Universitat de Lleida
Escola Politècnica Superior
Enginyeria en Informàtica

Sistemes Informàtics
Treball de Final de Carrera

Estudi/Implementació de la Qualitat de Servei en els entorns de xarxa i CPU

Autor: Alfred Fraile Agustí
Director: Francesc Giné de Sola

Setembre de 2007

Índex

Índex **1**

Índex de figures **4**

Índex de taules **6**

Índex d'algorismes **7**

1 Introducció **8**

1.1 La QoS en l'entorn de xarxa . . . 9

1.2 La QoS en l'entorn de CPU . . . 10

1.3 Objectius . . . 10

1.4 Entorn de treball . . . 11

1.4.1 Hardware . . . 11

1.4.2 El Sistema Operatiu . . . 12

1.5 Continguts de la memòria . . . 14

2 La QoS en l'entorn de xarxa **16**

2.1 Introducció . . . 16

2.2 Implementació del protocol TCP/IP . . . 19

2.2.1 El dispositiu lògic net_device . . . 20

2.2.2 Els paquets a l'interior del kernel 23
2.3 Disciplines de cues per la gestió de l'ample de banda 26
2.3.1 Disciplines de cues sense classes 29
2.3.2 Disciplines de cues amb classes 35
2.3.3 Classificar paquets amb filtres 45
2.4 Experimentació 49
2.4.1 Resultats 50

3 La QoS en l'entorn de CPU **59**

3.1 Introducció 59
3.2 Els processos 60
3.2.1 El Bloc de Control de Procés 61
3.2.2 Estats 62
3.2.3 Creació i destrucció 63
3.2.4 Estructures organitzatives 65
3.2.5 Els fils d'execució 66
3.3 El planificador de processos 66
3.3.1 La cua d'execució 68
3.3.2 Despertant i adormint processos 70
3.3.3 L'algorisme de planificació 74
3.3.4 La temporització del planificador 79
3.3.5 La inicialització del planificador 80
3.3.6 La prioritat dels processos 81
3.4 El nou planificador de processos 84
3.5 Experimentació 88
3.5.1 Resultats 88

4 Conclusions i treball futur **98**

Bibliografia **100**

A Software **102**

A.1 Script QoS 102

A.1.1 Iproute 103

A.1.2 Patch Con Kolivas 103

A.1.3 Schedtool 104

A.2 Eina provesNETWORK.sh 104

A.2.1 Iperf 104

A.3 Benchmark InterBench 105

A.4 Eina provesCPU.sh 106

B Script QoS **107**

C Scripts de proves **115**

C.1 L'entorn de xarxa 115

C.2 L'entorn de CPU 119

Índex de figures

2.1 Estructura de capes de les comunicacions TCP/IP. 21
2.2 L'estructura sk_buff. 23
2.3 L'estructura sk_buff_head. 25
2.4 Diagrama de paquets amb disciplines de cues. 29
2.5 Funcionament Qdisc pfifo_fast. 30
2.6 Octet TOS (Type Of Service). 31
2.7 Funcionament Qdisc SFQ. 34
2.8 Exemple d'una estructura de Qdisc amb classes. 36
2.9 Funcionament Qdisc PRIO. 37
2.10 Exemple Qdisc PRIO. 39
2.11 Exemple Qdisc CBQ. 43
2.12 Exemple Qdisc HTB. 45
2.13 Qdisc del Script de QoS. 50
2.14 Prova xarxa: pfifo_fast 1. 51
2.15 Prova xarxa: pfifo_fast 2. 52
2.16 Prova xarxa: Script QoS 1. 53
2.17 Prova xarxa: Script QoS 2. 53
2.18 Prova xarxa: Script QoS 3. 54
2.19 Prova xarxa: Script QoS 4. 55

2.20 Prova xarxa: Script QoS 5. 55
2.21 Prova xarxa: Script QoS 6. 56
2.22 Prova xarxa: Script QoS 7. 57
2.23 Prova xarxa: Script QoS 8. 57
2.24 Prova xarxa: Script QoS 9. 58

3.1 Diagrama de processos en un sistema multifil. 60
3.2 Espai de memòria del nucli reservat a un procés. 62
3.3 Diagrama de flux d'estats d'un procés. 63
3.4 Representació d'una llista. 65
3.5 Planificador multiprocessador. 69
3.6 Implementació interna de la cua d'espera. 71
3.7 Latència mitjana de scheduling. 89
3.8 Prova CPU: Script QoS 1. 90
3.9 Prova CPU: Script QoS 2. 90
3.10 Prova CPU: Script QoS 3. 91
3.11 Prova CPU: Script QoS 6. 92
3.12 Prova CPU: Script QoS 7. 92
3.13 Prova CPU: Script QoS 8. 93
3.14 Prova CPU: Script QoS 9. 94
3.15 Prova CPU: Script QoS 10. 94
3.16 Prova CPU: Script QoS 11. 95
3.17 Prova CPU: Script QoS 12. 96
3.18 Prova CPU: Script QoS 13. 96
3.19 Prova CPU: Script QoS 14. 97

Índex de taules

2.1 Els quatre bits del camp TOS. 30
2.2 Assignació de bandes. 31

Índex d'algorismes

B.1 qos.sh 107

B.2 qosNETWORK.sh 109

B.3 qosCPU.sh 112

C.1 provesNETWORK.sh 115

C.2 provesCPU.sh 119

C.3 provesCPU.c 123

Capítol 1

Introducció

La **QoS** (*Quality of Service* o *Qualitat de Servei*) no gaudeix d'una definició comú o formal. Malgrat que existeix gran nombre de definicions en el món de les comunicacions, on la notació va sorgir inicialment per descriure característiques tècniques de transmissions, principalment no dependents del temps. Algunes definicions habituals de QoS són les següents:

- La *ITU* (International Telecommunication Union) [1] que defineix l'estàndard X.902, descriu la QoS com "A set of quality requirements on the collective behavior of one or more objects". Un nombre de paràmetres relacionats amb la QoS descriuen la velocitat i la fiabilitat de les transmissions de dades. Alguns d'aquests paràmetres són el retard, la taxa d'error, ...
- La *ATM Lexicon* [2] descriu la QoS com "A term which refers to the set of ATM performance parameters that characterize the traffic over a given virtual connection". Els paràmetres de QoS s'apliquen majoritariament a les capes inferiors del protocol, per tant, no poden observar-se o verificar-se des de l'aplicació. Alguns d'aquests paràmetres són *cell loss ratio*, *cell error rate*, ... En termes d'aquests paràmetres de QoS s'han definit cinc tipus de serveis diferents.
- La *CITR* (Canadian Institute for Telecommunication Research) [3] mostra una descripció més general de QoS per aplicacions que necessiten comunicació en temps real: "The set of those quantitative and qualitative characteristics of a distributed multimedia system, which are necessary in order to achieve the required functionality of an application". Podem entendre-ho com les característiques d'un servei que determinen el grau de satisfacció d'un usuari davant d'aquest servei.

D'acord amb l'última definició, podem afirmar que la QoS està integrada per un conjunt de tecnologies que permeten garantir en tot moment un determinat flux de dades, és a

dir, que realment disposin dels recursos suficients quan els sol·licitin.

El present TFC busca implantar la QoS en dos entorns diferents, el primer entorn és la xarxa, on l'objectiu serà garantir determinats amples de banda segons l'usuari. El segon entorn correspon a la CPU, on l'objectiu és desenvolupar un sistema avançat d'ús de la CPU segons l'usuari, realitzant la reserva de porcions CPU. D'aquesta manera els usuaris privilegiats del sistema, amb aquest tipus de QoS, podran obtenir una execució més eficient de les seves aplicacions.

1.1 La QoS en l'entorn de xarxa

Les limitacions que actualment presenta la xarxa són prou conegudes, algunes d'elles són la congestió i els colls d'ampolla. Des del punt de vista de l'usuari, aquest no està capacitat per resoldre aquests tipus de problemes plantejats. Però si que està capacitat per reduir l'impacte considerablement, si s'utilitzen eines d'alt nivell per a la gestió de tràfic IP.

Les eines de gestió del tràfic IP s'encarreguen, entre altres tasques, de la gestió de l'ample de banda. Actualment, es disposa d'un gran nombre de disciplines de cues, les quals poden utilitzar-se individualment o combinades, amb la finalitat de gestionar com es desitgi l'ample de banda disponible.

Usualment, les configuracions típiques de QoS intenten trobar una solució que s'aproximi a un funcionament òptim a nivell de xarxa, però cal destacar que no existeix una fórmula perfecta per aconseguir el *tràfic ideal*. Els objectius més comuns d'aquest tràfic són:

- *Mantenir una baixa latència pel tràfic interactiu*: evitar que el tràfic massiu utilitzi tot l'ample de banda, en cas contrari els serveis que requereixen una resposta ràpida perdran la interactivitat.
- *Permetre funcionar alguns serveis a velocitats raonables, mentre existeix tràfic massiu*: evitar que el tràfic escollit sigui ofegat pel tràfic massiu.
- *Assegurar-se que els enviaments no perjudiquin les descàrregues i viceversa*: s'observa molt sovint que el tràfic de sortida destrueix la velocitat de descàrrega.

Els objectius de la nostra QoS en l'entorn de xarxa disten d'aquests principis de funcionament. No es tracta d'oferir QoS a nivell d'aplicació en termes generals, com es descriu als punts mencionats anteriorment, sinó de garantir una determinada QoS a nivell d'usuaris del sistema. Els usuaris del sistema escollits per gaudir de la QoS disposaran d'un determinat ample de banda reservat, però si aquests usuaris no necessiten el recurs el repartiran equitativament entre les diferents connexions dels sistema.

1.2 La QoS en l'entorn de CPU

En un SO de temps compartit hi conviuen un gran nombre de processos, per tant, en neixen i moren constantment. Durant el cicle de vida de qualsevol procés aquest pot tenir necessitats de còmput, però els recursos de processament d'un computador són limitats. El planificador de processos és la part encarregada de repartir els recursos de processament del sistema entre el conjunt de processos amb necessitats de còmput.

El SO Linux proporciona un planificador de processos orientat i optimitzat per aplicacions de temps compartit, per tant, està limitat sota aquest principi. Aquest planificador no oferirà els mecanismes suficients per implementar un sistema QoS en l'entorn CPU.

Com la QoS en l'entorn de xarxa, els objectius de la nostra QoS en l'entorn de CPU no tracten d'oferir QoS a nivell d'aplicació en termes generals, sinó de garantir una determinada QoS a nivell d'usuaris del sistema. Els usuaris del sistema escollits per gaudir de la QoS, en funció de la càrrega del sistema, obtindran el temps mínim possible d'execució per una determinada aplicació.

Per altra banda, el planificador de processos proposat per **Con Kolivas** [14] oferirà mecanismes avançats que permetran la reserva de porcions CPU.

1.3 Objectius

Els objectius principals que es pretenen assolir amb aquest treball són:

- La QoS en l'entorn de xarxa:
  - Conèixer la implementació teòrica del protocol TCP/IP sota el SO Linux (nucli 2.6), amb la finalitat d'analitzar acuradament l'última capa representada a través del dispositiu de xarxa i com aquest s'implementa a través del dispositiu lògic net_device.
  - Un cop s'ha entès la implementació interna del dispositiu lògic net_device, fer un estudi exhaustiu de les diferents disciplines de cues per a la correcta gestió de l'ample de banda.
  - Assolits aquests coneixements, desenvolupar un sistema de QoS per establir una estructura de repartiment de l'ample de banda en funció dels usuaris del sistema. Aquesta implementació la trobem al **Script QoS**.
  - Implementar una eina (**provesNETWORK.sh**) que permeti realitzar les connexions de tràfic massiu indicades, sota la identitat d'un usuari determinat amb un interval de mostreig determinat durant un temps concret. La

finalitat d'aquesta eina consisteix en demostrar com la nova estructura de cua desenvolupada al **Script QoS** és molt superior a la disciplina de cua per defecte en termes de QoS. Els resultats són presentats en format gràfic.

- La QoS en l'entorn de CPU:
  - Conèixer a grans trets la implementació interna dels processos i fils d'execució sota el SO Linux (nucli 2.6), així com estudiar els cicles de vida dels processos i els canvis d'estat que sofreixen durant la seva execució.
  - Fer un estudi del funcionament intern del planificador de processos (nucli 2.6), així com argumentar com les polítiques de planificació estàndards no estan dotades dels mecanismes suficients per desenvolupar un sistema de QoS.
  - Fer un estudi de les diferències entre el planificador de processos estàndard i el planificador proposat per **Con Kolivas**. Analitzar les noves polítiques del nou planificador de processos, les quals estan dotades dels mecanismes suficients per desenvolupar un determinat sistema de QoS.
  - Assolits aquests coneixements, desenvolupar un sistema de QoS que permetrà, en funció de l'usuari, realitzar *reserves de porcions* de CPU i obtenir un menor temps d'execució en aquelles aplicacions privilegiades. Aquesta implementació la trobem al **Script QoS**.
  - Implementar una eina (**provesCPU.sh**) que s'encarrega de crear i gestionar una determinada quantitat de processos, els quals utilitzen la CPU massivament sota la identitat d'un usuari determinat, amb un interval de mostreig determinat durant un temps concret. La finalitat d'aquesta eina consisteix en demostrar com el nou planificador de processos utilitzat, conjuntament amb el **Script QoS**, és molt superior al planificador de processos estàndard en termes de QoS. Els resultats són presentats en format gràfic.

1.4 Entorn de treball

Aquesta secció està constituïda per dues parts. Per una banda, s'exposa l'equip hardware utilitzat per realitzar l'experimentació. Per altra banda, s'estudien les característiques principals del SO Linux (nucli 2.6), les quals proporcionen mecanismes suficients per instaurar la QoS sota els entorns de xarxa i CPU.

1.4.1 Hardware

Les proves experimentals s'han efectuat sobre dos computadors:

- **Computador 1**:
  - *Processador*: Intel Core 2 Duo - 1.66 GHz.
  - *Memòria principal*: 1 GHz - DDR2 SDRAM (533MHz).
  - *Memòria secundària*: 2 GBytes.
  - *Cache*: L2 de 2048Kbytes.
- **Computador 2**:
  - *Processador*: AMD Sempron 3000 - 1.6 GHz
  - *Memòria principal*: 1 GHz - DDR SDRAM (400MHz)
  - *Memòria secundària*: 2 GBytes.
  - *Cache*: L1 de 128kbytes i L2 de 256kbytes.

Aquests dos computadors estan interconnectats en una xarxa *ethernet* d'ample de banda de 100Mbps, mitjançant el dispositiu especial router i formant una topologia clàssica d'estrella.

1.4.2 El Sistema Operatiu

Linux és el nucli d'un SO de temps compartit. Va ser desenvolupat per Linus Torvalds l'any 1991 i actualment ell mateix també intervé en el seu manteniment i a més, és desenvolupat per un teixit dispers de programadors comunicats a través d'Internet. És software lliure i està sota la llicència GNU GPL. Disposar del codi font permet veure com funciona realment el nucli d'un SO complex. A més a més, el fet de poder modificar el codi, i posteriorment poder distribuir-lo sota la llicència GPL, permet poder experimentar amb ell.

Cal destacar que Linux està escrit gairebé en la seva totalitat en un llenguatge de programació d'alt nivell com és C. Això permet que sigui fàcilment portable a diferents arquitectures. La part del SO no escrita en C està escrita en assemblador i no és independent de l'arquitectura. Si naveguem a través del codi font del nucli podrem veure que les instruccions en assemblador estan inserides dintre de les funcions escrites en C. Això és possible gràcies al fet que Linux no està escrit en ANSI C estricte, sinó en una extensió del llenguatge anomenat GNU C. Aquesta variant de C és suportada pel compilador GCC i la seva característica més important, respecte el llenguatge C estàndard, és l'assemblador en línia, el qual permet l'inserció d'instruccions escrites en assemblador dins d'un programa en C. Aquestes instruccions podran utilitzar variables o macros del codi escrit en C.

Linux posseeix totes les característiques dels sistemes comercials utilitzats en entorns de treball més exigents, i es troba actualment en unes cotes d'eficàcia que el permeten ser uns dels protagonistes del mercat de servidors. Està cada cop més present als equips d'usuari domèstic. Aquest nivell de prestacions ha estat possible gràcies a la disponibilitat del seu codi font, que permet a qualsevol programador del món un accés total a la seva correcció i millora.

Algunes de les principals característiques són:

- *Multiprocés*: permet l'execució de vàries aplicacions simultàniament.
- *Multiusuari*: diferents usuaris poden accedir als recursos del sistema simultàniament.
- *Multiplataforma*: funciona amb la majoria de plataformes del mercat.
- *Shells*: programables, que el converteix en el Sistema Operatiu més flexible que existeix.
- *Dispositius*: suport per qualsevol tipus de dispositiu inclòs directament al nucli.
- *Fitxers*: suport per a la majoria de sistema de fitxers.

Un cop vista l'organització interna del SO Linux i les seves principals característiques, la versió del nucli més adient és la 2.6, donat que ofereix noves funcionalitats implementades respecte la versió 2.4. Algunes d'aquestes funcionalitats seran molt útils per desenvolupar un sistema de QoS sota els entorns de xarxa i CPU.

En referència al sistema de QoS en l'entorn de xarxa, el nucli incorpora una interfície per desenvolupar eines d'alt nivell per la gestió de paquets IP. Aquestes eines es desenvolupen a partir del socket anomenat *NETLINK*, el qual permet implementar en espai d'usuari codi de gestió de paquets i tràfic IP. Actualment, aquestes funcionalitats ja estan implementades i s'aconsegueixen, en gran mesura, gràcies al software **Iproute** (Apèndix A).

En referència al sistema de QoS en l'entorn de CPU, el nucli segueix una arquitectura modular, la qual evita els sistemes monolítics i permet que l'usuari pugui crear-se un nucli a la seva mida. Mitjançant aquesta característica podrem substituir el planificador de processos estàndard pel planificador proposat per **Con Kolivas** (Apèndix A).

Aquestes propietats el converteixen, segurament, en el SO més adient per desenvolupar un sistema de QoS sota els entorns de xarxa i CPU.

Els dos computadors descrits anteriorment disposen del SO Linux **Debian** sota els nuclis següents:

- **Computador 1**:
  - *2.6.21*: nucli estàndard.
  - *2.6.21.ck2*: nucli compilat, el qual s'ha obtingut a partir del codi font del nucli 2.6.21 més l'aplicació del *patch-2.6.21-ck2.bz2* per obtenir el nou planificador de processos utilitzat al capítol QoS en l'entorn de CPU.
- **Computador 2**:
  - *2.6.18*: nucli estàndard.

1.5 Continguts de la memòria

Els continguts que tracten els capítols d'aquesta memòria són els següents:

- **Capítol 2**: es dóna una visió general sobre la implementació del protocol TCP/IP sota el SO Linux (nucli 2.6). A continuació s'analitza a fons la implementació de l'última capa representada pel dispositiu lògic net_device, el qual implementa un sistema de disciplines de cues per a la gestió de l'ample de banda. S'analitzen les disciplines de cues (implementades al nucli 2.6), les quals permetran desenvolupar una estructura de cua (**Script QoS**) per instaurar un particular sistema de QoS en l'entorn de xarxa. Per demostrar la validesa de la nova estructura de cua vers la cua estàndard en termes de QoS en l'entorn de xarxa, es mostraran els resultats obtinguts amb el script de proves **provesNETWORK.sh**.
- **Capítol 3**: es dóna una visió general sobre els processos i els fils d'execució en el SO Linux. Es veu el cicle de vida d'un procés i els canvis d'estat que sofreix. A continuació s'explica el funcionament del planificador de processos (nucli 2.6), a partir d'una descripció a fons del conjunt d'algorismes i estructures de dades involucrades. Un cop analitzat el planificador de processos, s'explica el funcionament del planificador de processos desenvolupat per **Con Kolivas**, el qual permetrà instaurar un particular sistema de QoS en l'entorn de CPU. Per demostrar la validesa del conjunt format pel nou planificador de processos amb el **Script QoS**, vers el planificador estàndard en termes de QoS en l'entorn de CPU, es mostraran els resultats obtinguts amb el script de proves **provesCPU.sh**.
- **Capítol 4**: conclusions i treball futur; recull les conclusions extretes al finalitzar el TFC, per finalitzar amb futures línies d'investigació obertes en la realització del mateix.

- **Apèndix A**: es descriu el software utilitzat, on destaca el **Script QoS** i les eines desenvolupades per realitzar les proves (**provesNETWORK.sh** i **provesCPU.sh**). També es mostra com dotar el nucli del SO del nou planificador de processos.

- **Apèndix B**: mostra el codi font del **Script QoS** que engloba els scripts **qosNETWORK.sh** i **qosCPU.sh**, per poder instaurar un sistema de QoS en l'entorn de xarxa i CPU respectivament.

- **Apèndix C**: mostra el codi font de totes les eines desenvolupades per poder realitzar les proves del treball. L'apèndix C.1 mostra el codi de l'eina **provesNETWORK.sh** i l'apèndix C.2 mostra el codi de les eines **provesCPU.sh** i **provesCPU.c**.

Capítol 2

La QoS en l'entorn de xarxa

2.1 Introducció

Des de l'aparició d'Internet, aquesta gran xarxa de computadors ha revolucionat molts aspectes de la societat i de la vida de les persones. Des d'usuaris que utilitzen pocs minuts, passant pels aficionats als programes de missatgeria instantània, aplicacions P2P (*Peer Two Peer*), ... fins arribar als usuaris que necessiten una connexió a aquesta gran xarxa mundial per motius de treball. Tot plegat, ha esdevingut un canvi a les vides de moltes persones.

Internet es caracteritza per tenir un gran número d'avantatges que s'incrementen com més gran i global esdevé, però també compta amb alguns problemes, alguns d'ells cada cop són més presents, els quals posen al descobert alguns dels seus punts dèbils. Un d'aquests problemes consisteix en la congestió i els colls d'ampolla, que es tradueixen en connexions cada cop més lentes.

Un cop la connexió a Internet d'alta velocitat ha arribat a les llars, molts d'aquests problemes, que havien estat emmascarats, s'han convertit en més greus, degut al major ample de banda del que disposa cada usuari. Aquesta gran xarxa està articulada en base a unes màquines que dirigeixen el tràfic de tots els seus usuaris cap a altres xarxes, amb l'objectiu de trobar el seu destí. Aquestes màquines es coneixen com routers, els quals s'encarreguen de rebre tràfic i encaminar-lo segons unes regles preestablertes.

Els routers, encara que han evolucionat ràpidament gràcies als avenços tecnològics, i de ser capaços de gestionar el tràfic d'un gran número de màquines, tenen un les seves limitacions. Aquest límit depèn de molts factors i entre ells es troben els associats a la gestió de les seves cues. En l'actualitat, els routers solen funcionar amb una cua de tipus FIFO (*First Input First Output*). Segons van arribant els paquets al router, es van

col·locant al final de la cua, esperant a ser encaminats fins l'enllaç següent. El volum màxim d'aquesta cua determina la quantitat de paquets que serà capaç d'emmagatzemar el router a la seva memòria. El problema es presenta quan aquesta cua està plena i, al mateix temps, segueixen arribant més paquets des de la xarxa. A partir d'aquest moment, el router fins que no disposi de lloc per emmagatzemar aquests nous paquets, tot el tràfic que arriba serà eliminat.

Si ens aturem davant d'aquest detall, i imaginem una connexió que ha de passar per un gran número d'aquestes màquines, podrem entendre un dels problemes subjacents que existeixen. Les conseqüències d'aquesta pèrdua de paquets implica la necessitat de retransmissió del router citat, i les possibles noves pèrdues en routers successius. Podem comprendre que aquesta és una de les causes del ralentitzament de les nostres connexions.

Un cop vistos alguns dels problemes actuals de la xarxa, podem plantejar-nos trobar una forma de solucionar-los o almenys suavitzar-los. Actualment, el nucli del SO incorpora una interfície que permet implementar eines professionals d'alt nivell en qüestions de gestió de tràfic IP. Aquestes noves funcionalitats permeten realitzar tasques com:

- Creació de túnels IP.
- Ús de taules de routing múltiples.
- **Reserva d'ample de banda**.
- Monitorització de perifèrics, direccions i rutes.
- Gestió de taules ARP.
- Unificar comandes relacionades amb la gestió de tràfic IP.

Aquestes funcionalitats estaven disponibles en routers propietaris d'alta gamma i de preus molt elevats. El nucli de Linux ens permet implementar-les de forma molt més segura, més econòmica i amb més rendiment, a més de poder desenvolupar les nostres pròpies eines específiques.

L'idea principal consisteix en traslladar la cua d'enviament del router, el qual utilitza la política FIFO, al nostre computador que s'encarregarà de realitzar les tasques pròpies del router. D'aquesta forma podrem modelar i configurar la cua d'enviament al nostre gust, la qual cosa permetrà desenvolupar un sistema de QoS en l'entorn de xarxa.

La QoS en l'entorn de xarxa és un conjunt de protocols i tecnologies que garanteixen l'entrega de dades a través de la xarxa en un moment donat. A partir d'aquest principi assegurem que les aplicacions que requereixen un baix temps de latència, o un major

ample de banda, realment disposin dels recursos suficients quan els sol·licitin. Per aquest motiu, un dels principals objectius de la QoS és prioritzar unes connexions respecte d'altres.

Els beneficis que podem obtenir a l'introduir QoS al nostre sistema són:

- **Control sobre els recursos**: podem limitar l'ample de banda consumit per algunes transferències concretes i donar més prioritat a d'altres.
- **Ús més eficient dels recursos de xarxa**: al poder establir prioritats en funció dels tipus de servei, llindar de tasses de transferència, ...
- **Menor latència**: en aplicacions de tràfic interactiu, les quals requereixen un temps de resposta curt.

Els problemes d'una xarxa, a grans trets, són els descrits a continuació:

- **Alguns processos col·lapsen la connexió**: si tenim més d'un procés utilitzant la mateixa sortida a internet o xarxa local. Normalment alguna d'aquestes connexions creades per part d'aquests processos comencen a consumir molt ample de banda i acaparen la connexió, la qual cosa implica que els altres processos no tinguin quasi ample de banda amb el conseqüent augment de la latència.
- **Les connexions interactives**: si tenim la connexió a internet, o xarxa local, col·lapsada tal i, com s'ha explicat anteriorment, realitzar qualsevol tipus de connexió que requereixi interactivitat es converteix en una tasca impossible.
- **Els buffers del router/modem no donen abast**: actualment les connexions a internet o xarxa local són de banda ampla, això significa que hauríem de poder enviar i rebre a la vegada sense problemes. Però en la majoria d'ocasions quan se satura qualsevol dels dos canals, l'altre se'n recent. Aquesta situació és deguda a què els paquets que no pot enviar el router/modem els guarda en buffers interns. Aquests buffers solen estar bastant limitats i es comporten d'una manera que no pot controlar.

A grans trets, les solucions per evitar els problemes plantejats anteriorment són:

- **Repartiment equitatiu**: La forma d'evitar que una sola connexió col·lapsi tot l'ample de banda consisteix en realitzar una planificació i repartiment entre totes les connexions que requereixin d'ample de banda.

- **Prioritzant els tipus de trâfic**: Si volem que cert tipus de connexions tinguin prioritat sobre altres, caldrà classificar els seus paquets en una cua que tindrà prioritat respecte les altres.

- **Limitar el caudal que enviarem al router/modem**: traslladarem la cua d'enviament al nostre ordinador, el qual realitzarà les tasques pròpies del router, d'aquesta forma podem modelar-la i configurar-la al nostre gust, per gestionar l'ample de banda de manera més eficient. Els paquets es classificaran a l'ordinador, el qual realitzarà tasques pròpies de router abans d'arribar al mòdem. Per evitar saturar les cues d'enviament del mòdem, s'haurà de limitar lleugerament per sota de la velocitat de pujada.

A grans trets, la QoS en l'entorn de xarxa consistirà en establir una estructura de repartiment de l'ample de banda entre els diferents usuaris del sistema, a través de sistema de disciplines de cua disponibles al nucli del SO. L'objectiu serà analitzar el seu comportament i, basant-nos amb els resultats obtinguts, demostrar que l'estructura desenvolupada presenta millors característiques que FIFO, el qual tindria un impacte negatiu sobre els objectius de desitjats.

2.2 Implementació del protocol TCP/IP

Actualment, les comunicacions entre processos locals i remots s'implementen a partir d'una estructura basada en capes. Aquesta estructura permet realitzar una abstracció de les comunicacions com un conjunt individual de problemes i les seves solucions. Llavors, cada capa s'encarrega d'una determinada tasca de les comunicacions, amb les seves pròpies estructures i operacions. Les capes presenten la informació a la resta de nivells de comunicació de forma transparent.

El plantejament de les comunicacions per capes permet comunicar dos processos, independentment del Hardware que utilitzin els punts de comunicació. Les diferents capes del nucli del SO s'encarreguen de tractar aquesta informació.

Els *sockets* representen els extrems d'un enllaç entre dos processos que es volen comunicar. El SO aporta diferents tipus de *sockets* segons quin sigui el domini de les comunicacions. La família de direccions són *Unix Domain*, *AX25*, *IPX*, *Appeltalk*, *X25* i *INET*. Els més utilitzats són *Unix Domain* i *INET*, el qual s'encarrega de les comunicacions entre processos d'un mateix computador i de les comunicacions entre processos de diferents computadors a través de la família de protocols TCP/IP.

Quan un procés necessita comunicar-se amb un altre procés, local o remot, aquest procés s'encarrega de crear un canal de comunicació entre ell i l'altre procés. Aquest canal de

comunicacions es realitza mitjançant *sockets* i se'n crea un a cadascun dels costats.

La figura 2.1 mostra l'estructura de capes TCP/IP que implementa el SO Linux. La primera en entrar en funcionament és la capa **socket BSD**, la qual s'encarrega de crear els *sockets*. Defineix un conjunt d'operacions necessàries, per tal que aquest *socket* sigui accessible per la resta de capes. Aquesta capa representa cada *socket* mitjançant una estructura de tipus **socket**. Cada estructura forma part d'un *inode* i s'assigna un descriptor de fitxer. Aleshores, es converteix en un fitxer més del sistema de fitxers, tolerant als processos tractar per igual els *sockets* que els fitxers.

Per sota de la capa **socket BSD**, es troba la capa **socket INET**. Aquesta capa associa a cada estructura **socket** una altra estructura de tipus **sock**. Aquesta nova estructura, permet a la capa **socket INET** donar significat a les operacions que l'usuari cridi de la capa **socket BSD**, en funció de la família de direccions elegida. Assigna les funcions definides per la capa **socket BSD** a les seves pròpies per obtenir el resultat que la capa **socket INET** desitgi. Aquesta capa també realitza l'enrutament de les dades, alliberant i rebent les dades al protocol de xarxa **TCP**.

La capa **TCP** (*Transmision Control Protocol*) implementa un tipus de protocol amb control de transmissió per enllaços segurs. A continuació trobem la capa de xarxa, on trobem el protocol **IP** (*Internet Protocol*), el qual proporciona un servei de transmissió/recepció de paquets.

Per sota de la capa **IP** trobem la interfície amb els dispositius de xarxa. Aquesta capa representa cada dispositiu mitjançant una estructura **net_device**, la qual proporciona la informació i les operacions necessàries per poder comunicar-se amb els dispositius físics. Els dispositius físics formen l'últim escalafó en l'estructura de capes i la seva missió es transformar les dades per poder viatjar a través del mitjà físic.

2.2.1 El dispositiu lògic net_device

Un dispositiu de xarxa és una entitat que envia i rep paquets. Existeixen una gran varietat de dispositius HW creats amb aquesta finalitat i, per tant, una gran quantitat de formes de gestionar aquests dispositius. Per aquest motiu el SO crea una interfície genèrica, la qual permet implementar l'estructura de capes, independentment del dispositiu de xarxa utilitzat. D'aquesta forma, cada dispositiu de xarxa està representat per una estructura comú a tots, anomenada **net_device**. A través d'aquesta estructura s'accedeix a tota la informació del dispositiu físic que tingui associat. Per tant, l'estructura **net_device** és el dispositiu de xarxa lògic. El conjunt de tots els dispositius de xarxa lògics són gestionats, i accedits, per una llista anomenada *dev_base*. Els seus camps són els següents:

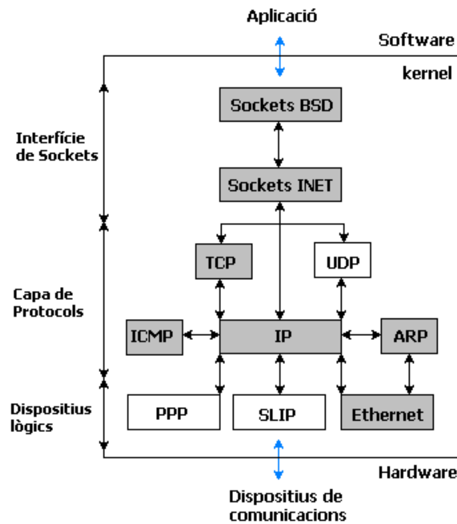

Figura 2.1: Estructura de capes de les comunicacions TCP/IP.

- *char *name*: és un camp únic que identifica un dispositiu en el sistema.
- *unsigned short base_addr*: defineix la direcció base del dispositiu.
- *unsigned char irq*: és el número d'interrupció del qual disposarà el dispositiu.
- *struct device *next*: s'utilitza per enllaçar tots els dispositius en la llista de dispositius *dev_base*.
- *int (*int) (struct device *dev)*: és la funció encarregada de la inicialització de l'estructura device i de l'obtenció de recursos per la mateixa. Aquesta funció es crida per cadascun dels dispositius en la *dev_base* durant la inicialització del sistema.
- *struct net_device_stats* (*gets_stats) (struct device *dev)*: permet obtenir les estadístiques del dispositiu en qualsevol moment. Únicament s'utilitza per a la generació d'una operació de lectura sobre l'entrada en el sistema de fitxers */proc/-net/dev*.
- *unsigned short flage, family, mtu*: són variables utilitzades pel protocol IP.
  - *Family*: indica la família de direccions que utilitza el dispositiu.
  - *MTU*: indica la mida màxima del paquet que el dispositiu pot llançar sense tenir en compte la capçalera d'ethernet. El valor *mtu* per una ethernet és de 1500 octets.

  - *Flags*: descriu el comportament del dispositiu.

- *unsigend short hard_header_len*: indica la longitud de la capçalera que afegeix el dispositiu.
- *void *priv*: és un punter a una estructura pròpia de cada dispositiu. Aquesta estructura normalment la proporciona el driver en la inicialització i permet l'accés a variables que mantenen el driver per a la gestió i control del dispositiu.
- *unsigned char dev_addr[MAX_ADDR_LEN]*: conté la direcció física, o MAC, de cada dispositiu.
- *unsigned char dev_len[MAX_ADDR_LEN]*: indica la longitud de la direcció física. MAX_ADDR_LEN permet una longitud màxima de 8 bytes.
- *unsigned char broadcast[MAX_ADDR_LEN]*: conté el valor que ha de sumar per enviar mitjançant broadcast.
- *int promiscuity*: indica si el dispositiu està funcionant en mode promiscu. És a dir, si captura tots els paquets que circulen per la xarxa, o solament els dirigits a la seva direcció MAC.
- *unsigned long tx_queue_len*: indica el número màxim de paquets permesos en les cues que la interfície, amb el dispositiu lògic, utilitza per emmagatzemar els paquets pendents de transmissió.
- **struct Qdisc *qdisc**: és la disciplina de cua associada al dispositiu. És el pas intermedi entre l'enviament per part del nucli i la sortida cap al driver. Implementa part del QoS destinat a prioritzar l'enviament de paquets cap a la xarxa.
- *int (*open) (struct device *dev)*: és la funció encarregada d'enviar i rebre paquets.
- *int (*stop) (struct device *dev)*: és la funció per les transferències, deshabilita les interrupcions i allibera recursos.
- *int (*hard_start_xmit) (struct sk_buff *skb, struct device *dev)*: és la funció que el dispositiu utilitza per alliberar paquets al dispositiu físic, enviant-los a la xarxa. De la gestió d'aquesta funció s'encarrega el driver.
- *int (*hard_header) (struct sk_buff *skb, struct net_device *dev, unsigned short type, void *saddr, unsigned len)*: és la funció cridada pel protocol IP per generar les capçaleres ethernet al buffer skb.
- *int (*rebuild_header) (struct sk_buff *skb)*: és la funció utilitzada per actualitzar les dades de la capçalera física del buffer, amb les dades contingudes en l'estructura **net_device**.

