CONCEPTION DU SOUS-SYSTÈME RÉSEAU DES SYSTÈMES D'EXPLOITATION : LES CAS LINUX VOLUME UN ETHERNET, IP et UDP

Patrick Cegielski
cegielski@univ-paris12.fr

Juin 2005

Pour Irène et Marie

Legal Notice

Copyright © 2005 Patrick Cegielski
Université Paris 12 - IUT
Route forestière Hurtault
F-77300 Fontainebleau
cegielski@univ-paris12.fr

Préface

On peut distinguer quatre niveaux en ce qui concerne les rapports entre une personne et le sous-système réseau d'un système d'exploitation :

- Au *niveau utilisateur*, on utilise des applications (logiciels) faisant intervenir une connexion informatique, telles que le courrier électronique, la consultation de pages web, le transfert de fichiers.
- Le *niveau administrateur* consiste à paramétrer le sous-système réseau du système d'exploitation, à le tenir à jour et à apporter des ressources aux utilisateurs (attribution d'une adresse IP, par exemple).
- Le *niveau programmation réseau* permet de concevoir des applications, que celles-ci soient utilisées localement ou connaissent une grande diffusion. Cette programmation se fait presque toujours en langage C en utilisant des *appels système* constituant une API (*Application Programmer Interface*).
- Le *niveau conception du noyau réseau* consiste à concevoir et à implémenter cette API réseau.

Le but de ce livre est de faire comprendre l'implémentation du sous-système réseau des systèmes d'exploitation, c'est-à-dire de l'API réseau. Il existe de très nombreux livres (en nombre trop important même) pour décrire le niveau utilisateur, qui fait de toute façon maintenant partie de la culture de base de tout un chacun. Il est même partie intégrante du "socle fondamental" de l'école tel que défini en France en 2004. L'administration réseau a également donnée lieu à une littérature abondante, par exemple [K-D–01]. La littérature sur la programmation réseau est moins fournie : la référence est [STE–90] dans le cas Unix. La littérature sur l'implémentation du sous-sytème réseau est plus que maigrichonne ([HER–00], [WPRMB–02], [C-P–02]). Il n'existe aucun livre général à notre connaissance.

Le choix du système d'exploitation permettant d'illustrer notre propos s'est tout naturellement porté sur Linux, puisque nous pouvons nous en procurer les sources très facilement.

Notre but n'est pas d'étudier TOUTE l'implémentation du sous-système réseau (Linux), ce qui occuperait plusieurs volumes comme celui-ci, dont le nombre de pages est déjà suffisamment imposant. Nous voulons simplement, si l'on peut dire, étudier un cas réel depuis le début jusqu'à la fin.

Dans la première partie, nous donnons une vue d'ensemble sur les réseaux et les modèles d'architecture (OSI à sept couches, TCP/IP à trois couches et le modèle mixte à cinq couches), nous décrivons très rapidement la couche physique (première couche) puisque la programmation (et donc le sous-système réseau) n'y a que faire. Pour illustrer la couche liaison (deuxième couche), nous avons choisi Ethernet, qui est la référence pour Linux. La couche réseau (troisième couche) est illustrée avec IP, devenu la référence absolue quelle que soit le réseau. Bien que le modèle de référence s'appelle TCP/IP, nous avons choisi d'illustrer la couche transport (la quatrième

couche) avec UDP et non TCP, ce qui est plus simple bien que déjà suffisamment complexe comme cela. Il n'y a pas à s'interroger sur le choix de l'API réseau puisque l'API des sockets est, comme IP, devenu la référence presque exclusive. Lors de la présentation de cette API, nous donnons quelques programmes très simples, ce qui nous aura permis de décrire le sous-système réseau d'un bout à l'autre.

La première partie reste générale, l'implémentation Linux n'y joue aucun rôle. Dans la seconde partie, nous étudions l'implémentation Linux de la réception d'un datagramme UDP, depuis la trame Ethernet et en passant par le paquet IP ordinaire. Nous commençons par la réception, même s'il a bien fallu l'envoyer, car son implémentation est plus simple que l'envoi.

La troisième partie traite de l'envoi d'un datagramme UDP.

La figure 0-1 montre la hiérarchie entre les chapitres.

	Réception	Général	Linux	Envoi
Généralités		ch1	ch10	
TCP/IP		ch5 ch9	ch11 ch12 ch13	
Couche physique		ch2		
Couche Liaison	 ch19	ch3 ch4 ch8	ch14 ch15 ch16 ch17	 ch34
Couche réseau	 ch23 ch36		ch18 ch20 ch21 ch22	 ch33 ch36
Couche transport	 ch25		ch24	 ch32
API réseau	 ch30	ch6 ch7	ch26 ch27 ch28 ch29 ch35	 ch31

FIG. 1 – *Hiérarchie entre les chapitres*

Le sous-système réseau est devenu partie intégrante des systèmes d'exploitation actuels. Nous supposons donc que le lecteur ait quelques notions sur l'implémentation des systèmes d'exploitation en général et de Linux en particulier. Il existe une littérature assez conséquente sur ce sujet, souvent excellente. Nous nous permettons de recommander notre livre [CEG–03], qui illustre les propos par le tout premier noyau Linux, qui ne concerne donc que l'essentiel, ce qui est suffisant pour aborder ce volume.

Ce livre n'a d'autre but que de publier en un seul volume les aspects suivants de l'implémentation d'un sous-système réseau :

- les concepts généraux sous-jacents à l'implémentation d'un sous-système réseau ;
- les concepts fondamentaux d'Ethernet et de TCP/IP ;
- la documentation sur un contrôleur de périphérique réseau, à savoir le carte 3Com 501 pour compatible PC ;
- une présentation des choix faits pour l'implémentation de Linux, suivie d'extraits de fichiers sources, repérables facilement par l'indication Code Linux située en marge, paraphrasés en français ; ces paraphrases ne sont pas théoriquement indispensables mais oh combien appréciables en pratique.

Nous parlerons quelquefois de "fonction générale" en décrivant rapidement le comportement de celle-ci mais sans nous reporter aux sources dans le cas où cela ne concerne pas les réseaux à proprement parler. Cet ouvrage est déjà suffisamment imposant comme cela par sa taille pour que nous n'ayons pas à revoir toute l'implémentation du système d'exploitation.

L'index est une partie fondamentale du livre, et pas seulement par le volume occupé (à peu près 30 pages) :

- Il s'agit tout d'abord d'un index des concepts fondamentaux concernant les réseaux. On s'y réfèrera pour chercher rapidement un mot dont on ne connaît pas la signification ou pour lequel on voudrait plus d'informations. Il renvoie à la page où il est défini.
- Il s'agit également d'un dictionnaire français-anglais et anglais-français de ces mêmes concepts. La dénomination anglaise de tout concept est toujours indiquée entre parenthèse lorsque celui-ci est introduit (en français). Cet index servira certainement également à ceux qui, soucieux de la langue française, veulement éviter les anglicismes. Pour tel mot anglais rencontré, l'index indique à quel page le mot français correspondant (et sa traduction anglaise entre parenthèses) est défini.
- Il s'agit également d'un index des grands noms qui ont participé aux réseaux, bien que cet aspect n'a pas encore été assez développé à notre goût (peut-être pour une édition ultérieure ?).
- Les sources, de façon générale et de Linux en particulier, sont souvent d'approche un peu repoussante car il est difficile de s'y retrouver. On dispose maintenant pour cela de l'excellent outil `lxr` que nous décrivons au chapitre 10. Celui-ci connaît cependant quelques limites (il n'indexe pas les constantes définies dans un type énuméré par exemple) et, surtout, il ne renvoie pas aux commentaires adéquats. Notre index référence **toutes** les constantes, macros, variables, fonctions et tous les types fondamentaux de l'implémentation du sous-système réseau de Linux, renvoyant à la page où la définition est citée et commentée. La version du noyau choisie pour cette édition est 2.6.10, c'est-à-dire la dernière version au moment de la fin de la rédaction de ce livre.
- On n'a certainement pas intérêt à étudier les sources fichier par fichier car la philosophie de mise en fichier n'est pas suffisamment cohérente pour Linux. On peut cependant, à l'occasion de l'étude de tel ou tel point d'un fichier, se demander si cette partie du fichier est commentée dans ce livre. C'est pourquoi les fichiers sont indexés, avec les lignes des fichiers comme second niveau d'indexation.

Notre espoir serait, tel l'aigle, de pouvoir survoler cet immense champ qu'est le réseau pour avoir une vue d'ensemble et, là où ça nous intéresse plus particulièrement, piquer jusqu'au sol pour en voir le moindre détail.

Cet espoir est-il réalisé ? C'est autre histoire mais c'est en tous cas le but vers lequel nous devons tendre.

Table des matières

Préface **v**

1 Les réseaux informatiques **1**
1.1 Introduction aux réseaux 1
1.1.1 Origine des réseaux informatique 1
1.1.2 Notion de réseau informatique 2
1.1.3 Matériel et logiciel pour réseau 2
1.1.4 Protocoles 2
1.1.5 Systèmes propriétaires et systèmes ouverts 2
1.2 Réseau et commutation 3
1.2.1 Notion de commutation 3
1.2.2 Types de commutation 4
1.2.3 Réseaux hiérarchisés et réseaux maillés 6
1.3 Modèles en couches 7
1.3.1 Étude générale 7
1.3.2 Modèle OSI 9
1.3.3 Suite TCP/IP 10
1.3.4 Le modèle des sockets 11
1.3.5 Modèle hybride 11
1.3.6 Cas de Linux 11
1.3.7 Les en-têtes de protocole 12
1.4 Historique 12
1.4.1 Naissance des ordinateurs 12
1.4.2 Terminaux distants 13
1.4.3 Première mise en réseau : 1965 13
1.4.4 Réseaux de communication 13
1.4.5 Commutation par paquets 14
1.4.6 Le premier réseau : ARPANET en 1971 14

I Couches physique et de liaison **17**

2 La couche physique **21**
2.1 Les supports physiques 22
2.2 Interface support physique/ordinateur 22
2.2.1 Rappels sur les liaisons série 22
2.2.2 La synchronisation et le codage Manchester 24
2.3 Hiérarchie des réseaux 25

2.3.1 Hiérarchie suivant la taille . . . 25
2.3.2 Réseau à diffusion et réseau point-à-point . . . 26
2.4 Réseaux locaux . . . 27
2.4.1 Technologie des réseaux locaux . . . 27
2.4.2 Topologie des réseaux locaux . . . 27
2.5 Historique . . . 30
2.5.1 Émergence de réseaux de recherche . . . 30
2.5.2 Naissance des réseaux à diffusion : ALOHANET . . . 31
2.5.3 Naissance des réseaux locaux : Ethernet . . . 32
2.5.4 Interréseaux . . . 33

3 Vue d'ensemble sur la couche liaison **35**
3.1 Détermination des trames . . . 36
3.1.1 Paquet, trame et somme de contrôle . . . 36
3.1.2 Découpage en trames . . . 37
3.2 Garantie de transmission correcte . . . 39
3.2.1 Première idée : confirmation ou infirmation . . . 39
3.2.2 Deuxième idée : confirmation et délai . . . 40
3.2.3 Troisième idée : fenêtre glissante . . . 40
3.3 Contrôle de flux . . . 41
3.4 Gestion des canaux . . . 42
3.4.1 Allocation statique des canaux . . . 42
3.4.2 Gestion des canaux à accès multiples . . . 43

4 Ethernet **45**
4.1 Vue générale sur Ethernet . . . 45
4.1.1 Couche physique : types de câblage Ethernet . . . 45
4.1.2 Couche de liaison : détection des collisions . . . 46
4.2 Le standard IEEE . . . 47
4.2.1 IEEE et les réseaux locaux . . . 47

II Vue d'ensemble sur TCP/IP **51**

5 L'architecture TCP/IP **55**
5.1 Vue d'ensemble . . . 55
5.1.1 Historique . . . 55
5.1.2 Définition des standards . . . 57
5.1.3 Vue d'ensemble de l'architecture TCP/IP . . . 58
5.2 Protocoles de la suite TCP/IP . . . 59
5.2.1 Protocoles de la couche d'accès . . . 59
5.2.2 Protocoles de la couche réseau . . . 59
5.2.3 Protocoles de la couche de transport . . . 60
5.2.4 Les protocoles d'application . . . 60
5.3 Les adresses réseau Internet . . . 61
5.3.1 Adresse IP . . . 61
5.3.2 Multiplexage au niveau transport . . . 62

6 API des sockets : l'interface BSD **65**
6.1 Modèles de sockets . . . 66

6.1.1 Notion de socket 66
6.1.2 Types de communication 66
6.2 Paire de sockets locales 67
6.2.1 Création 67
6.2.2 Lecture et écriture sur des sockets 69
6.2.3 Fermeture des sockets 69
6.3 Socket pour communication connectée 70
6.3.1 Cycles de vie dans le cas des communications connectées 70
6.3.2 Création d'une socket 71
6.3.3 Spécification des adresses 72
6.3.4 L'ordre réseau des octets 73
6.3.5 Connexion au serveur 74
6.3.6 Initialisation d'un serveur 75
6.3.7 Attente de client 76
6.3.8 Acceptation de client 76
6.3.9 Ouverture d'un fichier de socket 77
6.3.10 Exemple de serveur 77
6.4 Données urgentes 78
6.4.1 Notion 78
6.4.2 Envoi et réception des données urgentes 79
6.5 Socket pour communication non connectée 80
6.5.1 Cycles de vie dans le cas des communications non connectées 80
6.5.2 Les fonctions d'envoi et de réception de message 80
6.5.3 Exemple 81
6.6 Semi-arrêt d'une socket 84

7 API des sockets : niveau avancé 85
7.1 Données dispersées 86
7.1.1 Les entrées-sorties sur socket 86
7.1.2 Vecteur d'entrée-sortie 86
7.1.3 Syntaxe des fonctions 86
7.1.4 Exemple 87
7.2 Données ancillaires 87
7.2.1 Notion 87
7.2.2 Les fonctions de transmission de messages 88
7.2.3 Structure des données ancillaires 89
7.3 Les options 91

8 Un exemple de carte réseau : 3Com 501 93
8.1 Description de la carte 94
8.1.1 Interface logicielle 94
8.1.2 Le registre de commande d'émission : TCR 95
8.1.3 Le registre de statut d'émission : TSR 95
8.1.4 Le registre de commande de réception : RCR 95
8.1.5 Le registre de statut de réception : RSR 96
8.1.6 Le registre de commande auxiliaire : ACR 96
8.1.7 Le registre de statut auxiliaire : ASR 97
8.2 Émission et réception 97
8.2.1 Choix de l'adresse de base et de l'IRQ 97
8.2.2 Émission d'une trame 98

8.2.3 Réception d'une trame 98

9 Quelques protocoles réseau 99
9.1 Protocole Ethernet 100
9.1.1 Protocole de sous-couche MAC pour Ethernet 100
9.1.2 Protocole 802.3 de la sous-couche LLC 102
9.2 Le protocole de couche réseau IPv4 102
9.2.1 Étude générale de la couche réseau 102
9.2.2 Ce que ne fait pas la couche réseau 102
9.2.3 Les tribulations d'un paquet IP 103
9.2.4 Fragmentation 104
9.2.5 Le routage 105
9.2.6 Routage et adresse IP 106
9.2.7 En-tête IPv4 108
9.3 Le protocole de couche de transport UDP 111
9.3.1 L'en-tête UDP 111
9.3.2 Sécurisation optionnelle par somme de contrôle 112

III Vue générale sur l'implémentation Linux 115

10 Implémentation générale de Linux 117
10.1 Organisation du code source 118
10.1.1 Premier niveau 118
10.1.2 Répertoire concernant les réseaux 118
10.1.3 Structure d'un fichier source 119
10.1.4 Aide au parcours du code source 121
10.2 Définition de types portables 121
10.2.1 Types Posix 121
10.2.2 Types entiers 123
10.2.3 Tailles 125
10.3 Gestion de la mémoire 125
10.3.1 Réservation et libération de mémoire noyau 125
10.3.2 Copie entre espace noyau et espace utilisateur 126
10.3.3 Caches mémoire 126
10.4 La gestion du temps dans le noyau Linux 127
10.4.1 Durée 127
10.4.2 File d'attente de minuteurs 128
10.5 SMP 128
10.6 Gestion des activités dans le noyau Linux 128
10.6.1 Activités de base 129
10.6.2 Parties basses et tasklets 130
10.6.3 Interruptions logicielles 131
10.7 Opérations atomiques 132
10.7.1 Opérations atomiques sur les bits 132
10.7.2 Opérations atomiques sur les entiers 133
10.7.3 Verrous rotatifs 134
10.7.4 Verrous rotatifs de lecture-écriture 136
10.7.5 Sémaphore 138
10.8 Modules 138

10.9 Initialisation du système Linux 139
10.9.1 Fonctions d'initialisation 139
10.9.2 Code d'initialisation 140

11 Vue d'ensemble **141**
11.1 Réception de données *via* un datagramme UDP 142
11.1.1 Réception de la trame Ethernet par la carte réseau 142
11.1.2 Récupération du paquet par l'ordinateur 142
11.1.3 Traitement du paquet par la couche réseau 143
11.1.4 Traitement du datagramme par la couche de transport 143
11.1.5 Récupération par l'utilisateur 143
11.2 Envoi de données *via* un datagramme UDP 143
11.2.1 Envoi par l'utilisateur 143
11.2.2 Traitement du datagramme par la couche de transport 144
11.2.3 Envoi des paquets ordinaires sous IPv4 144
11.2.4 Envoi de la trame 144
11.3 Historique 144

12 Les tampons de socket **147**
12.1 Structure d'un tampon de socket 148
12.1.1 Espace de tête et espace de queue d'un tampon de socket 148
12.1.2 Fragmentation d'un tampon de socket 148
12.2 Descripteur de tampon 149
12.2.1 Définition du type 149
12.2.2 Champs relatifs aux listes de descripteurs de tampon 152
12.2.3 Provenance du tampon 153
12.2.4 En-têtes des couches de protocole 153
12.2.5 Attributs divers 154
12.3 Files d'attente de tampons de socket 157

13 Gestion des tampons de socket **159**
13.1 Génération et libération des descripteurs de tampon 160
13.1.1 Sources 160
13.1.2 Allocation d'un tampon de socket 160
13.1.3 Libération rapide d'un tampon de socket 162
13.1.4 Libération propre d'un tampon de socket 164
13.2 Manipulation du tampon 166
13.2.1 Espaces de tête et de queue 166
13.2.2 Ajustement de la taille d'un tampon vide 167
13.2.3 Manipulation du compteur de références 170
13.2.4 Détection de clonage 170
13.3 Gestion des files d'attente de paquets 172
13.3.1 Caractères généraux 172
13.3.2 Gestion des structures de files d'attente 172
13.3.3 Gestion des tampons d'une file d'attente 173
13.4 Initialisation de l'antémémoire des tampons de socket 181
13.5 Fonctions de copie 181
13.5.1 Duplication d'un descripteur de tampon 181
13.5.2 Copie d'un tampon et de son descripteur 183
13.5.3 Copie du tampon sans linéarisation 188

13.5.4 Copie d'un tampon et de son descripteur avec modification 189
13.6 Ajustement d'une zone des données occupée 190
13.6.1 Retrait de données au début 190
13.6.2 Retrait à la fin 195
13.6.3 Extension de l'espace de tête 196
13.7 Allocation d'un tampon de socket pour l'émission 197

14 Structures de données pour le pilote 199
14.1 Descripteur d'interface de périphérique réseau 200
14.1.1 Philosophie Linux des fichiers de périphérique réseau 200
14.1.2 Définition du type 201
14.1.3 Partie visible 205
14.1.4 Champs généraux 209
14.1.5 Membres concernant la sous-couche MAC 214
14.1.6 Membres concernant la couche réseau 219
14.1.7 Méthodes du pilote de carte réseau 220
14.1.8 Autres champs 222
14.2 Entrée de cache d'en-tête matériel 222
14.3 Implémentation de l'ordre réseau des octets 224
14.3.1 Implémentation générique 224
14.3.2 Cas des microprocesseurs Intel 227

15 Détection d'une carte réseau (1) Aspect général 229
15.1 Première étape 230
15.1.1 Cas des périphériques liés statiquement 230
15.1.2 Cas des périphériques modularisés 232
15.2 Deuxième étape : allocation d'un nom 233
15.2.1 Allocation du nom et passage à la troisième étape 233
15.2.2 Manipulation d'un périphérique désigné par un nom 236
15.3 Attribution d'un index et insertion dans la liste 238
15.3.1 Chaînes de notification 239
15.3.2 Fonction principale 241
15.3.3 Allocation d'une trame de déroutement 245
15.3.4 Attribution d'un index 246
15.3.5 Rattachement au gestionnaire des tâches 247
15.3.6 Initialisation du minuteur 249
15.4 Mise en fonctionnement des périphériques réseau 249
15.4.1 Fonction principale 249
15.4.2 Montage du système de fichiers 251

16 Détection d'une carte réseau (2) Le cas d'Ethernet 255
16.1 Structures de données pour Ethernet 256
16.1.1 Constantes 256
16.1.2 Les types de protocole 256
16.1.3 Structure d'en-tête de trame 257
16.2 Détection et initialisation des cartes Ethernet 258
16.2.1 Détection des périphériques Ethernet 258
16.2.2 Initialisation d'une carte Ethernet 261
16.2.3 Allocation d'un descripteur de périphérique réseau 262

16.3 Opérations liées à Ethernet 264

17 Détection d'une carte réseau (3) Cas 3Com 501 **267**
17.1 Description du contrôleur de la carte 268
17.1.1 Commentaire général 268
17.1.2 Les registres 269
17.2 Détection et initialisation de la carte 3Com 271
17.2.1 Fonction principale 271
17.2.2 Établissement des paramètres au démarrage 273
17.2.3 Libération d'un descripteur de périphérique réseau 274
17.2.4 Vérification de la présence physique de la carte 274

18 Attribution d'une adresse à une carte réseau **279**
18.1 La commande `ifconfig` 280
18.1.1 Fonctionnalité 280
18.1.2 Liste des périphériques réseau en activité 280
18.1.3 Activation des périphériques du noyau 281
18.1.4 Activation et désactivation d'un périphérique modularisé 281
18.2 Aspect général de l'activation 281
18.2.1 Fonction d'ouverture 281
18.2.2 Activation de la file d'attente en émission 283
18.2.3 Démarrage du minuteur de problèmes d'émission 285
18.3 Cas de la carte 3Com 501 286
18.3.1 Installation du gestionnaire d'interruption 286
18.3.2 Réinitialisation 287

IV Réception **289**

19 Réception des trames **291**
19.1 Action du gestionnaire d'interruption 292
19.1.1 Démultiplexage : réception ou émission 292
19.1.2 Traitement en cas de réception 294
19.2 Création d'un nouveau tampon de socket 296
19.3 Détermination du protocole de couche réseau 298
19.4 Mise du nouveau tampon en file d'attente 300
19.4.1 File d'attente d'un microprocesseur 300
19.4.2 Fonction de mise en file d'attente 302
19.5 Passage à la couche supérieure 304
19.5.1 Type de paquet 304
19.5.2 Processus de passage à la couche supérieure 309
19.6 Préparation au traitement dans la couche supérieure 312
19.6.1 Détermination du paquet à traiter 312
19.6.2 Traitement d'un paquet 314
19.6.3 Livraison à la couche supérieure 317
19.7 Récupération des informations statistiques 318
19.7.1 Cas général 318
19.7.2 Cas de la carte 3Com 501 318

20 Implémentation Linux de IPv4 **319**

20.1 Implémentation Linux de l'en-tête IP 319
20.2 Calcul de la somme de contrôle IP 320

21 Tables de routage **323**
21.1 Remplissage des tables de routage 324
21.1.1 Calcul de la table de redirection 324
21.1.2 La commande `ip` 324
21.2 Structure des tables de routage 327
21.2.1 Tables de routage et sources Linux 327
21.2.2 Description de la structure des tables de routage 328
21.3 Initialisation des tables de routage statiques 332
21.3.1 Déclaration 332
21.3.2 Initialisation lors du démarrage du système 333
21.3.3 Numéros des tables 334
21.3.4 Attribution des paramètres par défaut 335
21.4 Insertion d'un élément dans une table 335
21.4.1 Structures de données pour l'insertion 336
21.4.2 Fonction interne d'insertion 340
21.5 Fonction interne de retrait d'une entrée 355
21.6 Consultation d'une table 357
21.6.1 Structure de données pour la consultation 357
21.6.2 Fonction interne de consultation 359
21.6.3 Interface avec les fonctions de redirection 362

22 Cache de routage **363**
22.1 Structure du cache de routage 364
22.1.1 Cache de destination 364
22.1.2 Entrée de cache de routage 367
22.1.3 Cache de routage 368
22.2 Initialisation du cache de routage IP 368
22.3 Interface cache de routage/fonctions de redirection 372
22.3.1 Calcul de l'index dans la table de routage 372
22.3.2 Insertion dans la table de hachage 372

23 Réception des paquets ordinaires sous IPv4 **377**
23.1 Tri et contrôle de l'intégrité du paquet 378
23.1.1 Tri et contrôle 378
23.1.2 Étape optionnelle : les points d'ancrage 381
23.2 Deuxième étape : routage et traitement des options 382
23.2.1 Vue d'ensemble 382
23.2.2 Fonction de redirection en entrée 384
23.2.3 Choix de la fonction de traitement du paquet 397
23.3 Remise locale des paquets 397
23.3.1 Première étape : réassemblage des fragments 397
23.3.2 Gestion des protocoles de transport dans la couche IP 398
23.3.3 Deuxième étape : démultiplexage de la couche de transport 399

24 Les descripteurs de couche transport **403**
24.1 Structure des descripteurs de couche transport 404
24.1.1 Définition du type 404

24.1.2 Attributs communs 406
24.1.3 Attributs de contrôle 408
24.1.4 Attributs associés aux options des sockets 410
24.1.5 Fonctions membre 411
24.2 Allocation et libération 411
24.2.1 Allocation 411
24.2.2 Gestion du compteur de référence 413
24.2.3 Libération 414
24.3 Gestion des données associées 416
24.3.1 Initialisation des données 416
24.3.2 Verrouillage 418
24.3.3 Initialisation du délai 419

25 Réception des datagrammes UDP **421**
25.1 Implémentation Linux générale d'UDP 421
25.1.1 Implémentation Linux de l'en-tête UDP 421
25.1.2 Calcul de la somme de contrôle 422
25.2 Réception des datagrammes UDP 428
25.2.1 Fonction de traitement principale 428
25.2.2 Vérification rapide de la somme de contrôle 431
25.2.3 Consultation de la table de hachage UDP 433
25.2.4 Traitement du datagramme UDP 436
25.2.5 Mise dans la file d'attente de réception UDP 438

26 Installation de la famille de protocoles IPv4 **441**
26.1 Manipulation des familles de protocoles 442
26.1.1 Familles de protocoles 442
26.1.2 Les types de communication 444
26.1.3 Les protocoles de la suite TCP/IPv4 449
26.2 Implémentation 449
26.2.1 Tableau des familles de protocoles 449
26.2.2 Enregistrement d'un type de communication IPv4 452
26.2.3 Installation de la suite de protocoles TCP/IPv4 453

27 Implémentation des fichiers de type socket **459**
27.1 Implémentation générale des fichiers 460
27.1.1 Système de fichiers virtuel 460
27.1.2 Super-bloc 461
27.1.3 Nœud d'information 463
27.1.4 Descripteur de fichier 465
27.1.5 Répertoire 466
27.1.6 Types de fichiers 466
27.1.7 Déclaration d'un système de fichiers 467
27.1.8 Enregistrement d'un système de fichiers 467
27.2 Descripteur de socket 468
27.2.1 Définition du type 468
27.2.2 Les états d'une socket 469
27.2.3 Les drapeaux 469
27.2.4 Le type opérations sur une socket 469
27.3 Déclaration du système de fichiers des sockets 470

27.3.1 La structure . . . 470
27.3.2 Enregistrement . . . 470
27.4 Système de fichiers de sockets . . . 471
27.4.1 Obtention du super-bloc . . . 471
27.4.2 Libération d'un super-bloc . . . 474
27.5 Opérations sur les super-blocs . . . 475
27.5.1 Ensemble des opérations . . . 475
27.5.2 Statut du système de fichiers . . . 475
27.5.3 Allocation d'un descripteur de nœud d'information . . . 476
27.5.4 Libération d'un descripteur de nœud d'information . . . 477
27.6 Opérations sur les répertoires de socket . . . 477
27.7 Opérations sur les fichiers de type socket . . . 477
27.8 Premières implémentations de fonctions . . . 479
27.8.1 Implémentation du positionnement . . . 479
27.8.2 Implémentation de la scrutation . . . 479
27.8.3 Implémentation du non mappage en mémoire vive . . . 480
27.8.4 Implémentation de l'ouverture de fichier . . . 481
27.8.5 Implémentation des événements asynchrones . . . 481
27.8.6 Implémentation de la libération de fichier . . . 482

28 Création et fermeture des sockets 485
28.1 Création d'une socket . . . 486
28.1.1 Traitement général . . . 486
28.1.2 Cas de la famille de protocoles IPv4 . . . 495
28.2 Semi-arrêt d'une socket . . . 500
28.2.1 Traitement général . . . 500
28.2.2 Cas de IPv4 . . . 502
28.2.3 Fonction de déconnexion spécifique à UDP . . . 503
28.3 Fermeture d'une socket . . . 506

29 Initialisation d'un serveur 509
29.1 Partie générale . . . 510
29.1.1 Fonction d'appel . . . 510
29.1.2 Passage espace utilisateur/espace noyau . . . 510
29.2 Cas de IPv4 . . . 511
29.2.1 Fonction spécifique d'initialisation du serveur . . . 511
29.2.2 Détermination du type d'adresse . . . 513
29.3 Obtention et vérification du port dans le cas UDP . . . 514

30 Réception d'un datagramme par une socket 519
30.1 Types liés aux fonctions d'entrée-sortie . . . 520
30.1.1 Vecteurs d'entrée-sortie . . . 520
30.1.2 En-tête de message . . . 520
30.1.3 En-tête de contrôle . . . 520
30.2 Implémentation générale de la réception des datagrammes . . . 521
30.2.1 Fonction d'appel . . . 521
30.2.2 Réception de message de socket . . . 522
30.2.3 Passage espace noyau/espace utilisateur . . . 523
30.2.4 Fonction interne de réception d'un message . . . 524
30.3 Cas de IPv4 . . . 525

30.4 Cas de UDP 526
30.5 Lien avec les tampons de socket 529
30.5.1 Instantiation de descripteur de tampon en réception 529
30.5.2 Copie d'un datagramme dans l'espace utilisateur 533
30.5.3 Libération d'un tampon de datagramme 537

V Envoi **539**

31 Envoi d'un datagramme par une socket **541**
31.1 Niveau implémentation des sockets 542
31.1.1 Fonction d'appel 542
31.1.2 Envoi de message de socket 543
31.1.3 Fonction interne d'envoi de message 543
31.2 Cas de IPv4 544
31.2.1 Appel à la fonction spécifique à la couche de transport 544
31.2.2 Attribution automatique de numéro de port 545

32 Envoi de datagrammes ordinaires sous UDP **547**
32.1 Fonction d'envoi spécifique à UDP 548
32.1.1 Description 548
32.1.2 Implémentation 548
32.2 Regroupement de données UDP 555
32.2.1 Fonction principale 555
32.2.2 Instantiation d'un tampon de socket pour l'émission 561
32.2.3 Récupération des fragments 564
32.3 Constitution du datagramme 566
32.4 Envoi du datagramme à la couche réseau 569

33 Envoi de paquets ordinaires sous IPv4 **571**
33.1 Première étape : routage d'un paquet ordinaire 572
33.1.1 Recherche dans le cache de routage 572
33.1.2 Recherche étendue à la FIB 573
33.2 Seconde étape : constitution du paquet IP 581
33.2.1 Implémentation des options IP 581
33.2.2 Récupération des fragments 581
33.3 Transfert de la couche réseau à la couche inférieure 585
33.3.1 Première sous-étape : détermination de la fonction de transfert 585
33.3.2 Deuxième sous-étape : fragmentation éventuelle 586
33.3.3 Troisième sous-étape : positionnement du type du paquet 586
33.3.4 Quatrième sous-étape : passage à la couche inférieure 586

34 Envoi de trames **589**
34.1 Transmission à la carte réseau : partie générale 590
34.1.1 Première étape : mise en file d'attente 590
34.1.2 Deuxième étape : récupération des paquets 596
34.1.3 Troisième étape : envoi 601
34.2 Partie spécifique à la carte 603
34.2.1 Étude générale 603
34.2.2 Cas de la carte 3Com 501 : Mise en file d'attente 604

34.3 Réaction à l'envoi 607
34.3.1 Réponse du gestionnaire d'interruption 607
34.3.2 Action lorsque le délai est écoulé 609

35 Désactivation et retrait **611**
35.1 Désactivation 612
35.1.1 Cas général 612
35.1.2 Cas de la carte 3Com 501 614
35.2 Retrait d'un périphérique réseau 615
35.2.1 Démontage du système de fichier 619
35.2.2 Démontage du système de fichiers 619
35.2.3 Attente des références 619

VI Complément sur IP : la fragmentation **621**

36 Fragmentation pour la couche réseau : le cas de IPv4 **623**
36.1 Fragmentation 624
36.2 Réassemblage 632
36.2.1 Cache de fragment 632
36.2.2 Traitement du réassemblage 634
36.2.3 Traitement lors de l'expiration du délai 643
36.3 Initialisation de la fragmentation 645

VII Appendices **647**

Bibliographie **649**

Index **654**

Table des figures

1 Hiérarchie entre les chapitres vi

1.1 Graphe complet 3
1.2 Réseau avec commutateur central 4
1.3 Réseau distribué 6
1.4 Pile réseau 8

2.1 Niveaux logiques 23
2.2 Synchronisation 23
2.3 Codage Manchester 24
2.4 Réseau étendu 26
2.5 Réseau machine à machine 28
2.6 Réseau en bus 29
2.7 Réseau en anneau 29
2.8 Réseau en étoile 30
2.9 Croissance d'Arpanet : (a) décembre 1969 (b) juillet 1970 (c) mars 1971 (d) avril 1972 (e) septembre 1972 31
2.10 Èpine dorsale NSFNET en 1988 32
2.11 Architecture du premier réseau Ethernet 33

3.1 Structure d'une trame 36
3.2 Décomptage des octets 37
3.3 Fanion de signalisation 38
3.4 Confirmation et infirmation 39
3.5 Fenêtre glissante 41

4.1 Types de câblages Ethernet 46
4.2 Types de connexions Ethernet 47
4.3 Sous-couches MAC et LLC 49

5.1 Numéros de port bien connus des serveurs 63

7.1 Données ancillaires 89

9.1 Fragmentation d'un paquet IP 105
9.2 Table de hachage 106
9.3 Pseudo en-tête UDP 112

12.1 Tampon de socket et son descripteur 150
12.2 File d'attente de paquets 157

19.1 Gestion des protocoles de réseau 306
21.1 Table de routage 328
22.1 Structure du cache de routage 368
36.1 Cache de fragment 632

Chapitre 1

Les réseaux informatiques

1.1 Introduction aux réseaux

1.1.1 Origine des réseaux informatique

La notion de calcul est l'une des grandes conquêtes de l'humanité, certainement apparue au néolithique pour les besoins de comptabilité des ressources indispensables pour les premières cités, concernant le cheptel et les réserves de nourriture. Les grandes quantités manipulées pour cette comptabilité a conduit à rechercher rapidement des outils d'aides au calcul (petits cailloux, puis abaque, puis machine arithmétique de Pascal...). La notion d'ordinateur, outil universel d'aide aux calculs dans la mesure où, par définition, un ordinateur est capable de calculer ce qui est calculable par n'importe quel outil, date de 1936 avec la définition par Alan TURING de sa célèbre machine (en fait un modèle, car il s'agit d'une machine imaginaire). La réalisation effective d'ordinateurs attendra encore quelques années, puisque le premier date de 1949[1].

Quelques années après, il fut envisagé de partager l'unité centrale d'un ordinateur entre plusieurs sites, soit pour des raisons de bonne économie, soit pour des raisons intrinsèquement liées à l'application envisagée (projet SAGE [pour *Semi-Automatic Ground Environment*, environnement au sol semi-automatique] de surveillance radar des États-Unis). L'alliance des télécommunications et de l'informatique était née.

Au milieu des années 1960 vint l'idée de relier les unités centrales elles-mêmes.

1. Ce que l'on vend sous le nom d'ordinateur n'est pas réellement une machine universelle. Mathématiquement ce sont des automates finis à un très grand nombre d'état et non des machines de Turing.

1.1.2 Notion de réseau informatique

Un **réseau informatiques**[2] (*computer network* en anglais) est un ensemble d'ordinateurs et de périphériques (imprimantes, traceurs, scanners, etc.) connectés ensemble par le biais d'un support physique (en anglais *medium*). La connexion peut être directe (par câble coaxial, par exemple) ou indirecte (par modem).

1.1.3 Matériel et logiciel pour réseau

Comme pour les systèmes informatiques[3], la distinction entre *matériel* et *logiciel* est importante dans le cas des réseaux.

L'objet de ce livre concerne le logiciel et non le matériel. Le matériel réseau nous apparaîtra comme une boîte noire fonctionnant parfaitement. Nous ne dirons du matériel que ce qui en est strictement nécessaire pour comprendre le logiciel associé.

1.1.4 Protocoles

1.1.4.1 Notion de protocole

Dans un réseau informatique, lorsqu'un ordinateur envoie des informations à un autre, les matériels et les logiciels sont en général différents. On a donc besoin d'un ensemble de règles pour coordonner l'échange de ces informations. Celles-ci forment un **protocole**, nom donné par Tom Marill en 1966 ([HL–96], p.83).

Le mot provient évidemment du langage diplomatique. Les diplomates suivent des règles lors des discussions entre nations, appelées un protocole. Le protocole diplomatique indique qu'il ne faut pas insulter ses hôtes et qu'il faut tâcher de respecter les coutumes locales. La plupart des ambassades et des consulats abritent des spécialistes du protocole, dont le rôle est de s'assurer que tout se passe harmonieusement lors des rencontres. Le protocole est un ensemble de règles qui doivent être suivies pour "jouer le jeu", pour reprendre une expression diplomatique.

1.1.4.2 Suite de protocoles

Les protocoles ont évolué. De processus très simples ("je t'envoie un caractère, tu me le renvoies, et je m'assure que les deux correspondent") au départ, ils sont devenus des mécanismes complexes qui prennent en compte tous les problèmes et conditions de transfert possibles. Un protocole unique qui couvrirait tous les aspects du transfert serait de taille trop importante, difficile d'emploi et trop spécialisé. Plusieurs protocoles ont donc été développés, chacun s'acquittant d'une tâche spécifique.

On appelle **suite de protocoles** un ensemble de protocoles cohérents qui couvre pratiquement tous les besoins de communication.

1.1.5 Systèmes propriétaires et systèmes ouverts

Au début, il n'y avait qu'un réseau (ARPANET) avec ses protocoles associés. Devant le succès de celui-ci, des produits commerciaux furent développés, plus particulièrement pour les réseaux

2. Désormais nous ne parlerons que de *réseau* au lieu de *réseau informatique*. L'histoire des premiers réseaux de télécommunication (premiers essais, télégraphe optique, télégraphe électrique, télégraphie d'images, téléphone, transmission radio et systèmes de commutation) est contée dans [HUU–03].

3. On appelle **système informatique** un poste de travail informatique complet : ordinateur, bien sûr, mais aussi périphériques (tels que imprimante ou scanner) et logiciels. De nos jours, les outils réseau font partie intégrante du système informatique, dans un sens plus large.

locaux et étendus. Chaque constructeur avait sa ligne de produits, matériels comme logiciels. On parle de **système propriétaire**. Les sources, et même les spécifications complètes, des systèmes propriétaires sont rarement diffusées. Ceci présente une difficulté pour la conception des interréseaux.

Par opposition, un **système ouvert** est un système dont au moins les spécifications complètes sont diffusées, éventuellement les sources des logiciels.

Le mouvement des systèmes ouverts n'est pas né à propos des réseaux, même si c'est là qu'il a connu son plus grand succès puisqu'il n'existe plus de système propriétaire, mais à propos des systèmes d'exploitation. Jusqu'aux années 1980, chaque constructeur de matériel avait sa ligne de produits, et on était dans une très large mesure dépendant de ce constructeur pour tout ce qui concernait le logiciel et le matériel. Certains constructeurs profitaient de cette situation pour pratiquer des tarifs prohibitifs ou pour imposer certaines configurations à leurs clients. Le ressentiment des clients prit une telle ampleur que ces derniers finirent par imposer leurs souhaits. IBM commença par diffuser le code du BIOS de ses micro-ordinateurs. DEC (*Digital Equipment Corporation*) passa d'un système d'exploitation propriétaire, VMS, sur ses mini-ordinateurs à un système d'exploitation ouvert de type UNIX. Ses clients lui en furent gré et l'entreprise vendit plus de machines. Microsoft essaie de faire passer sa technologie .NET comme un standard ECMA (*European Computer Manufacturer Association*), sans totalement y réussir pour l'instant (seule la spécification du langage C# est définie par un standard ECMA pour l'instant, début 2005).

1.2 Réseau et commutation

Les éléments d'un réseau de télécommunication doivent être reliés entre eux et ceci sur une distance quelquefois très éloignée. Comment réaliser ceci?

1.2.1 Notion de commutation

En 1876, peu de temps après qu'Alexander Graham Bell eut déposé son brevet d'invention du téléphone, il y eut une vague énorme de demandes. Au début, les téléphones étaient vendus par paire et il appartenait au client de les relier à l'aide d'un fil électrique (la voie du retour se faisait par la terre). Si une personne souhaitait converser avec n autres interlocuteurs, il lui fallait être raccordé par autant de fils aux n domiciles de ces derniers. On aboutit à ce qu'on appelle un graphe complet, représenté à la figure 1.1.

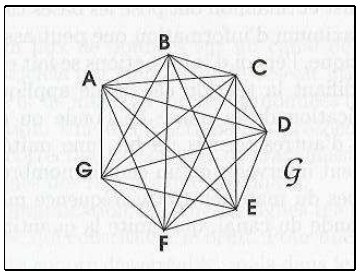


FIG. 1.1 – *Graphe complet*

En l'espace d'un an les villes se trouvèrent prises dans un enchevêtrement sauvage de fils passant par-dessus les toits et les arbres. Il devint très vite clair qu'un modèle d'interconnexion où chaque téléphone était relié à tous les autres n'était pas viable.

Bell réagit à cette situation en fondant la société Bell Telephone Company, qui créa la notion de **central téléphonique** et mit le premier central en place à New Heaven (Connecticut) en 1878. La société amena un fil électrique à chaque domicile ou bureau d'*abonné*. Pour effectuer un appel, le client devait tourner une manivelle qui produisait une sonnerie au niveau du central téléphonique, attirant l'attention d'une opératrice. Celle-ci se chargeait ensuite de raccorder manuellement la ligne de l'appelant à celle du correspondant appelé au moyen d'un câble de jonction sur un *tableau de commutation*. Ce modèle centralisé est illustré à la figure 1.2.

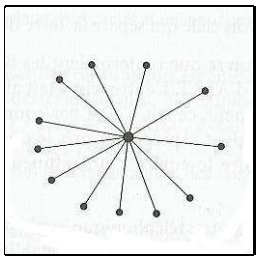


FIG. 1.2 – *Réseau avec commutateur central*

Très rapidement d'autres centraux furent créés un peu partout. Et comme les clients souhaitaient pouvoir appeler des correspondants dans d'autres villes, il a fallu interconnecter ces centraux. Ce fut, là encore, d'abord un graphe complet, puis des centraux de second niveau et ainsi de suite. Il y eut jusqu'à cinq niveaux de centraux.

1.2.2 Types de commutation

Les divers réseaux de télécommunication ont conduit à plusieurs types de commutation, comme nous allons le voir.

1.2.2.1 Commutation de circuits

Si d'un poste téléphonique de New-York vous demandez le 22 à Asnières à une opératrice, celle-ci contacte le central interurbain, qui va contacter le central reliant le câble transatlantique à un central en Europe, qui contacte un central en France, qui contacte un central dans la région parisienne, qui contacte le numéro demandé. Par un jeu de fiches dans chacun de ces six centraux, un chemin physique est établi de l'appelant vers l'appelé. Un tel chemin est appelé un **circuit** dans le jargon de la téléphonie (même s'il ne s'agit pas d'un circuit au sens de la théorie des graphes). On parlera plus tard de **commutation de circuits** (*circuit switching* en anglais) pour ce type de commutation lorsque d'autres types de commutation apparaîtront. Une ligne physique est dédiée à cette communication durant tout le temps de la communication, d'où le paiement à la durée de la communication.

Au début de l'ère de la téléphonie, la connexion physique était établie manuellement comme nous venons de le rappeler. On parlera plus tard de **commutation manuelle**. Peu après l'invention du téléphone, un équipement automatique de commutation électro-mécanique fut inventé par Almon Strowger et fut utilisé pendant près de cent ans. Au début des années 1970, il fut remplacé par un équipement électronique, mais le principe est le même.

Le gros avantage de la communication de circuits est qu'une ligne est entièrement dédiée à la communication. Elle présente cependant également trois gros inconvénients :

– Le temps d'établissement d'une ligne, en passant à travers tous les centraux, n'est pas

négligeable. Ce qui est acceptable pour une communication téléphonique le serait moins pour un réseau informatique.
- Il peut arriver que toutes les liaisons d'un central à l'autre soient occupées. On vous demande alors de renouveler votre appel ultérieurement.
- Le débit n'est pas utilisé de façon optimale. Lors des silences dans la communication téléphonique, qui peuvent parfois être longs lorsqu'un des correspondants recherche quelque chose, le circuit est toujours dédié à cette communication.

1.2.2.2 Commutation de messages

Une autre technique de commutation est la **commutation de messages** (*message switching* en anglais). Lorsqu'elle est mise en œuvre, aucun chemin physique n'est établi au préalable entre l'émetteur et le récepteur. Lorsque l'émetteur envoie un bloc de données, celui-ci est stocké dans le premier central de commutation (appelé **routeur** dans ce cas), contrôlé pour vérifier qu'il ne comporte pas d'erreurs puis transmis au routeur suivant. On parle de **transmission différée** (*store-and-forward* en anglais).

Cette technique de commutation fut mise en œuvre dans les premiers systèmes électromécaniques de télécommunications pour transmettre les télégrammes.

L'énorme avantage est que le débit est utilisé au mieux et que le temps d'établissement du circuit n'existe plus. Cependant les stockages successifs entraînent des délais qui empêcheraient l'utilisation d'une telle commutation pour la voix, par exemple.

1.2.2.3 Commutation par paquets

Dans la commutation de messages, la taille des blocs n'est pas limitée. Cela a pour conséquence que les routeurs doivent disposer de disques pour placer en mémoire tampon les longs blocs. Un seul bloc peut monopoliser une ligne entre deux routeurs pendant des minutes. Cette technique de commutation est donc inadaptée au trafic interactif. De plus, si un gros message a été altéré, il est totalement perdu ou il faut le renvoyer intégralement.C'est pourquoi la **commutation par paquets** (*packet switching* en anglais) a été developpée.

La commutation de paquets est analogue à la commutation de messages mais les messages sont découpés en paquets d'une taille maximale, ce qui évite les disques tampon et la congestion durant plusieurs minutes due à un gros message.

Les paquets empruntent éventuellement des chemins différents, ce qui peut entraîner des arrivées désordonnées. On numérote donc les paquets.

Le premier paquet d'un message qui en comporte plusieurs peut être transmis par un routeur avant que le deuxième paquet ne soit complètement arrivé, ce qui réduit le délai et améliore le débit.

La commutation de paquets offre une meilleure tolérance aux pannes que la commutation de circuits. Si un routeur devient indisponible, les paquets peuvent le contourner et poursuivre leur chemin.

1.2.2.4 Cas des réseaux informatiques

Dans le cas des réseaux informatique, on utilise le plus souvent la commutation de paquets (c'est le cas pour Internet) et plus rarement la commutation de circuits (c'est le cas de ATM), mais jamais la commutation de messages.

On appelle **routage** l'algorithme utilisé par un routeur pour déterminer l'ordinateur ou le routeur auquel il faut envoyer le paquet qu'il vient de recevoir pour que celui-ci arrive à destination au vu de l'adresse réseau.

1.2.3 Réseaux hiérarchisés et réseaux maillés

Les réseaux informatiques se distinguent des autres réseaux de télécommunication dans la mesure où ils sont maillés et non hiérarchisés. L'utilisation de réseaux distribués au lieu des réseaux hiérarchisés utilisés en téléphonie fut prônée par Paul Baran avant de devenir réalité lors de la réalisation des réseaux informatiques.

Paul Baran s'intéressa dès 1960 à la capacité de survie des systèmes de communication en cas d'attaque nucléaire. À l'époque, les réseaux de communication à longue distance étaient extrèmement vulnérables et hors d'état de supporter une attaque nucléaire. Pourtant le pouvoir qu'avait le président des États-Unis de commander ou d'annuler l'envoi de missiles nucléaires reposait sur ces systèmes de communications vulnérables. Baran a été l'un des premiers à établir, au moins de façon théorique, que le problème pouvait être résolu. Il était arrivé à l'idée qu'un réseau de transmission de données pouvait être rendu plus robuste et plus fiable en introduisant des niveaux de redondance élévés. Le mécanisme de défense qui consiste à diviser une vaste structure, unique et vulnérable, en de nombreuses parties, se retrouve dans maintes applications, comme le compartimentage de sécurité des navires. Théoriquement, il était possible d'installer un réseau avec de nombreuses connexions redondantes. Mais il y avait une restriction technique : tous les signaux sur le réseau téléphonique étaient analogiques. Le plan d'acheminement sur ce réseau interdisait d'utiliser plus de cinq liaisons les unes à la suite des autres à cause de l'altération du signal. Baran proposa donc un agencement en **réseau réparti** (dit aussi **distribué**) comme le montre la figure 1.3 [BAR–60, BAR–64].

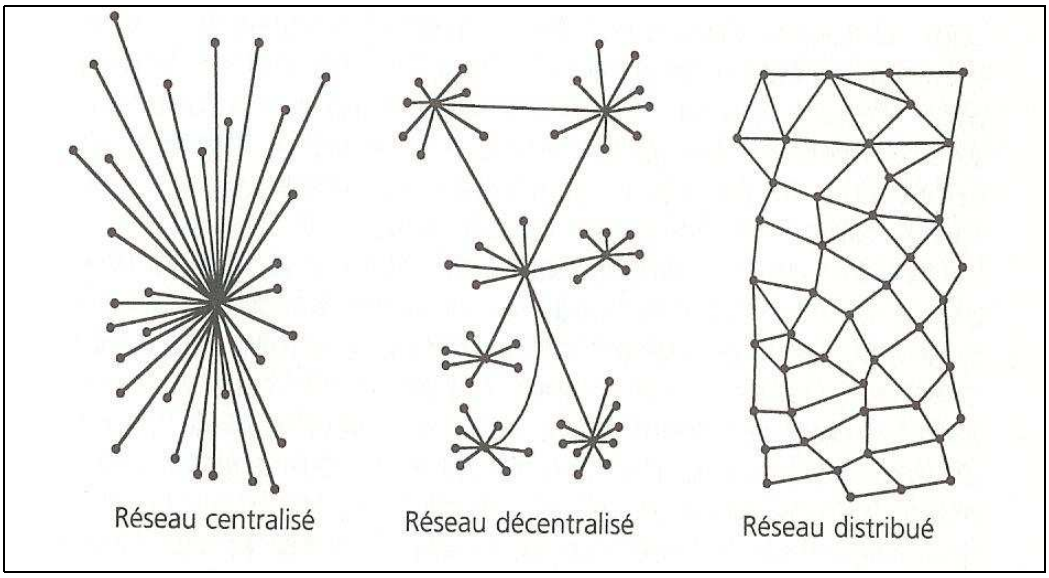


FIG. 1.3 – *Réseau distribué*

Néanmoins une question demeurait, nous faisant passer du qualitatif au quantitatif : quel degré de redondance fallait-il introduire dans les connexions entre nœuds voisins pour garantir la persistance du réseau ? Baran effectua de nombreuses simulations. Il en conclut qu'il suffirait d'un faible niveau de redondance : que chaque nœud soit connecté à trois ou quatre autres permet d'offrir un niveau exceptionnelement élévé de résistance et de fiabilité (voir [HL–96], pp.64–72).

1.3 Modèles en couches

La partie logicielle des réseaux comprend un grand nombre de fonctions, chacune relevant d'un protocole. Un ensemble de protocoles cohérent couvrant l'ensemble des besoins pour un réseau s'appelle une **suite** de protocoles. L'ensemble d'une telle suite s'appelle un **modèle réseau** ou **architecture réseau.**

1.3.1 Étude générale

1.3.1.1 Notion

Imaginez que vous deviez écrire un programme qui fournisse des fonctions de réseau à toutes les machines de votre réseau local. L'écriture d'un logiciel unique se chargeant de toutes les tâches nécessaires à la communication entre plusieurs ordinateurs serait un vrai cauchemar. De plus, dans l'hypothèse où vous arriveriez à gérer tous les matériels présents sur le réseau, le programme résultant serait bien trop grand pour être maintenu ou même exécuté.

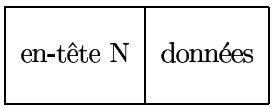
Il est plus raisonnable de diviser chaque domaine d'opérations en groupes de natures proches. Les groupes sont assez évidents à discerner. Un groupe traite du transport des données, un autre de l'empaquetage des messages, un autre des applications utilisateur, etc. Chaque groupe de tâches proches est appelé une **couche**. On obtient ainsi un **modèle en couches.**

Les couches d'une architecture réseau sont censées être des entités autonomes et indépendantes. Une couche ne peut évidemment pas effectuer une tâche observable sans interagir avec d'autres couches mais, du point de vue de la programmation, elles sont indépendantes. Elles ne doivent interagir les unes avec les autres que grâce à leur **interface** proprement définie.

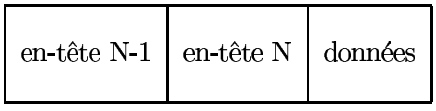
1.3.1.2 Pile réseau

Dans un modèle en couches, les couches sont numérotées de 1 à N, allant du niveau le plus proche du matériel (concernant le port série, puis la carte réseau) au niveau le plus proche de l'application de l'utilisateur (courrier électronique, transfert de fichier...). Plus on est proche du matériel, plus le numéro de couche est bas.

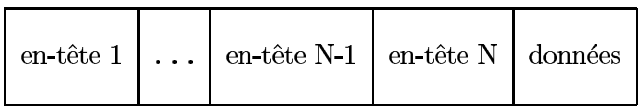
Chaque protocole encapsule les données dans un ensemble plus grand comprenant en général un **en-tête** et quelquefois un **suffixe.** Supposons qu'il n'y ait pas de suffixe. Les données de l'utilisateur sont encapsulées avec l'en-tête du niveau N :



Cela devient des données de niveau N qui sont encapsulées avec l'en-tête de niveau N-1 pour obtenir des données de niveau N-1 :



et ainsi de suite jusqu'aux données encapsulées de niveau 1 :



L'intérêt des modèles en couche est, qu'à chaque niveau, il n'est pas besoin de passer en revue l'ensemble des en-têtes mais uniquement celui du niveau correspondant.

On parle aussi de **pile réseau**, *stack* en anglais, car on peut considérer que les en-têtes des niveaux N à 1 sont empilés au-dessus des données de base lors de l'expédition et qu'elles sont dépilées lors de la réception, comme le montre la figure 1.3 ([K-R–01], p.52).

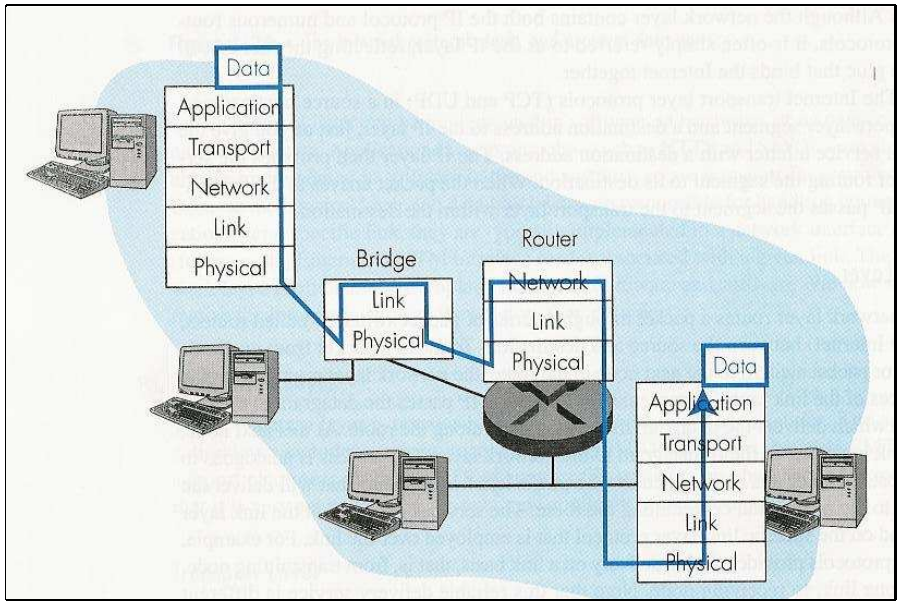


FIG. 1.4 – *Pile réseau*

1.3.1.3 Intérêts et réalisations

Les modèles en couches présentent beaucoup d'intérêt. La suite de protocoles est beaucoup plus simple à concevoir : le protocole d'un niveau donné peut être conçu par une équipe différente du protocole d'un autre niveau. Il en est de même de la suite de logiciels mettant en place ce modèle. De plus, les matériels actifs du réseau n'ont à considérer que les niveaux les plus bas, par exemple :

- un répéteur ne considérera que les données brutes, sans même entrer dans le détail, autrement dit ne concernera que la couche physqiue ;
- un routeur ne considèrera que les en-têtes des couches les plus basses pour obtenir l'adresse de destination.

Deux modèles furent développés pratiquement en parallèle : le **modèle OSI** (par l'organisme de standardisation international ISO) et la **suite TCP/IP** (d'après le nom de deux des protocoles de la suite). Rétrospectivement on s'aperçoit qu'un modèle est largement dominant, pour ne pas dire exclusif : la suite TCP/IP. Il est quand même intéressant de dire quelques mots du modèle OSI, même s'il est désormais pratiquement abandonné, car les deux couches les plus basses ne sont pas prises en considération par TCP/IP, ce qui a conduit à un modèle mixte.

1.3.2 Modèle OSI

L'ISO (*International Standardisation Organisation*), l'organisation de standardisation la plus prioritaire dans le monde entier, fondée en 1947, a proposé un modèle en sept couches en 1984 [ISO 7498-1], appelé **modèle OSI**, en fait **OSI–RM** pour *Open Systems Interconnection Reference Model* (modèle de référence d'interconnection des systèmes ouverts) :

7	Application	Couches supérieures
6	Signification	
5	Session	
4	Transport	Couches inférieures
3	Réseau	
2	Liaison des données	
1	Physique	

Le modèle OSI ne décrit aucune implémentation réelle d'un système particulier, mais se contente de définir les tâches des différentes couches.

Les couches d'application, de présentation et de session sont toutes les trois orientées application, c'est-à-dire qu'elles présentent l'interface de l'application à l'utilisateur. Ces trois couches sont totalement indépendantes des couches situées sous elles, elles ne connaissent rien de la façon dont les données parviennent à l'application. Elles sont appelées **couches supérieures.**

Les quatre **couches inférieures** se chargent de la transmission des données, en gérant l'empaquetage, le routage, la vérification et la transmission de chaque ensemble de données. Elles ne font aucune différence entre les diverses applications.

Détaillons maintenant le rôle de chacune des sept couches :

- La **couche physique** (*physical layer* en anglais) contrôle la transmission des différents bits *via* un support physique (*media* en anglais). C'est dans cette couche qu'on s'occupe de la façon dont les suites de bits sont converties (sans structuration) en signaux physiques et transmises *via* un support physique (câble de cuivre, fibre de verre, radio, etc.). La couche physique définit les procédures de codage physique (telle ou telle différence de potentiel par exemple), la géométrie des connecteurs enfichables et les types des supports spéciaux. Les protocoles de cette couche dépendent du support physique.
- La **couche de liaison des données** (*data link layer* en anglais) est chargée de la transmission correcte des données d'un point du réseau à un autre relié directement à lui *via* un support physique. Elle s'occupe de la correction des erreurs survenant lors de cette transmission (différentes des erreurs dans les données elles-mêmes, traitées dans la couche de transport). Elle doit tenir compte des interférences de signal (fréquentes, pouvant provenir de plusieurs sources, parmi lesquelles les rayons cosmiques et les interférences magnétiques provenant d'autres équipements).
- La **couche réseau** (*network layer* en anglais) est chargée de la transmission des données d'un point du réseau à un autre, que celui-ci soit relié directement à lui ou non. Il s'agit d'abord de déterminer un chemin (ou **route**, comme on préfère l'appeler dans le vocabulaire des réseaux) de l'expéditeur vers le destinataire en utilisant des machines intermédiaires : on parle du **routage** physique des données. Elle est chargée également de l'adaptation des unités de données à la taille admissible de la couche liaison existante : on parle de **fragmentation.**
- La **couche transport** (*transport layer* en anglais) se charge du multiplexage (lors de l'envoi) et du démultiplexage (lors de la réception), c'est-à-dire de distribuer aux différentes applications les paquets arrivés sans encombre.

Il y a **multiplexage** lorsque plusieurs communications transitent par un support physique unique. Le **démultiplexage** est l'inverse du multiplexage. Le multiplexage est indispensable pour prendre en charge de nombreuses connexions simultanément, tout en ne disposant que de ressources limitées. Un exemple classique est un bureau distant comprenant vingt terminaux. Chaque terminal pourrait être connecté au bureau principal par le biais d'une ligne téléphonique dédiée. Au lieu d'utiliser vingt lignes, on peut aussi s'arranger pour multiplexer les connexions afin de n'utiliser que trois ou quatre lignes téléphoniques.

Elle peut, de plus, assurer d'autres tâches. Elle peut établir, maintenir et terminer les communications entre deux machines (dans le cas des **communications** dites **connectées**). Elle peut se charge d'assurer que les données envoyées correspondent aux données reçues, tout au moins modulo certains points de comparaison et, si elles ne correspondent pas, demander à ce que les données soient envoyées à nouveau. Elle peut gèrer l'envoi des données, en déterminant l'ordre et la priorité de l'envoi.

- La **couche de session** (*session layer* en anglais) contrôle l'échange structuré de dialogues *via* les liaisons de communication. Il est, par exemple, possible de contrôler au cours d'une session si le transfert des données peut avoir lieu simultanément dans les deux sens ou si un seul partenaire à la fois dispose du droit d'émission. Dans ce dernier cas, la couche session gère le droit d'émission. Elle opère avec la couche d'application pour fournir des ensembles de données simples, appelés **points de synchronisation**, qui permettent à l'application de connaître l'état de progression de la transmission et de la réception des données. Il s'agit donc d'une couche de temporisation et de contrôle des flux.
- La **couche de présentation** (*presentation layer* en anglais) contrôle la présentation des données à transmettre sous une forme ne dépendant pas des systèmes. Elle convertit les données de l'application en un format commun, souvent appelé la **représentation canonique**, par exemple pour éviter le problème du code (Unicode, ASCII ou EBDIC) de représentation des caractères, le problème du format (petit-boutien ou grand-boutien) des entiers ou la façon d'indiquer le passage à la ligne.

  Le seul problème qui relève de cette couche que nous aborderons sera celui concernant le format (petit-boutien ou grand-boutien).
- La **couche d'application** (*application layer* en anglais) est l'interface utilisateur vers le système OSI. C'est là que résident les applications telles que le courrier électronique. La tâche de la couche d'application est d'afficher les informations reçues et d'envoyer aux couches inférieures les données fournies par l'utilisateur.

1.3.3 Suite TCP/IP

L'architecture TCP/IP est similaire au modèle OSI mais ne met en jeu que trois couches, car elle combine les couches supérieures OSI en une seule et ne s'occupe pas des couches en-dessous de la couche réseau, les protocoles des ces couches étant propres au réseau sous-jacent. Elle date de 1974 mais elle est définie, après coup en 1989, dans la section 1.1.3 de [RFC 1122] :

Application
Transport
Internet

- La **couche application** (*application layer* en anglais) regroupe toutes les tâches orientées application, c'est-à-dire celles des couches 5 à 7 du modèle OSI.
- La **couche transport** (*transport layer* en anglais), comme dans le modèle OSI, permet le multiplexage et le démultiplexage entre les applications de systèmes d'extrémité.

– La **couche Internet** (*Internet layer* en anglais) est principalement chargée de de router les paquets IP de l'expéditeur au destinataire à travers le réseau. Elle correspond à la couche réseau du modèle OSI.

1.3.4 Le modèle des sockets

Les modèles ci-dessus datent de 1984 et 1989. Tout se complique encore par le fait que les sockets, mises en place en 1983, comme nous le verrons plus loin, donnent lieu à un modèle du sous-système réseau des systèmes d'exploitation en trois couches avec un vocabulaire différent, mais que l'on peut relier aux modèles précédents :

– La couche physique n'interfère pas du tout avec le sous-système réseau des systèmes d'exploitation, donc on n'en parle pas.
– Les protocoles de la couche liaison sont traités dans la partie électronique (le contrôleur) des cartes réseau. Il n'y a donc pas besoin non plus d'en parler au niveau du sous-système réseau.
– La couche réseau s'appelle **famille d'adresses**, la mise en place des sockets retenant essentiellement la structure des adresses.
– La couche transport s'appelle **type de communication**.
– Une troisième couche, appelée **protocole**, avait été prévue si les deux couches précédentes n'étaient pas suffisantes. On la retrouve comme paramètre (presque toujours égal à zéro) mais ne sert pas à grand chose.
– Les couches supérieures concernent les applications et non le sous-système réseau, donc on ne s'en occupe pas.

1.3.5 Modèle hybride

L'Internet et la plupart des réseaux intranets d'entreprise ont recours à une **architecture hybride** TCP/IP–OSI qui s'appuie sur les couches basses de l'architecture OSI pour spécifier les infrastructures de réseau de type LAN ou autres :

Application
Transport
Réseau
Liaison
Physique

Nous verrons de plus que la couche liaison de données est divisée en deux sous-couches dans le standard IEEE.

1.3.6 Cas de Linux

Linux se réfère au modèle hybride mais la mise en place conduit à deux commentaires :

– Linux ne s'occupe pas du tout de la couche physique puisque aucun élément de programmation n'intervient à ce niveau. La mise en œuvre des protocoles de la couche liaison est presque entièrement traitée dans les cartes réseau de nos jours ; Linux n'a donc pas à s'en préoccuper si ce n'est sous la forme d'une interface avec celle-ci (il faut bien commencer par quelque chose).

– Linux ne respecte pas entièrement la philosophie des couches avec une interface bien définie. Nous verrons, par exemple, que l'on passe directemnt de la couche application à la couche réseau dans certains cas (plus exactement lors de l'envoi de données UDP).

1.3.7 Les en-têtes de protocole

Une unité d'information est composée de données et d'informations de contrôle de cette unité d'information. Ces informations de contrôle sont généralement assemblées dans un bloc venant avant les données, ce qui est le cas pour tous les protocoles de TCP/IP. On appelle ces informations un **en-tête de protocole.**

Lorsque l'unité d'information est passée à la couche inférieure, celle-ci ajoute son en-tête à l'unité d'information passée, considérée comme les données de cette couche inférieure. De cette manière, lorsqu'une unité d'information est partie de la couche d'application, au moment où il atteint la couche physique, elle contient quatre en-têtes de protocole (sept avec le modèle OSI).

Pour mieux visualiser ce processus, on peut se représenter les différentes couches d'un oignon. L'intérieur est constitué par les données à envoyer. À chaque fois que l'unité d'information passe à travers une des couches, une couche d'oignon est ajoutée. Lorsqu'elle a parcouru toutes les couches, plusieurs en-têtes de protocole entourent les données initiales. Lorsque l'unité d'information est envoyée à travers toutes les couches en partant du bas (sur une autre machine en général), chaque couche pèle l'en-tête de protocole lui correspondant. Lorsque la couche d'application de destination est atteinte, il ne reste plus que les données initiales.

OSI dispose d'une description formelle pour tout ce processus. La couche en cours porte le numéro N. Les données N-utilisateur à transférer doivent être précédées d'informations de contrôle de N-protocole (**N-PCI** pour *Protocol Control Information*) pour former une unité de données de N-protocole (**N-PDU** pour *Protocol Data Unit*). Les N-PDU sont passéss par l'intermédiaire d'un point d'accès de N-service (**N-SAP** pour *Service Access Point*) sous la forme d'un ensemble de paramètres de service comprenant une unité de données de N-service (**N-SDU** pour *Service Data Unit*). Les paramètres de service comprenant les N-SDU sont appelés les données d'utilisateur de N-service (**N-SUD** pour *Service User Data*), mis devant le (N-1)-PCI pour former un autre (N-1)-PDU.

1.4 Historique

Une bonne introduction à l'histoire des réseaux est [HL–96]. Elle a été écrite par des journalistes qui ont pu, d'une part, consulter les documents de littérature grise très difficilement accessibles et, d'autre part, interviewer les premiers acteurs de cette histoire. Elle contient une très grande documentation dont le seul reproche que l'on puisse faire est la non visibilité de la structuration par le manque de titres et sous-titres suffisamment explicites.

1.4.1 Naissance des ordinateurs

Il n'existe pas, à ma connaissance, d'étude abordable sur l'origine de la comptabilité au Néolithique, ni sur l'origine du calcul.

La notion de machine de calcul universel et l'apparition des premiers ordinateurs sont contés, à un niveau de haute vulgarisation avec des pointeurs sur la littérature primaire, dans [DAV–00], malheureusement non traduit en français.

1.4.2 Terminaux distants

Nous avons vu qu'une première alliance entre informatique et télécommunication est l'utilisation de terminaux distants de l'unité centrale, parfois situés à plusieurs milliers de kilomètres.

Le premier problème à résoudre pour cela est celui des utilisateurs multiples (*time-sharing* en anglais). Ceci est un problème du système d'exploitation. En 1961 est développé le **CTSS** (*Compatible Time Sharing System*) au MIT (*Massachussets Institute of Technology*), première réalisation importante mais encore expérimentale dans ce domaine.

Ceci conduira, en 1964, un groupe de chercheurs du MIT, des Bell Laboratories et de General Electric à s'associer pour initier le développement d'un système d'exploitation multi-utilisateur. Ils baptisèrent leur ambitieux projet **MULTICS** (*Multiplexed Information and Computing System*). Ce projet n'a pas vraiment abouti et sera abandonné en 1968. Il est cependant à l'origine de Unix, qui date de 1971.

Du point de vue matériel, dans les premiers essais d'utilisation des terminaux distants, on utilisait des lignes télex.

En 1974, IBM lance le **SNA** (*Systems Network Architecture*) qui normalise la communication entre un ordinateur et les terminaux distants. **VTAM** (*Virtual Telecommunication Access Method*) tourne sur l'ordinateur (un IBM 370) alors que **NCP** (*Network Control Program*) tourne sur le contrôleur de communication pour établir et surveiller en permanence le trafic.

1.4.3 Première mise en réseau : 1965

La première mise en réseau, entre deux ordinateurs (et non une unité centrale et un terminal), eut lieu en 1965. Le psychologue Tom Marill lança cette année-là une petite société de systèmes en temps partagé. Mais son principal investisseur ayant fait défaut à la dernière minute, il dut chercher un contrat de recherche et développement. Il proposa donc à l'ARPA de mener une expérience de mise en réseau, en joignant l'ordinateur TX-2 du Lincoln Laboratory et le SDC Q-32 situé à Santa Monica. La société de Marill était si petite que l'ARPA lui recommanda de procéder à son expérience sous l'égide du Lincoln Laboratory. L'idée plut aux responsables du Lincoln et ils chargèrent Larry Roberts de surveiller le projet.

La liaison entre les deux ordinateurs était réalisée grâce à un service spécial de la Western Union : quatre fils en duplex intégral. Marill branchait sur cette liaison un type de modem rudimentaire, opérant à 2 000 bits par seconde, qu'il appelait un composeur automatique (*automatic dialer*). Marill établit une procédure pour grouper les caractères en messages, les envoyer et vérifier qu'ils arrivent. Si aucun accusé de réception ne suivait, le message était transmis à nouveau. Il appela " protocole" de message l'ensemble des procédures pour faire circuler l'information dans les deux sens. En dépit de leurs efforts, lorsque Marill et Roberts connectèrent effectivement les deux machines, le résultat fut mitigé : le temps de réponse était médiocre (voir [HL–96], pp. 82–83).

Même si le résultat n'est pas vraiment celui escompté, il s'agit bien de la première mise en réseau : les ordinateurs ont des systèmes d'exploitation différents et on utilise des protocoles.

1.4.4 Réseaux de communication

De nos jours, "réseau" sans adjectif désigne toujours un réseau informatique. Le réseau est un concept général très utilisé depuis le réseau de nos connaissances personnelles jusqu'aux réseaux de communication et de télécommunication.

Les réseaux de communication cohérents commencent avec les Romains pour les routes, revus aux dix-neuvième siècle avec le réseau de routes nationales (départementales et vicinales) et celui des autoroutes au vingtième siècle sans oublier les chemins de fer.

Les réseaux de télécommunication commencent avec les feux dans l'Antiquité, les signaux de fumée des indiens d'Amérique, les relais de poste au début des Temps modernes. Il prit vraiment son essor avec le télégraphe optique de Chappe sous le Révolution puis avec le télégraphe électrique qui le détronera dans la seconde moitié du dix-neuvième siècle. C'est aussi le début des premiers câbles transatlantiques. Le réseau téléphonique suivra avec la naissance des grandes sociétés telles que AT&T aux États-Unis et les PTT dépendant directement du gouvernement dans de nombreux pays dont la France.

1.4.5 Commutation par paquets

L'utilisation de la commutation par paquets fut pronée indépendanmment par Paul Baran aux États-Unis et Donald Davies à Londes.

Nous avons vu ci-dessus comment Paul Baran en est venue à l'idée de l'utilisation des réseaux maillés. La seconde contribution de Baran aux réseaux a été plus révolutionnaire encore : disloquer également les messages pour obtenir la commutation par paquets, qu'il appelait " blocs-messages". Dans le modèle de Baran, chaque nœud de communication contenait une table de routage qui se comportait comme une sorte de coordinateur ou de régulateur. Il espérait persuader AT&T des avantages de son projet mais ce ne fut pas le cas. En 1965 cependant, cinq ans après s'être lancé dans le projet, Baran obtint l'appui complet de la RAND qui voulût construire un réseau de commutation distribué. Malheureusement AT&T, qui disposait du monopole des télécommunications, s'y opposa. Baran se tourna alors vers d'autres projets (voir [HL–96], pp.72–78).

À Londres, à l'automne 1965, juste après que Baran eut laissé tomber son projet, Donald Watts Davies, quarante et un ans, physicien au British National Physical Laboratory (NPL), écrivait une première note exposant ses idées à propos d'un nouveau réseau d'ordinateurs fort proche de celui de Baran. Le printemps suivant il donnait à Londres une conférence publique où il décrivait l'envoi de petits blocs de données – qu'il appelait "paquets" – à travers un réseau numérique sur le principe du stockage et retransmission. En 1966, après l'annonce de son travail précurseur sur la commutation par paquets, il fut nommé chef de la division informatique du NPL. Les motifs qui avaient conduit Davies à l'idée d'un réseau de commutation par paquets n'avaient rien à voir avec les préoccupations militaires qui avaient poussé Baran. Davies voulait simplement créer un nouveau réseau public de communication. Il a prévu la nécessité de maîtriser la diversité des matériels et des logiciels, autrement dit les différences séparant les langages informatiques ou les systèmes d'exploitation des machines. Contrairement à la réaction très fraîche d'AT&T à l'égard de Baran, les télécommunications britanniques ont épousé les idées de Davies. Cela l'a encouragé à chercher un financement pour bâtir un réseau expérimental au NPL (voir [HL–96], pp.78–81).

La théorie de la commutation par paquets a été également étudiée par Leonard KLEINROCK en 1959 lorsqu'il était étudiant de troisième cycle au MIT ([KLE–61], [KLE–64]). Il effectua un travail théorique important qui décrivait une série de modèles analytiques pour les réseaux de communication.

1.4.6 Le premier réseau : ARPANET en 1971

Le vendredi 4 octobre 1957, l'Union Soviétique lance le premier satellite artificiel, appelé Spoutnik. Les Américains considèrent cela comme une menace. Le président Einsenhower demande le 7 janvier 1958 au Congrès les fonds nécessaires pour créer l'ARPA (*Advanced Research Projects Agency*). Directement lié au président at au secrétaire à la défense, cet organisme de recherche avait pour but de garantir, par la promptitude de sa réaction, que les américains ne

soient désormais plus jamais en retard dans aucun domaine technologique ([HL–96], pp. 19–31).

En 1961, le directeur de l'ARPA cherchait quelqu'un pour gérer le contrat d'un nouvel ordinateur commandé par l'ARPA, le Q-32, ainsi que quelqu'un qui pût diriger un nouveau programme, demandé par le département de la Défense et axé sur les sciences du comportement. À l'automne 1962, il trouve enfin un candidat susceptible d'occuper les deux postes, un éminent psychologue nommé Joseph Carl Robnett Licklider. D'après son contrat, sa tâche principale était de trouver pour l'ordinateur des utilisations qui en fassent autre chose qu'un outil destiné aux calculs numériques des scientifiques. En un rien de temps, il prit contact avec les meilleurs informaticiens du moment, à Stanford, au MIT, à Berkeley, à UCLA (*University of California, Los Angeles*), ainsi qu'avec une poignée de sociétés. Six mois après son arrivée à l'ARPA, Licklider envoya une longue note aux membres de cet entourage à propos de la dispersion excessive des thèmes de recherche ([HL–96], pp.32–49). Comment parvenir à y remédier ? Il discuta de l'hypothèse d'un réseau (*network*) d'ordinateurs :

> "Considérez la situation où plusieurs centres sont réunis dans un même filet [*netted*], chaque centre étant tout à fait singulier, avec son propre langage et sa propre façon de façon de faire les choses. N'est-il pas désirable ou même nécessaire que tous les centres s'accordent sur un quelconque langage ou, du moins, sur quelques conventions pour poser des questions telles que : "Quel langage parlez-vous ?" À ce point, le problème est avant tout celui dont débattent les auteurs de science-fiction : comment amorcer des communications entre des êtres doués de raison, mais privés de toute forme de correspondance ?"

L'une des personnes récrutées par Licklider, Robert Taylor, devint le directeur du service de celui-ci, le LIPTO, en 1966. Il décida de mettre en pratique les idées de Licklider et demanda un financement pour faire l'expérience d'un réseau d'ordinateurs. Il suggéra que l'ARPA finance un petit réseau à titre d'essai : on commencerait, par exemple, avec quatre nœuds, et on poursuivrait jusqu'à une douzaine environ ([HL–96], pp.49–53). Il obtînt un million de dollars pour mettre en place son système.

L'architecte d'un tel réseau devrait être également un expert en télécommunication. Taylor recruta Larry Roberts en décembre 1966 ([HL–96], pp.55–63). Celui-ci commença par écrire une note dans laquelle il appelait les ordinateurs intermédiaires qui contrôleraient le réseau des "serveurs de messages", des **IMP** pour *Interface Message Processor*. Ils devaient remplir les fonctions suivantes : interconnecter le réseau, envoyer et recevoir des données, effectuer des tests d'erreur, retransmettre en cas d'erreur, acheminer les données et vérifier que les messages arrivent aux destinations voulues ([HL–96], pp.81–91). Un protocole serait établi pour définir avec exactitude comment les IMP devraient communiquer avec les ordinateurs hôtes (il ne le sera jamais).

Fin 1967, Roberts présenta son premier exposé sur ce qu'il appela l'"ARPA net", le réseau de l'ARPA, à un colloque d'informatique de l'ACM à Gatlinburg, dans le Tennessee.

Roberts estimait que le réseau devait démarrer avec quatre sites : UCLA, le SRI (*Stanford Research Institute*), l'université d'Utah et l'UCSB (*University of California, Santa Barbara*). Dans un second temps, il se développerait jusqu'à en réunir dix-neuf environ. Il décida de passer un appel d'offres, qu'il termina fin juillet 1968. Les premières réactions, négatives, à l'appel d'offres vinrent des deux plus grands constructeurs d'ordinateurs, IBM et Control Data Corporation (CDC). Les deux sociétés refusèrent de soumissionner car elles estimaient que le réseau ne pourrait jamais être construit parce qu'il n'existait pas d'ordinateur assez petit pour rendre l'affaire rentable. Plus d'une douzaine d'offres furent soumises. Juste avant noël 1968, l'ARPA annonça que le contrat pour la construction des serveurs de messages était attribué à Bolt, Beranek & Newman (BBN), une petite société de conseils de Cambridge, Massachussetts ([HL–96],

pp.92–97).

Les cinq années suivantes furent consacrées à des tests et à des mises au point. En 1971, ARPANET entra en service régulier. Les machines utilisaient ARPANET en se connectant à un IMP. La façon de connecter un IMP au réseau se faisait à l'aide du **protocole "1822"**, dont le nom provient du nombre de pages techniques décrivant le système. Nous verrons que la façon dont un hôte devait se relier à l'IMP ne fut jamais décrit dans un protocole de la part des concepteurs de l'ARPANET.

Notons qu'en 1980 l'agence change de nom, passant de ARPA à DARPA (*Defense Advanced Research Projects Agency*).

Références

[TAN–81], à travers ses éditions successives, est devenu le classique pour une vue d'ensemble sur les réseaux. Il contient beaucoup d'informations mais pas d'exemple étudié en détail. [K-R–01] peut devenir un concurrent sérieux et au minimum un très bon complément. [PUJ–04] est le classique français.

La norme ISO 7498, *Open System Interconnection Model standard* est décrite dans quatre documents, disponibles en anglais et en français : [ISO 7498-1], [ISO 7498-2], [ISO 7498-3] et [ISO 7498-4].

L'architecture TCP/IP a été décrite, après coup, dans [RFC 1122].

Première partie

Couches physique et de liaison

Nous avons décidé, après mûre réflexion, d'inclure cette première partie dans cet ouvrage (consacré à l'implémentation du sous-système réseau de Linux) à titre de complétude sur l'étude des réseaux. Elle ne joue aucun rôle dans le sous-système réseau des systèmes d'exploitation. Elle peut donc être passée sans problème pour la lecture de la suite.

Chapitre 2

La couche physique

La *couche physique* est chargée de la transmission des bits à l'état brut sur un canal de communication. Les problèmes de conception concernent les interfaces mécaniques (le nombre de broches d'un connecteur et leur rôle) et électriques (le nombre de volts à fournir pour représenter un 1 et un 0), la synchronisation ainsi que le support physique de transmission (*medium* en anglais). La prise en compte de cette couche concerne donc le matériel, les concepteurs des cartes réseau, des micro-ordinateurs (et des routeurs) et non les logiciels que sont les sous-systèmes réseau des systèmes d'exploitation. Par conséquent Linux n'est pas partie prenante de la couche physique.

Nous allons cependant mettre en avant un minimum de connaissances concernant la couche physique qui nous serons utiles, sans être à proprement nécessaires, pour la suite.

2.1 Les supports physiques

Un bit peut être représenté par une donnée mécanique, électrique (par une certaine différence de potentiel), hydraulique, pneumatique (c'est-à-dire par une certaine pression), optique ou autre. Actuellement, et sauf à titre tout à fait expérimental, les bits sont toujours représentés au niveau de l'ordinateur électroniquement en jouant sur une certaine différence de potentiel. Ce n'est pas le cas dans les canaux de communication où l'on utilise une fréquence (cas du réseau téléphonique ou des lignes spécialisées), des ondes radio ou des rayons lumineux (dans le cas des fibres optiques). Cependant les signaux sont toujours transformés à l'arrivée et, au niveau de l'ordinateur, on n'a à tenir compte que de différences de potentiel.

Nous renvoyons, par exemple, au chapitre deux de la dernière édition en cours de [TAN–81] pour une vue d'ensemble des medias physiques utilisés pour la couche physique. Pour ce qui nous concerne, nous allons nous contenter de les citer :

- les supports avec guide physique (*transmission filaire*) :
  - avec une âme en cuivre comprennent les paires torsadées et les câbles coaxiaux ;
  - remplacés pour les distances moyennes par des fibres optiques ;
- les supports sans guide physique (*transmission sans fil*) utilisent les ondes radio :
  - *via* des relais hertziens ;
  - ou des satellites de communication.

2.2 Interface support physique/ordinateur

Le modèle que l'on peut prendre pour l'interface entre le support transportant les données réseau et l'ordinateur est un flux de bits, comme dans le cas d'une liaison série. La liaison série fut d'ailleurs longtemps utilisée, par exemple dans le cas d'un accès au réseau *via* un modem (avant l'arrivée du câble et de ADSL).

2.2.1 Rappels sur les liaisons série

Nous nous permettons de renvoyer au chapitre 24 de notre livre [CEG–03] pour une étude des liaisons série. Rappelons-en ce qui est essentiel pour la suite.

Dans une **liaison série**, les données sont transmises un bit à la fois (et non pas un ou plusieurs octets à la fois). On distingue deux types de communications séries : les *communications séries asynchrone* et les *communications séries synchrones*.

2.2.1.1 Liaisons série asynchrone

Dans une **communication série asynchrone**, les octets sont transmis de façon irrégulière ; il faut donc un **marqueur** pour indiquer le début de l'envoi d'un certain nombre de bits, convenu à l'avance, et un marqueur pour en indiquer la fin.

Une communication série asynchrone se caractérise par la nature et le nombre de ses bits de début et d'arrêt et par son taux de transmission :

Bit de début et bit d'arrêt. La technique courante pour les communications série asynchrones consiste à maintenir le signal à un niveau haut (une **marque**, en anglais *mark*) jusqu'à ce qu'une donnée soit transmise. Chaque caractère transmis doit commencer par un bit de niveau 0 (un **espace**, en anglais *space*), appelé le **bit de début** (en anglais *start bit*). Il

24.1.2 Attributs communs 406
24.1.3 Attributs de contrôle 408
24.1.4 Attributs associés aux options des sockets 410
24.1.5 Fonctions membre 411
24.2 Allocation et libération 411
24.2.1 Allocation 411
24.2.2 Gestion du compteur de référence 413
24.2.3 Libération 414
24.3 Gestion des données associées 416
24.3.1 Initialisation des données 416
24.3.2 Verrouillage 418
24.3.3 Initialisation du délai 419

25 Réception des datagrammes UDP **421**
25.1 Implémentation Linux générale d'UDP 421
25.1.1 Implémentation Linux de l'en-tête UDP 421
25.1.2 Calcul de la somme de contrôle 422
25.2 Réception des datagrammes UDP 428
25.2.1 Fonction de traitement principale 428
25.2.2 Vérification rapide de la somme de contrôle 431
25.2.3 Consultation de la table de hachage UDP 433
25.2.4 Traitement du datagramme UDP 436
25.2.5 Mise dans la file d'attente de réception UDP 438

26 Installation de la famille de protocoles IPv4 **441**
26.1 Manipulation des familles de protocoles 442
26.1.1 Familles de protocoles 442
26.1.2 Les types de communication 444
26.1.3 Les protocoles de la suite TCP/IPv4 449
26.2 Implémentation 449
26.2.1 Tableau des familles de protocoles 449
26.2.2 Enregistrement d'un type de communication IPv4 452
26.2.3 Installation de la suite de protocoles TCP/IPv4 453

27 Implémentation des fichiers de type socket **459**
27.1 Implémentation générale des fichiers 460
27.1.1 Système de fichiers virtuel 460
27.1.2 Super-bloc 461
27.1.3 Nœud d'information 463
27.1.4 Descripteur de fichier 465
27.1.5 Répertoire 466
27.1.6 Types de fichiers 466
27.1.7 Déclaration d'un système de fichiers 467
27.1.8 Enregistrement d'un système de fichiers 467
27.2 Descripteur de socket 468
27.2.1 Définition du type 468
27.2.2 Les états d'une socket 469
27.2.3 Les drapeaux 469
27.2.4 Le type opérations sur une socket 469
27.3 Déclaration du système de fichiers des sockets 470

27.3.1 La structure . . . 470
27.3.2 Enregistrement . . . 470
27.4 Système de fichiers de sockets . . . 471
27.4.1 Obtention du super-bloc . . . 471
27.4.2 Libération d'un super-bloc . . . 474
27.5 Opérations sur les super-blocs . . . 475
27.5.1 Ensemble des opérations . . . 475
27.5.2 Statut du système de fichiers . . . 475
27.5.3 Allocation d'un descripteur de nœud d'information . . . 476
27.5.4 Libération d'un descripteur de nœud d'information . . . 477
27.6 Opérations sur les répertoires de socket . . . 477
27.7 Opérations sur les fichiers de type socket . . . 477
27.8 Premières implémentations de fonctions . . . 479
27.8.1 Implémentation du positionnement . . . 479
27.8.2 Implémentation de la scrutation . . . 479
27.8.3 Implémentation du non mappage en mémoire vive . . . 480
27.8.4 Implémentation de l'ouverture de fichier . . . 481
27.8.5 Implémentation des événements asynchrones . . . 481
27.8.6 Implémentation de la libération de fichier . . . 482

28 Création et fermeture des sockets 485
28.1 Création d'une socket . . . 486
28.1.1 Traitement général . . . 486
28.1.2 Cas de la famille de protocoles IPv4 . . . 495
28.2 Semi-arrêt d'une socket . . . 500
28.2.1 Traitement général . . . 500
28.2.2 Cas de IPv4 . . . 502
28.2.3 Fonction de déconnexion spécifique à UDP . . . 503
28.3 Fermeture d'une socket . . . 506

29 Initialisation d'un serveur 509
29.1 Partie générale . . . 510
29.1.1 Fonction d'appel . . . 510
29.1.2 Passage espace utilisateur/espace noyau . . . 510
29.2 Cas de IPv4 . . . 511
29.2.1 Fonction spécifique d'initialisation du serveur . . . 511
29.2.2 Détermination du type d'adresse . . . 513
29.3 Obtention et vérification du port dans le cas UDP . . . 514

30 Réception d'un datagramme par une socket 519
30.1 Types liés aux fonctions d'entrée-sortie . . . 520
30.1.1 Vecteurs d'entrée-sortie . . . 520
30.1.2 En-tête de message . . . 520
30.1.3 En-tête de contrôle . . . 520
30.2 Implémentation générale de la réception des datagrammes . . . 521
30.2.1 Fonction d'appel . . . 521
30.2.2 Réception de message de socket . . . 522
30.2.3 Passage espace noyau/espace utilisateur . . . 523
30.2.4 Fonction interne de réception d'un message . . . 524
30.3 Cas de IPv4 . . . 525

30.4 Cas de UDP 526
30.5 Lien avec les tampons de socket 529
30.5.1 Instantiation de descripteur de tampon en réception 529
30.5.2 Copie d'un datagramme dans l'espace utilisateur 533
30.5.3 Libération d'un tampon de datagramme 537

V Envoi **539**

31 Envoi d'un datagramme par une socket **541**
31.1 Niveau implémentation des sockets 542
31.1.1 Fonction d'appel 542
31.1.2 Envoi de message de socket 543
31.1.3 Fonction interne d'envoi de message 543
31.2 Cas de IPv4 544
31.2.1 Appel à la fonction spécifique à la couche de transport 544
31.2.2 Attribution automatique de numéro de port 545

32 Envoi de datagrammes ordinaires sous UDP **547**
32.1 Fonction d'envoi spécifique à UDP 548
32.1.1 Description 548
32.1.2 Implémentation 548
32.2 Regroupement de données UDP 555
32.2.1 Fonction principale 555
32.2.2 Instantiation d'un tampon de socket pour l'émission 561
32.2.3 Récupération des fragments 564
32.3 Constitution du datagramme 566
32.4 Envoi du datagramme à la couche réseau 569

33 Envoi de paquets ordinaires sous IPv4 **571**
33.1 Première étape : routage d'un paquet ordinaire 572
33.1.1 Recherche dans le cache de routage 572
33.1.2 Recherche étendue à la FIB 573
33.2 Seconde étape : constitution du paquet IP 581
33.2.1 Implémentation des options IP 581
33.2.2 Récupération des fragments 581
33.3 Transfert de la couche réseau à la couche inférieure 585
33.3.1 Première sous-étape : détermination de la fonction de transfert 585
33.3.2 Deuxième sous-étape : fragmentation éventuelle 586
33.3.3 Troisième sous-étape : positionnement du type du paquet 586
33.3.4 Quatrième sous-étape : passage à la couche inférieure 586

34 Envoi de trames **589**
34.1 Transmission à la carte réseau : partie générale 590
34.1.1 Première étape : mise en file d'attente 590
34.1.2 Deuxième étape : récupération des paquets 596
34.1.3 Troisième étape : envoi 601
34.2 Partie spécifique à la carte 603
34.2.1 Étude générale 603
34.2.2 Cas de la carte 3Com 501 : Mise en file d'attente 604

34.3 Réaction à l'envoi 607
34.3.1 Réponse du gestionnaire d'interruption 607
34.3.2 Action lorsque le délai est écoulé 609

35 Désactivation et retrait **611**
35.1 Désactivation 612
35.1.1 Cas général 612
35.1.2 Cas de la carte 3Com 501 614
35.2 Retrait d'un périphérique réseau 615
35.2.1 Démontage du système de fichier 619
35.2.2 Démontage du système de fichiers 619
35.2.3 Attente des références 619

VI Complément sur IP : la fragmentation **621**

36 Fragmentation pour la couche réseau : le cas de IPv4 **623**
36.1 Fragmentation 624
36.2 Réassemblage 632
36.2.1 Cache de fragment 632
36.2.2 Traitement du réassemblage 634
36.2.3 Traitement lors de l'expiration du délai 643
36.3 Initialisation de la fragmentation 645

VII Appendices **647**

Bibliographie **649**

Index **654**

Table des figures

1 Hiérarchie entre les chapitres vi

1.1 Graphe complet 3
1.2 Réseau avec commutateur central 4
1.3 Réseau distribué 6
1.4 Pile réseau 8

2.1 Niveaux logiques 23
2.2 Synchronisation 23
2.3 Codage Manchester 24
2.4 Réseau étendu 26
2.5 Réseau machine à machine 28
2.6 Réseau en bus 29
2.7 Réseau en anneau 29
2.8 Réseau en étoile 30
2.9 Croissance d'Arpanet : (a) décembre 1969 (b) juillet 1970 (c) mars 1971 (d) avril 1972 (e) septembre 1972 31
2.10 Èpine dorsale NSFNET en 1988 32
2.11 Architecture du premier réseau Ethernet 33

3.1 Structure d'une trame 36
3.2 Décomptage des octets 37
3.3 Fanion de signalisation 38
3.4 Confirmation et infirmation 39
3.5 Fenêtre glissante 41

4.1 Types de câblages Ethernet 46
4.2 Types de connexions Ethernet 47
4.3 Sous-couches MAC et LLC 49

5.1 Numéros de port bien connus des serveurs 63

7.1 Données ancillaires 89

9.1 Fragmentation d'un paquet IP 105
9.2 Table de hachage 106
9.3 Pseudo en-tête UDP 112

12.1 Tampon de socket et son descripteur 150
12.2 File d'attente de paquets 157

19.1 Gestion des protocoles de réseau 306
21.1 Table de routage 328
22.1 Structure du cache de routage 368
36.1 Cache de fragment 632

Première partie

Couches physique et de liaison

Nous avons décidé, après mûre réflexion, d'inclure cette première partie dans cet ouvrage (consacré à l'implémentation du sous-système réseau de Linux) à titre de complétude sur l'étude des réseaux. Elle ne joue aucun rôle dans le sous-système réseau des systèmes d'exploitation. Elle peut donc être passée sans problème pour la lecture de la suite.

Chapitre 2

La couche physique

La *couche physique* est chargée de la transmission des bits à l'état brut sur un canal de communication. Les problèmes de conception concernent les interfaces mécaniques (le nombre de broches d'un connecteur et leur rôle) et électriques (le nombre de volts à fournir pour représenter un 1 et un 0), la synchronisation ainsi que le support physique de transmission (*medium* en anglais). La prise en compte de cette couche concerne donc le matériel, les concepteurs des cartes réseau, des micro-ordinateurs (et des routeurs) et non les logiciels que sont les sous-systèmes réseau des systèmes d'exploitation. Par conséquent Linux n'est pas partie prenante de la couche physique.

Nous allons cependant mettre en avant un minimum de connaissances concernant la couche physique qui nous serons utiles, sans être à proprement nécessaires, pour la suite.

2.1 Les supports physiques

Un bit peut être représenté par une donnée mécanique, électrique (par une certaine différence de potentiel), hydraulique, pneumatique (c'est-à-dire par une certaine pression), optique ou autre. Actuellement, et sauf à titre tout à fait expérimental, les bits sont toujours représentés au niveau de l'ordinateur électroniquement en jouant sur une certaine différence de potentiel. Ce n'est pas le cas dans les canaux de communication où l'on utilise une fréquence (cas du réseau téléphonique ou des lignes spécialisées), des ondes radio ou des rayons lumineux (dans le cas des fibres optiques). Cependant les signaux sont toujours transformés à l'arrivée et, au niveau de l'ordinateur, on n'a à tenir compte que de différences de potentiel.

Nous renvoyons, par exemple, au chapitre deux de la dernière édition en cours de [TAN–81] pour une vue d'ensemble des medias physiques utilisés pour la couche physique. Pour ce qui nous concerne, nous allons nous contenter de les citer :

- les supports avec guide physique (*transmission filaire*) :
  - avec une âme en cuivre comprennent les paires torsadées et les câbles coaxiaux ;
  - remplacés pour les distances moyennes par des fibres optiques ;
- les supports sans guide physique (*transmission sans fil*) utilisent les ondes radio :
  - *via* des relais hertziens ;
  - ou des satellites de communication.

2.2 Interface support physique/ordinateur

Le modèle que l'on peut prendre pour l'interface entre le support transportant les données réseau et l'ordinateur est un flux de bits, comme dans le cas d'une liaison série. La liaison série fut d'ailleurs longtemps utilisée, par exemple dans le cas d'un accès au réseau *via* un modem (avant l'arrivée du câble et de ADSL).

2.2.1 Rappels sur les liaisons série

Nous nous permettons de renvoyer au chapitre 24 de notre livre [CEG–03] pour une étude des liaisons série. Rappelons-en ce qui est essentiel pour la suite.

Dans une **liaison série**, les données sont transmises un bit à la fois (et non pas un ou plusieurs octets à la fois). On distingue deux types de communications séries : les *communications séries asynchrone* et les *communications séries synchrones*.

2.2.1.1 Liaisons série asynchrone

Dans une **communication série asynchrone**, les octets sont transmis de façon irrégulière ; il faut donc un **marqueur** pour indiquer le début de l'envoi d'un certain nombre de bits, convenu à l'avance, et un marqueur pour en indiquer la fin.

Une communication série asynchrone se caractérise par la nature et le nombre de ses bits de début et d'arrêt et par son taux de transmission :

Bit de début et bit d'arrêt. La technique courante pour les communications série asynchrones consiste à maintenir le signal à un niveau haut (une **marque**, en anglais *mark*) jusqu'à ce qu'une donnée soit transmise. Chaque caractère transmis doit commencer par un bit de niveau 0 (un **espace**, en anglais *space*), appelé le **bit de début** (en anglais *start bit*). Il

est utilisé pour synchroniser le transmetteur et le receveur. La donnée, d'un octet, est alors envoyée en commençant par le bit de poids le plus faible. On termine par un ou deux **bits d'arrêt** (en anglais *stop bit*) à niveau haut.

Exemple.- Le schéma 2-1 ([UFF–87], p. 515) montre les niveaux logiques pour la transmis-

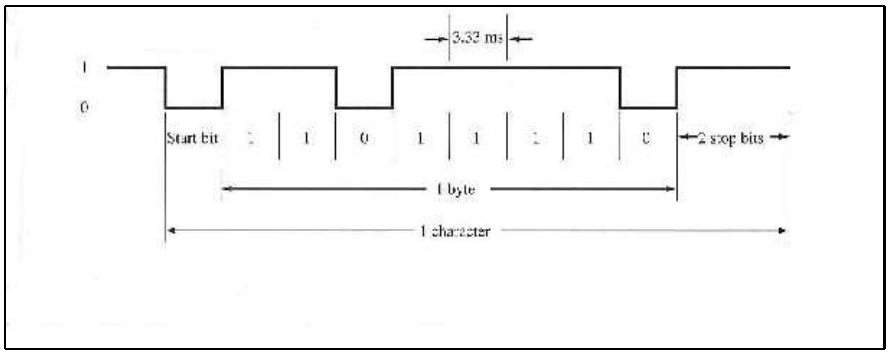


FIG. 2.1 – *Niveaux logiques*

sion de l'octet `7Bh` dans le cas de deux bits d'arrêt.

Taux de transmission. Le **taux de transmission**, ou **débit** (en anglais *data rate*), est exprimé en bits par seconde (**bps**) ou en caractères par seconde.

2.2.1.2 Synchronisation du transmetteur et du receveur

La transmission serait facile si les débits du tansmetteur et du receveur étaient parfaitement synchronisés. Mais il y a peu de chance que ceci soit réalisé.

La figure 2-2 ([UFF–87], p. 519) montre le résultat lorsque le débit est trop rapide ou trop

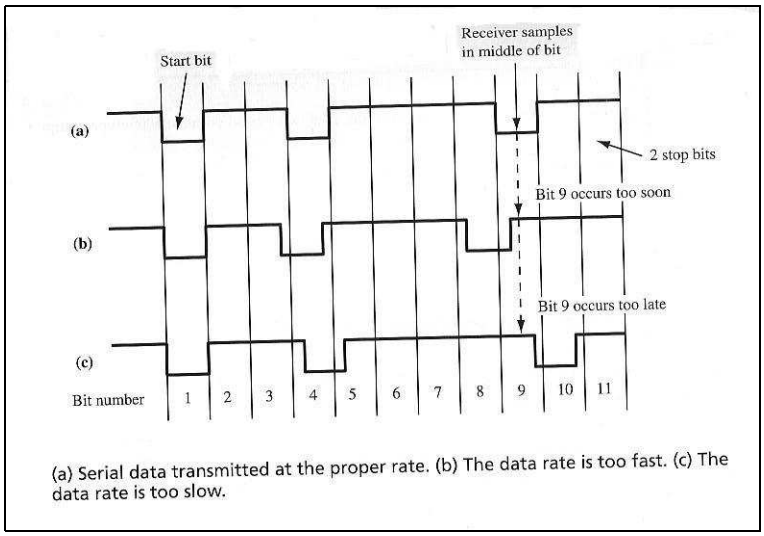


FIG. 2.2 – *Synchronisation*

lent. Dans ces deux cas, les erreurs s'accumulent. Si on regarde ce qui se passe au milieu de la transmission d'un bit, l'erreur maximum permise fait que le neuvième bit sera décalé à droite ou à gauche d'un demi `DELAY`. Si tous les bits sont décalés de la même façon, la quantité d'erreur sur un bit sera de 1/2 durée / 9 = 1/18 durée d'un bit. Ceci signifie quue les débits du transmetteur et du receveur doivent coïncider avec une erreur d'au plus 5.6 %.

La technique asynchrone doit **s'auto-synchroniser** après chaque caractère. Bien sûr le prix à payer est que la longueur de chaque octet de donnée doit être accrue de deux bits, soit de 25 %.

2.2.2 La synchronisation et le codage Manchester

Pour résoudre le problème de la synchronisation, Ethernet, par exemple, n'emploie pas directement un codage binaire simple avec des tensions de 0 et V volts pour coder les valeurs de bit 0 et 1, respectivement, mais le **codage Manchester**. Il s'agit d'une méthode qui permet au récepteur d'identifier avec exactitude le début, la fin ou le milieu de chaque bit, sans devoir recourir à une horloge externe.

Chaque période représentative d'un bit est divisée en deux intervalles égaux. Pour coder un bit à 1, on envoie une tension haute pendant le premier intervalle, suivie d'une tension basse durant le second intervalle. Pour coder un bit à 0, on fait l'inverse: on envoie d'abord une tension basse, puis une tension haute. Cette stratégie garantit que chaque période de codage d'un bit comprend une transition en son milieu, ce qui permet au récepteur de se synchroniser plus facilement avec l'émetteur.

La figue 2.3 illustre le codage Manchester.

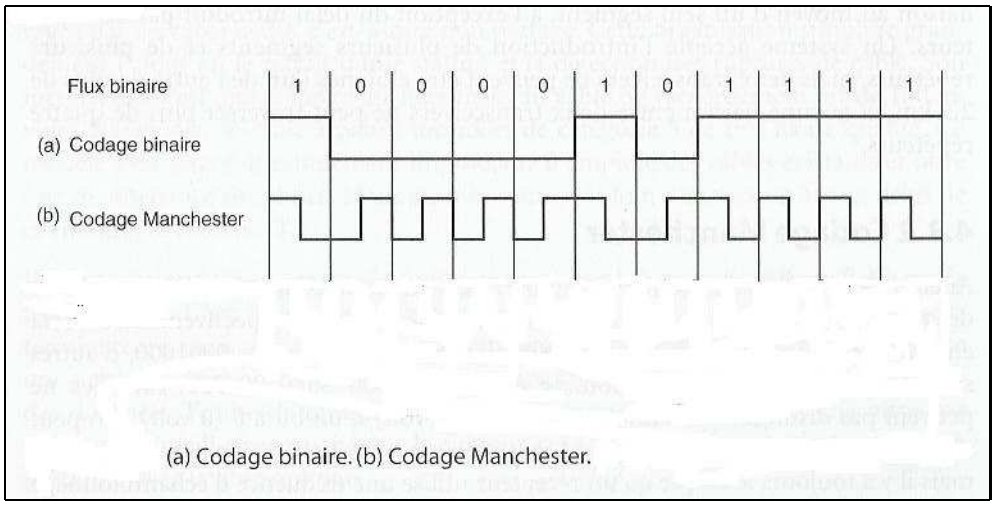


FIG. 2.3 – *Codage Manchester*

L'inconvénient de cette technique est qu'elle nécessite un débit double par rapport à un codage binaire simple, car chaque impulsion n'occupe qu'une moitié de période significative.

Tous les systèmes Ethernet utilisent le codage Manchester en raison de sa simplicité. Le signal de tension haute est +0,85 V et celui de tension basse - 0,85 V, ce qui donne une valeur de courant direct de 0 V.

2.3 Hiérarchie des réseaux

Nous avons déjà parlé du premier réseau de Tom Marril reliant deux ordinateurs et du réseau ARPANET réparti sur tous les États-Unis. Les réseaux informatique ont connu un tel succès que les entreprises ont utilisées des réseaux au sein de celles-ci (pour partager les fichiers et les ressources matérielles telles que les imprimantes). Depuis le début du vingt-et-unième siècle, même les particuliers utilisent un réseau dans leur appartement.

2.3.1 Hiérarchie suivant la taille

On hiérarchise les réseaux, plus ou moins suivant la distance entre les ordinateurs, de la façon suibante :

- Les **réseaux personnels** (**PAN** pour l'anglais *Personal Area Network*) se destinent à une personne. Ils utilisent des techniques de réseau sans fil, en particulier le protocole *Bluetooth*.
- Les **réseaux familiaux** se destinent à une famille dans un appartement. Ces réseaux, apparus à la fin du vingtième siècle, ont d'abord utilisé les techniques des réseaux locaux puis on est passé à des techniques de réseau sans fil, et en particulier à l'un de ses protocoles appelé **Wi-FI** (pour *WIreless FIdelity*).
- Les **réseaux locaux** (**LAN** pour l'anglais *Local Area Network*) sont des réseaux privés implantés dans un même immeuble ou dans un même campus dont la taille peut atteindre quelques kilomètres.

  Ils se distinguent d'autres catégories de réseaux par trois caractéristiques : leur taille, leur technologie de transmission et leur topologie. Nous venons de parler de leur taille. Il s'agit de réseau de diffusion, et non de réseau point à point, technique sur laquelle nous reviendrons dans la sous-section suivante. Nous reviendrons également sur leur topologie ci-dessous.
- Les **réseaux métropolitains** (**MAN** pour *Metropolitan Area Network*) couvrent une ville. L'exemple le plus connu de MAN est le réseau de télévision par câble que l'on trouve dans nombre de villes. Celui-ci s'est développé à partir de l'ancien système d'antenne collective qui était utilisé dans les zones souffrant d'une mauvaise réception : on plaçait une grosse antenne en haut d'une colline pour ensuite conduire le signal par câble jusqu'au domicile des abonnés.

  Il s'agissait au départ de systèmes *ad hoc* dont la conception ne répondait qu'à des besoins locaux. Lorsque les entreprises se sont attaquées au marché, elles ont obtenu des gouvernements des contrats leur permettant de câbler des villes entières. L'étape suivante a été la programmation d'émissions de télévision qui a donné naissance à des chaînes expressément conçues pour le câble. Lorsque le public a afflué sur l'internet, les câblo-opérateurs ont réalisé qu'à l'aide de peu de changements, ils pourraient offrir un service d'accès à l'internet bidirectionnel dans certaines parties inutilisées du spectre.
- Les **réseaux longue distance** ou **réseaux étendus** (**WAN** pour *Wide Area Network*) s'étendent sur une vaste zone géographique (un pays, voire même un continent).

  On ne sait pas concevoir un WAN comme une seule entité. Il s'agit toujours de plusieurs réseaux locaux regroupés au moyen d'une méthode de connexion rapide (ligne téléphonique à haute vitesse, câble réseau spécialisé et très rapide, liaison hertzienne, fibre optique, etc.) appelée **dorsale** (*backbone* en anglais). Les réseaux locaux formant le réseau étendu s'appellent **sous-réseaux**.

  Un réseau étendu composé de deux réseaux locaux peut être vu de la façon schématisée à la figure 2.4 :

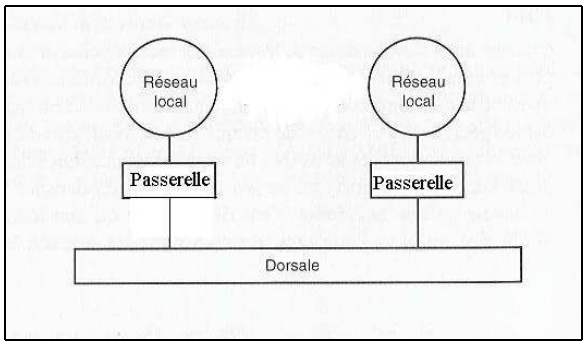


FIG. 2.4 – *Réseau étendu*

- chacun des réseaux locaux possède un ordinateur spécialisé, appelé **passerelle** (*gateway* en anglais) ;
- les deux passerelles sont reliées par une dorsale.

Le rôle d'une passerelle est triple :

- lorsqu'un message est envoyé d'un ordinateur A vers un ordinateur B du réseau étendu, la passerelle du réseau sur lequel se trouve l'ordinateur A vérifie si le message est destiné à un ordinateur d'un autre réseau local, si c'est le cas elle l'envoie sur la dorsale ;
- avant de l'envoyer, elle transforme les données du format de la suite réseau du réseau local en un format commun du réseau étendu si les deux réseaux utilisent des protocoles différents ;
- lors de l'arrivée sur le réseau local de l'ordinateur B, elle transforme les données du format commun en des données du format local.

- L'étape suivante consiste à relier tous les réseaux entre eux à l'échelle planétaire pour obtenir un **interréseau** (*internet* avec un 'i' minuscule en anglais).

  Du point de vue de la technologie, un interréseau n'est rien d'autre qu'un énorme WAN.

  Il a existé plusieurs interréseaux avec de jolis noms tels que ARPANET, BITNET ou INTERNET (avec un 'i' majuscule). Ils utilisent tous des sytèmes ouverts, mais différents les uns des autres. Comme le lecteur le sait sûrement, Internet demeure pratiquement le seul interréseau à l'heure actuelle, s'étendant sur toute la planète. **Internet** se caractérise par l'utilisation de la suite de logiciels TCP/IP.

2.3.2 Réseau à diffusion et réseau point-à-point

On distingue deux types de réseau du point de vue de leur **technologie de transmission** :

- Un **réseau à diffusion** (*broadcast* en anglais) dispose d'un seul canal de transmission qui est partagé par tous les équipements qui y sont connectés. Sur un tel réseau, chaque message envoyé est reçu par toutes les machines du réseau. Le message doit donc comporter un indice permettant d'identifier le destinataire réel. À réception de ce message, une machine cherche l'indice et procède au traitement du message si celui-ci la concerne, ou l'ignore dans le cas contraire.

– Par opposition, un **réseau point-à-point** consiste en un grand nombre de connexions, chacune faisant intervenir deux machines. Pour aller de sa source à sa destination, un message peut alors transiter par plusieurs machines intermédiaires. Ce type de réseau offre souvent plusieurs **routes** possibles de longueurs différentes pour atteindre la même destination.

Bien entendu, le débit souffre de l'utilisation d'un réseau à diffusion. D'une manière générale, le système à diffusion sera utilisé sur des petits réseaux géographiquement limités alors que le système point-à-point sera plutôt implanté pour les points d'accès aux sous-réseaux d'un réseau étendu.

Le type de transmission concerne à la fois la couche physique (nous avons vu comment les bus permettaient la diffusion) et la couche liaison. En effet la transmission de bits n'est pas suffisante à cause de l'indice que le message doit contenir quant à l'adresse du destinataire dans le cas d'un réseau à diffusion. Nous verrons qu'on utilise les trames pour résoudre ce problème : l'adresse est un des champs de l'en-tête de chaque trame.

Une transmission point-à-point entre un expéditeur et un destinataire est appelée **diffusion individuelle** (*unicast sending* en anglais).

2.4 Réseaux locaux

2.4.1 Technologie des réseaux locaux

Dans un réseau local les divers composants ont été reliés successivement dans le temps par le port série, un câble coaxial puis maintenant une paire torsadée **Ethernet.**

Les diverses technologies employées font que chaque élément ne doit pas être éloigné de l'élément auquel il est relié directement de plus de quelques centaines de mètres, comme l'indique le tableau suivant :

Câble	Longueur maximum
série	15 m
coaxial fin	185 m
10base T (paire torsadée)	100 m
coaxial gros (jaune)	500 m

On comprend que les composants d'un réseau local sont rassemblés dans une même pièce ou un même immeuble. Il s'agit quelquefois même de la définition d'un réseau local, qui peut alors utiliser des supports physiques réservés aux réseaux étendus (la fibre optique en particulier).

Tous les composants d'un réseau local sont interconnectés par le biais d'un type unique de câble réseau.

2.4.2 Topologie des réseaux locaux

La disposition du réseau, c'est-à-dire les composants eux-mêmes et la façon dont ils sont interconnectés, est appelée la **topologie du réseau.**

Bien qu'il existe de nombreuses topologies de réseaux locaux, on en compte trois principales : machine à machine, en bus, en anneau et en étoile.

2.4.2.1 Réseau machine à machine

Le **réseau machine à machine** (*peer-to-peer* en anglais) est le plus simple de tous. Un ordinateur est relié à un de ses voisins en utilisant un **câble Eternet fin** (*Thin Ethernet* en anglais, semblable au câble coaxial utilisé pour la télévision) ou un câble à paires torsadées (semblable aux câbles téléphoniques). Ce type de réseau consiste en une certaine longueur de câble coaxial qui court d'une machine à une autre. Chaque composant est connecté directement au câble à l'aide d'un connecteur en forme de T sur la carte interface du réseau, appelé **connecteur BNC** (pour *Bayonet Neill Concelman*). Celui-ci connecte la machine à deux de ses voisins au moyen d'un câble pour chaque voisin. Aux extrémités du réseau, là où il n'y a qu'un seul voisin, on ajoute une terminaison sur l'un des connecteurs en T. La figure suivante ([PAR–96], p. 7) montre le schéma de ce type de réseau :

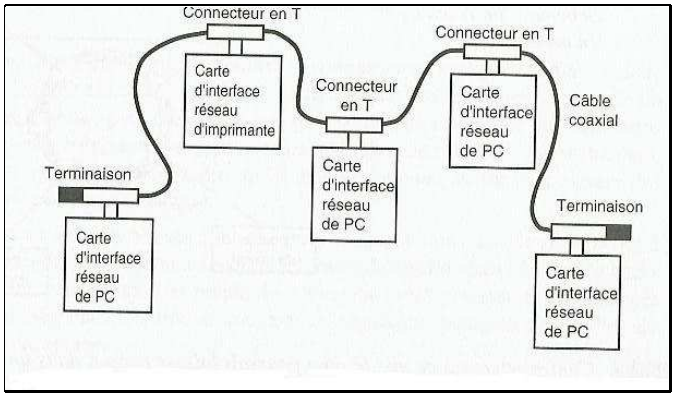


FIG. 2.5 – *Réseau machine à machine*

L'avantage des réseaux machine à machine tient à leur simplicité. Pour lui ajouter une nouvelle machine, il suffit d'installer une carte réseau et de connecter la nouvelle machine à un emplacement logique sur la dorsale. Ils ont aussi l'avantage d'être peu coûteux : c'est la topologie de réseau local la moins chère.

L'inconvénient principal de cette topologie est que si une machine est coupée du câble réseau, si la carte réseau ne fonctionne plus, ou si l'ordinateur est tout simplement éteint, la dorsale est coupée et doit être remise en place d'une façon ou d'une autre pour que le réseau fonctionne à nouveau.

De plus ce type de réseau n'est pas capable d'atteindre la vitesse des topologies suivantes en raison du câble réseau utilisé. Le débit des réseaux machine à machine fut longtemps limité à 10 Mbps (Mégabits par seconde). Des solutions telles que **Ethernet rapide** (*Fast Ethernet* en anglais) permettent des débits allant jusqu'à 100 Mbps.

2.4.2.2 Réseau en bus

Le **réseau en bus** permet de résoudre le principal inconvénient du réseau machine à machine. Il existe un unique chemin de communication principal, appelé **bus** ; chaque composant est rattaché au bus (matérialisé par un câble) par le biais d'un périphérique actif appelé **transmetteur/récepteur**, en abrégé **transcepteur** (*transceiver* en anglais) ou boîte de jonction. Le bus est également appelé une **dorsale** (*backbone* en anglais) car sa topologie fait penser à une épine dorsale. De chaque transcepteur du bus part un autre câble (très court la plupart du temps) qui se connecte à la carte réseau du composant. La figure 2.6 ([PAR–96], p. 6) montre le schéma d'un

réseau en bus. Chaque extrémité du bus est terminée par une résistance, appelée **terminaison**, qui marque électriquement la fin du bus. Chaque composant du chemin principal dispose d'un nombre identificateur – ou **adresse** – spécial qui lui permet de savoir si les informations qui lui arrivent lui sont bien destinées.

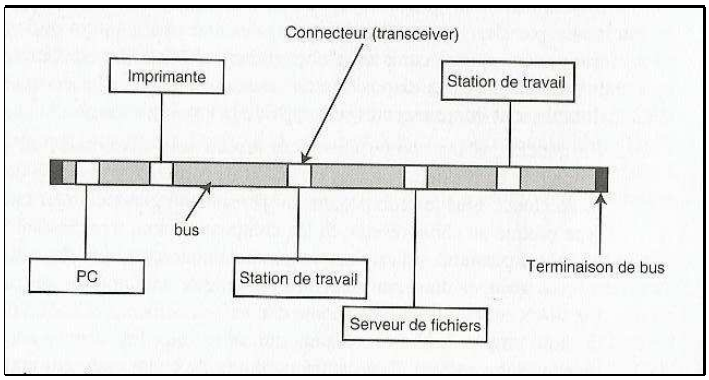


FIG. 2.6 – *Réseau en bus*

Le principal avantage de cette topologie est qu'elle permet un bus à haute vitesse. Autre avantage, le réseau en bus n'est généralement pas affecté si la carte réseau d'un des composants a des problèmes. En effet, le transcepteur assure l'intégrité du trafic à travers la dorsale, qu'un composant lui soit attaché (et est allumé) ou non.

Pour sa mise en place effective, un réseau en bus est rarement composé d'un câble rectiligne. Il suit les contours des murs et la disposition des immeubles en fonction des besoins. Ce qui résulte de la topologie est que son chemin est unique et les deux extrémités en sont marquées par un dispositif de terminaison. La figure ci-dessus montrait une représentation logique du réseau, en n'utilisant que des lignes droites et en ne respectant pas l'échelle.

2.4.2.3 Réseau en anneau

Comme son nom le suggère, un **réseau en anneau** (*ring* en anglais) est souvent représenté sous la forme d'un anneau, comme le montre la figure 2-7 ([PAR–96], p. 8).

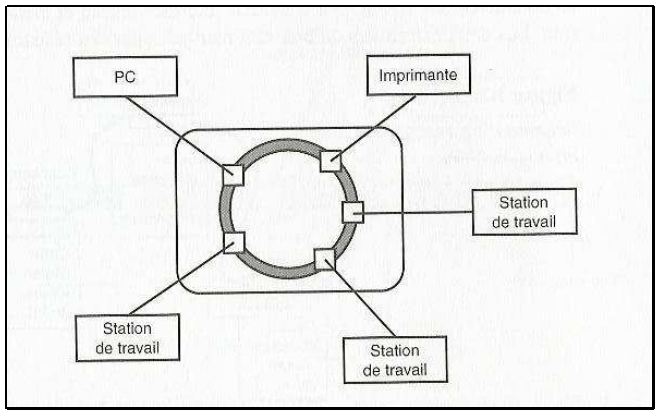


FIG. 2.7 – *Réseau en anneau*

Contrairement à ce que le nom pourrait laisser penser, dans un réseau en anneau les extrémités de la dorsale ne se rejoignent pas pour former une boucle ; le terme d'anneau s'applique à l'unité de contrôle centrale, qui dépend de la couche liaison et non de la couche physique.

2.4.2.4 Réseau en étoile

Un **réseau en étoile** utilise un câble principal, très proche de celui en bus, appelé **dorsale** (*backplane* en anglais) ; un ensemble de câbles va de la dorsale aux **concentrateurs** (*hub* en anglais), qui sont des boîtes contenant plusieurs **ports** dans lesquels sont branchés les composants. Les câbles menant à un point de connexion sont souvent appelés des **câbles de descente** (*drop* en anglais) car à l'origine ils descendaient des faux-plafonds. La figure 2.8 montre le schéma d'un réseau en étoile ([PAR–96], p. 9).

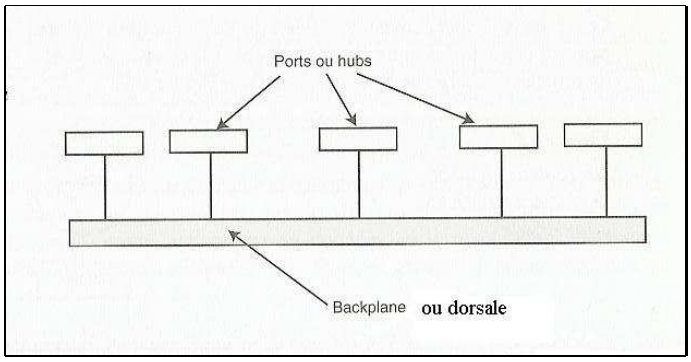


FIG. 2.8 – *Réseau en étoile*

Les réseaux en étoile peuvent atteindre de grandes tailles, comportant alors une dorsale à fibre optique et des câbles des descente Ethernet moins rapides. Ils peuvent également être plus petits, ne comporter qu'un seul concentrateur prenant en charge quelques composants interconnectés au moyen de câbles Ethernet standard. Le réseau en étoile est facilement extensible, il peut donc être étendu au fur et à mesure des besoins, ce qui n'est pas un mince avantage.

Les réseaux en étoile sont souvent utilisés dans les grandes installations, car ils sont faciles à mettre en place et à entretenir. Le coût en est réduit.

2.5 Historique

2.5.1 Émergence de réseaux de recherche

Le premier réseau, ARPANET, s'est rapidement développé comme le montre la figure 2.9 ([TAN–81], p. 58).

Les années 1970 ont vu l'émergence de nombreux réseaux à commutation de paquets autres que ARPANET, chacun d'eux au service d'une communauté scientifique, tels que **CSNET** (entre informaticiens), **BITNET** (entre professeurs d'économie et de science du comportement) et une myriade de petits réseaux de recherche situés dans d'autres pays que les États-Unis.

À la fin des années 1970, la NSF (*National Science Foundation*) a réalisé combien l'impact d'ARPanet sur la recherche universitaire était grand, puisqu'il permettait aux scientifiques de partager des données et de travailler sur des projets communs à travers tout le pays. Toutefois, pour pouvoir participer à ce réseau, une université devait être sous contrat avec le DoD (*Department of Defence*), ce qui n'était pas le cas pour beaucoup d'entre elles. La NSF a donc décidé de

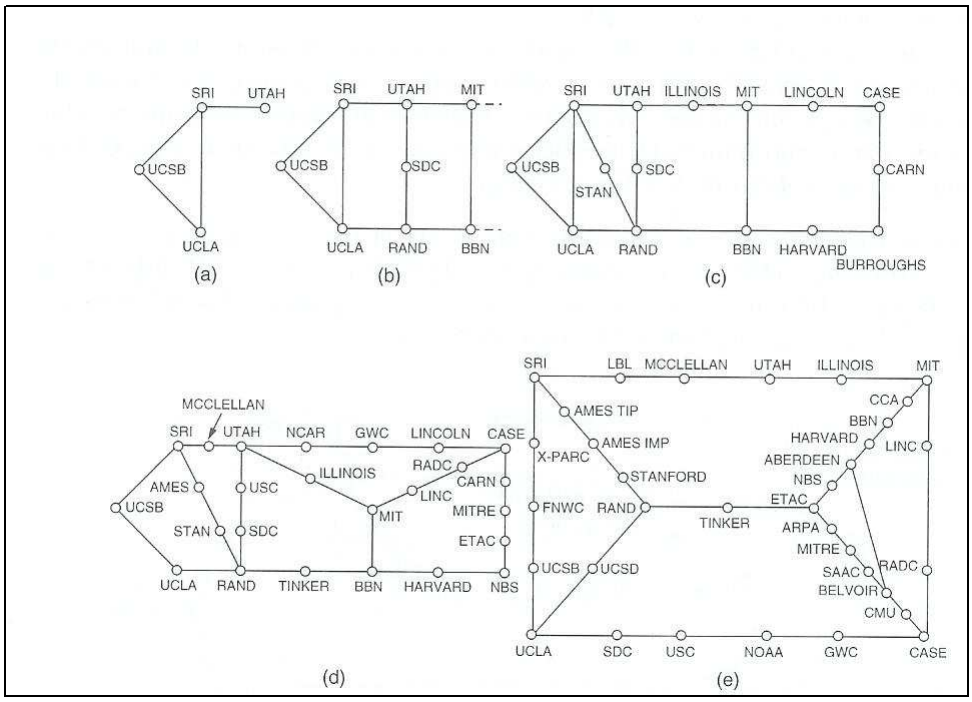


FIG. 2.9 – *Croissance d'Arpanet : (a) décembre 1969 (b) juillet 1970 (c) mars 1971 (d) avril 1972 (e) septembre 1972*

créer un réseau fédérateur pour connecter les six centres de calcul basés à San Diego, Boulder, Champaign, Pittsburgh, Ithaca et Princeton. À chacun des six supercalculateurs était associé un micro-ordinateur LSI-11 ou **fuzzball.** Ces fuzzballs étaient reliés au moyen de liaisons louées à 56 kbit/s. Sur le plan logiciel, les fuzzballs supportaient dès le départ TCP/IP.

