconférence. Le service assuré est censé apporter aux applications non temps réel un meilleur service que le "Best Effort", notamment par un taux de perte plus réduit et des délais plus faibles. Le service assuré est lui-même divisé en quatre sous-catégories dont les sémantiques peuvent être définies localement à chaque domaine. On pourrait imaginer par exemple que les sous-

catégories se distinguent par leurs taux de perte ou leurs délais, absolus ou relatifs. Chaque sous-catégorie dispose, en outre, de trois niveaux de précédence permettant de relativiser l'importance des paquets en cas de congestion. Ceci nous conduit finalement à 12 services assurés distincts.

Pour remplir ces services, les routeurs doivent réaliser des comportements définis, appelés PHB (Per-Hop Behaviour). Ainsi, le service Premium est réalisé par le PHB EF (Expedited Forwarding) [JNP 99], tandis que les services assurés le sont par le groupe de PHB AF (Assured Forwarding) [HBW+ 99]. Le PHB BE (Best-Effort) complète ceux-ci. Nous reviendrons plus loin sur les mécanismes à mettre en œuvre dans chacun de ces PHB, mais notons que l'ordre des paquets est censé être respecté localement par le PHB EF et les quatre PHB AF.

Pour mieux comprendre le fonctionnement de DiffServ, nous allons considérer un scénario typique de communication. La figure 3.1 montre deux domaines périphériques interconnectés par un ISP. Aux deux interfaces de l'ISP, des SLA sont normalement établis. Pour simplifier, nous supposerons que le réseau de l'ISP ne comporte que trois routeurs : un routeur de cœur de réseau (*CR* = core router) et deux routeurs de frontière (*BR* = Border Router). Pour un flux dont l'origine est dans le domaine de gauche et la destination dans celui de droite, les routeurs de frontière *IBR* (Ingress Border Router) et *EBR* (Egress Border Router) constituent respectivement les points d'entrée et de sortie de l'ISP. Les réseaux des deux organisations seront aussi simplifiés. On y trouvera essentiellement une station émettrice *S* (ou réceptrice *D*), le routeur R_S (ou R_D) le plus proche de cette station sur le chemin, et le routeur de frontière (*ER* = Edge Router) qui est connecté à l'ISP.

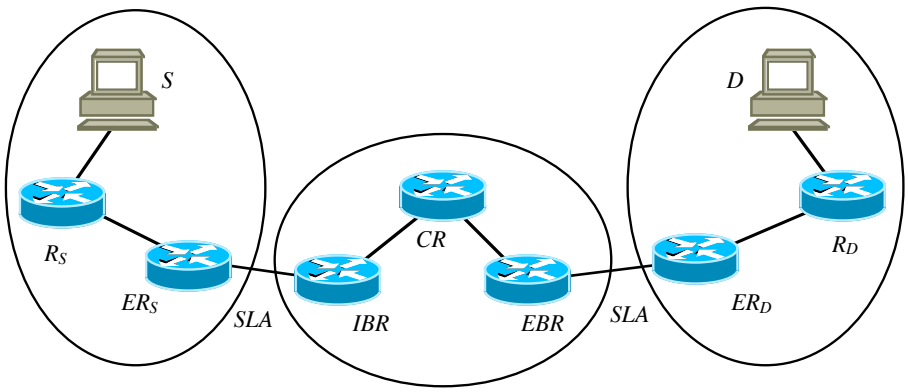

Figure 3.1. *Architecture typique de réseau*

Nous ne décrirons que les mécanismes mis en œuvre pendant la transmission de données, mais nous ferons allusion à la signalisation lorsque nous le jugerons opportun. La source *S*, qui souhaite envoyer des données à *D* via une classe de service DiffServ, peut estampiller elle-même ses paquets avec le DSCP adéquat. En pratique, il faudra veiller à ce que cette application dispose bien du droit d'usage de cette classe de service sans quoi toutes les applications pourraient être tentées d'utiliser les meilleures classes, ce qui conduirait à une rapide saturation de celles-ci, et à un phénomène de famine préjudiciable aux applications qui ont de réels besoins de QdS. Pour cette raison, il peut être plus commode de laisser le premier routeur R_S se charger du marquage des paquets, à condition qu'il soit informé, par un contrôleur d'admission, de l'autorisation de marquer ce flux. Ceci exige donc une signalisation (p.ex. avec RSVP) entre la source *S*, le contrôleur d'admission (non représenté à la figure 3.1) et le routeur R_S. Il faut aussi caractériser le flux par un ensemble de champs du paquet. Ceci rejoint la technique de classification multichamps déjà décrite dans le contexte d'IntServ. La différence ici est que le premier routeur R_S sera le seul à effectuer cette classification, alors que dans IntServ, chaque routeur la reproduisait. Dans DiffServ, les autres routeurs peuvent faire l'économie d'une classification en se basant simplement sur le DSCP attribué par le premier routeur. Le premier routeur placera aussi les paquets dans des files d'attente selon les classes de service. Nous aurons typiquement une file pour EF, quatre files pour AF_1, AF_2, AF_3 et AF_4 respectivement, et une file pour BE. Nous reviendrons plus loin sur la gestion de ces files en cas de congestion et sur les techniques d'ordonnancement de celles-ci. Selon la classe de service, le routeur R_S peut aussi se charger d'une régulation de flux. Ceci est très important pour la classe Premium, car le PHB EF ne peut assurer de faibles (gigues de) délais que si le trafic de la source est très régulier (p.ex. rafales très courtes). Une autre forme de contrôle d'admission peut aussi être utile pour la classe Premium (entre autres). Si nous supposons qu'une fraction des ressources de l'ISP est dédicacée à cette classe, et que chaque flux spécifie son débit, ce contrôle d'admission aura pour but de vérifier que la classe Premium dispose des réserves suffisantes. A cet égard, il est nécessaire de vérifier qu'il en est ainsi dans le domaine local, mais aussi dans l'ISP et dans le domaine d'arrivée. Ceci nécessite un protocole de signalisation (p.ex. RSVP) entre la source et un courtier en bande passante de son domaine, ainsi qu'entre courtiers de domaines voisins.

Le routeur de frontière ER_S ne se base que sur les DSCP des paquets pour déterminer les politiques de QdS et de gestion des files d'attente. Comme R_S, il peut (re)mettre en forme les trafics de certaines classes avant que ceux-ci n'entrent dans l'ISP, sans quoi les paquets non conformes risqueraient d'être dégradés ou détruits.

Insistons sur le fait qu'ici, la régulation est réalisée sur le trafic agrégé d'une classe et non sur des flux individuels.

Le routeur de frontière *IBR* se comporte comme ER_S sauf que la fonction de régulation de trafic est remplacée par la fonction duale de contrôle de trafic (*policing* en anglais). En effet, *IBR* est le gardien de la SLS définie à cette interface. Si aucun SLA n'a été défini, il attribuera le DSCP 0 (= BE) à tous les paquets entrants. Si un SLS existe, son rôle consistera à vérifier que le client le respecte. Par exemple, pour la classe Premium, il détruira tous les paquets non conformes. Pour les classes de service assuré, il dégradera les paquets non conformes en les marquant explicitement comme tel. Nous avons vu que chaque classe AF_i dispose de trois niveaux de précédence qui peuvent être utilisés à cet effet. Les DSCP des paquets conformes sont laissés tels quels (p.ex. de couleur verte : $AF_{i,1}$), alors que les autres seraient marqués $AF_{i,2}$ (couleur jaune) ou $AF_{i,3}$ (couleur rouge) selon la gravité.

Les routeurs de cœur de réseau ne s'occupent ni de classification, ni de mise en forme, ni de contrôle de trafic. Ils peuvent ainsi dédier l'essentiel de leurs ressources à la commutation, à la gestion et à l'ordonnancement des files d'attente.

Le routeur de frontière *EBR* est comparable au routeur de frontière ER_S. Enfin, par ses fonctions, le routeur R_D est comparable à un routeur de cœur de réseau.

Notons encore qu'il est concevable de faire coexister les modèles IntServ et DiffServ, notamment en utilisant IntServ dans les réseaux périphériques où le nombre de flux est plus restreint, et DiffServ dans le réseau dorsal. Dans ce cas, la classification, qui était opérée par le routeur R_S, serait transférée au routeur ER_S. De même, le routeur ER_D démultiplexerait les flux agrégés venant du réseau dorsal en flux individuels qui disposeront alors de ressources et de QdS dédicacées.

3.3. Mécanismes de support des qualités de service

Dans la section 3.2, nous avons évoqué les mécanismes principaux de support de la qualité de service que sont la gestion des files d'attente en cas de congestion, la régulation et le contrôle de trafic, et l'ordonnancement des paquets. Le but de cette section est de décrire et formaliser les algorithmes classiques utilisés pour réaliser ces fonctions.

3.3.1. Gestion des files d'attente

Nous commencerons par la description d'un ensemble de files d'attente muni d'une politique de décongestion, c'est-à-dire une politique de destruction ou de marquage de paquets en cas de congestion.

Le système le plus simple est un ensemble de files qui partagent une même zone de mémoire de taille fixée, et où la politique de décongestion se réduit au strict minimum : on stocke les paquets tant qu'il y a de la place en mémoire commune, sinon on les détruit. Cette politique de trop-plein (ou *drop tail* en anglais) est la plus répandue. Une alternative presqu'aussi simple serait de fixer une taille limite à chaque file et d'appliquer la politique de trop-plein à chacune d'elle.

3.3.1.1. RED

Toutefois, il existe de nombreuses autres politiques. Une des plus connues est RED (Random Early Detection) [FJ93]. Comme le trop-plein, RED peut s'appliquer à une file ou à un ensemble de files. Considérons tout d'abord le cas d'une file.

RED, contrairement au trop-plein, permet d'anticiper les congestions, en contrôlant la taille moyenne de la file d'attente (*avg*) et en n'attendant pas qu'une file soit pleine pour pénaliser (c'est-à-dire marquer ou jeter) les paquets entrants. Le système consiste à pénaliser avec une certaine probabilité les paquets qui font trop augmenter avg. Ainsi, la source est avertie suffisamment tôt d'un risque de congestion et peut ralentir son débit. Cette manière de procéder permet aux sources qui s'asservissent (p.ex. TCP) de réduire le délai moyen subi par leurs paquets tout en conservant un débit élevé. Elle assure aussi une meilleure équité entre les sources et empêche le phénomène de synchronisation de celles-ci.