2.2.2 Els paquets a l'interior del kernel

L'estructura sk_buff

Un dels principals problemes, a l'hora de treballar amb múltiples capes de protocols, és la gestió dels buffers de dades. Cada protocol necessita afegir o extreure informació dels buffers, tant en transmissions com les recepcions. El pas dels buffers entre les diferents capes, i la localització de la informació que correspon a cada protocol, són punts claus en l'eficiència de les comunicacions.

La solució que Linux implementa es denomina **sk_buff**. Els **sk_buffs** són el mitjà de transport que utilitzen els paquets per creuar el kernel. Un sk_buff és una estructura de dades formada per uns camps de control i un bloc de dades associat. La gestió dels **sk_buffs** es produeix mitjançant un conjunt específic de funcions definides pel nucli. A grans trets, un **sk_buff** representa un paquet al kernel.

La figura 2.2 mostra l'esquema bàsic d'aquesta estructura i el conjunt de camps que proporciona el control sobre les dades contingudes al buffer.

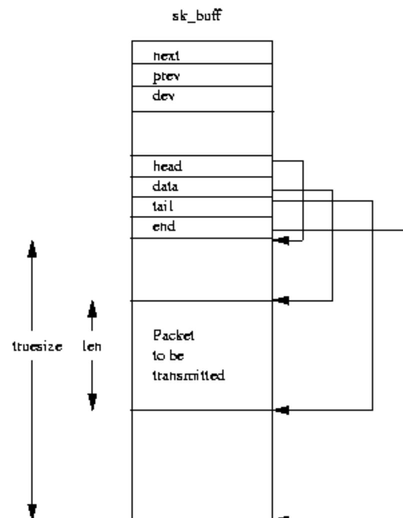

Figura 2.2: L'estructura sk_buff.

- *next* i *prev*: permeten encadenaments dels buffers en llistes circulars.
- *len*: indica la longitud actual de les dades a l'interior del buffer, *trusize* indica la mida total reservada per dades.

- *head*: és el punter a l'inici del bloc de dades. Data apunta al final de la capçalera i l'inici de l'àrea de dades.
- *tail*: és el punter al final del bloc de dades i end al final de l'espai de memòria reservat per dades.
- Altres camps interesants a destacar i que proporcionen informació sobre el buffer són:
  - *struct sock *sk*: indica el socket del qual prové el buffer.
  - *struct sock *dev*: és el punter del dispositiu de xarxa encarregat de la transmissió i recepció.
  - *struct sk_buff_head *list*: apunta al primer element de la llista al qual pertany, en aquest mateix instant de temps el *sk_buff*.
  - *union h*: conté l'apuntador a l'estructura que conté informació sobre la capçalera del paquet, respecte a la capa de transport. El punter *raw* conté la informació de la capçalera per sockets UNIX Domain.
  - *union nh*: conté l'apuntador a l'estructura que conté informació sobre la capçalera del paquet respecte la capa de xarxa.
  - *union mac*: conté l'apuntador a l'estructura que disposa de la informació sobre la capçalera del paquet respecte la capa d'enllaç.
  - *unsigned long priority*: indica la prioritat del paquet per tal de ser encuat en les disciplines de prioritat definides per la implementació de la Qdisc.
  - *unsigned short protocol*: indica el protocol utilitzat pel paquet. Aquest camp s'inicialitza per part del driver.

L'estructura sk_buff_head

Els paquets, durant el seu trajecte per les diferents capes del subsistema de comunicacions, són enllaçats a diferents llistes d'espera per ser utilitzats. Aquestes poden ser llistes de sockets, protocols, interfícies de dispositius lògics i físics. Les llistes on els buffers esperen, són llistes doblement enllaçades, on una estructura *sk_buff_head* representa l'inici de la mateixa. La variable *qlen* conté la mida de la llista que referència. La figura 2.3 la funcionalitat d'aquesta estructura.

Funcions per la gestió dels paquets

El conjunt de funcions, que el nucli defineix per gestionar els buffers, es pot dividir en tres grups: **gestió de dades**, **gestió de llistes d'espera** i **funcions d'alt nivell**.

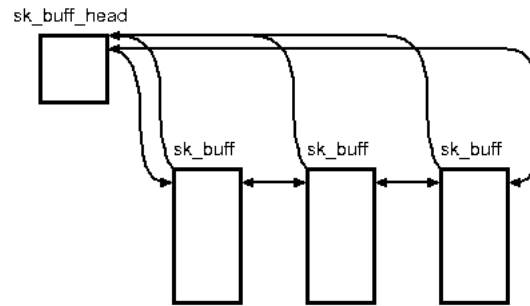

Figura 2.3: L'estructura sk_buff_head.

Funcions per la **gestió de dades**:

- *alloc_skb (tailspace+headspace)*: obté un buffer de *tailspace+headspace* bytes, amb zero bytes d'espai inicial, zero bytes de dades i *tailspace+headspace* bytes d'espai final (tail).
- *sk_reserve (headspace)*: reserva *headspace* bytes a l'espai inicial (head).
- *skb_push()*: afegeix dades o capçaleres al principi de les dades a transmetre. Mou el punter data a l'inici de l'àrea de dades i incrementa *len*.
- *skb_pull()*: borra les dades o capçaleres al principi de les dades rebudes. Mou el punter data al final de l'àrea de dades i decrementa *len*.
- *skb_put()*: afegeix dades o informació final de les dades a transmetre. Mou el punter tail fins al final de l'àrea de dades i incrementa *len*.
- *skb_trim()*: elimina les dades o informació del final de dades rebudes. Mou el punter tail al inici de l'àrea de dades i decrementa *len*.

El nucli ha de gestionar gran quantitat de buffers. En alguns casos, aquestos han de romandre en alguna llista abans de ser processats. El següent conjunt de funcions, entre moltes existents, ajuden a **gestionar llistes d'espera**:

- *skb_dequeue (struct sk_buff_head *list)*: retorna un punter al primer buffer de la llista. No allibera memòria.
- *skb_queue_empty (struct sk_buff_head *list)*: retorna si la llista està buida.
- *skb_queue_head_init (struct sk_buff_head *list)*: inicialitza la capçalera de la llista.

- *skb_queue_len (struct sk_buff_head *list)*: retorna el número de buffers que hi ha en una llista.
- *skb_queue_tail (struct sk_buff *skb)*: afegeix el buffer al final de la llista.
- *skb_ulink (struct sk_buff *skb)*: elimina un buffer de la llista d'on està situat, però sense alliberar memòria.

Les **funcions d'alt nivell** s'utilitzen per treballar amb els buffers en nivells més alts de l'estructura de capes. Són utilitzats per protocols, per si necessiten modificar directament els buffers o enviar-los a altres llistes.

- *sock_queue_rcv_skb*: és utilitzada pels protocols per emmagatzemar els *sk_buffs* provinents de la interfície física, a la cua de recepció del socket corresponent. En aquells casos en què no existeix suficient espai a la cua de recepció, els *sk_buffs* es descarten.
- *sock_alloc_send_skb*: reserva memòria destinada als *sk_buffs* de les aplicacions d'usuari, mentre existeixi espai en memòria.
- *kfree_skb()*: allibera la memòria ocupada, en el moment en què es lliura un paquet al dispositiu físic i el receptor confirma la seva correcta recepció. A continuació es comprova si està associat a la cua d'algun socket (*skb->sk*), per tal d'indicar al socket que actualitzi la capacitat de memòria disponible de la seva cua.

2.3 Disciplines de cues per la gestió de l'ample de banda

Les disciplines de cues, a través de l'encuament dels paquets, determinem el mode com s'envien les dades. A través de la nostra interfície de xarxa *ethernet*, únicament estem capacitats en seleccionar tot el que enviem. Seguint el funcionament d'Internet no disposarem mai del control directe de les dades que ens envien. Per altra banda, Internet es basa en el protocol TCP/IP, el qual proporcionarà algunes característiques per contrarestar aquests inconvenients.

El protocol TCP/IP no disposa de cap mecanisme per esbrinar la capacitat de la xarxa entre dos sistemes, de forma que simplement comença a enviar dades cada cop més ràpid fins que comença a perdre paquets. Donat que no existeix espai suficient per enviar-los tots junts, redueix la velocitat.

La comunicació entre la interfície de xarxa i el router s'ha de realitzar de manera que ens assegurem la tramesa de dades al router. Seguint aquesta idea, podem controlar el coll d'ampolla que es produeix si s'envien més dades al router de les que està capacitat a tractar. Per altra banda, el responsable real de controlar l'enllaç, i ajustar l'ample de banda, continua sent el router. Per tant, necessitem crear un sistema de cues que passi les dades de la interfície de xarxa al router d'acord als requeriments de QoS dels usuaris del sistema.

Com s'ha vist en la implementació que realitza el SO Linux de la pila TCP/IP, cada estructura **net_device** conté una disciplina de cua associada que controla com són tractats els paquets encuats en aquest dispositiu. Cada *net_device* diferent conté una disciplina de cua associada i representada per l'estructura **Qdisc**, utilitzada per la QoS.

La QoS amb Linux està construïda mitjançant una complexa combinació de disciplines de cues, classes i filtres que controlen els paquets que són enviats cap a la xarxa. La complexitat d'una disciplina de cua varia des d'una simple FIFO a una disciplina que utilitzi filtres per diferenciar entre diferents tipus de paquets, per processar-los segons els criteris desitjats. Per entendre correctament les configuracions, haurem d'introduir primerament alguns conceptes fonamentals:

- **Disciplina de cua** (Qdisc): és l'algoritme que gestiona el procés d'encuar paquets en un dispositiu, usualment una interfície de xarxa. Aquesta gestió pot ser tant a la cua d'entrada (*ingress*) o sortida (*egress*), però habitualment únicament s'utilitza la darrera. Cada Qdisc conté internament la Qdisc arrel (*root qdisc*) que no conté subdivisions internes configurables.

- **Qdisc sense classes**: és aquella Qdisc que no admet una subdivisió interna que pugui ser configurada per l'usuari. S'encarrega d'acceptar paquets i es limiten a reordenar-los, endarrerir-los o descartar-los.

- **Qdisc amb classes**: una Qdisc amb classes conté múltiples classes. Algunes d'elles contenen altres Qdisc, que a la vegada poden ser amb classes o no. S'utilitza per realitzar tractaments de tràfic diferents.

- **Classes**: una Qdisc amb classes pot contenir moltes classes, cadascuna de les quals és interna. Una classe està capacitada per contenir diverses classes. De forma que una classe pot tenir com a pare una Qdisc o una altra classe. Una classe terminal o fulla és una classe que no té classes filles, i conté una Qdisc adjunta. Aquesta Qdisc és responsable d'enviar dades a la classe. Quan es crea una classe, s'adjunta per defecte la Qdisc **pfifo_fast**. Quan s'afegeix una classe filla, aquesta Qdisc s'elimina. Per classes terminals, aquesta Qdisc pot reemplaçar-se per una altra més adequada. Fins i tot es pot reemplaçar aquesta Qdisc per una classe de forma que puguem afegir més classes.

- **Classificador** i **Filtres**: cada Qdisc amb classes necessita determinar a quina classe necessita enviar paquet, aquest objectiu s'aconsegueix utilitzant el classificador. Els filtres realitzen la classificació que conté vàries condicions que es poden cumplir.

- Altres conceptes implicats en el context de les Qdisc són:

  - **Scheduling**: una Qdisc amb el suport d'un classificador és capaç de decidir quins paquets necessiten sortir abans que altres. Aquest procés es denomina scheduling o reordenament.
  - **Shaping**: és el procés d'endarreriment de paquets abans de la seva sortida, per tal que el tràfic respecti una taxa màxima configurada. El shapping es realitza durant la sortida. Usualment, el descart de paquets per alentir el tràfic també s'anomena *shapping*.
  - **Policing**: permet descartar paquets perquè el tràfic es mantingui per sota d'un ample de banda configurat.
  - **Conservativa de treball**: una Qdisc conservativa de treball sempre distribueix paquets si estan disponibles. És a dir, mai endarrereix un paquet si l'adaptador de xarxa està preparat per enviar-lo.
  - **No conservativa de treball**: algunes cues poden necessitar endarrerir un paquet durant un cert temps per limitar l'ample de banda. Això significa que algunes vegades no enviaran un paquet, encara que estiguin disponibles.

La presència de classes, i la semàntica de les mateixes, són propietats fonamentals de les Qdisc. Per altra banda, els filtres poden ser combinats arbitràriament amb les classes i les Qdisc, sempre i quan aquestes tinguin classes, donat que no totes les Qdisc estan associades a alguna classe. Els filtres són utilitzats per una Qdisc per assignar els paquets provinents del nucli a alguna de les classes. Els paquets que arriben a la Qdisc són seleccionats pel filtre per tal de ser encuats cap a una classe determinada. La cua on s'introdueixen és propietat de la classe a la qual pertany. Aquests paquets poden ser filtrats i posats en més classes internes que la primera. Aleshores, el nucli demana enviar un paquet, que sortirà de la cua més interna de la classe que correspon, i serà enviat a través de la interfície de xarxa.

Un cop coneguda la terminologia bàsica, la figura 2.4 mostra el recorregut que segueix un paquet, des que arriba al nucli fins que l'abandona, a través de la interfície de xarxa:

1. El paquet arriba al nucli del SO.
2. (Pas Opcional) Si s'utilitza el SW **Netfilter** (**iptables**) s'encarrega de posar una marca al paquet segons les condicions desitjades. Aquesta marca és necessària, si

np el SW **Iproute** (**tc**) no podria classificar adequadament el paquet durant el pas següent.

3. El paquet arriba a la Qdisc, on s'analitza a través dels diferents filtres establerts. Si es produeix alguna coincidència, aleshores s'envia a la classe que indica el filtre.

4. Un cop el paquet surt de la Qdisc, aleshores torna al nucli, el qual s'encarrega d'enviar-lo a través de la interfície de xarxa.

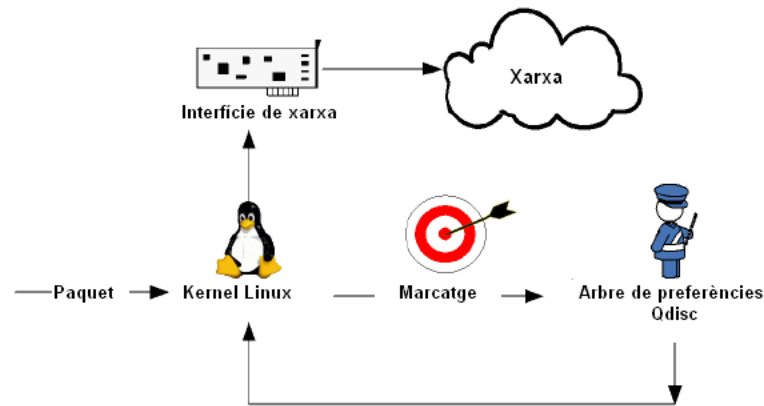

Figura 2.4: Diagrama de paquets amb disciplines de cues.

2.3.1 Disciplines de cues sense classes

Mitjançant les disciplines de cues canviem la forma de tramesa de les dades. Les disciplines de cua sense classes són aquelles que accepten dades i es limiten a reordenar-les, endarrerir-les o descartar-les. A partir d'aquesta idea podem ajustar el tràfic d'una interfície sencera. Sense haver de realitzar subdivisions internes.

La disciplina per utilitzada defecte pel nucli de Linux, i la més utilitzada, és la Qdisc **pfifo_fast**. Les altres cues no representen simplement una altra col·lecció de cues, sinó un conjunt de cues robustes amb característiques pròpies. Cadascuna d'aquestes cues proporcionarà punts forts i dèbils segons la funcionalitat desitjada.

La Qdisc pfifo_fast

Aquesta cua segueix el funcionament de FIFO, la qual cosa significa que cap paquet rep un tractament especial. Aquesta cua està constituïda per tres bandes. Una banda és

una classe, però no és configurable per part de l'usuari. Dins de cada banda, s'apliquen les regles FIFO, el nucli obeeix la marca **TOS** (*Type Of Service*) dels paquets TCP/IP; però, no es processarà una banda fins que l'anterior estigui finalitzada completament. La figura 2.5 mostra el funcionament intern.

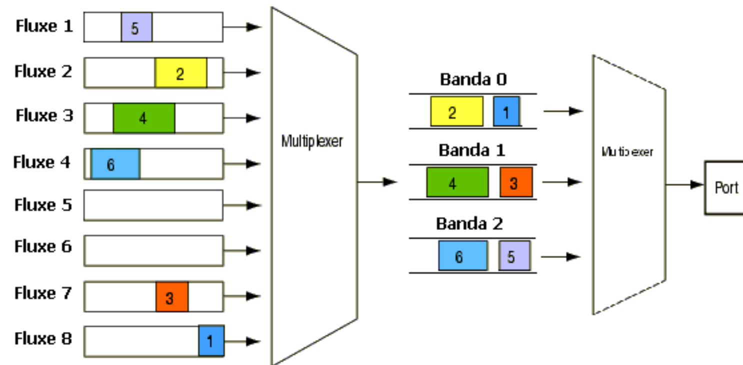

Figura 2.5: Funcionament Qdisc pfifo_fast.

La Qdisc **pfifo_fast** no es pot configurar, però disposa de paràmetres i formes d'ús concretes. Aquesta Qdisc disposa d'una cua configurada per defecte. Configuració de sèrie:

- **Priomap**: determina les prioritats dels paquets. Segueix els mecanismes d'assignació del nucli a les bandes. Aquesta correspondència es basa amb l'octet *TOS* del paquet TCP/IP. La taula 2.1 mostra els quatre bits del camp TOS:

Binari	Decimal	Significat
1000	8	Minimitzar retard (md)
0100	4	Maximitzar transferència (mt) salts
0010	2	Maximitzar fiablitat (mr)
0001	1	Minimitzar el cost monetari (mmc)
0000	0	Servei normal

Taula 2.1: Els quatre bits del camp TOS.

La figura 2.6 mostra l'estructura interna d'aquest octet. L'octet *TOS* està format pel camp TOS de quatre bits, però a la seva dreta conté el camp *MBZ* d'un bit. Aleshores el valor real del camp *TOS* és el doble del valor dels seus bits.

La taula 2.2 mostra aquest fet, el qual permetrà esbrinar a quina banda s'envia un determinat paquet. La primera columna indica el valor doble del nou camp *TOS*.

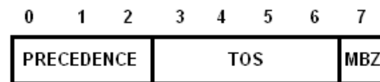

Figura 2.6: Octet TOS (Type Of Service).

La segona columna conté el valor dels quatre bits *TOS* rellevants, seguits per la tercera que conté significat traduït. Per exemple, el 10 significa que un paquet espera la màxima fiabilitat i mínim retard. La quarta columna indica la forma en què el nucli interpreta els bits del TOS, mostrant quina prioritat s'assigna. L'última columna indica el resultat final del paràmetre **priomap**. Per exemple, la prioritat 4 s'assigna la banda 1.

TOS	Bits	Significat	Prioritat Linux	Banda
0x0	0	Servei normal	0 Millor esforç	1
0x2	1	Minimitzar cost monetari	1 Farciment	2
0x4	2	Maximitzar fiabilitat	0 Millor esforç	1
0x6	3	mmc+mr	0 Millor esforç	1
0x8	4	Maximitzar transferència	2 En massa	2
0xa	5	mmc+mt	2 En massa	2
0xc	6	mr+mt	2 En massa	2
0xe	7	mmc+mr+mt	2 En massa	2
0x10	8	Minimitzar retards	6 Interactiu	0
0x12	9	mmc+md	6 Interactiu	0
0x14	10	mr+md	6 Interactiu	0
0x16	11	mmc+mr+md	6 Interactiu	0
0x18	12	mt+md	4 Int. en massa	1
0x1a	13	mmc+mt+md	4 Int. en massa	1
0x1c	14	mr+mt+md	4 Int. en massa	1
0x1e	15	mmc+mr+mt+md	4 Int. en massa	1

Taula 2.2: Assignació de bandes.

- **Txqueuelen**: la longitud d'aquesta cua s'obté a través de la configuració de la interfície de xarxa.

Com s'ha mencionat anteriorment, la Qdisc **pfifo_fast** està configurada per defecte, per tant, no es necessari executar cap instrucció. El codi següent mostra la configuració per defecte:

```
# tc qdisc show dev eth1
qdisc pfifo_fast 0: root bands 3 priomap 1 2 2 2 1 2 0 0 1 1 1 1 1 1 1 1
```

La Qdisc TBF

La **TBF** (*Token Bucket Filter*) és una Qdisc senzilla, limitada a deixar passar els paquets fins que no arribin a una taxa límit imposada administrativament. Aquesta Qdisc disposa de la possibilitat de permetre ràfegues curtes que excedeixin aquesta taxa. La implementació de **TBF** consisteix en un buffer, el qual s'omple constantment de peces virtuals d'informació denominades *tokens*. Aquest buffer s'omple a una taxa específica denominada *token-rate*. El paràmetre més important del buffer és la seva capacitat, la qual correspon al nombre de *tokens-capacitat* per emmagatzemar.

Cada *token* que arriba pren un paquet de dades entrant de la cua i s'elimina del buffer. L'associació d'aquest algoritme amb els dos fluxos (tokens i dades), proporciona tres possibles situacions:

- Les dades arriben a una taxa idèntica a la de *tokens* entrants. En aquest cas, cada paquet entrant disposa del seu *token* corresponent i passa a la cua sense endarreriment.
- Les dades arriben amb una taxa inferior a la dels *tokens*. Únicament una part dels *tokens* s'eliminaran per sortida de cada paquet, que s'envia fora la cua, de forma que s'acumulen els *tokens*, fins omplir del tot el buffer. Els *tokens* sense utilitzar poden enviar dades a velocitats superiors a la taxa dels *tokens*, la qual cosa produeix una curta ràfega de dades.
- Les dades arriben amb una taxa superior a la dels *token*. Això significa que el buffer es quedarà aviat sense *tokens*, la qual cosa causarà que la classe s'acceleri per una estona. Si es produeix aquesta situació, i continuen arribant paquets, se superarà el límit establert i els paquets es començaran a descartar.

Aquesta última situació és molt important, perquè permet ajustar administrativament l'ample de banda disponible a les dades que s'estan passant pel filtre. L'acumulació de *tokens* permet ràfegues curtes de dades extralimitades, perquè passin sense pèrdues. Però qualsevol sobrecàrrega restant ocasionarà que els paquets s'endarrereixin constantment i al final siguin descartats.

La Qdisc **TBF** disposa d'alguns paràmetres ajustables, però en la seva majoria de necessitats no haurem de modificar-los. Paràmetres disponibles en qualsevol moment:

- **limit** o **latency**: *limit* és el número de bytes que poden ser encuats a l'espera de *tokens* disponibles. També ho podem especificar a través del paràmetre *latency*, el qual indica el període màxim de temps que pot romandre un paquet al buffer. Aquest càlcul considera la grandària del buffer, la taxa i el *peakrate*, en cas d'estar configurat.

- **burst/buffer/maxburst**: mida del *buffer* en bytes. Aquesta és la màxima quantitat de bytes que poden contenir *tokens* disponibles instantàniament. En general, grans taxes necessiten grans buffers. Per altra banda, si el buffer és massa petit, es descartaran paquets.

- **MPU** (Minimum Packet Unit): determina l'ús mínim de *tokens* per paquet. Un paquet de mida zero no utilitza un ample de banda zero. En Ethernet cap paquet utilitza menys de 64 bytes.

- **rate**: valor de la velocitat mitjana que s'haurà de mantenir.

- Si el buffer conté *tokens* i està habilitat per buidar-se, llavors s'han de configurar els paràmetres següents:

  - **peakrate**: si disposem de *tokens*, i arriben paquets, per defecte s'enviaran immediatament. Si el buffer és de mida considerable, en alguns casos, pot produir un efecte indesitjable. Per altra banda, la taxa de pics la podem utilitzar per especificar la velocitat del buffer per buidar-se.
  - **mtu/minburst**: un *peakrate* inferior a la taxa normal és inútil. És possible tenir una taxa de pics superior enviant més paquets per fracció del temporitzador. El *peakrate* màxim disponible l'obtenim multiplicant la *MTU* configurada per l'arquitectura de la CPU.

Aquesta Qdisc no es configurable aïlladament, és a dir, haurà de configurar-se com alguna classe terminal d'una Qdisc amb classes. Per altra banda, podem posar un exemple per plasmar els conceptes d'aquesta cua. Suposem que tenim una Qdisc amb classes, la qual necessita configurar la classe 1:X, llavors hem de crear una classe terminal Y: per associar-li la Qdisc **TBF**. La sentència següent permet realitzar aquesta idea:

```
# tc qdisc add dev eth1 parent 1:X handle Y: tbf rate 0.5mbit
burst 5kb latency 70ms peakrate 1mbit minburst 1540
```

La Qdisc **TBF** està configurada amb un *rate* de 0.5Mbps, 5kb de buffer i un *peak rate* de 1.0Mbps per ràfegues curtes de paquets. La grandària de la cua de paquets es calcula a partir d'un màxim 70 ms de *latency* que un paquet estarà a la cua. El *minburst* se selecciona com la MTU de la interfície.

La Qdisc SFQ

La **SFQ** (*Stochastic Fairness Queuing*) és una implementació senzilla de la família d'algorismes de cues justes (*fair queuing*). És menys precís que altres, però també necessita menys càlculs, mentre que resulta ser gairebé perfectament just.

Com mostra la figura 2.7, el tràfic es divideix en un número bastant gran de cues FIFO, una per cada connexió. Aleshores s'envia el tràfic d'una manera semblant a *Round Robin*, donant a cada sessió per torns la oportunitat d'enviar dades. Aquest comportament és molt equitatiu i evita que una única connexió ofegui les altres. SFQ no crea una cua per cada sessió, sinó que disposa d'un algorisme que fracciona el tràfic en un número limitat de cues, utilitzant un algoritme de Hash.

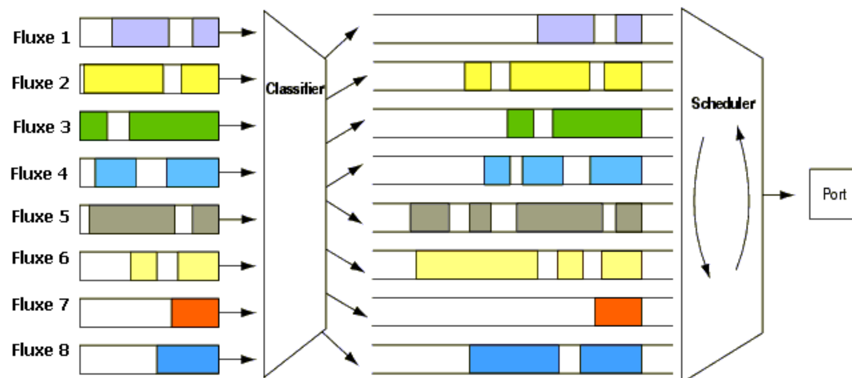

Figura 2.7: Funcionament Qdisc SFQ.

Degut al Hash, vàries sessions poden acabar a la mateixa cua, que dividirà per dos les possibilitats d'enviar un paquet per a cada sessió, reduint a la meitat, d'aquesta forma, la velocitat efectiva disponible. Per evitar que aquesta situació sigui detectable, SFQ canvia sovint l'algoritme de Hash de forma que dos sessions no col·lisionin gairebé mai.

Paràmetres disponibles en qualsevol moment:

- **pertub**: reconfigurar el Hash segons el temps establert. Si no s'indica res, el Hash no es reconfigurarà mai, per aquest motiu no és recomanable realitzar aquesta acció. El valor per defecte en segons és 10.
- **quantum**: quantitat en bytes d'un flux que es permeten treure d'una cua abans del torn de la següent cua.
- **limit**: número total de paquets que seran encuats per aquesta SFQ.

Configurar la disciplina SFQ, específicament com la Qdisc principal, no té molt sentit, perquè solament és útil si la interfície de xarxa de sortida està realment saturada. En cas contrari, no s'encuaran paquets i no tindrà cap efecte. Usualment es combina amb altres Qdisc per obtenir resultats òptims, però com mostra el codi podem configurar-la de la forma següent:

```
# tc qdisc add dev eth1 root sfq
# tc qdisc show dev eth1
qdisc sfq 8001: limit 128p quantum 1514b perturb 10sec
```

2.3.2 Disciplines de cues amb classes

Les Qdisc es caracteritzen per tenir una subdivisió interna en la seva estructura, la qual és molt útil si necessitem diferents tipus de tràfic, als quals desitgem donar un tracte per separat. El nucli del SO Linux proporciona un gran ventall de disciplines de cues amb classes, malgrat tot les més conegudes són les Qdisc's **PRIO**, **CBQ** i **HTB**.

El flux a l'interior de les Qdisc i les seves classes.

Quan entra tràfic a l'interior d'una Qdisc amb classes, és necessari enviar-lo a alguna de les seves classes. Aquest tràfic serà necessari classificar-lo. Per determinar quina acció realitzar amb un paquet, es consulta amb els filtres. És important remarcar, que els filtres són cridats des de l'interior de la Qdisc i no a l'inrevés.

Els filtres associats a aquesta Qdisc retornen una decisió, i aquesta s'utilitza per encuar el paquet en una de les classes. Cada subclasse pot provar altres filtres per veure si imparteixen més instruccions. En cas contrari, la classe encua el paquet en la Qdisc que conté.

Malgrat contenir altres Qdisc, la majoria de les Qdisc amb classes també realitzen *shaping*. Això és útil tan per reordenar paquets com per controlar la velocitat. Necessitarem això en cas de disposar d'una interfície a gran velocitat que envia a un dispositiu més lent.

La família Qdisc: Arrels, Controladors, Pares i Germans.

Qualsevol interfície configurada disposa d'una Qdisc arrel, la qual s'encarrega de comunicar-se amb el nucli del SO. A cada Qdisc i classe s'assigna un controlador (*handle*), que pot utilitzar-se en posteriors sentències de configuració per referir-se a la Qdisc.

Els controls d'aquestes Qdisc consisteixen en dos parts, un número major i un número menor: <major>:<menor>. És costum donar-li a la Qdisc d'arrel el nom 1: que és sinònim de 1:0. El numero menor d'una Qdisc sempre és 0.

Les classes han de tenir el mateix número superior als seus pares i aquest número superior ha de ser únic dins d'una mateixa configuració. El número inferior ha de ser únic dins d'una Qdisc i les seves classes.

La figura 2.8 mostra l'estructura jeràrquica de classes d'una Qdisc d'exemple. Com podem veure, la Qdisc arrel 1:0 té la classe 1:1 com a filla. La classe 1:1 disposa de les classes filles 1:10 i 1:20. La classe 1:10 té una classe terminal 10:, mentre que la classe 1:20 té una nova Qdisc. La nova Qdisc arrel 20:0 té dos classes terminals, que són 20:1 i 20:2.

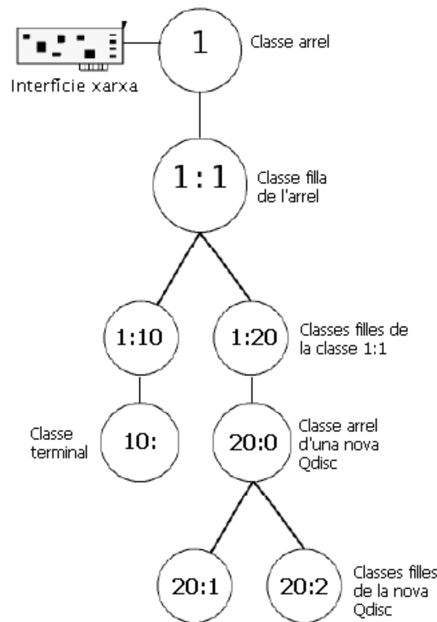

Figura 2.8: Exemple d'una estructura de Qdisc amb classes.

Un cop construït l'arbre, s'ha d'interpretar correctament. És incorrecte interpretar que el nucli està al capdamunt de l'arbre i la xarxa sota. Els paquets s'encuen i desencuen a l'interior de la Qdisc arrel, que és l'únic element que es comunica amb el nucli.

Desencuar els paquets per enviar-los a la interfície.

En el moment que el nucli decideix que necessita extreure paquets per enviar-los a una interfície, la Qdisc arrel 1:0 rep una petició de desencuar, que s'envia a la classe 1:1 que, a la seva vegada, la passa a les subclasses, cadascuna de les quals consulta als seus descendents, i intenta fer un desencuament sobre ells. Les classes niuades només es comuniquen amb les seves Qdisc paternes, i mai amb una interfície. Solament la Qdisc arrel rep peticions de desencuat per part del nucli.

La Qdisc PRIO

La Qdisc **PRIO** en realitat no realitza ajustaments, sinó que simplement subdivideix el tràfic a partir de la configuració dels filtres. Podem considerar la Qdisc **PRIO** com una **pfifo_fast** optimitzada, on cada banda és una classe separada, en lloc d'una simple FIFO.

Quan s'encua un paquet a la Qdisc PRIO, s'escull una classe segons les ordres de filtrat que s'hagin donat. Per defecte es creen tres bandes i aquestes són purament cues FIFO sense estructura interna, però poden substituir-se per qualsevol altra Qdisc disponible. La figura 2.9 mostra el seu funcionament intern.

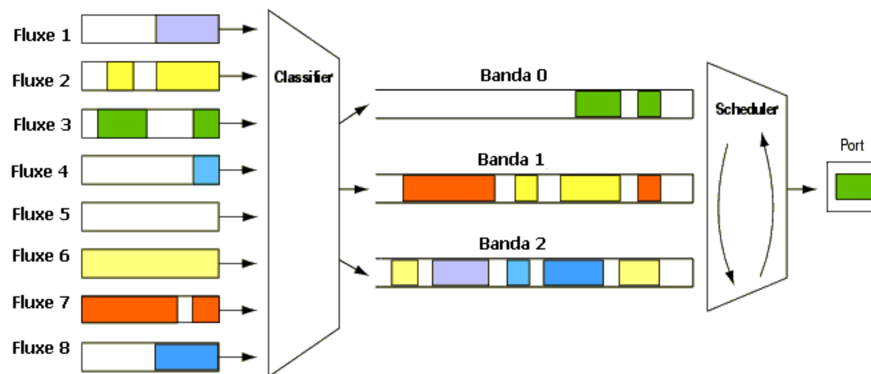

Figura 2.9: Funcionament Qdisc PRIO.

Aquesta Qdisc és molt útil en cas de desitjar donar prioritat a un determinat tràfic, sense utilitzar solament les marques TOS, sinó utilitzant el potencial ofert pels filtres d'**Iproute**. També pot contenir qualsevol Qdisc, mentre que la Qdisc **pfifo_fast** està limitada a Qdisc FIFO senzilles.

Donat que realment no realitza ajustaments, s'aplica el mateix avís que la Qdisc **SFQ**. S'utilitza únicament si l'enllaç físic està realment saturat o s'introdueix a l'interior d'una Qdisc amb classes. Aquest últim s'aplica a la majoria de dispositius de xarxa. Formalment, la Qdisc **PRIO** és un reorganitzador conservatiu. Les bandes són classes, i totes es criden de major a menor per defecte.

La Qdisc **PRIO** disposa de pocs paràmetres per realitzar una configuració eficient:

- **bands**: número de bandes a crear. Cada banda representa una classe. Si s'utilitza aquest paràmetre, també s'haurà d'utilitzar el paràmetre *priomap*.
- **priomap**: si no proporciona filtres **tc** per classificar el tràfic, la Qdisc **PRIO** examina la prioritat per decidir com encuar el tràfic. El funcionament segueix el mateix principi que la Qdisc **pfifo_fast**.

La figura 2.10 mostra un exemple d'estructura de classes per implantar la Qdisc **PRIO**. Per obtenir la configuració anterior són necessàries les sentències següents:

```
# tc qdisc add dev eth1 root handle 1: prio
# tc qdisc add dev eth1 parent 1:1 handle 10: pfifo
# tc qdisc add dev eth1 parent 1:2 handle 10: sfq
# tc qdisc add dev eth1 parent 1:3 handle 30: tbf rate 0.5mbit
burst 5kb latency 70ms peakrate 1mbit minburst 1540
```

La primera sentència crea la Qdisc **PRIO** amb tres classes, les quals són 1:1, 1:2, 1:3. La resta d'instruccions estableixen una determinada Qdisc sense classes a cadascuna de les classes creades anteriorment. La classe 1:1 està enllaçada amb la fulla :10 i disposa de la Qdisc **pfifo_fast**. La classe 1:2 està enllaçada amb la fulla :20 i disposa de la Qdisc **SFQ**. Per finalitzar, la classe 1:3 està enllaçada amb la fulla :30 i disposa de la Qdisc **TBF** sota un *rate* de 0.5Mbps amb 5kb de buffer i un *peak rate* de 1.0Mbps per ràfegues curtes de paquets. La grandària de la cua de paquets es calcula a partir d'un màxim 70 ms de *latency* que un paquet estarà a la cua. El *minburst* se selecciona com la MTU de la interfície. El codi següent mostra la configuració:

```
# tc qdisc show dev eth1
qdisc prio 1: bands 3 priomap  1 2 2 2 1 2 0 0 1 1 1 1 1 1 1 1
qdisc pfifo 10: parent 1:1 limit 1000p
qdisc sfq 20: parent 1:2 limit 128p quantum 1514b
qdisc tbf 30: parent 1:3 rate 500000bit burst 5Kb peakrate 1000Kbit
 minburst 1539b lat 70.0ms
```

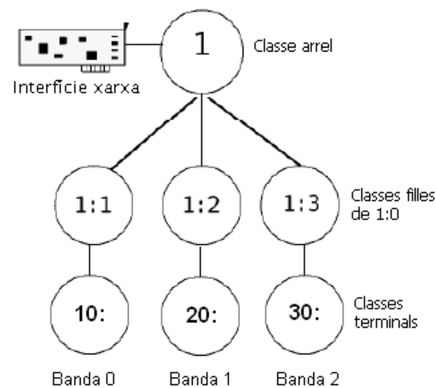

Figura 2.10: Exemple Qdisc PRIO.