La NSF a également financé plusieurs réseaux régionaux (une vingtaine au total), qui ont été reliés à l'épine dorsale pour permettre aux utilisateurs de milliers d'universités, de laboratoires de recherche, de bibliothèques et de musées d'accéder à n'importe quel supercalculateur et de communiquer entre eux. Le réseau complet, incluant l'épine dorsale et les réseaux régionaux, a été nommé **NSFNET**. Il a été raccordé à ARPAnet au moyen d'une liaison entre un IMP et un fuzzball dans la salle des machines de l'université de Carnegie-Mellon. La figure 2.10 ([TAN–81], p. 60) illustre l'épine dorsale NSFNET initiale.

2.5.2 Naissance des réseaux à diffusion : ALOHANET

Norman Abramson et ses collègues de l'université d'Hawaï tentaient de relier les utilisateurs d'îles éloignées à l'ordinateur central situé à Honolulu au début des années 1970. Cependant l'archipel ne disposait pas encore d'un système téléphonique opérationnel.

L'option de tirer des câbles sous l'océan Pacifique paraissant peu envisageable, ils ont cherché une autre solution. Ils en sont venus à utiliser des ondes radio de faible portée. Chaque terminal utilisateur était équipé d'un petit dispositif radio à deux fréquences : l'une montante (vers l'ordinateur central) et l'autre descendante (depuis l'ordinateur central). Lorsqu'un utilisateur souhaitait contacter le central, il transmettait simplement un paquet contenant les données sur le canal montant. Si personne d'autre ne transmettait au même moment, le paquet était acheminé

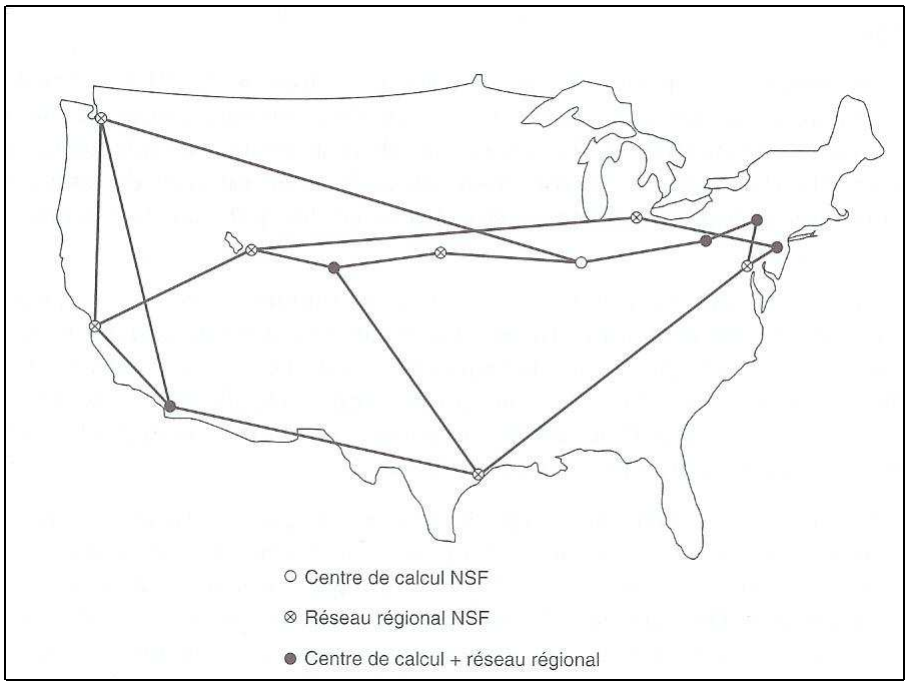


FIG. 2.10 – *Épine dorsale NSFNET en 1988*

vers sa destination, puis était acquitté par voie descendante. Lorsqu'une situation de conflit se présentait pour l'utilisation du canal montant, le terminal remarquait l'absence d'acquittement et essayait à nouveau d'émettre. Il ne pouvait pas y avoir de conflit sur le canal descendant puisque seul l'ordinateur central y émettait. Ce système, baptisé ALOHANET, fonctionnait assez bien dans des conditions de faible fréquentation mais réagit par contre très mal à l'intensification du trafic montant [ABR–70, ABR–85].

2.5.3 Naissance des réseaux locaux : Ethernet

2.5.3.1 Le PARC

À peu près à la même époque, un étudiant nommé Bob Metcalfe partait décrocher un doctorat à Harvard après avoir obtenu son diplôme de licence au MIT. Confronté aux travaux d'Abramson durant ses études, il les a trouvés si intéressants qu'une fois sorti de Harvard, il a décidé de passer l'été à Hawaï avec le chercheur avant d'intégrer le centre de recherches de Palo Alto (PARC, *Palo Alto Research Center*) de Xerox. À son arrivée au PARC, il a découvert que les chercheurs avaient conçu et construit un ordinateur personnel. Mais les machines étaient isolées. Mettant à profit sa connaissance des travaux d'Abramson, Metcalfe et son collègue David Boggs ont développé et implémenté le premier réseau local [M-R–76].

Ils l'ont appelé `Ethernet` d'après l'éther luminifère à travers lequel les scientifiques ont pensé à une époque que les ondes se propageaient. Le support de transmission n'était pas ici le vide mais un câble coaxial épais (l'Ether) long de 2,5 km avec des répéteurs tous les 500 m. On pouvait relier 256 machines au système en même temps *via* des transcepteurs fixés au câble à l'aide d'une vis comme le montre la figure 2.11 ci-dessous ([TAN–81], p.72).

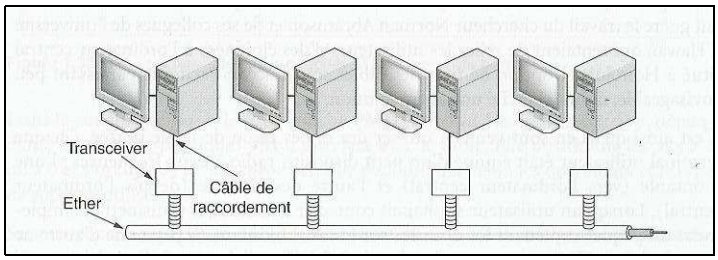


FIG. 2.11 – *Architecture du premier réseau Ethernet*

2.5.3.2 3Com

La société Xerox était déjà connue par ses inventions originales qu'elle ne commercialisait jamais, par exemple l'ordinateur personnel dont nous avons parlé ci-dessus. Comme elle ne semblait intéressée que par la normalisation d'Ethernet, Metcalfe a fondé sa propre entreprise, **3Com**, pour commercialiser des adaptateurs Ethernet pour les micro-odinateurs. Il en a vendu plus de 100 millions.

2.5.3.3 Autres types de réseaux locaux

En 1977, la Datapoint Corporation lance sur le marché le système de réseau local **ARCnet.** En 1979, la société Ungermann-Bass Inc. est fondée par Ralph K. Ungermann et Charlie Bass, à Santa Clara (Californie), en vue du développement et de la vente de systèmes de réseaux locaux. Son produit Net/one sera lancé en 1980. En 1980, la société Novell Data Systems Inc. est créée à Provo (Utah).

2.5.4 Interréseaux

Vinton Cerf a formulé l'idée de relier entre eux les divers réseaux afin de constituer un vaste réseau mondial, qui va devenir **Internet.** La DARPA s'en charge en se fondant sur l'architecture d'ARPANET.

Chapitre 3

Vue d'ensemble sur la couche liaison

La couche liaison de données a pour but de permettre à deux machines adjacentes dans un réseau de communiquer de façon fiable et efficace. Par **machines adjacentes**, il faut entendre que les deux machines sont physiquement connectées par un canal de transmission (câble coaxial, ligne téléphonique...) qui délivre les bits dans l'ordre dans lequel ils ont été émis.

Dans le cas d'un **canal de communication parfait**, la couche liaison de données n'a rien à faire : la machine A se contente d'émettre ses données sur le support physique, et la machine B de les collecter. Malheureusement, un canal de communication présente en général deux défauts :

- il est **bruité**, ce qui a pour conséquence des erreurs de transmission ;
- les deux machines n'ont pas le même débit, ce qui a pour conséquence qu'il faut réguler les flux de données échangées pour éviter que les destinataires lents ne soient submergés par des expéditeurs rapides, tout en essayant d'être le plus rapide possible ; on parle de **régulation** (ou **contrôle**) **des flux.**

3.1 Détermination des trames

3.1.1 Paquet, trame et somme de contrôle

Pour être en mesure de fournir un service à la couche réseau, la couche liaison des données doit utiliser le service de la couche physique. Celle-ci assure le transport de flux de bits sur le support et leur remise à la station de destination. Ces flux de bits peuvent comporter des erreurs : le nombre de bits reçus peut être inférieur, égal ou supérieur au nombre de bits émis ; les bits peuvent de plus avoir changé de valeur.

Pour résoudre les problèmes liés aux erreurs de transmission, la couche liaison des données n'accepte d'émettre qu'un certain nombre d'octets à la fois, appelé **paquet** (*packet* en anglais). La taille (fixe ou plus usuellement comprise entre un minimum et un maximum) d'un paquet dépend du protocole de liaison utilisé.

De plus chaque paquet est encapsulé dans une **trame** (*frame* en anglais) : chaque trame comprend un **en-tête** de trame (*header* en anglais), un champ de données pour héberger le paquet et un **en-queue** de trame (*trailer* en anglais), comme le montre la figure 3.1. L'en-tête

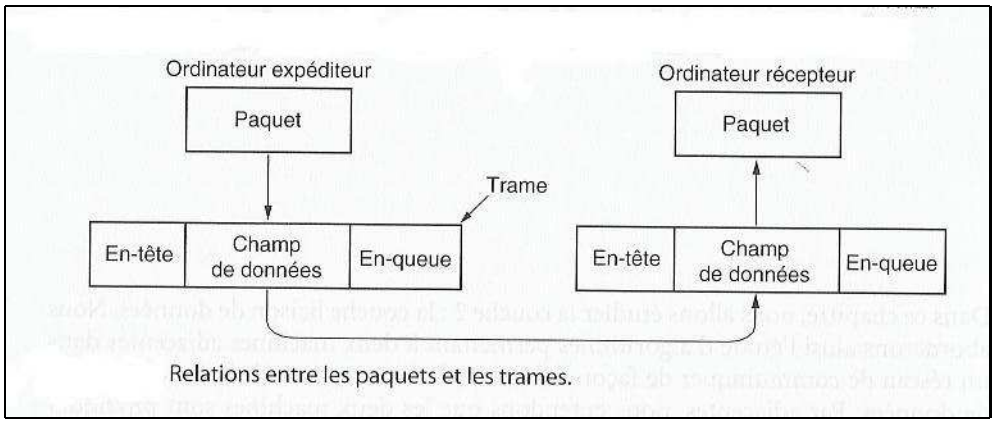


FIG. 3.1 – *Structure d'une trame*

et l'en-queue comportent des données de contrôle.

La donnée de contrôle la plus utilisée est la **somme de contrôle** (*checksum* en anglais) : la machine émettrice calcule un entier à partir des données suivant un algorithme bien précis qu'elle place parmi les donnée de contrôle ; lorsque la trame arrive à destination, la machine réceptrice calcule un entier suivant le même algorithme à partir des données reçues ; si le résultat obtenu est différent de celui calculé par la machine émettrice tel que reçu par la machine réceptrice, la couche liaison de données sait qu'une erreur de transmission s'est produite et peut prendre des mesures : ne pas tenir compte de la trame reçue et envoyer à l'émetteur un rapport d'erreur, par exemple. La façon de calculer la somme de contrôle dépend du protocole. Si le résultat est le même que celui calculé par la machine émettrice, on ne peut rien dire mais on considère que la trame reçue est identique à celle envoyée.

3.1.2 Découpage en trames

Le découpage en trames des flux de bits est plus difficile qu'il n'y paraît à première vue. Plusieurs méthodes ont été envisagées.

3.1.2.1 Insertion de silences

La première idée à laquelle on peut penser est d'insérer des silences pour délimiter les trames, exactement comme on place des espaces entre les mots. Malheureusement la couche physique garantit rarement les délais : il est donc possible que des silences disparaissent ou soient insérés dans les trames durant les transmissions.

3.1.2.2 Décomptage des octets

La seconde idée consiste à utiliser un champ dans l'en-tête de la trame qui indique le nombre d'octets que celle-ci contient. Lorsque la couche liaison de données de la station réceptrice lit ce champ, elle en déduit le nombre d'octets de la trame et donc la délimitation de celle-ci. Cette technique est illustrée à la figure 3.2 ([TAN–81], p. 204), qui montre quatre trames contenant

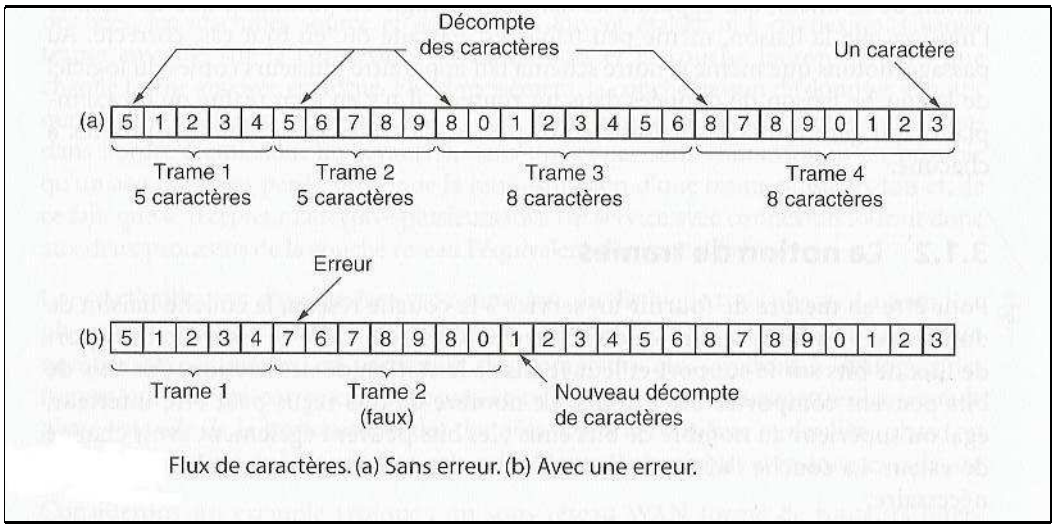


FIG. 3.2 – *Décomptage des octets*

respectivement 5, 5, 8 et 8 octets.

Cette méthode pose un problème lorsque l'octet indiquant le nombre d'octets de la trame est affecté par la transmission. Par exemple, si l'octet "nombre d'octets" de la seconde trame est changé de 5 en 7, le récepteur perd la synchronisation et ne peut plus trouver le début de la trame suivante. Grâce à la somme de contrôle, le récepteur se rendra certainement compte que cette trame est mauvaise, mais il ne sait pas où commence la suivante. S'il décide de demander à l'émetteur de retransmettre les données, celui-ci ne connaît pas le nombre d'octets à laisser passer pour trouver le début de la retransmission. Pour cette raison, cette méthode est rarement utilisée.

3.1.2.3 Fanions de signalisation avec caractères d'échappement

La troisième idée résout le problème de la **resynchronisation** après une erreur de transmission en délimitant chaque trame par des octets spéciaux de début et de fin. Par le passé, les protocoles employaient souvent des octets différents de début et de fin. À présent, ils utilisent le même octet, appelé **fanion de signalisation** (*flag* en anglais). Si le récepteur perd la synchronisation, il lui suffit de rechercher le fanion de signalisation pour trouver la fin de la trame (ou d'une trame suivante). Deux fanions consécutifs désignent la fin d'une trame et le début de la suivante.

Reste un problème de mise en œuvre. N'oublions pas que les données à transmettre sont binaires. Il se peut donc que le profil binaire du fanion de signalisation se présente dans les données, interférant avec l'encadrement. Pour résoudre ce problème, la couche liaison des données de l'émetteur insère un octet (caractère) spécial d'échappement (ESC) avant chaque fanion "accidentel". La couche liaison de données du récepteur élimine le caractère d'échappement avant de transmettre les données à la couche réseau. On peut ainsi distinguer l'encadrement du fanion de celui des données par l'absence ou la présence du caractère d'échappement qui le précède.

Ceci ne résout pas tout le problème de la mise en œuvre. La question suivante est : que se passe-t-il si un caractère d'échappement se présente dans les données binaires ? La mise en œuvre consiste à le faire précéder également d'un caractère d'échappement. Ainsi un caractère d'échappement seul désigne-t-il une séquence d'échappement, alors qu'un double caractère d'échappement indique qu'un échappement apparaît dans les données. La figure 3.3 montre quelques

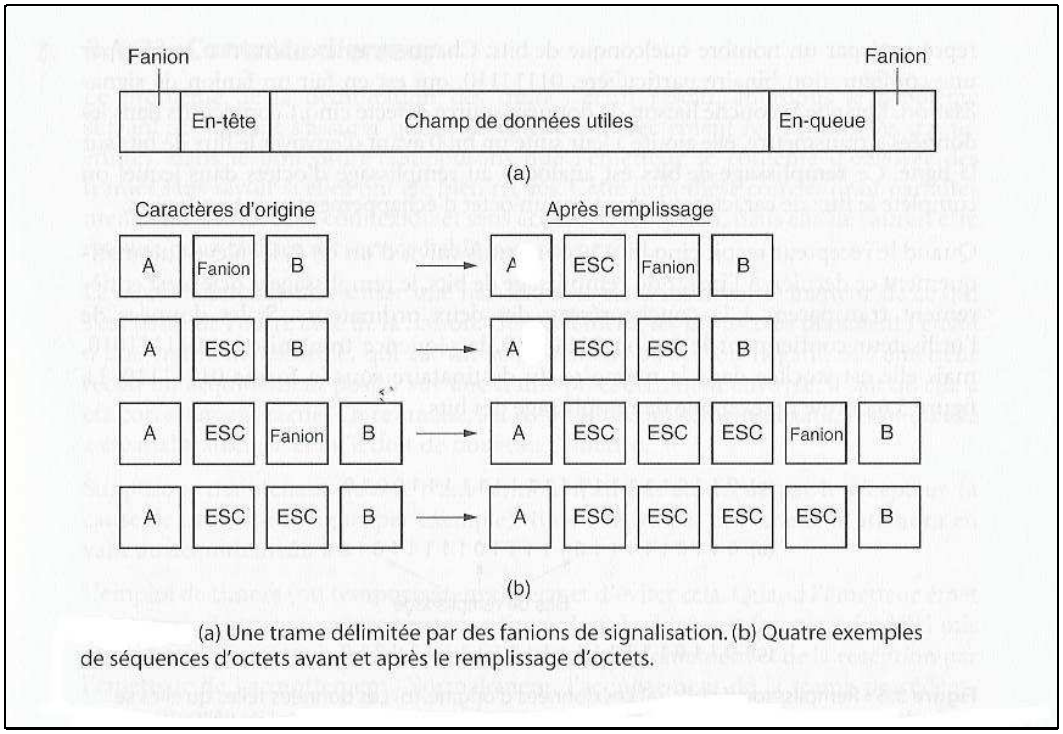


FIG. 3.3 – *Fanion de signalisation*

exemples de mise en place de cette **technique de remplissage.** Dans tous les cas, la séquence d'octets transmis après le **vidage** des caractères d'échappement est exactement identique à la

séquence d'octets d'origine.

3.2 Garantie de transmission correcte

Le problème de la délimitation des trames étant résolu, abordons le problème suivant : comment s'assurer que le récepteur a correctement reçu toutes les trames émises, dans le bon ordre?

La tâche essentielle est de garantir autant que faire se peut la communication en répétant la transmission de paquets de données erronés ou perdus. Cela ne peut pas se faire sans le concours du destinataire, car lui seul sait quels paquets sont déjà arrivés. Comment mettre ceci en place?

3.2.1 Première idée : confirmation ou infirmation

Le destinataire peut transmettre une **confirmation** (*Acknowledge* en anglais, abrégé en `ACK`) à son correspondant pour chaque paquet parvenu à bon port et dont la somme de contrôle a permis d'établir que ce paquet est *a priori* intact. En revanche, si la vérification donne lieu à la découverte d'une erreur, il envoie une **infirmation** (*Non-Acknowledge* dans l'anglais de TCP, abrégé en `NAK`).

L'expéditeur attend de recevoir un message de confirmation ou d'infirmation en provenance de son correspondant. Après réception de la confirmation, celui-ci expédie le paquet suivant. Dans le cas d'une infirmation, l'expéditeur reprend la transmission du dernier paquet jusqu'à ce qu'il obtienne une confirmation, avant de passer au paquet suivant, comme le montre la figure 3-4 ([TJ–97], p.139).

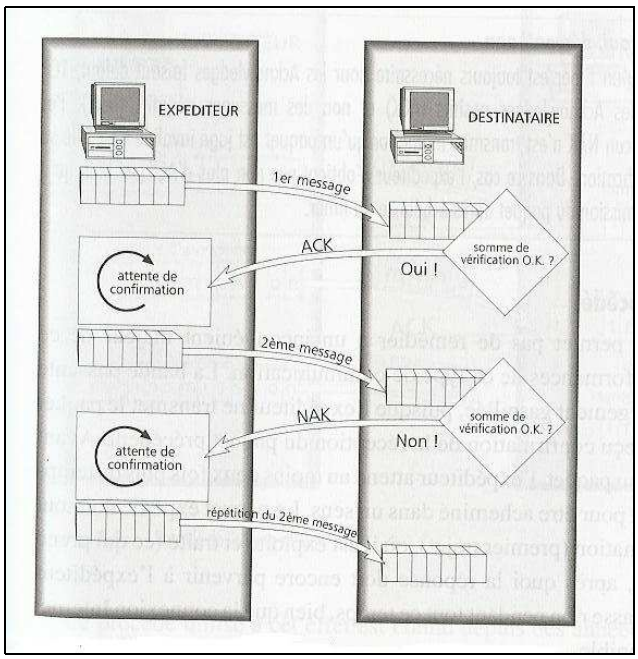


FIG. 3.4 – *Confirmation et infirmation*

Ce procédé ne peut pas être mis en place tel quel. En effet, qu'advient-il des messages non reçus, qui se sont perdus en cours de route?

3.2.2 Deuxième idée : confirmation et délai

Dans une version améliorée, l'expéditeur envoie son message et initialise un minuteur (*timer* en anglais). Il attend alors soit une confirmation, soit qu'un certain délai (*time out* en anglais) se soit écoulé. S'il reçoit la confirmation avant que le délai ne se soit écoulé, il envoie le message suivant. Sinon, une fois le délai échu, il considère qu'aucun message du destinataire ne lui parviendra et il transmet une nouvelle fois le message.

Si l'on transmet plusieurs fois le même message, il se peut que le récepteur l'accepte plus d'une fois et qu'il le délivre ainsi plusieurs fois. Pour éviter une telle situation, on numérote généralement les messages de telle sorte que le récepteur ne tienne pas compte des messages retransmis s'il n'y a pas lieu d'être. Ceci est un deuxième élément de contrôle (le premier étant la somme de contrôle).

Cependant l'utilisation d'un minuteur conduit à réduire fortement le débit. En effet la bande passante mobilisée est alors largement gaspillée puisque l'expéditeur ne transmet le paquet suivant qu'après avoir reçu confirmation de la réception du paquet précédent. Avant de transmettre un nouveau paquet, l'expéditeur attend au moins deux fois plus de temps qu'il n'en faut au paquet pour être acheminé dans un sens : le paquet expédié doit tout d'abord parvenir à destination (premier trajet), où il est exploité et traité (ce qui prend également un moment), après quoi la réponse doit encore parvenir à l'expéditeur (second trajet). Il ne se passe rien pendant tout ce temps, bien que la connexion soit intégralement disponible.

3.2.3 Troisième idée : fenêtre glissante

Pour remédier à ce gaspillage de la bande passante, on envoie plusieurs paquets avant d'attendre confirmation : trois, cinq ou huit, par exemple. On accorde ainsi au destinataire une sorte de crédit de confiance avant d'en attendre confirmation. Dès que la confirmation du premier paquet est reçue, le paquet suivant non encore expédié est envoyé sur la ligne, comme le montre la figure 3-5 ([TJ–97], p. 141). L'opération se répète ainsi pour les confirmations suivantes. Pour chaque nouvelle confirmation concernant un paquet précédent, le paquet suivant non encore transmis est expédié.

Il s'agit du procédé dit de la **fenêtre glissante** (*sliding window* en anglais). Il doit son nom au fait que l'expéditeur place une sorte de fenêtre sur le flux d'entités devant être expédié. Au début de la transmission, ce dernier envoie en premier lieu tous les paquets inclus dans cette fenêtre (d'expédition). À chaque confirmation, la fenêtre se décale sur la droite de manière à exclure le paquet confirmé situé sur la gauche. Cependant la fenêtre n'est pas réduite. Elle se déplace simplement sur la droite du flux de paquets devant être expédiés, incluant ainsi un nouveau paquet du côté droit de la fenêtre.

Pendant la durée de la transmission, la fenêtre d'expédition défile par conséquent de gauche à droite au-dessus du flux de paquets devant être transmis, tout en divisant celui-ci en trois groupes :

- tous les paquets situés à gauche de la fenêtre ont déjà été expédiés et confirmés ; l'expéditeur n'a plus à s'en occuper ;
- tous les paquets compris dans la fenêtre ont déjà été expédiés, mais pas encore confirmés ; leur nombre dépend de la largeur de la fenêtre ;
- tous les paquets situés à droite de la fenêtre attendent d'être inclus dans la fenêtre pour être envoyés ; plus les confirmations en provenance du destinataire parviennent rapidement à l'expéditeur, plus vite ces paquets se rapprochent de la fenêtre.

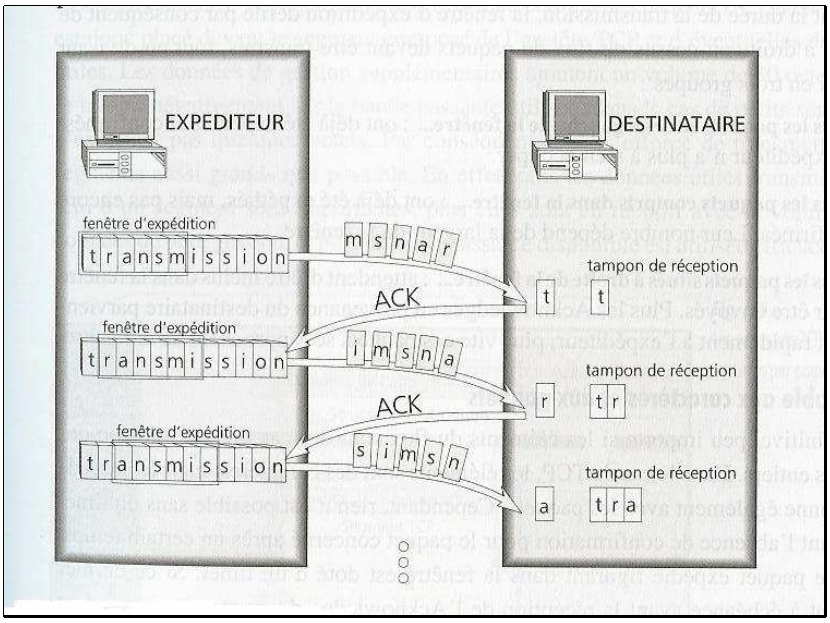


FIG. 3.5 – *Fenêtre glissante*

3.3 Contrôle de flux

Il se peut qu'un émetteur émette plus de messages que le récepteur ne peut en accepter. C'est le cas lorsque l'émetteur est sur un ordinateur rapide (ou peu chargé) et que le récepteur est sur une machine lente (ou très chargée). Dans ce cas, même si la transmission s'effectue sans erreur, le récepteur ne peut pas traiter toutes les messages émis et en perdra certains. Il faut donc mettre en œuvre un mécanisme pour éviter cette situation.

Il existe deux approches pour résoudre ce problème :

- La première consiste à instaurer un **contrôle de flux avec retour d'information**, ou rétroaction (*feed-back based flow control* en anglais), pour contraindre l'émetteur à ne pas envoyer plus de messages que le récepteur ne peut en accepter. L'émetteur peut ainsi savoir, grâce à un mécanisme de rétroaction, si le récepteur est en mesure ou non de recevoir les messages émis.
- Dans la seconde méthode, le **contrôle de flux basé sur le débit** (*rate based flow control* en anglais), un mécanisme intégré au protocole limite le débit de transmission des données, sans exploiter le retour d'information.

La seconde méthode n'est jamais employée dans la couche liaison de données.

Il y a de nombreuses variantes de contrôle de flux avec rétroaction. La plupart se fondent sur le principe suivant : il est interdit à l'émetteur d'envoyer des messages s'il n'a pas reçu auparavant une permission implicite ou explicite du récepteur. Par exemple, sur une réception déjà établie, le récepteur peut envoyer le message suivant : "Tu peux m'envoyer maintenant n messages, mais après ces n émissions, suspend tes envois jusqu'à ce que je te dise de continuer".

La couche liaison s'occupe de la régulation des flux entre deux machines adjacentes alors que, bien sûr, le problème le plus important est certainement la régulation des flux entre la machine

émettrice et la machine réceptrice (qui n'ont aucune raison d'être adjacentes). Cette régulation des flux-là est l'une des fonctionnalités éventuelles de la couche transport.

3.4 Gestion des canaux

Une fois que l'on a mis en place un support physique entre deux points, reste à choisir la meilleure façon de l'utiliser. On peut décider d'y envoyer les messages les uns à la suite des autres. Mais on s'est très vite aperçu que l'on ne rentabilise pas correctement l'utilisation du support physique dans ce cas, à cause des moments où il n'y a rien à dire. On préfère envoyer plusieurs messages à la fois, sur des **canaux** différents.

Il existe deux méthodes de détermination des canaux : l'allocation statique des canaux dans lesquels il ne peut pas y avoir collision entre deux messages sur le support donné et les **systèmes à contention** dans lesquels le partage d'un canal commun entre plusieurs utilisateurs peut engendrer des situations de conflits.

3.4.1 Allocation statique des canaux

Dans le cas de l'allocation statique d'un canal, on permet à plusieurs messages de passer dans le même support physique au même instant en **multiplexant** les messages, un par canal. À l'arrivée, il faut **démultiplexer** les messages. Il existe deux techniques fondamentales de multiplexage : le multiplexage en fréquence et le multiplexage temporel.

3.4.1.1 Multiplexage en fréquence

La méthode de partage d'un canal entre plusieurs usagers choisie à l'avénement du réseau téléphonique est le **multiplexage en fréquence** ou **FDM** (pour l'anglais *Frequency Division Multiplexing*). Les messages sont transportés à travers un support physique sous la forme d'une onde. L'ensemble des fréquences possibles des ondes pour un support physique donné est un intervalle, appelé **bande passante**. Lorsqu'il y a N utilisateurs, la bande passante est divisée en N portions de même taille, et chacune est affectée exclusivement à l'un d'eux. Il ne se produit donc pas d'interférences entre les différentes communications.

Lorsque le nombre des usagers est constant et peu élevé, et que chacun d'eux est la source d'un fort trafic, FDM offre un mécanisme simple et efficace de multiplexage sur un support de transmission partagé. Toutefois, lorsque le nombre d'émetteurs est élevé et varie sans cesse, ou que le trafic est sporadique, FDM pose quelques problèmes. Pour commencer, si le spectre des fréquences disponibles est divisé en N portions et, qu'à un instant donné, moins de N utilisateurs souhaitent transmettre, une grande partie des ressources est gaspillée. Ensuite, si plus de N nœuds souhaitent communiquer, certains ne seront pas autorisés à le faire par manque de bande passante, même si parmi les émetteurs bénéficiaires d'une bande, celle-ci est peu ou pas du tout exploitée.

3.4.1.2 Multiplexage temporel

Dans le cas d'un **multiplexage temporel**, ou **TDM** pour l'anglais *Time Division Multiplexing*, on alloue à chaque utilisateur un intervalle de temps fixe qu'il voit se renouveler tous les N intervalles de temps.

Les arguments contre FDM sont également valables contre TDM. Comme aucune méthode d'allocation statique n'est satisfaisante avec un trafic sporadique, on a pensé à des méthodes dynamiques.

3.4.2 Gestion des canaux à accès multiples

Nous avons vu que les réseaux peuvent être divisés en deux catégories selon le mode de communication qu'ils utilisent : mode en point-à-point ou mode à diffusion. Dans le cas d'un réseau à diffusion, dans lequel tous les participants entrent en compétition pour utiliser un canal partagé, on doit déterminer qui, à un instant donné, est autorisé à émettre.

Les canaux de diffusion sont parfois désignés par les termes **canaux à accès multiple** ou **canaux à accès aléatoire**. Voyons les méthodes de gestion de ce type de canal.

3.4.2.1 ALOHA pur

Bien que le système ALOHA d'origine se fondait sur la diffusion radio terrestre, son principe reste applicable à n'importe quel autre système dans lequel des utilisateurs non coordonnés se disputent l'usage d'un canal unique.

L'idée de base du système ALOHA est simple : laisser les utilisateurs accéder librement au canal lorsqu'ils ont des données à transmettre. Dans ces conditions, il est clair que des **collisions** se produisent et que les trames qui en sont victimes sont détruites. Toutefois la technologie utilisée permet toujours à un émetteur de savoir si une trame a été ou non victime d'un incident. Lorsqu'une collision se produit, les participants en cause attendent un laps de temps aléatoire avant de tenter une nouvelle transmission. Si la durée d'attente doit être aléatoire, c'est pour éviter que les mêmes trames se percutent de façon répétitive, ce qui conduirait à une situation de blocage.

3.4.2.2 ALOHA discrétisé

En 1972, Larry Roberts publiait une méthode pour doubler la capacité d'un système ALOHA. Sa proposition était de diviser le temps en **slots**, ou intervalles finis, chacun d'eux correspondant à une trame. Cette approche demande que les utilisateurs s'accordent sur la délimitation des slots. Une façon de procéder à la synchronisation des stations était de disposer d'une station spéciale chargée d'émettre un bit au départ de chaque slot, telle une horloge. Chaque station doit attendre le début du prochain slot pour émettre. Le système ALOHA est ainsi transformé en système discret.

Ce système a été utilisé dans quelques sytèmes expérimentaux, puis pratiquement oublié. Lorsque l'accès à internet par le câble fut possible, ALOHA discrétisé a été sorti de son carton.

3.4.2.3 Protocoles d'accès par écoute de porteuse CSMA

Le problème avec ALOHA est qu'un système où des stations transmettent à volonté, sans égard pour l'activité des autres stations, est évidemment propice à de nombreuses collisions. Sur un réseau local, la technologie est telle que chaque participant peut détecter l'activité des autres stations et adapter son comportement en conséquence. Les protocoles qui permettent aux stations d'adapter leurs décisions à l'activité en cours sont appelés **protocoles à détection de porteuse** (CSP pour l'anglais *Carrier Sense Protocol*).

Le premier des protocoles à détection de porteuse est appelé **CSMA 1-persistant** (pour l'anglais *Carrier Sense Multiple Access*). Quand une station doit expédier des données, elle commence par écouter le support de transmission pour savoir si une autre station n'est pas déjà en train de transmettre. Si c'est le cas, elle attend que le canal se libère. Quand il est de nouveau disponible, elle transmet sa trame. Lorsqu'une collision se produit, l'émetteur d'une trame observe un temps de pause aléatoire, puis recommence la procédure depuis le début. Le nom de ce protocole vient du fait que la probabilité qu'une station souhaitant transmettre trouve le canal disponible est égale à 1.

ALOHANET ne pouvait pas fonctionner de cette manière car un terminal sur une île était dans l'impossibilité de déterminer si un terminal sur une autre île était ou non en train d'émettre.

3.4.2.4 Le protocole d'accès multiple avec détection de collision

Un autre problème survient néanmoins lorsque, après avoir attendu la fin de la transmission en cours, deux ou plusieurs ordinateurs décident d'émettre simultanément. La solution est de faire en sorte que chaque machine écoute pendant qu'elle-même transmet et alerte tous les autres émetteurs si elle détecte une interférence. Chacun attend ensuite pendant une période de temps aléatoire avant d'émettre à nouveau. Si une nouvelle collision survient, l'intervalle aléatoire est doublé (et le sera à chaque collision successive) pour donner à l'une des machines l'occasion d'émettre en premier. On parle alors de **CSMA/CD** pour *CSMA with Collision Detection*.

Chapitre 4

Ethernet

Prenons comme exemple des couches physique et de liaison, non prises en compte par l'architecture TCP/IP rappelons-le, celui d'Ethernet.

4.1 Vue générale sur Ethernet

Nous avons déjà vu comment le protocole Ethernet est né, à propos de la naissance des réseaux locaux.

L'Ethernet de Xerox a eu tellement de succès que DEC, Intel et Xerox ont défini en 1978 un standard Ethernet à 10 Mbits/s, appelé le standard **DIX** d'après les initiales des protagonistes. Après deux modifications mineures, il est devenu en 1983 la norme IEEE 802.3, puis plus tard une norme ISO.

Ethernet a poursuivi son évolution. De nouvelles versions à 100 Mbit/s, 1 000 Mbit/s et plus ont vu le jour. Son câblage a été amélioré et la commutation ainsi que d'autres fonctionnalités ont été ajoutées.

4.1.1 Couche physique : types de câblage Ethernet

Le terme "Ethernet" se réfère au câble (l'éther), qui concerne la couche physique. Quatre types de câbles sont généralement installés, décrits dans le tableau de la figure 7.1.

- Le câble **10Base5**, appelé plus familièrement **Ethernet épais**, est apparu le premier. Il ressemble à un tuyau d'arrosage jaune, avec des repères tous les 2,5 m pour indiquer l'emplacement des prises de raccordement du câble d'interface (la couleur jaune n'est qu'une suggestion, non une exigence de la norme 802.3). Les raccordements au câble se

Nom	Type	Longueur maximale d'un segment	Nombre de nœuds par segment	Remarques
10Base5	Coaxial épais	500 m	100	Câble originel, maintenant obsolète
10Base2	Coaxial fin	185 m	30	Pas de hub nécessaire
10Base-T	Paire torsadée	100 m	1024	Système le moins cher
10Base-F	Fibre optique	2000 m	1024	Le mieux adapté pour relier les immeubles

FIG. 4.1 – *Types de câblages Ethernet*

font généralement au moyen de **prises vampires** dont une broche est enfoncée avec beaucoup de précaution à mi-course dans le cœur du câble coaxial. La désignation 10Base5 est une référence mnémotechnique qui signifie que le câble offre un débit de 10 Mbit/s, utilise une signalisation en bande de base et peut supporter des segments d'une longueur maximale de 500 m.

- Le deuxième type de câble à apparaître fut **10Base2**, ou **Ethernet fin**, qui présente une meilleure souplesse au pliage que le câble épais. Sur ce câble, les connexions se font au moyen de **connecteurs BNC** standard pour former les jonctions de la forme d'un "T", plus faciles à utiliser et plus fiables que les prises vampires. L'Ethernet fin est plus simple à installer et beaucoup moins onéreux, mais la longueur maximale d'un segment est de 185 m, et chaque segment n'accepte que 30 stations.
- Avec ces deux types de câbles, la détection des ruptures de câble, des longueurs excessives, des mauvais raccordements ou des connecteurs mal fixés est très difficile. Ces problèmes ont conduit à une autre méthode d'interconnexion, dans laquelle chaque station est reliée par un câble à un dispositif central appelé **concentrateur** (*hub* en anglais). Le raccordement se fait généralement au moyen de simples câbles téléphoniques à paires torsadées. Cette technique de câblage est appelée **10Base-T** (avec 'T' pour l'anglais *twisted* mais également valable pour la traduction française 'torsadé'). On utilise alors des prises **RJ45.**
- La quatrième option de configuration pour Ethernet, le **10Base-F** à base de fibre optique, est une solution plus onéreuse en raison du prix des connecteurs et des terminateurs mais qui offre une excellente immunité contre le bruit. Elle offre par ailleurs une meilleure confidentialité que le cuivre, car l'écoute clandestine n'est pas facile à pratiquer avec la fibre optique.

4.1.2 Couche de liaison : détection des collisions

Les connexions des trois premiers types de câblages Ethernet sont schématisées sur la figure 4.2 ([TAN–81], p.295). La détection des collisions est liée à ces connexions :

- Pour le câble 10Base5, on utilise un connecteur spécial appelé **transcepteur** (*transceiver* en anglais), qui est fermement pincé autour du câble pour provoquer un contact électrique avec le cœur du câble. Le transcepteur contient une carte électronique se chargeant de l'écoute des porteuses et de la détection des collisions. Lorsqu'une collision est détectée, il envoie un signal spécial sur le câble pour que les autres transcepteurs soient informés de l'incident.

  Un **câble transcepteur**, ou **cordon AUI** (pour l'anglais *Attachment Unit Interface*), relie le transcepteur à la carte réseau de l'ordinateur. Ce cordon peut avoir une longueur

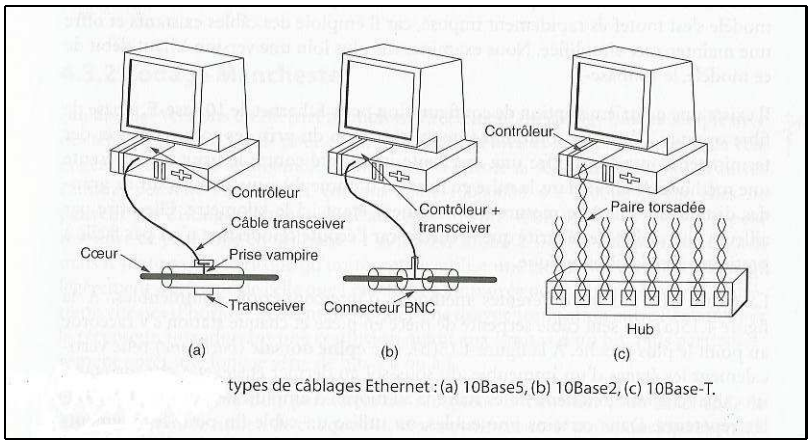


FIG. 4.2 – *Types de connexions Ethernet*

maximale de 50 m et renferme cinq paires torsadées blindées. Deux paires servent respectivement à l'envoi et à la réception des données. Deux autres sont utilisées pour les signalisations entrante et sortante. La cinquième paire, qui n'est pas toujours exploitée, permet à l'ordinateur d'alimenter l'électronique du transcepteur. Certains transcepteurs permettent même le raccordement de huit ordinateurs voisins pour limiter le nombre de prises.

Le câble transcepteur aboutit à une **carte réseau** (ou **carte Ethernet**), qui est l'une des cartes additionnelles à la carte mère de l'ordinateur. Cette carte réseau contient un circuit contrôleur qui se charge de la transmission des trames au transcepteur et d'en recevoir de sa part. Le contrôleur reçoit un paquet de la part de la couche réseau et est responsable de l'assemblement des données dans le format de trame adéquat, ainsi que du calcul des sommes de contrôle, aussi bien pour les trames sortantes que pour la vérification de l'intégrité des trames entrantes. Certains contrôleurs disposent également de zones de mémoire tampon servant au stockage temporaire des trames reçues et des trames en partance.

- Avec le câble 10Base2, le raccordement d'une station au câble se fait au moyen d'un connecteur BNC passif. L'électronique du transcepteur est alors intégrée à la carte contrôleur et chaque station dispose de son propre transcepteur.
- C'est le cas également du câblage 10Base-T.

4.2 Le standard IEEE

4.2.1 IEEE et les réseaux locaux

Nous avons vu que Ethernet, conçu à l'origine par Xerox, est devenu un standard IEEE. L'**IEEE** (*Institute of Electrical and Electronics Engineers*) est la plus grande institution professionnelle au monde. Elle publie de nombreuses revues, organise des centaines de conférences par an et conduit une section chargée de développer des standards dans les domaines de l'électronique et de l'informatique. La commission IEEE 802 a notamment standardisé plusieurs types de réseaux locaux.

4.2.1.1 Diversité des solutions

Ethernet est le plus célèbre des types de réseaux locaux standardisés par IEEE mais ce n'est pas le seul : il existe aussi, entre autres, les bus à jeton (*token bus*), l'anneau à jeton (*token ring*), tous les deux maintenant pratiquement abandonnés et surtout Wi-Fi (*Wireless Fidelity*, fidélité sans fil) et Bluetooth.

Le tableau ci-dessous présente les groupes de travail dans le cadre du comité IEEE 802 et l'avancement des travaux :

Numéro	Sujet	Avancement
802.1	Vue d'ensemble sur l'architecture des LAN	
802.2	Contrôle logique de liaison (LLC)	↓
802.3	Ethernet	*
802.4	Bus à jeton	↓
802.5	Anneau à jeton	
802.6	DQDB (premier MAN)	↓
802.7	Technologies à large bande	↓
802.8	Fibre optique	†
802.9	LAN en temps réel	↓
802.10	LAN virtuel (VLAN) et sécurité	↓
802.11	LAN sans fil (Wi-Fi)	*
802.12	AnyLAN de Hewlett-Packard	↓
802.13	Numéro non affecté par superstition	
802.14	Modem câble	↓
802.15	Réseaux personnels (PLAN, Bluetoooth)	*
802.16	Sans fil à large bande	*
802.17	RPR (Resilient Packet Ring)	

*: groupes les plus importants, ↓: groupes en suspens, †: groupes ayant abandonné.

4.2.1.2 Deux sous-couches de la couche liaison

Le standard IEEE 802 (figure 4.3, [WPRMB–02], p. 136) distingue deux sous-couches dans la couche de liaison des données [IEEE-802] :

- la **sous-couche de contrôle d'accès au canal**, ou **sous-couche MAC** (pour l'anglais *Medium Access Control*), ou encore sous-couche 2a, la plus proche de la couche physique comme son nom le laisse entendre, est chargée de la gestion des accès à un canal partagé.
- La **sous-couche de contrôle de liaison logique** (**LLC** pour *Logical Link Control*), ou sous-couche 2b, doit dissimuler les différences spécifiques aux supports et offrir une interface uniforme aux protocoles des couches supérieures. C'est cette sous-couche également qui s'occupe du contrôle de flux et des erreurs.

Ce découpage en deux sous-couches présente des avantages :

- Pour transporter des paquets IP, par exemple, aucune garantie de livraison n'est requise ni attendue. On n'a donc pas besoin d'utiliser la sous-couche LLC.
- La couche LLC est indépendante du type de réseau local : Ethernet ou WiFi.

Lors de l'implémentation :

- L'interface avec la couche 1 (couche physique) et la sous-couche 2a (sous-couche MAC) sont souvent implémentées dans les cartes réseau.

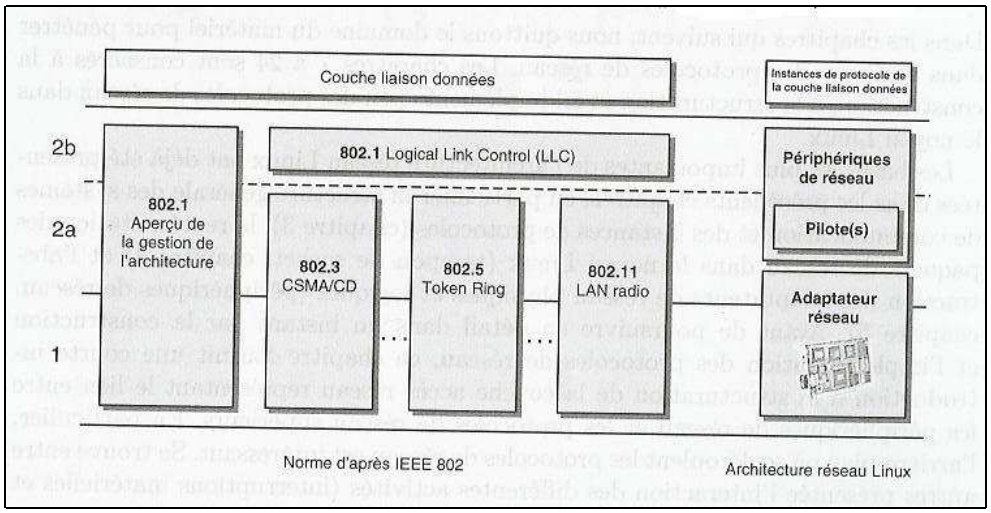


FIG. 4.3 – *Sous-couches MAC et LLC*

– La sous-couche 2b (LLC) est implémentée dans le sous-système réseau du noyau du système d'exploitation.

Deuxième partie

Vue d'ensemble sur TCP/IP

Cette seconde partie est, contrairement à la première, indispensable à la compréhension de l'implémentation du sous-système réseau de Linux. Elle regroupe les notions générales liées à cette dernière.

Chapitre 5

L'architecture TCP/IP

Nous avons abordé l'aspect matériel de la mise en place des réseaux au chapitres 2. Voyons maintenant ce qu'il en est de l'aspect logiciel. Nous avons vu au chapitre un qu'il existe deux grands modèles : OSI et TCP/IP. Ces modèles ne font que définir les fonctionnalités mais n'indiquent rien sur la réalisation de celles-ci. Il existe de nombreuses suites de logiciels réseau, mettant en œuvre tout ou partie de ces modèles. On peut citer, parmi les systèmes propriétaires, **XNS** (pour *Xerox Networking Systems*), **SNA** (déjà cité) et **NetBIOS** d'IBM et, parmi les systèmes ouverts, **UUCP** (pour *Unix-to-Unix Copy Protocol*) et l'architecture TCP/IP (souvent également appelée **architecture Internet** car la suite de protocoles TCP/IP et l'interréseau Internet sont étroitement liés).

Il ne faut pas confondre le modèle TCP/IP (qui est un modèle), l'architecture TCP/IP (suite de protocoles) et l'implémentation de TCP/IP sur tel ou tel système (suite de logiciels).

5.1 Vue d'ensemble

5.1.1 Historique

5.1.1.1 Origine : 1974

Lors des premières années d'utilisation du premier réseau, l'ARPANET, de nombreuses – et parfois vigoureuses – discussions portèrent sur la finalité et l'utilité du réseau, ce qui eut pour effet d'affiner et de modifier le logiciel réseau à mesure que les utilisateurs demandaient de nouvelles fonctionnalités.

Les deux fonctionnalités les plus demandées furent la possibilité de transférer des fichiers d'une machine à une autre et la possibilité de se connecter à distance. Les connexions à distance permettaient à un utilisateur de Santa Barbara de se connecter par le biais du réseau à une

machine située à Los Angeles, et d'utiliser cette dernière comme s'il se trouvait devant. Les logiciels et protocoles utilisés alors n'étaient pas en mesure de gérer ces nouvelles fonctionnalités. Il fallut donc continuellement développer, affiner et tester de nouveaux protocoles.

La connexion à distance fut finalement implémentée grâce à un protocole appelé **NCP** (déjà rencontré au chapitre 1) et le transfert de fichiers à distance au moyen du **FTP** (pour *File Transfer Protocol*).

Vers 1973, il devint clair que l'ensemble de protocoles utilisés n'était pas capable de gérer le volume de trafic et les nouvelles fonctionnalités requises par les utilisateurs. On s'attela au développement d'une nouvelle suite de protocoles. L'architecture TCP/IP fut proposée pour la première fois en 1974. L'article [CK–74] publié par Vincent Cerf et Kahn décrivait un système fournissant un protocole d'applications standardisé, qui prenait aussi en compte les besoins des utilisateurs.

Cerf et Kahn proposaient de plus que la nouvelle suite de protocoles soit indépendante du réseau sous-jacent et du matériel informatique. Il s'agissait d'une idée audacieuse dans un monde informatique où le logiciel et le matériel étaient propriétaires, car elles permettaient de faire participer n'importe quelle plate-forme au réseau. La suite de protocoles fut développée et reçut par la suite le nom de TCP/IP.

5.1.1.2 Envolée : 1982

Une série de RFC, dont nous reparlerons tout au long de ce livre, fut proposée en 1981 pour standardiser la version 4 de TCP/IP, plus particulièrement destinée à l'ARPANET. En 1982, TCP/IP a pris la place de NCP comme protocole dominant dans un réseau maintenant étendu, puisqu'il était composé de machines situées un peu partout aux États-Unis. On estime en effet qu'au cours de la première décennie d'existence d'ARPANET, un IMP y était raccordée tous les 20 jours.

TCP/IP devint important lorsque le Département de la Défense commença à inclure les protocoles de TCP/IP dans les standards militaires, qui étaient obligatoires dans de nombreux contrats.

5.1.1.3 L'implémentation BSD : 1983

La définition des protocoles, c'est bien, leur implémentation c'est mieux. L'université de Californie à Berkeley (UCB) reçut au début des années 1980 une subvention de la DARPA pour qu'elle modifie son système d'exploitation UNIX, connu sous le nom de **BSD** (pour *Berkeley System Distribution*), de manière à ce qu'il inclut la prise en charge d'IP. La version 4.2BSD sortit en 1983 avec une implémentation des quatres protocoles TCP, IP, SMTP et ARP.

Cette version de BSD fut mise dans le domaine public. Le succès de 4.2BSD entraîna celui de TCP/IP, lui-même dopé par le développement de l'ARPANET.

La prise en charge d'IP par 4.2BSD était fort bonne mais l'usage en était limité aux seuls petits réseaux locaux. Pour augmenter les capacités de prise en charge d'IP, BSD ajouta des capacités de retransmission, des informations de durée de vie (TTL pour *Time To Live*) et des messages de redirection. D'autres fonctions furent également ajoutées pour fonctionner avec des réseaux plus grands, des interréseaux et des systèmes étendus connectés par des lignes spécialisées. Ceci donna lieu à une version améliorée (qui contenait ce qu'on appelle les **utilitaires de Berkeley**, ou *Berkeley Utilities*) de son système en 1986, sous le nom de 4.3BSD. Une implémentation optimisée de TCP suivit en 1988 (4.3BSD/Tahoe). Pratiquement toutes les implémentation de TCP/IP actuellement disponibles plongent leurs racines dans les versions de Berkeley, même si BSD n'a pas connu de nouvelle version depuis 1993.

5.1.2 Définition des standards

5.1.2.1 Les RFC

Nous avons vu ci-dessus que relier un IMP à l'ARPANET était l'objet d'un protocole. Par contre rien n'était défini pour savoir comment un utilisateur (*hôte* dans le langage des réseaux) se reliait à l'IMP. En 1968 le réseau de l'ARPA commençait à se mettre en place. Plusieurs nœuds étaient opérationnels et cela commençait à se savoir. Au cours de l'été 1968, un petit groupe d'étudiants de second cycle s'est réuni à Santa Barbara; ils venaient des quatre sites hôtes – UCLA, le SRI, l'université de Californie à Santa Barbara et l'université de l'Utah. Ils savaient que le réseau était en préparation, mais n'avaient guère de détails. Il est sorti de cette réunion un corps de jeunes chercheurs qui allaient se consacrer aux communications d'hôte à hôte sur le réseau. Pour activer les choses, ils avaient décidé de se rencontrer régulièrement. Un mois après la formation du groupe, il est devenu évident qu'ils avaient tout intérêt à rassembler des notes sur leurs discussions. L'un d'entre eux, Crocker, s'offrit pour rédiger les premiers comptes rendus. Pour ne pas froisser les concepteurs officiels du réseau et pour éviter de paraître trop catégorique, il appela sa première note *Request for Comments* (ou **RFC**, demande de commentaires), et l'expédia le 7 avril 1969. Intitulée "*le logiciel de l'hôte*", la note [RFC 1] a été distribuée aux autres sites comme allaient l'être les premiers RFC : par la poste, dans une enveloppe en léchant le timbre.

"Demande de commentaires" s'est révélé un choix parfait pour un titre. Cela avait l'air à la fois attentif et sérieux. Et cela a tenu. Le langage du RFC était chaleureux et accueillant. L'idée était de favoriser la coopération, sans pontifier. Le fait que Crocker ait laissé son ego à l'écart lors du premier RFC a donné le ton et inspiré ceux qui ont suivi le mouvement dans les centaines de RFC amicaux et serviables qui sont venus après. Les RFC allaient devenir le principal moyen de libre expression chez les gens du réseau. Il y a maintenant plus de 3 000 RFC.

5.1.2.2 Le NWG

Bientôt, le collectif né au cours de l'été 1968 s'est désigné sous le nom de **Network Working Group** (ou **NWG**, le groupe de travail du réseau). Le défi qu'il fallait relever était de trouver un accord de principe sur les protocoles – comment partager les ressources, comment transférer les données, comment faire fonctionner le système. Cela signifiait écrire des programmes ou, du moins, adopter certaines règles sur la façon de les écrire, des règles qui recueilleraient une large adhésion.

L'accord était la condition *sine qua non*. On était en présence d'une communauté fondée sur l'égalité des compétences. Tout le monde pouvait écrire un code – ou réécrire le code déjà écrit par un autre. Le NWG était une "adhocratie", une aristocratie de spécialistes, une aristocratie égalitaire pour férus de l'informatique.

Devançant la construction du réseau, le NWG a continué à se réunir régulièrement, et des termes neufs comme des inventions nouvelles sont souvent nés d'un commun accord.

5.1.2.3 Le NIC

En 1967, lorsque Taylor et Larry Roberts avaient annoncé au colloque d'Ann Arbor leur projet de réseau en douze sites, Doug Engelbart se trouvait dans l'auditoire. Il dirigeait à l'époque un laboratoire de recherches informatiques au SRI. Ce qui l'intéressait, c'était l'emploi de l'ordinateur pour élargir l'intellect humain. Sous contrat avec l'ARPA, il s'occupait de mettre au point un système appelé **NLS** (pour *oNLine System*, système en ligne). Engelbart considérait le NLS comme l'instrument naturel d'un bureau central de renseignements pour le réseau de l'ARPA. Si on devait partager des ressources, il était important de faire savoir à chacun ce qui

était disponible. À la réunion du Michigan, il offrit de monter le centre d'information du réseau, le *Network Information Center*, lequel allait finir par être connu sous le nom de **NIC** ([HL–96], pp.93–94).

5.1.2.4 Les organismes de contrôle

Lorsque le DARPA fut créé en 1980, un groupe fut formé pour développer un ensemble de standards pour Internet. Ce groupe, appelé **Internet Configuration Control Board** (**ICCB**) fut réorganisé en 1983, date à laquelle il devint l'**Internet Activities Board** (**IAB**), dont la tâche devint de concevoir, de mettre en œuvre et de gérer Internet.

En 1986, l'IAB se déchargea du développement des standards sur l'**Internet Engineering Task Force** (**IETF**) et la recherche à long terme fut confiée à l'**Internet Research Task Force** (**IRTF**). L'IAB se réserve le droit de donner ou non son autorisation à tout ce que peuvent proposer les deux organisations qui dépendent de lui.

La dernière étape de cette saga fut la formation de l'**Internet Society** en 1992, date à laquelle l'IAB devint l'**Internet Architecture Board.** Ce groupe reste responsable des standards existants et à venir, et soumet ses travaux à la direction de l'Internet Society.

Pratiquement dès ses origines, Internet fut défini comme une "collaboration internationale à organisation souple de réseaux autonomes et interconnectés", qui prenait en charge des communications d'hôte à hôte "par le biais de l'adoption volontaire de protocoles et procédures ouverts" définis dans un document technique appelé *Internet Standards*, RFC 1310,2 ([RFC 1310]).

L'IETF continue à affiner les standards de communication sur Internet par le biais de divers groupes de travail, chacun spécialisé dans un aspect du protocole. Certains groupes sont spécialisés dans la gestion des réseaux, d'autres dans la sécurité, les services utilisateurs ou le routage. La plupart du temps, les groupes de l'IETF se forment, créent une recommandation et se dissolvent en moins d'un an.

5.1.3 Vue d'ensemble de l'architecture TCP/IP

5.1.3.1 Noms des PDU

Rappelons que l'architecture TCP/IP utilise quatre couches au lieu des sept du modèle OSI. Plutôt que d'utiliser la terminologie générale des n-PDU du modèle OSI pour désigner les données encapsulées au niveau de chaque couche, TCP/IP utilise les dénominations suivantes :

Application	Message	
Transport	Segment Datagramme	dans le cas connecté dans le cas non connecté
Réseau	Paquet	
Accès	Trame	

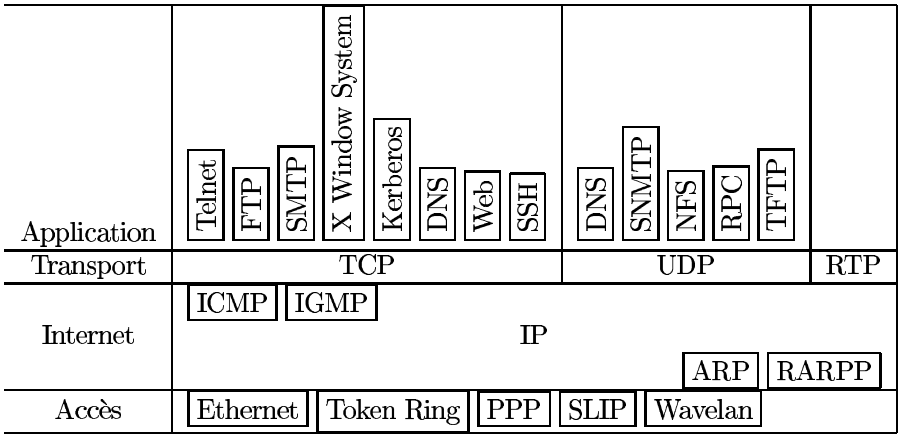
trame se disant *frame* en anglais.

5.1.3.2 Mode d'implémentation

Le traitement d'une couche de protocole peut être implémenté logiciellement ou matériellement ou en utilisant une combinaison des deux. Les couches application et transport sont presque toujours traitées logiciellement. Les couches physique et de liaison sont presque toujours traitées matériellement sur la carte réseau. La couche réseau est souvent implémentée de façon mixte.

5.2 Protocoles de la suite TCP/IP

La figure suivante montre les composants de base de l'architecture TCP/IP :



5.2.1 Protocoles de la couche d'accès

L'architecture TCP/IP proprement dite ne s'occupe ni de la couche physique, ni de la couche liaison. Les cartes réseau et leurs pilotes se trouvent sous la couche Internet. Pour TCP/IP au sens étendu, c'est-à-dire les protocoles définis par des RFC, les protocoles des couches physique et de liaison sont fortement liés : ils forment la couche d'accès. Il s'agit des protocoles **Ethernet** et **Token Ring** pour les réseaux à diffusion, **PPP** (pour *Point-to-Point Protocol*) pour les réseaux point à point et **SLIP** (pour *Serial Line Internet Protocol*) pour les liaisons série (par modem ou par RNIS, numeris en France).

Chacun de ces protocoles possède une partie qui dépend du support, et donc de la couche physique. Par exemple, Ethernet possède des protocoles pour les paires torsadées, pour les câbles coaxiaux, pour les fibres optiques, et ainsi de suite.

Linux implémente également **ATM** (pour *Asynchronous Transfer Mode*), **X.25** et **Frame Relay** pour les réseaux étendus par exemple, même si ces protocoles ne font pas partie de la suite TCP/IP.

5.2.2 Protocoles de la couche réseau

La couche réseau comprend trois protocoles :

- Le protocole principal **IP** (pour *Internet Protocol*) est chargé de déplacer sur les réseaux les paquets assemblés par les protocoles de la couche du dessus. Il utilise un ensemble d'adresses uniques, appelées **adresses IP**, pour chaque composant du réseau afin de déterminer le routage et les destinations.
- Le protocole **ICMP** (pour *Internet Control Message Protocol*) est chargé de vérifier l'état des composants sur un réseau et de générer des messages sur ces états. Il peut être utilisé pour informer d'autres composants du dysfonctionnement d'une machine donnée. Lorsque IP ne peut pas transmettre un paquet à destination, il charge ICMP d'en avertir l'expéditeur et en reste là (la couche supérieure décidera s'il faut renvoyer le paquet ou non). Bien que considéré comme un protocole de la couche réseau, ICMP repose sur IP dans la mesure où ses messages sont envoyés *via* IP. On l'utilise directement à travers le service **ping**.

- La plupart du temps, les protocoles concernent un expéditeur et un destinaire. On parle de **protocole à diffusion unique** (*unicast* en anglais). Quelquefois on veut envoyer des messages à tout un groupe de destinataires. C'est le cas du transfert audio, vidéo, des téléconférences ou des jeux en ligne. On parle alors de **protocole à diffusion restreinte** ou de **protocole de multidiffusion** (*multicast* en anglais).

  Le protocole **IGMP** (pour *Internet Group Management Protocol*) est le protocole réseau de multidiffusion de TCP/IP.

En fait IP se décline en plusieurs versions dont trois sont actuellement utilisées :

- la version 4, **IPv4**, avec ses adresses IP sur 32 bits ;
- la version 6, **IPv6** ou **IPng** (pour *IP New Generation*), avec ses adresses IP sur 128 bits ;
- la version sécurisée, **IPsec**, qui n'envoie pas les informations en clair.

Par défaut, IP signifie IPv4.

ARP (pour *Address Resolution Protocol*) est installé entre la couche d'accès et la couche Internet. Ce protocole est mis en place pour traduire les adresses réseau (les adresses IP par exemple) en adresses physiques (les deux couches, réseau et liaison, n'utilisant pas le même format d'adresses).

5.2.3 Protocoles de la couche de transport

La couche de transport comprend deux protocoles fondamentaux :

- **TCP** (pour *Transmission Control Protocol*) est un protocole de communication qui fait au mieux pour permettre un transfert de données fiable. Il se charge d'assembler les données provenant des applications en les plaçant dans des segments standard et en essayant de s'assurer qu'elles sont transférées correctement.
- **UDP** (pour *User Datagram Protocol*) est un protocole qui n'assure pas la retransmission des datagrammes si ceux-ci n'arrivent pas à destination. Le composant émetteur n'a même aucun moyen de savoir si un message a été reçu correctement ou non. UDP est donc moins fiable que TCP mais plus rapide et surtout plus facile à mettre en œuvre. Il n'a pas non plus de fonction de réparation d'erreur.

On dit que UDP est **orienté sans connexion** (*connectionless* en anglais) alors que TCP est **orienté connexion**.

Il existe un troisième protocole qui sert à envoyer les flux radio ou télé à travers Internet. Appelé **RTP** (pour *Real-Time Protocol*), il est décrit dans [RFC 1889]. En fait les datagrammes RTP sont souvent acheminés dans des segments UDP avec RTP implémenté dans l'espace utilisateur et non dans le noyau.

5.2.4 Les protocoles d'application

Les applications s'appuient sur TCP (telnet ou FTP, par exemple), sur UDP (TFTP et NFS, par exemple) ou sur les deux (DNS, par exemple) :

- Les **connexions à distance** (*remote connexion* en anglais) permettent à un utilisateur basé sur un système de se connecter – par l'intermédiaire du réseau – à un autre système l'acceptant comme utilisateur. C'est différent d'une **connexion distribuée**, comme dans le système SAGE, dans lequel le terminal est situé à une certaine distance mais les systèmes sont les mêmes. Le protocole **telnet** fut le premier à permettre la connexion à distance sous TCP/IP. En raison de la transmission sans garantie du mot de passe et des données, on préfère aujourd'hui le protocole **SSH** (*Secure Socket Shell*) à telnet.

- Les **transferts de fichiers** permettent aux utilisateurs de partager des fichiers rapidement et efficacement, sans duplication excessive ni nécessité de se préoccuper de la méthode de transport. Il est bien plus rapide de transférer un fichier par réseau que par la poste, et même que de copier le fichier sur une disquette pour la porter d'une pièce à une autre. **FTP** (pour *File Transport Protocol*) est le premier protocole utilisé sous TCP/IP. Ayant l'inconvénient de transmettre le mot de passe en clair, il est de plus en plus fréquemment remplacé par **SCP** (pour *Secure Copy Protocol*) et **SFTP** (pour *Secure File Transport Protocol*).
- Le **courrier électronique** est bon marché (pas d'enveloppe, ni papier, ni timbre) et rapide (le tour du monde en une minute ou presque). Le premier protocole fut **SMTP** (pour *Simple Mail Transfer Protocol*), totalement transparent pour l'utilisateur. En coulisses, SMTP se connecte à une machine distante et transfère les messages électroniques.
- Au fil du temps, d'autres protocoles s'y sont ajoutés, comme **DNS** (pour *Domain Name System*) qui convertit les noms (tels que `www.linux-france.org`) en adresses IP et *vice-versa*.
- **HTTP** (pour *HyperText Transfer Protocol*) est le protocole le plus fréquemment utilisé actuellement dans la couche application. Il permet l'échange de données dans le **World Wide Web**, c'est-à-dire le chargement de pages web par l'intermédiaire d'un navigateur (Netscape, Mozilla, Lynx, etc.).
- **NFS** (pour *Network File System*) permet à plusieurs ordinateurs d'accéder à un seul et même système de fichiers. Le service NFS représente une extension des systèmes de fichiers locaux au-delà des limites du réseau.

5.3 Les adresses réseau Internet

Dans le cas d'Internet, une adresse réseau est précisée par deux paramètres : l'adresse IP liée à la couche réseau et le port lié à la couche transport.

5.3.1 Adresse IP

5.3.1.1 Définition

IPv4 utilise une adresse 32 bits pour identifier une machine sur l'interréseau, appelée **adresse IP** ou **adresse Internet.**

Les adresses IP sont attribuées par **NIC.** Un réseau qui n'est pas connecté à Internet peut déterminer son propre adressage mais, pour tous les accès Internet, l'adresse doit être enregistrée auprès du NIC.

Le NIC désigne un représentant par pays, et une plage d'adresses, celui-ci étant chargé d'attribuer les adresses dans ce pays, par exemple :

```
http://www.afnic.fr
```

pour la France.

5.3.1.2 Représentation

Puisqu'il est difficile de lire un nombre de 32 bits, on représente souvent ces nombres dans les applications sous la forme de quatre nombres décimaux de 8 bits, séparés par des points, par exemple :

```
127.40.8.72
```

127 représentant dans ce cas l'octet de poids fort.

5.3.2 Multiplexage au niveau transport

5.3.2.1 Notion de port

Un ordinateur peut communiquer en même temps avec plusieurs autres (en utilisant des processus différents et du pseudo-parallélisme), par exemple recevoir une page web, du courrier tout en envoyant un fichier par `ftp`. Les paquets qui arrivent sont triés :

- ceux qui ne correspondent pas à l'adresse IP sont rejetés ;
- les autres sont envoyés vers telle ou telle (instance d') application.

Comment déterminer l'instance d'application vers laquelle on doit envoyer tel paquet ? L'en-tête de la couche de transport (que ce soit TCP ou UDP) contient un champ indiquant celle-ci. Il s'agit d'une entier de 16 bits, donc compris entre 0 et 65 535, appelé **port**.

Sous Internet, une **connexion** est caractérisée, de façon univoque, par un quadruplet : adresse IP source, numéro de port source, adresse IP de destination, numéro de port de destination. Un couple adresse IP, numéro de port s'appelle en anglais un *peer*.

5.3.2.2 Modèle client/serveur

La manipulation des sockets se fait toujours, sauf dans le cas des paires de sockets locales, dans le cadre du modèle client/serveur : la **socket client** est celle qui tente d'établir la communication ; la **socket serveur** est celle qui attend que l'on tente d'établir une communication. Il faut donc prévoir pour la socket serveur un procédé pour cette attente, ne serait-ce qu'une simple boucle.

5.3.2.3 Attribution des numéros de port aux serveurs

Si un client situé sur un ordinateur A veut envoyer une requête à un serveur situé sur un ordinateur B, comment peut-il connaître son port ? Chaque service peut-il choisir son propre numéro ? L'attribution des numéros est-elle centralisée ? Comment une application peut-elle obtenir le numéro de port d'une autre application ?

Un organisme, appelé **IANA** pour *Internet Assigned Number Authority*, attribue des ports fixes aux serveurs des applications connues. On parle des ports bien connus (*well bnown ports*), auxquels chaque application peut se référer lors de l'établissement de la connexion. Au départ, les valeurs situées entre 0 et 256 étaient réservées à l'attribution des ports normalisés. Mais, en 1994, ce quota a été étendu aux valeurs contenues entre 0 et 1 023, en raison de la demande toujours croissante de ports réservés. La liste des ports normalisés peut être consultée dans [RFC 1700] de 1994. La figure 5.1 montre quelques-uns des numéros de ports les plus utilisés par les serveurs.

5.3.2.4 Attribution dynamique pour les clients

Si certains services des numéros fixés par IANA, les clients par contre se voient attribuer un numéro de port de façon dynamique par le sous-système réseau du système d'exploitation au moment où ils ont besoin (situés évidemment situé au-delà de 1 024).

Application	Port
daytime (heure du jour)	13
ftp	21
telnet	23
smtp	25
time (heure)	37
DNS	53
finger	79
web	80
pop3	110
nntp (news)	119
snmp	161
irc (Internet Relay Chat)	194

FIG. 5.1 – *Numéros de port bien connus des serveurs*

Chapitre 6

API des sockets : l'interface BSD

Commençons par la fin, à savoir par l'interface de programmation (ou API). Il devrait presque s'agir d'un rappel puisqu'il vaut mieux commencer par programmer des applications réseau (très simples) avant d'aborder l'implémentation elle-même. Cependant nous ne supposons aucune connaissance préalable et donc ce chapitre et le suivant peuvent servir d'introduction à la programmation réseau.

Deux API réseau principaux ont été utilisées : les sockets de BSD et le **TLI** (pour *Transport Layer Interface*) du System V. On pourra consulter le chapitre 7 de la première édition de [STE–90] pour TLI, maintenant abandonné au profit des sockets. Nous allons nous intéresser dans ce chapitre et le suivant à l'API des sockets.

Notre but est d'étudier la mise en place des sockets non connectées (qui correspondent à UDP) mais, pour amener les notions petit à petit, nous commençons par les paires de socket locales, les sockets pour communication connectée (ce qui correspond à TCP) et la transmission de données urgentes.

6.1 Modèles de sockets

6.1.1 Notion de socket

Nous avons vu que les versions 4.1BSD et 4.2BSD d'UNIX incluaient une implémentation de TCP/IP. L'interface utilisateur de celle-ci est constituée de l'API des sockets. En UNIX, tout est fichier. Les sockets sont donc implémentés comme un type particulier de fichiers.

Une **socket**[1] est une généralisation des tubes de communication entre processus (*pipe* en anglais), sauf que la communication s'effectue maintenant entre ordinateurs distants (ou, plus exactement, processus distants et non plus nécessairement situés sur le même ordinateur). C'est évidemment surtout au niveau de l'implémentation qu'il y a quelque chose à faire. Le programmeur, quant à lui, voit une socket comme un fichier. Bien entendu sa création exige une image de communication, puisqu'il doit indiquer l'adresse réseau et le numéro de port d'application.

L'API est constituée comme d'habitude par une bibliothèque (et un fichier en-tête) du langage C, formée d'appels système.

Dans ce premier chapitre consacré à l'API des sockets, nous allons étudier l'essentiel pour concevoir un programme qui fonctionne. C'est ce que l'on peut appeler le niveau fondamental. Il correspond à l'interface BSD d'origine.

Pour établir une communication téléphonique, on appelle son correspondant en faisant son numéro, on attend qu'il réponde puis on peut parler. Schématiquement, on a donc deux postes téléphoniques, le poste local et le poste distant (en anglais *remote telephone*), et une ligne téléphonique. Pour une communication par sockets, on a l'analogue :

- une **socket locale**, qui correspond au poste téléphonique local ;
- une **socket distante** (*remote socket* en anglais), qui correspond au téléphone distant ;
- une ligne de communication.

Bien entendu on utilisera une adresse réseau (constituée d'une adresse IP et d'un numéro de port dans le cas de TCP/IP) au lieu d'un numéro de téléphone.

6.1.2 Types de communication

Il ya deux types de communication avec les sockets :

- Dans une **communication connectée** (*connection-oriented communication*), on commence par établir un canal entre les deux points. Il s'agit de TCP dans le cas de IP.

- Dans les autres cas on parle de **communication non connectée** (*connectionless-oriented communication* en anglais). Dans ce cas on doit donner l'adresse de destination lors de cahque envoi et l'adresse source si on veut une réponse. Il s'agit de UDP dans le cas de IP.

1. Le mot *socket* se traduit par *prise* (de courant) en français mais le mot anglais est devenu un terme consacré.

6.2 Paire de sockets locales

Pour débuter avec quelque chose de simple, nous allons créer deux sockets situées sur le même ordinateur, communiquant entre elles. Dans ce cas, nous n'aurons pas besoin d'utiliser les adresses réseau (il n'y a pas besoin de numéro de téléphone dans le cas de deux postes internes). Il s'agit en fait d'une façon sophistiquée de faire appel aux tubes de communication, mais c'est également une très bonne propédeutique à l'étude des sockets dans toute leur généralité.

6.2.1 Création

Syntaxe.- La création d'une paire de sockets locales s'effectue grâce à l'appel système :

```
#include <sys/socket.h>

int socketpair(int domain, int type, int protocol, int sv[2]);
```

Sémantique.- Le domaine indique la famille de protocoles qui sera utilisée lors de la communication (locale, IP, IPX de Novell, Appletalk ou autre). Il correspond au protocole de la couche réseau du modèle OSI mais rappelons que l'interface des sockets a été conçue avant la définition de celui-ci.

Le type de communication indique si la communication supporte les erreurs (cas d'une émission de télévision ou de radio transmise par réseau), si elle est unidirectionnelle ou bidirectionnelle et ainsi de suite. Il correspond au protocole de la couche transport du modèle OSI.

Le protocole est un paramètre supplémentaire qui peut être nécessaire dans certains cas mais la plupart du temps il prend la valeur nulle. Nous indiquerons les valeurs possibles lorsque cela sera nécessaire. Il n'a pas d'équivalent dans le modèle OSI.

L'appel système renvoie deux **numéros de socket** à travers le tableau `sv[]`, qui seront utilisés dans les appels systèmes ultérieurs pour faire référence à ces sockets. Il s'agit de numéros de fichiers.

Domaines.- Donnons quelques familles de protocoles implémentées sous Linux dans le tableau suivant :

Constante	Signification	Documentation
PF_UNIX	communication locale	unix(7)
PF_LOCAL	synonnyme de PF_UNIX	
PF_INET	protocole Internet IPv4	ip(7)
PF_INET6	protocole Internet IPv6	
PF_IPX	protocole Novell IPX	
PF_NETLINK	interface noyau/utilisateur de périphérique	netlink(7)
PF_X25	protocole ITU-T X.25 / ISO-8208	x25(7)
PF_AX25	protocole radio-amateur AX.25	
PF_ATMPVC	accès à ATM de façon brute	
PF_APPLETALK	protocole du réseau Apple d'origine	ddp(7)
PF_PACKET	interface bas niveau aux paquets (abandonné)	packet(7)

Rappelons que `unix(7)`, par exemple, indique que l'on obtiendra une documentation du manuel grâce à la commande :

```
man 7 unix
```

Ces constantes symboliques commencent toutes par `PF` pour l'anglais *Protocol Family*. On les retrouve aussi avec l'en-tête `AF` pour *Address Family*, par exemple `AF_UNIX`.

Nous ne nous intéresserons dans ce livre qu'aux communications locales (il s'agit en fait des tubes nommés) et aux protocoles Internet version 4.

Pour la création d'une paire de sockets, on ne peut utiliser que la famille de protocoles `PF_UNIX` (et ses trois variantes `AF_UNIX`, `PF_LOCAL` et `AF_LOCAL`).

Types de communication.- Les types acceptés sont représentés par l'une des constantes symboliques suivantes :

- `SOCK_STREAM` : flux d'octets, avec connexion, bidirectionnel et sûr (ou presque). Il s'agit de la couche de transport TCP dans le cas de IP.
- `SOCK_DGRAM` : support de datagrammes (sans connexion, sans être sûr que le message soit reçu, données d'une longueur maximum fixée). Il s'agit de la couche de transport UDP dans le cas de IP.
- `SOCK_SEQPACKET` : comme pour `SOCK_STREAM` sauf qu'ici le destinataire doit lire un paquet en entier par appel système de lecture.
- `SOCK_RAW` : accès à bas niveau.
- `SOCK_RDM` : comme pour `SOCK_DGRAM` mais sans que l'ordre d'arrivée des paquets ne soit garantie.
- `SOCK_PACKET` : abandonné. Ressemblait à `SOCK_RAW`.

Nous nous intéresserons évidemment plus particulièrement à `SOCK_STREAM` et à `SOCK_DGRAM`.

Erreurs.- La valeur renvoyée par la fonction est le numéro de fichier lorsque tout s'est bien déroulé. La valeur -1 indique une erreur ; la valeur prise par `errno` dans ce cas est l'une des suivantes :

- `EINVAL` : le domaine n'est pas valide ;
- `EPROTONOSUPPORT` : le type n'est pas cohérent avec le protocole ou le domaine ;
- `EACCES` : on n'a pas l'autorisation de créer une socket du type demandé ;
- `EMFILE`, `ENFILE`, `ENOMEM` de signification habituelle.

Exemple.- L'exemple suivant nous permet de créer une paire de sockets locales, sans utilisation particulière pour l'instant :

```
/* paire.c */

#include <stdio.h>
#include <sys/socket.h>

void main(void)
{
  int s[2];

  /* Creation d'une paire de sockets locales */

  socketpair(AF_LOCAL, SOCK_STREAM, 0, s);

  printf("s[0] = %d\n", s[0]);
  printf("s[1] = %d\n", s[1]);
}
```

L'exécution de ce programme donne :

```
# ./a.out
s[0] = 3
s[1] = 4
```

ce qui est prévisible puisque les numéros de fichier 0, 1 et 2 sont réservés pour les fichiers standard.

6.2.2 Lecture et écriture sur des sockets

Syntaxe.- On peut lire et écrire sur une socket comme sur n'importe quel fichier grâce aux appels systèmes suivants :

```
#include <unistd.h>

ssize_t read(int fd, void *buf, size_t count);
ssize_t write(int fd, const void *buf, size_t count);
```

La valeur de retour de `read()` est le nombre de caractères effectivement reçus.

Il n'y a pas d'accès direct avec les sockets, mais uniquement des accès séquentiels, aussi ne peut-on pas utiliser la fonction `lseek()`.

Exemple.- Dans l'exemple suivant on envoie un message depuis l'une des sockets, celui-ci étant récupéré par l'autre :

```
/* message.c */

#include <stdio.h>
#include <unistd.h>
#include <sys/socket.h>
#include <string.h>

void main(void)
{
  int s[2];
  char buf1[80], buf2[80];
  int n;

  /* Creation d'une paire de sockets locales */

  socketpair(AF_LOCAL, SOCK_STREAM, 0, s);

  printf("s[0] = %d\n", s[0]);
  printf("s[1] = %d\n", s[1]);

  /* Envoi d'un message */

  strcpy(buf1, "Bonjour");
  printf("Message envoye : %s\n", buf1);
  write(s[0], buf1, 7);

  /* Reception d'un message */

  n = read(s[1], buf2, 80);
  buf2[n] = '\0';              /* Terminaison de la chaine recue */
  printf("Message recu : %s\n", buf2);
}
```

6.2.3 Fermeture des sockets

Le programme de l'exemple précédent présente un défaut. Comme tout fichier, une socket doit être fermée lorsqu'on n'en a plus besoin. Ceci permet de libérer le descripteur de fichier associé et autres choses analogues.

Syntaxe.- On peut pour cela utiliser l'appel système classique :

```
#include <unistd.h>

int close(int fd);
```

bien que nous verrons un peu plus loin que ce ne soit pas le plus intéressant.

Exemple.- Un meilleur programme est donc :

```
/* ferme.c */

#include <stdio.h>
#include <unistd.h>
#include <sys/socket.h>
#include <string.h>

void main(void)
{
  int s[2];
  char buf1[80], buf2[80];
  int n;

  /* Creation d'une paire de sockets locales */

  socketpair(AF_LOCAL, SOCK_STREAM, 0, s);

  printf("s[0] = %d\n", s[0]);
  printf("s[1] = %d\n", s[1]);

  /* Envoi d'un message */

  strcpy(buf1, "Bonjour");
  printf("Message envoye : %s\n", buf1);
  write(s[0], buf1, 7);

  /* Reception d'un message */

  n = read(s[1], buf2, 80);
  buf2[n] = '\0';
  printf("Message recu : %s\n", buf2);

  /* Fermeture des sockets */

  close(s[0]);
  close(s[1]);
}
```

Sémantique.- Lorsqu'on ferme une socket, la socket qui se trouve à l'autre bout reçoit un indicateur de fin de fichier pour indiquer qu'il n'y a plus de données à recevoir.

6.3 Socket pour communication connectée

6.3.1 Cycles de vie dans le cas des communications connectées

Dans le cas d'une communication connectée, les étapes du cycle de vie d'un serveur sont les suivantes :

- création de la socket serveur ;
- initialisation du serveur (concrétisée par l'association d'une adresse IP et d'un numéro de port dans le cas de TCP) ;
- attente de client ;
- acceptation de la connexion ;
- communication (écriture, lecture) ;

– fermeture.

Les étapes du cycle de vie d'un client sont les suivantes :

– création de la socket côté client ;
– connexion à la socket distante (serveur) ;
– communication (écriture, lecture) ;
– fermeture.

6.3.2 Création d'une socket

Syntaxe.- La création d'une socket (qu'elle soit client ou serveur) s'effectue grâce à l'appel système :

```
#include <sys/socket.h>

int socket(int domain, int type, int protocol);
```

dont les paramètres ont été décrits à propos de la création d'une paire de sockets. Le numéro de socket est renvoyé, ou -1 dans le cas d'une erreur.

En cas d'erreur, la variable `errno` prend l'une des valeurs suivantes :

– `EAFNOSUPPORT` : type de famille d'adresses non implémenté ;
– `EINVAL` : type de communication non valide, lorsque `type` n'est pas compris entre 0 et une certaine valeur maximum ;
– `ENFILE` : il n'y a plus de descripteur de socket disponible.

Protocoles.- Dans le cas de la famille de protocoles `PF_INET`, correspondant à IPv4 :

– pour le type de communication `SOCK_STREAM`, le seul protocole existant est `IPPROTO_TCP`, égal à 0 ; il correspond à TCP ;
– pour le type de communication `SOCK_DGRAM`, le seul protocole existant est `IPPROTO_UDP`, égal à 0 ; il correspond à UDP.

Exemple.- Créons une socket pour le protocole TCP :

```
/* client1.c */

#include <stdio.h>
#include <sys/socket.h>

void main(void)
{
  int s;

  /* Creation d'une socket */

  s = socket(AF_INET, SOCK_STREAM, 0);

  printf("s = %d\n", s);

  /* Fermeture de la socket */

  close(s);
}
```

Ce programme a peu d'intérêt. Nous allons le compléter petit à petit.

6.3.3 Spécification des adresses

Voyons comment on spécifie l'adresse réseau.

6.3.3.1 Adresses génériques

Puisque les sockets BSD furent développées avant que le standard C ANSI ne soit adopté, le type pointeur `(void *)` n'était pas accepté pour faire face aux différents types d'adresses. La solution fut donc de définir une structure d'**adresse générique** :

```
#include <sys/socket.h>

struct sockaddr
  {
    unsigned short int sa_family;
    unsigned char  sa_data[14];
  };
```

en réservant deux octets pour la famille d'adresse et 14 octets au maximum pour l'adresse proprement dite.

6.3.3.2 Adresses locales

Pour les communications locales, on utilise la structure d'**adresse Unix** :

```
#include <sys/un.h>

struct sockaddr_un
{
    sa_family_t sun_family;        /* Famille d'adresses */
    char        sun_path[108];     /* Chemin d'acces     */
};
```

où `sun_path` indique le chemin du fichier local (chaîne de caractères, qui n'a pas besoin de se terminer par le caractère nul puisque, comme nous le verrons, la longueur de l'adresse est également toujours passée en paramètre) et où `sun_family` doit avoir la valeur `AF_LOCAL` ou `AF_UNIX`.

Le type `sa_family_t` est un entier court non signé, ce qui représente deux octets.

Des informations sur les adresses locales se trouvent dans la page de manuel `unix(4)`.

Le service `lpr` d'impression, par exemple, utilise une socket locale pour communiquer avec le fichier de spool.

6.3.3.3 Adresses Internet

Pour les sockets de la famille `AF_INET` reposant sur le protocole IPv4, on utilise une structure précisant le port d'application et l'adresse IP :

```
#include <netinet/in.h>

struct sockaddr_in
{
    sa_family_t    sin_family;   /* Famille d'adresses    */
    uint16_t       sin_port;     /* Port d'application    */
    struct in_addr sin_addr;     /* Adresse Internet      */
    unsigned char  sin_zero[8];  /* octets de remplissage */
};

struct in_addr
{
```

```
    uint32_t        s_addr;        /* Adresse Internet      */
}
```

où `sin` est une abréviation de *Socket INternet* et `netinet` une abréviation de *NET InterNET*. Le membre `sin_family` doit prendre la valeur `AF_INET`.

La description des types d'adresses se trouvent dans la page de manuel `ip(4)`.

6.3.3.4 Autres familles d'adresses

Les autres familles de protocoles utilisent les structures suivantes, que nous n'étudierons pas dans ce livre :

Famille	Structure	Fichier
AF_AX25	sockaddr_ax25	<netax24/ax25.h>
AF_INET6	sockaddr_in6	<netinet/in.h>
AF_IPX	sockaddr_ipx	<netipx/ipx.h>

6.3.3.5 Constantes symboliques d'adresses Internet

Certaines adresses Internet, souvent utilisées, sont disponibles sous forme de constantes symboliques :

- l'**adresse Internet par défaut** (*wild address* en anglais) permet de déterminer l'adresse de votre ordinateur, ou plus exactement l'une de celles-ci lorsqu'il dispose de plusieurs cartes réseau par exemple. Il s'agit de la constante `INADDR_ANY`.
- L'adresse Internet locale correspond au périphérique *loopback* d'adresse IP 127.0.0.1. Elle est repérée par la constante symbolique `INADDR_LOOPBACK`.

6.3.4 L'ordre réseau des octets

Rappelons que la valeur décimale `4660`, soit `1234h`, s'écrit `12h, 34h` dans l'ordre grand boutien des octets et `34h, 12h` dans l'ordre petit boutien. Il a été décidé qu'on utiliserait l'ordre grand boutien des octets sur les réseaux.

Des fonctions sont fournies pour faciliter la conversion.
Il y a deux directions de conversion à considérer :

- de l'hôte vers le réseau ;
- du réseau vers l'hôte.

Il y a deux catégories de fonctions à considérer :

- la conversion d'entiers courts (de seize bits) ;
- la conversion d'entiers longs (de trente-deux bits).

On utilise donc les fonctions suivantes dont les noms sont suffisamment parlants :

```
#include <netinet/in.h>

unsigned long htonl(unsigned long hostlong);
unsigned short htons(unsigned short hostshort);
unsigned long ntohl(unsigned long netlong);
unsigned short ntohs(unsigned netshort);
```

Ces fonctions sont décrites dans la page de manuel `byteorder(3)`.

6.3.5 Connexion au serveur

Créer une socket revient à réserver un espace mémoire pour entreposer temporairement les données en transit et initialiser un certain nombre de paramètres. Par contre aucune adresse réseau ne lui est encore associée. Une fois qu'une socket client est créée, on doit la connecter au serveur en lui indiquant l'adresse de la socket serveur. Cette adresse est un chemin dans le cas d'une communication locale et une adresse de réseau dans le cas d'une communication entre machines distantes.

6.3.5.1 Appel système

Syntaxe.- Pour établir une connexion vers une socket distante, on utilise l'appel système suivant :

```
#include <sys/types.h>
#include <sys/socket.h>

int connect(int socket, struct sockaddr *adresse_destination, int longueur_adresse);
```

Sémantique.- Le premier argument est le numéro de fichier socket obtenu lors de la création de celle-ci. Le second argument est un pointeur à une structure décrivant l'adresse de la socket distante ; les deux premiers octets de celle-ci déterminent la famille d'adresse et la façon dont les octets suivants sont interprétés. Le troisième argument indique la longueur de cette structure, en octets. Ce dernier argument est indispensable, car avant d'analyser le type de la socket, le noyau doit copier l'adresse depuis l'espace de l'utilisateur vers son propre espace mémoire, et doit donc connaître la longueur de la structure.

Lorsque la connexion s'établit, la valeur de retour est nulle. Sinon elle prend la valeur -1 et la nature du problème est indiquée grâce à `errno`.

6.3.5.2 Exemple de serveur : le service date et heure d'Unix

Comme exemple nous allons nous connecter au serveur, toujours présent sous Unix, donnant la date et l'heure. Commençons par étudier celui-ci. Les systèmes `Unix` possèdent un certain nombre de services sous forme de serveur. C'est le cas de `daytime` qui indique la date et l'heure.

Premier essai.- Ce service correspond au port 13 de l'ordinateur. On peut l'essayer avec `telnet` :

```
# telnet 127.0.0.1 13
Trying 127.0.0.1...
telnet: connect to address 127.0.0.1: Connection refused
```

Mise en service.- Dans notre cas le service n'est pas mis en place. Il suffit de décommenter la ligne commençant par 'daytime' du fichier `/etc/inetd.conf` :

```
# daytime stream tcp nowait root internal
```

puis de remettre à jour le démon réseau :

```
# /etc/init.d/inetd reload
Reload service inetd                          done
```

en ayant les droits de super-utilisateur.

Deuxième essai.- Maintenant nous devrions avoir une réponse :

```
# telnet 127.0.0.1 13
Trying 127.0.0.1...
Connected to 127.0.0.1.
Escape character is '^]'.
Sat Feb  7 10:25:04 2004
Connection closed by foreign host.
```

6.3.5.3 Exemple

Le programme suivant permet d'obtenir la date et l'heure :

```
/* client2.c */

#include <stdio.h>
#include <sys/socket.h>
#include <netinet/in.h>

void main(void)
{
  int s;
  struct sockaddr_in serveur;
  int longueur;
  const unsigned char IPno[] = {127,0,0,1}; /* local */
  char buf[80];
  int n;

  /* Creation d'une socket */

  s = socket(AF_INET, SOCK_STREAM, 0);

  /* Connexion */

  memset(&serveur, 0, sizeof(serveur));

  serveur.sin_family = AF_INET;
  serveur.sin_port = htons(13);
  memcpy(&serveur.sin_addr.s_addr, IPno, 4);

  longueur = sizeof(serveur);

  n = connect(s, (struct sockaddr *) &serveur, longueur);

  /* Reception */

  printf("Message recu : ");
  n = -1;
  while (n != 0)
    {
    n = read(s, buf, 80);
    buf[n] = '\0';
    printf("%s\n", buf);
    }

  /* Fermeture de la socket */

  close(s);
}
```

On remarquera la façon de définir ici l'adresse IP par un tableau d'octets, ceux-ci étant ordonnés suivant l'ordre réseau.

6.3.6 Initialisation d'un serveur

Notion.- Lorsqu'une socket serveur est créée, on se retrouve un peu dans la position d'un téléphone auquel on n'a pas encore attribué de numéro. Elle est peu utile alors car personne ne peut l'appeler. On doit donc lui attribuer une adresse réseau (un numéro IP et un numéro de port dans le cas d'Internet).

Syntaxe.- Pour initialiser un serveur, on utilise l'appel système :

```
bind(socket, adresse, longueur_adresse);
```

ou, plus exactement :

```
#include <sys/socket.h>
#include <sys/types.h>

int bind(int socket, struct sockaddr * addresse, socklen_t longueur);
```

Sémantique.- Comme pour `connect()`, le premier argument est le numéro de fichier de la socket serveur obtenu lors de la création de celle-ci, le second argument est un pointeur à une structure décrivant l'adresse et le troisième indique la longueur de cette structure, en octets.

Si tout s'est bien passé, la fonction renvoie la valeur 0. Sinon elle renvoie la valeur -1 et la valeur de `errno` donne des indications sur ce qui n'a pas marché.

6.3.7 Attente de client

Notion.- Une socket serveur doit créer une file d'attente de connexions puis attendre qu'un client essaie de se connecter.

Syntaxe.- On utilise pour cela l'appel système suivant :

```
#include <sys/socket.h>

int listen(int s, int backlog);
```

Cet appel ne s'applique qu'aux sockets des types `SOCK_STREAM` et `SOCK_SEQPACKET`.

Sémantique.- Le premier argument est le numéro de fichier de socket serveur à utiliser, le deuxième spécifie la longueur de la file d'attente en nombre d'entrées.

La fonction renvoie 0 si tout s'est bien passé, -1 sinon. Dans ce dernier cas, la variable `errno` prend l'une des valeurs suivantes :

- `EBADF` : l'argument `s` n'est pas un descripteur valide ;
- `ENOTSOCK` : l'argument `s` n'est pas une socket ;
- `EOPNOTSUPP` (pour *Error OPtion NOT SUPPorted*) : la socket n'est pas d'un type qui supporte l'opération d'attente.

6.3.8 Acceptation de client

Syntaxe.- Pour accepter un client et renvoyer le numéro de fichier socket qui lui est attribué, on utilise l'appel système suivant :

```
#include <sys/types.h>
#include <sys/socket.h>

int accept(int s, struct sockaddr *addr, int *longueur);
```

Sémantique.- Le premier argument `s` désigne le numéro de fichier de la socket serveur.

Lorsqu'une socket distante (d'un des types `SOCK_STREAM`, `SOCK_SEQPACKET` ou `SOCK_RDM`) demande à être connectée et que le serveur l'accepte (entre autre parce qu'il reste de place dans sa file d'attente), une nouvelle socket est créée du côté du serveur, dont le numéro de fichier est renvoyé par la fonction. De plus l'adresse de la socket distante et sa longueur sont renvoyées par

les deux derniers arguments de la fonction. La nouvelle socket sort alors de la file d'attente (pour vivre une vie indépendante).

Lorsque la demande de connexion n'est pas acceptée, la valeur -1 est renvoyée.

L'argument `longueur` sert à renvoyer la longueur de l'adresse de la socket distante, qui dépend de la famille d'adresses. Mais elle sert également comme entrée. On a intérêt à l'initialiser avec la longueur maximum parmi celles qui sont possibles.

6.3.9 Ouverture d'un fichier de socket

Vous connaissez la fonction :

```
#include <stdio.h>

FILE * fopen(const char *path, const char *mode);
```

permettant d'ouvrir un fichier, et surtout de l'associer à un nom physique. Les sockets ne possèdent pas de nom physique. On utilisera donc à la place la fonction :

```
#include <stdio.h>

FILE * fopen(int fd, const char *mode);
```

qui joue le même rôle mais dont le premier paramètre est un numéro de fichier.

6.3.10 Exemple de serveur

Créons un serveur TCP sur notre ordinateur associé au port 9999 qui se contente de renvoyer 'Bonjour' au format HTML lors de toute demande de connection :

```
/* serveur.c */

#include <stdio.h>
#include <sys/socket.h>
#include <unistd.h>
#include <netinet/in.h>
#include <string.h>
#include <stdlib.h>

void main(void)
{
  int s;                      /* descripteur de socket serveur */
  int c;                      /* descripteur de socket accepte */
  struct sockaddr_in serveur;
  struct sockaddr_in client;
  int longueur;
  int n;
  FILE * rx;                  /* Flux de lecture */
  FILE * tx;                  /* Flux d'ecriture */
  char entree[2048];


  /* Creation d'une socket (serveur) */

  s = socket(AF_INET, SOCK_STREAM, 0);

  /* Association d'une adresse sur le port 9999 */

  memset(&serveur, 0, sizeof(serveur));

  serveur.sin_family = AF_INET;
```

```
  serveur.sin_port = ntohs(9999);
  serveur.sin_addr.s_addr = ntohl(INADDR_ANY);

  longueur = sizeof(serveur);

  n = bind(s, (struct sockaddr *) &serveur, longueur);

  /* Mise du serveur a l'ecoute */

  listen(s, 5);

  /* Accepte une ligne de texte
   * l'ignore
   * puis envoie une page Web */

  for (;;)
    {
      /* Attend une connexion */

      longueur = sizeof(client);
      c = accept(s, (struct sockaddr *) &client, &longueur);

      /* Creation des flux */

      rx = fdopen(c, "r");
      tx = fdopen(dup(c), "w");

      /* Accepte une ligne de texte */

      fgets(entree, sizeof(entree), rx);

      /* Envoie une page HTML */

      fputs("<HTML>\n<BODY>\n<H1>Bonjour</H1>\n</BODY>\n</HTML>\n", tx);
      fclose(tx);
      fclose(rx);
    }
}
```

Pour le tester, il suffit de remplir la partie URL de votre navigateur préféré avec :

```
http://127.0.0.1:9999/
```

6.4 Données urgentes

6.4.1 Notion

Imaginons une file d'attente chez le boulanger. Chacun attend son tour pour être servi. Un gros client arrive, le restaurateur du coin, et va directement auprès de la boulangère pour être servi en priorité. Ce comportement (anti-social) a permis de court-circuiter la file d'attente. On parle de *out-of-band* en anglais. Par opposition, on parle de *in-band* lorsqu'on se place correctement dans la file d'attente.

Les **données urgentes** sur une connexion ont un comportement analogue. Normalement, le flux de données est récupéré par le destinataire dans l'ordre d'envoi de l'expéditeur. L'API des sockets, cependant, permet que des données soient envoyées en urgence au destinataire, de la même façon que ce que nous venons de voir ci-dessus.

Ceci se justifie. Un destinataire peut avoir une grande quantité de données à recevoir en attente. L'expéditeur peut décider d'annuler cette réception. Il veut donc envoyer un ordre d'an-

nulation de façon urgente, c'est-à-dire qui va court-circuiter la file d'attente en réception. Les applications qui utilisent les données urgentes sont principalement `telnet`, `rlogin` et `ftp`.

6.4.2 Envoi et réception des données urgentes

Les fonctions `write()` et `read()` sont des fonction *in-band*. On se sert des fonctions `send()` et `recv()` pour les données urgentes.

6.4.2.1 La fonction `send()`

Syntaxe.- La fonction `send()`:

```
#include <sys/types.h>
#include <sys/socket.h>

int send(int s, const void *msg, int len, unsigned int flags);
```

possède quatre arguments:

- le numéro de socket sur laquelle il faut écrire;
- l'adresse du tampon d'écriture;
- la longueur du tampon d'écriture;
- les drapeaux d'option.

Valeurs des drapeaux.- Pour une valeur 0 du vecteur des drapeaux, on se retrouve avec la fonction `write()`. On peut également y placer une conjonction des constantes symboliques suivantes:

- `MSG_OOB` (pour *Out-Of-Band*): envoyer des données urgentes;
- `MSG_DONTROUTE`: ne pas utiliser le routeur pour envoyer les données (mais la route que l'on précise); intéressant pour des diagnostics;
- `MSG_DONTWAIT`: opération non bloquante;
- `MSG_NOSIGNAL`: ne pas envoyer de signal `SIGPIPE` lorsque la machine distante a fermé la connexion, mais le code d'erreur `EPIPE` est renvoyé;
- `MSG_CONFIRM`: on veut une confirmation de la part de la machine distante (depuis Linux 2.3).

On trouve une description de ces valeurs dans la page `man` de `sendmsg`. C'est surtout `MSG_OOB` qui nous intéresse ici.

6.4.2.2 La fonction `recv()`

Syntaxe.- Le prototype de la fonction `recv()` est analogue à celui de la fonction `send()`:

```
#include <sys/types.h>
#include <sys/socket.h>

int recv(int s, const void *msg, int len, unsigned int flags);
```

Valeurs du vecteur des drapeaux.- Pour une valeur 0 du vecteur des drapeaux, on se retrouve avec la fonction `read()`. On peut également y placer une conjonction des constantes symboliques suivantes:

- `MSG_OOB` (pour *Out-Of-Band*): recevoir des données urgentes;
- `MSG_PEEK`: récupérer les données sans les enlever de la file d'attente;

- `MSG_WAITALL` : l'opération est bloquante jusqu'à ce que tout soit reçu ;
- `MSG_NOSIGNAL` : ne pas envoyer de signal `SIGPIPE` lorsque la machine distante a fermé la connexion, mais le code d'erreur `EPIPE` est renvoyé ;
- `MSG_TRUNC` : renvoyer la taille réelle du paquet ;
- `MSG_ERRQUEUE` : recevoir les erreurs dans la file d'attente.

On trouve une description de ces valeurs dans la page `man` de `recv`. C'est surtout `MSG_OOB` qui nous intéresse ici.

6.5 Socket pour communication non connectée

6.5.1 Cycles de vie dans le cas des communications non connectées

Dans le cas des communications non connectées, les étapes du cycle de vie d'un serveur sont les suivantes :

- création de la socket serveur ;
- initialisation du serveur (concrétisé par l'association d'une adresse Internet et d'un numéro de port dans le cas de UDP) ;
- en général une boucle infinie du type :
  - attente d'une réception ;
  - envoi de la réponse ;
- fermeture éventuelle.

Dans le cas d'une communication non connecté, l'attente de client est émulée librement au gré du programmeur. Une façon simple consiste en une **scrutation** (*polling* en anglais) : une boucle infinie attend l'arrivée de quelque chose en utilisant un ordre de lecture ; le serveur répond par un ordre d'écriture.
Les étapes du cycle de vie d'un client sont les suivantes :

- création de la socket client ;
- envoi d'un message au serveur ;
- attente d'une réponse ;
- éventuellement d'autres cycles envoi–réception ;
- fermeture.

6.5.2 Les fonctions d'envoi et de réception de message

Les fonctions `read()` et `write()`, utilisées pour l'envoi et la réception de messages dans le cas d'une paire de sockets locales, supposent l'utilisation d'un protocole orienté connexion (tel que TCP). Dans le cas d'un protocole sans connexion (tel que UDP), on doit spécifier l'adresse de destination à chaque fois. On doit donc utiliser un couple différent de fonctions de lecture-écriture : il s'agit de `sendto()` et de `recvfrom()`.

6.5.2.1 Fonction d'envoi pour communication non connectée

La fonction `sendto()`:

```
#include <sys/types.h>
#include <sys/socket.h>

int sendto(int s, const void *msg, int len, unsigned int flags,
           const struct sockaddr *to,
           int tolen);
```

possède six arguments:

- les quatre premiers sont ceux de la fonction `send()`;
- un pointeur à une adresse générique de socket, indiquant où doit être envoyé le datagramme;
- la longueur de cette adresse.

La valeur renvoyée par la fonction est le nombre d'octets envoyés (mais rien n'assure qu'ils soient reçus) ou -1 lorsqu'une erreur est survenue.

6.5.2.2 Fonction de réception pour communication non connectée

La fonction `recvfrom()`:

```
#include <sys/types.h>
#include <sys/socket.h>

int recvfrom(int s, const void *msg, int len, unsigned int flags,
             const struct sockaddr *from,
             int *fromlen);
```

possède également six arguments:

- les quatre premiers sont ceux de la fonction `recv()`;
- un pointeur à une adresse générique de socket, indiquant où doit être déposé l'adresse de l'expéditeur du datagramme;
- la longueur de cette adresse, ou plus exactement un pointeur puisqu'on utilise un passage par adresse.

La valeur renvoyée par la fonction est le nombre d'octets reçus ou -1 lorsqu'une erreur est survenue.

Remarquons que l'on doit initialiser `*fromlen` avec la taille maximum des adresses de socket car cette valeur est utilisée pour réserver de la place pour l'adresse.

6.5.3 Exemple

Écrivons un serveur UDP qui attend des clients et qui renvoie 'Essai' à chacun d'eux sans tenir compte de ce qui est reçu.

6.5.3.1 Programmation du serveur

Le client peut être créé grâce au programme suivant:

```
/* udpServer.c */

#include <stdio.h>
#include <sys/socket.h>
```

```
#include <netinet/in.h>
#include <string.h>
#include <time.h>

void main(void)
{
  int s;
  struct sockaddr_in serveur;
  struct sockaddr_in client;
  int longueur;
  const unsigned char IPno[] = {127,0,0,1}; /* local */
  char buf[80];

  /* Creation d'une socket */

  s = socket(AF_INET, SOCK_DGRAM, 0);

  /* Creation d'une adresse Internet */

  memset(&serveur, 0, sizeof(serveur));

  serveur.sin_family = AF_INET;
  serveur.sin_port = htons(8888);
  memcpy(&serveur.sin_addr.s_addr, IPno, 4);

  longueur = sizeof(serveur);

  /* Initialisation du serveur */

  bind(s, (struct sockaddr *) &serveur, longueur);

  /* Attente de client */

  for (;;)
    {
      longueur = sizeof(client);
      recvfrom(s, buf, sizeof(buf), 0, (struct sockaddr *) &client, &longueur);

      /* Envoi */

      strcpy(buf,"Essai\n");
      sendto(s, buf, strlen(buf), 0, (struct sockaddr *) &client, longueur);
    }

  /* Fermeture de la socket */

  close(s);
}
```

6.5.3.2 Programmation du client

Le client peut être programmé de la façon suivante :

```
/* udpClient.c */

#include <stdio.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <string.h>

void main(void)
{
```

```
  int s;
  struct sockaddr_in serveur;
  struct sockaddr_in adr;
  int longueur;
  const unsigned char IPno[] = {127,0,0,1}; /* local */
  char buf[80];
  int n;
  int x;

  /* Creation d'une socket */

  s = socket(AF_INET, SOCK_DGRAM, 0);

  /* Creation d'une adresse */

  memset(&serveur, 0, sizeof(serveur));

  serveur.sin_family = AF_INET;
  serveur.sin_port = htons(8888);
  memcpy(&serveur.sin_addr.s_addr, IPno, 4);

  longueur = sizeof(serveur);

  /* Envoi d'un message */

  strcpy(buf, " ");
  n = sendto(s, buf, strlen(buf), 0, (struct sockaddr *) &serveur, longueur);

  /* Reception */

  printf("Message recu : ");
  n = recvfrom(s, buf, 80, 0, (struct sockaddr *) &adr, &x);
  buf[n] = '\0';
  printf("%s\n", buf);

  /* Fermeture de la socket */

  close(s);
}
```

6.5.3.3 Mise en place

Dans une première fenêtre de terminal, compiler le serveur et le faire exécuter :

```
linux # gcc udpServer.c -o udpS
udpServer.c: In function 'main':
udpServer.c:10: warning: return type of 'main' is not 'int'
linux # ./udpS
```

Dans une seconde fenêtre, compiler le client et le faire exécuter :

```
linux # gcc udpClient.c -o udpC
udpClient.c: In function 'main':
udpClient.c:9: warning: return type of 'main' is not 'int'
linux: # ./udpC
Message recu : Essai

linux: #
```

6.6 Semi-arrêt d'une socket

Introduction.- Nous avons vu comment fermer une socket, comme on ferme tout fichier. Cette façon de faire pose problème lorsqu'un processus local veut signaler à une socket distante qu'il n'a plus de données à envoyer. Si ce processus veut recevoir une confirmation de la socket distante, il ne le peut pas puisque la socket est fermée. Il faut donc trouver un moyen qui ferme la socket à moitié.

Syntaxe.- On utilise pour cela la fonction :

```
#include <sys/socket.h>

int shutdown(int s, int how);
```

Le premier argument est le numéro de la socket à arrêter. Le second argument indique comment cette socket doit être arrêtée, la valeur devant être l'une des trois suivantes :

Valeur	Constante symbolique	Signification
0	SHUT_RD	On ne pourra plus lire sur la socket
1	SHUT_WR	On ne pourra plus écrire sur la socket
2	SHUT_RDWR	On ne pourra plus ni lire ni écrire

La valeur 2 correspond à `close()`.

L'appel à `shutdown` ne libère pas le numéro de fichier, même avec la valeur `SHUT_RDWR`. Il faut un appel à `close()` pour le libérer.

Valeur de retour.- La valeur de retour est 0 si l'appel de la fonction réussit. Elle est -1 en cas d'échec, la variable `errno` prenant alors l'une des valeurs suivantes :

- `EBADF` si le numéro de fichier n'est pas un numéro valide ;
- `ENOTSOCK` si le numéro de fichier ne correspond à une socket ;
- `ENOTCONN` si la socket spécifiée n'est pas connectée.

Chapitre 7

API des sockets : niveau avancé

Dans le premier chapitre consacré à l'API des sockets, nous avons étudié l'essentiel pour concevoir un programme qui fonctionne. Dans ce chapitre nous passons en revue d'autres appels système de cette API. Nous n'en avons pas vraiment besoin dans ce volume mais nous devrons y faire référence dans l'implémentation Linux, y compris pour le seul cas qui nous intéresse, c'est-à-dire l'envoi de datagrammes UDP.

7.1 Données dispersées

7.1.1 Les entrées-sorties sur socket

Nous avons vu que l'on peut utiliser les fonctions `write()` et `read()` sur les sockets dans le cas de communications connectées et les fonctions `sendto()` et `recvfrom()` dans le cas de communications non connectées. En fait il existe cinq types de fonctions, indiquées ci-dessous dans l'ordre de complexité de mise en œuvre pour l'utilisateur:

Écriture	Lecture	Fonctionnalités ajoutées
write()	read()	Fonction de base
send()	recv()	Ajoute un argument de drapeaux
sendto()	recvfrom()	Ajoute une adresse de socket et une longueur
writev()	readv()	Ni socket ni drapeaux, mais un vecteur
sendmsg()	recvmsg()	Drapeaux, socket, vecteur et données ancillaires.

Nous allons étudier le cas des données dispersées dans cette section. Les ordres de lecture-écriture de données dispersées (*scatter* en anglais) permettent de manipuler des données éparpillées en plusieurs endroits. Nous avons besoin de la notion de vecteur d'entrée-sortie pour travailler avec ces fonctions.

7.1.2 Vecteur d'entrée-sortie

Un **vecteur d'entrée-sortie** est un tableau d'entités du type `struct iovec`:

```
#include <sys/uio.h>

struct iovec
  {
  ptr_t   iov_base;     /* adresse de debut */
  size_t  iov_len;      /* taille en octets */
  };
```

chaque instance spécifiant un tampon de données, grâce à son adresse de base et sa taille (en octets).

7.1.3 Syntaxe des fonctions

Les fonctions d'entrée-sortie de données dispersées:

```
#include >sys/uio.h>

int readv(int fd, const struct iovec *vector, int count);
int writev(int fd, const struct iovec *vector, int count);
```

possèdent trois arguments:

- le numéro de fichier sur lequel lire ou écrire;
- le vecteur d'entrée-sortie;
- le nombre d'éléments de ce vecteur.

La valeur de retour est le nombre d'octets lus ou écrits ou -1 en cas d'erreur.

7.1.4 Exemple

Le programme suivant permet d'écrire des données qui sont dispersées en trois endroits différents :

```
/* scatter.c */

#include <sys/uio.h>
#include <string.h>

void main(void)
{
  char part1[] = "Essai d'";
  char part2[] = "ecriture ";
  char part3[] = "dispersee\n";
  struct iovec iov[3];

  iov[0].iov_base = part1;
  iov[0].iov_len = strlen(part1);

  iov[1].iov_base = part2;
  iov[1].iov_len = strlen(part2);

  iov[2].iov_base = part3;
  iov[2].iov_len = strlen(part3);

  writev(1, iov, 3);
}
```

7.2 Données ancillaires

7.2.1 Notion

Supposons que l'un des utilisateurs de votre système UNIX soit chargé du serveur Web mais, que pour des raisons de sécurité, vous ne voulez pas lui confier le mot de passe du super-utilisateur. Il y a un petit problème car UNIX réserve tous les numéros de port inférieurs à 1 024, et donc le port 80, au super-utilisateur.

Heureusement le problème peut être réglé en utilisant les **données ancillaires** (*ancillary data* en anglais, c'est-à-dire relatives aux servantes), qui sont des données supplémentaires ou auxiliaires envoyées en même temps que d'autres données, dites données normales. On les appelle aussi **données auxiliaires** ou **données de contrôle.**

La solution au problème énoncé ci-dessus se trouve dans la réalisation d'un serveur de sockets. Il s'agit d'un serveur local possédant les droits du super-utilisateur (et donc pouvant accéder au port 80). Le client local, par exemple celui qui doit s'occuper du serveur Web, lui envoie un numéro de port. Le serveur vérifie son identité pour savoir s'il est bien accrédité ; si c'est le cas, il crée une socket pour le port demandé et lui envoie le descripteur de fichier associé à celle-ci.

Les données ancillaires doivent donc porter sur au moins deux types de données : les descripteurs de fichier et les identificateurs. On utilise en fait un regroupement d'identificateurs appelé pièces d'identité (*credentials* en anglais). Les pièces d'identités sont fournies par le noyau Linux lui-même, jamais par l'utilisateur. Elles peuvent donc être considérées comme sûres sur un système local (mais pas pour un hôte distant, puisque le noyau distant peut être manipulé). Les données ancillaires ne sont implémentées que dans le cas d'un système local.

7.2.2 Les fonctions de transmission de messages

La dernière forme de fonctions d'entrée-sortie sur socket utilise une nouvelle entité, dite en-tête de message, qui permet la transmission d'une adresse de socket, de données dispersées et de données ancillaires.

7.2.2.1 Les en-têtes de message

Un en-tête de message est une entité du type `struct msghdr`:

```
struct msghdr
  {
  void          *msg_name;
  socklen_t     msg_namelen;
  struct iovec  *msg_iov;
  size_t        msg_iovlen;
  void          *msg_control;
  size_t        msg_controllen;
  int           msg_flags;
  };
```

comprenant:

- l'adresse de la socket et sa longueur. Ces deux premiers arguments ne sont nécessaires que dans le cas des communications non connectées (UDP par exemple). Puisque le type du champ de l'adresse est `(void *)`, on n'a pas besoin de convertir celle-ci au type `(struct sockaddr *)`.
- l'adresse et la dimension du vecteur des données dispersées.
- le tampon des données ancillaires et sa taille (en octets).
- le vecteur des drapeaux. Ce dernier champ n'est utile que pour la fonction de lecture. Les coordonnées possibles sont les suivantes:
  - `MSG_EOR` (pour *End Of Record*): la fin de l'enregistrement a été reçue; utile pour le type de communication `SOCK_SEQPACKET`;
  - `MSG_TRUNC`: la fin du datagramme a été tronquée car le tampon de réception était trop petit pour le recevoir en entier;
  - `MSG_CTRUNC`: les données ancillaires (dites aussi de contrôle, rappelons-le) ont été tronquées car le tampon de réception était trop petit pour les recevoir en entier;
  - `MSG_OOB`: des données urgentes ont été reçues;
  - `MSG_ERRQUEUE`: aucune donnée n'a été reçue mais une erreur a été renvoyée.

Les données ancillaires doivent répondre à une syntaxe sur laquelle nous reviendrons ci-dessous, après avoir vu les fonctions qui permettent de les transmettre.

7.2.2.2 Les fonctions

Les fonctions de lecture-écriture permettant les données ancillaires sont les suivantes:

```
#include <sys/types.h>
#include <sys/socket.h>

int sendmsg(int s, const struct msghdr *msg, unsigned int flags);
int recvmsg(int s, const struct msghdr *msg, unsigned int flags);
```

possédant trois arguments:

- le numéro du fichier de socket auquel envoyer ou duquel recevoir le message;

- l'adresse de l'en-tête de message ;
- le vecteur des drapeaux dont nous avons vu les valeurs possibles à propos des données urgentes.

La valeur de retour indique le nombre d'octets envoyés ou reçus et -1 en cas d'erreur.

7.2.3 Structure des données ancillaires

7.2.3.1 Structure générale

Le champ des données ancillaires peut comprendre 0, une, deux ou plusieurs données. Chacune d'entre elles comprend un en-tête de contrôle, éventuellement des octets de remplissage, suivi des données elles-mêmes, elles-mêmes suivies éventuellement d'octets de remplissage, comme l'indique la figure ci-dessous ([GRA–00], p. 436) :

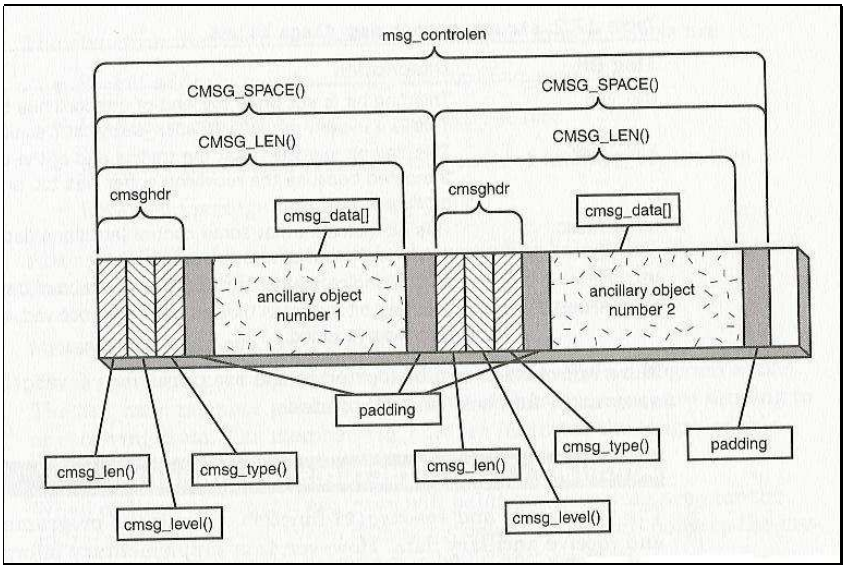


FIG. 7.1 – *Données ancillaires*

7.2.3.2 En-tête de contrôle

L'en-tête de contrôle est une entité du type `struct cmsghdr` (pour *Control MeSsaGe HeaDeR*) :

```
struct cmsghdr
    {
    socklen_t cmsg_len;
    int       cmsg_level;
    int       cmsg_type;
/*  u_char    smsg_data[]; */
    };
```

comprenant trois champs :

- la taille, en nombre d'octets, de la donnée ancillaire, incluant l'en-tête ;
- le niveau de protocole d'origine, seul `SOL_SOCKET` nous intéressant ici ;

- le type de message de contrôle; les deux types possibles pour `SOL_SOCKET` sont:
  - `SCM_RIGHTS`: la donnée ancillaire est un descripteur de fichier;
  - `SCM_CREDENTIALS`: la donnée ancillaire est une structure contenant des pièces d'identité.

Le champ `cmsg_data` n'existe pas vraiment; il est là pour indiquer où se trouvent placées physiquement les données.

On peut obtenir plus d'informations avec l'entrée `cmsg` du manuel en ligne et la [RFC 2292].

7.2.3.3 Les macros concernant les données ancillaires

Du fait de la complexité de la structuration des données ancillaires, un certain nombre de macros C sont fournies pour en faciliter l'utilisation. En voici les protocoles:

```
#include <sys/socket.h>

struct cmsghdr *CMSG_FIRSTHDR(struct msghdr *msgh);
struct cmsghdr *CMSG_NXTHDR(struct msghdr *msgh, struct cmsghdr *cmsg);
size_t CMSG_ALIGN(size_t length);
size_t CMSG_SPACE(size_t length);
size_t CMSG_LEN(size_t length);
void *CMSG_DATA(struct cmsghdr *cmsg);
```

- La macro `CMSG_LEN()` accepte en paramètre la taille de l'objet que l'on veut placer dans le tampon des données ancillaires. Elle renvoie la valeur à placer dans le champ `cmsg_len` de l'en-tête.
- La macro `CMSG_SPACE()` est utilisée pour calculer l'espace total requis par la donnée ancillaire et son en-tête, sans inclure les octets de remplissage de données (par contre ceux concernant l'en-tête sont inclus). Elle est utile pour déterminer la taille du tampon.
- La macro `CMSG_DATA()` prend un pointeur sur un en-tête de message de contrôle en entrée. La valeur de pointeur renvoyée est l'adresse du premier octet de donnée ancillaire qui suit cet en-tête (et éventuellement les octets de remplissage).
- La macro `CMSG_ALIGN()` est une extension de Linux qui ne fait pas partie du standard Posix.1g. Elle prend une longueur en octets en entrée et renvoie une longueur qui tient compte des octets de remplissage pour que l'alignement soit maintenu.
- La macro `CMSG_FIRSTHDR()` prend en entrée un pointeur sur un en-tête de message. Elle évalue les champs de cet en-tête pour savoir si des données ancillaires existent et, le cas échéant, renvoye le pointeur de l'en-tête de message de contrôle de la première donnée ancillaire du tampon, la valeur `NULL` sinon.
- La macro `CMSG_NXTHDR()` prend en entrée un pointeur sur un en-tête de message et un pointeur sur l'en-tête de message de contrôle en cours. Elle renvoie le pointeur de l'en-tête de message de contrôle suivant s'il existe, la valeur `NULL` sinon.

7.2.3.4 Pièces d'identité d'un utilisateur

Une pièce d'identité est une entité du type `struct ucred` (pour *User CREDentials*):

```
struct ucred {
        __u32   pid;
        __u32   uid;
        __u32   gid;
};
```

qui comprend trois champs: un identificateur de processus, un identificateur d'utilisateur et un identificateur de groupe d'utilisateurs.