Plus précisément, le type de comportement adopté par l'algorithme lors de la réception d'un paquet dépend de la valeur de *avg* à cet instant par rapport aux deux seuils min_{th} et max_{th} dont l'illustration est donnée à la figure 3.2. Le paramètre max_{qlen} est le nombre maximal de paquets que la file peut contenir et p_b est la probabilité que le paquet soit pénalisé. On distingue trois cas :

- Si $avg < min_{th}$, $p_b := 0$.
- Si $min_{th} < avg < max_{th}$, la probabilité p_b se calcule par extrapolation linéaire d'après la valeur de *avg* à l'aide de la formule $max_p \frac{avg - min_{th}}{max_{th} - min_{th}}$ où max_p est la probabilité maximale atteinte lorsque *avg* vaut max_{th}. Si la taille de la file est mesurée en octets (et non en paquets), il est nécessaire d'ajuster p_b, pour garantir que la probabilité de pénalité soit proportionnelle à la taille du paquet, en faisant $p_b := p_b \frac{PacketSize}{MaxPacketSize}$.
- Si $avg > max_{th}$, $p_b := 1$.

La taille moyenne de la file est calculée à l'aide de la formule $(1 - w_q)\ avg + w_q\ q$ où q représente la taille courante de la file et w_q le poids qui détermine l'influence d'une modification de q sur *avg*. Plus w_q est grand, plus les changements de valeur

de q influeront sur la valeur de avg. Toutefois, si w_q est trop grand, les congestions momentanées et les courtes rafales de paquets auront une trop grande importance sur la taille moyenne de la file. Dans le cas contraire, avg ne convergera pas assez rapidement vers la taille moyenne réelle de la file d'attente.

Il faut noter que ce sont les arrivées de paquets qui provoquent des mises à jour de la taille moyenne de la file, rythmant ainsi l'évolution de avg. C'est pourquoi lors de la réception d'un paquet dans une file vide, il est nécessaire d'ajuster la valeur de la taille moyenne de la file en tenant compte de la période pendant laquelle la file est restée vide pour éviter d'avoir un écart trop grand entre l'approximation de la taille de la file et sa taille réelle. Concrètement, la nouvelle valeur de avg sera $avg(1-w_q)^{f(time-qtime)}$ où $time$ est le temps courant, $qtime$ est le moment où la file est devenue vide, et f est une fonction linéaire déterminant le nombre de mises à jour nécessaires sur cette période.

3.3.1.2. RIO et WRED

Supposons que les paquets arrivant dans une file d'attente puissent avoir un niveau de précédence. Par exemple, les paquets “in” sont conformes et doivent être traités avec davantage d'égards, et les paquets “out” sont non conformes et seront les premiers à être détruits en cas de congestion. La politique de décongestion RIO (RED with In and Out) [CW 97, CF 97] est une extension de RED qui permet de définir des seuils différents pour les deux niveaux de précédence, de sorte que les paquets “out” seront détruits plus tôt et plus souvent que les paquets “in”. Une généralisation de RIO à plus de deux niveaux est WRED (Weighted RED). Cette politique peut entre autres s'appliquer à DiffServ qui définit trois niveaux de précédence par classe AF.

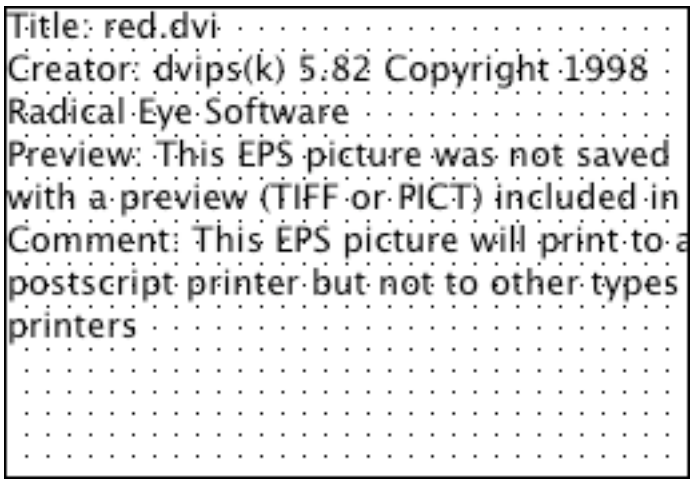

Figure 3.2. *Probabilité de pénalité d'un paquet en fonction de la taille moyenne de la file d'attente dans RED.*

3.3.1.3. Description des files d'attente en E-LOTOS

Une file d'attente réelle à toujours une capacité finie, et la politique de trop-plein est donc intrinsèquement liée à la file. Toutefois, pour la modularité de la spécification, nous avons préféré factoriser le problème en séparant la gestion FIFO d'une part, de la politique de gestion de congestion d'autre part, fut-elle aussi simple qu'un trop-plein. C'est pourquoi, nous définissons deux processus synchronisés : `FIFO_Queue` et `Drop_Tail`.

Le processus `FIFO_Queue`

Ce processus (voir figure 3.3) reçoit les paquets sur sa porte `input`. D'autre part, tant que la file n'est pas vide, il est prêt à émettre le paquet se trouvant en tête de file par la porte `output`.

La structure de données utilisée pour conserver les paquets présents dans une file est une simple file FIFO infinie. Le typage des portes de la file tire parti du sous-typage. Celles-ci ont le type `gen_packet` qui est un enregistrement extensible contenant *au moins* un champ de type `paquet`. Ceci permet d'instancier le processus `FIFO_Queue` avec des portes dont le type est un enregistrement disposant de champs supplémentaires. L'exemple typique d'un tel champ est une balise identifiant la file d'attente à utiliser, ou encore une estampille temporelle indiquant un temps de service. Nous en verrons des exemples dans la suite. Le type `classified_packet` de la figure 3.4 est un exemple de sous-type de `gen_packet`. Remarquons que le

type `packet` est lui-même extensible, mais comprend obligatoirement un champ de données (éventuellement vide), sa taille (en octets) et un niveau de précédence.

```
interface Packet_Queue
   type packet is (data => bytestring, size => rational,
                   precedence => level, etc) endtype
   (* "size" est la taille des paquets en bytes. Le fait
      qu'elle soit fractionnaire simplifie les manipulations.
      "precedence" est le niveau de précédence du paquet *)
   type level is low | high endtype
   (* dans le cas de deux niveaux *)
   type gen_packet is (pkt => packet, etc) endtype
   type fifo
   value empty_queue : fifo
   function enqueue (fifo, gen_packet) : fifo
   function dequeue (fifo) raises [cannot_dequeue] : fifo
   function first (fifo) raises [no_packet] : gen_packet
   function isempty (fifo) : bool
endint
process FIFO_Queue [input, output : gen_packet] (q : fifo) is
  input ?x; FIFO_Queue [...] (enqueue (q, x))
  []
  output !first(q) [not empty(q)] ;
     FIFO_Queue [...] (dequeue (q))
endproc
```

Figure 3.3. *Modélisation d'une file FIFO (infinie)*

Le processus `FIFO_Queues`

Un système à files multiples est très simple à représenter. Il est composé de plusieurs processus en parallèle : un par file (voir figure 3.4). Chaque file est ici modélisée par le processus `FIFO_Queue` auquel on adjoint un processus `Q_Selector` qui permet, grâce à sa synchronisation avec `FIFO_Queue`, de garantir que la file ne traitera que les paquets la concernant : ceux dont la balise `Qid` correspond au numéro attribué à cette file.

```
type classified_packet is (pkt => packet, qid => nat, etc)
endtype

process FIFO_Queues [input, output : classified_packet]
                    (n : nat) is
  if n > 0 then
    (Q_Selector [input, output] (n)
     |[input, output]|
     FIFO_Queue [input, output] (empty_queue)
    )
    |||
    FIFO_Queues [...] (n-1)
  endif
endproc

process Q_Selector [input, output : classified_packet]
                   (My_Qid : nat) is
loop
  input (qid => !My_Qid, etc)
  []
  output (qid => !My_Qid, etc)
endloop
endproc
```

Figure 3.4. *Modélisation d'un système à files multiples*

3.3.1.4. Description des politiques de décongestion en E-LOTOS

Le processus en charge de cette politique sera placé avant la file d'attente. Cette architecture est représentée à la figure 3.5 dans le cas simple d'une seule file et de la politique de trop-plein, mais restera valable dans des cas plus compliqués que nous décrirons plus loin. La description en E-LOTOS est donnée à la figure 3.6.

Le processus `Drop_Tail`

Ce processus modélise la politique de trop-plein (voir figure 3.7). Il réceptionne un `gen_paquet (x)` sur la porte `input`. Il vérifie si, en fonction de sa taille (`x.pkt.size`), de l'occupation (`qlen`) et de la capacité (`maxqlen`) de la file d'attente, il est possible de mémoriser ce paquet. Si c'est le cas, le processus dirige celui-ci vers la file d'attente via la porte `forward`. Le processus observe aussi la sortie de la file d'attente (porte `output`), afin de garder à jour l'état (`qlen`). La figure 3.7 représente le cas où la politique de décongestion s'applique à une file. La même structure permettrait d'appliquer la politique à un ensemble de files (partageant une

même zone mémoire) en remplaçant simplement, dans le processus `Drop_Tail_Queue`, le processus `FIFO_Queue` par `FIFO_Queues` défini à la figure 3.4. Pour appliquer une politique de trop-plein à chaque file séparément, il conviendrait de procéder un peu différemment. Dans ce cas, on composerait en parallèle autant de processus `Drop_Tail` qu'il y a de files dans le système, et chacun serait paramétré avec la capacité attribuée à cette file. En parallèle sur chaque processus `Drop_Tail`, on ajouterait un processus `Q_Selector` semblable à celui de la figure 3.4, de sorte que chaque `Drop_Tail` n'appliquera sa politique qu'à la file adéquate.