La Qdisc CBQ

La Qdisc **cbq** (*Class Based Queueing*) és la disciplina més complexa i difícil de configurar correctament, però actualment està obsoleta. En el seu lloc s'utilitza la disciplina **HTB** que veurem a la següent secció. Aquesta disciplina no és molt precisa i no s'adapta correctament al funcionament del SO Linux.

Al ser la Qdisc **CBQ** amb classes, realitza tasques d'ajustament (*shapping*) i és en aquest aspecte que la implementació no funciona perfectament, però en la majoria de circumstàncies funciona correctament.

La Qdisc **CBQ**, per realitzar el *shapping*, treballa assegurant-se que l'enllaç està ocupat solament el temps necessari per reduir l'ample de banda real fins assolir la taxa configurada. Per aconseguir-ho, calcula el temps que hauria de passar entre paquets.

Mentre calcula, es mesura el temps ociós efectiu utilitzant una mitjana de moviment exponencial proporcional, anomenada EWMA (*Exponential Weighted Moving Average*), la qual considera els paquets recents exponencialment més importants que els passats. Els temps ociós calculat es resta al mesurat mitjançant EWMA i el número resultant s'anomena *avgidle*. Un enllaç carregat perfectament conté un *avgidle* de zero, perquè els paquets arriben exactament un cop cada interval calculat. Un enllaç sobrecarregat té un *avgidle* negatiu, la Qdisc necessita aturar-lo durant una estona i aleshores es produeix un *sobrelímit*. Per evitar-ho, el *avgidle* es trunca en *maxidle*. Si existeix una situació de *sobrelímit*, la Qdisc **CBQ** hauria d'accelerar-se automàticament durant exactament el temps calculat que passa entre paquets, aleshores passa un paquet i torna a accelerar-se.

Aquests són els paràmetres que podem especificar per configurar l'ajust:

- **avpkt**: mida mitjana d'un paquet mesurat en bytes. Es necessita per calcular *maxidle* que deriva de *maxburst* que s'especifica en paquets.

- **bandwidth**: l'ample de banda físic del dispositiu, necessari per càlculs de temps ociósos.

- **cell**: el temps que tarda un paquet en ser transmès sobre un dispositiu és escalonat, segons la mida del paquet.

- **maxburst**: aquest número de paquets s'utilitza per calcular *maxidel* de forma que quan *avgidle* estigui a *maxidel*, es pugui enviar una ràfega d'aquesta quantitat de paquets abans que *avgidle* es redueixi a zero. Un valor alt d'aquest paràmetre serà tolerant a ràfegues. No es pot establir *maxidel* directament, sinó solament mitjançant aquest paràmetre.

- **minburst**: com s'ha explicat anteriorment, la Qdisc **CBQ** necessita accelerar en cas de *sobrelímit*. La solució ideal consisteix en fer-ho exactamnet durant el temps ociós calculat, i passar un paquet. Encara que, als nuclis usualment els resulta difícil organitzar esdeveniments menors de 10ms, de forma que és millor accelerar durant un període més gran, i aleshores fer passar *minburst* paquets d'una tanda, per després parar *minburst* cops més. El temps d'espera s'anomena *offtime*. Els valors alts de *minburst* arriben a ajustaments més precisos a llarg plaç, però amb ràfegues més grans a escala de milisegons.

- **minidle**: si *avgidle* està per sota de zero, estarem en situació de sobrelímit i necessitarem esperar fins que *avgidle* sigui suficientment gran com per enviar un paquet. Per prevenir una ràfega sobtada, després d'haver detingut l'enllaç durant un període prolongat, *avgidlle* es reinicia a *minidle* si es redueix massa. El paràmetre *minidle* s'especifica en microsegons negatius.

- **mpu**: mida mínim del paquet, encara que un paquet sigui de mida zero s'omple amb 64Bytes sobre TCP/IP i, per tant, comporta cert temps en transmetre'ls. La Qdisc **CBQ** necessita conèixer aquest valor per calcular de forma adequada el temps ociós.

- **rate**: la taxa desitjada sortint d'aquesta Qdisc. Realment, aquest és el paràmetre de control de velocitat.

Internament, la Qdisc **CBQ** disposa de gran quantitat d'ajustaments molt més precisos, però que es realitzen automàticament. Per exemple, les classes que no contenen dades encuades no se'ls pregunta. Es penalitzen les classes sobrelimitades reduint la seva prioritat efectiva.

A part de l'ajustament, utilitzant les aproximacions de temps ociós, la Qdisc **CBQ** també actua igual que la Qdisc **PRIO** en el sentit de què les seves classes poden tenir diferents prioritats i que els números petits de prioritat s'analitzen abans que els grans. Cada cop que es requereix un paquet per enviar-lo a la xaxa, s'inicia un procés anomenat **WRR** (*Weigth Round Robin*), començant per les classes de menor prioritat.

Aquestes s'agrupen i se'ls pregunta si tenen dades disponibles. En cas afirmatiu, es retornen. Després d'haver permés desencuar una sèrie de bytes a una classe, es prova amb la següent classe d'aquella mateixa prioritat. Els paràmetres següents controlen el procés **WRR**:

- **allot**: en el moment que es demani a la Qdisc **CBQ** un paquet per enviar-lo a la interfície, buscarà per torns en totes les seves Qdisc internes (a les classes), amb l'ordre del paràmetre de *prioritat*. En cada torn d'una classe, solament pot enviar una quantitat limitada de dades. Aquest paràmetre és la unitat bàsica d'aquesta quantitat.
- **prio**: la Qdisc **CBQ** també pot actuar com un dispositiu **PRIO**. Primer es prova amb les classes internes de menor prioritat i mentre tingui tràfic, no es miren les altres classes.
- **weight**: aquest paràmetre ajuda durant el procés de **WRR**. Cada classe conté una oportunitat per torns per enviar. Si una classe conté un ample de banda significativament inferior a les altres, té sentit permetre-li enviar més dades a la seva ronda que a les altres. Una Qdisc **CBQ** suma tot el pes sota una classe, els normalitza de forma que pugui utilitzar nombres aleatoris. Popularment, s'utilitza un valor de taxa 10 i generalment funciona correctament. El pes renormalitzat es multiplica pel paràmetre *allot* per determinar quantes dades s'envien en cada ronda.

A més de limitar determinats tipus de tràfic, la Qdisc **CBQ** està capacitada per especificar quines classes poden agafar l'ample de banda cedit d'altres classes, o cedir-lo. Els paràmetres que determinen la compartició i préstec de l'enllaç són:

- **isolated/sharing**: una classe configurada com *isolated* no cedirà ample de banda a les seves germanes. S'utilitza en cas de tenir diversos agents competidors o mútuament hostils sobre l'enllaç que es deixen espai entre ells. El paràmentre *sharing* correspon al criteri invers.
- **bounded/borrow**: un classe pot configurar-se com *bounded* o limitada, la qual cosa significa que no tractarà d'agafar ample cedit de les classes germanes. El paràmetre *borrow* correspon al criteri invers.

En una situació típica, ens podríem trobar dos agents sobre un mateix enllaç, que al mateix temps són *isolated* i *bounded*, la qual cosa significa que estan realment limitades a les seves taxes aïllades i que permetran cedir-les entre elles. Juntament a aquestes classes, poden haver altres amb drets per cedir ample de banda.

La figura 2.11 mostra un exemple d'estructura de classes per implantar la Qdisc **CBQ**. A grans trets, la configuració s'encarrega de limitar el tràfic d'un servei a una velocitat de 50Mbit i d'un altre a 50Mbit. Junts, no poden superar una velocitat de 100Mbit. Disposem d'un ample de banda 100Mbit i les classes poden prendre cedit ample de banda una de l'altra. Per obtenir la configuració anterior són necessàries les sentències següents:

```
# tc qdisc add dev eth1 root handle 1:0 cbq bandwidth 100Mbit avpkt 1000 cell 8
# tc class add dev eth1 parent 1:0 classid 1:1 cbq bandwidth 100Mbit
rate 100Mbit weight 0.6Mbit prio 8 allot 1514 cell 8 maxburst 20
avpkt 1000 bounded
# tc class add dev eth1 parent 1:1 classid 1:2 cbq bandwidth 100Mbit
rate 50Mbit weight 0.5Mbit prio 5 allot 1514 cell 8 maxburst 20
avpkt 1000
# tc class add dev eth1 parent 1:1 classid 1:3 cbq bandwidth 100Mbit
rate 50Mbit weight 0.5Mbit prio 5 allot 1514 cell 8 maxburst 20
avpkt 1000
# tc qdisc add dev eth1 parent 1:2 handle 20: pfifo
# tc qdisc add dev eth1 parent 1:3 handle 30: pfifo
```

La primera sentència crea la Qdisc **CBQ** amb l'arrel a la classe *1:0*. La segona instrucció s'encarrega de limitar la classe *1:0*, de forma que el seu ample de banda no superi 100Mbit. La tercera i quarta sentència creen les classes 1:2 i 1:3 respectivament, cadascuna de les quals disposa d'una velocitat mínima de 50Mbit, que pot augmentar fins 100Mbits una altra classe no requereix del seu ample de banda. Les dues últimes instruccions creen les classes terminals 20: i 30: amb Qdisc's **pfifo_fast** a partir de les classes 1:2 i 1:3 respectivament. El codi següent mostra la configuració:

```
# tc qdisc show dev eth1
qdisc cbq 1: rate 100000Kbit (bounded,isolated) prio no-transmit
qdisc pfifo 20: parent 1:2 limit 1000p
qdisc pfifo 30: parent 1:3 limit 1000p
```

La implantació d'una Qdisc **CBQ** requereix la configuració de molts paràmetres, la majoria del quals són extranys i poc entenedors per l'usuari. La Qdisc **CBQ** està actualment obsoleta, donat que disposem de la Qdisc **HTB**, la qual és una evolució de

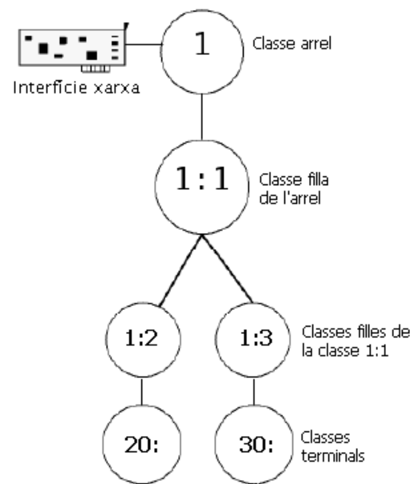

Figura 2.11: Exemple Qdisc CBQ.

la primera. Com veurem a continuació, la Qdisc **HTB** es configura fàcilment i s'obtenen resultats més eficients.

La Qdisc HTB

Donada la complexitat que presentava la Qdisc **CBQ**, aquesta no era òptima en vàries situacions típiques. La jerarquia **HTB** (*Hierarchical Token Bucket*) s'ajusta a configuracions on existeix una quantitat fixa d'ample de banda, a dividir entre diferents propòsits, donant-li a cadascun d'aquests propòsits un ample de banda garantit, amb la possibilitat d'especificar la quantitat d'ample de banda establert.

A grans trets, la Qdisc **HTB** és més fàcil de comprendre, més intuïtiva i ràpida de reemplaçar que la Qdisc **CBQ**. Totes dues Qdisc serveixen per controlar l'ús de l'ample de banda d'un enllaç donat. Les dues estan capacitades per utilitzar l'enllaç físic, com simulacions de múltiples enllaços més lents i enviar diferents tipus de tràfic per cadascun d'aquests enllaços virtuals. En tots dos casos s'ha d'indicar explícitament com dividir l'enllaç físic en diferents enllaços virtuals, i com decidir en quin enllaç virtual s'envia un determinat paquet.

Per compartir l'enllaç, la Qdisc **HTB** assegura que la capacitat destinada a un enllaç virtual és almenys el mínim establert durant la seva configuració. Quan una classe requereix menys de l'assignat, la resta d'ample de banda se cedeix a altres classes que en requereixin.

La Qdisc **HTB** disposa de pocs paràmetres per la seva configuració:

- **default**: tot el tràfic que no s'ha classificat s'envia a la classe indicada per aquest paràmetre.
- **rate**: la taxa de velocitat mínima establerta per una classe.
- **ceil**: velocitat màxima que pot assolir en cas que altres classes no utilitzin el seu *rate*.
- **r2q**: és un paràmetre global, amb un valor per defecte de 10, que correspon al valor típic d'una MTU de 1500 per *rates* a partir de 15Kbps. Per valors inferiors, s'especifica *r2q 1* crear la Qdisc.
- **burst** i **cburst**: paràmetre relacionat amb l'anterior *rate* i *ceil* respectivament. Aquest paràmetre controla la quantitat d'informació que podem enviar a la màxima velocitat sense servir cap altra classe. S'utilitza per millorar el temps de resposta en xarxes molt congestionades. Aquest paràmetre depèn del rellotge del sistema, és a dir, depèn de l'arquitectura del computador. Actualment, s'utilitzen valors de 15kbits.

La figura 2.12 mostra un exemple d'estructura de classes per implantar la Qdisc **HTB**, el qual aconsegueix la mateixa estructura de classes que la Qdisc **CBQ** amb pocs paràmetres de configuració, donat que la Qdisc **HTB** disposa dels mecanismes per realitzar la configuració automàtica dels paràmetres més específics. El codi següent mostra la seva simplicitat:

```
# tc qdisc add dev eth1 root handle 1: htb default 20
# tc class add dev eth1 parent 1: classid 1:1 htb rate 100Mbit
# tc class add dev eth1 parent 1:1 classid 1:2 htb rate 50Mbit ceil 100Mbit
# tc class add dev eth1 parent 1:1 classid 1:3 htb rate 50Mbit ceil 100Mbit
# tc qdisc add dev eth1 parent 1:2 handle 20: pfifo
# tc qdisc add dev eth1 parent 1:3 handle 30: pfifo
```

La primera sentència crea la Qdisc **HTB** amb l'arrel a la classe *1:0* i enviarà per defecte a la classe terminal :20 aquells paquets que no siguin classificats per cap filtre. La segona instrucció s'encarrega de limitar la classe *1:0*, de forma que el seu ample de banda no superi 100Mbit. La tercera i quarta sentència creen les classes 1:2 i 1:3 respectivament cadascuna de les quals disposa d'una velocitat mínima de 50Mbit, que pot augmentar fins 100Mbits si l'altre classe no requereix del seu ample de banda. Les dues últimes instruccions creen les classes terminals 20: i 30: amb Qdisc's **pfifo_fast** a partir de les classes 1:2 i 1:3 respectivament. El codi següent mostra la configuració:

```
# tc qdisc show dev eth1
```

```
qdisc htb 1: r2q 10 default 20 direct_packets_stat 23
qdisc pfifo 20: parent 1:2 limit 1000p
qdisc pfifo 30: parent 1:3 limit 1000p
```

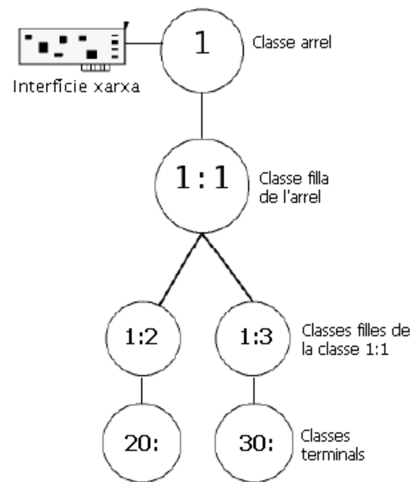

Figura 2.12: Exemple Qdisc HTB.

2.3.3 Classificar paquets amb filtres

Per determinar quina classe s'haurà d'encarregar de processar un paquet, es crida a una *cadena classificadora* cada cop que sigui necessària una decisió. Aquesta cadena està constituïda per tots els filtres associats a la Qdisc amb classes que ha de decidir en aquell moment. Els filtres per classificar paquets són necessaris per reorganitzar els paquets en qualsevol de les subcues. A continuació es mostren els dos classificadors més utilitzats actualment:

- **fw**: Basa la decisió en la forma que el tallafocs **iptables** ha marcat el paquet. Aquest tipus de filtres és una alternativa simple a la sintaxi establerta pel filtre u32.
- **u32**: Basa la decisió en camps de l'interior del paquet TCP/IP.
- Altres classificadors menys utilitzats, com són:
  - **route**: Basa la decisió en la ruta on serà enviat el paquet.

  - **rsvp**: Encamina els paquets basant-se en RSVP. Internet no està basat en el protocol RSVP i solament és útil en xarxes que segueixin estrictament aquest protocol.

Com acabem de veure, existeixen molts tipus de filtres per classificar els paquets, però tots accepten alguns arguments comuns:

- **protocol**: El protocol que acceptarà aquest classificador. Donat que internet funciona sobre TCP/IP, el protocol acceptat serà IP.
- **parent**: El controlador on estarà associat aquest classificador. Aquest controlador ha de ser una classe existent.
- **prio**: La prioritat d'aquest classificador. Els números més baixos es comproven abans.
- **handle**: Aquest controlador obté el significat segons el filtre utilitzat.

A continuació es descriuen en detall els classificadors més utilitzats, **fw** i **u32**.

El classificador fw

El classificador **fw** està constituït per una combinació de SW **Netfilter** i **Iproute**. El SW Netfilter conté l'aplicatiu **iptables** que ens permetrà definir polítiques de filtrat des de l'espai d'usuari. Com veurem, en realitat realitzen modificacions directes sobre les capçaleres dels paquets mitjançant una capacitat especial per marcar paquets a través d'un número, a partir de l'estructura *–set-mark*.

Iptables agrupa les regles en cadenes, cada cadena és una llista ordenada de regles. Les cadenes s'agrupen en taules i cadascuna d'aquestes taules està associada amb un tipus diferent de processament de paquets. Per defecte, internament incorpora tres taules i cadascuna incorpora certes cadenes predefinides. Podem crear i eliminar cadenes definides per usuaris a l'interior de qualsevol taula. Inicialment, totes les cadenes estan buides i tenen una política de destí que permet que tots els paquets passin sense ser bloquejats o alterats. Aquestes tres taules són:

- **filter table**: és la responsable del filtrat. Tots els paquets passen a través de la taula de filtres.
- **nat table**: Aquesta taula és la responsable de configurar les regles de reescriptura de direccions o de ports dels paquets.

- **mangle table**: és la responsable d'ajustar les opcions dels paquets, com pot ser adientment la QoS. Tots els paquets passen per aquesta taula.

Com s'ha comentat, cada cadena conté una llista de regles. Quan un paquet s'envia a una cadena, se'l compara contra cada regla de la cadena. La regla especifica quines propietats haurà de tenir el paquet perquè la regla concordi. Si la regla no concorda, el processament continua amb la regla següent. Si la regla concorda amb el paquet, les instruccions de destí de les regles se segueixen. El codi següent mostra com marcar els paquets sobre **iptables**:

```
iptables -A PREROUTING -i INTERFICIE -t mangle OPCIONS_PAQUETS
-j MARK --set-mark NUMERO_MARCA
```

Un cop tenim els paquets marcats segons els nostres criteris, actua Iproute. Aquest software, a través de l'eina **tc**, conté un mecanisme per filtrar els paquets marcats prèviament a través d'Iptables i enviar-los a la classe corresponent. El codi següent mostra com l'eina **tc** filtra els paquets marcats i els envia a la classe desitjada:

```
tc filter add dev ITERFICIE protocol PROTOCOL parent PARE prio PRIORITAT
   handle NUMERO_MARCA fw flowid CLASSE
```

Actualment el classificador **u32** disposa d'una sintaxi molt estricta i eficaç, però poc comprensible. El classificador **u32** és el més modern i no disposa de tots els mecanismes de filtrat implementats. Aquests mecanismes sí estan disponibles a través del classificador **fw**. Per tant, en funció del tràfic a filtrar s'haurà d'utilitzar un o altre classificador, o fins i tot els dos alhora.

El classificador u32

El filtre **u32** és el més avançat dell que es disposa en la implementació actual. Basa el seu funcionament en taules Hash, les quals converteixen aquest filtre en un sistema de filtrat molt robust.

En la seva forma més simple, el filtre **u32** és una llista de registres, cadascun d'ells consisteix en dos camps: un selector i una acció. A grans trets, els selectors es comparen amb el paquet IP que s'està processant fins trobar la primera coincidència, i aleshores s'executa l'acció associada. El tipus més senzill d'acció seria dirigir el paquet a una classe definida.

La línia d'ordres de programa ***tc*** filter, que s'utilitza per configurar el filtre, consisteix en tres parts: especificació del filtre, selector i acció. L'especificació del filtre pot definir-se així:

```
tc filter add dev INTERFICIE protocol PROTOCOL priority PRIORITAT
   parent CLASSE u32
```

El camp *protocol* descriu el protocol al qual s'aplicarà el filtre. Donat que internet es basa sobre el protocol TCP/IP, únicament es comentarà sobre aquest. El camp *priority* estableix la prioritat del filtre escollit. Aquest camp és important, perquè podem tenir varis filtres amb diferents prioritats. Es passarà per cada llista en l'ordre en què s'agreguin les regles, i aleshores es processaran les llistes de menor prioritat. El camp *parent* defineix la classe de l'arbre on s'associarà el filtre.

El *selector u32* conté definicions dels patrons que seran comparats amb el paquet processat en cada moment. Per ser més precisos, defineix quins són els bits que s'han de comparar a la capçalera del paquet. Aquest mètode sembla molt simple, però en realitat és molt potent. Un cop conegut el preàmbul de l'ordre d'ajustament, les opcions de filtrat més habituals són:

- **Sobre la direcció**:
  - *origen*:

    `match ip src XARXA/MASCARA`
  - *destí*:

    `match ip dst XARXA/MASCARA`
- **Sobre el port**:
  - *origen*:

    `match ip sport PORT 0xffff`
  - *destí*:

    `match ip dport PORT 0xffff`
- **Sobre el protocol IP**:

  `match ip PROTOCOL ID_PROTOCOL 0xff`
  - *TCP*
  - *UDP*
  - *ICMP*
  - *GRE*
- **Sobre fwmark**: podem marcar els paquets amb iptables i que la marca sobrevisqui a l'enrutament a través d'interfícies. Usualment s'utilitza per ajustaments de tràfic entre diferents interfícies d'una mateixa màquina.

2.4 Experimentació

Un cop vistos els diferents conceptes que formen l'entorn de la QoS a nivell de xarxa, per assolir els objectius de QoS, s'ha desenvolupat una nova Qdisc que permet aconseguir aquests propòsits. El script és **qosNETWORK.sh**, el qual forma part del **Script QoS** (Apèndix B).

En aquesta secció s'exposen els resultats obtinguts amb l'execució del script **proves-NETWORK.sh** (Apèndix C.1). L'objectiu es comprovar el funcionament de la nova Qdisc vers la Qdisc estàndard del SO Linux (**pfifo_fast**). Aquesta nova Qdisc s'adaptarà correctament a les exigències marcades de QoS a nivell d'usuari, amb la qual cosa obtindrem resultats òptims. Per altra banda, amb la Qdisc **pfifo_fast** obtindrem resultats gens satisfactoris.

Donat que és necessari regular l'ample de banda, per dividir-lo entre diferents propòsits i donar-li a cadascun d'aquests propòsits un ample de banda garantit, aleshores és necessària una Qdisc amb classes per tal de crear la jerarquia de classes desitjada.

La Qdisc amb classes principal elegida per construir la jerarquia de classes és **HTB**, perquè és la més avançada i requereix pocs paràmetres de configuració. Per altra banda, la Qdisc **CBQ** requerix configurar explícitament molts paràmetres i ha esdevingut gairebé obsoleta des de l'aparició de **HTB**. La Qdisc **PRIO** no disposa dels mecanismes suficients per crear una jerarquia de classes, donat que realment és una optimització de la Qdisc **pfifo_fast**. Un cop construïda la jerarquia de classes, totes les classes terminals tenen associada una Qdisc sense classes **SFQ** per tal que el tràfic d'una mateixa classe sigui repartit equitativament.

La figura 2.13 mostra l'estructura interna de la Qdisc construïda pel **Script QoS**. La Qdisc **HTB** conté l'arrel a la classe 1:0. L'arrel disposa de la classe filla 1:1, per tal d'establir l'ample de banda màxim de l'arrel. La classe 1:1 té dos classes filles; la classe filla 1:10 s'utilitza pel tràfic de xarxa local, mentre que la 1:20 s'utilitza pel tràfic d'Internet. Per poder diferenciar el tràfic dels diferents usuaris, les classes 1:10 i 1:20 tenen dues noves classes cadascuna. La classe 1:10 té les classes 1:110 i 1:120, que són utilitzades utilitzades pels usuaris privilegiats amb QoS i la resta d'usuaris, respectivament. La classe 1:20 té les classes 1:210 i 1:220, que són utilitzades pels usuaris privilegiats amb QoS i la resta d'usuaris, respectivament. Les classes terminals de l'arbre, que són 1:110, 1:120, 1:210 i 1:220, tenen totes associades la Qdisc **SQF**, llavors les noves classes terminals de l'arbre són 110:, 120:, 210: i 220: respectivament. Per altra banda, els filtres s'encarregaran d'enviar els paquets en funció de l'usuari i la destinació.

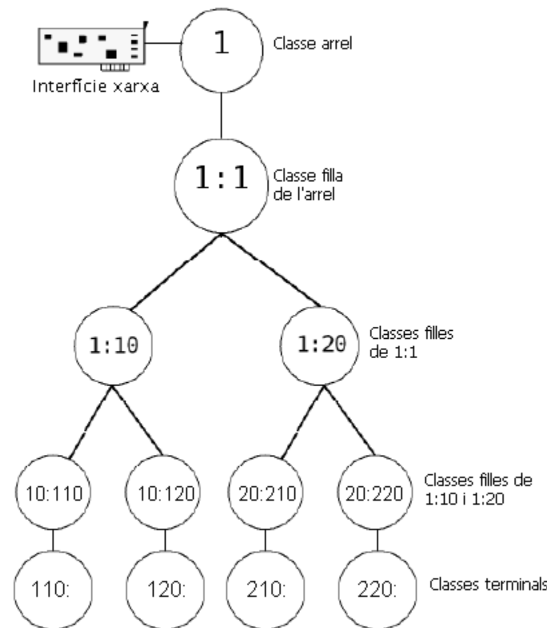

Figura 2.13: Qdisc del Script de QoS.

2.4.1 Resultats

A continuació es mostren els gràfics obtinguts de les proves realitzades a partir del script **provesNETWORK.sh** (Apèndix B.1). A grans trets, aquest script de proves s'encarrega de generar connexions de tràfic massiu durant un màxim de 50 segons sota la identitat d'un usuari determinat. Si aquest usuari està privilegiat amb QoS rebrà un determinat ample de banda i en cas contrari en rebrà un altre.

Les proves solament s'han realitzat per la branca del tràfic de xarxa local. Donat que els resultats en teoria haurien de ser simètrics per la branca d'Internet. Però en realitat els resultats no serien tant eficients, perquè depèn de la congestió d'Internet al moment de realitzar les proves. Les proves s'han realitzat sota la configuració següent:

- Usuari privilegiat:
  - Velocitat mínima de 70Mbits/s, és a dir, reserva del 70% d'ample de banda. Si algun altre usuari privilegiat realitza una connexió, aleshores estaran obligats a repartir-se aquest 70% equitativament.
  - Velocitat escalable de 100Mbits/s, per tant, podrà augmentar del 70% al 100%. Solament podrà augmentar la velocitat si cap usuari normal requereix d'ample de banda.

- Usuari normal:
  - Velocitat mínima de 30Mbits/s, és a dir, reserva del 30% d'ample de banda. Si algun altre usuari normal realitza una connexió, aleshores estaran obligats a repartir-se aquest 30% equitativament.
  - Velocitat escalable de 100Mbits/s, per tant, podrà augmentar del 30% al 100%. Solament podrà augmentar la velocitat si cap usuari privilegiat requereix d'ample de banda.

Per demostrar la ineficàcia de la Qdisc **pfifo_fast** (**Script QoS** desactivat) respecte de la gestió de QoS, les figures 2.14 i 2.15 mostren dues situacions típiques. La primera situació mostra com dos usuaris realitzen una connexió simultàniament, però aquestes dues no es reparteixen l'ample de banda equitativament. La segona situació mostra com dos usuaris realitzen una connexió no simultàniament, el segon usuari que realitza la connexió 15 segons després necessita molt de temps per obtenir una mica més d'ample de banda, donat que la primera connexió acapararà l'ample de banda i el reduirà progressivament.

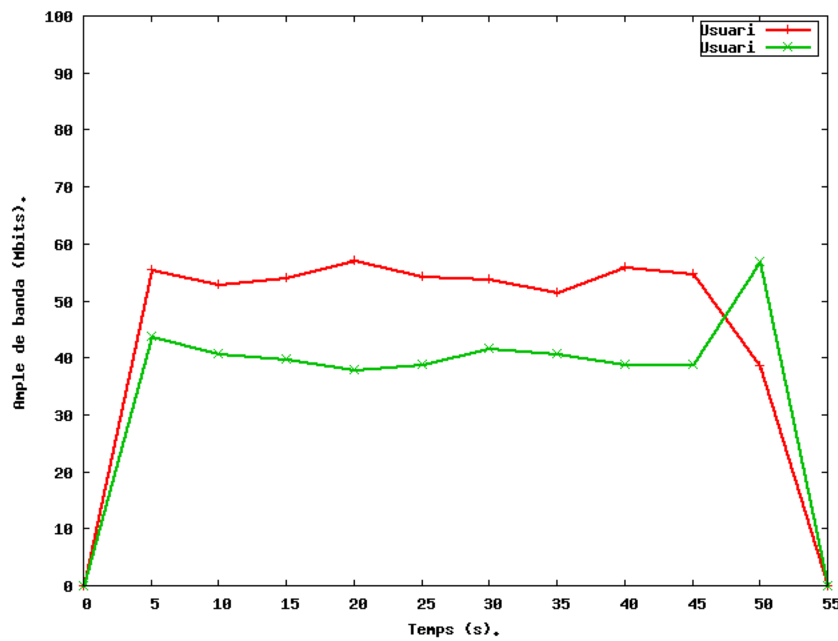

Figura 2.14: Prova xarxa: pfifo_fast 1.

Un cop s'ha demostrat que la Qdisc **pfifo_fast** no és molt eficient, fins i tot en situacions habituals, aleshores s'activarà el **Script QoS** per implantar la Qdisc detallada anteriorment i obtenir millors resultats que els presentats anteriorment.

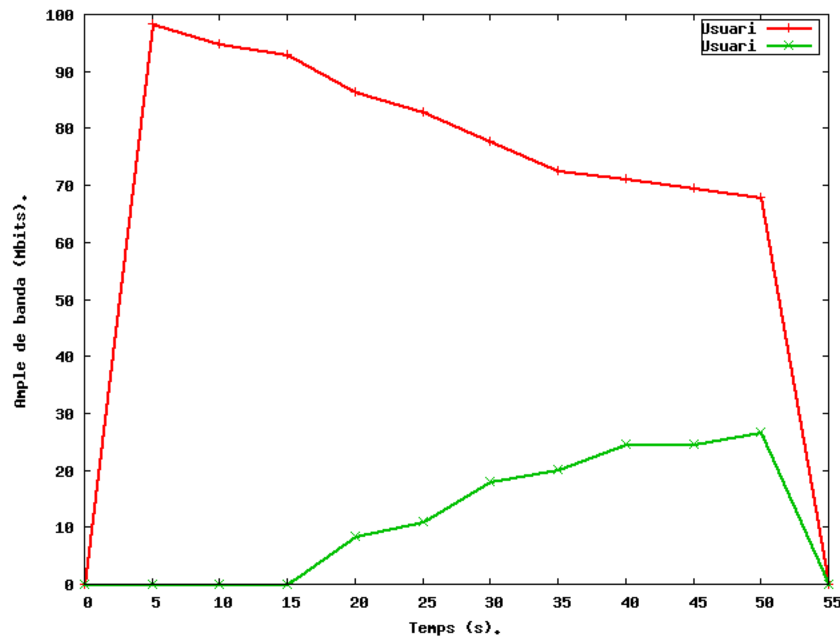

Figura 2.15: Prova xarxa: pfifo_fast 2.

La figura 2.16 mostra una connexió d'un usuari privilegiat ocupant la totalitat de l'ample de banda, supera el 70% mínim fins arribar al 100%. Donat que no existeixen altres connexions actives.

La figura 2.17 mostra dues connexions realitzades per dos usuaris privilegiats, per tant, aquestes dues connexions es repartiran l'ample de banda equitativament. Donat que no existeix cap connexió d'un usuari normal, supera el 70% mínim fins arribar al 100%. Però al ser dos connexions privilegiades disposaran del 50% d'ample de banda cadascuna. La Qdisc **SFQ** utilitzada per part de les classes terminals és la responsable del repartiment just de l'ample de banda.

La figura 2.18 mostra dues connexions realitzes per dos usuaris privilegiats, però el segon usuari inicia la connexió 15 segons després. En el primer període situat entre 0 i 15 segons solament existeix una connexió, aleshores el primer usuari obtindrà el 100% d'ample de banda. Durant el segon període situat entre 15 i 50 segons existeixen dues connexions privilegiades, per tant, cada connexió obtindrà el 50% d'ample de banda. Donat que les classes terminals de l'arbre tenen associades la Qdisc **SFQ**, el repartiment just de l'ample de banda serà instantani.

Per altra banda, si les classes terminals tenen associades la Qdisc **pfifo_fast** en lloc de la Qdisc **SFQ**, no es produirà el repartiment equitatiu de l'ample de banda entre usuaris

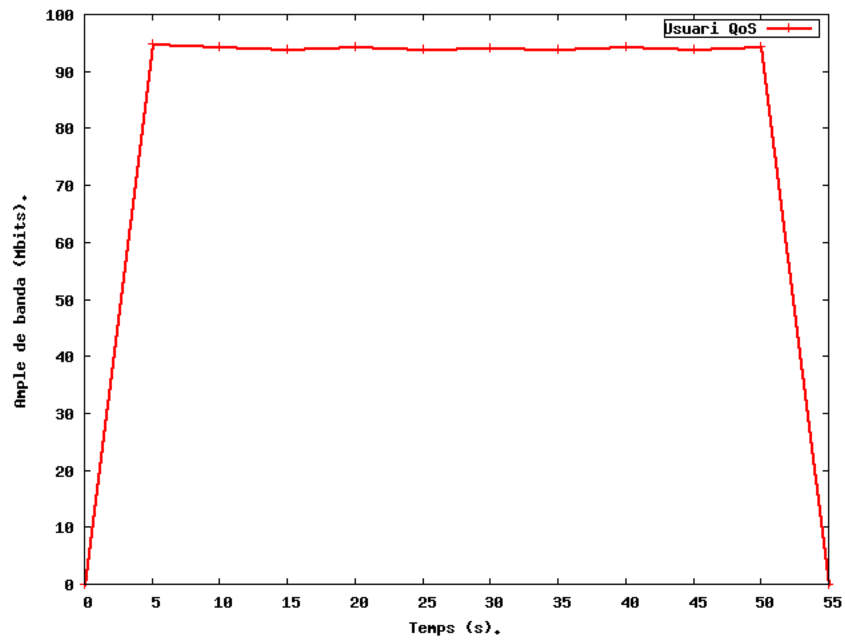

Figura 2.16: Prova xarxa: Script QoS 1.