7.3 Les options

Pour récupérer et positionner les options d'une socket, on utilise les fonctions suivantes :

```
#include <sys/types.h>
#include <sys/socket.h>

int getsockopt(int s,
               int level,
               int optname,
               void *opval,
               socklen_t *optel);
int setsockopt(int s,
               int level,
               int optname,
               void *opval,
               socklen_t optel);
```

dont la signification de chacun des cinq arguments est la suivante :

- `s` est le numéro de fichier duquel inspecter (auquel positionner) l'option.
- `level` est le niveau de protocole, qui prend l'une des deux valeurs suivantes :
  - `SOL_SOCKET` pour le niveau utilisateur ;
  - `SOL_TCP` pour le niveau transport TCP.
- `optname` est le nom de l'option. Dans le cas du niveau `SOL_SOCKET`, il s'agit de l'une des valeurs suivantes :
  - `SO_REUSEADDR` permet à un serveur d'accepter un nouveau client voulant utiliser la même adresse (même adresse IP et même numéro de port) qu'un client déjà actif. Ceci permet, par exemple, à deux processus distants d'accéder à l'ordinateur *via* telnet ou le Web.
  - `SO_KEEPALIVE` spécifie à un serveur d'envoyer un message à une socket distante si elle est inactive depuis assez longtemps (en général deux heures).
  - `SO_LINGER` permet de contrôler comment une socket est arrêtée lors de l'appel de `close()`, pour les types de communication orientés connexion (tel que TCP) uniquement.
  - `SO_BROADCAST` permet la diffusion générale avec un type de communication non connecté, UDP par exemple.
  - `SO_OOBINLINE` permet l'envoi de données urgentes.
  - `SO_SNDBUF` pour la taille du tampon d'envoi, la valeur étant un entier (le nombre d'octets).
  - `SO_RCVBUF` pour la taille du tampon de réception, la valeur étant un entier (le nombres d'octets).
  - `SO_TYPE` pour le type de socket (récupération uniquement), dont les deux valeurs entières principales sont `SOCK_STREAM` et `SOCK_DGRAM`.
  - et enfin `SO_ERROR`.
- `optval` est un pointeur sur le tampon de réception (ou d'envoi) de la valeur de l'option.
- `optlen` est (un pointeur, dans le cas de la récupération, car il s'agit d'un passage par adresse sur) la longueur de l'option.

Chapitre 8

Un exemple de carte réseau : 3Com 501

Dans le cas d'Ethernet, la couche liaison est en grande partie mise en œuvre au sein d'un **adaptateur** ou **carte réseau**, l'une des nombreuses cartes insérées sur la carte mère (comme la carte graphique ou la carte son). Il s'agit d'une unité semi-autonome qui reçoit un paquet de la part de la couche réseau de l'émetteur. La carte réseau encapsule ce paquet au sein d'une trame et la transmet au support physique. À l'autre extrémité, la carte du destinataire extrait le paquet de la trame, le place quelque part en mémoire vive et déclenche une interruption matérielle pour prévenir qu'un paquet est arrivé.

La carte réseau peut être plus ou moins perfectionnée. Elle peut ajouter les bits de détection d'erreurs en émission et les contrôler en réception. Le service de transport fiable peut également être entièrement mis en œuvre à son niveau (notamment l'insertion de numéros de séquence, l'activation de temporisateurs et l'échange d'acquittements). Elle peut supprimer des trames réceptionnées contenant des erreurs sans en avertir l'hôte.

L'une des premières cartes réseau conçue par 3Com (l'entreprise créée par les inventeurs d'Ethernet) pour les micro-ordinateurs compatibles PC fut la carte 3Com 501, sortie en 1988. C'est aussi l'une des premières à avoir été prise en compte par Linux.

8.1 Description de la carte

Nous n'allons pas donner de description physique de cette carte ni comment l'installer sur un PC. Il s'agit de toute façon d'une carte maintenant ancienne. Les éléments pour la programmation de cette carte sont décrits, incomplètement, dans [3Com501].

Cette carte est un contrôleur/transcepteur pour les compatibles PC qui est conforme à la spécification Ethernet, version 1.0 du 30 septembre 1980. Elle implémente les couches physique et liaison du modèle ISO. Elle comprend un contrôleur pour la couche liaison, appelé **8001** ou **EDLC** et un tampon de trame de 2 Ko, suffisant pour contenir une trame Ethernet, en émission ou en réception. Le tampon peut opérer avec le contrôleur de DMA 8237A des compatibles PC. Le contrôleur peut utiliser le contrôleur d'interruption 8259A des compatibles PC.

8.1.1 Interface logicielle

L'adresse de base, comme port d'entrée-sortie du microprocesseur, peut être choisie parmi un certain nombre de valeurs à l'aide de cavaliers.

L'interface logicielle s'effectue à l'aide de 16 registres, accessibles comme octets à partir de l'adresse de base. Ces registres portent les noms suivants, suivant qu'on lit ou qu'on écrit :

Décalage	Lecture	Écriture
0		Adresse de la station 0
1		Adresse de la station 1
2		Adresse de la station 2
3		Adresse de la station 3
4		Adresse de la station 4
5		Adresse de la station 5
6	Receive Status	Receive Command
7	Transmit Status	Transmit Command
8	GP Buffer Pointer [LSB]	GP Buffer Pointer [LSB]
9	GP Buffer Pointer [MSB]	GP Buffer Pointer [MSB]
A	RCV Buffer Pointer [LSB]	RCV Buffer Pointer Clear
B	RCV Buffer Pointer [MSB]	
C	Ethernet Address Prom Window	
D		
E	Auxiliary Status (CSR)	Auxiliary Command (CSR)
F	Buffer Window	Buffer Window

- Le registre F, *Buffer Window*, permet d'accéder, en écriture comme en lecture, à un octet du tampon parmi les 2 K possibles, d'où son nom de fenêtre.
- Le pointeur GP (*General Purpose buffer pointer*) contient la position de cette fenêtre dans le tampon. On a besoin de onze bits pour balayer l'amplitude 0 à 2047. L'octet de poids faible constitue le registre 8 et l'octet de poids fort (avec 5 bits ayant une valeur nulle) le registre 9. Une lecture ou une écriture dans la fenêtre incrémente automatiquement GP.
- Le pointeur RP (*Receive Pointer*) est également un pointeur de onze bits. Il est incrémenté à chaque fois que la carte reçoit un octet. On peut en lire la valeur grâce aux registres A et B (en lecture) et le remettre à zéro grâce au registre A (en écriture).
- L'adresse MAC de la carte est entièrement libre, stockée en mémoire PROM. On écrit ses six octets en utilisant les registres 0 à 5. On lit l'adresse à travers le registre C de taille un octet en positionnant les bits 0–2 de GP pour indiquer l'octet que l'on veut lire. La lecture de ce registre n'incrémente pas automatiquement GP.

– Les autres registres s'appellent RSR (*Receive Status Register*), RCR (*Receive Command Register*), TSR (*Transmit Status Register*), TCR (*Transmit Command Register*), ASR (*Auxiliary Status Register*) et ACR (*Auxiliary Command Register*) avec une dénomination qui rappelle leur fonction.

8.1.2 Le registre de commande d'émission : TCR

Lorsqu'on envoie un paquet à la carte 3Com 501 pour qu'elle émette une trame, celle-ci essaie de l'émettre et renvoie l'un des diagnostics suivants : émission réussie (c'est-à-dire que la trame a quittée la carte, mais on ne sait pas si elle est parvenue à destination), collision, seizième collision ou dépassement de capacité.

Après chaque collision, les données demeurent dans le tampon mais le pilote doit remettre GP à zéro et redémarrer explicitement l'émission. Le contrôleur attend alors le délai approprié et essaie d'émettre à nouveau.

Après la seizième collision, l'essai d'émission doit être abandonné d'après les règles du standard Ethernet.

Le dépassement de capacité survient lorsqu'on essaie d'émettre une trame avec un mauvais CRC ou pour essayer de diagnostiquer quelque chose.

Le pilote peut choisir d'ignorer ou de détecter chacun de ces diagnostics. Il l'indique grâce au TCR :

7	6	5	4	3	2	1	0
non utilisés				réussi	collision 16	collision	dépassement

Un bit à 1 invite à révéler le diagnostic dans le TSR. Les raisons du dépassement de capacité à détecter sont précisées dans le registre ACR. Si on veut qu'une interruption soit déclenchée lors de la détection d'un de ces diagnostics, il faut le préciser dans le registre ACR.

8.1.3 Le registre de statut d'émission : TSR

Le contrôleur change la valeur du TSR après chaque essai d'émission. La structure de ce registre est la suivante :

7	6	5	4	3	2	1	0
non utilisés				prêt	16 collisions	collision	dépassement

8.1.4 Le registre de commande de réception : RCR

Le pilote peut programmer la carte pour qu'elle n'accepte que les trames de certaines classes ; les autres trames sont alors ignorées automatiquement. Lorsqu'une trame est ignorée, RP est remis à zéro et le contrôleur est prêt à recevoir la trame suivante.

La structure du registre RCR est la suivante :

7	6	5	4	3	2	1	0
mode		accepte	EOF	trop court	trop lent	CRC	dépassement

– Le mode d'acceptation des adresses (*Address Match Mode*) détermine quelles sont les trames que l'on veut accepter au vu de leur adresse de destination :

  – 0 – receveur inhibé, ne reçoit aucune trame ;
  – 1 – reçoit toutes les adresses ;

- 2 – reçoit l'adresse de la station et la diffusion générale ;
- 3 – reçoit l'adresse de la station et la diffusion restreinte.

- Le bit 5 positionné à 1 indique que l'on ne doit accepter que les trames bien constituées (taille légale, pas d'erreur de CRC et pas de dépassement de capacité).
- Le bit 4 indique que l'on ne doit accepter que les trames sans erreur de dépassement de capacité, c'est-à-dire celles pour lesquelles un **EOF** (*End Of Frame*) a été détecté.
- Le bit 3 indique que l'on ne doit pas accepter les trames trop courtes (moins de 60 octets, en excluant le préambule et le CRC). On considère en effet qu'il s'agit certainement d'un fragment dû à un problème de collision.
- Le bit 2 indique qu'il faut détecter un débit trop lent (*dribble error*). La trame sera cependant chargée.
- Le bit 1 indique que l'on ne doit pas accepter les trames présentant une erreur de CRC.
- Le bit 0 indique qu'il faut détecter les erreurs de dépassement de capacité, c'est-à-dire que le contrôleur détecte une trame intéressante alors que le tampon n'est pas disponible.

8.1.5 Le registre de statut de réception : RSR

Le pilote définit les trames ayant un intérêt en positionnant le registre de commande en réception. Le contrôleur change le registre de statut de réception après chaque trame détectée sur le réseau, qu'elle présente ou non un intérêt. La structure de ce registre est la suivante :

7	6	5	4	3	2	1	0
Vieux	non défini	bonne trame	erreur EOF	trop court	trop lent	erreur CRC	erreur de dépassement

- Lorsque le contrôleur détecte une trame intéressante, le bit 7 (*Stale Receive Status*) est positionné à 0. Le contrôleur n'accepte plus alors de trame jusqu'à la lecture du registre. La lecture positionne ce bit à 1.
- Le bit 5 vaut 1 si la trame reçue est intéressante.
- Les bits 4, 3 et 1 spécifient la cause de l'erreur sinon : trame trop longue, trame trop courte, erreur de CRC.
- Si la trame a été reçue trop lentement, le bit 2 est positionné à 1.
- Si le contrôleur n'est pas en position de réception lorsqu'une trame arrive, le bit 0 est positionné à 1.

8.1.6 Le registre de commande auxiliaire : ACR

La structure de ce registre est la suivante :

7	6	5	4	3	2	1	0
Reset	RIDE	DMA	non utilisé	contrôle		CRC	IRE

- Positionner le bit `reset` à 1 permet de réinitialiser tous les registres de statut et de contrôle. Le pilote doit absolument le repositionner à 0 ensuite.

- Le bit RIDE (pour l'anglais *Request Interrupt and DMA Enable*) spécifie à la carte de positionner les signaux IRQ (*Interrupt ReQuest*) et DRQ (*Dma service ReQuest*) sur le bus système. Lorsque le RIDE vaut 0, le contrôleur ne peut pas engendrer de transferts DMA et ne peut engendrer des interruptions que si le bit IRE (pour l'anglais *Interrupt Request Enable*) vaut 1.
- Les bits 2 et 3 contrôlent l'accès au tampon :
  - 0 – Le bus système a accès au tampon, qui n'est plus relié au réseau dans ce cas.
  - 1 – Mode transmission (suivi automatiquement du mode réception).
  - 2 – Mode réception.
  - 3 – Mode périphérique en boucle.
- Si le bit 1 vaut 1, le contrôleur peut transmettre des trames ayant un mauvais CRC.

8.1.7 Le registre de statut auxiliaire : ASR

La structure du registre ASR est la suivante :

7	6	5	4	3	2	1	0
Occupé en émission	RIDE	requête DMA	DMA effectuée	contrôle du tampon		CRC	Occupé en réception

- Le bit 7 n'a de signification que lorsque le contrôle de tampon est positionné pour le périphérique en boucle ou pour émission. Il prend la valeur 1 lorsque le pilote démarre une émission (en positionnant le contrôle de tampon du RCA à 1). Il revient à zéro en cas de collision ou lorsque l'émission se termine avec succès. Le pilote peut distinguer ces deux cas en examinant le registre TSR.
- Le pilote démarre un transfert DMA en programmant le canal adéquat du contrôleur de DMA et en positionnant les bits RIDE et DMA Request du registre RCA du contrôleur de la carte réseau. Lorsque le transfert DMA est terminé, le bit 4 est positionné à 1. Le pilote efface ce bit en positionnant le bit DMA Request du registre RCA à zéro.
- Le bit 0 est positionné à un lorsque le contrôleur est armé pour recevoir une trame. Ceci se fait automatiquement après la transmission d'une trame ou lorsque le pilote positionne le contrôle de tampon en réception ou pour le périphérique en boucle. Ce bit revient à zéro après que le contrôleur ait accepté une trame. Le pilote doit alors attendre 800 nanosecondes avant de lire le registre RSR.

8.2 Émission et réception

8.2.1 Choix de l'adresse de base et de l'IRQ

On peut choisir l'adresse de base de la carte parmi 32 (ou même 64 suivant le modèle choisi) valeurs possibles, le canal IRQ parmi deux valeurs (3 ou 5) et le canal DMA parmi deux valeurs (1 ou 3) en positionnant physiquement des cavaliers.

Nous verrons que, en ce qui concerne l'implémentation Linux, on commence par considérer les valeurs par défaut à la sortie d'usine (à savoir 280, 1 et 3 respectivement) dans le noyau. L'administrateur système peut changer ces valeurs grâce à la commande `ifconfig` de configuration de l'interface physique.

8.2.2 Émission d'une trame

Pour émettre une trame, il faut :

- Positionner le contrôle de tampon (registre ACR) à zéro. Ceci procure au bus système un accès au tampon.
- Charger le paquet dans le tampon de façon telle que le dernier octet du paquet coïncide avec le dernier octet du tampon, à savoir `7FFh` pour `GP`.
- Positionner `GP` pour qu'il pointe sur le premier octet du paquet.
- Démarrer l'émission en positionnant le contrôle de tampon (registre ACR) à un.
- Lire le `TSR` pour déterminer s'il y a eu collision ou succès. L'émission est terminée lorsque le bit *occupé en émission* de `ASR` est revenu à zéro.

En cas de collision, il faut positionner le contrôle de tampon à zéro, repositionner le `GP` puis positionner le contrôle de tampon à un. Ceci permet d'émettre à nouveau la trame.

8.2.3 Réception d'une trame

Pour recevoir une trame, il faut :

- Positionner `RP` à zéro et le contrôle de tampon (registre ACR) à deux.
- Attendre 800 nanosecondes après que le bit *occupé en réception* de `ASR` soit revenu à zéro pour lire le registre `RSR` et déterminer ainsi le statut de la trame qui vient d'être reçue.
- La taille du paquet, en octets sans préambule et sans CRC, est alors contenue dans `RP`. Le début du paquet se trouve à l'adresse `000h` pour `GP`. Le pilote doit positionner le contrôle de tampon (registre ACR) à zéro avant de lire le tampon.

Chapitre 9

Quelques protocoles réseau

Nous allons décrire quelques protocoles réseau dans ce chapitre, plus précisément Ethernet, le protocole de couche réseau IPv4 et le protocole de couche transport UDP.

9.1 Protocole Ethernet

9.1.1 Protocole de sous-couche MAC pour Ethernet

9.1.1.1 Adresse physique MAC

Ethernet [IEEE-802.3] et quelques autres standards utilisent une adresse physique sur 48 bits, appelée **adresse MAC**, dont la structure est donnée sur la figure suivante :

1 bit	1 bit	22 bits	24 bits
I/G	U/L	Adresse de sous-réseau	Partie locale

- Le bit `I/G`, le moins significatif de l'adresse, est le bit d'adresse d'individu ou de groupe. Si ce bit vaut 0, l'adresse est celle d'un individu. S'il vaut 1, le reste du champ de l'adresse identifie une adresse de groupe qui nécessite une résolution supplémentaire.

  Lorsqu'une trame est adressée à un groupe, toutes les stations du groupe la reçoivent. Une diffusion de ce type est appelée **diffusion restreinte**, **diffusion multidestinataire** ou **diffusion multidiffusion** (*multicast* en anglais).
- Le deuxième bit, `U/L`, est le bit local ou universel. S'il vaut 0, l'adresse de sous-réseau a été définie par l'organisme d'administration universel. S'il vaut 1, l'adresse de sous-réseau a été définie localement.

  L'organisme d'administration universel fut d'abord Xerox (le créateur d'Ethernet, comme nous l'avons vu) puis ensuite l'IEEE au niveau mondial.
- Dans le cas où le deuxième bit vaut 0, les 24 premiers bits constituent l'**OUI** (pour *Organization Unique Identifier*).
- Les 24 derniers bits identifient l'adresse dans le réseau local et est gérée localement.

Si la totalité de l'adresse est composée de bits à 1, l'adresse a une signification spéciale : elle indique que toutes les stations du réseau sont considérées comme destinataires. On parle de **diffusion générale** (*broadcast* en anglais).

Un sous-réseau est donc identifié par 22 bits. Ce système autorise donc 2^{22} combinaisons – soit 4 millions environ – et il est fort possible qu'au rythme de croissance actuel d'Internet, on se retrouve à court d'OUI. Si une organisation est à court d'adresses physiques, l'IEEE a la possibilité de lui affecter une deuxième adresse de sous-réseau.

9.1.1.2 Les trames Ethernet DIX

Commençons par décrire une trame de l'Ethernet d'origine, appelée **trame DIX** (pour *DEC-Intel-Xerox*, rappelons-le). Une trame Ethernet ne comprend pas seulement un en-tête et des données. Elle est suivie d'un suffixe comme le montre le schéma suivant :

Préambule	Adresse destinataire	Adresse émetteur	Type	Données	CRC
64 bits	48 bits	48 bits	16 bits	Longueur variable	32 bits

- Le **préambule** est un ensemble de bits dont la fonction principale est de synchroniser le processus de communication et de mettre en évidence tout bruit parasite affectant les premiers bits envoyés.

  Dans le cas d'une trame de l'Ethernet d'origine, chacun des 8 octets du préambule contient la séquence binaire `10101010`. Le codage Manchester de cette séquence produit un signal de forme rectangulaire de 10 MHz pendant 6,4μs qui permet à l'horloge du récepteur

de se caler avec celle de l'émetteur. Les parties doivent ensuite rester synchronisées pour le reste de la trame et utiliser le code Manchester pour identifier les délimitations des bits.

- Les adresses MAC du destinataire et de l'expéditeur suivent. La norme autorise des adresses sur 2 ou 6 octets, mais les paramètres définis pour la norme à 10 Mbit/s en bande de base n'emploient que des adresses sur 6 octets.
- Dans le cas d'une trame DIX, le champ suivant de 16 bits, le **type**, permet au récepteur le démultiplexage au niveau de la couche réseau, autrement dit ce qu'il doit faire avec la trame. En effet plusieurs protocoles de couche réseau peuvent être employés en même temps sur un même ordinateur. Ainsi, lorsqu'une trame Ethernet arrive, le noyau du système doit savoir à quel protocole la remettre. C'est ce champ qui lui permet de le savoir.

  Voici quelques valeurs de ce champ :

Valeur décimale	Description
512	Xerox PUP
2048	Internet Protocol (IP)
2051	ECMA
2053	X.25 Level 3
2054	Address Resolution Protocol (ARP)
32821	Reverse ARP
32823	Appletalk

  dont 2048 et 2054 seront les seules valeurs qui nous intéresseront.
- Les données suivent l'indicateur de type, dont la longueur varie entre 46 et 1 500 octets suivant le type de protocole.

  La valeur maximum de 1 500 a été choisie de façon arbitraire, principalement parce qu'un transcepteur devait disposer de suffisamment de mémoire pour contenir une trame entière et qu'en 1978 la mémoire RAM était coûteuse.

  Outre une longueur maximale, une trame doit respecter une longueur minimale. Bien qu'un champ de données de zéro octet soit parfois utile, cela pose problème. Lorsqu'un transcepteur détecte une collision, il tronque la trame en cours, si bien que des bits ou des fragments de trame égarés apparaissent continuellement sur le câble. Pour faciliter la distinction entre les trames valides et les rebuts parasites, Ethernet exige que les trames valides possèdent au moins 64 octets, du champ d'adresse de destination au champ de somme de contrôle, ces deux champs inclus. Si le champ des données a une taille inférieure à 46 octets, elles sont complétées par des 0 jusqu'à obtenir 46 octets, ce qui constitue le **champ de remplissage.**
- À la fin de la trame se trouvent quatre octets de vérification, (**CRC** pour *Cyclic Redundancy Check*), qui permet de s'assurer plus ou moins que le contenu de la trame n'a pas été modifié lors de la transmission. Chaque passerelle qui figure sur la route de transmission doit calculer une valeur de CRC pour la trame et la comparer à la valeur qui figure à la fin de celle-ci. Si les deux sont égales, la trame est envoyée un peu plus loin dans le réseau. Sinon, c'est que la trame a été altérée en cours de route et elle doit donc être détruite (elle sera retransmise – après expiration d'un temporisateur (*timer*) – par la machine qui l'a envoyée).

9.1.2 Protocole 802.3 de la sous-couche LLC

La sous-couche LLC comprend un certain nombre de protocoles. Dans le cas de 802.3, le champ données de la trame ne comporte pas d'en-tête de sous-couche 2b ; par contre la signification des champs de l'en-tête Ethernet (*a priori* de la sous-couche MAC) a légèrement changé. On parle alors de **trame Ethernet 802.3.** Il y a deux changements :

- Le premier changement concerne le préambule. Les sept premiers octets restent inchangés ; le huitième octet sert de délimiteur de début de trame (**SFD** pour *Start Frame Delimiter*) afin de rendre le format compatible avec les standards 802.4 et 802.5.
- Le second changement a d'abord transformé le champ de type en champ de **longueur** de trame. Bien sûr, il n'était plus possible pour un récepteur de savoir ce qu'il fallait faire avec une trame entrante, mais ce problème a été résolu par l'ajout d'un petit en-tête dans la portion des données pour apporter cette information.

  Malheureusement, à l'époque où la norme 802.3 fut publiée, il y avait déjà tellement d'équipements et de logiciels en circulation pour l'Ethernet DIX que peu de fabricants et d'utilisateurs ont accueilli avec enthousiasme le changement du champ de type en champ de longueur. En 1997, l'IEEE jeta l'éponge et annonça que les deux formats étaient valides. Heureusement, la plupart des champs de type utilisés avant 1997 étaient d'une valeur supérieure à 1 500 (la longueur maximum des données). Par conséquent, une valeur inférieure ou égale à 1 500 peut être considérée comme étant une longueur et une valeur supérieure à 1 500 comme étant un type.

9.2 Le protocole de couche réseau IPv4

9.2.1 Étude générale de la couche réseau

9.2.1.1 Ce que fait la couche réseau

La couche réseau gère le déplacement des paquets dans un réseau, c'est-à-dire sur des machines qui ne sont pas nécessairement adjacantes. Elle **fragmente** éventuellement les données de la couche transport, pour obtenir des paquets d'une taille standard, et est responsable du **routage** des paquets, c'est-à-dire de la détermination du chemin à suivre par le paquet en proposant des itinéraires de substitution en cas de problème.

9.2.2 Ce que ne fait pas la couche réseau

Nous venons de voir que la couche réseau s'occupe de la fragmentation et du routage. Elle ne se préoccupe pas de la fiabilité de la livraison des paquets. Cette dernière n'est pas garantie car le paquet peut être retardé, mal routé ou altéré lors du fractionnement et du réassemblage des fragments. Il n'existe pas de fonctionnalité permettant de vérifier qu'un paquet envoyé a été correctement reçu. Le protocole IP de couche réseau dispose d'une somme de contrôle du contenu de l'en-tête d'un paquet mais pas de son contenu. Elle peut essayer de deviner le meilleur itinéraire jusqu'au prochain nœud sur le chemin, mais ne vérifie pas que le chemin choisi est le plus rapide ou le plus efficace.

La couche réseau est **sans connexion**, ce qui signifie qu'elle ne se soucie pas de savoir par quels nœuds passe un paquet, ni même quelles sont ses machines de départ et d'arrivée. Ces informations figurent dans l'en-tête, mais l'analyse et la transmission d'un paquet n'ont rien à voir avec l'analyse faite par la couche réseau de l'adresse d'émission et de réception.

9.2.3 Les tribulations d'un paquet IP

9.2.3.1 Hôte de départ

Lorsqu'une application doit envoyer un paquet sur le réseau, le protocole IPv4 de la couche réseau effectue un certain nombre de tâches simples :

- Elle vérifie tout d'abord si le paquet doit être fragmenté. IP autorise une taille de paquet maximale de 65 535 octets, ce qui est bien supérieur à ce que peuvent traiter la plupart des couches inférieures. Si c'est le cas elle crée les fragments.

  Remarquons que la fragmentation ne concerne que le protocole de transport UDP. Le protocole TCP n'en a pas besoin car la connexion lui permet de savoir le nombre d'octets qu'il peut envoyer.
- Elle construit ensuite, pour chaque fragment, le paquet IP en respectant les longueurs valides, telles qu'elles sont spécifiées par l'implémentation IP locale. Une **somme de contrôle** est calculée pour les données, puis l'en-tête IP est construit.
- Ensuite commence le routage. Il est nécessaire de déterminer le premier "saut" (*hop* en anglais), c'est-à-dire la première machine de la route vers la destination, afin de router le paquet vers la machine de destination : la machine elle-même ou une autre machine du réseau local, la machine de destination ou une **passerelle** si on utilise un interréseau. Si un routage précis est prédéterminé (à titre de contrôle en général), les informations sur celui-ci sont ajoutées à l'en-tête au moyen d'une **option**.
- Enfin le paquet est transmis à la couche inférieure pour être envoyé sur le réseau physique.

9.2.3.2 Passerelles et routeurs

À mesure qu'un paquet traverse l'interréseau, chaque passerelle effectue une série de tests :

- Une fois que la couche inférieure a enlevé l'en-tête de la trame reçue, la couche IP de la passerelle calcule la somme de contrôle et vérifie l'intégrité du paquet. Si les sommes de contrôle ne correspondent pas, le paquet est détruit et un message d'erreur est envoyé au composant émetteur *via* ICMP.
- Ensuite, un champ de **durée de vie** (TTL pour *Time To Live* en anglais) de l'en-tête IP est décrémenté puis examiné. Si la durée de vie du paquet a expirée, le paquet est écarté et un message d'erreur est envoyé au composant émetteur *via* ICMP.
- Après avoir déterminé le prochain saut de la route, en analysant l'adresse de destination ou à partir du routage spécifique indiqué dans le champ option de l'en-tête IP, le paquet est reconstruit avec une nouvelle valeur de TTL et donc une nouvelle somme de contrôle.
- Si une fragmentation est nécessaire, soit parce que la longueur du paquet a augmentée, soit du fait d'une limitation du logiciel d'envoi de la passerelle, le paquet est divisé, et de nouveaux paquets, contenant les informations d'en-tête adéquates, sont créés.
- Si un routage ou une estampille temporelle est nécessaire, il est ajouté.
- Enfin, le paquet est renvoyé à la couche inférieure.

9.2.3.3 Hôte de destination

Lorsque le paquet est finalement reçu par le composant récepteur :

- Le système effectue un calcul de la somme de contrôle et vérifie que celle-ci concorde avec le champ adéquat de l'en-tête. Si ce n'est pas le cas, un message ICMP est envoyé à l'émetteur.

- Il regarde ensuite s'il y a des fragments supplémentaires. L'opération inverse de la fragmentation s'appelle le **réassemblage.** Si d'autres paquets sont nécessaires pour réassembler le paquet originel, le système attend, après avoir lancé un **minuteur de réassemblage**, pour ne pas être bloqué si les paquets suivants mettent trop de temps à arriver.
- Si le composant ne peut réassembler toutes les parties du paquet originel avant que le minuteur n'atteigne 0, la partie du paquet arrivée est détruite et un message d'erreur est envoyé à l'émetteur *via* ICMP.

  Une des conséquences de cette étape est qu'un paquet fragmenté a moins de chances d'arriver à bon port qu'un paquet non fragmenté, ce qui explique pourquoi la plupart des applications essaient d'éviter autant que possible la fragmentation.
- Enfin l'en-tête IP est enlevé, le paquet originel est reconstruit (s'il a été fragmenté) et il passe à travers les couches pour arriver à la couche d'application.

9.2.4 Fragmentation

Pour pouvoir envoyer des paquets IP sur tous les types de réseau physique, le protocole IP doit être en mesure d'adapter la dimension de ceux-ci au type de réseau en place. Nous avons déjà vu que tout réseau physique comporte une taille de trame maximale, qualifiée de *MTU* (pour l'anglais *Maximum Transfert Unit*, unité de transfert maximale). Seuls des trames de cette taille peuvent être envoyées sur le réseau.

Si la MTU d'un support de transfert est inférieure à la taille d'un paquet à expédier, ce dernier doit être divisé en paquets IP plus petits, appelés **fragments.**

Il ne suffit pas que les protocoles de la couche de transport n'expédient que des paquets convenant à la MTU, tout au moins dans le cas non connecté. Un paquet peut traverser plusieurs réseaux différents avec des MTU différentes sur le trajet de l'hôte source à l'hôte cible. C'est pourquoi il faut appliquer une procédure plus flexible capable également de générer des paquets plus petits au sein d'un système de transmission (routeur) sur la couche IP. Cette procédure est appelée **fragmentation.**

La fragmentation sous-entend que le protocole IP de tout ordinateur IP (que ce soit un routeur ou un système d'extrémité) doit être en mesure de reconstituer ces fragments pour obtenir le paquet d'origine (*réassemblage*).

Chaque fragment d'un paquet IP découpé est un paquet IP complet qui contient un en-tête IP. Le paquet originel auquel appartient un fragment se reconnait grâce au champ identificateur de l'en-tête IP. Ce champ seul ne suffit pas à déterminer le fragment. On utilise également les champs adresse de l'émetteur, adresse cible et protocole.

Les différents fragments d'un datagramme peuvent emprunter des itinéraires différents pour parvenir à l'ordinateur cible et être également fragmentés plusieurs fois sur leurs trajets. La position des données d'un fragment au sein du datagramme IP d'origine est identifiée grâce au champ `décalage de fragment` de leur en-tête IP. Les sous-paquets sont repérés par un décalage et non par un numéro parce qu'un sous-paquet peut être fragmenté à tout moment le long de sa route. On ne peut pas revenir en arrière pour renuméroter les fragments.

Tous les fragments, jusqu'à l'avant-dernier, comportent un bit `MF` (pour l'anglais *More Fragments*) pour indiquer que d'autres fragments vont suivre.

La figure 9.1 ([WPRMB–02], p. 280) montre l'exemple d'un paquet IP qui doit être fragmenté plusieurs fois.

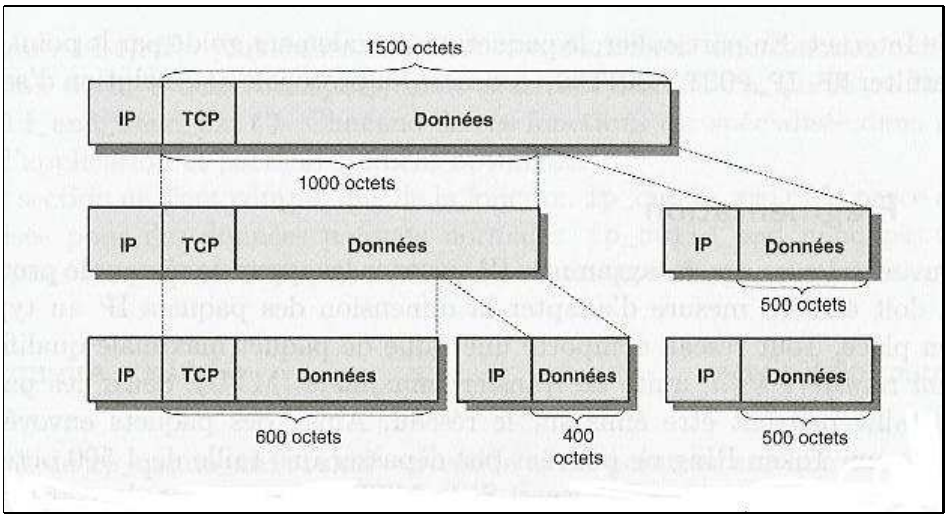


FIG. 9.1 – *Fragmentation d'un paquet IP*

9.2.5 Le routage

9.2.5.1 Notion

Le **routage** (*routing* en anglais) est l'une des tâches de la couche réseau. Il s'agit de déterminer la route à suivre pour aller d'un point A à un point B. Ceci est l'objet des ordinateurs spécialisés appelés **routeurs** (*router* en anglais) et non des hôtes d'extrémité. Le routage ne devrait donc pas nous intéresser dans ce livre mais en fait il a également des répercussions sur les hôtes d'extrémité.

Le routage comprend deux modules :

- La **redirection** (*forwarding* en anglais) recherche, pour chaque paquet, à partir de la destination de celui-ci dans une table l'interface sur laquelle transmettre ce paquet. Cette table est appelée **table de redirection** (*forwarding table* en anglais).
- La tâche de routage proprement dit consiste à construire et à maintenir la table de redirection.

La redirection est une procédure du noyau alors que la construction de la table est une procédure utilisateur. Le routage proprement dit ne nous intéressera ici qu'à travers la façon d'ajouter ou de retrancher un élément à cette table.

En gros le rôle de la table de redirection est le suivant : on lui fournit une adresse IP et elle renvoie le prochain saut (*next hop* en anglais, abrégé en `nh`), sous la forme de l'interface réseau sur laquelle envoyer le paquet (ce qui détermine le réseau local) et une adresse IP (ce qui détermine l'élément de ce réseau local). La détermination de l'interface réseau est importante car un routeur possède au minimum deux interfaces réseau et en moyenne quatre à cinq.

9.2.5.2 Routage et tables de hachage

Les adresses IPv4 ont une longueur de 32 bits. Une table à deux colonnes, une colonne pour placer une adresse IP par ligne et une colonne pour le prochain saut, est peu réaliste. Rien que la première colonne occuperait 4×2^{32} octets, soit 16 Go de mémoire.

L'idée est donc d'utiliser une table de hachage (figure 9-2, [C-P–02], p. 548). Connaissant la distribution approximative des adresses IP actuellement assignées, une fonction de hachage peut envoyer une adresse IP, parmi les 2^{32} adresses possibles, dans un ensemble plus petit.

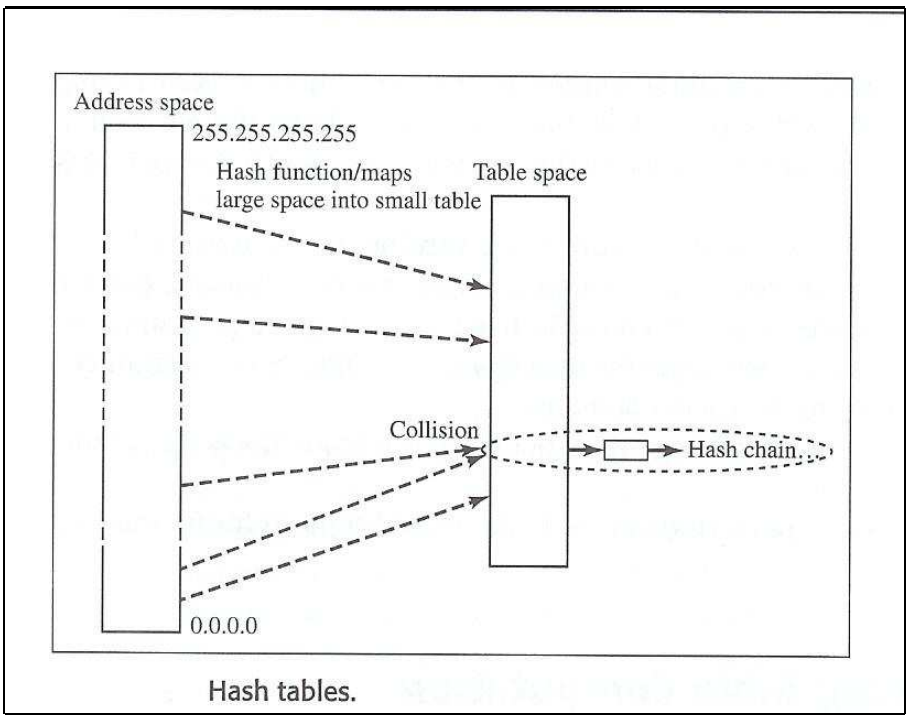


FIG. 9.2 – *Table de hachage*

Bien entendu il y aura inévitablement des collisions, aussi prevoit-on un chaînage sous la forme d'une liste chaînée pour les adresses qui donnent lieu au même index de hachage.

9.2.6 Routage et adresse IP

La distribution des adresses IP simplifie la conception de la fonction de hachage. En effet les adresses IPv4 ne sont pas de simple identifiants de nœuds réseau. Elles sont distribuées de façon à faciliter le routage. Les adresses IP sont structurées de manière hiérarchique et sont formées de deux parties distinctes : une **adresse de sous-réseau** (*network part* en anglais) et un **identifiant de système terminal** (*host part* en anglais). L'adresse de sous-réseau est identique pour tous les éléments présents dans un sous-réseau donné. Quant à l'identifiant de système terminal, il est destiné à distinguer les différents éléments au sein d'un sous-réseau donné.

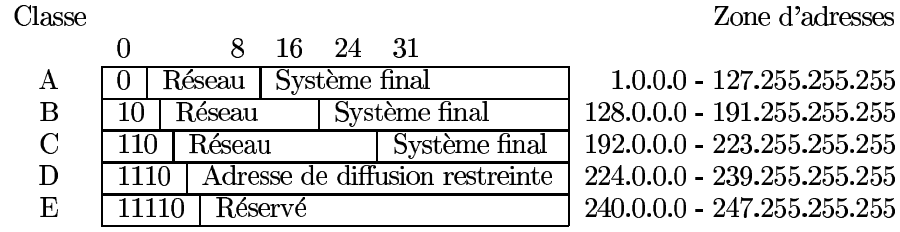
Dans le cas du routage, l'identifiant du système terminal peut être ignoré jusqu'à ce que le paquet soit parvenu dans le bon sous-réseau. De ce fait, les routeurs n'ont pas besoin de connaître les identifiants des systèmes terminaux, ce qui permet ainsi de réduire de manière notable la quantité d'informations à enregistrer, grâce au découpage de l'adresse IP en deux parties.

Il existe deux façons de savoir quelle est la taille de la partie de l'adresse IP correspondant au sous-réseau et celle correspondant à l'identifiant de système terminal, comme nous allons le voir. Dans les deux cas la partie sous-réseau d'une adresse IP se trouve toujours au début, d'où

le synonyme **préfixe réseau** que l'on rencontre quelquefois.

9.2.6.1 Classes d'adresses

Le premier système ([RFC 791]), datant de 1981, consiste à distinguer trois **classes d'adresses**, comme le montre la figure ci-dessous :



Les classes d'adresses A, B et C comprennent une partie sous-réseau respectivement de 7, 14 et 21 bits et un identifiant de système terminal de 24, 16 et 8 bits. L'appartenance à l'une de ces classes est déterminée par les premiers bits (forts) de l'adresse.

Ce schéma d'adressage a été conçu de manière à permettre à chaque réseau physique d'obtenir un identifiant sous-réseau appartenant à l'une des trois classes A, B ou C en fonction de la taille du réseau physique. L'idée était qu'il pouvait y avoir 256 sous-réseaux de classe A, tels que ARPANET et SATNET, 65 536 sous-réseaux de classe B, tels que Berkeley, le MIT, UCLA et UCL, et 16 777 216 sous-réseaux de classe C.

Cependant, il est très rapidement apparu que cette approche épuiserait très rapidement les préfixes réseau disponibles. Par ailleurs, les classes présentes se sont fréquemment montrées peu adaptées : un réseau de classe A pourrait compter près de 2^{24}, c'est-à-dire 16 777 216 systèmes terminaux, une valeur qui dépasse les besoins des plus grandes entreprises – indépendamment du fait qu'aucune technique réseau n'est en mesure de gérer un aussi grand nombre de systèmes terminaux. Les sous-réseaux de classe C – qui sont très nombreux – ne peuvent en revanche gérer que 254 systèmes terminaux, ce qui est généralement insuffisant.

9.2.6.2 Adressage sans classe

C'est la raison pour laquelle des techniques visant à une meilleure utilisation de l'espace d'adressage disponible ont été développées. L'idée fondamentale est de permettre que la limite entre le préfixe réseau et l'identifiant du système terminal soit placé à n'importe quelle position de bit, au lieu de ne l'autoriser qu'aux trois positions prévues par les classes d'adresses A, B et C. On utilise actuellement sur Internet le système **CIDR** (*Classless Inter-Domain Routing*) défini dans [RFC 1518] et [RFC 1519].

Les préfixes réseau peuvent avoir une longueur quelconque. L'information concernant la longueur de l'identifiant du sous-réseau ne peut plus être déterminée à partir des premiers bits de l'adresse. Cette information doit être transmise par ailleurs. On a le choix pour cela entre deux notations utilisateur :

- Dans la première notation, le nombre de bits correspondant au préfixe du sous-réseau est indiqué sous forme d'un nombre décimal, séparé de l'adresse par une barre oblique. Ainsi, l'adresse 192.168.152.0/21 désigne un sous-réseau dont le préfixe est représenté par les 21 premiers bits de l'adresse IP.
- La seconde notation consiste à ajouter à l'adresse IP un masque de bits de même longueur, le **masque réseau**, dans lequel tous les bits appartenant au préfixe du sous-réseau dans

l'adresse IP ont la valeur 1. Par exemple le réseau évoqué ci-dessus est noté :

192.168.152.0/255.255.248.0.

9.2.6.3 Remarque

Ce système a fonctionné parfaitement durant de nombreuses années, tant qu'un sous-réseau ne possèdait qu'un seul préfixe réseau (*single homed* en anglais). Ce n'est plus le cas de nos jours puisqu'il est apparu plus simple d'attribuer plusieurs préfixes réseau à un même sous-réseau (*multihoming* en anglais).

9.2.7 En-tête IPv4

Rappelons que l'unité de transfert utilisée par IP est appelée **paquet**, ou plus précisément **paquet Internet** ou **paquet IP**. Un paquet est constitué d'un en-tête IP suivi des données. La figure suivante montre l'agencement de l'en-tête IP, telle que définie par [RFC 791] ("*Internet Protocol*" écrite par Postel en 1981) :

Version (4 bits)	Longueur (4 bits)	Type (8 bits)	Longueur du paquet (16 bits)		
Identificateur				DF	MF	Décalage
TTL		Transport	Somme de contrôle		
Adresse émetteur					
Adresse destinataire					
Options	Remplissage				

Commentons les champs de cet en-tête un à un :

- Le **numéro de version** est un champ de 4 bits contenant le numéro de version IP que le paquet utilise.

  La version la plus utilisée est la version 4 mais quelques systèmes effectuent des tests avec la version 6. Nous décrivons la version 4 dans la suite.

  Ce champ est *a priori* nécessaire pour que le logiciel IP récepteur sache décoder le reste de l'en-tête, qui varie à chaque nouvelle version de protocole IP. En fait nous avons vu que la distinction se fait par le champ *protocole* de l'en-tête Ethernet au niveau de la couche MAC (`0x800` pour IPv4, `0x86DD` pour IPv6).

- La **longueur de l'en-tête** (`IHL` pour *Internet Header Length*) est un champ de 4 bits qui indique la longueur totale de l'en-tête IP construit par la machine émettrice. Cette longueur permet de déterminer où commencent les données puisqu'il n'existe pas de marqueur de début des données.

  Les spécifications d'IP (ainsi que de la plupart des autres protocoles de la suite TCP/IP) définissent comme unité de longueur le **mot**, un mot représentant 32 bits, soit 4 octets.

  L'en-tête IP a une longueur maximale de quinze mots, soit 60 octets. Le plus court en-tête autorisé par IP (sans aucune option) utilise cinq mots, soit 20 octets.

- Le champ suivant, de 8 bits (un octet), spécifiait à l'origine le **type de service** (`TOS` pour *Type Of Service*), qui indiquait aux routeurs comment traiter correctement le paquet.

  Les 8 bits de ce champ étaient répartis de la façon suivante :

Priorité	Délai	Débit	Fiabilité	non utilisé

  - Les trois premiers bits indiquaient l'**ordre de priorité** (ou précédence) du paquet, avec des valeurs comprises entre 0 (normal) et 7 (contrôle réseau). Plus cette valeur est grande, plus le datagramme est important, ce qui, en théorie du moins, le fera arriver plus vite à destination.
  - Les trois bits suivants sont des drapeaux d'un bit qui contrôlent le délai, le débit et la fiabilité du paquet. Un bit positionné à 1 indique un délai bas, un grand débit et une haute fiabilité respectivement.
  - Les deux derniers bits ne sont pas utilisés.

  En pratique, la plupart des implémentations des routeurs TCP/IP ignoraient ce champ et traitaient tous les paquets avec les mêms valeurs de priorité, de délai, de débit et de fiabilité. Il prenait donc la plupart du temps la valeur 0.

  La signification de ce champ a été changée dans [RFC 2474] pour devenir *Differentiation Services Codepoint*. Il indique maintenant le comportement de transmission utilisé, ce qui n'intéresse que les routeurs, aussi ne nous nous en occuperons pas dans ce livre.

- La **longueur du paquet** est un champ de 16 bits, soit deux octets, qui indique la longueur totale du paquet, en-tête compris, en octets.

  On a donc des paquets d'au plus 65 535 octets.

- L'**identificateur de paquet originel** (*Fragment-ID*) est un champ de 16 bits, soit deux octets, qui contient un identificateur unique créé par le nœud émetteur pour chaque paquet, avant que celui-ci ne soit éventuellement fragmenté. Ce numéro est nécessaire lors du réassemblage des fragments, afin de garantir que les fragments d'un paquet ne soient pas mêlés à ceux d'un autre.

  Comme nous l'avons déjà dit, un paquet est identifié par cet identificateur mais également par l'adresse de l'émetteur, l'adresse cible et le protocole de transport.

- Le champ **drapeaux** est un champ de trois bits qui contrôle la gestion des paquets lorsqu'une fragmentation est requise. Le premier bit n'est pas utilisé, les deux bits restants permettent de spécifier des drapeaux appelés `DF` (pour *Don't Fragment*, ne pas fragmenter) et `MF` (pour *More Fragments*, encore des fragments) :

  - Si le drapeau `DF` vaut 1, le paquet ne peut en aucun cas être fragmenté. Si la couche de logiciel IP ne peut pas envoyer le paquet à une autre machine sans le fragmenter et que ce bit vaut 1, le paquet est détruit et un message d'erreur doit être envoyé au composant émetteur *via* ICMP.
  - Si le drapeau `MF` vaut 1, c'est que le paquet en cours est suivi d'autres paquets (parfois appelés **sous-paquets**) qui devront être réassemblés pour former le paquet originel. Le dernier fragment d'un paquet envoyé doit avoir son drapeau `MF` égal à 0, de manière à ce que le composant récepteur sache qu'il doit cesser d'attendre de nouveaux sous-paquets. Comme l'ordre d'arrivée des fragments ne correspond pas forcément à leur ordre d'envoi, le drapeau `MF` est utilisé en conjonction avec le champ suivant pour indiquer à la machine réceptrice à quoi ressemble le paquet originel.

- Si le drapeau `MF` est égal à 1, ce qui indique un paquet fragmenté, le champ `décalage de fragment` de 13 bits contient la position, l'unité étant deux mots ou huit octets, au sein du paquet complet, du début des données contenues dans le paquet en cours. Ceci permet à IP de réassembler les paquets fragmentés dans leur ordre correct.

Puisque ce champ comporte 13 bits et que les décalages sont calculés en unités de 8 octets, on retrouve bien la longueur de paquet maximale de 65 535 octets. Puisque ce champ a une taille de 13 bits, seulement 8 192 fragments au maximum peuvent appartenir à un paquet IP originel. Tous les fragments, sauf le dernier, doivent avoir une longueur multiple de 8 octets, l'unité élémentaire de fragmentation.

- Le champ TTL (pour *Time To Live*, durée de vie), de taille un octet, indiquait à l'origine la période, en secondes, durant laquelle un paquet pouvait rester sur le réseau avant d'être détruit par le nœud en cours avec, dans ce cas, un message ICMP envoyé à la machine émettrice. Cette durée de vie permet d'éviter que des paquets IP circulent sans fin sur le réseau et ne saturent celui-ci.

  Cette période est fixée par le nœud émetteur lors de l'assemblage du paquet. Elle a généralement pour valeur 15 ou 30 secondes.

  Les standards TCP/IP stipulent que le champ TTL doit être diminué d'au moins une seconde à chaque nœud qui traite le paquet, même si le temps de traitement est inférieur à une seconde. Par ailleurs, lorsqu'un paquet est reçu par une passerelle, le temps d'arrivée est noté, pour que dans l'hypothèse où le paquet doive attendre avant d'être traité, ce temps d'attente soit comptabilisé dans le TTL. En conséquence, si une passerelle est surchargée et ne peut pas traiter rapidement tous les paquets, le minuteur du TTL peut expirer alors que le paquet attend d'être traité, et ce dernier est détruit.

  Ce compteur indique aujourd'hui le nombre maximal de systèmes intermédiaires (routeurs ou plutôt sauts).

- Le **protocole de transport** est un champ d'un octet qui contient le numéro d'identification du protocole de transport duquel provient ou sera confié le paquet. Les numéros sont définis par le IANA. Il existe actuellement environ 50 protocoles affectés d'un numéro de transport, codifiés dans [RFC 1700]. Les quatre protocoles les plus importants sont ICMP, qui porte le numéro 1, TCP le numéro 6, UDP le numéro 17 et IGMP le numéro 2.

- La **somme de contrôle** ne porte que sur l'en-tête de protocole (et non sur les données) pour accélérer les traitements.

  L'algorithme de contrôle prend le complément à un de la somme sur 16 bits de tous les mots de 16 bits.

  Comme le champ TTL est décrémenté lors du passage de chaque routeur, la somme de contrôle doit être recalculée par chaque routeur par lequel passe le paquet.

- Les champs **adresse de l'émetteur** et **adresse de destination** contiennent les adresses IP à 32 bits des composants émetteur et récepteur.

- Le champ **Options** est optionnel et ne nous intéressera pas vraiment dans cet ouvrage. Il est composé de plusieurs codes de longueurs variables. Si plus d'une option est utilisée dans un paquet, les options apparaissent les unes à la suite des autres dans l'en-tête IP. Toutes les options sont contrôlées par un octet, généralement divisé en trois champs : un *drapeau de copie* sur un bit, une *classe d'options* sur deux bits et un *numéro d'option* sur cinq bits.

  - Le **drapeau de copie** est utilisé pour indiquer la manière dont l'option est traitée lorsqu'une fragmentation est nécessaire dans une passerelle. Lorsque ce bit vaut 0, l'option doit être copiée sur le premier fragment mais pas sur les suivants. Si le bit vaut 1, elle doit être copiée sur tous les fragments.
  - Les deux bits de la **classe d'options** permet quatre classes. Pour le moment il n'existe que deux classes. La classe 0 indique que l'option s'applique au contrôle du paquet ou du réseau. La classe 2 indique que l'option a trait au débogage ou à l'administration.

– Les valeurs actuellement prises en charge pour les classes et numéros d'options figurent dans le tableau suivant :

Classe	Numéro	Description
0	0	Marque la fin de la liste d'options
0	1	Aucune option (utilisée pour le remplissage de caractères)
0	2	Options de sécurité (utilisation militaire seulement)
0	3	Routage tolérant
0	7	Active l'enregistrement du routage (ajoute des champs)
0	9	Routage strict
2	4	Marquage de la date activé (ajoute des champs)

Les options les plus intéressantes sont celles qui activent l'enregistrement du routage et de l'heure. Elles permettent de tracer le passage d'un datagramme au travers de l'interréseau, ce qui peut être utile pour émettre des diagnostics.

Le **routage tolérant** (*loose routing* en anglais) fournit une série d'adresses IP par lesquelles le paquet doit passer, mais il permet de prendre n'importe quelle route pour parvenir à chacune de ces adresses (correspondant à des passerelles en général).

Le **routage strict** n'autorise aucune déviation de la route spécifiée. Si cette route ne peut pas être suivie, le paquet est détruit et un message d'erreur est envoyé *via* ICMP. Le routage strict est fréquemment utilisé pour tester les routes mais rarement pour transmettre des paquets utilisateur.

– Le contenu de la **zone de remplissage** dépend des options sélectionnées. On utilise le remplissage pour s'assurer que la longueur de l'en-tête est bien un multiple de mots.

9.3 Le protocole de couche de transport UDP

Le protocole UDP (*User Datagram Protocol*), décrit dans [RFC 768], constitue un protocole de transport minimal. Il est basé sur le protocole de réseau Internet (IP) et offre pour l'essentiel la même fonctionnalité qu'IP lui-même : un service de datagrammes sans connexion et non fiable. Le protocole UDP possède cependant une fonctionnalité essentielle, celle de multiplexage/démultiplexage : il transmet les datagrammes aux différentes applications d'un système terminal donné, grâce à la spécification des numéros de port.

Le protocole UDP est utilisé d'une part pour les applications orientées transaction telles que DNS, dans lesquelles seules une requête et une réponse associée doivent être transmise, de telle sorte qu'il est inutile d'établir une connexion pour cela. D'autre part, le protocole UDP est également utilisé dans les contextes où la rapidité de la transmission des données prédomine sur la fiabilité. C'est ainsi que dans le cas des flux audio, qui sont transmis sous forme de petits paquets, il n'est pas excessivement gênant que des paquets isolés se perdent ; au contraire, un contrôle automatique de flux et des erreurs (assorti de la retransmission des paquets perdus) s'avérerait souvent très pénalisant pour la régularité du flux.

9.3.1 L'en-tête UDP

L'en-tête UDP est défini dans [RFC 768]. Sa structure, très simple, n'occupe que huit octets, comme le montre la figure suivante :

Port source (16 bits)	Port Destination (16 bits)
Longueur (16 bits)	Somme de contrôle (16 bits)
Données	Remplissage

- Le **port source** est un champ facultatif contenant le numéro de port de l'expéditeur, compris entre 1 et 65 535. Si aucun numéro de port n'est spécifié, le champ est mis à 0. Ce champ est nécessaire au destinataire s'il doit renvoyer des données.

  Dans le cas de l'implémentation Linux, le port source sera cependant nécessairement indiqué.

- Le **port de destination** est le numéro de port sur la machine de destination.

- La **longueur** est la taille du datagramme, exprimée en nombre d'octets, comprenant en-tête et données. Sa valeur minimum est 8 (taille de l'en-tête UDP) et le datagramme UDP le plus long peut transporter 65 535 - 8 = 65 527 octets de données utiles.

- la **somme de contrôle** est un champ facultatif sur lequel nous allons revenir ci-dessous. De nombreuses applications n'y ont pas recours. S'il n'est pas utilisé, sa valeur doit être zéro.

- Les **données** sont de longueur variable.

- Le **remplissage** assure que le datagramme a une longueur multiple de 16 bits.

9.3.2 Sécurisation optionnelle par somme de contrôle

UDP permet une connexion non fiable dans le sens où il n'existe aucun mécanisme permettant de reconnaître et de traiter les datagrammes perdus ou dupliqués. Les datagrammes peuvent cependant être protégés, de manière optionnelle, des erreurs à l'aide d'une somme de contrôle. Contrairement au protocole IP, cette somme de contrôle ne se rapporte pas uniquement à l'en-tête du datagramme mais également aux données utiles de celui-ci.

Si le datagramme expédié comporte une somme de contrôle, celle-ci est le complément à 1 de 16 bits du complément à 1 de la somme du datagramme (en-tête et données utiles) ainsi que d'un **pseudo en-tête**, dont la structure est montrée à la figure 9-1 ([TJ–97], p.117).

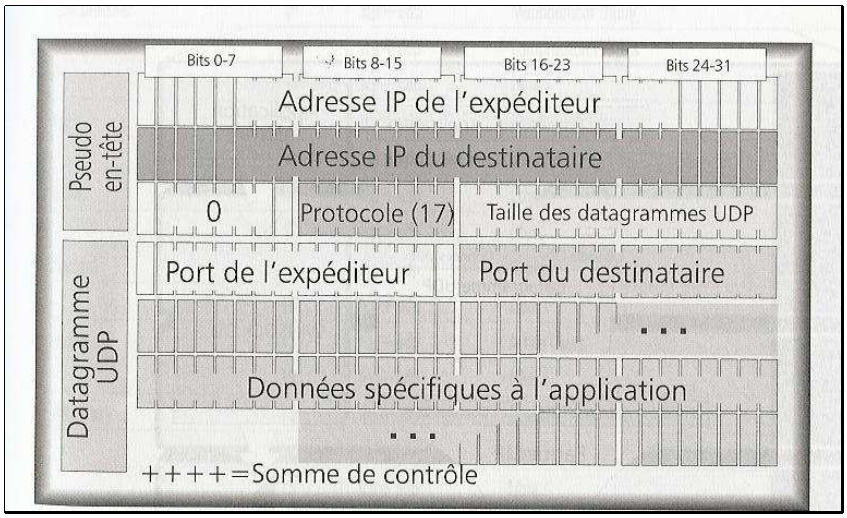


FIG. 9.3 – *Pseudo en-tête UDP*

Le pseudo en-tête contient les adresses IP de l'expéditeur et du destinataire, la taille du datagramme UDP et le code de protocole IP pour UDP (à savoir 17). Toutes ces informations proviennent de la couche réseau.

Lorsque le calcul de la somme de contrôle donne la valeur nulle, on indique `0xFFFF`, soit -1, à la place, qui équivaut également à 0 sans toutefois coïncider avec la valeur constante 0 signalant l'absence de somme de contrôle.

La méthode de calcul de la somme de contrôle s'implémente très efficacement pour tous les types de processeurs. Elle est décrite dans les recommandations [RFC 1071], [RFC 1141] et [RFC 1624].

Troisième partie

Vue générale sur l'implémentation Linux

Chapitre 10

Implémentation générale de Linux

Les systèmes d'exploitation modernes, tels que Linux, implémentent tous des un sous-système réseau. Avant de passer à l'implémentation de celui-ci en ce qui concerne Linux, disons quelques mots de l'implémentation générale de Linux.

Il serait malheureusement trop long ne serait-ce que d'en faire un résumé en un chapitre. Heureusement il existe de bons livres de ce point de vue, entre autres [BOV–01]. Nous avons la faiblesse de penser que notre livre consacré à ce sujet [CEG–03], qui commente la toute première version 0.01 de Linux, est bien adapté. La suite de ce livre suppose une connaissance de l'implémentation de Linux, acquise dans [CEG–03] ou autre.

La connaissance du noyau 0.01 est largement suffisante pour comprendre l'implémentation réseau de Linux. Cependant certains points généraux ont évolué et influent sur celle-ci. Ce sont d'eux dont nous allons parler ici et, parsemé ici et là, sous le nom de *fonction générale*.

Les quatre sous-systèmes les plus importants de Linux sont : la gestion de la mémoire (`MM` pour l'anglais *Memory Management*), la gestion des processus (devenue gestion des activités avec la diversification de nature de celles-ci), la gestion des fichiers et la gestion des périphériques. Il existe d'autres sous-systèmes tels que le sous-système réseau, objet de cet ouvrage, la gestion du temps, la gestion des processeurs sur un système avec plusieurs microprocesseurs ou la gestion des modules.

Après une vue d'ensemble de l'organisation des fichiers source de Linux, nous consacrerons une section à la définition des types portables (pour les rendre indépendants du microprocesseur utilisé), puis à la gestion de la mémoire et à la gestion du temps. La gestion des fichiers sera revue au début du chapitre 27, consacré à l'implémentation du type de fichiers socket. Nous aborderons ensuite la gestion de plusieurs microprocesseurs, puis celle des activités. Nous consacrerons une section à l'implémentation des opérations atomiques, à celle des modules et enfin à l'initialisation du système Linux.

10.1 Organisation du code source

Nous renvoyons à [CEG–03] pour une description complète de l'organisation du code source d'un des noyaux de Linux, le tout premier. Évidemment ça se complique de version en version : les sources du noyau 2.4.18, par exemple, occupent 122 Mo, celles du noyau 2.6.0 pas moins de 212 Mo.

10.1.1 Premier niveau

Linux 2.6.10

Le premier niveau de l'aborescence du noyau 2.6.10 est assez simple :

- `./Documentation` contient de la documentation, en particulier sur les périphériques pris en compte.
- `./arch` concerne tout qui dépend de l'architecture du microprocesseur, Linux ayant été adapté à plusieurs microprocesseurs. C'est dans ce répertoire qu'on trouve ce qui concerne le démarrage du système.
- `./crypto` concerne la cryptographie.
- `./drivers` contient les divers pilotes de périphériques.
- `./fs` contient ce qui concerne les différents systèmes de fichiers.
- `./include` contient les fichiers en-tête, dont beaucoup dépendent d'une architecture de microprocesseur donnée.
- `./init` ne contient qu'un seul fichier `main.c`, le fichier principal du code.
- `./ipc` contient la mise en place d'un mode de communication entre processus appelé IPC V.
- `./kernel` contient tout ce qu'on pourrait appelé le micro-noyau ou cœur du système d'exploitation.
- `./lib` contient le code des appels systèmes qui seront utilisés lors du démarrage du système.
- `./mm` (pour *Memory Management*) contient le code de gestion de la mémoire.
- `./net` concerne la mise en place du sous-système réseau.
- `./scripts` contient un certain nombre de programmes de commandes (*script* en anglais).
- `./security` contient ce qui concerne la sécurité.
- `./sound` contient ce qui concerne le son.
- `./usr` contient des fichiers auxiliaires.

Nous allons évidemment nous intéresser dans ce livre plus particulièrement au répertoire `./net` et aux parties des répertoires `./include` et `./drivers` qui concernent les réseaux.

10.1.2 Répertoire concernant les réseaux

Le répertoire `./net` contient un grand nombre de sous-répertoires. Nous aborderons les répertoires suivants :

- `./core` concerne le code commun à l'accès des périphériques réseau, au filtrage, aux sockets, etc.
- `./ipv4` contient le code concernant IP version 4.
- `./sched` contient le code concernant l'ordonnancement des paquets, qu'ils soient IP ou non.
- `./packet` contient le code des familles de protocole d'accès aux paquets bruts.
- `./netlink` contient le code pour ce protocole.

- `./ethernet` contient le code général de l'implémentation d'Ethernet, le code spécifique à chaque carte Ethernet se trouvant dans `../drivers`.
- `./802` contient le code générique du protocole de la couche de liaison spécifique à IEEE 802.
- `./unix` contient l'implémentation des sockets de type local.
- `./llc` contient l'implémentation de la sous-couche LLC.

Décrivons en quelques mots les autres répertoires, ceux qui ne nous intéresseront pas dans cet ouvrage :

- `./ipv6` contient le code concernant IP version 6.
- `./sunrpc` contient l'implémentation de l'appel de procédure distante (*Remote Procedure Call*) Sun (utile pour NFS).
- `./wanrouter` contient le support de routeur multi-protocole pour les réseaux étendus.
- `./atm` contient le code générique du protocole de la couche de liaison/réseau du mode de transfert asynchrone, incluant MPOA et les spécificités IP.
- `./ax25` contient le code du protocole de liaison X.25 des paquets pour les radio-amateurs.
- `./x25` contient le code du protocole de réseau x.25.
- `./rose` contient le code du service des opérations distantes sur x.25, non IP.
- `./lapb` contient le code du protocole de liaison *Link Access Procedure, Balanced*.
- `./netrom` contient le code du périphérique IP appelé *netrom*.
- `./bridge` contient l'implémentation des ponts suivant la spécification IEEE 802.1d en dessous du niveau IP.
- `./ipx` contient l'implémentation du protocole réseau propriétaire de Novell appelé IPX.
- `./irda` contient l'implémentation des pilotes de périphériques infra-rouges (*Infra-Red Digital Access*).
- `./decnet` contient l'implémentation de l'ancienne suite de protocoles DECNET de *Digital Equipment Corporation*.
- `./econet` contient l'implémentation de l'ancienne suite de protocoles ECONET (non IP).
- `./appletalk` contient l'implémentation de la suite de protocoles qui fut conçue pour les MAC d'Apple (non IP).
- `./key` contient l'implémentation de `PF_KEY` version 2.
- `./rxrpc` contient l'implémentation de `Rx RPC`.
- `./sctp` contient l'implémentation de `SCTP` d'IBM.
- `./xfrm` contient l'implémentation de `XFRM`.
- `./8021q` contient l'implémentation générique de la couche *VLAN* de réseau local virtuel.
- `./bluetooth` contient l'implémentation de la couche physique pour les matériels utilisant le protocole Bluetooth.

10.1.3 Structure d'un fichier source

Montrons à quoi ressemble un fichier de l'implémentation Linux à propos du fichier principal `socket.c` du répertoire `linux/net`. L'API des sockets pour Linux est due à un grand nombre d'auteurs. Ceux-ci sont cités au début de ce fichier : Code Linux 2.6.10

```
1  /*
2   * NET          Une implementation du protocole d'acces au reseau SOCKET.
3   *
4   * Version :    @(#)socket.c    1.1.93  18/02/95
```

```
5   *
6   * Auteurs :    Orest Zborowski, <obz@Kodak.COM>
7   *              Ross Biro, <bir7@leland.Stanford.Edu>
8   *              Fred N. van Kempen, <waltje@uWalt.NL.Mugnet.ORG>
9   *
10  * Rectifications :
11  *              Anonyme         :       Nettoyage NOTSOCK/BADF. Erreur rectifiee dans
12  *                                      shutdown()
13  *              Alan Cox        :       verify_area() rectifie
14  *              Alan Cox        :       DDI enleve
15  *              Jonathan Kamens :       Bogue sur reconnect SOCK_DGRAM
16  *              Alan Cox        :       Deplacement d'un grand nombre de verifications
17  *                                      au plus haut niveau.
18  *              Alan Cox        :       Deplacement des structures d'adresses vers/depuis le
19  *                                      mode utilisateur au-dessus des couches de protocole.
20  *              Rob Janssen     :       Permet d'envoyer une longueur 0.
21  *              Alan Cox        :       Support des E/S asynchrones (plaggie des
22  *                                      pilotes tty).
23  *              Niibe Yutaka    :       E/S asynchrones pour les ecritures (style 4.4BSD)
24  *              Jeff Uphoff     :       Rendu le nombre max de sockets configurable
25  *                                      en ligne de commande.
26  *              Matti Aarnio    :       Rendu le nombre de sockets dynamique,
27  *                                      a etre alloue lorsque necessaire. Le max de M.
28  *                                      Uphoff est utilise comme max
29  *                                      permis en allocation.
30  *              Linus           :       Argh. efface toutes les allocation de socket par
31  *                                      ailleurs : dans le noeud d'information maintenant.
32  *              Alan Cox        :       Rendu sock_alloc()/sock_release() public
33  *                                      pour NetROM et le type nfsd du futur
34  *                                      noyau.
35  *              Alan Cox        :       Bases de sendmsg/recvmsg.
36  *              Tom Dyas        :       Exporte les symboles du reseau.
37  *              Marcin Dalecki  :       Problemes rectifies avec CONFIG_NET="n".
38  *              Alan Cox        :       Ajoute le verrouillage des threads pour les appels
39  *                                      sys_* des sockets. Il peut y avoir des erreurs
40  *                                      en ce moment.
41  *              Kevin Buhr      :       Rectifie les erreurs betes ci-dessus.
42  *              Andi Kleen      :       Quelques petits nettoyages, optimisations
43  *                                      et rectifie un bogue dans copy_from_user().
44  *              Tigran Aivazian :       sys_send(args) appelle sys_sendto(args, NULL, 0)
45  *              Tigran Aivazian :       Rendu les verifications du nombre de clients dans
46  *                                      listen(2) independant du protocole
47  *
48  *
49  *              Ce programme est un logiciel libre ; vous pouvez le redistribuer et/ou
50  *              le modifier sous les termes de la Licence Publique Generale GNU
51  *              publiee par la Free Software Foundation ; soit la version
52  *              2 de cette licence, soit (a votre convenance) toute version ulterieure.
53  *
54  *
55  *      Ce module est effectivement l'interface de plus haut niveau pour le
56  *      paradigme des sockets BSD.
57  *
58  */
58  *      Fonde sur NET3.039 de la Swansea University Computer Society
59  */
```

Le fichier commence par un commentaire comprenant un titre et une description rapide de son objet (ligne 2) suivi de la version et de la date de celle-ci (ligne 4). On remarquera que l'implémentation de cette partie n'aurait pas changée depuis 1995. En fait les dates sont rarement à jour.

Les auteurs de la première version du fichier sont cités, suivis des auteurs des rectifications,

avec une description rapide du rôle de chacun d'eux.

Les lignes 49 à 52 sur la licence sont reprises dans la plupart des fichiers.

Dans cet ouvrage, nous traduirons les commentaires en français. On pourra retrouver la version originelle anglaise en consultant les sources.

10.1.4 Aide au parcours du code source

Dans la version 2.4.9, le noyau Linux compte 9 837 fichiers représentant un volume de 3 857 319 lignes de code source. Quant aux modifications intervenant d'une version à la suivante, elles peuvent représenter plusieurs méga-octets. L'important volume représenté par ce code source fait qu'il est très difficile de s'y retrouver et de naviguer à l'emplacement souhaité.

C'est pour simplifier la manipulation du code source du noyau Linux qu'a été développé à l'université d'Oslo le navigateur de code source **LXR** (pour *Linux cross (X) Reference*) accessible sur le site Internet *Cross-Referencing Linux*:

`http://lxr.linux.no/`

en cliquant sur "Browse the code" ou:

`http://fxr.watson.org/fxr/`

ou encore, mais le serveur est plus lent, *Linux Cross Reference*:

`http://www.iglu.org.il/lxr/`

10.2 Définition de types portables

Les systèmes d'exploitation modernes, tels que Linux, doivent être indépendant de l'architecture matérielle. Il existe une très petite part du noyau qui dépend de l'architecture. Linux définit un ensemble de types C pour utilisation dans le noyau (et ailleurs si on le désire).

10.2.1 Types Posix

Ceci commence par la définition des types POSIX pour lesquels on n'a pas besoin de connaître l'architecture, dans le fichier `linux/include/linux/posix_types.h` que nous reproduisons intégralement:

Code Linux 2.6.10

```
 1 #ifndef _LINUX_POSIX_TYPES_H
 2 #define _LINUX_POSIX_TYPES_H
 3
 4 #include <linux/stddef.h>
 5
 6 /*
 7  * On permet 1 024 descripteurs de fichier : si NR_OPEN est un jour augmente
 8  * au-dela, vous aurez a changer ceci egalement. Mais 1024 fd semble suffisant
 9  * pour des unix "reels" comme OSF/1, aussi peut-on esperer que cette
10  * limite n'aura pas a etre changee [une fois de plus].
11  *
12  * Noter que POSIX veut que FD_CLEAR(fd,fdsetp) soit defini dans
13  * <sys/time.h> (en ainsi dans <linux/time.h>) - mais ici est une place plus
14  * logique. Resolu en ayant une definition muette dans <sys/time.h>.
15  */
16
17 /*
18  * Ces macros peuvent avoir ete definies dans <gnu/types.h>. Mais nous
19  * utiliserons toujours celles qui sont ici.
20  */
```

```
21 #undef __NFDBITS
22 #define __NFDBITS       (8 * sizeof(unsigned long))
23
24 #undef __FD_SETSIZE
25 #define __FD_SETSIZE    1024
26
27 #undef __FDSET_LONGS
28 #define __FDSET_LONGS   (__FD_SETSIZE/__NFDBITS)
29
30 #undef __FDELT
31 #define __FDELT(d)      ((d) / __NFDBITS)
32
33 #undef __FDMASK
34 #define __FDMASK(d)     (1UL << ((d) % __NFDBITS))
35
36 typedef struct {
37         unsigned long fds_bits [__FDSET_LONGS];
38 } __kernel_fd_set;
39
40 /* Type d'une routine de signal.  */
41 typedef void (*__kernel_sighandler_t)(int);
42
43 /* Type d'une cle SYSV IPC.  */
44 typedef int __kernel_key_t;
45 typedef int __kernel_mqd_t;
46
47 #include <asm/posix_types.h>
48
49 #endif /* _LINUX_POSIX_TYPES_H */
```

Les types pour lesquels la définition repose sur l'architecture sont définis dans le répertoire `asm` adéquat. Par exemple pour les microprocesseurs Intel, on trouve dans le fichier `linux/include/asm-i386/posix_types.h`:

Code Linux 2.6.10

```
1  #ifndef __ARCH_I386_POSIX_TYPES_H
2  #define __ARCH_I386_POSIX_TYPES_H
3
4  /*
5   * Ce fichier est en general utilise par les logiciels au niveau utilisateur, aussi devez-vous
6   * etre un petit peu soigneux quant a la pollution de l'espace des noms etc. De plus, vous ne
7   * pouvez pas supposer que GCC l'utilise.
8   */
9
10 typedef unsigned long   __kernel_ino_t;
11 typedef unsigned short  __kernel_mode_t;
12 typedef unsigned short  __kernel_nlink_t;
13 typedef long            __kernel_off_t;
14 typedef int             __kernel_pid_t;
15 typedef unsigned short  __kernel_ipc_pid_t;
16 typedef unsigned short  __kernel_uid_t;
17 typedef unsigned short  __kernel_gid_t;
18 typedef unsigned int    __kernel_size_t;
19 typedef int             __kernel_ssize_t;
20 typedef int             __kernel_ptrdiff_t;
21 typedef long            __kernel_time_t;
22 typedef long            __kernel_suseconds_t;
23 typedef long            __kernel_clock_t;
24 typedef int             __kernel_timer_t;
25 typedef int             __kernel_clockid_t;
26 typedef int             __kernel_daddr_t;
27 typedef char *          __kernel_caddr_t;
28 typedef unsigned short  __kernel_uid16_t;
29 typedef unsigned short  __kernel_gid16_t;
```

```
30 typedef unsigned int    __kernel_uid32_t;
31 typedef unsigned int    __kernel_gid32_t;
32
33 typedef unsigned short  __kernel_old_uid_t;
34 typedef unsigned short  __kernel_old_gid_t;
35 typedef unsigned short  __kernel_old_dev_t;
36
37 #ifdef __GNUC__
38 typedef long long       __kernel_loff_t;
39 #endif
```

10.2.2 Types entiers

Linux utilise les types entiers suivants :

Type	Signification	Variantes
s8	entier relatif sur 8 bits	__s8, int8_t
u8	entier naturel sur 8 bits	__u8, u_int8_t, uint8_t, u_char, unchar
s16	entier relatif sur 16 bits	__s16, int16_t
u16	entier naturel sur 16 bits	__u16, u_int16_t, uint16_t, u_short, ushort
s32	entier relatif sur 32 bits	__s32, int32_t
u32	entier naturel sur 32 bits	__u32, u_int32_t, uint32_t, u_int, uint, u_long, ulong
s64	entier relatif sur 64 bits	__s64, int64_t
u64	entier naturel sur 64 bits	__u64, uint64_t, u_int64_t

avec 's' pour *signed* et 'u' pour *unsigned*.

La définition de base de ces types dépend de l'architecture. Pour les microprocesseurs Intel, par exemple, elle se trouve dans le fichier `linux/include/asm-i386/types.h`:

Code Linux 2.6.10

```
8  /*
9   * __xx est ok : il ne pollue pas l'espace des noms POSIX. Utiliser ceci dans les
10  * fichiers en-tete exportes vers l'espace utilisateur
11  */
12
13 typedef __signed__ char __s8;
14 typedef unsigned char __u8;
15
16 typedef __signed__ short __s16;
17 typedef unsigned short __u16;
18
19 typedef __signed__ int __s32;
20 typedef unsigned int __u32;
21
22 #if defined(__GNUC__) && !defined(__STRICT_ANSI__)
23 typedef __signed__ long long __s64;
24 typedef unsigned long long __u64;
25 #endif
26
27 #endif /* __ASSEMBLY__ */
28
29 /*
30  * Ceux-ci ne sont pas exportes en-dehors du noyau pour eviter des problemes avec l'espace
      des noms
31  */
32 #ifdef __KERNEL__
33
34 #define BITS_PER_LONG 32
35
```

```
36 #ifndef __ASSEMBLY__
37
38 #include <linux/config.h>
39
40 typedef signed char s8;
41 typedef unsigned char u8;
42
43 typedef signed short s16;
44 typedef unsigned short u16;
45
46 typedef signed int s32;
47 typedef unsigned int u32;
48
49 typedef signed long long s64;
50 typedef unsigned long long u64;
```

Code Linux 2.6.10

La définition des variantes se trouve dans le fichier `linux/include/linux/types.h`:

```
  1 #ifndef _LINUX_TYPES_H
  2 #define _LINUX_TYPES_H
  3
  4 #ifdef  __KERNEL__
  5 #include <linux/config.h>
[...]
 13 #include <linux/posix_types.h>
 14 #include <asm/types.h>
 15
 16 #ifndef __KERNEL_STRICT_NAMES
[...]
 92 /* bsd */
 93 typedef unsigned char           u_char;
 94 typedef unsigned short          u_short;
 95 typedef unsigned int            u_int;
 96 typedef unsigned long           u_long;
 97
 98 /* sysv */
 99 typedef unsigned char           unchar;
100 typedef unsigned short          ushort;
101 typedef unsigned int            uint;
102 typedef unsigned long           ulong;
103
104 #ifndef __BIT_TYPES_DEFINED__
105 #define __BIT_TYPES_DEFINED__
106
107 typedef         __u8            u_int8_t;
108 typedef         __s8            int8_t;
109 typedef         __u16           u_int16_t;
110 typedef         __s16           int16_t;
111 typedef         __u32           u_int32_t;
112 typedef         __s32           int32_t;
113
114 #endif /* !(__BIT_TYPES_DEFINED__) */
115
116 typedef         __u8            uint8_t;
117 typedef         __u16           uint16_t;
118 typedef         __u32           uint32_t;
119
120 #if defined(__GNUC__) && !defined(__STRICT_ANSI__)
121 typedef         __u64           uint64_t;
122 typedef         __u64           u_int64_t;
123 typedef         __s64           int64_t;
124 #endif
```

10.2.3 Tailles

Les types liés à des tailles sont définis dans le fichier `linux/include/linux/types.h`: Code Linux 2.6.10

```
54 #if defined(__GNUC__) && !defined(__STRICT_ANSI__)
55 typedef __kernel_loff_t         loff_t;
56 #endif
57
58 /*
59  * Les typedef suivants sont egalement proteges par des ifdef individuels pour
60  * des raisons historiques :
61  */
62 #ifndef _SIZE_T
63 #define _SIZE_T
64 typedef __kernel_size_t         size_t;
65 #endif
66
67 #ifndef _SSIZE_T
68 #define _SSIZE_T
69 typedef __kernel_ssize_t        ssize_t;
70 #endif
```

Il s'agit de la longueur d'un décalage et des types entiers utilisés pour exprimer les tailles.

10.3 Gestion de la mémoire

Rappelons ce qui est nécessaire pour la gestion de la mémoire (uniquement du point de vue du programmeur système) : réservation et libération de mémoire de l'espace noyau, copie entre l'espace d'adressage du noyau et celui de l'utilisateur et enfin les caches mémoire.

10.3.1 Réservation et libération de mémoire noyau

10.3.1.1 Réservation

La fonction :

`kmalloc(size, priority)`

essaie de réserver une zone de mémoire connexe de `size` octets dans l'espace mémoire du noyau. En fait quelques octets supplémentaires sont réservés à cause des caches mémoire dont nous reparlerons ci-dessous.

Le paramètre `priority` permet d'indiquer des options. Le préfixe `GFP_` indique qu'il faut éventuellement utiliser la fonction `get_free_pages()` en vue de réservation de mémoire :

- `GFP_USER` pour allouer l'emplacement mémoire dans l'espace utilisateur.
- `GFP_KERNEL` pour allouer l'emplacement mémoire dans l'espace noyau. L'activité qui en fait la demande peut être interrompue au cours de la réservation.
- `GFP_ATOMIC` est l'analogue de `GFP_KERNEL` mais tout l'emplacement mémoire doit être attribué de façon atomique (en une seule fois).
- `GFP_DMA` pour indiquer que l'emplacement mémoire doit être adéquat pour un accès direct en mémoire.

Ces valeurs générales sont commentées dans le fichier `linux/mm/slab.c`: Code Linux 2.6.10

```
2406 /**
2407  * kmalloc - alloue de la memoire
2408  * @size : le nombre d'octets requis.
2409  * @flags: le type de memoire a allouer.
```

```
2410  *
2411  * kmalloc est la methode normale d'allocation de la memoire
2412  * du noyau.
2413  *
2414  * L'argument @flags est l'un des suivants :
2415  *
2416  * %GFP_USER - Alloue de la memoire pour l'utilisateur. Peut s'assoupir.
2417  *
2418  * %GFP_KERNEL - Alloue de la ram noyau normale. Peut s'assoupir.
2419  *
2420  * %GFP_ATOMIC - L'allocation ne s'assoupira pas. A utiliser pour les routines
                    d'interruption.
2421  *
2422  * De plus, le drapeau %GFP_DMA peut etre positionne pour indiquer que le memoire
2423  * doit etre adequate pour la DMA. Ceci peut signifier des choses differentes suivant
2424  * la plateforme. Par exemple, sur i386, ceci signifie que la memoire doit provenir
2425  * des 16 premiers Mo.
2426  */
```

La valeur de renvoi de `malloc()` est un pointeur sur la zone de mémoire réservée avec succès ou `NULL` s'il n'y a pas de mémoire disponible.

10.3.1.2 Libération

La fonction :

`kfree(objp)`

libère la zone de mémoire réservée à l'adresse `objp`. Cette zone de mémoire doit avoir été réservée au préalable à l'aide de `kmalloc()`.

10.3.2 Copie entre espace noyau et espace utilisateur

Les fonctions suivantes sont utilisables pour échanger des données entre l'espace d'adressage noyau et l'espace d'adressage utilisateur :

- `copy_from_user(to, from, count)` copie `count` octets de la partie mémoire de l'espace utilisateur d'adresse `from` vers la partie mémoire de l'espace noyau d'adresse `to`.
- `copy_to_user(to, from, count)` copie `count` octets de la partie mémoire de l'espace noyau d'adresse `from` vers la partie mémoire de l'espace utilisateur d'adresse `to`.

10.3.3 Caches mémoire

10.3.3.1 Notion

La réservation de zones de mémoire à l'aide de l'appel système `kmalloc()` peut durer un certain temps. Si on prévoit qu'une zone mémoire de même taille sera nécessaire dans un très proche avenir, on a intérêt à ne pas libérer la zone mémoir avec l'appel système `kfree()` après utilisation, mais à enregistrer le fait dans une liste puis à la réutiliser à la demande.

Ce mode d'action est implémenté sous Linux par les **caches par plaque** (*slab* en anglais).

10.3.3.2 Création d'un cache par plaque

La fonction :

`kmem_cache_create(name, size, offset, flags, ctor, dstor)`

crée un cache par plaque pour les zones de mémoire de `size` octets. Le paramètre `name` pointe sur une chaîne de caractères contenant le nom du cache. Le paramètre `offset` permet d'indiquer

le décalage de la première zone de mémoire dans une page de mémoire (la plupart du temps de valeur nulle). Le paramètre `flags` sert à préciser ce qu'on souhaite pour la réservation des zones de mémoire, par exemple:

- `SLAB_HWCACHE_ALIGN` our s'aligner sur la taille du cache du premier niveau du processeur;
- `SLAB_NO_REAP` pour emêcher la réduction du cache lorsque le noyau a besoin de mémoire;
- `SLAB_CACHE_DMA` pour que les zones de mémoire réservée se trouvent dans la zone compatible DMA.

Les paramètres `ctor` et `dtor` permettent d'indiquer un constructeur et un destructeur pour la zone de mémoire. Ils servent à l'initialisation et au déblaiement des zones de mémoire réservées.

La valeur de renvoi est un pointeur sur la structure de cache de plaque, de type `kmem_cache_t`.

10.3.3.3 Allocation

La fonction:

`kmem_cache_alloc(cachep, flags)`

permet de demander une zone de mémoire à partir du cache par plaque `cachep`. S'il s'en trouve un de disponible, un pointeur est immédiatement renvoyé à l'appelant. Lorsque le cache est plein, `kmalloc()` réserve une nouvelle zone de mémoire; les valeurs `GFP_` sont alors utilisées à titre de drapeaux.

10.3.3.4 Libération

La fonction:

`kmem_cache_free(kmem_cache_t *cachep, void *objp)`

renvoie au cache `cachep` la zone de mémoire commençant à l'adresse `objp`, réservée au préalable avec `kmem_cache_alloc()`.

10.4 La gestion du temps dans le noyau Linux

10.4.1 Durée

Une durée est exprimée soit en *jiffies*, soit en seconde et microseconde à travers la structure `struct timeval`. Cette dernière est définie dans le fichier `linux/include/linux/time.h`: Code Linux 2.6.10

```
18 struct timeval {
19         time_t          tv_sec;         /* secondes */
20         suseconds_t     tv_usec;        /* microsecondes */
21 };
```

Les types `time_t` et `suseconds_t` sont définis dans le fichier `linux/include/linux/types.h`: Code Linux 2.6.10

```
29 typedef __kernel_suseconds_t    suseconds_t;
[...]
77 #ifndef _TIME_T
78 #define _TIME_T
79 typedef __kernel_time_t         time_t;
80 #endif
```

qui renvoie, pour les microprocesseurs Intel, au fichier `linux/include/asm-i386/posix_types.h`: Code Linux 2.6.10

```
21 typedef long           __kernel_time_t;
22 typedef long           __kernel_suseconds_t;
```

10.4.2 File d'attente de minuteurs

Les ordinateurs ne possèdent en général que d'un seul minuteur alors qu'on en a besoin de plusieurs en programmation système. On résout de problème en utilisant une **file d'attente de minuteurs** : lorsqu'une interruption d'horloge se produit, la routine de gestion de celle-ci ne se contente pas d'actualiser la variable `jiffies` ; elle vérifie aussi la file d'attente de minuteurs à la recherche d'éventuelles routines de traitement.

La file d'attente de minuteurs est constituée de structures `timer_list` représentant une fonction (routine de traitement) devant être exécutée à un certain moment (`expires`). Ce type est défini dans le fichier `linux/include/linux/timer.h` :

Code Linux 2.6.10

```
11 struct timer_list {
12         struct list_head entry;
13         unsigned long expires;
14
15         spinlock_t lock;
16         unsigned long magic;
17
18         void (*function)(unsigned long);
19         unsigned long data;
20
21         struct tvec_t_base_s *base;
22 };
```

Voici les fonctions disponibles pour la gestion de la file d'attente des minuteurs :

- `add_timer()` insère une structure `timer_list` dans la file d'attente.
- `del_timer()` supprime une structure `timer_list` de la file d'attente.
- `init_timer()` initialise une structure `timer_list`. Cette fonction doit toujours être appelée lorsqu'une structure `timer_list` a été générée. Elle est définie dans le fichier `linux/include/linux/timer.h` :

Code Linux 2.6.10

```
35 /***
36  * init_timer - initialise un minuteur.
37  * @timer : le minuteur a initialiser
38  *
39  * init_timer() doit etre appliquee a un minuteur avant d'appeler *toutes* les
40  * autres fonctions de minutage.
41  */
42 static inline void init_timer(struct timer_list * timer)
43 {
44         timer->base = NULL;
45         timer->magic = TIMER_MAGIC;
46         spin_lock_init(&timer->lock);
47 }
```

10.5 SMP

Linux est l'un des rares systèmes d'exploitation à prendre en charge plusieurs processeurs, depuis la version 2.0 du noyau pour les processeurs Intel et Sparc, fonction améliorée dans les versions ultérieures. Il s'agit du **SMP** (pour l'anglais *Symetric Multi-Processing*).

Cependant nous ne commenterons pas le code qui dépend de SMP dans cet ouvrage.

10.6 Gestion des activités dans le noyau Linux

Linux est un système multitâche. Pendant plusieurs versions du noyau Linux, les seules **activités** possibles étaient les processus, les appels systèmes et les interruptions matérielles. De nos

jours, on a un plus grand choix :

- processus ;
- appels systèmes ;
- interruptions matérielles ;
- interruptions logicielles ;
- tasklets ;
- parties basses.

10.6.1 Activités de base

Les trois types d'activités de base existent depuis le tout premier noyau Linux et sont décrits, par exemple, dans [CEG–03]. Nous n'insisterons pas ici.

10.6.1.1 Processus

Un **processus** est une activité démarrée dans le but d'exécuter une instance d'une application donnée. Un processus est décrit par une entité du type `struct task_struct`.

10.6.1.2 Appels système

Les processus n'agissent que dans l'espace d'adressage utilisateur. Si un processus veut accéder aux pilotes ou utiliser une fonctionnalité du noyau du système d'exploitation, il doit avoir recours à un **appel système**. Tout n'est pas permis mais seulement ce qui a été prévu par le concepteur du noyau.

10.6.1.3 Interruptions matérielles

Les périphériques informent un système d'exploitation de la survenue d'un événement à l'aide d'une **interruption matérielle** (**HW-IRQ** pour *HardWare Interrupt ReQuest*). Ceci provoque l'interruption de l'activité en cours sur un des microprocesseurs et l'exécution d'une **routine de traitement d'interruption**.

La routine de traitement d'une interruption peut être décrite statiquement dans le noyau ou enregistrée au moment de l'exécution à l'aide de la fonction `request_irq()`. Dans ce dernier cas, elle peut être retirée grâce à la fonction `free_irq()`.

La fonction générale `request_irq()` est commentée dans le fichier `linux/kernel/irq/manage.c` :

Code Linux 2.6.10

```
279 /**
280  *      request_irq - alloue une ligne d'interruption
281  *      @irq : Ligne d'interruption a allouer
282  *      @handler : Fonction a appeler lorsque l'IRQ survient
283  *      @irqflags : Drapeaux du type d'interruption
284  *      @devname : Un nom ascii pour le peripherique reclamant
285  *      @dev_id : Un cafteur passe a la routine de gestion
286  *
287  *      Cet appel alloue des ressources d'interruption et active la
288  *      ligne d'interruption et le traitement de l'IRQ. Pour ce dernier point,
289  *      cet appel donne la routine de gestion qui doit etre invoquee. Puisque
290  *      votre routine de gestion peut effacer toute interruption que la carte
291  *      a leve, vous devez prendre soin a la fois d'initialiser votre materiel
292  *      et de positionner la routine d'interruption dans le bon ordre.
293  *
294  *      Dev_id doit etre globalement unique. Normalement l'adresse de la
```

```
295  *      structure de donnees du peripherique est utilisee comme cafteur. Puisque la
296  *      routine recoit cette valeur, il n'est pas bete de l'utiliser.
297  *
298  *      Si votre interruption est partagee, vous devez passer un dev_id non NULL
299  *      comme ceci est exige lorsqu'on libere l'interruption.
300  *
301  *      Drapeaux :
302  *
303  *      SA_SHIRQ                L'interruption est partagee
304  *      SA_INTERRUPT            Inhibe les interruptions locales lors du traitement
305  *      SA_SAMPLE_RANDOM        L'interruption peut etre utilisee pour l'entropie
306  *
307  */
308 int request_irq(unsigned int irq,
309                 irqreturn_t (*handler)(int, void *, struct pt_regs *),
310                 unsigned long irqflags, const char * devname, void *dev_id)
```

Code Linux 2.6.10

La fonction générale `free_irq()` est commentée dans le même fichier :

```
218 /**
219  *      free_irq - libere une interruption
220  *      @irq : ligne d'interruption a liberer
221  *      @dev_id : identite du peripherique a liberer
222  *
223  *      Retire un gestionnaire d'interruption. Le gestionnaire est retire et si
224  *      la ligne d'interruption n'est plus utilisee par aucun peripherique, elle es inhibee.
225  *      Pour une IRQ partagee, l'appelant doit s'assurer que l'interruption est inhibee
226  *      sur la carte qu'il pilote avant d'appeler cette fonction. La fonction
227  *      ne renvoie rien jusqu'a ce que toutes les interruptions en cours pour cette IRQ
228  *      soient traitees.
229  *
230  *      Cette fonction ne peut pas etre appelee dans un contexte d'interruption.
231  */
232 void free_irq(unsigned int irq, void *dev_id)
```

Le noyau Linux distingue deux types d'interruptions matérielles :

- Les **interruptions rapides** comportent une routine de traitement d'interruption très courtes et interrompent l'activité en cours très rapidement. Toutes les autres interruptions sont verrouillées au cours de leur exécution. La routine ne doit pas être interrompue.

  Les interruptions rapides sont identifiées lors de l'enregistrement par `request_irq()` à l'aide du drapeau `SA_INTERRUPT`.
- Les **interruptions lentes** peuvent être interrompues au cours de leur exécution par d'autres interruptions. Elles comportent souvent une routine de traitement d'interruption plus longue. Dans ce cas on distingue une partie haute et une partie basse.

10.6.2 Parties basses et tasklets

10.6.2.1 Parties hautes et parties basses

Les routines de traitement d'interruptions doivent s'exécuter dès que possible après le déclenchement de l'interruption et n'interrompre l'activité en cours que le plus brièvement possible. Toutefois, il est impossible d'exécuter certaines tâches avec peu d'instructions. Ceci a conduit à diviser certaines routines en deux parties :

- Dans la **partie haute** (*top half* en anglais), seules les tâches les plus importantes sont réalisées après déclenchement d'une interruption. La partie haute correspond à la routine d'interruption proprement dite.

– Dans la **partie basse** (*bottom half* en anglais, abrégé en **BH**) s'exécutent toutes les opérations pour lesquelles le temps n'est pas la préoccupation majeure et qu'il n'est pas nécessaire de réaliser dans la routine de traitement d'interruption proprement dite. La partie basse correspond à une activité qui sera prise en compte par l'ordonnanceur des tâches.

Dans les noyaux antérieurs à la version 2.4, les parties basses constituaient la forme d'activité principale au sein du noyau. La `NET_BH` était par exemple responsable du traitement du sous-système réseau et de l'envoi des paquets. Les parties basses ont ensuite été remplacées par les tasklets.

10.6.2.2 Tasklets

Les **tasklets**, introduites pour remplacer les anciennes parties basses, possèdent les propriétés suivantes :

– Elles possèdent une instantiation du type `struct tasklet_struct`.
– Elles sont mises en activité grâce à la fonction `tasklet_schedule(&tasklet_struct)`.
– Il est garanti qu'une tasklet ne peut s'exécuter à un instant donné que sur un seul microprocesseur.
– Différentes tasklets peuvent s'exécuter simultanément sur plusieurs microprocesseurs.

10.6.3 Interruptions logicielles

Les interruptions logicielles ne sont pas des interruptions à proprement parler mais des activités dont l'exécution est démarrée à l'aide de la fonction `do_softirq()` lorsque l'ordonnanceur des tâches lui passe la main.

On peut définir au maximum 32 interruptions logicielles dans le noyau Linux. Il n'en existe que six dans le noyau 2.6 dont `NET_RX_SOFTIRQ` (pour *NET ReCeive SOFT IRQ*), `NET_TX_SOFTIRQ` (pour *NET TransmiT SOFT IRQ*), qui nous intéressent tout particulièrement dans ce livre, et `TASKLET_SOFTIRQ` utilisée en vue de l'implémentation des tasklets décrites ci-dessus, comme le montre le fichier `linux/include/linux/interrupt.h`:

Code Linux 2.6.10

```
79 /* Evitez, S'IL VOUS PLAIT, d'ajouter de nouvelles interruptions logicielles, si vous
80    n'avez pas _reellement_ besoin d'ordonnancer des travaux a traiter a haute
      frequence. Pour presque tous les propos, les tasklets sont plus que
81    suffisantes. Par exemple tous les BH des peripheriques serie et analogues
82    devraient etre converties en tasklets, pas en interruptions logicielles.
83  */
84
85 enum
86 {
87         HI_SOFTIRQ=0,
88         TIMER_SOFTIRQ,
89         NET_TX_SOFTIRQ,
90         NET_RX_SOFTIRQ,
91         SCSI_SOFTIRQ,
92         TASKLET_SOFTIRQ
93 };
```

Voici les propriétés les plus importantes des interruptions logicielles :

– Une interruption logicielle peut s'exécuter simultanément sur plusieurs microprocesseurs. Par conséquent lorsqu'il existe des zones critiques dans une interruption logicielle, elles doivent être protégées par un verrou, ainsi que toutes les variables globales auxquelles elle accède.

- Une interruption logicielle ne peut pas s'interrompre au cours de son exécution sur un microprocesseur.
- Une interruption logicielle ne peut, au cours de son traitement sur un microprocesseur, être interrompue que par une interruption matérielle.

10.7 Opérations atomiques

Plusieurs formes d'activités différentes, capables de s'interrompre mutuellement, peuvent opérer en pseudo-parallélisme. Sur les systèmes multiprocesseurs, différentes activités opèrent même en vrai parallèlisme. Tant que les activités opèrent séparément dans le noyau Linux, il n'y a aucun problème. Toutefois, dès que plusieurs activités accèdent aux mêmes structures de données, des effets indésirables sont susceptibles de se produire.

On peut, par exemple, commencer par tester un champ avec un processus A. Le processus B change la valeur de celui-ci. Le processus A agit alors sur la structure de données avec une mauvaise information.

Il est extrêmement important, pour la stabilité du système, de s'assurer que ces opérations parallèles se déroulent sans effet secondaire indésirable. Pour éviter le genre de problèmes cité ci-dessus lors de l'opération de plusieurs activités sur une structure de données communes, appelée **zone critique**, ces activités doivent avoir lieu de manière **atomique**, c'est-à-dire qu'une opération comportant plusieurs étapes est exécutée de manière indivisible. Au cours d'une opération atomique, aucune autre instance ne peut agir sur la structure de données.

La plupart des microprocesseurs prévoient l'atomicité des opérations sur les bits et les entiers, encore faut-il les encapsuler dans des macros et fonctions de Linux. Pour compléter ceci, Linux utilise des verrous rotatifs et des sémaphores.

10.7.1 Opérations atomiques sur les bits

Les opérations atomiques sur les bits constituent la base des concepts de verrous rotatifs et des sémaphores. Par conséquent, les instructions atomiques "Test and Set" sont nécessaires. Ces opérations portent les noms suivants dans Linux :

- `set_bit(nr, void *addr)` positionne à 1 le bit numéro `nr` de l'emplacement mémoire d'adresse `addr`.
- `clear_bit(nr, void *addr)` positionne à 0 le bit numéro `nr` de l'emplacement mémoire d'adresse `addr`.
- `change_bit(nr, void *addr)` change la valeur du bit numéro `nr` de l'emplacement mémoire d'adresse `addr`.
- `test_bit(nr, void *addr)` renvoie la valeur du bit numéro `nr` de l'emplacement mémoire d'adresse `addr`.
- `test_and_set_bit(nr, void *addr)` positionne à 1 le bit numéro `nr` de l'emplacement mémoire d'adresse `addr`. La valeur précédente de ce bit est renvoyée.
- `test_and_clear_bit(nr, void *addr)` positionne à 0 le bit numéro `nr` de l'emplacement mémoire d'adresse `addr`. La valeur précédente de ce bit est renvoyée.
- `test_and_change_bit(nr, void *addr)` change la valeur du bit numéro `nr` de l'emplacement mémoire d'adresse `addr`. La valeur précédente de ce bit est renvoyée.

Ces fonctions sont définies, dans le cas des microprocesseurs Intel, dans le fichier `linux/include/asm-386/bitops.h`:

Code Linux 2.6.10

```
27 /**
28  * set_bit - Positionne un bit en memoire de facon atomique
29  * @nr : le bit a positionner
30  * @addr : l'adresse a partir de laquelle le decompter
31  *
32  * Cette fonction est atomique et ne peut pas etre rearrangee. Voir __set_bit()
33  * si vous n'avez pas besoin d'une garantie d'atomicite.
34  *
35  * Note : il n'y a aucune garantie que cette fonction ne sera pas rearrangee
36  * sur les architectures non x86, donc si vous etes en train d'ecrire du code portable,
37  * assurez-vous de ne pas vous fier a cette garantie.
38  *
39  * Noter que @nr peut etre presque arbitrairement aussi grand que l'on veut ; cette fonction
40  * n'est pas restreinte a agir sur un mot.
41  */
42 static inline void set_bit(int nr, volatile unsigned long * addr)
43 {
44         __asm__ __volatile__( LOCK_PREFIX
45                 "btsl %1,%0"
46                 :"=m" (ADDR)
47                 :"Ir" (nr));
48 }
[...]
130 /**
131  * test_and_set_bit - Positionne un bit et renvoie son ancienne valeur
132  * @nr : Bit a positionner
133  * @addr : Adresse a partir de laquelle decompter
134  *
135  * Cette operation est atomique et ne peut pas etre rearrangee.
136  * Elle peut etre rearrangee sur des architectures autres que x86.
137  * Elle implique egalement une barriere de memoire.
138  */
139 static inline int test_and_set_bit(int nr, volatile unsigned long * addr)
140 {
141         int oldbit;
142
143         __asm__ __volatile__( LOCK_PREFIX
144                 "btsl %2,%1\n\tsbbl %0,%0"
145                 :"=r" (oldbit),"=m" (ADDR)
146                 :"Ir" (nr) : "memory");
147         return oldbit;
148 }
```

10.7.2 Opérations atomiques sur les entiers

Pour exécuter des opérations atomiques sur les entiers, il faut utiliser le type `atomic_t`. Ce type dépend de l'architecture; il est défini dans le fichier `linux/include/asm-i386/atomic.h` pour les microprocesseurs Intel:

Code Linux 2.6.10

```
 8 /*
 9  * Operations atomiques que le C ne peut pas nous garantir. Utile pour
10  * le denombrement de ressources etc.
11 */
[...]
19 /*
20  * Assurez-vous que gcc n'essaie pas d'etre intelligent en deplacant des choses autour
21  * de nous. Nous avons besoin d'utiliser _exactement_ l'adresse que l'utilisateur
22  * nous a donnee et non quelque alias qui contient les memes informations.
23  */
```

```
24 typedef struct { volatile int counter; } atomic_t;
```

On dispose des opérations suivantes :

- `ATOMIC_INIT()` permet d'initialiser un entier lors de sa déclaration.
- `atomic_set(atomic_t *var, int i)` affecte de façon atomique la valeur `i` à l'entier d'adresse `var`.
- `atomic_read(atomic_t *var)` renvoie de façon atomique la valeur de l'entier d'adresse `var`.
- `atomic_add(int i, atomic_t *var)` et `atomic_sub(int i, atomic_t *var)` s'utilisent pour l'addition et la soustraction atomiques.
- `atomic_inc(atomic_t *var)` et `atomic_dec(atomic_t *var)` s'utilisent pour l'incrémentation et la décrémentation atomiques.
- `atomic_...and_test()` sont les opérations analogues à celles sur les bits.

Ces macros dépendent de l'architecture. Pour les microprocesseurs Intel, elles sont définies dans le fichier en-tête `linux/include/asm-i386/atomic.h`:

Code Linux 2.6.10

```
26 #define ATOMIC_INIT(i)  { (i) }
27
28 /**
29  * atomic_read - lit une variable atomique
30  * @v : pointeur du type atomic_t
31  *
32  * Lit atomiquement la valeur de @v.
33  */
34 #define atomic_read(v)          ((v)->counter)
35
36 /**
37  * atomic_set - positionne une variable atomique
38  * @v : pointeur du type atomic_t
39  * @i : valeur requise
40  *
41  * Positionne atomiquement la valeur de @v a @i.
42  */
43 #define atomic_set(v,i)         (((v)->counter) = (i))
44
45 /**
46  * atomic_add - ajoute un entier a une variable atomique
47  * @i : valeur entiere a ajouter
48  * @v : pointeur du type atomic_t
49  *
50  * Ajoute atomiquement @i a @v.
51  */
52 static __inline__ void atomic_add(int i, atomic_t *v)
53 {
54         __asm__ __volatile__(
55                 LOCK "addl %1,%0"
56                 :"=m" (v->counter)
57                 :"ir" (i), "m" (v->counter));
58 }
```

10.7.3 Verrous rotatifs

10.7.3.1 Notion

Si une portion critique de programme commence et si une variable de verrouillage, appelée **verrou rotatif** (*busy wait lock* en anglais), est déjà positionnée (par un autre processus), le processus attend que celle-ci se libère en la vérifiant dans une boucle sans fin (scrutation).

On gaspille ainsi du temps machine car le processus vérifie la variable de verrouillage continuellement. Il peut toutefois être plus efficace, sur un système multiprocesseur, d'effectuer une courte vérification au lieu d'appeler l'ordonnanceur. Il faut évidemment appliquer les verrous rotatifs pour de petites portions de programme.

10.7.3.2 Déclaration et initialisation

Sous Linux, les variables de verrouillage sont du type `spinlock_t`. Celui-ci est défini dans le fichier `linux/include/linux/spinlock.h`:

Code Linux 2.6.10

```
89 #define SPINLOCK_MAGIC  0x1D244B3C
90 typedef struct {
91         unsigned long magic;
92         volatile unsigned long lock;
93         volatile unsigned int babble;
94         const char *module;
95         char *owner;
96         int oline;
97 } spinlock_t;
```

L'utilisation d'un verrou rotatif exige la création et l'initialisation d'une variable du type précédent de l'une des deux façons:

```
#include <linux/spinlock.h>
spinlock_t s = SPIN_LOCK_UNLOCKED;
```

ou:

```
#include <linux/spinlock.h>
spinlock_t s;
spinlock_init(&s);
```

La constante et la macro ci-dessus sont définis dans le fichier `linux/include/linux/spinlock.h`:

Code Linux 2.6.10

```
98  #define SPIN_LOCK_UNLOCKED (spinlock_t) { SPINLOCK_MAGIC, 0, 10, __FILE__ , NULL, 0}
99
100 #define spin_lock_init(x) \
101         do { \
102                 (x)->magic = SPINLOCK_MAGIC; \
103                 (x)->lock = 0; \
104                 (x)->babble = 5; \
105                 (x)->module = __FILE__; \
106                 (x)->owner = NULL; \
107                 (x)->oline = 0; \
108         } while (0)
```

10.7.3.3 Gestion des verrous rotatifs

On peut demander un verrou rotatif au moyen des macros suivantes:

- `spin_lock(spinlock_t *s)` essaie d'occuper le verrou rotatif `s`. S'il n'est pas libre, il est mis en attente ou testé jusqu'à ce qu'il le soit. Le verrou rotatif est occupé dès qu'il se libère.
- `spin_lock_irqsave(spinlock_t *s, unsigned long flags)` fonctionne de la même façon que `spin_lock()`, mais en outre les interruptions sont bloquées et la valeur en cours du registre d'état du processeur est enregistrée dans la variable `flags`.

- `spin_lock_irq(spinlock_t *s)` fonctionne de la même façon que `spin_lock_irqsave()` mais la valeur du registre d'état du processeur n'est pas mise en sécurité. On part du principe que les interruptions sont déjà bloquées.
- `spin_lock_bh(spinlock_t *s)` essaie d'occuper le verrou de la même manière que `spin_lock()`, bien que l'exécution simultanée des parties basses soit bloquée.

Les macros suivantes permettent d'identifier la fin de la zone critique selon la zone d'utilisation proprement dite. Elles libèrent le verrou rotatif actuellement occupé :

- `spin_unlock(spinlock_t *s)` libère un verrou rotatif occupé.
- `spin_unlock_irqrestore(spinlock *s, unsigned long flags)` libère le verrou rotatif et autorise les interruptions, si elles étaient activées lors de la mise en sécurité du registre d'état du processeur dans la variable `flags`, sinon non.
- `spin_unlock_irq(spinlock_t *s)` libère le verrou rotatif et autorise les interruptions.
- `spin_unlock_bh(spinlock_t *s)` libère le verrou et autorise le traitement immédiat des parties basses.

Les macros suivantes permettent d'accéder aux verrous rotatifs :

- `spin_is_locked(spinlock_t *s)` interroge l'état en cours du verrou, sans le modifier. La valeur renvoyée est non nulle lorsque le verrou est en place, nulle s'il est libre.
- `spin_trylock(spinlock_t *s)` occupe le verrou rotatif s'il ne l'était pas déjà lors de l'appel. Sinon une valeur non nulle est renvoyée.
- `spin_unlock_wait(spinlock_t *s)` attend la libération d'un verrou occupé. Celui-ci n'est pas occupé cependant.

10.7.4 Verrous rotatifs de lecture-écriture

Les verrous rotatifs constituent un élément simple et utile de protection contre les effets secondaires indésirables lors d'opérations parallèles sur la plupart des structures de données. Toutefois, ils freinent la progression des activités, parce que celles-ci doivent attendre la libération des verrous par scrutation. Dans certaines situations, l'attente active n'est pas toujours nécessaire. Il existe, par exemple, des structures de données à lesquelles on accède très souvent en lecture, mais rarement en écriture. Il ne s'agit pas d'arrêter les activités en lecture seule si aucune activité d'écriture n'opère à ce moment sur la structure de données.

Dans ce cas, on utilise des **verrous rotatifs de lecture-écriture** qui autorisent plusieurs activités de lecture à pénétrer dans une zone critique si aucune activité d'écriture n'est en cours. Cependant, dès qu'une activité occupe le verrou dans un but d'écriture, aucune activité de lecture ou autre activité d'écriture ne doit se trouver dans la zone critique jusqu'à ce que le verrou ait été libéré.

Les verrous de lecture-écriture sont implémentés grâce au type `rwlock_t`. Celui-ci, qui dépend de l'architecture, est défini dans le fichier `linux/include/asm-i386/spinlock.h`, dans le cas des microprocesseurs Intel :

Code Linux 2.6.10

```
154 /*
155  * Verrous rotatifs de lecture-ecriture, permettant plusieurs lectures
156  * mais une seule ecriture.
157  *
158  * NOTE ! il est relativement frequent d'avoir plusieurs lectures dans une interruption
159  * mais pas d'ecriture. Pour ces raisons nous
160  * pouvons "mixer" des verrous irq-safe - toute ecriture necessitant d'obtenir
```

```
161  * un verrou en ecriture irq-safe, mais les lectures peuvent obtenir
162  * des verrous en lecture non-irqsafe.
163  */
164 typedef struct {
165         volatile unsigned int lock;
166 #ifdef CONFIG_DEBUG_SPINLOCK
167         unsigned magic;
168 #endif
169 } rwlock_t;
170
171 #define RWLOCK_MAGIC    0xdeaf1eed
```

L'utilisation d'un verrou rotatif de lecture-écriture exige la création et l'initialisation d'une variable du type précédent de l'une des deux façons :

```
#include <linux/spinlock.h>
rwlock_t s = RW_LOCK_UNLOCKED;
```

ou :

```
#include <linux/spinlock.h>
rwlock_t s;
rwlock_init(&s);
```

La constante et la macro ci-dessus dépendent de l'architecture. Elles sont définies dans le fichier `linux/include/asm-i386/spinlock.h` dans le cas des microprocesseurs Intel : Code Linux 2.6.10

```
173 #ifdef CONFIG_DEBUG_SPINLOCK
174 #define RWLOCK_MAGIC_INIT       , RWLOCK_MAGIC
175 #else
176 #define RWLOCK_MAGIC_INIT       /* */
177 #endif
178
179 #define RW_LOCK_UNLOCKED (rwlock_t) { RW_LOCK_BIAS RWLOCK_MAGIC_INIT }
180
181 #define rwlock_init(x)  do { *(x) = RW_LOCK_UNLOCKED; } while(0)
```

Les fonctions disponibles pour l'occupation et la libération des verrous de lecture-écriture sont les suivantes :

- `read_lock()` essaie d'entrer dans une zone critique à des fins de lecture. La pénétration dans cette zone s'effectue immédiatement s'il ne s'y trouve aucune activité ou des activités de lecture seule. Si une activité d'écriture se trouve dans la zone critique, l'activité attend que le verrou soit libéré.
- `read_unlock()` quitte la zone critique dans laquelle elle était entrée en lecture seule.
- `write_lock()` essaie d'entrer dans une zone critique à des fins d'écriture. Si une activité (en écriture ou en lecture) se trouve dans la zone critique, l'activité attend que toutes les activités aient quitté la zone. L'occupation de la zone critique s'effectue alors de façon exclusive.
- `write_unlock()` fournit le verrou (en écriture) et libère ainsi la zone critique.

Il existe également différentes expressions des fonctions de verrous de lecture-écriture tenant compte de l'utilisation des interruptions et des parties basses (`..._irq()`, `.../bh()`, rtc.).

10.7.5 Sémaphore

L'inconvénient des verrous est qu'on utilise du temps machine pour la scrutation. Une autre façon est de faire appel à l'ordonnanceur. Ce concept est connu en informatique sous le nom de **sémaphore**. On parle de **mutex** dans le cas binaire (deux positions seulement).

Les sémaphores sont réalisés à l'aide de la structure de données `semaphore`. Le type, qui dépend de l'architecture, est défini dans le fichier `linux/include/asm-i386/semaphore.h`, pour les microprocesseurs Intel :

Code Linux 2.6.10

```
44 struct semaphore {
45         atomic_t count;
46         int sleepers;
47         wait_queue_head_t wait;
48 };
```

On initialise un sémaphore grâce à la macro `DECLARE_MUTEX()`.

On abaisse et on lève un sémaphore, au sens d'une barrière, grâce aux fonctions en ligne `down()` et `up()`.

10.8 Modules

Linux est un système d'exploitation à couches mais sans micro-noyau, considérant que les communications entre les extensions du micro-noyau sont beaucoup trop onéreuses en temps. Par contre il utilise la notion de **modules**, c'est-à-dire des portions de code du noyau qui peuvent être ajoutées ou retirées sans avoir à redémarrer le système, depuis la version 2.0.

Pour la suite, nous n'avons pas besoin de savoir comment ces modules sont gérés en interne, mais nous avons besoin de connaître comment concevoir et mettre en place de tels modules.

Le source d'un module s'écrit en langage C. Écrivons un module, de nom `hello.c`, qui se contente d'afficher 'Bonjour' lorsqu'on le charge et 'Au revoir' lorsqu'on le retire :

```
#define MODULE
#include <linux/module.h>

int init_module(void)
{
  printk("Bonjour\n");
}

void cleanup_module(void)
{
  printk("Au revoir\n");
}
```

La compilation du module se fait de façon habituelle, en ne générant le code objet :

```
root# gcc -c hello.c
```

On obtient ainsi un module de nom `hello.o`.

Le chargement d'un module s'effectue grâce à la commande `insmod` (pour *INSert MODule*) :

```
root# insmod ./hello.c
Bonjour
root#
```

Le retrait d'un module s'effectue grâce à la commande `rmmod` (pour *ReMove MODule*) :

```
root# rmmod hello
Au revoir
root#
```

On retrouvera souvent, dans la définition des structures, un champ déclaré de la façon suivante :

```
struct lodule *owner;
```

Il représente un pointeur sur le module qui est "propriétaire" de cette structure. Il est utilisé par le noyau pour conserver le comptage d'utilisation du module.

Le type `struct module` est défini dans le fichier en-tête `linux/scripts/modpost.h`: Code Linux 2.6.10

```
71 struct module {
72         struct module *next;
73         const char *name;
74         struct symbol *unres;
75         int seen;
76         int skip;
77         int has_init;
78         int has_cleanup;
79         struct buffer dev_table_buf;
80         char         srcversion[25];
81 };
```

10.9 Initialisation du système Linux

10.9.1 Fonctions d'initialisation

Les fonctions qui ne servent que pour l'initialisation du système Linux sont indiquées grâce au modificateur `__init`, comme indiqué dans le fichier `linux/include/linux/init.h`: Code Linux 2.6.10

```
7  /* Ces macros sont utilisees pour designer certaines fonctions ou
8   * donnees initialisees (ne s'applique pas aux donnees non initialisees)
9   * comme fonctions d'‘initialisation'. Le noyau peut prendre ceci comme
10  * une indication que la fonction est utilisee seulement durant la phase d'initialisation
11  * et liberer la memoire utilisee par les ressources apres.
12  *
13  * Utilisation :
14  * Pour les fonctions :
15  *
16  * Vous devez ajouter __init immediatement avant le nom de la fonction, comme ceci :
17  *
18  * static void __init initme(int x, int y)
19  * {
20  *    extern int z; z = x * y;
21  * }
22  *
23  * Si la fonction a un prototype quelque part, vous devez aussi ajouter
24  * __init entre la parenthese fermante du prototype et la virgule :
25  *
26  * extern int initialize_foobar_device(int, int, int) __init;
27  *
28  * Pour les donnees initialisees :
29  * Vous devez inserer __initdata entre le nom de la variable name et le signe
30  * d'egalite suivi par la valeur, par exemple :
31  *
32  * static int init_variable __initdata = 0;
33  * static char linux_logo[] __initdata = { 0x32, 0x36, ... };
34  *
35  * N'oubliez pas d'initialiser la donnee non pas dans la portee, c'est-a-dire dans une
36  * fonction, car sinon gcc place la donnee dans la section bss section et non dans
37  * la section init.
38  *
39  * Noter aussi que cette donnee ne peut pas etre "const".
40  */
```

10.9.2 Code d'initialisation

Lorsqu'on rencontre `module_init()` dans une partie du code source, la fonction indiquée entre les deux parenthèses sera placée dans un fichier spécial qui servira à initialiser le système lors du démarrage. Cette macro est définie dans le fichier `linux/include/linux/init.h`:

Code Linux 2.6.10

```
148 /**
149  * module_init() - point d'entree d'initialisation d'un peripherique
150  * @x : fonction a faire tourner lors du demarrage du noyau ou de l'insertion du module
151  *
152  * module_init() sera appele soit durant do_initcalls (si
153  * construit) ou au moment de l'insertion du module (s'il s'agit d'un module).
154  * Il peut seulement y en avoir une par module.
155  */
156 #define module_init(x)  __initcall(x);
```

La macro auxiliaire `__initcall()` est définie dans le même fichier:

Code Linux 2.6.10

```
78 /* Les initcalls sont maintenant groupes par fonctionalites dans des sous-sections
79  * separees. L'ordre a l'interieur des sous-sections est determine
80  * par l'ordre des liens.
81  * Par compatibilite ascendante, initcall() place l'appel dans la
82  * sous-section init du peripherique.
83  */
84
85 #define __define_initcall(level,fn) \
86         static initcall_t __initcall_##fn __attribute_used__ \
87         __attribute__((__section__(".initcall" level ".init"))) = fn
[...]
94 #define device_initcall(fn)             __define_initcall("6",fn)
[...]
97 #define __initcall(fn) device_initcall(fn)
```

Chapitre 11

Vue d'ensemble sur l'implémentation du sous-système réseau de Linux

Comme nous l'avons dit dans la préface, nous n'allons pas nous intéresser dans cet ouvrage au sous-système réseau de Linux dans son intégralité mais uniquement à l'envoi d'un datagramme UDP et à sa réception par un ordinateur d'extrémité.

La structuration en couches devrait nous permettre d'étudier l'envoi et la réception pour chacune des couches. L'expérience montre que ceci n'est pas pédagogique, au moins pour l'implémentation Linux.

La logique voudrait qu'on s'intéresse d'abord à l'envoi d'un datagramme UDP puis à sa réception. En fait l'expérience montre qu'il vaut mieux, toujours pour des raisons pédagogiques, le contraire.

11.1 Réception de données *via* un datagramme UDP

Lorsqu'un utilisateur envoie des données *via* le protocole UDP, celles-ci sont encapsulées dans un datagramme UDP, puis dans un paquet IP puis dans une trame. Nous supposerons que l'ordinateur récepteur est relié au réseau par une carte Ethernet.

11.1.1 Réception de la trame Ethernet par la carte réseau

La trame voyage sur le support physique à travers le réseau. Il y a trois possibilités :

- elle se perd lors de ses pérégrinations ;
- elle arrive corrompue sur la carte Ethernet du destinataire ;
- elle arrive dans son intégrité sur la carte Ethernet du destinataire.

Nous n'avons vu que le protocole Ethernet exige qu'un minuteur soit déclenché lors de l'émission d'une trame. Dans le premier cas, la machine émettrice ne recevra pas d'accusé de réception. Au bout d'un certain délai, elle enverra à nouveau la trame (en itérant un certain nombre de fois).

Dans le second cas, si ceci est détecté grâce au CRC, la carte réseau de la machine réceptrice écarte la trame. La machine émettrice ne recevra donc pas d'accusé de réception et on se retrouvera dans la situation du premier cas.

Dans le second cas, si ceci n'est pas détecté par le CRC, et dans le troisième cas, le contrôleur de la carte réseau compare l'adresse matérielle du destinataire de la trame avec celle de la carte réseau de la machine réceptrice sur laquelle est arrivée la trame. Si celles-ci concordent, il déclenche une interruption matérielle pour annoncer au microprocesseur de l'ordinateur qu'un paquet est disponible dans la mémoire de la carte réseau. Il laisse le pilote de cette carte agir à sa guise.

Si les deux adresses ne concordent pas, il y a trois cas dans lesquels le contrôleur de la carte a quand même un comportement analogue : dans le cas d'une trame de diffusion générale, dans celui d'une trame de multidiffusion dont l'adresse est acceptée par la carte et dans le cas d'un mode spécial, dit **mode de promiscuité**. Nous ne nous occuperons pas de ces cas dans cet ouvrage.

Le mode de promiscuité, comme son nom le laisse entendre, indique à la carte réseau d'accepter toutes les trames qui parviennent à celles-ci. Ceci est utile pour les **renifleurs**, qui analysent tout ce qui passe sur le réseau, soit dans un but avouable de contrôle, soit dans un but d'espionnage.

Dans les autres cas, la trame est rejetée par le contrôleur.

11.1.2 Récupération du paquet par l'ordinateur

Nous venons de voir que lorsqu'une trame arrive en bon état sur la carte réseau de l'ordinateur qui lui est destiné, le contrôleur de la carte déclenche une interruption. Le gestionnaire de cette interruption, qui est une partie du **pilote** de la carte réseau, alloue un emplacement en mémoire vive pour ce paquet, appelé **tampon de socket** (voir chapitres 12 et 13), effectue quelques vérifications sur ce paquet et, si elles sont concluantes, le place dans ce tampon de socket, ainsi qu'une partie de l'en-tête de trame, comprenant les adresses matérielles source et de destination ainsi que le type du paquet. Il alloue également un **descripteur de tampon de socket** contenant des informations sur celui-ci (début de son emplacement, sa taille, son heure d'arrivée...).

Suivant le type du paquet, le gestionnaire fait appel à une fonction de traitement de celui-ci, ayant pour seul paramètre l'adresse du descripteur de tampon de socket. Dans le cas d'un paquet IPv4, il s'agit de `ip_rcv()`.

11.1.3 Traitement du paquet par la couche réseau

Dans le cas d'une trame destinée à IPv4, la couche réseau traite le paquet grâce à la fonction `ip_rcv()`. Celle-ci vérifie d'abord s'il s'agit d'un paquet entier ou d'un sous-paquet dû à une fragmentation. Dans ce dernier cas, elle place le sous-paquet dans une file d'attente et déclenche un minuteur en attendant de recevoir tous les sous-paquets contituant le paquet originel. Si le délai s'écoule sans que tous les sous-paquets soient arrivés, elle détruit les sous-paquets arrivés se trouvant dans la file d'attente et envoie un message ICMP à la machine émettrice pour indiquer de renvoyer le paquet. Sinon elle réassemble les sous-paquets pour obtenir le paquet originel qui sera placé dans un seul tampon de socket puis démultiplexie la couche de transport grâce au champ concernant celle-ci de l'en-tête IP. Ceci consiste à transmettre le descripteur de tampon de socket à la fonction `udp_rcv()` dans le cas de UDP.

11.1.4 Traitement du datagramme par la couche de transport

Dans le cas de UDP, la couche de transport exécute une vérification minimale puis transmet le message à l'application concernée.

On commence par vérifier la longueur du datagramme. Si celle-ci est inférieure à celle d'un en-tête UDP, on se contente d'afficher un message noyau et de détruire le tampon de socket. Sinon, si la longueur du datagramme est supérieure à celle indiquée dans l'en-tête UDP, on se contente également d'afficher un message noyau et de détruire le tampon de socket. On calcule ensuite la somme de contrôle du datagramme. Si elle ne concorde pas avec celle spécifiée dans l'en-tête UDP, on se contente également d'afficher un message noyau et de détruire le tampon de socket.

Si toutes les vérifications sont concluantes, on place le tampon de socket (en fait un nouveau descripteur, celui de couche de transport) dans la file d'attente de réception des datagrammes UDP.

11.1.5 Récupération par l'utilisateur

L'utilisateur crée une socket de la façon vue au chapitre 6 et lit, dans le cas non connecté, à travers celle-ci grâce à la fonction `recvfrom()`. Celle-ci fait appel à la fonction de réception spécifique à la famille de protocoles, à savoir `sock_common_recvmsg()` dans le cas de IPv4. Cette dernière se contente de démultiplexer suivant la couche de transport : elle fait appel à la fonction `udp_recvmsg()` dans le cas de UDP.

La fonction `udp_recvmsg()` essaie de récupérer un tampon de socket dans la file d'attente de réception UDP, en faisant appel à `skb_recv_datagram()` et en attendant que le bon tampon de socket arrive s'il ne s'y trouve pas déjà. On lit alors le nombre de caractères voulu, que l'on place dans l'espace utilisateur.

11.2 Envoi de données *via* un datagramme UDP

11.2.1 Envoi par l'utilisateur

L'utilisateur crée une socket de la façon vue au chapitre 6 et envoie, dans le cas non connecté, à travers celle-ci grâce à la fonction `sendto()`. Celle-ci crée un message à partir des données, comprenant en particulier un en-tête de message, puis fait appel à la fonction `sock_sendmsg()`, elle-même faisant appel à la fonction `__sock_sendmsg()`, qui fait appel à la fonction d'envoi spécifique à la famille de protocoles, à savoir `inet_sendmsg()` dans le cas de IPv4. Cette dernière

attribue un numéro de port au message puis démultiplexie suivant la couche de transport : elle fait appel à la fonction `udp_sendmsg()` dans le cas de UDP.