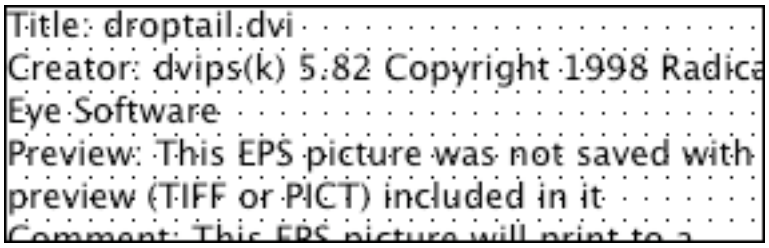

Figure 3.5. *Système composé d'une file et d'une politique de trop-plein*

```
process Drop_Tail_Queue [input, output : gen_packet]
                        (maxqlen : rational) is
hide forward : gen_packet in
  Drop_Tail [input, forward, output] (maxqlen, 0)
  |[forward, output]|
  FIFO_Queue [forward, output] (empty_queue)
endhide
endproc
```

Figure 3.6. *Le processus qui modélise une file avec la politique de trop-plein*

Le processus `RED_dropper`

Le processus `RED_dropper` (figure 3.8) est censé être instancié avec une file vide (`qlen` = 0, `avg` = 0) et des paramètres statiques `maxqlen`, `minth`, `maxth`, `maxp` et `wq`. L'algorithme RED, tel qu'il a été décrit, peut être séparé en deux : d'une part, un processus `Q_Param` qui se charge du calcul de la taille moyenne de la file, et d'autre part, le processus `RED_curve` qui calcule la probabilité de pénalité et qui l'applique le cas échéant. Le premier doit communiquer au second les résultats de ses calculs, ce qui est réalisé à la volée par la porte `input` au moment où les paquets arrivent. Pour cela, il est toutefois nécessaire d'étendre le typage de la porte `input` afin que ces valeurs puissent être communiquées. Le type de la porte `input` est un

`PacketAndQparam` (figure 3.8) dont les deux derniers composants sont respectivement les tailles moyenne et courante de la file. Toutefois, comme ces deux composants supplémentaires ne sont utilisés que par ces deux processus, ils sont cachés à l'environnement par l'opérateur de renommage.

```
Process Drop_Tail [input, forward, output : gen_packet]
                  (maxqlen, qlen: rational) is
input ?x [qlen + x.pkt.size <= maxqlen];
   forward !x;
   Drop_Tail [...] (maxqlen, qlen + x.pkt.size)
[]
input ?x [qlen + x.pkt.size > maxqlen];
   Drop_Tail [...] (...)
[]
output ?x;
   Drop_Tail [...] (maxqlen, qlen - x.pkt.size)
endproc
```

Figure 3.7. *Processus qui modélise la politique de trop-plein*

Le calcul de la moyenne est réalisé par le processus `Q_Param` (figure 3.9). Remarquons que nous distinguons le cas où un paquet arrive dans une file vide (`qlen` = 0). Dans ce cas, il est important de connaître depuis quand cette file est vide. Cette durée est donnée directement par la valeur de la variable `t`, qui représente dans ce cas le temps écoulé depuis la dernière action `output`. La variable `t` est de portée locale (par la sémantique de l'opérateur **`var`**). Ceci est nécessaire afin que les deux branches du premier opérateur de choix se terminent avec les mêmes liaisons de variables.

Le calcul de la probabilité de pénalité et son application sont du ressort du processus `RED_curve` (figure 3.9). Remarquons que lorsque la taille moyenne de la file est entre les deux seuils, la décision de pénaliser le paquet est normalement probabiliste, mais que E-LOTOS ne peut exprimer cela formellement. Le seul moyen d'approcher la solution est de laisser ce choix non déterministe. Nous avons toutefois écrit le processus de telle sorte que ce choix non déterministe puisse aisément être remplacé par un choix probabiliste, si un tel opérateur était introduit dans le langage. Cette structure faciliterait aussi une transcription automatique de la spécification vers un modèle stochastique. Il suffirait pour cela de définir une annotation spécifique à l'opérateur de choix qui permettrait d'y associer une probabilité.

```
type PacketAndQparam is (gen_pkt => gen_packet,
                         aQsize => rational,
                         Qsize => rational) endtype

process RED_dropper [input, forward, output : gen_packet]
                    (maxqlen, qlen, minth, maxth, maxp, wq,
                     avg : rational) is
rename gate input ((gen_pkt => ?x, etc) : PacketAndQparam) is
            input !x in
     (* supprime tous les champs autres que gen_pkt *)
hide drop : none in
   RED_curve [input, forward, drop] (minth, maxth, maxp)
   |[input, forward, drop]|
   Q_Param [input, forward, drop, output] (maxqlen, 0, wq, 0)
endhide
endren
endproc
```

Figure 3.8. *Processus modélisant l'algorithme RED*

```
process RED_curve [input: PacketAndQparam, forward : gen_packet,
                   drop : none] (minth, maxth, maxp : rational) is
loop
   input (gen_pkt => ?x, MQsize => ?avg, Qsize => ?qlen) ;
   if (avg < minth) then forward !x    (* pas de pénalité *)
   elseif (new_avg > maxth) then drop  (* pénalité *)
   else (* minth <= new_avg <= maxth *)
   ? pb :=
     maxp *((avg - minth)/(maxth - minth))*(x.pkt.size/maxsize);
         (* maxsize est une constante globale dont la valeur
            est la taille du plus grand paquet en octets *)
   (i ;   (* pénalité *)
    drop
    [] (* devrait être un choix probabiliste avec une
          probabilité pb de choisir la première clause *)
    i ;   (* pas de pénalité *)
    forward !x)
   endif
endloop
endproc
```

```
process Q_Param [input : PacketAndQparam, forward : gen_packet,
                 drop : none, output : gen_packet]
                (maxqlen, qlen, wq, avg : rational) is
  (var t : time in
      input (gen_pkt => ?x, aQsize => ?new_avg,
             Qsize => ?new_qlen) @?t
            [(qlen == 0) and
             (new_avg == avg * ((1 - wq) ** f(t))) and
             (new_qlen == x.pkt.size)] ;
         (* arrivée d'un paquet dans une file vide,
            f est une simple fonction linéaire *)
   endvar
   []
   input (gen_pkt => ?x, aQsize => ?new_avg, Qsize => ?new_qlen)
          [(qlen > 0) and
           (new_avg == (1 - wq) * avg + wq *(qlen + x.pkt.size))
            and (new_qlen == qlen + x.pkt.size)] ;
        (* arrivée d'un paquet dans une file non vide *)
  ) ;
  (forward ?any : gen_packet ;
          Q_param [...] (maxqlen, new_qlen, wq, new_avg)
   []
   drop ; Q_param [...] (maxqlen, qlen, wq, avg)
  )
  []
  output ?x ; Q_param [...] (maxqlen, qlen - x.pkt.size, wq, avg)
endproc
```

Figure 3.9. *Les deux composantes de l'algorithme RED*

A l'instar du processus `Drop_Tail`, le processus `RED_dropper` peut s'appliquer indifféremment à une file d'attente représentée par le processus `FIFO_Queue` ou à un ensemble de files représenté par le processus `FIFO_Queues`.

Le processus `RIO`

Ce processus (figure 3.10) a une structure comparable à `RED_dropper`, mais où le processus `RED_curve` est simplement dédoublé. Ceci permet d'appliquer deux politiques RED distinctes aux deux niveaux de précédence (`low` ou `high`). Chaque `RED_curve` ne traite que les paquets de son niveau grâce aux processus `Precedence_Selector` qui jouent le rôle de filtres.

```
process RIO_dropper [input, forward, output : gen_packet]
                    (maxqlen, minth_l, maxth_l, maxp_l, minth_h,
                     maxth_h, maxp_h, wq : rational) is
rename gate input ((gen_pkt => ?x, etc) : PacketAndQparam) is
            input !x
     (* supprime tous les champs autres que gen_pkt *)
in
hide drop : none in
  ((RED_curve [input, forward, drop] (minth_l, maxth_l, maxp_l)
    |[input]|
    Precedence_Selector [input] (low)
   )
   |||
   (RED_curve [input, forward, drop] (minth_h, maxth_h, maxp_h)
    |[input]|
    Precedence_Selector [input] (high)
  ))
  |[input, forward, drop]|
  Q_param [input, forward, drop, output] (maxqlen, 0, wq, 0)
endhide
endren
endproc

process Precedence_selector [input : PacketAndQparam]
        (dp : level) is
  loop
  input (gen_pkt => ?x, etc) [x.pkt.precedence == dp]
  endloop
endproc
```

Figure 3.10. *Processus RIO construit à partir de deux composants RED_curve*

3.3.2. Régulation de trafic

Une des causes principales de la congestion dans les réseaux est le fait que les paquets peuvent être transmis par rafales. Cette augmentation soudaine du débit provoque une accumulation de paquets dans les files des routeurs, qui peut aller jusqu'au débordement de celles-ci.

3.3.2.1. Régulateur à seau

L'algorithme le plus connu pour réguler un trafic avant de l'émettre dans un réseau est le seau à jetons (*token bucket* en anglais) [TUR 86]. Les différents éléments constituant le système sont présentés à la figure 3.11. Conceptuellement, le régulateur comprend une file de taille finie (*Queue*) et un seau qui contient des jetons (*Token Bucket*).

Les paquets entrants transitent par la file d'attente et ne pourront en sortir que si le seau contient suffisamment de jetons. Le seau peut contenir $\square$ jetons au maximum et est initialement plein. Chaque jeton représente le droit d'émettre un octet. D'autres variantes existent où un jeton représente le droit d'émettre un paquet. C'est le cas notamment lorsque les paquets sont de taille fixe, comme dans les réseaux ATM. Le taux d'arrivée des jetons dans le seau est de $\square$ jetons par seconde. Lorsqu'un paquet est émis, on retire du seau le nombre de jetons correspondant à la taille du paquet. Ainsi, le débit moyen de transmission des paquets sur la ligne de sortie est fixé par le paramètre $\square$ quel que soit leur taux d'arrivée. D'autre part, si suffisamment de jetons ont été accumulés dans le seau, il est concevable qu'une (petite) rafale de paquets soit émise.