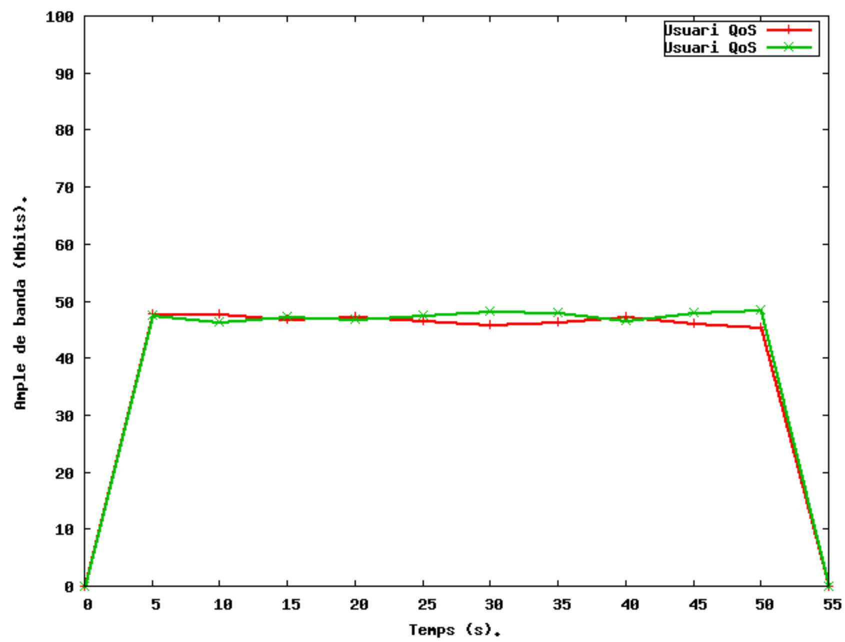

Figura 2.17: Prova xarxa: Script QoS 2.

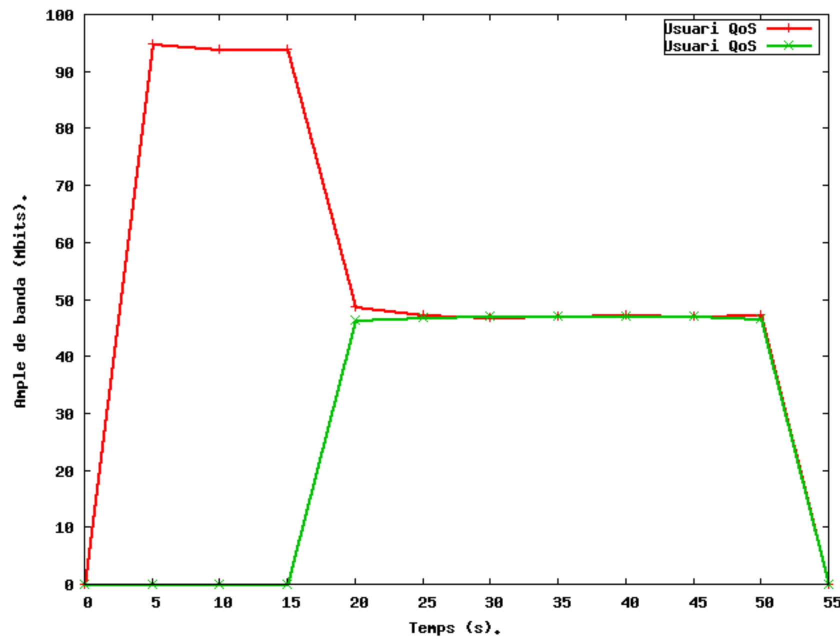

Figura 2.18: Prova xarxa: Script QoS 3.

dels mateix tipus. La figura 2.19 mostra aquest fet.

La figura 2.20 mostra dues connexions, la primera realitzada per un usuari privilegiat i la segona per un usuari normal. Donat que existeix un usuari de cada tipus, les dues connexions no podran escalar la velocitat. Per tant, l'usuari privilegiat obtindrà el 70% i l'usuari normal el 30% restant d'ample de banda.

La figura 2.21 mostra dues connexions, la primera d'un usuari privilegiat i la segona d'usuari normal, però l'usuari normal inicia la connexió 15 segons després. En el primer període situat entre 0 i 15 segons solament existeix una connexió, aleshores el primer usuari obtindrà el 100% d'ample de banda. Durant el segon període situat entre 15 i 50 segons existeixen les dues connexions, per tant, la primera connexió passarà del 100% escalable a la velocitat mínima establerta del 70% i la segona connexió disposa del 30% mínim reservat. Donat que les classes terminals de l'arbre tenen associades la Qdisc **SFQ**, el repartiment just de l'ample de banda serà instantani.

La figura 2.22 mostra dues connexions, la primera d'un usuari privilegiat i la segona d'usuari normal, però l'usuari privilegiat inicia la connexió 15 segons després. En el primer període situat entre 0 i 15 segons solament existeix una connexió, aleshores l'usuari normal escala fins el 100% d'ample de banda. Durant el segon període, situat entre 15 i 50 segons, existeixen les dues connexions, per tant, la connexió de l'usuari

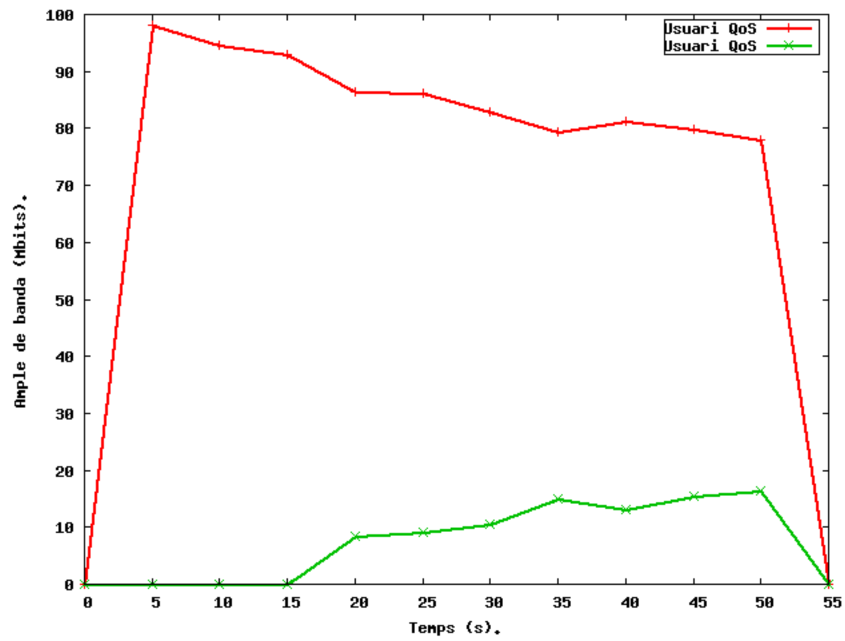

Figura 2.19: Prova xarxa: Script QoS 4.

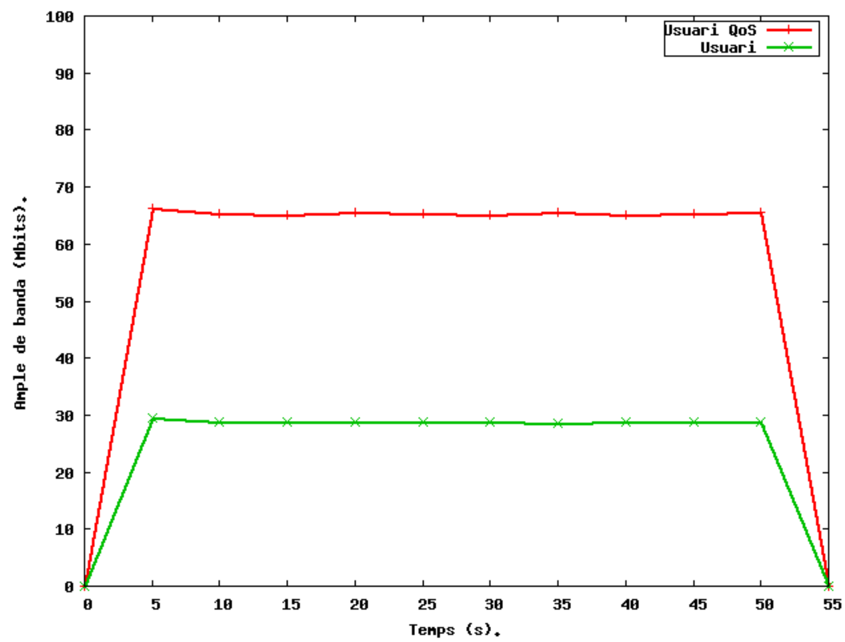

Figura 2.20: Prova xarxa: Script QoS 5.

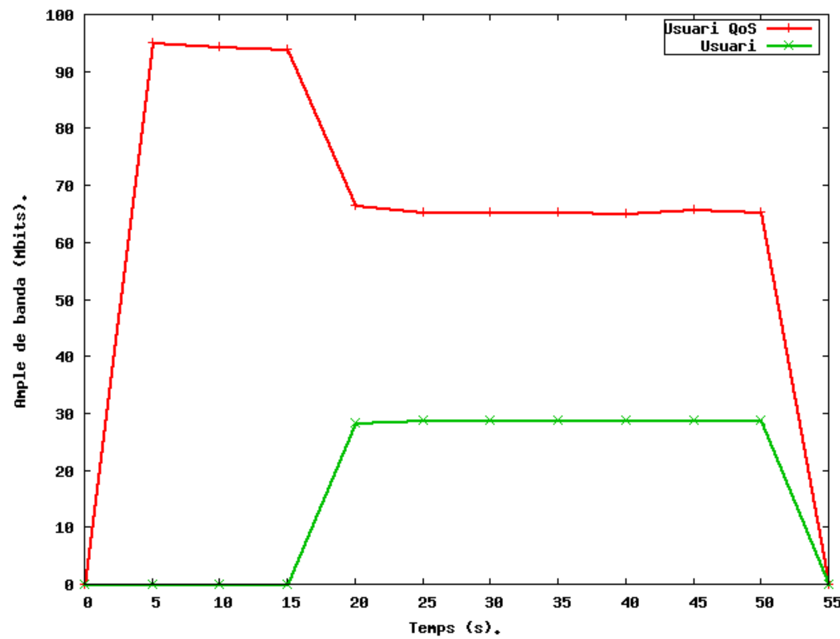

Figura 2.21: Prova xarxa: Script QoS 6.

normal passarà del 100% escalable a la velocitat mínima establerta del 30% i l'usuari privilegiat passarà al 70% mínim reservat. Donat que les classes terminals de l'arbre tenen associades la Qdisc **SFQ**, el repartiment just de l'ample de banda serà instantani.

La figura 2.23 mostra quatre connexions, dues de les quals d'usuaris privilegiats i les altres dues d'usuaris normals. Per una banda, les dues connexions privilegiades es repartiran el 70% d'ample de banda, aconseguint un 35% cadascuna. Per altra banda, les dues connexions normals es repartiran el 30% restant, aconseguint un 15% cadascuna.

La figura 2.24 mostra quatre connexions, dues de les quals d'usuaris privilegiats i les altres dues d'usuaris normals. Però un usuari privilegiat inicia la connexió 15 segons després i un usuari normal inicia la connexió 30 segons després. Durant el primer període situat entre 0 i 15 segons existeix una connexió privilegiada i una normal, llavors obtindran el 70% i 30% respectivament. Durant el segon període situat entre 15 i 30 segons existeixen dues connexions privilegiades i una normal. Per una banda, les dues connexions privilegiades hauran de repartir-se el 70% d'ample de banda, obtenint un 35% cadascuna. Per altra banda, la connexió normal disposarà del 30% d'ample de banda. Durant el tercer període situat entre 30 i 50 segons existeixen dues connexions privilegiades i dues normals, mentre les dues privilegiades no modificaran l'ample de

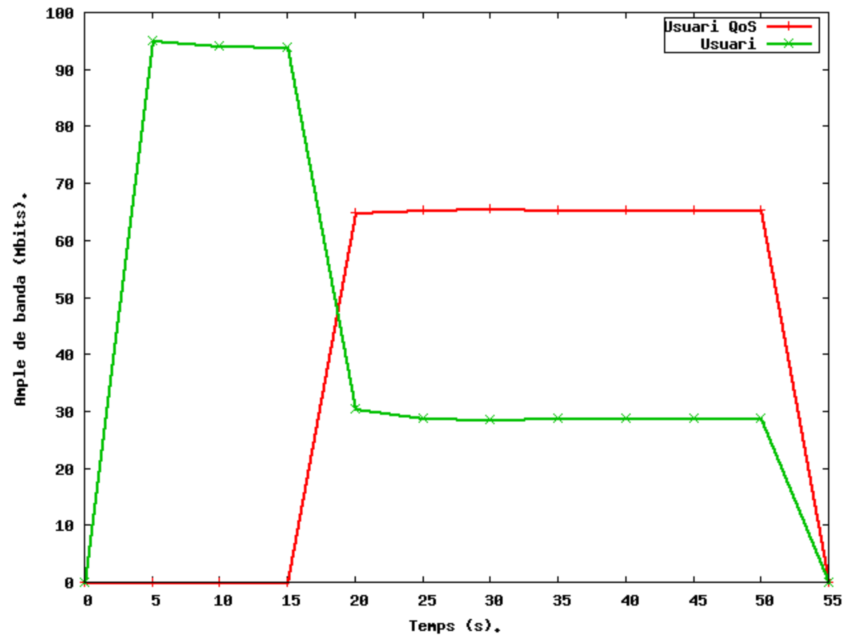

Figura 2.22: Prova xarxa: Script QoS 7.

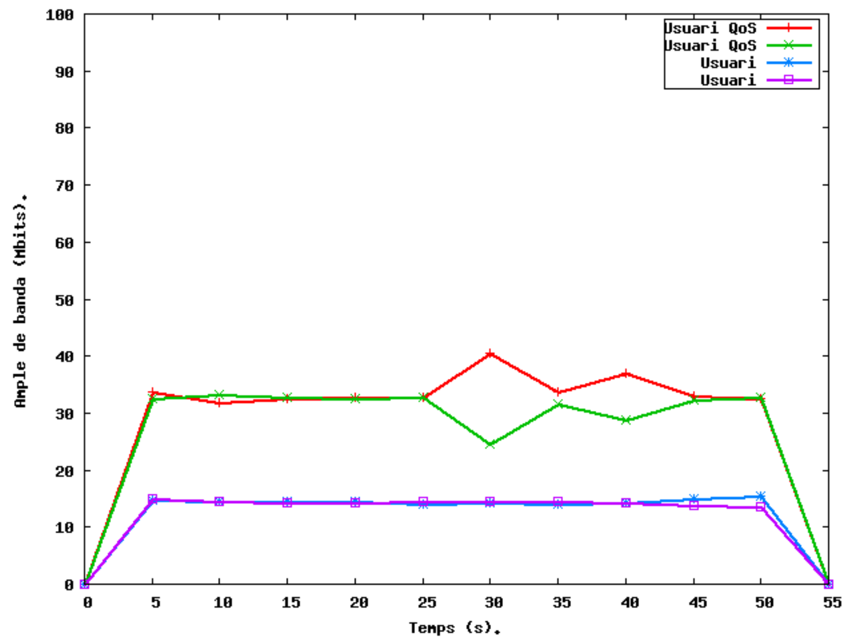

Figura 2.23: Prova xarxa: Script QoS 8.

banda assignat, la incorporació d'una nova connexió normal obligarà a la connexió activa reduir del 30% al 15%, per tant, l'altre 15% restant serà assignat a la nova connexió entrant.

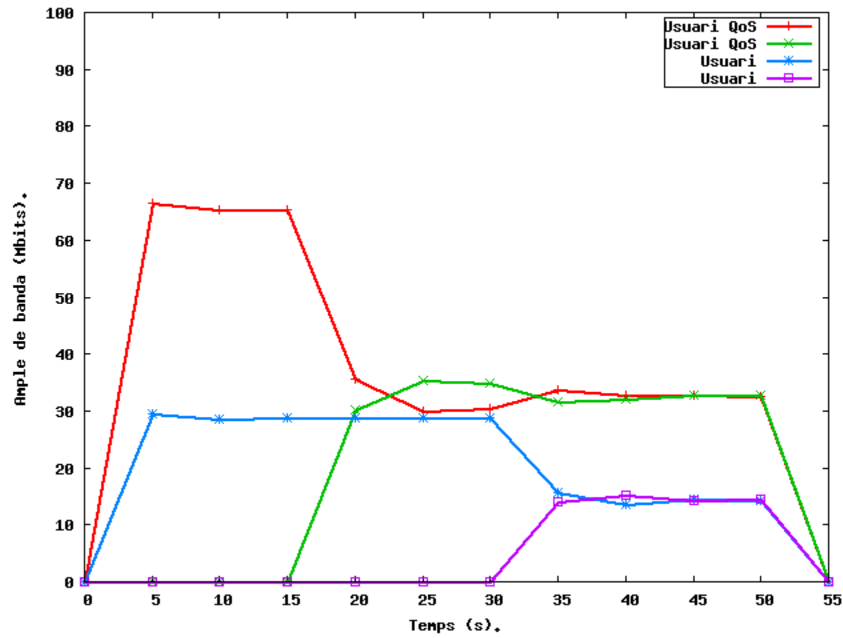

Figura 2.24: Prova xarxa: Script QoS 9.

Capítol 3

La QoS en l'entorn de CPU

3.1 Introducció

Un SO (*Sistema Operatiu*) de temps compartit és un SO multiprogramat que proporciona un temps de resposta relativament correcte a tots els processos d'usuari, proporcionant d'aquesta forma la sensació que múltiples processos s'estan executant concurrentment en un computador. S'aconsegueix un SO de temps compartit a través de la compartició dels recursos.

El recursos d'un computador no són infinits, per tant la CPU (*Central Processing Unit*) segueix aquesta limitació. En un determinat instant de temps cadascun dels processadors d'un computador estarà executant solament un procés, mentre la resta de processos preparats per l'execució estaran a l'espera de l'assignació d'una CPU.

La part del SO que s'ocupa de repartir el temps de les CPU's entre el conjunt de processos executables s'anomena **scheduler**, o simplement, **planificador de processos**. La seva funció principal és seleccionar quin procés s'ha d'executar en una CPU en un instant determinat, durant quant temps i quin és el millor candidat d'entre tots els processos preparats per obtenir una CPU quan el procés actual l'alliberi. El planificador estàndard del SO Linux disposa d'un mecanisme de gestió de prioritats que afavoreix l'execució dels processos *interactius* d'usuari davant els processos *batch*.

Com es mostra en aquest capítol, el planificador de processos estàndard està clarament limitat, i no ofereix els mecanismes suficients per reservar en tot moment porcions de CPU per aplicacions concretes d'usuaris privilegiats, i així poder implantar un sistema de QoS en l'entorn de CPU.

Per altra banda, **Con Kolivas** [14] ha desenvolupat un nou planificador de processos, el

qual està disponible pel nucli 2.6 i aplicable a través d'un *patch*. Aquest nou planificador està basat amb l'estàndard, però introdueix una sèrie de millores que ens permeten assolir els reptes esmentats anteriorment. Aquests canvis no formen part del nucli estàndard, per tant, és necessari compilar el nou nucli optimitzat.

3.2 Els processos

Un *programa* correspon a un algorisme codificat en un llenguatge de programació que, una vegada compilat, està preparat per l'execució en un computador.

Un *procés* és un programa en execució. Disposa d'un espai virtual d'adreces, un conjunt de recursos i un o més fils d'execució. L'espai virtual d'adreces d'un procés està dividit pels segments de codi, de dades i la pila.

Un *fil d'execució* està definit com l'entitat encarregada de realitzar l'execució en un procés. Un procés pot constar d'un o varis fils d'execució (*multifil*), els quals disposen del seu propi comptador de programa, la seva pròpia pila i el conjunt de registres de la CPU. Un fil és considerat com un procés qualsevol amb algunes particularitats, com són la compartició dels recursos i de l'espai de dades, entre la resta de fils d'un mateix procés. La figura 3.1 mostra el diagrama de processos en un sistema multifil.

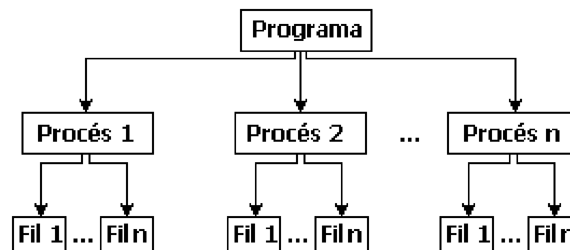

Figura 3.1: Diagrama de processos en un sistema multifil.

El nucli resideix en un nivell de sistema superior als dels processos d'usuari. Aquest nivell superior s'anomena *espai de nucli*. Sota l'espai de nucli s'executen els processos de sistema gestors del SO i les rutines de servei d'interrupcions. Els principals privilegis que dóna són:

- Un espai de memòria protegit.
- Permet accés total al hardware.
- No hi ha protecció de memòria.

Els processos d'usuari s'executen sota l'*espai d'usuari*. Aquest espai es caracteritza per:

- Protecció de memòria.
- No permet l'accés als recursos hardware.

Per tant, a través de l'espai d'usuari no podríem executar la majoria de processos d'usuari. Aleshores, es fa necessari que els processos d'usuari s'executin en l'espai de nucli en alguns instants, per tal de poder dur a terme determinades operacions. Aquesta situació s'assoleix a través de les *crides a sistema*.

Les crides a sistema són un conjunt de funcions que formen part del SO. Proporcionen una interfície de comunicació amb el nucli. A través de les crides a sistema es permet l'execució dels processos d'usuari sota l'espai del nucli amb restriccions. D'aquesta forma el nucli resta protegit.

3.2.1 El Bloc de Control de Procés

El **PCB** (*Bloc de Control de Procés*), anomenat descriptor de procés, és una estructura de dades que conté tota la informació que necessita el SO per tal de gestionar un procés. El PCB esdevé la visió del nucli d'un procés.

El bloc de control de procés és una estructura del tipus **task_struct**. Aquesta estructura està definida dins *<linux/sched.h>*.

Cada procés disposa d'un espai de memòria del nucli que li té reservat. Aquest espai té una mida fixa, és reservat durant la creació del procés i posteriorment alliberat quan el procés acaba d'executar-se. Conté la pila del nucli i el descriptor del procés.

La figura 3.2 mostra com està distribuït l'espai de memòria del nucli reservat per un procés. A l'adreça més alta està situat el principi de la pila, per tant, la pila creix de dalt cap a baix. A l'adreça més baixa s'hi troba l'estructura **thread_info**. El camp **task** d'aquesta estructura conté un punter al descriptor del procés.

Quan un procés s'està executant, al registre ESP del processador s'hi emmagatzema l'adreça del top de la pila del nucli. Això, juntament amb el fet que l'àrea continguda estigui alineada en una adreça múltiple de 2, fa possible obtenir l'adreça de l'estructura **thread_info** aplicant una màscara al registre ESP. Si després accedim al camp **task** d'aquesta estructura, s'obté un punter al descriptor de procés. D'aquesta forma s'aconsegueix una forma d'accés ràpid al PCB del procés en execució. La macro **current**, que retorna un punter al PCB del procés en execució, està implementada d'aquesta forma.

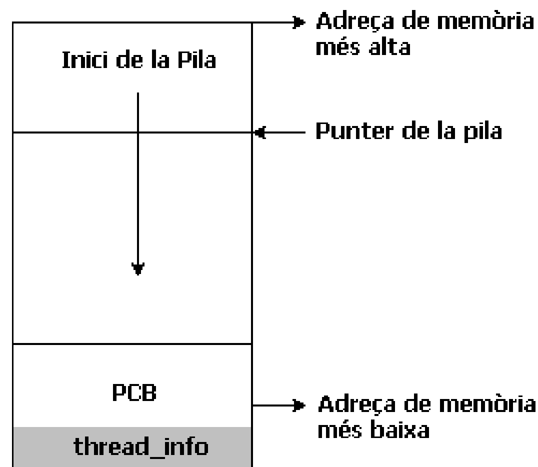

Figura 3.2: Espai de memòria del nucli reservat a un procés.

3.2.2 Estats

L'estat d'un procés descriu com es troba el procés en un determinat moment. S'emmagatzema al camp **state** del descriptor de procés. Els diferents estats possibles d'un procés són els següents:

- **TASK_RUNNING**: el procés pot estar executar-se, o bé pot estar situat en la cua de preparats en espera de ser escollit pel planificador de processos.
- **TASK_INTERRUPTIBLE**: el procés està bloquejat en espera de complir una condició donada. Quan s'esdevé aquesta condició, el procés es desperta i se situa en estat TASK_RUNNING. El procés pot ser despertat prematurament si rep un senyal.
- **TASK_UNINTERRUPTIBLE**: aquest estat es idèntic al TASK_INTERRUPTIBLE, excepte que el procés no es desperta prematurament si rep un senyal.
- **TASK_ZOMBIE**: el procés ha acabat d'executar-se i està en espera que el seu procés pare rebi informació referent a l'acabament de la seva execució. Aquesta informació es rebuda pel pare mitjançant la crida a sistema **wait**.
- **TASK_STOPPED**: el procés està parat degut a què ha rebut el senyal SIGSTOP, SIGTSTP, SIGTTIN o SIGTOOU.
- **TASK_DEAD**: el procés ha acabat la seva execució i ja ha enviat la informació referent a l'acabament de la seva execució al procés pare. Arribat en aquest punt, pot ser desallotjat de la CPU.

Tots aquests estats són exclusius, és a dir, un procés en un determinat instant solament pot trobar-se en un d'aquests estats. El diagrama de flux representat per la figura 3.3 mostra els diferents canvis d'estat que es poden dur a terme durant la vida d'un procés.

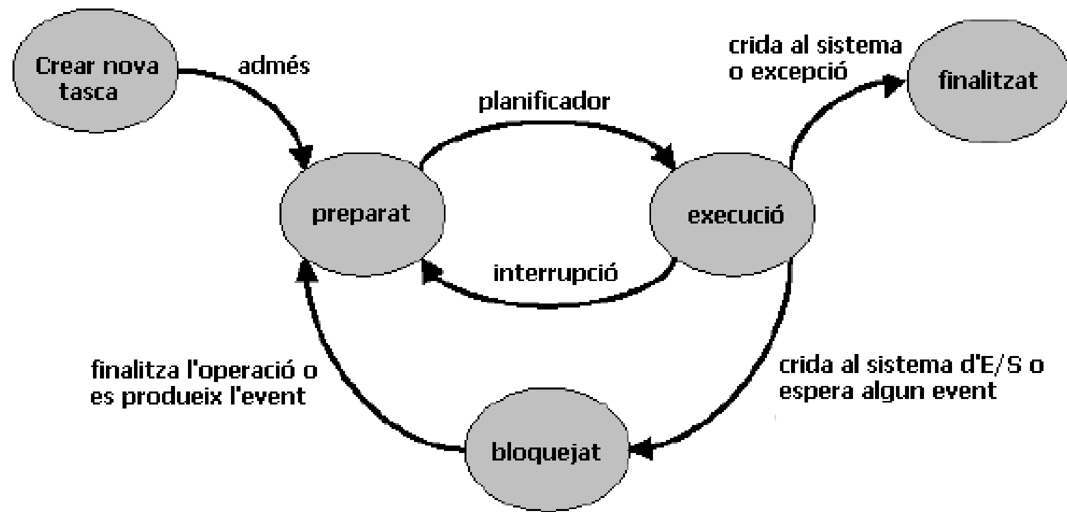

Figura 3.3: Diagrama de flux d'estats d'un procés.

3.2.3 Creació i destrucció

En un SO de temps compartit, constantment es creen i moren processos. Per tant, si volem aconseguir un bon rendiment del sistema, cal que les crides a sistema encarregades de crear i destruir processos estiguin implementades eficientment. En aquest apartat se citaran aquestes crides i s'explicarà el seu funcionament superficialment.

El procés init

El procés **init** és un procés que s'executa al final de l'arrancada del SO. La seva funció principal és executar els *scripts* d'inici i executar els processos pertinents. Aquest és el primer procés que crea el SO i el darrer en ser eliminat. És el pare de tots el processos i el seu PID és 1.

Creació d'un procés

El SO crea tots el processos seguint la mateixa metodologia. La creació de processos es realiza en dues parts independents i de forma seqüencial, mitjançant les crides a sistema de les famílies de **fork** i **exec**.

Per començar, la funció **fork** realitza una còpia idèntica del procés que la crida, però el nou procés tindrà un PID diferent i serà fill del procés en execució, del qual no heretarà els senyals pendents. Després **exec** carregarà el nou procés a l'espai d'adreces reservat per **fork** i començarà a executar-lo.

Al crear **fork** una còpia idèntica del procés que l'ha cridat, implicarà la còpia del contingut de l'espai d'adreces del procés pare a l'espai d'adreces que s'ha reservat pel procés fill. Això comporta un cost temporal inassequible per una crida com **fork**, que alhora pot resultar inútil si immediatament després de **fork** es crida a **exec**. Per tal d'evitar que això succeeixi s'utilitza el mètode **COW** (*Copy-on-Write*).

El mètode **COW** consisteix en la compartició dels recursos entre pares i fills, mentre aquests no tinguin modificacions. Quan es realitza **fork**, el nou procés estarà format pel seu PCB i dins l'espai d'adreces únicament contindrà una còpia idèntica de la taula de pàgines del procés pare. Això li permetrà accedir a l'espai d'adreces del pare, però solament podrà accedir-hi en mode lectura. Quan el fill hagi d'escriure en alguna pàgina de memòria, es copiarà la pàgina de l'espai d'adreces del pare al seu espai d'adreces, posteriorment es realitzarà la modificació de dades i finalment modificarà la seva taula de pàgines. També es compartiran els recursos fins que no tinguin modificacions. Utilitzant **COW** s'aconsegueix que la creació d'un procés es realitzi de forma ràpida i eficient.

Destrució d'un procés

Quan un procés finalitza la seva execució, el SO allibera tots els recursos reservats per part del procés i informa al procés pare del final de la seva execució. D'altra banda, si mentre un procés està en execució, aquest rep un senyal o una excepció que no està capacitat per manejar o ignorar, aleshores acabarà la seva execució de forma prematura.

La crida a sistema responsable de la finalització d'un procés és **exit**. Les funcions principals que realitza són:

- Alliberar l'espai d'adreces del procés de l'àrea d'usuari i tots els recursos associats al procés.
- Introduir el procés en estat TASK_ZOMBIE i enviar un senyal de finalització al procés pare.

- Cridar al planificador de processos perquè seleccioni el següent procés a executar.

La zona de memòria del nucli no s'elimina fins que el pare del procés utilitza una crida de sistema de la família **wait**, rebent així informació referent a la finalització del fill. A continuació es cridarà la funció **release_task** i els fragments que quedin del procés *zombie* seran eliminats. Aleshores, el procés abandonarà definitivament el sistema.

3.2.4 Estructures organitzatives

Anteriorment, s'ha comentat com els PCB's són la representació dels processos a l'interior del nucli del sistema. Els PCB's dels diferents processos estan organitzats sota diferents tipus d'estructures de dades, per tal de facilitar un accés ràpid als diferents processos, tant per part de l'usuari com per part del sistema. L'existència d'aquestes estructures de dades farà possible que el nucli pugui dur a terme la gestió i la planificació del sistema. La llista és l'estructura de dades més simple, i a partir d'aquesta es construeixen estructures de dades més complexes.

Les llistes

El tipus de llistes utilitzades al nucli són les llistes circulars doblement enllaçades. La implementació de les llistes és única, i definida a l'estructura **list_head**.

Com indica la figura 3.4, els camps **next** i **prev** apunten al següent i l'anterior element de la llista, respectivament. Gràcies a aquest tipus de llista genèrica, aconseguirem estalviar i simplificar el codi del nucli, perquè solament declarant un camp **list_head** en el context de qualsevol tipus d'estructura aconseguirem poder construir una llista d'elements d'aquella estructura.

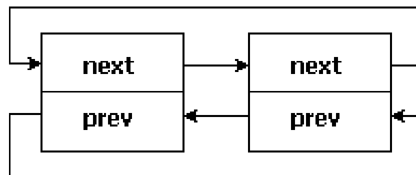

Figura 3.4: Representació d'una llista.

L'arbre jeràrquic

Les relacions familiars entre tots el processos estan representades en el context del nucli mitjançant una estructura de dades de tipus arborescent. Com s'ha comentat anteriorment, l'arrel de l'arbre, i pare de tots els processos, és el procés **init**.

Els camps del PCB encarregats d'establir l'arbre familiar dels processos són:

- **task_struct* parent**: punter al PCB del procés pare.
- **list_head children**: estructura de tipus llista que fa de capçalera a la llista que conté tots els processos fills.
- **list_head sibling**: estructura de tipus llista que conté tots els processos germans. La llista de tots els germans està referenciada per la llista *children* del procés pare.

3.2.5 Els fils d'execució

Com s'ha comentat anteriorment, els fils d'execució són tractats com processos normals. Cada fil d'execució té el seu **task_struct** i, en conseqüència, disposa d'un PID propi. Això implica que cada context d'execució pugui identificar-se de forma única.

En un procés multifil, el conjunt de recursos de tots els fils d'execució és comú i compartit. El conjunt de fils d'execució d'un procés s'anomena grup de fils.

Per poder ser identificats com un únic procés, tots els fils d'execució que conformen un grup de fils, tenen el valor del PID del primer fil del grup, emmagatzemat al camp **tgid** del PCB. Per tant, la crida a sistema **getpid** retorna *current->tgid* en lloc de *current->pid*. Per tant, el camp **tgid** en els processos normals és el pid del procés.

Un tipus particular de fils d'execució són els anomenats *fils del nucli*, els quals són processos que solament treballen sota espai de nucli. Un fil del nucli és un procés que treballa en segon pla (*background*), realitzant tasques de sistema i no disposa d'espai d'adreces en la zona d'usuari.

3.3 El planificador de processos

El **planificador de processos** és la part del nucli d'un SO multiprogramat, que assumeix la responsabilitat de distribuir el temps de CPU dels processadors dels quals consta el computador. Aquesta distribució es realitza entre el conjunt de processos del sistema

que esperen obtenir el control d'una CPU per tal de poder executar-se. Aquests són els processos preparats i el seu estat és TASK_RUNNING. L'estructura de dades que conté tots els processos preparats s'anomena **cua de preparats**. Aquesta cua de preparats està implementada mitjançant uns arrays de llistes anomenats arrays de prioritats.

A grans trets, l'algorisme de planificació estàndard del SO segueix els requeriments bàsics següents:

- Proporcionar un bon temps de resposta als processos interactius.
- Bon rendiment dels processadors que s'executen en segon pla.
- Repartiment equitatiu de les tasques entre diferents processadors en els sistemes multiprocessador.

El planificador és preemptiu, és a dir, apropiatiu. L'execució d'un procés es pot interrompre en qualsevol moment per l'arribada d'un procés més prioritari, el qual assolirà el control de la CPU.

Un procés en execució abandona la CPU i provoca l'execució del planificador de processos, segons les dues tendències següents:

- Per voluntat pròpia:
  - Ha finalitzat la seva execució.
  - Abandonament voluntari de la CPU tornant a la cua de preparats.
  - Suspès perquè necessita realitzar E/S.
- Abandonament forçat:
  - Arribada a la cua de preparats d'un procés més prioritari que l'actual en execució.
  - El planificador ha assignat un temps màxim de CPU al procés i aquest s'ha exhaurit.

L'abandonament de la CPU per part d'un procés, voluntari o no, provocarà una nova planificació. El planificador seleccionarà quin serà el procés en executar-se. El criteri principal, que utilitza el planificador per decidir el procés elegit enlloc d'un altre, és la prioritat. Cada procés té una prioritat associada.

El conjunt de regles, utilitzades per determinar quan i com seleccionar un nou procés per executar-se, s'anomena **política de planificació**. Per defecte, conviuen tres polítiques

de planificació diferents. Totes es regeixen per la mateixa escala de prioritats i, per tant, el planificador sempre seleccionarà el procés més prioritari, indiferentment de la política de planificació que aquest segueixi. La política de planificació que segueix un procés està especificada al camp **policy** del seu PCB. Les tres polítiques de planificació del nucli estàndard del SO són les següents:

- **SCHED_FIFO**: política FIFO (*First Input First Output*) per a processos en temps real. El procés tindrà un temps d'execució indefinit, abandonant solament la CPU per voluntat pròpia o per l'arribada d'un procés de temps real que tingui més prioritat.
- **SCHED_RR**: política *Round Robin* per a processos de temps real. Aquesta política garanteix la distribució equitativa de la CPU entre processos de temps real amb la mateixa prioritat.
- **SCHED_NORMAL**: política *Round Robin* per a processos de temps compartit.