11.2.2 Traitement du datagramme par la couche de transport

La fonction `udp_sendmsg()` commence par constituer le datagramme UDP à partir du message. Si la taille des données est supérieure à 65 535 octets, celles-ci ne peuvent pas tenir dans un datagramme ; on s'arrête donc purement et simplement. Il en est de même si on essaie d'envoyer des données urgentes (grâce à un positionnement des drapeaux), puisque ceci n'est pas prévu dans le cas de UDP. On récupère l'adresse et le numéro de port pour renseigner l'en-tête UDP. On essaie d'initialiser une entrée de cache de routage ; on est sûr d'y parvenir si la table de routage possède une adresse par défaut. On fait enfin appel à la fonction `udp_flush_pending_frames()` pour constituer le datagramme UDP puis à la fonction `udp_push_pending_frames()` pour envoyer le datagramme à la couche réseau. Il n'y a pas beoin de démultiplexer dans ce sens, puisque la couche réseau est nécessairement IPv4.

11.2.3 Envoi des paquets ordinaires sous IPv4

La première partie consiste à router le paquet grâce à la fonction `ip_route_output_flow()`, chargée de fournir une entrée de cache de routage. On remarquera que l'on a déjà parlé de cette action dans la sous-section précédente ; ceci est dû au fait que Linux ne tient pas compte des couches de façon stricte. L'entrée de cache de routage précise en particulier que le paquet devra être transmis, dans le cas d'un paquet ordinaire, à la fonction `ip_output()`.

La seconde étape consiste à constituer le paquet IP. Pour cela on essaie de récupérer le premier descripteur de tampon de socket de la file d'attente en écriture du descripteur de couche de transport passé en argument à la fonction `ip_push_pending_frames()`. On récupère ensuite tous les descripteurs de cette file d'attente et on les place dans la liste des fragments. On renseigne les champs d'un nouvel en-tête IP puis on fait appel à la fonction `ip_output()` déterminée précédemment.

La troisième étape consiste à transférer le paquet de la couche réseau à la couche inférieure. Pour cela, on fragment éventuellement le paquet, on positionne le type du paquet de la sous-couche 2b à `ETH_P_IP` et on transmet à la fonction `ip_finish_output2()`, qui fait appel à la fonction `dev_queue_xmit()` d'envoi d'une trame.

11.2.4 Envoi de la trame

La fonction `dev_queue_xmit()`, où `skb` désigne le descripteur de tampon de socket, indique à la carte réseau qu'il y a des données à envoyer et leur emplacement en mémoire vive. La carte s'occupe du reste, utilisant l'accès direct à la mémoire (DMA) pour récupérer les données. À la fin de l'émission, la carte lève une interruption pour prévenir le noyau que la trame a été envoyée avec succès ou que le délai s'est écoulé.

Les données consistent en un paquet de la couche réseau. La carte s'occupe d'encapsuler celui-ci pour obtenir une trame.

11.3 Historique

Une implémentation UUCP a tourné sous Linux presque dès le début. L'implémentation de TCP/IP a commencé en 1992 lorsque Ross Biro et d'autres ont créé ce qui fut ensuite connu sous le nom de **Net-1.**

Ross Biro ne s'occupa plus du développement de Linux à partir de mai 1993. Fred van Kempen commença alors à travailler sur une nouvelle implémentation, qui sera connue sous la dénomination de **Net-2**. La première version publique, Net-2d, apparut dans le noyau 0.99.10 de l'été 1993. Alan Cox commença a s'intéresser au projet, concevant Net-2Debugged, puis passa à la version **Net-3** pour Linux 1.2 et Linux 2.0. Le noyau 2.2 utilise `Net-4`.

Chapitre 12

Les tampons de socket

Nous avons vu que chaque paquet (reçu ou à envoyer) occupe une partie de la mémoire vive, avec un enrobage d'en-têtes (et éventuellement d'en-queues) en couches d'oignon. Linux ne réserve pas un emplacement en mémoire vive pour chaque couche, mais un seul emplacement pour le paquet, dont le début peut s'accroître ou rétrécir à souhait pour tenir compte des en-têtes des différentes couches. Il en est de même de la fin pour tenir compte des en-queues des différentes couches. On parle de *tampon* pour cet emplacement, comme pour tout emplacement en mémoire vive, et plus précisément de **tampon de socket** (*socket buffer* en anglais) pour les distinguer, par exemple, des tampons de l'antémémoire des disques.

À chaque tampon de socket est associé un **descripteur de tampon de socket**, permettant de le manipuler.

Ces descripteurs sont placés dans diverses files d'attente.

Dans ce chapitre nous allons décrire la structure des tampons de socket, des descripteurs de tampon de socket et des files d'attente de descripteurs de tampon de socket, c'est-à-dire l'aspect statique. Dans le chapitre suivant nous en étudierons la gestion, c'est-à-dire l'aspect dynamique.

12.1 Structure d'un tampon de socket

Un tampon de socket est un emplacement mémoire sur lequel il n'y a rien à dire *a priori* (contrairement à la structure de son descripteur). Les notions d'espace de tête, d'espace de queue et de fragmentation sont cependant nécessaires pour la suite.

12.1.1 Espace de tête et espace de queue d'un tampon de socket

Un tampon de socket en émission contiendra à un certain moment le paquet à transmettre à la carte réseau. Cependant, durant le déroulement de la pile réseau, ce tampon commence par recevoir les données de l'application, puis l'en-tête de la couche transport, puis l'en-tête de la couche réseau, puis l'en-tête MAC. De même, les données de l'application seront éventuellement suivies d'en-queues divers.

Sous Linux, la zone de mémoire vive non encore remplie, située avant la partie du paquet valide à un moment donné, est appelée **espace de tête** (*headroom* en anglais). De même, la zone de mémoire placée après la partie du paquet valide à cet instant est appelée **espace de queue** (**tailroom** en anglais).

12.1.2 Fragmentation d'un tampon de socket

Les tampons de socket peuvent être trop grand pour tenir dans une page. Dans ce cas ils sont dispersés sur plusieurs pages, on dit qu'ils sont **fragmentés**. De plus nous avons vu qu'il peut s'agir du regroupement de plusieurs tampons ; il est plus simple dans ce cas de conserver cette structure marquant le regoupement (tout au moins momentanément) plutôt que de "linéariser" immédiatement.

12.1.2.1 Descripteur de fragment

Un fragment de tampon de socket sera contenu entièrement dans une page. Cependant il n'occupe pas nécessairement la page en entier, ne serait-ce que parce que la taille du tampon n'est pas un multiple de la taille d'une page.

Le type `struct skb_frag_t` permet de détenir les informations sur les fragments. Il est défini dans le fichier en-tête `linux/include/linux/skbuff.h`:

Code Linux 2.6.10

```
129 typedef struct skb_frag_struct skb_frag_t;
130
131 struct skb_frag_struct {
132         struct page *page;
133         __u16 page_offset;
134         __u16 size;
135 };
```

qui spécifie le numéro de page, le **décalage** ou adresse du début des données dans cette page et la taille, c'est-à-dire le nombre d'octets utiles dans cette page à partir du décalage. Ceci détermine une zone contiguë de mémoire, entièrement contenue dans une même page.

12.1.2.2 Informations sur la fragmentation

Les informations sur la fragmentation font l'objet d'une entité du type structuré `struct skb_shared_info`, qui comprend un tableau de fragments et une liste de descripteurs de tampon de socket. Celui-ci est défini dans le fichier en-tête `linux/include/linux/skbuff.h`:

Code Linux 2.6.10

```
137 /* Ces donnees ne varient pas d'un clone a l'autre et sont situees a la
138  * fin de l'en-tete des donnees, c'est-a-dire a skb->end.
```

```
139  */
140 struct skb_shared_info {
141         atomic_t        dataref;
142         unsigned int    nr_frags;
143         unsigned short  tso_size;
144         unsigned short  tso_segs;
145         struct sk_buff  *frag_list;
146         skb_frag_t      frags[MAX_SKB_FRAGS];
147 };
```

dont les champs spécifient :

- s'il est effectivement fait référence à cette entité ou non (autrement dit si on peut l'effacer de la mémoire vive si besoin est) ;
- le nombre de fragments dans le tableau de fragments, autrement dit la taille effective de celui-ci ;
- la taille et les segments concernat le protocole de transport TCP (avec `tso` pour *TCP Segmentation On*) ;
- une **liste de fragments**, chaque fragment étant un (descripteur de) tampon de socket ;
- un **tableau de fragments** représentant des emplacements en mémoire vive permettant de détenir 64 Ko de données sans avoir à recourir à une liste de fragments (champ précédent).

  Le nombre `MAX_SKB_FRAGS` de pages nécessaires pour détenir 64 Ko est également défini dans le fichier en-tête `linux/include/linux/skbuff.h`: Code Linux 2.6.10

```
126 /* Pour permettre une trame de 64K d'etre empaquetee comme un seul skb sans frag_list */
127 #define MAX_SKB_FRAGS (65536/PAGE_SIZE + 2)
```

Ordre de citation du code.- Nous nous arrangerons toujours, lors de la citation du code de Linux, pour ne pas avoir de référence en avant, c'est-à-dire à ne pas utiliser de type ou de fonction qui ne serait étudié que plus loin. Remarque

Lors de la définition du type `struct skb_frag_t`, on rencontre le type `__u16`, mais celui-ci a déjà été introduit précédemment. Il est vrai que l'on rencontre également le type `page`. Il s'agit d'un type général concernant la gestion de la mémoire. Nous ne pouvons pas revoir tout le noyau Linux.

On rencontre également, lors de la définition du type `struct skb_shared_info`, *a priori* le type `sk_buff` non défini antérieurement. En fait si on observe bien, ce n'est pas ce type qui intervient mais un pointeur sur ce type, c'est-à-dire une adresse. On pourrait tout aussi bien le remplacer par le type `void *`. On n'a donc pas besoin de connaître explicitement ce type à ce moment. Cette situation se retrouvera très souvent dans la suite.

12.2 Descripteur de tampon

Linux définit un type de descripteur de tampon de socket assez complexe, différent des autres implémentations dont les sources sont disponibles.

La figure 10-1 ([WPRMB–02], p. 66) montre les relations entre un tampon et son descripteur.

Commençons par citer la définition du type descripteur de tampon de socket avant d'en commenter les champs un à un dans les sections suivantes.

12.2.1 Définition du type

Le type `struct sk_buff` est défini dans le fichier `linux/include/linux/skbuff.h`: Code Linux 2.6.10

```
1 /*
```

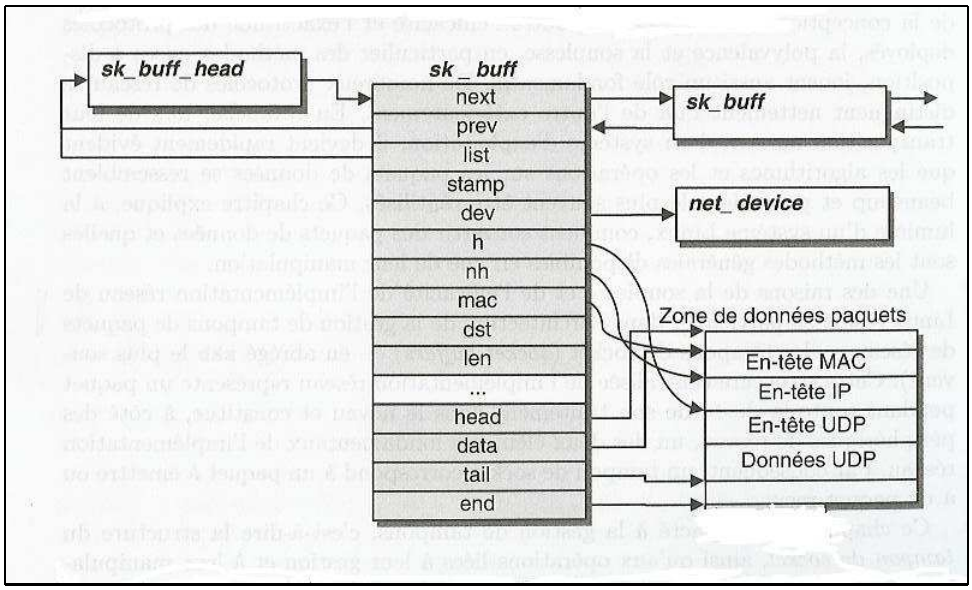


FIG. 12.1 – *Tampon de socket et son descripteur*

```
2   *      Definitions pour les routines de memoire de 'struct sk_buff'.
3   *
4   *      Auteurs :
5   *              Alan Cox, <gw4pts@gw4pts.ampr.org>
6   *              Florian La Roche, <rzsfl@rz.uni-sb.de>
[...]
14 #ifndef _LINUX_SKBUFF_H
15 #define _LINUX_SKBUFF_H
[...]
149 /**
150  *      struct sk_buff - tampon de socket
151  *      @next : Tampon suivant dans la liste
152  *      @prev : Tampon precedent dans la liste
153  *      @list : Liste dans laquelle nous nous trouvons
154  *      @sk : Socket dont nous sommes la propriete
155  *      @stamp : Heure a laquelle nous sommes arrives
156  *      @dev : Peripherique par lequel nous sommes arrives/nous partirons
157  *      @input_dev : Peripherique par lequel nous sommes arrives
158  *      @real_dev : Le peripherique effectif que nous sommes en train d'utiliser
159  *      @h : En-tete de la couche transport
160  *      @nh : En-tete de la couche reseau
161  *      @mac : En-tete de la couche liaison
162  *      @dst : A RECTIFIER : Decrire ce champ
163  *      @cb : Tampon de controle. Utilisation libre par chaque couche. Mettre les vars
        privees ici
164  *      @len : Longueur des donnees en cours
165  *      @data_len : Longueur des donnees
166  *      @mac_len : Longueur de l'en-tete de couche de liaison
167  *      @csum : Somme de controle
168  *      @__unused : Champ decede, peut etre reutilise
169  *      @cloned : La tete peut etre clonee (verifier refcnt pour en etre sur)
170  *      @pkt_type : Classe de paquet
171  *      @ip_summed : Le peripherique nous a fourni une somme de controle IP
172  *      @priority : Priorite de mise en file d'attente du paquet
```

```
173  *      @users : Compteur d'utilisateurs - voir {datagram,tcp}.c
174  *      @protocol : Protocole du paquet depuis le peripherique
175  *      @security : Niveau de securite du paquet
176  *      @truesize : Taille du tampon
177  *      @head : Tete du tampon
178  *      @data : Pointeur sur le debut des donnees
179  *      @tail : Pointeur sur la fin des donnees
180  *      @end : Pointeur sur la fin
181  *      @destructor : Fonction de destruction
182  *      @nfmark : Peut etre utilise pour la communication entre points d'ancrage
183  *      @nfcache : Infos sur le cache
184  *      @nfct : Connexion associee, si elle existe
185  *      @nfctinfo : Relations de ce skb avec la connexion
186  *      @nf_debug : Debogage Netfilter
187  *      @nf_bridge : Donnees sauvegardees sur une trame de pont - voir br_netfilter.c
188  *      @private : Donnees qui sont privees pour l'implementation de HIPPI
189  *      @tc_index : Index de controle du trafic
190  */
191
192 struct sk_buff {
193         /* Ces deux membres doivent apparaitre en premier. */
194         struct sk_buff          *next;
195         struct sk_buff          *prev;
196
197         struct sk_buff_head     *list;
198         struct sock             *sk;
199         struct timeval          stamp;
200         struct net_device       *dev;
201         struct net_device       *input_dev;
202         struct net_device       *real_dev;
203
204         union {
205                 struct tcphdr   *th;
206                 struct udphdr   *uh;
207                 struct icmphdr  *icmph;
208                 struct igmphdr  *igmph;
209                 struct iphdr    *ipiph;
210                 struct ipv6hdr  *ipv6h;
211                 unsigned char   *raw;
212         } h;
213
214         union {
215                 struct iphdr    *iph;
216                 struct ipv6hdr  *ipv6h;
217                 struct arphdr   *arph;
218                 unsigned char   *raw;
219         } nh;
220
221         union {
222                 unsigned char   *raw;
223         } mac;
224
225         struct  dst_entry       *dst;
226         struct  sec_path        *sp;
227
228         /*
229          * Ceci est le tampon de controle. Son utilisation par chaque couche est
230          * libre. Mettre vos variables privees ici s'il vous plait. Si vous
231          * voulez les conserver en passant d'une couche a une autre, vous devez d'abord
232          * faire un skb_clone(). Ce tampon est possede par celui qui a mis le skb dans la file
              d'attente ATM.
233          */
```

```
234         char                    cb[40];
235
236         unsigned int            len,
237                                 data_len,
238                                 mac_len,
239                                 csum;
240         unsigned char           local_df,
241                                 cloned,
242                                 pkt_type,
243                                 ip_summed;
244         __u32                   priority;
245         unsigned short          protocol,
246                                 security;
247
248         void                    (*destructor)(struct sk_buff *skb);
249 #ifdef CONFIG_NETFILTER
250         unsigned long           nfmark;
251         __u32                   nfcache;
252         __u32                   nfctinfo;
253         struct nf_conntrack     *nfct;
254 #ifdef CONFIG_NETFILTER_DEBUG
255         unsigned int            nf_debug;
256 #endif
257 #ifdef CONFIG_BRIDGE_NETFILTER
258         struct nf_bridge_info   *nf_bridge;
259 #endif
260 #endif /* CONFIG_NETFILTER */
261 #if defined(CONFIG_HIPPI)
262         union {
263                 __u32           ifield;
264         } private;
265 #endif
266 #ifdef CONFIG_NET_SCHED
267         __u32                   tc_index;       /* index de controle du trafic */
268 #ifdef CONFIG_NET_CLS_ACT
269         __u32           tc_verd;                /* verdict de controle de trafic */
270         __u32           tc_classid;             /* id de classe de controle de trafic */
271 #endif
272
273 #endif
274
275
276         /* Ces elements doivent etre a la fin, voir alloc_skb() pour les details.  */
277         unsigned int            truesize;
278         atomic_t                users;
279         unsigned char           *head,
280                                 *data,
281                                 *tail,
282                                 *end;
283 };
```

12.2.2 Champs relatifs aux listes de descripteurs de tampon

12.2.2.1 Parcours de la liste de descripteurs de tampon

Les deux premiers champs `next` et `prev` permettent au descripteur de tampon de se situer dans une liste doublement chaînée.

12.2.2.2 File d'attente à laquelle appartient le tampon

Le champ `list` indique la file d'attente dans laquelle le descripteur de tampon se trouve en ce moment. Les files d'attente doivent par conséquent toujours être du type `struct sk_buff_head`, afin que les opérations de tampons de socket puissent les gérer. Si un tampon n'est assigné à aucune file d'attente, ce champ prend la valeur `NULL`. Nous reviendrons dans la troisième section sur la structure des files d'attente de descripteurs de tampon de socket.

12.2.3 Provenance du tampon

La provenance du tampon est indiquée par plusieurs champs :

- L'attribut `sk` indique le descripteur de couche transport associé au tampon, qu'il soit un tampon d'émission ou de réception. Le type `struct sock`, dont nous n'avons pas vraiment besoin ici (seul le type adresse est en fait utilisé), sera étudié au moment opportun, c'est-à-dire au chapitre 28.
- L'attribut `stamp` indique le moment précis, en `jiffies`, où il est arrivé (de l'extérieur en cas de réception, depuis le processus en cas d'émission).
- L'attribut `dev` spécifie le périphérique réseau auquel il est attaché : depuis lequel il provient en cas de réception, auquel il est destiné en cas d'émission. Le type `struct net_device`, dont nous n'avons pas vraiment besoin ici (seul le type adresse est en fait utilisé), sera étudié au moment opportun.
- Dans le cas d'un routeur (ce qui ne nous intéressera pas dans cet ouvrage), l'attribut `input_dev` spécifie le périphérique réseau duquel il provient, l'attribut précédent le périphérique par lequel il partira.
- L'attribut `real_dev` spécifie le périphérique réseau que nous sommes en train d'utiliser, qui peut être différent des périphériques ci-dessus.

12.2.4 En-têtes des couches de protocole

Nous avons vu que les tampons de socket contiennent les données de l'application ainsi que les encapsulations successives. Les trois champs suivants permettent donc de préciser les en-têtes de la couche transport, de la couche réseau et de la sous-couche MAC.

12.2.4.1 En-tête de la couche de transport

L'en-tête `h` (pour *Header*) de la couche de transport dépend évidemment du protocole de transport. Le type associé est donc une union, dont l'élément est l'en-tête des protocoles TCP (type `tcphdr`), UDP (type `udphdr`), ICMP (type `icmphdr`), IGM (type `igmhdr`, défini dans le fichier en-tête `linux/include/linux/igm.h`), IPv4 (type `iphdr`, qui concerne normalement la couche réseau et non la couche de transport sauf dans le cas de données brutes), IPv6 (type `ipv6hdr`, défini dans le fichier en-tête `linux/include/linux/ipv6.h`) ou brut (qui est alors tout simplement une chaîne de caractères). La structure de chacun de ces en-têtes, qui n'est pas utile ici (seul le type adresse est utilisé), sera décrite aux chapitres consacrés à ces protocoles.

12.2.4.2 En-tête de la couche réseau

L'en-tête `nh` (pour *Net Header*) de la couche réseau dépend lui aussi du protocole choisi, protocole de réseau dans ce cas. Le type associé est donc également une union, dont l'élément est l'en-tête des protocoles IP version 4 (type `iphdr`), IP version 6, ARP (type `arphdr` défini dans le fichier en-tête `linux/include/if_arp.h`) ou brut (qui est alors tout simplement une chaîne de

caractères). Comme pour le champ précédent, la structure de chacun de ces en-têtes, qui n'est pas utile ici, sera décrite aux chapitres consacrés à ces protocoles.

12.2.4.3 En-tête de la sous-couche MAC

L'en-tête `mac` de la sous-couche MAC dépend lui aussi du protocole choisi. Le type associé est donc également une union, dont l'élément a été longtemps soit un en-tête Ethernet (type `ethhdr`) soit brut (qui est alors tout simplement une chaîne de caractères). On n'a plus le choix depuis la version 2.6.10, mais on a gardé l'union pour la compatibilité avec les versions antérieures.

12.2.5 Attributs divers

12.2.5.1 Entrée de cache de destination

Le champ `dst` renvoie à une entrée dans le cache de routage. Il contient des informations sur l'itinéraire du paquet, par exemple la carte réseau sur laquelle le paquet doit quitter l'ordinateur (dans le cas où il y en a plusieurs) et éventuellement un renvoi sur un en-tête MAC tout fait déjà stocké comme champ de cette entrée. Nous étudierons le type `dst_entry`, qui n'est pas vraiment utile ici, avec les structures de données pour IP.

12.2.5.2 Chemin sécurisé

Le champ `sp` est utile lorsqu'on veut un chemin sécurisé, ce qui ne nous intéressera pas dans ce livre. Le type `struct sec_path` n'est pas vraiment utile, puisqu'il s'agit d'une adresse.

12.2.5.3 Bloc de contrôle du protocole

Le champ `cb` contient des informations concernant la couche que l'on est en train de traiter et à usage libre par celle-ci. Ces informations sont surchargées lors du passage d'une couche à une autre, d'où l'avertissement concernant le clonage s'il est nécessaire de ne pas pas perdre celles-ci.

12.2.5.4 Attributs généraux

Les attributs généraux sont les suivants :

- La taille `len` désigne la longueur de la partie du tampon qui est renseignée à ce moment. Il s'agit d'un attribut redondant puisque ce n'est rien d'autre que `tail - data`. Seules les données accessibles par le noyau sont prises en considération. Avec une trame Ethernet par exemple, seules les deux adresses MAC et le champ type/longueur sont pris en considération. Les autres champs (préambule, champ de remplissage et somme de contrôle) sont traités par la carte réseau, par conséquent le noyau n'en tient pas compte.
- La taille `data_len` désigne la longueur de la partie du paquet qui a un sens lors du traitement dans la couche dans laquelle nous nous trouvons.
- Le champ `mac_len` spécifie la longueur de l'en-tête de la sous-couche MAC de la couche liaison.
- Le champ `csum` spécifie la somme de contrôle.
- Le champ `local_df` (auparavant `__unused`, comme encore référencé dans le commentaire) est à usage local.

– Le champ `cloned` indique si le descripteur de tampon a été cloné. Si c'est le cas, on se trouve alors en présence de plusieurs descripteurs de tampon faisant référence au même tampon de socket.

– Le type ou classe du paquet `pkt_type` est utilisé lors de la livraison du paquet. Le pilote doit lui assigner l'une des valeurs `PACKET_HOST` (paquet à remettre localement), `PACKET_BROADCAST` (paquet de diffusion générale), `PACKET_MULTICAST` (paquet de multidiffusion), `PACKET_OTHERHOST` (paquet qui n'était pas destiné à l'ordinateur local, mais dont la réception a été rendue possible grâce au un mode spécial de promiscuité), `PACKET_OUTGOING` (paquet devant quitter l'ordinateur), `PACKET_LOOPBACK` (paquet provenant du système local vers lui-même) ou `PACKET_FASTROUTE` (transmission rapide entre cartes réseau spéciales).

Les constantes symboliques représentant les types de paquets sont définies dans le fichier en-tête `linux/include/linux/if_packet.h`: Code Linux 2.6.10

```
22 /* Types de paquet */
23
24 #define PACKET_HOST            0              /* Pour nous                */
25 #define PACKET_BROADCAST       1              /* Pour tous                */
26 #define PACKET_MULTICAST       2              /* Pour le groupe           */
27 #define PACKET_OTHERHOST       3              /* Pour quelqu'un d'autre   */
28 #define PACKET_OUTGOING        4              /* Sortie de tout type      */
29 /* Ceux-ci sont invisibles au niveau utilisateur */
30 #define PACKET_LOOPBACK        5              /* MC/BRD frame looped back */
31 #define PACKET_FASTROUTE       6              /* Trame Fastroutee         */
```

– Le comportement `ip_summed` du pilote de périphérique à l'égard de la somme de contrôle IP est représenté par l'une des des constante symboliques suivantes (aucune, fournie par le matériel ou non nécessaire), définies dans le fichier `linux/include/linux/skbuff.h`: Code Linux 2.6.10

```
36 #define CHECKSUM_NONE       0
37 #define CHECKSUM_HW         1
38 #define CHECKSUM_UNNECESSARY 2
```

Un commentaire se trouve dans le fichier en-tête `linux/include/linux/skbuff.h`: Code Linux 2.6.10

```
48 /* A. Verification de la somme des paquets recus par le peripherique.
49  *
50  *      NONE : le peripherique a echoue en verifiant la somme de ce paquet.
51  *             skb->csum n'est pas defini.
52  *
53  *      UNNECESSARY : le peripherique a etudie le paquet et a du verifie la somme.
54  *             skb->csum n'est pas defini.
55  *           C'est une mauvaise option, mais malheureusement beaucoup de vendeurs font
56  *           ceci. Apparemment dans le but secret de vous vendre un nouveau peripherique
57  *           lorsque vous ajouterez un nouveau protocole a votre hote, par exemple
                 IPv6. 8)
58  *
59  *      HW : La facon la plus generique. Le peripherique supplee la somme de controle de
60  *          _tout_ le paquet comme vu par netif_rx dans skb->csum.
61  *          NOTE : Meme si le peripherique supporte seulement quelques protocoles, mais
62  *          est capable de produire quelques skb->csum, il DOIT utiliser HW,
63  *          et non UNNECESSARY.
64  *
65  * B. Verification en sortie.
66  *
67  *      NONE : la somme de controle de skb est calculee par le protocole ou csum n'est
                 pas necessaire.
68  *
69  *      HW : le peripherique est necessaire pour calculer la somme csum du paquet comme
70  *      vu par hard_start_xmit a partir de skb->h.raw a la fin et pour enregister la
71  *      somme de controle a skb->h.raw+skb->csum.
```

Il est un peu curieux de rencontrer ainsi un champ relatif à une couche donnée, et même à un seul protocole de cette couche, mais c'est ainsi.

- Le champ `priority` spécifie la priorité du paquet.
- Le champ `protocol` sert à transmettre le tampon de la sous-couche LLC à la couche réseau. La valeur qui nous intéresse principalement est `ETH_P_IP`. Les valeurs possibles seront définies lors de l'étude de l'implémentation du protocole Ethernet.
- Le champ `security` spécifie le niveau de sécurité du paquet, ce qui ne nous intéressera pas dans cet ouvrage.

12.2.5.5 Fonction de destruction du tampon

Le tampon devra être détruit après utilisation. La façon de faire dépend d'un certain nombre de paramètres, d'où l'intérêt de la fonction membre `destructor()`.

12.2.5.6 Attributs optionnels

Un certain nombre d'attributs complémentaires existent suivant les options choisies pour la compilation du noyau, dont aucune ne nous intéressera dans cet ouvrage :

- `nfmark` pour la communication entre points d'ancrage si on a choisi le filtrage réseau ;
- `nfcache` pour des informations sur le cache si on a choisi le filtrage réseau ;
- `nfctinfo` pour des informations sur le cache si on a choisi le filtrage réseau ;
- `nfct` pour la connexion associée si on a choisi le filtrage réseau ;
- `nf_debug` si on a choisi le filtrage réseau avec option de débogage ;
- `nf_bridge` pour sauvegarder les données d'une trame dans le cas d'un pont ;
- `private` si on a choisi l'option des périphériques à haut débit `HIPPI` (*High-Performance Parallel Interface*) ;
- l'index de contrôle du trafic si on a choisi l'option `CONFIG_NET_SCHED`.
- et enfin le verdict et l'identificateur de classe de contrôle du trafic si on a choisi l'option `CONFIG_NET_CLS_ACT`.

12.2.5.7 Attributs finaux

Les derniers attributs doivent être placés à la fin pour faciliter l'implémentation de la fonction `alloc_skb()`. Ils concernent :

- La taille `truesize` du tampon, données, en-têtes et attributs de contrôle compris.
- Le nombre `users` d'utilisateurs du tampon.
- L'entreposage du tampon. L'emplacement du tampon en mémoire vive est caractérisé par quatre attributs :
  - un pointeur `head` sur le début du tampon (espace alloué), autrement dit le début de l'espace de tête (*headroom*) ;
  - un pointeur `data` sur le début des données remplies de façon valide à ce moment, dont l'adresse est donc supérieure à l'adresse précédente ;
  - un pointeur `tail` sur l'octet suivant la fin des données remplies de façon valide à ce moment, autrement dit le début de l'espace de queue (*tailroom*) ;
  - un pointeur `end` sur l'adresse de fin de l'espace alloué, supérieur ou égal à `tail`.

12.2.5.8 Informations sur la fragmentation du tampon

Une entité du type `struct skb_shared_info` est repérée par le champ `end`, qui joue donc deux rôles.

12.3 Files d'attente de tampons de socket

Lorsqu'un paquet n'est pas traité immédiatement, il est placé dans une file d'attente. Linux prend en charge la gestion des paquets grâce à la structure de file d'attente de descripteurs de socket `sk_buff_head` et à des opérations afférentes.

Comme le montre la figure 12-2 ([WPRMB–02], p.79), les descripteurs de tampon de socket stockés dans une file d'attente sont doublement chaînés l'un à l'autre au moyen d'une structure en anneau. Le double chaînage permet une navigation rapide dans les deux sens. La structure en anneau facilite le chaînage et évite l'apparition du pointeur `NULL`.

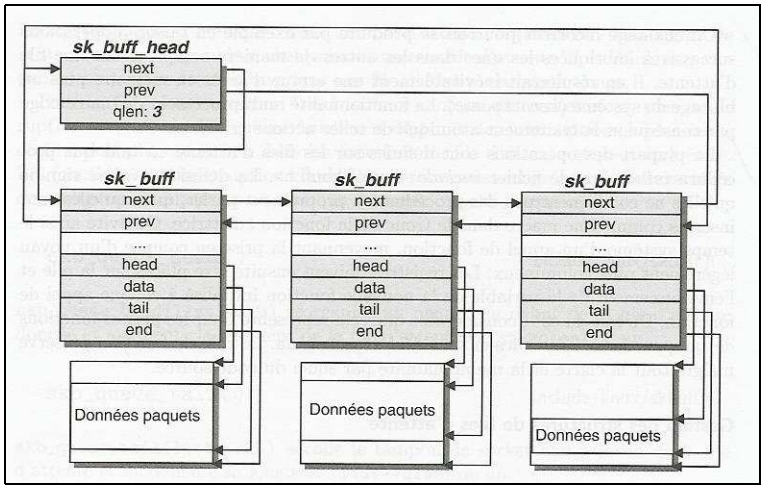


FIG. 12.2 – *File d'attente de paquets*

Le type `sk_buff_head` est défini dans le fichier en-tête `linux/include/linux/skbuff.h`: Code Linux 2.6.10

```
115 struct sk_buff_head {
116         /* Ces deux membres doivent etre en premier. */
117         struct sk_buff  *next;
118         struct sk_buff  *prev;
119
120         __u32           qlen;
121         spinlock_t      lock;
122 };
```

- les champs `next` et `prev` des descripteurs de tampon de socket servent au chaînage des descripteurs; le champ `next` de `sk_buff_head` indique le premier paquet placé dans la file d'attente et `prev` le dernier;
- le champ `qlen` (pour l'anglais *Queue LENgth*) indique le nombre d'éléments présents dans la file d'attente;
- le champ `lock` est un verrou rotatif utilisé lors de l'exécution d'opérations atomique sur la file d'attente. Dans ce cas, lors d'un accès critique, le verrou rotatif doit être libre ou attendu jusqu'à sa libération.

Chapitre 13

Gestion des tampons de socket

Nous avons vu au chapitre 12 la structure d'un descripteur de tampon de socket. Nous ne nous nous sommes pas occupés de l'aspect dynamique jusqu'à maintenant. C'est ce que nous allons faire maintenant.

Des fonctions de gestion des descripteurs de tampon de socket sont à disposition du noyau. Elles se classent en quatre catégories :

- *Génération, libération et duplication des descripteurs* : ces fonctions se chargent de l'ensemble de la gestion mémoire du tampon de socket et de son optimisation.
- *Manipulation des paramètres* au sein de la structure `sk_buff`, en particulier les opérations visant la modification de la zone de données des paquets.
- *Gestion des files d'attente* de descripteurs de tampon de socket.
- *Copie* des descripteurs de tampon et des tampons eux-mêmes.

Nous regroupons l'étude de ces fonctions dans ce chapitre car c'est leur place naturelle et que nous allons en utiliser certaines très rapidement. Mais on ne peut pas dire que leur étude soit passionnante. Cela n'a presque rien à voir avec les réseaux, si ce n'est leur utilisation, mais beaucoup plus avec la gestion de la mémoire. On pourra se contenter de parcourir rapidement ce chapitre et y revenir au fur et à mesure des besoins.

13.1 Génération et libération des descripteurs de tampon

Les opérations sur un descripteur de tampon de socket sont celles à lesquelles on peut s'attendre: allocation et libération.

13.1.1 Sources

Code Linux 2.6.10

L'aspect dynamique de l'antémémoire des sockets se trouve dans les fichiers `linux/include/linux/skbuff.h` (que nous avons déjà rencontré) et `linux/net/core/skbuff.c`:

```
1  /*
2   *      Routines ayant a voir avec les manipulations memoire de 'struct sk_buff'.
3   *
4   *      Auteurs :       Alan Cox <iiitac@pyr.swan.ac.uk>
5   *                      Florian La Roche <rzsfl@rz.uni-sb.de>
6   *
7   *      Version :       $Id: skbuff.c,v 1.90 2001/11/07 05:56:19 davem Exp $
8   *
9   *      Rectifications :
10  *              Alan Cox        :       A rectifie le pire des bogues de mise
11  *                                      d'aplomb.
12  *              Dave Platt      :       A rectifie la pile d'interruption.
13  *      Richard Kooijman        :       A rectifie les estampilles temporelles.
14  *              Alan Cox        :       A change le format des tampons.
15  *              Alan Cox        :       ancre de destruction pour AF_UNIX etc.
16  *              Linus Torvalds  :       Un meilleur skb_clone.
17  *              Alan Cox        :       A ajoute skb_copy.
18  *              Alan Cox        :       A ajoute toutes les routines changees que Linus
19  *                                      avait seulement placees dans les en-tetes
20  *              Ray VanTassle   :       A rectifie --skb->lock en free
21  *              Alan Cox        :       champ arp de copie skb_copy
22  *              Andi Kleen      :       A slabifie.
23  *              Robert Olsson   :       A retire skb_head_pool
24  *
25  *      NOTE :
26  *              Les routines __skb_ doivent etre appelees avec les interruptions
27  *      inhibees, sinon vous devez *reellement* vous assurer que l'operation est atomique
28  *      a l'egard de la liste en train d'etre traitee (par exemple via lock_sock()
29  *      ou en inhibant les routines des moities basses, etc).
[...]
37 /*
38  *      Les fonctions de ce fichier ne compileront pas correctement avec gcc 2.4.x
39  */
```

13.1.2 Allocation d'un tampon de socket

13.1.2.1 Description

La fonction:

```
struct sk_buff * alloc_skb(unsigned int size, int gfp_mask);
```

alloue un nouveau tampon de socket sans espace de tête, sans données et dont l'espace de queue a une taille de `size` octets (plus alignement jusqu'à la prochaine adresse 16 bits). Elle renvoie l'adresse du descripteur associé (ou la valeur `NULL` s'il y a un problème).

Les valeurs des drapeaux de `gfp_mask` sont `GFP_USER`, `GFP_KERNEL`, `GFP_ATOMIC` ou `GFP_DMA`.

13.1.2.2 Emplacement des descripteurs de tampon

Les descripteurs de tampon de socket forment une liste chaînée, comme nous l'avons vu au chapitre 12. Le début de cette liste est située à un emplacement de la mémoire vive réservée au noyau, repéré par la variable `skbuff_head_cache`. Celle-ci est définie au début du fichier `linux/net/core/skbuff.c`:

Code Linux 2.6.10

```
71 static kmem_cache_t *skbuff_head_cache;
```

13.1.2.3 Définition de la fonction

La fonction `alloc_skb()` est définie dans le fichier `linux/net/core/skbuff.c`:

Code Linux 2.6.10

```
110 /*      Alloue un nouveau skbuff. Nous faisons ceci nous-memes pour que nous puissions
111  *      renseigner quelques champs 'prives' et aussi faire des statistiques memoire pour
112  *      trouver toutes les fuites [BEEP].
113  *
114  */
115
116 /**
117  *      alloc_skb       -       alloue un tampon de reseau
118  *      @size : taille a allouer
119  *      @gfp_mask : masque d'allocation
120  *
121  *      Alloue un nouveau &sk_buff. Le tampon renvoye n'a pas d'espace de tete et un
122  *      espace de queue de size octets. L'objet a un compteur de reference de un.
123  *      La valeur renvoyee est le tampon. En cas d'echec, la valeur renvoyee est %NULL.
124  *
125  *      Les tampons peuvent etre alloues dans des interruptions seulement en utilisant
126  *      un @gfp_mask de %GFP_ATOMIC.
127  */
128 struct sk_buff *alloc_skb(unsigned int size, int gfp_mask)
129 {
130         struct sk_buff *skb;
131         u8 *data;
132
133         /* Obtenir la HEAD */
134         skb = kmem_cache_alloc(skbuff_head_cache,
135                                gfp_mask & ~__GFP_DMA);
136         if (!skb)
137                 goto out;
138
139         /* Obtenir les DATA. La taille doit concorder avec skb_add_mtu(). */
140         size = SKB_DATA_ALIGN(size);
141         data = kmalloc(size + sizeof(struct skb_shared_info), gfp_mask);
142         if (!data)
143                 goto nodata;
144
145         memset(skb, 0, offsetof(struct sk_buff, truesize));
146         skb->truesize = size + sizeof(struct sk_buff);
147         atomic_set(&skb->users, 1);
148         skb->head = data;
149         skb->data = data;
150         skb->tail = data;
151         skb->end  = data + size;
152
153         atomic_set(&(skb_shinfo(skb)->dataref), 1);
154         skb_shinfo(skb)->nr_frags  = 0;
155         skb_shinfo(skb)->tso_size = 0;
156         skb_shinfo(skb)->tso_segs = 0;
157         skb_shinfo(skb)->frag_list = NULL;
158 out:
```

```
159         return skb;
160 nodata:
161         kmem_cache_free(skbuff_head_cache, skb);
162         skb = NULL;
163         goto out;
164 }
```

Autrement dit :

- L'adresse `skb` à renvoyer est déclarée ainsi que celle `data` du tampon proprement dit.
- On essaie d'obtenir un emplacement en mémoire vive pouvant contenir le descripteur de tampon, au-delà du début indiqué. Si on n'y parvient pas, on renvoie `NULL`.
- On essaie d'obtenir de la place en mémoire vive pour le tampon proprement dit.
  - La taille devant être un multiple de 16, on ajoute un complément à `size` si nécessaire, en utilisant la macro auxiliaire `SKB_DATA_ALIGN()`.

    Code Linux 2.6.10

    Celle-ci est définie dans le fichier `linux/include/linux/skbuff.h`:

    ```
    40 #define SKB_DATA_ALIGN(X)       (((X) + (SMP_CACHE_BYTES - 1)) & \
    41                                   ~(SMP_CACHE_BYTES - 1))
    ```

  - On essaie d'obtenir de l'emplacement mémoire pour le tampon. Si on n'y parvient pas, on libère le descripteur de tampon obtenu ci-dessus et on renvoie `NULL`.
- On initialise l'emplacement mémoire du descripteur de tampon avec des zéros.
- On renseigne les champs indispensables du descripteur de tampon de socket à ce niveau, c'est-à-dire la taille effective, le nombre d'utilisateurs (égal à 1) et les quatre adresses du tampon.
- On renseigne les champs concernant la fragmentation : référencé une fois, taille effective du tableau des fragments nulle, de taille zéro, liste de fragmentation vide.

  La macro `skb_shinfo()` permet de se référer au champ `end` du descripteur (qui détient les informations sur la fragmentation) :

  Code Linux 2.6.10

  ```
  313 /* Interne */
  314 #define skb_shinfo(SKB)         ((struct skb_shared_info *)((SKB)->end))
  ```
- On renvoie l'adresse du descripteur de tampon.

13.1.3 Libération rapide d'un tampon de socket

13.1.3.1 Description

La fonction :

```
void kfree_skbmem(struct sk_buff *skb);
```

libère le tampon spécifié par la descripteur `skb` ainsi que le descripteur lui-même.

13.1.3.2 Définition

Code Linux 2.6.10

La fonction `kfree_skbmem()` est définie dans le fichier `linux/net/core/skbuff.c`:

```
205 /*
206  *      Libere un skbuff de la memoire sans nettoyer l'etat.
207  */
208 void kfree_skbmem(struct sk_buff *skb)
209 {
210         skb_release_data(skb);
211         kmem_cache_free(skbuff_head_cache, skb);
212 }
```

Autrement dit :

- le tampon est libéré, en utilisant la fonction auxiliaire `skb_release_data()` étudiée ci-après ;
- la partie de mémoire vive allouée au descripteur de tampon est également libérée.

13.1.3.3 Libération du tampon proprement dit

La fonction `skb_release_data()` est définie dans le fichier `linux/net/core/skbuff.c`: Code Linux 2.6.10

```
188 void skb_release_data(struct sk_buff *skb)
189 {
190         if (!skb->cloned ||
191             atomic_dec_and_test(&(skb_shinfo(skb)->dataref))) {
192                 if (skb_shinfo(skb)->nr_frags) {
193                         int i;
194                         for (i = 0; i < skb_shinfo(skb)->nr_frags; i++)
195                                 put_page(skb_shinfo(skb)->frags[i].page);
196                 }
197
198                 if (skb_shinfo(skb)->frag_list)
199                         skb_drop_fraglist(skb);
200
201                 kfree(skb->head);
202         }
203 }
```

Autrement dit si le descripteur n'est pas cloné ou si le nombre de références est nul :

- si le nombre de fragments n'est pas nul, on libère chacune des pages du tableau de fragments ;
- si la liste de fragmentation est non nulle, on la libère, en utilisant la fonction auxiliaire `skb_drop_fraglist()` étudiée ci-après ;
- on libère la tête de liste, en utilisant la fonction générale `kfree()`.

13.1.3.4 Libération de la liste de fragmentation

La fonction `skb_drop_fraglist()` est définie dans le fichier `linux/net/core/skbuff.c`: Code Linux 2.6.10

```
167 static void skb_drop_fraglist(struct sk_buff *skb)
168 {
169         struct sk_buff *list = skb_shinfo(skb)->frag_list;
170
171         skb_shinfo(skb)->frag_list = NULL;
172
173         do {
174                 struct sk_buff *this = list;
175                 list = list->next;
176                 kfree_skb(this);
177         } while (list);
178 }
```

autrement dit on libère chaque descripteur de la liste grâce à la fonction `kfree_skb()` décrite ci-après et on indique que la liste est vide.

13.1.4 Libération propre d'un tampon de socket

13.1.4.1 Description

La fonction :

```
void kfree_skb(struct sk_buff *skb);
```

libère le tampon et le descripteur de tampon, spécifiés par l'adresse `skb` du descripteur, proprement, c'est-à-dire en mettant à zéro les champs du descripteur de tampon. Elle vérifie en outre que le descripteur ne se trouve pas dans une file d'attente, sinon un message d'erreur est affiché.

13.1.4.2 Définition

Code Linux 2.6.10

La fonction `kfree_skb()` est définie dans le fichier en-tête `linux/include/linux/skbuff.h`:

```
340 /*
341  * Si users == 1, nous sommes le seul proprietaire et nous pouvons eviter les
342  * changements atomiques redondants.
343  */
344
345 /**
346  *      kfree_skb - libere un sk_buff
347  *      @skb : tampon a liberer
348  *
349  *      Ecarte une reference au tampon et le libere si le compteur d'utilisation a
350  *      atteint zero.
351  */
352 static inline void kfree_skb(struct sk_buff *skb)
353 {
354         if (atomic_read(&skb->users) == 1 || atomic_dec_and_test(&skb->users))
355                 __kfree_skb(skb);
356 }
```

Code Linux 2.6.10

qui renvoie essentiellement à la fonction `__kfree_skb()`, définie dans le fichier `linux/net/core/skbuff.c`:

```
214 /**
215  *      __kfree_skb - fonction privee
216  *      @skb : tampon
217  *
218  *      Libere un sk_buff. Relache toute chose attachee au tampon.
219  *      Nettoie l'etat. Ceci est une fonction d'aide interne. Les utilisateurs
220  *      doivent toujours appeler kfree_skb
221  */
222
223 void __kfree_skb(struct sk_buff *skb)
224 {
225         if (skb->list) {
226                 printk(KERN_WARNING "Warning: kfree_skb passed an skb still "
227                        "on a list (from %p).\n", NET_CALLER(skb));
228                 BUG();
229         }
230
231         dst_release(skb->dst);
232 #ifdef CONFIG_XFRM
233         secpath_put(skb->sp);
234 #endif
235         if(skb->destructor) {
236                 if (in_irq())
237                         printk(KERN_WARNING "Warning: kfree_skb on "
238                                             "hard IRQ %p\n", NET_CALLER(skb));
```

```
239                 skb->destructor(skb);
240         }
241 #ifdef CONFIG_NETFILTER
242         nf_conntrack_put(skb->nfct);
243 #ifdef CONFIG_BRIDGE_NETFILTER
244         nf_bridge_put(skb->nf_bridge);
245 #endif
246 #endif
247 /* XXX : Ceci EST-IL encore necessaire ? - JHS */
248 #ifdef CONFIG_NET_SCHED
249         skb->tc_index = 0;
250 #ifdef CONFIG_NET_CLS_ACT
251         skb->tc_verd = 0;
252         skb->tc_classid = 0;
253 #endif
254 #endif
255
256         kfree_skbmem(skb);
257 }
```

Autrement dit :

- Si le descripteur se trouve dans une file d'attente, on ne devrait pas libérer le descripteur de tampon. Le noyau affiche donc un message d'avertissement.

  La macro générale `BUG()` dépend de l'architecture. Pour les microprocesseurs Intel, elle est définie dans le fichier `linux/include/asm-i386/bug.h`: Code Linux 2.6.10

```
 6 /*
 7  * Dit a l'utilisateur qu'il y a un probleme.
 8  * Le fichier responsable ainsi que la ligne sont codes d'apres l'opcode "officiallement
 9  * indefini" pour l'analyse dans la routine.
10  */
11
12 #if 1   /* Positionne a zero pour un noyau legerement plus petit */
13 #define BUG()                           \
14  __asm__ __volatile__(  "ud2\n"         \
15                         "\t.word %c0\n" \
16                         "\t.long %c1\n" \
17                          : : "i" (__LINE__), "i" (__FILE__))
18 #else
19 #define BUG() __asm__ __volatile__("ud2\n")
20 #endif
```

- On libère l'entrée du cache de destination, grâce à la fonction `dst_release()` étudiée ci-après.
- Si une fonction de destruction est associée au descripteur de tampon, on y fait appel.
- On libère enfin le tampon et le descripteur de tampon grâce à la fonction `kfree_skbmem()` étudiée ci-dessus.

13.1.4.3 Libération d'une entrée de cache de destination

La fonction `dst_release()` est définie dans le fichier en-tête `linux/include/net/dst.h`: Code Linux 2.6.10

```
145 static inline
146 void dst_release(struct dst_entry * dst)
147 {
148         if (dst) {
149                 WARN_ON(atomic_read(&dst->__refcnt) < 1);
150                 atomic_dec(&dst->__refcnt);
151         }
152 }
```

Elle se contente de décrémenter le nombre d'utilisateurs de cette entrée.

La macro générale `WARN_ON()` dépend de l'architecture. Pour les microprocesseurs Intel, elle est définie dans le fichier `linux/include/asm-i386/bug.h`:

Code Linux 2.6.10

```
25 #ifndef HAVE_ARCH_WARN_ON
26 #define WARN_ON(condition) do { \
27         if (unlikely((condition)!=0)) { \
28                 printk("Badness in %s at %s:%d\n", __FUNCTION__, __FILE__, __LINE__); \
29                 dump_stack(); \
30         } \
31 } while (0)
32 #endif
```

Les macros générales `likely()` et `unlikely()` sont définies dans le fichier `linux/include/linux/compiler.h`:

Code Linux 2.6.10

```
54 /*
55  * Des macros generiques dependant du compilateur requises pour la construction du
56  * noyau en-dessous de ce commentaire. Les implementations specifiques a la version du
57  * compilateur/compilateur reel specifique proviennent des fichiers en-tete ci-dessus
58  */
59
60 #define likely(x)       __builtin_expect(!!(x), 1)
61 #define unlikely(x)     __builtin_expect(!!(x), 0)
```

13.1.4.4 Macro de libération

La macro `dev_kfree_skb()` se contente de renvoyer à la fonction étudiée ci-dessus.

Elle est définie dans le fichier `linux/include/linux/skbuff.h`:

Code Linux 2.6.10

```
307 #define dev_kfree_skb(a)        kfree_skb(a)
```

13.2 Manipulation du tampon

13.2.1 Espaces de tête et de queue

13.2.1.1 Taille de l'espace de queue

La fonction `skb_tailroom()` indique la taille de l'espace de queue. Elle est définie dans le fichier `linux/include/linux/skbuff.h`:

Code Linux 2.6.10

```
804 /**
805  *      skb_tailroom - octets a la fin du tampon
806  *      @skb : tampon a verifier
807  *
808  *      Renvoie le nombre d'octets de l'espace libre a la fin d'un sk_buff
809  */
810 static inline int skb_tailroom(const struct sk_buff *skb)
811 {
812         return skb_is_nonlinear(skb) ? 0 : skb->end - skb->tail;
813 }
```

La fonction en ligne `skb_is_nonlinear()` est définie un peu plus haut dans le même fichier :

Code Linux 2.6.10

```
655 static inline int skb_is_nonlinear(const struct sk_buff *skb)
656 {
657         return skb->data_len;
658 }
```

13.2.1.2 Taille de l'espace de tête

La fonction `skb_headroom()` indique la taille de l'espace de tête. Elle est définie dans le fichier `linux/include/linux/skbuff.h`: Code Linux 2.6.10

```
793 /**
794  *      skb_headroom - octets en tete du tampon
795  *      @skb : tampon a verifier
796  *
797  *      Renvoie le nombre d'octets de l'espace libre en tete d'un &sk_buff.
798  */
799 static inline int skb_headroom(const struct sk_buff *skb)
800 {
801         return skb->data - skb->head;
802 }
```

13.2.2 Ajustement de la taille d'un tampon vide

13.2.2.1 Ajustement de la taille de l'espace de tête

La fonction `skb_reserve(len)` ajoute `len` octets à l'espace de tête, pris sur l'espace de queue. Cette fonction n'est bien sûr appropriée que s'il n'existe pas de données dans la zone actuelle, mais seulement si sa taille initiale doit être corrigée.

Elle est définie dans le fichier `linux/include/linux/skbuff.h`: Code Linux 2.6.10

```
815 /**
816  *      skb_reserve - ajuste l'espace de tete
817  *      @skb : tampon a alterer
818  *      @len : octets a deplacer
819  *
820  *      Accroit l'espace de tete d'un &sk_buff vide en reduisant l'espace
821  *      de queue. Ceci est seulement permis pour un tampon vide.
822  */
823 static inline void skb_reserve(struct sk_buff *skb, unsigned int len)
824 {
825         skb->data += len;
826         skb->tail += len;
827 }
```

13.2.2.2 Ajout à la fin de la zone des données

La fonction `skb_put(skb, len)` permet d'augmenter la zone des données à partir de la fin de celle-ci au dépend de l'espace de queue. Ceci se produit rarement car la majorité des protocoles utilisent un en-tête mais rarement un en-queue. Seuls les pointeurs sont positionnés en conséquence; l'utilisateur est responsable de la copie des données elles-mêmes. Avant l'appel de `skb_put()`, il faut vérifier que la taille de l'espace de queue est encore suffisant, sinon le noyau affiche un message d'erreur.

Elle est définie dans le fichier `linux/include/linux/skbuff.h`: Code Linux 2.6.10

```
701 /**
702  *      skb_put - ajoute des donnees a un tampon
703  *      @skb : tampon a utiliser
704  *      @len : quantite de donnees a ajouter
705  *
706  *      Cette fonction etend la zone des donnees du tampon. Si ceci conduit a
707  *      depasser la taille totale du tampon, le noyau panique. Un pointeur sur le
708  *      premier octet de la zone supplementaire est renvoye.
709  */
710 static inline unsigned char *skb_put(struct sk_buff *skb, unsigned int len)
```

```
711 {
712         unsigned char *tmp = skb->tail;
713         SKB_LINEAR_ASSERT(skb);
714         skb->tail += len;
715         skb->len  += len;
716         if (unlikely(skb->tail>skb->end))
717                 skb_over_panic(skb, len, current_text_addr());
718         return tmp;
719 }
```

Code Linux 2.6.10

La macro `SKB_LINEAR_ASSERT()` est définie un peu plus haut dans le même fichier :

```
687 #define SKB_LINEAR_ASSERT(skb)  BUG_ON(skb_is_nonlinear(skb))
```

Code Linux 2.6.10

La définition de la macro générale `BUG_ON()` est définie dans le fichier `linux/include/asm-generic/bug.h`:

```
21 #ifndef HAVE_ARCH_BUG_ON
22 #define BUG_ON(condition) do { if (unlikely((condition)!=0)) BUG(); } while(0)
23 #endif
```

Code Linux 2.6.10

La fonction `skb_over_panic()`, qui gèle le système et affiche un message d'erreur à l'écran, est définie dans le fichier `linux/net/core/skbuff.c`:

```
79 /**
80  *      skb_over_panic  -       fonction privee
81  *      @skb : tampon
82  *      @sz : taille
83  *      @here : adresse
84  *
85  *      Code de support non en ligne pour skb_put(). Non appelable par un utilisateur.
86  */
87 void skb_over_panic(struct sk_buff *skb, int sz, void *here)
88 {
89         printk(KERN_INFO "skput:over: %p:%d put:%d dev:%s",
90                here, skb->len, sz, skb->dev ? skb->dev->name : "<NULL>");
91         BUG();
92 }
```

Code Linux 2.6.10

La fonction générale `current_text_addr()` dépend de l'architecture. Elle est définie dans le fichier `linux/include/asm-i386/processor.h` pour les microprocesseurs Intel :

```
36 /*
37  * Implementation par defaut de la macro qui renvoie le pointeur d'instruction
38  * en cours ("compteur de programmer").
39  */
40 #define current_text_addr() ({ void *pc; __asm__("movl $1f,%0\n1:":"=g" (pc)); pc; })
```

13.2.2.3 Ajout en début de la zone des données

La fonction `skb_push()` opère comme la fonction précédente mais en déplaçant le début de la zone des données, au dépend de l'espace de tête. La taille de l'espace de tête est vérifiée au préalable.

Code Linux 2.6.10

Elle est définie dans le fichier `linux/include/linux/skbuff.h`:

```
728 /**
729  *      skb_push - ajoute des donnees au debut d'un tampon
730  *      @skb : tampon a utiliser
731  *      @len : quantite de donnees a ajouter
732  *
733  *      Cette fonction etend la zone des donnees du tampon au debut du
734  *      tampon. Si ceci conduit a depasser la taille totale de l'espace de tete du tampon,
```

```
735  *      le noyau panique. Un pointeur sur le premier octet des donnees supplementaires
             est renvoye.
736  */
737 static inline unsigned char *skb_push(struct sk_buff *skb, unsigned int len)
738 {
739         skb->data -= len;
740         skb->len  += len;
741         if (unlikely(skb->data<skb->head))
742                 skb_under_panic(skb, len, current_text_addr());
743         return skb->data;
744 }
```

La fonction `skb_under_panic()` est définie dans le fichier `linux/net/core/skbuff.c`: Code Linux 2.6.10

```
94  /**
95   *      skb_under_panic -       fonction privee
96   *      @skb : tampon
97   *      @sz : taille
98   *      @here : adresse
99   *
100  *      Code de support hors ligne pour skb_push(). Non appelable par un utilisateur.
101  */
102
103 void skb_under_panic(struct sk_buff *skb, int sz, void *here)
104 {
105         printk(KERN_INFO "skput:under: %p:%d put:%d dev:%s",
106                here, skb->len, sz, skb->dev ? skb->dev->name : "<NULL>");
107         BUG();
108 }
```

13.2.2.4 Retrait de données au début

La fonction `skb_pull(skb, len)` tronque de `len` octets le début de la zone des données d'un tampon au profit de l'espace de tête, après avoir vérifié qu'on demeurera après ceci dans la zone de mémoire affectée au tampon. Elle est définie dans le fichier `linux/include/linux/skbuff.h`: Code Linux 2.6.10

```
746 static inline unsigned char *__skb_pull(struct sk_buff *skb, unsigned int len)
747 {
748         skb->len -= len;
749         BUG_ON(skb->len < skb->data_len);
750         return skb->data += len;
751 }
752
753 /**
754  *      skb_pull - retire des donnees au debut d'un tampon
883  *      @skb : tampon a utiliser
884  *      @len : quantite de donnees a retirer
757  *
758  *      Cette fonction efface le debut des donnees d'un tampon, les renvoyant a
759  *      l'espace de tete. Un pointeur a la donnee suivante dans le tampon
760  *      est renvoye. Une fois que les donnees ont ete retirees, les placements suivants
761  *      surchargeront les anciennes donnees.
762  */
763 static inline unsigned char *skb_pull(struct sk_buff *skb, unsigned int len)
764 {
765         return unlikely(len > skb->len) ? NULL : __skb_pull(skb, len);
766 }
```

13.2.3 Manipulation du compteur de références

13.2.3.1 Incrémentation du nombre de références

Code Linux 2.6.10

La fonction `skb_get()` augmente de un le nombre de références au descripteur de tampon. Elle est définie dans le fichier `linux/include/linux/skbuff.h`:

```
327 /**
328  *      skb_get - reference un tampon
329  *      @skb : tampon a referencer
330  *
331  *      Ajoute une autre reference a un tampon de socket et renvoie un pointeur
332  *      au tampon.
333  */
334 static inline struct sk_buff *skb_get(struct sk_buff *skb)
335 {
336         atomic_inc(&skb->users);
337         return skb;
338 }
```

13.2.3.2 Détection de la multiplicité des références

Code Linux 2.6.10

La fonction `skb_shared()` vérifie si le descripteur de tampon possède plusieurs références ou une seule. Elle est définie dans le fichier `linux/include/linux/skbuff.h`:

```
378 /**
379  *      skb_shared - le tampon est-il partage ?
380  *      @skb : tampon a verifier
381  *
382  *      Renvoie vrai si plus d'une personne fait reference a ce
383  *      tampon.
384  */
385 static inline int skb_shared(const struct sk_buff *skb)
386 {
387         return atomic_read(&skb->users) != 1;
388 }
```

Remarquons que le code n'est correct que si le compteur est non nul, mais ceci est le cas puisque celui qui pose la question fait référence au descripteur de tampon.

13.2.4 Détection de clonage

13.2.4.1 Disponibilité du clone

Code Linux 2.6.10

La fonction `skb_cloned()` indique si le descripteur de tampon est un clone et si le tampon associé est disponible. Elle est définie dans le fichier `linux/include/linux/skbuff.h`:

```
365 /**
366  *      skb_cloned - le tampon est-il un clone ?
367  *      @skb : tampon a verifier
368  *
369  *      Renvoie vrai si le tampon a ete genere avec skb_clone() et est
370  *      une des copies partagees du tampon. Les tampons clones sont
371  *      des donnes partagees, aussi ne doit-on pas ecrire dessus en circonstances normales.
372  */
373 static inline int skb_cloned(const struct sk_buff *skb)
374 {
375         return skb->cloned && atomic_read(&skb_shinfo(skb)->dataref) != 1;
376 }
```

13.2.4.2 Déclonage

La fonction `skb_unshare()` vérifie si le descripteur de tampon est un clone. Si c'est le cas, une copie du tampon est effectuée que l'on associe à ce clone, dont le compteur de renvois prend la valeur un. Le compteur de renvoi de l'ancien tampon est décrémenté. Elle est définie dans le fichier `linux/include/linux/skbuff.h`:

Code Linux 2.6.10

```
414 /*
415  *      Copie des tampons partages dans un nouveau sk_buff. Nous faisons effectivement COW sur
416  *      les paquets pour traiter les cas ou nous avons une ecriture locale et une
417  *      retransmission et d'autres trucs du meme genre. Le cas usuel est le vidage tcp
418  *      d'un paquet en train d'etre retransmis.
419  */
420
421 /**
422  *      skb_unshare - fait une copie d'un tampon partage
423  *      @skb : tampon a verifier
424  *      @pri : priorite pour l'allocation de la memoire
425  *
426  *      Si le tampon de socket est un clone, cette fonction cree une nouvelle
427  *      copie des donnees, decremente le compteur de reference de l'ancienne copie et
428  *      renvoie la nouvelle copie avec un compteur de reference a 1. Si le tampon
429  *      n'est pas un clone, le tampon originel est renvoye. Si elle est appelee avec un verrou
430  *      tournant depuis une interruption, l'etat @pri doit etre %GFP_ATOMIC
431  *
432  *      %NULL est renvoye en cas d'echec d'allocation memoire.
433  */
434 static inline struct sk_buff *skb_unshare(struct sk_buff *skb, int pri)
435 {
436         might_sleep_if(pri & __GFP_WAIT);
437         if (skb_cloned(skb)) {
438                 struct sk_buff *nskb = skb_copy(skb, pri);
439                 kfree_skb(skb); /* Libere notre copie partagee */
440                 skb = nskb;
441         }
442         return skb;
443 }
```

La fonction `skb_copy()` sera étudiée dans une section ultérieure.

13.2.4.3 Clonage

La fonction `skb_share_check()` vérifie si le tampon est partagé. Si c'est le cas, il est cloné. Elle est définie dans le fichier `linux/include/linux/skbuff.h`:

Code Linux 2.6.10

```
390 /**
391  *      skb_share_check - verifie si un tampon est partage et le clone s'il l'est
392  *      @skb : tampon a verifier
393  *      @pri : priorite pour l'allocation memoire
394  *
395  *      Si le tampon est partage, le tampon est clone et la reference a l'ancienne
396  *      copie est ecartee. Un nouveau clone avec une reference de 1 est renvoyee.
397  *      Si le tampon n'est pas partage, le tampon originel est renvoye. Si elle est
398  *      appelee avec un statut d'interruption ou avec un verrou tournant detenu,
399  *      pri doit etre GFP_ATOMIC.
400  *
401  *      NULL est renvoye en cas d'echec d'allocation memoire.
402  */
403 static inline struct sk_buff *skb_share_check(struct sk_buff *skb, int pri)
404 {
405         might_sleep_if(pri & __GFP_WAIT);
406         if (skb_shared(skb)) {
```

```
407                 struct sk_buff *nskb = skb_clone(skb, pri);
408                 kfree_skb(skb);
409                 skb = nskb;
410         }
411         return skb;
412 }
```

La fonction `skb_clone()` sera également étudiée dans une section ultérieure.

13.3 Gestion des files d'attente de paquets

Nous allons décrire les méthodes destinées à la gestion des descripteurs de tampon de socket dans les files d'attente.

13.3.1 Caractères généraux

13.3.1.1 Atomicité des opérations sur les tampons de socket

La plupart des opérations sur les tampons de socket sont mises en œuvre durant les phases critiques ou peuvent être interrompues par des opérations de priorité supérieure (traitement d'interruption, interruption logicielle, etc.). Un chaînage incorrect pourrait se produire par exemple en raison d'opérations successives imbriquées les unes dans les autres de manière complexe sur une file d'attente. Il en résulterait inévitablement une erreur d'accès à la mémoire puis un blocage du système. Cela explique pourquoi le traitement des paquets doit avoir lieu atomiquement.

Il en résulte certes un coût supplémentaire parce qu'il faut implémenter des mécanismes particuliers comme les verrous rotatifs ou les sémaphores pour être en mesure d'obtenir des états mieux sécurisés. C'est toutefois la seule façon de garantir qu'il n'existe aucun état incohérent susceptible de mettre en péril la stabilité du sous-système réseau et, par conséquent, celle du système d'exploitation. Sous Linux, la sécurité et la stabilité de l'ensemble du système d'exploitation prennent toujours le pas sur la performance et les bancs de test.

13.3.1.2 Opérations *inline*

La plupart des opérations sur les files d'attente sont définies en tant que procédure *inline*. La définition *inline* signifie qu'elles ne constituent pas des procédures à proprement parler, mais qu'elles sont insérées comme une macro dans le tronc de la fonction émettrice. On évite ainsi le temps système d'un appel de fonction, moyennant la prise en compte d'un noyau légèrement plus volumineux. Le rôle de fonction préserve malgré tout la clarté et la maintenabilité du code source.

13.3.2 Gestion des structures de files d'attente

13.3.2.1 Initialisation

La fonction `skb_queue_head_init(list)` initialise une instance de la structure `skb_queue_head` de façon à pouvoir l'utiliser en tant que file d'attente. Les pointeurs sont positionnés sur la structure et la longueur définie à zéro : dans une file d'attente vide, `next` et `prev` pointent sur la file d'attente `list` et non sur `NULL`.

Code Linux 2.6.10

Elle est définie dans le fichier `linux/include/linux/skbuff.h`:

```
498 static inline void skb_queue_head_init(struct sk_buff_head *list)
499 {
500         spin_lock_init(&list->lock);
```

```
501         list->prev = list->next = (struct sk_buff *)list;
502         list->qlen = 0;
503 }
```

13.3.2.2 Test de vacuité

La fonction `skb_queue_empty(list)` vérifie si la file d'attente `list` est vide ou si elle contient encore un descripteur de tampon. Le champ `list->qlen` est renvoyé par souci de facilité.

Elle est définie dans le fichier `linux/include/linux/skbuff.h`: Code Linux 2.6.10

```
316 /**
317  *      skb_queue_empty - verifie si la file d'attente est vide
318  *      @list : tete de file d'attente
319  *
320  *      Renvoie vrai si la file d'attente est vide, faux sinon.
321  */
322 static inline int skb_queue_empty(const struct sk_buff_head *list)
323 {
324         return list->next == (struct sk_buff *)list;
325 }
```

13.3.2.3 Longueur de la file d'attente

La fonction `skb_queue_len()` renvoie la longueur de la file d'attente passée en paramètre.

Elle est définie dans le fichier `linux/include/linux/skbuff.h`: Code Linux 2.6.10

```
487 /**
488  *      skb_queue_len   - obtient la longueur de la file d'attente
489  *      @list_ : liste a mesurer
490  *
491  *      Renvoie la longueur d'une file d'attente &sk_buff.
492  */
493 static inline __u32 skb_queue_len(const struct sk_buff_head *list_)
494 {
495         return list_->qlen;
496 }
```

13.3.3 Gestion des tampons d'une file d'attente

Pour les fonctions suivantes, il existe deux versions différentes : lorsqu'une fonction a déjà mis une interruption hors circuit, ceci n'a plus à être effectué dans la fonction concernant les files d'attente et peut être évité. Les fonctions sans interruption verrouillée se reconnaissent grâce à deux caractères de soulignement en préfixe, par exemple `__skb_dequeue()`.

13.3.3.1 Ajout d'un élément au début

La fonction `skb_queue_head(list, skb)` permet d'ajouter le descripteur de tampon d'adresse `skb` au début de la file d'attente `list`.

Elle est définie dans le fichier `linux/net/core/skbuff.c`: Code Linux 2.6.10

```
1235 /**
1236  *      skb_queue_head -  met un tampon en file d'attente en debut de liste
1237  *      @list : liste a utiliser
1238  *      @newsk : tampon a mettre en file d'attente
1239  *
1240  *      Met un tampon en file d'attente au debut de la liste. Cette fonction tire le
1241  *      verrou de la liste et peut donc etre utilisee de facon sure avec d'autres
1242  *      fonctions de de &sk_buff verrouillant de facon sure.
1243  *
```

```
1244  *      Un tampon ne peut pas etre place dans deux listes en meme temps.
1245  */
1246 void skb_queue_head(struct sk_buff_head *list, struct sk_buff *newsk)
1247 {
1248         unsigned long flags;
1249
1250         spin_lock_irqsave(&list->lock, flags);
1251         __skb_queue_head(list, newsk);
1252         spin_unlock_irqrestore(&list->lock, flags);
1253 }
```

Code Linux 2.6.10

Elle fait appel à la fonction `__skb_queue_head()` définie dans le fichier `linux/include/linux/skbuff.h`:

```
505 /*
506  *      Insere un sk_buff au debut d'une liste.
507  *
508  *      Les fonctions "__skb_xxxx()" sont celles qui sont non atomiques et qui
509  *      peuvent seulement etre appelees avec les interruptions inhibees.
510  */
511
512 /**
513  *      __skb_queue_head - met en tete d'une file d'attente un tampon
514  *      @list : liste a utiliser
515  *      @newsk : tampon a mettre en file d'attente
516  *
517  *      Met en file d'attente un tampon au debut d'une liste. Cette fonction ne tire aucun
518  *      verrou, vous devez donc detenir les verrous requis avant de l'appeler.
519  *
520  *      Un tampon ne peut pas etre place dans deux listes en meme temps.
521  */
522 extern void skb_queue_head(struct sk_buff_head *list, struct sk_buff *newsk);
523 static inline void __skb_queue_head(struct sk_buff_head *list,
524                                     struct sk_buff *newsk)
525 {
526         struct sk_buff *prev, *next;
527
528         newsk->list = list;
529         list->qlen++;
530         prev = (struct sk_buff *)list;
531         next = prev->next;
532         newsk->next = next;
533         newsk->prev = prev;
534         next->prev  = prev->next = newsk;
535 }
```

On utilise l'algorithme standard d'insertion dans une liste.

13.3.3.2 Ajout d'un élément à la fin

La fonction `skb_queue_tail(list, skb)` permet d'ajouter le descripteur de tampon d'adresse `skb` à la fin de la file d'attente `list`.

Code Linux 2.6.10

Elle est définie dans le fichier `linux/net/core/skbuff.c`:

```
1255 /**
1256  *      skb_queue_tail - met un tampon en file d'attente a la fin d'une liste
1257  *      @list : liste a utiliser
1258  *      @newsk : tampon a mettre en file d'attente
1259  *
1260  *      Met un tampon en file d'attente a la fin de la liste. Cette fonction tire le
1261  *      verrou de la liste et peut donc etre utilisee de facon sure avec d'autres
1262  *      fonctions de &sk_buff verrouillant de facon sure.
```

```
1263  *
1264  *      Un tampon ne peut pas etre place dans deux listes en meme temps.
1265  */
1266 void skb_queue_tail(struct sk_buff_head *list, struct sk_buff *newsk)
1267 {
1268         unsigned long flags;
1269
1270         spin_lock_irqsave(&list->lock, flags);
1271         __skb_queue_tail(list, newsk);
1272         spin_unlock_irqrestore(&list->lock, flags);
1273 }
```

Elle fait appel à la fonction `__skb_queue_tail()` définie dans le fichier `linux/include/linux/skbuff.h`:

Code Linux 2.6.10

```
537 /**
538  *      __skb_queue_tail - met un tampon en file d'attente a la fin de la liste
539  *      @list : liste a utiliser
540  *      @newsk : tampon a placer dans la file d'attente
541  *
542  *      Met un tampon en file d'attente a la fin d'une liste. Cette fonction ne tire aucun
543  *      verrou, vous devez donc detenir les verrous requis avant de l'appeler.
544  *
545  *      Un tampon ne peut pas etre place dans deux listes en meme temps.
546  */
547 extern void skb_queue_tail(struct sk_buff_head *list, struct sk_buff *newsk);
548 static inline void __skb_queue_tail(struct sk_buff_head *list,
549                                     struct sk_buff *newsk)
550 {
551         struct sk_buff *prev, *next;
552
553         newsk->list = list;
554         list->qlen++;
555         next = (struct sk_buff *)list;
556         prev = next->prev;
557         newsk->next = next;
558         newsk->prev = prev;
559         next->prev  = prev->next = newsk;
560 }
```

13.3.3.3 Retrait d'un élément au début

La fonction `skb_dequeue(list)` permet de récupérer le premier descripteur de tampon élément de la file d'attente. Elle renvoie l'adresse de celui-ci ou, si aucun élément ne se trouve dans la file d'attente, le pointeur `NULL`.

Elle est définie dans le fichier `linux/net/core/skbuff.c`:

Code Linux 2.6.10

```
1181 /**
1182  *      skb_dequeue - retire de la tete d'une file d'attente
1183  *      @list : liste de laquelle il faut retirer
1184  *
1185  *      Retire la tete de la liste. Le verrou de la liste est tire, donc la fonction
1186  *      peut etre utilisee de facon sure avec d'autres fonctions de liste qui verrouillent.
1187  *      L'item de tete est renvoye, ou %NULL si la liste est vide.
1188  */
1189
1190 struct sk_buff *skb_dequeue(struct sk_buff_head *list)
1191 {
1192         unsigned long flags;
1193         struct sk_buff *result;
1194
1195         spin_lock_irqsave(&list->lock, flags);
```

```
1196         result = __skb_dequeue(list);
1197         spin_unlock_irqrestore(&list->lock, flags);
1198         return result;
1199 }
```

Elle fait appel à la fonction `__skb_dequeue()` définie dans le fichier `linux/include/linux/skbuff.h`:

Code Linux 2.6.10

```
563 /**
564  *      __skb_dequeue - retire de la tete de la file d'attente
565  *      @list : liste de laquelle il faut retirer
566  *
567  *      Retire la tete de la liste. Cette fonction ne tire aucun verrou,
568  *      elle doit donc etre utilisee uniquement avec les verrous appropries detenus.
569  *      L'item de tete est renvoye, ou %NULL si la liste est vide.
570  */
571 extern struct sk_buff *skb_dequeue(struct sk_buff_head *list);
572 static inline struct sk_buff *__skb_dequeue(struct sk_buff_head *list)
573 {
574         struct sk_buff *next, *prev, *result;
575
576         prev = (struct sk_buff *) list;
577         next = prev->next;
578         result = NULL;
579         if (next != prev) {
580                 result       = next;
581                 next         = next->next;
582                 list->qlen--;
583                 next->prev   = prev;
584                 prev->next   = next;
585                 result->next = result->prev = NULL;
586                 result->list = NULL;
587         }
588         return result;
589 }
```

13.3.3.4 Retrait d'un élément à la fin

La fonction `skb_dequeue_tail(list)` permet de retirer le dernier descripteur de tampon élément de la file d'attente passée en argument. Elle renvoie l'adresse de celui-ci ou, si aucun élément ne se trouve dans la file d'attente, le pointeur `NULL`.

Elle est définie dans le fichier `linux/net/core/skbuff.c`:

Code Linux 2.6.10

```
1201 /**
1202  *      skb_dequeue_tail - retire a la fin de la file d'attente
1203  *      @list : liste de laquelle retirer
1204  *
1205  *      Retire la queue de la liste. Le verrou de la liste est tire, la fonction
1206  *      peut donc etre utilisee de facon sure avec d'autres fonctions de liste qui
1207  *      verrouillent. L'item de queue est renvoye, ou %NULL si la liste est vide.
1208  */
1209 struct sk_buff *skb_dequeue_tail(struct sk_buff_head *list)
1210 {
1211         unsigned long flags;
1212         struct sk_buff *result;
1213
1214         spin_lock_irqsave(&list->lock, flags);
1215         result = __skb_dequeue_tail(list);
1216         spin_unlock_irqrestore(&list->lock, flags);
1217         return result;
1218 }
```

Elle fait appel à la fonction `__skb_dequeue_tail()` définie dans le fichier `linux/include/linux/skbuff.h`:

Code Linux 2.6.10

```
637 /**
638  *      __skb_dequeue_tail - retire a la fin de la file d'attente
639  *      @list : liste de laquelle retirer
640  *
641  *      Retire la fin de la liste. Cette fonction ne tire aucun verrou,
642  *      elle ne doit donc etre utilisee qu'avec les verrous appropries detenus.
643  *      L'item de queue est renvoye ou %NULL si la liste est vide.
644  */
645 extern struct sk_buff *skb_dequeue_tail(struct sk_buff_head *list);
646 static inline struct sk_buff *__skb_dequeue_tail(struct sk_buff_head *list)
647 {
648         struct sk_buff *skb = skb_peek_tail(list);
649         if (skb)
650                 __skb_unlink(skb, list);
651         return skb;
652 }
```

qui renvoie à la fonction `__skb_unlink()` définie un peu plus haut dans le même fichier:

Code Linux 2.6.10

```
616 /*
617  * retire un sk_buff d'une liste. _Doit_ etre appelee atomiquement, et avec
618  * la liste connue...
619  */
620 extern void        skb_unlink(struct sk_buff *skb);
621 static inline void __skb_unlink(struct sk_buff *skb, struct sk_buff_head *list)
622 {
623         struct sk_buff *next, *prev;
624
625         list->qlen--;
626         next       = skb->next;
627         prev       = skb->prev;
628         skb->next  = skb->prev = NULL;
629         skb->list  = NULL;
630         next->prev = prev;
631         prev->next = next;
632 }
```

13.3.3.5 Vidage d'une file d'attente

La fonction `skb_queue_purge(list)` permet de vider la file d'attente passée en argument. Elle est définie dans le fichier `linux/net/core/skbuff.c`:

Code Linux 2.6.10

```
1220 /**
1221  *      skb_queue_purge - vide une liste
1222  *      @list : liste a vider
1223  *
1224  *      Enleve tous les tampons d'une liste &sk_buff. Chaque tampon est enleve de
1225  *      la liste en ecartant une reference. Cette fonction tire le verrou de la
1226  *      liste et est donc atomique a l'egard des autres fonctions de liste qui verrouillent.
1227  */
1228 void skb_queue_purge(struct sk_buff_head *list)
1229 {
1230         struct sk_buff *skb;
1231         while ((skb = skb_dequeue(list)) != NULL)
1232                 kfree_skb(skb);
1233 }
```

La fonction `__skb_queue_purge()`, quant à elle, est définie dans le fichier `linux/include/linux/skbuff.h`:

Code Linux 2.6.10

```
910 /**
911  *      __skb_queue_purge - vide une liste
912  *      @list : liste a vider
913  *
914  *      Enleve tous les tampons d'une liste &sk_buff. Chaque tampon est enleve de
915  *      la liste en ecartant une reference. Cette fonction ne tire pas le verrou de
916  *      la liste, aussi l'appelant doit-il detenir les verrous adequats pour l'utiliser.
917  */
918 extern void skb_queue_purge(struct sk_buff_head *list);
919 static inline void __skb_queue_purge(struct sk_buff_head *list)
920 {
921         struct sk_buff *skb;
922         while ((skb = __skb_dequeue(list)) != NULL)
923                 kfree_skb(skb);
924 }
```

13.3.3.6 Insertion d'un élément avant un autre

La fonction `skb_insert(oldskb, newskb)` permet d'insérer le descripteur de tampon d'adresse `newskb` avant le descripteur de tampon d'adresse `oldskb` dans la file d'attente à laquelle appartient `*oldskb`. Rappelons qu'un descripteur de tampon ne peut se trouver que dans une seule file d'attente à la fois et que celle-ci fait l'objet d'un des champs du descripteur.

Elle est définie dans le fichier `linux/net/core/skbuff.c`:

Code Linux 2.6.10

```
1321 /**
1322  *      skb_insert      -       insere un tampon
1323  *      @old : tampon devant lequel inserer
1324  *      @newsk : tampon a inserer
1325  *
1326  *      Place un paquet avant un paquet donne dans une liste. Les verrous de la liste sont
1327  *      tires, donc cette fonction est atomique a l'egard des autres appels verrouilles a la
1328  *      liste. Un tampon ne peut pas etre place dans deux listes en meme temps.
1329  */
1330
1331 void skb_insert(struct sk_buff *old, struct sk_buff *newsk)
1332 {
1333         unsigned long flags;
1334
1335         spin_lock_irqsave(&old->list->lock, flags);
1336         __skb_insert(newsk, old->prev, old, old->list);
1337         spin_unlock_irqrestore(&old->list->lock, flags);
1338 }
```

Elle fait appel à la fonction `__skb_insert()` définie dans le fichier `linux/include/linux/skbuff.h`:

Code Linux 2.6.10

```
592 /*
593  *      Insere un paquet dans une liste.
594  */
595 extern void        skb_insert(struct sk_buff *old, struct sk_buff *newsk);
596 static inline void __skb_insert(struct sk_buff *newsk,
597                                 struct sk_buff *prev, struct sk_buff *next,
598                                 struct sk_buff_head *list)
599 {
600         newsk->next = next;
601         newsk->prev = prev;
602         next->prev  = prev->next = newsk;
603         newsk->list = list;
```

```
604         list->qlen++;
605 }
```

13.3.3.7 Insertion d'un élément après un autre

La fonction skb_append(oldskb, newskb) permet d'insérer le descripteur de tampon d'adresse newskb après le descripteur de tampon d'adresse oldskb dans la file d'attente à laquelle appartient *oldskb.

Elle est définie dans le fichier linux/net/core/skbuff.c: Code Linux 2.6.10

```
1301 /**
1302  *      skb_append      -       place un tampon apres un autre
1303  *      @old : tampon apres lequel inserer
1304  *      @newsk : tampon a inserer
1305  *
1306  *      Place un paquet apres un paquet donne dans une liste. Les verrous de la liste sont
1307  *      tires, donc cette fonction est atomique a l'egard des autres appels verrouilles de
1308  *      liste. Un tampon ne peut pas etre place dans deux listes en meme temps.
1309  */
1310
1311 void skb_append(struct sk_buff *old, struct sk_buff *newsk)
1312 {
1313         unsigned long flags;
1314
1315         spin_lock_irqsave(&old->list->lock, flags);
1316         __skb_append(old, newsk);
1317         spin_unlock_irqrestore(&old->list->lock, flags);
1318 }
```

Elle fait appel à la fonction __skb_append() définie dans le fichier linux/include/linux/-skbuff.h: Code Linux 2.6.10

```
607 /*
608  *      Place un paquet apres un paquet donne dans une liste.
609  */
610 extern void        skb_append(struct sk_buff *old, struct sk_buff *newsk);
611 static inline void __skb_append(struct sk_buff *old, struct sk_buff *newsk)
612 {
613         __skb_insert(newsk, old, old->next, old->list);
614 }
```

13.3.3.8 Retrait dans une file d'attente

La fonction skb_unlink(skb) permet de retirer le descripteur de tampon dont l'adresse est passée en paramètre de la file d'attente à laquelle il appartient.

Elle est définie dans le fichier linux/net/core/skbuff.c: Code Linux 2.6.10

```
1274 /**
1275  *      skb_unlink      -       retire un tampon d'une liste
1276  *      @skb : tampon a retirer
1277  *
1278  *      [Retire un paquet d'une liste]. Les verrous de la liste sont tires,
1279  *      donc cette fonction est atomique a l'egard d'autres appels verrouilles de liste.
1280  *
1281  *      Marche meme sans connaitre la liste dans laquelle il se trouve, qui doit etre
1282  *      accessible. Ceci signifie aussi que LA LISTE DOIT EXISTER lorsque vous
1283  *      retirez. Ainsi une liste doit avoir son contenu non lie avant qu'il ne soit
1284  *      detruit.
1285  */
1286 void skb_unlink(struct sk_buff *skb)
1287 {
```

```
1288         struct sk_buff_head *list = skb->list;
1289
1290         if (list) {
1291                 unsigned long flags;
1292
1293                 spin_lock_irqsave(&list->lock, flags);
1294                 if (skb->list == list)
1295                         __skb_unlink(skb, skb->list);
1296                 spin_unlock_irqrestore(&list->lock, flags);
1297         }
1298 }
```

13.3.3.9 Pointage sur le premier élément d'une liste

La fonction `skb_peek(list)` renvoie un pointeur sur le premier élement de la file d'attente passée en paramètre, si celle-ci n'est pas vide. Sinon `NULL` est renvoyé. L'élément n'est pas retiré de la file d'attente.

Code Linux 2.6.10

Elle est définie dans le fichier `linux/include/linux/skbuff.h`:

```
445 /**
446  *      skb_peek
447  *      @list_ :  liste sur laquelle pointer
448  *
449  *      Pointe sur un &sk_buff. Contrairement a beaucoup d'autres operations, vous _DEVEZ_
450  *      etre soigneux avec celle-ci. Pointer laisse le tampon dans la
451  *      liste et quelqu'un d'autre peut se l'approprier. Vous devez detenir
452  *      les verrous appropries ou avoir une file d'attente privee pour faire ca.
453  *
454  *      Renvoie %NULL pour une liste vide ou un pointeur sur l'element de tete.
455  *      Le compteur de reference n'est pas incremente et la reference est ainsi
456  *      volatile. A utiliser avec precaution.
457  */
458 static inline struct sk_buff *skb_peek(struct sk_buff_head *list_)
459 {
460         struct sk_buff *list = ((struct sk_buff *)list_)->next;
461         if (list == (struct sk_buff *)list_)
462                 list = NULL;
463         return list;
464 }
```

13.3.3.10 Pointage sur le dernier élément d'une liste

La fonction `skb_peek_tail(list)` renvoie un pointeur sur le dernier élement de la file d'attente passée en paramètre, si celle-ci n'est pas vide. Sinon `NULL` est renvoyé. L'élément n'est pas retiré de la file d'attente.

Code Linux 2.6.10

Elle est définie dans le fichier `linux/include/linux/skbuff.h`:

```
466 /**
467  *      skb_peek_tail
468  *      @list_ : liste sur laquelle pointer
469  *
470  *      Pointe sur un &sk_buff. Contrairement a beaucoup d'autres operations vous _DEVEZ_
471  *      etre soigneux avec celle-ci. Pointer laisse le tampon dans la
472  *      liste et quelqu'un d'autre peut se l'approprier. Vous devez detenir
473  *      les verrous appropries ou avoir une file d'attente privee pour ca.
474  *
475  *      Renvoie %NULL pour une liste vide ou un pointeur sur l'element de queue.
476  *      Le compteur de reference n'est pas incremente et la reference est ainsi
477  *      volatile. A utiliser avec precaution.
478  */
```

```
479 static inline struct sk_buff *skb_peek_tail(struct sk_buff_head *list_)
480 {
481         struct sk_buff *list = ((struct sk_buff *)list_)->prev;
482         if (list == (struct sk_buff *)list_)
483                 list = NULL;
484         return list;
485 }
```

13.4 Initialisation de l'antémémoire des tampons de socket

L'antémémoire de tampons de socket est initialisée grâce à la fonction `skb_init()`, définie dans le fichier `linux/net/core/skbuff.c`:

Code Linux 2.6.10

```
1428 void __init skb_init(void)
1429 {
1430         skbuff_head_cache = kmem_cache_create("skbuff_head_cache",
1431                                               sizeof(struct sk_buff),
1432                                               0,
1433                                               SLAB_HWCACHE_ALIGN,
1434                                               NULL, NULL);
1435         if (!skbuff_head_cache)
1436                 panic("cannot create skbuff cache");
1437 }
```

On se contente d'essayer d'instantier la variable `skbuff_head_cache` pour qu'elle pointe sur un emplacement mémoire de la taille d'un descripteur de socket. Si on n'y parvient pas, on affiche un message noyau.

13.5 Fonctions de copie

13.5.1 Duplication d'un descripteur de tampon

13.5.1.1 Description

La fonction:

```
struct sk_buff *skb_clone(struct sk_buff *skb, int gfp_mask);
```

duplique le descripteur de tampon de socket `skb` et renvoie l'adresse du nouveau (ou la valeur `NULL` s'il y a un problème). On spécifie que les deux descripteurs de tampon en présence possèdent un clone. Les pointeurs du descripteur originel et ceux du nouveau descripteur désignent le même tampon. Puisqu'il n'existe aucune référence inverse dans le tampon à son ou ses descripteurs de tampon, la zone mémoire du tampon doit être traitée en lecture seule.

Cette fonction est requise entre autres au cours de l'implémentation de la multidiffusion. On évite ainsi la fastidieuse copie du tampon lors de l'émission d'un paquet sur plusieurs périphériques réseau.

13.5.1.2 Définition

La fonction `skb_clone()` est définie dans le fichier `linux/net/core/skbuff.c`:

Code Linux 2.6.10

```
259 /**
260  *      skb_clone       -       duplique un sk_buff
261  *      @skb : tampon a cloner
262  *      @gfp_mask : priorite d'allocation
263  *
```

```
264  *      Duplique un &sk_buff. Le nouveau n'est pas propriete d'une socket. Les
265  *      copies partagent les memes donnees de paquet mais pas la meme structure. Le nouveau
266  *      tampon a un compteur de reference de 1. Si l'allocation echoue, la
257  *      fonction renvoie %NULL, sinon le nouveau tampon est renvoye.
268  *
269  *      Si cette fonction est appelee depuis une interruption, gfp_mask() doit etre
270  *      %GFP_ATOMIC.
271  */
272
273 struct sk_buff *skb_clone(struct sk_buff *skb, int gfp_mask)
274 {
275         struct sk_buff *n = kmem_cache_alloc(skbuff_head_cache, gfp_mask);
276
277         if (!n)
278                 return NULL;
279
280 #define C(x) n->x = skb->x
281
282         n->next = n->prev = NULL;
283         n->list = NULL;
284         n->sk = NULL;
285         C(stamp);
286         C(dev);
287         C(real_dev);
288         C(h);
289         C(nh);
290         C(mac);
291         C(dst);
292         dst_clone(skb->dst);
293         C(sp);
294 #ifdef CONFIG_INET
295         secpath_get(skb->sp);
296 #endif
297         memcpy(n->cb, skb->cb, sizeof(skb->cb));
298         C(len);
299         C(data_len);
300         C(csum);
301         C(local_df);
302         n->cloned = 1;
303         C(pkt_type);
304         C(ip_summed);
305         C(priority);
306         C(protocol);
307         C(security);
308         n->destructor = NULL;
309 #ifdef CONFIG_NETFILTER
310         C(nfmark);
311         C(nfcache);
312         C(nfct);
313         nf_conntrack_get(skb->nfct);
314         C(nfctinfo);
315 #ifdef CONFIG_NETFILTER_DEBUG
316         C(nf_debug);
317 #endif
318 #ifdef CONFIG_BRIDGE_NETFILTER
319         C(nf_bridge);
320         nf_bridge_get(skb->nf_bridge);
321 #endif
322 #endif /*CONFIG_NETFILTER*/
323 #if defined(CONFIG_HIPPI)
324         C(private);
325 #endif
```

```
326 #ifdef CONFIG_NET_SCHED
327         C(tc_index);
328 #ifdef CONFIG_NET_CLS_ACT
329         n->tc_verd = SET_TC_VERD(skb->tc_verd,0);
330         n->tc_verd = CLR_TC_OK2MUNGE(skb->tc_verd);
331         n->tc_verd = CLR_TC_MUNGED(skb->tc_verd);
332         C(input_dev);
333         C(tc_classid);
334 #endif
335
336 #endif
337         C(truesize);
338         atomic_set(&n->users, 1);
339         C(head);
340         C(data);
341         C(tail);
342         C(end);
343
344         atomic_inc(&(skb_shinfo(skb)->dataref));
345         skb->cloned = 1;
346
347         return n;
348 }
```

dont le code est suffisamment parlant (en gros les champs du nouveau descripteur prennent la même valeur que ceux du descripteur à dupliquer).

La fonction en ligne `dst_clone()` est définie dans le fichier `linux/include/net/dst.h`: Code Linux 2.6.10

```
137 static inline
138 struct dst_entry * dst_clone(struct dst_entry * dst)
139 {
140         if (dst)
141                 atomic_inc(&dst->__refcnt);
142         return dst;
143 }
```

13.5.2 Copie d'un tampon et de son descripteur

13.5.2.1 Description

La fonction :

```
struct sk_buff *skb_copy(const struct sk_buff *skb, int gfp_mask);
```

effectue une copie du descripteur de tampon passé en argument, ainsi que du tampon lui-même. Elle renvoie l'adresse du nouveau descripteur de tampon de socket ou la valeur `NULL` s'il y a un problème.

13.5.2.2 Définition

La fonction `skb_copy()` est définie dans le fichier `linux/net/core/skbuff.c`: Code Linux 2.6.10

```
401 /**
402  *      skb_copy        -       cree une copie privee d'un sk_buff
403  *      @skb : tampon a copier
404  *      @gfp_mask : priorite d'allocation
405  *
406  *      Fait une copie a la fois d'un &sk_buff et de ses donnees. Ceci est utilise
407  *      lorsque l'appelant desire modifier les donnees et a besoin d'une copie privee
408  *      des donnees a alterer. Renvoie %NULL en cas d'echec, le pointeur au tampon
```

```
409  *      en cas de succes. Le tampon renvoye a un compteur de reference de 1.
410  *
411  *      Comme sous-produit, cette fonction transforme un &sk_buff non lineaire en
412  *      un qui est lineaire, de sorte que ce &sk_buff devient completement prive
413  *      et que l'appelant peut modifier toutes les donnees du tampon renvoye. Ceci
414  *      signifie qu'il n'est pas recommande d'utiliser cette fonction dans des
415  *      circonstances ou seul l'en-tete doit etre modifie. Utiliser pskb_copy() alors.
416  */
417
418 struct sk_buff *skb_copy(const struct sk_buff *skb, int gfp_mask)
419 {
420         int headerlen = skb->data - skb->head;
421         /*
422          *      Alloue le tampon de copie
423          */
424         struct sk_buff *n = alloc_skb(skb->end - skb->head + skb->data_len,
425                                       gfp_mask);
426         if (!n)
427                 return NULL;
428
429         /* Positionne le pointeur des donnees */
430         skb_reserve(n, headerlen);
431         /* Positionne le pointeur tail et la longueur */
432         skb_put(n, skb->len);
433         n->csum      = skb->csum;
434         n->ip_summed = skb->ip_summed;
435
436         if (skb_copy_bits(skb, -headerlen, n->head, headerlen + skb->len))
437                 BUG();
438
439         copy_skb_header(n, skb);
440         return n;
441 }
```

Autrement dit :

- On déclare une taille d'en-tête, que l'on initialise avec la différence entre l'adresse des données et celle de l'en-tête, ce qui donne bien ce qui est voulu.
- On déclare l'adresse `n` (comme *new*) du nouveau descripteur de tampon.
- On essaie d'allouer de la place en mémoire vive pour le nouveau tampon. Si on n'y parvient pas, on renvoie `NULL`.
- On ajuste la taille de l'espace de tête du tampon.
- On ajuste la taille effective des données du nouveau tampon.
- On renseigne les champs `csum` et `ip_summed` du nouveau descripteur de tampon, en copiant les valeurs de ceux de l'ancien.
- On copie les données dans le nouveau tampon, grâce à la fonction `skb_copy_bits()`, décrite ci-dessous. Si une erreur intervient lors de la copie, on interrompt tout.
- On copie l'en-tête du descripteur à dupliquer vers le nouveau, grâce à la fonction `copy_skb_header()`, décrite ci-dessous.
- On renvoie l'adresse du nouveau descripteur de tampon.

13.5.2.3 Copie des données du tampon

La fonction :

`skb_copy_bits(const struct sk_buff *skb, int offset, void *to, int len);`

permet de copier `len` octets de la partie des données contenues dans le tampon repéré par le descripteur de tampon `skb` à partir du décalage `offset` à l'emplacement mémoire d'adresse `to`.

Elle est définie dans le fichier `linux/net/core/skbuff.c`: Code Linux 2.6.10

```
855 /* Copie les bits de donnees d'un skb vers un tampon du noyau. */
856
857 int skb_copy_bits(const struct sk_buff *skb, int offset, void *to, int len)
858 {
859         int i, copy;
860         int start = skb_headlen(skb);
861
862         if (offset > (int)skb->len - len)
863                 goto fault;
864
865         /* Copie l'en-tete. */
866         if ((copy = start - offset) > 0) {
867                 if (copy > len)
868                         copy = len;
869                 memcpy(to, skb->data + offset, copy);
870                 if ((len -= copy) == 0)
871                         return 0;
872                 offset += copy;
873                 to     += copy;
874         }
875
876         for (i = 0; i < skb_shinfo(skb)->nr_frags; i++) {
877                 int end;
878
879                 BUG_TRAP(start <= offset + len);
880
881                 end = start + skb_shinfo(skb)->frags[i].size;
882                 if ((copy = end - offset) > 0) {
883                         u8 *vaddr;
884
885                         if (copy > len)
886                                 copy = len;
887
888                         vaddr = kmap_skb_frag(&skb_shinfo(skb)->frags[i]);
889                         memcpy(to,
890                                vaddr + skb_shinfo(skb)->frags[i].page_offset+
891                                offset - start, copy);
892                         kunmap_skb_frag(vaddr);
893
894                         if ((len -= copy) == 0)
895                                 return 0;
896                         offset += copy;
897                         to     += copy;
898                 }
899                 start = end;
900         }
901
902         if (skb_shinfo(skb)->frag_list) {
903                 struct sk_buff *list = skb_shinfo(skb)->frag_list;
904
905                 for (; list; list = list->next) {
906                         int end;
907
908                         BUG_TRAP(start <= offset + len);
909
910                         end = start + list->len;
911                         if ((copy = end - offset) > 0) {
912                                 if (copy > len)
913                                         copy = len;
914                                 if (skb_copy_bits(list, offset - start,
915                                                   to, copy))
```

```
916                                     goto fault;
917                             if ((len -= copy) == 0)
918                                     return 0;
919                             offset += copy;
920                             to     += copy;
921                     }
922                     start = end;
923             }
924     }
925     if (!len)
926             return 0;
927
928 fault:
929     return -EFAULT;
930 }
```

Autrement dit :

- On déclare une variable de début, que l'on initialise avec la longueur de l'en-tête du tampon.

Code Linux 2.6.10

La fonction `skb_headlen()` est définie dans le fichier `linux/include/linux/skbuff.h`:

```
660 static inline unsigned int skb_headlen(const struct sk_buff *skb)
661 {
662         return skb->len - skb->data_len;
663 }
```

- Si la somme du décalage et de la longueur passés en argument est supérieure à la longueur des données du tampon, on renvoie l'opposé du code d'erreur `EFAULT`.
- On détermine le nombre d'octets à copier pour l'en-tête. S'il est non nul, on copie l'en-tête. Si ce qu'on a copié est suffisant, on a terminé, on renvoie donc 0. Sinon on déplace le décalage.
- On copie chacun des fragments du tableau des fragments.

Code Linux 2.6.10

La macro générale `BUG_TRAP()` est définie dans le fichier `linux/include/linux/rtnetlink.h`:

```
821 #define BUG_TRAP(x) do { \
822         if (unlikely(!(x))) { \
823                 printk(KERN_ERR "KERNEL: assertion (%s) failed at %s (%d)\n", \
824                         #x,  __FILE__ , __LINE__); \
825         } \
826 } while(0)
```

Code Linux 2.6.10

Les fonctions `kmap_skb_frag()` et `kunmap_skb_frag()` sont définies dans le fichier `linux/include/linux/skbuff.h`:

```
1055 static inline void *kmap_skb_frag(const skb_frag_t *frag)
1056 {
1057 #ifdef CONFIG_HIGHMEM
1058         BUG_ON(in_irq());
1059
1060         local_bh_disable();
1061 #endif
1062         return kmap_atomic(frag->page, KM_SKB_DATA_SOFTIRQ);
1063 }
1064
1065 static inline void kunmap_skb_frag(void *vaddr)
1066 {
1067         kunmap_atomic(vaddr, KM_SKB_DATA_SOFTIRQ);
1068 #ifdef CONFIG_HIGHMEM
1069         local_bh_enable();
1070 #endif
1071 }
```

- On copie les données des tampons associés aux descripteurs de la liste des fragments.
- On renvoie le nombre d'octets copiés.

13.5.2.4 Copie de l'en-tête du descripteur

La fonction `copy_skb_header()` est définie dans le fichier `linux/net/core/skbuff.c`: Code Linux 2.6.10

```
350 static void copy_skb_header(struct sk_buff *new, const struct sk_buff *old)
351 {
352         /*
353          *      Decalage entre les deux zones des donnees, en octets
354          */
355         unsigned long offset = new->data - old->data;
356
357         new->list       = NULL;
358         new->sk         = NULL;
359         new->dev        = old->dev;
360         new->real_dev   = old->real_dev;
361         new->priority   = old->priority;
362         new->protocol   = old->protocol;
363         new->dst        = dst_clone(old->dst);
364 #ifdef CONFIG_INET
365         new->sp         = secpath_get(old->sp);
366 #endif
367         new->h.raw      = old->h.raw + offset;
368         new->nh.raw     = old->nh.raw + offset;
369         new->mac.raw    = old->mac.raw + offset;
370         memcpy(new->cb, old->cb, sizeof(old->cb));
371         new->local_df   = old->local_df;
372         new->pkt_type   = old->pkt_type;
373         new->stamp      = old->stamp;
374         new->destructor = NULL;
375         new->security   = old->security;
376 #ifdef CONFIG_NETFILTER
377         new->nfmark     = old->nfmark;
378         new->nfcache    = old->nfcache;
379         new->nfct       = old->nfct;
380         nf_conntrack_get(old->nfct);
381         new->nfctinfo   = old->nfctinfo;
382 #ifdef CONFIG_NETFILTER_DEBUG
383         new->nf_debug   = old->nf_debug;
384 #endif
385 #ifdef CONFIG_BRIDGE_NETFILTER
386         new->nf_bridge  = old->nf_bridge;
387         nf_bridge_get(old->nf_bridge);
388 #endif
389 #endif
390 #ifdef CONFIG_NET_SCHED
391 #ifdef CONFIG_NET_CLS_ACT
392         new->tc_verd = old->tc_verd;
393 #endif
394         new->tc_index   = old->tc_index;
395 #endif
396         atomic_set(&new->users, 1);
397         skb_shinfo(new)->tso_size = skb_shinfo(old)->tso_size;
398         skb_shinfo(new)->tso_segs = skb_shinfo(old)->tso_segs;
399 }
```

Elle se contente tout simplement de recopier un certain nombre de champs de l'ancien descripteur vers le nouveau.

13.5.3 Copie du tampon sans linéarisation

Code Linux 2.6.10

La fonction `pskb_copy()` est définie dans le fichier `linux/net/core/skbuff.c`:

```
444 /**
445  *      pskb_copy       -       cree une copie d'un sk_buff avec une tete privee.
446  *      @skb : tampon a copier
447  *      @gfp_mask : priorite d'allocation
448  *
449  *      Fait une copie a la fois d'un &sk_buff et d'une partie de ses donnees, celles
450  *      situees dans l'en-tete. Les donnees fragmentees restent partagees. Ceci est
451  *      utilise lorsque l'appelant ne desire modifier que l'en-tete du &sk_buff et a besoin
452  *      d'une copie privee de l'en-tete a alterer. Renvoie %NULL en cas d'echec
453  *      ou l'adresse du tampon en cas de succes.
454  *      Le tampon renvoye a un compteur de reference de 1.
455  */
456
457 struct sk_buff *pskb_copy(struct sk_buff *skb, int gfp_mask)
458 {
459         /*
460          *      Alloue le tampon de copie
461          */
462         struct sk_buff *n = alloc_skb(skb->end - skb->head, gfp_mask);
463
464         if (!n)
465                 goto out;
466
467         /* Positionne le pointeur des donnees */
468         skb_reserve(n, skb->data - skb->head);
469         /* Positionne le pointeur tail et la longueur */
470         skb_put(n, skb_headlen(skb));
471         /* Copie les octets */
472         memcpy(n->data, skb->data, n->len);
473         n->csum      = skb->csum;
474         n->ip_summed = skb->ip_summed;
475
476         n->data_len  = skb->data_len;
477         n->len       = skb->len;
478
479         if (skb_shinfo(skb)->nr_frags) {
480                 int i;
481
482                 for (i = 0; i < skb_shinfo(skb)->nr_frags; i++) {
483                         skb_shinfo(n)->frags[i] = skb_shinfo(skb)->frags[i];
484                         get_page(skb_shinfo(n)->frags[i].page);
485                 }
486                 skb_shinfo(n)->nr_frags = i;
487         }
488
489         if (skb_shinfo(skb)->frag_list) {
490                 skb_shinfo(n)->frag_list = skb_shinfo(skb)->frag_list;
491                 skb_clone_fraglist(n);
492         }
493
494         copy_skb_header(n, skb);
495 out:
496         return n;
497 }
```

Le code se comprend aisément.

13.5.4 Copie d'un tampon et de son descripteur avec modification

13.5.4.1 Description

La fonction :

```
struct sk_buff *skb_copy_expand(const struct sk_buff *skb,
                                int newheadroom,
                                int newtailroom,
                                int gfp_mask);
```

effectue une copie du tampon et du descripteur du tampon désigné par le descripteur `skb`, en en profitant pour modifier la taille de l'espace de tête (passant à `newheadroom` octets) et celle de l'espace d'en-queue (passant à `newtailroom` octets). On renvoie l'adresse du nouveau descripteur de tampon de socket ou la valeur `NULL` s'il y a un problème.

13.5.4.2 Définition

La fonction `skb_copy_expand()` est définie dans le fichier `linux/net/core/skbuff.c`: Code Linux 2.6.10

```
582 /**
583  *      skb_copy_expand -       copie et etend un sk_buff
584  *      @skb : tampon a copier
585  *      @newheadroom : nouveaux octets libres en tete
586  *      @newtailroom : nouveaux octets libres en queue
587  *      @gfp_mask : priorite d'allocation
588  *
589  *      Fait une copie a la fois d'un &sk_buff et de ses donnees tout en
590  *      allouant un espace supplementaire.
591  *
592  *      Ceci est utilise lorsque l'appelant desire modifier les donnees et a besoin d'une
593  *      copie privee des donnees a alterer ainsi que de plus d'espace pour de nouveaux champs.
594  *      Renvoie %NULL en cas d'echec ou l'adresse du tampon en cas
595  *      de succes. Le tampon renvoye a un compteur de reference de 1.
596  *
597  *      Vous devez passer %GFP_ATOMIC comme priorite d'allocation si cette fonction
598  *      est appelee depuis une interruption.
599  *
600  *      ALERTE DE BOGUE : ip_summed n'est pas copie. Pourquoi cela marche-t-il ? Est-elle
601  *      seulement utilisee par netfilter dans les cas ou checksum est recalculee ? --ANK
602  */
603 struct sk_buff *skb_copy_expand(const struct sk_buff *skb,
604                                 int newheadroom, int newtailroom, int gfp_mask)
605 {
606         /*
607          *      Alloue le tampon de copie
608          */
609         struct sk_buff *n = alloc_skb(newheadroom + skb->len + newtailroom,
610                                       gfp_mask);
611         int head_copy_len, head_copy_off;
612
613         if (!n)
614                 return NULL;
615
616         skb_reserve(n, newheadroom);
617
618         /* Positionne le pointeur tail et la longueur */
619         skb_put(n, skb->len);
620
621         head_copy_len = skb_headroom(skb);
622         head_copy_off = 0;
```

```
623         if (newheadroom <= head_copy_len)
624                 head_copy_len = newheadroom;
625         else
626                 head_copy_off = newheadroom - head_copy_len;
627
628         /* Copie l'en-tete et les donnees lineairement. */
629         if (skb_copy_bits(skb, -head_copy_len, n->head + head_copy_off,
630                           skb->len + head_copy_len))
631                 BUG();
632
633         copy_skb_header(n, skb);
634
635         return n;
636 }
```

Par rapport à la fonction de copie simple, on voit surtout la différence au niveau de l'instantiation du nouveau descripteur de tampon.

13.6 Ajustement d'une zone des données occupée

Nous avons vu ci-dessus comment ajuster la taille d'une zone des données vides. C'est plus difficile lorsque celle-ci contient des données. Nous allons voir comment faire dans ce cas ici.

13.6.1 Retrait de données au début

13.6.1.1 Fonction d'appel

La fonction `pskb_pull(skb, len)` tronque de `len` octets le début de la zone des données d'un tampon, après avoir vérifié qu'on demeurera après ceci dans la zone de mémoire affectée au tampon. Elle est définie dans le fichier `linux/include/linux/skbuff.h`:

Code Linux 2.6.10

```
768 extern unsigned char *__pskb_pull_tail(struct sk_buff *skb, int delta);
769
770 static inline unsigned char *__pskb_pull(struct sk_buff *skb, unsigned int len)
771 {
772         if (len > skb_headlen(skb) &&
773             !__pskb_pull_tail(skb, len-skb_headlen(skb)))
774                 return NULL;
775         skb->len -= len;
776         return skb->data += len;
777 }
778
779 static inline unsigned char *pskb_pull(struct sk_buff *skb, unsigned int len)
780 {
781         return unlikely(len > skb->len) ? NULL : __pskb_pull(skb, len);
782 }
783
784 static inline int pskb_may_pull(struct sk_buff *skb, unsigned int len)
785 {
786         if (likely(len <= skb_headlen(skb)))
787                 return 1;
788         if (unlikely(len > skb->len))
789                 return 0;
790         return __pskb_pull_tail(skb, len-skb_headlen(skb)) != NULL;
791 }
```

Elle fait essentiellement appel à la fonction interne `__pskb_pull_tail()`.

13.6.1.2 Fonction interne

La fonction `__pskb_pull_tail()` est définie dans le fichier `linux/net/core/skbuff.c`: Code Linux 2.6.10

```
716 /**
717  *      __pskb_pull_tail - avance la fin d'un en-tete de skb
718  *      @skb : tampon a reallouer
719  *      @delta : nombre d'octets a avancer a la fin
720  *
721  *      Cette fonction n'a de sens que pour un &sk_buff fragmente,
722  *      elle etend l'en-tete en deplacant la fin et en copiant les donnees
723  *      necessaires de la partie fragmentee.
724  *
725  *      &sk_buff DOIT avoir un compteur de reference de 1.
726  *
727  *      Renvoie %NULL (et &sk_buff ne change pas) si l'etirement echoue
728  *      ou la valeur de la nouvelle fin du skb en cas de succes.
729  *
730  *      Tous les pointeurs pointant sur l'en-tete de skb peuvent changer et doivent
731  *      donc etre charges a nouveau apres appel de cette fonction.
732  */
733
734 /* Deplace la fin de la tete du skb, en copiant les donnees de la partie fragmentee
735  * lorsque c'est necessaire.
736  * 1. On peut echouer a cause d'un echec memoire.
737  * 2. On peut changer les pointeurs de skb.
738  *
739  * C'est gentiment complique. Heureusement, elle est appelee seulement dans des cas
      exceptionels.
740  */
741 unsigned char *__pskb_pull_tail(struct sk_buff *skb, int delta)
742 {
743         /* Si skb n'a pas suffisamment d'espace libre a la fin, en prendre plus
744          * ainsi que 128 octets pour les futures extensions. Si nous n'avons pas assez
745          * de place a la fin, reallouer sans extension seulement si skb est clone.
746          */
747         int i, k, eat = (skb->tail + delta) - skb->end;
748
749         if (eat > 0 || skb_cloned(skb)) {
750                 if (pskb_expand_head(skb, 0, eat > 0 ? eat + 128 : 0,
751                                      GFP_ATOMIC))
752                         return NULL;
753         }
754
755         if (skb_copy_bits(skb, skb_headlen(skb), skb->tail, delta))
756                 BUG();
757
758         /* Optimisation : pas de fragments, pas de raisons de preestimer
759          * la taille des pages a placer. Superbe.
760          */
761         if (!skb_shinfo(skb)->frag_list)
762                 goto pull_pages;
763
764         /* Estimer la taille des pages a placer. */
765         eat = delta;
766         for (i = 0; i < skb_shinfo(skb)->nr_frags; i++) {
767                 if (skb_shinfo(skb)->frags[i].size >= eat)
768                         goto pull_pages;
769                 eat -= skb_shinfo(skb)->frags[i].size;
770         }
771
772         /* Si nous avons besoin de mettre a jour la frag_list, nous sommes ennuyes.
773          * Il est certainement possible d'ajouter un decalage aux donnees de skb,
```

```
774          * mais on s'attend a ce que prendre en compte ce depilage soit
775          * une operation tres rare, il est pire de combattre contre les tetes de
776          * skb gonflees et de nous crucifier ici a la place.
777          * Pur masochisme, en effet. 8)8)
778          */
779         if (eat) {
780                 struct sk_buff *list = skb_shinfo(skb)->frag_list;
781                 struct sk_buff *clone = NULL;
782                 struct sk_buff *insp = NULL;
783
784                 do {
785                         if (!list)
786                                 BUG();
787
788                         if (list->len <= eat) {
789                                 /* Mange en entier. */
790                                 eat -= list->len;
791                                 list = list->next;
792                                 insp = list;
793                         } else {
794                                 /* Mange partiellement. */
795
796                                 if (skb_shared(list)) {
797                                         /* Tres mauvais ! Nous avons besoin de
                                            bifurquer la liste. :-( */
798                                         clone = skb_clone(list, GFP_ATOMIC);
799                                         if (!clone)
800                                                 return NULL;
801                                         insp = list->next;
802                                         list = clone;
803                                 } else {
804                                         /* Ceci peut etre place sans
805                                          * probleme. */
806                                         insp = list;
807                                 }
808                                 if (!pskb_pull(list, eat)) {
809                                         if (clone)
810                                                 kfree_skb(clone);
811                                         return NULL;
812                                 }
813                                 break;
814                         }
815                 } while (eat);
816
817                 /* Liberer les fragments. */
818                 while ((list = skb_shinfo(skb)->frag_list) != insp) {
819                         skb_shinfo(skb)->frag_list = list->next;
820                         kfree_skb(list);
821                 }
822                 /* Et inserer un nouveau clone en tete. */
823                 if (clone) {
824                         clone->next = list;
825                         skb_shinfo(skb)->frag_list = clone;
826                 }
827         }
828         /* Succes ! Maintenant nous pouvons faire subir des changements aux donnees de skb. */
829
830 pull_pages:
831         eat = delta;
832         k = 0;
833         for (i = 0; i < skb_shinfo(skb)->nr_frags; i++) {
834                 if (skb_shinfo(skb)->frags[i].size <= eat) {
```

```
835                         put_page(skb_shinfo(skb)->frags[i].page);
836                         eat -= skb_shinfo(skb)->frags[i].size;
837                 } else {
838                         skb_shinfo(skb)->frags[k] = skb_shinfo(skb)->frags[i];
839                         if (eat) {
840                                 skb_shinfo(skb)->frags[k].page_offset += eat;
841                                 skb_shinfo(skb)->frags[k].size -= eat;
842                                 eat = 0;
843                         }
844                         k++;
845                 }
846         }
847         skb_shinfo(skb)->nr_frags = k;
848
849         skb->tail     += delta;
850         skb->data_len -= delta;
851
852         return skb->tail;
853 }
```

13.6.1.3 Extension de l'espace de tête

La fonction `pskb_expand_head()` est définie dans le fichier `linux/net/core/skbuff.c`: Code Linux 2.6.10

```
499 /**
500  *      pskb_expand_head - realloue l'en-tete d'un &sk_buff
501  *      @skb : tampon a reallouer
502  *      @nhead : place a ajouter en tete
503  *      @ntail : place a ajouter a la fin
504  *      @gfp_mask : priorite d'allocation
505  *
506  *      Etend (ou cree une copie identique, si &nhead et &ntail sont nuls)
507  *      un en-tete de skb. &sk_buff lui-meme n'est pas change. &sk_buff DOIT avoir
508  *      son compteur de reference a 1. Renvoie zero en cas de succes, ou une erreur
509  *      si l'extension echoue. Dans ce dernier cas, &sk_buff n'est pas change.
510  *
511  *      Tous les pointeurs pointant sur l'en-tete de skb peuvent changer et doivent
512  *      donc etre charges a nouveau apres appel de cette fonction.
513  */
514
515 int pskb_expand_head(struct sk_buff *skb, int nhead, int ntail, int gfp_mask)
516 {
517         int i;
518         u8 *data;
519         int size = nhead + (skb->end - skb->head) + ntail;
520         long off;
521
522         if (skb_shared(skb))
523                 BUG();
524
525         size = SKB_DATA_ALIGN(size);
526
527         data = kmalloc(size + sizeof(struct skb_shared_info), gfp_mask);
528         if (!data)
529                 goto nodata;
530
531         /* Copier seulement les donnees effectives... et, helas, l'en-tete. Ceci devrait
532          * etre optimise dans les cas ou l'en-tete est vide. */
533         memcpy(data + nhead, skb->head, skb->tail - skb->head);
534         memcpy(data + size, skb->end, sizeof(struct skb_shared_info));
535
536         for (i = 0; i < skb_shinfo(skb)->nr_frags; i++)
```

```
537                 get_page(skb_shinfo(skb)->frags[i].page);
538
539         if (skb_shinfo(skb)->frag_list)
540                 skb_clone_fraglist(skb);
541
542         skb_release_data(skb);
543
544         off = (data + nhead) - skb->head;
545
546         skb->head     = data;
547         skb->end      = data + size;
548         skb->data    += off;
549         skb->tail    += off;
550         skb->mac.raw += off;
551         skb->h.raw   += off;
552         skb->nh.raw  += off;
553         skb->cloned   = 0;
554         atomic_set(&skb_shinfo(skb)->dataref, 1);
555         return 0;
556
557 nodata:
558         return -ENOMEM;
559 }
```

Code Linux 2.6.10

La fonction `skb_clone_fraglist()` est définie dans le fichier `linux/net/core/skbuff.c`:

```
180 static void skb_clone_fraglist(struct sk_buff *skb)
181 {
182         struct sk_buff *list;
183
184         for (list = skb_shinfo(skb)->frag_list; list; list = list->next)
185                 skb_get(list);
186 }
```

13.6.1.4 Réallocation

La fonction `skb_realloc_headroom(skb, newheadroom)` permet de générer un nouveau tampon de socket dont l'espace de tête a maintenant la taille `newheadroom`. La portion de données de l'ancien tampon est copiée dans le nouveau et la plupart des champs du descripteur sont pris en compte. Seuls les champs `sk` et `list` sont définis à `NULL`.

Code Linux 2.6.10

Elle est définie dans le fichier `linux/net/core/skbuff.c`:

```
561 /* Fait une copie privee de skb avec une tete sur laquelle on peut ecrire
       et un peu d'espace de queue */
562
563 struct sk_buff *skb_realloc_headroom(struct sk_buff *skb, unsigned int headroom)
564 {
565         struct sk_buff *skb2;
566         int delta = headroom - skb_headroom(skb);
567
568         if (delta <= 0)
569                 skb2 = pskb_copy(skb, GFP_ATOMIC);
570         else {
571                 skb2 = skb_clone(skb, GFP_ATOMIC);
572                 if (skb2 && pskb_expand_head(skb2, SKB_DATA_ALIGN(delta), 0,
573                                              GFP_ATOMIC)) {
574                         kfree_skb(skb2);
575                         skb2 = NULL;
576                 }
577         }
578         return skb2;
579 }
```

13.6.2 Retrait à la fin

13.6.2.1 Fonction d'appel

La fonction `skb_trim(skb, len)` taille (*to trim* en anglais) la fin de la zone des données d'un tampon pour que celle-ci fasse `len` octets. Elle est définie dans le fichier `linux/include/linux/skbuff.h`:

Code Linux 2.6.10

```
853 extern int ___pskb_trim(struct sk_buff *skb, unsigned int len, int realloc);
854
855 static inline void __skb_trim(struct sk_buff *skb, unsigned int len)
856 {
857         if (!skb->data_len) {
858                 skb->len  = len;
859                 skb->tail = skb->data + len;
860         } else
861                 ___pskb_trim(skb, len, 0);
862 }
863
864 /**
865  *      skb_trim - efface la fin d'un tampon
866  *      @skb : tampon a alterer
867  *      @len : nouvelle longueur
868  *
869  *      Coupe la longueur d'un tampon en effacant la fin des donnees. Si
870  *      le tampon est deja en-dessous de la longueur specifiee, il n'est pas modifie.
871  */
872 static inline void skb_trim(struct sk_buff *skb, unsigned int len)
873 {
874         if (skb->len > len)
875                 __skb_trim(skb, len);
876 }
877
878
879 static inline int __pskb_trim(struct sk_buff *skb, unsigned int len)
880 {
881         if (!skb->data_len) {
882                 skb->len  = len;
883                 skb->tail = skb->data+len;
884                 return 0;
885         }
886         return ___pskb_trim(skb, len, 1);
887 }
888
889 static inline int pskb_trim(struct sk_buff *skb, unsigned int len)
890 {
891         return (len < skb->len) ? __pskb_trim(skb, len) : 0;
892 }
```

13.6.2.2 Fonction interne

La fonction `__pskb_trim()` est définie dans le fichier `linux/net/core/skbuff.c`:

Code Linux 2.6.10

```
667 /* Taille skb a la longueur len. On peut changer les pointeurs de skb si "realloc" est 1.
668  * Si realloc==0 et que la taille est impossible sans changer les donnees,
669  * c'est BUG().
670  */
671
672 int ___pskb_trim(struct sk_buff *skb, unsigned int len, int realloc)
673 {
674         int offset = skb_headlen(skb);
675         int nfrags = skb_shinfo(skb)->nr_frags;
```

```
676         int i;
677
678         for (i = 0; i < nfrags; i++) {
679                 int end = offset + skb_shinfo(skb)->frags[i].size;
680                 if (end > len) {
681                         if (skb_cloned(skb)) {
682                                 if (!realloc)
683                                         BUG();
684                                 if (pskb_expand_head(skb, 0, 0, GFP_ATOMIC))
685                                         return -ENOMEM;
686                         }
687                         if (len <= offset) {
688                                 put_page(skb_shinfo(skb)->frags[i].page);
689                                 skb_shinfo(skb)->nr_frags--;
690                         } else {
691                                 skb_shinfo(skb)->frags[i].size = len - offset;
692                         }
693                 }
694                 offset = end;
695         }
696
697         if (offset < len) {
698                 skb->data_len -= skb->len - len;
699                 skb->len       = len;
700         } else {
701                 if (len <= skb_headlen(skb)) {
702                         skb->len      = len;
703                         skb->data_len = 0;
704                         skb->tail     = skb->data + len;
705                         if (skb_shinfo(skb)->frag_list && !skb_cloned(skb))
706                                 skb_drop_fraglist(skb);
707                 } else {
708                         skb->data_len -= skb->len - len;
709                         skb->len       = len;
710                 }
711         }
712
713         return 0;
714 }
```

13.6.3 Extension de l'espace de tête

La fonction `skb_cow(skb, headroom)` vérifie si le tampon possède au moins `headroom` octets dans l'espace de tête et s'il s'agit d'un clone. Si une des deux situations est réalisée, un nouveau tampon est généré. Elle est définie dans le fichier `linux/include/linux/skbuff.h`:

Code Linux 2.6.10

```
964 /**
965  *      skb_cow - copie l'en-tete de skb lorsque c'est necessaire
966  *      @skb : tampon a intimider
967  *      @headroom : headroom necessaire
968  *
969  *      Si le skb passe manque de suffisamment de headroom ou si sa partie donnees
970  *      est partagee, les donnees sont reallouees. Si la reallocation echoue, une erreur
971  *      est renvoyee et le skb originel n'est pas change.
972  *
973  *      Le resultat est un skb avec une zone skb->head...skb->tail sur laquelle on
974  *      peut ecrire et au moins @headroom d'espace de tete.
975  */
976 static inline int skb_cow(struct sk_buff *skb, unsigned int headroom)
977 {
978         int delta = (headroom > 16 ? headroom : 16) - skb_headroom(skb);
979
```

```
980         if (delta < 0)
981                 delta = 0;
982
983         if (delta || skb_cloned(skb))
984                 return pskb_expand_head(skb, (delta + 15) & ~15, 0, GFP_ATOMIC);
985         return 0;
986 }
```

13.7 Allocation d'un tampon de socket pour l'émission

La fonction en ligne `dev_alloc_skb(length)` génère un tampon de socket dont l'espace de tête a une taille de 16 octets et l'espace de queue une taille de `length` octets.