Ce régulateur de trafic répond donc à l'équation suivante : sur *tout* intervalle de durée $\square$, le nombre d'octets transmis est borné par $\square\square+\square$

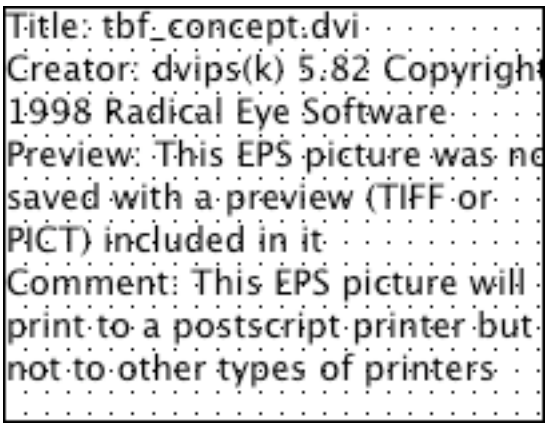

Figure 3.11. *Les différents éléments du régulateur à seau*

Il est à noter que, contrairement à ce que l'on pourrait penser de prime abord, la taille maximale d'une rafale sortant du régulateur n'est pas $\square$, mais est légèrement

plus grande étant donné que lors de la transmission, le seau continue à se remplir. Il est possible de déterminer cette taille maximale analytiquement en résolvant le système d'équations suivant où [illegible] est la durée de la plus grande rafale. Pour un [illegible], un [illegible] et un débit de la ligne de sortie *P* donnés, la taille de la plus grande rafale *MBS* (Maximum Burst Size) doit vérifier les deux équations suivantes :

$MBS = P.$[illegible] (équation de la ligne de sortie)

$MBS =$ [illegible] + [illegible] (équation du régulateur)

En égalisant et en résolvant par rapport à [illegible], on trouve après remplacement :

$$MBS = \frac{\square.P}{P \square \square}$$

Cette valeur est obtenue en considérant les flux de paquets comme des fluides. Si on tient compte du fait que la transmission se fait par paquets entiers (et non par octets successifs), la formule doit être quelque peu adaptée.

3.3.2.2. Description en E-LOTOS

Ce système est illustré à la figure 3.12. Il est composé de trois processus. Le processus `TB_Queue` modélise la file finie FIFO dans laquelle les paquets entrants sont reçus et éventuellement stockés, et `Tokens` qui modélise le seau contenant les jetons. La ligne de sortie est modélisée par la contrainte additionnelle `Line_Rate` qui représente le débit de la ligne et est donc une borne supérieure sur la rapidité avec laquelle les paquets peuvent être transmis.

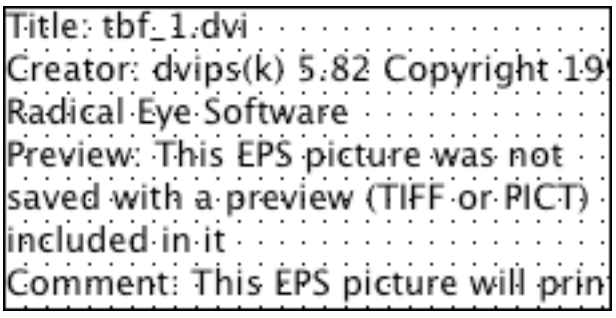

Figure 3.12. *Les processus constituant le régulateur TB_shaper*

La porte `input` permet au processus `TB_shaper` de se synchroniser avec l'émetteur des paquets pour les recevoir, et la porte `output` permet de se synchroniser avec la ligne de sortie pour émettre les paquets. La structure de ce processus est donnée à la figure 3.13. On peut voir que les processus tournent en parallèle et sont synchronisés sur la porte `output`. En effet, la synchronisation permet de respecter les deux contraintes suivantes : (a) existence d'un paquet dans la file, et (b) seau suffisamment approvisionné.

Le processus `TB_shaper`

Le régulateur est initialisé avec une file d'attente vide de taille `maxqlen`, et un seau rempli, c'est-à-dire contenant `sigma` (σ) octets. Le taux de remplissage du seau est de `rho` (ρ) octets/s. Le système est composé du processus `TB_Queue` qui modélise la file d'entrée munie de sa politique de décongestion, et du processus `Tokens` qui modélise la disponibilité des jetons. Le premier processus est simplement le processus `Drop_Tail_Queue` défini à la figure 3.6. Le renommage de la porte `output` a simplement pour but de supprimer tous les attributs autres que `pkt`. Ceci se justifie si on considère que la sortie de `TB_shaper` est une ligne de transmission sur laquelle seuls les paquets transitent. Les autres attributs, s'il en existe, n'étaient censés avoir qu'une utilité locale au système.

```
process TB_shaper [input : gen_packet, output : packet]
                  (sigma, rho, maxqlen : rational) is
rename gate output ((pkt => ?p, etc) : gen_packet) is output !p
     (* supprime tous les champs autres que pkt *)
in
   TB_Queue [input, output] (maxqlen)
   |[output]|
   Tokens [output] (sigma, rho, sigma)
endren
endproc
process TB_Queue [input, output : gen_packet]
                 (maxqlen : rational) is
   Drop_Tail_Queue [input, output] (maxqlen)
endproc
```

Figure 3.13. *Le processus qui modélise le régulateur à seau*

Le processus `Tokens`

Le processus `Tokens` (voir figure 3.14) constitue le cœur du régulateur. Celui-ci ne permet l'occurrence de l'événement `output` et donc la transmission éventuelle du paquet au sommet de la file que si suffisamment de jetons ont été accumulés pour couvrir sa taille. Un jeton étant équivalent à un octet, il est nécessaire que la taille du paquet soit inférieure ou égale au nombre de jetons présents dans le seau, ce qui explique la présence du prédicat `p.size <= min (sigma, content + rho * t)`. Le paramètre `content` représente le nombre de jetons présents dans le seau lors de l'instanciation du processus, c'est-à-dire après la dernière transmission de paquet. Dès lors, `content + rho * t` représente le nombre de jetons présents dans le seau `t` secondes après l'instanciation. Il faut noter que `content` est un nombre rationnel, car nous modélisons le remplissage du seau comme un fluide. Cette abstraction est tout à fait valide, même si les paquets sont bien sûr composés d'un nombre entier d'octets, car le test du régulateur assure que le nombre (fractionnaire) de jetons est supérieur (ou égal) au nombre entier d'octets du paquet.

```
type time renames rational endtype

process Tokens [output : gen_packet]
               (sigma, rho, content : rational) is
   output (pkt => ?p, etc) @?t
                    [p.size <= min (sigma, content + rho * t)] ;
   Tokens [output] (sigma, rho,
                   min (sigma, content + rho * t) - p.size)
endproc
```

Figure 3.14. *Le processus qui modélise le seau*

Le processus `Line_Rate`

Pour décrire le processus `Line_Rate`, il est important de déterminer la signification associée à l'occurence d'un événement sur la porte `input` du processus. Marque-t-elle la fin ou le début de la transmission du paquet ? C'est une question de convention. Nous avons choisi le début de la transmission du paquet qui permet d'exprimer plus simplement le processus `Line_Rate`. Si on considère un débit de *rate* octets/s et une taille de paquet de *n* octets, la transmission dure *n*/*rate* secondes. Donc la ligne ne peut pas accepter de nouveau paquet avant que cette période ne soit écoulée. C'est ce qui est exprimé par `Line_Rate` (figure 3.15).

```
process Line_Rate [input : packet] (rate : rational) is
   loop
      input ?p ;
      wait (p.size/rate) ;
   endloop
endproc
```

Figure 3.15. *Le processus qui impose le débit maximal de la ligne de sortie*

3.3.3. Contrôle de trafic

Les fonctions de régulation et de contrôle de trafic sont duales. Si une source est censée respecter un contrat de trafic défini par les paramètres d'un seau à jetons, elle mettra certainement son trafic en forme en utilisant le processus `TB_shaper` vu à la figure 3.13. De son côté, le réseau vérifiera si le trafic entrant est conforme, en utilisant un algorithme assez semblable et basé sur les mêmes paramètres. Cet algorithme a pour but de marquer les paquets non conformes avec un niveau de précédence bas et de laisser intact le niveau de précédence des paquets conformes. Nous supposerons que l'en-tête d'un paquet contient un champ (ou un bit dans ce cas simple) de précédence. Un contrôleur basé sur un seau à jetons observera tous les paquets entrants. S'il y a moins de jetons dans son seau que d'octets dans le paquet, celui-ci est non conforme et marqué en conséquence. Un paquet marqué de la sorte ne consomme pas de jetons. Dans le cas contraire, le paquet n'est pas modifié, mais le seau est vidé d'un nombre de jetons égal à la taille du paquet en octets.

Lorsque deux routeurs de frontière sont interconnectés par une ligne, le régulateur du routeur amont et le contrôleur du routeur aval sont aux extrémités de cette ligne. Si de plus les deux processus sont instanciés avec les mêmes paramètres σ et ρ, le contrôleur de trafic ne marquera logiquement aucun paquet.

Le contrôleur peut aussi avoir une attitude plus brutale et détruire les paquets non conformes au lieu de les marquer. Nous avons vu que ce type de contrôleur est utilisé dans DiffServ pour la classe EF par exemple. Les deux versions de ce contrôleur : `TB_dropper` et `TB_marker`, sont illustrées aux figures 3.16 et 3.17. Ils ont la même structure, mais le second est légèrement plus compliqué, car il lie deux variables (`rest` et `other_fields`) à des ensembles de champs d'enregistrements (*record matching*). Ceci a pour but de mémoriser de tels ensembles de champs sans connaître leur structure (notamment quand celle-ci est extensible), afin de pouvoir les restituer lors d'une interaction ultérieure.

```
process TB_dropper [input, output : gen_packet]
                   (sigma, rho, content : rational) is
   input ?x @?t [x.pkt.size <= min (sigma, content + rho * t)] ;
   output !x ;   (* pas de destruction *)
   TB_dropper [input] (sigma, rho,
                       min (sigma, content + rho * t)
                       - x.pkt.size)
   []
   input ?x @?t [x.pkt.size > min (sigma, content + rho * t)] ;
                (* destruction *)
   TB_dropper [input] (sigma, rho,
                       min (sigma, content + rho * t))
endproc
```

Figure 3.16: Filtrage à l'aide d'un seau à jetons

```
process TB_marker [input, output : gen_packet]
                  (sigma, rho, content : rational) is
   input ?x @?t [x.pkt.size <= min (sigma, content + rho * t)] ;
   output !x;   (* pas de marquage *)
   TB_marker [input] (sigma, rho,
                      min (sigma, content + rho * t)
                       - x.pkt.size)
   []
   input (pkt => (precedence => any : level, size => ?s,
                  ?rest as etc),
          ?other_fields as etc) @?t
                          [s > min (sigma, content + rho * t)] ;
                (* marquage *)
   output (pkt => (precedence => !low,
                   size => !s,
                   !rest),
           !other_fields);
   TB_marker [input] (sigma, rho,
                      min (sigma, content + rho * t))
endproc
```

Figure 3.17: Marquage à l'aide d'un seau à jetons

3.3.4. Ordonnancement

Chaque routeur du réseau utilise une certaine politique d'ordonnancement pour déterminer l'ordre dans lequel les paquets de différents flux partageant une même ligne de sortie seront transmis. Dans le plus simple des cas, on utilise une discipline de type FIFO qui permet de transmettre les paquets dans l'ordre dans lequel ils sont arrivés. Mais cette technique ne permet pas de donner plus d'importance (de poids) à un flux donné par rapport aux autres, alors que c'est nécessaire pour implanter des qualités de service. Il existe une panoplie d'algorithmes qui visent à résoudre ce problème. Dans cette section, nous décrirons l'algorithme d'ordonnancement à équité pondérée, *Weighted Fair Queuing* (WFQ) [DKS 90].