La prioritat de qualsevol procés està situat a l'interval [0, MAX_PRIO-1]. Les prioritats [0, MAX_RT_PRIO-1] estan reservades als processos de temps real que segueixen una política de planificació **SCHED_RR** o **SCHED_FIFO**. Les prioritats [MAX_RT_PRIO, MAX_PRIO-1] estan reservades als processos normals de temps compartit que segueixen una política de planificació **SCHED_NORMAL**. Com més petit és el valor de la prioritat d'un procés, més prioritari és aquest. La prioritat dels processos de temps real és estàtica, invariable en el temps, mentre que la dels processos de temps compartit és dinàmica.

El planificador divideix el temps de CPU en èpoques. Al principi de cada època cada procés que segueix una política de planificació *Round Robin* (**SCHED_RR** i **SCHED_NORMAL**) té un *quantum* assignat. El *quantum* és la porció màxima de temps de CPU assignada a un procés durant una època. Cada cop que s'executa el rellotge del SO es decrementa el valor del *quantum* del procés en execució. Quan s'expira el *quantum* d'un procés, aquest abandona la cua de preparats, es calcula de nou el seu *quantum* per a la següent època i finalment es crida el planificador de processos. Cada cop que tots els processos preparats expiren el seu quantum, es produeix un canvi d'època.

3.3.1 La cua d'execució

La **cua d'execució** o **runqueue** és l'estructura de dades principal del planificador de processos. En els sistemes multiprocessador, els processos estan distribuïts equitativament entre tots els processadors del sistema. Cada CPU, com es pot veure en la figura 3.5, conté la seva cua d'execució pròpia i la missió de la qual és la gestió i planificació dels

processos que li han estat assignats. Una cua d'execució conté la cua de preparats d'una CPU i, a més a més, informació addicional per a la planificació. Cada procés preparat es troba exactament en un processador, i per tant solament en una cua d'execució.

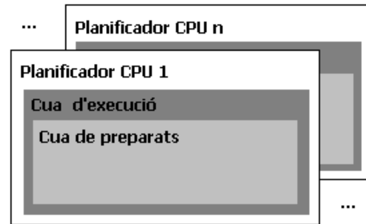

Figura 3.5: Planificador multiprocessador.

Com s'ha esmentat anteriorment, la cua de preparats forma part de la cua d'execució, concretament en un tipus d'estructura de dades que s'anomena array de prioritats.

Els camps més importants d'aquesta estructura són:

- **lock**: serveix per bloquejar la cua d'execució.
- **nr_running**: nombre de processos en estat TASK_RUNNING que es troben a la CPU de la cua d'execució.
- **nr_switches**: nombre de canvis de context entre processos que s'han dut a terme a la CPU on pertany la cua d'execució.
- **expired_timestamp**: guarda l'instant de temps d'inici d'una nova època. Aquest instant està expessat mitjançant el nombre de *jiffies* que han passat des de l'inici del SO.
- **nr_uninterruptible**: nombre de processos en estat TASK_UNINTERRUPTIBLE que es troben a la CPU de la cua d'execució.
- **nr_lowait**: nombre de processos de la cua d'execució que es troben suspesos realitzant E/S.
- **curr**: punter al procés que s'està executant actualment a la CPU, correspon al procés *current*.
- **idle**: punter al procés *swapper*. Aquest procés s'anomena procés ociós, el qual s'executa quan no exiteix cap procés en estat TASK_RUNNING.
- **arrays[2]**: array unidimensional que conté els arrays de prioritats de preparats i d'expirats.

- **active**: punter a l'array de prioritats de preparats.
- **expired**: punter a l'array de prioritats d'expirats. Els processos expirats són aquells que segueixen una política de planificació de tipus *Round Robin* i ha exhaurit el seu *quantum*. Es troben en aquest array esperant el canvi d'època.

3.3.2 Despertant i adormint processos

Un procés està dormint quan està bloquejat en espera d'un event. L'estat d'un procés bloquejat és TASK_INTERRUPTIBLE o TASK_UNINTERRUPTIBLE. Les raons per les quals un procés suspèn la seva execució i se situa a dormir són vàries:

- Fer E/S amb un dispositiu hardware.
- Esperar algun esdeveniment hardware.
- Bloqueig durant un interval de temps especificat.

Un procés bloquejat es despertat quan:

- Succeïx l'event causant del bloqueig.
- El procés és TASK_INTERRUPTIBLE i rep un senyal d'activació. Aleshores aquest serà despertat prematurament, sense que hagi succeït l'event causant del bloqueig.

Quan un procés adormit es desperta, el seu estat passa a ser TASK_RUNNING i torna a la cua de preparats. Els processos adormits es troben agrupats en un tipus d'estructures de dades anomenades **cues d'espera** o **waitqueues**. La figura 3.6 mostra com està formada per dues estructures clarament diferenciades, aquestes són la capçalera i el node. Està implementada com una llista doblement enllaçada. El primer element de la cua és la capçalera i els altres són els nodes.

Bloqueig d'un procés

Quan un procés esdevé adormit passa a un estat suspès (TASK_INTERRUPTIBLE o TASK_UNINTERRUPTIBLE), se situa en una cua d'espera i finalment es crida el planificador per seleccionar el nou procés que ocuparà la CPU. Tots els algorismes de bloqueig de processos seran similars, independentment de la naturalesa i del tipus

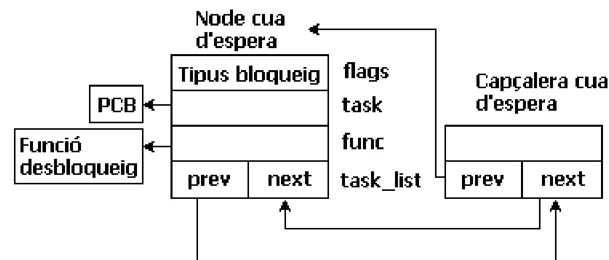

Figura 3.6: Implementació interna de la cua d'espera.

del bloqueig. El conjunt d'algorismes encarregats d'enviar a dormir els processos està implementat per la família de crides a sistema **wait_event**.

La funció **wait_event(wait_queue_head *wq, int condition)** col·loca a dormir un procés TASK_UNINTERRUPTIBLE amb un bloqueig de tipus no exclusiu. El seu funcionament detallat és el següent:

1. Es verifica si es compleix la condició *condition*. Si es compleix el procés *current* no es bloqueja i se surt de la funció.
2. Es crea un node de cua d'espera que apunta al procés *current*.
3. El procés és inserit al principi de la cua d'espera *wq*.
4. Se situa *current* en estat TASK_UNINTERRUPTIBLE.
5. Es verifica la condició *condition*:
   - Si es compleix, anar al pas 7.
   - Si no es compleix, es crida al planificador. El planificador traurà el procés suspès de la cua d'execució i seleccionarà el següent procés a executar. En aquest punt el procés perd el control de la CPU i es posiciona a dormir.
6. El procés ha despertat, ha tornat a la cua d'execució i el planificador li ha retornat el control de la CPU. Per tant, torna a ser *current*. Anar al pas 4.
7. Se situa current en estat TASK_RUNNING.
8. Es treu el procés de la cua d'espera *wq*. Finalitza la funció i el node de cua d'espera associat a *current*, com que és una variable local, és eliminat.

La macro **wait_event_interruptible(wait_queue_head *wq, int condition, int ret)** posa a dormir un procés TASK_INTERRUPTIBLE, amb un bloqueig de tipus no exclusiu. El seu funcionament pas a pas és el següent:

1. S'assigna *ret* el valor 0. Després es verifica si es compleix la condició *condition*. Si es compleix *current* no es bloqueja i se surt de la funció.
2. Es crea un node de cua d'espera que apunta al procés *current*.
3. El procés s'insereix al principi de la cua d'espera *wq*.
4. Se posiciona *current* en estat TASK_INTERRUPTIBLE.
5. Es verifica la condició *condition*:
   - Si es compleix, anar a 8.
   - Si no, anar a 6.
6. Es mira si *current* ha rebut un senyal:
   - Si l'ha rebut a la variable *ret*, li assignarem *-ERESTARSYS*. Anar a 8.
   - Si no l'ha rebut es crida el planificador. El planificador traurà el procés suspès de la **cua d'execució** i seleccionarà el següent procés a ser executat. En aquest context, el procés perd el control de la CPU i es posa a dormir.
7. El procés ha despertat, ha tornat a la cua d'execució i el planificador li ha tornat el control de la CPU. Per tant torna a ser *current*. Anar a 4.
8. Es posa *current* en estat TASK_RUNNING.
9. Es treu el procés de la cua d'espera *wq*. Es finalitza la funció i el node de cua d'espera associat a *current*, com que és una variable local, és eliminat.

Com es pot veure, aquestes funcions verifiquen si s'ha complert la condició de desbloqueig, abans de despertar del tot al procés. Això és degut a què un procés pot ser desbloquejat mitjançant una crida de la família **wake_up**, sense que realment hagi succeït l'event de desbloqueig.

Com podem observar, la interfície **wait_event** està orientada al bloqueig de processos no exclusiu. Similarment, la funció **add_wait_queue_exclusive** correspon a l'algorisme que realitza el bloqueig exclusiu de processos.

Desbloqueig d'un procés

Quan un procés es troba dormint, esperant un event, i aquest succeeix, aleshores aquest procés ha de tornar a la cua de preparats, canviar d'estat i sortir de la cua d'espera

on es trobava bloquejat. La família de crides **wake_up** és la interfície responsable de despertar un o més processos d'una determinada cua d'espera.

Un procés adormit està totalment bloquejat, per tant no pot despertar-se a sí mateix. L'encarregat de despertar un procés, i per tant qui utilitzarà **wake_up**, serà un altre procés o una RSI (*Rutina Servei d'Interrupció*).

Cada procés que sigui despertat en **wake_up**, serà despertat mitjançant la funció de desbloqueig especificada en el camp *func* del seu node de cua d'espera.

La funció del conjunt de crides que conformen la interfície **wake_up**, va orientada a l'activació d'un o més processos que es troben dintre d'una mateixa cua d'espera. Hem de tenir en compte que tots els processos que es troben en la mateixa cua d'espera estan dormits, esperant el mateix esdeveniment, que pot ser de naturalesa exclusiva o no. Un exemple d'event de naturalesa exclusiva és la utilització d'un recurs que no pot ser compartit, i per l'altra banda, un de naturalesa no exclusiva és l'expiració d'un temporitzador. Cada crida de la interfície **wake_up** actua directament sobre una cua d'espera determinada.

Una funció de desbloqueig és la funció encarregada de despertar un procés quan aquest està dormint. Tot procés bloquejat, conté al camp *func* una funció associada. La funció més important és **try_to_wake_up(wait_queue_t *p, unsigned state, int sync)**. El funcionament d'aquesta és el següent:

1. Mirar si l'estat del procés *p* està en *state*:
   - Si està especificat, anar a 2.
   - Si l'estat no està especificat en *state* (màscara que indica els possibles estats d'un procés a despertar), sortir de la funció i retornar a 0.
2. Mirar si el procés està lligat a un array de prioritats:
   - Si està lligat a un array de prioritats ja està despert. Es col·loca *p* en estat TASK_RUNNING i es retorna 0.
   - Altrament anar a 3.
3. Si l'estat en què està *p* és TASK_UNINTERRUPTIBLE aleshores *p->activated=-1*.
   - Mirar si el flag *sync* (indica un desbloqueig síncron) està activat:
     - Si està activat, crida a la funció **activate_task(rq,p)** on *rq* és un punter a la cua d'execució del procés al que apunta *p*. Aquesta funció situa *p* a l'array de prioritats actiu de *rq*.

  - Si no està activat:
    (a) Es crida la funció **activate_task(rq,p)**. Aquesta funció recalcula la prioritaritat dinàmica del procés *p* i:
      i. Si *p->activated=0*, és a dir, si no estava bloquejat en estat TASK_UNINTERRU i el procés és despertat per una interrupció hardware assignarem a *p->activated=2*.
      ii. Altrament, si el procés no és despertat per una interrupció Hardware, aleshores *p->activated=1*.

      Finalment se situa *p* a l'array de prioritats actiu de *rq*.
    (b) Es compara la prioritat de *p* amb la de *current*. Si el procés *p* és més prioritari que *current*, aleshores es cridarà a **resched_task(rq->curr)**. Aquesta funció activarà el flag *need_resched* del procés en execució, el qual provocarà una replanificació.

4. Finalment se situarà *p* en estat TASK_RUNNING i es retornarà 1, tot indicant que s'ha produït el desbloqueig del procés amb èxit.

Sobre aquest algorisme cal destacar dos aspectes:

- Encara que el procés sigui desbloquejat, aquest no s'elimina de la cua d'espera on estava dormint.
- El valor del camp *activated* del PCB és utilitzat posteriorment pel planificador, per tal de calcular les prioritats dinàmiques dels processos.

3.3.3 L'algorisme de planificació

La part principal del planificador de processos de qualsevol SO multiprogramat és l'**algorisme de planificació**. Aquest algorisme està implementat a la funció **schedule** dins de l'arxiu <*linux/sched.c*>.

Les funcions principals que ha de realitzar l'algorisme de planificació d'un SO són:

- Treure el control de la CPU al procés en execució.
- Seleccionar el nou procés que ocuparà la CPU d'entre tots els processos disponibles de la cua de preparats.
- Realitzar el canvi de context entre els processos.
- Donar el control de la CPU al procés nou.

El funcionament detallat de la funció **schedule** és el següent:

1. El punter *prev* senyala al descriptor del procés que perdrà el control de la CPU, és a dir, el procés *current*.
2. Es bloqueja l'accés a la cua d'execució i es desactiva la preemptivitat.
3. Si *prev* no està en estat TASK_RUNNING, és a dir, si aquest ha d'abandonar la cua d'execució, per bloqueig o per finalització de l'execució, aleshores:
   - Si *prev* està bloquejat en estat TASK_INTERRUPTIBLE, es verifica si ha rebut algun senyal de desbloqueig, i si és així, es posiciona *prev* a estat TASK_RUNNING.
   - Altrament, traurem *prev* de la cua d'execució.
4. Si no hi ha cap procés a la cua d'execució, *next*, que és el punter al següent procés que prendrà el control de la CPU, apuntarà al procés *swapper* (PID 0). Col·locarem el camp *expired_timestamp* a zero, per indicar que aquest ha de ser recalculat a l'inici d'una nova època. Anar a 8.
5. Si no hi ha cap procés a l'array de prioritats d'actius, aleshores s'intercanviaran els punters entre els arrays de prioritats d'actius i d'expirats. Com que s'ha produït un canvi d'època, posarem el camp *expired_timestamp* a zero, per tal que aquest sigui recalculat per la funció *scheduler_tick*.
6. Mitjançant la funció **sched_find_bit** sobre l'array de prioritats d'actius trobarem la prioritat del procés més prioritari de l'array. Posteriorment, accedirem a la llista de l'array d'actius corresponent a la prioritat obtinguda, i *next* apuntarà al primer procés d'aquesta llista.
7. Si el procés és de temps compartit, acaba d'entrar a la cua d'execució i no està bloquejat en estat TASK_UNINTERRUPTIBLE aleshores el valor del camp *acivated* del PCB serà més gran que zero. Al moment es recalcularà la prioritat dinàmica del procés.
8. Si *prev* és diferent de *next* es realitza el canvi de context entre aquests processos mitjançant les funcions **context_switch** i **finish_task_switch**.
9. Es desbloqueja l'accés a la cua d'execució i s'activa la preemptivitat.
10. Es verifica el bit de preemptivitat del nou procés *current* (*next*):
    - Si està activat, serà necessària la replanificació. Per tant anirem a 1.
    - Si està desactivat s'acaba l'execució de la funció **schedule**.

Com es pot observar en aquesta funció, s'aconsegueix l'acompliment de dos requeriments que són fonamentals per aconseguir un algorisme de planificació eficient. Aquests són:

- Rapidesa d'execució deguda a la simplicitat de l'algorisme.
- El temps d'execució de l'algorisme serà constant, independentment del nombre de processos que es trobin a la cua d'execució.

Un cop tenim l'algorisme de planificació especificat, ens centrarem en qui, com i sota quines circumstàncies pot ser invocat aquest. Es distingeixen dos tipus diferents d'invocacions a la funció **schedule**:

- **Directa**: el procés en execució decideix abandonar la CPU i crida **schedule**.
- **Indirecta**: la preemptivitat fa que el SO cridi **schedule** per tal d'expropiar el control de la CPU al procés que està actualment en execució.

Invocació directa

Es parla d'invocació directa del planificador quan el procés *current* decideix voluntàriament abandonar la CPU, cedint així el control d'aquesta al planificador del SO per tal que elegeixi un nou procés. Els diferents casos d'invocacions directes al planificador són els següents:

- **Final d'execució d'un procés**: quan un procés acaba la seva execució se situa en estat TASK_ZOMBIE i crida la funció **schedule**. El procés ja ha finalitzat la seva execució i, per tant han de ser desallotjat de la cua d'execució. Aquesta tasca és portada a terme per la funció **finish_task_switch**, que està integrada a **schedule**. Aquesta funció, juntament amb la funció **context_switch**, forma la part del planificador de processos, responsable de realitzar el canvi de context entre els processos.
- **Senyal parada**: el procés que s'està executant quan rep el senyal SIGSTOP, SIGSTP, SIGTTIN, SIGTTOU o quan rep qualsevol senyal, quan està essent depurat per un altre programa, passa a estat TASK_STOPPED i és bloquejat. Llavors es crida la funció **schedule** que s'ocuparà de traure *current* de la cua d'execució i d'elegir el nou procés *current*. Quan el procés rebi el senyal d'activació SIGCONT es desbloquejarà i es tornarà a situar a la cua de preparats en estat TASK_RUNNING.

- **Bloqueig d'un procés**: el procés se situa en estat TASK_INTERRUPTIBLE o TASK_UNINTERRUPTIBLE i és bloquejat, en espera que succeeixi un event.

- **Abandonament voluntari**: Mitjançant la crida a sistema **yield**, el procés *current* abandonarà la CPU sense bloquejar-se, és a dir, es mantindrà a la cua d'execució en estat TASK_RUNNING. El funcionament d'aquesta funció és el següent:

  1. Bloqueig de la cua d'execució a la qual pertany *current*.
  2. Si *current* és un procés de temps compartit, aleshores aquest és extret de l'inici de la seva llista i inserit a l'array d'expirats. Per tant, aquest procés no podrà tornar a ser planificat fins la següent època.
  3. Si *current* és un procés de temps real, aleshores aquest és extret de l'array d'actius i inserit a l'array d'expirats. Per tant, aquest procés no podrà tornar a ser planificat fins a la següent època.
  4. Es desbloqueja la cua d'execució i es crida la funció **schedule**.
  5. Quan el procés recupera el control de la CPU finalitza l'execució de la funció retornant 0.

Invocació indirecta

Es parla d'invocació indirecta, o bé ociosa, quan el planificador de processos del SO decideix que el procés *current* ha d'abandonar la CPU, cridant la funció **schedule** per tal que elegeixi el nou procés *current*.

Si la funció de planificació solament pogués ser invocada de forma directa, és a dir, pel procés en execució, aleshores un procés podria estar executant-se indefinidament en una CPU. La solució a aquest problema radica al nucli del SO, concretament el planificador de processos, per tal que determini quan un procés ha d'abandonar la CPU i cridi **schedule**. Aquesta tasca s'anomena preemptivitat.

Els diferents casos d'invocacions al planificador, causades per la preemptivitat, que es poden trobar són els següents:

- **Expiració quantum**: el procés en execució ha expirat el seu *quantum*. Es cridarà al planificador i d'aquesta forma el procés perdrà el control de la CPU. La funció del planificador, encarregada del decrement del *quantum*, i de controlar quan aquest ha expirat és **scheduler_tick**. Es pot donar el cas en què el planificador consideri que un procés porta massa estona a la CPU i, per tal d'afavorir la interactivitat, aquest procés abandoni la CPU sense haver expirat el seu *quantum*.

- **Procés nou més prioritari**: arribada a la cua de preparats d'un procés més prioritari que *current*. Això provocarà que es produeixi una replanificació. La funció **try_to_wake_up** és la responsable de comparar la prioritat de cada procés que entra a al cua d'execució amb la de *current*, i activar la preemptivitat, si així s'escau.

- **Canvi prioritat procés**: es pot donar el cas en el qual es canviï la prioritat d'un procés que es trobava a la cua d'execució, i que aquest passi a ser més prioritari que *current*. Això provocarà una replanificació.

El funcionament de la preemptivitat és el següent:

- Quan el planificador determina que ha d'expropiar la CPU a *current*, activa el flag *need_resched* d'aquest procés mitjançant la funció **set_tsk_need_resched**. Aquest flag activat és un missatge pel planificador de processos, el qual li indica que s'ha de produir una replanificació el més aviat possible. Cada procés té un bit *need_resched* propi, i aquest es troba al camp flags de l'estructura *thread_info*.

- Quan es validi el bit *need_resched*, mitjançant la funció **need_resched**, actuarà la preemptivitat i es cridarà **schedule** si cal. Segons el lloc on es produeixi aquesta validació, distingirem dos tipus diferents de preemptivitats. Aquestes són:

  - **Preemptivitat d'usuari**: la validació es produeix mentre el procés s'executa en l'espai d'usuari. Concretament, es verifica el valor del camp *need_resched* cada cop que el procés acaba d'executar una crida a sistema, o un manegador d'interrupcions i desitja tornar a executar-se en l'espai d'usuari.
  - **Preemptivitat del nucli**: la validació es produeix mentre el procés s'executa en l'espai de nucli. El fet que un procés pugui ser interromput, mentre s'executa en espai de nucli, és una característica desitjable en tots els SO de temps real. La preemptivitat a nivell de nucli no és total, ja que hi ha regions del nucli on aquesta ha d'estar desactivada. Cada procés té el camp *preempt_count* a l'interior de l'estructura **thread_info**, associada al seu PCB. El seu valor indica el nombre de bloquejos a la preemptivitat que té el procés. Per tant, la preemptivitat solament estarà activada quan *preempt_count* valgui zero.

    Es verifica el valor de *need_resched* quan es torna a l'espai de nucli després d'atendre un manegador d'interrupcions, i cada cop que *preempt_cout* es col·loca a zero.

- Finalment, en el context de la funció **schedule**, com que ja s'està produint l'exploració, es col·loca el bit *need_resched* del procés, que abandona la CPU a zero, mitjançant la funció **clear_task_need_resched**.

3.3.4 La temporització del planificador

La noció del temps és un element molt important a tenir en compte en el nucli d'un SO, ja que molts dels esdeveniments que tenen lloc dintre d'aquest són orientats al temps. Aquests esdeveniments poden ser de naturalesa periòdica o executar-se de forma eventual.

El **temporitzador del sistema** és la part del hardware d'un computador encarregada de controlar el pas del temps. Aquest temporitzador funciona utilitzant una font de temps, que pot ser un rellotge digital o la mateixa freqüència del processador. La funció del temporitzador és disparar-se amb una freqüència programada, és a dir, cada cert nombre de *ticks* de rellotge, el temporitzador del sistema llença una interrupció hardware. Aquesta freqüència s'anomena *tick rate* i el seu valor es indicat al temporitzador durant l'arrencada del SO.

El *tick rate* està definit en la constant simbòlica HZ i el seu valor depèn de l'arquitectura. El valor de HZ ha de complir el compromís de ser suficientment gran perquè els events, que succeeixin en un determinat instant de temps, siguin acotats el més ràpidament possible, i alhora ser suficientment petit perquè l'execució massa continuada de les rutines relacionades amb el temporitzador no afectin negativament al rendiment del sistema. Actualment, el valor de HZ és 1000, és a dir, aquest es dispara cada milisegon.

El **temporitzador d'interrupcions** és el software del SO encarregat de manegar les interrupcions del temporitzador del sistema. Algunes de les funcions que s'hi duen a terme són:

- Actualització de la data i l'hora.
- Actualització del temps absolut transcorregut des de l'arrancada del sistema. Aquest és expressat en *jiffies* i s'emmagatzema a la variable global *jiffies*. Un *jiffie* correspon a un *tick* del temporitzador del sistema.
- Execució de les rutines del planificador de processos depenents del temps.
- Execució de les rutines relacionades amb els temporitzadors dinàmics quan aquests expiren.

Les funcions més importants del planificador de processos depenents del temps, és a dir, les gestionades pel temporitzador d'interrupcions són:

- **schedule_timeout**: afegeix un temporitzador dinàmic al procés que la crida. D'aquesta forma assegurem que al finalitzar el temporitzador, el procés que ha abandonat la CPU torni a la cua d'execució si encara està bloquejat.

- **schedule_tick**: tots els processos que segueixen una política d'execució *Round Robin* disposen d'un temps màxim d'execució anomenat *quantum*. Cada *tick* de rellotge provoca que el *quantum* del procés en execució es decrementi una unitat. Aquest decrement es realitzat per aquesta funció, la qual és periòdica, ja que es cridarà pel temporitzador d'interrupcions a cada *tick* de rellotge.

- **load_balance**: cada processador disposa d'una cua d'execució pròpia i aquesta s'encarrega de planificar el conjunt de processos que li han assignat. Per tant, cada processador té el seu propi planificador de processos. El balancejador de càrrega és la part del SO responsable de mantenir una càrrega de processos equitativa entre totes les cues d'execució del sistema.

  Sota sistemes uniprocessador aquesta funció mai és invocada. Per altra banda, en sistemes multiprocessador aquesta funció és invocada per:

  - La funció **schedule** si la cua d'execució està buïda.
  - El temporitzador mitjançant la funció **schedule_tick**. Si el sistema està ociós la crida es realitzarà cada *tick* de rellotge i sino es realitzarà cada cop que expiri el *quantum* del procés en execució.

3.3.5 La inicialització del planificador

Durant l'arrancada del SO totes les parts d'aquest, entre elles el planificador de processos, són inicialitzades. La funció que conté les inicialitzacions de la cua d'execució s'anomena **sched_init** i el seu funcionament, pas a pas, és el següent:

1. Per cada cua d'execució del sistema:
   - Els punters dels arrays de prioritats *active* i *expired* apuntaran a *arrays[0]* i *arrays[1]* respectivament.
   - Es col·loca *nr_lowait* a 0.
   - S'inicialitza el camp *lock*, el qual serveix per bloquejar la cua d'execució.
   - Cadascun dels arrays de prioritats s'inicialitza.
2. Es col·loca *current* com a procés en execució, i com a procés ociós *idle* de la cua d'execució de la CPU que executa aquesta funció.

Una altra funció interesant de l'arrencada del sistema és **init_idle(task_struct *idle, int cpu)**. Aquesta funció inicialitza el procés *idle* perquè faci de procés *swapper* a la CPU especificada al camp *cpu*. El seu funcionament pas a pas és el següent:

1. Es bloqueja la cua d'execució on es troba el procés *idle* i la cua d'execució corresponent a la CPU que executa la funció.

2. S'extreu *idle* de la cua d'execució. Com que el procés *swapper* no es trobarà a cap array de prioritats el camp array apuntarà a NULL.

3. Es col·loca el procés en estat TASK_RUNNING amb una prioritat MAX_PRIO.

4. Es desbloquegen les cues d'execució prèviament.

5. S'activa el flag *need_resched* al procés *idle* i es posa el seu camp *preempt_count* a 0.

3.3.6 La prioritat dels processos

La planificació de processos està basada en prioritats, és a dir, cada procés té una prioritat associada. La funció de planificació, per tal d'elegir quin serà el procés a qui cedirà el control de la CPU, escollirà el procés més prioritari de la cua de preparats.

El valor de la prioritat que pot estar associat a un procés està inclòs a l'interval **[0,MAX_PRIO-1]**. Com més petit sigui aquest valor, més prioritari serà el procés que hi estigui associat. La prioritat que pot tenir un procés que segueix una política de planificació de temps real (**SCHED_RR** o **SCHED_FIFO**) està situat al rang **[0, MAX_RT_PRIO-1]**. La prioritat dels processos que segueixen una política de planificació de temps compartit (**SCHED_NORM**) està situat al rang **[MAX_USER_RT_PRIO, MAX_PRIO-1]**. El procés *swapper*, com que solament s'executa quan el sistema està ociós, és el menys prioritari i la seva prioritat és **MAX_PRIO**.

- **MAX_RT_PRIO=MAX_USER_RT_PRIO=100**
- **MAX_PRIO=140**

La prioritat dels processos de temps compartit es pot expressar amb una altra escala de prioritats anomenada **nice**. El seu rang és [-20,19] i el *nice* assignat a un procés per defecte és 0. El mapeig entre l'escala de prioritats normal i l'escala *nice* és total, ja que aquesta última representa un subconjunt de la primera. Per tant, el rang [-20,19] del *nice* representa el rang de prioritats **[MAX_USER_RT_PRIO, MAX_PRIO-1]**. La prioritat expressada en l'escala *nice* no es guarda al PCB.

En els processos de temps compartit ens trobem dos tipus de prioritats:

- **Prioritat estàtica**: és la prioritat que té el procés des del moment de la seva creació i es manté invariable en el temps fins que el procés finalitza la seva execució. A partir d'ella es calcula el *quantum*. La prioritat estàtica d'un procés es modificable sota privilegis *root* a través d'una específica crida a sistema o amb la comanda **schedtool** (Apèndix A.1.3). Es troba al camp *static_prio* del PCB.

- **Prioritat dinàmica**: és la prioritat real que té un procés en un determinat instant de temps. Inicialment val el mateix que la prioritat estàtica però, segons el comportament del procés (temps que està en la CPU, temps d'espera, temps de bloqueig), aquesta s'incrementarà o es decrementarà lleugerament. És la prioritat que té en compte l'algorisme de planificació. S'emmagatzema al camp *prio* del PCB.

En els processos de temps real solament hi ha un tipus de prioritat. Aquesta és estàtica i, a més a més d'expressar-se en l'escala de prioritats normal, s'expressa amb l'escala de prioritats de temps real. La prioritat de temps real s'emmagatzema al camp *rt_priority* del procés. Al contrari que en l'escala de prioritats normal, com més gran és el valor emmagatzemat a *rt_priority* més priotari més prioritari és el procés. L'interval de prioritats de temps real vàlides és **[1, MAX_USER_RT_PRIO-1]**. La prioritat en escala normal d'un procés de temps real es troba expressada al camp *prio* del PCB. La correspondència entre *prio* i *rt_priority* és la següent:

- **prio=MAX_USER_RT_PRIO-1 - rt_priority**

Per tant si *MAX_USER_RT_PRIO=100* el conjunt de prioritats de temps real [1,99] es representarà de la prioritat 98 fins a la 0.

En els processos que segueixen una política de planificació *SCHED_NORMAL* el camp *rt_priority* val 0.

Els processos de temps real també tenen *nice*, i aquest està expressat en l'escala normal de prioritats dintre del camp *static_prio* del PCB, de la mateixa forma que en els processos de temps compartit. La diferència recau en què el valor *static_prio* en els processos de temps real no indica la prioritat d'aquest, únicament serveix per a calcular el *quantum* del procés a l'inici de cada època. Tots els processos del sistema que utilitzen una política de planificació *Round Robin* (**SCHED_RR** i **SCHED_NORMAL**) calculen el seu *quantum* mitjançant la funció **task_timeslice**. Aquesta funció transforma el *nice* emmagatzemat al camp *static_prio* en un *quantum* que s'emmagatzemarà al camp *time_slice*. La funció **task_timeslice** assignarà *quantums* compresos entre 10 i 200 milisegons, depenent del *nice* del procés. Com més petit sigui el valor de *static_prio* més gran serà el *time_slice*.

Prioritat dinàmica

Com s'ha esmentat anteriorment, la prioritat d'un procés de temps compartit és dinàmica, és a dir, varia en el temps segons el comportament del procés. La prioritat dinàmica d'un procés es calcula a partir de la prioritat estàtica i del grau d'interactivitat del procés.

És desitjable que el planificador de processos d'un SO de temps compartit doni més prioritat als processos interactius, és a dir, orientats a l'E/S, que als processos CPU intensius. El PCB no té especificada en cap camp la interactivitat d'un procés i, per tant, ha de ser el propi SO qui calculi la interactivitat d'un procés.

El grau d'interactivitat d'un procés es calcula mitjançant un conjunt d'heurístiques que utilitzen com a base les següents mesures:

- Temps d'execució en CPU.
- Temps d'espera a la cua de preparats.
- Temps de bloqueig d'E/S.

El planificador de processos és qui obté tots aquests valors. Per tal de calcular aquests intervals de temps es necessiten emmagatzemar determinats instants de temps. Aquests són emmagatzemats a la variable *timestamp* del PCB. Aquesta variable emmagatzema els següents instants de temps:

- Procés entra a la cua d'execució.
- Procés pren el control de la CPU.
- Procés perd el control de la CPU.

El temps que un procés s'executi en una CPU decrementarà el camp *sleep_avg* del PCB d'aquest. El temps que el procés estigui dormint s'incrementarà al camp *sleep_avg*. A partir del valor d'aquest camp s'avaluarà l'interactivitat del procés:

- Si el procés és interactiu, és a dir, si ha estat molt temps bloquejat, i poca estona executat-se, rebrà una bonificació en la seva prioritat, que pot variar de -1 a -5.
- Si el procés és CPU intensiu, és a dir, si s'ha executat molt temps en una CPU i ha estat bloquejat poca estona, rebrà una penalització en la seva prioritat de +1 a +5.

La funció encarregada de calcular la prioritat dinàmica d'un procés de temps compartit s'anomena **effective_prio**. La prioritat dinàmica d'un procés es calcula quan:

- El *quantum* del procés expira: es recalcularà el *quantum* i la prioritat dinàmica.
- El procés entra a la cua d'execució: Quan un procés vol entrar a la cua de preparats, es calcula la seva prioritat dinàmica. Posteriorment aquest és inserit a l'array de prioritats que li correspon.
- El planificador selecciona un procés que ha entrat a la cua d'execució després d'un bloqueig NO_UNINTERRUPTIBLE: es recalcula la prioritat dinàmica d'aquest, tenint en compte el temps que ha estat esperant ser elegit pel planificador de la cua d'execució.

3.4 El nou planificador de processos

Com veurem a continuació, el planificador de processos estàndard està clarament limitat, i no ofereix els mecanismes suficients per reservar en tot moment porcions de CPU per aplicacions concretes d'usuaris privilegiats, i així poder implantar un sistema de QoS en l'entorn de CPU.

A través del planificador de processos de **Con Kolivas**, el qual està disponible per al nucli estàndard i aplicable a través d'un *patch*, ens permet assolir els reptes esmentats anteriorment. Aquests canvis no formen part del nucli estàndard, per tant, es necessari compilar el nou nucli optimitzat.

L'acció de reservar una porció de CPU no significa disposar d'un percentatge de CPU per a l'execució d'un procés en tot moment, perquè no existeix planificador de processos que contempli aquesta idea. Els motius d'aquesta limitació són variats, però el planificador de processos està basat a partir d'un sistema d'assignació de prioritats que beneficia la multitasca és el principal responsable. Per altra banda, amb el nou planificador de processos, podem extreure una idea que permetrà obtenir un concepte semblant. El nou planificador de processos, proporciona una nova política que permet seleccionar els processos més prioritaris del sistema i limitar l'ús de la CPU sota un determinat percentatge d'utilització, a través d'uns paràmetres de configuració.

El nou planificador de processos incorpora cinc polítiques de planificació. Per una banda, disposem de les tres polítiques del planificador estàndard, que són **SCHED_RR**, **SCHED_FIFO** i **SCHED_NORMAL**. Per altra banda, incorpora dues noves polítiques, que són **SCHED_ISO** i **SCHED_IDLEPRIO**.

Un cop s'han esmentat les sis polítiques de planificació disponibles del nou planificador de processos, aquestes són classificables sota dos grups:

- Polítiques de temps real:
  - **SCHED_FIFO**
  - **SCHED_RR**
- Polítiques de temps compartit.
  - **SCHED_NORMAL**
  - **SCHED_ISO**: definida detalladament més endavant.
  - **SCHED_BATCH**: els processos sota aquesta política disposen d'un valor més gran del camp *time_slice*, en referència al valor retornat per la funció **task_timeslice**. Per altra banda, l'execució es pot interrompre per part d'altres processos, si aquests garanteixen l'interactivitat del sistema.
  - **SCHED_IDLEPRIO**: és una política similar a **SCHED_BATCH**, però a diferència d'aquesta, únicament utilitza la CPU durant el temps *idle*, és a dir, si s'executa el procés *swapper*. Aquesta política s'empra habitualment per a aplicacions no interactives.

Aquestes cinc polítiques conviuen de la mateixa forma que al planificador estàndard, és a dir, totes es regeixen per la mateixa escala de prioritats i, per tant, el planificador sempre seleccionarà el procés més prioritari, independentment de la política de planificació que aquest segueixi. Com s'ha esmentat anteriorment, la política de planificació que segueix un procés està especificada al camp *policy* del seu PCB.