Elle est définie dans le fichier `linux/include/linux/skbuff.h`: Code Linux 2.6.10

```
926 /**
927  *      __dev_alloc_skb - alloue un skbuff pour l'emission
928  *      @length : longueur a allouer
929  *      @gfp_mask : masque get_free_pages, passe a alloc_skb
930  *
931  *      Alloue un nouveau &sk_buff et lui assigne un compteur d'utilisation de un. Le
932  *      tampon a un espace de tete non specifie. Les utilisateurs doivent allouer
933  *      l'espace de tete qu'ils pensent necessaire sans comptabiliser l'espace de
934  *      construction. L'espace de construction est utilise pour des optimisations.
935  *
936  *      %NULL est renvoye s'il n'y a pas d'emplacement memoire libre.
937  */
938 static inline struct sk_buff *__dev_alloc_skb(unsigned int length,
939                                               int gfp_mask)
940 {
941         struct sk_buff *skb = alloc_skb(length + 16, gfp_mask);
942         if (likely(skb))
943                 skb_reserve(skb, 16);
944         return skb;
945 }
946
947 /**
948  *      dev_alloc_skb - alloue un skbuff pour l'emission
949  *      @length : longueur a allouer
950  *
951  *      Alloue un nouveau &sk_buff et lui assigne un compteur d'utilisation de un. Le
952  *      tampon a un headroom non specifie. Les utilisateurs doivent allouer
953  *      le headroom qu'ils pensent necessaire sans comptabiliser l'espace de
954  *      construction. L'espace de construction est utilise pour des optimisations.
955  *
956  *      %NULL est renvoye s'il n'y a pas d'emplacement memoire libre. Bien que cette
957  *      fonction alloue de la memoire, elle peut etre appelee depuis une interruption.
958  */
959 static inline struct sk_buff *dev_alloc_skb(unsigned int length)
960 {
961         return __dev_alloc_skb(length, GFP_ATOMIC);
962 }
```

Chapitre 14

Les structures de données Linux pour le pilote de périphérique réseau

La première action à entreprendre avant de pouvoir utiliser le réseau est de détecter la carte réseau (éventuellement les cartes réseau) de l'ordinateur, ce qui est un problème de gestion de périphérique, comme détecter un disque dur, une carte graphique ou une clé USB.

Unix distingue traditionnellement deux types seulement de fichiers de périphériques: les fichiers caractère et les fichiers bloc. Linux préfère introduire un troisième type de fichiers de périphériques pour les réseaux. Il serait possible de considérer un périphérique réseau comme un périphérique bloc mais avec une certaine gymnastique inutile.

Les structures de données fondamentales concernant les pilotes de périphériques réseau sous Linux sont au nombre de deux:

- La carte réseau est représentée par un descripteur, de type `struct net_device`. Comme pour tout périphérique, la carte réseau sera vue par le système d'exploitation comme un fichier.
- L'en-tête de la trame, appelé **en-tête matériel** ou **en-tête physique**, est placée dans un cache afin d'éviter de le recomposer sans cesse.

Nous terminerons par l'ordre des octets utilisé sur les réseaux.

14.1 Descripteur d'interface de périphérique réseau

14.1.1 Philosophie Linux des fichiers de périphérique réseau

Comme nous l'avons dit dans l'introduction, Linux préfère définir un troisième type de fichiers de périphériques pour les réseaux. Ce type présente quelques différences par rapport aux fichiers de périphériques caractère ou bloc :

- Le rôle d'un périphérique réseau dans le système est semblable à celui d'un périphérique bloc monté. Un périphérique bloc enregistre ses caractéristiques dans le tableau `blk_array[]` puis reçoit et transmet des blocs sur requête, au moyen de la fonction `request()`. De même un périphérique réseau doit s'enregistrer avant que des paquets soient échangés avec le monde extérieur. Nous verrons au chapitre 15 comment on enregistre un périphérique réseau.
- Sous Linux, un pilote de carte réseau peut être lié statiquement au noyau ou chargé comme module, comme pour tout pilote de périphérique. Cependant, contrairement aux périphériques caractère et bloc, le pilote de carte réseau ne se déclare pas de la même façon suivant qu'il est lié statiquement ou modularisé.
- Un périphérique réseau n'est pas repéré par un nombre majeur et un nombre mineur, comme le sont les périphériques caractère et bloc, mais par un **index** comme nous le verrons.
- Chaque pilote correspond à un descripteur de type `struct net_device` inséré dans une liste chaînée globale de périphériques réseau, appelée `dev_base` et déclarée dans le fichier `linux/net/core/dev.c`:

Code Linux 2.6.10

```
185 struct net_device *dev_base;
```

- Un disque possède un fichier spécial dans le répertoire `/dev` mais les périphériques réseau ne possèdent pas de telle entrée. Il n'existe donc, par exemple, aucun périphérique réseau tel que `/dev/eth0` ou `/dev/atm1`.
- Les opérations habituelles sur les fichiers (en particulier les appels système de lecture et d'écriture simples `read()` et `write()`) n'ont pas de sens pour les périphériques réseau (à part les sockets locales). Ceux-ci utilisent un ensemble différent d'opérations, comme indiqué au début du fichier `linux/drivers/net/net_init.c`:

Code Linux 2.6.10

```
55 /* Les peripheriques reseau existant a l'heure actuelle uniquement dans l'espace de
56    noms des sockets, ces entrees ne sont pas utilisees. Les seules qui ont un sens sont :
57     open        demarre la carte Ethernet
58     close       arrete la carte Ethernet
59     ioctl       Pour obtenir des statistiques et peut-etre positionner le port de
      l'interface (AUI, BNC, etc.)
60    On peut aussi imaginer obtenir des paquets bruts en utilisant
61     read & write
62    mais il vaut probalement mieux les manipuler par une socket de paquet brut.
63
64    Etant donne que presque toutes ces fonctions sont manipulees dans le schema
65    socket actuel, mettre les peripheriques carte Etherner dans /dev/ semble sans interet.
66
67    [Effacer tous les supports pour les peripheriques reseau de /dev. Lorsque quelqu'un
68     envoie des flux, nous les recuperons par magie, mais sinon ils ne sont pas
69         necessaires et constituent une perte d'espace]
70 */
```

Les descripteurs de périphérique réseau ne représentent pas tous une carte physique. Il existe également des périphériques comme le périphérique réseau en boucle `loopback` qui offrent une fonctionnalité de réseau logique.

14.1.2 Définition du type

Un périphérique réseau apparaît à Linux sous la forme d'un descripteur d'interface constituée d'un certain nombre d'attributs et de méthodes (ou fonctions de service). Il s'agit d'une classe au sens de la programmation orientée objet, mais implémentée en C.

Le descripteur de l'interface d'un périphérique réseau est une entité du type `struct net_device`, défini dans le fichier en-tête `linux/include/linux/netdevice.h`. Commençons par en reproduire la définition que nous commenterons ensuite :

Code Linux 2.6.10

```
6    *              Definitions pour les interfaces de routines.
7    *
8    * Version :    @(#)dev.h       1.0.10  08/12/93
9    *
10   * Auteurs :    Ross Biro, <bir7@leland.Stanford.Edu>
11   *              Fred N. van Kempen, <waltje@uWalt.NL.Mugnet.ORG>
12   *              Corey Minyard <wf-rch!minyard@relay.EU.net>
13   *              Donald J. Becker, <becker@cesdis.gsfc.nasa.gov>
14   *              Alan Cox, <Alan.Cox@linux.org>
15   *              Bjorn Ekwall. <bj0rn@blox.se>
16   *              Pekka Riikonen <priikone@poseidon.pspt.fi>
[...]
23   *              Deplace a /usr/include/linux pour NET3
24   */
25  #ifndef _LINUX_NETDEVICE_H
26  #define _LINUX_NETDEVICE_H
[...]
254 /*
255  *      La structure PERIPHERIQUE.
256  *      En fait cette structure entiere est une grosse erreur. Elle melange des donnees
257  *      d'E/S avec des donnees strictement de "haut niveau", et elle doit connaitre
258  *      presque toutes les structures de donnees utilisees dans le module INET.
259  *
260  *      A RECTIFIER : nettoyer struct net_device de facon a ce que les infos sur
261  *      le protocole reseau soient deplacees.
262  */
263
264 struct net_device
265 {
266
267         /*
268          * Ceci est le premier champ de la partie "visible" de cette structure
269          * (c'est-a-dire vue par les utilisateurs dans le fichier "Space.c"). C'est le nom
270          * de l'interface.
271          */
272         char                    name[IFNAMSIZ];
273
274         /*
275          *      Champs specifiques aux E/S
276          *      A RECTIFIER : Fusionner ceci et struct ifmap en une seule
277          */
278         unsigned long           mem_end;        /* Fin de la memoire partagee    */
279         unsigned long           mem_start;      /* Debut de la memoire partagee  */
280         unsigned long           base_addr;      /* adresse d'E/S du peripherique */
281         unsigned int            irq;            /* numero d'IRQ du peripherique  */
282
283         /*
284          *      Certains materiels necessitent egalement ces champs, mais ils ne font
285          *      pas partie de l'ensemble usuel specifie dans Space.c.
286          */
287
288         unsigned char           if_port;        /* AUI, TP,... selectionnable */
289         unsigned char           dma;            /* canal DMA                  */
```

```
290
291         unsigned long           state;
292
293         struct net_device       *next;
294
295         /* La fonction d'initialisation du peripherique. Appelee seulement une fois. */
296         int                     (*init)(struct net_device *dev);
297
298         /* ------- Les champs preinitialises dans Space.c se terminent ici --------- */
299
300         struct net_device       *next_sched;
301
302         /* Index d'interface. Identificateur unique de peripherique                 */
303         int                     ifindex;
304         int                     iflink;
305
306
307         struct net_device_stats* (*get_stats)(struct net_device *dev);
308         struct iw_statistics*   (*get_wireless_stats)(struct net_device *dev);
309
310         /* Liste de fonctions pour gerer les extensions sans fil (au lieu de).
311          * Voir <net/iw_handler.h> pour les details. Jean II */
312         const struct iw_handler_def *   wireless_handlers;
313         /* Donnees de l'instance gerees par le coeur des Extensions Sans Fil. */
314         struct iw_public_data * wireless_data;
315
316         struct ethtool_ops *ethtool_ops;
317
318         /*
319          * Ceci marque la fin de la partie "visible" de la structure. Tous
320          * les champs ci-apres sont internes au systeme, et peuvent etre changes a
321          * volonte (lire : peuvent etre enleves a volonte).
322          */
323
324         /* Ceux-ci peuvent etre necessaire pour du code futur de debranchement du reseau
               a distance. */
325         unsigned long           trans_start;    /* Heure (en jiffies) de la derniere Tx */
326         unsigned long           last_rx;        /* Heure de la derniere Rx              */
327
328         unsigned short          flags;  /* Drapeaux de l'interface (a la BSD)   */
329         unsigned short          gflags;
330         unsigned short          priv_flags; /* Comme 'flags' mais dans l'espace utilisateur
                                                   invisible. */
331         unsigned short          unused_alignment_fixer; /* Puisque nous avons besoin de
                                                               priv_flags,
332                                                          * et que nous voulons etre alignes
                                                             sur 32 bits.
333                                                          */
334
335         unsigned                mtu;    /* valeur MTU de l'interface    */
336         unsigned short          type;   /* type materiel de l'interface */
337         unsigned short          hard_header_len;        /* taille de l'en-tete materiel */
338         void                    *priv;  /* pointeur sur des donnees privees             */
339
340         struct net_device       *master; /* Pointeur sur le peripherique maitre du groupe
341                                           * dont ce peripherique est membre.
342                                           */
343
344         /* Info sur l'adresse de l'interface. */
345         unsigned char           broadcast[MAX_ADDR_LEN];        /* adresse materielle de
                                                                       diffusion generale */
346         unsigned char           dev_addr[MAX_ADDR_LEN]; /* adresse materielle         */
```

```
347         unsigned char           addr_len;       /* longueur de l'adresse physique       */
348
349         struct dev_mc_list      *mc_list;       /* Adresses mac de multidiffusion       */
350         int                     mc_count;       /* Nombre de multidiffusions installees */
351         int                     promiscuity;
352         int                     allmulti;
353
354         int                     watchdog_timeo;
355         struct timer_list       watchdog_timer;
356
357         /* Pointeurs sur des protocoles specifiques */
358
359         void                    *atalk_ptr;     /* Lien AppleTalk               */
360         void                    *ip_ptr;        /* Donnees specifiques a IPv4   */
361         void                    *dn_ptr;        /* Donnees specifiques a DECnet */
362         void                    *ip6_ptr;       /* Donnees specifiques a IPv6   */
363         void                    *ec_ptr;        /* Donnees specifiques a Econet */
364         void                    *ax25_ptr;      /* Donnees specifiques a AX.25  */
365
366         struct list_head        poll_list;      /* Lien a la liste d'election   */
367         int                     quota;
368         int                     weight;
369
370         struct Qdisc            *qdisc;
371         struct Qdisc            *qdisc_sleeping;
372         struct Qdisc            *qdisc_ingress;
373         struct list_head        qdisc_list;
374         unsigned long           tx_queue_len;   /* Nombre max de trames permis par file
                                                       d'attente */
375
376         /* Synchronisateur de chemin ingress */
377         spinlock_t              ingress_lock;
378         /* Synchronisateur de hard_start_xmit */
379         spinlock_t              xmit_lock;
380         /* id du microprocesseur entre a hard_start_xmit ou -1
381            s'il n'y en a pas.
382          */
383         int                     xmit_lock_owner;
384         /* verrou de file d'attente du peripherique */
385         spinlock_t              queue_lock;
386         /* Nombre de references a ce peripherique */
387         atomic_t                refcnt;
388         /* enregistrement/retrait retarde */
389         struct list_head        todo_list;
390         /* chaine de hachage des noms de peripherique */
391         struct hlist_node       name_hlist;
392         /* chaine de hachage des index de peripherique */
393         struct hlist_node       index_hlist;
394
395         /* etat de la machine d'enregistrement/retrait */
396         enum { NETREG_UNINITIALIZED=0,
397                NETREG_REGISTERING,      /* register_netdevice appele          */
398                NETREG_REGISTERED,       /* enregistrement complete a realiser */
399                NETREG_UNREGISTERING,    /* unregister_netdevice appele        */
400                NETREG_UNREGISTERED,     /* retrait complete a realiser        */
401                NETREG_RELEASED,         /* free_netdev appele                 */
402         } reg_state;
403
404         /* Caracteristiques du peripherique reseau */
405         int                     features;
406 #define NETIF_F_SG              1       /* Scatter/gather IO. */
407 #define NETIF_F_IP_CSUM         2       /* Doit controler la somme seulement pour TCP/UDP
```

```
                                                sur IPv4. */
408 #define NETIF_F_NO_CSUM         4       /* Ne demande pas de controle de la somme. Par
                                                exemple loopack. */
409 #define NETIF_F_HW_CSUM         8       /* Doit verifier la sommme pour tous les paquets. */
410 #define NETIF_F_HIGHDMA         32      /* Doit utiliser DMA pour la memoire haute. */
411 #define NETIF_F_FRAGLIST        64      /* Scatter/gather IO. */
412 #define NETIF_F_HW_VLAN_TX      128     /* Acceleration materielle pour une transmission
                                                VLAN */
413 #define NETIF_F_HW_VLAN_RX      256     /* Acceleration materielle pour une reception
                                                VLAN */
414 #define NETIF_F_HW_VLAN_FILTER  512     /* Filtrage des receptions sur VLAN */
415 #define NETIF_F_VLAN_CHALLENGED 1024    /* Le peripherique ne peut pas manipuler les
                                                paquets VLAN */
416 #define NETIF_F_TSO             2048    /* Peut se charger de la segmentation TCP/IP */
417 #define NETIF_F_LLTX            4096    /* LockLess TX */
418
419         /* Appele apres qu'un peripherique soit detache du reseau. */
420         void                    (*uninit)(struct net_device *dev);
421         /* Appele apres que la derniere reference utilisateur ait disparue. */
422         void                    (*destructor)(struct net_device *dev);
423
424         /* Pointeurs vers les routines de service de l'interface.      */
425         int                     (*open)(struct net_device *dev);
426         int                     (*stop)(struct net_device *dev);
427         int                     (*hard_start_xmit) (struct sk_buff *skb,
428                                                     struct net_device *dev);
429 #define HAVE_NETDEV_POLL
430         int                     (*poll) (struct net_device *dev, int *quota);
431         int                     (*hard_header) (struct sk_buff *skb,
432                                                 struct net_device *dev,
433                                                 unsigned short type,
434                                                 void *daddr,
435                                                 void *saddr,
436                                                 unsigned len);
437         int                     (*rebuild_header)(struct sk_buff *skb);
438 #define HAVE_MULTICAST
439         void                    (*set_multicast_list)(struct net_device *dev);
440 #define HAVE_SET_MAC_ADDR
441         int                     (*set_mac_address)(struct net_device *dev,
442                                                    void *addr);
443 #define HAVE_PRIVATE_IOCTL
444         int                     (*do_ioctl)(struct net_device *dev,
445                                             struct ifreq *ifr, int cmd);
446 #define HAVE_SET_CONFIG
447         int                     (*set_config)(struct net_device *dev,
448                                               struct ifmap *map);
449 #define HAVE_HEADER_CACHE
450         int                     (*hard_header_cache)(struct neighbour *neigh,
451                                                      struct hh_cache *hh);
452         void                    (*header_cache_update)(struct hh_cache *hh,
453                                                        struct net_device *dev,
454                                                        unsigned char *  haddr);
455 #define HAVE_CHANGE_MTU
456         int                     (*change_mtu)(struct net_device *dev, int new_mtu);
457
458 #define HAVE_TX_TIMEOUT
459         void                    (*tx_timeout) (struct net_device *dev);
460
461         void                    (*vlan_rx_register)(struct net_device *dev,
462                                                     struct vlan_group *grp);
463         void                    (*vlan_rx_add_vid)(struct net_device *dev,
464                                                    unsigned short vid);
```

```
465         void                    (*vlan_rx_kill_vid)(struct net_device *dev,
466                                                     unsigned short vid);
467
468         int                     (*hard_header_parse)(struct sk_buff *skb,
469                                                      unsigned char *haddr);
470         int                     (*neigh_setup)(struct net_device *dev, struct neigh_parms *);
471         int                     (*accept_fastpath)(struct net_device *, struct dst_entry*);
472 #ifdef CONFIG_NETPOLL
473         int                     netpoll_rx;
474 #endif
475 #ifdef CONFIG_NET_POLL_CONTROLLER
476         void                    (*poll_controller)(struct net_device *dev);
477 #endif
478
479         /* Chose relative aux ponts */
480         struct net_bridge_port  *br_port;
481
482 #ifdef CONFIG_NET_DIVERT
483         /* Ceci sera initialise par la routine init de chaque type d'interface */
484         struct divert_blk       *divert;
485 #endif /* CONFIG_NET_DIVERT */
486
487         /* entree classe/reseau/nom */
488         struct class_device     class_dev;
489         /* Combien de [caracteres de] remplissage ont ete ajoutes par alloc_netdev() */
490         int padded;
491 };
```

14.1.3 Partie visible

Les membres de la structure `net_device` peuvent être répartis en deux groupes : la **partie visible** par un utilisateur et la partie invisible par un utilisateur, mais évidemment visible par le noyau. La partie visible de la structure est constituée des membres qui peuvent être assignés statiquement ; toutes les structures définies dans le fichier `drivers/net/Space.c` sont initialisés de cette façon. Les autres membres ne sont pas initialisés au moment de la compilation : certains le sont au moment de l'initialisation et d'autres par le périphérique lui-même à tout moment (par exemple les statistiques sur le nombre de paquets reçus).

14.1.3.1 Nom du périphérique réseau

Sous Linux, tout périphérique réseau dispose d'un nom sans équivoque. Le tableau ci-dessous montre quelques-uns des noms actuellement utilisés :

Nom	Type de périphérique réseau
eth	Ethernet
tr	Token Ring
atm	ATM
sl	SLIP
ppp	PPP
plip	PLIP (*Parallel Line Interface Protocol*)
tunl	Tunnel IPIP
isdn	ISDN (NUMERIS en France)
dummy	Périphérique factice
lo	Périphérique en boucle (*loopback*)

Les périphériques de même type sont généralement numérotés à la file de 0 à n, par ordre

croissant, par exemple `isdn0`, `isdn1`, etc. Certains périphériques, comme le périphérique en boucle (`lo`), n'apparaissent qu'une fois et comportent par conséquent des noms fixes.

Du fait que Linux est très étroitement lié à Ethernet et que quelques fonctions du noyau en simplifient l'utilisation, certaines cartes non Ethernet se servent également de la catégorie `eth`*n* grâce à des adaptateurs simulant Ethernet.

Le schéma de désignation des périphériques réseau offre quelques avantages. Lors de la conception d'applications et l'établissement de fichiers de commandes de configuration, il est plus facile de solliciter les périphériques réseau sans avoir à connaître quel est leur fabricant et quels sont leurs paramètres (numéro d'interruption, numéro de port).

Le nom d'un périphérique réseau constitue le champ `name` du descripteur.

Le nombre maximum de caractères pour ce nom est repéré par la constante symbolique `IFNAMESIZ` (pour *Interface PHysique NAME SIZe*), définie dans le fichier en-tête `linux/include/linux/if.h`:

Code Linux 2.6.10

```
26 #define IFNAMSIZ        16
```

14.1.3.2 Champs spécifiques au matériel

Les attributs liés aux entrées-sorties correspondent aux champs:

- `mem_end` et `mem_start` de fin et de début de la mémoire partagée entre adaptateur et noyau. La zone de mémoire `[mem_start, mem_end]` désigne le tampon destiné aux paquets à envoyer ou à réceptionner.

  Ces valeurs peuvent être spécifiées sur la ligne de commande du noyau lors du démarrage du système et peuvent être retrouvées grâce à la commande `ifconfig`. On s'arrange, par convention, pour que `mem_end - mem_start` soit égal à la quantité de mémoire vive disponible.

- `base_addr` correspondant au numéro de port d'entrée-sortie du microprocesseur relié au périphérique. En général une carte réseau utilise plusieurs ports d'entrée-sortie du microprocesseur mais seul le premier est important puisque les autres sont toujours les suivants, leur nombre, quant à lui, dépendant de la carte.

  Cette valeur est en général définie au moment de la détection de la carte. La commande `ifconfig` peut également être utilisée pour afficher ou modifier cette valeur.

- `irq` correspondant au numéro d'interruption du périphérique.

  Cette valeur est en général définie au moment de la recherche du périphérique dans la routine de détection du pilote ou par une indication explicite lors du chargement du module ou du démarrage du noyau. La valeur `dev->irq` est affichée par `ifconfig` lorsque les interfaces sont listées. Elle peut être modifiée par cette commande.

- `if_port` permet de repérer le port du périphérique si celui-ci en possède plusieurs. C'est souvent le cas avec des cartes qui possèdent à la fois une connexion coaxiale (`IF_PORT_10BASE2`) et une connexion en paire torsadée (`IF_PORT_10BASET`).

  La liste des paramètres possibles est définie dans le fichier `linux/include/linux/netdevice.h`:

Code Linux 2.6.10

```
142 /* Options de selection du support. */
143 enum {
144         IF_PORT_UNKNOWN = 0,
145         IF_PORT_10BASE2,
146         IF_PORT_10BASET,
147         IF_PORT_AUI,
148         IF_PORT_100BASET,
```

```
149         IF_PORT_100BASETX,
150         IF_PORT_100BASEFX
151 };
```

– `dma` spécifie le canal d'accès direct à la mémoire si la carte réseau prend en charge le mode de transfert DMA.

14.1.3.3 État du périphérique

Le champ `state` indique l'état du périphérique. Il a été réintroduit depuis la version du noyau 2.3.43, remplaçant les champs précédents `start` (la carte réseau est ouverte), `interrupt` (le pilote traite directement une interruption) et `tbusy` (tous les tampons de paquets sont occupés).

Les champs précédemment décrits sont à présent remplacés par sept drapeaux dont les bits correspondants sont repérés par les constantes symboliques suivantes, définies dans le fichier `linux/include/linux/netdevice.h`:

Code Linux 2.6.10

```
226 /* Ces bits de drapeau sont reserves au niveau file d'attente de reseau generique,
227  * ils ne peuvent pas etre explicitement references par du code
228  * autre.
229  */
230
231 enum netdev_state_t
232 {
233         __LINK_STATE_XOFF=0,
234         __LINK_STATE_START,
235         __LINK_STATE_PRESENT,
236         __LINK_STATE_SCHED,
237         __LINK_STATE_NOCARRIER,
238         __LINK_STATE_RX_SCHED,
239         __LINK_STATE_LINKWATCH_PENDING
240 };
```

La signification des deux premiers drapeaux est la suivante:

– `__LINK_STATE_START` indique que la carte a été ouverte avec la fonction `dev_open()` que nous étudierons plus loin, c'est-à-dire qu'elle est activée et peut être utilisée. Un tel état ne signifie toutefois pas automatiquement que des paquets peuvent être émis: il est possible que tous les tampons de la carte soient occupés (voir drapeau suivant). Seul l'accès en lecture est possible sur ce drapeau, puisqu'il ne doit être modifié que par des méthodes destinées à la gestion des périphériques. La méthode `netif_running(dev)` sert à vérifier ce drapeau.

– `__LINK_STATE_XOFF` indique que la carte réseau peut accepter un tampon de socket à émettre. Ici aussi, le drapeau ne doit être accessible qu'en lecture. La méthode `netif_queue_stopped(dev)` sert à vérifier ce drapeau.

  Ce drapeau remplace le champ `dev->tbusy` des versions antérieures. Dans les versions antérieures, il existait trois situations particulières pour accéder au champ `tbusy`. Après son remplacement, les fonctions suivantes ont été introduites, améliorant ainsi nettement la clarté et le style de programmation:

  – *Clôture de la transmission*: lorsque les tampons de paquet d'une carte réseau sont occupés, la remise ultérieure de paquets à la carte était arrêtée avec `dev->busy = 1`. Désormais, il existe pour ce faire la fonction (en ligne) `netif_stop_queue(dev)`, dont le but est de positionner à 1 le bit `__LINK_STATE_XOFF`. Aucun paquet n'est ainsi retiré de la file d'attente et remis à la carte.

- *Autorisation de la transmission*: après avoir émis un paquet provenant du tampon (circulaire) de paquet, la carte réseau peut à nouveau accepter des paquets provenant du noyau. Pour ce faire, il existe la méthode `netif_start_queue(dev)` qui supprime le drapeau `__LINK_STATE_XOFF`.
- *Début de la transmission*: la méthode `netif_start_queue()` n'autorise à nouveau que la remise de tampons de socket à la carte réseau. La méthode `netif_wake_queue(dev)`, quant à elle, autorise à nouveau la remise de paquets et déclenche l'interruption logicielle `NET_TX` qui met en œuvre la remise des paquets à la carte réseau.

Le champ `interrupt` n'a aucun équivalent dans les versions ultérieures à 2.3.43.

14.1.3.4 Liste chaînée des périphériques réseau

Nous avons vu que les périphériques réseau doivent être enregistrés sous la forme d'une entité du type `struct net_device` dans une liste chaînée de nom `dev_base`. Le champ `next` permet la constitution de cette liste chaînée.

Le parcours de cette liste sera effectué par une boucle commençant par :

```
for (dev = dev_base; dev; dev = dev->next)
```

14.1.3.5 Fonction d'initialisation du périphérique

La fonction `init()` est chargée de la vérification de la présence physique d'un adaptateur réseau et de l'initialisation de sa structure `struct net_device` avec les informations de pilote appropriées.

L'argument de la fonction `init()` est un pointeur sur le descripteur de périphérique à initialiser. La valeur de renvoi est 0 ou un code d'erreur négatif (par exemple `-ENODEV` si aucun adaptateur n'a été découvert).

14.1.3.6 Liste chaînée des périphériques en attente

Le champ `next_sched` sert (utilisé deux fois uniquement dans tout le code) pour la liste chaînée des périphériques réseau en attente de réception, à travers le type structuré `struct sofnet_data` que nous étudierons plus tard.

14.1.3.7 Index d'interfaçage physique

Nous avons déjà dit qu'un périphérique réseau n'est pas repéré, contrairement aux périphériques caractère et bloc, par un nombre majeur et un nombre mineur. Cependant à chaque périphérique réseau enregistré est associé un numéro unique qui constitue son champ `ifindex`.

Lors de la génération d'un nouveau périphérique réseau, la fonction `dev_get_index()` lui assigne un index nouveau et non utilisé. En comparaison de la recherche par le nom, l'index permet une découverte plus rapide d'un périphérique dans la liste de tous les périphériques.

Le champ `iflink` indique l'index du périphérique réseau à l'aide duquel un paquet est émis. Normalement, il s'agit de l'index `ifindex`, mais pour les périphériques de tunnelisation, comme `ipip`, l'index de périphérique réseau est consigné dans `iflink` à l'aide du paquet duquel le paquet encapsulé est finalement émis. Ce champ ne nous intéressera pas dans cet ouvrage.

14.1.3.8 Statistiques

Un certain nombre d'informations (telles que le nombre total de paquets reçus par le périphérique) sont détenues dans une entité du type structuré `net_device_stats`. Celui-ci est défini

dans le fichier `linux/include/linux/netdevice.h`: Code Linux 2.6.10

```
103 /*
104  *      Statistiques sur le peripherique reseau. Analogues aux stats ether 2.0 mais
105  *      avec des compteurs d'octets.
106  */
107
108 struct net_device_stats
109 {
110         unsigned long   rx_packets;             /* paquets recus au total               */
111         unsigned long   tx_packets;             /* paquets transmis au total            */
112         unsigned long   rx_bytes;               /* octets recus au total                */
113         unsigned long   tx_bytes;               /* octets transmis au total             */
114         unsigned long   rx_errors;              /* mauvais paquets recus                */
115         unsigned long   tx_errors;              /* problemes de transmission de paquet  */
116         unsigned long   rx_dropped;             /* pas d'espace dans les tampons linux  */
117         unsigned long   tx_dropped;             /* pas d'espace disponible dans linux   */
118         unsigned long   multicast;              /* paquets multidiffusion recus         */
119         unsigned long   collisions;
120
121         /* Details sur les erreurs en reception : */
122         unsigned long   rx_length_errors;
123         unsigned long   rx_over_errors;         /* tampon de reception en anneau sature */
124         unsigned long   rx_crc_errors;          /* paquet recu avec erreur sur crc      */
125         unsigned long   rx_frame_errors;        /* erreur d'alignement dans trame recue */
126         unsigned long   rx_fifo_errors;         /* fifo de reception saturee            */
127         unsigned long   rx_missed_errors;       /* paquet manque en reception           */
128
129         /* Details sur les erreurs en transmission */
130         unsigned long   tx_aborted_errors;
131         unsigned long   tx_carrier_errors;
132         unsigned long   tx_fifo_errors;
133         unsigned long   tx_heartbeat_errors;
134         unsigned long   tx_window_errors;
135
136         /* pour cslip etc */
137         unsigned long   rx_compressed;
138         unsigned long   tx_compressed;
139 };
```

La méthode `get_stats()`, spécifique à chaque carte réseau, permet d'obtenir ces statistiques et ces informations sur la carte réseau et ses activités.

14.1.3.9 Périphériques sans fil

Les trois champs `get_wirelees_stats()`, `wireless_handlers` et `wireless_data` concernent les périphériques sans fil. Ils ne nous intéresserons pas dans cet ouvrage.

14.1.3.10 Opérations liées à Ethernet

Le champ `ethtool_ops` concerne les opérations liées à Ethernet, égal à `NULL` dans le cas d'un périphérique non Ethernet. Nous reviendrons sur le type `struct ethtool_ops` lors de l'étude de l'implémentation d'Ethernet au chapitre 16.

14.1.4 Champs généraux

14.1.4.1 Estampilles temporelles

Le champ `trans_start` spécifie le moment précis (en *jiffies*) du début de la dernière transmission. Si au bout d'un certain temps, le pilote n'a reçu aucune confirmation de l'émission du paquet

(interruption de confirmation), il prend les mesures appropriées. Le minuteur `watchdog_timer` existe dans ce but depuis la version 2.4 du noyau.

Le champ `last_rx` devrait contenir le moment précis (en *jiffies*) d'arrivée du dernier paquet. En fait il n'est pas utilisé.

14.1.4.2 Partie privée

Le champ `priv` permet d'adjoindre une structure spécifique à la carte réseau. Il s'agira en général d'une entité du type `struct net_local` qui dépend, lui aussi, du type de carte utilisée.

Ce type est, par exemple, défini dans le fichier `linux/drivers/net/3c501.h` dans le cas de la carte 3Com 501 :

Code Linux 2.6.10

```
26 /*
27  *      Informations specifiques a la carte dans dev->priv.
28  */
29
30 struct net_local
31 {
32         struct net_device_stats stats;
33         int             tx_pkt_start;   /* La longueur du paquet emis en cours. */
34         int             collisions;     /* Collisions en emission dans ce paquet */
35         int             loading;        /* Tampon d'eclaboussure qui charge les
                                              collisions */
36         int             txing;          /* Vrai si la carte est en mode d'emission */
37         spinlock_t      lock;           /* Verrou de serialisation */
38 };
```

14.1.4.3 Carte réseau maître

Certains ordinateurs possèdent plusieurs cartes réseau, par exemple les serveurs Web. Le champ `master` spécifie alors la carte maître d'un groupe de cartes réseau. Ce champ ne nous intéressera pas dans ce livre.

14.1.4.4 Liste d'élection

Le champ `poll_list` permet de se référer à la file d'attente, en réception et en émission, de la carte réseau. En effet, la carte possède souvent un tampon permettant de contenir quelques paquets (mais un seul paquet pour la carte 3Com 501), toutefois cela est insuffisant. On doit donc s'aider de la mémoire vive de l'ordinateur.

Les champs `quota` et `weight` permettent de gérer celle-ci.

Le type général `struct list_head` est défini dans le fichier en-tête `linux/include/linux/list.h`:

Code Linux 2.6.10

```
18 /*
19  * Implementation d'une liste doublement chainee simple.
20  *
21  * Certaines des fonctions internes ("__xxx") sont utiles lorsqu'on
22  * manipule les listes en entier plutot que des entrees parce que nous
23  * connaissons deja les entrees next/prev et que nous pouvons
24  * generer du meilleur code en les utilisant directement au lieu d'utiliser
25  * les routines a une entree generique.
26  */
27
28 struct list_head {
29         struct list_head *next, *prev;
30 };
```

14.1.4.5 Stratégie de mise en file d'attente

Les champs `qdisc`, `qdisc_sleeping`, `qdisc_list`, `qdisc_ingress` et `tx_queue_len` permettent la gestion des files d'attente.

Le périphérique réseau ne comporte généralement qu'une seule file d'attente et fonctionne suivant le principe du FIFO. Toutefois, il est possible de définir plusieurs files d'attente et de les utiliser grâce à une stratégie (le champ `qdisc`) particulière.

Le type `struct Qdisc` reflète la stratégie de contrôle des files d'attente du périphérique réseau en cours. Il est défini dans le fichier `linux/net/sch_generic.h`:

Code Linux 2.6.10

```
27 struct Qdisc
28 {
29         int                     (*enqueue)(struct sk_buff *skb, struct Qdisc *dev);
30         struct sk_buff *        (*dequeue)(struct Qdisc *dev);
31         unsigned                flags;
32 #define TCQ_F_BUILTIN   1
33 #define TCQ_F_THROTTLED 2
34 #define TCQ_F_INGRESS   4
35         int                     padded;
36         struct Qdisc_ops        *ops;
37         u32                     handle;
38         u32                     parent;
39         atomic_t                refcnt;
40         struct sk_buff_head     q;
41         struct net_device       *dev;
42         struct list_head        list;
43
44         struct gnet_stats_basic bstats;
45         struct gnet_stats_queue qstats;
46         struct gnet_stats_rate_est      rate_est;
47         spinlock_t              *stats_lock;
48         struct rcu_head         q_rcu;
49         int                     (*reshape_fail)(struct sk_buff *skb,
50                                         struct Qdisc *q);
51
52         /* Ce champ est desavoue mais il est encore utilise par CBQ
53          * et il restera en vie jusqu'a ce qu'une meilleure solution soit inventee.
54          */
55         struct Qdisc            *__parent;
56 };
```

La signification des champs est la suivante:

- La méthode `enqueue()` permet d'ajouter un élément à la file d'attente.
- La méthode `dequeue()` permet de retirer un élément de la file d'attente.
- Le champ `flags` est un champ de bits dont trois seulement sont significatifs, repérés par les constantes symboliques citées.
- Le champ `padded` spécifie le rembourrage utilisé.
- Nous allons revenir ci-dessous sur l'ensemble des opérations, le champ `ops`, permises sur une instance de cette structure.
- La poignée `handle` permet de repérer l'instance.
- Le parent `parent` permet de se repérer dans une liste chaînée.
- Le compteur de référence `refcnt` permet de ne pas détruire l'instance si quelqu'un y fait encore référence.
- Le champ `q` permet de pointer sur le premier élément de la file d'attente.
- Le champ `dev` spécifie la carte réseau associée à la file d'attente.

– Les champ `bstats`, `qstats` et `rate_est` permettent de détenir des informations.

Code Linux 2.6.10

Les types `struct gnet_stats_basic`, `struct gnet_stats_queue` et `struct gnet_stats_rate_est` sont définis dans le fichier en-tête `linux/include/linux/gen_stats.h`:

```
16 /**
17  * struct gnet_stats_basic - statistiques sur les octets/paquets
18  * @bytes : nombre d'octets vus
19  * @packets : nombre de paquets vus
20  */
21 struct gnet_stats_basic
22 {
23         __u64   bytes;
24         __u32   packets;
25 };
26
27 /**
28  * struct gnet_stats_rate_est - estimateur de taux
29  * @bps : taux d'octets en cours
30  * @pps : taux de paquet en cours
31  */
32 struct gnet_stats_rate_est
33 {
34         __u32   bps;
35         __u32   pps;
36 };
37
38 /**
39  * struct gnet_stats_queue - statistiques de file d'attente
40  * @qlen : longueur de la file d'attente
41  * @backlog : taille maximale d'elements que peut contenir la file d'attente
42  * @drops : nombre de paquets ecartes
43  * @requeues : nombre de remises dans la file d'attente
44  * @overlimits : nombre de mises en file d'attente depassant la limite
45  */
46 struct gnet_stats_queue
47 {
48         __u32   qlen;
49         __u32   backlog;
50         __u32   drops;
51         __u32   requeues;
52         __u32   overlimits;
53 };
```

– Le champ `stats_lock` permet de verrouiller les informations statistiques.

– Le champ `q_rcu` sert au mécanisme de mise à jour en copie et écriture d'exlusion mutuelle (*Read-Copy Update* en anglais).

Code Linux 2.6.10

Le type général `struct rcu_head` est défini dans le fichier en-tête `linux/include/linux/rcupdate.h`:

```
45 /**
46  * struct rcu_head - structure de rappel a utiliser avec la RCU
47  * @next : requete de mise a jour suivante dans une liste
48  * @func : fonction de mise a jour en cours a appeler apres la periode de grace.
49  */
50 struct rcu_head {
51         struct rcu_head *next;
52         void (*func)(struct rcu_head *head);
53 };
```

– La méthode `reshape_fail()` permet d'intervenir en cas d'échec.

– Le champ `__parent` n'est plus utilisé, comme indiqué.

Le type `struct Qdisc_ops` des opérations permises sur une instance du type défini ci-dessus est également défini dans le fichier `linux/net/sch_generic.h`:

Code Linux 2.6.10

```
58 struct Qdisc_class_ops
59 {
60         /* Manipulation du qdisc fils */
61         int                     (*graft)(struct Qdisc *, unsigned long cl,
62                                         struct Qdisc *, struct Qdisc **);
63         struct Qdisc *          (*leaf)(struct Qdisc *, unsigned long cl);
64
65         /* Routines de manipulation de la classe */
66         unsigned long           (*get)(struct Qdisc *, u32 classid);
67         void                    (*put)(struct Qdisc *, unsigned long);
68         int                     (*change)(struct Qdisc *, u32, u32,
69                                         struct rtattr **, unsigned long *);
70         int                     (*delete)(struct Qdisc *, unsigned long);
71         void                    (*walk)(struct Qdisc *, struct qdisc_walker * arg);
72
73         /* Manipulation du filtre */
74         struct tcf_proto **     (*tcf_chain)(struct Qdisc *, unsigned long);
75         unsigned long           (*bind_tcf)(struct Qdisc *, unsigned long,
76                                         u32 classid);
77         void                    (*unbind_tcf)(struct Qdisc *, unsigned long);
78
79         /* Specifique a rtnetlink */
80         int                     (*dump)(struct Qdisc *, unsigned long,
81                                         struct sk_buff *skb, struct tcmsg*);
82         int                     (*dump_stats)(struct Qdisc *, unsigned long,
83                                         struct gnet_dump *);
84 };
85
86  struct Qdisc_ops
87  {
88          struct Qdisc_ops        *next;
89          struct Qdisc_class_ops  *cl_ops;
90          char                    id[IFNAMSIZ];
91          int                     priv_size;
92
93          int                     (*enqueue)(struct sk_buff *, struct Qdisc *);
94          struct sk_buff *        (*dequeue)(struct Qdisc *);
95          int                     (*requeue)(struct sk_buff *, struct Qdisc *);
96          unsigned int            (*drop)(struct Qdisc *);
97
98          int                     (*init)(struct Qdisc *, struct rtattr *arg);
99          void                    (*reset)(struct Qdisc *);
100         void                    (*destroy)(struct Qdisc *);
101         int                     (*change)(struct Qdisc *, struct rtattr *arg);
102
103         int                     (*dump)(struct Qdisc *, struct sk_buff *);
104         int                     (*dump_stats)(struct Qdisc *, struct gnet_dump *);
105
106         struct module           *owner;
107 };
```

14.1.4.6 Synchronisation de l'émission

Les champs `xmit_lock`, `xmit_lock_owner` et `queue_lock` permettent de synchroniser l'émission des trames:

– Le verrou rotatif `xmit_lock` est utilisé pour éviter des appels multiples simultanés à la fonction `hard_start_xmit()` du pilote de périphérique.

- Le champ `xmit_lock_owner` est le numéro du microprocesseur qui a obtenu `xmit_lock`, utile dans le cas où il y a plusieurs microprocesseurs. Si aucun microprocesseur n'émet momentanément, ce champ prend la valeur -1.
- Le verrou rotatif `queue_lock` est utilisé pour éviter des appels multiples simultanés à la file d'attente.

14.1.4.7 Installation/désinstallation

Le champ `refcnt` permet de maintenir le nombre de renvois à cette carte, ce qui évite de la désinstaller si celui-ci est non nul.

La liste `todo_list` permet de savoir si la carte est installée ou non.

Les chaînes de hachage `name_hlist` et `index_hlist` permettent de repérer plus facilement l'instance.

Code Linux 2.6.10

Le type général `struct hlist_node` est défini dans le fichier `linux/include/linux/list.h`:

```
503 struct hlist_node {
504         struct hlist_node *next, **pprev;
505 };
```

L'état `reg_state` (pour *REGister STATE*) permet de savoir où l'on en est du point de vue de l'installation de la carte: non initialisée, en train d'être enregistrée, enregistrée, en train d'être désintallée, désinstallée ou libérée.

14.1.5 Membres concernant la sous-couche MAC

14.1.5.1 Drapeaux de la sous-couche MAC

Les drapeaux `flags` et `gflags` sont des champs de bits, décrivant d'une part les propriétés de la carte réseau et transmettant l'état temporaire (par exemple `IFF_UP`). Les bits sont repérés par des constantes symboliques définies dans le fichier `linux/include/linux/if.h`:

Code Linux 2.6.10

```
29 /* Drapeaux standard de l'interface (netdevice->flags). */
30 #define IFF_UP          0x1             /* l'interface est active                */
31 #define IFF_BROADCAST   0x2             /* adresse de diffusion generale valide  */
32 #define IFF_DEBUG       0x4             /* en phase de debogage                  */
33 #define IFF_LOOPBACK    0x8             /* est un reseau loopback                */
34 #define IFF_POINTOPOINT 0x10            /* l'interface a un lien p-p             */
35 #define IFF_NOTRAILERS  0x20            /* eviter l'utilisation de remorques     */
36 #define IFF_RUNNING     0x40            /* ressources allouees                   */
37 #define IFF_NOARP       0x80            /* pas de protocole ARP                  */
38 #define IFF_PROMISC     0x100           /* recoit tous les paquets               */
39 #define IFF_ALLMULTI    0x200           /* recoit tous les paquets multidiffusion */
40
41 #define IFF_MASTER      0x400           /* maitre d'un equilibreur de chargement  */
42 #define IFF_SLAVE       0x800           /* esclave d'un equilibreur de chargement  */
43
44 #define IFF_MULTICAST   0x1000          /* Supporte la multidiffusion            */
45
46 #define IFF_VOLATILE
       (IFF_LOOPBACK|IFF_POINTOPOINT|IFF_BROADCAST|IFF_MASTER|IFF_SLAVE|IFF_RUNNING)
47
48 #define IFF_PORTSEL     0x2000          /* peut positionner le type de support    */
49 #define IFF_AUTOMEDIA   0x4000          /* auto-selection du support active       */
50 #define IFF_DYNAMIC     0x8000          /* peripherique telephonique avec adresse
                                           changeante                             */
```

avec un préfixe `IFF` pour *InterFace Flag*. La signification lorsque le bit est à 1 est la suivante:

- `IFF_UP`: le noyau positionne ce bit (qui peut être lu, mais pas écrit par le périphérique) lorsque l'interface est active et prête à transférer des paquets.

- IFF_BROADCAST: ce bit indique que la carte réseau permet la diffusion générale et que l'adresse de diffusion générale est valide.
- IFF_DEBUG: ce bit indique qu'on est en mode de débogage. Il peut être positionné ou remis à zéro grâce à la commande utilisateur `ioctl()`.
- IFF_LOOPBACK: ce bit ne doit être positionné que pour l'interface en boucle.
- IFF_POINTOPOINT: ce bit signale que l'interface est connectée à une liaison point-à-point. Il est positionné par la commande utilisateur `ifconfig`.
- IFF_NOTRAILERS: ce bit n'est pas utilisé par Linux mais il existe pour des raisons de compatibilité avec BSD.
- IFF_RUNNING: ce bit indique que l'interface est active et est en train de travailler. Il est présent pour des raisons de compatibilité avec BSD mais Linux en fait peu d'usage.
- IFF_NOARP: ce bit signale que l'interface ne peut pas effectuer d'ARP, ne serait-ce que, par exemple, parce que les interfaces point-à-point n'ont pas besoin d'effectuer d'ARP.
- IFF_PROMISC: ce bit est positionné pour activer le mode promiscuité.
- IFF_MULTICAST: ce bit est positionné lorsque l'interface peut effectuer des multidiffusions.
  Il est positionné par défaut lors de l'initialisation. Il doit donc être remis à zéro pour les cartes qui ne permettent pas la diffusion restreinte (ce qui est le cas de la carte 3Com 501, par exemple).
- IFF_ALLMULTI: ce bit indique à l'interface qu'il faut recevoir tous les paquets de diffusion restreinte. Il ne peut être positionné, par le noyau, que si IFF_MULTICAST l'est également. Ce bit ne peut qu'être lu par la carte.
- IFF_MASTER et IFF_SLAVE: ces bits ne sont utilisés que par le programme. Le pilote de périphérique ne s'en sert pas.
- IFF_PORTSEL et IFF_AUTOMEDIA: ces bits signalent que le périphérique est capable de passer d'un type de support à un autre, par exemple de paire torsadée à câble coaxial. Si IFF_AUTOMEDIA est positionné, le périphérique sélectionne le support automatiquement.
- IFF_DYNAMIC: ce bit indique que l'adresse de la carte peut être changée. Ce mode est utilisé par les périphériques de connexion par appel téléphonique.

Lorsqu'un programme change IFF_UP, la méthode de périphérique `open()` ou `stop()` est appelée. Lorsqu'un drapeau est modifié, la méthode `set_multicast_list()` est appelée. Si le pilote de périphérique doit effectuer une action suite à la modification d'un drapeau, cette action doit donc être placée dans `set_multicast_list()`. Par exemple, lorsque IFF_PROMISC est positionné ou remis à zéro, `set_multicast_list()` doit le notifier au filtre matériel du périphérique.

Ces commutateurs peuvent être positionnés à l'aide de la commande `ifconfig`.

Les positions des drapeaux de `priv_flags` sont également définies dans le fichier `linux/include/linux/if.h`:

Code Linux 2.6.0

```
52 /* Drapeaux prives (de l'utilisateur) pour l'interface (netdevice->priv_flags). */
53 #define IFF_802_1Q_VLAN 0x1             /* peripherique VLAN 802.1Q.    */
54 #define IFF_EBRIDGE     0x2             /* peripherique pont Ethernet.  */
55
56 #define IF_GET_IFACE    0x0001          /* pour les requetes uniquement */
57 #define IF_GET_PROTO    0x0002
58
59 /* Pour les definitions voir hdlc.h */
60 #define IF_IFACE_V35    0x1000          /* interface serie V.35         */
61 #define IF_IFACE_V24    0x1001          /* interface serie V.24         */
```

```
62 #define IF_IFACE_X21    0x1002          /* interface serie X.21         */
63 #define IF_IFACE_T1     0x1003          /* interface serie telco T1     */
64 #define IF_IFACE_E1     0x1004          /* interface serie telco E1     */
65 #define IF_IFACE_SYNC_SERIAL 0x1005     /* ne peut pas etre positionne par logiciel */
66 #define IF_IFACE_X21D   0x1006          /* X.21 Dual Clocking (FarSite) */
67
68 /* Pour les definitions voir hdlc.h */
69 #define IF_PROTO_HDLC   0x2000          /* protocole HDLC brut          */
70 #define IF_PROTO_PPP    0x2001          /* protocole PPP                */
71 #define IF_PROTO_CISCO  0x2002          /* protocole HDLC Cisco         */
72 #define IF_PROTO_FR     0x2003          /* protocole Frame Relay        */
73 #define IF_PROTO_FR_ADD_PVC 0x2004      /*    Creer   FR PVC            */
74 #define IF_PROTO_FR_DEL_PVC 0x2005      /*    Retirer FR PVC            */
75 #define IF_PROTO_X25    0x2006          /* X.25                         */
76 #define IF_PROTO_HDLC_ETH 0x2007        /* HDLC brut, emulation d'Ethernet */
77 #define IF_PROTO_FR_ADD_ETH_PVC 0x2008  /*  Creer   FR pont-Ethernet PVC */
78 #define IF_PROTO_FR_DEL_ETH_PVC 0x2009  /*  Retirer FR pont-Ethernet PVC */
79 #define IF_PROTO_FR_PVC 0x200A          /* pour lire le statut PVC       */
80 #define IF_PROTO_FR_ETH_PVC 0x200B
81 #define IF_PROTO_RAW    0x200C          /* RAW Socket                   */
```

14.1.5.2 Unité de transfert d'information maximale

Chaque type de réseau possède une **unité de transfert d'information maximale**, ou **MTU** (pour l'anglais *Maximum Transfert Unit*), dans laquelle doit s'insérer chaque trame. Le champ `mtu` précise cette valeur pour le réseau auquel la carte est reliée.

Par exemple Ethernet a une MTU de 1 500 octets.

14.1.5.3 Type matériel

Le type matériel de l'interface est utilisé par le protocole de résolution d'adresse **ARP** pour déterminer quelle sorte d'adresse physique est supportée par le périphérique réseau. La valeur pour les cartes Ethernet est `ARPHDR_ETHER`.

Les valeurs des types matériels ont été codifiées dans [RFC 1700] concernant le protocole ARP. Linux définit des constantes supplémentaires qui ne sont pas codifiées dans cette RFC.

Les types pris en charge par Linux sont repérés par l'une des constantes symboliques définies dans le fichier en-tête `linux/include/linux/if_arp.h`:

Code Linux 2.6.10

```
28 /* Identificateurs MATERIELS de protocole ARP. */
29 #define ARPHRD_NETROM   0               /* de KA9Q : pseudo NET/ROM           */
30 #define ARPHRD_ETHER    1               /* Ethernet 10Mbps                    */
31 #define ARPHRD_EETHER   2               /* Ethernet Experimental              */
32 #define ARPHRD_AX25     3               /* AX.25 niveau 2                      */
33 #define ARPHRD_PRONET   4               /* PROnet token ring                  */
34 #define ARPHRD_CHAOS    5               /* Chaosnet                           */
35 #define ARPHRD_IEEE802  6               /* IEEE 802.2 Ethernet/TR/TB          */
36 #define ARPHRD_ARCNET   7               /* ARCnet                             */
37 #define ARPHRD_APPLETLK 8               /* APPLEtalk                          */
38 #define ARPHRD_DLCI     15              /* Frame Relay DLCI                   */
39 #define ARPHRD_ATM      19              /* ATM                                */
40 #define ARPHRD_METRICOM 23              /* Metricom STRIP (nouvel id IANA)    */
41 #define ARPHRD_IEEE1394 24              /* IEEE 1394 IPv4 - RFC 2734          */
42 #define ARPHRD_EUI64    27              /* EUI-64                             */
43 #define ARPHRD_INFINIBAND 32            /* InfiniBand                         */
44
45 /* Types muets pour du materiel non ARP */
46 #define ARPHRD_SLIP     256
47 #define ARPHRD_CSLIP    257
48 #define ARPHRD_SLIP6    258
```

```
49 #define ARPHRD_CSLIP6   259
50 #define ARPHRD_RSRVD    260             /* Notional KISS type                */
51 #define ARPHRD_ADAPT    264
52 #define ARPHRD_ROSE     270
53 #define ARPHRD_X25      271             /* CCITT X.25                        */
54 #define ARPHRD_HWX25    272             /* Cartes avec X.25 dans le firmware   */
55 #define ARPHRD_PPP      512
56 #define ARPHRD_CISCO    513             /* HDLC Cisco                        */
57 #define ARPHRD_HDLC     ARPHRD_CISCO
58 #define ARPHRD_LAPB     516             /* LAPB                              */
59 #define ARPHRD_DDCMP    517             /* protocole DDCMP de Digital        */
60 #define ARPHRD_RAWHDLC  518             /* HDLC brut                         */
61
62 #define ARPHRD_TUNNEL   768             /* Tunnel IPIP                       */
63 #define ARPHRD_TUNNEL6  769             /* Tunnel IP6IP6                     */
64 #define ARPHRD_FRAD     770             /* Frame Relay Access Device         */
65 #define ARPHRD_SKIP     771             /* SKIP vif                          */
66 #define ARPHRD_LOOPBACK 772             /* Peripherique Loopback             */
67 #define ARPHRD_LOCALTLK 773             /* Peripherique Localtalk            */
68 #define ARPHRD_FDDI     774             /* Fiber Distributed Data Interface  */
69 #define ARPHRD_BIF      775             /* AP1000 BIF                        */
70 #define ARPHRD_SIT      776             /* Peripherique sit0 - IPv6-in-IPv4  */
71 #define ARPHRD_IPDDP    777             /* Tunellisation IP sur DDP          */
72 #define ARPHRD_IPGRE    778             /* GRE sur IP                        */
73 #define ARPHRD_PIMREG   779             /* PIMSM register interface          */
74 #define ARPHRD_HIPPI    780             /* High Performance Parallel Interface */
75 #define ARPHRD_ASH      781             /* Nexus 64Mbps Ash                  */
76 #define ARPHRD_ECONET   782             /* Acorn Econet                      */
77 #define ARPHRD_IRDA     783             /* Linux-IrDA                        */
78 /* ARP fonctionne differemment sur des supports FC differents ... donc  */
79 #define ARPHRD_FCPP     784             /* Point to point fibrechannel       */
80 #define ARPHRD_FCAL     785             /* Fibrechannel arbitrated loop      */
81 #define ARPHRD_FCPL     786             /* Fibrechannel public loop          */
82 #define ARPHRD_FCFABRIC 787             /* Fibrechannel fabric               */
83         /* 787->799 reserve pour les types de media fibrechannel */
84 #define ARPHRD_IEEE802_TR 800           /* Ident de type magique pour tr     */
85 #define ARPHRD_IEEE80211 801            /* IEEE 802.11                       */
86 #define ARPHRD_IEEE80211_PRISM 802      /* IEEE 802.11 + en-tete Prism2      */
87
88 #define ARPHRD_VOID       0xFFFF        /* Type vide, rien n'est connu       */
89 #define ARPHRD_NONE       0xFFFE        /* Longueur d'en-tete nulle          */
```

Seuls quelques types (ARPHRD_ETHER, ARPHRD_IEEE802, ARPHRD_SLIP, ARPHRD_PPP, ARPHRD_LOOPBACK et ARPHRD_VOID) nous intéresserons dans ce livre.

14.1.5.4 Longueur de l'en-tête matériel

Le champ hard_header_len indique la longueur de l'en-tête et du suffixe d'une trame, c'est-à-dire ce qui n'est pas occupé par les données, en ne comptant pas les emplacements occupés par les adresses de l'émetteur et du destinataire.

Par exemple, pour Ethernet on a ETH_HLEN, que nous reverrons dans le chapitre 16, égal à 14, correspondant à 8 octets de préambule, 2 octets de type et 4 octets de CRC.

14.1.5.5 Adresses de l'interface

L'adresse physique de la carte, le champ dev_addr, est seulement structurée par sa longueur maximale. La constante symbolique MAX_ADDR_LEN est définie dans le fichier linux/include/linux/netdevice.h:

Code Linux 2.6.10

```
74 #define MAX_ADDR_LEN    32              /* Plus grande longueur d'adresse physique */
```

ce qui suffit pour les 48 bits (ou six octets) d'une adresse physique MAC.

La longueur effective de l'adresse, le champ `addr_len`, prendra la valeur `ETH_LEN`, c'est-à-dire 6, dans le cas d'Ethernet.

L'adresse physique doit être lue sur le périphérique par une méthode qui dépend du type de périphérique.

Ces valeurs sont fixées, à l'aide de la commande `ifconfig`, lors de l'activation d'une carte réseau.

14.1.5.6 Attribut lié à la diffusion générale

L'adresse de diffusion générale, le champ `broadcast`, dépend du protocole. Rappelons que tous les bits de l'adresse de diffusion générale sont égaux à 1 dans le cas d'Ethernet, c'est-à-dire qu'elle est constituée de six octets de valeur `FFh`.

14.1.5.7 Attributs liés à la multidiffusion

Les adresses de multidiffusion appartiennent à une liste chaînée du type `struct dev_mc_list`. Celui-ci est défini dans le fichier `linux/include/linux/netdevice.h`:

Code Linux 2.6.10

```
179 /*
180  *      Nous marquons la multidiffusion avec ces structures.
181  */
182
183 struct dev_mc_list
184 {
185         struct dev_mc_list      *next;
186         __u8                    dmi_addr[MAX_ADDR_LEN];
187         unsigned char           dmi_addrlen;
188         int                     dmi_users;
189         int                     dmi_gusers;
190 };
```

Le nombre d'adresses dans cette liste est indiqué par le champ `mc_count`.

Si la carte réseau reçoit un paquet dont l'adresse de destination est contenue dans la liste `dev_mc_list`, elle doit le transmettre aux couches supérieures. Les adresses de cette liste sont transmises à la carte à l'aide de la méthode du pilote `set_multicast_list()`.

Le champ `allmulti` spécifie si la carte réseau accepte ou non les messages envoyés en multidiffusion. Il est manipulé à travers la fonction `dev_set_allmulti()`, définie dans le fichier `linux/net/core/dev.c`:

Code Linux 2.6.10

```
2176 /**
2177  *      dev_set_allmulti        - met a jour le compteur allmulti sur un peripherique
2178  *      @dev : peripherique
2179  *      @inc : modificateur
2180  *
2181  *      Active ou inhibe la reception de toutes les trames de multidiffusion pour un
2182  *      peripherique. Tant que le compteur du peripherique reste en dessous de zero,
2183  *      l'interface reste a l'ecoute de toutes les interfaces. Une fois qu'il atteint zero,
2184  *      le peripherique revient a l'operation normale de filtrage. Une valeur @inc negative
2185  *      est utilisee pour ecarter le compteur lorsqu'on libere une ressource necessaitant
              toutes les adresses de multidiffusion.
2186  */
2187
2188 void dev_set_allmulti(struct net_device *dev, int inc)
```

14.1.5.8 Acceptation de paquets destinés à d'autres hôtes

Le champ `promiscuity` spécifie si la carte doit accepter les trames destinées à d'autres ordinateurs, autrement dit si on se trouve dans le mode spécial de promiscuité.

Il est manipulé à travers la fonction `dev_set_promiscuity()`, définie dans le fichier `linux/net/core/dev.c`:

Code Linux 2.6.10

```
2151 /**
2152  *      dev_set_promiscuity     - met a jour le compteur de promiscuite sur un peripherique
2153  *      @dev : peripherique
2154  *      @inc : modificateur
2155  *
2156  *      Active ou inhibe la promiscuite pour un peripherique. Tant que le compteur du
2157  *      peripherique reste en-dessous de zero, l'interface admet la promiscuite.
2158  *      Des qu'il atteint zero, le peripherique revient a l'operation normale de
2159  *      filtrage. Une valeur negative de inc est utilisee pour inhiber le
              mode promiscuite du peripherique.
2160  */
2161 void dev_set_promiscuity(struct net_device *dev, int inc)
```

14.1.5.9 Attributs liés aux délais

La carte réseau utilise un certain nombre de temporisateurs, pour déterminer si le délai s'est écoulé depuis l'expédition d'une trame par exemple. Ils sont placés dans une liste, le champ `watchdog_timer`.

Le délai minimum (en *jiffies*) que l'on doit attendre avant que la couche réseau décide qu'une émission a échouée fait l'objet du champ `watchdog_timeo`.

La routine de traitement `dev_watchdog()` vérifie si, depuis la dernière émission d'un paquet, `watchdog_timeo` unités de temps se sont écoulées. Si tel est le cas, des problèmes sont apparus au moment de l'émission du dernier paquet et la carte réseau nécessite une vérification. On appelle pour ce faire la méthode de pilote `tx_timeout()`.

14.1.5.10 Nombre maximum de trames en émission

Le champ `tx_queue_len` spécifie le nombre maximum de trames qui peuvent être placées dans la file d'attente en émission du périphérique. Cette valeur est positionnée à 100 dans le cas d'Ethernet mais on peut la changer si nécessaire.

Il ne faut pas confondre `tx_queue_len` avec les tampons de la carte réseau proprement dite. Normalement une carte réseau dispose elle-même d'un tampon circulaire supplémentaire pour 16 ou 32 paquets (mais seulement 1 pour la carte 3Com 501).

14.1.6 Membres concernant la couche réseau

A priori la couche réseau ne devrait rien avoir à faire avec le pilote de périphérique. Cependant certaines cartes prennent en charge certains protocoles de la couche réseau.

14.1.6.1 Pointeurs sur des protocoles spécifiques

Ces protocoles nécessitent des données supplémentaires. C'est l'objet des six champs `atalk_ptr`, `ip_ptr`, `dn_ptr`, `ip6_ptr`, `ec_ptr` et `ax25_ptr` concernant respectivement Appletalk, IPv4, DECnet, IPv6, Econet et AX.25.

Si la carte réseau est entre autre configurée pour le protocole Internet, `ip_ptr` renvoie à une structure du type `in_device` dans laquelle sont gérées les informations et les paramètres de configuration de l'instance IP en question. La structure `in_device` gère par exemple la liste des

adresses IP de la carte réseau, la liste des groupes de multidiffusion IP ainsi que les paramètres du protocole ARP.

14.1.6.2 Comportement à l'égard de la somme de contrôle

Le champ de bits `features` permet de connaître les caractéristiques de la carte réseau : ne peut vérifier les sommes de contrôle que pour TCP/UDP sur IPv4, n'exige pas de somme de contrôle, peut vérifier la somme de contrôle de tous les paquets.

Code Linux 2.6.10

Un commentaire se trouve dans le fichier en-tête `linux/include/linux/skbuff.h`:

```
73  *      Le peripherique doit montrer ses possibilites dans dev->features, positionne
74  *      au moment du demarrage du peripherique.
75  *      NETIF_F_HW_CSUM - c'est un peripherique intelligent, il est capable de calculer
76  *                        la somme de controle dans tous les cas.
77  *      NETIF_F_NO_CSUM - loopback ou simple support de saut digne de confiance.
78  *      NETIF_F_IP_CSUM - le peripherique est bete. Il est seulement capable de
                              calculer la somme de controle
79  *                        pour TCP/UDP sur IPv4. Soupir. Les vendeurs aiment cette
80  *                        facon pour une raison inconnue. Quoique, voir le commentaire
81  *                        ci-dessus sur CHECKSUM_UNNECESSARY. 8)
82  *
83  *      Des questions ? Pas de question, bien.              --ANK
84  */
```

14.1.7 Méthodes du pilote de carte réseau

Il existe un certain nombre de fonctions permettant la gestion de la carte, ces fonctions dépendant du type de celle-ci, dont le nom explique le comportement. Certaines de ces méthodes peuvent être laissées à `NULL`.

Quelques-unes de ces méthodes dépendent du matériel de la carte réseau, aussi sont-elles initialisées par la fonction d'initialisation du pilote de la carte réseau. Les autres fonctions sont spécifiques au protocole MAC de la carte réseau employée et on peut les initialiser à l'aide de méthodes spécialisées, comme `eth_setup()` par exemple.

14.1.7.1 Méthodes dépendant du matériel

Les méthodes dépendant du matériel sont les suivantes :

- `uninit()` est appelée lors de la déconnexion d'une carte réseau (*via* la fonction `unregister_netdevice()`). Des fonctions spécifiques au pilote éventuellement requises lors du retrait d'un périphérique réseau peuvent être appelées. Cette méthode n'est présente que depuis la version 2.4 du noyau Linux et aucun pilote ne l'utilise actuellement.
- `destructor()` est appelée lorsque la dernière référence à un périphérique réseau a été éliminée, c'est-à-dire lorsque aucune instance de protocole ou d'autres composants ne renvoie plus au noyau Linux sur la structure `net_device`. Cette fonction permet donc de mettre en œuvre des tâches de nettoyage comme la libération de mémoire ou autre opération similaire. Également récente, aucun pilote ne l'utilise actuellement.
- `open()` ouvre (active) une carte réseau. Au moment de l'activation, les ressources système nécessaires (ports d'entrée-sortie, IRQ, DMA, etc.) sont demandées (et marquées comme occupées), le périphérique est démarré et le compteur d'utilisation du module est incrémenté. Seuls les périphériques préalablement enregistrés peuvent être ouverts.

- `stop()` met fin à l'activité d'une carte réseau et libère les ressources système occupées. Le périphérique réseau n'est dorénavant plus actif ; cependant, il se trouve encore dans la liste des périphériques réseau enregistrés.
- `hard_start_xmit()` initie la transmission d'un paquet. En cas de succès (le paquet a été transmis à la carte mais on ne sait pas s'il est parvenu au destinataire), la méthode revient à l'état antérieur avec une valeur de renvoi 0, autrement avec 1.
- `set_multicast_list()` est appelée soit lorsque la liste de multidiffusion change, soit lorsque le drapeau afférent (`IFF_MULTICAST`) change. Elle transmet la liste des adresses MAC de multidiffusion à la carte réseau qui doit réceptionner les paquets.
- `do_ioctl()` permet d'implémenter la commande `ioctl()` spécifique à l'interface.
- `set_config()` change la configuration de l'interface. Cette possibilité peut être utilisée par l'administrateur système. Les pilotes pour les cartes modernes n'ont pas besoin d'implémenter cette méthode en général.
- `tx_timeout()` est appelée lorsqu'on considère que la transmission d'un paquet a échouée parce que le délai s'est écoulé.

14.1.7.2 Méthodes dépendant du protocole MAC

Les méthodes dépendant du protocole MAC sont les suivantes :

- `hard_header()` permet de construire l'en-tête matériel à partir des adresses source et de destination qui ont été récupérées antérieurement.

  La fonction `eth_header()` est l'instance par défaut pour les cartes Ethernet.
- `rebuild_header()` est utilisée pour reconstruire l'en-tête matériel avant qu'un paquet ne soit transmis.

  Dans les versions de Linux antérieures à 2.4, cette méthode était le point d'entrée dans le protocole ARP. Depuis, grâce à l'adaptation au cache `neighbour`, il devrait déjà exister un en-tête matériel enregistré. Par conséquent cette méthode n'est plus appelée que pour de fausses informations dans le cache d'en-tête matériel.
- `set_mac_address()` sert à modifier l'adresse matérielle d'une carte réseau si l'interface permet le changement de celle-ci.
- `hard_header_cache()` est appelée pour remplir une entrée de cache d'en-tête matériel, de type `struct hh_cache` étudié dans la section suivante, avec les résultats obtenus par une requête ARP. Elle remplit un en-tête de trame à l'aide des données transmises. On peut ainsi accéder, lors des processus d'émission suivants, à un en-tête de trame préfabriqué.
- `header_cache_update()` met à jour l'adresse de destination de l'entrée de cache d'en-tête matériel, de type `struct hh_cache`.
- `change_mtu()` modifie l'unité maximale de transfert (MTU) d'un périphérique réseau et met en œuvre les modifications nécessaires.
- `hard_header_parse(skb, haddr)` extrait l'adresse source du paquet contenu dans le paramètre `skb` et la copie dans le tampon situé à l'adresse `haddr`. Elle renvoie la longueur de cette adresse.

14.1.8 Autres champs

14.1.8.1 Champs relatifs aux ponts

Le champ `br_port` indique le numéro de port du pont associé à la carte. Le type `struct net_bridge_port`, défini dans le fichier `linux/net/bridge/br_private.h`, ne nous intéressera pas dans cet ouvrage.

14.1.8.2 Trame de déroutement

Le type `struct divert_blk` du champ `divert` est défini dans le fichier `linux/include/linux/divert.h`:

Code Linux 2.6.10

```
14 #define MAX_DIVERT_PORTS        8       /* Nombre max de ports a derouter (tcp, udp) */
15
16 /* Protocoles pouvant etre deroutes */
17 #define DIVERT_PROTO_NONE       0x0000
18 #define DIVERT_PROTO_IP         0x0001
19 #define DIVERT_PROTO_ICMP       0x0002
20 #define DIVERT_PROTO_TCP        0x0004
21 #define DIVERT_PROTO_UDP        0x0008
22
23 /*
24  *      Ceci est un bloc d'option Ethernet Frame Diverter
25  */
26 struct divert_blk
27 {
28         int           divert;  /* sommes-nous actif */
29         unsigned int  protos;  /* protocoles */
30         u16           tcp_dst[MAX_DIVERT_PORTS]; /* ports de dst tcp specifiques a derouter */
31         u16           tcp_src[MAX_DIVERT_PORTS]; /* ports src tcp specifiques a derouter    */
32         u16           udp_dst[MAX_DIVERT_PORTS]; /* ports de dst udp specifiques a derouter */
33         u16           udp_src[MAX_DIVERT_PORTS]; /* ports src udp specifiques a derouter    */
34 };
```

14.1.8.3 Classe du périphérique

Les périphériques peuvent appartenir à une classe. Le type général `struct class_device` du champ `class_dev` était défini dans le fichier `linux/include/linux/device.h` dans le cas du noyau 2.6.0. Il ne l'est plus pour le noyau 2.6.10, ce qui montre qu'il n'est plus utilisé.

14.2 Entrée de cache d'en-tête matériel

Pour éviter d'avoir à recomposer à chaque fois l'en-tête matériel de la trame, on utilise une entrée de cache d'en-tête matériel. Celle-ci est une entité du type `struct hh_cache`, défini dans le fichier `linux/include/linux/netdevice.h`:

Code Linux 2.6.10

```
192 struct hh_cache
193 {
194         struct hh_cache *hh_next;       /* Entree suivante                  */
195         atomic_t        hh_refcnt;      /* nombre d'utilisateurs            */
196         unsigned short  hh_type;        /* identificateur de protocole, par exemple ETH_P_IP
197                                          *  NOTE : Pour les VLAN, ceci sera le
198                                          *  type encapsule. --BLG
199                                          */
200         int             hh_len;         /* longueur de l'en-tete            */
201         int             (*hh_output)(struct sk_buff *skb);
202         rwlock_t        hh_lock;
```

```
203
204         /* en-tete physique en cache ; utiles pour les necessites d'alignement des
               machines. */
205 #define HH_DATA_MOD     16
206 #define HH_DATA_OFF(__len) \
207         (HH_DATA_MOD - ((__len) & (HH_DATA_MOD - 1)))
208 #define HH_DATA_ALIGN(__len) \
209         (((__len)+(HH_DATA_MOD-1))&~(HH_DATA_MOD - 1))
210         unsigned long   hh_data[HH_DATA_ALIGN(LL_MAX_HEADER) / sizeof(long)];
211 };
```

qui comporte à la fois des champs de contrôle (qui constitueraient un descripteur dans d'autres circonstances) et des données. Les champs de contrôle sont les suivants :

- Les entrées du cache sont placées dans une liste chaînée. Le champ `hh_next` permet de retrouver l'entrée suivante.
- Le champ `hh_refcnt` est le compteur de références habituel, qui permet de ne pas détruire l'entrée si celui-ci n'est pas nul.
- Le verrou rotatif `hh_lock` permet d'éviter un accès simultané à l'entrée du cache.

Les champs de données sont les suivants :

- Le type `hh_type` est l'identificateur du protocole, par exemple `ETH_P_IP` pour un paquet IP.
- La taille de l'en-tête est spécifiée par le champ `hh_len`.
- La méthode `hh_output()` permet de récupérer un en-tête à partir d'un descripteur de tampon.
- L'en-tête matériel lui-même constitue le champ `hh_data[]`.

  La constante `LL_MAX_HEADER` est définie dans le fichier `linux/include/linux/netdevice.h`:

Code Linux 2.6.10

```
81 /*
82  *      Calcule la longueur de l'en-tete dans le pire cas en fonction des protocoles
73  *      utilises.
84  */
85
86 #if !defined(CONFIG_AX25) && !defined(CONFIG_AX25_MODULE) && !defined(CONFIG_TR)
87 #define LL_MAX_HEADER   32
88 #else
89 #if defined(CONFIG_AX25) || defined(CONFIG_AX25_MODULE)
90 #define LL_MAX_HEADER   96
91 #else
92 #define LL_MAX_HEADER   48
93 #endif
94 #endif
```

14.3 Implémentation de l'ordre réseau des octets

14.3.1 Implémentation générique

Code Linux 2.6.10

Les fonctions relatives à l'ordre réseau sont définies de façon générique dans le fichier `linux/include/linux/byteorder/generic.h`:

```
1  #ifndef _LINUX_BYTEORDER_GENERIC_H
2  #define _LINUX_BYTEORDER_GENERIC_H
3
4  /*
5   * linux/byteorder_generic.h
6   * Support pour reordonner les octets de facon generique
7   *
8   * Francois-Rene Rideau <fare@tunes.org> 19970707
9   *    a rassemble toutes les bonnes idees depuis tous les asm-foo/byteorder.h en un fichier,
10  *    les a nettoyes.
11  *    J'espere que c'est compatible avec les compilateurs non-GCC.
12  *    J'ai decide de mettre __BYTEORDER_HAS_U64__ dans byteorder.h,
13  *    puisque je n'etais pas sur que ce serait ok de le mettre dans types.h
14  *    A mis a jour pour 2.1.43
15  * Francois-Rene Rideau <fare@tunes.org> 19971012
16  *    A mis a jour pour 2.1.57
17  *    pour faire plaisir a Linus T., a remplace l'enorme #ifdef entre little/big endian
18  *    par des fichiers #include imbriques.
19  * Francois-Rene Rideau <fare@tunes.org> 19971205
20  *    Fait pour 2.1.71 ; maintenant cosmetique :
21  *    Mis les fichiers dans include/linux/byteorder/
22  *    Ventile swab du support generique.
23  *
24  * A FAIRE :
25  *   = Les mainteneurs reguliers du noyau devraient aussi remplacer toutes les macros
26  *    byteswap manuelles qui restent, disseminees parmi les pilotes,
27  *    apres quelques grep sur les sources...
28  *   = Linus peut vouloir renommer tous ces macros et fichiers a son gout,
29  *    pour correspondre a son schema de nommage personnel.
30  *   = il semble que quelques pilotes devraient aussi apprecier
31  *    le support d'echange des demi-octets...
32  *   = chaque architecture devrait ajouter sa macro byteswap dans asm/byteorder.h
33  *    voir comment quelques architectures l'ont deja fait (i386, alpha, ppc, etc)
34  *   = cpu_to_beXX et beXX_to_cpu pourraient un jour avoir besoin d'etre bien
35  *    distinguees dans le noyau. Ceci n'est pas le cas a l'heure actuelle,
36  *    puisque les machines petit-boutiennes, grand-boutiennes et pdp-boutiennes n'en ont pas
        besoin.
37  *    Mais ceci pourrait etre le cas pour, disons, un portage de Linux sur les
38  *    architectures a 20/21 bits (et Linux F21 en a besoin ?).
39  */
40
41 /*
42  * Les macros suivantes doivent etre definies dans <asm/byteorder.h>:
43  *
44  * Conversion de long et short int entre reseau et format hote
45  *      ntohl(__u32 x)
46  *      ntohs(__u16 x)
47  *      htonl(__u32 x)
48  *      htons(__u16 x)
49  * Il semble que quelques programmes (lesquels ? ou ? ou peut-etre un standard ? POSIX ?)
50  * pourraient aimer que celles-ci soient des fonctions, pas des macros (pourquoi ?).
51  * si c'est vrai, alors detecter-les et prenez les mesures necessaires.
52  * En tout cas, la mesure est : definir seulement ___ntohl en tant que macro,
53  * et dans un fichier separe, puis
54  * unsigned long inline ntohl(x){return ___ntohl(x);}
```

```
55   *
56   * De meme pour les arguments constants
57   *      __constant_ntohl(__u32 x)
58   *      __constant_ntohs(__u16 x)
59   *      __constant_htonl(__u32 x)
60   *      __constant_htons(__u16 x)
61   *
62   * Conversion des entiers XX-bits (16- 32- or 64-)
63   * entre un format de CPU natif et le format petit/grand boutien
64   * chose 64-bits seulement definie pour les architectures memes
65   *      cpu_to_[bl]eXX(__uXX x)
66   *      [bl]eXX_to_cpu(__uXX x)
67   *
68   * De meme, mais prendre un pointeur sur la valeur a convertir
69   *      cpu_to_[bl]eXXp(__uXX x)
70   *      [bl]eXX_to_cpup(__uXX x)
71   *
72   * De meme, mais changer in situ
73   *      cpu_to_[bl]eXXs(__uXX x)
74   *      [bl]eXX_to_cpus(__uXX x)
75   *
76   * Voir asm-foo/byteorder.h pour des exemples de comment fournir
77   * des versions optimisees pour l'architecture
78   *
79   */
80
81
82  #if defined(__KERNEL__)
83  /*
84   * a l'interieur du noyau nous pouvons utiliser des alias ;
85   * au dehors, nous devons eviter de polluer l'espace de noms POSIX...
86   */
87  #define cpu_to_le64 __cpu_to_le64
88  #define le64_to_cpu __le64_to_cpu
89  #define cpu_to_le32 __cpu_to_le32
90  #define le32_to_cpu __le32_to_cpu
91  #define cpu_to_le16 __cpu_to_le16
92  #define le16_to_cpu __le16_to_cpu
93  #define cpu_to_be64 __cpu_to_be64
94  #define be64_to_cpu __be64_to_cpu
95  #define cpu_to_be32 __cpu_to_be32
96  #define be32_to_cpu __be32_to_cpu
97  #define cpu_to_be16 __cpu_to_be16
98  #define be16_to_cpu __be16_to_cpu
99  #define cpu_to_le64p __cpu_to_le64p
100 #define le64_to_cpup __le64_to_cpup
101 #define cpu_to_le32p __cpu_to_le32p
102 #define le32_to_cpup __le32_to_cpup
103 #define cpu_to_le16p __cpu_to_le16p
104 #define le16_to_cpup __le16_to_cpup
105 #define cpu_to_be64p __cpu_to_be64p
106 #define be64_to_cpup __be64_to_cpup
107 #define cpu_to_be32p __cpu_to_be32p
108 #define be32_to_cpup __be32_to_cpup
109 #define cpu_to_be16p __cpu_to_be16p
110 #define be16_to_cpup __be16_to_cpup
111 #define cpu_to_le64s __cpu_to_le64s
112 #define le64_to_cpus __le64_to_cpus
113 #define cpu_to_le32s __cpu_to_le32s
114 #define le32_to_cpus __le32_to_cpus
115 #define cpu_to_le16s __cpu_to_le16s
116 #define le16_to_cpus __le16_to_cpus
```

```
117 #define cpu_to_be64s __cpu_to_be64s
118 #define be64_to_cpus __be64_to_cpus
119 #define cpu_to_be32s __cpu_to_be32s
120 #define be32_to_cpus __be32_to_cpus
121 #define cpu_to_be16s __cpu_to_be16s
122 #define be16_to_cpus __be16_to_cpus
123 #endif
124
125
126 #if defined(__KERNEL__)
127 /*
128  * Manipule ntohl et autres. Ceci a divers problemes de compatibilite -
129  * comme nous voulons donner le prototype avant meme que nous ayons
130  * une macro pour lui dans le cas ou quelque etrange programme
131  * veut prendre l'adresse de la chose ou de quelque chose..
132  *
133  * Noter que ceci utilisait un "long" a renvoyer dans libc5, alors que
134  * long est souvent 64 bits de nos jours... D'ou les conversions.
135  *
136  * On a besoin que ce soit des macros afin que tous les pliages
137  * soient corrects - si l'argument passe dans une fonction inline
138  * ce n'est plus constant avec gcc...
139  */
140
141 #undef ntohl
142 #undef ntohs
143 #undef htonl
144 #undef htons
145
146 /*
147  * Fait les prototypes. Quelqu'un peut vouloir prendre l'adresse
148  * ou quelque sottise de la sorte...
149  */
150 extern __u32                    ntohl(__be32);
151 extern __be32                   htonl(__u32);
152 extern __u16                    ntohs(__be16);
153 extern __be16                   htons(__u16);
154
155 #if defined(__GNUC__) && (__GNUC__ >= 2) && defined(__OPTIMIZE__)
156
157 #define ___htonl(x) __cpu_to_be32(x)
158 #define ___htons(x) __cpu_to_be16(x)
159 #define ___ntohl(x) __be32_to_cpu(x)
160 #define ___ntohs(x) __be16_to_cpu(x)
161
162 #define htonl(x) ___htonl(x)
163 #define ntohl(x) ___ntohl(x)
164 #define htons(x) ___htons(x)
165 #define ntohs(x) ___ntohs(x)
166
167 #endif /* OPTIMIZE */
168
169 #endif /* KERNEL */
170
171
172 #endif /* _LINUX_BYTEORDER_GENERIC_H */
```

14.3.2 Cas des microprocesseurs Intel

On se sert de plusieurs drapeaux pour connaître l'ordre adopté par l'architecture du microprocesseur, définis dans les fichiers en-tête `linux/include/linux/byteorder/little_endian.h`: Code Linux 2.6.10

```
1 #ifndef _LINUX_BYTEORDER_LITTLE_ENDIAN_H
2 #define _LINUX_BYTEORDER_LITTLE_ENDIAN_H
3
4 #ifndef __LITTLE_ENDIAN
5 #define __LITTLE_ENDIAN 1234
6 #endif
7 #ifndef __LITTLE_ENDIAN_BITFIELD
8 #define __LITTLE_ENDIAN_BITFIELD
9 #endif
```

et `linux/include/linux/byteorder/big_endian.h`: Code Linux 2.6.10

```
1 #ifndef _LINUX_BYTEORDER_BIG_ENDIAN_H
2 #define _LINUX_BYTEORDER_BIG_ENDIAN_H
3
4 #ifndef __BIG_ENDIAN
5 #define __BIG_ENDIAN 4321
6 #endif
7 #ifndef __BIG_ENDIAN_BITFIELD
8 #define __BIG_ENDIAN_BITFIELD
9 #endif
```

On se réfère enfin au fichier `linux/include/asm/byteorder.h`, par exemple au fichier `linux/include/asm-i386/byteorder.h` pour les microprocesseurs Intel : Code Linux 2.6.10

```
1  #ifndef _I386_BYTEORDER_H
2  #define _I386_BYTEORDER_H
3
4  #include <asm/types.h>
5  #include <linux/compiler.h>
6
7  #ifdef __GNUC__
8
9  /* Pour eviter bswap sur i386 */
10 #ifdef __KERNEL__
11 #include <linux/config.h>
12 #endif
13
14 static __inline__ __attribute_const__ __u32 ___arch__swab32(__u32 x)
15 {
16 #ifdef CONFIG_X86_BSWAP
17         __asm__("bswap %0" : "=r" (x) : "" (x));
18 #else
19         __asm__("xchgb %b0,%h0\n\t"     /* echange les octets bas   */
20                 "rorl $16,%0\n\t"       /* echange les mots         */
21                 "xchgb %b0,%h0"         /* echange les octets hauts */
22                 :"=q" (x)
23                 : "" (x));
24 #endif
25         return x;
26 }
27
28 static __inline__ __attribute_const__ __u64 ___arch__swab64(__u64 val)
29 {
30         union {
31                 struct { __u32 a,b; } s;
32                 __u64 u;
33         } v;
```

```
34         v.u = val;
35 #ifdef CONFIG_X86_BSWAP
36         asm("bswapl %0 ; bswapl %1 ; xchgl %0,%1"
37             : "=r" (v.s.a), "=r" (v.s.b)
38             : "" (v.s.a), "1" (v.s.b));
39 #else
40    v.s.a = ___arch__swab32(v.s.a);
41         v.s.b = ___arch__swab32(v.s.b);
42         asm("xchgl %0,%1" : "=r" (v.s.a), "=r" (v.s.b) : "" (v.s.a), "1" (v.s.b));
43 #endif
44         return v.u;
45 }
46
47 /* Ne pas definir swab16. Gcc est assez fin pour reconnaitre la version "C" et
48    les convertir en rotation ou echange.  */
49
50 #define __arch__swab64(x) ___arch__swab64(x)
51 #define __arch__swab32(x) ___arch__swab32(x)
52
53 #define __BYTEORDER_HAS_U64__
54
55 #endif /* __GNUC__ */
56
57 #include <linux/byteorder/little_endian.h>
58
59 #endif /* _I386_BYTEORDER_H */
```

Chapitre 15

Détection d'une carte réseau (1) Aspect général

Comme nous l'avons vu, la première action pour mettre en place le sous-système réseau est de détecter la carte réseau, déterminer ses paramètres (port d'entrée-sortie, IRQ, canal DMA, adresse MAC...) et lui associer un pilote de périphérique.

Comme nous l'avons vu dans le chapitre précédent, un pilote de périphérique réseau est implémenté sous la forme d'une structure (pour ne pas dire d'une classe au sens de la programmation orientée objet) possédant un certain nombre de champs, qui sont soit des attributs, soit des méthodes. Lors de la phase de démarrage, le pilote est initialisé et enregistré dans le système d'exploitation. Ensuite on gère les entrées-sorties du périphérique avec les appels système et les interruptions.

Nous allons étudier l'initialisation dans ce chapitre, c'est-à-dire voir comment l'on associe un nom à une carte réseau, comment on lui associe un index et comment on place les adresses des fonctions qui seront utiles lors de la partie dynamique (envoi et réception de trames) dans le descripteur de périphérique réseau associé.

Nous avons au chapitre 14 qu'une carte réseau est décrite par un descripteur de périphérique réseau, de type `struct net_device`, et que les descripteurs de ce type sont placés dans une liste chaînée `dev_base`. La première étape consiste à créer et à initialiser la structure `net_device` associée au périphérique. Ceci peut se faire de deux manières différentes :

- S'il a été décidé lors de la configuration du noyau que le pilote de périphérique réseau doit être intégré au noyau de manière fixe, une structure `net_device` pour celui-ci existe dans le fichier `linux/drivers/net/Space.c`. Les descripteurs de périphérique prévus à l'amorçage pour les cartes réseau prédéfinies sont créées grâce à un mécanisme comportant des définitions de préprocesseur, en fonction de la configuration du noyau.

  Par exemple huit descripteurs de périphérique réseau, qui ne sont à ce moment attribués à aucune carte, sont créés pour les périphériques réseau Ethernet.
- Si le pilote est écrit en tant que module du noyau, celui-ci doit le créer chaque fois que le module est appelé.

15.1 Première étape

15.1.1 Cas des périphériques liés statiquement

15.1.1.1 Démarrage

Tout commence par le lancement de l'initialisation des périphériques prévu lors du démarrage du système grâce à un certain nombre d'appels à la fonction générale `device_initcall()` répartie tout au long du code. Pour les périphériques réseau, il s'agit de la ligne 457 du fichier `linux/drivers/net/Space.c`:

Code Linux 2.6.10

```
423 device_initcall(net_olddevs_init);
```

qui fait appel à une fonction `net_olddevs_init()` définie juste au-dessus dans le même fichier :

Code Linux 2.6.10

```
389 /*  Pilotes configures statiquement -- mettre un peu d'ordre ici. */
390 static int __init net_olddevs_init(void)
391 {
392         int num;
393
394         if (loopback_init()) {
395                 printk(KERN_ERR "Network loopback device setup failed\n");
396         }
397
398
399 #ifdef CONFIG_SBNI
400         for (num = 0; num < 8; ++num)
401                 sbni_probe(num);
402 #endif
403 #ifdef CONFIG_TR
404         for (num = 0; num < 8; ++num)
405                 if (!trif_probe(num))
406                         trif_probe2(num);
407 #endif
408         for (num = 0; num < 8; ++num)
409                 ethif_probe2(num);
410
411 #ifdef CONFIG_COPS
412         cops_probe(0);
413         cops_probe(1);
414         cops_probe(2);
415 #endif
416 #ifdef CONFIG_LTPC
417         ltpc_probe();
418 #endif
419
420         return 0;
421 }
```

dont le but est d'initialiser le périphérique en boucle, huit périphériques Ethernet (lignes 408–409) et éventuellement d'autres périphériques suivant les options de compilation du noyau.

Nous reviendrons plus tard sur les détails de l'initialisation des huit périphériques Ethernet. Intéressons-nous pour l'instant au périphérique en boucle (*loopback* en anglais) à titre d'exemple.

15.1.1.2 Descripteur du périphérique en boucle

L'instance de structure `net_device` pour le périphérique *loopback*, à savoir `loopback_dev`, est définie dans le fichier `linux/drivers/net/loopback.c`:

Code Linux 2.6.10

```
198 struct net_device loopback_dev = {
199         .name                   = "lo",
```

```
200         .mtu                    = (16 * 1024) + 20 + 20 + 12,
201         .hard_start_xmit        = loopback_xmit,
202         .hard_header            = eth_header,
203         .hard_header_cache      = eth_header_cache,
204         .header_cache_update    = eth_header_cache_update,
205         .hard_header_len        = ETH_HLEN,     /* 14   */
206         .addr_len               = ETH_ALEN,     /* 6    */
207         .tx_queue_len           = 0,
208         .type                   = ARPHRD_LOOPBACK,      /* 0x0001*/
209         .rebuild_header         = eth_rebuild_header,
210         .flags                  = IFF_LOOPBACK,
211         .features               = NETIF_F_SG|NETIF_F_FRAGLIST
212                                   |NETIF_F_NO_CSUM|NETIF_F_HIGHDMA
213                                   |NETIF_F_LLTX,
214         .ethtool_ops            = &loopback_ethtool_ops,
215 };
```

Les valeurs des champs sont presque toutes liées à Ethernet (`eth_header()`, `eth_header_cache()`, `eth_header_cache_update()`, `eth_rebuild_header()`, `ETH_HLEN` et `ETH_ALEN`). Nous y reviendrons à propos de l'implémentation d'Ethernet au fur et à mesure des besoins. Seuls le nom, la fonction `loopback_xmit()` et l'ensemble d'opérations `loopback_ethtool_ops` sont propres au périphérique en boucle.

15.1.1.3 Initialisation et enregistrement du périphérique en boucle

La fonction d'initialisation et d'enregistrement du périphérique en boucle, `loopback_init()` dont il est fait référence ci-dessus, est définie dans le même fichier : Code Linux 2.6.10

```
217 /* Initialise et enregistre le peripherique LOOPBACK. */
218 int __init loopback_init(void)
219 {
220         struct net_device_stats *stats;
221
222         /* Peut survivre sans statistiques */
223         stats = kmalloc(sizeof(struct net_device_stats), GFP_KERNEL);
224         if (stats) {
225                 memset(stats, 0, sizeof(struct net_device_stats));
226                 loopback_dev.priv = stats;
227                 loopback_dev.get_stats = &get_stats;
228         }
229
230         return register_netdev(&loopback_dev);
231 };
```

Autrement dit :

- On déclare une structure pour les statistiques. On essaie de réserver de la place en mémoire centrale pour celle-ci. Si on n'y parvient pas, ce n'est pas grave.
- Si on y parvient, on initialise cet emplacement mémoire à zéro et on renseigne les deux champs du descripteur de périphérique concernant les statistiques.

  La fonction `get_stats()` pour le périphérique en boucle est définie dans le même fichier : Code Linux 2.6.10

```
161 static struct net_device_stats *get_stats(struct net_device *dev)
162 {
163         struct net_device_stats *stats = dev->priv;
164         int i;
165
166         if (!stats) {
167                 return NULL;
```

```
168         }
169
170         memset(stats, 0, sizeof(struct net_device_stats));
171
172         for (i=0; i < NR_CPUS; i++) {
173                 struct net_device_stats *lb_stats;
174
175                 if (!cpu_possible(i))
176                         continue;
177                 lb_stats = &per_cpu(loopback_stats, i);
178                 stats->rx_bytes   += lb_stats->rx_bytes;
179                 stats->tx_bytes   += lb_stats->tx_bytes;
180                 stats->rx_packets += lb_stats->rx_packets;
181                 stats->tx_packets += lb_stats->tx_packets;
182         }
183
184         return stats;
185 }
```

Autrement dit :

- On essaie de récupérer la partie privée du descripteur de périphérique en boucle. Si on n'y parvient pas, on renvoie `NULL`.
- On réinitialise à zéro cette partie de la mémoire vive.
- On récupère les données concernant le périphérique en boucle pour chaque microprocesseur, on les cumule et on renvoie l'adresse de la partie privée.

– On enregistre le pilote grâce à la fonction `register_netdev()` étudiée ci-dessous dans la deuxième étape.

15.1.2 Cas des périphériques modularisés

15.1.2.1 Déclaration du descripteur de périphérique réseau

Dans le cas d'un périphérique modularisé, un descripteur de périphérique est déclaré quelque part. Par exemple le descripteur du périphérique réseau 3Com 501 est déclaré dans le fichier `linux/drivers/net/3c501.c`:

Code Linux 2.6.10

```
891 #ifdef MODULE
892
893 static struct net_device *dev_3c501;
894
895 MODULE_PARM(io, "i");
896 MODULE_PARM(irq, "i");
897 MODULE_PARM_DESC(io, "EtherLink I/O base address");
898 MODULE_PARM_DESC(irq, "EtherLink IRQ number");
```

15.1.2.2 Module d'initialisation

Chaque pilote de carte réseau modularisé possède un module d'initialisation, appelé soit lors du démarrage du système, soit lors de l'appel d'un module.

Pour la carte 3Com 501, que nous prenons en exemple, ce module est situé à la fin du fichier `linux/drivers/net/3c501.c`:

Code Linux 2.6.10

```
900 /**
901  * init_module :
902  *
903  * Cette fonction est appelee lorsque le pilote est charge en tant que module. Nous
```

```
904  * improvisons une structure de peripherique avec l'adresse de base d'I/O et l'interruption
       positionnes comme si il avait ete appele
905  * depuis Space.c. Ceci minimise le code supplementaire qui serait requis
906  * sinon.
907  *
908  * Renvoie 0 en cas de succes et -EIO si la carte n'est pas trouvee. Renvoyer une
909  * erreur ici a pour consequence de ne pas charger le module
910  */
911
912 int init_module(void)
913 {
914         dev_3c501 = el1_probe(-1);
915         if (IS_ERR(dev_3c501))
916                 return PTR_ERR(dev_3c501);
917         return 0;
918 }
```

qui fait essentiellement appel à la fonction `el1_probe()`, sur laquelle nous reviendrons au chapitre 17. Retenons seulement, pour l'instant, qu'elle fait, comme le périphérique en boucle, appel à la fonction `register_netdev()`.

Les fonctions en ligne générales `IS_ERR()` et `PTR_ERR()` sont définies dans le fichier `linux/-include/linux/err.h`:

Code Linux 2.6.10

```
21 static inline long PTR_ERR(const void *ptr)
22 {
23         return (long) ptr;
24 }
25
26 static inline long IS_ERR(const void *ptr)
27 {
28         return unlikely((unsigned long)ptr > (unsigned long)-1000L);
29 }
```

15.2 Deuxième étape : allocation d'un nom

La deuxième étape de l'installation d'une carte réseau, après celle de la création du descripteur de périphérique, consiste à lui allouer un nom. Cette seconde étape est effectuée par la fonction `register_netdev()`, dont nous venons de voir qu'elle est appelée soit lors de l'initialisation (cas du périphérique en boucle par exemple), soit à partir du module d'initialisation dans le cas des pilotes modularisés.

15.2.1 Allocation du nom et passage à la troisième étape

15.2.1.1 Enregistrement d'un pilote de périphérique réseau

La fonction `register_netdev()` d'enregistrement d'un pilote (plus exactement d'un descripteur) de périphérique réseau est définie dans le fichier `linux/drivers/net/net_init.c`:

Code Linux 2.6.10

```
108 int register_netdev(struct net_device *dev)
109 {
110         int err;
111
112         rtnl_lock();
113
114         /*
115          *      Si le nom est une chaine de caracteres format, l'appelant veut que
116          *      nous allouions un nom
117          */
```

```
118
119         if (strchr(dev->name, '%'))
120         {
121                 err = dev_alloc_name(dev, dev->name);
122                 if (err < 0)
123                         goto out;
124         }
125
126         /*
127          *      Point d'ancrage de compatibilite ascendante. Enlever ceci dans 2.5
128          */
129
130         if (dev->name[0]==0 || dev->name[0]==' ')
131         {
132                 err = dev_alloc_name(dev, "eth%d");
133                 if (err < 0)
134                         goto out;
135         }
136
137         err = register_netdevice(dev);
138
139 out:
140         rtnl_unlock();
141         return err;
142 }
```

Elle a pour but d'allouer un nom au périphérique et d'appeler l'étape suivante. Plus précisément :

– On se réserve le maniement de la liste chaînée des périphériques réseau car on va en avoir besoin pour vérifier les noms existants.

Code Linux 2.6.10

Comme indiqué dans le fichier `linux/net/core/dev.c`:

```
166 /*
167  * La liste @dev_base est protegee par @dev_base_lock et le
168  * semaphore rtln.
169  *
170  * Pour les lectures pures, on doit detenir dev_base_lock pour lire.
171  *
172  * Pour les ecritures, on doit detenir le semaphore rtnl tant qu'on parcourt la
173  * liste dev_base, et detenir dev_base_lock pour ecrire lorsqu'on fait les
174  * mises a jour effectives. Ceci permet aux lecteurs pures d'acceder a la liste meme
175  * lorsqu'une ecriture se prepare a la mettre a jour.
176  *
177  * En d'autres mots, dev_base_lock est detenu pour l'ecriture seulement pour
178  * se proteger contre les lectuers pures ; le semaphore rtnl fournit la
179  * protection contre les autres ecritures.
180  *
181  * Voir, pour des exemples d'utilisation, register_netdevice() et
182  * unregister_netdevice(), qui doivent etre appeles avec le semaphore rtnl
183  * detenu.
184  */
```

l'écriture et la lecture dans cette liste sont protégées par le verrou rotatif `dev_base_lock` et le sémaphore `rtln_sem`.

Code Linux 2.6.10

Ceux-ci sont définis dans les fichiers `linux/net/core/dev.c`:

```
187 rwlock_t dev_base_lock = RW_LOCK_UNLOCKED;
```

Code Linux 2.6.10

et `linux/net/core/netlink.c` (`rtnl` signifiant *RouTing NetLink*) :

```
54 DECLARE_MUTEX(rtnl_sem);
55
```

```
56 void rtnl_lock(void)
57 {
58         rtnl_shlock();
59 }
```

La macro `rtnl_shlock()` est définie dans le fichier `linux/include/linux/netlink.h`: Code Linux 2.6.10

```
800 #define rtnl_shlock()           down(&rtnl_sem)
```

- Si le champ `name` du périphérique passé en argument ne comporte pas d'occurrence du caractère `'%'`, c'est que l'on veut que le contenu de ce champ soit le nom du périphérique, non suivi d'un numéro (cas de `lo` par exemple). On fait alors appel à la fonction `dev_alloc_name()`, étudiée ci-dessous, pour vérifier que ce nom n'est pas déjà attribué à un autre périphérique et le lui attribuer si c'est bien le cas. Sinon cette fonction renvoie une erreur, on ne se réserve plus alors le maniement de la liste chaînée des périphériques réseau et on renvoie le code d'erreur fourni par cette fonction.
- Si le champ `name` commence par `'0'`, autrement dit si aucun nom n'a été attribué, ou par une espace, on fait appel à la même fonction mais le nom du périphérique sera `'eth'` suivi d'un numéro. Si une erreur intervient au cours de cette allocation, on ne se réserve plus le maniement de la liste chaînée des périphériques réseau et on renvoie le code fourni par cette fonction.
- On passe à la troisième étape en faisant appel à la fonction `register_netdevice()` étudiée ci-dessous.
- On ne se réserve plus le maniement de la liste chaînée des périphériques réseau et on renvoie le code fourni par la fonction ci-dessus.

La fonction `rtnl_unlock()` est définie dans le fichier `linux/net/core/netlink.c`: Code Linux 2.6.10

```
61 void rtnl_unlock(void)
62 {
63         rtnl_shunlock();
64
65         netdev_run_todo();
66 }
```

La macros `retnl_shunlock()` est définie dans le fichier `linux/include/linux/netlink.h`: Code Linux 2.6.10

```
803 #define rtnl_shunlock() do { up(&rtnl_sem); \
804                              if (rtnl && rtnl->sk_receive_queue.qlen) \
805                                      rtnl->sk_data_ready(rtnl, 0); \
806                          } while(0)
```

La variable `rtnl` est déclarée dans le fichier `linux/net/core/rtnetlink.c`: Code Linux 2.6.10

```
81 struct sock *rtnl;
```

dont le type `struct sock` sera étudié plus tard. Seule l'adresse est importante pour l'instant.

Enfin la fonction `netdev_run_todo()` correspond à la quatrième étape et sera étudiée ci-dessous.

15.2.2 Manipulation d'un périphérique désigné par un nom

15.2.2.1 Allocation en précisant le nom

La fonction `dev_alloc_name()` détermine un nom pour le périphérique à partir de celui qui est passé en argument et elle vérifie qu'il n'est pas déjà attribué. Elle est définie dans le fichier `linux/net/core/dev.c`:

Code Linux 2.6.10

```
655 /**
656  *      dev_alloc_name - alloue un nom a un peripherique
657  *      @dev : peripherique
658  *      @name: chaine de caracteres directive de format du nom
659  *
660  *      Passe une directive de format - par exemple "lt%d" - elle essaiera de trouver
670  *      un id convenable. Pas efficace pour certains peripheriques, ne pas appeler beaucoup.
671  *      L'appelant doit detenir le verrou dev_base ou rtnl pendant l'allocation du nom
672  *      et l'ajout du peripherique afin d'eviter les doublons. Renvoie le numero
673  *      de l'unite assignee ou un code errno negatif.
665  */
666
667 int dev_alloc_name(struct net_device *dev, const char *name)
668 {
669         int i = 0;
670         char buf[IFNAMSIZ];
671         const char *p;
672         const int max_netdevices = 8*PAGE_SIZE;
673         long *inuse;
674         struct net_device *d;
675
676         p = strnchr(name, IFNAMSIZ-1, '%');
677         if (p) {
678                 /*
679                  * Verifie la chaine de caracteres car elle peut provenir de
680                  * l'utilisateur. Elle doit soit contenir "%d" et pas d'autre caracteres "%",
681                  * ou aucun caractere "%" du tout.
682                  */
683                 if (p[1] != 'd' || strchr(p + 2, '%'))
684                         return -EINVAL;
685
686                 /* Utilise une page en tant que tableau de bits des slots possibles */
687                 inuse = (long *) get_zeroed_page(GFP_ATOMIC);
688                 if (!inuse)
689                         return -ENOMEM;
690
691                 for (d = dev_base; d; d = d->next) {
692                         if (!sscanf(d->name, name, &i))
693                                 continue;
694                         if (i < 0 || i >= max_netdevices)
695                                 continue;
696
697                         /*  evite les cas ou sscanf n'est pas l'inverse exact de printf */
698                         snprintf(buf, sizeof(buf), name, i);
699                         if (!strncmp(buf, d->name, IFNAMSIZ))
700                                 set_bit(i, inuse);
701                 }
702
703                 i = find_first_zero_bit(inuse, max_netdevices);
704                 free_page((unsigned long) inuse);
705         }
706
707         snprintf(buf, sizeof(buf), name, i);
708         if (!__dev_get_by_name(buf)) {
709                 strlcpy(dev->name, buf, IFNAMSIZ);
```

```
710                 return i;
711         }
712
713         /* Il est possible d'aller au-dela des slots possibles
714          * lorsque le nom est long et qu'il n'y a pas assez d'espace laisse
715          * pour les chiffres ou si tous les bits sont utilises.
716          */
717         return -ENFILE;
718 }
```

Autrement dit :

- On déclare le numéro du périphérique qui sera assigné, une chaîne de caractères qui contiendra le nom du périphérique, une chaîne de caractères qui contiendra le nom passé en argument, le nombre maximum de périphériques permis, l'adresse de la page qui sera utilisée en tant que tableau de bits des emplacements possibles et un descripteur de périphérique réseau.
- On recherche l'adresse de la première occurrence du caractère `'%'` dans la chaîne de caractères passée en argument. Si celle-ci existe :
  - si elle n'est pas suivie du caractère `'d'` ou s'il existe une autre occurrence de `'%'`, on renvoie l'opposé du code d'erreur `EINVAL`.
  - On recherche une page non utilisée (qui ne contient que des zéros). Si on n'y parvient pas, on renvoie l'opposé du code d'erreur `ENOMEM`.
  - On décrit la liste chaînée des descripteurs de périphérique réseau, on place dans `buf` la chaîne de caractères obtenue à partir de `name` en remplaçant l'occurrence de `'%d'` par l'index `i`, s'il existe un périphérique portant le nom ainsi obtenu, on positionne le bit correspondant de la page à 1.
  - On détermine la position du premier bit nul de la page et on libère la page.
- On place dans `buf` la chaîne de caractères obtenue à partir de `name` en remplaçant l'occurrence de `'%d'` par l'index `i`.
- Si le nom obtenu est déjà utilisé par un autre périphérique, on renvoie l'opposé du code d'erreur `ENFILE`. La fonction `__dev_get_by_name()` sera étudiée ci-dessous.
- Sinon on renseigne le champ `name` du descripteur de périphérique passé en argument avec ce nom.

15.2.2.2 Recherche d'un périphérique réseau par son nom

La fonction `__dev_get_by_name()` renvoie l'adresse du descripteur du périphérique dont le nom est celui passé en argument s'il existe et `NULL` sinon. Elle est définie dans le fichier `linux/net/core/dev.c` :

Code Linux 2.6.10

```
476 /**
477  *      __dev_get_by_name       - trouver un peripherique grace a son nom
478  *      @name : nom a trouver
479  *
480  *      Trouver une interface grace a son nom. Doit etre appelee avec le semaphore RTNL
481  *      ou @dev_base_lock. Si le nom est trouve, un pointeur au peripherique
482  *      est renvoye. Si le nom n'est pas trouve alors %NULL est renvoye. Les
483  *      compteurs de reference ne sont pas incrementes, aussi l'appelant doit-il etre
484  *      soigneux avec les verrous.
485  */
486
487 struct net_device *__dev_get_by_name(const char *name)
```

```
488 {
489         struct hlist_node *p;
490
491         hlist_for_each(p, dev_name_hash(name)) {
492                 struct net_device *dev
493                         = hlist_entry(p, struct net_device, name_hlist);
494                 if (!strncmp(dev->name, name, IFNAMSIZ))
495                         return dev;
496         }
497         return NULL;
498 }
```

Autrement dit on décrit la liste chaînée des périphériques réseau d'index de hachage correspondant au nom passé en argument à la recherche d'un descripteur de périphérique réseau dont le nom coïncide avec celui passé en argument. Si on le trouve, on renvoie l'adresse de ce descripteur.

Remarquons l'utilisation d'une astuce: si on ne trouve pas le périphérique, la variable `dev` prendra la valeur `NULL` à la fin de la liste.

Code Linux 2.6.10

Les macros générales `hlist_for_each()` et `hlist_entry()` sont définies dans le fichier `linux/include/linux/list.h`:

```
631 #define hlist_entry(ptr, type, member) container_of(ptr,type,member)
632
633 #define hlist_for_each(pos, head) \
634         for (pos = (head)->first; pos && ({ prefetch(pos->next); 1; }); \
635              pos = pos->next)
```

Code Linux 2.6.10

La fonction en ligne `dev_name_hash()` est définie dans le fichier `linux/net/core/dev.c`:

```
192 #define NETDEV_HASHBITS 8
193 static struct hlist_head dev_name_head[1<<NETDEV_HASHBITS];
194 static struct hlist_head dev_index_head[1<<NETDEV_HASHBITS];
195
196 static inline struct hlist_head *dev_name_hash(const char *name)
197 {
198         unsigned hash = full_name_hash(name, strnlen(name, IFNAMSIZ));
199         return &dev_name_head[hash & ((1<<NETDEV_HASHBITS)-1)];
200 }
```

La fonction générale:

`unsigned int full_name_hash(const unsigned char *name, unsigned int len)`

permet de calculer l'index de hachage d'une chaîne de caractères.

15.3 Attribution d'un index et insertion dans la liste

La troisième étape de l'installation d'un périphérique réseau consiste à lui attribuer un index et à insérer son descripteur dans la liste chaînée des descripteurs de périphériques réseau ainsi que dans la table de hachage.

La liste `dev_base` gère les périphériques réseau. Ils doivent tous être enregistrés dans cette liste, qu'ils soient ou non activés. Par contre, ils ne doivent être enregistrés que si la carte réseau correspondante est réellement disponible sur le système informatique. C'est pourquoi on vérifie avant l'insertion qu'il est possible de trouver une carte réseau pour le pilote. Pour ce faire, chacun des descripteurs de périphériques comporte une fonction `init()`. Elle est très spécifique à la carte. Les périphériques de réseau logiques comme *loopback* (`lo`) ou PPP (`ppp0`), qui ne sont pas assignés au matériel sous-jacent, constituent une exception.

Cette étape débute par l'appel à la fonction `register_netdevice()`, effectuée depuis la fonction `register_netdev()`. Avant de l'étudier, nous allons parler des chaînes de notification.

15.3.1 Chaînes de notification

Les périphériques réseau sont déclarés et retirés dynamiquement. De plus, leut état peut changer au fil du temps : un périphérique réseau peut par exemple modifier son adresse matérielle, voire son nom. Au niveau des périphériques réseau, une modification d'état n'entraîne aucun problème. Mais ceci n'est pas le cas pour les couches supérieures. En effet, pour des raisons d'efficacité et de simplicité, les protocoles de celles-ci stockent fréquemment des renvois aux périphériques réseau dans des caches. Si l'état d'un périphérique change, ces états stockés ne sont plus valides. Les instances de protocoles doivent donc être informées de tout changement d'état. Ceci s'effectue à l'aide des **chaînes de notification** (*notifier-chain* en anglais).

15.3.1.1 Type associé

Les chaînes de notification sont implémentées sous la forme d'une liste `netdev_chain` de structures `notifier_block`.

La chaîne `netdev_chain` est déclarée dans le fichier `linux/net/core/dev.c`: Code Linux 2.6.10

```
207 /*
208  *      Notre liste de notification
209  */
210
211 static struct notifier_block *netdev_chain;
```

Le type général `struct notifier_block` est défini dans le fichier en-tête `linux/include/linux/notifier.h`: Code Linux 2.6.10

```
1 /*
2  *      Routines pour gerer les chaines de notification, pour passer les changements de
3  *      statut a toutes les routines interessees. Nous en avons besoin au lieu des listes
4  *      codees en dur car les modules peuvent s'en meler. Les peripheriques reseau
5  *      en ont besoin.
6  *
7  *                              Alan Cox <Alan.Cox@linux.org>
8  */
[...]
14 struct notifier_block
15 {
16         int (*notifier_call)(struct notifier_block *self, unsigned long, void *);
17         struct notifier_block *next;
18         int priority;
19 };
```

- Le champ `notifier_call()` est un pointeur sur une routine de traitement chargée de gérer la communication au cours du changement d'état du périphérique réseau. Toute instance de protocole devrait comporter une telle fonction de stockage d'états des périphériques réseau.

  Si un changement d'état a lieu dans un périphérique réseau, toutes les routines de traitement stockées dans la liste `netdev_chain` sont appelées. On passe les arguments suivants :

  - un pointeur sur la structure `notifier_block` dont la routine de traitement vient d'être appelée ;
  - un pointeur sur le descripteur de périphérique réseau (`net_device`) concerné, dont l'état change ;
  - l'état qui est à l'origine de la notification.

- `priority` spécifie une priorité, à partir de laquelle la notification sera traitée.

– `next` désigne le prochain élément dans la liste.

On emploie le concept de chaînes de notification pour les périphériques réseau, ce qui nous intéresse ici, mais également pour d'autres états susceptibles de changer. Cela explique pourquoi l'implémentation de ces chaînes de notification est générique. Il existe, par exemple, une chaîne `reboot_notifier_list` pour informer d'un redémarrage imminent du système.

Dans le domaine des réseaux, la liste suivante montre les causes possibles :

– `NETDEV_UP` : un périphérique réseau est activé (*via* `dev_open()`).
– `NETDEV_DOWN` : un périphérique réseau est désactivé.
– `NETDEV_CHANGE` : un périphérique réseau change d'état.
– `NETDEV_REGISTER` : un périphérique réseau a été enregistré mais aucune instance n'a encore été ouverte.
– `NETDEV_UNREGISTER` : un pilote de périphérique réseau a été retiré.
– `NETDEV_CHANGEMTU` : la MTU a été changée.
– `NETDEV_CHANGEADDR` : l'adresse matérielle d'un périphérique réseau a été changée.
– `NETDEV_CHANGENAME` : la désignation d'un périphérique réseau a été changée.

Code Linux 2.6.10

Ces constantes symboliques sont définies dans le fichier en-tête `linux/include/linux/notifier.h` :

```
45 /* chaines de notification des peripheriques reseau */
46 #define NETDEV_UP          0x0001  /* A partir de maintenant vous ne pouvez pas empecher
                                         un peripherique d'etre active/desactive */
47 #define NETDEV_DOWN        0x0002
48 #define NETDEV_REBOOT      0x0003  /* Dit a une pile de protocoles qu'une interface reseau
49                                         a detecte un accident materiel et qu'il a redemarre
50                                         - nous pouvons par exemple utiliser ceci pour bombarder
51                                         des sessions tcp une fois fait */
52 #define NETDEV_CHANGE      0x0004  /* Notifie un changement d'etat de peripherique */
53 #define NETDEV_REGISTER    0x0005
54 #define NETDEV_UNREGISTER  0x0006
55 #define NETDEV_CHANGEMTU   0x0007
56 #define NETDEV_CHANGEADDR  0x0008
57 #define NETDEV_GOING_DOWN  0x0009
58 #define NETDEV_CHANGENAME  0x000A
```

15.3.1.2 Appel de la chaîne de notification

Code Linux 2.6.10

L'appel de la chaîne de notification est mis en œuvre à l'aide de la fonction générale `notifier_call_chain()`, définie dans le fichier `linux/kernel/sys.c` :

```
149 /**
150  *      notifier_call_chain - Fonction d'appel d'une chaine de notification
151  *      @n : Pointeur au pointeur racine de la chaine de notification
152  *      @val : Valeur passee sans modification a la fonction de notification
153  *      @v : Pointeur passe sans modification a la fonction de notification
154  *
155  *      Appelle chaque fonction l'une apres l'autre dans une chaine de notification.
156  *
157  *      Si la valeur de retour de la notification peut etre mise en conjonction
158  *      avec %NOTIFY_STOP_MASK, alors notifier_call_chain
159  *      la renverra immediatement, avec la valeur de retour de
160  *      la fonction de notification qui a stoppe l'execution.
161  *      Sinon la valeur de retour est la valeur de retour
162  *      de la derniere fonction de notification appelee.
163  */
```

```
164
165 int notifier_call_chain(struct notifier_block **n, unsigned long val, void *v)
166 {
167         int ret=NOTIFY_DONE;
168         struct notifier_block *nb = *n;
169
170         while(nb)
171         {
172                 ret=nb->notifier_call(nb,val,v);
173                 if(ret&NOTIFY_STOP_MASK)
174                 {
175                         return ret;
176                 }
177                 nb=nb->next;
178         }
179         return ret;
180 }
```

Elle informe toutes les méthodes de traitement enregistrées dans la liste `n` (`netdev_chain` pour nous) de l'événement `val`. Toutes les routines sont appelées les unes après les autres. Si une de ces routines renvoie la constante `NOTIFY_STOP_MASK`, la notification est interrompue. Les routines enregistrées décident si elles entreprennent quoi que ce soit à la suite de l'avertissement du changement d'état.

Les constantes symboliques générales `NOTIFY_DONE` et `NOTIFY_STOP_MASK` sont définies dans le fichier `linux/include/linux/notifier.h`:

Code Linux 2.6.10

```
28 #define NOTIFY_DONE             0x0000          /* Ne pas en tenir compte */
29 #define NOTIFY_OK               0x0001          /* Me convient */
30 #define NOTIFY_STOP_MASK        0x8000          /* Ne pas rappeler plus tard */
31 #define NOTIFY_BAD              (NOTIFY_STOP_MASK|0x0002)       /* Action Mauvais/Veto */
32 /*
33  * Nettoyer le chemin pour revenir du notificateur et arreter les appels utlerieurs.
34  */
35 #define NOTIFY_STOP             (NOTIFY_OK|NOTIFY_STOP_MASK)
```

15.3.2 Fonction principale

La fonction `register_netdevice()` est définie dans le fichier `linux/net/core/dev.c`:

Code Linux 2.6.10

```
2687 /**
2688  *      register_netdevice      - enregistre un peripherique reseau
2689  *      @dev : peripherique a enregistrer
2690  *
2691  *      Prend une structure de peripherique reseau completee et l'ajoute aux interfaces
2692  *      du noyau. Un message %NETDEV_REGISTER est envoye a la chaine de notification
2693  *      netdev. 0 est renvoye en cas de succes. Un code errno negatif est renvoye
2694  *      en cas d'echec d'installation du peripherique ou si le nom est un doublon.
2695  *
2696  *      Les appelants doivent posseder le semaphore rtnl. Voir le commentaire a la
2697  *      fin de Space.c pour les details sur le verrouillage. Vous pouvez vouloir
2698  *      register_netdev() au lieu de celle-ci.
2699  *
2700  *      BOGUES :
2701  *      Le verrouillage apparait insuffisant pour garantir que deux registres paralleles
2702  *      ne porteront pas le meme nom.
2703  */
2704
2705 int register_netdevice(struct net_device *dev)
2706 {
2707         struct hlist_head *head;
2708         struct hlist_node *p;
```

```
2709         int ret;
2710
2711         BUG_ON(dev_boot_phase);
2712         ASSERT_RTNL();
2713
2714         /* Lorsque des net_device sont persistants, ceci sera fatal. */
2715         BUG_ON(dev->reg_state != NETREG_UNINITIALIZED);
2716
2717         spin_lock_init(&dev->queue_lock);
2718         spin_lock_init(&dev->xmit_lock);
2719         dev->xmit_lock_owner = -1;
2720 #ifdef CONFIG_NET_CLS_ACT
2721         spin_lock_init(&dev->ingress_lock);
2722 #endif
2723
2724         ret = alloc_divert_blk(dev);
2725         if (ret)
2726                 goto out;
2727
2728         dev->iflink = -1;
2729
2730         /* Initialiser, si cette fonction est disponible */
2731         if (dev->init) {
2732                 ret = dev->init(dev);
2733                 if (ret) {
2734                         if (ret > 0)
2735                                 ret = -EIO;
2736                         goto out_err;
2737                 }
2738         }
2739
2740         if (!dev_valid_name(dev->name)) {
2741                 ret = -EINVAL;
2742                 goto out_err;
2743         }
2744
2745         dev->ifindex = dev_new_index();
2746         if (dev->iflink == -1)
2747                 dev->iflink = dev->ifindex;
2748
2749         /* Verification de l'existence du nom */
2750         head = dev_name_hash(dev->name);
2751         hlist_for_each(p, head) {
2752                 struct net_device *d
2753                         = hlist_entry(p, struct net_device, name_hlist);
2754                 if (!strncmp(d->name, dev->name, IFNAMSIZ)) {
2755                         ret = -EEXIST;
2756                         goto out_err;
2757                 }
2758         }
2759
2760         /* Rectifier les combinaisons SG+CSUM illegales. */
2761         if ((dev->features & NETIF_F_SG) &&
2762             !(dev->features & (NETIF_F_IP_CSUM |
2763                                NETIF_F_NO_CSUM |
2764                                NETIF_F_HW_CSUM))) {
2765                 printk("%s: Dropping NETIF_F_SG since no checksum feature.\n",
2766                        dev->name);
2767                 dev->features &= ~NETIF_F_SG;
2768         }
2769
2770         /* TSO exige que SG soit egalement present. */
```

```
2771        if ((dev->features & NETIF_F_TSO) &&
2772            !(dev->features & NETIF_F_SG)) {
2773                printk("%s: Dropping NETIF_F_TSO since no SG feature.\n",
2774                       dev->name);
2775                dev->features &= ~NETIF_F_TSO;
2776        }
2777
2778        /*
2779         *      routine rebuild_header par defaut,
2780         *      qui ne devrait jamais etre appelee et seulement utilisee lors du debogage.
2781         */
2782
2783        if (!dev->rebuild_header)
2784                dev->rebuild_header = default_rebuild_header;
2785
2786        /*
2787         *      L'etat initial par defaut lors de l'enregistrement est que le
2788         *      peripherique est present.
2789         */
2790
2791        set_bit(__LINK_STATE_PRESENT, &dev->state);
2792
2793        dev->next = NULL;
2794        dev_init_scheduler(dev);
2795        write_lock_bh(&dev_base_lock);
2796        *dev_tail = dev;
2797        dev_tail = &dev->next;
2798        hlist_add_head(&dev->name_hlist, head);
2799        hlist_add_head(&dev->index_hlist, dev_index_hash(dev->ifindex));
2800        dev_hold(dev);
2801        dev->reg_state = NETREG_REGISTERING;
2802        write_unlock_bh(&dev_base_lock);
2803
2804        /* Notifie aux protocoles qu'un nouveau peripherique est apparu. */
2805        notifier_call_chain(&netdev_chain, NETDEV_REGISTER, dev);
2806
2807        /* Termine l'enregistrement apres le deverrouillage */
2808        net_set_todo(dev);
2809        ret = 0;
2810
2811 out:
2812        return ret;
2813 out_err:
2814        free_divert_blk(dev);
2815        goto out;
2816 }
```

Autrement dit :

- On déclare une liste de hachage, un élément d'une telle liste et un code de retour.
- On ne permet pas d'enregistrer deux périphériques réseau en même temps ; si on est déjà en train d'en enregistrer un autre, on attend jusqu'à ce qu'on ait terminé l'enregistrement de ce dernier.

  La variable `dev_boot_phase` est déclarée un peu plus haut dans le même fichier : Code Linux 2.6.10

  ```
  2674 static int dev_boot_phase = 1;
  ```

- De même on attend si on ne dispose pas du sémaphore RTNL.

  La macro `ASSERT_RTNL()` est définie dans le fichier `linux/include/linux/rtnetlink.h`: Code Linux 2.6.10

  ```
  812 #define ASSERT_RTNL() do { \
  ```

```
813         if (unlikely(down_trylock(&rtnl_sem) == 0)) { \
814                 up(&rtnl_sem); \
815                 printk(KERN_ERR "RTNL: assertion failed at %s (%d)\n", \
816                        __FILE__,  __LINE__); \
817                 dump_stack(); \
818         } \
819 } while(0)
```

- On attend également si l'état d'enregistrement n'est pas "non enregistré", car sinon on risquerait de se placer dans une boucle infinie.
- On verrouille les files d'attente en lecture et en écriture associées au descripteur de périphérique réseau.
- On indique qu'aucun paquet en émission n'a été transmis.
- On essaie d'allouer une trame de diversion au descripteur de périphérique. Si on n'y parvient pas, on renvoie le code fourni par la fonction `alloc_divert_blk()`, étudiée ci-dessous.
- On positionne le champ `iflink` du descripteur de périphérique à -1.
- Si une fonction spécifique d'initialisation est associée au descripteur de périphérique, on fait appel à elle. Si une erreur intervient lors de son exécution, on renvoie soit le code fourni par cette fonction spécifique, soit l'opposé du code d'erreur `EIO`.

  On a vu, par exemple, que dans le cas de la carte 3Com 501, cette fonction spécifique d'initialisation s'appelle `el1_probe()`.
- On vérifie la validité du nom du périphérique. Si ce n'est pas le cas, on renvoie l'opposé du code d'erreur `EINVAL`.

Code Linux 2.6.10

La fonction `dev_valid_name()` est définie dans le fichier `linux/net/core/dev.c`:

```
640 /**
641  *      dev_valid_name - verifie si le nom est ok pour le peripherique reseau
642  *      @name : chaine de caracteres nom
643  *
644  *      On a besoin que les noms des peripheriques reseau soient des noms de fichiers
645  *      valides pour permettre a sysfs de marcher
646  */
647 int dev_valid_name(const char *name)
648 {
649         return !(*name == '\0'
650                  || !strcmp(name, ".")
651                  || !strcmp(name, "..")
652                  || strchr(name, '/'));
653 }
```

- On attribue un index à ce périphérique, en faisant appel à la fonction `dev_new_index()` étudiée ci-dessous.
- Si le champ `iflink` n'a pas été renseigné (autrement dit si sa valeur est -1) par la fonction spécifique d'initialisation, on lui attribue la valeur de l'index.
- On parcourt la liste de hachage des périphériques réseau d'index associé au nom pour s'assurer qu'il n'avait pas déjà été déclaré; s'il l'avait été, on renvoie l'opposé du code d'erreur `EEXIST`.
- On vérifie la cohérence du champ des caractéristiques et on le modifie éventuellement.
- Si le champ `rebuild_header` n'est pas déjà renseigné, on lui attribue la fonction `default_rebuild_header()` comme valeur. Celle-ci est définie dans le fichier `linux/net/core/dev.c`:

Code Linux 2.6.10

```
799 static int default_rebuild_header(struct sk_buff *skb)
800 {
801         printk(KERN_DEBUG "%s: default_rebuild_header called -- BUG!\n",
```

```
802                skb->dev ? skb->dev->name : "NULL!!!");
803        kfree_skb(skb);
804        return 1;
805 }
```

- On renseigne l'état du périphérique en spécifiant qu'il est présent.
- On met le champ `next` à `NULL` puisqu'il s'agira du dernier de la liste des périphériques réseau.
- On indique l'existence du périphérique au gestionnaire des tâches pour qu'il en tienne compte, grâce à la fonction `dev_init_scheduler()` étudiée ci-dessous.
- On verrouille la liste des périphériques réseau en écriture et on ajoute le périphérique à la fin de celle-ci ainsi que dans la liste de hachage.
- On initialise le nombre d'utilisateurs de ce périphérique.