*3.3.4.1. Critère d'**Erreur! Signet non défini.**équité max-min*

Ce critère permet de répondre à la question de savoir "Comment diviser équitablement une ressource entre différents utilisateurs ayant des demandes différentes ?". Intuitivement, l'équité max-min est réalisée lorsque chaque utilisateur dont la demande n'est pas totalement satisfaite obtient une même part p (la plus grande possible) de la ressource, tandis que ceux dont la demande est inférieure à p sont totalement satisfaits. En d'autres termes, les utilisateurs les moins demandeurs obtiennent ce qu'ils veulent, et les autres obtiennent une part égale du reste. Ce principe de répartition des ressources est appelé une allocation équitable max-min (*max-min fair*) parce qu'il maximise la fraction minimale de ressources allouée à un utilisateur dont la demande n'est pas complètement satisfaite.

On peut donner une interprétation géométrique simple (voir figure 3.18) à l'algorithme permettant d'effectuer le partage de la ressource en respectant le critère d'équité max-min. Chaque rectangle (à base unitaire) de la figure 3.18 représente la demande (exprimée en % de la ressource) de chacun des trois flux. L'allocation max-min consiste à déterminer l'ordonnée de la ligne en pointillés, qui représente ce qui est alloué à chaque flux, en tenant compte des contraintes suivantes :

- La somme des surfaces hachurées (ce qui est alloué à chaque flux) doit être inférieure ou égale à 100 % de la ressource.
- Les différentes surfaces hachurées doivent être les plus grandes possibles (sans dépasser la demande de chaque flux).

Dans notre exemple, il y a trois flux qui demandent respectivement une fraction des ressources égale à 66 %, 83 % et 23 %. Il est manifestement impossible de satisfaire les trois demandes en même temps. En appliquant le principe, le flux le moins gourmand prend ses 23 %, car $23 \leq 100/3$, et le reste, soit 77 %, sera alloué aux deux autres flux; ce qui donne 38,5 % de la ressource à chacun.

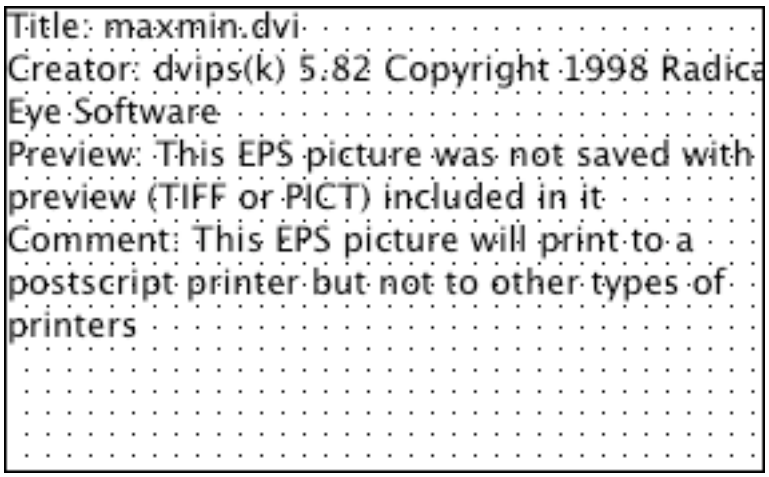

Figure 3.18. *Illustration du critère d'équité max-min*

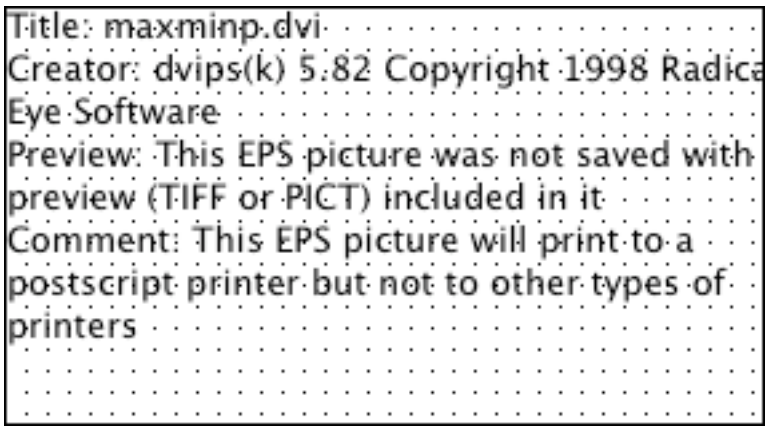

Figure 3.19. *Illustration du critère d'équité max-min pondéré*

Si on doit tenir compte de poids conférant à certains flux une plus grande importance qu'à d'autres, il suffit de normaliser les poids en les divisant par le plus petit poids (ce qui est permis étant donné que les poids sont relatifs), et de prendre pour chaque flux une base de rectangle égale au poids normalisé du flux. On peut alors raisonner de la même manière que précédemment, mais en lisant les fractions de la ressource totale non plus sur l'ordonnée mais en terme de surfaces hachurées.

Reprenons le même exemple en fixant des poids respectifs de 6, 3 et 9 (voir figure 3.19). Après normalisation, ces poids deviennent respectivement 2, 1 et 3. Donc, par exemple, le rectangle du premier flux sera deux fois plus large que celui

du second. Sa demande de 66 % sera donc représentée par un rectangle de base 2 (son poids relatif) et de hauteur 33 %. Le flux le moins gourmand aura ses 23 % de la ressource, et les deux autres se partageront les 77 % restant à raison de 51,3 % pour le premier et 25,7 % pour le second.

3.3.4.2. Generalized Processor Sharing (GPS)

Si nous supposons que tous les poids sont identiques, le système GPS peut allouer une ressource équitablement selon le principe max-min en utilisant la politique d'ordonnancement suivante : le système place les paquets de flux distincts dans des files séparées et examine tour à tour celles qui contiennent au moins un paquet pour servir une quantité de données *infinitésimale* de chacune d'elles. De cette manière, dans tout intervalle de temps non nul, il a examiné chaque file au moins une fois. Lorsque l'on utilise des poids, il suffit de servir chaque file proportionnellement à son poids. Cette façon de procéder conduit à une allocation max-min.

Cette politique est évidemment irréalisable en pratique car les paquets ne peuvent être considérés comme des fluides, mais elle donne la politique idéale à approcher le plus possible.

3.3.4.3. Description de WFQ

WFQ est un ordonnanceur qui tente d'approximer GPS en le simulant.

Intuitivement, pour chaque paquet entrant, WFQ calcule et attache au paquet le moment (appelé *numéro de service* ou *finish number* en anglais) auquel ce paquet aurait été servi par GPS, et utilise ces numéros pour servir les paquets par ordre de numéros croissants. Ce calcul est basé sur une variable appelée le *numéro de tour* (ou *round number* en anglais) qui indique le nombre de passages (tours) que GPS a déjà effectués dans l'ensemble de ses files. Cette variable est fonction du temps et croît à une vitesse inversement proportionnelle aux nombres de files actives. Une file étant active dans le système GPS si un de ses paquets est en cours de service. Ceci peut également s'exprimer comme suit. Une file est active au sens GPS si le dernier numéro de service attribué à un paquet de cette file est plus grand que le numéro de tour.

Nous avons dit que la croissance du numéro de tour est inversement proportionnelle au nombre de files actives. Cela s'explique par le raisonnement suivant. S'il y a n (> 0) files actives, chaque file apporte une contribution de $1/n$ dans la réalisation du tour, étant donné que lorsque toutes les files ont été examinées, nous avons effectué un tour. Si chaque examen prend une unité de temps, le tour prend n unités de temps, c'est-à-dire que le temps nécessaire pour servir une quantité infinitésimale de données de chaque flux prend n fois plus de temps que s'il n'y avait

qu'un flux actif. Notons toutefois que s'il n'y a pas de file active, aucune file n'est examinée et le numéro de tour ne change pas.

Si P_i^k est la taille du *k*ème paquet du flux *i* qui arrive au temps *t*, *R*(*t*) le numéro de tour (*Round*) à cet instant, et F_i^k le numéro de service (*Finish number*) du *k*ème paquet de ce flux *i*, alors

$$F_i^k = \max[F_i^{k-1}, R(t)] + P_i^k$$

La formule est facile à comprendre si on considère qu'à chaque tour, GPS sert un bit de chaque paquet au lieu de servir une quantité de données infinitésimale. Cette hypothèse est légitime étant donné que, hormis l'hypothèse de croissance inversement proportionnelle, la seule condition imposée au numéro de tour est d'évoluer de la même manière pour tous les flux. De plus, la plus petite quantité d'information accessible dans un paquet est un bit. Si un paquet entrant fait *p* bits, il faudra *p* tours pour qu'il soit servi après que le paquet précédent du même flux ait été servi s'il y en a un, ou dans le cas contraire, à partir du moment où il est arrivé. C'est ce qu'exprime la formule.

Si on tient compte des poids w_i associés aux flux, le numéro de tour croît avec un facteur $1/w_i$ et la formule devient :

$$F_i^k = \max[F_i^{k-1}, R(t)] + \frac{P_i^k}{w_i}$$

3.3.4.4. Description de WFQ en E-LOTOS

Nous donnerons tout d'abord une version du système en pseudo-code pour permettre d'avoir une vision claire et précise de ce que les processus devront réaliser. L'idée est de simuler GPS pour attribuer un numéro de service à chaque paquet entrant. Toute la difficulté réside dans le calcul du numéro de tour, et plus précisément dans la détermination du moment où le nombre de files actives (au sens GPS) change. Ceci peut se produire lors de l'émission du dernier paquet d'une file (active), diminuant ainsi le nombre de files actives d'une unité, ou lors de l'arrivée d'un paquet dans une file vide (inactive), augmentant ce nombre d'une unité.