Cal remarcar que el nou planificador de processos segueix estrictament el funcionament intern del planificador estàndard, és a dir, criteris com la cua d'execució, els mecanismes per despertar i adormir els processos, i l'algorisme de planificació no canvien en absolut sota el nou planificador de processos. Aquest nou planificador de processos simplement defineix un nou sistema de prioritats dels processos, per tal que les noves polítiques de planificació treguin un benefici exclusiu no ofert per la resta de polítiques.

El nou planificador de processos introdueix un sistema propi de prioritats dels processos, donat que la selecció de la política de planificació és una decisió realitzada automàticament per l'aplicació a través de la crida a sistema **sys_setscheduler** a l'interior del codi. Encara que es desitjable especificar la política desitjada manualment quan s'inicia una aplicació, usualment els processos interns no utilitzen una política exclusiva del planificador de processos proposat per **Con Kolivas** (**SCHED_ISO** o **SCHED_IDLEPRIO**), sinó les definides sota el planificador estàndard (**SCHED_RR**, **SCHED_FIFO** i **SCHED_NORMAL**).

Com s'ha mencionat anteriorment, la planificació de processos està basada en prioritats, és a dir, cada procés té una prioritat associada. La funció de planificació, en el moment de seleccionar quin serà el procés a qui cedir el control de la CPU, elegirà el procés més prioritari de la cua de preparats.

De la mateixa forma que el planificador de processos estàndard, el nou planificador situa dos tipus de prioritats destinades als processos de temps compartit, l'estàtica i la dinàmica. Per altra banda, els processos de temps real solament disposen de la prioritat estàtica.

Sota el nou planificador de processos, el valor de la prioritat que pot estar associat a un procés està dintre de l'interval **[-1,MAX_PRIO-1]**. Com més petit sigui aquest valor més prioritari serà el procés que el tingui associat. La prioritat que pot tenir un procés que segueixi una política de planificació de temps real (**SCHED_FIFO** i **SCHED_RR**) oscil·la entre **[0,MAX_RT_PRIO-1]**. La prioritat dels processos que segueixen una política de planificació de temps compartit (**SCHED_NORM** i **SCHED_IDLEPRIO**) oscil·la entre **[MAX_USER_RT_PRIO, MAX_PRIO-1]**. Aquest nou sistema de prioritats defineix aquestes constants simbòliques amb els valors per defecte:

- **MAX_RT_PRIO=MAX_USER_RT_PRIO=100**
- **MAX_PRIO=140**

Com hem definit anteriorment, en el context del rang de possibles prioritats existeix el valor **-1**. Aquest valor està reservat exclusivament a la política **SCHED_ISO**, per tant, un procés amb aquesta prioritat assignada serà el més prioritari del sistema. D'aquesta manera aconseguim que els processos de temps compartit sota la política **SCHED_ISO** puguin assolir una prioritat superior a qualsevol procés del sistema.

La política de planificació **SCHED_ISO** és una política dissenyada per llançar processos sota la màxima prioritat, sense necessitat de convertir-se en una política de temps real (**SCHED_RR** o **SCHED_FIFO**). Quan un procés està sota la política **SCHED_ISO** obté baixa latència i pot obtenir la totalitat del recurs de CPU com la política de temps real **SCHED_RR**, però a diferencia d'aquesta existeixen els paràmetres de configuració *iso_cpu* i *iso_period*, els quals permeten limitar l'ús del recurs de CPU als processos.

L'ús incontrolat d'aquesta política, per part d'alguns processos de CPU intensiu que la tinguin associada, pot provocar una indesitjable sobrecàrrega del sistema. Per evitar aquest problema, disposa de dos paràmetres de configuració:

- **echo limit > /proc/sys/kernel/iso_cpu**: correspon al percentatge de CPU

que un determinat procés, amb aquesta política de planificació, no pot superar. En cas de superar aquest límit establert, serà penalitzat durant el temps indicat a través del següent paràmetre. El valor per defecte és 70, que correspon al 70%.

- **echo temps > /proc/sys/kernel/iso_period**: correspon al temps de penalització que rebrà un procés que superi el percentatge de CPU assignat. La penalització consisteix en situar-lo dintre del rang de **[MAX_USER_RT_PRIO, MAX_PRIO-1]** durant el temps indicat. El valor per defecte és 5, que correspon a 5 segons.

Sota aquesta política, mentre no sigui superat el limit establert, solament es pot reduir la prioritat del procés. Per altra banda, si el límit s'ha superat, s'ignorarà qualsevol mecanisme d'augment de la prioritat.

A través de la manipulació d'aquests paràmetres, podem simular un sistema de reserva de porcions de CPU.

Donat que necessita modificar les prioritats d'aquells processos que superin el limit establert, la política **SCHED_ISO** no correspon a una política de temps real. Les polítiques de temps real solament disposen de la prioritat estàtica. Per altra banda, les polítiques de temps compartit disposen de dos tipus de prioritats, les estàtiques i les dinàmiques. Per aquest motiu, funciona com la política **SCHED_NORMAL**, perquè necessita modificar les prioritats per tal de realitzar les penalitzacions.

Un cop vist el funcionament intern de la política **SCHED_ISO**, podem observar com l'escala dels processos de temps compartit *nice* no és suficient per realitzar reserves de porcions de CPU. Perquè el seu rang real és **[MAX_USER_RT_PRIO, MAX_PRIO-1]**, malgrat que s'utilitzi una escala diferent com *nice*, la qual correspon al rang [-20,19]. La correspondència entre l'escala de prioritats normal i l'escala *nice* és total, perquè aquesta última escala representa un subconjunt de la primera. Com podem observar, qualsevol procés de temps real sempre serà més prioritari, donat que tindrà una prioritat situada sota el rang **[0, MAX_USER_RT_PRIO-1]**.

Com s'ha comentat anteriorment, el processos per defecte no utilitzen les polítiques del nou planificador. Per aquest motiu s'ha desenvolupat el script **qosCPU.sh** (Apèndix B) que detecta automàticament els processos i fills d'aquests del grup d'usuaris privilegiats, per sotmetre'ls sota la política **SCHED_ISO**. A l'interior del script es configuren els paràmetres *iso_cpu* i *iso_period* mencionats anteriorment. Aquest s'activa a través del **Script QoS** (Apèndix B), el qual s'encarrega de l'activació i desactivació dels scripts **qosNETWORK.sh** i **qosCPU.sh**.

3.5 Experimentació

En aquesta secció s'hi exposen els resultats obtinguts amb l'execució del script **provesCPU.sh** (Apèndix C.2). També es mostraran els resultats sobre la latència a través del benchmark software **InterBench** [16] (Apèndix A.3). L'objectiu es comprovar el funcionament del nou planificador de processos respecte el planificador estàndard. A grans trets, es mostrarà com realitzant proves idèntiques simultàniament, entre usuaris privilegiats amb QoS i usuaris normals, els primers obtindran un temps de còmput inferior. Aquests resultats s'obtenen gràcies al nou planificador de processos de **Con Kolivas**, conjuntament amb el **Script QoS** (Apèndix B) que s'encarrega de proporcionar QoS als usuaris privilegiats d'aquest recurs.

Com s'ha explicat anteriorment, reservar una porció de CPU no consisteix en disposar d'un percentatge de CPU per l'execució d'un procés en tot moment, perquè no existeix planificador de processos que contempli aquesta idea. Per altra banda, el nou planificador de processos proporciona una idea que permetrà obtenir un concepte semblant. A partir de la política **SCHED_ISO** gestionada exclusivament pel script **Script QoS**, el qual s'encarrega de seleccionar a nivell d'usuari quins seran els processos més prioritaris del sistema i limitar l'ús de la CPU d'aquests processos segons els valors dels paràmetres *iso_cpu* i *iso_period* obtenim una nova definició del concepte.

Les proves s'han realitzat sota valors de *iso_cpu=90* i *iso_period=5*. Altres valors d'aquest paràmetres no han produït diferències significatives, per aquest motiu no es mostraran. Per altra banda, si configurem *iso_cpu=100* els processos no seran mai penalitzats, perquè un procés no pot superar l'ús del 100% de CPU. Per tant, les proves realitzades sota aquest valor destruien totalment la interactivitat del sistema, i aquest no responia fins que tots els processos privilegiats finalitzaven l'execució.

Malgrat que l'entorn d'experimentació s'ha exposat a l'introducció, és molt important recordar que el computador elegit per realitzar les proves està dotat de 2 CPU's, donat que és un *Intel Core 2 Duo*.

3.5.1 Resultats

La figura 3.7 mostra la latència mitjana de scheduling. La latència es considera com el temps transcorregut des que es posa a dormir un procés fins que és executat de nou. Aquest prova s'ha realitzat a partir del benchmark software **InterBench** (Apèndix A.3), com usuari privilegiat amb QoS i com usuari normal del sistema. La diferència és substancial, però clarament raonable, donat que l'usuari privilegiat executa aquest procés sota la prioritat màxima del sistema, mentre que l'usuari normal està subjecte al sistema d'assignació de prioritats del planificador de processos.

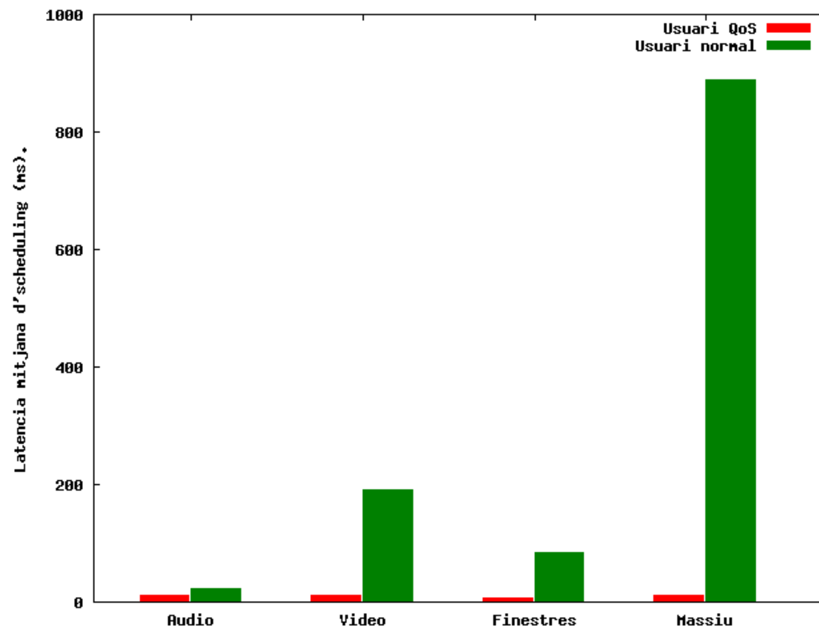

Figura 3.7: Latència mitjana de scheduling.

A continuació es mostren els gràfics obtinguts i la sortida de l'execució de les proves realitzades amb el script **provesCPU.sh**.

La figura 3.8 mostra l'execució de dos processos privilegiats, per tant, cada procés pren una CPU diferent. En aquest cas, un processador acaparà tota la CPU i l'atra CPU contindrà a la seva cua d'execució l'altre procés, més la resta de processos del sistema, per aquest motiu tarda 2 segons més. El temps d'execució del primer és 52 segons i el temps del segon és 54 segons. de Com podem observar, l´ús de les dos CPU's està clarament limitat al 90%.

La figura 3.9 mostra l'execució de dos processos, un privilegiat i l'altre no. El primer s'apoderà d'una CPU i l'altre procés haurà de compartir la CPU amb la resta de processos del sistema, els quals usalment seran més prioritaris. El temps d'execució del procés privilegiat és 53 segons i el temps del procés normal és 58 segons. Per tant, obtenim 5 segons de pèrdues respecte la prova anterior.

La figura 3.10 mostra l'execució de dos processos normals. Els dos processos es reparteixen les 2 CPU's equitativament, més la càrrega de la resta de processos del sistema. Per tant, finalitzen al mateix temps. El temps d'execució dels dos processos és 55 segons cadascun.

La figura 3.11 mostra l'execució de tres processos privilegiats. Un procés acapararà l´ús

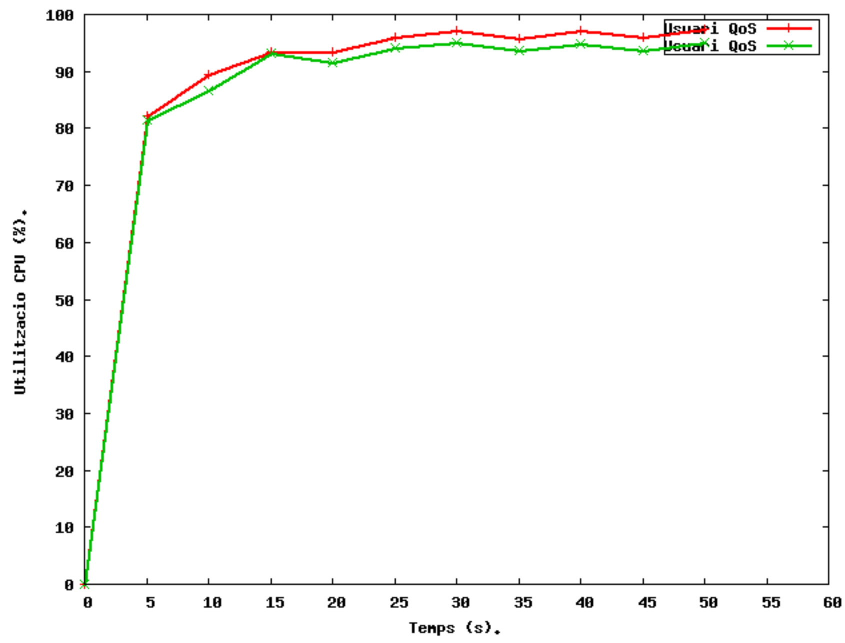

Figura 3.8: Prova CPU: Script QoS 1.

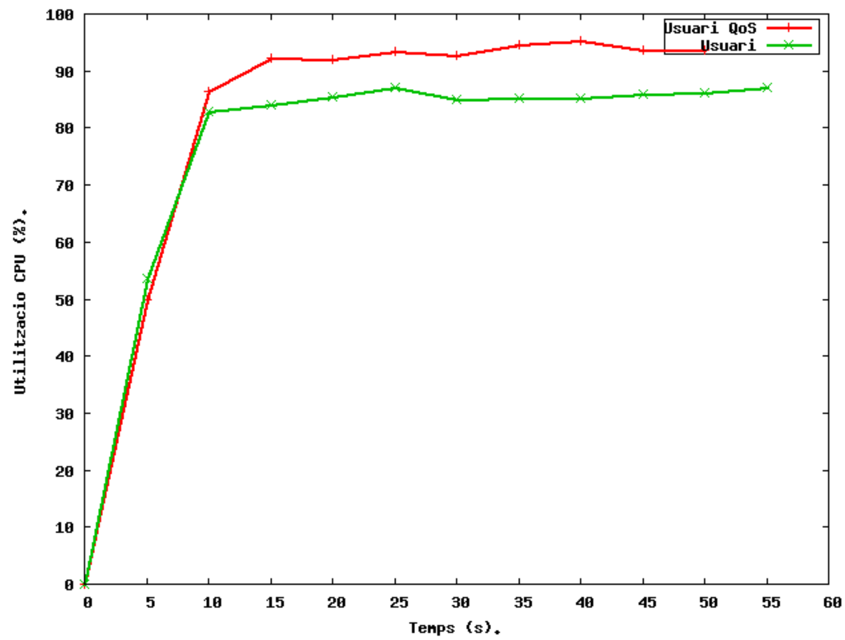

Figura 3.9: Prova CPU: Script QoS 2.

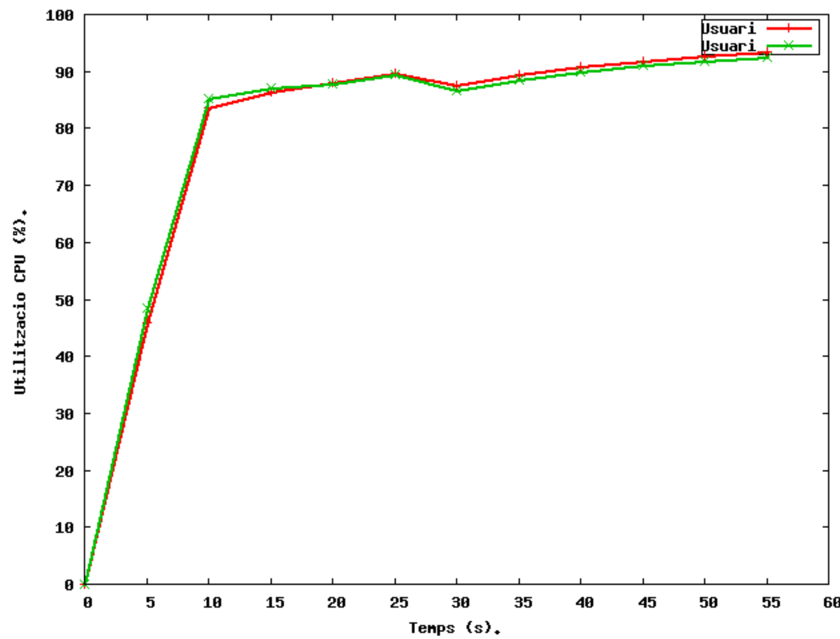

Figura 3.10: Prova CPU: Script QoS 3.

d'una CPU sota la limitació establerta del 90%, mentre que els altres dos processos hauran de repartir-se l'atra CPU equitativament. Un procés surt beneficiat, respecte els altres dos, perquè llancem un nombre senar de processos i per contra disposem de 2 CPU's. Un cop finalitzat el primer procés, una CPU queda lliure, per tant, cada procés privilegiat se situa en una CPU diferent. Els temps d'execució dels tres processos són 62, 80 i 83 segons respectivament.

La figura 3.12 mostra l'execució de tres processos, dos dels quals privilegiats i un de normal. Un procés privilegiat acapararà l´ús d'una CPU sota la limitació establerta del 90%, com la prova anterior, però a diferència de la prova anterior no es repartiran equitativament l'altra CPU, sinó que el procés privilegiat l'acaparà i el procés normal l'utilitzarà durant les penalitzacions. Un cop finalitza algun procés privilegiat, el procés normal augmenta significativament l'ús de la CPU. Els temps d'execució dels dos processos privilegiats són 61 segons cadascun i el temps del procés normal és 98 segons.

La figura 3.13 mostra l'execució de tres processos, un dels quals és privilegiat i la resta normals. El procés privilegiat acapararà una CPU i els altres dos processos es repartiran equitativament la CPU. Un cop finalitzi l'execució del procés privilegiat, un procés normal passarà a l'altra CPU, d'aquesta forma els dos processos incrementaran els seu rendiment. El temps d'execució del procés privilegiat és 53 segons i els temps dels dos

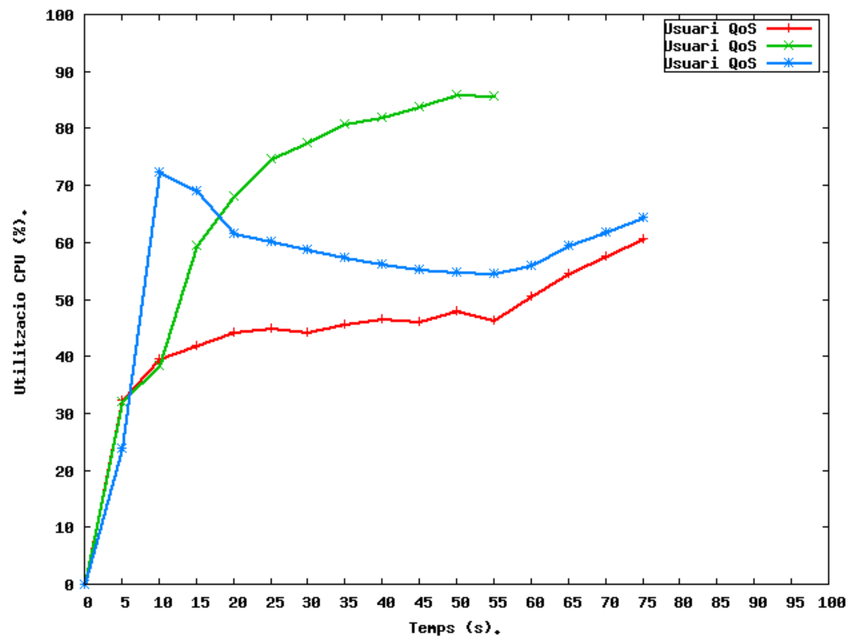

Figura 3.11: Prova CPU: Script QoS 6.

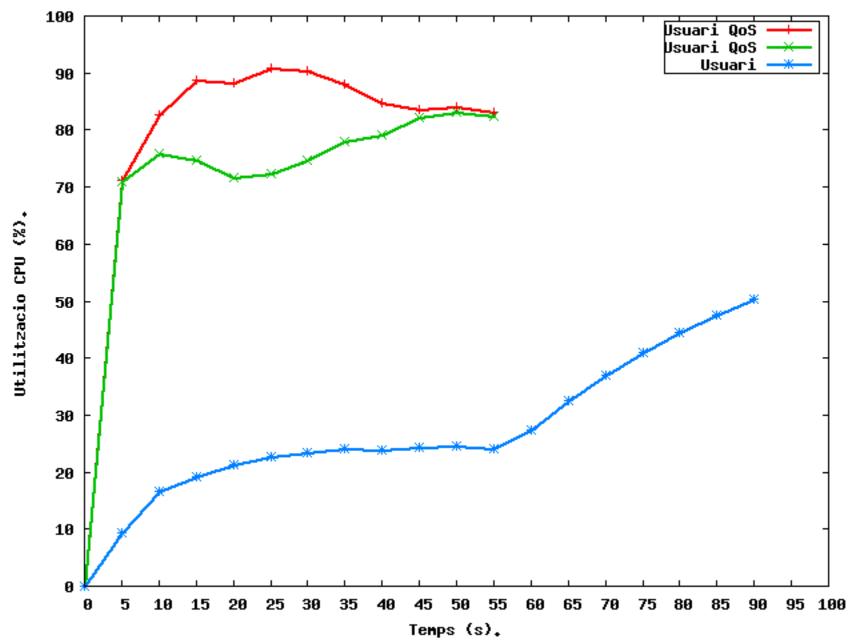

Figura 3.12: Prova CPU: Script QoS 7.

processos normals són 80 segons cadascun.

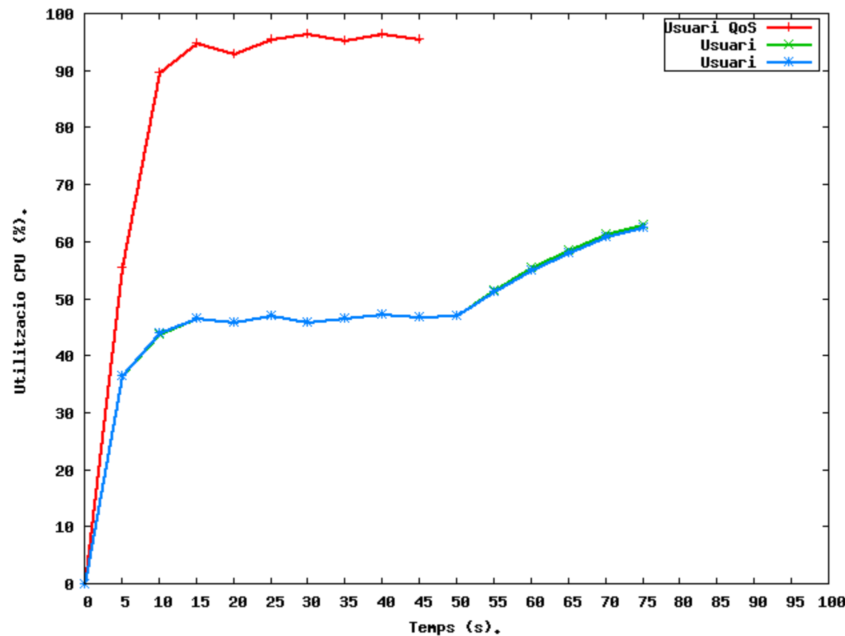

Figura 3.13: Prova CPU: Script QoS 8.

La figura 3.14 mostra l'execució de tres processos normals. Entre els tres processos hauran de turnar-se l'ús de les 2 CPU's donat que es tracta d'un nombre senar de processos, sempre haurà un procés que sortirà lleugerament beneficiat respecte als altres dos. Però aquest benefici es incomparable al proporciat als processos privilegiats. Els temps d'execució dels tres processos normals són 64, 80 i 85 segons respectivament.

La figura 3.15 mostra l'execució de quatre processos privilegiats. Per tant, al ser un nombre parell de processos es repartiran les 2 CPU's equitativament. Els temps d'execució dels quatre processos són de 101, 106, 108 i 108 segons respectivament.

La figura 3.16 mostra l'execució de quatre processos, tres dels quals són privilegiats i un de normal. La primera CPU contrindrà un procés privilegiat i el procés normal, el qual solament utilitzarà la CPU durant el període de penalització aplicat al procés privilegiat. Al finalitzar aquest procés privilegiat, el procés normal disposarà d'aquesta CPU, juntament a la resta de processos del sistema. Per altra banda, els altres dos processos es repartiran equitativament l'altra CPU, ja que disposen de la mateixa prioritat. El temps d'execució del primer procés privilegiat és 80 segons, els temps dels altres dos processos privilegiats són 98 segons cadascun i el temps del procés normal és 110 segons.

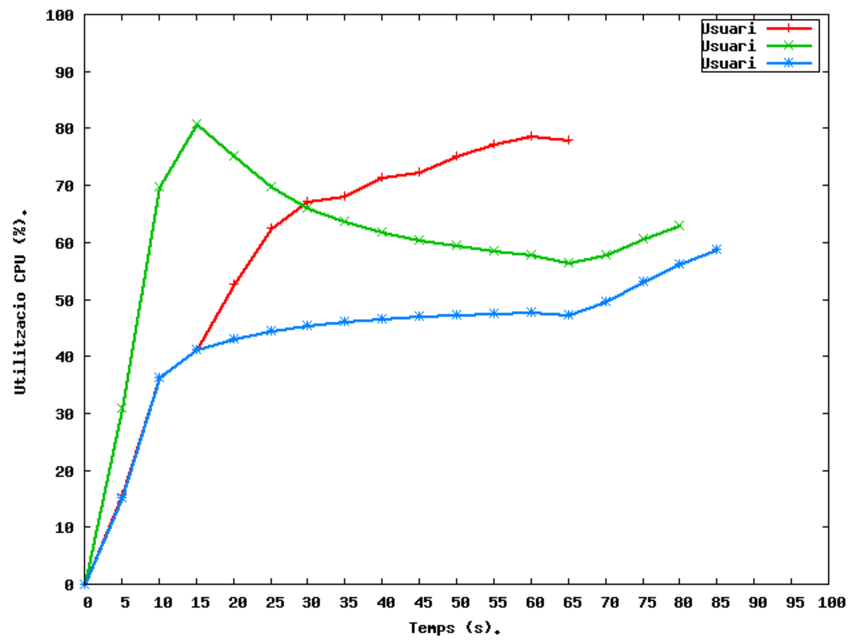

Figura 3.14: Prova CPU: Script QoS 9.

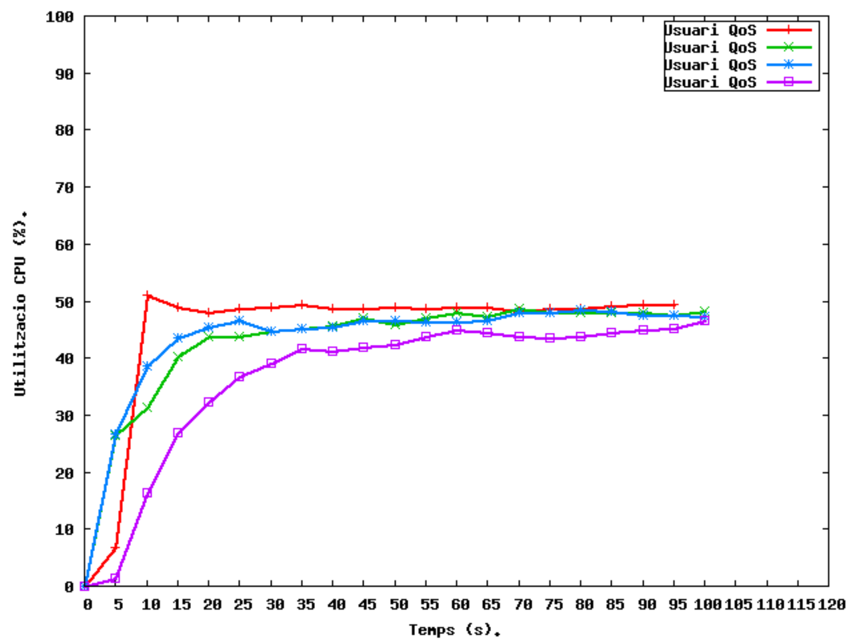

Figura 3.15: Prova CPU: Script QoS 10.

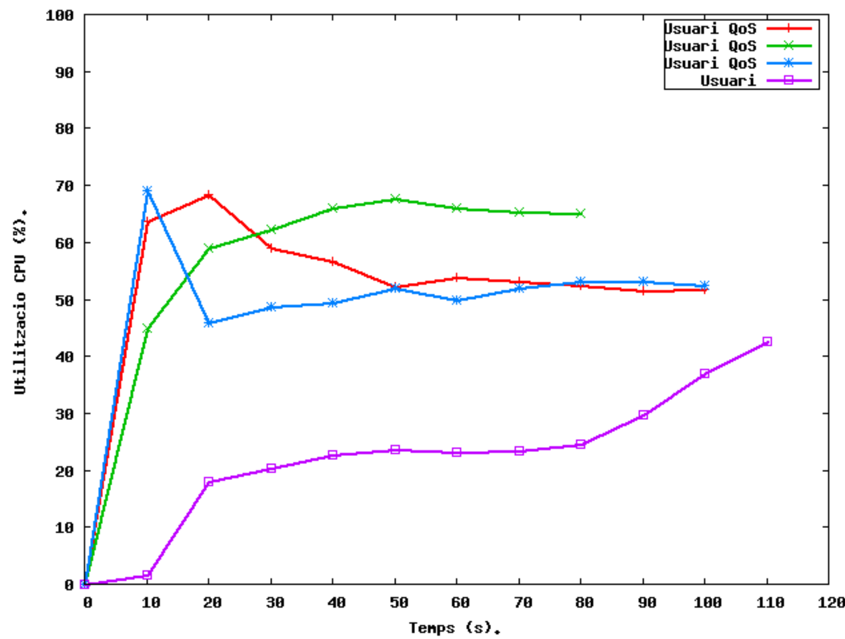

Figura 3.16: Prova CPU: Script QoS 11.

La figura 3.17 mostra l'execució de quatre processos, dos dels quals són privilegiats i els altres dos són normals. Cada CPU contindrà un procés privilegiat i un procés normal. Els processos normals principalment utilitzaran la CPU durant el període de penalització dels processos privilegiats. Un cop finalitza el procés privilegiat, l'altre procés normal incrementarà la utilització de la CPU. Els temps d'execució dels dos processos privilegiats són 70 segons cadascun i els temps dels dos processos normals són 102 i 103 segons respectivament.

La figura 3.18 mostra l'execució de quatre processos, un dels quals és procés privilegiat i els altres tres són normals. El procés privilegiat acaparà una CPU i el temps de pernalització serà utilitzat per un procés normal. L'altra CPU contindrà els altres dos processos, donat que tenen la mateixa prioritat, se la repatiran equitativament. El temps d'execució del procés privilegiat és 68 segons i els temps dels tres processos normals són 97, 104 i 107 segons respectivament.

La figura 3.19 mostra l'execució de quatre processos normals. Donat que disposem de 2 CPU's i un nombre parell de processos, cada CPU executarà dos processos sota les mateixes condicions. Els temps d'execució dels quatre processos normals són 106, 106, 107 i 108 segons respectivament.

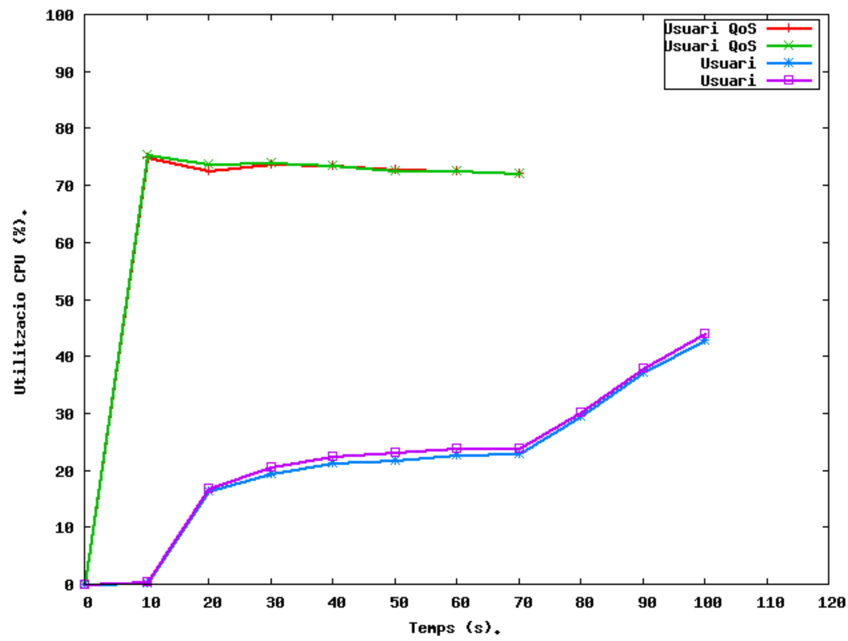

Figura 3.17: Prova CPU: Script QoS 12.

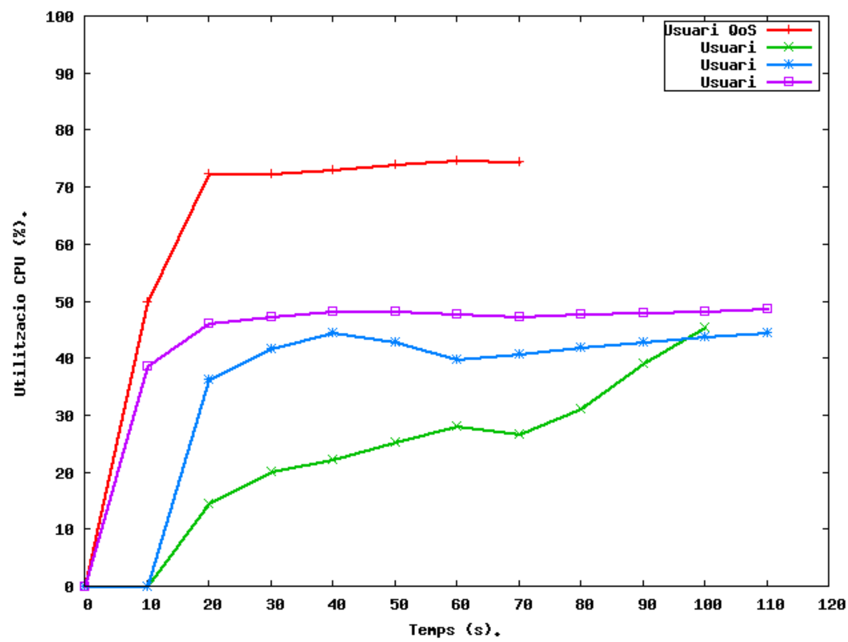

Figura 3.18: Prova CPU: Script QoS 13.

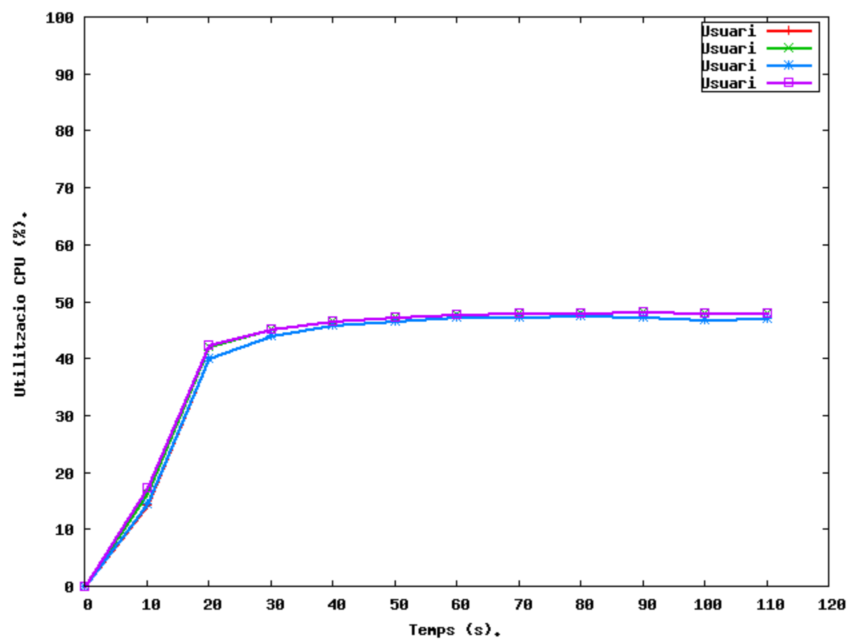

Figura 3.19: Prova CPU: Script QoS 14.