  La macro `dev_hold()` est définie dans le fichier `linux/include/linux/netdevice.h`: Code Linux 2.6.10

```
678 #define dev_hold(dev) atomic_inc(&(dev)->refcnt)
```

- L'état d'enregistrement du périphérique prend maintenant la valeur "en train d'être enregistré".
- On déverrouille la liste des périphériques réseau.
- On notifie aux protocoles qu'un nouveau périphérique est présent.
- On termine l'installation en ajoutant l'adresse du champ `dev->todo_list` du descripteur de périphérique réseau à la liste `net_todo_list`.

  Cette liste est déclarée et instantiée dans le fichier `linux/net/core/dev.c`. La fonction `set_set_todo()` est définie dans le même fichier: Code Linux 2.6.10

```
2678 static struct list_head net_todo_list = LIST_HEAD_INIT(net_todo_list);
2679
2680 static inline void net_set_todo(struct net_device *dev)
2681 {
2682        spin_lock(&net_todo_list_lock);
2683        list_add_tail(&dev->todo_list, &net_todo_list);
2684        spin_unlock(&net_todo_list_lock);
2685 }
```

- On renvoie 0.

15.3.3 Allocation d'une trame de déroutement

La fonction `alloc_divert_blk()` est définie dans le fichier `linux/net/core/dv.c`: Code Linux 2.6.10

```
6   *              Deroutement generique de trame
7   *
8   * Auteurs :
9   *              Benoit LOCHER :  integration initiale dans le noyau avec support pour ethernet
10  *              Dave Miller :    amelioration du code (correction, performance et
                                     fichiers source)
11  *
12  */
[...]
48 /*
49  * Alloue un divert_blk a un peripherique, qui doit etre une carte ethernet.
50  */
51 int alloc_divert_blk(struct net_device *dev)
52 {
53         int alloc_size = (sizeof(struct divert_blk) + 3) & ~3;
54
```

```
55          if (dev->type == ARPHRD_ETHER) {
56                  printk(KERN_DEBUG "divert: allocating divert_blk for %s\n",
57                         dev->name);
58
59                  dev->divert = (struct divert_blk *)
60                          kmalloc(alloc_size, GFP_KERNEL);
61                  if (dev->divert == NULL) {
62                          printk(KERN_DEBUG "divert: unable to allocate divert_blk for %s\n",
63                                 dev->name);
64                          return -ENOMEM;
65                  } else {
66                          memset(dev->divert, 0, sizeof(struct divert_blk));
67                  }
68                  dev_hold(dev);
69          } else {
70                  printk(KERN_DEBUG "divert: not allocating divert_blk for
                           non-ethernet device %s\n",
71                         dev->name);
72
73                  dev->divert = NULL;
74          }
75          return 0;
76 }
```

Autrement dit :

- On calcule la taille de l'emplacement pour l'allocation.
- Si le périphérique est du type carte réseau Ethernet, on indique qu'on va allouer une trame de déroutement pour ce périphérique et on essaie d'instantier une telle trame dont on place l'adresse dans le champ `divert` du descripteur de périphérique passé en argument. Si on n'y parvient pas, on l'indique par un message d'erreur et on renvoie l'opposé du code d'erreur `ENOMEM`. Si on y parvient, on met à zéro cet emplacement mémoire.
- S'il ne s'agit pas d'une carte réseau Ethernet, on affiche un message noyau indiquant qu'on ne va pas allouer de trame de déroutement pour ce périphérique et on renseigne le champ `divert` avec `NULL`.
- On renvoie 0.

15.3.4 Attribution d'un index

Code Linux 2.6.10

La fonction `dev_new_index()` est définie dans le fichier `linux/net/core/dev.c`:

```
2656 /**
2657  *      dev_new_index   -       alloue un ifindex
2658  *
2659  *      Renvoie une valeur unique convenable pour un nouveau numero d'interface de
2660  *      peripherique. L'appelant doit avoir le semaphore rtnl ou le verrou
2661  *      dev_base_lock pour etre sur qu'il soit unique.
2662  */
2663 static int dev_new_index(void)
2664 {
2665         static int ifindex;
2666         for (;;) {
2667                 if (++ifindex <= 0)
2668                         ifindex = 1;
2669                 if (!__dev_get_by_index(ifindex))
2670                         return ifindex;
2671         }
2672 }
```

qui fait essentiellement appel à la fonction `__dev_get_by_index()`. Celle-ci est définie au début du même fichier :

Code Linux 2.6.10

```
523 /**
524  *      __dev_get_by_index - trouve un peripherique grace a son ifindex
525  *      @ifindex : index du peripherique
526  *
527  *      Cherche une interface grace a son index. Renvoie %NULL si le peripherique
528  *      n'est pas trouve, un pointeur au peripherique sinon. Le peripherique n'a pas eu
529  *      son compteur de reference accru, aussi l'appelant doit-il etre soigneux
530  *      sur le verrouillage. L'appelant doit avoir soit le semaphore RTNL,
531  *      soit @dev_base_lock.
532  */
533
534 struct net_device *__dev_get_by_index(int ifindex)
535 {
536         struct hlist_node *p;
537
538         hlist_for_each(p, dev_index_hash(ifindex)) {
539                 struct net_device *dev
540                         = hlist_entry(p, struct net_device, index_hlist);
541                 if (dev->ifindex == ifindex)
542                         return dev;
543         }
544         return NULL;
545 }
```

qui est l'analogue de `__dev_get_by_name()` pour l'index au lieu du nom.

15.3.5 Rattachement au gestionnaire des tâches

La fonction `dev_init_scheduler()` met en place la procédure d'ordonnancement du nouveau périphérique réseau sur le mécanisme FIFO par défaut. Elle est définie dans le fichier `linux/net/sched/sch_generic.c` :

Code Linux 2.6.10

```
571 void dev_init_scheduler(struct net_device *dev)
572 {
573         qdisc_lock_tree(dev);
574         dev->qdisc = &noop_qdisc;
575         dev->qdisc_sleeping = &noop_qdisc;
576         INIT_LIST_HEAD(&dev->qdisc_list);
577         qdisc_unlock_tree(dev);
578
579         dev_watchdog_init(dev);
580 }
```

Autrement dit :

– On verrouille l'accès en écriture à l'arbre des stratégies de mise en file d'attente.

Le verrou `qdisc_tree_lock` et la fonction `qdisc_lock_tree()` sont définis plus haut dans le même fichier :

Code Linux 2.6.10

```
40 /* Verrou principal de structure qdisc.
41
42    Cependant les modifications
43    aux donnees participant a l'ordonnancement doivent etre protegees
44    en plus avec le verrou tournant dev->queue_lock.
45
46    L'idee est la suivante :
47    - empiler et depiler sont serialises via le verrou tournant du
48      peripherique dev->queue_lock.
```

```
49    - le parcours de l'arbre est protege par read_lock_bh(qdisc_tree_lock)
50      et ce verrou est utilise dans le contexte des processus.
51    - les mises a jour de l'arbre sont faites sous semaphore rtnl ou
52      dans un contexte softirq (__qdisc_destroy rcu-callback),
53      donc ce verrou a besoin d'inhiber les bh locales.
54
55    qdisc_tree_lock doit etre attrape AVANT dev->queue_lock !
56  */
57 rwlock_t qdisc_tree_lock = RW_LOCK_UNLOCKED;
58
59 void qdisc_lock_tree(struct net_device *dev)
60 {
61         write_lock_bh(&qdisc_tree_lock);
62         spin_lock_bh(&dev->queue_lock);
63 }
```

– On renseigne le champ de stratégie de mise en file d'attente du descripteur de périphérique avec la stratégie qui ne fait rien.

Code Linux 2.6.10

La stratégie `noop_qdisc` est définie dans le fichier `linux/net/sched/sch_generic.c`:

```
267 struct Qdisc_ops noop_qdisc_ops = {
268         .next           =       NULL,
269         .cl_ops         =       NULL,
270         .id             =       "noop",
271         .priv_size      =       0,
272         .enqueue        =       noop_enqueue,
273         .dequeue        =       noop_dequeue,
274         .requeue        =       noop_requeue,
275         .owner          =       THIS_MODULE,
276 };
277
278 struct Qdisc noop_qdisc = {
279         .enqueue        =       noop_enqueue,
280         .dequeue        =       noop_dequeue,
281         .flags          =       TCQ_F_BUILTIN,
282         .ops            =       &noop_qdisc_ops,
283         .list           =       LIST_HEAD_INIT(noop_qdisc.list),
284 };
```

– On initialise la liste de la stratégie de mise en file d'attente du descripteur de périphérique.

Code Linux 2.6.10

La macro générale `INIT_LIST_HEAD()` est définie dans le fichier `linux/include/linux/list.h`:

```
37 #define INIT_LIST_HEAD(ptr) do { \
38         (ptr)->next = (ptr); (ptr)->prev = (ptr); \
39 } while (0)
```

– On déverrouille le verrou de file d'attente associé au descripteur de périphérique passé en argument.

Code Linux 2.6.10

La fonction `qdisc_unlock_tree()` est définie plus haut dans le même fichier que la fonction principale:

```
65 void qdisc_unlock_tree(struct net_device *dev)
66 {
67         spin_unlock_bh(&dev->queue_lock);
68         write_unlock_bh(&qdisc_tree_lock);
69 }
```

– on initialise le minuteur en vue de l'identification des problèmes d'émission.

15.3.6 Initialisation du minuteur

La fonction `dev_watchdog_init()` est définie dans le fichier `linux/net/sched/sch_generic.c`: Code Linux 2.6.10

```
208 static void dev_watchdog_init(struct net_device *dev)
209 {
210         init_timer(&dev->watchdog_timer);
211         dev->watchdog_timer.data = (unsigned long)dev;
212         dev->watchdog_timer.function = dev_watchdog;
213 }
```

Autrement dit:

- On initialise le minuteur associé au descripteur de périphérique passé en argument.
- L'adresse des données est l'adresse du descripteur de périphérique elle-même.
- La fonction à appliquer lorsque le délai est écoulé est `dev_watchdog()`, que nous étudierons au moment opportun.

15.4 Mise en fonctionnement des périphériques réseau

La quatrième étape, appelée par la fonction `rtln_unlock()` comme nous l'avons vu ci-dessus, correspond à la remise en fonctionnement de tous les périphériques réseau.

15.4.1 Fonction principale

La fonction `netdev_run_todo()` est définie dans le fichier `linux/net/core/dev.c`: Code Linux 2.6.10

```
2893 void netdev_run_todo(void)
2894 {
2895         struct list_head list = LIST_HEAD_INIT(list);
2896         int err;
2897
2898
2899         /* Exige de prendre garde aux autres cpu. */
2900         down(&net_todo_run_mutex);
2901
2902         /* Pas sain de le faire en dehors du semaphore. Nous ne devons rien renvoyer
2903          * avant que tous les evenements de retrait invoques par le processeur local
2904          * aient ete accomplis (soit par ce todo run, soit par un
2905          * autre cpu).
2906          */
2907         if (list_empty(&net_todo_list))
2908                 goto out;
2909
2910         /* Liste instantanee, permet les requetes ulterieures */
2911         spin_lock(&net_todo_list_lock);
2912         list_splice_init(&net_todo_list, &list);
2913         spin_unlock(&net_todo_list_lock);
2914
2915         while (!list_empty(&list)) {
2916                 struct net_device *dev
2917                         = list_entry(list.next, struct net_device, todo_list);
2918                 list_del(&dev->todo_list);
2919
2920                 switch(dev->reg_state) {
2921                 case NETREG_REGISTERING:
2922                         err = netdev_register_sysfs(dev);
2923                         if (err)
```

```
2924                                    printk(KERN_ERR "%s: failed sysfs registration (%d)\n",
2925                                           dev->name, err);
2926                            dev->reg_state = NETREG_REGISTERED;
2927                            break;
2928
2929                    case NETREG_UNREGISTERING:
2930                            netdev_unregister_sysfs(dev);
2931                            dev->reg_state = NETREG_UNREGISTERED;
2932
2933                            netdev_wait_allrefs(dev);
2934
2935                            /* paranoia */
2936                            BUG_ON(atomic_read(&dev->refcnt));
2937                            BUG_TRAP(!dev->ip_ptr);
2938                            BUG_TRAP(!dev->ip6_ptr);
2939                            BUG_TRAP(!dev->dn_ptr);
2940
2941
2942                            /* Cela doit etre la toute derniere action,
2943                             * apres cela 'dev' peut pointer sur de la memoire liberee.
2944                             */
2945                            if (dev->destructor)
2946                                    dev->destructor(dev);
2947                            break;
2948
2949                    default:
2950                            printk(KERN_ERR "network todo '%s' but state %d\n",
2951                                   dev->name, dev->reg_state);
2952                            break;
2953                    }
2954            }
2955
2956 out:
2957            up(&net_todo_run_mutex);
2958 }
```

Autrement dit :

- On déclare une liste que l'on instantie.
- On s'assure qu'aucun autre microprocesseur ne va agir.

Code Linux 2.6.10

Le sémaphore `net_todo_run_mutex` est défini juste au-dessus dans le même fichier :

```
2892 static DECLARE_MUTEX(net_todo_run_mutex);
```

- Si la liste `net_todo_list` est vide, on a terminé puisqu'il n'y a aucun périphérique à mettre en fonctionnement.
- On agit sur un certain nombre de listes pour préparer les requêtes ultérieures.

Code Linux 2.6.10

Le verrou rotatif liste `net_todo_list_lock` est également déclaré et initialisé dans le même fichier :

```
2676 /* Enregistrement/retrait retarde */
2677 static spinlock_t net_todo_list_lock = SPIN_LOCK_UNLOCKED;
```

Code Linux 2.6.10

La fonction générale `list_splice_init()` est également déclaré et initialisé dans le fichier `linux/include/linux/list.h`:

```
298 /**
299  * list_splice_init - joint deux listes et reinitialise la liste videe.
300  * @list : la nouvelle liste a ajouter.
301  * @head : l'emplacement ou l'ajouter dans la premiere liste.
302  *
```

```
303  * La liste @list est reinitialisee
304  */
305 static inline void list_splice_init(struct list_head *list,
306                                      struct list_head *head)
```

- Pour chaque élément de la liste `net_todo_list` :
  - On récupère le descripteur de périphérique réseau associé.
  - On le retire de la liste.
  - Suivant la valeur de l'état d'enregistrement de ce périphérique :
    - S'il est en train de s'enregistrer, on monte le système de fichiers associé grâce à la fonction `netdev_register_sysfs()` étudiée ci-après et on change l'état en "enregistré".
    - Nous étudierons le cas en train d'être désinstallé au chapitre 35.
    - Dans les autres cas, on affiche un message indiquant qu'on ne fait rien.
- On lève le sémaphore `net_todo_run_mutex`.

15.4.2 Montage du système de fichiers

15.4.2.1 Fonction principale

La fonction `netdev_register_sysfs()` est définie dans le fichier `linux/net/core/sys-fs.c` : Code Linux 2.6.10

```
412 /* Cree les entrees sysfs pour un peripherique reseau. */
413 int netdev_register_sysfs(struct net_device *net)
414 {
415         struct class_device *class_dev = &(net->class_dev);
416         int i;
417         struct class_device_attribute *attr;
418         int ret;
419
420         class_dev->class = &net_class;
421         class_dev->class_data = net;
422
423         strlcpy(class_dev->class_id, net->name, BUS_ID_SIZE);
424         if ((ret = class_device_register(class_dev)))
425                 goto out;
426
427         for (i = 0; (attr = net_class_attributes[i]) != NULL; i++) {
428                 if ((ret = class_device_create_file(class_dev, attr)))
429                     goto out_unreg;
430         }
431
432
433         if (net->get_stats &&
434             (ret = sysfs_create_group(&class_dev->kobj, &netstat_group)))
435                 goto out_unreg;
436
437 #ifdef WIRELESS_EXT
438         if (net->get_wireless_stats &&
439             (ret = sysfs_create_group(&class_dev->kobj, &wireless_group)))
440                 goto out_cleanup;
441
442         return 0;
443 out_cleanup:
444         if (net->get_stats)
445                 sysfs_remove_group(&class_dev->kobj, &netstat_group);
```

```
446 #else
447         return 0;
448 #endif
449
450 out_unreg:
451         printk(KERN_WARNING "%s: sysfs attribute registration failed %d\n",
452                net->name, ret);
453         class_device_unregister(class_dev);
454 out:
455         return ret;
456 }
```

Autrement dit :

- On déclare une entité du type général `struct class_device` que l'on initialise avec le champ adéquat du périphérique réseau passé en argument.
- On déclare une entité du type général `struct class_device_attribute`.
- On renseigne le champ `class` de `class_dev` avec l'adresse de l'instance `net_class`. Cette dernière est définie dans le fichier `linux/net/core/net-sysfs.c` :

Code Linux 2.6.10

```
389 static struct class net_class = {
390         .name = "net",
391         .release = netdev_release,
392 #ifdef CONFIG_HOTPLUG
393         .hotplug = netdev_hotplug,
394 #endif
395 };
```

- On renseigne le champ `class_data` de `class_dev` avec l'adresse passée en argument.
- On renseigne le champ `class_id` de `class_dev` avec le nom du périphérique passé en argument.
  La constante symbolique `BUS_ID_SIZE` est définie dans le fichier `linux/include/linux/-device.h` :

Code Linux 2.6.10

```
28 #define BUS_ID_SIZE             KOBJ_NAME_LEN
```

- On esaie d'enregistrer la classe, grâce à la fonction générale `class_device_register()`. Si on n'y parvient pas, on renvoie le code fourni par cette dernière fonction.
- On essaie d'enregistrer les attributs, grâce à la fonction générale `class_device_create_-file()`. Si on n'y parvient pas pour l'un des attributs, on affiche un message d'erreur, on désintalle la classe de pilotes (grâce à la fonction générale `class_device_unregister()`) et on renvoie le code fourni par la fonction d'enregistrement.

  Le tableau `net_class_attributes[]` est défini dans le fichier `linux/net/core/net--sysfs.c` :

Code Linux 2.6.10

```
188 static struct class_device_attribute *net_class_attributes[] = {
189         &class_device_attr_ifindex,
190         &class_device_attr_iflink,
191         &class_device_attr_addr_len,
192         &class_device_attr_tx_queue_len,
193         &class_device_attr_features,
194         &class_device_attr_mtu,
195         &class_device_attr_flags,
196         &class_device_attr_type,
197         &class_device_attr_address,
198         &class_device_attr_broadcast,
199         &class_device_attr_carrier,
200         NULL
201 };
```

– On essaie de créer le groupe, grâce à la fonction générale `sysfs_create_group()`. Si on n'y parvient pas, on affiche un message d'erreur, on désintalle la classe de pilotes et on renvoie le code fourni par cette dernière fonction.

L'instance `netstat_group` est définie dans le fichier `linux/net/core/net-sysfs.c`: Code Linux 2.6.10

```
286 static struct attribute_group netstat_group = {
287         .name  = "statistics",
288         .attrs  = netstat_attrs,
289 };
```

L'instance `netstat_attrs` est définie dans le fichier `linux/net/core/net-sysfs.c`: Code Linux 2.6.10

```
258 static struct attribute *netstat_attrs[] = {
259         &class_device_attr_rx_packets.attr,
260         &class_device_attr_tx_packets.attr,
261         &class_device_attr_rx_bytes.attr,
262         &class_device_attr_tx_bytes.attr,
263         &class_device_attr_rx_errors.attr,
264         &class_device_attr_tx_errors.attr,
265         &class_device_attr_rx_dropped.attr,
266         &class_device_attr_tx_dropped.attr,
267         &class_device_attr_multicast.attr,
268         &class_device_attr_collisions.attr,
269         &class_device_attr_rx_length_errors.attr,
270         &class_device_attr_rx_over_errors.attr,
271         &class_device_attr_rx_crc_errors.attr,
272         &class_device_attr_rx_frame_errors.attr,
273         &class_device_attr_rx_fifo_errors.attr,
274         &class_device_attr_rx_missed_errors.attr,
275         &class_device_attr_tx_aborted_errors.attr,
276         &class_device_attr_tx_carrier_errors.attr,
277         &class_device_attr_tx_fifo_errors.attr,
278         &class_device_attr_tx_heartbeat_errors.attr,
279         &class_device_attr_tx_window_errors.attr,
280         &class_device_attr_rx_compressed.attr,
281         &class_device_attr_tx_compressed.attr,
282         NULL
283 };
```

– Nous ne nous intéresserons pas aux réseaux sans fil dans cet ouvrage.
– On renvoie 0 pour indiquer que tout s'est bien déroulé.

Chapitre 16

Détection d'une carte réseau (2) Le cas d'Ethernet

Nous venons de voir comment charger un pilote de périphérique. Voyons ce qu'il en est dans le cas d'Ethernet.

16.1 Structures de données pour Ethernet

16.1.1 Constantes

Code Linux 2.6.10

Les constantes nécessaires à l'implémentation Linux d'Ethernet sont définies dans le fichier `linux/include/linux/if_ether.h`:

```
6    *              Definitions globales pour l'interface Ethernet IEEE 802.3.
7    *
8    * Version :    @(#)if_ether.h  1.0.1a  02/08/94
9    *
10   * Auteurs :    Fred N. van Kempen, <waltje@uWalt.NL.Mugnet.ORG>
11   *              Donald Becker, <becker@super.org>
12   *              Alan Cox, <alan@redhat.com>
13   *              Steve Whitehouse, <gw7rrm@eeshack3.swan.ac.uk>
[...]
21   #ifndef _LINUX_IF_ETHER_H
22   #define _LINUX_IF_ETHER_H
23
24   /*
25    *      Constantes magiques Ethernet IEEE 802.3. Les tailles des trames omettent
26    *      le preambule et FCS/CRC (frame check sequence).
27    */
28
29   #define ETH_ALEN        6               /* Octets dans une adresse ethernet      */
30   #define ETH_HLEN        14              /* Nombre total d'octets dans l'en-tete. */
31   #define ETH_ZLEN        60              /* Min. octets dans la trame sans FCS    */
32   #define ETH_DATA_LEN    1500            /* Max. octets dans les donnees          */
33   #define ETH_FRAME_LEN   1514            /* Max. octets dans la trame sans FCS    */
```

Autrement dit:

- Une adresse MAC n'ayant pas de structure particulière, à part le nombre de bits ou d'octets de celle-ci, Linux définit une constante symbolique `ETH_ALEN` (pour *Address LENgth*) spécifiant le nombre d'octets d'une telle adresse.
- La longueur `HLEN` (pour *Header LENgth*) d'un en-tête Ethernet (sans préambule at sans CRC) est de 14 octets.
- La longueur minimum d'une trame (sans préambule et sans CRC) acceptable est de 60 octets pour éviter de prendre en compte les débuts de trames qui subissent une collision.
- La longueur maximum des données, ou MTU Ethernet, est de 1 500 octets, comme nous l'avons déjà vu.
- La longueur maximum de la trame (sans préambule et sans CRC) est donc de 1 514 octets.

16.1.2 Les types de protocole

Nous avons vu que l'en-tête d'une trame Ethernet comprend un champ `type` qui permet de déterminer à quelle suite de protocoles doit être livré la trame. Les types de protocole permis sont repérés par des constantes symboliques définies dans le fichier `linux/include/linux/if_ether.h`:

Code Linux 2.6.10

```
35   /*
36    *      Ce sont les ID des protocoles Ethernet definis.
37    */
38
39   #define ETH_P_LOOP      0x0060          /* Paquet Loopback Ethernet                 */
40   #define ETH_P_PUP       0x0200          /* Paquet Xerox PUP                         */
41   #define ETH_P_PUPAT     0x0201          /* Paquet Trans Adresse Xerox PUP           */
42   #define ETH_P_IP        0x0800          /* Paquet du protocole Internet             */
```

```
43  #define ETH_P_X25       0x0805          /* CCITT X.25                                */
44  #define ETH_P_ARP       0x0806          /* Paquet de Resolution d'Adresse            */
45  #define ETH_P_BPQ       0x08FF          /* Paquet Ethernet G8BPQ AX.25
                                               [ ID NON OFFICIELLEMENT ENREGISTRE]       */
46  #define ETH_P_IEEEPUP   0x0a00          /* Paquet PUP Xerox IEEE802.3                */
47  #define ETH_P_IEEEPUPAT 0x0a01          /* Paquet Trans d'Adresse Xerox IEEE802.3 PUP */
48  #define ETH_P_DEC       0x6000          /* DEC Assigned proto                        */
49  #define ETH_P_DNA_DL    0x6001          /* DEC DNA Dump/Load                         */
50  #define ETH_P_DNA_RC    0x6002          /* DEC DNA Remote Console                    */
51  #define ETH_P_DNA_RT    0x6003          /* Routage DEC DNA                           */
52  #define ETH_P_LAT       0x6004          /* DEC LAT                                   */
53  #define ETH_P_DIAG      0x6005          /* Diagnostic DEC                            */
54  #define ETH_P_CUST      0x6006          /* DEC Customer use                          */
55  #define ETH_P_SCA       0x6007          /* DEC Systems Comms Arch                    */
56  #define ETH_P_RARP      0x8035          /* Paquet de Resolution d'adresse inverse    */
57  #define ETH_P_ATALK     0x809B          /* Appletalk DDP                             */
58  #define ETH_P_AARP      0x80F3          /* Appletalk AARP                            */
59  #define ETH_P_8021Q     0x8100          /* En-tete etendu VLAN 802.1Q                */
60  #define ETH_P_IPX       0x8137          /* IPX sur DIX                               */
61  #define ETH_P_IPV6      0x86DD          /* IPv6 sur bluebook                         */
62  #define ETH_P_WCCP      0x883E          /* Protocole de coordination de cache Web
63                                           * defini dans draft-wilson-wrec-wccp-v2-00.txt */
64  #define ETH_P_PPP_DISC  0x8863          /* Messages de decouverte PPPoE              */
65  #define ETH_P_PPP_SES   0x8864          /* Messages de session PPPoE                 */
66  #define ETH_P_MPLS_UC   0x8847          /* Trafic individuel MPLS                    */
67  #define ETH_P_MPLS_MC   0x8848          /* Trafic multidiffusion MPLS                */
68  #define ETH_P_ATMMPOA   0x884c          /* MultiProtocol sur ATM                     */
69  #define ETH_P_ATMFATE   0x8884          /* Transport fonde sur des trames ATM
70                                           * sur Ethernet
71                                           */
72  #define ETH_P_EDP2      0x88A2          /* Coraid EDP2                               */
73
74  /*
75   *      Types non DIX. Ne devrait pas interferer avec les types 1500.
76   */
77
78  #define ETH_P_802_3     0x0001          /* Type muet pour les trames 802.3           */
79  #define ETH_P_AX25      0x0002          /* Id de protocole muet pour AX.25           */
80  #define ETH_P_ALL       0x0003          /* Chaque paquet (soyez soigneux !!!)        */
81  #define ETH_P_802_2     0x0004          /* Trames 802.2                              */
82  #define ETH_P_SNAP      0x0005          /* En interne seulement                      */
83  #define ETH_P_DDCMP     0x0006          /* DEC DDCMP : en interne seulement          */
84  #define ETH_P_WAN_PPP   0x0007          /* Type muet pour les trames WAN PPP         */
85  #define ETH_P_PPP_MP    0x0008          /* Type muet pour les trames PPP MP          */
86  #define ETH_P_LOCALTALK 0x0009          /* Pseudo-type Localtalk                     */
87  #define ETH_P_PPPTALK   0x0010          /* Type muet pour Atalk sur PPP              */
88  #define ETH_P_TR_802_2  0x0011          /* Trames 802.2                              */
89  #define ETH_P_MOBITEX   0x0015          /* Mobitex (kaz@cafe.net)                    */
90  #define ETH_P_CONTROL   0x0016          /* Trames de controle specifiques a la carte */
91  #define ETH_P_IRDA      0x0017          /* Linux-IrDA                                */
92  #define ETH_P_ECONET    0x0018          /* Acorn Econet                              */
93  #define ETH_P_HDLC      0x0019          /* Trames HDLC                               */
```

dont nous ne retiendrons que les types ETH_P_LOOP, ETH_P_IP et ETH_P_ARP.

16.1.3 Structure d'en-tête de trame

Linux spécifie la structure d'un en-tête de trame Ethernet, constituée uniquement des deux adresses source et de destination et du type, sous le nom ethhdr à la suite dans le même fichier linux/include/linux/if_ether.h:

Code Linux 2.6.10

```
95  /*
```

```
96   *      Ceci est l'en-tete de trame Ethernet.
97   */
98
99  struct ethhdr
100 {
101         unsigned char   h_dest[ETH_ALEN];       /* adr eth de destination     */
102         unsigned char   h_source[ETH_ALEN];     /* adr eth source             */
103         unsigned short  h_proto;                /* champ ID de type de paquet */
104 } __attribute__((packed));
```

16.2 Détection et initialisation des cartes Ethernet

Nous avons vu que, lors du démarrage du système Linux, huit cartes Ethernet sont sondées et leurs pilotes de périphérique installées. Étudions ceci plus en détail maintenant.

16.2.1 Détection des périphériques Ethernet

16.2.1.1 Fonction principale

Code Linux 2.6.10

La fonction `ethif_probe2()` est définie dans le fichier `linux/drivers/net/Space.c`:

```
315 /*
316  * Detection unifiee des peripheriques ethernet, segmente par architecture et
317  * par interface de bus. Ceci pilote uniquement les peripheriques legaux pour l'instant.
318  */
319
320 static void __init ethif_probe2(int unit)
321 {
322         unsigned long base_addr = netdev_boot_base("eth", unit);
323
324         if (base_addr == 1)
325                 return;
326
327         (void)( probe_list2(unit, m68k_probes, base_addr == 0) &&
328                 probe_list2(unit, mips_probes, base_addr == 0) &&
329                 probe_list2(unit, eisa_probes, base_addr == 0) &&
330                 probe_list2(unit, mca_probes, base_addr == 0) &&
331                 probe_list2(unit, isa_probes, base_addr == 0) &&
332                 probe_list2(unit, parport_probes, base_addr == 0));
333 }
```

Autrement dit :

- On déclare une adresse de base, que l'on instantie grâce à la fonction `netdev_boot_base()` étudiée ci-dessous. S'il existe déjà un périphérique portant ce nom ("eth0" pour l'unité égale à zéro), on a terminé.
- On essaie de détecter les cartes pour les microprocesseurs 68000 de Motorola et MIPS, puis pour différents systèmes de bus (EISA, MCA et ISA) et enfin pour le port parallèle, grâce à la fonction `probe_list()` étudiée ci-dessous.

Rappelons que `unit` prenait les valeurs 0 à 7 dans la fonction `net_olddevs_init()`.

16.2.1.2 Instantiation de l'adresse de base

Code Linux 2.6.10

La fonction `netdev_boot_base()` est définie dans le fichier `linux/net/core/dev.c`:

```
410 /**
411  *      netdev_boot_base        - obtient une adresse au moment du demarrage
```

```
412  *      @prefix : prefixe du peripherique reseau
413  *      @unit : id du peripherique reseau
414  *
415  *      Initialisation au moment du demarrage de l'adresse de base du peripherique.
416  *      Les initialisations trouvees sont positionnees pour le peripherique a utiliser
417  *      plus tard lors du sondage du peripherique.
418  *      Renvoie 0 si aucune initialisation n'est trouvee.
419  */
420 unsigned long netdev_boot_base(const char *prefix, int unit)
421 {
422         const struct netdev_boot_setup *s = dev_boot_setup;
423         char name[IFNAMSIZ];
424         int i;
425
426         sprintf(name, "%s%d", prefix, unit);
427
428         /*
429          * Si le peripherique est deja enregistre alors renvoyer une base de 1
430          * pour indiquer de ne pas sonder cette interface
431          */
432         if (__dev_get_by_name(name))
433                 return 1;
434
435         for (i = 0; i < NETDEV_BOOT_SETUP_MAX; i++)
436                 if (!strcmp(name, s[i].name))
437                         return s[i].map.base_addr;
438         return 0;
439 }
```

Autrement dit :

– On déclare un tableau de structures `netdev_boot_setup`, que l'on instantie avec le tableau `dev_boot_setup[]`.

Le type `struct netdev_boot_setup` est défini dans le fichier `linux/include/linux/netdevice.h`: Code Linux 2.6.10

```
243 /*
244  * Cette structure contient les indications des peripheriques configures au moment du
245  * demarrage. Elles sont utilisees pour le sondage des peripheriques.
246  */
247 struct netdev_boot_setup {
248         char name[IFNAMSIZ];
249         struct ifmap map;
250 };
251 #define NETDEV_BOOT_SETUP_MAX 8
```

Le type `struct ifmap` est défini dans le fichier `linux/include/linux/if.h`: Code Linux 2.6.10

```
84  /*
85   *      Structure des parametres du peripherique. Je n'ai fait que concevoir un
86   *      beau schema utilisant seulement les modules chargeables avec des arguments
87   *      pour les options des peripheriques et alors vinrent les gens du PCMCIA 8)
88   *
89   *      Ah bien. Le cote get() de ceci est bon pour WDSETUP, et il sera manipulable
90   *      pour deboguer des choses. Le cote set() est bien pour l'instant et etant
91   *      tres petit il pourrait etre pire de le garder pour une configuration propre.
92   */
93
94  struct ifmap
95  {
96          unsigned long mem_start;
97          unsigned long mem_end;
```

```
98          unsigned short base_addr;
99          unsigned char irq;
100         unsigned char dma;
101         unsigned char port;
102         /* 3 octets disperses */
103 };
```

Code Linux 2.6.10

Le tableau `dev_boot_setup[]` est défini dans le fichier `linux/net/core/dev.c`:

```
346 /************************************************************************
347
348           Routines pour le parametrage des peripheriques au moment du demarrage
349
350 ************************************************************************/
351
352 /* Table de configuration au moment du demarrage */
353 static struct netdev_boot_setup dev_boot_setup[NETDEV_BOOT_SETUP_MAX];
```

- On déclare un nom de périphérique réseau et un index.
- On initialise le nom avec le préfxe et l'unité passés en argument.
  On obtiendra donc "eth0" pour le premier.
- Si un périphérique portant ce nom est déjà enregistré, on renvoie 1.
- Si ce nom apparaît dans le tableau `dev_boot_setup[]`, on renvoie l'adresse de base spécifiée dans ce tableau.
- Sinon on renvoie 0.

16.2.1.3 Détection suivant une liste

Code Linux 2.6.10

La fonction `probe_list2()` est définie dans le fichier `linux/drivers/net/Space.c`:

```
111 static int __init probe_list2(int unit, struct devprobe2 *p, int autoprobe)
112 {
113         struct net_device *dev;
114         for (; p->probe; p++) {
115                 if (autoprobe && p->status)
116                         continue;
117                 dev = p->probe(unit);
118                 if (!IS_ERR(dev))
119                         return 0;
120                 if (autoprobe)
121                         p->status = PTR_ERR(dev);
122         }
123         return -ENODEV;
124 }
```

Autrement dit:

- On parcourt la liste passée en argument à la recherche d'un périphérique:
  - Si la valeur `autoprobe` de l'adresse de base passée en argument est non nulle ainsi que le statut, on ne fait rien pour cet élément.
    Ce n'est pas le cas des six listes que nous avons à considérer puisque l'adresse de base était toujours zéro.
  - On essaie de détecter le périphérique associé à l'élément de cette liste et à l'unité passée en argument.
  - Si on n'a pas trouvé ce périphérique, on renvoie 0. Sinon, si l'adresse de base passée en argument était non nulle, on renseigne le champ statut de cet élément.
- Si aucune des fonctions de détection n'a réussi, on renvoie l'opposé du code d'erreur `ENODEV`.

Le type `struct devprobe2` est défini dans le fichier `linux/drivers/net/Space.c`: Code Linux 2.6.10

```
106 struct devprobe2 {
107         struct net_device *(*probe)(int unit);
108         int status;     /* non-nul si l'auto-detection a echouee */
109 };
```

Les six listes (en fait tableaux) sont définies dans le fichier `linux/drivers/net/Space.c`, par exemple pour `isa_probes[]` : Code Linux 2.6.10

```
165 /*
166  * ISA detecte les adresses < 0x400 (y compris celles qui recherchent aussi
167  * les cartes EISA/PCI/MCA en plus des cartes ISA).
168  */
169 static struct devprobe2 isa_probes[] __initdata = {
170 #ifdef CONFIG_HP100             /* ISA, EISA & PCI */
171         {hp100_probe, 0},
172 #endif
173 #ifdef CONFIG_3C515
174         {tc515_probe, 0},
175 #endif
176 #ifdef CONFIG_ULTRA
177         {ultra_probe, 0},
178 #endif
179 #ifdef CONFIG_WD80x3
180         {wd_probe, 0},
181 #endif
182 #ifdef CONFIG_EL2               /* 3c503 */
183         {el2_probe, 0},
[...]
231 #ifdef CONFIG_EL1               /* 3c501 */
232         {el1_probe, 0},
233 #endif
[...]
258         {NULL, 0},
259 };
```

16.2.2 Initialisation d'une carte Ethernet

Lors de la conception du pilote d'une carte particulière, on fait heureusement, pour la plus grande part, appel à une initialisation par défaut, implémentée dans le noyau Linux pour les grands types de cartes (située dans le fichier `linux/drivers/net/net_init.c`), ce qui facilite le travail de conception d'un pilote de périphérique :

- `ether_setup()` pour les cartes Ethernet, que nous allons étudier ici ;
- `ltalk_setup()` pour les cartes *LocalTalk* ;
- `fc_setup()` pour les périphériques utilisant de la fibre optique ;
- `fddi_setup()` pour les réseaux FDDI (*Fiber Distributed Data Interface*) ;
- `hippi_setup()` pour les périphériques à haut débit HIPPI ;
- `tr_configure()` pour configurer les interfaces réseau *token ring*.

La mise en place de valeurs par défaut dans le cas de Ethernet est effectuée grâce à la fonction `ether_setup()`. Celle-ci est définie dans le fichier `linux/net/ethernet/eth.c`: Code Linux 2.6.10

```
266 /*
267  * Renseigne les champs de la structure de peripherique avec les valeurs generiques
             d'ethernet.
268  */
269 void ether_setup(struct net_device *dev)
```

```
270 {
271         dev->change_mtu         = eth_change_mtu;
272         dev->hard_header        = eth_header;
273         dev->rebuild_header     = eth_rebuild_header;
274         dev->set_mac_address    = eth_mac_addr;
275         dev->hard_header_cache  = eth_header_cache;
276         dev->header_cache_update= eth_header_cache_update;
277         dev->hard_header_parse  = eth_header_parse;
278
279         dev->type               = ARPHRD_ETHER;
280         dev->hard_header_len    = ETH_HLEN;
281         dev->mtu                = 1500; /* eth_mtu */
282         dev->addr_len           = ETH_ALEN;
283         dev->tx_queue_len       = 1000; /* Ethernet exige de grandes files d'attente */
284         dev->flags              = IFF_BROADCAST|IFF_MULTICAST;
285
286         memset(dev->broadcast,0xFF, ETH_ALEN);
287
288 }
```

Autrement dit :

- elle assigne des valeurs par défaut au descripteur de périphérique réseau pour certaines fonctions de gestions ;
- elle indique le type matériel de l'interface (utilisé par ARP pour déterminer quelle sorte d'adresse physique est supportée par le périphérique réseau) ; il s'agit ici évidemment du type associé à Ethernet ;
- elle spécifie la longueur de l'en-tête d'une trame, à savoir 14 octets pour Ethernet ;
- elle spécifie la longueur maximale d'une trame, ou MTU, à savoir 1 500 octets pour Ethernet ; remarquons qu'on aurait pu utiliser la constante symbolique `ETH_DATA_LEN` ;
- elle spécifie la longueur effective d'une adresse, à savoir six octets dans le cas d'Ethernet ;
- elle spécifie la taille maximale de la file d'attente en transmission, à savoir 1000 trames pour Ethernet ;
- elle spécifie l'adresse de diffusion générale, à savoir six octets de valeur `FFh`, c'est-à-dire que des 1.

Les fonctions de gestion Ethernet par défaut devront être étudiées au fur et à mesure des besoins. En fait nous n'en aurons pas besoin dans cet ouvrage.

16.2.3 Allocation d'un descripteur de périphérique réseau

16.2.3.1 Cas de Ethernet

Code Linux 2.6.10

La fonction `alloc_etherdev()` est définie dans le fichier `linux/net/ethernet/eth.c`:

```
291 /**
292  * alloc_etherdev - Alloue et initialise un peripherique ethernet
293  * @sizeof_priv : Taille de la structure privee additionnelle du pilote a allouer
294  *      pour ce peripherique ethernet
295  *
296  * Renseigne les champs de la structure du peripherique avec les valeurs generiques
297  * ethernet. Ne fait rien d'autre de fondamental que d'enregistrer le peripherique.
298  *
299  * Construit un nouveau peripherique reseau, le complete avec une zone de donnees
300  * privees de taille @sizeof_priv. Un alignement de 32 octets (pas bits) est force
301  * pour cette zone de donnees privees.
302  */
303
```

```
304 struct net_device *alloc_etherdev(int sizeof_priv)
305 {
306         return alloc_netdev(sizeof_priv, "eth%d", ether_setup);
307 }
```

qui renvoie à la fonction générale `alloc_netdev()` avec les deux paramètres propres à Ethernet, à savoir le nom et la fonction d'initialisation `ether_setup()`.

16.2.3.2 Cas général

La fonction générale est définie dans le fichier `linux/drivers/net/net_init.c`: Code Linux 2.6.10

```
73  struct net_device *alloc_netdev(int sizeof_priv, const char *mask,
74                                  void (*setup)(struct net_device *))
75  {
76          void *p;
77          struct net_device *dev;
78          int alloc_size;
79
80          /* On s'assure d'un alignement sur 32 octets de la zone privee */
81
82          alloc_size = (sizeof(struct net_device) + NETDEV_ALIGN_CONST)
83                          & ~NETDEV_ALIGN_CONST;
84          alloc_size += sizeof_priv + NETDEV_ALIGN_CONST;
85
86          p = kmalloc (alloc_size, GFP_KERNEL);
87          if (!p) {
88                  printk(KERN_ERR "alloc_dev: Unable to allocate device.\n");
89                  return NULL;
90          }
91
92          memset(p, 0, alloc_size);
93
94          dev = (struct net_device *)(((long)p + NETDEV_ALIGN_CONST)
95                                  & ~NETDEV_ALIGN_CONST);
96          dev->padded = (char *)dev - (char *)p;
97
98          if (sizeof_priv)
99                  dev->priv = netdev_priv(dev);
100
101         setup(dev);
102         strcpy(dev->name, mask);
103
104         return dev;
105 }
```

Autrement dit :

- On déclare une adresse, un descripteur de périphérique et une taille.
- On initialise cette taille en tenant compte de la taille d'un descripteur de périphérique, de la zone des données privées et de l'alignement sur 32 octets.

  La constante symbolique `NETDEV_ALIGN_CONST` est définie dans le fichier `linux/include/linux/netdevice.h`: Code Linux 2.6.10

  ```
  493 #define NETDEV_ALIGN            32
  494 #define NETDEV_ALIGN_CONST      (NETDEV_ALIGN - 1)
  ```

- On essaie de réserver de la place en mémoire vive pour le descripteur de périphérique. Si on n'y parvient pas, on affiche un message noyau et on renvoie `NULL`.
- On initialise cette zone de mémoire à zéro.

- On initialise l'adresse du descripteur de périphérique.
- On renseigne le champ de remplissage de ce descripteur de périphérique.
- Si la taille de la zone privée passée en argument est non nulle, on renseigne le champ concernant celle-ci du descripteur de périphérique.

Code Linux 2.6.10

La fonction en ligne `netdev_priv()` est définie dans le fichier `linux/include/linux/-netdevice.h`:

```
496 static inline void *netdev_priv(struct net_device *dev)
497 {
498         return (char *)dev + ((sizeof(struct net_device)
499                                         + NETDEV_ALIGN_CONST)
500                                 & ~NETDEV_ALIGN_CONST);
501 }
```

- On fait appel à la fonction d'initialisation passée en argument, qui dépend du type de carte réseau.
- On renseigne le champ nom du descripteur de périphérique réseau avec le nom passé en argument.
- On renvoie l'adresse du descripteur de périphérique ainsi créé.

16.3 Opérations liées à Ethernet

Nous avons vu au chapitre 14 que le descripteur de périphérique réseau prévoie un champ faisant référence à un ensemble d'opérations Ethernet (champ égal à `NULL` dans le cas d'un périphérique non Ethernet).

Code Linux 2.6.10

Le type `struct ethtool_ops` de ce champ est défini dans le fichier `linux/include/linux/-ethtool.h`:

```
264 /**
265  * &ethtool_ops - Change et rappelle les positionnements du peripherique reseau
266  * get_settings : recupere les positionnements specifiques au peripherique
267  * set_settings : positionne ce qui est specifique au peripherique
268  * get_drvinfo : recupere les informations sur le peripherique
269  * get_regs : recupere les registres du peripherique
270  * get_wol : dit si Wake-on-Lan est active
271  * set_wol: active ou desactive Wake-on-Lan
272  * get_msglevel : recupere la couche message du peripherique
273  * set_msglevel : positionne la couche message du peripherique
274  * nway_reset : redemarre l'autonegociation
275  * get_link : recupere le statut de lien
276  * get_eeprom : lit les donnees depuis l'EEPROM du peripherique
277  * set_eeprom : ecrit des donnees sur l'EEPROM du peripherique
278  * get_coalesce : recupere les parametres de combinaison de l'interruption
279  * set_coalesce : positionne les parametres de combinaison de l'interruption
280  * get_ringparam : recupere les tailles de l'anneau
281  * set_ringparam : positionne les tailles de l'anneau
282  * get_pauseparam : recupere les parametres de pause
283  * set_pauseparam : positionne les parametres de pause
284  * get_rx_csum : dit si le controle de la somme de controle en reception est activee ou non
285  * set_rx_csum : active ou desactive le controle de la somme de controle en reception
286  * get_tx_csum : dit si le controle de la somme de controle en emission est active ou non
287  * set_tx_csum : active ou desactive le controle de la somme de controle en emission
288  * get_sg : dit si la dispersion-rassemblement est activee
289  * set_sg : active ou desactive la dispersion-rassemblement
290  * get_tso : dit si la segmentation TCP est activee
291  * set_tso : active ou desactive la segmentation TCP
292  * self_test : effectue les auto-tests specifies
```

```
293  * get_strings : renvoie un ensemble de chaines de caracteres qui decrit les objets demandes
294  * phys_id : identifie le peripherique
295  * get_stats : renvoie les statistiques concernant le peripherique
296  *
297  * Description :
298  *
299  * get_settings :
300  *      @get_settings passe une &ethtool_cmd a renseigner. Elle renvoie
301  *      un errno negatif ou zero.
302  *
303  * set_settings :
304  *      @set_settings passe une &ethtool_cmd et doit essayer de positionner
305  *      tout ce que supporte ce peripherique. Elle renvoie une valeur d'erreur
306  *      si quelque chose se passe mal (0 sinon).
307  *
308  * get_eeprom :
309  *      Doit renseigner le champ magic. Ne necessite pas de verifier si len est nulle
310  *      ou l'emballage. Renseigne l'argument des donnees avec les valeurs eeprom
311  *      de offset a offset + len. Met a jour len sur la quantite lue.
312  *      Renvoie une erreur ou zero.
313  *
314  * set_eeprom :
315  *      Doit valider le champ magic. Ne necessite pas de verifier si len est nulle
316  *      ou l'emballage. Met a jour la quantite ecrite. Renvoie une erreur
317  *      ou zero.
318  */
319 struct ethtool_ops {
320         int     (*get_settings)(struct net_device *, struct ethtool_cmd *);
321         int     (*set_settings)(struct net_device *, struct ethtool_cmd *);
322         void    (*get_drvinfo)(struct net_device *, struct ethtool_drvinfo *);
323         int     (*get_regs_len)(struct net_device *);
324         void    (*get_regs)(struct net_device *, struct ethtool_regs *, void *);
325         void    (*get_wol)(struct net_device *, struct ethtool_wolinfo *);
326         int     (*set_wol)(struct net_device *, struct ethtool_wolinfo *);
327         u32     (*get_msglevel)(struct net_device *);
328         void    (*set_msglevel)(struct net_device *, u32);
329         int     (*nway_reset)(struct net_device *);
330         u32     (*get_link)(struct net_device *);
331         int     (*get_eeprom_len)(struct net_device *);
332         int     (*get_eeprom)(struct net_device *, struct ethtool_eeprom *, u8 *);
333         int     (*set_eeprom)(struct net_device *, struct ethtool_eeprom *, u8 *);
334         int     (*get_coalesce)(struct net_device *, struct ethtool_coalesce *);
335         int     (*set_coalesce)(struct net_device *, struct ethtool_coalesce *);
336         void    (*get_ringparam)(struct net_device *, struct ethtool_ringparam *);
337         int     (*set_ringparam)(struct net_device *, struct ethtool_ringparam *);
338         void    (*get_pauseparam)(struct net_device *, struct ethtool_pauseparam*);
339         int     (*set_pauseparam)(struct net_device *, struct ethtool_pauseparam*);
340         u32     (*get_rx_csum)(struct net_device *);
341         int     (*set_rx_csum)(struct net_device *, u32);
342         u32     (*get_tx_csum)(struct net_device *);
343         int     (*set_tx_csum)(struct net_device *, u32);
344         u32     (*get_sg)(struct net_device *);
345         int     (*set_sg)(struct net_device *, u32);
346         u32     (*get_tso)(struct net_device *);
347         int     (*set_tso)(struct net_device *, u32);
348         int     (*self_test_count)(struct net_device *);
349         void    (*self_test)(struct net_device *, struct ethtool_test *, u64 *);
350         void    (*get_strings)(struct net_device *, u32 stringset, u8 *);
351         int     (*phys_id)(struct net_device *, u32);
352         int     (*get_stats_count)(struct net_device *);
353         void    (*get_ethtool_stats)(struct net_device *, struct ethtool_stats *,
                                     u64 *);
```

```
354         int     (*begin)(struct net_device *);
355         void    (*complete)(struct net_device *);
356 };
```

Code Linux 2.6.10

Le type `ethtool_cmd` est défini dans le fichier `linux/include/linux/ethtool.h`:

```
16 /* Ceci doit marcher a la fois pour les espaces utilisateur 32 et 64 bits. */
17 struct ethtool_cmd {
18         u32     cmd;
19         u32     supported;      /* Caracteristiques que cette interface supporte */
20         u32     advertising;    /* Caracteristiques dont cette interface fait la publicite */
21         u16     speed;          /* La vitesse forcee : 10Mb, 100Mb, gigabit */
22         u8      duplex;         /* Duplex, half ou full */
23         u8      port;           /* Quel port connecteur */
24         u8      phy_address;
25         u8      transceiver;    /* Quel transcepteur utiliser */
26         u8      autoneg;        /* Active ou desactive l'auto-negociation */
27         u32     maxtxpkt;       /* Paquets transmis avant de lever l'interruption de
                                      transmission */
28         u32     maxrxpkt;       /* Paquets recus avant de lever l'interruption de reception */
29         u32     reserved[4];
30 };
```

Ces fonctions devraient être étudiées au fur et à mesure des besoins. En fait aucune ne sera utilisée dans cet ouvrage.

Chapitre 17

Détection d'une carte réseau (3) Cas de la carte 3Com 501

Intéressons-nous à un pilote en particulier pour voir à quoi cela ressemble. Nous choisissons celui de la carte 3Com 501, qui a été l'une des premières à être implémentées sous Linux.

17.1 Description du contrôleur de la carte

17.1.1 Commentaire général

Le pilote de périphérique de la carte 3Com 501 fait l'objet des deux fichiers `linux/drivers/net/3c501.h` et `linux/drivers/net/3c501.c`. Un commentaire général se trouve au début du second fichier :

Code Linux 2.6.10

```
1   /* 3c501.c : un pilote Ethernet pour Linux de 3Com 3c501. */
2   /*
3       Ecrit en 1992,1993,1994 par Donald Becker
4
5       Copyright 1993 United States Government, represente par le
6       Directeur de la National Security Agency. Ce logiciel peut etre utilise et
7       distribue conformement aux termes de la GNU General Public License,
8       incorporee ici pour reference.
9
10      Ceci est le pilote de peripherique pour 3Com Etherlink 3c501.
11      N'achetez pas cette carte, meme par plaisanterie. Ses performances sont horribles
12      et elle s'arrete souvent.
13
14      L'auteur originel peut etre contacte par becker@scyld.com ou C/O
15          Scyld Computing Corporation
16          410 Severn Ave., Suite 210
17          Annapolis MD 21403
18
19      Rectifie (encore !) le verrouillage manquant de l'interruption sur le decalage TX/RX.
20                  Alan Cox <Alan.Cox@linux.org>
21
22      Appels a init_etherdev supprimes puisqu'ils ne sont plus necessaires et
23      un peu de menage sur la modularisation. Le pilote ne permet encore que
24      l'adresse par defaut pour les cartes chargees comme module, mais ceci est
25      reellement moins stupide que quiconque utilisant une carte 3c501. :)
26                      19950208 (invid@msen.com)
27
28      Trappes ajoutees pour les interruption frappant a la fenetre lorsque nous effacons et
29      chargeons en TX la carte. Obtenons maintenant 150K/seconde pour FTP avec une carte 3c501.
30      Je joue encore a l'optimisation TX-TX pour voir si nous pouvons atteindre 180-200K/seconde
31      qui semble etre le maximum theorique.
32                  19950402 Alan Cox <Alan.Cox@linux.org>
33
34      Nettoye pour 2.3.x puisque nous brisons maintenant la SMP.
35                  20000208 Alan Cox <alan@redhat.com>
36
37      Verification pour 2.5. Aucun changement significatif
38                  20021009 Alan Cox <alan@redhat.com>
39
40      Rectifie le cas du coin rempli avec des zeros
41                  20030104 Alan Cox <alan@redhat.com>
42
43
44     L'evidence des doutes sur la "forme preferee" de ce code a conduit a
45     un format ouvert non encombre de brevet. Puisque la cle cryptographique de signature
46     fait partie du processus de creation d'un executable, les informations
47     incluant les cles necessaires pour generer un executable fonctionnellement equivalent
48     sont considerees comme faisant parties du code source.
49
50  */
51
52
53  /**
54   * DOC : Notes sur la carte 3c501
```

```
55  *
56  *  Quelques notes sur cette chose si vous avez a l'etudier.  [Alan]
57  *
58  *  De la documentation est disponible chez 3Com. Du a l'age de la carte,
59  *  la reponse standard lorsque vous demandez celle-ci ira de 'soyez serieux'
60  *  a 'donnez-la a un musee'. La documentation est incomplete et pour une bonne part
61  *  d'un interet purement historique.
62  *
63  *  Le systeme de base est un unique tampon qui peut etre utilise pour recevoir ou
64  *  transmettre un paquet. Un troisieme mode de commande existe lorsque vous activez
65  *  la carte.
66  *
67  *  Si elle est en train d'emettre, elle ne peut pas recevoir et vice versa.
68  *  En effet le temps de retourner la carte dans un etat utile apres une operation
69  *  est tres grand.
70  *
71  *  Le pilote marche en gardant la carte en mode reception en attendant qu'un
72  *  paquet arrive. Lorsqu'il en arrive un, il est copie hors du tampon
73  *  et remis au noyau. La carte est rechargee et on y va.
74  *
75  *  Lorsqu'on emet, lp->txing est positionne et la carte est reinitialisee (du
76  *  mode reception) [en perdant eventuellement un paquet juste recu] au mode
77  *  commande. Un paquet est charge et le mode d'emission est lance. La routine
78  *  d'interruption execute un code different pour les interruptions d'emission et
79  *  peut s'occuper de revenir au mode reception ou emission (oui vous devez
80  *  l'aider pour ceci aussi).
81  *
82  * DOC : Problemes
83  *
84  *  Il y a de nombreux retours sur erreur non documentes depuis la carte,
85  *  aussi avez-vous fondamentalement a lui donner un coup de pied et a prier si ca se
86  *  passe mal. Beacoup d'erreurs apparaissent seulement en charge extreme ou si vous
87  *  faites quelque chose que la carte n'aime pas (par exemple toucher a un registre
       au mauvais moment).
88  *
89  *  Le pilote est moins efficace que ce qu'il pourrait etre. Il passe en
90  *  mode reception meme s'il y a encore a emettre. Si cela vous ennuie, achetez
91  *  une vraie carte Ethernet.
92  *
93  *  La combinaison d'un redemarrage en reception lent et pas de vrai filtre multidiffusion
94  *  fait que la carte est inutilisable avec un noyau compile pour la multidiffusion IP
95  *  dans un vrai environnement de multidiffusion. Ceci est du a la carte,
96  *  mais meme sans programme de multidiffusion executant une multidiffusion IP, le
97  *  noyau est dans le groupe 224.0.0.1 et vous entendrez toutes les multidiffusions.
98  *  Une conference s'executant sur cette carte Ethernet et vous renoncerez.
99  *
100 */
```

17.1.2 Les registres

17.1.2.1 Adresses des registres

Les adresses des registres de la carte 3Com 501 sont repérées, par rapport à l'adresse de base, par des constantes symboliques définies dans le fichier `linux/drivers/net/3c501.h`:

Code Linux 2.6.10

```
41 #define RX_STATUS (ioaddr + 0x06)
42 #define RX_CMD    RX_STATUS
43 #define TX_STATUS (ioaddr + 0x07)
44 #define TX_CMD    TX_STATUS
45 #define GP_LOW    (ioaddr + 0x08)
46 #define GP_HIGH   (ioaddr + 0x09)
47 #define RX_BUF_CLR (ioaddr + 0x0A)
```

```
48 #define RX_LOW    (ioaddr + 0x0A)
49 #define RX_HIGH   (ioaddr + 0x0B)
50 #define SAPROM    (ioaddr + 0x0C)
51 #define AX_STATUS (ioaddr + 0x0E)
52 #define AX_CMD    AX_STATUS
53 #define DATAPORT  (ioaddr + 0x0F)
[...]
56 #define EL1_DATAPTR     0x08
57 #define EL1_RXPTR       0x0A
58 #define EL1_SAPROM      0x0C
59 #define EL1_DATAPORT    0x0f
```

Remarquons que `ioaddr` n'est pas défini globalement, mais dans chaque contexte qui utilise les adresses des registres, ce qui est possible puisqu'il s'agit de macros et non de fonctions. Nous verrons le cas, par exemple, pour la fonction `el_receive()` que nous étudierons plus loin.

17.1.2.2 Les registres TCR et TSR

Code Linux 2.6.10

Les bits des registres TCR et TSR sont repérés grâce aux constantes symboliques définies dans le fichier `linux/drivers/net/3C501.h`:

```
54 #define TX_RDY          0x08      /* Dans TX_STATUS */
[...]
82 /*
83  *      Registre TX_STATUS.
84  */
85
86 #define TX_COLLISION 0x02
87 #define TX_16COLLISIONS 0x04
88 #define TX_READY 0x08
```

17.1.2.3 Le registre RCR

Code Linux 2.6.10

Les valeurs du registre RCR sont repérées grâce aux constantes symboliques définies dans le fichier `linux/drivers/net/3C501.h`:

```
72 /*
73  *      Mode de reception normal ecrit dans RX_STATUS. Nous devons lever une interruption
74  *      sur les petits paquets pour eviter les verrouillages en reception bidons.
75  */
76
77 #define RX_NORM 0xA8            /* 0x68 == toutes les adresses, 0xA8 seulement pour moi. */
78 #define RX_PROM 0x68            /* Promiscuite senior, hum mode promiscuite.             */
79 #define RX_MULT 0xE8            /* Accepte les paquets multidiffusion.                   */
```

17.1.2.4 Le registre RSR

Code Linux 2.6.10

Les valeurs du registre RSR sont repérées grâce aux constantes symboliques définies dans le fichier `linux/drivers/net/3C501.h`:

```
90 #define RX_RUNT   0x08
91 #define RX_MISSED 0x01       /* A manque un paquet a cause d'une lesion cerebrale de 3c501. */
92 #define RX_GOOD   0x30       /* Bon paquet 0x20 ou depassement simple 0x10.                 */
```

17.1.2.5 Le registre RCA

Code Linux 2.6.10

Les valeurs du registre RCA sont repérées grâce aux constantes symboliques définies dans le fichier `linux/drivers/net/3C501.h`:

```
61 /*
```

```
62  *      Ecrit sur le registre de commande ax.
63  */
64
65 #define AX_OFF   0x00                  /* Irq inhibe, acces au tampon active    */
66 #define AX_SYS   0x40                  /* Charge le tampon                      */
67 #define AX_XMIT  0x44                  /* Emet un paquet                    */
68 #define AX_RX    0x48                  /* Recoit un paquet                      */
69 #define AX_LOOP  0x0C                  /* Mode loopback                         */
70 #define AX_RESET 0x80
[...]
80 #define TX_NORM  0x0A                  /* Leve une interruption pour chaque chose
                                             qui peut accrocher le microprocesseur */
```

17.2 Détection et initialisation de la carte 3Com

Nous avons vu au chapitre 15 que les pilotes de carte réseau font appel à une fonction de détection et d'initialisation. Nous avons vu également qu'il est fait appel à la fonction `init()` dans le corps de la fonction `register_netdevice()` ainsi qu'à la fonction de destruction `destructor()` spécifiques à la carte réseau. La carte 3Com 501 ne possède pas de fonction de destruction spécifique. Nous avons vu, à propos de `isa_probes[]` et de `dev_3c501`, que la fonction de détection et d'initialisation s'appelle `el1_probe()` dans le cas de la carte 3Com501 (que le pilote de celle-ci soit lié statiquement au noyau ou modularisé).

17.2.1 Fonction principale

La fonction `el1_probe()` est définie dans le fichier `linux/drivers/net/3c501.c`: Code Linux 2.6.10

```
139 /*
140  *      Le code de detection [de la chaudiere].
141  */
142
143 static int io=0x280;
144 static int irq=5;
145 static int mem_start;
146
147 /**
148  * el1_probe :          -       detecte une 3c501
149  * @dev : la structure de peripherique passee pour la detection.
150  *
151  * Ceci peut etre appele depuis deux endroits. La couche reseau sondera en utilisant
152  * une structure de peripherique passee avec les informations de sondage completees. Pour un
153  * peripherique modularise, nous utilisons #init_module pour renseigner notre propre structure
154  * et pour la detecter.
155  *
156  * Renvoie 0 en cas de succes, ENXIO s'il n'est pas demande de sonder et ENODEV s'il est
157  * demande de sonder et qu'on echoue a trouver quelque chose.
158  */
159
160 struct net_device * __init el1_probe(int unit)
161 {
162         struct net_device *dev = alloc_etherdev(sizeof(struct net_local));
163         static unsigned ports[] = { 0x280, 0x300, 0};
164         unsigned *port;
165         int err = 0;
166
167         if (!dev)
168                 return ERR_PTR(-ENOMEM);
169
170         if (unit >= 0) {
```

```
171                 sprintf(dev->name, "eth%d", unit);
172                 netdev_boot_setup_check(dev);
173                 io = dev->base_addr;
174                 irq = dev->irq;
175                 mem_start = dev->mem_start & 7;
176         }
177
178         SET_MODULE_OWNER(dev);
179
180         if (io > 0x1ff) {       /* Verifie un seul emplacement specifie. */
181                 err = el1_probe1(dev, io);
182         } else if (io != 0) {
183                 err = -ENXIO;           /* Ne pas sonder du tout. */
184         } else {
185                 for (port = ports; *port && el1_probe1(dev, *port); port++)
186                         ;
187                 if (!*port)
188                         err = -ENODEV;
189         }
190         if (err)
191                 goto out;
192         err = register_netdev(dev);
193         if (err)
194                 goto out1;
195         return dev;
196 out1:
197         release_region(dev->base_addr, EL1_IO_EXTENT);
198 out:
199         free_netdev(dev);
200         return ERR_PTR(err);
201 }
```

Autrement dit :

- On déclare une adresse de descripteur de périphérique réseau, que l'on essaie d'instantier pour une carte Ethernet. Si on n'y parvient pas, on s'arrête en pointant l'opposé du code d'erreur ENOMEM.

Code Linux 2.6.10

La macro générale ERR_PTR() est définie dans le fichier `linux/include/linux/err.h`:

```
8  /*
9   * Les pointeurs noyau ont de l'information redondante, aussi pouvons-nous utiliser un
10  * schema ou nous pouvons renvoyer soit un code d'erreur, soit un pointeur
11  * dentry avec la meme valeur renvoyee.
12  *
13  * Ceci doit etre une chose pour l'architecture, de permettre des decisions
14  * pointeur et erreur differentes.
15  */
16 static inline void *ERR_PTR(long error)
17 {
18         return (void *) error;
19 }
```

- On définit le tableau des numéros de port d'entrée-sortie du microprocesseur relié à la carte pouvant être utilisés pour une carte 3Com 501.
- On déclare un numéro de port d'entrée-sortie et un code d'erreur.
- Si l'unité passée en argument est positive ou nulle (rappelons qu'elle était de -1 dans le cas du module et entre 0 et 7 dans le cas des huit cartes Ethernet de départ) :
  - On renseigne le champ nom du descripteur de périphérique.

  - On vérifie les paramètres de démarrage, grâce à la fonction `netdev_boot_setup_check()` étudiée ci-dessous.
  - On détermine l'adresse d'entrée-sortie de base, le numéro d'interruption et le début de mémoire vive à utiliser pour les tampons à partir du descripteur de périphérique.

- On se rend maître du module dans lequel se trouve le pilote.

  La macro `SET_MODULE_OWNER()` est définie dans le fichier `linux/include/linux/netdevice.h`: Code Linux 2.6.10

```
503 #define SET_MODULE_OWNER(dev) do { } while (0)
504 /* Positionne la reference du peripherique physique sysfs pour le peripherique logique
       de reseau.
505  * Positionner avant l'enregistrement causera un symlink pendant l'initialisation.
506  */
```

- Si l'adresse de base est strictement supérieure à `1FFh`, on essaie de détecter la présence physique de la carte au seul emplacement spécifié, grâce à la fonction `el1_probe1()` étudiée ci-après.
- Si l'adresse de base est comprise entre 1 et `1FFh`, ceci signifie qu'il ne faut pas sonder. On libère donc le descripteur de périphérique et on renvoie l'opposé du code d'erreur `ENXIO`.

  La fonction `free_netdev()` sera étudiée ci-dessous.
- Si l'adresse de base est nulle, on essaie de détecter la présence physique de la carte pour les valeurs du tableau `ports[]`. En cas de succès, on renvoie 0. Si cela échoue pour toutes ces valeurs, on libère le descripteur de périphérique et on renvoie l'opposé du code d'erreur `ENODEV`.
- On essaie d'enregistrer le périphérique. Si on n'y parvient pas, on libère les numéros de port d'entrée-sortie du microprocesseur occupés par cette carte, on libère le descripteur de périphérique et on renvoie le code d'erreur fourni par la fonction `register_netdev()`.

  La constante symbolique `EL1_IO_EXTENT`, spécifiant le nombre de ports d'entrée-sortie consécutifs nécessaire pour une carte 3Com 501, est définie dans le fichier `linux/drivers/net/3c501.h`: Code Linux 2.6.0

```
18 #define EL1_IO_EXTENT   16
```

- Sinon on renvoie l'adresse du descripteur de périphérique.

17.2.2 Établissement des paramètres au démarrage

La fonction `netdev_boot_setup_check()` est définie dans le fichier `linux/net/core/dev.c`: Code Linux 2.6.10

```
382 /**
383  *      netdev_boot_setup_check - verifie les parametres au moment du demarrage
384  *      @dev : le peripherique reseau
385  *
386  *      Verifie les parametres au moment du demarrage pour ce peripherique.
387  *      Les parametres trouves sont positionnes pour que le peripherique les utilise
388  *      plus tard lors de la detection du peripherique.
389  *      Renvoie 0 si aucun parametre trouve, 1 s'il y en a.
390  */
391 int netdev_boot_setup_check(struct net_device *dev)
392 {
393         struct netdev_boot_setup *s = dev_boot_setup;
394         int i;
395
396         for (i = 0; i < NETDEV_BOOT_SETUP_MAX; i++) {
397                 if (s[i].name[0] != '\0' && s[i].name[0] != ' ' &&
```

```
398                     !strncmp(dev->name, s[i].name, strlen(s[i].name))) {
399                         dev->irq        = s[i].map.irq;
400                         dev->base_addr  = s[i].map.base_addr;
401                         dev->mem_start  = s[i].map.mem_start;
402                         dev->mem_end    = s[i].map.mem_end;
403                         return 1;
404                 }
405         }
406         return 0;
407 }
```

Autrement dit :

- On déclare un tableau de structures de paramètres de démarrage, que l'on initialise avec `dev_boot_setup[]`.
- On cherche si le nom du périphérique réseau passé en argument apparaît dans le tableau des paramétrages. Si c'est le cas, on renseigne les champs du descripteur de périphérique passé en argument avec ces paramètres et on renvoie 1.
- Si ce nom n'apparaît pas, on renvoie 0.

17.2.3 Libération d'un descripteur de périphérique réseau

Code Linux 2.6.10

La fonction `free_netdev()` est définie dans le fichier `linux/net/core/dev.c`:

```
2960 /**
2961  *      free_netdev - libere un peripherique reseau
2962  *      @dev : peripherique
2963  *
2964  *      Cette fonction effectue la derniere etape de destruction d'une interface de
2965  *      peripherique allouee. La reference a l'objet peripherique est relachee.
2966  *      S'il s'agit de la derniere reference, il est libere.
2967  */
2968 void free_netdev(struct net_device *dev)
2969 {
2970 #ifdef CONFIG_SYSFS
2971         /*  Compatibilite avec la manipulation des erreurs dans le peripherique */
2972         if (dev->reg_state == NETREG_UNINITIALIZED) {
2973                 kfree((char *)dev - dev->padded);
2974                 return;
2975         }
2976
2977         BUG_ON(dev->reg_state != NETREG_UNREGISTERED);
2978         dev->reg_state = NETREG_RELEASED;
2979
2980         /* sera libere via le relachement de la classe */
2981         class_device_put(&dev->class_dev);
2982 #else
2983         kfree((char *)dev - dev->padded);
2984 #endif
2985 }
```

17.2.4 Vérification de la présence physique de la carte

Code Linux 2.6.10

La fonction `el1_probe1()` est définie dans le fichier `linux/drivers/net/3c501.c`:

```
203 /**
204  *      el1_probe1 :
205  *      @dev : la structure de peripherique a utiliser
206  *      @ioaddr : une adresse d'E/S a tester.
207  *
```

```
208  *      La detection elle-meme. Ceci est itere par #el1_probe afin de
209  *      verifier tous les emplacements de peripheriques applicables.
210  *
211  *      Renvoie 0 en cas de succes, en quel cas le peripherique est active,
212  *      EAGAIN si le IRQ est utilise par un autre peripherique et ENODEV si la
213  *      carte n'est pas detectee.
214  */
215
216 static int __init el1_probe1(struct net_device *dev, int ioaddr)
217 {
218         struct net_local *lp;
219         const char *mname;              /* Nom du vendeur */
220         unsigned char station_addr[6];
221         int autoirq = 0;
222         int i;
223
224         /*
225          *      Reserve les ressources d'E/S exclusivement pour ce peripherique
226          */
227
228         if (!request_region(ioaddr, EL1_IO_EXTENT, DRV_NAME))
229                 return -ENODEV;
230
231         /*
232          *      Lit les donnees PROM de l'adresse de la station a partir du port special.
233          */
234
235         for (i = 0; i < 6; i++)
236         {
237                 outw(i, ioaddr + EL1_DATAPTR);
238                 station_addr[i] = inb(ioaddr + EL1_SAPROM);
239         }
240         /*
241          *      Verifie les trois premiers octets de l'adresse de la station a la
242          *      recherche du prefixe 3Com ou du prefixe Sager NP943.
243          */
244
245         if (station_addr[0] == 0x02  &&  station_addr[1] == 0x60
246                 && station_addr[2] == 0x8c)
247         {
248                 mname = "3c501";
249         } else if (station_addr[0] == 0x00  &&  station_addr[1] == 0x80
250         && station_addr[2] == 0xC8)
251         {
252                 mname = "NP943";
253         }
254         else {
255                 release_region(ioaddr, EL1_IO_EXTENT);
256                 return -ENODEV;
257         }
258
259         /*
260          *      Nous auto-IRQ en fermant la ligne d'interruption et en la positionnant
261          *      a la valeur haute.
262          */
263
264         dev->irq = irq;
265
266         if (dev->irq < 2)
267         {
268                 unsigned long irq_mask;
269
```

```
270                 irq_mask = probe_irq_on();
271                 inb(RX_STATUS);         /* Efface les interruptions en souffrance. */
272                 inb(TX_STATUS);
273                 outb(AX_LOOP + 1, AX_CMD);
274
275                 outb(0x00, AX_CMD);
276
277                 mdelay(20);
278                 autoirq = probe_irq_off(irq_mask);
279
280                 if (autoirq == 0)
281                 {
282                         printk(KERN_WARNING "%s probe at %#x failed to detect IRQ line.\n",
283                                 mname, ioaddr);
284                         release_region(ioaddr, EL1_IO_EXTENT);
285                         return -EAGAIN;
286                 }
287         }
288
289         outb(AX_RESET+AX_LOOP, AX_CMD);                 /* Mode loopback. */
290         dev->base_addr = ioaddr;
291         memcpy(dev->dev_addr, station_addr, ETH_ALEN);
292
293         if (mem_start & 0xf)
294                 el_debug = mem_start & 0x7;
295         if (autoirq)
296                 dev->irq = autoirq;
297
298         printk(KERN_INFO "%s: %s EtherLink at %#lx, using %sIRQ %d.\n",
               dev->name, mname, dev->base_addr,
299                         autoirq ? "auto":"assigned ", dev->irq);
300
301 #ifdef CONFIG_IP_MULTICAST
302         printk(KERN_WARNING "WARNING: Use of the 3c501 in a multicast kernel is
               NOT recommended.\n");
303 #endif
304
305         if (el_debug)
306                 printk(KERN_DEBUG "%s", version);
307
308         memset(dev->priv, 0, sizeof(struct net_local));
309         lp = netdev_priv(dev);
310         spin_lock_init(&lp->lock);
311
312         /*
313          *      Les entrees specifiques a EL1 dans la structure de peripherique.
314          */
315
316         dev->open = &el_open;
317         dev->hard_start_xmit = &el_start_xmit;
318         dev->tx_timeout = &el_timeout;
319         dev->watchdog_timeo = HZ;
320         dev->stop = &el1_close;
321         dev->get_stats = &el1_get_stats;
322         dev->set_multicast_list = &set_multicast_list;
323         dev->ethtool_ops = &netdev_ethtool_ops;
324         return 0;
325 }
```

Autrement dit :

– On déclare une entité, du type `struct net_local`, pour les informations spécifiques à ce type de cartes.

Le type dépend du type de cartes. Il est défini dans le fichier `linux/drivers/net/3c501.h` pour la carte qui nous intéresse: Code Linux 2.6.10

```
26 /*
27  *      Informations specifiques a la carte dans dev->priv.
28  */
29
30 struct net_local
31 {
32         struct net_device_stats stats;
33         int             tx_pkt_start;   /* La longueur du paquet emis en cours. */
34         int             collisions;     /* Collisions en emission dans ce paquet */
35         int             loading;        /* Tampon d'eclaboussure qui charge les
                                              collisions */
36         int             txing;          /* Vrai si la carte est en mode d'emission */
37         spinlock_t      lock;           /* Verrou de serialisation */
38 };
```

- On déclare des variables pour le nom du vendeur et l'adresse MAC de la station.
- Si la plage d'adresses d'entrée-sortie passée en argument chevauche une plage utilisée par un autre pilote, on renvoie l'opposé du code d'erreur `ENODEV`.

  Les détails de la macro générale `request_region()` ne nous intéresserons pas dans cet ouvrage.

  La constante symbolique `DRV_NAME` est définie, pour la carte qui nous intéresse, dans le fichier `linux/drivers/net/3c501.h`: Code Linux 2.6.10

```
102 #define DRV_NAME        "3c501"
```

- On lit l'adresse MAC de la station à partir de la mémoire PROM de la carte.
- On vérifie les trois premiers caractères pour déterminer le préfixe du vendeur (deux possibilités) et initialiser le nom du vendeur de la carte. S'il ne s'agit pas de l'un de ces deux vendeurs, on libère la plage de ports d'entrée-sortie et on renvoie le code d'erreur `ENODEV`.
- On essaie d'initialiser l'interruption matérielle de la carte. Si on n'y parvient pas, on libère la plage de ports d'entrée-sortie et on renvoie l'opposé du code d'erreur `EAGAIN`.
- On renseigne les champ adresse d'entrée-sortie de base et adresse MAC de la station du descripteur de périphérique passé en argument (lignes 290 et 291).
- On change éventuellement la valeur de la variable `el_debug`, celle-ci étant définie dans le fichier `linux/drivers/net/3c501.h`: Code Linux 2.6.10

```
20 #ifndef EL_DEBUG
21 #define EL_DEBUG  0     /* utiliser 0 pour production, 1 pour developpement, >2 pour
                              debogage */
22 #endif                 /* Tout ce qui est dessus de 5 est mort ! */
23 #define debug el_debug
24 static int el_debug = EL_DEBUG;
```

- Si on est parvenu à déterminer le numéro d'interruption, on renseigne le champ `irq` du descripteur de périphérique passé en argument.
- On affiche un message noyau indiquant que la carte a été détectée.
- Si les options de compilation du noyau comprennent la multidiffusion, un message noyau est affiché, indiquant que ce n'est pas une bonne idée d'utiliser cette carte pour la multidiffusion.
- Si la valeur de débogage est non nulle, on affiche un message noyau spécifiant la version.

  La constante `version` est définie dans le fichier `linux/drivers/net/3c501.h`: Code Linux 2.6.10

```
103 #define DRV_VERSION     "2002/10/09"
```

```
104
105
106 static const char version[] =
107         DRV_NAME ".c: " DRV_VERSION " Alan Cox (alan@redhat.com).\n";
```

- On initialise la structure privée à zéro.
- On verrouille cette structure.
- On renseigne les champs du descripteur de périphérique spécifiques à la carte : les fonctions `el_open()` d'ouverture (en fait d'activation) du périphérique, `el_start_xmit()` de transmission des paquets, `el1_close()` de fermeture (désactivation) du périphérique, `el_timeout()` d'action à entreprendre lorsque le délai est écoulé, `el1_get_stats()` de récupération des informations statistiques et `set_multicast_list()` de mise en place de la diffusion restreinte ; le délai est initialisé à `HZ`.

  Ces fonctions spécifiques seront étudiées au fur et à mesure des besoins.
- On renseigne le champ de l'ensemble des opérations liées à Ethernet du descripteur de périphérique et on renvoie 0.

  L'ensemble d'opérations `netdev_ethtool_ops` propre à cette carte est défini dans le fichier `linux/drivers/net/3c501.c` :

Code Linux 2.6.10

```
885 static struct ethtool_ops netdev_ethtool_ops = {
886         .get_drvinfo            = netdev_get_drvinfo,
887         .get_msglevel           = netdev_get_msglevel,
888         .set_msglevel           = netdev_set_msglevel,
889 };
```

Les fonctions génériques `netdev_get_drvinfo()`, `netdev_get_msglevel()` et `netdev_set_msglevel()` devront être étudiées au fur et à mesure des besoins. En fait nous n'en aurons pas besoin dans cet ouvrage.

Chapitre 18

Attribution d'une adresse à une carte réseau

Après avoir détecter une carte réseau, la seconde étape consiste à lui attribuer une adresse. Il s'agit d'une adresse IP dans le cas d'Internet et pour presque tous les réseaux actuels.

Nous avons vu au chapitre 15 comment les pilotes de périphériques sont enregistrés. Une fois un périphérique réseau enregistré, avant que le noyau puisse gérer les trames (réception et émission), il doit l'activer et lui assigner une adresse. De même le noyau le désactivera lorsqu'il n'en a plus besoin, ce qui est important pour l'utilisateur dans le cas d'une liaison par téléphone. Comme pour les autres périphériques, on parle également sous Linux d'*ouverture* et de *fermeture*. Nous allons voir dans ce chapitre comment ceci est réalisé.

Le noyau n'a aucune raison d'ouvrir ou de fermer le périphérique par lui-même. C'est l'utilisateur qui décide du moment pour cela, heureusement d'ailleurs quand il s'agit d'une connexion téléphonique sur une ligne très onéreuse. Unix BSD a prévu une commande, appelée `ifconfig`, inclue dans les utilitaires `net-tools`, pour ce faire. Cette commande est souvent appelée par un script, aussi est-elle cachée à l'utilisateur lambda.

Nous allons commencer par dire ce qui est nécessaire de cette commande (dont le code ne fait pas parti du noyau Linux) pour l'ouverture et la fermeture d'un périphérique réseau. Celle-ci sert à bien d'autres propos.

18.1 La commande `ifconfig`

18.1.1 Fonctionnalité

La commande `ifconfig` (pour *InterFace CONFIGuration*) n'est pas seulement nécessaire pour l'activation et la désactivation des périphériques réseau, mais également pour leur configuration. Elle permet de définir les paramètres spécifiques aux protocoles, les adresses et les masques de sous-réseau ainsi que de modifier les paramètres d'interface de la carte réseau (paramètres matériels tels que ports d'entrée-sortie, interruptions, etc.). Elle peut également modifier les drapeaux (`ARP`, `PROMISC`, etc.) d'un périphérique réseau. Le périphérique doit être enregistré avant que l'on puisse utiliser cette commande à son égard.

Notre but n'est pas ici d'étudier cette commande dans son intégralité, mais uniquement la partie qui concerne l'activation et la désactivation d'un périphérique réseau. Pour une étude complète, on pourra se référer à la page `man` concernant `ifconfig`.

18.1.2 Liste des périphériques réseau en activité

À l'appel de cette commande sans paramètre, toutes les interfaces qui ont été précédemment ouvertes sont listées l'une après l'autre avec un certain nombre de renseignements :

```
linux:~ # ifconfig
eth0      Link encap:Ethernet  HWaddr 00:04:75:EA:1C:7B
          inet addr:192.168.5.10  Bcast:192.168.5.255  Mask:255.255.255.0
          inet6 addr: fe80::204:75ff:feea:1c7b/10 Scope:Link
          UP BROADCAST NOTRAILERS RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:528 errors:0 dropped:0 overruns:0 frame:0
          TX packets:495 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:100
          RX bytes:487357 (475.9 Kb)  TX bytes:127052 (124.0 Kb)
          Interrupt:5 Base address:0xb000

lo        Link encap:Local Loopback
          inet addr:127.0.0.1  Mask:255.0.0.0
          inet6 addr: ::1/128 Scope:Host
          UP LOOPBACK RUNNING  MTU:16436  Metric:1
          RX packets:30 errors:0 dropped:0 overruns:0 frame:0
          TX packets:30 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:0
          RX bytes:1956 (1.9 Kb)  TX bytes:1956 (1.9 Kb)

linux:~ #
```

Dans le cas que nous prenons en exemple, une carte Ethernet et le périphérique en boucle sont activés. Les paramètres en sont les suivants :

- Les noms sont `eth0` et `lo`, ce que l'on pouvait prévoir.
- Les types matériels sont `Ethernet` et `Local Loopback`.
- L'adresse MAC de la carte Ethernet est `00:04:75:ea:1c:7b`.
- L'adresse Internet qui a été affectée à la carte (par un script, en fait ici par DHCP) est `192.168.5.10`. Celle du périphérique en boucle est `127.0.0.1`, par tradition.
- L'adresse de diffusion générale de la carte Ethernet est `192.168.5.255`, ce que l'on peut déduire du masque.
- Le masque de sous-réseau est `255.255.255.0` pour la carte Ethernet et `255.0.0.0` pour le périphérique en boucle, ce qui correspond à des valeurs classiques.

- L'adresse IPv6 de la carte Ethernet est `fe80::204:75ff:feea:1c7b/10` et celle du périphérique en boucle `::1/128`.
- Les portées sont *Link* pour la carte Ethernet et *Host* pour le périphérique en boucle.
- On comprend la valeur des drapeaux : `UP` puisque les interfaces sont activées, `BROADCAST` lorsqu'il y a possibilité de diffusion générale, `MULTICAST` lorsqu'il y a possibilité de multidiffusion et ainsi de suite.
- L'unité maximum de transfert (MTU) est de 1 500 octets pour la carte Ethernet et 16 436 octets pour le périphérique en boucle, ce qui est classique.
- La **métrique** est de 1, c'est-à-dire que cette voie de communication n'est pas encombrée. Une métrique élevée indique que la voie de communication qui part de la carte réseau n'est pas très bonne.
- L'interface Eternet a reçu 528 paquets (RX), sans aucune erreur survenue.
- Elle a transmis 495 (TX), également sans aucune erreur survenue.
- Aucune collision n'est survenue pour la carte Ethernet. La longueur de sa file d'attente en émission est de 100 trames, la valeur par défaut comme nous l'avons vu.
- On peut consulter les nombres d'octets transmis et reçus.
- On a le numéro d'interruption et l'adresse de base d'entrée-sortie de la carte Ethernet.

18.1.3 Activation des périphériques du noyau

L'option `-a` permet d'activer les périphériques réseau qui n'ont pas encore été activés mais qui sont reconnus dans le noyau (pas dans les modules) :

```
linux:~ # ifconfig -a
```

18.1.4 Activation et désactivation d'un périphérique modularisé

L'activation d'un périphérique réseau s'effectue avec la commande :

`ifconfig` *name address* `up`

où *nom* est le nom de l'interface (par exemple `eth0`) et *address* l'adresse à lui attribuer.

Pour désactiver le périphérique réseau, on utilise la commande :

`ifconfig` *name* `down`

Nous n'allons pas étudier ici le lien avec les fonctions des deux sections suivantes : ceci est l'objet d'un programme utilisateur. Toujours est-il que les commandes d'activation et de désactivation finissent par faire appel à ces fonctions.

18.2 Aspect général de l'activation

Nous allons étudier la fonction d'ouverture d'un périphérique réseau, qui renvoie à une fonction spécifique à la carte réseau, puis la fonction spécifique dans le cas de la carte 3Com 501.

18.2.1 Fonction d'ouverture

L'activation d'un périphérique réseau s'effectue par l'intermédiaire de la fonction `dev_open()`. Elle est définie dans le fichier `linux/net/core/dev.c` : Code Linux 2.6.10

```
808 /**
809  *      dev_open        - prepare une interface pour son utilisation.
810  *      @dev :   peripherique a ouvrir
```

```
811  *
812  *      Passe un peripherique de l'etat desactive a l'etat active. La fonction
813  *      privee du peripherique est appelee et les listes de multidiffusion sont chargees.
814  *      Puis le peripherique est amene dans l'etat actif et un message %NETDEV_UP est
815  *      envoye a la chaine de notification netdev.
816  *
817  *      L'appel de cette fonction pour une interface activee est une nop. En cas d'echec
818  *      un code errno negatif est renvoye.
819  */
820 int dev_open(struct net_device *dev)
821 {
822         int ret = 0;
823
824         /*
825          *      Est-elle deja activee ?
826          */
827
828         if (dev->flags & IFF_UP)
829                 return 0;
830
831         /*
832          *      Est-elle deja presente ?
833          */
834         if (!netif_device_present(dev))
835                 return -ENODEV;
836
837         /*
838          *      Appelle la methode open privee du peripherique
839          */
840         set_bit(__LINK_STATE_START, &dev->state);
841         if (dev->open) {
842                 ret = dev->open(dev);
843                 if (ret)
844                         clear_bit(__LINK_STATE_START, &dev->state);
845         }
846
847         /*
848          *      Si l'ouverture est OK alors :
849          */
850
851         if (!ret) {
852                 /*
853                  *      Positionner les drapeaux.
854                  */
855                 dev->flags |= IFF_UP;
856
857                 /*
858                  *      Initialiser le statut de multidiffusion
859                  */
860                 dev_mc_upload(dev);
861
862                 /*
863                  *      Reveiller l'engin de file d'attente en transmission
864                  */
865                 dev_activate(dev);
866
867                 /*
868                  *      ... et annoncer la nouvelle interface.
869                  */
870                 notifier_call_chain(&netdev_chain, NETDEV_UP, dev);
871         }
872         return ret;
```

```
873 }
```

Autrement dit :

- Si le périphérique passé en argument est déjà ouvert, on renvoie 0.
- Si le périphérique n'est pas présent, on renvoie l'opposé du code d'erreur `ENODEV`.

  La fonction `netif_device_present()` est définie dans le fichier `linux/include/linux/netdevice.h`: Code Linux 2.6.10

```
72608 /* Chargement a chaud. */
727 static inline int netif_device_present(struct net_device *dev)
728 {
729         return test_bit(__LINK_STATE_PRESENT, &dev->state);
730 }
```

- On spécifie comme champ d'état du descripteur de périphérique que ce dernier est en train de démarrer.
- S'il existe une fonction spécifique d'ouverture correspondant à la carte réseau, on fait appel à elle. Si tout ne s'est pas bien déroulé, on met à zéro le bit `__LINK_STATE_START` du champ d'état du descripteur de périphérique.

  Nous étudierons ci-dessous le cas de la fonction spécifique d'ouverture pour la carte 3Com 501.
- Si tout s'est bien passé, on positionne le drapeau `IFF_UP`, on initialise le statut de multidiffusion, qui ne nous intéressera pas dans l'immédiat, on active la file d'attente en émission et l'ordonnanceur du périphérique (par appel à la fonction `dev_activate()` étudiée ci-dessous) et on annonce la nouvelle interface.

18.2.2 Activation de la file d'attente en émission

La fonction `dev_activate()` est définie dans le fichier `linux/net/sched/sch_generic.c`: Code Linux 2.6.10

```
515 void dev_activate(struct net_device *dev)
516 {
517         /* Aucune strategie de mise en file d'attente n'est attachee au peripherique ;
518            en creer une par defaut, a savoir pfifo_fast pour les peripheriques,
519            ce qui necessite une mise en file d'attente, et noqueue_qdisc pour les
520            interfaces virtuelles
521          */
522
523         if (dev->qdisc_sleeping == &noop_qdisc) {
524                 struct Qdisc *qdisc;
525                 if (dev->tx_queue_len) {
526                         qdisc = qdisc_create_dflt(dev, &pfifo_fast_ops);
527                         if (qdisc == NULL) {
528                                 printk(KERN_INFO "%s: activation failed\n", dev->name);
529                                 return;
530                         }
531                         write_lock_bh(&qdisc_tree_lock);
532                         list_add_tail(&qdisc->list, &dev->qdisc_list);
533                         write_unlock_bh(&qdisc_tree_lock);
534                 } else {
535                         qdisc =  &noqueue_qdisc;
536                 }
537                 write_lock_bh(&qdisc_tree_lock);
538                 dev->qdisc_sleeping = qdisc;
539                 write_unlock_bh(&qdisc_tree_lock);
540         }
541
```

```
542         spin_lock_bh(&dev->queue_lock);
543         rcu_assign_pointer(dev->qdisc, dev->qdisc_sleeping);
544         if (dev->qdisc != &noqueue_qdisc) {
545                 dev->trans_start = jiffies;
546                 dev_watchdog_up(dev);
547         }
548         spin_unlock_bh(&dev->queue_lock);
549 }
```

Autrement dit :

- Si le champ `qdisc_sleeping` du descripteur de périphérique passé en argument est `noop_qdisc` (stratégie de mise en file d'attente qui ne fait rien, placée lors de l'installation du pilote de périphérique comme nous l'avons vu) :
  - Si la longueur de la file d'attente en émission est non nulle : on essaie d'instantier une stratégie de file d'attente par défaut ; si on n'y parvient pas, on affiche un message noyau et on a terminé ; sinon on ajoute cette stratégie à la liste des stratégies du descripteur de périphérique.

    La fonction `qdisc_create_dflt()` et `pfifo_fast_ops` sont définis dans le fichier `linux/net/sched/sch_generic.c` :

Code Linux 2.6.10

```
399 static struct Qdisc_ops pfifo_fast_ops = {
400         .next           =       NULL,
401         .cl_ops         =       NULL,
402         .id             =       "pfifo_fast",
403         .priv_size      =       3 * sizeof(struct sk_buff_head),
404         .enqueue        =       pfifo_fast_enqueue,
405         .dequeue        =       pfifo_fast_dequeue,
406         .requeue        =       pfifo_fast_requeue,
407         .init           =       pfifo_fast_init,
408         .reset          =       pfifo_fast_reset,
409         .dump           =       pfifo_fast_dump,
410         .owner          =       THIS_MODULE,
411 };
412
413 struct Qdisc * qdisc_create_dflt(struct net_device *dev, struct Qdisc_ops *ops)
414 {
415         void *p;
416         struct Qdisc *sch;
417         int size;
418
419         /* S'assurer que le Qdisc et les donnees privees sont alignees sur 32
               octets */
420         size = ((sizeof(*sch) + QDISC_ALIGN_CONST) & ~QDISC_ALIGN_CONST);
421         size += ops->priv_size + QDISC_ALIGN_CONST;
422
423         p = kmalloc(size, GFP_KERNEL);
424         if (!p)
425                 return NULL;
426         memset(p, 0, size);
427
428         sch = (struct Qdisc *)(((unsigned long)p + QDISC_ALIGN_CONST)
429                                & ~QDISC_ALIGN_CONST);
430         sch->padded = (char *)sch - (char *)p;
431
432         INIT_LIST_HEAD(&sch->list);
433         skb_queue_head_init(&sch->q);
434         sch->ops = ops;
435         sch->enqueue = ops->enqueue;
436         sch->dequeue = ops->dequeue;
```

```
437         sch->dev = dev;
438         dev_hold(dev);
439         sch->stats_lock = &dev->queue_lock;
440         atomic_set(&sch->refcnt, 1);
441         if (!ops->init || ops->init(sch, NULL) == 0)
442                 return sch;
443
444         dev_put(dev);
445         kfree(p);
446         return NULL;
447 }
```

La constante QDISC_ALIGN_CONST est définie dans le fichier linux/include/net/pkt_sched.h: Code Linux 2.6.10

```
16 #define QDISC_ALIGN             32
17 #define QDISC_ALIGN_CONST       (QDISC_ALIGN - 1)
```

– On renseigne le champ de stratégie assoupie du descripteur de périphérique, soit avec noqueue_qdisc, soit avec celle qui vient d'être instantiée.

La stratégie noqueue_qdisc est définie dans le fichier linux/net/sched/sch_generic.c: Code Linux 2.6.10

```
286 struct Qdisc_ops noqueue_qdisc_ops = {
287         .next           =       NULL,
288         .cl_ops         =       NULL,
289         .id             =       "noqueue",
290         .priv_size      =       0,
291         .enqueue        =       noop_enqueue,
292         .dequeue        =       noop_dequeue,
293         .requeue        =       noop_requeue,
294         .owner          =       THIS_MODULE,
295 };
296
297 struct Qdisc noqueue_qdisc = {
298         .enqueue        =       NULL,
299         .dequeue        =       noop_dequeue,
300         .flags          =       TCQ_F_BUILTIN,
301         .ops            =       &noqueue_qdisc_ops,
302         .list           =       LIST_HEAD_INIT(noqueue_qdisc.list),
303 };
```

– On verrouille la file d'attente du descripteur de périphérique.
– On renseigne le champ qdisc de celui-ci avec la stratégie en attente de ce même descripteur.
– Si celle-ci est différente de noqueue_disc, on renseigne l'heure de la dernière émission avec l'heure en cours et on démarre le minuteur permettant l'identification de problèmes en émission.

La fonction dev_watchdog_up() sera étudiée ci-dessous.

– On déverrouille la file d'attente du descripteur de périphérique.

18.2.3 Démarrage du minuteur de problèmes d'émission

La méthode dev_watchdog_up() démarre le minuteur permettant l'identification de problèmes d'émission. Elle est définie dans le fichier linux/net/sched/sch_generic.c: Code Linux 2.6.10

```
215 void __netdev_watchdog_up(struct net_device *dev)
216 {
217         if (dev->tx_timeout) {
218                 if (dev->watchdog_timeo <= 0)
```

```
219                     dev->watchdog_timeo = 5*HZ;
220             if (!mod_timer(&dev->watchdog_timer, jiffies + dev->watchdog_timeo))
221                     dev_hold(dev);
222         }
223 }
224
225 static void dev_watchdog_up(struct net_device *dev)
226 {
227         spin_lock_bh(&dev->xmit_lock);
228         __netdev_watchdog_up(dev);
229         spin_unlock_bh(&dev->xmit_lock);
230 }
```

18.3 Cas de la carte 3Com 501

18.3.1 Installation du gestionnaire d'interruption

Code Linux 2.6.10

Nous avons vu au chapitre 17 que la fonction d'ouverture spécifique à la carte réseau Ethernet 3Com 501 s'appelle `el_open()`. Elle est définie dans le fichier `linux/drivers/net/3c501.c`:

```
327 /**
328  *      el1_open :
329  *      @dev : peripherique qui est en train d'etre ouvert
330  *
331  *      Lorsqu'un ifconfig change les drapeaux du peripherique pour inclure
332  *      IFF_UP, cette fonction est appelee. Elle est seulement appelee lorsque ce changement
333  *      apparait, pas lorsque l'interface reste activee. #el1_close sera appelee
334  *      lorsqu'il est desactive.
335  *
336  *      Renvoie 0 en cas de succes d'ouverture, ou -EAGAIN si quelqu'un a tire
337  *      notre ligne d'interruption.
338  */
339
340 static int el_open(struct net_device *dev)
341 {
342         int retval;
343         int ioaddr = dev->base_addr;
344         struct net_local *lp = netdev_priv(dev);
345         unsigned long flags;
346
347         if (el_debug > 2)
348                 printk(KERN_DEBUG "%s: Doing el_open()...", dev->name);
349
350         if ((retval = request_irq(dev->irq, &el_interrupt, 0, dev->name, dev)))
351                 return retval;
352
353         spin_lock_irqsave(&lp->lock, flags);
354         el_reset(dev);
355         spin_unlock_irqrestore(&lp->lock, flags);
356
357         lp->txing = 0;          /* Carte en mode RX */
358         outb(AX_RX, AX_CMD);    /* Controle Aux : irq et reception actives */
359         netif_start_queue(dev);
360         return 0;
361 }
```

Autrement dit :

- On déclare une valeur de retour.
- On déclare un numéro de port d'entrée-sortie, que l'on initialise avec celui fourni par le descripteur de périphérique passé en argument.

- On déclare une structure locale de réseau, que l'on initialise avec la partie privée du descripteur de périphérique passé en argument.
- On déclare un vecteur de drapeaux.
- Si le niveau de débogage est strictement supérieur à 2, on affiche un message noyau indiquant qu'on est en train d'ouvrir le périphérique.
- On essaie de mettre en place le gestionnaire d'interruption `el_interrupt()` correspondant au numéro d'interruption de la carte (celui-ci étant fourni par le descripteur de périphérique passé en argument). Si une erreur se produit, on renvoie le code d'erreur fourni par la fonction `request_irq()`.

  Le gestionnaire d'interruption sera étudié au chapitre 19.
- On sauve les valeurs des registres dans le vecteur des drapeaux.
- On (ré-)initialise le périphérique, grâce à la fonction `el_reset()` étudiée ci-après.
- On restaure les valeurs des registres.
- On se place en mode de réception et on renvoie 0.

  La fonction `netif_start_queue()` est définie dans le fichier `linux/include/linux/-netdevice.h`:

Code Linux 2.6.10

```
606 static inline void netif_start_queue(struct net_device *dev)
607 {
608         clear_bit(__LINK_STATE_XOFF, &dev->state);
609 }
```

18.3.2 Réinitialisation

La fonction `el_reset()` de réinitialisation spécifique à la carte 3Com 501 est définie dans le fichier `linux/drivers/net/3c501.c`:

Code Linux 2.6.10

```
756 /**
757  * el_reset : Reinitialise une carte 3c501
758  * @dev : La carte 3c501 a traiter
759  *
760  * Meme la reinitialisation d'un 3c501 n'est pas simple. Lorsque vous activez la
761  * reinitialisation, elle perd tout de sa configuration. Vous devez detenir le verrou
       lorsque vous faites ca. La fonction
762  * ne peut pas tirer le verrou elle-meme car elle est appelable depuis la routine d'irq.
763  */
764
765 static void  el_reset(struct net_device *dev)
766 {
767         struct net_local *lp = netdev_priv(dev);
768         int ioaddr = dev->base_addr;
769
770         if (el_debug> 2)
771                 printk(KERN_INFO "3c501 reset...");
772         outb(AX_RESET, AX_CMD);         /* Reinitialise la puce */
773         outb(AX_LOOP, AX_CMD);          /* Controle aux : irq et loopback actives */
774         {
775                 int i;
776                 for (i = 0; i < 6; i++) /* Positionne l'adresse de la station. */
777                         outb(dev->dev_addr[i], ioaddr + i);
778         }
779
780         outw(0, RX_BUF_CLR);            /* Initialise la zone de rx des paquets a 0. */
781         outb(TX_NORM, TX_CMD);          /* Active l'irq en tx, collision */
782         outb(RX_NORM, RX_CMD);          /* Initialise les commandes de Rx. */
783         inb(RX_STATUS);                 /* Efface le statut. */
784         inb(TX_STATUS);
```

```
785         lp->txing = 0;
786 }
```

Autrement dit :

- on déclare une structure locale de réseau, que l'on initialise avec la partie privée du descripteur de périphérique passé en argument ;
- on déclare un port d'entrée-sortie, que l'on initialise avec celui fourni par le descripteur de périphérique passé en argument ;
- si le niveau de débogage est strictement supérieur à 2, on affiche un message noyau ;
- on passe l'ordre de réinitialisation à la carte du périphérique, spécifique à la carte 3Com 501 ;
- on active les fonctions de la carte (IRQ et périphérique en boucle) ;
- on positionne l'adresse de la station sur la carte ;
- on initialise le pointeur du tampon de réception à zéro ;
- on indique qu'il n'y a pas de traitement en transmission à effectuer ;
- on envoie la commande de réception ;
- on met à zéro le statut ;
- on se place en mode de réception.

Quatrième partie

Réception

Chapitre 19

Réception des trames

Nous allons étudier dans ce chapitre comment les trames sont reçues en prenant comme exemple notre carte Ethernet 3Com 501.

Une fois que l'interface est activée, lors de la réception d'une trame par la carte réseau, celle-ci lève une interruption pour indiquer qu'une trame est arrivée. Le gestionnaire d'interruption la vérifie, la rejette ou la place en mémoire vive en utilisant l'accès direct à la mémoire puis elle passe les données et la main à la couche supérieure (la couche réseau en général).

19.1 Action du gestionnaire d'interruption

L'itinéraire de toute trame qui n'a pas été générée en local dans l'ordinateur commence dans un adaptateur réseau (ou une interface comparable, comme le port parallèle). Cet adaptateur reçoit la trame dans sa mémoire interne propre et informe le noyau de son arrivée en déclenchant une interruption.

Prenons comme exemple celui de la carte 3c501 de 3Com.

19.1.1 Démultiplexage : réception ou émission

Le même numéro d'interruption est utilisé pour la réception et l'émission. La première action du gestionnaire de cette interruption est donc de déterminer la cause de l'interruption. Nous avons vu que le gestionnaire d'interruption de la carte 3Com 501 s'appelle `el_interrupt()`. Il est défini dans le fichier `linux/drivers/net/3c501.c`:

Code Linux 2.6.10

```
496 /**
497  * el_interrupt :
498  * @irq : Numero d'interruption
499  * @dev_id : La 3c501 qui a fait un renvoi
500  * @regs : Registre des donnees (surplus a nos exigences)
501  *
502  * Gestionnaire des interruptions de l'interface ether. La 3c501 a besoin de beaucoup plus
503  * que la plupart des cartes. En particulier nous obtenons une interruption en emission par
504  * erreur due a une collision puisque le micrologiciel de la carte n'est pas capable de
505  * rembobiner ses propres pointeurs de tampon. Il peut cependant compter jusqu'a 16 pour nous.
506  *
507  * Du cote de la reception, la carte est egalement tres bete. Elle n'a pas de tampon pour
508  * indiquer. Nous tirons tout simplement le paquet hors de son tampon PIO (qui est lent)
509  * et le mettons dans une file d'attente a l'attention du noyau. Nous reinitialison alors
       la carte pour le paquet suivant.
510  *
511  * Nous obtenons quelquefois des interruptions suprises tardives, a la fois parce que la
512  * livraison des IRQ SMP est un passage de message et parce que la carte semble quelquefois
513  * les livrer en retard. Je pense que ceci est du en partie a la facon de receptionner et au
514  * fait que le mode est change lorsqu'il sort d'une reception et nous envoie une interruption.
515  * Nous avons a regrouper tous ces cas pour obtenir une performamce d'a peu pres 150
516  * koctets/seconde. Meme alors vous devez avoir une petite fenetre TCP.
517  */
518
519 static irqreturn_t el_interrupt(int irq, void *dev_id, struct pt_regs *regs)
520 {
521         struct net_device *dev = dev_id;
522         struct net_local *lp;
523         int ioaddr;
524         int axsr;                       /* Registre de statut aux. */
525
526         ioaddr = dev->base_addr;
527         lp = netdev_priv(dev);
528
529         spin_lock(&lp->lock);
530
531         /*
532          *      Qu'est ce qui se passe ?
533          */
534
535         axsr = inb(AX_STATUS);
536
537         /*
538          *      Inscrivons-le !
539          */
```

```
540
541         if (el_debug > 3)
542                 printk(KERN_DEBUG "%s: el_interrupt() aux=%#02x", dev->name, axsr);
543
544         if(lp->loading==1 && !lp->txing)
545                 printk(KERN_WARNING "%s: Inconsistent state loading while not in tx\n",
546                         dev->name);
547
548         if (lp->txing)
549         {
550
551                 /*
552                  *      La carte est en mode emission. Peut etre en train de charger.
[...]
633         }
634         else
635         {
636                 /*
637                  *      En mode reception.
638                  */
[...]
673         }
674
675         /*
676          *      Passer dans l'etat reception
677          */
678
679         outb(AX_RX, AX_CMD);
680         outw(0x00, RX_BUF_CLR);
681         inb(RX_STATUS);         /* Etre certain que les interruptions sont mises a zero. */
682         inb(TX_STATUS);
683         spin_unlock(&lp->lock);
684 out:
685         return IRQ_HANDLED;
686 }
```

Le type général `pt_regs` spécifie la façon dont les registres sont stockés dansla pile lors d'un appel système. Il dépend de l'architecture. Il est défini dans le fichier `linux/include/asm-i386/ptrace.h` dans le cas des microprocesseurs Intel :

Code Linux 2.6.10

```
23 /* Cette struct definit la facon dont les registres sont stockes sur la
24    pile lors d'un appel systeme. */
25
26 struct pt_regs {
27         long ebx;
28         long ecx;
29         long edx;
30         long esi;
31         long edi;
32         long ebp;
33         long eax;
34         int  xds;
35         int  xes;
36         long orig_eax;
37         long eip;
38         int  xcs;
39         long eflags;
40         long esp;
41         int  xss;
42 };
```

Venons-en au commentaire du code :

– On déclare une adresse de descripteur de périphérique réseau, que l'on initialise avec celle

passée en argument.
- On déclare une adresse de structure réseau locale, un port d'entrée-sortie et l'adresse du registre auxiliaire de statut.
- On initialise le port d'entrée-sortie avec celui fourni par le descripteur de périphérique passé en argument.
- On initialise l'adresse de structure de réseau locale avec celle de la partie privée du descripteur de périphérique passé en argument.
- On verrouille la structure de réseau locale.
- On récupère le contenu du registre auxiliaire de statut pour connaître la raison de l'interruption matérielle (début de réception ou fin d'émission?).
- Si le niveau de débogage est strictement supérieur à 3, on affiche un message système précisant le contenu de ce registre.
- Si la structure de réseau locale précise qu'on est en train de recevoir des données sans être dans l'état émission, on affiche un message noyau à propos de cette incohérence.
- On traite le cas de l'émission ou de la réception suivant le cas, de la façon détaillée plus loin.
- On passe en mode réception dans les deux cas.
- On déverrouille la structure locale de réseau et on renvoie `IRQ_HANDLED`.

Code Linux 2.6.10

Les constantes symboliques générales `IRQ_HANDLED` et `IRQ_NONE`, la macro `IRQ_RETVAL()` et le type `irqreturn_t` sont définis dans le fichier `linux/include/linux/interrupt.h`:

```
16 /*
17  * Pour une compatibilite avec 2.4.x, 2.4.x peut utiliser
18  *
19  *      typedef void irqreturn_t;
20  *      #define IRQ_NONE
21  *      #define IRQ_HANDLED
22  *      #define IRQ_RETVAL(x)
23  *
24  * Pour melanger l'ancien style et le nouveau style, la routine d'irq renvoie quelque
      chose.
25  *
26  * IRQ_NONE signifie que nous ne l'avons pas traitee.
27  * IRQ_HANDLED signifie que nous avions une interruption valide et qu'elle a ete traitee.
28  * IRQ_RETVAL(x) selectionne une des deux suivant que x est non nul (pour traite)
29  */
30 typedef int irqreturn_t;
31
32 #define IRQ_NONE        (0)
33 #define IRQ_HANDLED     (1)
34 #define IRQ_RETVAL(x)   ((x) != 0)
```

19.1.2 Traitement en cas de réception

Code Linux 2.6.10

Reprenons le code pour nous intéresser à la partie concernant la réception:

```
519 static irqreturn_t el_interrupt(int irq, void *dev_id, struct pt_regs *regs)
[...]
635         {
636                 /*
637                  *      En mode reception.
638                  */
639
640                 int rxsr = inb(RX_STATUS);
641                 if (el_debug > 5)
642                         printk(KERN_DEBUG " rxsr=%02x txsr=%02x rp=%04x", rxsr,
```

```
                                inb(TX_STATUS),inw(RX_LOW));
643                 /*
644                  *      Juste lire rx_status cause beaucoup d'erreurs.
645                  */
646                 if (rxsr & RX_MISSED)
647                         lp->stats.rx_missed_errors++;
648                 else if (rxsr & RX_RUNT)
649                 {       /* Manipule pour eviter le verrouillage de la carte. */
650                         lp->stats.rx_length_errors++;
651                         if (el_debug > 5)
652                                 printk(KERN_DEBUG " runt.\n");
653                 }
654                 else if (rxsr & RX_GOOD)
655                 {
656                         /*
657                          *      La reception a marche.
658                          */
659                         el_receive(dev);
660                 }
661                 else
662                 {
663                         /*
664                          *      Rien ? Quelque chose a disparu !
665                          */
666                         if (el_debug > 2)
667                                 printk(KERN_DEBUG "%s: No packet seen,
                                               rxsr=%02x **resetting 3c501***\n",
668                                         dev->name, rxsr);
669                         el_reset(dev);
670                 }
671                 if (el_debug > 3)
672                         printk(KERN_DEBUG ".\n");
673         }
[...]
686 }
```

Autrement dit :

- Le gestionnaire d'interruption de cette carte récupère le contenu du registre de statut en réception.
- Si le niveau de débogage est supérieur à 6, on affiche un message noyau.
- Suivant le contenu du registre de statut en réception :
  - S'il indique qu'on a manqué quelque chose, on met à jour les statistiques à ce propos dans la structure locale de réseau.
  - S'il indique qu'il y a une erreur sur la longueur du paquet reçu, on met à jour les statistiques à ce propos dans la structure locale de réseau. Si le niveau de débogage est strictement supérieur à 5, on affiche un message noyau.
  - Si tout s'est bien déroulé, on fait appel à la fonction `el_receive()`, étudiée ci-après, de contrôle de la trame et de création d'un nouveau tampon de socket.
  - Sinon, quelque chose s'est mal déroulé : si le niveau de débogage est strictement supérieur à 2, on affiche un message noyau ; on réinitialise le périphérique.
- Si le niveau de débogage est supérieur à 3, on passe à la ligne dans les messages noyau.

19.2 Création d'un nouveau tampon de socket

Dans le cas d'une réception réussie, on doit vérifier la trame, allouer un nouveau tampon de socket, placer les données de la trame dans celui-ci et placer son descripteur dans une file d'attente.

Les trois premières actions sont l'objet de la fonction `el_receive()` dans le cas de la carte 3c501 de 3Com, qui est appelée par le gestionnaire d'interruption comme nous venons de le voir. Elle fait appel à une nouvelle fonction pour continuer le traitement. La fonction `el_receive()` est définie dans le fichier `linux/drivers/net/3c501.c`:

Code Linux 2.6.10

```
689 /**
690  * el_receive :
691  * @dev : [Descripteur de] peripherique depuis lequel extraire les paquets entrants
692  *
693  * Nous avons un bon paquet. En fait, pas exactement "bon", juste non perdu.
694  * Nous devons verifier chaque chose pour voir s'il est bon. En particulier nous
695  * obtenons un paquet brut convenant a la carte. Si le paquet semble sain, nous le retirons
696  * de la carte et le placons dans une file d'attente pour les couches de protocole.
697  */
698
699 static void el_receive(struct net_device *dev)
700 {
701         struct net_local *lp = netdev_priv(dev);
702         int ioaddr = dev->base_addr;
703         int pkt_len;
704         struct sk_buff *skb;
705
706         pkt_len = inw(RX_LOW);
707
708         if (el_debug > 4)
709                 printk(KERN_DEBUG " el_receive %d.\n", pkt_len);
710
711         if ((pkt_len < 60)  ||  (pkt_len > 1536))
712         {
713                 if (el_debug)
714                         printk(KERN_DEBUG "%s: bogus packet, length=%d\n",
                                dev->name, pkt_len);
715                 lp->stats.rx_over_errors++;
716                 return;
717         }
718
719         /*
720          *      Mode commande, ainsi nous pouvons vider le tampon
721          */
722
723         outb(AX_SYS, AX_CMD);
724         skb = dev_alloc_skb(pkt_len+2);
725
726         /*
727          *      Debut de la trame
728          */
729
730         outw(0x00, GP_LOW);
731         if (skb == NULL)
732         {
733                 printk(KERN_INFO "%s: Memory squeeze, dropping packet.\n",
                        dev->name);
734                 lp->stats.rx_dropped++;
735                 return;
736         }
737         else
```

```
738         {
739                 skb_reserve(skb,2);     /* Force l'alignement 16 octets */
740                 skb->dev = dev;
741                 /*
742                  *      La lecture incremente [l'adresse des] octets. Le gestionnaire
743                  *      d'interruption rectifiera le pointeur lorsqu'il reviendra en
744                  *      mode reception.
745                  */
746                 insb(DATAPORT, skb_put(skb,pkt_len), pkt_len);
747                 skb->protocol=eth_type_trans(skb,dev);
748                 netif_rx(skb);
749                 dev->last_rx = jiffies;
750                 lp->stats.rx_packets++;
751                 lp->stats.rx_bytes+=pkt_len;
752         }
753         return;
754 }
```

Autrement dit :

- On déclare une adresse de structure locale de réseau, que l'on initialise avec celle de la partie privée du descripteur de périphérique réseau passé en argument.
- On déclare un port d'entrée-sortie, que l'on initialise avec celui associé au descripteur de périphérique réseau passé en argument.
- On déclare une longueur de paquet et un descripteur de tampon de socket.
- On renseigne la longueur du paquet reçu à partir de la valeur du pointeur de réception de la carte 3Com 501.
- Si le niveau de débogage est strictement supérieur à 4, on affiche celle-ci comme message noyau.
- Si cette longueur est strictement inférieure à 60 ou strictement supérieure à 1 536, il ne s'agit pas d'une longueur de trame Ethernet permise : si le niveau de débogage est non nul, on affiche un message noyau à ce propos ; on met à jour les informations statistiques sur ce type d'erreur dans la structure locale de réseau et on a terminé.

  Remarquons qu'il aurait mieux vallu utiliser une constante plutôt qu'un nombre magique : 1 536, c'est 1 514 plus deux adresses MAC plus le CRC.
- On se place en mode commande.
- On essaie d'instantier un nouveau descripteur de tampon de socket, dans le champ données duquel on placera le paquet. Si on n'y parvient pas, on affiche un message noyau, on met à jour les informations statistiques sur ce type d'erreur dans la structure locale de réseau et on a terminé.
- On réinitialise le pointeur de tampon de la carte à zéro.
- On initialise le tampon de socket : on aligne sur un multiple de 16 octets, on détermine l'adresse de l'emplacement du champ des données que l'on communique au contrôleur de la carte, on récupère les données de la trame reçue que l'on place dans le champ données du tampon.
- On renseigne les champs du descripteur de tampon de socket : le descripteur de périphérique associé est celui passé en argument ; on positionne le type de protocole de la couche réseau, déterminé grâce à la fonction `eth_type_trans()` étudiée ci-dessous.
- On place le descripteur du nouveau tampon dans la file d'attente en réception du microprocesseur en cours, grâce à la fonction `netif_rx()` étudiée ci-dessous.
- On met à jour l'heure de la dernière réception de paquet par le périphérique ainsi que les informations statistiques dans la structure locale de réseau du descripteur de périphérique et on a terminé.

19.3 Détermination du protocole de couche réseau

La fonction `short eth_type_trans(skb, dev)` est l'un des éléments les plus importants de l'implémentation Linux de la sous-couche LLC. Deux tâches essentielles sont exécutées:

- on détermine le protocole de la couche réseau au moyen des paramètres placés dans la trame et on renvoie cette valeur;
- on détermine le type de paquet (individuel, diffusion restreinte, diffusion générale) et on vérifie si le paquet est destiné à l'ordinateur présent.

La fonction `eth_type_trans()`, utilisable pour toutes les cartes réseau compatibles Ethernet, est en général appelée par le pilote de réseau dans la méthode de réception des paquets, ainsi que lors de l'émission dans le cas du périphérique en boucle. Tous les périphériques réseau d'un type de protocole MAC utilisent la même méthode `type_trans()`. Les périphériques *Token Ring* et FDDI disposent de méthodes correspondantes (`tr_type_trans()` et `fddi_type_trans()` respectivement).

Les réseaux Ethernet n'utilisent aucune des normes LLC. Elles transmettent directement l'identification de protocole de la couche réseau trouvée dans la trame MAC. Le seul mécanisme de protocole est le démultiplexage des différents protocoles de couche réseau. Par conséquent `eth_type_trans()` est assez simple et claire.

Code Linux 2.6.10

La fonction `eth_type_trans()` est définie dans le fichier `linux/net/ethernet/eth.c`:

```
153 /*
154  *      Determine l'identificateur de protocole du paquet. La regle est de considerer ici
155  *      qu'il s'agit de 802.3 si le champ type est suffisamment court pour etre
156  *      une taille. C'est la pratique normale et cela marche pour les protocoles
         'actuellement en usage'.
157  */
158
159 unsigned short eth_type_trans(struct sk_buff *skb, struct net_device *dev)
160 {
161         struct ethhdr *eth;
162         unsigned char *rawp;
163
164         skb->mac.raw=skb->data;
165         skb_pull(skb,ETH_HLEN);
166         eth = eth_hdr(skb);
167         skb->input_dev = dev;
168
169         if(*eth->h_dest&1)
170         {
171                 if(memcmp(eth->h_dest,dev->broadcast, ETH_ALEN)==0)
172                         skb->pkt_type=PACKET_BROADCAST;
173                 else
174                         skb->pkt_type=PACKET_MULTICAST;
175         }
176
177         /*
178          *      Cette verification ALLMULTI devrait etre redondante d'apres 1.4
179          *      aussi n'oubliez pas de l'enlever.
180          *
181          *      Il semblerait que vous ayez oublie de l'enlever. Tous les
182          *      peripheriques stupides semblent positionner IFF_PROMISC.
183          */
184
185         else if(1 /*dev->flags&IFF_PROMISC*/)
186         {
187                 if(memcmp(eth->h_dest,dev->dev_addr, ETH_ALEN))
```

```
188                         skb->pkt_type=PACKET_OTHERHOST;
189         }
190
191         if (ntohs(eth->h_proto) >= 1536)
192                 return eth->h_proto;
193
194         rawp = skb->data;
195
196         /*
197          *      Ceci est un truc magique pour reperer les paquets IPX. Les anciens Novell ne
199          *      respectent pas le protocole et realisent IPX sur 802.3 sans couche LLC 802.2.
199          *      Nous recherchons FFFF qui n'est pas un SSAP/DSAP 802.2 utilise. Ceci ne
200          *      marchera pas pour les reseaux tolerant les fautes mais marchera pour les
                    autres.
201          */
202         if (*(unsigned short *)rawp == 0xFFFF)
203                 return htons(ETH_P_802_3);
204
205         /*
206          *      Vrai LLC 802.2
207          */
208         return htons(ETH_P_802_2);
209 }
```

Autrement dit :

- On déclare une adresse d'en-tête Ethernet et une chaîne de caractères.
- On positionne le début des données brutes du tampon de socket au début des données du tampon de socket passé en argument.
- On tronque le début du tampon de la taille d'un en-tête Ethernet.
- On initialise l'adresse d'en-tête Ethernet avec celle du champ en-tête Ethernet du tampon de socket.

  La fonction en ligne `eth_hdr()` est définie dans le fichier `linux/include/linux/if_ether.h` :

Code Linux 2.6.10

```
108 static inline struct ethhdr *eth_hdr(const struct sk_buff *skb)
109 {
110         return (struct ethhdr *)skb->mac.raw;
111 }
```

- On renseigne le champ périphérique d'entrée du descripteur de tampon avec le descripteur de périphérique passé en argument.
- Si le type du paquet est diffusion générale, multidiffusion ou mode promiscuité, ceci est consigné dans le champ adéquat du descripteur de tampon passé en argument.
- Le protocole réseau du paquet admis est ensuite déterminé :
  - Lorsqu'il existe dans le champ longueur et/ou protocole de l'en-tête de la trame Ethernet une valeur supérieure à la longueur de la trame maximale (1 536 octets), on part du principe qu'il s'agit d'une carte Ethernet compatible avec la norme 802.3. Nous avons vu que le protocole 802.3 intègre le type de protocole de la couche réseau dans le champ protocole de la trame 802.3. Par conséquent, la valeur du champ est tout simplement renvoyée.
  - Les cartes plus anciennes ne stockent pas l'identification du protocole de couche réseau dans le champ longueur et/ou protocole, mais la longueur de la trame. Linux renvoie tout simplement `ETH_P_802_2` à titre d'identification de protocole de couche réseau car 802.2 est plutôt un cas exceptionnel. Le processus de démultiplexage proprement dit aura lieu dans la routine de traitement du protocole 802.2, à savoir `llc_rcv()`.

19.4 Mise du nouveau tampon en file d'attente

Les paquets reçus par l'ordinateur sont placés dans une file d'attente, une par microprocesseur dans le cas d'un système multiprocesseurs. Commençons par étudier celles-ci.

19.4.1 File d'attente d'un microprocesseur

19.4.1.1 Structure de données

Code Linux 2.6.10

Ces files d'attente sont des entités du type `struct softnet_data`, défini dans le fichier `linux/include/linux/netdevice.h`:

```
563 /*
564  * Les paquets recus sont places dans des files d'attente, une par microprocesseur,
565  * donc aucun verrouillage n'est necessaire.
566  */
567
568 struct softnet_data
569 {
570         int                     throttle;
571         int                     cng_level;
572         int                     avg_blog;
573         struct sk_buff_head     input_pkt_queue;
574         struct list_head        poll_list;
575         struct net_device       *output_queue;
576         struct sk_buff          *completion_queue;
577
578         struct net_device       backlog_dev;    /* Desole. 8) */
579 };
```

dont les champs ont la signification suivante:

- Une valeur non nulle pour `throttle` signifie que la file d'attente est pleine. On ne peut plus y placer de nouvel élément.
- Le champ `cng_level` spécifie le niveau de congestion de la file d'attente. Cette information est utile avant d'en arriver au cas extrême représenté par le champ précédent.

  Les valeurs possibles sont représentées par l'une des trois constantes symboliques parmi les suivantes:

  - `NET_RX_SUCCESS` en cas de réception réussie;
  - `NET_RX_DROP` si le paquet a été détruit ('écarté' est certes plus beau, mais c'est la même chose) pour une raison quelconque, par exemple s'il n'est pas destiné à l'ordinateur;
  - `NET_RX_CN_LOW` (avec `CN` pour *CoNgested*) lorsque le canal commence à être congestionné (mais on a quand même reçu le paquet);
  - `NET_RX_CN_MOD` (avec `MOD` pour *MODerate*) lorsque le canal commence vraiment à être congestionné (on a eu du mal à récupérer le paquet);
  - `NET_RX_CN_HIGH` lorsque le canal est vraiment congestionné;
  - `NET_RX_BAD` lorsque le paquet a été détruit à cause d'une erreur concernant le noyau.

  Comme on le voit, cette série de constantes symboliques sert également à d'autres propos.

  Code Linux 2.6.10

  Ces constantes symboliques spécifiant l'état de la réception sont définies dans le fichier `linux/include/linux/netdevice.h`:

```
62 /* Niveaux de congestion d'accumulation */
63 #define NET_RX_SUCCESS          0   /* Bien arrive, cherie */
64 #define NET_RX_DROP             1   /* paquet ecarte */
```

```
65 #define NET_RX_CN_LOW          2   /* alerte de tempete, on ne sait jamais */
66 #define NET_RX_CN_MOD          3   /* La tempete arrive */
67 #define NET_RX_CN_HIGH         4   /* La tempete est la */
68 #define NET_RX_BAD             5   /* paquet ecarte du a une erreur du noyau */
```

- Le champ `input_pkt_queue` est la file d'attente de descripteurs de tampon de socket proprement dite. Il s'agit donc du champ le plus important de cette structure. Les autres champs correspondent à un descripteur de cette file d'attente.
- Le champ `poll_list` est une liste d'élection.
- Le champ `output_queue` est un pointeur sur le périphérique d'où provient l'élément de la file d'attente en train d'être traité.
- Le champ `completion_queue` est un pointeur sur le descripteur du tampon de socket.
- Le périphérique (virtuel) `backlog_dev` permet de passer la main à la couche supérieure, c'est-à-dire à la couche réseau, pour continuer le traitement du paquet, grâce à sa fonction `poll()`, comme nous le verrons.