Comme WFQ n'est qu'une approximation de GPS, le départ *réel* d'un paquet ne coincide presque jamais avec son départ *au sens GPS*. Il est donc nécessaire de provoquer artificiellement un événement correspondant au départ d'un paquet *au sens GPS* pour mettre à jour le numéro de tour.

Nous utiliserons les variables et ensembles suivants :

- t_{old} représente le moment auquel est survenu le dernier événement (arrivée ou départ d'un paquet) au sens GPS.
- R_{old} est le numéro de tour au temps t_{old}.
- t_{next} ($\geq t_{old}$) représente le premier instant après t_{old} auquel un paquet sera complètement servi (i.e. transmis) au sens GPS.
- *Act* est l'ensemble des files actives au sens GPS.
- *FNset* est un ensemble de paires composées d'un "finish number" et d'un numéro de file. Chaque paire correspond à un paquet présent (non encore complètement transmis) dans le système GPS.
- F_{min} est le plus petit numéro de service des paquets présents dans le système GPS, ou encore dans l'ensemble *FNset*. Nous utilisons une variable séparée pour plus de clarté.

La marche à suivre lors de la phase d'initialisation et en réponse à l'un des deux événements qui peuvent se produire dans le système GPS est la suivante :

Initialisation

- Les files GPS sont initialement vides
- $t := 0, t_{old} := 0, t_{next} :=$, $R_{old} := 0, Act := \varnothing$, $FNset := \varnothing$, $F_{min} := 0$

Arrivée du *k*ème paquet de taille P_i^k de la file *i* au temps *t*

- Calcul du nouveau numéro de tour : $R := R_{old} + \frac{t - t_{old}}{\sum_{i \in Act} w_i}$ si $Act \neq \varnothing$, sinon $R := R_{old}$.
- Calcul du numéro de service de ce paquet : $F_i^k := \max(F_i^{k-1}, R) + \frac{P_i^k}{w_i}$
- On ajoute le paquet dans sa file
- Mises à jour : $t_{old} := t$, $R_{old} := R$, $FNset := FNset \sqcup (F_i^k, i)$, $Act := Act \sqcup \{i\}$, $F_{min} := \min(FNset)$
- Temps de sortie GPS du prochain paquet : $t_{next} := t + (F_{min} - R) \sum_{i \in Act} w_i$

***t* devient égal à t_{next} (fin de transmission d'un paquet au sens GPS)**

- Il s'agit d'un paquet de la file *j* telle que $(F_{min}, j) \in FNSet$.
- Calcul du nouveau numéro de tour : $R := R_{old} + \frac{t - t_{old}}{\sum_{i \in Act} w_i}$
- Mises à jour : $t_{old} := t$, $R_{old} := R$, $FNset := FNset - \{(F_{min}, j)\}$, $F_{min} := \min(FNset)$
- Si la file *j* devient inactive : $Act := Act - \{j\}$, sinon *Act* est inchangé.

- Temps de sortie GPS du prochain paquet : $t_{next} := t + (F_{min} \cdot R) \cdot \sum_{i \in Act} w_i$ si $Act \neq \emptyset$, sinon $t_{next} :=$.

Le système d'ordonnancement `WFQSystem` est illustré à la figure 3.20 et décrit en E-LOTOS à la figure 3.21. Il est composé des processus `Filter`, `Dropper`, `GPS_tagging`, `FIFO_Queues` et `WFQ_Order`. Le processus `GPS_tagging` implante le simulateur GPS détaillé plus haut et ajoute un numéro de service aux paquets. Le processus `FIFO_Queues` représente les files d'attentes du système. Le processus `WFQ_Order` fixe l'ordre de service des files par ordre croissant de numéros de service. Le processus `Dropper` spécifie la politique de decongestion. Enfin, le processus `Filter` élimine les paquets dont le numéro de file associé correspond à une file non approvisionnée, c'est-à-dire dont le poids est nul.

Notons aussi que vu la présence des numéros de service associés aux paquets, il est nécessaire d'étendre le type `classified_packet` avec un champ nouveau `tag`. Ceci donne le type `tagged_packet` qui est lui-même extensible. On notera que le champ `tag` est masqué à la sortie du système, vu qu'il n'a d'intérêt que pour l'algorithme WFQ. Ce masquage est réalisé par le renommage de la porte `output`.

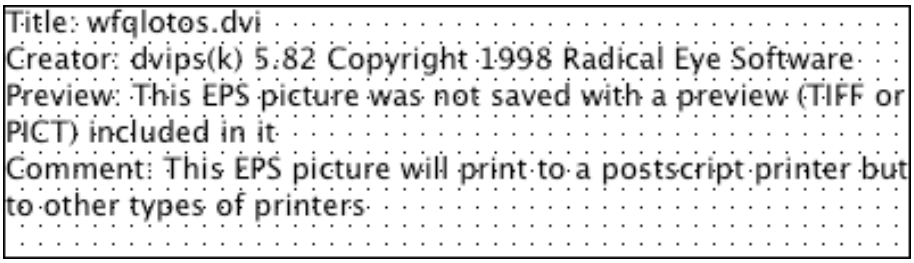

Figure 3.20. *Les processus constituant le système d'ordonnancement ainsi que la contrainte sur le débit*

Le processus `GPS_tagging`

Ce processus (voir figure 3.22) simule GPS et suit la structure décrite ci-dessus. L'initialisation apparaît lors de l'instanciation dans le processus `WFQSystem` (voir figure 3.21). La partie exécutée lors de l'arrivée d'un paquet correspond au comportement qui débute avec la porte `input`. La partie qui correspond à la fin de transmission d'un paquet du système GPS correspond au comportement qui débute par l'action interne `i`.

```
Type command is (code => opcode, qid => nat, weight => rational,
                 etc) endtype
Type opcode is add | remove endtype

process WFQSystem [input, output : classified_packet,
                   cmd : command]
                  (nbqueues : nat, maxqlen : rational) is
   hide m1, m2 : classified_packet in
      Filter [input, m1, cmd] ({})
      |[m1, cmd]|
      (Dropper [m1, m2, output] (maxqlen, 0)
       |[m2, output]|
       WFQ [m2, output, cmd] (nb_queues))
   endhide
endproc

Type tagged_packet is (pkt => packet, qid => nat,
                       tag => rational, etc)
endtype

Process WFQ [m2, output : classified_packet,
             cmd : command] (nb_queues : nat) is
rename gate output ((tag => any : rational, ?other_fields as etc)
                    : tagged_packet)
            is output (!other_fields)
     (* suppression de l'attribut "tag" à la porte output *)
in
hide m3 : tagged_packet in
      (GPS_tagging [m2, m3, cmd] (0, 0, 0, 0, [], [])
       |[m3]|
       (FIFO_Queues [m3, output] (nb_queues)
        |[m3, output]|
        WFQ_Order [m3, output] ([])
      ))
endhide
endren
endproc
```

Figure 3.21. *Le processus qui modélise le système d'ordonnancement*

Par rapport au pseudo-code, on notera les différences suivantes :

- La variable t_{next} est remplacée par `delta_next` qui vaut $t_{next} - t_{old}$. Ceci permet de simplifier la description. En outre, puisqu'une valeur infinie n'est pas acceptable pour `delta_next`, nous remplaçons un délai infini par une garde fausse.
- L'ensemble *Act* n'apparaît pas explicitement, mais fait partie d'une structure de donnée plus globale, appelée `GPS_state`, qui est une liste de descripteurs de files. Chaque descripteur de file, de type `tflow`, a la structure suivante : `(qid => nat, weight => rational, size => rational, lastfn => rational)` où `qid` est le numéro de la file, `weight` est le poids associé à cette file, `size` est le nombre de paquets dans cette file au sens GPS, et `lastfn` est le dernier numéro de service attribué à cette file. Une file est donc active quand `size` > 0.
- La fonction `addweights` retourne $\sum_{i \in Act} w_i$
- La fonction `get` extrait de `GPS_state` le descripteur d'une file donnée.
- Une interface `cmd` permet à un processus extérieur (p.ex. RSVP) de modifier le `GPS_state` lorsqu'une file doit être mise en ou hors service.

```
type tflow is (qid => nat, weight => rational, size => rational,
               lastfn => rational) endtype
type fn_flow is (tag => rational, qid => nat) endtype
type GPS_state is list of tflow endtype
type fn_state is list of fn_flow endtype
(* nous supposerons que des fonctions infixées telles que
   "++", "--" et "isin" ont été définies sur ces listes
   respectivement comme "concat" a été définie sur les listes et
   "diff" et "isin" sur les ensembles
*)

function addWeights (s : GPS_state) : rational is
if isempty (s) then 0
else
   var flow : tflow := nth (s, 0) in
       (if flow.size > 0 then flow.weight else 0 endif)
        + addWeights (tail (s))
   endvar
endif
endfun
```

```
function get (s : GPS_state, index : nat) : tflow
         raises [no_such_flow] is
if isempty (s) then raise no_such_flow
else
   var flow : tflow := nth (s, 0) in
       if flow.qid == index then flow
                           else get (tail (s), index)
       endif
   endvar
endif
endfun

function isin infix (index : nat, s : GPS_state) : Bool is
trap exception no_such_flow is false endexn
in var flow := get (s, index) in true endvar
endtrap
endfun

function notin infix (index : nat, s : GPS_state) : Bool is
not (index isin s)
endfun

function minfn (fnset : fn_state) : rational
         raises [no_minimum] is
(* donne le fn minimum contenu dans les paires de
   l'ensemble fnset *)
if isempty (s) then raise no_minimum
else
   min (nth (s, 0).tag, minfn (tail (s)))
endif
endfun

process GPS_tagging [input : classified_packet,
                     output : tagged_packet,
                     cmd : command]
                    (told, rold, delta_next, weights : rational,
                     fnset : fn_state, flows : GPS_state) is
   input (qid => ?j, pkt => ?p, ?other_fields as etc) @?t;
   ?flow := get (flows, j) ;
   ?r := if weights == 0 then rold
                         else rold + (t / weights) endif ;
   ?fn := max(flow.lastfn, r) + (p.size / flow.weight) ;
   output (qid => !j, pkt => !p, tag => !fn, !other_fields) ;
```

```
    ?ffnset := fnset ++ [(tag => fn, qid => j)];
    ?fflows := (flows -- [flow]) ++ [(qid => flow.qid,
                                      weight => flow.weight,
                                      size => flow.size + 1,
                                      lastfn => fn)];
    ?new_weights := addWeights(fflows) ;
    GPS_tagging [...] (told + t, r,
                       (minfn(ffnset) - r) * new_weights,
                        new_weights, ffnset, fflows)
    []
    if weights > 0 then
      wait (delta_next);
      ?r := rold + (delta_next / weights) ;
      ?Fmin := min(fnset) ;
      ?j := any : nat [(tag => Fmin, qid => j) isin fnset]
      ?flow := get(flows, j)
      ?fflows := (flows -- [flow]) ++ [(qid => flow.qid,
                                        weight => flow.weight,
                                        size => flow.size - 1,
                                        lastfn => flow.lastfn)];
      ?ffnset := fnset -- [(tag => Fmin, qid => j)] ;
      ?new_weights := addWeights (fflows) ;
      GPS_tagging [...] (told + delta_next, r,
                         (minfn(ffnset) - r) * new_weights,
                         new_weights, ffnset, fflows)
    endif
    []
    cmd (code => !add, qid => ?j, weight => ?w, etc)
                                    [(j notin flows) and (w > 0)];
    GPS_tagging [...] (told, rold, delta_next, weights, fnset,
                       flows ++ [(qid => j, weight => w,
                                  size => 0, maxfn => 0)])
    []
    cmd (code => !remove, qid => ?j, etc)
              [(j isin flows) andalso (get(flows, j).size == 0)];
    GPS_tagging [...] (told, rold, delta_next, weights, fnset,
                       flows -- [get(flows, j)])
endproc
```