Capítol 4

Conclusions i treball futur

En aquest treball final de carrera s'ha realitzat el desenvolupament d'un sistema de QoS a nivell d'usuari sota els entorns de xarxa i CPU. Consisteix en garantir uns determinats amples de banda de xarxa, així com reservar porcions de CPU a través del nucli del SO Linux, per a determinades aplicacions d'usuaris privilegiats. D'aquesta manera obtenir una execució més eficient d'aquestes aplicacions.

Malgrat les limitacions que presenta la xarxa, com són la congestió i els colls d'ampolla, mitjançant la interfície de gestió de tràfic IP incorporada al nucli 2.6 podem solucionar, o suavitzar almenys, aquests problemes.

Per tal de satisfer els objectius plantejats s'ha utilitzat la funcionalitat dedicada a la reserva d'ample de banda. Aquesta funcionalitat està formada per les disciplines de cues que, a través de l'encuament dels paquets, determinen el mode com s'envien les dades. Amb els coneixements obtinguts i els mecanismes de gestió de l'ample de banda, s'ha desenvolupat un sistema efectiu de QoS a nivell d'usuari sota l'entorn de xarxa a través del **Script QoS**. Gràcies a aquesta eina es podran reservar en tot moment determinats amples de banda i destinar-los a connexions d'usuaris privilegiats. Per altra banda, si els usuaris privilegiats no requereixen del recurs serà repartit equitativament entre la resta de connexions.

Un cop s'ha realitzat una anàlisi minuciosa i detallada sobre el funcionament del planificador de processos del nucli 2.6, podem afirmar que aquest està orientat a l'execució d'aplicacions de temps compartit. Per tant, no disposa de cap mecanisme efectiu per al desenvolupament d'un sistema de QoS a nivell d'usuari sota l'entorn de CPU.

Per satisfer els objectius plantejats s'ha utilitzat el planificador de processos desenvolupat per **Con Kolivas**, el qual disposa d'un nou sistema d'assignació de prioritats i de noves polítiques d'execució. Amb els coneixements obtinguts i els mecanismes del nou

planificador de processos, s'ha desenvolupat un sistema efectiu de QoS a nivell d'usuari sota l'entorn de CPU a través del **Script QoS**. Gràcies a aquesta eina es podran gestionar automàticament els processos d'usuaris privilegiats, els quals obtindran el mínim temps d'execució possible en funció de la càrrega del sistema.

Per poder realitzar les proves al sistema de QoS instaurat, s'ha desenvolupat per a cadascun dels dos entorns una eina específica que permet avaluar els resultats obtinguts. D'aquesta manera poder comparar el sistema estàndard amb el nou sistema dotat de QoS a nivell d'usuari.

Un cop la QoS a nivell d'usuari està instaurada sota els entorns de xarxa i CPU, l'altre recurs finit més important d'un computador és la memòria principal. Per tant, com a treball futur, seria necessari el desenvolupament d'un sistema de QoS a nivell d'usuari sota l'entorn de memòria.

Podem entendre la QoS a nivell d'usuari sota l'entorn de memòria com la reserva de porcions de memòria principal, les quals seran utilitzades per a l'execució d'aplicacions d'usuaris privilegiats.

Bibliografia

[1] *http://www.itu.int/* International Telecommunication Union.

[2] *http://capstone.spsbe.jhu.edu/telecommunications/* ATM Lexicon.

[3] Vogel, A.; Kerhervé, B.; Bochman, G.V.; Gecsei, J. (1994). *Distributed multimedia applications and quality of service: a survey.*

[4] Corbet, J.; Rubimi, A.; Kroah-Hartman, G. (2005). *Linux Device Drivers.* 3rd Edition. O'Reilly.

[5] Herbert, T.F. (2004). *The Linux TCP/IP Stack.* Charles River Media.

[6] Horman, N. (2004). *Understanding And Programming With NetLink Sockets.*

[7] Hubert, B. (2004). *Enrutamiento avanzado y control de tráfico en Linux.*

[8] Buggenhaut, E.V. (2005). *Routing avanzado con el núcleo de Linux.*

[9] *http://dast.nlanr.net/Projects/Iperf/* Software Iperf.

[10] Love, R. (2004). *Linux kernel Development.* Sams Publishing.

[11] Bouet, D.P. (2002). *Understanding the Linux kernel.* 2nd Edition. O'Reilly.

[12] Peterson, J.L,; Silberschatz, A. (1989). *Sistemas Operativos: Conceptos fundamentales.* Reverté.

[13] Milenkovic, M. (1994). *Sistemas Operativos: Conceptos y diseño.* Mc Graw Hill.

[14] *http://members.optusnet.com.au/ckolivas/* Con Kolivas.

[15] *http://guide.debianizzati.org/* Aplicar Patch.

[16] *http://members.optusnet.com.au/ckolivas/interbench/* Benchmark Interbench.

[17] *http://ck.wikia.com/wiki/SchedulingPolicies* Polítiques d'Scheduling.

[18] *http://freequaos.host.sk/schedtool/* Software Scheedtool.

Apèndix A

Software

Aquesta secció conté una descripció detallada dels diferents elements software que han intervingut durant l'elaboració del TFC. Es descriuen breument les característiques principals del SO Linux, descripció i implantació del **Script QoS** i les eines desenvolupades per realitzar les proves.

A.1 Script QoS

L'eina **Script QoS** (Apèndix B) està constituïda per un conjunt tres scripts:

- **qos.sh**: és el script principal, i únicament s'encarrega d'activar/desactivar els scripts de xarxa i CPU. Els paràmetres són *start*, *stop* i *restart*.
- **qosNETWORK.sh**: és el script que s'encarrega de substituir la disciplina de cua **pfifo_fast** per l'estructura de cua que implanta la QoS en l'entorn de xarxa. Aquests mecanismes els proporciona el software **Iproute** detallat a continuació.
- **qosCPU.sh**: és el script que s'encarrega automàticament de buscar els processos d'usuaris privilegiats per canviar-los-hi la política d'execució. D'aquesta manera poder implantar un sistema de QoS en l'entorn de CPU. El correcte funcionament depèn exclusivament de l'activació del nou planificador de processos. Els passos per obtenir-lo els trobem a la subsecció **Patch Con Kolivas**.

A.1.1 Iproute

Iproute és una eina de software que proporciona al SO Linux un rendiment i característiques amb poca competència en el panorama general dels Sistemes Operatius. Aquest codi d'enrutament, filtratge i classificació, conté més possibilitats que els proporcionats per molts routers i tallafocs dedicats i productes de control de tràfic.

Iproute disposa de vàries aplicacions amb funcionalitat molt variada. Disposem de comandes com **ip**, la qual mostra o manipula l'enrutament o els dispositius de xarxa. També s'encarrega de les polítiques d'encaminament i túnels. Una altra comanda disponible és *arp*, la qual manipula la cache *arp* del sistema.

Per altra banda, podem trobar l'aplicació **tc** (*Traffic Control*), la qual ofereix la possibilitat de mostrar i manipular el control absolut del tràfic de xarxa. A l'interior del nucli és capaç de controlar el flux de transmissió, organitzar la prioritat del tràfic depenent de la naturalesa del mateix, establir polítiques d'encuament i eliminar paquets.

A.1.2 Patch Con Kolivas

Per tal d'obtenir el planificador de processos proposat per **Con Kolivas** [14] serà necessari aplicar el seu *patch* i tot seguit recompilar el nucli. Un cop recompilat el nucli, estarà preparat i optimitzat. Els passos són els següents:

1. tar -jxf /usr/src/linux-source-2.6.21.tar.bz2
2. cd /usr/src/linux-source-2.6.21
3. bzcat ../patch-2.6.21-ck2.bz2 | patch -p1
4. cp /boot/config-2.6.21-2-686 /usr/src/linux-source-2.6.21/.config
5. make oldconfig
6. fakeroot make-kpkg –initrd -append-to-version .ck2 kernel_image
7. dpkg -i linux-image-2.6.21.ck2-10.00.Custom_i386.deb

Un cop tenim el nucli recompilat, la comanda **schedtool** gestionada pel **Script QoS** s'encarregarà de buscar els processos dels usuaris privilegiats i canviar-los-hi la política de planificació, amb la finalitat d'obtenir un temps de còmput final inferior.

A.1.3 Schedtool

Schedtool és una eina de software que proporciona al SO Linux els mecanismes suficients per canviar en temps real la política d'execució i la prioritat dels processos. Si s'utilitza sota un nucli estàndard solament podrà utilitzar les polítiques d'execució **SCHED_FIFO**, **SCHED_RR** i **SCHED_NORMAL**. Per altra banda, si s'utilitza conjuntament amb el nou planificador de processos implantat prèviament, llavors estarà capacitat per utilitzar les polítiques d'execució **SCHED_FIFO**, **SCHED_RR**, **SCHED_NORMAL**, **SCHED_BATCH**, **SCHED_ISO** i **SCHED_IDLEPRIO**.

En funció de la política d'execució seleccionada, la comanda **schedtool** necessitarà l'aportació d'algun paràmetre auxiliar. Per exemple, si durant l'execució d'un procés s'elegeix la política de temps real **SCHED_FIFO**, aleshores s'haurà de proporcionar al procés una prioritat estàtica. En aquest cas, al seleccionar una política de temps real, la prioritat haurà d'estar situada al rang [1-99].

A.2 Eina provesNETWORK.sh

L'eina **provesNETWORK.sh** (Apèndix C.1) disposa d'onze proves, les quals poden seleccionar-se a través d'un únic paràmetre d'entrada. Cada prova s'encarrega de crear un nombre de connexions sota la identitat d'usuaris privilegiats i/o normals. Aquestes connexions són generades a través del software **Iperf**, que s'explica a continuació. Mentre les connexions estiguin actives en segon pla, el script s'encarrega de recollir les dades necessàries per generar un gràfic on poder plasmar els resultats obtinguts. Cada gràfic mostra el seguiment, al llarg del temps, de l'ample de banda utilitzat per cadascuna de les connexions analitzades.

El funcionament d'aquesta eina de proves és **./provesNETWORK.sh** <*PROVA*>.

A.2.1 Iperf

El SO Linux disposa d'una gran varietat de generadors de tràfic, cadascun dels quals disposa d'unes característiques pròpies que els diferencien de la resta. Com s'ha esmentat anteriorment, l'eina **provesNETWORK.sh** necessita realitzar proves de tràfic massiu, llavors el generador de tràfic elegit és **Iperf**, les principals característiques del qual són:

- *Arquitectura client/servidor*: les proves es realitzen segons aquest principi. En un ordinador s'inicia el servidor i des dels altres els diferents clients que es connectaran al servidor i enviaran paquets de dades.

- *Connexions paral·leles*: possibilitat de generar connexions paral·leles, enviant un gran flux de dades cap a la mateixa direcció i port.

- *Ample de banda màxim*: permet especificar la mida màxima del segment, la qual és la MTU de 40bytes. També permet especificar la mida de la finestra TCP.

La utilització d'aquesta eina de proves permetrà demostrar que l'estructura de cua implantada a través **Qdisc QoS** compleix amb els objectius marcats i com la disciplina estàndard del SO Linux (**pfifo_fast**) no els assoleix.

A.3 Benchmark InterBench

InterBench [16] és el benchmark proporcionat per **Con Kolivas** per provar l'eficàcia de la política d'execució **SCHED_ISO** utilitzada al **Script QoS**.

Està dissenyat per realitzar el benchmark d'interactivitat sota el SO Linux. Podem entendre la interactivitat com la latència durant l'execució dels processos, la qual és creada pel planificador de processos a causa de la càrrega del sistema.

Els processos en tot moment poden executar-se, malgrat que el sistema disposi d'una càrrega molt elevada. Però la seva interactivitat es veurà clarament afectada en funció de la càrrega del moment.

Aquest benchmark s'encarrega de simular el funcionament del planificador de processos i mesurar la latència i el jitter produïts. Els càlculs els realitza a partir d'un seguit de tasques habituals, com són:

- *Àudio*: a partir d'un fil d'execució se simula l'intent d'execució a intervals de 50ms, la qual cosa requereix el 5% de CPU.

- *Vídeo*: a partir d'un fil d'execució se simula l'intent de rebre la CPU 60 cops cada segons i utilitza el 40% de CPU. Necessita 60 cops per segon la CPU ja que demana el vídeo a 60 frames/s.

- *Finestres*: a partir d'un fil d'execució se simula la creació d'una finestra arrossegada a través de la pantalla. La utilització de la CPU varia del 0% al 100%.

- *Massiu*: a partir d'un fil d'execució que simula l'execució d'un joc que s'encarrega de sol·licitar més CPU que l'obtenible.

Inicialment, el benchmark s'encarrega de calcular la millor forma de calcular els percentatges d'utilització de CPU. Introdueix aquesta informació en un fitxer per poder mantenir la posterior execució sota un ús constant de CPU.

El benchmark executa un fil temporal, el qual s'encarrega de despertar el fil d'execució de la simulació i calcular la latència. Donat que no existeix la planificació del temps sota el SO Linux, el fil temporal dorm tan acuradament com el nucli suporta. La latència és considera com el temps des que es posa a dormir el fil fins que és executat de nou.

A.4 Eina provesCPU.sh

L'eina **provesCPU.sh** (Apèndix C.2) disposa de catorze proves, seleccionables a partir de tres paràmetres d'entrada. El primer paràmetre indica la quantitat total de processos (quatre processos màxim), el segon i el tercer la quantitat d'aquests que són privilegiats i normals respectivament. Amb aquests paràmetres es crida el programa **provesCPU.c** (Apèndix C.2) que s'encarrega de crear i gestionar la quantitat de processos privilegiats i/o normals. Tots els processos fills, independentment del propietari del procés, executen la mateixa funció d'ús massiu de la CPU. Mentre les funcions d'ús massiu s'executen en segon pla, l'eina s'encarrega de recollir les dades necessàries per generar un gràfic on plasmar els resultats. Cada gràfic mostra el seguiment al llarg del temps dels processos implicats en referència al percentatge d'ús de la CPU.

El funcionament d'aquesta eina de proves és **./provesCPU.sh** *<PROCESSOS>* *<QoS>* *<NORMALS>* on PROCESSOS=QoS+NORMALS.

La utilització d'aquesta eina de proves permetrà demostrar com amb el nou planificador de processos, més el **Qdisc QoS**, obtindrem millors resultats que amb el planificador de processos estàndard en termes de QoS.

Apèndix B

Script QoS

En aquest apèndix s'exposa el codi font del **Script QoS** per implantar la QoS sota els entorns de xarxa i CPU desenvolupada durant l'elaboració d'aquest TFC. L'apèndix A.1 explica detalladament l'estructura interna del **Script QoS**. Aquest script s'estructura sota un conjunt de tres scripts específics, els quals són el script principal **qos.sh** (B.1), i els scripts **qosNETWORK.sh** (B.2) i **qosNETWORK.sh** (B.3) que s'ocupen de la QoS sota els entorns de xarxa i CPU respectivament.

Listing B.1: qos.sh

```
#!/bin/bash

# Interfície on netejar la Qdisc configurada.
DEV=eth1

# Camí absolut dels binaris.
TC=/sbin/tc
IPTABLES=/sbin/iptables

# Direcció dels scripts de xarxa i CPU respectivament.
QOS_NETWORK=/<direccio>/qosNETWORK.sh
QOS_CPU=/<direccio>/qosCPU.sh

# Funció de desactivació del script qosNETWORK.sh.
network_off() {
$TC qdisc del dev $DEV root    2> /dev/null > /dev/null
$TC qdisc del dev $DEV ingress 2> /dev/null > /dev/null
```

```
$IPTABLES -t mangle -F OUTPUT
$IPTABLES -t mangle -F qos_usuari_local01
$IPTABLES -t mangle -F qos_usuari_local02
$IPTABLES -t mangle -F qos_usuari_local03
$IPTABLES -t mangle -F qos_usuari_local04
$IPTABLES -t mangle -F qos_usuari_internet01
$IPTABLES -t mangle -F qos_usuari_internet02
$IPTABLES -t mangle -F qos_usuari_internet03
$IPTABLES -t mangle -F qos_usuari_internet04
$IPTABLES -t mangle -X
}

# Funció de desactivació del script qosCPU.sh.
cpu_off(){
killall qosCPU.sh
}

case "$1" in
   start)
      echo -n "QoS NETWORK ... "
      $QOS_NETWORK
      echo "ON."

echo -n "QoS CPU ... "
      $QOS_CPU &
      echo "ON."
      ;;
   stop)
      echo -n "QoS NETWORK ... "
      network_off
      echo "OFF."

  echo -n "QoS CPU ... "
      cpu_off
      echo "OFF."
      ;;
   restart)
      $0 stop
      $0 start
      ;;
```

```
   *)
      echo "Funcionament: qos {start|stop|restart}" >&2
      exit 1
      ;;
esac
exit 0
```

Listing B.2: qosNETWORK.sh

```
#!/bin/bash

# Interficie de xarxa on s'aplicarà la Qdisc.
DEV=eth1

# Xarxa local.
NETWORK=192.168.1.0/24

# Camí absolut dels binaris.
TC=/sbin/tc
IPTABLES=/sbin/iptables

# Usuaris amb privilegiats.
USER01=alfred01
USER02=alfred02
USER03=alfred03
USER04=alfred04

# Ample de banda de pujada Internet (en MBit).
UPBW=128

# Ample de banda xarxa local (en MBit)
ETHERBW=100

# Percentatge d'ample de banda pels RATES.
RATEQOS=70
RATENORMAL=30

# Percentatge d'ample de banda pels CEILS.
CEILQOS=100
CEILNORMAL=100
```

```
# Càlcul dels RATES Internet
RATEINTERNETQOS=$((($RATEQOS*$UPBW)/100))
RATEINTERNETNORMAL=$((($RATENORMAL*$UPBW)/100))

# Càlcul dels CEILS Internet
CEILINTERNETQOS=$((($CEILQOS*$UPBW)/100))
CEILINTERNETNORMAL=$((($CEILNORMAL*$UPBW)/100))

# Càlcul dels RATES Local
RATELOCALQOS=$((($RATEQOS*$ETHERBW)/100))
RATELOCALNORMAL=$((($RATENORMAL*$ETHERBW)/100))

# Càlcul dels CEILS Local
CEILLOCALQOS=$((($CEILQOS*$ETHERBW)/100))
CEILLOCALNORMAL=$((($CEILNORMAL*$ETHERBW)/100))

# Ample de banda màxim
TOTALBW=$ETHERBW

# Creem la Qdisc de tipus HTB, amb l'arrel a la classe 1:0 i enviant
# el tràfic sense classificar a la classe 220.
$TC qdisc add dev $DEV root handle 1: htb r2q 1 default 220

# L'arrel disposa de la classe filla 1:1, per tal d'establir l'ample de
# banda màxim de l'arrel.
$TC class add dev $DEV parent 1: classid 1:1 htb rate ${TOTALBW}Mbit burst 25k

# La classe 1:1 té dos classes filles; la classe filla 1:10 s'utilitza pel
#tràfic de xarxa local, mentre que la 1:20 s'utilitza pel tràfic d'Internet.
$TC class add dev $DEV parent 1:1 classid 1:10 htb rate ${ETHERBW}Mbit burst 25k
$TC class add dev $DEV parent 1:1 classid 1:20 htb rate ${UPBW}kbit burst 10k

# La classe 1:10 té les classes 1:110 i 1:120, que són utilitzades utilitzades
# pels usuaris privilegiats amb QoS i la resta d'usuaris, respectivament.
$TC class add dev $DEV parent 1:10 classid 1:110 htb
rate ${RATELOCALQOS}Mbit ceil ${CEILLOCALQOS}Mbit burst 25k prio 0

$TC class add dev $DEV parent 1:10 classid 1:120 htb
rate ${RATELOCALNORMAL}Mbit ceil ${CEILLOCALNORMAL}Mbit burst 25k prio 0
```

```
# La classe 1:20 té les classes 1:210 i 1:220, que són utilitzades pels
# usuaris privilegiats amb QoS i la resta d'usuaris, respectivament.
$TC class add dev $DEV parent 1:20 classid 1:210 htb
rate ${RATEINTERNETQOS}kbit ceil ${CEILINTERNETQOS}kbit burst 10k prio 0

$TC class add dev $DEV parent 1:20 classid 1:220 htb
rate ${RATEINTERNETNORMAL}kbit ceil ${CEILINTERNETNORMAL}kbit burst 10k prio 0

# Les classes terminals de l'arbre, que són 1:110, 1:120, 1:210 i 1:220, tenen
# totes associades la Qdisc SFQ, llavors les noves classes terminals de l'arbre
# són 110:, 120:, 210: i 220 respectivament.
$TC qdisc add dev $DEV parent 1:110 handle 110: sfq
$TC qdisc add dev $DEV parent 1:120 handle 120: sfq

$TC qdisc add dev $DEV parent 1:210 handle 210: sfq
$TC qdisc add dev $DEV parent 1:220 handle 220: sfq

# Envia a la classe terminal 120: els paquets locals de la resta d'usuaris.
$TC filter add dev $DEV parent 1:0 protocol ip prio 2 u32 match ip dst $NETWORK
flowid 1:120

# Regles que marquen els paquets de xarxa local dels usuaris privilegiats.
$IPTABLES -t mangle -N qos_usuari_local01
$IPTABLES -t mangle -A OUTPUT -m owner --uid-owner $USER01
--destination $NETWORK -j qos_usuari_local01
$IPTABLES -t mangle -A qos_usuari_local01 -j MARK --set-mark 110

$IPTABLES -t mangle -N qos_usuari_local02
$IPTABLES -t mangle -A OUTPUT -m owner --uid-owner $USER02
--destination $NETWORK -j qos_usuari_local02
$IPTABLES -t mangle -A qos_usuari_local02 -j MARK --set-mark 110

$IPTABLES -t mangle -N qos_usuari_local03
$IPTABLES -t mangle -A OUTPUT -m owner --uid-owner $USER03
--destination $NETWORK -j qos_usuari_local03
$IPTABLES -t mangle -A qos_usuari_local03 -j MARK --set-mark 110

$IPTABLES -t mangle -N qos_usuari_local04
$IPTABLES -t mangle -A OUTPUT -m owner --uid-owner $USER04
--destination $NETWORK -j qos_usuari_local04
$IPTABLES -t mangle -A qos_usuari_local04 -j MARK --set-mark 110
```

```
# Envia a la classe terminal 110: els paquets marcats amb el valor 110.
$TC filter add dev $DEV parent 1:0 protocol ip prio 1 handle 110 fw flowid 1:110

# Regles que marquen els paquets de xarxa local dels usuaris privilegiats.
$IPTABLES -t mangle -N qos_usuari_internet01
$IPTABLES -t mangle -A OUTPUT -m owner --uid-owner $USER01
-j qos_usuari_internet01
$IPTABLES -t mangle -A qos_usuari_internet01 -j MARK --set-mark 210

$IPTABLES -t mangle -N qos_usuari_internet02
$IPTABLES -t mangle -A OUTPUT -m owner --uid-owner $USER02
-j qos_usuari_internet02
$IPTABLES -t mangle -A qos_usuari_internet02 -j MARK --set-mark 210

$IPTABLES -t mangle -N qos_usuari_internet03
$IPTABLES -t mangle -A OUTPUT -m owner --uid-owner $USER03
-j qos_usuari_internet03
$IPTABLES -t mangle -A qos_usuari_internet03 -j MARK --set-mark 210

$IPTABLES -t mangle -N qos_usuari_internet04
$IPTABLES -t mangle -A OUTPUT -m owner --uid-owner $USER04
-j qos_usuari_internet04
$IPTABLES -t mangle -A qos_usuari_internet04 -j MARK --set-mark 210

# Envia a la classe terminal 210: els paquets marcats amb el valor 210.
$TC filter add dev $DEV parent 1:0 protocol ip prio 3 handle 210 fw flowid 1:210
```

Listing B.3: qosCPU.sh

```
#!/bin/bash

# Eina que s'encarrega de canviar la política d'execució
# de SCHED_NORMAL a SCHED_ISO als processos privilegiats.
SCHEDTOOL=/usr/bin/schedtool

# Percentatge d'ús de CPU que no poden superar els processos
# sota la política SCHED_ISO. Si és superat es procedeix a la
# penalització. Valors entre 70-90 són els habituals.
CPU=90
```

```
# Periode de penalització per aquells processos que superin
# el percentatge d'ús indicat anteriorment. La penalització
# consisteix en la modificació de la prioritat, aleshores
# es produeix la pèrdua de la CPU. Valors entre 3-5 són els
# habituals.
PERIOD=5

# Modifica els valors del nucli amb els paràmentres anteriors.
echo $CPU > /proc/sys/kernel/iso_cpu
echo $PERIOD > /proc/sys/kernel/iso_period

while true; do

# Busca aquells processos d'usuaris privilegiats que encara no han
# rebut el canvi de la política SCHED_NORMAL a SCHED_ISO. A l'interior
# d'aquesta instrucció poden afegir i eliminar usuaris privilegiats.
pids=`ps -wweALo user,pid,priority | awk '(/alfred01/ || /alfred02/ ||
/alfred03/ || /alfred04/) && !/-1/ {print $2}'`

# S'encarrega de filtrar aquells processos que s'han activat anteriorment,
# però ara estan en perïode de penalització.
for i in $pids
do
for j in $fets
do
if [ $i -eq $j ]; then
trobat="TRUE"
fi
done
if [ -z $trobat ]; then
falten="$falten $i"
fi
trobat=""
done

# Canvia la política d'execució de SCHED_NORMAL a SCHED_ISO
# dels processos seleccionats anteriorment.
for i in $falten
do
$SCHEDTOOL -I $i;
```

```
fets="$fets $i"
falten=""
done

sleep 1;
done

exit 0
```

Apèndix C

Scripts de proves

En aquest apèndix s'exposa el codi font de les eines que s'ha implementat per poder realitzar les proves desitjades sobre la QoS en l'entorn de xarxa i CPU, ja que no s'han trobat eines específiques amb aquesta finalitat.

El primer apartat descriu el codi font del script **provesNETWORK.sh**, eina utilitzada per realitzar les proves de xarxa. Per altra banda, el segon apartat descriu el codi font del script **provesCPU.sh** i del programa **provesCPU.c**, els quals són combinats per realitzar les proves de CPU.

C.1 L'entorn de xarxa

En aquesta secció s'exposa tot el codi font del script **provesNETWORK.sh**, representat a C.1 i implementat amb llenguatge *shell*. A grans trets, el script en funció de la prova elegida s'encarregarà de crear les connexions de tràfic massiu sota una identitat determinada. Aquestes connexions de tràfic massiu les realitzà el software client-servidor **Iperf**. Mentre les connexions estiguin actives en segon pla, el script s'encarrega de recollir les dades necessàries per generar un gràfic on poder plasmar els resultats obtinguts. Cada gràfic mostra el seguiment al llarg del temps de l'ample de banda utilitzat per cadascuna de les connexions analitzades.

Listing C.1: provesNETWORK.sh

```
#!/bin/bash

IPERF=/<direccio>/iperf
GNUPLOT=/usr/bin/gnuplot

GRAFIC_T1=/<direccio>/gp1.sh
GRAFIC_T2=/<direccio>/gp2.sh
GRAFIC_T3=/<direccio>/gp3.sh
GRAFIC_T4=/<direccio>/gp4.sh
```

```
10 GRAFIC_T5=/<direccio>/gp5.sh
   GRAFIC_T6=/<direccio>/gp6.sh

   SERVIDOR=atlantis
   PORT=8080

   QOS1=qos1
   QOS2=qos2
   NORMAL1=normal1
   NORMAL2=normal2
20
   case "$1" in
   servidor)
   echo -n "Activant servidor ... "
   ssh $SERVIDOR $IPERF -s -p $PORT &
   echo "OK."
   ;;

   1)
   echo "PROVA 1:";
30 echo "Usuaris normals: 1."
   echo "Usuaris QoS: 0."
   echo -n "Calculant ... "
   su $NORMAL1 -c "$IPERF -c $SERVIDOR -p $PORT -i 5 -t 50 > /tmp/tmp01.aux &";
   sleep 52;
   echo "OK."

   sed -n '7,16p' /tmp/tmp01.aux | sed -e 's/- /-/' -e 's/-/ /' -e 's/  / /' \
   | awk '/sec/ {print $4" "$8}' | sed -e '1i\0.0 0.0' -e '$a\55.0 0.0' > /tmp/p01.out

40 echo -n "Generant grafica ..."
   $GNUPLOT $GRAFIC_T1
   echo "OK."
   rm /tmp/tmp01.aux /tmp/p01.out
   ;;

   2)
   echo "PROVA 2:";
   echo "Usuaris normals: 2."
   echo "Usuaris QoS: 0."
50 echo -n "Calculant ... "
   su $NORMAL1 -c "$IPERF -c $SERVIDOR -p $PORT -i 5 -t 50 > /tmp/tmp01.aux &";
   su $NORMAL2 -c "$IPERF -c $SERVIDOR -p $PORT -i 5 -t 50 > /tmp/tmp02.aux &";
   sleep 52;
   echo "OK."

   sed -n '7,16p' /tmp/tmp01.aux | sed -e 's/- /-/' -e 's/-/ /' -e 's/  / /' \
   | awk '/sec/ {print $4" "$8}'| sed -e '1i\0.0 0.0' -e '$a\55.0 0.0' > /tmp/p01.out

   sed -n '7,16p' /tmp/tmp02.aux | sed -e 's/- /-/' -e 's/-/ /' -e 's/  / /' \
60 | awk '/sec/ {print $4" "$8}' | sed -e '1i\0.0 0.0' -e '$a\55.0 0.0' > /tmp/p02.out

   echo -n "Generant grafica ..."
   $GNUPLOT $GRAFIC_T2
   echo "OK."
   rm /tmp/tmp01.aux /tmp/tmp02.aux /tmp/p01.out /tmp/p02.out
   ;;

   3)
   echo "PROVA 3:";
70 echo "Usuaris normals: 1 + 1 (15s retard)."
   echo "Usuaris QoS: 0."
   echo -n "Calculant ... "
   su $NORMAL1 -c "$IPERF -c $SERVIDOR -p $PORT -i 5 -t 50 > /tmp/tmp01.aux &";
   su $NORMAL2 -c "sleep 15 && $IPERF -c $SERVIDOR -p $PORT -i 5 -t 35 > \
   /tmp/tmp02.aux &";
   sleep 52;
   echo "OK."

   sed -n '7,16p' /tmp/tmp01.aux | sed -e 's/- /-/' -e 's/-/ /' -e 's/  / /' \
80 | awk '/sec/ {print $4" "$8}' | sed -e '1i\0.0 0.0' -e '$a\55.0 0.0' > /tmp/p01.out

   sed -n '7,13p' /tmp/tmp02.aux | sed -e 's/- /-/' -e 's/-/ /' -e 's/  / /' \
   | awk '/sec/ {print $4" "$8}' | sed -e 's/35.0/50.0/' -e 's/30.0/45.0/' \
   -e 's/25.0/40.0/' -e 's/20.0/35.0/' -e 's/15.0/30.0/' -e 's/10.0/25.0/' \
   -e 's/^5.0/20.0/' -e '1i\0.0 0.0' -e '1i\5.0 0.0' -e '1i\10.0 0.0' \
   -e '1i\15.0 0.0' -e '$a\55.0 0.0' > /tmp/p02.out

   echo -n "Generant grafica ..."
   $GNUPLOT $GRAFIC_T2
90 echo "OK."
   rm /tmp/tmp01.aux /tmp/tmp02.aux /tmp/p01.out /tmp/p02.out
   ;;
```

```
    4)
    echo "PROVA 4:";
    echo "Usuaris normals: 0."
    echo "Usuaris QoS: 1."
    echo -n "Calculant ... "
    su $QOS1 -c "$IPERF -c $SERVIDOR -p $PORT -i 5 -t 50 > /tmp/tmp01.aux &";
100 sleep 52;
    echo "OK."

    sed -n '7,16p' /tmp/tmp01.aux | sed -e 's/- /-/' -e 's/-/ /' -e 's/  / /' \
    | awk '/sec/ {print $4" "$8}' | sed -e '1i\0.0 0.0' -e '$a\55.0 0.0' > /tmp/p01.out

    echo -n "Generant grafica ..."
    $GNUPLOT $GRAFIC_T3
    echo "OK."
    rm /tmp/tmp01.aux /tmp/p01.out
110 ;;

    5)
    echo "PROVA 5:";
    echo "Usuaris normals: 0."
    echo "Usuaris QoS: 2."
    echo -n "Calculant ... "
    su $QOS1 -c "$IPERF -c $SERVIDOR -p $PORT -i 5 -t 50 > /tmp/tmp01.aux &";
    su $QOS2 -c "$IPERF -c $SERVIDOR -p $PORT -i 5 -t 50 > /tmp/tmp02.aux &";
    sleep 52;
120 echo "OK."

    sed -n '7,16p' /tmp/tmp01.aux | sed -e 's/- /-/' -e 's/-/ /' -e 's/  / /' \
    | awk '/sec/ {print $4" "$8}' | sed -e '1i\0.0 0.0' -e '$a\55.0 0.0' > /tmp/p01.out

    sed -n '7,16p' /tmp/tmp02.aux | sed -e 's/- /-/' -e 's/-/ /' -e 's/  / /' \
    | awk '/sec/ {print $4" "$8}' | sed -e '1i\0.0 0.0' -e '$a\55.0 0.0' > /tmp/p02.out

    echo -n "Generant grafica ..."
    $GNUPLOT $GRAFIC_T4
130 echo "OK."
    rm /tmp/tmp01.aux /tmp/tmp02.aux /tmp/p01.out /tmp/p02.out
    ;;

    6)
    echo "PROVA 6:";
    echo "Usuaris normals: 0."
    echo "Usuaris QoS: 1 + 1 (15s retard)."
    echo -n "Calculant ... "
    su $QOS1 -c "$IPERF -c $SERVIDOR -p $PORT -i 5 -t 50 > /tmp/tmp01.aux &";
140 su $QOS2 -c "sleep 15 && $IPERF -c $SERVIDOR -p $PORT -i 5 -t 35 > /tmp/tmp02.aux &";
    sleep 52;
    echo "OK."

    sed -n '7,16p' /tmp/tmp01.aux | sed -e 's/- /-/' -e 's/-/ /' -e 's/  / /' \
    | awk '/sec/ {print $4" "$8}' | sed -e '1i\0.0 0.0' -e '$a\55.0 0.0' > /tmp/p01.out

    sed -n '7,13p' /tmp/tmp02.aux | sed -e 's/- /-/' -e 's/-/ /' -e 's/  / /' \
    | awk '/sec/ {print $4" "$8}' | sed -e 's/35.0/50.0/' -e 's/30.0/45.0/' \
    -e 's/25.0/40.0/' -e 's/20.0/35.0/' -e 's/15.0/30.0/' -e 's/10.0/25.0/' \
150 -e 's/^5.0/20.0/' -e '1i\0.0 0.0' -e '1i\5.0 0.0' -e '1i\10.0 0.0' \
    -e '1i\15.0 0.0' -e '$a\55.0 0.0' > /tmp/p02.out

    echo -n "Generant grafica ..."
    $GNUPLOT $GRAFIC_T4
    echo "OK."
    rm /tmp/tmp01.aux /tmp/tmp02.aux /tmp/p01.out /tmp/p02.out
    ;;

    7)
160 echo "PROVA 7:";
    echo "Usuaris normals: 1."
    echo "Usuaris QoS: 1."
    echo -n "Calculant ... "
    su $QOS1 -c "$IPERF -c $SERVIDOR -p $PORT -i 5 -t 50 > /tmp/tmp01.aux &";
    su $NORMAL2 -c "$IPERF -c $SERVIDOR -p $PORT -i 5 -t 50 > /tmp/tmp02.aux &";
    sleep 52;
    echo "OK."

    sed -n '7,16p' /tmp/tmp01.aux | sed -e 's/- /-/' -e 's/-/ /' -e 's/  / /' \
170 | awk '/sec/ {print $4" "$8}' | sed -e '1i\0.0 0.0' -e '$a\55.0 0.0' > /tmp/p01.out

    sed -n '7,16p' /tmp/tmp02.aux | sed -e 's/- /-/' -e 's/-/ /' -e 's/  / /' \
    | awk '/sec/ {print $4" "$8}' | sed -e '1i\0.0 0.0' -e '$a\55.0 0.0' > /tmp/p02.out

    echo -n "Generant grafica ..."
    $GNUPLOT $GRAFIC_T5
    echo "OK."
```

```
    rm /tmp/tmp01.aux /tmp/tmp02.aux /tmp/p01.out /tmp/p02.out
    ;;
180
    8)
    echo "PROVA 8:";
    echo "Usuaris normals: 1 (15s retard)."
    echo "Usuaris QoS: 1."
    echo -n "Calculant ... "
    su $QOS1 -c "$IPERF -c $SERVIDOR -p $PORT -i 5 -t 50 > /tmp/tmp01.aux &";
    su $NORMAL1 -c "sleep 15 && $IPERF -c $SERVIDOR -p $PORT -i 5 -t 35 > /tmp/tmp02.aux &";
    sleep 52;
    echo "OK."
190
    sed -n '7,16p' /tmp/tmp01.aux | sed -e 's/- /-/' -e 's/-/ /' -e 's/  / /' \
    | awk '/sec/ {print $4" "$8}' | sed -e '1i\0.0 0.0' -e '$a\55.0 0.0' > /tmp/p01.out

    sed -n '7,13p' /tmp/tmp02.aux | sed -e 's/- /-/' -e 's/-/ /' -e 's/  / /' \
    | awk '/sec/ {print $4" "$8}' | sed -e 's/35.0/50.0/' -e 's/30.0/45.0/' \
    -e 's/25.0/40.0/' -e 's/20.0/35.0/' -e 's/15.0/30.0/' -e 's/10.0/25.0/' \
    -e 's/^5.0/20.0/' -e '1i\0.0 0.0' -e '1i\5.0 0.0' -e '1i\10.0 0.0' \
    -e '1i\15.0 0.0' -e '$a\55.0 0.0' > /tmp/p02.out

200 echo -n "Generant grafica ..."
    $GNUPLOT $GRAFIC_T5
    echo "OK."
    rm /tmp/tmp01.aux /tmp/tmp02.aux /tmp/p01.out /tmp/p02.out
    ;;

    9)
    echo "PROVA 9:";
    echo "Usuaris normals: 1."
    echo "Usuaris QoS: 1 (15s retard)."
210 echo -n "Calculant ... "
    su $QOS1 -c "sleep 15 && $IPERF -c $SERVIDOR -p $PORT -i 5 -t 35 > /tmp/tmp01.aux &";
    su $NORMAL1 -c "$IPERF -c $SERVIDOR -p $PORT -i 5 -t 50 > /tmp/tmp02.aux &";
    sleep 52;
    echo "OK."

    sed -n '7,13p' /tmp/tmp01.aux | sed -e 's/- /-/' -e 's/-/ /' -e 's/  / /' \
    | awk '/sec/ {print $4" "$8}' | sed -e 's/35.0/50.0/' -e 's/30.0/45.0/' \
    -e 's/25.0/40.0/' -e 's/20.0/35.0/' -e 's/15.0/30.0/' -e 's/10.0/25.0/' \
    -e 's/^5.0/20.0/' -e '1i\0.0 0.0' -e '1i\5.0 0.0' -e '1i\10.0 0.0' \
220 -e '1i\15.0 0.0' -e '$a\55.0 0.0' > /tmp/p01.out

    sed -n '7,16p' /tmp/tmp02.aux | sed -e 's/- /-/' -e 's/-/ /' -e 's/  / /' \
    | awk '/sec/ {print $4" "$8}' | sed -e '1i\0.0 0.0' -e '$a\55.0 0.0' > /tmp/p02.out

    echo -n "Generant grafica ..."
    $GNUPLOT $GRAFIC_T5
    echo "OK."
    rm /tmp/tmp01.aux /tmp/tmp02.aux /tmp/p01.out /tmp/p02.out
    ;;
230
    10)
    echo "PROVA 10:";
    echo "Usuaris normals: 2."
    echo "Usuaris QoS: 2."
    echo -n "Calculant ... "
    su $QOS1 -c "$IPERF -c $SERVIDOR -p $PORT -i 5 -t 50 > /tmp/tmp01.aux &";
    su $QOS2 -c "$IPERF -c $SERVIDOR -p $PORT -i 5 -t 50 > /tmp/tmp02.aux &";
    su $NORMAL1 -c "$IPERF -c $SERVIDOR -p $PORT -i 5 -t 50 > /tmp/tmp03.aux &";
    su $NORMAL2 -c "$IPERF -c $SERVIDOR -p $PORT -i 5 -t 50 > /tmp/tmp04.aux &";
240 sleep 52;
    echo "OK."

    sed -n '7,16p' /tmp/tmp01.aux | sed -e 's/- /-/' -e 's/-/ /' -e 's/  / /' \
    | awk '/sec/ {print $4" "$8}' | sed -e '1i\0.0 0.0' -e '$a\55.0 0.0' > /tmp/p01.out

    sed -n '7,16p' /tmp/tmp02.aux | sed -e 's/- /-/' -e 's/-/ /' -e 's/  / /' \
    | awk '/sec/ {print $4" "$8}' | sed -e '1i\0.0 0.0' -e '$a\55.0 0.0' > /tmp/p02.out

    sed -n '7,16p' /tmp/tmp03.aux | sed -e 's/- /-/' -e 's/-/ /' -e 's/  / /' \
250 | awk '/sec/ {print $4" "$8}' | sed -e '1i\0.0 0.0' -e '$a\55.0 0.0' > /tmp/p03.out

    sed -n '7,16p' /tmp/tmp04.aux | sed -e 's/- /-/' -e 's/-/ /' -e 's/  / /' \
    | awk '/sec/ {print $4" "$8}' | sed -e '1i\0.0 0.0' -e '$a\55.0 0.0' > /tmp/p04.out

    echo -n "Generant grafica ..."
    $GNUPLOT $GRAFIC_T6
    echo "OK."
    rm /tmp/tmp01.aux /tmp/tmp02.aux /tmp/tmp03.aux /tmp/tmp04.aux \
    /tmp/p01.out /tmp/p02.out /tmp/p03.out /tmp/p04.out
260 ;;
```

```
11)
echo "PROVA 11:";
echo "Usuaris normals: 1 + 1 (30s retard).."
echo "Usuaris QoS: 1 + 1 (15s retard).."
echo -n "Calculant ... "
su $QOS1 -c "$IPERF -c $SERVIDOR -p $PORT -i 5 -t 50 > /tmp/tmp01.aux &";
su $QOS2 -c "sleep 15 && $IPERF -c $SERVIDOR -p $PORT -i 5 -t 35 > /tmp/tmp02.aux &";
su $NORMAL1 -c "$IPERF -c $SERVIDOR -p $PORT -i 5 -t 50 > /tmp/tmp03.aux &";
270 su $NORMAL2 -c "sleep 30 && $IPERF -c $SERVIDOR -p $PORT -i 5 -t 20 > /tmp/tmp04.aux &";
sleep 52;
echo "OK."

sed -n '7,16p' /tmp/tmp01.aux | sed -e 's/- /-/' -e 's/-/ /' -e 's/  / /' \
| awk '/sec/ {print $4" "$8}' | sed -e '1i\0.0 0.0' -e '$a\55.0 0.0' > /tmp/p01.out

sed -n '7,13p' /tmp/tmp02.aux | sed -e 's/- /-/' -e 's/-/ /' -e 's/  / /' \
| awk '/sec/ {print $4" "$8}' | sed -e 's/35.0/50.0/' -e 's/30.0/45.0/' \
-e 's/25.0/40.0/' -e 's/20.0/35.0/' -e 's/15.0/30.0/' -e 's/10.0/25.0/' \
280 -e 's/^5.0/20.0/' -e '1i\0.0 0.0' -e '1i\5.0 0.0' -e '1i\10.0 0.0' \
-e '1i\15.0 0.0' -e '$a\55.0 0.0' > /tmp/p02.out

sed -n '7,16p' /tmp/tmp03.aux | sed -e 's/- /-/' -e 's/-/ /' -e 's/  / /' \
| awk '/sec/ {print $4" "$8}' | sed -e '1i\0.0 0.0' -e '$a\55.0 0.0' > /tmp/p03.out

sed -n '7,10p' /tmp/tmp04.aux | sed -e 's/- /-/' -e 's/-/ /' -e 's/  / /' \
| awk '/sec/ {print $4" "$8}' | sed -e 's/20.0/50.0/' -e 's/15.0/45.0/' \
-e 's/10.0/40.0/' -e 's/^5.0/35.0/' -e '1i\0.0 0.0' -e '1i\5.0 0.0' -e '1i\10.0 0.0' \
-e '1i\15.0 0.0' -e '1i\20.0 0.0' -e '1i\25.0 0.0' -e '1i\30.0 0.0' \
290 -e '$a\55.0 0.0' > /tmp/p04.out

echo -n "Generant grafica ..."
$GNUPLOT $GRAFIC_T6
echo "OK."
rm /tmp/tmp01.aux /tmp/tmp02.aux /tmp/tmp03.aux /tmp/tmp04.aux \
/tmp/p01.out /tmp/p02.out /tmp/p03.out /tmp/p04.out
;;

*)
300 echo "Funcionament: proves.sh #numero" >&2
exit 1
;;

esac

exit 0
```