19.4.1.2 Déclaration

Une file d'attente de ce type est créée pour chaque microprocesseur, comme le montre l'extrait suivant du fichier `linux/include/linux/netdevice.h`: Code Linux 2.6.10

```
581 DECLARE_PER_CPU(struct softnet_data,softnet_data);
```

La taille maximale d'une telle file d'attente est limitée à 300. Cette constante est représentée par la constante symbolique (en fait la variable) `netdev_max_backlog` définie dans le fichier `linux/net/core/dev.c`: Code Linux 2.6.10

```
1340 int netdev_max_backlog = 300;
```

Son nom provient de ce qu'elle désignait le nombre de clients pouvant se connecter simultanément à un serveur (*backlog* en anglais) dans les versions antérieures du noyau.

19.4.1.3 Initialisation des files d'attente

Ces files d'attente sont initialisées lors du démarrage du système de la façon indiquée dans la fonction `net_dev_init()`, définie dans le fichier `linux/net/core/dev.c`: Code Linux 2.6.10

```
3139 static int __init net_dev_init(void)
3140 {
[...]
3163         /*
3164          *      Initialise les files d'attente des paquets en reception.
3165          */
3166
3167         for (i = 0; i < NR_CPUS; i++) {
3168                 struct softnet_data *queue;
3169
3170                 queue = &per_cpu(softnet_data, i);
3171                 skb_queue_head_init(&queue->input_pkt_queue);
3172                 queue->throttle = 0;
3173                 queue->cng_level = 0;
3174                 queue->avg_blog = 10; /* non nul arbitraire */
3175                 queue->completion_queue = NULL;
3176                 INIT_LIST_HEAD(&queue->poll_list);
3177                 set_bit(__LINK_STATE_START, &queue->backlog_dev.state);
3178                 queue->backlog_dev.weight = weight_p;
3179                 queue->backlog_dev.poll = process_backlog;
3180                 atomic_set(&queue->backlog_dev.refcnt, 1);
```

```
3181        }
[...]
3199 }
```

Code Linux 2.6.10

La constante (en fait variable) `weight_p` est définie dans le même fichier :

```
1341 int weight_p = 64;            /* Ancien nombre maximum de clients */
```

Nous reviendrons ci-dessous sur l'importante fonction `process_backlog()`.

19.4.2 Fonction de mise en file d'attente

La fonction `netif_rx(skb)` place le descripteur de tampon de socket passé en argument dans la file d'attente en réception du microprocesseur en cours, passe la main à la couche supérieure et renvoie une indication sur le niveau de congestion de la file d'attente, correspondant à l'une des constantes symboliques vues ci-dessus.

Code Linux 2.6.10

Elle est définie dans le fichier `linux/net/core/dev.c` :

```
1405 /**
1406  *      netif_rx        -       met a disposition le tampon au code du reseau
1407  *      @skb : [descripteur de] tampon a mettre a disposition
1408  *
1409  *      Cette fonction recoit un paquet d'un peripherique reseau et le place dans une file
1410  *      d'attente pour etre traitee par les couches (de protocole) superieures. Elle
1411  *      reussit toujours. Le tampon peut etre ecarte durant le traitement par le
1412  *      controle de congestion ou par les couches de protocole.
1413  *
1414  *      valeurs de retour :
1415  *      NET_RX_SUCCESS  (pas de congestion)
1416  *      NET_RX_CN_LOW   (congestion basse)
1417  *      NET_RX_CN_MOD   (congestion moderee)
1418  *      NET_RX_CN_HIGH  (congestion importante)
1419  *      NET_RX_DROP     (le paquet a ete ecarte)
1420  *
1421  */
1422
1423 int netif_rx(struct sk_buff *skb)
1424 {
1425         int this_cpu;
1426         struct softnet_data *queue;
1427         unsigned long flags;
1428
1429 #ifdef CONFIG_NETPOLL
1430         if (skb->dev->netpoll_rx && netpoll_rx(skb)) {
1431                 kfree_skb(skb);
1432                 return NET_RX_DROP;
1433         }
1434 #endif
1435
1436         if (!skb->stamp.tv_sec)
1437                 net_timestamp(&skb->stamp);
1438
1439         /*
1440          * Le code est arrange de telle sorte que le chemin soit le plus
1441          * court lorsque le microprocesseur est congestionne, mais encore operationnel.
1442          */
1443         local_irq_save(flags);
1444         this_cpu = smp_processor_id();
1445         queue = &__get_cpu_var(softnet_data);
1446
1447         __get_cpu_var(netdev_rx_stat).total++;
```

```
1448         if (queue->input_pkt_queue.qlen <= netdev_max_backlog) {
1449                 if (queue->input_pkt_queue.qlen) {
1450                         if (queue->throttle)
1451                                 goto drop;
1452
1453 enqueue:
1454                         dev_hold(skb->dev);
1455                         __skb_queue_tail(&queue->input_pkt_queue, skb);
1456 #ifndef OFFLINE_SAMPLE
1457                         get_sample_stats(this_cpu);
1458 #endif
1459                         local_irq_restore(flags);
1460                         return queue->cng_level;
1461                 }
1462
1463                 if (queue->throttle)
1464                         queue->throttle = 0;
1465
1466                 netif_rx_schedule(&queue->backlog_dev);
1467                 goto enqueue;
1468         }
1469
1470         if (!queue->throttle) {
1471                 queue->throttle = 1;
1472                 __get_cpu_var(netdev_rx_stat).throttled++;
1473         }
1474
1475 drop:
1476         __get_cpu_var(netdev_rx_stat).dropped++;
1477         local_irq_restore(flags);
1478
1479         kfree_skb(skb);
1480         return NET_RX_DROP;
1481 }
```

Autrement dit :

- On déclare un numéro de processeur, utile dans le cas d'un système multi-processeurs.
- On déclare une file d'attente en réception de microprocesseur.
- On déclare un vecteur de drapeaux.
- Si l'heure de réception du tampon dont le descripteur est passé en argument n'est pas spécifiée, on lui attribue l'heure en cours.
- On sauvegarde les valeurs des registres du microprocesseur en cours dans le vecteur de drapeaux.
- On détermine le numéro du microprocesseur en cours.
- On instantie la file d'attente avec celle associée à ce microprocesseur.
- On incrémente l'information sur le nombre total de trames reçues par ce microprocesseur.
- Si la taille de la file d'attente en réception du microprocesseur est inférieure à la taille maximale (autrement dit si cette file d'attente n'est pas pleine) :
  - Si la taille de la file d'attente en réception du microprocesseur est non nulle (autrement dit si la file d'attente n'est pas vide) :
    - S'il est indiqué que la file d'attente est pleine, on détruit le paquet : on incrémente l'information sur le nombre de paquets détruits pour ce microprocesseur, on restaure les valeurs des registres du microprocesseur, on libère le tampon de socket et on renvoie `NET_RX_DROP`.

- Sinon (c'est-à-dire dans le cas normal) :
  - On incrémente le compteur de références au périphérique.
  - On ajoute le descripteur de tampon passé en argument à la fin de la file d'attente en réception du microprocesseur.
  - On met éventuellement à jour les informations statistiques concernant le microprocesseur en cours.
  - On restaure les valeurs des registres du microprocesseur.
  - On renvoie le niveau de congestion du microprocesseur, tel qu'indiqué dans la file d'attente.
- Si la taille de la file d'attente en réception du microprocesseur est nulle (autrement dit si la file d'attente est vide) :
  - Si cette file d'attente est marquée comme pleine, on rectifie cette indication et, éventuellement, on réveille la file d'attente.
  - On fait appel à la fonction `netif_rx_schedule()`, étudiée dans la section suivante, en passant en argument le descripteur de périphérique virtuel, afin de passer à la couche supérieure.
  - On essaie à nouveau de placer le descripteur de tampon dans la file d'attente.
- Si la taille de la file d'attente est supérieure à la constante (autrement dit si elle est pleine) :
  - S'il n'est pas indiqué que la file d'attente est pleine, on le fait et on met à jour les informations statistiques à ce propos pour le microprocesseur.
  - On écarte le paquet.

19.5 Passage à la couche supérieure

La fonction de passage à la couche supérieure considère un à un les descripteurs de tampon de socket placés dans la file d'attente du microprocesseur. Elle les communique à la fonction de traitement associée au protocole indiqué dans le champs créé à cet effet. Avant d'entrer dans le détail de l'action, voyons les structures de données nécessaires à ce passage.

19.5.1 Type de paquet

Pour remettre un paquet à la couche supérieure, on regarde le champ `protocol` de son descripteur et on communique son descripteur à une fonction de traitement dépendant de ce champ. La détermination de cette fonction s'effectue en parcourant deux listes de *types de paquet*.

19.5.1.1 Structure de données de type de paquet

Le type d'un paquet est une instance du type `struct packet_type` défini dans le fichier `linux/include/linux/netdevice.h`:

Code Linux 2.6.10

```
509 struct packet_type {
510         unsigned short          type;   /* Il s'agit de htons(ether_type).          */
511         struct net_device       *dev;   /* NULL comme caractere de remplacement ici */
512         int                     (*func) (struct sk_buff *, struct net_device *,
513                                          struct packet_type *);
514         void                    *af_packet_priv;
515         struct list_head        list;
516 };
```

dont la signification des champs est la suivante :

- Le champ `type` spécifie l'identificateur de protocole. Il s'agit de l'une des constantes vues au chapitre 16.
- Le champ `dev` peut être occupé par un pointeur sur un périphérique réseau. Dans ce cas, seuls les paquets reçus par ce périphérique sont transmis au protocole. Si plusieurs adaptateurs réseau, mais pas tous, doivent bénéficier de cette priorité, il faut que le protocole soit enregistré séparément pour chaque périphérique réseau. Lorsque `dev` contient `NULL`, ce qui est le cas usuel, le périphérique réseau d'entrée ne joue aucun rôle dans le choix du protocole.
- Le champ `func()` est la routine de traitement prévue pour commencer le traitement du protocole.
- Le champ `af_packet_priv` peut contenir des données supplémentaires pour ce protocole. Il n'est pas utilisé en général.
- Le champ `list` sert à chaîner les différentes instances de cette structure.

Par exemple, le type de paquet associé à IP s'appelle `ip_packet_type` et est défini dans le fichier `linux/net/ipv4/ip_output.c` : Code Linux 2.6.10

```
1323 /*
1324  *      Initialiseur de la couche de protocole IP
1325  */
1326
1327 static struct packet_type ip_packet_type = {
1328         .type = __constant_htons(ETH_P_IP),
1329         .func = ip_rcv,
1330 };
```

qui nous donne `ip_rcv()` comme fonction de traitement.

19.5.1.2 Listes des types de paquet

Il existe deux structures de données concernant les types de paquets, comme le montre la figure 19.1 ([WPRMB–02], p.148) :

- L'une, une liste simplement chaînée appelée `ptype_all`, stocke les protocoles qui doivent recevoir tous les tampons de sockets entrants. En règle générale, aucun protocole n'est enregistré dans cette liste. Toutefois elle convient parfaitement à l'insertion d'outils d'analyse comme les renifleurs.
- L'autre, une table de hachage appelée `ptype_base[]`, comprend les protocoles qui ne reçoivent que les paquets comportant une identification de protocole correcte, par exemple `0x08 00` pour le protocole IP.

Ces structures de données sont définies dans le le fichier `linux/net/core/dev.c` : Code Linux 2.6.10

```
129 /*
130  *      La liste des types des paquets que nous recevons (par opposition a ceux abandonnes)
131  *      et les routines a invoquer.
132  *
133  *      Pourquoi 16 ? Parce qu'avec 16 le seul chevauchement que nous obtiendrons avec un
134  *      hachage du demi-octet de poids faible de la valeur du protocole est RARP/SNAP/X.25.
135  *
136  *      NOTE : Ceci n'est plus vrai avec l'addition des valeurs de VLAN.
137  *             Ce qui arrivera en premier n'est pas sur, mais je mise qu'il n'y aura
138  *             pas beaucoup de difference si nous utilisons les VLAN. La bonne nouvelle
139  *             est que ce protocole ne sera pas dans la liste a moins d'etre compile, donc
```

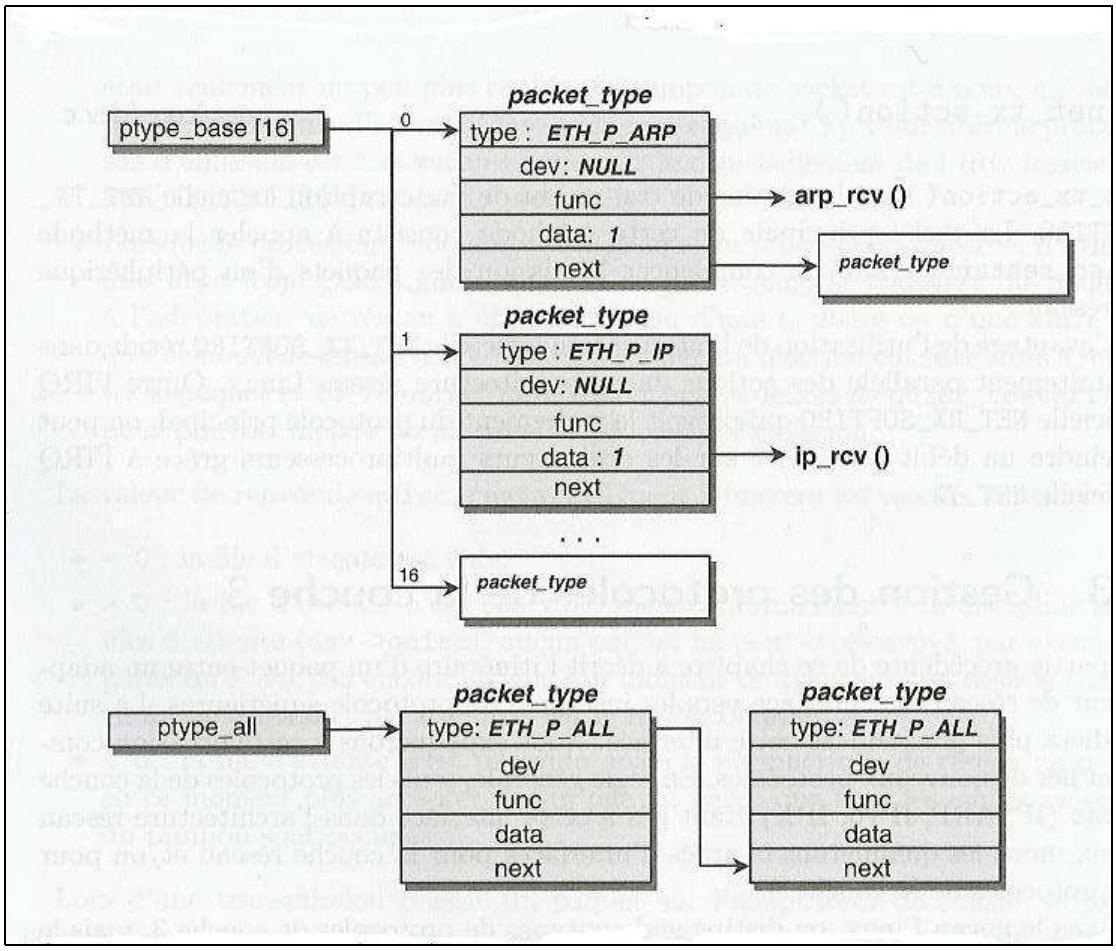


FIG. 19.1 – *Gestion des protocoles de réseau*

```
140  *             l'utilisateur moyen ne sera pas defavorablement affecte.
141  *             --BLG
142  *
143  *              0800    IP
144  *              8100    802.1Q VLAN
145  *              0001    802.3
146  *              0002    AX.25
147  *              0004    802.2
148  *              8035    RARP
149  *              0005    SNAP
150  *              0805    X.25
151  *              0806    ARP
152  *              8137    IPX
153  *              0009    Localtalk
154  *              86DD    IPv6
155  */
156
157 static spinlock_t ptype_lock = SPIN_LOCK_UNLOCKED;
158 static struct list_head ptype_base[16];  /* liste de hachage a 16 possibilites */
159 static struct list_head ptype_all;       /* Mise sur ecoute [Taps] */
```

Ces structures de données sont initialisées lors du démarrage du système de la façon indiquée dans la fonction `net_dev_init()`, définie dans le fichier `linux/net/core/dev.c`: Code Linux 2.6.10

```
3139 static int __init net_dev_init(void)
3140 {
[...]
3153         INIT_LIST_HEAD(&ptype_all);
3154         for (i = 0; i < 16; i++)
3155                 INIT_LIST_HEAD(&ptype_base[i]);
[...]
3199 }
```

19.5.1.3 Ajout et retrait d'un type de paquet

La gestion des types de paquet est effectuée grâce aux deux méthodes suivantes:

- `dev_add_pack(packet_type)` place le type de paquet dans l'une des deux structures de données. Si le champ `type` contient la valeur `ETH_P_ALL`, il est ajouté à la liste `ptype_all`. Sinon il est inséré sur la ligne correspondante de la table de hachage `ptype_base`.
- `dev_remove_pack(packet_type)` retire le type de paquet de la structure de données dans laquelle il se trouve.

La fonction `dev_add_pack()` est définie dans le fichier `linux/net/core/dev.c`: Code Linux 2.6.10

```
242 /*
243  *      Ajoute un identificateur de protocole a la liste. Maintenant que la routine d'entree
244  *      est plus maligne, nous pouvons nous dispenser de toutes les choses compliquees
245  *      qui ont ete utilisees jusqu'ici.
246  *
247  *      ATTENTION !!! Les routines de protocole mutilant les paquets d'entree
248  *      DOIVENT ETRE les dernieres dans les chaines de hachage. Verifier que les
249  *      routines de protocole DOIVENT demarrer par la chaine confuse ptype_all dans net_bh.
250  *      C'est vrai ici, ne pas changer.
251  *      Des explications suivent : si une routine de protocole mutilant les paquets etait
252  *      la premiere sur la liste, elle ne serait pas capable de savoir que le paquet
253  *      doit etre clone et copie, aussi serait-il change
254  *      et les lectures ulterieures obtiendraient des paquets incomplets.
255  *                                                      --ANK (980803)
256  */
257
258 /**
259  *      dev_add_pack - ajoute une routine de paquet
260  *      @pt : declaration du type de paquet
261  *
262  *      Ajoute une routine de protocole a la pile reseau. Le &packet_type passe
263  *      est lie dans les listes du noyau et ne peut pas etre libere avant qu'il ne soit
264  *      retire des listes du noyau.
265  *
266  *      Cet appel ne s'assoupit pas, aussi ne peut-il pas
267  *      garantir que tous les microprocesseurs qui sont en train de recevoir des paquets
268  *      verront le nouveau type de paquet (jusqu'au paquet recu suivant).
269  */
270
271 void dev_add_pack(struct packet_type *pt)
272 {
273         int hash;
274
275         spin_lock_bh(&ptype_lock);
276         if (pt->type == htons(ETH_P_ALL)) {
277                 netdev_nit++;
278                 list_add_rcu(&pt->list, &ptype_all);
```

```
279         } else {
280                 hash = ntohs(pt->type) & 15;
281                 list_add_rcu(&pt->list, &ptype_base[hash]);
282         }
283         spin_unlock_bh(&ptype_lock);
284 }
```

Code Linux 2.6.10

La fonction `dev_remove_pack()` est définie dans le fichier `linux/net/core/dev.c`:

```
290 /**
291  *      __dev_remove_pack        - retire une routine de paquet
292  *      @pt : declaration du type de paquet
293  *
294  *      Retire une routine de protocole qui a ete precedemment ajoutee aux routines de
295  *      protocole du noyau avec dev_add_pack(). Le &packet_type passe est retire
296  *      des listes du noyau et peut ainsi etre libere ou reutilise une fois que cette
297  *      fonction est terminee.
298  *
299  *      Le type de paquet peut encore etre en train d'etre utilise par les recepteurs
300  *      et ne doit pas etre libere avant que tous les microprocesseurs soient entres
301  *      dans un etat latent.
302  */
303 void __dev_remove_pack(struct packet_type *pt)
304 {
305         struct list_head *head;
306         struct packet_type *pt1;
307
308         spin_lock_bh(&ptype_lock);
309
310         if (pt->type == htons(ETH_P_ALL)) {
311                 netdev_nit--;
312                 head = &ptype_all;
313         } else
314                 head = &ptype_base[ntohs(pt->type) & 15];
315
316         list_for_each_entry(pt1, head, list) {
317                 if (pt == pt1) {
318                         list_del_rcu(&pt->list);
319                         goto out;
320                 }
321         }
322
323         printk(KERN_WARNING "dev_remove_pack: %p not found.\n", pt);
324 out:
325         spin_unlock_bh(&ptype_lock);
326 }
327 /**
328  *      dev_remove_pack  - retire une routine de paquet
329  *      @pt : declaration du type de paquet
330  *
331  *      Retire une routine de protocole qui avait ete anterieurement ajoutee aux routines
332  *      de protocole du noyau avec dev_add_pack(). Le &packet_type passe est retire
333  *      des listes du noyau et peut ainsi etre libere ou reutilise une fois que cette
334  *      fonction est terminee.
335  *
336  *      Cet appel s'assoupit afin de garantir qu'aucun microprocesseur n'est a la recherche
337  *      de type de paquet apres retour de la fonction.
338  */
339 void dev_remove_pack(struct packet_type *pt)
340 {
341         __dev_remove_pack(pt);
342
343         synchronize_net();
```

```
344 }
```

19.5.2 Processus de passage à la couche supérieure

Le passage à la couche supérieure a lieu par l'intermédiaire de l'interruption logicielle NET_RX_SOFTIRQ. Celle-ci n'a pas besoin de bloquer les interruptions et peut se faire au gré de l'ordonnanceur.

La fonction netif_rx_schedule() est définie dans le fichier linux/include/linux/netdevice.h: Code Linux 2.6.10

```
826 /* Essaie de reprogrammer l'election. Appelee par une routine d'interruption. */
827
828 static inline void netif_rx_schedule(struct net_device *dev)
829 {
830         if (netif_rx_schedule_prep(dev))
831                 __netif_rx_schedule(dev);
832 }
```

Elle prépare au passage puis elle passe la main à la couche réseau de façon proprement dite.

19.5.2.1 Préparation

La fonction de préparation netif_rx_schedule_prep() est définie dans le fichier linux/include/linux/netdevice.h: Code Linux 2.6.10

```
799 /* Programme l'interruption de reception maintenant ? */
800
801 static inline int netif_rx_schedule_prep(struct net_device *dev)
802 {
803         return netif_running(dev) &&
804                 !test_and_set_bit(__LINK_STATE_RX_SCHED, &dev->state);
805 }
```

Autrement dit :

- On vérifie que la carte est active et peut être utilisée.
  La fonction netif_running() est définie plus haut dans le même fichier : Code Linux 2.6.10

  ```
  635 static inline int netif_running(const struct net_device *dev)
  636 {
  637         return test_bit(__LINK_STATE_START, &dev->state);
  638 }
  ```

- On positionne l'état du descripteur de peripherique reseau pour indiquer que l'on va passer à l'interruption logicielle.

19.5.2.2 Appel de l'interruption logicielle

La fonction __netif_rx_schedule() est définie dans le fichier linux/include/linux/netdevice.h: Code Linux 2.6.10

```
807 /* Ajoute une interface en debut de liste d'election en reception. Ceci suppose que _prep a
808  * deja ete appele et qu'il a renvoye 1.
809  */
810
811 static inline void __netif_rx_schedule(struct net_device *dev)
812 {
813         unsigned long flags;
814
```

```
815         local_irq_save(flags);
816         dev_hold(dev);
817         list_add_tail(&dev->poll_list, &__get_cpu_var(softnet_data).poll_list);
818         if (dev->quota < 0)
819                 dev->quota += dev->weight;
820         else
821                 dev->quota = dev->weight;
822         __raise_softirq_irqoff(NET_RX_SOFTIRQ);
823         local_irq_restore(flags);
824 }
```

Autrement dit :

- On déclare un vecteur de drapeaux.
- On sauvegarde les valeurs des registres du microprocesseur en cours.
- On incrémente le compteur d'utilisation du descripteur de périphérique réseau passé en argument.
- On ajoute les éléments de la file d'attente du descripteur de périphérique passé en entrée `dev->poll_list` à la liste de files d'attente en réception du microprocesseur en cours.

  La fonction générale `list_add_tail()` est définie dans le fichier `linux/include/linux/list.h`:

Code Linux 2.6.10

```
70 /**
71  * list_add_tail - ajoute une nouvelle entree
72  * @new : nouvelle entree a etre ajoutee
73  * @head : tete de la liste a laquelle il faut ajouter
74  *
75  * Insere une nouvelle entree avant la tete specifiee.
76  * Ceci est utile pour implementer les files d'attente.
77  */
78 static inline void list_add_tail(struct list_head *new, struct list_head *head)
79 {
80         __list_add(new, head->prev, head);
81 }
```

- On met à jour le quota du descripteur de périphérique réseau.
- On fait appel à l'interruption logicielle `NET_RX_SOFTIRQ` pour passer au descripteur de périphérique virtuel.

  La macro générale `__raise_softirq_irqoff()` est définie dans le fichier `linux/include/linux/interrupt.h`:

Code Linux 2.6.10

```
108 #define __raise_softirq_irqoff(nr) do { local_softirq_pending() |= 1UL << (nr); }
                                        while (0)
```

- On restaure les valeurs des registres du microprocesseur.

19.5.2.3 Routine de service de l'interruption logicielle

La routine de service de l'interruption logicielle `NET_RX_SOFTIRQ` est précisée dans la fonction `net_dev_init()`, définie dans le fichier `linux/net/core/dev.c`:

Code Linux 2.6.10

```
3139 static int __init net_dev_init(void)
3140 {
[...]
3190         open_softirq(NET_TX_SOFTIRQ, net_tx_action, NULL);
3191         open_softirq(NET_RX_SOFTIRQ, net_rx_action, NULL);
[...]
3199 }
```

19.5.2.4 Appel du descripteur de périphérique virtuel

La routine de service `net_rx_action()` de l'interruption logicielle `NET_RX_SOFTIRQ` est définie dans le fichier `linux/net/core/dev.c`: Code Linux 2.6.10

```
1764 static void net_rx_action(struct softirq_action *h)
1765 {
1766         struct softnet_data *queue = &__get_cpu_var(softnet_data);
1767         unsigned long start_time = jiffies;
1768         int budget = netdev_max_backlog;
1769
1770
1771         local_irq_disable();
1772
1773         while (!list_empty(&queue->poll_list)) {
1774                 struct net_device *dev;
1775
1776                 if (budget <= 0 || jiffies - start_time > 1)
1777                         goto softnet_break;
1778
1779                 local_irq_enable();
1780
1781                 dev = list_entry(queue->poll_list.next,
1782                                  struct net_device, poll_list);
1783
1784                 if (dev->quota <= 0 || dev->poll(dev, &budget)) {
1785                         local_irq_disable();
1786                         list_del(&dev->poll_list);
1787                         list_add_tail(&dev->poll_list, &queue->poll_list);
1788                         if (dev->quota < 0)
1789                                 dev->quota += dev->weight;
1790                         else
1791                                 dev->quota = dev->weight;
1792                 } else {
1793                         dev_put(dev);
1794                         local_irq_disable();
1795                 }
1796         }
1797 out:
1798         local_irq_enable();
1799         return;
1800
1801 softnet_break:
1802         __get_cpu_var(netdev_rx_stat).time_squeeze++;
1803         __raise_softirq_irqoff(NET_RX_SOFTIRQ);
1804         goto out;
1805 }
```

Autrement dit:

- On déclare une file d'attente en réception de microprocesseur, que l'on instantie avec celle du microprocesseur en cours.
- On déclare une heure de début, que l'on instantie avec l'heure en cours.
- On déclare un nombre de paquets restants à traiter, que l'on initialise avec le nombre maximum d'éléments permis dans une telle file d'attente.
- On inhibe les interruptions sur le microprocesseur en cours.
- Tant que la liste d'élection de la file d'attente en réception du microprocesseur n'est pas vide:
  - On déclare un descripteur de périphérique réseau.

- Si le nombre de paquets à traiter est (négatif ou) nul ou si un tick s'est écoulé depuis le début, on donne la main au gestionnaire des tâches pour qu'une autre activité puisse se dérouler. On évite ainsi que le traitement bloque le reste des activités de l'ordinateur. De façon détaillée : on incrémente l'information statistique sur le nombre de pincements de la file d'attente du microproceseur en cours, on lève `NET_RX_SOFTIRQ` et enfin on restaure les interruptions.
- On accepte à nouveau les interruptions sur ce microprocesseur.
- On instantie le descripteur de périphérique réseau avec le descripteur de périphérique réseau virtuel de la file d'attente du microprocesseur en cours.
- On fait appel à la fonction d'élection du descripteur de périphérique virtuel.

  Nous avons vu lors de l'initialisation des files d'attente en réception de microprocesseur que cette fonction d'élection est `proces_backlog()`. Nous l'étudierons dans la section suivante.
- Si le quota associé au descripteur de périphérique réseau virtuel est (négatif ou) nul ou si la fonction d'élection renvoie une valeur non nulle : on inhibe les interruptions sur ce microprocesseur, on retire un élément de la liste d'élection du descripteur de périphérique réseau virtuel, on fait passer un élément de la liste d'élection de la file d'attente en réception du microprocesseur vers la liste d'élection du descripteur de périphérique réseau virtuel et on met à jour le quota du descripteur de périphérique réseau virtuel.
- Sinon on libère le descripteur de périphérique réseau virtuel et on inhibe les interruptions.

- On permet à nouveau les interruptions et on a terminé.

19.6 Préparation au traitement dans la couche supérieure

19.6.1 Détermination du paquet à traiter

Nous venons de voir que la fonction précédente fait appel à la fonction `process_backlog()`. Celle-ci est définie dans le fichier `linux/net/core/dev.c` :

Code Linux 2.6.10

```
1716 static int process_backlog(struct net_device *backlog_dev, int *budget)
1717 {
1718         int work = 0;
1719         int quota = min(backlog_dev->quota, *budget);
1720         struct softnet_data *queue = &__get_cpu_var(softnet_data);
1721         unsigned long start_time = jiffies;
1722
1723         for (;;) {
1724                 struct sk_buff *skb;
1725                 struct net_device *dev;
1726
1727                 local_irq_disable();
1728                 skb = __skb_dequeue(&queue->input_pkt_queue);
1729                 if (!skb)
1730                         goto job_done;
1731                 local_irq_enable();
1732
1733                 dev = skb->dev;
1734
1735                 netif_receive_skb(skb);
1736
1737                 dev_put(dev);
```

```
1738
1739                work++;
1740
1741                if (work >= quota || jiffies - start_time > 1)
1742                        break;
1743
1744        }
1745
1746        backlog_dev->quota -= work;
1747        *budget -= work;
1748        return -1;
1749
1750 job_done:
1751        backlog_dev->quota -= work;
1752        *budget -= work;
1753
1754        list_del(&backlog_dev->poll_list);
1755        smp_mb__before_clear_bit();
1756        netif_poll_enable(backlog_dev);
1757
1758        if (queue->throttle)
1759                queue->throttle = 0;
1760        local_irq_enable();
1761        return 0;
1762 }
```

Autrement dit :

- On déclare un nombre de paquets traités, que l'on initialise à zéro.
- On déclare un quota, que l'on initialise au minimum entre le quota du périphérique réseau virtuel passé en argument et le budget passé en argument.
- On déclare une file d'attente en réception de microprocesseur, que l'on instantie avec celle du microprocesseur en cours.
- On déclare une heure de début, que l'on initialise avec l'heure en cours.
- On entre dans une boucle (pseudo-)infinie :
  - On déclare un descripteur de tampon de socket et un descripteur de périphérique réseau.
  - On inhibe les interruptions matérielles du microprocesseur en cours.
  - On essaie d'instantier le descripteur de tampon avec le premier élément de la file d'attente en réception du microprocesseur.
  - Si on n'y parvient pas, c'est que cette file d'attente est vide. Dans ce cas :
    - On diminue le quota du périphérique réseau virtuel et le budget passé en argument du nombre de paquets traités.
    - On retire le premier élément de la liste d'élection du périphérique réseau virtuel.
    - On effectue un traitement dans le cas d'un système à plusieurs microprocesseurs.
    - On met en service l'élection pour le périphérique réseau virtuel.

      La fonction `netif_poll_enable()` est définie dans le fichier `linux/include/linux/netdevice.h`:

      Code Linux 2.6.10

```
879 static inline void netif_poll_enable(struct net_device *dev)
880 {
881         clear_bit(__LINK_STATE_RX_SCHED, &dev->state);
882 }
```

  - Si la file d'attente en réception du microprocesseur est indiquée comme pleine, on rectifie cette indication.
  - On permet à nouveau les interruptions matérielles sur le microprocesseur en cours et on renvoie 0.
- On permet à nouveau les interruptions matérielles sur le microprocesseur en cours.
- On instantie le descripteur de périphérique avec celui associé au descripteur de tampon de socket.
- On traite le paquet, grâce à la fonction `netif_receive_skb()` étudiée dans la sous-section ci-dessous.
- On libère le descripteur de tampon de socket.
- On incrémente le nombre de paquets traités.
- Si le nombre de paquets traités est inférieur au quota et si cela fait moins d'un tick que la fonction a débuté, on revient au début de la boucle pour traiter le paquet suivant.

- On diminue le quota du périphérique réseau virtuel et le budget passé en argument du nombre de paquets traités.
- On renvoie -1 pour indiquer qu'il reste des paquets à traiter.

19.6.2 Traitement d'un paquet

Code Linux 2.6.10

On vient de voir que la fonction précédente fait appel à la fonction `netif_receive_skb()` pour le traitement d'un paquet. Celle-ci est définie dans le fichier `linux/net/core/dev.c`:

```
1625 int netif_receive_skb(struct sk_buff *skb)
1626 {
1627         struct packet_type *ptype, *pt_prev;
1628         int ret = NET_RX_DROP;
1629         unsigned short type;
1630
1631 #ifdef CONFIG_NETPOLL
1632         if (skb->dev->netpoll_rx && skb->dev->poll && netpoll_rx(skb)) {
1633                 kfree_skb(skb);
1634                 return NET_RX_DROP;
1635         }
1636 #endif
1637
1638         if (!skb->stamp.tv_sec)
1639                 net_timestamp(&skb->stamp);
1640
1641         skb_bond(skb);
1642
1643         __get_cpu_var(netdev_rx_stat).total++;
1644
1645         skb->h.raw = skb->nh.raw = skb->data;
1646         skb->mac_len = skb->nh.raw - skb->mac.raw;
1647
1648         pt_prev = NULL;
1649
1650         rcu_read_lock();
1651
1652 #ifdef CONFIG_NET_CLS_ACT
1653         if (skb->tc_verd & TC_NCLS) {
1654                 skb->tc_verd = CLR_TC_NCLS(skb->tc_verd);
1655                 goto ncls;
1656         }
1657 #endif
```

```
1658
1659         list_for_each_entry_rcu(ptype, &ptype_all, list) {
1660                 if (!ptype->dev || ptype->dev == skb->dev) {
1661                         if (pt_prev)
1662                                 ret = deliver_skb(skb, pt_prev);
1663                         pt_prev = ptype;
1664                 }
1665         }
1666
1667 #ifdef CONFIG_NET_CLS_ACT
1668         if (pt_prev) {
1669                 ret = deliver_skb(skb, pt_prev);
1670                 pt_prev = NULL; /* noone else should process this after*/
1671         } else {
1672                 skb->tc_verd = SET_TC_OK2MUNGE(skb->tc_verd);
1673         }
1674
1675         ret = ing_filter(skb);
1676
1677         if (ret == TC_ACT_SHOT || (ret == TC_ACT_STOLEN)) {
1678                 kfree_skb(skb);
1679                 goto out;
1680         }
1681
1682         skb->tc_verd = 0;
1683 ncls:
1684 #endif
1685
1686         handle_diverter(skb);
1687
1688         if (handle_bridge(&skb, &pt_prev, &ret))
1689                 goto out;
1690
1691         type = skb->protocol;
1692         list_for_each_entry_rcu(ptype, &ptype_base[ntohs(type)&15], list) {
1693                 if (ptype->type == type &&
1694                     (!ptype->dev || ptype->dev == skb->dev)) {
1695                         if (pt_prev)
1696                                 ret = deliver_skb(skb, pt_prev);
1697                         pt_prev = ptype;
1698                 }
1699         }
1700
1701         if (pt_prev) {
1702                 ret = pt_prev->func(skb, skb->dev, pt_prev);
1703         } else {
1704                 kfree_skb(skb);
1705                 /* Jamal, maintenant vous n'echapperez pas a une explication sur
1706                  * la facon dont vous utilisez ceci. :-)
1707                  */
1708                 ret = NET_RX_DROP;
1709         }
1710
1711 out:
1712         rcu_read_unlock();
1713         return ret;
1714 }
```

Autrement dit :

- On déclare un type de paquet et un type de paquet précédent.
- On déclare une valeur de retour, que l'on initialise avec `NET_RX_DROP` pour le cas d'un paquet

écarté.

- On déclare un type de protocole.
- Si l'estampille temporelle n'est pas spécifiée pour le descripteur de tampon de socket passé en argument, on la renseigne avec l'heure en cours.
- On renseigne le champ périphérique réel du descripteur de tampon passé en argument.

Code Linux 2.6.10

La fonction `skb_bond()` est définie dans le fichier `linux/net/core/dev.c`:

```
1498 static __inline__ void skb_bond(struct sk_buff *skb)
1499 {
1500         struct net_device *dev = skb->dev;
1501
1502         if (dev->master) {
1503                 skb->real_dev = skb->dev;
1504                 skb->dev = dev->master;
1505         }
1506 }
```

- On met à jour l'information statistique sur le nombre total de paquets reçus et traités par le microprocesseur.
- On renseigne les champs en-tête matériel et en-tête réseau du descripteur de périphérique pour le cas d'un paquet brut.
- On renseigne le champ longueur d'en-tête matériel du descripteur de périphérique.
- On verrouille le processus de lecture et mise à jour.

Code Linux 2.6.10

Les macros générales `rcu_read_lock()` et `rcu_read_unlock()` sont définies dans le fichier `linux/include/linux/rcupdate.h`:

```
157 /**
158  * rcu_read_lock - marque le debut d'une section critique du cote lecture RCU.
159  *
160  * Lorsque synchronize_kernel() est invoque sur un CPU alors que d'autres CPU
161  * sont dans des sections critiques du cote lecture RCU,
162  * on est assure que synchronize_kernel() bloque jusqu'a ce que tous les autres
163  * CPU sortent de leurs sections critiques. De meme, si call_rcu() est invoque
164  * sur un CPU alors que d'autres CPU sont dans des sections critiques du cote lecture
165  * RCU, l'invocation de l'appel RCU correspondant est retarde
166  * jusqu'a ce que tous les autres CPU sortent de leurs sections critiques.
167  *
168  * Noter, cependant, qu'on permet que les appels RCU tournent de facon concurrente
169  * avec des sections critiques du cote lecture RCU. Une facon dont ceci peut arriver
170  * est via la sequence suivante d'evenements : (1) CPU 0 entre dans une section
171  * critique du cote lecture RCU, (2) CPU 1 invoque call_rcu() pour enregistrer
172  * un appel RCU, (3) CPU 0 sort de la section critique du cote lecture RCU,
173  * (4) CPU 2 entre dans une section lecture du cote lecture RCU, (5) l'appel RCU
174  * est invoque. Ceci est legal, puisque la section critique du cote lecture RCU
175  * qui est en train de tourner en meme temps que call_rcu() (et que ainsi
176  * pourrait etre en train de referencer quelque chose que l'appel RCU correspondant
177  * pourrait liberer) a termine avant que l'appel RCU correspondant
178  * soit invoque.
179  *
180  * Les sections critiques du cote lecture RCU peuvent etre emboitees. Toutes les actions
181  * retardees le seront jusqu'a ce que la section critique du cote lecture RCU la plus
182  * exterieure ait terminee.
183  *
184  * Il est illegal de bloquer alors qu'on est dans une section critique du cote
       lecture RCU.
185  */
186 #define rcu_read_lock()         preempt_disable()
187
188 /**
```

```
189  * rcu_read_unlock - marque la fin d'une section critique du cote lecture RCU.
190  *
191  * Voir rcu_read_lock() pour plus d'informations.
192  */
193 #define rcu_read_unlock()       preempt_enable()
```

- On cherche si le type du tampon apparaît dans la liste des types de paquet de traitement de tous les paquets. Si c'est le cas on dépose le tampon à l'intention de la couche supérieure, grâce à la fonction `deliver_skb()` étudiée dans la sous-section suivante.
- Si le déroutement est supporté par le périphérique, on traite celui-ci.

  La fonction `handle_diverter()` est définie dans le fichier `linux/include/linux/divert.h`:

Code Linux 2.6.10

```
111 #ifdef CONFIG_NET_DIVERT
112 #include <linux/netdevice.h>
113
114 int alloc_divert_blk(struct net_device *);
115 void free_divert_blk(struct net_device *);
116 int divert_ioctl(unsigned int cmd, struct divert_cf __user *arg);
117 void divert_frame(struct sk_buff *skb);
118 static inline void handle_diverter(struct sk_buff *skb)
119 {
120         /* si le deroutement est supporte sur le peripherique, alors allons-y */
121         if (skb->dev->divert && skb->dev->divert->divert)
122                 divert_frame(skb);
123 }
124
125 #else
126 # define alloc_divert_blk(dev)          (0)
127 # define free_divert_blk(dev)           do {} while (0)
128 # define divert_ioctl(cmd, arg)         (-ENOPKG)
129 # define handle_diverter(skb)           do {} while (0)
130 #endif
```

  La fonction `divert_frame()` de déroutement proprement dit ne nous intéressera pas ici.
- Lorsque l'ordinateur a été configuré en tant que passerelle (`CONFIG_BRIDGE`), le paquet est transmis à la méthode `handle_bridge()`, ce qui ne nous intéressera pas ici.
- On cherche si le type du tampon apparaît dans la liste des types de paquet de base. Si c'est le cas on dépose le tampon à l'intention de la couche supérieure adéquate.

  Dans le cas qui nous intéresse, on trouvera le type de paquet IP.
- Si on ne trouve aucun type adéquat, on libère le descripteur de tampon.
- On déverrouille le dispositif d'écriture et mise à jour et on renvoie `NET_RX_DROP` ou la valeur renvoyée par la fonction `deliver_skb()`.

19.6.3 Livraison à la couche supérieure

La fonction en ligne `deliver_skb()` est définie dans le fichier `linux/net/core/dev.c`:

Code Linux 2.6.10

```
1554 static __inline__ int deliver_skb(struct sk_buff *skb,
1555                                     struct packet_type *pt_prev)
1556 {
1557         atomic_inc(&skb->users);
1558         return pt_prev->func(skb, skb->dev, pt_prev);
1559 }
```

Elle se contente d'incrémenter le nombre d'utilisateurs du descripteur de tampon et de faire appel à la routine de traitement du protocole en question, `ip_rcv()` rappelons-le dans le cas de IPv4.

19.7 Récupération des informations statistiques

19.7.1 Cas général

Nous avons vu lors de l'étude des fonctions précédentes qu'un certain nombre d'informations statistiques étaient placées soit dans une variable dépendant du microprocesseur actif, soit dans la partie privée du decripteur de périphérique dans le cas de la carte 3Com 501. La fonction `get_stats()` spécifique à la carte permet de récupérer ces dernières.

19.7.2 Cas de la carte 3Com 501

Code Linux 2.6.10

Comme nous l'avons vu au chapitre 17, la fonction `get_stats()` spécifique à la carte 3Com 501 s'appelle `el1_get_stats()`. Celle-ci est définie dans le fichier `linux/drivers/net/3c501.c`:

```
818 /**
819  * el1_get_stats :
820  * @dev : La carte a partir de laquelle obtenir les statistiques
821  *
822  * Pour des peripheriques plus elegants, cette fonction est necessaire pour rapatrier les
823  * statistiques de la carte elle-meme. La 3c501 n'a pas de statistiques dans le materiel.
824  * Nous les maintenons tous, aussi sont-elles toujours par definition a jour.
825  *
826  * Renvoie les statistiques pour la carte a partir des donnes privees de la carte
827  */
828
829 static struct net_device_stats *el1_get_stats(struct net_device *dev)
830 {
831         struct net_local *lp = netdev_priv(dev);
832         return &lp->stats;
833 }
```

Comme la carte 3Com 501 ne conserve pas de données statistiques, cette fonction renvoie la partie privée du descripteur de périphérique passé en argument.

Chapitre 20

Implémentation Linux de IPv4

Nous allons commencer à étudier le début de l'implémentation de la couche IPv4 dans ce chapitre, à savoir l'en-tête IPv4 le calcul de la somme de contrôle.

20.1 Implémentation Linux de l'en-tête IP

L'en-tête IP est une entité du type `struct iphdr`. Celui-ci est défini dans le fichier en-tête `linux/include/linux/ip.h`:

Code Linux 2.6.10

```
6   *              Definitions pour le protocole IP.
7   *
8   * Version :    @(#)ip.h        1.0.2   04/28/93
9   *
10  * Auteurs :    Fred N. van Kempen, <waltje@uWalt.NL.Mugnet.ORG>
[...]
16  */
17 #ifndef _LINUX_IP_H
18 #define _LINUX_IP_H
19 #include <asm/byteorder.h>
[...]
167 struct iphdr {
168 #if defined(__LITTLE_ENDIAN_BITFIELD)
169         __u8    ihl:4,
170                 version:4;
171 #elif defined (__BIG_ENDIAN_BITFIELD)
172         __u8    version:4,
173                 ihl:4;
174 #else
175 #error  "Please fix <asm/byteorder.h>"
176 #endif
177         __u8    tos;
```

```
178         __u16   tot_len;
179         __u16   id;
180         __u16   frag_off;
181         __u8    ttl;
182         __u8    protocol;
183         __u16   check;
184         __u32   saddr;
185         __u32   daddr;
186         /* Les options debutent ici. */
187 };
```

Les drapeaux de cet en-tête IP sont définis comme constantes symboliques dans le en-tête `linux/include/net/ip.h`:

Code Linux 2.6.10

```
69 /* Drapeaux IP. */
70 #define IP_CE           0x8000          /* Drapeau : "Congestion"           */
71 #define IP_DF           0x4000          /* Drapeau : "Don't Fragment"       */
72 #define IP_MF           0x2000          /* Drapeau : "More Fragments"       */
73 #define IP_OFFSET       0x1FFF          /* partie "Decalage de fragment"    */
```

Les types de service sont définis comme constantes symboliques dans le fichier en-tête `linux/include/linux/ip.h`:

Code Linux 2.6.10

```
21 #define IPTOS_TOS_MASK                  0x1E
22 #define IPTOS_TOS(tos)                  ((tos)&IPTOS_TOS_MASK)
23 #define IPTOS_LOWDELAY                  0x10
24 #define IPTOS_THROUGHPUT                0x08
25 #define IPTOS_RELIABILITY               0x04
26 #define IPTOS_MINCOST                   0x02
27
28 #define IPTOS_PREC_MASK                 0xE0
29 #define IPTOS_PREC(tos)                 ((tos)&IPTOS_PREC_MASK)
30 #define IPTOS_PREC_NETCONTROL           0xe0
31 #define IPTOS_PREC_INTERNETCONTROL      0xc0
32 #define IPTOS_PREC_CRITIC_ECP           0xa0
33 #define IPTOS_PREC_FLASHOVERRIDE        0x80
34 #define IPTOS_PREC_FLASH                0x60
35 #define IPTOS_PREC_IMMEDIATE            0x40
36 #define IPTOS_PREC_PRIORITY             0x20
37 #define IPTOS_PREC_ROUTINE              0x00
```

20.2 Calcul de la somme de contrôle IP

Le calcul de la somme de contrôle IP est effectué par la fonction `ip_fast_csum()`. L'implémentation de celle-ci dépend de l'architecture. Pour les microprocesseurs Intel, elle est définie dans le fichier `linux/include/asm-i386/checksum.h`:

Code Linux 2.6.10

```
54 /*
55  *      Ceci est une version de ip_compute_csum() optimisee pour les en-tetes IP,
56  *      qui calcule toujours sur des frontieres de 4 octets.
57  *
58  *      Par Jorge Cwik <jorge@laser.satlink.net>, adapte pour linux par
59  *      Arnt Gulbrandsen.
60  */
61 static inline unsigned short ip_fast_csum(unsigned char * iph,
62                                           unsigned int ihl)
63 {
64         unsigned int sum;
65
66         __asm__ __volatile__(
67             "movl (%1), %0      ;\n"
```

```
68              "subl $4, %2        ;\n"
69              "jbe 2f             ;\n"
70              "addl 4(%1), %0     ;\n"
71              "adcl 8(%1), %0     ;\n"
72              "adcl 12(%1), %0    ;\n"
73 "1:          adcl 16(%1), %0     ;\n"
74              "lea 4(%1), %1      ;\n"
75              "decl %2            ;\n"
76              "jne 1b             ;\n"
77              "adcl $0, %0        ;\n"
78              "movl %0, %2        ;\n"
79              "shrl $16, %0       ;\n"
80              "addw %w2, %w0      ;\n"
81              "adcl $0, %0        ;\n"
82              "notl %0            ;\n"
83 "2:                             ;\n"
84         /* Puisque les registres d'entree qui sont charges avec iph et ipl
85            sont modifies, nous devons aussi les specifier comme sortie, sinon gcc
86            supposera qu'ils contiennent les valeurs originelles. */
87         : "=r" (sum), "=r" (iph), "=r" (ihl)
88         : "1" (iph), "2" (ihl)
89         : "memory");
90         return(sum);
91 }
```

Chapitre 21

Tables de routage

Sous Linux, la table de redirection est constituée de plusieurs **tables de routage :** au moins une **table de routage locale** (contenant les adresses IP de la machine locale) et une **table de routage principale** (contenant les autres adresses)**.**

Nous allons étudier dans ce chapitre comment est constituée la table de redirection, comment y ajouter et y retirer des éléments et enfin comment la consulter**.**

21.1 Remplissage des tables de routage

21.1.1 Calcul de la table de redirection

21.1.1.1 Principe

Dans le cas d'un système d'extrémité relié à un réseau par un système point à point, la table de redirection est très simple. On peut donc la remplir à la main (en fait grâce à un fichier de commandes).

Dans le cas d'un système d'extrémité situé dans un réseau local, le remplissage de la table est un peu plus long puisqu'il faut une ligne par machine sur le réseau local (en général moins de 255).

Dans le cas d'un routeur, c'est plus difficile puisqu'il y a plusieurs interfaces réseau et que la route peut changer suivant l'encombrement des différents chemins. On utilise une application utiliastauer (ceci n'est pas du ressort du noyau) pour maintenir les tables de routage. Il n'existe pas de logiciel libre standard pour cela bien que certains comment à émerger.

21.1.1.2 Routage statique et routage dynamique

Nous venons de voir que les informations concernées sur un système hôte sont peu volumineuses et ne sont appelées à être modifiées que lorsque la topologie du réseau change. Il s'agit uniquement de connaître quelques adresses locales ainsi que l'adresse du routeur derrière lequel se trouve le "reste du réseau Internet". On parle alors de **routage statique**, en opposition au **routage dynamique** faisant appel à des protocoles de routage, surtout utilisés sur les routeurs.

21.1.2 La commande `ip`

Les tables de routages sont renseignées au niveau utilisateur, et non au niveau noyau. Le système BSD a conçu la commande `route` pour cela. Celle-ci est remplacée, ainsi que anciennes commandes `ifconfig`, `netstat` et `arp` entre autres, de nos jours par la commande `ip` qui utilise les sockets Netlink.

21.1.2.1 Syntaxe générale

La syntaxe de la commande `ip` est :

```
ip [options] objet [commande] [arguments]
```

Options.- Les options possibles sont `-V`, `-s`, `-f` et `-r` pour obtenir le numéro de version de la commande, pour obtenir des statistiques, pour indiquer la famille d'adresses (`inet`, `inet6` ou `link`) et pour utiliser un nom au lieu d'une adresse IP (avec `r` pour *Resolve*).

Objet.- Ce paramètre spécifie ce que l'on veut gérer. Il s'agit de `link` (on obtient alors l'analogue de la commande `ifconfig`), `address` pour attibuer l'adresse à une interface, `neighbor` pour l'analogue de `arp`, `route` qui est ce qui nous intéresse ici, `multicast address`, `multicast route` et `tunnel`.

Commande.- Spécifie l'action sur l'objet. On a toujours `add`, `delete`, `set` et `show`.

21.1.2.2 Les interfaces

L'objet `link` ne nous intéresse pas vraiment ici. Voyons cependant les informations qu'il peut fournir :

```
# ip link show
1: lo: <LOOPBACK,UP> mtu 16436 qdisc noqueue
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
2: eth0: <BROADCAST,MULTICAST,NOTRAILERS,UP> mtu 1500 qdisc pfifo_fast qlen 100
    link/ether 00:04:75:ea:1c:7b brd ff:ff:ff:ff:ff:ff
3: sit0@NONE: <NOARP> mtu 1480 qdisc noop
    link/sit 0.0.0.0 brd 0.0.0.0
#
```

Le résultat de la commande nous fournit l'index de l'interface, son nom, son mode d'opération (par exemple `LOOPBACK`, `BROADCAST`, `MULTICAST`), son statut (par exemple `UP` pour actif), son unité de transfert maximum, sa stratégie de mise en file d'attente, la longueur de sa file d'attente et, dans le cas d'un système à diffusion, le type de son adresse MAC, sa valeur et la valeur de diffusion générale.

21.1.2.3 Adresse des interfaces

De même l'objet `address` ne nous intéresse pas vraiment ici. Voyons cependant les informations qu'il peut fournir :

```
# ip address show
1: lo: <LOOPBACK,UP> mtu 16436 qdisc noqueue
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
    inet 127.0.0.1/8 brd 127.255.255.255 scope host lo
    inet6 ::1/128 scope host
2: eth0: <BROADCAST,MULTICAST,NOTRAILERS,UP> mtu 1500 qdisc pfifo_fast qlen 100
    link/ether 00:04:75:ea:1c:7b brd ff:ff:ff:ff:ff:ff
    inet 192.168.5.10/24 brd 192.168.5.255 scope global eth0
    inet6 fe80::204:75ff:feea:1c7b/10 scope link
3: sit0@NONE: <NOARP> mtu 1480 qdisc noop
    link/sit 0.0.0.0 brd 0.0.0.0
#
```

Dans notre cas, outre l'interface en boucle, nous sommes reliés au réseau par une carte réseau Ethernet. L'adresse qui nous a été attribuée (dynamiquement) est `192.168.5.10` avec un masque de réseau `255.255.255.255`. L'adresse de diffusion générale est `193.253.5.255`, ce qui pouvait se prévoir. La portée est globale (elle pourrait être `site`, `link` ou `host`).

21.1.2.4 Découverte des voisins

L'objet `neighbor` nous fournit l'analogue de la commande `arp`. Dans notre cas :

```
# ip neighbor show
ip neighbor show
192.168.5.1 dev eth0 lladdr 00:60:4c:5b:f0:e8 nud stale
#
```

nous obtenons l'adresse du routeur.

21.1.2.5 Manipulation des tables de routage

L'ajout ou le retrait d'une entrée dans les tables de routage, ainsi que sa consultation, s'effectue à l'aide de l'objet `route`.

La consultation :

```
# ip route show
192.168.5.0/24 dev eth0  proto kernel  scope link  src 192.168.5.10
default via 192.168.5.1 dev eth0
#
```

nous montre qu'il y a deux entrées dans la *table de routage principale* :

- L'adresse `193.168.5.0` passe par l'interface réseau `eth0`, l'adresse source (à placer dans le paquet) étant `192.168.5.10`.
- Toutes les autres adresses (non locales) commenceront par être redirigées vers l'adresse `192.168.5.1` (qui est le routeur) en passant par l'interface réseau `eth0`.

Comme on le voit, on doit installer une route par défaut. Ceci permet de savoir comment acheminer les paquets dont l'adresse de destination ne se trouve pas dans les tables de routage. Il s'agit de l'adresse d'un routeur. On peut l'ajouter avec la commande suivante :

```
# ip route add -net 0.0.0.0 netmask 0.0.0.0 gw 192.168.5.1
```

ou :

```
# ip route add default gw 192.168.5.1
```

où le mot clé `default` signifie :

```
-net 0.0.0.0 netmask 0.0.0.0
```

On a également besoin de spécifier la route par défaut, par exemple :

```
# ip route 0.0.0.0 0.0.0.0 192.168.5.1
```

Remarquons qu'assigner une adresse IP à une interface place automatiquement la route appropriée dans la table de routage. La consultation de la *table de routage locale* nous montre les adresses de l'ordinateur hôte :

```
# ip route list table local
broadcast 192.168.5.0 dev eth0  proto kernel  scope link  src 192.168.5.10
broadcast 127.255.255.255 dev lo  proto kernel  scope link  src 127.0.0.1
broadcast 192.168.5.255 dev eth0  proto kernel  scope link  src 192.168.5.10
local 192.168.5.10 dev eth0  proto kernel  scope host  src 192.168.5.10
broadcast 127.0.0.0 dev lo  proto kernel  scope link  src 127.0.0.1
local 127.0.0.1 dev lo  proto kernel  scope host  src 127.0.0.1
local 127.0.0.0/8 dev lo  proto kernel  scope host  src 127.0.0.1
#
```

Il s'agit de 192.168.5.10.

21.2 Structure des tables de routage

La table de redirection est appelée **base de données de redirection**, ou **FIB** pour l'anglais *Forwarding Information Base*, sous Linux. Elle est constituée de tables de routage et de règles de routage dans le cas du routage dynamique. Dans le cas du routage statique, on n'a pas besoin de règles de routage.

21.2.1 Tables de routage et sources Linux

21.2.1.1 Configuration du noyau Linux

Le code source du routage dépend fortement d'un certain nombre de paramètres pouvant être définis lors de la configuration du noyau avant compilation du code source. Ces paramètres figurent à la rubrique `Networking options`. Nous allons nous intéresser ici à ceux qui permettent le routage statique :

- `CONFIG_INET` ("*TCP/IP networking*") est requis par quelques-unes de ces options.
- `CONFIG_NETLINK` ("*Kernel/user netlink socket*") : cette option n'agit pas directement sur le routage mais active l'**interface Netlink** bidirectionnelle entre le noyau et l'espace d'adresses utilisateur. En effet l'interface des sockets eut un tel succès qu'elle n'a pas seulement été utilisée pour le réseau proprement dit. Nous avons déjà vu qu'elle est également utilisée pour la commande `lpr` d'impression. Alexey Kuznetsov a conçu une famille de protocoles, `PF_NETLINK`, destinée à communiquer avec les différents domaines du noyau. Cette option nous permet, pour ce qui nous intéresse, l'utilisation de l'option suivante :
  - `CONFIG_RTNETLINK` ("*Routing messages*") : Les règles et tables de routage peuvent être modifiées en utilisant le "protocole" `NETLINK_ROUTE` par le biais de sockets appartenant à la famille de protocoles `PF_NETLINK`. Cette interface est exploitée par l'utilitaire de configuration `ip`.
- `CONFIG_IP_ADVANCED_ROUTER` ("*IP : advanced router*") : Cette option n'a pas d'effet immédiat mais représente un commutateur destiné à sélectionner un certain nombre d'options supplémentaires permettant de contrôler plus aisément la procédure de routage. Les options `CONFIG_NETLINK` et `CONFIG_RTNETLINK` sont activées automatiquement en sélectionnant cette option.
  - `CONFIG_IP_MULTIPLE_TABLES` ("*IP : policy routing*") : Cette option permet de lier le fichier `fib_rules.o` au noyau. Elle active le routage dynamique à base de règles. Lorsque cette option n'est pas activée, ce que nous supposerons dans ce livre non consacré aux routeurs, seules les deux tables de routage `local` et `main` sont créées par le noyau, la recherche dans ces tables de routage intervenant dans cet ordre.

21.2.1.2 Fichiers source

L'implémentation du traitement des règles pour le routage dynamique est intégralement contenue dans le fichier `fib_rules.c`, qui ne nous intéressera pas dans cet ouvrage. Dans le cas où le routage dynamique n'a pas été activé (autrement dit lorsque l'option `CONFIG_IP_MULTIPLE_TABLES` de configuration du noyau n'a pas été choisie), le routage statique (utilisation de deux tables de routage, `local` et `main`, seulement) est intégralement assurée par les fonctions en lignes du fichier `ip_fib.h`.

21.2.2 Description de la structure des tables de routage

Les tables de routage sont implémentées dans le noyau Linux par des structures de données qui peuvent paraître d'une grande complexité, dans lesquelles les entrées sont gérées à l'aide d'un certain nombre de tables de hachage, comme le montre la figure 21-1 ([WPRMB–02], p.366) :

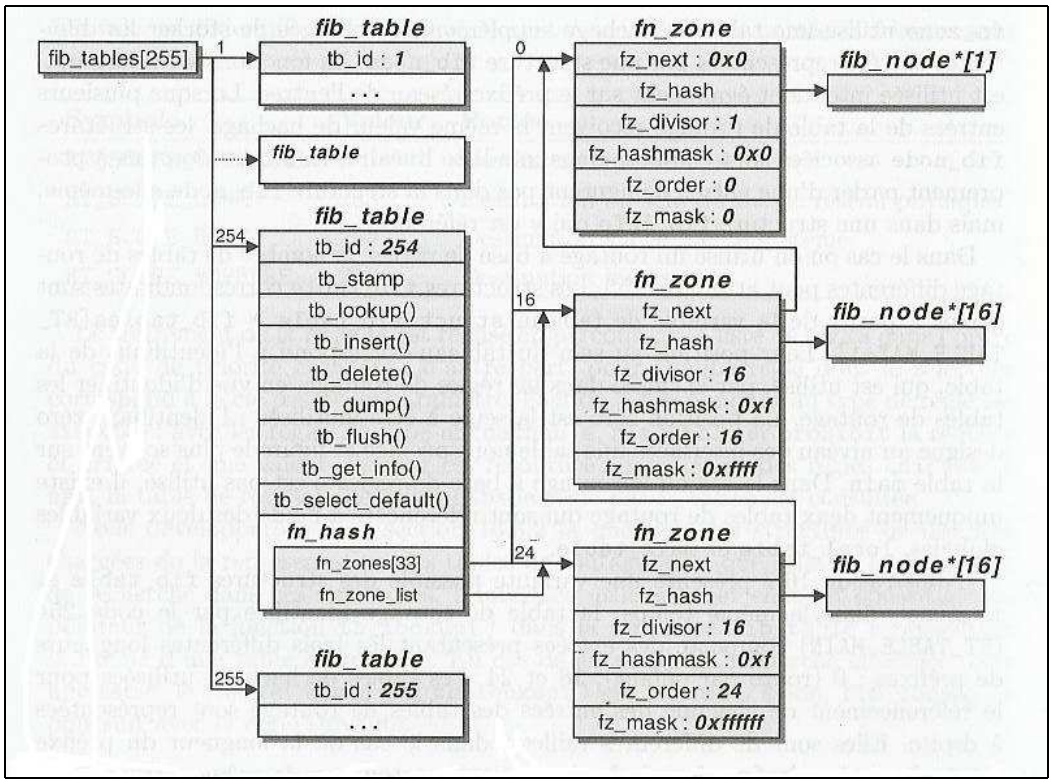


FIG. 21.1 – *Table de routage*

- La **base d'une table de routage** est formée d'une structure `fib_table`.
- Celle-ci comporte trente-trois **zones** : toutes les entrées d'une table de routage ayant une longueur de préfixe CIDR identique sont associées à une structure `fn_zone`, une pour chaque longueur de préfixe possibles (de 0 à 32 bits).
- La structure `fn_zone` utilise une table de hachage chargée de stocker les différentes entrées, ou **nœuds** représentés par une structure `fib_node`.
- Lorsque plusieurs entrées de la table de routage reçoivent la même valeur de hachage, les structures `fib_node` associées sont chaînées dans une liste linéaire.
- Enfin, les données à proprement parler d'une entrée ne figurent pas dans la structure `fib_node` elle-même, mais dans une structure `fib_info` qui y est référencée.

21.2.2.1 Base d'une table de routage

Code Linux 2.6.10

Le type `fib_table` de la base d'une table de routage est défini dans le fichier `linux/include/net/ip_fib.h` :

```
122 struct fib_table
```

```
123 {
124         unsigned char   tb_id;
125         unsigned        tb_stamp;
126         int             (*tb_lookup)(struct fib_table *tb, const struct flowi *flp,
                                         struct fib_result *res);
127         int             (*tb_insert)(struct fib_table *table, struct rtmsg *r,
128                                      struct kern_rta *rta, struct nlmsghdr *n,
129                                      struct netlink_skb_parms *req);
130         int             (*tb_delete)(struct fib_table *table, struct rtmsg *r,
131                                      struct kern_rta *rta, struct nlmsghdr *n,
132                                      struct netlink_skb_parms *req);
133         int             (*tb_dump)(struct fib_table *table, struct sk_buff *skb,
134                                      struct netlink_callback *cb);
135         int             (*tb_flush)(struct fib_table *table);
136         void            (*tb_select_default)(struct fib_table *table,
137                                              const struct flowi *flp, struct fib_result *res);
138
139         unsigned char   tb_data[0];
140 };
```

qui comprend un numéro de table (`tb_id`), une estampille temporelle (`tb_stamp`) spécifiant la durée de vie de la table de routage (sans intérêt pour les tables statiques et de toutes façons non utilisée) et six méthodes formant l'interface avec la gestion des entrées mémorisées dans la table :

- `tb_lookup()` permet la consultation dans la table.
- `tb_insert()` est chargée de l'insertion d'une entrée.
- `tb_delete()` est chargée de la suppression d'une entrée, ou plus exactement de la marquer comme supprimée.
- `tb_flush()` vide toutes les entrées de la table qui ont été marquées comme étant supprimées. Cette opération n'est effectuée que de temps en temps, et non à chaque fois qu'une entrée est marquée comme supprimée, car elle prend beaucoup de temps.
- `tb_dump()` est chargée de la sortie des entrées *via* Netlink RT.
- `tb_select_default()` est chargée de choisir parmi plusieurs routes par défaut présentes.

Le champ virtuel `tb_data` spécifie que les données se trouvent immédiatement après les champs précédents en mémoire vive. Il n'est utilisé que dans le fichier `linux/net/ipv4/fib_hash.c` et fait alors référence à une instance de la structure `fn_hash`. Celle-ci est définie dans le fichier `linux/net/ipv4/fib_hash.c` :

Code Linux 2.6.10

```
75 struct fn_hash {
76         struct fn_zone  *fn_zones[33];
77         struct fn_zone  *fn_zone_list;
78 };
```

dont la signification des champs est la suivante :

- Le tableau `fn_zones[]` spécifie l'adresse de chacune des zones (dont beaucoup auront la valeur `NULL`) : `fn_zones[0]` celle de la zone de masque réseau `0x0000`, ... , `fn_zones[32]` celle de la zone de masque `0xFFFF`.
- Le champ `fn_zone_list` spécifie l'adresse de la première zone non vide de la liste des zones non vides : `fn_zones[32]` si elle est non nulle, sinon `fn_zones[31]`, et ainsi de suite. On est sûr que `fn_zones[0]` est non nulle puisqu'elle contient l'adresse par défaut.

21.2.2.2 Zone de table de hachage

Code Linux 2.6.10

Le type `fn_zone` des zones est défini dans le fichier `linux/net/ipv4/fib_hash.c`:

```
57 struct fn_zone {
58         struct fn_zone          *fz_next;       /* Zone non vide suivante           */
59         struct hlist_head       *fz_hash;       /* Pointeur sur la table de hachage */
60         int                     fz_nent;        /* Nombre d'entrees                 */
61
62         int                     fz_divisor;     /* Diviseur de hachage              */
63         u32                     fz_hashmask;    /* (fz_divisor - 1)                 */
64 #define FZ_HASHMASK(fz)         ((fz)->fz_hashmask)
65
66         int                     fz_order;       /* Ordre de la zone                 */
67         u32                     fz_mask;
68 #define FZ_MASK(fz)             ((fz)->fz_mask)
69 };
70
71 /* NOTE. Sur les ordinateurs rapides l'evaluation de fz_hashmask et de fz_mask peut etre
72    meilleur marche que la consultation de la memoire. Dans ce cas les macrost FZ_* sont
       utilisees.
73  */
```

dont la signification des champs est la suivante:

- Nous venons de voir que le champ `fn_zone_list` de la structure `fn_hash` spécifie la première zone non vide. Le champ `fz_next` permet de connaître l'élément suivant dans cette liste.
- Pour une zone donnée, les items sont placés dans une table de hachage formée d'un tableau `fz_hash[]` d'instances du type `fib_node`.
- Le nombre d'entrées effectives dans cette zone est spécifiée par `fz_nent`.
- La dimension de la table de hachage est spécifiée par `fz_divisor`. Ce sera 16 pour la plupart des zones et 1 pour la première zone fn_zones[0].
- Le champ `fz_hashmark` est un masque de bits à l'aide duquel la valeur de hachage peut être déterminée. Ce sera 15 pour la plupart des zones et 0 pour la première zone.
- La longueur du préfixe est notée dans le champ `fz_order`.
- Le masque réseau associé, dont la valeur peut se déduire de celle du champ précédent $((1 << (32 - fz_order)) - 1)$, est indiqué dans le champ `fz_mask`.

21.2.2.3 Nœud de table de hachage

Code Linux 2.6.10

Le type `fib_node` d'un nœud est défini dans le fichier `linux/net/ipv4/fib_hash.c`:

```
51 struct fib_node {
52         struct hlist_node       fn_hash;
53         struct list_head        fn_alias;
54         u32                     fn_key;
55 };
```

21.2.2.4 Prochain saut

Comme nous l'avons dit, le rôle d'une table de routage est de fournir le prochain saut. La structure de données `struct fib_nh` concernant celui-ci (avec `nh` pour *next hop*) est définie un peu plus haut dans le même fichier:

Code Linux 2.6.10

```
44 struct fib_nh {
45         struct net_device       *nh_dev;
46         struct hlist_node       nh_hash;
47         struct fib_info         *nh_parent;
```

```
48         unsigned                nh_flags;
49         unsigned char           nh_scope;
50 #ifdef CONFIG_IP_ROUTE_MULTIPATH
51         int                     nh_weight;
52         int                     nh_power;
53 #endif
54 #ifdef CONFIG_NET_CLS_ROUTE
55         __u32                   nh_tclassid;
56 #endif
57         int                     nh_oif;
58         u32                     nh_gw;
59 };
```

Elle comprend l'interface de sortie à utiliser, représentée par son index (le champ `nh_oif`) et par le pointeur vers le descripteur de périphérique réseau (le champ `nh_dev`), ainsi que par l'adresse IP du routeur le plus proche (le champ `nh_gw`, avec *gw* pour *gateway*, passerelle). Dans le cas d'une liaison point à point, l'interface de sortie suffit ; dans le cas d'un réseau à diffusion, l'adresse de la passerelle est également nécessaire.

La portée `scope` indique la distance à la destination.

21.2.2.5 Structure d'information

Les informations contenues dans la FIB ne concernent pas que le prochain saut. Il y a d'autres données comme le montre le type `struct fib_info`, défini dans le fichier `linux/include/net/ip_fib.h`:

Code Linux 2.6.10

```
61 /*
62  * Cette structure contient des donnees partagees par plusieurs routes.
63  */
64
65 struct fib_info {
66         struct hlist_node       fib_hash;
67         struct hlist_node       fib_lhash;
68         int                     fib_treeref;
69         atomic_t                fib_clntref;
70         int                     fib_dead;
71         unsigned                fib_flags;
72         int                     fib_protocol;
73         u32                     fib_prefsrc;
74         u32                     fib_priority;
75         u32                     fib_metrics[RTAX_MAX];
76 #define fib_mtu fib_metrics[RTAX_MTU-1]
77 #define fib_window fib_metrics[RTAX_WINDOW-1]
78 #define fib_rtt fib_metrics[RTAX_RTT-1]
79 #define fib_advmss fib_metrics[RTAX_ADVMSS-1]
80         int                     fib_nhs;
81 #ifdef CONFIG_IP_ROUTE_MULTIPATH
82         int                     fib_power;
83 #endif
84         struct fib_nh           fib_nh[0];
85 #define fib_dev         fib_nh[0].nh_dev
86 };
```

Cette structure représente des informations relatives au résultat d'une requête FIB :

– Le champ le plus important concerne l'interface de sortie à utiliser (`fib_dev`) ainsi que, le cas échéant, le routeur le plus proche sur la route menant au système de destination. Ces informations figurent dans le champ `fib_nh`.

  Il convient de se préparer à l'existence éventuelle de plusieurs routes équivalentes aboutissant à une destination donnée et à ce qu'elles soient représentées dans la FIB : nous aurons

pour le champ `fib_nh` un tableau dont la dimension sera déclarée à zéro et pour lequel il sera réservé suffisamment d'espace pour que les différents chemins puissent y figurer.
- Le nombre de routes correspondantes sera alors enregistré dans le champ `fib_nhs`.
- La structure `fib_info` ne comporte pas de lien inverse vers son emplacement au sein des structures de données des tables de routage. Les pointeurs `fib_hash` et `fib_lhash` permettent de placer ses instances dans une table de hachage.
- Certains champs servent à choisir lorsqu'il y a plusieurs routes possibles : le protocole pour `fib_protocol`, les préférences pour `fib_prefsrc` et la priorité pour `fib_priority`.
- Les autres champs servent à la gestion interne : la référence à l'arbre `fib_treeref`, la référence au client `fib_clntref`, la vivacité `fib_dead` et les champs de drapeaux.

Code Linux 2.6.10

Les constantes symboliques `RTAX_MAX`, `RTAX_MTU`, `RTAX_WINDOW`, `RTAX_RTT` et `RTAX_ADVMSS` sont définies dans le fichier `linux/include/linux/rtnetlink.h` :

```
311 /* RTM_METRICS --- tableau de struct rtattr avec les types de RTAX_* */
312
313 enum
314 {
315         RTAX_UNSPEC,
316 #define RTAX_UNSPEC RTAX_UNSPEC
317         RTAX_LOCK,
318 #define RTAX_LOCK RTAX_LOCK
319         RTAX_MTU,
320 #define RTAX_MTU RTAX_MTU
321         RTAX_WINDOW,
322 #define RTAX_WINDOW RTAX_WINDOW
323         RTAX_RTT,
324 #define RTAX_RTT RTAX_RTT
325         RTAX_RTTVAR,
326 #define RTAX_RTTVAR RTAX_RTTVAR
327         RTAX_SSTHRESH,
328 #define RTAX_SSTHRESH RTAX_SSTHRESH
329         RTAX_CWND,
330 #define RTAX_CWND RTAX_CWND
331         RTAX_ADVMSS,
332 #define RTAX_ADVMSS RTAX_ADVMSS
333         RTAX_REORDERING,
334 #define RTAX_REORDERING RTAX_REORDERING
335         RTAX_HOPLIMIT,
336 #define RTAX_HOPLIMIT RTAX_HOPLIMIT
337         RTAX_INITCWND,
338 #define RTAX_INITCWND RTAX_INITCWND
339         RTAX_FEATURES,
340 #define RTAX_FEATURES RTAX_FEATURES
341         __RTAX_MAX
342 };
343
344 #define RTAX_MAX (__RTAX_MAX - 1)
```

21.3 Initialisation des tables de routage statiques

21.3.1 Déclaration

Code Linux 2.6.10

Comme nous l'avons déjà dit, en cas de routage statique deux tables, appelées *local* et *main* par l'utilisateur mais `ip_fib_local_table` et `ip_fib_main_table` dans le noyau, sont utilisées. Elles sont déclarées dans le fichier `linux/net/ipv4/fib_frontend.c` :

```
50 #ifndef CONFIG_IP_MULTIPLE_TABLES
```

```
51
52 #define RT_TABLE_MIN RT_TABLE_MAIN
53
54 struct fib_table *ip_fib_local_table;
55 struct fib_table *ip_fib_main_table;
```

Nous reviendrons sur RT_TABLE_MAIN ci-dessous.

21.3.2 Initialisation lors du démarrage du système

Les deux tables local et main sont initialisées lors du démarrage du système grâce à la fonction ip_fib_init(), définie dans le fichier linux/net/ipv4/fib_frontend.c: Code Linux 2.6.10

```
588 struct notifier_block fib_inetaddr_notifier = {
589         .notifier_call =fib_inetaddr_event,
590 };
591
592 struct notifier_block fib_netdev_notifier = {
593         .notifier_call =fib_netdev_event,
594 };
595
596 void __init ip_fib_init(void)
597 {
598 #ifndef CONFIG_IP_MULTIPLE_TABLES
599         ip_fib_local_table = fib_hash_init(RT_TABLE_LOCAL);
600         ip_fib_main_table  = fib_hash_init(RT_TABLE_MAIN);
601 #else
602         fib_rules_init();
603 #endif
604
605         register_netdevice_notifier(&fib_netdev_notifier);
606         register_inetaddr_notifier(&fib_inetaddr_notifier);
607 }
```

Nous allons revenir ci-dessous sur les numéros de table RT_TABLE_LOCAL et RT_TABLE_MAIN ainsi que sur la fonction fib_hash_init().

Les deux fonctions fib_inetaddr_event() et fib_netdev_event() sont définies au-dessus dans le même fichier: Code Linux 2.6.10

```
527 static int fib_inetaddr_event(struct notifier_block *this, unsigned long event, void *ptr)
528 {
529         struct in_ifaddr *ifa = (struct in_ifaddr*)ptr;
530
531         switch (event) {
532         case NETDEV_UP:
533                 fib_add_ifaddr(ifa);
534 #ifdef CONFIG_IP_ROUTE_MULTIPATH
535                 fib_sync_up(ifa->ifa_dev->dev);
536 #endif
537                 rt_cache_flush(-1);
538                 break;
539         case NETDEV_DOWN:
540                 fib_del_ifaddr(ifa);
541                 if (ifa->ifa_dev && ifa->ifa_dev->ifa_list == NULL) {
542                         /* La derniere adresse a ete retiree de cette interface.
543                            Desactiver IP.
544                          */
545                         fib_disable_ip(ifa->ifa_dev->dev, 1);
546                 } else {
547                         rt_cache_flush(-1);
548                 }
```

```
549                 break;
550         }
551         return NOTIFY_DONE;
552 }
553
554 static int fib_netdev_event(struct notifier_block *this, unsigned long event, void *ptr)
555 {
556         struct net_device *dev = ptr;
557         struct in_device *in_dev = __in_dev_get(dev);
558
559         if (event == NETDEV_UNREGISTER) {
560                 fib_disable_ip(dev, 2);
561                 return NOTIFY_DONE;
562         }
563
564         if (!in_dev)
565                 return NOTIFY_DONE;
566
567         switch (event) {
568         case NETDEV_UP:
569                 for_ifa(in_dev) {
570                         fib_add_ifaddr(ifa);
571                 } endfor_ifa(in_dev);
572 #ifdef CONFIG_IP_ROUTE_MULTIPATH
573                 fib_sync_up(dev);
574 #endif
575                 rt_cache_flush(-1);
576                 break;
577         case NETDEV_DOWN:
578                 fib_disable_ip(dev, 0);
579                 break;
580         case NETDEV_CHANGEMTU:
581         case NETDEV_CHANGE:
582                 rt_cache_flush(0);
583                 break;
584         }
585         return NOTIFY_DONE;
586 }
```

21.3.3 Numéros des tables

Les identificateurs 255 de la table locale et 254 de la table principale sont représentés par les constantes symboliques RT_TABLE_LOCAL et RT_TABLE_MAIN, définies dans le fichier linux/-include/linux/rtnetlink.h:

Code Linux 2.6.10

```
220 /* Identificateurs reserves des tables */
221
222 enum rt_class_t
223 {
224         RT_TABLE_UNSPEC=0,
225 /* Valeurs definies par l'utilisateur */
226         RT_TABLE_DEFAULT=253,
227         RT_TABLE_MAIN=254,
228         RT_TABLE_LOCAL=255,
229         __RT_TABLE_MAX
230 };
231 #define RT_TABLE_MAX (__RT_TABLE_MAX - 1)
```

21.3.4 Attribution des paramètres par défaut

La fonction `fib_hash_init()` est définie dans le fichier `linux/net/ipv4/fib_hash.c`: Code Linux 2.6.10

```
48 static kmem_cache_t *fn_hash_kmem;
49 static kmem_cache_t *fn_alias_kmem;
[...]
826 #ifdef CONFIG_IP_MULTIPLE_TABLES
827 struct fib_table * fib_hash_init(int id)
828 #else
829 struct fib_table * __init fib_hash_init(int id)
830 #endif
831 {
832         struct fib_table *tb;
833
834         if (fn_hash_kmem == NULL)
835                 fn_hash_kmem = kmem_cache_create("ip_fib_hash",
836                                                  sizeof(struct fib_node),
837                                                  0, SLAB_HWCACHE_ALIGN,
838                                                  NULL, NULL);
839
840         if (fn_alias_kmem == NULL)
841                 fn_alias_kmem = kmem_cache_create("ip_fib_alias",
842                                                   sizeof(struct fib_alias),
843                                                   0, SLAB_HWCACHE_ALIGN,
844                                                   NULL, NULL);
845
846         tb = kmalloc(sizeof(struct fib_table) + sizeof(struct fn_hash),
847                      GFP_KERNEL);
848         if (tb == NULL)
849                 return NULL;
850
851         tb->tb_id = id;
852         tb->tb_lookup = fn_hash_lookup;
853         tb->tb_insert = fn_hash_insert;
854         tb->tb_delete = fn_hash_delete;
855         tb->tb_flush = fn_hash_flush;
856         tb->tb_select_default = fn_hash_select_default;
857         tb->tb_dump = fn_hash_dump;
858         memset(tb->tb_data, 0, sizeof(struct fn_hash));
859         return tb;
860 }
```

qui renvoie la valeur `NULL` en cas d'échec ou, en cas de succès, l'adresse de la table initialisée avec la valeur passée en argument comme identificateur et les six méthodes initialisées respectivement à `fn_hash_lookup()`, `fn_hash_insert()`, `fn_hash_delete()`, `fn_hash_dump()`, `fn_hash_dump()` et à `fn_hash_select_default()` que nous étudierons au moment opportun.

21.4 Insertion d'un élément dans une table

Comme nous venons de le voir, la fonction interne, autrement dit au niveau du noyau, d'insertion d'un élément dans une table statique s'appelle `fn_hash_insert()`. Parmi les cinq arguments de cette fonction, quatre correspondent à des structures de données nouvelles. Commençons par les étudier.

21.4.1 Structures de données pour l'insertion

21.4.1.1 Attributs de routage

Code Linux 2.6.10

Le type `struct kern_rta` montre les informations que nous pouvons ajouter à la FIB. Il est défini dans le fichier `linux/include/net/ip_fib.h`:

```
23 /* ATTENTION! L'ordre de ces elements doit concorder avec l'ordre
24  *          des numeros d'attribut de RTA_* rtnetlink.
25  */
26 struct kern_rta {
27         void            *rta_dst;
28         void            *rta_src;
29         int             *rta_iif;
30         int             *rta_oif;
31         void            *rta_gw;
32         u32             *rta_priority;
33         void            *rta_prefsrc;
34         struct rtattr   *rta_mx;
35         struct rtattr   *rta_mp;
36         unsigned char   *rta_protoinfo;
37         u32             *rta_flow;
38         struct rta_cacheinfo *rta_ci;
39         struct rta_session *rta_sess;
40 };
```

Les champs concernent le routage de destination et source, les interfaces d'entrée et de sortie, le routeur `rta_gw`, la priorité, les préférences, les chemins multiples `rta_mx` et `rta_mp`, les informations sur le protocole, le flux, les informations sur le cache et sur la session.

Code Linux 2.6.10

Le type `struct rtattr` est défini dans le fichier `linux/include/linux/rtnetlink.h`:

```
96  /*
97     Structure generique pour l'encapsulation d'informations de routage optionnelles.
98     C'est une reminiscence de sockaddr, mais avec sa_family remplace
99     par le type attribut.
100  */
101
102 struct rtattr
103 {
104         unsigned short  rta_len;
105         unsigned short  rta_type;
106 };
```

Code Linux 2.6.10

Le type `struct rta_cacheinfo` est défini dans le même fichier `linux/include/linux/rtnetlink.h`:

```
295 /* RTM_CACHEINFO */
296
297 struct rta_cacheinfo
298 {
299         __u32   rta_clntref;
300         __u32   rta_lastuse;
301         __s32   rta_expires;
302         __u32   rta_error;
303         __u32   rta_used;
304
305 #define RTNETLINK_HAVE_PEERINFO 1
306         __u32   rta_id;
307         __u32   rta_ts;
308         __u32   rta_tsage;
309 };
```

Le type `struct rta_session` est également défini dans le fichier `linux/include/linux/rtnetlink.h`:

Code Linux 2.6.10

```
350 struct rta_session
351 {
352         __u8    proto;
353
354         union {
355                 struct {
356                         __u16   sport;
357                         __u16   dport;
358                 } ports;
359
360                 struct {
361                         __u8    type;
362                         __u8    code;
363                         __u16   ident;
364                 } icmpt;
365
366                 __u32           spi;
367         } u;
368 };
```

21.4.1.2 Message de routage

Le type `struct rtmsg` est défini dans le fichier `linux/include/linux/rtnetlink.h`:

Code Linux 2.6.10

```
125 /****************************************************************************
126  *              Definitions utilisees dans l'administration des tables de routage.
127  ****/
128
129 struct rtmsg
130 {
131         unsigned char           rtm_family;
132         unsigned char           rtm_dst_len;
133         unsigned char           rtm_src_len;
134         unsigned char           rtm_tos;
135
136         unsigned char           rtm_table;      /* Identificateur de table de routage     */
137         unsigned char           rtm_protocol;   /* Protocole de routage ; voir ci-dessous */
138         unsigned char           rtm_scope;      /* Voir ci-dessous */
139         unsigned char           rtm_type;       /* Voir ci-dessous */
140
141         unsigned                rtm_flags;
142 };
```

dont la signification des champs est la suivante:

- Le champ `rtm_family` spécifie la famille d'adresses.
- Les champs `rtm_dst_len` et `rtm_src_len` représentent la longueur du préfixe du sous-réseau de destination et du sous-réseau source.
- Le champ `rtm_tos` spécifie le type de service.
- Le champ `rtm_table` identifie la table de routage.
- Le champ `rtm_protocol` prend l'une des valeurs spécifiées un peu plus loin dans le même fichier:

  Code Linux 2.6.10

  ```
  168 /* rtm_protocol */
  169
  170 #define RTPROT_UNSPEC   0
  171 #define RTPROT_REDIRECT 1       /* Route installee par redirection ICMP ;
  ```

```
172                                     non utilise par IPv4 actuel          */
173 #define RTPROT_KERNEL   2       /* Route installee par le noyau         */
174 #define RTPROT_BOOT     3       /* Route installee au demarrage         */
175 #define RTPROT_STATIC   4       /* Route installee par l'administrateur */
176
177 /* Les valeurs de protocole >= RTPROT_STATIC ne sont pas interpretees par le noyau ;
178    elles sont juste passees depuis l'utilisateur et renvoyees comme ca.
179    Elle seront utilisees par les demons de routage multiple hypothetiques.
180    Noter que les valeurs de protocole devraient etre standardisees afin
181    d'eviter les conflits.
182  */
183
184 #define RTPROT_GATED    8       /* Apparemment, GateD */
185 #define RTPROT_RA       9       /* RDISC/ND router advertisements */
186 #define RTPROT_MRT      10      /* Merit MRT */
187 #define RTPROT_ZEBRA    11      /* Zebra */
188 #define RTPROT_BIRD     12      /* BIRD */
189 #define RTPROT_DNROUTED 13      /* DECnet routing daemon */
190 #define RTPROT_XORP     14      /* XORP */
```

– Le champ `rtm_scope` prend l'une des valeurs spécifiées un peu plus loin dans le même fichier :

Code Linux 2.6.10

```
192 /* rtm_scope
193
194    Il ne s'agit pas vraiment de portee mais d'une sorte de distance jusqu'a la
       destination.
195    NOWHERE est reserve pour les destinations qui n'existent pas, HOST est nos adresses
196    locales, LINK sont des destinations sur des liens directs
197    et UNIVERSE est partout dans l'univers.
198
199    Des valeurs intermediaires sont aussi possibles, par exemple on peut assigner
200    aux routes interieures une valeur comprise entre UNIVERSE et LINK.
201 */
202
203 enum rt_scope_t
204 {
205         RT_SCOPE_UNIVERSE=0,
206 /* Valeurs definies par l'utilisateur */
207         RT_SCOPE_SITE=200,
208         RT_SCOPE_LINK=253,
209         RT_SCOPE_HOST=254,
210         RT_SCOPE_NOWHERE=255
211 };
```

– Le champ `rtm_type` prend l'une des valeurs spécifiées un peu plus loin dans le même fichier :

Code Linux 2.6.10

```
144 /* rtm_type */
145
146 enum
147 {
148         RTN_UNSPEC,
149         RTN_UNICAST,            /* Passerelle ou route directe          */
150         RTN_LOCAL,              /* Accepte localement                   */
151         RTN_BROADCAST,          /* Accepte localement comme diffusion generale,
152                                    envoye comme diffusion generale      */
153         RTN_ANYCAST,            /* Accepte localement comme diffusion generale,
154                                    mais envoye comme individuel         */
155         RTN_MULTICAST,          /* Route de multidiffusion              */
156         RTN_BLACKHOLE,          /* Ecarte                               */
157         RTN_UNREACHABLE,        /* La destination n'est pas atteignable */
158         RTN_PROHIBIT,           /* Administrativement interdit          */
159         RTN_THROW,              /* Pas dans cette table                 */
```

```
160         RTN_NAT,                /* Traduire cette adresse               */
161         RTN_XRESOLVE,           /* Utiliser un resolveur externe        */
162         __RTN_MAX
163 };
164
165 #define RTN_MAX (__RTN_MAX - 1)
```

21.4.1.3 En-tête de message Netlink

Les informations de `kern_rta` sont ajoutés dans la FIB par un appel à une socket RTNetlink, soit par un démon de routage soit directement par le superutilisateur en utilisant la commande `ip`, d'où l'intérêt du champ portant sur l'en-tête de mesage Netlink.

Le type `struct nlmsghdr` est défini dans le fichier `linux/include/linux/rtnetlink.h`: Code Linux 2.6.10

```
33 struct nlmsghdr
34 {
35         __u32           nlmsg_len;      /* Longueur du message, y compris l'en-tete */
36         __u16           nlmsg_type;     /* Contenu du message */
37         __u16           nlmsg_flags;    /* Drapeaux supplementaires */
38         __u32           nlmsg_seq;      /* Numero de sequence */
39         __u32           nlmsg_pid;      /* PID du processus emetteur */
40 };
41
42 /* Valeurs des drapeaux */
43
44 #define NLM_F_REQUEST           1       /* C'est un message de requete.       */
45 #define NLM_F_MULTI             2       /* Message en plusieurs parties, termine par
                                           NLMSG_DONE */
46 #define NLM_F_ACK               4       /* Reponse par ack, avec zero ou un code d'erreur */
47 #define NLM_F_ECHO              8       /* Echo a cette requete            */
49 /* Modificateurs des requetes GET */
50 #define NLM_F_ROOT      0x100   /* specifie la racine de l'arbre   */
51 #define NLM_F_MATCH     0x200   /* renvoie toutes les concordances */
52 #define NLM_F_ATOMIC    0x400   /* GET atomique                    */
53 #define NLM_F_DUMP      (NLM_F_ROOT|NLM_F_MATCH)
54
55 /* Modificateurs des requetes NEW */
56 #define NLM_F_REPLACE   0x100   /* Surcharge l'existant       */
57 #define NLM_F_EXCL      0x200   /* Ne pas toucher s'il existe */
58 #define NLM_F_CREATE    0x400   /* Creer s'il n'existe pas    */
59 #define NLM_F_APPEND    0x800   /* Ajouter en fin de liste    */
60
61 /*
62    4.4BSD ADD           NLM_F_CREATE|NLM_F_EXCL
63    4.4BSD CHANGE        NLM_F_REPLACE
64
65    True CHANGE          NLM_F_CREATE|NLM_F_REPLACE
66    Append               NLM_F_CREATE
67    Check                NLM_F_EXCL
68  */
```

21.4.1.4 Pièces d'identité d'un utilisateur

Les pièces d'identité d'un utilisateur sont des entités du type `struct ucred` (pour *User CREDentials*), comme nous l'avons déjà vu. Ce type est défini dans le fichier en-tête `linux/include/linux/socket.h`: Code Linux 2.6.10

```
148 struct ucred {
149         __u32   pid;
150         __u32   uid;
151         __u32   gid;
```

```
152 };
```

Il comprend trois champs : un identificateur de processus, un identificateur d'utilisateur et un identificateur de groupe d'utilisateurs.

21.4.1.5 Paramètres Netlink

Code Linux 2.6.10

Le type `struct netlink_skb_parms` est défini dans le fichier `linux/include/linux/rtnetlink.h`:

```
105 struct netlink_skb_parms
106 {
107         struct ucred            creds;          /* Pieces d'identite du skb */
108         __u32                   pid;
109         __u32                   groups;
110         __u32                   dst_pid;
111         __u32                   dst_groups;
112         kernel_cap_t            eff_cap;
113 };
```

21.4.2 Fonction interne d'insertion

21.4.2.1 Vue d'ensemble sur l'insertion

Code Linux 2.6.10

La fonction `fn_hash_insert()` est définie dans le fichier `linux/net/ipv4/fib_hash.c`:

```
427 static int
428 fn_hash_insert(struct fib_table *tb, struct rtmsg *r, struct kern_rta *rta,
429                struct nlmsghdr *n, struct netlink_skb_parms *req)
430 {
431         struct fn_hash *table = (struct fn_hash *) tb->tb_data;
432         struct fib_node *new_f, *f;
433         struct fib_alias *fa, *new_fa;
434         struct fn_zone *fz;
435         struct fib_info *fi;
436         int z = r->rtm_dst_len;
437         int type = r->rtm_type;
438         u8 tos = r->rtm_tos;
439         u32 key;
440         int err;
441
442         if (z > 32)
443                 return -EINVAL;
444         fz = table->fn_zones[z];
445         if (!fz && !(fz = fn_new_zone(table, z)))
446                 return -ENOBUFS;
447
448         key = 0;
449         if (rta->rta_dst) {
450                 u32 dst;
451                 memcpy(&dst, rta->rta_dst, 4);
452                 if (dst & ~FZ_MASK(fz))
453                         return -EINVAL;
454                 key = fz_key(dst, fz);
455         }
456
457         if  ((fi = fib_create_info(r, rta, n, &err)) == NULL)
458                 return err;
459
460         if (fz->fz_nent > (fz->fz_divisor<<1) &&
461             fz->fz_divisor < FZ_MAX_DIVISOR &&
```

```
462             (z==32 || (1<<z) > fz->fz_divisor))
463                 fn_rehash_zone(fz);
464
465         f = fib_find_node(fz, key);
466         fa = fib_find_alias(f, tos, fi->fib_priority);
467
468         /* Maintenant fa, si non-NULL, pointe sur le premier alias de fib
469          * ayant les memes cles [prefixe, tos, priorite], si une telle cle existe
470          * deja ou sur le noeud avant lequel nous insererons le nouveau.
471          *
472          * Si fa est NULL, nous aurons besoin d'en allouer un nouveau et
473          * de l'inserer en tete de f.
474          *
475          * Si f est NULL, aucun noeud de fib ne concorde avec la cle de destination
476          * et nous avons besoin d'en allouer un nouveau egalement.
477          */
478
479         if (fa && fa->fa_tos == tos &&
480             fa->fa_info->fib_priority == fi->fib_priority) {
481                 struct fib_alias *fa_orig;
482
483                 err = -EEXIST;
484                 if (n->nlmsg_flags & NLM_F_EXCL)
485                         goto out;
486
487                 if (n->nlmsg_flags & NLM_F_REPLACE) {
488                         struct fib_info *fi_drop;
489                         u8 state;
490
491                         write_lock_bh(&fib_hash_lock);
492                         fi_drop = fa->fa_info;
493                         fa->fa_info = fi;
494                         fa->fa_type = type;
495                         fa->fa_scope = r->rtm_scope;
496                         state = fa->fa_state;
497                         fa->fa_state &= ~FA_S_ACCESSED;
498                         write_unlock_bh(&fib_hash_lock);
499
500                         fib_release_info(fi_drop);
501                         if (state & FA_S_ACCESSED)
502                                 rt_cache_flush(-1);
503                         return 0;
504                 }
505
506                 /* Erreur si nous trouvons une concordance parfaite qui
507                  * utilise les memes informations de portee, type et
508                  * prochain saut.
509                  */
510                 fa_orig = fa;
511                 fa = list_entry(fa->fa_list.prev, struct fib_alias, fa_list);
512                 list_for_each_entry_continue(fa, &f->fn_alias, fa_list) {
513                         if (fa->fa_tos != tos)
514                                 break;
515                         if (fa->fa_info->fib_priority != fi->fib_priority)
516                                 break;
517                         if (fa->fa_type == type &&
518                             fa->fa_scope == r->rtm_scope &&
519                             fa->fa_info == fi)
520                                 goto out;
521                 }
522                 if (!(n->nlmsg_flags & NLM_F_APPEND))
523                         fa = fa_orig;
```

```
524         }
525
526         err = -ENOENT;
527         if (!(n->nlmsg_flags&NLM_F_CREATE))
528                 goto out;
529
530         err = -ENOBUFS;
531         new_fa = kmem_cache_alloc(fn_alias_kmem, SLAB_KERNEL);
532         if (new_fa == NULL)
533                 goto out;
534
535         new_f = NULL;
536         if (!f) {
537                 new_f = kmem_cache_alloc(fn_hash_kmem, SLAB_KERNEL);
538                 if (new_f == NULL)
539                         goto out_free_new_fa;
540
541                 INIT_HLIST_NODE(&new_f->fn_hash);
542                 INIT_LIST_HEAD(&new_f->fn_alias);
543                 new_f->fn_key = key;
544                 f = new_f;
545         }
546
547         new_fa->fa_info = fi;
548         new_fa->fa_tos = tos;
549         new_fa->fa_type = type;
550         new_fa->fa_scope = r->rtm_scope;
551         new_fa->fa_state = 0;
552
553         /*
554          * Insere une nouvelle entree dans la liste.
555          */
556
557         write_lock_bh(&fib_hash_lock);
558         if (new_f)
559                 fib_insert_node(fz, new_f);
560         list_add_tail(&new_fa->fa_list,
561                  (fa ? &fa->fa_list : &f->fn_alias));
562         write_unlock_bh(&fib_hash_lock);
563
564         if (new_f)
565                 fz->fz_nent++;
566         rt_cache_flush(-1);
567
568         rtmsg_fib(RTM_NEWROUTE, f, new_fa, z, tb->tb_id, n, req);
569         return 0;
570
571 out_free_new_fa:
572         kmem_cache_free(fn_alias_kmem, new_fa);
573 out:
574         fib_release_info(fi);
575         return err;
576 }
```

Autrement dit :

- On déclare la partie de hachage d'une base de table de routage, que l'on instantie avec celle associée à la base de table de routage passée en argument.
- On déclare deux nœuds.
- On déclare deux alias.

Le type `fib_alias` est défini dans le fichier `linux/net/ipv4/fib_lookup.c`: Code Linux 2.6.10

```
8  struct fib_alias {
9          struct list_head        fa_list;
10         struct fib_info         *fa_info;
11         u8                      fa_tos;
12         u8                      fa_type;
13         u8                      fa_scope;
14         u8                      fa_state;
15 };
```

- On déclare une adresse de zone et une adresse de partie information.
- On déclare une longueur de préfixe de sous-réseau, que l'on instantie avec celle associée au message de routage passé en argument.
- On déclare un type, que l'on instantie avec celui associé au message de routage passé en argument.
- On déclare une clé et un code d'erreur.

  La **clé** de l'adresse 192.168.3.0 pour le masque 255.255.0.0 est 192.168.0.0, c'est-à-dire l'adresse obtenue en effectuant une disjonction bit à bit entre l'adresse et le masque.
- La longueur du préfixe doit être inférieure à 32. Si ce n'est pas le cas, on a terminé. On renvoie l'opposé du code d'erreur `EINVAL`.
- On instantie l'adresse de zone avec celle de la zone de la partie de hachage de la table associée à la longueur du préfixe. Si elle est nulle et qu'on ne parvient pas à créer une nouvelle zone, on renvoie l'opposé du code d'erreur `ENOBUFS`.

  La fonction `fn_new_zone()` de création d'une nouvelle zone sera étudiée ci-après.
- On initialise la clé à zéro.
- Si l'attribut de routage passé en argument comporte une destination: on copie celle-ci; si la partie sous-réseau de celle-ci est nulle, on ne peut pas aller plus loin, on renvoie donc l'opposé du code d'erreur `EINVAL`; sinon on détermine la clé à partir de l'adresse de destination et de la longueur du préfixe.

  La fonction en ligne `fz_key()` est définie dans le fichier `linux/net/ipv4/fib_hash.c`: Code Linux 2.6.10

```
90 static inline u32 fz_key(u32 dst, struct fn_zone *fz)
91 {
92         return dst & FZ_MASK(fz);
93 }
```

- On essaie de réserver de la place pour la partie information. Si on n'y parvient pas, on renvoie le code d'erreur fourni par la fonction `fib_create_info()`.

  La fonction `fib_create_info()` sera étudiée ci-dessous.
- Si le nombre d'entrées dans la zone est supérieur à la moitié du diviseur de hachage de celle-ci et d'autres conditions, on réaménage cette zone.

  La constante `FZ_MASK_DIVISOR` est définie dans le fichier `linux/net/ipv4/fib_hash.c`: Code Linux 2.6.10

```
97 #define FZ_MAX_DIVISOR ((PAGE_SIZE<<MAX_ORDER) / sizeof(struct hlist_head))
```

  La fonction `fn_rehash_zone()` sera étudiée ci-dessous.
- On cherche l'adrese de la partie information dans la zone qui correspond à la clé. Elle peut être nulle éventuellement.

  La fonction `fib_find_node()` est définie dans le fichier `linux/net/ipv4/fib_hash.c`: Code Linux 2.6.10

```
392 /* Renvoie le noeud de FZ concordant avec KEY. */
393 static struct fib_node *fib_find_node(struct fn_zone *fz, u32 key)
394 {
```

```
395         struct hlist_head *head = &fz->fz_hash[fn_hash(key, fz)];
396         struct hlist_node *node;
397         struct fib_node *f;
398
399         hlist_for_each_entry(f, node, head, fn_hash) {
400                 if (f->fn_key == key)
401                         return f;
402         }
403
404         return NULL;
405 }
```

– On cherche l'adresse de l'alias associé à cette partie information, au type de service et à la priorité. Elle peut être nulle éventuellement.

Code Linux 2.6.10

La fonction `fib_find_alias()` est définie dans le fichier `linux/net/ipv4/fib_hash.c`:

```
407 /* Renvoie le premier alias de fib concordant avec TOS dont la
408  * priorite est inferieure ou egale a PRIO.
409  */
410 static struct fib_alias *fib_find_alias(struct fib_node *fn, u8 tos, u32 prio)
411 {
412         if (fn) {
413                 struct list_head *head = &fn->fn_alias;
414                 struct fib_alias *fa;
415
416                 list_for_each_entry(fa, head, fa_list) {
417                         if (fa->fa_tos > tos)
418                                 continue;
419                         if (fa->fa_info->fib_priority >= prio ||
420                             fa->fa_tos < tos)
421                                 return fa;
422                 }
423         }
424         return NULL;
425 }
```

– Si l'adresse de l'alias est non nulle, si le type de service de celui-ci est celui obtenu à partir du message de routage et si sa priorité est celle de la partie information qui a été créée:

  – On déclare un alias originel.
  – Si le drapeau de l'en-tête de message passé en argument spécifie de ne pas toucher s'il existe, on libère la partie information et on renvoie l'opposé du code d'erreur `EEXIST`.

Code Linux 2.6.10

La fonction `fib_release_info()` est définie dans le fichier `linux/net/ipv4/fib_semantics.c`:

```
50 static rwlock_t fib_info_lock = RW_LOCK_UNLOCKED;
[...]
158 void fib_release_info(struct fib_info *fi)
159 {
160         write_lock(&fib_info_lock);
161         if (fi && --fi->fib_treeref == 0) {
162                 hlist_del(&fi->fib_hash);
163                 if (fi->fib_prefsrc)
164                         hlist_del(&fi->fib_lhash);
165                 change_nexthops(fi) {
166                         if (!nh->nh_dev)
167                                 continue;
168                         hlist_del(&nh->nh_hash);
169                 } endfor_nexthops(fi)
170                 fi->fib_dead = 1;
171                 fib_info_put(fi);
```

```
172         }
173         write_unlock(&fib_info_lock);
174 }
```

Les macros `change_nexthops()` et `endfor_nexthops()` sont définies dans le fichier `linux/net/ipv4/fib_semantics.c`:

Code Linux 2.6.10

```
72 /* Esperons que gcc l'optimisera en cas de boucle muette */
73
74 #define for_nexthops(fi) { int nhsel=0; const struct fib_nh * nh = (fi)->fib_nh; \
75 for (nhsel=0; nhsel < 1; nhsel++)
76
77 #define change_nexthops(fi) { int nhsel=0; struct fib_nh *
                                nh = (struct fib_nh*)((fi)->fib_nh); \
78 for (nhsel=0; nhsel < 1; nhsel++)
79
80 #endif /* CONFIG_IP_ROUTE_MULTIPATH */
81
82 #define endfor_nexthops(fi) }
```

La fonction `fib_info_put()` est définie dans le fichier `linux/include/net/ip_fib.h`:

Code Linux 2.6.10

```
254 static inline void fib_info_put(struct fib_info *fi)
255 {
256         if (atomic_dec_and_test(&fi->fib_clntref))
257                 free_fib_info(fi);
258 }
```

La fonction `free_fib_info()` est définie dans le fichier `linux/net/ipv4/fib_semantics.c`:

Code Linux 2.6.10

```
141 /* Libere un enregistrement d'info de saut suivant */
142
143 void free_fib_info(struct fib_info *fi)
144 {
145         if (fi->fib_dead == 0) {
146                 printk("Freeing alive fib_info %p\n", fi);
147                 return;
148         }
149         change_nexthops(fi) {
150                 if (nh->nh_dev)
151                         dev_put(nh->nh_dev);
152                 nh->nh_dev = NULL;
153         } endfor_nexthops(fi);
154         fib_info_cnt--;
155         kfree(fi);
156 }
```

– Si le drapeau de l'en-tête de message passé en argument spécifie de surcharger l'existant, on change l'alias et on renvoie 0 (lignes 487–503).

La fonction `rt_cache_flush()` est définie dans le fichier `linux/net/ipv4/route.c`:

Code Linux 2.6.0

```
585 static spinlock_t rt_flush_lock = SPIN_LOCK_UNLOCKED;
586
587 void rt_cache_flush(int delay)
588 {
589         unsigned long now = jiffies;
590         int user_mode = !in_softirq();
591
592         if (delay < 0)
```

```
593                 delay = ip_rt_min_delay;
594
595         spin_lock_bh(&rt_flush_lock);
596
597         if (del_timer(&rt_flush_timer) && delay > 0 && rt_deadline) {
598                 long tmo = (long)(rt_deadline - now);
599
600                 /* Si le minuteur de vidage est deja en action et
601                    si la requete de vidage n'est pas immediate (delay > 0):
602
603                    si le delai n'est pas atteint, prolonger le minuteur a "delay",
604                    sinon le fixer au delai.
605                  */
606
607                 if (user_mode && tmo < ip_rt_max_delay-ip_rt_min_delay)
608                         tmo = 0;
609
610                 if (delay > tmo)
611                         delay = tmo;
612         }
613
614         if (delay <= 0) {
615                 spin_unlock_bh(&rt_flush_lock);
616                 rt_run_flush(0);
617                 return;
618         }
619
620         if (rt_deadline == 0)
621                 rt_deadline = now + ip_rt_max_delay;
622
623         mod_timer(&rt_flush_timer, now+delay);
624         spin_unlock_bh(&rt_flush_lock);
625 }
```

  - Sinon on sauvegarde l'alias d'origine. On considère l'alias précédent dans la liste. On décrit les alias de la liste. Si l'un de ceux-ci concorde parfaitement (même type, même portée et même partie information), il y a une erreur. On libère la partie information et on renvoie l'opposé du code d'erreur `EEXIST`.
  - Si le drapeau de l'en-tête de message passé en argument spécifie d'ajouter en fin de liste, on revient à l'alias d'origine.
  - Si le drapeau de l'en-tête de message passé en argument spécifie de créer s'il n'existe pas, on libère la partie information et on renvoie l'opposé du code d'erreur `ENOENT`.
  - On essaie d'allouer de la mémoire vive à `new_fa`. Si on n'y parvient pas, on s'arrête en libérant la structure d'information et en renvoyant l'opposé du code d'erreur `ENOBUFS`.
  - On essaie d'allouer de la mémoire vive à `new_f`. Si on n'y parvient pas, on s'arrête en libérant la structure d'information et le nouvel alias et en renvoyant l'opposé du code d'erreur `ENOBUFS`.
  - On renseigne le champ clé de la nouvelle partie information et la partie information prend comme nouvelle valeur cette nouvelle partie information.

- On renseigne les champs du nouvel alias : partie information, type de service, type, portée et état.
- On insère cette nouvelle entrée dans la liste.

Code Linux 2.6.10

Le verrou `fib_hash_lock` est défini dans le fichier `linux/net/ipv4/fib_hash.c` :

```
95 static rwlock_t fib_hash_lock = RW_LOCK_UNLOCKED;
```

La fonction `fib_insert_node()` est défini dans le fichier `linux/net/ipv4/fib_hash.c`: Code Linux 2.6.10

```
384 /* Insere un noeud F a FZ. */
385 static inline void fib_insert_node(struct fn_zone *fz, struct fib_node *f)
386 {
387         struct hlist_head *head = &fz->fz_hash[fn_hash(f->fn_key, fz)];
388
389         hlist_add_head(&f->fn_hash, head);
390 }
```

– On incrémente le nombre d'entrées dans la table de hachage de la zone.
– On passe un message de routage et on renvoie 0.

La fonction `rtmsg_fib()` est définie dans le fichier `linux/net/ipv4/fib_hash.c`: Code Linux 2.6.10

```
799 static void rtmsg_fib(int event, struct fib_node *f, struct fib_alias *fa,
800                        int z, int tb_id,
801                        struct nlmsghdr *n, struct netlink_skb_parms *req)
802 {
803         struct sk_buff *skb;
804         u32 pid = req ? req->pid : 0;
805         int size = NLMSG_SPACE(sizeof(struct rtmsg)+256);
806
807         skb = alloc_skb(size, GFP_KERNEL);
808         if (!skb)
809                 return;
810
811         if (fib_dump_info(skb, pid, n->nlmsg_seq, event, tb_id,
812                           fa->fa_type, fa->fa_scope, &f->fn_key, z,
813                           fa->fa_tos,
814                           fa->fa_info) < 0) {
815                 kfree_skb(skb);
816                 return;
817         }
818         NETLINK_CB(skb).dst_groups = RTMGRP_IPV4_ROUTE;
819         if (n->nlmsg_flags&NLM_F_ECHO)
820                 atomic_inc(&skb->users);
821         netlink_broadcast(rtnl, skb, pid, RTMGRP_IPV4_ROUTE, GFP_KERNEL);
822         if (n->nlmsg_flags&NLM_F_ECHO)
823                 netlink_unicast(rtnl, skb, pid, MSG_DONTWAIT);
824 }
```

21.4.2.2 Première étape : création d'une nouvelle zone

La fonction `fn_new_zone()` est définie dans le fichier `linux/net/ipv4/fib_hash.c`: Code Linux 2.6.10

```
201 static struct fn_zone *
202 fn_new_zone(struct fn_hash *table, int z)
203 {
204         int i;
205         struct fn_zone *fz = kmalloc(sizeof(struct fn_zone), GFP_KERNEL);
206         if (!fz)
207                 return NULL;
208
209         memset(fz, 0, sizeof(struct fn_zone));
210         if (z) {
211                 fz->fz_divisor = 16;
212         } else {
213                 fz->fz_divisor = 1;
214         }
215         fz->fz_hashmask = (fz->fz_divisor - 1);
216         fz->fz_hash = fz_hash_alloc(fz->fz_divisor);
217         if (!fz->fz_hash) {
```

```
218                 kfree(fz);
219                 return NULL;
220         }
221         memset(fz->fz_hash, 0, fz->fz_divisor * sizeof(struct hlist_head *));
222         fz->fz_order = z;
223         fz->fz_mask = inet_make_mask(z);
224
225         /* Trouver la premiere zone non vide ayant ce masque specifique */
226         for (i=z+1; i<=32; i++)
227                 if (table->fn_zones[i])
228                         break;
229         write_lock_bh(&fib_hash_lock);
230         if (i>32) {
231                 /* Aucune avec ce masque specifique, nous sommes les premiers. */
232                 fz->fz_next = table->fn_zone_list;
233                 table->fn_zone_list = fz;
234         } else {
235                 fz->fz_next = table->fn_zones[i]->fz_next;
236                 table->fn_zones[i]->fz_next = fz;
237         }
238         table->fn_zones[z] = fz;
239         write_unlock_bh(&fib_hash_lock);
240         return fz;
241 }
```

qui n'exige pas vraiment de commentaires.

21.4.2.3 Deuxième étape : création de la partie information

Code Linux 2.6.10

La création de la partie information s'effectue à l'aide de la fonction `fib_create_info()`, définie dans le fichier `linux/net/ipv4/fib_semantics.c`:

```
572 struct fib_info *
573 fib_create_info(const struct rtmsg *r, struct kern_rta *rta,
574                 const struct nlmsghdr *nlh, int *errp)
575 {
576         int err;
577         struct fib_info *fi = NULL;
578         struct fib_info *ofi;
579 #ifdef CONFIG_IP_ROUTE_MULTIPATH
580         int nhs = 1;
581 #else
582         const int nhs = 1;
583 #endif
584
585         /* Verification rapide pour coincer les cas les plus mysterieux */
586         if (fib_props[r->rtm_type].scope > r->rtm_scope)
587                 goto err_inval;
588
589 #ifdef CONFIG_IP_ROUTE_MULTIPATH
590         if (rta->rta_mp) {
591                 nhs = fib_count_nexthops(rta->rta_mp);
592                 if (nhs == 0)
593                         goto err_inval;
594         }
595 #endif
596
597         err = -ENOBUFS;
598         if (fib_info_cnt >= fib_hash_size) {
599                 unsigned int new_size = fib_hash_size << 1;
600                 struct hlist_head *new_info_hash;
601                 struct hlist_head *new_laddrhash;
```

```
602                 unsigned int bytes;
603
604                 if (!new_size)
605                         new_size = 1;
606                 bytes = new_size * sizeof(struct hlist_head *);
607                 new_info_hash = fib_hash_alloc(bytes);
608                 new_laddrhash = fib_hash_alloc(bytes);
609                 if (!new_info_hash || !new_laddrhash) {
610                         fib_hash_free(new_info_hash, bytes);
611                         fib_hash_free(new_laddrhash, bytes);
612                 } else {
613                         memset(new_info_hash, 0, bytes);
614                         memset(new_laddrhash, 0, bytes);
615
616                         fib_hash_move(new_info_hash, new_laddrhash, new_size);
617                 }
618
619                 if (!fib_hash_size)
620                         goto failure;
621         }
622
623         fi = kmalloc(sizeof(*fi)+nhs*sizeof(struct fib_nh), GFP_KERNEL);
624         if (fi == NULL)
625                 goto failure;
626         fib_info_cnt++;
627         memset(fi, 0, sizeof(*fi)+nhs*sizeof(struct fib_nh));
628
629         fi->fib_protocol = r->rtm_protocol;
630
631         fi->fib_nhs = nhs;
632         change_nexthops(fi) {
633                 nh->nh_parent = fi;
634         } endfor_nexthops(fi)
635
636         fi->fib_flags = r->rtm_flags;
637         if (rta->rta_priority)
638                 fi->fib_priority = *rta->rta_priority;
639         if (rta->rta_mx) {
640                 int attrlen = RTA_PAYLOAD(rta->rta_mx);
641                 struct rtattr *attr = RTA_DATA(rta->rta_mx);
642
643                 while (RTA_OK(attr, attrlen)) {
644                         unsigned flavor = attr->rta_type;
645                         if (flavor) {
646                                 if (flavor > RTAX_MAX)
647                                         goto err_inval;
648                                 fi->fib_metrics[flavor-1] = *(unsigned*)RTA_DATA(attr);
649                         }
650                         attr = RTA_NEXT(attr, attrlen);
651                 }
652         }
653         if (rta->rta_prefsrc)
654                 memcpy(&fi->fib_prefsrc, rta->rta_prefsrc, 4);
655
656         if (rta->rta_mp) {
657 #ifdef CONFIG_IP_ROUTE_MULTIPATH
658                 if ((err = fib_get_nhs(fi, rta->rta_mp, r)) != 0)
659                         goto failure;
660                 if (rta->rta_oif && fi->fib_nh->nh_oif != *rta->rta_oif)
661                         goto err_inval;
662                 if (rta->rta_gw && memcmp(&fi->fib_nh->nh_gw, rta->rta_gw, 4))
663                         goto err_inval;
```

```
664 #ifdef CONFIG_NET_CLS_ROUTE
665                 if (rta->rta_flow && memcmp(&fi->fib_nh->nh_tclassid, rta->rta_flow, 4))
666                         goto err_inval;
667 #endif
668 #else
669                 goto err_inval;
670 #endif
671         } else {
672                 struct fib_nh *nh = fi->fib_nh;
673                 if (rta->rta_oif)
674                         nh->nh_oif = *rta->rta_oif;
675                 if (rta->rta_gw)
676                         memcpy(&nh->nh_gw, rta->rta_gw, 4);
677 #ifdef CONFIG_NET_CLS_ROUTE
678                 if (rta->rta_flow)
679                         memcpy(&nh->nh_tclassid, rta->rta_flow, 4);
680 #endif
681                 nh->nh_flags = r->rtm_flags;
682 #ifdef CONFIG_IP_ROUTE_MULTIPATH
683                 nh->nh_weight = 1;
684 #endif
685         }
686
687         if (fib_props[r->rtm_type].error) {
688                 if (rta->rta_gw || rta->rta_oif || rta->rta_mp)
689                         goto err_inval;
690                 goto link_it;
691         }
692
693         if (r->rtm_scope > RT_SCOPE_HOST)
694                 goto err_inval;
695
696         if (r->rtm_scope == RT_SCOPE_HOST) {
697                 struct fib_nh *nh = fi->fib_nh;
698
699                 /* L'adresse locale est ajoutee. */
700                 if (nhs != 1 || nh->nh_gw)
701                         goto err_inval;
702                 nh->nh_scope = RT_SCOPE_NOWHERE;
703                 nh->nh_dev = dev_get_by_index(fi->fib_nh->nh_oif);
704                 err = -ENODEV;
705                 if (nh->nh_dev == NULL)
706                         goto failure;
707         } else {
708                 change_nexthops(fi) {
709                         if ((err = fib_check_nh(r, fi, nh)) != 0)
710                                 goto failure;
711                 } endfor_nexthops(fi)
712         }
713
714         if (fi->fib_prefsrc) {
715                 if (r->rtm_type != RTN_LOCAL || rta->rta_dst == NULL ||
716                     memcmp(&fi->fib_prefsrc, rta->rta_dst, 4))
717                         if (inet_addr_type(fi->fib_prefsrc) != RTN_LOCAL)
718                                 goto err_inval;
719         }
720
721 link_it:
722         if ((ofi = fib_find_info(fi)) != NULL) {
723                 fi->fib_dead = 1;
724                 free_fib_info(fi);
725                 ofi->fib_treeref++;
```

```
726                 return ofi;
727         }
728
729         fi->fib_treeref++;
730         atomic_inc(&fi->fib_clntref);
731         write_lock(&fib_info_lock);
732         hlist_add_head(&fi->fib_hash,
733                        &fib_info_hash[fib_info_hashfn(fi)]);
734         if (fi->fib_prefsrc) {
735                 struct hlist_head *head;
736
737                 head = &fib_info_laddrhash[fib_laddr_hashfn(fi->fib_prefsrc)];
738                 hlist_add_head(&fi->fib_lhash, head);
739         }
740         change_nexthops(fi) {
741                 struct hlist_head *head;
742                 unsigned int hash;
743
744                 if (!nh->nh_dev)
745                         continue;
746                 hash = fib_devindex_hashfn(nh->nh_dev->ifindex);
747                 head = &fib_info_devhash[hash];
748                 hlist_add_head(&nh->nh_hash, head);
749         } endfor_nexthops(fi)
750         write_unlock(&fib_info_lock);
751         return fi;
752
753 err_inval:
754         err = -EINVAL;
755
756 failure:
757         *errp = err;
758         if (fi) {
759                 fi->fib_dead = 1;
760                 free_fib_info(fi);
761         }
762         return NULL;
763 }
```

Autrement dit :

- On déclare un code d'erreur de retour, une adresse de nouvelle structure d'information (que l'on initialise à `NULL`), une adresse d'ancienne structure d'information et une taille de saut suivant (que l'on initialise à 1).
- Si la portée spécifiée dans le message de routage passé en argument est situé en-dehors de l'amplitude permise, on positionne le code d'erreur de retour (en argument) à l'opposé du code d'erreur `EINVAL` et on renvoie `NULL`.

Le tableau `fib_props[]` est défini dans le fichier `linux/net/ipv4/fib_semantics.c`: Code Linux 2.6.10

```
85  static struct
86  {
87          int     error;
88          u8      scope;
89  } fib_props[RTA_MAX + 1] = {
90          {
91                  .error  = 0,
92                  .scope  = RT_SCOPE_NOWHERE,
93          },      /* RTN_UNSPEC */
94          {
95                  .error  = 0,
96                  .scope  = RT_SCOPE_UNIVERSE,
```

```
97          },      /* RTN_UNICAST */
98          {
99                  .error  = 0,
100                 .scope  = RT_SCOPE_HOST,
101         },      /* RTN_LOCAL */
102         {
103                 .error  = 0,
104                 .scope  = RT_SCOPE_LINK,
105         },      /* RTN_BROADCAST */
106         {
107                 .error  = 0,
108                 .scope  = RT_SCOPE_LINK,
109         },      /* RTN_ANYCAST */
110         {
111                 .error  = 0,
112                 .scope  = RT_SCOPE_UNIVERSE,
113         },      /* RTN_MULTICAST */
114         {
115                 .error  = -EINVAL,
116                 .scope  = RT_SCOPE_UNIVERSE,
117         },      /* RTN_BLACKHOLE */
118         {
119                 .error  = -EHOSTUNREACH,
120                 .scope  = RT_SCOPE_UNIVERSE,
121         },      /* RTN_UNREACHABLE */
122         {
123                 .error  = -EACCES,
124                 .scope  = RT_SCOPE_UNIVERSE,
125         },      /* RTN_PROHIBIT */
126         {
127                 .error  = -EAGAIN,
128                 .scope  = RT_SCOPE_UNIVERSE,
129         },      /* RTN_THROW */
130         {
131                 .error  = -EINVAL,
132                 .scope  = RT_SCOPE_NOWHERE,
133         },      /* RTN_NAT */
134         {
135                 .error  = -EINVAL,
136                 .scope  = RT_SCOPE_NOWHERE,
137         },      /* RTN_XRESOLVE */
138 };
```

Le nombre de structures d'information présentes en mémoire vive est conservé dans la variable globale `fib_info_cnt`, la taille de la table est hachage est conservée dans la variable globale `fib_hash_size`, toutes deux définies dans le fichier `linux/net/ipv4/fib_semantics.c`:

Code Linux 2.6.10

```
53 static unsigned int fib_hash_size;
54 static unsigned int fib_info_cnt;
```

– Si le nombre de parties information est supérieur à la dimension de la table de hachage, on essaie de multiplier la dimension de la table de hachage par deux. Si on n'y parvient pas, on libère la partie information et on renvoie `NULL` (lignes 599–621).

Les fonctions `fib_hash_alloc()`, `fib_hash_free()` et `fib_hash_move()` sont définies dans le fichier `linux/net/ipv4/fib_semantics.c`:

Code Linux 2.6.10

```
503 static struct hlist_head *fib_hash_alloc(int bytes)
504 {
505         if (bytes <= PAGE_SIZE)
506                 return kmalloc(bytes, GFP_KERNEL);
```

```
507         else
508                 return (struct hlist_head *)
509                         __get_free_pages(GFP_KERNEL, get_order(bytes));
510 }
511
512 static void fib_hash_free(struct hlist_head *hash, int bytes)
513 {
514         if (!hash)
515                 return;
516
517         if (bytes <= PAGE_SIZE)
518                 kfree(hash);
519         else
520                 free_pages((unsigned long) hash, get_order(bytes));
521 }
522
523 static void fib_hash_move(struct hlist_head *new_info_hash,
524                           struct hlist_head *new_laddrhash,
525                           unsigned int new_size)
526 {
527         unsigned int old_size = fib_hash_size;
528         unsigned int i;
529
530         write_lock(&fib_info_lock);
531         fib_hash_size = new_size;
532
533         for (i = 0; i < old_size; i++) {
534                 struct hlist_head *head = &fib_info_hash[i];
535                 struct hlist_node *node, *n;
536                 struct fib_info *fi;
537
538                 hlist_for_each_entry_safe(fi, node, n, head, fib_hash) {
539                         struct hlist_head *dest;
540                         unsigned int new_hash;
541
542                         hlist_del(&fi->fib_hash);
543
544                         new_hash = fib_info_hashfn(fi);
545                         dest = &new_info_hash[new_hash];
546                         hlist_add_head(&fi->fib_hash, dest);
547                 }
548         }
549         fib_info_hash = new_info_hash;
550
551         for (i = 0; i < old_size; i++) {
552                 struct hlist_head *lhead = &fib_info_laddrhash[i];
553                 struct hlist_node *node, *n;
554                 struct fib_info *fi;
555
556                 hlist_for_each_entry_safe(fi, node, n, lhead, fib_lhash) {
557                         struct hlist_head *ldest;
558                         unsigned int new_hash;
559
560                         hlist_del(&fi->fib_lhash);
561
562                         new_hash = fib_laddr_hashfn(fi->fib_prefsrc);
563                         ldest = &new_laddrhash[new_hash];
564                         hlist_add_head(&fi->fib_lhash, ldest);
565                 }
566         }
567         fib_info_laddrhash = new_laddrhash;
568
```

```
569         write_unlock(&fib_info_lock);
570 }
```

- On essaie de réserver de la place en mémoire centrale pour la nouvelle structure d'information. Si on n'y parvient pas, on positionne le code d'erreur de retour (en argument) à l'opposé du code d'erreur ENOBUFS et on renvoie NULL.
- On incrémente l'information statistique sur le nombre de parties information présentes en mémoire vive.
- On initialise à zéro cette zone de la mémoire centrale.
- On remplit les premiers champs de cette nouvelle structure d'information: protocole et drapeaux avec les champs correspondants du message de routage passé en argument, taille du prochain saut égale à 1 d'après ce que nous avons vu ci-dessus, priorité avec celle de l'attribut de routage passé en argument et ainsi de suite.