Figure 3.22. *Le processus qui simule GPS et attribue les numéros de service*

Les processus FIFO_Queues et Dropper

Le processus Dropper décide si un paquet fourni à FIFO_Queues doit être stocké ou non par celui-ci selon une politique particulière de décongestion. Nous renvoyons le lecteur à la section 3.3.1 pour plus de détails.

Le processus WFQ_Order

Ce processus (voir figure 3.23) désigne la file d'attente qui doit émettre un paquet. Il connaît tous les numéros de service présents dans les files, qu'il conserve dans l'ensemble RealFNSet. Contrairement au FNSet du processus GPS_tagging, il s'agit ici des numéros de service effectivement présents dans la file et non ceux qui y seraient au sens GPS. Cette connaissance permet à WFQ_Order de n'autoriser une émission qu'à partir d'une file (il pourrait y en avoir plusieurs) dont le premier paquet a ce numéro de service.

```
Process WFQ_Order [input, output : tagged_packet]
                  (RealFNSet : fn_state) is
input (qid => ?j, tag => ?fn, etc);
   WFQ_Order [...] (RealFNSet ++ [(tag => fn, qid => j)])
[]
output (qid => ?j, tag => ?fn, etc)
                    [(tag => fn, qid => j)
                     isin subset(RealFNSet, minfn (RealFNSet))];
   WFQ_Order [...] (RealFNSet -- [(tag => fn, qid => j)])
endproc

function subset (s : fn_state, fn : rational) : fn_state is
(* sous-ensemble de s contenant les paires ayant le numéro de
   service fn *)
if isempty (s) then []
else
   case nth (s, 0) in
       (tag => !fn, qid => ?j)
             -> [(tag => fn, qid => j)] ++ subset (tail (s), fn)
     | any : fn_flow -> subset (tail (s), fn)
   endcase
endif
endfun
```

Figure 3.23. *Processus qui désigne l'ordre d'émission des paquets*

3.4. Mise en œuvre des composants

Tous les mécanismes décrits dans la section 3.3 doivent être combinés pour fournir des QdS. Nous allons montrer comment le fonctionnement simplifié d'un routeur DiffServ pourrait être décrit.

La figure 3.24 montre une architecture générique en supposant qu'il s'agit d'un routeur de frontière aussi bien pour le trafic entrant que sortant. Ceci permet de placer toutes les fonctions. Un routeur de cœur de réseau serait beaucoup plus simple. L'élément central du routeur, la matrice de commutation (*Switch Fabric*), se charge du routage des paquets vers les bonnes sorties. En amont, on trouve les fonctions de classification et de contrôle de trafic. Dans DiffServ, la classification peut consister à ajouter une balise au paquet en fonction de sa classe, de façon à simplifier les traitements ultérieurs dans le routeur. Cette balise pourrait par exemple être utilisée comme numéro de file d'attente à utiliser à l'interface de sortie. Ainsi, la classe BE pourrait se voir attribuer le numéro 0, les classes AF_i (i = 1..4) les numéros 1..4, et la classe EF, le numéro 5. Selon la classe, le contrôleur de trafic sera de nature différente. Pour EF, il est inexistant. Pour EF, il détruira les paquets non conformes. Et pour AF_i, il marquera les paquets non conformes. Il conviendra donc de combiner plusieurs contrôleurs et de leur attribuer les flux à contrôler. Ceci est montré à la figure 3.25 où l'on n'a montré qu'une classe AF pour la lisibilité. On y voit les contrôleurs des classes EF et AF. Ceux-ci n'opèrent que sur les paquets de leur classe grâce aux sélecteurs associés.

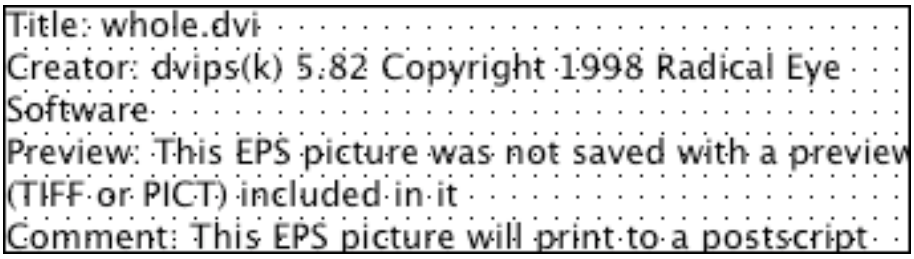

Figure 3.24. *Architecture d'un routeur DiffServ*

En aval du cœur du routeur, lorsque les paquets ont été commutés vers la bonne sortie, il faut stocker les paquets dans la file adéquate, y appliquer préalablement la politique de décongestion, organiser l'ordonnancement des files et éventuellement mettre en forme le trafic avant l'émission. Cette découpe est visible à la figure 3.24 car les paquets passent successivement par les processus *DSQueuingAndScheduling* et *DSShaping*.

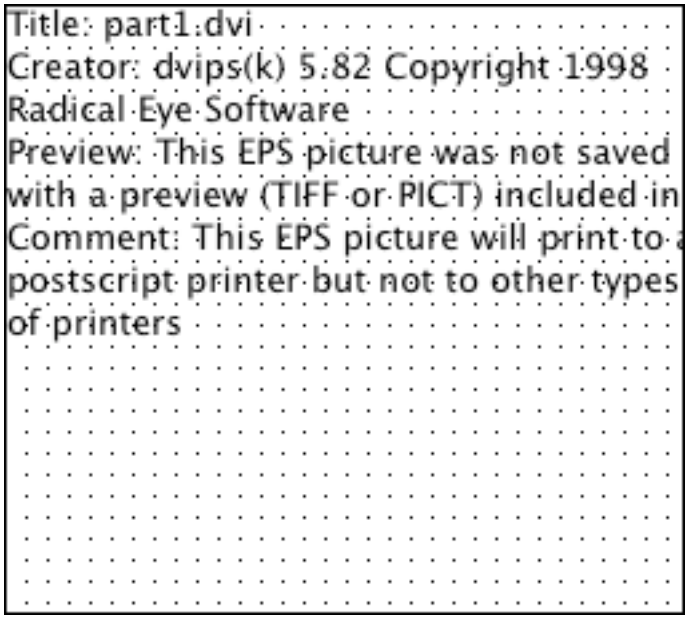

Figure 3.25. *Agencement des contrôleurs d'entrée DiffServ*

La figure 3.26 montre le premier plus en détail. On y voit les différentes politiques de décongestion des files et leur sélecteur. Pour EF, une simple file avec trop-plein suffit. Pour AF, la politique WRED permet de tenir compte des trois niveaux de précédence. Pour BE, on peut envisager un trop-plein ou encore RED comme indiqué. Les classes AF et BE sont servies par un ordonnanceur WFQ qui dispose d'une file et d'un poids par classe. L'ordonnanceur *PriorityQueuing* a pour but de servir la classe EF en priorité, tout en protégeant les autres classes contre une famine en n'appliquant cette priorité que si le trafic EF ne dépasse pas une certaine proportion de la bande passante de sortie. Son principe est donc simple : tant qu'il sait que la file EF n'est pas vide et que EF ne dépasse pas son quota, il prélève des paquets dans cette file, sinon il s'occupe des files AF et BE gérée par WFQ. Les deux ordonnanceurs forment donc une hiérarchie.

La figure 3.27 montre le cas d'une remise en forme des trafics EF et AF à la sortie du routeur. Même si ces deux régulateurs sont des seaux à jetons, ils n'ont certainement pas les mêmes paramètres, car les rafales émises par EF doivent être très courtes.