C.2 L'entorn de CPU

En aquesta secció s'exposa tot el codi font del script **provesCPU.sh**, representat a C.2 i implementat amb llenguatge *shell*. A grans trets, aquest script s'encarrega de cridar, amb els paràmetres adequats, el programa **provesCPU.c**, el qual està exposat al següent apartat. Mentre aquest programa s'executa en segon pla, el script s'encarrega de recollir les dades necessàries per generar un gràfic on plasmar els resultats. Cada gràfic mostra el seguiment al llarg del temps dels processos implicats en referència al percentatge d'ús de la CPU.

Listing C.2: provesCPU.sh

```
#!/bin/bash

PROVES=/<direccio>/provesCPU
4 GNUPLOT=/usr/bin/gnuplot

GRAFIC_T01=/<direccio>/gp01.sh
GRAFIC_T02=/<direccio>/gp02.sh
GRAFIC_T03=/<direccio>/gp03.sh
GRAFIC_T04=/<direccio>/gp04.sh
GRAFIC_T05=/<direccio>/gp05.sh
```

```
   GRAFIC_T06=/<direccio>/gp06.sh
   GRAFIC_T07=/<direccio>/gp07.sh
   GRAFIC_T08=/<direccio>/gp08.sh
14 GRAFIC_T09=/<direccio>/gp09.sh
   GRAFIC_T10=/<direccio>/gp10.sh
   GRAFIC_T11=/<direccio>/gp11.sh
   GRAFIC_T12=/<direccio>/gp12.sh
   GRAFIC_T13=/<direccio>/gp13.sh
   GRAFIC_T14=/<direccio>/gp14.sh

   percentatges(){
   i=10;
   while [ $i -le 130 ]; do
24    ps -eo pcpu,user | awk '/alfred01/ {print $1}' | xargs echo -e $i >> /tmp/qos01.aux
      ps -eo pcpu,user | awk '/alfred02/ {print $1}' | xargs echo -e $i >> /tmp/qos02.aux
      ps -eo pcpu,user | awk '/alfred03/ {print $1}' | xargs echo -e $i >> /tmp/qos03.aux
      ps -eo pcpu,user | awk '/alfred04/ {print $1}' | xargs echo -e $i >> /tmp/qos04.aux

      ps -eo pcpu,user | awk '/usuari01/ {print $1}' | xargs echo -e $i >> /tmp/usuari01.aux
      ps -eo pcpu,user | awk '/usuari02/ {print $1}' | xargs echo -e $i >> /tmp/usuari02.aux
      ps -eo pcpu,user | awk '/usuari03/ {print $1}' | xargs echo -e $i >> /tmp/usuari03.aux
      ps -eo pcpu,user | awk '/usuari04/ {print $1}' | xargs echo -e $i >> /tmp/usuari04.aux
      let i=i+10;
34    sleep 10;
   done
   }

   pulir(){
   sed -n '/[0-9]*\.[0-9]*/p' $1 > /tmp/tmp.aux
   sed /tmp/tmp.aux -e '1i\0.0 0.0' > $1
   rm /tmp/tmp.aux
   }

44 if [ $1 == '1' ]; then
      echo "PROVA 1:"
      if [ $2 == '1' ] && [ $3 == '0' ]; then
         echo "Usuaris QoS: 1."
         echo "Usuaris normals: 0."
         echo "Calculant:"
         $PROVES $1 $2 $3 &
         sleep 2;
         percentatges
         pulir /tmp/qos01.aux
54       echo "OK."
         echo -n "Generant grafica: "
         $GNUPLOT $GRAFIC_T01
         echo "OK."
      fi

      if [ $2 == '0' ] && [ $3 == '1' ]; then
         echo "Usuaris QoS: 0."
         echo "Usuaris normals: 1."
         echo "Calculant:"
64       $PROVES $1 $2 $3 &
         sleep 2;
         percentatges
         pulir /tmp/usuari01.aux
         echo "OK."
         echo -n "Generant grafica: "
         $GNUPLOT $GRAFIC_T02
         echo "OK."
      fi
   fi
74
   if [ $1 == '2' ]; then
      if [ $2 == '2' ] && [ $3 == '0' ]; then
         echo "Usuaris QoS: 2."
         echo "Usuaris normals: 0."
         echo "Calculant:"
         $PROVES $1 $2 $3 &
         sleep 2;
         percentatges
         pulir /tmp/qos01.aux
84       pulir /tmp/qos02.aux
         echo "OK."
         echo -n "Generant grafica: "
         $GNUPLOT $GRAFIC_T03
         echo "OK."
      fi

      if [ $2 == '1' ] && [ $3 == '1' ]; then
         echo "Usuaris QoS: 1."
         echo "Usuaris normals: 1."
94       echo "Calculant:"
```

```
        $PROVES $1 $2 $3 &
        sleep 2;
        percentatges
        pulir /tmp/qos01.aux
        pulir /tmp/usuari01.aux
        echo "OK."
        echo -n "Generant grafica: "
        $GNUPLOT $GRAFIC_T04
        echo "OK."
104     fi

    if [ $2 == '0' ] && [ $3 == '2' ]; then
        echo "Usuaris QoS: 0."
        echo "Usuaris normals: 2."
        echo "Calculant:"
        $PROVES $1 $2 $3 &
        sleep 2;
        percentatges
        pulir /tmp/usuari01.aux
114     pulir /tmp/usuari02.aux
        echo "OK."
        echo -n "Generant grafica: "
        $GNUPLOT $GRAFIC_T05
        echo "OK."
    fi
fi

if [ $1 == '3' ]; then
    if [ $2 == '3' ] && [ $3 == '0' ]; then
124     echo "Usuaris QoS: 3."
        echo "Usuaris normals: 0."
        echo "Calculant:"
        $PROVES $1 $2 $3 &
        sleep 2;
        percentatges
        pulir /tmp/qos01.aux
        pulir /tmp/qos02.aux
        pulir /tmp/qos03.aux
        echo "OK."
134     echo -n "Generant grafica: "
        $GNUPLOT $GRAFIC_T06
        echo "OK."
    fi

    if [ $2 == '2' ] && [ $3 == '1' ]; then
        echo "Usuaris QoS: 2."
        echo "Usuaris normals: 1."
        echo "Calculant:"
        $PROVES $1 $2 $3 &
144     sleep 2;
        percentatges
        pulir /tmp/qos01.aux
        pulir /tmp/qos02.aux
        pulir /tmp/usuari01.aux
        echo "OK."
        echo -n "Generant grafica: "
        $GNUPLOT $GRAFIC_T07
        echo "OK."
    fi
154
    if [ $2 == '1' ] && [ $3 == '2' ]; then
        echo "Usuaris QoS: 1."
        echo "Usuaris normals: 2."
        echo "Calculant:"
        $PROVES $1 $2 $3 &
        sleep 2;
        percentatges
        pulir /tmp/qos01.aux
        pulir /tmp/usuari01.aux
164     pulir /tmp/usuari02.aux
        echo "OK."
        echo -n "Generant grafica: "
        $GNUPLOT $GRAFIC_T08
        echo "OK."
    fi

    if [ $2 == '0' ] && [ $3 == '3' ]; then
        echo "Usuaris QoS: 0."
        echo "Usuaris normals: 3."
174     echo "Calculant:"
        $PROVES $1 $2 $3 &
        sleep 2;
        percentatges
        pulir /tmp/usuari01.aux
```

```
        pulir /tmp/usuari02.aux
        pulir /tmp/usuari03.aux
        echo "OK."
        echo -n "Generant grafica: "
        $GNUPLOT $GRAFIC_T09
184     echo "OK."
    fi
fi


if [ $1 == '4' ]; then
    if [ $2 == '4' ] && [ $3 == '0' ]; then
        echo "Usuaris QoS: 4."
        echo "Usuaris normals: 0."
        echo "Calculant:"
194     $PROVES $1 $2 $3 &
        sleep 2;
        percentatges
        pulir /tmp/qos01.aux
        pulir /tmp/qos02.aux
        pulir /tmp/qos03.aux
        pulir /tmp/qos04.aux
        echo "OK."
        echo -n "Generant grafica: "
        $GNUPLOT $GRAFIC_T10
204     echo "OK."
    fi

    if [ $2 == '3' ] && [ $3 == '1' ]; then
        echo "Usuaris QoS: 3."
        echo "Usuaris normals: 1."
        echo "Calculant:"
        $PROVES $1 $2 $3 &
        sleep 2;
        percentatges
214     pulir /tmp/qos01.aux
        pulir /tmp/qos02.aux
        pulir /tmp/qos03.aux
        pulir /tmp/usuari01.aux
        echo "OK."
        echo -n "Generant grafica: "
        $GNUPLOT $GRAFIC_T11
        echo "OK."
    fi

224 if [ $2 == '2' ] && [ $3 == '2' ]; then
        echo "Usuaris QoS: 2."
        echo "Usuaris normals: 2."
        echo "Calculant:"
        $PROVES $1 $2 $3 &
        sleep 2;
        percentatges
        pulir /tmp/qos01.aux
        pulir /tmp/qos02.aux
        pulir /tmp/usuari01.aux
234     pulir /tmp/usuari02.aux
        echo "OK."
        echo -n "Generant grafica: "
        $GNUPLOT $GRAFIC_T12
        echo "OK."
    fi

    if [ $2 == '1' ] && [ $3 == '3' ]; then
        echo "Usuaris QoS: 1."
        echo "Usuaris normals: 3."
244     echo "Calculant:"
        $PROVES $1 $2 $3 &
        sleep 2;
        percentatges
        pulir /tmp/qos01.aux
        pulir /tmp/usuari01.aux
        pulir /tmp/usuari02.aux
        pulir /tmp/usuari03.aux
        echo "OK."
        echo -n "Generant grafica: "
254     $GNUPLOT $GRAFIC_T13
        echo "OK."
    fi

    if [ $2 == '0' ] && [ $3 == '4' ]; then
        echo "Usuaris QoS: 0."
        echo "Usuaris normals: 4."
        echo "Calculant:"
        $PROVES $1 $2 $3 &
```

```
        sleep 2;
264     percentatges
        pulir /tmp/usuari01.aux
        pulir /tmp/usuari02.aux
        pulir /tmp/usuari03.aux
        pulir /tmp/usuari04.aux
        echo "OK."
        echo -n "Generant grafica: "
        $GNUPLOT $GRAFIC_T14
        echo "OK."
    fi
274

fi

rm /tmp/qos01.aux /tmp/qos02.aux /tmp/qos03.aux /tmp/qos04.aux \
   /tmp/usuari01.aux /tmp/usuari02.aux /tmp/usuari03.aux /tmp/usuari04.aux

exit 0
```

En aquest apart s'exposa tot el codi font del programa **provesCPU.c**, mostrat a C.3 i implementat sota el llenguatge de programació C. Aquest programa és la base del script **provesCPU.sh** mencionat anteriorment. A grans trets, el programa principal o pare s'encarrega de crear la quantitat de processos fills indicats al primer paràmetre rebut. Els altres dos paràmetres s'utilitzen per canviar els propietaris dels processos, d'aquesta forma el script **qos.sh** podrà diferenciar entre processos privilegiats, els quals gaudiran de la QoS i els processos normals. Tots els processos fills, independentment del propietati del procés, executen la mateixa funció d'ús massiu de la CPU.

Listing C.3: provesCPU.c

```
/*
   PROVES DE CPU
   Fitxer: procesCPU.c

   Autor: Alfred Fraile Agusti
   afraile@alumnes.udl.cat

   Compilar: gcc -o provesCPU provesCPU.c
9
   Executar: ./provesCPU <processos> <QoS> <normals>
   (processos=QoS+normals, processos<5)

                                                        */

#include <stdlib.h>
#include <stdio.h>
#include <signal.h>
#include <string.h>
19
int pid01;
int pid02;
int pid03;
int pid04;

int estat01;
int estat02;
int estat03;
int estat04;
29
time_t tini, tfi;

void cpu_massiu();

int main(int argc, char *argv[]) {

if(argc==4){

   if((atoi(argv[1])==1)){
39    if((pid01=fork())==0){
         if((atoi(argv[2])==1) && (atoi(argv[3])==0)){
            setuid(1000);
            printf("Pid proces QoS: %d\n",getpid());
         } else if((atoi(argv[2])==0) && (atoi(argv[3])==1)){
```

```
            setuid(1004);
            printf("Pid proces normal: %d\n",getpid());
         } else {
            printf("Error"); exit(0);
         }
49       sleep(1);
         tini = time(NULL);
         cpu_massiu();
         tfi = time(NULL);
         if((atoi(argv[2])==1) && (atoi(argv[3])==0)){
            printf("Pid proces QoS: %d - Temps: %d\n",getpid(),(int)(tfi-tini));
         } else {
            printf("Pid proces normal: %d - Temps: %d\n",getpid(),(int)(tfi-tini));
         }
      } else {
59       wait(&estat01);
         exit(0);
      }
   }


   if((atoi(argv[1])==2)){
      if((pid01=fork())==0){
         if((atoi(argv[2])==2) && (atoi(argv[3])==0)){
            setuid(1000);
69          printf("Pid proces QoS: %d\n",getpid());
         } else if((atoi(argv[2])==1) && (atoi(argv[3])==1)){
            setuid(1000);
            printf("Pid proces QoS: %d\n",getpid());
         } else if((atoi(argv[2])==0) && (atoi(argv[3])==2)){
            setuid(1004);
            printf("Pid proces normal: %d\n",getpid());
         } else {
            printf("Error"); exit(0);
         }
79       sleep(1);
         tini = time(NULL);
         cpu_massiu();
         tfi = time(NULL);
         if((atoi(argv[2])==2) && (atoi(argv[3])==0)){
            printf("Pid proces QoS: %d - Temps: %d\n",getpid(),(int)(tfi-tini));
         } else if((atoi(argv[2])==1) && (atoi(argv[3])==1)){
            printf("Pid proces QoS: %d - Temps: %d\n",getpid(),(int)(tfi-tini));
         } else {
            printf("Pid proces normal: %d - Temps: %d\n",getpid(),(int)(tfi-tini));
89       }
      } else if ((pid02=fork())==0){
         if((atoi(argv[2])==2) && (atoi(argv[3])==0)){
            setuid(1001);
            printf("Pid proces QoS: %d\n",getpid());
         } else if((atoi(argv[2])==1) && (atoi(argv[3])==1)){
            setuid(1004);
            printf("Pid proces normal: %d\n",getpid());
         } else if((atoi(argv[2])==0) && (atoi(argv[3])==2)){
            setuid(1005);
99          printf("Pid proces normal: %d\n",getpid());
         } else {
            printf("Error"); exit(0);
         }
         sleep(1);
         tini = time(NULL);
         cpu_massiu();
         tfi = time(NULL);
         if((atoi(argv[2])==2) && (atoi(argv[3])==0)){
            printf("Pid proces QoS: %d - Temps: %d\n",getpid(),(int)(tfi-tini));
109      } else if((atoi(argv[2])==1) && (atoi(argv[3])==1)){
            printf("Pid proces normal: %d - Temps: %d\n",getpid(),(int)(tfi-tini));
         } else {
            printf("Pid proces normal: %d - Temps: %d\n",getpid(),(int)(tfi-tini));
         }
      } else {
         wait(&estat01);
         wait(&estat02);
         exit(0);
      }
119 }


   if((atoi(argv[1])==3)){
      if((pid01=fork())==0){
         if((atoi(argv[2])==3) && (atoi(argv[3])==0)){
            setuid(1000);
            printf("Pid proces QoS: %d\n",getpid());
         } else if((atoi(argv[2])==2) && (atoi(argv[3])==1)){
```

```
            setuid(1000);
129         printf("Pid proces QoS: %d\n",getpid());
         } else if((atoi(argv[2])==1) && (atoi(argv[3])==2)){
            setuid(1000);
            printf("Pid proces QoS: %d\n",getpid());
         } else if((atoi(argv[2])==0) && (atoi(argv[3])==3)){
            setuid(1004);
            printf("Pid proces normal: %d\n",getpid());
         } else {
            printf("Error"); exit(0);
         }
139      sleep(1);
         tini = time(NULL);
         cpu_massiu();
         tfi = time(NULL);
         if((atoi(argv[2])==3) && (atoi(argv[3])==0)){
            printf("Pid proces QoS: %d - Temps: %d\n",getpid(),(int)(tfi-tini));
         } else if((atoi(argv[2])==2) && (atoi(argv[3])==1)){
            printf("Pid proces QoS: %d - Temps: %d\n",getpid(),(int)(tfi-tini));
         } else if((atoi(argv[2])==1) && (atoi(argv[3])==2)){
            printf("Pid proces QoS: %d - Temps: %d\n",getpid(),(int)(tfi-tini));
149      } else {
            printf("Pid proces normal: %d - Temps: %d\n",getpid(),(int)(tfi-tini));
         }

      } else if ((pid02=fork())==0){
         if((atoi(argv[2])==3) && (atoi(argv[3])==0)){
            setuid(1001);
            printf("Pid proces QoS: %d\n",getpid());
         } else if((atoi(argv[2])==2) && (atoi(argv[3])==1)){
            setuid(1001);
159         printf("Pid proces QoS: %d\n",getpid());
         } else if((atoi(argv[2])==1) && (atoi(argv[3])==2)){
            setuid(1004);
            printf("Pid proces normal: %d\n",getpid());
         } else if((atoi(argv[2])==0) && (atoi(argv[3])==3)){
            setuid(1005);
            printf("Pid proces normal: %d\n",getpid());
         } else {
            printf("Error"); exit(0);
         }
169      sleep(1);
         tini = time(NULL);
         cpu_massiu();
         tfi = time(NULL);
         if((atoi(argv[2])==3) && (atoi(argv[3])==0)){
            printf("Pid proces QoS: %d - Temps: %d\n",getpid(),(int)(tfi-tini));
         } else if((atoi(argv[2])==2) && (atoi(argv[3])==1)){
            printf("Pid proces QoS: %d - Temps: %d\n",getpid(),(int)(tfi-tini));
         } else if((atoi(argv[2])==1) && (atoi(argv[3])==2)){
            printf("Pid proces normal: %d - Temps: %d\n",getpid(),(int)(tfi-tini));
179      } else {
            printf("Pid proces normal: %d - Temps: %d\n",getpid(),(int)(tfi-tini));
         }


      } else if ((pid03=fork())==0){
         if((atoi(argv[2])==3) && (atoi(argv[3])==0)){
            setuid(1002);
            printf("Pid proces QoS: %d\n",getpid());
         } else if((atoi(argv[2])==2) && (atoi(argv[3])==1)){
189         setuid(1004);
            printf("Pid proces normal: %d\n",getpid());
         } else if((atoi(argv[2])==1) && (atoi(argv[3])==2)){
            setuid(1005);
            printf("Pid proces normal: %d\n",getpid());
         } else if((atoi(argv[2])==0) && (atoi(argv[3])==3)){
            setuid(1006);
            printf("Pid proces normal: %d\n",getpid());
         } else {
            printf("Error"); exit(0);
199      }
         sleep(1);
         tini = time(NULL);
         cpu_massiu();
         tfi = time(NULL);
         if((atoi(argv[2])==3) && (atoi(argv[3])==0)){
            printf("Pid proces QoS: %d - Temps: %d\n",getpid(),(int)(tfi-tini));
         } else if((atoi(argv[2])==2) && (atoi(argv[3])==1)){
            printf("Pid proces normal: %d - Temps: %d\n",getpid(),(int)(tfi-tini));
         } else if((atoi(argv[2])==1) && (atoi(argv[3])==2)){
209         printf("Pid proces normal: %d - Temps: %d\n",getpid(),(int)(tfi-tini));
         } else {
            printf("Pid proces normal: %d - Temps: %d\n",getpid(),(int)(tfi-tini));
```

```
            }

         } else {
            wait(&estat01);
            wait(&estat02);
            wait(&estat03);
            exit(0);
219      }
      }


      if((atoi(argv[1])==4)){
         if((pid01=fork())==0){
            if((atoi(argv[2])==4) && (atoi(argv[3])==0)){
               setuid(1000);
               printf("Pid proces QoS: %d\n",getpid());
            } else if((atoi(argv[2])==3) && (atoi(argv[3])==1)){
229            setuid(1000);
               printf("Pid proces QoS: %d\n",getpid());
            } else if((atoi(argv[2])==2) && (atoi(argv[3])==2)){
               setuid(1000);
               printf("Pid proces QoS: %d\n",getpid());
            } else if((atoi(argv[2])==1) && (atoi(argv[3])==3)){
               setuid(1000);
               printf("Pid proces QoS: %d\n",getpid());
            } else if((atoi(argv[2])==0) && (atoi(argv[3])==4)){
               setuid(1004);
239            printf("Pid proces normal: %d\n",getpid());
            } else {
               printf("Error"); exit(0);
            }
            sleep(1);
            tini = time(NULL);
            cpu_massiu();
            tfi = time(NULL);
            if((atoi(argv[2])==4) && (atoi(argv[3])==0)){
               printf("Pid proces QoS: %d - Temps: %d\n",getpid(),(int)(tfi-tini));
249         } else if((atoi(argv[2])==3) && (atoi(argv[3])==1)){
               printf("Pid proces QoS: %d - Temps: %d\n",getpid(),(int)(tfi-tini));
            } else if((atoi(argv[2])==2) && (atoi(argv[3])==2)){
               printf("Pid proces QoS: %d - Temps: %d\n",getpid(),(int)(tfi-tini));
            } else if((atoi(argv[2])==1) && (atoi(argv[3])==3)){
               printf("Pid proces QoS: %d - Temps: %d\n",getpid(),(int)(tfi-tini));
            } else {
               printf("Pid proces normal: %d - Temps: %d\n",getpid(),(int)(tfi-tini));
            }

259      } else if ((pid02=fork())==0){
            if((atoi(argv[2])==4) && (atoi(argv[3])==0)){
               setuid(1001);
               printf("Pid proces QoS: %d\n",getpid());
            } else if((atoi(argv[2])==3) && (atoi(argv[3])==1)){
               setuid(1001);
               printf("Pid proces QoS: %d\n",getpid());
            } else if((atoi(argv[2])==2) && (atoi(argv[3])==2)){
               setuid(1001);
               printf("Pid proces QoS: %d\n",getpid());
269         } else if((atoi(argv[2])==1) && (atoi(argv[3])==3)){
               setuid(1004);
               printf("Pid proces normal: %d\n",getpid());
            } else if((atoi(argv[2])==0) && (atoi(argv[3])==4)){
               setuid(1005);
               printf("Pid proces normal: %d\n",getpid());
            } else {
               printf("Error"); exit(0);
            }
            sleep(1);
279         tini = time(NULL);
            cpu_massiu();
            tfi = time(NULL);
            if((atoi(argv[2])==4) && (atoi(argv[3])==0)){
               printf("Pid proces QoS: %d - Temps: %d\n",getpid(),(int)(tfi-tini));
            } else if((atoi(argv[2])==3) && (atoi(argv[3])==1)){
               printf("Pid proces QoS: %d - Temps: %d\n",getpid(),(int)(tfi-tini));
            } else if((atoi(argv[2])==2) && (atoi(argv[3])==2)){
               printf("Pid proces QoS: %d - Temps: %d\n",getpid(),(int)(tfi-tini));
            } else if((atoi(argv[2])==1) && (atoi(argv[3])==3)){
289            printf("Pid proces normal: %d - Temps: %d\n",getpid(),(int)(tfi-tini));
            } else {
               printf("Pid proces normal: %d - Temps: %d\n",getpid(),(int)(tfi-tini));
            }


         } else if ((pid03=fork())==0){
```

```
            if((atoi(argv[2])==4) && (atoi(argv[3])==0)){
               setuid(1002);
               printf("Pid proces QoS: %d\n",getpid());
299         } else if((atoi(argv[2])==3) && (atoi(argv[3])==1)){
               setuid(1002);
               printf("Pid proces QoS: %d\n",getpid());
            } else if((atoi(argv[2])==2) && (atoi(argv[3])==2)){
               setuid(1004);
               printf("Pid proces normal: %d\n",getpid());
            } else if((atoi(argv[2])==1) && (atoi(argv[3])==3)){
               setuid(1005);
               printf("Pid proces normal: %d\n",getpid());
            } else if((atoi(argv[2])==0) && (atoi(argv[3])==4)){
309            setuid(1006);
               printf("Pid proces normal: %d\n",getpid());
            } else {
               printf("Error"); exit(0);
            }
            sleep(1);
            tini = time(NULL);
            cpu_massiu();
            tfi = time(NULL);
            if((atoi(argv[2])==4) && (atoi(argv[3])==0)){
319            printf("Pid proces QoS: %d - Temps: %d\n",getpid(),(int)(tfi-tini));
            } else if((atoi(argv[2])==3) && (atoi(argv[3])==1)){
               printf("Pid proces QoS: %d - Temps: %d\n",getpid(),(int)(tfi-tini));
            } else if((atoi(argv[2])==2) && (atoi(argv[3])==2)){
               printf("Pid proces normal: %d - Temps: %d\n",getpid(),(int)(tfi-tini));
            } else if((atoi(argv[2])==1) && (atoi(argv[3])==3)){
               printf("Pid proces normal: %d - Temps: %d\n",getpid(),(int)(tfi-tini));
            } else {
               printf("Pid proces normal: %d - Temps: %d\n",getpid(),(int)(tfi-tini));
            }
329
         } else if ((pid04=fork())==0){
            if((atoi(argv[2])==4) && (atoi(argv[3])==0)){
               setuid(1003);
               printf("Pid proces QoS: %d\n",getpid());
            } else if((atoi(argv[2])==3) && (atoi(argv[3])==1)){
               setuid(1004);
               printf("Pid proces normal: %d\n",getpid());
            } else if((atoi(argv[2])==2) && (atoi(argv[3])==2)){
               setuid(1005);
339            printf("Pid proces normal: %d\n",getpid());
            } else if((atoi(argv[2])==1) && (atoi(argv[3])==3)){
               setuid(1006);
               printf("Pid proces normal: %d\n",getpid());
            } else if((atoi(argv[2])==0) && (atoi(argv[3])==4)){
               setuid(1007);
               printf("Pid proces normal: %d\n",getpid());
            } else {
               printf("Error"); exit(0);
            }
349         sleep(1);
            tini = time(NULL);
            cpu_massiu();
            tfi = time(NULL);
            if((atoi(argv[2])==4) && (atoi(argv[3])==0)){
               printf("Pid proces QoS: %d - Temps: %d\n",getpid(),(int)(tfi-tini));
            } else if((atoi(argv[2])==3) && (atoi(argv[3])==1)){
               printf("Pid proces normal: %d - Temps: %d\n",getpid(),(int)(tfi-tini));
            } else if((atoi(argv[2])==2) && (atoi(argv[3])==2)){
               printf("Pid proces normal: %d - Temps: %d\n",getpid(),(int)(tfi-tini));
359         } else if((atoi(argv[2])==1) && (atoi(argv[3])==3)){
               printf("Pid proces normal: %d - Temps: %d\n",getpid(),(int)(tfi-tini));
            } else {
               printf("Pid proces normal: %d - Temps: %d\n",getpid(),(int)(tfi-tini));
            }
         } else {
            wait(&estat01);
            wait(&estat02);
            wait(&estat03);
            wait(&estat04);
369         exit(0);
         }

      }


   } else {
      printf("Error - Nombre parametres incorrecte\n");
      exit(1);
   }
379
```

```
}

void cpu_massiu(){
int i,j;
while(i<1000000000){
    i=i+1000; i=i-1000;
    while(j<1000000000){
        j=j+1000; j=j-1000;
        j++;
    }
    i++;
}

}
```

389