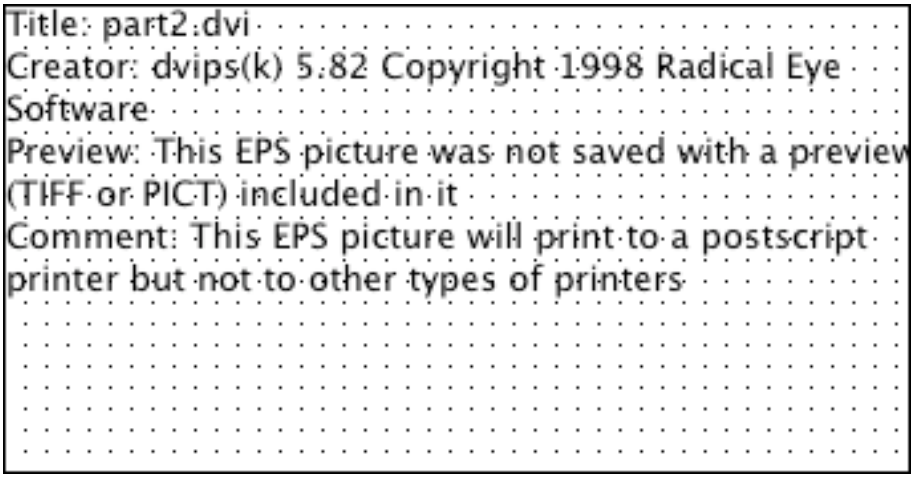

Figure 3.26. *Agencement des politiques de décongestion DiffServ*

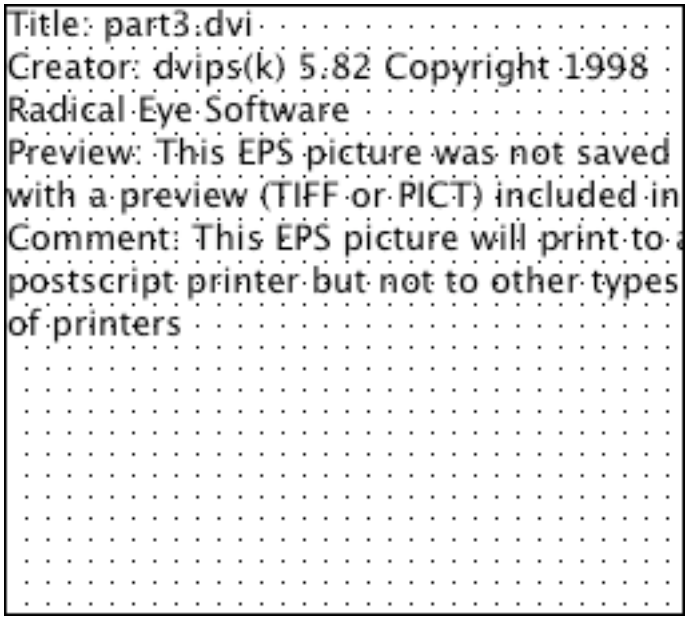

Figure 3.27. *Agencement des régulateurs de trafic DiffServ*

3.5. Conclusion

L'architecture de l'Internet évolue et devrait être en mesure de fournir des QdS différenciées et répondre ainsi aux besoins d'applications diverses. Cette évolution nécessaire sacrifie au passage la simplicité des mécanismes originels, pour les remplacer par des algorithmes plus sophistiqués. Nous avons vu que l'architecture DiffServ repose sur des mécanismes de régulation et de contrôle de trafic aux frontières des réseaux, et sur des politiques d'ordonnancement et décongestion dans les routeurs.

Nous avons décrit les algorithmes les plus connus, comme le seau à jetons (ou *token bucket*), RED, RIO et WFQ, et nous les avons formalisés à l'aide du langage ISO E-LOTOS. Enfin, nous avons montré comment ces algorithmes pouvaient être composés pour former des architectures aussi complexes que DiffServ.

Au passage, nous montrons que E-LOTOS dispose de toutes les facilités requises pour décrire ces algorithmes de façon compacte, élégante et structurée. Cependant, l'intérêt principal d'un tel langage est qu'il permet une analyse automatique de certaines propriétés. Des outils commencent à exister pour une variante de E-LOTOS, appelée LOTOS-NT [Sig 99]. L'outil TRAIAN peut effectuer les opérations suivantes :

- Analyse lexicale et syntaxique.
- Analyse sémantique statique, y compris pour les modules.
- Mise à plat des modules génériques.
- Génération de code C pour les types de données.

Une compilation de E-LOTOS vers des automates temporisés ouvrirait d'autres perspectives intéressantes. L'idée est bien sûr de définir une correspondence entre E-LOTOS et ces automates, afin de bénéficier des possibilités de *model-checking* offertes par ces derniers. Dans [Her 98], un sous-ensemble du noyau temporisé de E-LOTOS, appelé ET_LOTOS [LL 97, LL 98a, LL 98b], est compilé en un modèle d'automate hybride. Ce modèle est une extension des automates temporisés, et peut être analysé par simulations. De plus, dans certaines conditions, il est possible de le convertir en un automate temporisé plus classique pour lequel des outils de vérification existent (p.ex. HyTech [HHW+ 95], Kronos [DOT+ 96] et UPPAAL [LPW 97]).

Bibliographie

[ACE+ 00] AWDUCHE D., CHIU A., ELWALID A., WIDJAJA I., XIAO X., *A Framework for Internet Traffic Engineering,* Internet draft draft-ietf-tewg-framework-01.txt, May 2000.

[AMA+ 99] AWDUCHE D., MALCOLM J., AGOGBUA J. O'DELL M., MCMANUS J., *Requirements for Traffic Engineering Over MPLS,* RFC 2702, September 1999.

[BCS94] BRADEN R., CLARK D., SHENKER S., *Integrated Services in the Internet Architecture: an Overview,* RFC 1633, June 1994.

[BLA98] BLAKE ET AL., *An Architecture for Differentiated Services,* RFC 2475, Dec. 1998.

[BRA97] BRADEN R. ET AL., *Resource ReSerVation Protocol (RSVP) — Version 1 Functional Specification,* RFC 2205, Sept. 1997.

[CDF+ 99] CALLON R., DOOLAN P., FELDMAN N., FREDETTE A., SWALLOW G., VISWANATHAN A., *A Framework for Multiprotocol Label Switching,* Internet draft draft-ietf-mpls-framework-05.txt, September 1999.

[CW 97] CLARK D., WROCLAWSKI J., *An Approach to Service Allocation in the Internet,* Internet draft draft-clark-different-svc-alloc-00.txt, July 1997.

[CF 97] CLARK D., FANG W., *Explicit Allocation of Best Effort Packet Delivery Service,* Internet draft, September 1997.

[deP 93] DE PRYCKER M., *Asynchronous Transfer Mode - Solution for Broadband ISDN, 2nd edition,* Ellis Horwood, 1993.

[DKS 90] DEMERS A., KESHAV S., SHENKER S., *Analysis and Simulation of a Fair Queueing Algorithm,* J. Internetworking Res. and Experience, Vol. 1, 3-26, Oct. 1990. Egalement dans les actes d'ACM SIGCOMM'89, 3-12.

[DOT+ 96] DAWS C., OLIVERO A., TRIPAKIS S., YOVINE S., *The tool Kronos,* Hybrid Systems III, Verification and Control, LNCS 1066, Springer, 1996.

[FJ 93] FLOYD S., JACOBSON V., *Random Early Detection Gateways for Congestion Avoidance,* IEEE/ACM Transactions on Networking, Vol. 1, No. 4, 397-413, August 1993.

[HBW+ 99] HEINANEN J., BAKER F., WEISS W., WROCLAWSKI J., *Assured Fowarding PHB Group,* RFC 2597, June 1999.

[Her 98] HERNALSTEEN C.,*Specification, Validation and Verification of Real-Time Systems in ET-LOTOS,* Thèse de doctorat, Université Libre de Bruxelles, 1998.

[HHW+ 95] HENZINGER T., HO P.-H., WONG-TOI H., *A user guide to HyTech,* TACAS'95: Tools and Algorithms for the Construction and Analysis of Systems, Larsen K., Margaria T., Brinksma E., Cleaveland W., Steffen B. (eds), LNCS 1019, Springer, Berlin, 1995.

[ISO] ISO/IEC JTC1/SC7, *E-LOTOS,* ISO/IEC 15437, FDIS, July 2000.

[JNP 99] JACOBSON V., NICHOLS K., PODURI K., *Expedited Fowarding PHB,* RFC 2598, June 1999.

[Kes 97] KESHAV S., *An Engineering Approach to Computer Networking,* Addison-Wesley, 1997.

[LL 97] LEONARD L., LEDUC G., *An introduction to ET-LOTOS for the description of time-sensitive systems*, Computer Networks and ISDN Systems 29 (3) (1997), 271-292.

[LL 98a] LEONARD L., LEDUC G., *A Formal Definition of Time in LOTOS,* Formal Aspects of Computing (1998) 10: 248-266.

[LL 98b] LEONARD L., LEDUC G., *A Formal Definition of Time in LOTOS,* Formal Aspects of Computing (1998), 10E: 28-96, http://www.link.springer.de/link/service/journals/00165/supp/1998/8010030248.pdf

[LPW+ 97] LARSEN K., PETTERSSON P., WANG Y., *Uppaal in a nutshell,* International Journal of Software Tools for Technology Transfer, vol. 1, 1997.

[NIC 98] NICHOLS K. ET AL., *Definition of the Differentiated Services Field (DS Field) in the IPv4 and IPv6 Headers,* RFC 2474, Dec. 1998.

[PG 93] PAREKH A., GALLAGER R., *A Generalized Processor Sharing Approach to Flow Control in Integrated Services Networks: The Single-Node Case,* IEEE/ACM Transactions on Networking, Vol. 1, No. 3, 344-357, June 1993.

[PG 94] PAREKH A., GALLAGER R., *A Generalized Processor Sharing Approach to Flow Control in Integrated Services Networks: The Multiple Node Case,* IEEE/ACM Transactions on Networking, Vol. 2, No. 2, 137-150, April 1994.

[RVC 99] ROSEN E., VISWANATHAN A., CALLON R., *Multiprotocol Label Switching Architecture,* Internet draft draft-ietf-mpls-arch-06.txt, August 1999.

[Sig 99] SIGHIREANU M., *Contribution à la définition et à l'implémentation du langage Extended LOTOS,* Thèse de doctorat, Université Joseph Fourier, Grenoble, France, 1999.

[TUR 86] TURNER J., *New Directions in Communications, or Which Way to the Information Age?,* IEEE Communication Magazine, vol. 24, 28-15, 1986.

[XN 99] XIAO X., NI L., *Internet QoS: A Big Picture,* IEEE Network, March/April 1999, 8-18.

Index

Glossaire

AF…… Assured Forwarding
DiffServ ……Services différenciés
DSCP…… DiffServ Code Point
EF ……Expedited Forwarding

E-LOTOS ..... Enhanced LOTOS
GPS ..... Generalized Processor Sharing
IETF ..... Internet Engineering Task Force
IntServ ..... Services intégrés
ISP ..... Internet Service Provider
LOTOS ..... Language of Temporal Ordering Specification
PHB ..... Per Hop Behaviour
RED ..... Random Early Detection
RIO ..... RED with In and Out
RSVP ..... Resource ReSerVation Protocol
QdS ..... Qualité de Service
SLA ..... Service Level Agreement
SLS ..... Service Level Specification
WFQ ..... Weighted Fair Queuing
WRED ..... Weighted RED