Les macros RTA_PAYLOAD(), RTA_DATA() et RTA_OK() sont définies dans le fichier linux/include/linux/rtnetlink.h:

Code Linux 2.6.10

```
108 /* Macros pour manipuler les attributs de routage */
109
110 #define RTA_ALIGNTO     4
111 #define RTA_ALIGN(len) ( ((len)+RTA_ALIGNTO-1) & ~(RTA_ALIGNTO-1) )
112 #define RTA_OK(rta,len) ((len) >= (int)sizeof(struct rtattr) && \
113                          (rta)->rta_len >= sizeof(struct rtattr) && \
114                          (rta)->rta_len <= (len))
115 #define RTA_NEXT(rta,attrlen)   ((attrlen) -= RTA_ALIGN((rta)->rta_len), \
116                                  (struct rtattr*)(((char*)(rta))
                                    + RTA_ALIGN((rta)->rta_len)))
117 #define RTA_LENGTH(len) (RTA_ALIGN(sizeof(struct rtattr)) + (len))
118 #define RTA_SPACE(len)  RTA_ALIGN(RTA_LENGTH(len))
119 #define RTA_DATA(rta)   ((void*)(((char*)(rta)) + RTA_LENGTH(0)))
120 #define RTA_PAYLOAD(rta) ((int)((rta)->rta_len) - RTA_LENGTH(0))
```

- On insère la nouvelle structure d'information dans la liste (à partir de la ligne 721) et on en renvoie l'adresse.

21.4.2.4 Troisième étape: réorganisation d'une zone

Code Linux 2.6.10

La fonction fn_rehash_zone() est définie dans le fichier linux/net/ipv4/fib_hash.c:

```
143 static void fn_rehash_zone(struct fn_zone *fz)
144 {
145         struct hlist_head *ht, *old_ht;
146         int old_divisor, new_divisor;
147         u32 new_hashmask;
148
149         old_divisor = fz->fz_divisor;
150
151         switch (old_divisor) {
152         case 16:
153                 new_divisor = 256;
154                 break;
155         case 256:
156                 new_divisor = 1024;
157                 break;
158         default:
159                 if ((old_divisor << 1) > FZ_MAX_DIVISOR) {
160                         printk(KERN_CRIT "route.c: bad divisor %d!\n", old_divisor);
161                         return;
162                 }
163                 new_divisor = (old_divisor << 1);
```

```
164                 break;
165         }
166
167         new_hashmask = (new_divisor - 1);
168
169 #if RT_CACHE_DEBUG >= 2
170         printk("fn_rehash_zone: hash for zone %d grows from %d\n", fz->fz_order, old_divisor);
171 #endif
172
173         ht = fz_hash_alloc(new_divisor);
174
175         if (ht) {
176                 memset(ht, 0, new_divisor * sizeof(struct hlist_head));
177
178                 write_lock_bh(&fib_hash_lock);
179                 old_ht = fz->fz_hash;
180                 fz->fz_hash = ht;
181                 fz->fz_hashmask = new_hashmask;
182                 fz->fz_divisor = new_divisor;
183                 fn_rebuild_zone(fz, old_ht, old_divisor);
184                 write_unlock_bh(&fib_hash_lock);
185
186                 fz_hash_free(old_ht, old_divisor);
187         }
188 }
```

sur laquelle il n'y a pas grand chose à dire.

21.5 Fonction interne de retrait d'une entrée

La fonction `fn_hash_delete()` est définie dans le fichier `linux/net/ipv4/fib_hash.c`: Code Linux 2.6.10

```
579 static int
580 fn_hash_delete(struct fib_table *tb, struct rtmsg *r, struct kern_rta *rta,
581                struct nlmsghdr *n, struct netlink_skb_parms *req)
582 {
583         struct fn_hash *table = (struct fn_hash*)tb->tb_data;
584         struct fib_node *f;
585         struct fib_alias *fa, *fa_to_delete;
586         int z = r->rtm_dst_len;
587         struct fn_zone *fz;
588         u32 key;
589         u8 tos = r->rtm_tos;
590
591         if (z > 32)
592                 return -EINVAL;
593         if ((fz  = table->fn_zones[z]) == NULL)
594                 return -ESRCH;
595
596         key = 0;
597         if (rta->rta_dst) {
598                 u32 dst;
599                 memcpy(&dst, rta->rta_dst, 4);
600                 if (dst & ~FZ_MASK(fz))
601                         return -EINVAL;
602                 key = fz_key(dst, fz);
603         }
604
605         f = fib_find_node(fz, key);
606         fa = fib_find_alias(f, tos, 0);
607         if (!fa)
```

```
608                 return -ESRCH;
609
610         fa_to_delete = NULL;
611         fa = list_entry(fa->fa_list.prev, struct fib_alias, fa_list);
612         list_for_each_entry_continue(fa, &f->fn_alias, fa_list) {
613                 struct fib_info *fi = fa->fa_info;
614
615                 if (fa->fa_tos != tos)
616                         break;
617
618                 if ((!r->rtm_type ||
619                      fa->fa_type == r->rtm_type) &&
620                     (r->rtm_scope == RT_SCOPE_NOWHERE ||
621                      fa->fa_scope == r->rtm_scope) &&
622                     (!r->rtm_protocol ||
623                      fi->fib_protocol == r->rtm_protocol) &&
624                     fib_nh_match(r, n, rta, fi) == 0) {
625                         fa_to_delete = fa;
626                         break;
627                 }
628         }
629
630         if (fa_to_delete) {
631                 int kill_fn;
632
633                 fa = fa_to_delete;
634                 rtmsg_fib(RTM_DELROUTE, f, fa, z, tb->tb_id, n, req);
635
636                 kill_fn = 0;
637                 write_lock_bh(&fib_hash_lock);
638                 list_del(&fa->fa_list);
639                 if (list_empty(&f->fn_alias)) {
640                         hlist_del(&f->fn_hash);
641                         kill_fn = 1;
642                 }
643                 write_unlock_bh(&fib_hash_lock);
644
645                 if (fa->fa_state & FA_S_ACCESSED)
646                         rt_cache_flush(-1);
647                 fn_free_alias(fa);
648                 if (kill_fn) {
649                         fn_free_node(f);
650                         fz->fz_nent--;
651                 }
652
653                 return 0;
654         }
655         return -ESRCH;
656 }
```

qui se comprend aisément.

21.6 Consultation d'une table

On consulte la base de données de redirection grâce à une **requête de redirection** ou **requête FIB** *via* la fonction `fib_lookup()`.

21.6.1 Structure de données pour la consultation

La fonction de consultation manipule deux nouvelles structures de données : le flux Internet générique en entrée et une structure de données pour y placer le résultat de la consultation.

21.6.1.1 Flux Internet générique

Le type `struct flowi` de flux Internet générique est défini dans le fichier `linux/include/net/flow.h`:

Code Linux 2.6.10

```
1  /*
2   *
3   *      FLUX internet generique.
4   *
5   */
6
7  #ifndef _NET_FLOW_H
8  #define _NET_FLOW_H
9
10 #include <linux/in6.h>
11 #include <asm/atomic.h>
12
13 struct flowi {
14         int     oif;
15         int     iif;
16
17         union {
18                 struct {
19                         __u32                   daddr;
20                         __u32                   saddr;
21                         __u32                   fwmark;
22                         __u8                    tos;
23                         __u8                    scope;
24                 } ip4_u;
25
26                 struct {
27                         struct in6_addr         daddr;
28                         struct in6_addr         saddr;
29                         __u32                   flowlabel;
30                 } ip6_u;
31
32                 struct {
33                         __u16                   daddr;
34                         __u16                   saddr;
35                         __u32                   fwmark;
36                         __u8                    scope;
37                 } dn_u;
38         } nl_u;
39 #define fld_dst         nl_u.dn_u.daddr
40 #define fld_src         nl_u.dn_u.saddr
41 #define fld_fwmark      nl_u.dn_u.fwmark
42 #define fld_scope       nl_u.dn_u.scope
43 #define fl6_dst         nl_u.ip6_u.daddr
44 #define fl6_src         nl_u.ip6_u.saddr
45 #define fl6_flowlabel   nl_u.ip6_u.flowlabel
```

```
46 #define fl4_dst         nl_u.ip4_u.daddr
47 #define fl4_src         nl_u.ip4_u.saddr
48 #define fl4_fwmark      nl_u.ip4_u.fwmark
49 #define fl4_tos         nl_u.ip4_u.tos
50 #define fl4_scope       nl_u.ip4_u.scope
51
52         __u8    proto;
53         __u8    flags;
54         union {
55                 struct {
56                         __u16   sport;
57                         __u16   dport;
58                 } ports;
59
60                 struct {
61                         __u8    type;
62                         __u8    code;
63                 } icmpt;
64
65                 struct {
66                         __u16   sport;
67                         __u16   dport;
68                         __u8    objnum;
69                         __u8    objnamel;    /* Pas 16 bits puisque la valeur maximum est 16 */
70                         __u8    objname[16]; /* Pas termine par un zero */
71                 } dnports;
72
73                 __u32           spi;
74         } uli_u;
75 #define fl_ip_sport     uli_u.ports.sport
76 #define fl_ip_dport     uli_u.ports.dport
77 #define fl_icmp_type    uli_u.icmpt.type
78 #define fl_icmp_code    uli_u.icmpt.code
79 #define fl_ipsec_spi    uli_u.spi
80 } __attribute__((__aligned__(BITS_PER_LONG/8)));
```

Le flux Internet est une clé comportant les adresses source `saddr` et de destination `daddr`, les index de l'interface réseau d'entrée `iif` et de sortie `oif`, le type de service `tos` et éventuellement l'identifiant `fwmark` du paquet à rediriger. L'indication concernant la portée (`scope`) peut être utilisée afin de restreindre la zone de recherche.

21.6.1.2 Structure du résultat

Code Linux 2.6.10

Le résultat de la consultation d'une table est une instance de la structure `fib_result`, définie dans le fichier `linux/include/net/ip_fib.h`:

```
93  struct fib_result {
94          unsigned char   prefixlen;
95          unsigned char   nh_sel;
96          unsigned char   type;
97          unsigned char   scope;
98          struct fib_info *fi;
99  #ifdef CONFIG_IP_MULTIPLE_TABLES
100         struct fib_rule *r;
101 #endif
102 };
```

21.6.2 Fonction interne de consultation

21.6.2.1 Description

La fonction interne de consultation :

```
int fn_hash_lookup(struct fib_table *tb, const struct flowi *flp,
                   struct fib_result *res);
```

possède deux arguments : une base de table de routage et un flux Internet. Le résultat principal est passé par adresse. Elle renvoie par ailleurs 0 si on a trouvé un résultat, 1 si on n'a rien trouvé et un code d'erreur négatif en cas de problème.

Le flux Internet fournit l'adresse IP. On commence par chercher dans la table de routage dans la zone la plus grande (32, mais en général cela commence à 24) : on masque l'adresse avec le masque correspondant (grâce à un ET bit à bit) et on cherche si le résultat est présent. Si c'est le cas, on renvoie le résultat. Sinon on passe à la zone inférieure et ainsi de suite jusqu'à ce qu'on trouve.

Remarquons qu'on trouvera toujours un résultat s'il existe une adresse par défaut : l'adresse masquée avec 0.0.0.0 donne 0.0.0.0 et donc le résultat pour l'adresse par défaut sera renvoyé.

21.6.2.2 Implémentation

La fonction `fn_hash_lookup()` est définie dans le fichier `linux/net/ipv4/fib_hash.c` : Code Linux 2.6.10

```
243 static int
244 fn_hash_lookup(struct fib_table *tb, const struct flowi *flp, struct fib_result *res)
245 {
246         int err;
247         struct fn_zone *fz;
248         struct fn_hash *t = (struct fn_hash*)tb->tb_data;
249
250         read_lock(&fib_hash_lock);
251         for (fz = t->fn_zone_list; fz; fz = fz->fz_next) {
252                 struct hlist_head *head;
253                 struct hlist_node *node;
254                 struct fib_node *f;
255                 u32 k = fz_key(flp->fl4_dst, fz);
256
257                 head = &fz->fz_hash[fn_hash(k, fz)];
258                 hlist_for_each_entry(f, node, head, fn_hash) {
259                         if (f->fn_key != k)
260                                 continue;
261
262                         err = fib_semantic_match(&f->fn_alias,
263                                                  flp, res,
264                                                  fz->fz_order);
265                         if (err <= 0)
266                                 goto out;
267                 }
268         }
269         err = 1;
270 out:
271         read_unlock(&fib_hash_lock);
272         return err;
273 }
```

Autrement dit :

- On déclare un code d'erreur de renvoi et une zone.

- On déclare une partie données de table de routage, que l'on instantie avec celle de la base de table de routage passée en argument.
- On verrouille les tables de routage.
- On parcourt la liste des zones non vides en commençant par celle de plus grande longueur de masque. Pour chacune de ces zones :
  - On détermine la clé `k` associée à l'adresse de destination et à la zone, c'est-à-dire que l'on ne conserve que les n premiers bits de l'adresse s'il s'agit de la zone de longueur n.
  - On détermine le début de la liste chaînée correspondant à l'index de hachage associé à cette clé.
  - On parcourt la liste de hachage à la recherche d'un élément dont la clé correspond. Si on en trouve un, on vérifie si les données concordent grâce à la fonction `fib_semantic_match()` étudiée dans la sous-section suivante. S'il en est bien ainsi, on déverrouille les tables de routage et on renvoie 0 ; le résultat a été placé à la bonne adresse par la fonction `fib_semantic_match()`. Si une erreur est renvoyée par la fonction `fib_semantic_match()`, on déverrouille les tables de routage et on renvoie ce code d'erreur. Si 1 est renvoyé par la fonction `fib_semantic_match()`, on continue à parcourir la liste.
- Si on ne trouve aucun élément correspondant à la clé, on déverrouille les tables de routage et on renvoie 1.

21.6.2.3 Vérification de la concordance

Code Linux 2.6.10

La fonction `fib_semantic_match()` est définie dans le fichier `linux/net/ipv4/fib_semantics.c`:

```
765 int fib_semantic_match(struct list_head *head, const struct flowi *flp,
766                         struct fib_result *res, int prefixlen)
767 {
768         struct fib_alias *fa;
769         int nh_sel = 0;
770
771         list_for_each_entry(fa, head, fa_list) {
772                 int err;
773
774                 if (fa->fa_tos &&
775                     fa->fa_tos != flp->fl4_tos)
776                         continue;
777
778                 if (fa->fa_scope < flp->fl4_scope)
779                         continue;
780
781                 fa->fa_state |= FA_S_ACCESSED;
782
783                 err = fib_props[fa->fa_type].error;
784                 if (err == 0) {
785                         struct fib_info *fi = fa->fa_info;
786
787                         if (fi->fib_flags & RTNH_F_DEAD)
788                                 continue;
789
790                         switch (fa->fa_type) {
791                         case RTN_UNICAST:
792                         case RTN_LOCAL:
```

```
793                         case RTN_BROADCAST:
794                         case RTN_ANYCAST:
795                         case RTN_MULTICAST:
796                                 for_nexthops(fi) {
797                                         if (nh->nh_flags&RTNH_F_DEAD)
798                                                 continue;
799                                         if (!flp->oif || flp->oif == nh->nh_oif)
800                                                 break;
801                                 }
802 #ifdef CONFIG_IP_ROUTE_MULTIPATH
803                                 if (nhsel < fi->fib_nhs) {
804                                         nh_sel = nhsel;
805                                         goto out_fill_res;
806                                 }
807 #else
808                                 if (nhsel < 1) {
809                                         goto out_fill_res;
810                                 }
811 #endif
812                                 endfor_nexthops(fi);
813                                 continue;
814
815                         default:
816                                 printk(KERN_DEBUG "impossible 102\n");
817                                 return -EINVAL;
818                         };
819                 }
820                 return err;
821         }
822         return 1;
823
824 out_fill_res:
825         res->prefixlen = prefixlen;
826         res->nh_sel = nh_sel;
827         res->type = fa->fa_type;
828         res->scope = fa->fa_scope;
829         res->fi = fa->fa_info;
830         atomic_inc(&res->fi->fib_clntref);
831         return 0;
832 }
```

Autrement dit :

- On parcourt la liste des alias `head` passée en argument à la recherche d'un alias dont le type de service et la portée concordent avec ceux du flux Internet passé en argument. Si on n'en trouve aucun, on renvoie 1.
- Lorsqu'on en trouve un :
  - On spécifie qu'on est en train d'y accéder grâce au champ des drapeaux.
  - On détermine le type d'"erreur" associé en consultant le tableau `fib_props[]` en utilisant le type de l'alias comme index. Si elle est non nulle, on renvoie le code d'erreur trouvé.
  - On regarde la partie information associée à cet alias.
  - Si le champ drapeaux de cette partie information dit qu'elle est décédée, on revient au parcours de la liste.
  - Si le type de l'alias est l'un des types diffusion individuelle, locale, générale, partout ou restreinte, on recherche parmi les prochains sauts de la partie information (il peut y en avoir plusieurs). Si on en trouve un, on remplit la structure résultat et on renvoie 0.

– Si le type de l'alias n'est pas l'un de ces types, on affiche un message noyau et on renvoie l'opposé du code d'erreur `EINVAL`.

21.6.3 Interface avec les fonctions de redirection

La fonction en ligne `fib_lookup()` renvoie l'entrée de la table de redirection correspondant à la clé passée en argument. La clé est passée en paramètre sous forme d'un pointeur vers une structure `flowi`. La fonction est définie, dans le cas du routage statique, comme fonction en ligne dans le fichier `linux/include/net/ip_fib.h`:

Code Linux 2.6.10

```
158 static inline int fib_lookup(const struct flowi *flp, struct fib_result *res)
159 {
160         if (ip_fib_local_table->tb_lookup(ip_fib_local_table, flp, res) &&
161             ip_fib_main_table->tb_lookup(ip_fib_main_table, flp, res))
162                 return -ENETUNREACH;
163         return 0;
164 }
```

Elle recherche d'abord dans la table de routage `local` puis, si elle n'a rien trouvé, dans la table `main` à l'aide de la fonction `fn_hash_lookup()` étudiée ci-dessus.

Si elle n'a rien trouvé, elle renvoie l'opposé du code d'erreur `ENETUNREACH`. Sinon elle place le résultat dans une structure de résultat et renvoie 0.

Chapitre 22

Cache de routage

Bien que les structures de données de la FIB autorisent une interrogation relativement rapide, le coût qu'impliquerait l'utilisation de ce genre d'interrogation pour chaque paquet IP serait globalement très élevé. C'est la raison pour laquelle le noyau Linux comporte, parallèlement à la FIB, un cadre supplémentaire chargé de mémoriser le résultat des dernières (en général 256) demandes de redirection utilisées offrant l'avantage d'être très rapide d'accès: le **cache de routage.** Lors d'une redirection, le cache de routage est toujours consulté en premier. Dans un deuxième temps, mais uniquement s'il n'existe pas d'entrée correspondant à la demande, une requête est envoyée à la FIB. Une nouvelle entrée de cache est alors immédiatement générée avec le résultat de la requête FIB.

22.1 Structure du cache de routage

Le **cache de routage** contient des informations, indépendantes de tout protocole, sur l'itinéraire du paquet, entre autres la carte réseau sur laquelle le paquet doit quitter l'ordinateur (dans le cas où il y en a plusieurs) et éventuellement un renvoi sur un en-tête MAC déjà stocké dans le cache d'en-tête physique.

22.1.1 Cache de destination

22.1.1.1 Entrée du cache de destination

Une entrée du cache de destination est une entité du type `dst_entry`. Celui-ci est défini dans le fichier en-tête `linux/include/net/dst.h`:

Code Linux 2.6.10

```
1 /*
2  * net/dst.h    Definitions du cache de destination independant du protocole.
3  *
4  * Auteurs :    Alexey Kuznetsov, <kuznet@ms2.inr.ac.ru>
5  *
6  */
7
8 #ifndef _NET_DST_H
9 #define _NET_DST_H
[...]
29 /* Chaque dst_entry a un compteur de reference et est situee dans une (des) liste(s) parent.
30  * Lorsqu'elle est retiree d'une liste parent, elle est "liberee" (dst_free).
31  * Apres cela, elle entre dans un etat letal (dst->obsolete > 0) et si son refcnt
32  * est nul, elle peut etre detruite immediatement, sinon elle est ajoutee a
33  * la liste gc et le glaneur de cellules verifie periodiquement le refcnt.
34  */
35
36 struct sk_buff;
37
38 struct dst_entry
39 {
40         struct dst_entry        *next;
41         atomic_t                __refcnt;       /* references client    */
42         int                     __use;
43         struct dst_entry        *child;
44         struct net_device       *dev;
45         int                     obsolete;
46         int                     flags;
47 #define DST_HOST                1
48 #define DST_NOXFRM              2
49 #define DST_NOPOLICY            4
50 #define DST_NOHASH              8
51         unsigned long           lastuse;
52         unsigned long           expires;
53
54         unsigned short          header_len;     /* plus d'espace est requis en tete */
55         unsigned short          trailer_len;    /* espace a reserver en queue       */
56
57         u32                     metrics[RTAX_MAX];
58         struct dst_entry        *path;
59
60         unsigned long           rate_last;      /* taux limitant pour ICMP */
61         unsigned long           rate_tokens;
62
63         int                     error;
64
65         struct neighbour        *neighbour;
```

```
66          struct hh_cache         *hh;
67          struct xfrm_state       *xfrm;
68
69          int                     (*input)(struct sk_buff*);
70          int                     (*output)(struct sk_buff*);
71
72 #ifdef CONFIG_NET_CLS_ROUTE
73          __u32                   tclassid;
74 #endif
75
76          struct  dst_ops         *ops;
77          struct rcu_head         rcu_head;
78
79          char                    info[0];
80 };
```

Il s'agit d'un mélange de champs constituant les données proprement dites et de champs de contrôle (qui correspondraient à un descripteur en d'autres circonstances). Les champs constituant les données sont les suivants :

- Le champ `dev` spécifie l'interface de sortie vers le routeur.
- Les champs `header_len` et `trailer_len` spécifient la taille de l'en-tête et du suffixe d'une trame (pour cette interface).
- L'instance de voisinage `neighbour` spécifie le prochain routeur ou l'hôte de destination dans le cas d'un sysème à diffusion. Dans le cas d'un système point à point, le champ `dev` suffit.

  Nous n'avons pas besoin de connaître le type `neighbour` pour l'instant puisque seule l'adresse intervient.
- L'entrée du cache matériel `hh` comporte un en-tête de trame qu'il ne reste plus qu'à copier, comme nous l'avons déjà vu au chapitre 14.
- Les fonctions `input()` et `output()` sont chargées de spécifier comment poursuivre le traitement respectivement lors de la réception et de l'émission d'un paquet IP adapté à l'entrée du cache.

  Dans le cas des paquets IP entrants, le pointeur `input()` fera référence à `ip_local_deliver()` pour un paquet à diffusion individuelle à adresser localement, à `ip_mr_input()` pour un paquet de multidiffusion et à `ip_error()` lorsque la redirection est impossible, la destination n'étant pas accessible. Le pointeur `output()` fera référence à `ip_output()` pour un paquet à diffusion individuelle et à `ip_mc_output()` pour un paquet de multidiffusion.
- Nous reviendrons ci-dessous sur les opérations sur les entrées du cache de destination.

Les champs de contrôle sont les suivants :

- Les entrées du cache de routage appartiennent à une liste chaînée, d'où l'intérêt du champ `next`.
- Le compteur de references `__refcnt` permet de détruire l'entrée de cache lorsque celui-ci vaut zéro.
- Le compteur d'utilisation `__use` est incrémenté chaque fois que l'entrée de cache est utilisée.
- Le champ `child` permet l'utilisation d'une autre liste chaînée.
- Le champ `obsolete` est une autre sécurité évitant de détruire l'entrée de cache par inadvertance. On ne doit la détruire que si ce champ est strictement positif.
- Le champ `flags` est un champ de bits dont quatre seulement sont utilisés, représentés par les constantes symboliques `DST_HOST`, `DST_NOXFRM`, `DST_NOPOLICY` et `DST_NOHASH`.

- Le champ `lastuse` spécifie le moment, en jiffies, auquel l'entrée de cache a été utilisée pour la dernière fois.
- Le champ `expires` spécifie le moment, en jiffies, auquel l'entrée de cache ne sera plus valable.
- La liste d'entrée de cache `path` forme le chemin.
- Les champs `rate_last` et `rate_tokens` concernent ICMP, qui ne nous intéresse pas dans cet ouvrage.
- Le champ `rcu_head` permet un appel pour une lecture–mise à jour.
- Le champ `info[]` ne représente rien, il indique simplement que les informations suivent.

22.1.1.2 Opérations sur les entrées de cache de destination

Les opérations sur les entrées du cache de destination, le champ `ops` de la structure précédente, dépendent de la famille de protocoles. Le type `struct dst_ops` est defini dans le fichier `linux/include/net/dst.h`:

Code Linux 2.6.10

```
83 struct dst_ops
84 {
85         unsigned short          family;
86         unsigned short          protocol;
87         unsigned                gc_thresh;
88
89         int                     (*gc)(void);
90         struct dst_entry *      (*check)(struct dst_entry *, __u32 cookie);
91         void                    (*destroy)(struct dst_entry *);
92         void                    (*ifdown)(struct dst_entry *, int how);
93         struct dst_entry *      (*negative_advice)(struct dst_entry *);
94         void                    (*link_failure)(struct sk_buff *);
95         void                    (*update_pmtu)(struct dst_entry *dst, u32 mtu);
96         int                     (*get_mss)(struct dst_entry *dst, u32 mtu);
97         int                     entry_size;
98
99         atomic_t                entries;
100         kmem_cache_t            *kmem_cachep;
101 };
```

Il s'agit toujours, pour les entrées IPv4, de la structure `ipv4_dst_ops` définie globalement. Celle-ci comporte un certain nombre de fonctions par l'intermédiaire desquelles les utilisateurs d'entrée de cache de routage, tels que TCP ou ARP, peuvent rendre compte du routage, par exemple au sujet de connexions annulées. Elle est définie dans le fichier `linux/net/ipv4/route.c`:

Code Linux 2.6.0

```
148 static struct dst_ops ipv4_dst_ops = {
149         .family =               AF_INET,
150         .protocol =             __constant_htons(ETH_P_IP),
151         .gc =                   rt_garbage_collect,
152         .check =                ipv4_dst_check,
153         .destroy =              ipv4_dst_destroy,
154         .ifdown =               ipv4_dst_ifdown,
155         .negative_advice =      ipv4_negative_advice,
156         .link_failure =         ipv4_link_failure,
157         .update_pmtu =          ip_rt_update_pmtu,
158         .entry_size =           sizeof(struct rtable),
159 };
```

Nous étudierons ces fonctions au fur et à mesure des besoins.

22.1.2 Entrée de cache de routage

Une entrée de cache de routage est une entité du type `struct rtable`, défini dans le fichier `linux/include/net/route.h`:

Code Linux 2.6.10

```
6   *              Definitions pour les routeurs IP.
7   *
8   * Version :    @(#)route.h     1.0.4   05/27/93
9   *
10  * Auteurs :    Ross Biro, <bir7@leland.Stanford.Edu>
11  *              Fred N. van Kempen, <waltje@uWalt.NL.Mugnet.ORG>
12  * Rectifications :
13  *              Alan Cox         :      A reformate. A ajoute ip_rt_local()
14  *              Alan Cox         :      Support pour les parametres TCP.
15  *              Alexey Kuznetsov :      Changement majeur dans le code de routage.
16  *              Mike McLagan     :      Routage par la source
17  *              Robert Olsson    :      Ajout des statistiques rt_cache
[...]
50 struct rtable
51 {
52          union
53          {
54                  struct dst_entry        dst;
55                  struct rtable           *rt_next;
56          } u;
57
58          struct in_device        *idev;
59
60          unsigned                rt_flags;
61          unsigned                rt_type;
62
63          __u32                   rt_dst; /* Destination du circuit */
64          __u32                   rt_src; /* Source du circuit      */
65          int                     rt_iif;
66
67          /* Info sur la station voisine */
68          __u32                   rt_gateway;
69
70          /* Cles de consultation du cache */
71          struct flowi            fl;
72
73          /* Informations cachees diverses */
74          __u32                   rt_spec_dst; /* Destination specifique RFC1122 */
75          struct inet_peer        *peer;       /* info sur le partenaire durable */
76 };
```

dont la signification de certains des champs est précisée ci-dessous:

- Le champ `u` permet de référencer l'élément suivant de la table, soit globalement, soit une partie seulement.
- Les champs `rt_src` et `rt_dst` spécifient l'adresse source et l'adresse de destination des paquets IP traités à l'aide de l'entrée.
- L'identifiant d'interface physique `iif` désigne soit l'interface de sortie, soit l'interface d'entrée.
- Le champ `ret_gateway` comporte soit l'adresse de destination, si l'ordinateur de destination est situé dans le réseau local dans lequel nous nous trouvons, soit du routeur le plus proche de celui-ci sinon (soit situé dans le réseau local, soit en lien direct).
- Le champ `key` est utilisé comme clé lors de la recherche dans le cadre d'un routage.

22.1.3 Cache de routage

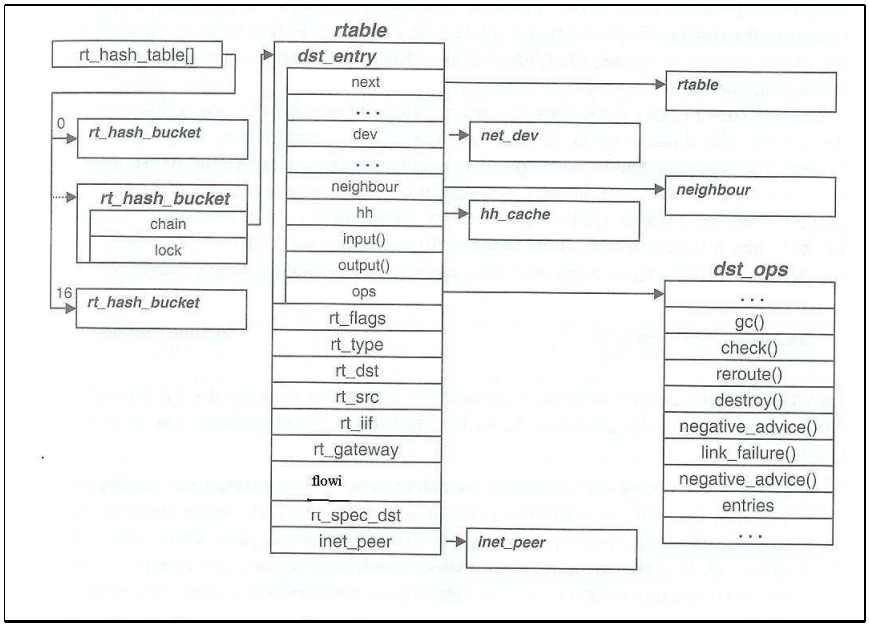


FIG. 22.1 – *Structure du cache de routage*

Comme le montre la figure 22.1, le cache de routage `rt_hash_table[]` est une table de hachage unique d'entrées de cache, présentées dans une suite linéaire pour une valeur de hachage identique.

Code Linux 2.6.10

Elle est déclarée dans le fichier `linux/net/ipv4/route.c`:

```
197 struct rt_hash_bucket {
198         struct rtable   *chain;
199         spinlock_t      lock;
200 } __attribute__((__aligned__(8)));
201
202 static struct rt_hash_bucket    *rt_hash_table;
```

Dans chaque seau de hachage, de type `rt_hash_bucket`, le champ `chain` forme le point d'ancrage d'une liste de structures `rtable` représentant les entrées du cache. L'accès à cette liste est régenté par le verrou tournant `lock`.

Code Linux 2.6.10

La dimension `rhash_entries` de ce tableau est déclarée à la fin du même fichier:

```
2696 static __initdata unsigned long rhash_entries;
```

22.2 Initialisation du cache de routage IP

L'initialisation du cache de routage IP est effectuée par la fonction `ip_rt_init()`, définie dans le fichier `linux/net/ipv4/route.c`:

Code Linux 2.6.10

```
2706 int __init ip_rt_init(void)
2707 {
2708         int i, order, goal, rc = 0;
2709
2710         rt_hash_rnd = (int) ((num_physpages ^ (num_physpages>>8)) ^
```

```
2711                            (jiffies ^ (jiffies >> 7)));
2712
2713 #ifdef CONFIG_NET_CLS_ROUTE
2714         for (order = 0;
2715              (PAGE_SIZE << order) < 256 * sizeof(struct ip_rt_acct) * NR_CPUS; order++)
2716                 /* RIEN */;
2717         ip_rt_acct = (struct ip_rt_acct *)__get_free_pages(GFP_KERNEL, order);
2718         if (!ip_rt_acct)
2719                 panic("IP: failed to allocate ip_rt_acct\n");
2720         memset(ip_rt_acct, 0, PAGE_SIZE << order);
2721 #endif
2722
2723         ipv4_dst_ops.kmem_cachep = kmem_cache_create("ip_dst_cache",
2724                                                      sizeof(struct rtable),
2725                                                      0, SLAB_HWCACHE_ALIGN,
2726                                                      NULL, NULL);
2727
2728         if (!ipv4_dst_ops.kmem_cachep)
2729                 panic("IP: failed to allocate ip_dst_cache\n");
2730
2731         goal = num_physpages >> (26 - PAGE_SHIFT);
2732         if (rhash_entries)
2733                 goal = (rhash_entries * sizeof(struct rt_hash_bucket)) >> PAGE_SHIFT;
2734         for (order = 0; (1UL << order) < goal; order++)
2735                 /* RIEN */;
2736
2737         do {
2738                 rt_hash_mask = (1UL << order) * PAGE_SIZE /
2739                         sizeof(struct rt_hash_bucket);
2740                 while (rt_hash_mask & (rt_hash_mask - 1))
2741                         rt_hash_mask--;
2742                 rt_hash_table = (struct rt_hash_bucket *)
2743                         __get_free_pages(GFP_ATOMIC, order);
2744         } while (rt_hash_table == NULL && --order > 0);
2745
2746         if (!rt_hash_table)
2747                 panic("Failed to allocate IP route cache hash table\n");
2748
2749         printk(KERN_INFO "IP: routing cache hash table of %u buckets, %ldKbytes\n",
2750                rt_hash_mask,
2751                (long) (rt_hash_mask * sizeof(struct rt_hash_bucket)) / 1024);
2752
2753         for (rt_hash_log = 0; (1 << rt_hash_log) != rt_hash_mask; rt_hash_log++)
2754                 /* RIEN */;
2755
2756         rt_hash_mask--;
2757         for (i = 0; i <= rt_hash_mask; i++) {
2758                 spin_lock_init(&rt_hash_table[i].lock);
2759                 rt_hash_table[i].chain = NULL;
2760         }
2761
2762         ipv4_dst_ops.gc_thresh = (rt_hash_mask + 1);
2763         ip_rt_max_size = (rt_hash_mask + 1) * 16;
2764
2765         rt_cache_stat = alloc_percpu(struct rt_cache_stat);
2766         if (!rt_cache_stat)
2767                 return -ENOMEM;
2768
2769         devinet_init();
2770         ip_fib_init();
2771
2772         init_timer(&rt_flush_timer);
```

```
2773         rt_flush_timer.function = rt_run_flush;
2774         init_timer(&rt_periodic_timer);
2775         rt_periodic_timer.function = rt_check_expire;
2776         init_timer(&rt_secret_timer);
2777         rt_secret_timer.function = rt_secret_rebuild;
2778
2779         /* Tous les minuteurs, demarres au depart du systeme, tendent
2780            a se synchroniser. Perturbons-les un peu.
2781          */
2782         rt_periodic_timer.expires = jiffies + net_random() % ip_rt_gc_interval +
2783                                     ip_rt_gc_interval;
2784         add_timer(&rt_periodic_timer);
2785
2786         rt_secret_timer.expires = jiffies + net_random() % ip_rt_secret_interval +
2787                 ip_rt_secret_interval;
2788         add_timer(&rt_secret_timer);
2789
2790 #ifdef CONFIG_PROC_FS
2791         {
2792         struct proc_dir_entry *rtstat_pde = NULL; /* keep gcc happy */
2793         if (!proc_net_fops_create("rt_cache", S_IRUGO, &rt_cache_seq_fops) ||
2794             !(rtstat_pde = create_proc_entry("rt_cache", S_IRUGO,
2795                                              proc_net_stat))) {
2796                 free_percpu(rt_cache_stat);
2797                 return -ENOMEM;
2798         }
2799         rtstat_pde->proc_fops = &rt_cpu_seq_fops;
2800         }
2801 #ifdef CONFIG_NET_CLS_ROUTE
2802         create_proc_read_entry("rt_acct", 0, proc_net, ip_rt_acct_read, NULL);
2803 #endif
2804 #endif
2805 #ifdef CONFIG_XFRM
2806         xfrm_init();
2807         xfrm4_init();
2808 #endif
2809         return rc;
2810 }
```

Autrement dit :

- On initialise la variable `rt_hash_rnd` (pour *RaNDom*), qui sert à pertuber les minuteurs pour qu'ils ne démarrent pas tous exactement au même moment, de façon plus ou moins aléatoire (ligne 2710–2711).

Code Linux 2.6.10

Cette variable est déclarée au début du même fichier :

```
205 static unsigned int            rt_hash_rnd;
```

- On essaie de renseigner le champ de début du cache de la structure d'opérations `ipv4_dst_ops`. Si on n'y parvient pas, on affiche un message de panique (lignes 2723–2729).
- On initialise la variable `goal` (lignes 2731–2733) et on détermine à la main le logarithme binaire `order` de celle-ci (lignes 2734–2735).
- On essaie de déterminer le masque de hachage de la table de hachage et d'obtenir de l'emplacement mémoire pour la table de hachage. Si on n'y parvient pas, on affiche un message de panique (lignes 2737–2747).

Code Linux 2.6.10

La variable `rt_hash_mask` est déclarée au début du même fichier :

```
203 static unsigned                rt_hash_mask;
```

- On affiche un message système spécifiant les caractéristiques du cache de routage IP.

– On calcule à la main le logarithme binaire du masque de hachage (lignes 2753–2754).
La variable `rt_hash_log` est déclarée au début du même fichier : Code Linux 2.6.10

```
204 static int                    rt_hash_log;
```

– On initialise la table de hachage (lignes 2756–2760).
– On renseigne le champ `gc_thresh` concernant le glaneur de cellules de `ipv4_dst_ops`.
– On initialise la taille maximum `ip_rt_max_size` du cache de routage.
Cette variable est définie dans le fichier `linux/net/ipv4/route.c` : Code Linux 2.6.10

```
113 int ip_rt_max_size;
```

– On essaie d'allouer de la place en mémoire vive pour les statistiques concernant le cache de routage sur chaque microprocesseur. Si on n'y parvient pas, on renvoie l'opposé du code d'erreur `ENOMEM` (lignes 2765–2767).
Les statistiques concernant le cache de routage sont détenues dans une structure du type `rt_cache_stat`, définie dans le fichier `linux/include/net/route.h` : Code Linux 2.6.10

```
86  struct rt_cache_stat
87  {
88          unsigned int in_hit;
89          unsigned int in_slow_tot;
90          unsigned int in_slow_mc;
91          unsigned int in_no_route;
92          unsigned int in_brd;
93          unsigned int in_martian_dst;
94          unsigned int in_martian_src;
95          unsigned int out_hit;
96          unsigned int out_slow_tot;
97          unsigned int out_slow_mc;
98          unsigned int gc_total;
99          unsigned int gc_ignored;
100         unsigned int gc_goal_miss;
101         unsigned int gc_dst_overflow;
102         unsigned int in_hlist_search;
103         unsigned int out_hlist_search;
104 };
```

Une variable du même nom est définie dans le fichier `linux/net/ipv4/route.c` : Code Linux 2.6.10

```
207 struct rt_cache_stat *rt_cache_stat;
```

– On initialise ce qui concerne les périphériques réseau pour IPv4.
La fonction `devinet_init()` est définie dans le fichier `linux/net/ipv4/devinet.c` : Code Linux 2.6.10

```
1482 void __init devinet_init(void)
1483 {
1484         register_gifconf(PF_INET, inet_gifconf);
1485         register_netdevice_notifier(&ip_netdev_notifier);
1486         rtnetlink_links[PF_INET] = inet_rtnetlink_table;
1487 #ifdef CONFIG_SYSCTL
1488         devinet_sysctl.sysctl_header =
1489                 register_sysctl_table(devinet_sysctl.devinet_root_dir, 0);
1490         devinet_sysctl_register(NULL, &ipv4_devconf_dflt);
1491 #endif
1492 }
```

– On initialise les deux tables de routage statiques, de la façon vue au chapitre 21.

– On initialise les minuteurs du cache de routage et les fonctions associées. On pertube un peu les minuteurs pour qu'ils ne ssoient pas synchronisés.

Les minuteurs `rt_flush_timer`, `rt_periodic_timer` et `rt_secret_timer` sont définis dans le fichier `linux/net/ipv4/route.c`:

Code Linux 2.6.10

```
131 static struct timer_list rt_flush_timer;
132 static struct timer_list rt_periodic_timer;
133 static struct timer_list rt_secret_timer;
```

Les fonctions `rt_run_flush()`, `rt_check_expire()` et `rt_secret_rebuild()` seront étudiées au fur et à mesure des besoins.

– On renvoie le code de retour, initialisé à 0.

22.3 Interface du cache de routage vers les fonctions de redirection

L'implémentation du cache de routage se trouve presque entièrement contenue dans le fichier `linux/net/ipv4/route.c`. Elle n'est pratiquement pas encapsulée. Ainsi il n'existe pas de fonction indépendante chargée de rechercher/créer une entrée du cache : dans les deux fonctions de l'interface principale concernant le traitement des paquets IP (à savoir `ip_route_input()` et `ip_route_output()`), cette recherche/création est réalisée directement sur la structure de données.

22.3.1 Calcul de l'index dans la table de routage

La fonction `rt_hash_code()` permet de calculer, à partir des adresses source et de destination et du type de service, l'index dans la table de hachage du cache de routage. Elle permet également de parcourir, le cas échéant, la liste ancrée dans le membre `chain`, ce qui permet de trouver éventuellement une entrée de cache identique au niveau des adresses, du type de service et éventuellement de l'identifiant `fwmark`.

Code Linux 2.6.10

Cette fonction est définie dans le fichier `linux/net/ipv4/route.c`:

```
212 static unsigned int rt_hash_code(u32 daddr, u32 saddr, u8 tos)
213 {
214         return (jhash_3words(daddr, saddr, (u32) tos, rt_hash_rnd)
215                 & rt_hash_mask);
216 }
```

22.3.2 Insertion dans la table de hachage

Les fonctions de redirection font appel à la fonction `rt_intern_hash()` afin d'insérer dans la table de hachage une structure `rtable` déjà remplie pour l'essentiel. Les nouvelles entrées sont toujours insérées en première position dans une liste de résolution de collisions éventuelles. Dans le cas où une entrée utilisant une clé identique figure déjà dans la liste, celle-ci passe en début de liste et la nouvelle entrée est rejetée.

Outre l'opération d'insertion, la fonction `rt_intern_hash()` se charge également d'obtenir le pointeur contenu dans la structure `rtable` sur une structure `neighbour` correspondant au routeur le plus proche ou au système de destination.

Cette fonction est définie dans le fichier `linux/net/ipv4/route.c`: Code Linux 2.6.10

```
776 static int rt_intern_hash(unsigned hash, struct rtable *rt, struct rtable **rp)
777 {
778         struct rtable   *rth, **rthp;
779         unsigned long   now;
780         struct rtable   *cand, **candp;
781         u32             min_score;
782         int             chain_length;
783         int attempts = !in_softirq();
784
785 restart:
786         chain_length = 0;
787         min_score = ~(u32)0;
788         cand = NULL;
789         candp = NULL;
790         now = jiffies;
791
792         rthp = &rt_hash_table[hash].chain;
793
794         spin_lock_bh(&rt_hash_table[hash].lock);
795         while ((rth = *rthp) != NULL) {
796                 if (compare_keys(&rth->fl, &rt->fl)) {
797                         /* Mettons-la en premier */
798                         *rthp = rth->u.rt_next;
799                         /*
800                          * Puisque la consultation s'effectue sans verrouillage, le retrait
801                          * doit etre visible par une autre CPU faiblement ordonnee avant
802                          * l'insertion au debut de la chaine de hachage.
803                          */
804                         rcu_assign_pointer(rth->u.rt_next,
805                                            rt_hash_table[hash].chain);
806                         /*
807                          * Puisque la consultation s'effectue sans verrouillage, la mise a
808                          * jour (ecriture) doit etre ordonnee pour la consistance sur SMP.
809                          */
810                         rcu_assign_pointer(rt_hash_table[hash].chain, rth);
811
812                         rth->u.dst.__use++;
813                         dst_hold(&rth->u.dst);
814                         rth->u.dst.lastuse = now;
815                         spin_unlock_bh(&rt_hash_table[hash].lock);
816
817                         rt_drop(rt);
818                         *rp = rth;
819                         return 0;
820                 }
821
822                 if (!atomic_read(&rth->u.dst.__refcnt)) {
823                         u32 score = rt_score(rth);
824
825                         if (score <= min_score) {
826                                 cand = rth;
827                                 candp = rthp;
828                                 min_score = score;
829                         }
830                 }
831
832                 chain_length++;
833
834                 rthp = &rth->u.rt_next;
835         }
836
```

```
837         if (cand) {
838                 /* ip_rt_gc_elasticity utilise comme longueur moyenne la longueur
839                  * de chaine. Lorsqu'elle est depassee le gc devient reellement agressif.
840                  *
841                  * La seconde limite est moins certaine. Pour l'instant elle permet
842                  * seulement 2 entrees par chaine. Nous verrons.
843                  */
844                 if (chain_length > ip_rt_gc_elasticity) {
845                         *candp = cand->u.rt_next;
846                         rt_free(cand);
847                 }
848         }
849
850         /* Essayer de lier la route a arp seulement si c'est une route de
851            sortie ou un chemin de reacheminement individuel.
852          */
853         if (rt->rt_type == RTN_UNICAST || rt->fl.iif == 0) {
854                 int err = arp_bind_neighbour(&rt->u.dst);
855                 if (err) {
856                         spin_unlock_bh(&rt_hash_table[hash].lock);
857
858                         if (err != -ENOBUFS) {
859                                 rt_drop(rt);
860                                 return err;
861                         }
862
863                         /* Les tables de voisins sont pleines et rien ne peut etre
864                            libere. Essayer de reduire le cache de routage, il est plus
865                            plausible qu'il detienne des enregistrements de voisins.
866                          */
867                         if (attempts-- > 0) {
868                                 int saved_elasticity = ip_rt_gc_elasticity;
869                                 int saved_int = ip_rt_gc_min_interval;
870                                 ip_rt_gc_elasticity     = 1;
871                                 ip_rt_gc_min_interval   = 0;
872                                 rt_garbage_collect();
873                                 ip_rt_gc_min_interval   = saved_int;
874                                 ip_rt_gc_elasticity     = saved_elasticity;
875                                 goto restart;
876                         }
877
878                         if (net_ratelimit())
879                                 printk(KERN_WARNING "Neighbour table overflow.\n");
880                         rt_drop(rt);
881                         return -ENOBUFS;
882                 }
883         }
884
885         rt->u.rt_next = rt_hash_table[hash].chain;
886 #if RT_CACHE_DEBUG >= 2
887         if (rt->u.rt_next) {
888                 struct rtable *trt;
889                 printk(KERN_DEBUG "rt_cache @%02x: %u.%u.%u.%u", hash,
890                        NIPQUAD(rt->rt_dst));
891                 for (trt = rt->u.rt_next; trt; trt = trt->u.rt_next)
892                         printk(" . %u.%u.%u.%u", NIPQUAD(trt->rt_dst));
893                 printk("\n");
894         }
895 #endif
896         rt_hash_table[hash].chain = rt;
897         spin_unlock_bh(&rt_hash_table[hash].lock);
898         *rp = rt;
```

```
899         return 0;
900 }
```

Autrement dit :

- On déclare une liste d'entrées de cache de routage, que l'on initialise avec la liste chaînée correspondant à l'index de hachage passé en argument (ligne 783).
- On verrouille cette liste chaînée.
- On parcourt cette liste chaînée à la recherche d'une entrée dont les clés concordent avec l'entrée passé en argument. Si on en trouve une (en fait elle est unique), on la place en premier dans la liste, on incrémente son nombre d'utilisateurs, on renseigne l'heure de dernière utilisation avec l'heure en cours, on déverrouille la liste chaînée, on décrémente le compteur de référence de l'entrée passée en argument et on renvoie 0 (lignes 786–827).

  La fonction `compare_keys()` est définie dans le fichier `linux/net/ipv4/route.c`: Code Linux 2.6.10

```
769 static inline int compare_keys(struct flowi *fl1, struct flowi *fl2)
770 {
771         return memcmp(&fl1->nl_u.ip4_u, &fl2->nl_u.ip4_u, sizeof(fl1->nl_u.ip4_u)) == 0
                &&
772                fl1->oif     == fl2->oif &&
773                fl1->iif     == fl2->iif;
774 }
```

  La fonction `rt_drop()` est définie dans le fichier `linux/net/ipv4/route.c`: Code Linux 2.6.10

```
452 static __inline__ void rt_drop(struct rtable *rt)
453 {
454         ip_rt_put(rt);
455         call_rcu_bh(&rt->u.dst.rcu_head, dst_rcu_free);
456 }
```

- Si on ne l'a pas trouvée dans la liste, si le type est diffusion individuelle et si l'interface d'entrée n'est pas spécifiée, on essaie de trouver un voisin (ligne 846)
-
-

Chapitre 23

Réception des paquets ordinaires sous IPv4

Les trames entrant dans l'ordinateur *via* la carte réseau et destinées à l'ordinateur stockent, comme décrit au chapitre 19, le paquet correspondant dans la file d'attente d'entrée du processeur actif. Une fois le protocole réseau déterminé dans la couche liaison (`ETH_P_IP` dans le cas qui nous intéresse), le paquet est transmis à la fonction `ip_rcv()`. Nous allons étudier l'itinéraire de ce paquet dans ce chapitre.

23.1 Première étape : tri et contrôle de l'intégrité du paquet

Nous avons vu au chapitre 19 que, lorsqu'une trame est reçue, elle est traitée au niveau de la couche de liaison puis passée à la couche supérieure, c'est-à-dire à la couche réseau. Ceci signifie qu'un tampon de socket est créé et qu'une fonction est appelée, `ip_rcv()` dans le cas de IPv4 comme l'a montré `ip_packet_type`, avec en particulier le descripteur de ce tampon de socket comme paramètre.

23.1.1 Tri et contrôle

La fonction `ip_rcv(skb, dev, pkt_type)` rejette, en premier lieu, les paquets qui ne sont pas destinés à cet ordinateur. Elle vérifie ensuite les critères d'exactitude fondamentaux du paquet : comporte-t-il au moins la taille d'un en-tête IP ? s'agit-il de la version 4 du protocole Internet ? la somme de contrôle est-elle correcte ? la longueur du paquet est-elle erronée ? Elle passe la main, enfin, à l'étape suivante.

La fonction renvoie un entier, correspondant à l'une des constantes symboliques suivantes, déjà vues au chapitre 19 : `NET_RX_SUCCESS`, `NET_RX_DROP`, `NET_RX_CN_LOW`, `NET_RX_CN_MOD`, `NET_RX_CN_HIGH` ou `NET_RX_BAD`.

Code Linux 2.6.10

La fonction `ip_rcv()` est définie dans le fichier `linux/net/ipv4/ip_input.c` :

```
357 /*
358  *      Routine principale de reception IP.
359  */
360 int ip_rcv(struct sk_buff *skb, struct net_device *dev, struct packet_type *pt)
361 {
362         struct iphdr *iph;
363
364         /* Lorsque l'interface est en mode promiscuite, ecarter tous les dechets
365          * qu'elle recoit, ne pas essayer de les analyser.
366          */
367         if (skb->pkt_type == PACKET_OTHERHOST)
368                 goto drop;
369
370         IP_INC_STATS_BH(IPSTATS_MIB_INRECEIVES);
371
372         if ((skb = skb_share_check(skb, GFP_ATOMIC)) == NULL) {
373                 IP_INC_STATS_BH(IPSTATS_MIB_INDISCARDS);
374                 goto out;
375         }
376
377         if (!pskb_may_pull(skb, sizeof(struct iphdr)))
378                 goto inhdr_error;
379
380         iph = skb->nh.iph;
381
382         /*
383          *      RFC1122: 3.1.2.2 DOIT ecarter sans faire de bruit toutes les trames IP dont la
                        somme de controle n'est pas bonne.
384          *
385          *      Le datagramme est-il acceptable ?
386          *
387          *      1.      Longueur d'au moins la taille d'un en-tete ip
388          *      2.      Version 4
389          *      3.      Somme de controle correcte. [Accelerer l'optimisation pour plus tard,
                            passer le controle pour le loopback]
390          *      4.      N'a pas une longueur bidon
```

```
391          */
392
393         if (iph->ihl < 5 || iph->version != 4)
394                 goto inhdr_error;
395
396         if (!pskb_may_pull(skb, iph->ihl*4))
397                 goto inhdr_error;
398
399         iph = skb->nh.iph;
400
401         if (ip_fast_csum((u8 *)iph, iph->ihl) != 0)
402                 goto inhdr_error;
403
404         {
405                 __u32 len = ntohs(iph->tot_len);
406                 if (skb->len < len || len < (iph->ihl<<2))
407                         goto inhdr_error;
408
409                 /* On peut avoir rembourre le tampon de notre support de transport. Maintenant
410                  * que nous savons qu'il s'agit d'IP, nous pouvons tailler a la vraie
                           longueur de la trame.
411                  * Noter que ceci signifie maintenant que skb->len contient
412                    ntohs(iph->tot_len).*/
413                 if (skb->len > len) {
414                         __pskb_trim(skb, len);
415                         if (skb->ip_summed == CHECKSUM_HW)
416                                 skb->ip_summed = CHECKSUM_NONE;
417                 }
418         }
419
420         return NF_HOOK(PF_INET, NF_IP_PRE_ROUTING, skb, dev, NULL,
421                        ip_rcv_finish);
422
423 inhdr_error:
424         IP_INC_STATS_BH(IPSTATS_MIB_INHDRERRORS);
425 drop:
426         kfree_skb(skb);
427 out:
428         return NET_RX_DROP;
429 }
```

Rappelons que dans le cas de IPv4, le paramètre `pt` a la valeur `ip_packet_type`. Décrivons maintenant le corps de la fonction :

- On déclare une adresse d'en-tête IP.
- S'il s'agit d'un paquet IP non destiné à cet ordinateur (le mode promiscuité l'a laissé passé mais il est destiné à un renifleur ; il ne doit pas être traité comme un paquet ordinaire), on libère le descripteur de tampon passé en argument et on renvoie l'état `NET_RX_DROP`. Il sera traité par une autre fonction de réception (liée au reniflage).
- On incrémente le nombre de paquets reçus dans les informations statistiques relatives à la couche IP en réception.

  La macro `IP_INC_STATS_BH()` est définie dans le fichier `linux/include/net/ip.h`: Code Linux 2.6.10

```
155 #define IP_INC_STATS_BH(field)          SNMP_INC_STATS_BH(ip_statistics, field)
```

  La variable `ip_statistics` est déclarée dans le fichier `linux/net/ipv4/ip_input.c`: Code Linux 2.6.10

```
149 /*
150  *      Gestion des statistiques SNMP
151  */
152
153 DEFINE_SNMP_STAT(struct ipstats_mib, ip_statistics);
```

Code Linux 2.6.10

La structure `ipstats_mib` et la macro `DEFINE_SNMP_STAT()` sont définies dans le fichier `linux/include/net/snmp.h`:

```
1  /*
2   *
3   *              Entrees MIB SNMP pour le sous-systeme IP.
4   *
5   *              Alan Cox <gw4pts@gw4pts.ampr.org>
6   *
7   *              Nous ne choisissons pas d'implementer SNMP dans le noyau (Ceci serait
8   *              idiot puisque SNMP est dur a l'heure actuelle). Nous avons
9   *              cependant besoin de collecter les statistiques MIB et de les exporter
10  *              hors de /proc (a un certain moment)
[...]
71 /* IPstats */
72 #define IPSTATS_MIB_MAX __IPSTATS_MIB_MAX
73 struct ipstats_mib {
74         unsigned long   mibs[IPSTATS_MIB_MAX];
75 } __SNMP_MIB_ALIGN__;
[...]
122 #define DEFINE_SNMP_STAT(type, name)    \
123          __typeof__(type) *name[2]
```

Code Linux 2.6.10

Les constantes `__IPSTATS_MIB_MAX`, `IPSTATS_MIB_INRECEIVES`, `IPSTATS_MIB_INDISCARDS` et autres sont définies dans le fichier `linux/include/linux/snmp.h`:

```
10 /* Definitions mib ipstats */
11 /*
12  * RFC 1213 :  MIB-II
13  * RFC 2011 (met a jour 1213) :  SNMPv2-MIB-IP
14  * RFC 2863 :  Interfaces Group MIB
15  * RFC 2465 :  IPv6 MIB : General Group
16  * draft-ietf-ipv6-rfc2011-update-10.txt: MIB for IP: IP Statistics Tables
17  */
18 enum
19 {
20         IPSTATS_MIB_NUM = 0,
21         IPSTATS_MIB_INRECEIVES,                 /* InReceives */
22         IPSTATS_MIB_INHDRERRORS,                /* InHdrErrors */
23         IPSTATS_MIB_INTOOBIGERRORS,             /* InTooBigErrors */
24         IPSTATS_MIB_INNOROUTES,                 /* InNoRoutes */
25         IPSTATS_MIB_INADDRERRORS,               /* InAddrErrors */
26         IPSTATS_MIB_INUNKNOWNPROTOS,            /* InUnknownProtos */
27         IPSTATS_MIB_INTRUNCATEDPKTS,            /* InTruncatedPkts */
28         IPSTATS_MIB_INDISCARDS,                 /* InDiscards */
29         IPSTATS_MIB_INDELIVERS,                 /* InDelivers */
30         IPSTATS_MIB_OUTFORWDATAGRAMS,           /* OutForwDatagrams */
31         IPSTATS_MIB_OUTREQUESTS,                /* OutRequests */
32         IPSTATS_MIB_OUTDISCARDS,                /* OutDiscards */
33         IPSTATS_MIB_OUTNOROUTES,                /* OutNoRoutes */
34         IPSTATS_MIB_REASMTIMEOUT,               /* ReasmTimeout */
35         IPSTATS_MIB_REASMREQDS,                 /* ReasmReqds */
36         IPSTATS_MIB_REASMOKS,                   /* ReasmOKs */
37         IPSTATS_MIB_REASMFAILS,                 /* ReasmFails */
38         IPSTATS_MIB_FRAGOKS,                    /* FragOKs */
39         IPSTATS_MIB_FRAGFAILS,                  /* FragFails */
40         IPSTATS_MIB_FRAGCREATES,                /* FragCreates */
41         IPSTATS_MIB_INMCASTPKTS,                /* InMcastPkts */
42         IPSTATS_MIB_OUTMCASTPKTS,               /* OutMcastPkts */
43         __IPSTATS_MIB_MAX
44 };
```

- Si le tampon est partagé, on essaie de le cloner et on travaillera désormais sur le clone. Si on ne parvient pas à le cloner, on incrémente les informations statistiques concernant les paquets détruits lors de la réception et on renvoie `NET_RX_DROP`.
- Si le tampon est de taille inférieure à celle d'un en-tête IP, on incrémente l'information statistique concernant le nombre d'erreurs sur l'en-tête des paquets IP entrants, on libère le descripteur de tampon et on renvoie `NET_RX_DROP`. Sinon on déplace le début de la zone des données du tampon d'une taille convenant à un en-tête IP. La zone des données du tampon de socket comportera donc maintenant un datagramme et non plus un paquet.
- On instantie l'adresse d'en-tête IP déclarée avec celle du champ adéquat du descripteur de tampon passé en paramètre, ce qui nous permet de nous positionner au début de l'en-tête IP du paquet contenu dans le tampon.
- En conformité avec [RFC 1122], si la longueur de cet en-tête IP est inférieure à 5 mots (soit 20 octets) ou si la version n'est pas IPv4, on incrémente l'information statistique concernant le nombre d'erreurs sur l'en-tête des paquets entrants, on libère le descripteur de tampon et on renvoie `NET_RX_DROP`.
- On vérifie que la longueur du tampon est supérieure à la longueur indiquée dans l'en-tête IP. Si ce n'est pas le cas, on incrémente l'information statistique concernant le nombre d'erreurs sur l'en-tête des paquets entrants, on libère le descripteur de tampon et on renvoie `NET_RX_DROP`.
- On déplace à nouveau le début de la zone des données du tampon pour tenir compte de la taille réelle de l'en-tête IP (options comprises) à partir de ce qui est indiqué dans l'en-tête IP.
- On vérifie la somme de contrôle de l'en-tête. Si elle ne cononcorde avec ce qui est indiqué dans l'en-tête, on incrémente l'information statistique concernant le nombre d'erreurs sur l'en-tête des paquets entrants, on libère le descripteur de tampon et on renvoie `NET_RX_DROP`.
- On déclare une longueur que l'on instantie avec la longueur totale du paquet. Si la taille du tampon est inférieure à la longueur du paquet, on incrémente l'information statistique concernant le nombre d'erreurs sur l'en-tête des paquets entrants, on libère le descripteur de tampon et on renvoie `NET_RX_DROP`.
- Si la taille du tampon est supérieure à la longueur du paquet, c'est que le paquet contenait un champ de remplissage. On déplace donc la fin de la zones des données du tampon pour retirer celle-ci du datagramme et on change le mode de vérification de la somme de contrôle.
- Le point d'ancrage `NF_IP_PRE_ROUTING` est appelé, pour traitement éventuel, avec `ip_rcv_finish()` comme fonction pour continuer après ce traitement éventuel.

  Nous allons parler des points d'ancrage dans la sous-section suivante.

23.1.2 Étape optionnelle : les points d'ancrage

L'option de compilation `NETFILTER` du noyau permet d'étendre le déroulement de certains protocoles, si cela est souhaité. Les **points d'ancrage** (*hook* en anglais) Netfilter sont placés à des points stratégiques de l'implémentation de certains protocoles et sont déployés par exemple pour les fonctions pare-feu, qualité de service ou transformation d'adresse. Un point d'ancrage Netfilter est appelé par la macro `NF_HOOK()` à laquelle sera transmise la fonction à venir après traitement de l'extension Netfilter sous la forme d'un pointeur de fonction.

Si Netfilter n'a pas été configuré, la macro veille à ce que cette fonction à venir soit immédiatement démarrée, comme le montre les lignes 147–148 de la définition de la macro `NF_HOOK()`, définie dans le fichier `linux/include/linux/netfilter.h`:

Code Linux 2.6.10

```
121 /* Activation d'un point d'ancrage ; on appelle okfn ou kfree_skb, a moins qu'un point
```

```
122    d'ancrage ne renvoie NF_STOLEN (dans un tel cas, c'est au point d'ancrage de reagir avec
123    les consequences possibles).
124
125    Renvoie -ERRNO si le paquet est ecarte. Zero signifie mise en file d'attente, vole ou
126    accepte.
127 */
128
129 /* RR :
130    > Je ne veux pas que nf_hook renvoie quelque chose parce que les gens pourraient l'oublier
131    > avec async et compter sur le fait qu'une valeur de retour signifie "le paquet etait ok".
132
133    AK :
134    Il suffit de le documenter clairement, vous pouvez alors esperer un peu de bon sens
135    de la part des codeurs du noyau :)
136 */
137
138 /* Ceci est fruste, mais les fonctions en ligne ne l'arreteront pas pour eviter l'appel de
139    fonction en routage rapide : gcc ne traite pas les fonctions en ligne (necessite des
       valeurs de pistage ?). --RR */
140 #ifdef CONFIG_NETFILTER_DEBUG
141 #define NF_HOOK(pf, hook, skb, indev, outdev, okfn)                    \
142  nf_hook_slow((pf), (hook), (skb), (indev), (outdev), (okfn), INT_MIN)
143 #define NF_HOOK_THRESH nf_hook_slow
144 #else
145 #define NF_HOOK(pf, hook, skb, indev, outdev, okfn)                    \
146 (list_empty(&nf_hooks[(pf)][(hook)])                                   \
147  ? (okfn)(skb)                                                         \
148  : nf_hook_slow((pf), (hook), (skb), (indev), (outdev), (okfn), INT_MIN))
149 #define NF_HOOK_THRESH(pf, hook, skb, indev, outdev, okfn, thresh)     \
150 (list_empty(&nf_hooks[(pf)][(hook)])                                   \
151  ? (okfn)(skb)                                                         \
152  : nf_hook_slow((pf), (hook), (skb), (indev), (outdev), (okfn), (thresh)))
153 #endif
154
155 int nf_hook_slow(int pf, unsigned int hook, struct sk_buff *skb,
156                  struct net_device *indev, struct net_device *outdev,
157                  int (*okfn)(struct sk_buff *), int thresh);
```

C'est tout ce qui nous intéressera sur les points d'ancrage pour l'instant.

23.2 Deuxième étape : routage et traitement des options

23.2.1 Vue d'ensemble

Nous avons vu ci-dessus que la fonction à venir dans le cas de notre point d'ancrage est `ip_rcv_finish()`. Celle-ci commence par déterminer la route du paquet, c'est-à-dire s'il doit être délivré localement, s'il s'agit d'une diffusion individuelle ou d'une multidiffusion, ou s'il doit être acheminé vers un autre ordinateur. Puis elle vérifie si des options sont indiquées dans l'en-tête IP et les traite le cas échéant.

Code Linux 2.6.10

Cette fonction est définie dans le fichier `linux/net/ipv4/ip_input.c`:

```
285 static inline int ip_rcv_finish(struct sk_buff *skb)
286 {
287         struct net_device *dev = skb->dev;
288         struct iphdr *iph = skb->nh.iph;
289
290         /*
291          *      Initialise le cache de circuit virtuel pour le paquet. Decrit
292          *      comment le paquet voyage dans le reseau Linux.
293          */
```

```
294         if (skb->dst == NULL) {
295                 if (ip_route_input(skb, iph->daddr, iph->saddr, iph->tos, dev))
296                         goto drop;
297         }
298
299 #ifdef CONFIG_NET_CLS_ROUTE
300         if (skb->dst->tclassid) {
301                 struct ip_rt_acct *st = ip_rt_acct + 256*smp_processor_id();
302                 u32 idx = skb->dst->tclassid;
303                 st[idx&0xFF].o_packets++;
304                 st[idx&0xFF].o_bytes+=skb->len;
305                 st[(idx>>16)&0xFF].i_packets++;
306                 st[(idx>>16)&0xFF].i_bytes+=skb->len;
307         }
308 #endif
309
310         if (iph->ihl > 5) {
311                 struct ip_options *opt;
312
313                 /* Il semble surestime, puisque toutes les options IP
314                    n'exigent pas une mutilation.
315                    Mais c'est plus facile pour l'instant, en particulier en prenant
316                    en compte le fait que cette combinaison des options IP
317                    et l'execution d'un renifleur est une condition extremement rare.
318                                                 --ANK (980813)
319                 */
320
321                 if (skb_cow(skb, skb_headroom(skb))) {
322                         IP_INC_STATS_BH(IPSTATS_MIB_INDISCARDS);
323                         goto drop;
324                 }
325                 iph = skb->nh.iph;
326
327                 if (ip_options_compile(NULL, skb))
328                         goto inhdr_error;
329
330                 opt = &(IPCB(skb)->opt);
331                 if (opt->srr) {
332                         struct in_device *in_dev = in_dev_get(dev);
333                         if (in_dev) {
334                                 if (!IN_DEV_SOURCE_ROUTE(in_dev)) {
335                                         if (IN_DEV_LOG_MARTIANS(in_dev) && net_ratelimit())
336                                                 printk(KERN_INFO
                        "source route option %u.%u.%u.%u -> %u.%u.%u.%u\n",
337                                                        NIPQUAD(iph->saddr),
                                                       NIPQUAD(iph->daddr));
338                                         in_dev_put(in_dev);
339                                         goto drop;
340                                 }
341                                 in_dev_put(in_dev);
342                         }
343                         if (ip_options_rcv_srr(skb))
344                                 goto drop;
345                 }
346         }
347
348         return dst_input(skb);
349
350 inhdr_error:
351         IP_INC_STATS_BH(IPSTATS_MIB_INHDRERRORS);
352 drop:
353         kfree_skb(skb);
```

```
354         return NET_RX_DROP;
355 }
```

Autrement dit :

- On déclare un descripteur de périphérique réseau, que l'on instantie avec celui associé au descripteur de tampon passé en argument.
- On déclare un en-tête IP, que l'on instantie avec celui associé au descripteur de tampon passé en argument.
- Si l'entrée de cache de destination `skb->dst` associée au descripteur de tampon passé en argument n'est pas définie, on essaie de la déterminer à l'aide de la fonction de redirection `ip_route_input()`, étudiée dans la sous-section suivante. Si ceci ne permet pas de lui en attribuer une, on libère le descripteur de tampon et on renvoie `NET_RX_DROP`.

  Cette entrée de cache n'est pas définie dans le cas d'un paquet provenant d'une trame, aussi fait-on appel à la fonction de redirection.
- Si la longueur de l'en-tête IP, telle qu'annoncée par celui-ci, est strictement supérieure à 5 mots, c'est que l'en-tête contient des options. On traite alors celles-ci (lignes 310–346), ce qui ne nous intéressera pas ici.
- On fait appel à la fonction chargée de poursuivre le traitement du paquet *via* la fonction `dst_input()`, étudiée dans une sous-section suivante, et on renvoie le code d'erreur fourni par celle-ci.

23.2.2 Fonction de redirection en entrée

La requête de redirection dans le traitement des paquets IP entrants s'effectue grâce à la fonction `ip_route_input()`, appelée par la fonction `ip_rcv_finish()` pour chaque paquet entrant. Elle est chargée d'identifier une entrée du cache de destination afin de déterminer la suite du parcours du paquet.

23.2.2.1 Ca de l'existence d'une entrée de cache

La fonction `ip_route_input()` prend en argument l'adresse du descripteur de tampon de socket du paquet ainsi que les adresses de destination et source du paquet, le type de service et l'adresse du descripteur de périphérique réseau chargé de la réception. Les quatre derniers arguments peuvent être déterminés à partir du premier. Cependant, comme ils doivent être à nouveau passés séparément, il est plus rapide qu'ils apparaissent en tant qu'arguments de la fonction `ip_route_input()`. Ceci permet de plus de traiter la structure `sk_buff` comme une "boîte noire" ou presque. Elle détermine l'entrée de cache adéquate si celle-ci apparaît dans le cache IP ou la suite du traitement avec `ip_route_input_mc()` ou `ip_route_input_slow()` sinon.

Code Linux 2.6.10

La fonction `ip_route_input()` est définie dans le fichier `linux/net/ipv4/route.c`:

```
1816 int ip_route_input(struct sk_buff *skb, u32 daddr, u32 saddr,
1817                     u8 tos, struct net_device *dev)
1818 {
1819         struct rtable * rth;
1820         unsigned        hash;
1821         int iif = dev->ifindex;
1822
1823         tos &= IPTOS_RT_MASK;
1824         hash = rt_hash_code(daddr, saddr ^ (iif << 5), tos);
1825
1826         rcu_read_lock();
1827         for (rth = rcu_dereference(rt_hash_table[hash].chain); rth;
```

```
1828             rth = rcu_dereference(rth->u.rt_next)) {
1829                 if (rth->fl.fl4_dst == daddr &&
1830                     rth->fl.fl4_src == saddr &&
1831                     rth->fl.iif == iif &&
1832                     rth->fl.oif == 0 &&
1833 #ifdef CONFIG_IP_ROUTE_FWMARK
1834                     rth->fl.fl4_fwmark == skb->nfmark &&
1835 #endif
1836                     rth->fl.fl4_tos == tos) {
1837                         rth->u.dst.lastuse = jiffies;
1838                         dst_hold(&rth->u.dst);
1839                         rth->u.dst.__use++;
1840                         RT_CACHE_STAT_INC(in_hit);
1841                         rcu_read_unlock();
1842                         skb->dst = (struct dst_entry*)rth;
1843                         return 0;
1844                 }
1845                 RT_CACHE_STAT_INC(in_hlist_search);
1846         }
1847         rcu_read_unlock();
1848
1849         /* La logique de reconnaissance de la multidiffusion est deplacee du cache
                de routage a ici.
1850             Le probleme etait que beaucoup trop de cartes Ethernet possedent des filtres
1851             materiels de multidiffusion mutiles/perdus :-( Il en resulte que l'hote du
1852             reseau de multidiffusion recoit beaucoup d'entrees de cache de routage
1853             inutiles, sortes de messages SDR provenant du monde entier. Nous essayons
                de nous en debarasser des maintenant.
1854             De fait, le logiciel IP de filtrage de multidiffussion ainsi fourni est
1855             organise raisonnablement. Il n'en resulte pas de ralentissement
1856             compare avec les entrees du cache de routage rejetees.
1857             Noter que les routeurs de multidiffusion ne sont pas leses puisque
1858             l'entree de cache de routage finit par etre creee.
1859          */
1860         if (MULTICAST(daddr)) {
1861                 struct in_device *in_dev;
1862
1863                 rcu_read_lock();
1864                 if ((in_dev = __in_dev_get(dev)) != NULL) {
1865                         int our = ip_check_mc(in_dev, daddr, saddr,
1866                                 skb->nh.iph->protocol);
1867                         if (our
1868 #ifdef CONFIG_IP_MROUTE
1869                             || (!LOCAL_MCAST(daddr) && IN_DEV_MFORWARD(in_dev))
1870 #endif
1871                             ) {
1872                                 rcu_read_unlock();
1873                                 return ip_route_input_mc(skb, daddr, saddr,
1874                                                          tos, dev, our);
1875                         }
1876                 }
1877                 rcu_read_unlock();
1878                 return -EINVAL;
1879         }
1880         return ip_route_input_slow(skb, daddr, saddr, tos, dev);
1881 }
```

Autrement dit :

- On déclare une entrée de cache de routage et un numéro de hachage.
- On déclare un index de périphérique réseau d'entrée, que l'on instantie avec l'index de périphérique réseau du descripteur de périphérique passé en argument.

– On ne garde que les deux premier bits du type de service passé en argument.

Le masque IPTOS_RT_MASK pour ce faire est défini dans le fichier linux/include/net/route.h:

Code Linux 2.6.10

```
137 #define IPTOS_RT_MASK   (IPTOS_TOS_MASK & ~3)
```

– Un appel à la fonction rt_hash_code() permet de calculer, à partir des adresses et du type de service, l'index dans la table de hachage du cache de routage.

– On verrouille en écriture le dispositif de mise à jour/écriture.

La macro générale rcu_dereference() eet définie dans le fichier linux/include/linux/rcupdate.h:

Code Linux 2.6.10

```
225 /**
226  * rcu_dereference - charge un pointeur RCU-protege dans une
227  * section critique du cote lecture RCU. Ce pointeur peut etre
228  * dereference de facon sure plus tard.
229  *
230  * Insere des barrieres de memoire sur les architectures qui l'exigent
231  * (a l'heure actuelle seulement les Alpha) et, plus important, documente
232  * exactement les pointeurs qui sont proteges par RCU.
233  */
234
235 #define rcu_dereference(p)     ({ \
236                                 typeof(p) _________p1 = p; \
237                                 smp_read_barrier_depends(); \
238                                 (_________p1); \
239                                 })
```

– On parcourt la liste chaînée associée à cet index à la recherche d'une concordance avec les cinq paramètres de la clé (adresse de destination, adresse source, carte réseau d'entrée, carte réseau de sortie égale à zéro et type de service). Si on trouve une telle entrée de cache de routage:

  – On modifie quelques-uns de ses champs, en particulier l'heure de dernier accès et son nombre de références.

  – On incrémente les informations statistiques sur le nombre de fois qu'on est entré dans le cache.

  La macro RT_CACHE_STAT_INC() eet définie dans le fichier linux/include/net/route.h:

  Code Linux 2.6.10

```
107 #define RT_CACHE_STAT_INC(field)                                        \
108                 (per_cpu_ptr(rt_cache_stat, smp_processor_id())->field++)
```

  – On déverrouille en écriture le dispositif de mise à jour/écriture.

  – On renseigne le champ entrée de cache de destination du descripteur de périphérique passé en argument avec le début de cette entrée de cache de routage, ce qui était le but recherché, et on renvoie 0.

– Dans le cas où aucune entrée de cache de routage qui concorde n'est trouvée, ce qui arrive nécessairement pour le premier paquet de la session, la suite du traitement est assurée par l'une des deux fonctions suivantes:

  – La fonction ip_route_imput_mc() est appelée dans le cas d'une adresse de destination de multidiffusion, ce qui ne nous intéressera pas pour l'instant.

  La macro MULTICAST() est définie dans le fichier linux/include/linux/in.h:

  Code Linux 2.6.10

```
246 #define MULTICAST(x)    (((x) & htonl(0xf0000000)) == htonl(0xe0000000))
```

– La fonction `ip_route_input_slow()` est chargée de la gestion des adresses de destination individuelle. Ceci est le cas du premier paquet entrant en particulier. Nous allons étudier ce cas dans la sous-section suivante.

23.2.2.2 Structure de données pour la redirection

Rappelons qu'une entité du type `in_device` permet de détenir les paramètres propres à Internet pour un descripteur de périphérique. Le type est défini dans le fichier `linux/include/linux/inetdevice.h`:

Code Linux 2.6.10

```
37 struct in_device
38 {
39         struct net_device       *dev;
40         atomic_t                refcnt;
41         int                     dead;
42         struct in_ifaddr        *ifa_list;      /* chaine ifaddr IP                  */
43         rwlock_t                mc_list_lock;
44         struct ip_mc_list       *mc_list;       /* chaine de filtres de
                                                      multidiffusion IP                 */
45         spinlock_t              mc_tomb_lock;
46         struct ip_mc_list       *mc_tomb;
47         unsigned long           mr_v1_seen;
48         unsigned long           mr_v2_seen;
49         unsigned long           mr_maxdelay;
50         unsigned char           mr_qrv;
51         unsigned char           mr_gq_running;
52         unsigned char           mr_ifc_count;
53         struct timer_list       mr_gq_timer;    /* minuteur general de requete       */
54         struct timer_list       mr_ifc_timer;   /* minuteur de changement d'interface */
55
56         struct neigh_parms      *arp_parms;
57         struct ipv4_devconf     cnf;
58         struct rcu_head         rcu_head;
59 };
```

dont les champs ont la signification suivante :

– Le champ `dev` est une référence arrière permettant de savoir à quel descripteur de périphérique est associée cette entité.
– Le champ `refcnt` est le compteur de références habituel. On modifie ce champ essentiellement à l'aide des fonctions `in_dev_get()` et `in_dev_put()` définies dans le même fichier.
– Le champ `dead` indique si le périphérique réseau IP est encore valide ou non.
– Le champ `ifa_list` renvoie à une liste de structures `in_ifaddr` contenant l'enregistrement des adresses IP de ce périphérique réseau. En effet Linux permet d'affecter plusieurs adresses IP à un périphérique réseau (*alias*).

Le type `struct in_ifaddr` est défini dans le même fichier :

Code Linux 2.6.10

```
85 struct in_ifaddr
86 {
87         struct in_ifaddr        *ifa_next;
88         struct in_device        *ifa_dev;
89         struct rcu_head         rcu_head;
90         u32                     ifa_local;
91         u32                     ifa_address;
92         u32                     ifa_mask;
93         u32                     ifa_broadcast;
94         u32                     ifa_anycast;
95         unsigned char           ifa_scope;
96         unsigned char           ifa_flags;
```

```
97         unsigned char           ifa_prefixlen;
98         char                    ifa_label[IFNAMSIZ];
99 };
```

Outre l'adresse IP (`ifa_address`), la structure `in_ifaddr` contient l'enregistrement de plusieurs autres paramètres, comme le masque de sous-réseau (`ifa_mask`), l'adresse de diffusion générale (`ifa_broadcast`), etc.

- Le champ `mc_list` est une liste constituée d'entités du type `ip_mc_list`. Chaque élément de cette liste mémorise les informations d'un groupe de multidiffusion IP. La multidiffusion ne nous intéressera pas pour l'instant.
- `arp_parms` renvoie sur une structure du type `neigh_parms` dans laquelle les principaux paramètres du protocole ARP sont mémorisés.

23.2.2.3 Cas d'une adresse de destination individuelle

Code Linux 2.6.10

La fonction `ip_route_input_slow()` est définie dans le fichier `linux/net/ipv4/route.c`:

```
1536 /*
1537  *      NOTE. Nous ecartons tous les paquets qui ont des adresses source
1538  *      locales, puisque tout paquet envoye en boucle de facon propre
1539  *      doit deja avoir une destination correcte attachee par la routine de sortie.
1540  *
1541  *      Un tel point de vue resoud deux gros problemes :
1542  *      1. Les peripheriques non simplexes sont traites correctement.
1543  *      2. Les essais de parodie IP sont filtre avec 100% de garantie.
1544  */
1545
1546 static int ip_route_input_slow(struct sk_buff *skb, u32 daddr, u32 saddr,
1547                                u8 tos, struct net_device *dev)
1548 {
1549         struct fib_result res;
1550         struct in_device *in_dev = in_dev_get(dev);
1551         struct in_device *out_dev = NULL;
1552         struct flowi fl = { .nl_u = { .ip4_u =
1553                                       { .daddr = daddr,
1554                                         .saddr = saddr,
1555                                         .tos = tos,
1556                                         .scope = RT_SCOPE_UNIVERSE,
1557 #ifdef CONFIG_IP_ROUTE_FWMARK
1558                                         .fwmark = skb->nfmark
1559 #endif
1560                                       } },
1561                             .iif = dev->ifindex };
1562         unsigned        flags = 0;
1563         u32             itag = 0;
1564         struct rtable * rth;
1565         unsigned        hash;
1566         u32             spec_dst;
1567         int             err = -EINVAL;
1568         int             free_res = 0;
1569
1570         /* IP est inhibee sur le peripherique. */
1571
1572         if (!in_dev)
1573                 goto out;
1574
1575         hash = rt_hash_code(daddr, saddr ^ (fl.iif << 5), tos);
1576
1577         /* Verification pour la plupart des martiens mysterieux, qui peuvent ne
1578            pas etre detectes par fib_lookup.
```

```
1579          */
1580
1581         if (MULTICAST(saddr) || BADCLASS(saddr) || LOOPBACK(saddr))
1582                 goto martian_source;
1583
1584         if (daddr == 0xFFFFFFFF || (saddr == 0 && daddr == 0))
1585                 goto brd_input;
1586
1587         /* Accepte l'adresse zero uniquement pour la diffusion generale ;
1588          * je ne sais meme pas le rectifier. J'attends les reclamations :-)
1589          */
1590         if (ZERONET(saddr))
1591                 goto martian_source;
1592
1593         if (BADCLASS(daddr) || ZERONET(daddr) || LOOPBACK(daddr))
1594                 goto martian_destination;
1595
1596         /*
1597          *      Maintenant nous sommes prets a router le paquet.
1598          */
1599         if ((err = fib_lookup(&fl, &res)) != 0) {
1600                 if (!IN_DEV_FORWARD(in_dev))
1601                         goto e_inval;
1602                 goto no_route;
1603         }
1604         free_res = 1;
1605
1606         RT_CACHE_STAT_INC(in_slow_tot);
1607
1608         if (res.type == RTN_BROADCAST)
1609                 goto brd_input;
1610
1611         if (res.type == RTN_LOCAL) {
1612                 int result;
1613                 result = fib_validate_source(saddr, daddr, tos,
1614                                              loopback_dev.ifindex,
1615                                              dev, &spec_dst, &itag);
1616                 if (result < 0)
1617                         goto martian_source;
1618                 if (result)
1619                         flags |= RTCF_DIRECTSRC;
1620                 spec_dst = daddr;
1621                 goto local_input;
1622         }
1623
1624         if (!IN_DEV_FORWARD(in_dev))
1625                 goto e_inval;
1626         if (res.type != RTN_UNICAST)
1627                 goto martian_destination;
1628
1629 #ifdef CONFIG_IP_ROUTE_MULTIPATH
1630         if (res.fi->fib_nhs > 1 && fl.oif == 0)
1631                 fib_select_multipath(&fl, &res);
1632 #endif
1633         out_dev = in_dev_get(FIB_RES_DEV(res));
1634         if (out_dev == NULL) {
1635                 if (net_ratelimit())
1636                         printk(KERN_CRIT "Bug in ip_route_input_slow(). "
1637                                          "Please, report\n");
1638                 goto e_inval;
1639         }
1640
```

```
1641         err = fib_validate_source(saddr, daddr, tos, FIB_RES_OIF(res), dev,
1642                                   &spec_dst, &itag);
1643         if (err < 0)
1644                 goto martian_source;
1645
1646         if (err)
1647                 flags |= RTCF_DIRECTSRC;
1648
1649         if (out_dev == in_dev && err && !(flags & (RTCF_NAT | RTCF_MASQ)) &&
1650             (IN_DEV_SHARED_MEDIA(out_dev) ||
1651              inet_addr_onlink(out_dev, saddr, FIB_RES_GW(res))))
1652                 flags |= RTCF_DOREDIRECT;
1653
1654         if (skb->protocol != htons(ETH_P_IP)) {
1655                 /* Non IP (par exemple ARP). Ne pas creer de route si ceci n'est pas
1656                  * valide pour le proxy arp. Les routes DNAT sont toujours valides.
1657                  */
1658                 if (out_dev == in_dev && !(flags & RTCF_DNAT))
1659                         goto e_inval;
1660         }
1661
1662         rth = dst_alloc(&ipv4_dst_ops);
1663         if (!rth)
1664                 goto e_nobufs;
1665
1666         atomic_set(&rth->u.dst.__refcnt, 1);
1667         rth->u.dst.flags= DST_HOST;
1668         if (in_dev->cnf.no_policy)
1669                 rth->u.dst.flags |= DST_NOPOLICY;
1670         if (in_dev->cnf.no_xfrm)
1671                 rth->u.dst.flags |= DST_NOXFRM;
1672         rth->fl.fl4_dst = daddr;
1673         rth->rt_dst     = daddr;
1674         rth->fl.fl4_tos = tos;
1675 #ifdef CONFIG_IP_ROUTE_FWMARK
1676         rth->fl.fl4_fwmark= skb->nfmark;
1677 #endif
1678         rth->fl.fl4_src = saddr;
1679         rth->rt_src     = saddr;
1680         rth->rt_gateway = daddr;
1681         rth->rt_iif     =
1682         rth->fl.iif     = dev->ifindex;
1683         rth->u.dst.dev  = out_dev->dev;
1684         dev_hold(rth->u.dst.dev);
1685         rth->idev       = in_dev_get(rth->u.dst.dev);
1686         rth->fl.oif     = 0;
1687         rth->rt_spec_dst= spec_dst;
1688
1689         rth->u.dst.input = ip_forward;
1690         rth->u.dst.output = ip_output;
1691
1692         rt_set_nexthop(rth, &res, itag);
1693
1694         rth->rt_flags = flags;
1695
1696 intern:
1697         err = rt_intern_hash(hash, rth, (struct rtable**)&skb->dst);
1698 done:
1699         in_dev_put(in_dev);
1700         if (out_dev)
1701                 in_dev_put(out_dev);
1702         if (free_res)
```

```
1703                fib_res_put(&res);
1704 out:    return err;
1705
1706 brd_input:
1707        if (skb->protocol != htons(ETH_P_IP))
1708                goto e_inval;
1709
1710        if (ZERONET(saddr))
1711                spec_dst = inet_select_addr(dev, 0, RT_SCOPE_LINK);
1712        else {
1713                err = fib_validate_source(saddr, 0, tos, 0, dev, &spec_dst,
1714                                          &itag);
1715                if (err < 0)
1716                        goto martian_source;
1717                if (err)
1718                        flags |= RTCF_DIRECTSRC;
1719        }
1720        flags |= RTCF_BROADCAST;
1721        res.type = RTN_BROADCAST;
1722        RT_CACHE_STAT_INC(in_brd);
1723
1724 local_input:
1725        rth = dst_alloc(&ipv4_dst_ops);
1726        if (!rth)
1727                goto e_nobufs;
1728
1729        rth->u.dst.output= ip_rt_bug;
1730
1731        atomic_set(&rth->u.dst.__refcnt, 1);
1732        rth->u.dst.flags= DST_HOST;
1733        if (in_dev->cnf.no_policy)
1734                rth->u.dst.flags |= DST_NOPOLICY;
1735        rth->fl.fl4_dst = daddr;
1736        rth->rt_dst     = daddr;
1737        rth->fl.fl4_tos = tos;
1738 #ifdef CONFIG_IP_ROUTE_FWMARK
1739        rth->fl.fl4_fwmark= skb->nfmark;
1740 #endif
1741        rth->fl.fl4_src = saddr;
1742        rth->rt_src     = saddr;
1743 #ifdef CONFIG_NET_CLS_ROUTE
1744        rth->u.dst.tclassid = itag;
1745 #endif
1746        rth->rt_iif     =
1747        rth->fl.iif     = dev->ifindex;
1748        rth->u.dst.dev  = &loopback_dev;
1749        dev_hold(rth->u.dst.dev);
1750        rth->idev       = in_dev_get(rth->u.dst.dev);
1751        rth->rt_gateway = daddr;
1752        rth->rt_spec_dst= spec_dst;
1753        rth->u.dst.input= ip_local_deliver;
1754        rth->rt_flags   = flags|RTCF_LOCAL;
1755        if (res.type == RTN_UNREACHABLE) {
1756                rth->u.dst.input= ip_error;
1757                rth->u.dst.error= -err;
1758                rth->rt_flags   &= ~RTCF_LOCAL;
1759        }
1760        rth->rt_type    = res.type;
1761        goto intern;
1762
1763 no_route:
1764        RT_CACHE_STAT_INC(in_no_route);
```

```
1765         spec_dst = inet_select_addr(dev, 0, RT_SCOPE_UNIVERSE);
1766         res.type = RTN_UNREACHABLE;
1767         goto local_input;
1768
1769         /*
1770          *      Ne pas mettre les adresses de martiens en cache : elles devraient etre
                    loguees (RFC1812)
1771          */
1772 martian_destination:
1773         RT_CACHE_STAT_INC(in_martian_dst);
1774 #ifdef CONFIG_IP_ROUTE_VERBOSE
1775         if (IN_DEV_LOG_MARTIANS(in_dev) && net_ratelimit())
1776                 printk(KERN_WARNING "martian destination %u.%u.%u.%u from "
1777                         "%u.%u.%u.%u, dev %s\n",
1778                         NIPQUAD(daddr), NIPQUAD(saddr), dev->name);
1779 #endif
1780 e_inval:
1781         err = -EINVAL;
1782         goto done;
1783
1784 e_nobufs:
1785         err = -ENOBUFS;
1786         goto done;
1787
1788 martian_source:
1789
1790         RT_CACHE_STAT_INC(in_martian_src);
1791 #ifdef CONFIG_IP_ROUTE_VERBOSE
1792         if (IN_DEV_LOG_MARTIANS(in_dev) && net_ratelimit()) {
1793                 /*
1794                  *      Recommandation RFC1812 : si la source vient de Mars,
1795                  *      la seule indication est l'en-tete MAC.
1796                  */
1797                 printk(KERN_WARNING "martian source %u.%u.%u.%u from "
1798                         "%u.%u.%u.%u, on dev %s\n",
1799                         NIPQUAD(daddr), NIPQUAD(saddr), dev->name);
1800                 if (dev->hard_header_len) {
1801                         int i;
1802                         unsigned char *p = skb->mac.raw;
1803                         printk(KERN_WARNING "ll header: ");
1804                         for (i = 0; i < dev->hard_header_len; i++, p++) {
1805                                 printk("%02x", *p);
1806                                 if (i < (dev->hard_header_len - 1))
1807                                         printk(":");
1808                         }
1809                         printk("\n");
1810                 }
1811         }
1812 #endif
1813         goto e_inval;
1814 }
```

Autrement dit :

- On déclare un résultat de consultation des tables de routage.
- On déclare une adresse de structure de périphérique Internet entrant, que l'on instantie avec celle du champ correspondant du descripteur de périphérique passé en argument. Si on ne parvient pas à l'instantier, on renvoie l'opposé du code d'erreur `EINVAL` (lignes 1572–1573).

Code Linux 2.6.10

La fonction en ligne `in_dev_get()` est définie dans le même fichier :

```
143 static __inline__ struct in_device *
144 in_dev_get(const struct net_device *dev)
```

```
145 {
146         struct in_device *in_dev;
147
148         rcu_read_lock();
149         in_dev = dev->ip_ptr;
150         if (in_dev)
151                 atomic_inc(&in_dev->refcnt);
152         rcu_read_unlock();
153         return in_dev;
154 }
```

- On déclare une adresse de structure de périphérique Internet sortant, que l'on initialise à `NULL` puisqu'on est en train de s'occuper d'un paquet entrant.
- On déclare un descripteur de flux Internet `fl` que les paramètres passés à la fonction permettent d'instantier (lignes 1552–1561).
- On détermine l'index de hachage (ligne 1575).
- On vérifie, avant de procéder à une requête FIB, que les adresses ne comportent pas de valeurs non autorisées telles que des adresses source de multidiffusion ou des adresses dont le premier octet du préfixe réseau est nul (lignes 1581–1594). Ces paquets sont détruits et, dans le cas où la compilation du noyau est configurée avec le paramètre `CONFIG_IP_ROUTE_VERBOSE`, ceci est consigné dans le journal système. Le seul cas où le zéro peut être utilisé est celui de la diffusion générale, lorsqu'on utilise 0.0.0.0 comme adresse source et adresse de destination.

  Les macros `BADCLASS()`, `LOOPBACK()` et `ZERONET()` sont définies dans le fichier `linux/include/linux/in.h`:

Code Linux 2.6.10

```
244 /* Quelques cas speciaux utiles dans le noyau. */
245 #define LOOPBACK(x)     (((x) & htonl(0xff000000)) == htonl(0x7f000000))
246 #define MULTICAST(x)    (((x) & htonl(0xf0000000)) == htonl(0xe0000000))
247 #define BADCLASS(x)     (((x) & htonl(0xf0000000)) == htonl(0xf0000000))
248 #define ZERONET(x)      (((x) & htonl(0xff000000)) == htonl(0x00000000))
249 #define LOCAL_MCAST(x)  (((x) & htonl(0xFFFFFF00)) == htonl(0xE0000000))
```

- On effectue une requête FIB (ligne 1599). Dans le cas où aucune entrée appropriée n'est trouvée, la fonction est interrompue. Elle retourne alors un code d'erreur, ce qui donne lieu ensuite au rejet du paquet, dans la fonction `ip_rcv_finish()` à partir de laquelle cette fonction avait été appelée.

  Dans le cas où le paquet est bien destiné à l'ordinateur hôte, une entrée, correspondant à celle de l'ordinateur, est bien trouvée dans la table locale, la première consultée.

  La macro `IN_DEV_FORWARD()` est définie dans le fichier `linux/include/linux/netdevice.h`:

Code Linux 2.6.10

```
61 #define IN_DEV_FORWARD(in_dev)          ((in_dev)->cnf.forwarding)
```

- On spécifie qu'on pourra libérer le résultat.
- On met à jour les informations statistiques sur le nombre total de requêtes effectuées avec succès.
- À partir du résultat de la requête FIB, il est possible de voir si l'adresse de destination est une adresse locale (`RTN_LOCAL`), et donc si le paquet correspondant est destiné au système local. Nous ne nous intéresserons pas ici, par exemple, aux paquets destinés à la diffusion générale (ligne 1608).
- Dans le cas où le résultat de la requête FIB indique que la destination est locale (ligne 1611) :
  - On effectue un contrôle de l'adresse source, à l'aide de la fonction `fib_validate_source()` étudiée dans la sous-section suivante. Si cette fonction a renvoyé une erreur,

on incrémente les informations statistiques à propos des sources “martiennes” et on renvoie l'opposé du code d'erreur `EINVAL`.

- L'adresse IP de destination spécifique prend comme valeur l'adresse de destination, c'est-à-dire une de celles de l'ordinateur hôte (il peut en posséder plusieurs).
- On essaie d'allouer une nouvelle entrée de cache de destination avec `ipv4_dst_ops` comme ensemble d'opérations (ligne 1725). Si on n'y parvient pas, on renvoie l'opposé du code d'erreur `ENOBUFS`.

Code Linux 2.6.10

La fonction `dst_alloc()` est définie dans le fichier `linux/net/core/dst.c`:

```
115 void * dst_alloc(struct dst_ops * ops)
116 {
117         struct dst_entry * dst;
118
119         if (ops->gc && atomic_read(&ops->entries) > ops->gc_thresh) {
120                 if (ops->gc())
121                         return NULL;
122         }
123         dst = kmem_cache_alloc(ops->kmem_cachep, SLAB_ATOMIC);
124         if (!dst)
125                 return NULL;
126         memset(dst, 0, ops->entry_size);
127         atomic_set(&dst->__refcnt, 0);
128         dst->ops = ops;
129         dst->lastuse = jiffies;
130         dst->path = dst;
131         dst->input = dst_discard_in;
132         dst->output = dst_discard_out;
133 #if RT_CACHE_DEBUG >= 2
134         atomic_inc(&dst_total);
135 #endif
136         atomic_inc(&ops->entries);
137         return dst;
138 }
```

- On renseigne les champs de cette nouvelle entrée de cache (lignes 1729–1760), dont seuls quelques-uns demandent une explication :
  - On attribue comme pointeur de fonction `output()` l'adresse de la fonction `ip_rt_bug()` afin que le paquet ne puisse pas sortir du système.

    Code Linux 2.6.10

    Cette fonction `ip_rt_bug()` est définie dans le fichier `linux/net/ipv4/route.c`:

```
1369 static int ip_rt_bug(struct sk_buff *skb)
1370 {
1371         printk(KERN_DEBUG "ip_rt_bug: %u.%u.%u.%u -> %u.%u.%u.%u, %s\n",
1372                 NIPQUAD(skb->nh.iph->saddr), NIPQUAD(skb->nh.iph->daddr),
1373                 skb->dev ? skb->dev->name : "?");
1374         kfree_skb(skb);
1375         return 0;
1376 }
```

  - Rappelons que le champ `rt_gateway` concerne l'adresse de la passerelle ou celle du destinataire dans le réseau local à diffusion dans lequel nous nous trouvons. Ce champ prend donc dans notre cas l'adresse de destination, qui est une des adresses IP de l'ordinateur sur lequel nous nous trouvons (ligne 1751).
  - On attribue comme valeur du pointeur `input()` l'adresse de la fonction `ip_local_deliver()` (ligne 1753) afin de permettre au paquet d'être livré localement.

– La structure `rtable`, remplie dans son ensemble, est intégrée à la table de hachage du cache de routage (ligne 1697).
– On décrémente le compteur de référence des ressources utilisées jusque-là et le code d'erreur fourni par la fonction `rt_intern_hash()` est renvoyé, ce qui termine l'action de la fonction étudiée dans le cas d'une adresse de destination locale.

La fonction `in_dev_put()` est définie dans le fichier `linux/include/linux/netdevice.h` :

Code Linux 2.6.10

```
164 static inline void in_dev_put(struct in_device *idev)
165 {
166         if (atomic_dec_and_test(&idev->refcnt))
167                 in_dev_finish_destroy(idev);
168 }
```

23.2.2.4 Validation de l'adresse source

La fonction `fib_validate_source()` est définie dans le fichier `linux/net/ipv4/fib_frontend.c` :

Code Linux 2.6.10

```
153 /* Etant donnes (source du paquet source, interface d'entree) et optionnellement
       (dst, oif, tos) :
154    - (principalement) verifier que la source est valide, c'est-a-dire qu'il ne
155      s'agit pas d'une adresse de diffusion generale ou de notre adresse locale.
156    - determiner depuis quelle interface "logique" ce paquet est arrive
157      et calculer l'adresse de "destination specifique".
158    - verifier que ce paquet provient d'une interface physique attendue.
159  */
160
161 int fib_validate_source(u32 src, u32 dst, u8 tos, int oif,
162                         struct net_device *dev, u32 *spec_dst, u32 *itag)
163 {
164         struct in_device *in_dev;
165         struct flowi fl = { .nl_u = { .ip4_u =
166                                       { .daddr = src,
167                                         .saddr = dst,
168                                         .tos = tos } },
169                             .iif = oif };
170         struct fib_result res;
171         int no_addr, rpf;
172         int ret;
173
174         no_addr = rpf = 0;
175         read_lock(&inetdev_lock);
176         in_dev = __in_dev_get(dev);
177         if (in_dev) {
178                 no_addr = in_dev->ifa_list == NULL;
179                 rpf = IN_DEV_RPFILTER(in_dev);
180         }
181         read_unlock(&inetdev_lock);
182
183         if (in_dev == NULL)
184                 goto e_inval;
185
186         if (fib_lookup(&fl, &res))
187                 goto last_resort;
188         if (res.type != RTN_UNICAST)
189                 goto e_inval_res;
190         *spec_dst = FIB_RES_PREFSRC(res);
191         fib_combine_itag(itag, &res);
192 #ifdef CONFIG_IP_ROUTE_MULTIPATH
```

```
193         if (FIB_RES_DEV(res) == dev || res.fi->fib_nhs > 1)
194 #else
195         if (FIB_RES_DEV(res) == dev)
196 #endif
197         {
198                 ret = FIB_RES_NH(res).nh_scope >= RT_SCOPE_HOST;
199                 fib_res_put(&res);
200                 return ret;
201         }
202         fib_res_put(&res);
203         if (no_addr)
204                 goto last_resort;
205         if (rpf)
206                 goto e_inval;
207         fl.oif = dev->ifindex;
208
209         ret = 0;
210         if (fib_lookup(&fl, &res) == 0) {
211                 if (res.type == RTN_UNICAST) {
212                         *spec_dst = FIB_RES_PREFSRC(res);
213                         ret = FIB_RES_NH(res).nh_scope >= RT_SCOPE_HOST;
214                 }
215                 fib_res_put(&res);
216         }
217         return ret;
218
219 last_resort:
220         if (rpf)
221                 goto e_inval;
222         *spec_dst = inet_select_addr(dev, 0, RT_SCOPE_UNIVERSE);
223         *itag = 0;
224         return 0;
225
226 e_inval_res:
227         fib_res_put(&res);
228 e_inval:
229         return -EINVAL;
230 }
```

Autrement dit :

- On déclare l'adresse d'une structure de périphérique Internet, que l'on essaie d'instantier avec l'adresse du champ associé du descripteur de périphérique passé en argument. Si on n'y parvient pas, on renvoie l'opposé du code d'erreur `EINVAL`.

  La fonction en ligne `__in_dev_get()` est définie dans le fichier `linux/include/linux/netdevice.h` :

Code Linux 2.6.10

```
156 static __inline__ struct in_device *
157 __in_dev_get(const struct net_device *dev)
158 {
159         return (struct in_device*)dev->ip_ptr;
160 }
```

- On déclare un flux Internet, dont on renseigne les champs grace aux valeurs passées en argument.
- On effectue une requête FIB (ligne 186). Si celle-ci réussit, on renseigne l'argument (passé en adresse) `spec_dst` et on renvoie 0.

23.2.3 Choix de la fonction de traitement du paquet

La fonction `dst_input()` est définie dans le fichier `linux/include/net/dst.h`: Code Linux 2.6.10

```
234 /* Paquet entrant de reseau a transport.  */
235 static inline int dst_input(struct sk_buff *skb)
236 {
237         int err;
238
239         for (;;) {
240                 err = skb->dst->input(skb);
241
242                 if (likely(err == 0))
243                         return err;
244                 /* Oh, Jamal... Je ne vous pardonnerai pas cette pagaille. :-) */
245                 if (unlikely(err != NET_XMIT_BYPASS))
246                         return err;
247         }
248 }
```

Elle se contente de faire appel à la fonction `skb->dst->input()` associée au descripteur de tampon et de renvoyer le code d'erreur fourni par celle-ci.

La fonction pointée, comme nous venons de le voir est l'une des trois suivantes :

- `ip_local_deliver()` pour les paquets locaux à diffusion individuelle.
- `ip_forward()` en cas d'acheminement d'un paquet à diffusion individuelle.
- `ip_mr_input()` pour les paquets de multidiffusion.

Dans la suite de ce chapitre, nous ne nous intéressons qu'aux paquets à remettre localement. Les deux autres cas seront traités plus tard.

23.3 Remise locale des paquets

23.3.1 Première étape : réassemblage des fragments

La fonction `ip_local_deliver()` est définie dans le fichier `linux/net/ipv4/ip_input.c`: Code Linux 2.6.10

```
266 /*
267  *      Livre les paquets IP aux couches superieures de protocole.
268  */
269 int ip_local_deliver(struct sk_buff *skb)
270 {
271         /*
272          *      Reassemble les fragments IP.
273          */
274
275         if (skb->nh.iph->frag_off & htons(IP_MF|IP_OFFSET)) {
276                 skb = ip_defrag(skb);
277                 if (!skb)
278                         return 0;
279         }
280
281         return NF_HOOK(PF_INET, NF_IP_LOCAL_IN, skb, skb->dev, NULL,
282                        ip_local_deliver_finish);
283 }
```

Autrement dit la première tâche consiste à réassembler d'éventuels fragments du paquet à l'aide de la fonction `ip_defrag()`, ce qui sera étudié dans au chapitre 36 consacré à la fragmentation IP. Nous supposerons dans une première étape que ceci n'est pas nécessaire. Suit un appel d'un point d'ancrage `NF_IP_LOCAL_IN`, la fonction à venir étant `ip_local_deliver_finish()`.

23.3.2 Gestion des protocoles de transport dans la couche IP

23.3.2.1 Descripteur de protocole de transport

Sous Linux, chaque protocole de couche de transport de la suite TCP/IP est enregistré à travers un **descripteur de protocole** (*protocol handler* en anglais), entité du type `struct net_protocol`. Ce type est défini dans le fichier en-tête `linux/include/net/protocol.h`:

Code Linux 2.6.10

```
36 /* Ceci est utilise pour enregistrer les protocoles. */
37 struct net_protocol {
38         int                     (*handler)(struct sk_buff *skb);
39         void                    (*err_handler)(struct sk_buff *skb, u32 info);
40         int                     no_policy;
41 };
```

Il comprend essentiellement les fonction de réception des tampons de socket et de traitement des messages d'erreurs ICMP transmis par la couche IP.

Par exemple, l'entité `udp_protocol` pour UDP est définie dans le fichier `linux/net/ipv4/af_inet.c`:

Code Linux 2.6.10

```
981 static struct net_protocol udp_protocol = {
982          .handler =      udp_rcv,
983          .err_handler =  udp_err,
984          .no_policy =    1,
985 };
```

23.3.2.2 Table de hachage des protocoles de transport

La couche IP gère les descripteurs de protocole de transport grâce à une table de hachage appelée `inet_protos[]`, déclarée dans le fichier `linux/net/ipv4/protocol.c`:

Code Linux 2.6.10

```
51 struct net_protocol *inet_protos[MAX_INET_PROTOS];
```

Le nombre maximum de protocoles de couche de transport est spécifié par la constante `MAX_INET_PROTOS`, définie dans le fichier `linux/include/net/protocol.h`:

Code Linux 2.6.10

```
33 #define MAX_INET_PROTOS 256             /* Doit etre une puissance de 2         */
```

23.3.2.3 Ajout d'un protocole de transport

L'ajout d'un descripteur de protocole à la table de hachage s'effectue grâce à la fonction `inet_add_protocol()`, définie dans le fichier `include/net/ipv4/protocol.c`:

Code Linux 2.6.10

```
54 /*
55  *      Ajoute un descripteur de protocole aux tables de hachage
56  */
57
58 int inet_add_protocol(struct inet_protocol *prot, unsigned char protocol)
59 {
60         int hash, ret;
61
62         hash = protocol & (MAX_INET_PROTOS - 1);
63
64         spin_lock_bh(&inet_proto_lock);
65         if (inet_protos[hash]) {
66                 ret = -1;
67         } else {
68                 inet_protos[hash] = prot;
69                 ret = 0;
70         }
71         spin_unlock_bh(&inet_proto_lock);
```

```
72
73          return ret;
74 }
```

23.3.2.4 Initialisation de la table de hachage des protocoles de transport

La table de hachage des protocoles de transport est initialisée dans le corps de la fonction `inet_init()`, déjà rencontrée, définie dans le fichier `linux/net/ipv4/af_inet.c`:

Code Linux 2.6.10

```
1017 static int __init inet_init(void)
1018 {
[...]
1051          /*
1052           *      Ajouter tous les protocoles de base.
1053           */
1054
1055          if (inet_add_protocol(&icmp_protocol, IPPROTO_ICMP) < 0)
1056                  printk(KERN_CRIT "inet_init: Cannot add ICMP protocol\n");
1057          if (inet_add_protocol(&udp_protocol, IPPROTO_UDP) < 0)
1058                  printk(KERN_CRIT "inet_init: Cannot add UDP protocol\n");
1059          if (inet_add_protocol(&tcp_protocol, IPPROTO_TCP) < 0)
1060                  printk(KERN_CRIT "inet_init: Cannot add TCP protocol\n");
1061 #ifdef CONFIG_IP_MULTICAST
1062          if (inet_add_protocol(&igmp_protocol, IPPROTO_IGMP) < 0)
1063                  printk(KERN_CRIT "inet_init: Cannot add IGMP protocol\n");
1064 #endif
[...]
1122 }
```

23.3.3 Deuxième étape : démultiplexage de la couche de transport

La fonction `ip_local_deliver_finish()` est chargée du démultiplexage, c'est-à-dire de déterminer à quel protocole de couche de transport envoyer la partie données du paquet, c'est-à-dire le datagramme dans le cas UDP. Elle vérifie d'abord si le paquet est brut, c'est-à-dire s'il ne correspond à aucun des protocoles enregistrés. Dans le cas contraire, le protocole de transport est déterminé et la partie données du paquet (le tampon de socket dans le cas de Linux) est passé à ce protocole.

Cette fonction est définie dans le fichier `linux/net/ipv4/ip_input.c`:

Code Linux 2.6.10

```
199 static inline int ip_local_deliver_finish(struct sk_buff *skb)
200 {
201          int ihl = skb->nh.iph->ihl*4;
202
203 #ifdef CONFIG_NETFILTER_DEBUG
204          nf_debug_ip_local_deliver(skb);
205 #endif /*CONFIG_NETFILTER_DEBUG*/
206
207          __skb_pull(skb, ihl);
208
209          /* Reference libre auparavant : nous n'en avons pas besoin plus longtemps,
210             et elle peut contenir le module ip_conntrack charge indefiniment. */
211          nf_reset(skb);
212
213          /* Pointe sur le datagramme IP, juste passer l'en-tete. */
214          skb->h.raw = skb->data;
215
216          rcu_read_lock();
217          {
218                  /* Note : voir raw.c et net/raw.h, RAWV4_HTABLE_SIZE==MAX_INET_PROTOS */
219                  int protocol = skb->nh.iph->protocol;
```

```
220                 int hash;
221                 struct sock *raw_sk;
222                 struct net_protocol *ipprot;
223
224         resubmit:
225                 hash = protocol & (MAX_INET_PROTOS - 1);
226                 raw_sk = sk_head(&raw_v4_htable[hash]);
227
228                 /* S'il peut y avoir une socket brute, nous devons verifier -
229                  * sinon inutile de prendre des precautions
230                  */
231                 if (raw_sk)
232                         raw_v4_input(skb, skb->nh.iph, hash);
233
234                 if ((ipprot = rcu_dereference(inet_protos[hash])) != NULL) {
235                         int ret;
236
237                         if (!ipprot->no_policy &&
238                             !xfrm4_policy_check(NULL, XFRM_POLICY_IN, skb)) {
239                                 kfree_skb(skb);
240                                 goto out;
241                         }
242                         ret = ipprot->handler(skb);
243                         if (ret < 0) {
244                                 protocol = -ret;
245                                 goto resubmit;
246                         }
247                         IP_INC_STATS_BH(IPSTATS_MIB_INDELIVERS);
248                 } else {
249                         if (!raw_sk) {
250                                 if (xfrm4_policy_check(NULL, XFRM_POLICY_IN, skb)) {
251                                         IP_INC_STATS_BH(IPSTATS_MIB_INUNKNOWNPROTOS);
252                                         icmp_send(skb, ICMP_DEST_UNREACH,
253                                                   ICMP_PROT_UNREACH, 0);
254                                 }
255                         } else
256                                 IP_INC_STATS_BH(IPSTATS_MIB_INDELIVERS);
257                         kfree_skb(skb);
258                 }
259         }
260  out:
261         rcu_read_unlock();
262
263         return 0;
264 }
```

Autrement dit :

- On déclare une longueur d'en-tête IP, que l'on instantie avec celle de l'en-tête IP associé au descripteur de tampon passé en argument.
- Si on a choisi l'option de compilation du noyau concernant le débogage du filtrage réseau, ce qui ne nous intéressera pas ici, on fait appel à la fonction `nf_debug_ip_local_deliver()`.
- On ajuste la zone des données du tampon de socket pour qu'elle puisse contenir l'en-tête.
- On positionne le compteur de références au filtrage à `NULL`.

La fonction en ligne `nf_reset()` est définie dans le fichier `linux/include/linux/skbuff.h` :

Code Linux 2.6.10

```
1151 static inline void nf_reset(struct sk_buff *skb)
1152 {
1153         nf_conntrack_put(skb->nfct);
```

```
1154         skb->nfct = NULL;
1155 #ifdef CONFIG_NETFILTER_DEBUG
1156         skb->nf_debug = 0;
1157 #endif
1158 }
```

- On passe l'en-tête IP en faisant pointer l'en-tête pour un paquet brut au début de la zone des données du paquet.
- On positionne le verrou d'écriture pour le mécanisme de mise à jour de l'écriture-copie.
- On déclare un numéro de protocole de la couche transport, que l'on instantie avec celui associé à l'en-tête IP, lui-même associé au descripteur de tampon passé en argument.
- On déclare un numéro de hachage, un descripteur de couche de transport (non vraiment utilisé) et une adresse de descripteur de protocole.
- On détermine le numéro de hachage du protocole de la couche transport.
- On essaie d'instantier le descripteur de couche transport à partir de ce numéro de hachage en cherchant d'abord dans la table des paquets bruts.

  La table `raw_v4_htable[]` est déclarée dans le fichier `linux/net/ipv4/raw.c`: Code Linux 2.6.10

```
30 #define RAWV4_HTABLE_SIZE       MAX_INET_PROTOS
31 extern struct hlist_head raw_v4_htable[RAWV4_HTABLE_SIZE];
```

  La fonction en ligne `sk_head()` est définie dans le fichier `linux/include/net/sock.h`: Code Linux 2.6.10

```
269 /*
270  * Routines d'aide pour les listes de hachage
271  */
272 static inline struct sock *__sk_head(struct hlist_head *head)
273 {
274         return hlist_entry(head->first, struct sock, sk_node);
275 }
276
277 static inline struct sock *sk_head(struct hlist_head *head)
278 {
279         return hlist_empty(head) ? NULL : __sk_head(head);
280 }
```

- Si on a obtenu ainsi un descripteur de couche de transport de paquet brut, on fait appel à la fonction `raw_v4_input()`, que nous devrions étudier à propos des paquets bruts, mais qui ne nous intéressera pas dans cet ouvrage.
- On essaie d'obtenir le descripteur de protocole grâce au numéro de hachage.

  - Si on en obtient un :

    - On déclare une valeur de retour.
    - Dans le cas où le descripteur de protocole exige une police mais qu'on ne peut pas obtenir celle-ci, on libère le descripteur de tampon, on déverrouille le dispositif de mise à jour de l'écriture-copie et on renvoie 0.
    - On fait appel à la routine de traitement correspondante de la couche de transport, ce qui est le but principal de cette fonction.

      Nous avons vu, par exemple, qu'il s'agit de `udp_rcv()` dans le cas de UDP, c'est-à-dire d'un numéro de protocole égal à 17.
    - Si une erreur s'est produite lors de l'appel à cette routine, on essaie à nouveau.
    - On incrémente l'information statistique sur le nombre de paquets IP reçus et livrés à la couche transport correspondante.

- Si on n'obtient pas de descripteur de protocole de couche de transport :
  - S'il s'agit d'un paquet brut, on incrémente l'information statistique sur le nombre de paquets IP reçus et livrés à la couche transport correspondante.
  - Sinon on incrémente l'information statistique sur le nombre de protocoles inconnus et on envoie un message ICMP "Destination non atteignable" à l'expéditeur du paquet, grâce à la fonction `icmp_send()`.
- Dans les deux cas, on libère le descripteur de tampon de socket.

- On déverrouille le dispositif de mise à jour de l'écriture-copie et on renvoie 0.

Chapitre 24

Les descripteurs de couche transport

Nous veons de voir comment un paquet entrant, dépecé de son en-tête IP, est transmis en tant que datagramme à une fonction de traitement de couche de transport, `udp_rcv()` dans le cas de UDP. Avant de poursuivre le traitement de celui-ci, étudions une nouvelle structure de données utile pour la couche transport.

On a besoin d'entreposer un certain nombre de renseignements utiles pour les couches de réseau (un peu) et transport (surtout), tels que l'adresse de destination et le numéro de port. Ceci s'effectue grâce à un **descripteur de couche transport**, de type `struct sock`.

Nous allons étudier la structure des descripteurs de couche transport et leur gestion dans ce chapitre qui, comme la gestion des descripteurs de tampon, ressort plus de la gestion de la mémoire que du réseau proprement dit.

24.1 Structure des descripteurs de couche transport

24.1.1 Définition du type

Code Linux 2.6.10

Le type `struct sock` est défini dans le fichier en-tête `linux/include/net/sock.h`:

```
115 /**
116   *     struct sock - representation de la couche reseau des sockets
117   *     @__sk_common - structure partagee avec tcp_tw_bucket
118   *     @sk_zapped - ax25 & ipx signifie !linked
119   *     @sk_shutdown - masque de %SEND_SHUTDOWN et/ou %RCV_SHUTDOWN
120   *     @sk_use_write_queue - si on doit appeler sk->sk_write_space dans sock_wfree
121   *     @sk_userlocks - positionnements de %SO_SNDBUF et de %SO_RCVBUF
122   *     @sk_lock -      synchronisateur
123   *     @sk_rcvbuf - taille du tampon de reception en octets
124   *     @sk_sleep - sock attend une file d'attente
125   *     @sk_dst_cache - cache de destination
126   *     @sk_dst_lock - verrou du cache de destination
127   *     @sk_policy - police de flux
128   *     @sk_rmem_alloc - octets de la file d'attente en reception remis
129   *     @sk_receive_queue - paquets entrants
130   *     @sk_wmem_alloc - octets de la file d'attente en emission remis
131   *     @sk_write_queue - File d'attente d'envoi des paquets
132   *     @sk_omem_alloc - "o" est "option" ou "other"
133   *     @sk_wmem_queued - taille de la file d'attente persistante
134   *     @sk_forward_alloc - espace alloue en avant
135   *     @sk_allocation - mode d'allocation
136   *     @sk_sndbuf - taille du tampon d'envoi en octets
137   *     @sk_flags - positionnements de %SO_LINGER (l_onoff), %SO_BROADCAST, %SO_KEEPALIVE,
                                    %SO_OOBINLINE
138   *     @sk_no_check - positionnement de %SO_NO_CHECK s'il y a ou non des paquets a verifier
139   *     @sk_debug - positionnement de %SO_DEBUG
140   *     @sk_rcvtstamp - positionnement de %SO_TIMESTAMP
141   *     @sk_no_largesend - si on doit envoyer de grands segments ou non
142   *     @sk_route_caps - possibilites de la route (par exemple %NETIF_F_TSO)
143   *     @sk_lingertime - positionnement de %SO_LINGER l_linger
144   *     @sk_hashent - entree de hachage dans plusieurs tables (par exemple tcp_ehash)
145   *     @sk_backlog - toujours utilise avec le verrou tournant per-socket detenu
146   *     @sk_callback_lock - utilise avec les retours d'appel a la fin de cette struct
147   *     @sk_error_queue - rarement utilise
148   *     @sk_prot - gestionnaires de protocole dans une famille de reseau
149   *     @sk_err - derniere erreur
150   *     @sk_err_soft - erreurs non vouees a l'echec mais qui sont la cause
           d'un echec persistant, pas juste 'delai ecoule'
151   *     @sk_ack_backlog - nombre de clients en attente possibles en cours
152   *     @sk_max_ack_backlog - positionne le nombre de clients en attente possible dans
      listen()
153   *     @sk_priority - positionnement de %SO_PRIORITY
154   *     @sk_type - type de socket (%SOCK_STREAM, etc)
155   *     @sk_localroute - route locale seulement, positionnement de %SO_DONTROUTE
156   *     @sk_protocol - protocole auquel cette socket appartient dans cette famille reseau
157   *     @sk_peercred - positionnement de %SO_PEERCRED
158   *     @sk_rcvlowat - positionnement de %SO_RCVLOWAT
159   *     @sk_rcvtimeo - positionnement de %SO_RCVTIMEO
160   *     @sk_sndtimeo - positionnement de %SO_SNDTIMEO
161   *     @sk_filter - instructions de filtage de la socket
162   *     @sk_protinfo - zone privee, propre a la famille reseau, lorsqu'on n'utilise pas
                          le slab
163   *     @sk_slab - le cache de slab a partir duquel cette instance a ete allouee
164   *     @sk_timer - minuteur de nettoyage de sock
165   *     @sk_stamp - estampille temporelle du dernier paquet recu
166   *     @sk_socket - Ident et report des signaux d'E/S
167   *     @sk_user_data - donnees privees de la couche RPC
```

```
168   *     @sk_owner - module qui possede cette socket
169   *     @sk_sndmsg_page - page en cache pour sendmsg
170   *     @sk_sndmsg_off - decalage en cache pour sendmsg
171   *     @sk_send_head - debut des choses a emettre
172   *     @sk_write_pending - une ecriture pour une socket de flux attend de demarrer
173   *     @sk_queue_shrunk - une file d'attente en ecriture a ete reduite recemment
174   *     @sk_state_change - appel pour indiquer le changment dans l'etat de la sock
175   *     @sk_data_ready - appel pour indiquer qu'il y a des donnees a traiter
176   *     @sk_write_space - appel pour indiquer qu'il y a de l'espace d'envoi disponible
177   *     @sk_error_report - appel pour indiquer les erreurs (par exemple %MSG_ERRQUEUE)
178   *     @sk_backlog_rcv - appel pour traiter le nombre de clients en attente
179   *     @sk_destruct - appele au moment de la liberation de sock, c'est-a-dire lorsque tous
          les refcnt == 0
180   */
181  struct sock {
182          /*
183           * struct tcp_tw_bucket utilise maintenant egalement sock_common, donc n'ajouter
184           * rien s'il vous plait avant ce premier membre (__sk_common) --acme
185           */
186          struct sock_common      __sk_common;
187  #define sk_family               __sk_common.skc_family
188  #define sk_state                __sk_common.skc_state
189  #define sk_reuse                __sk_common.skc_reuse
190  #define sk_bound_dev_if         __sk_common.skc_bound_dev_if
191  #define sk_node                 __sk_common.skc_node
192  #define sk_bind_node            __sk_common.skc_bind_node
193  #define sk_refcnt               __sk_common.skc_refcnt
194          volatile unsigned char  sk_zapped;
195          unsigned char           sk_shutdown;
196          unsigned char           sk_use_write_queue;
197          unsigned char           sk_userlocks;
198          socket_lock_t           sk_lock;
199          int                     sk_rcvbuf;
200          wait_queue_head_t       *sk_sleep;
201          struct dst_entry        *sk_dst_cache;
202          rwlock_t                sk_dst_lock;
203          struct xfrm_policy      *sk_policy[2];
204          atomic_t                sk_rmem_alloc;
205          struct sk_buff_head     sk_receive_queue;
206          atomic_t                sk_wmem_alloc;
207          struct sk_buff_head     sk_write_queue;
208          atomic_t                sk_omem_alloc;
209          int                     sk_wmem_queued;
210          int                     sk_forward_alloc;
211          unsigned int            sk_allocation;
212          int                     sk_sndbuf;
213          unsigned long           sk_flags;
214          char                    sk_no_check;
215          unsigned char           sk_debug;
216          unsigned char           sk_rcvtstamp;
217          unsigned char           sk_no_largesend;
218          int                     sk_route_caps;
219          unsigned long           sk_lingertime;
220          int                     sk_hashent;
221          /*
222           * La file d'attente des clients en attente est particuliere, elle est toujours
223           * utilisee avec le verrou tournant per-socket detenu et exige un acces a latence
224           * basse. D'ou le cas particulier de son implementation.
225           */
226          struct {
227                  struct sk_buff *head;
228                  struct sk_buff *tail;
```

```
229         } sk_backlog;
230         rwlock_t                sk_callback_lock;
231         struct sk_buff_head     sk_error_queue;
232         struct proto            *sk_prot;
233         int                     sk_err,
234                                 sk_err_soft;
235         unsigned short          sk_ack_backlog;
236         unsigned short          sk_max_ack_backlog;
237         __u32                   sk_priority;
238         unsigned short          sk_type;
239         unsigned char           sk_localroute;
240         unsigned char           sk_protocol;
241         struct ucred            sk_peercred;
242         int                     sk_rcvlowat;
243         long                    sk_rcvtimeo;
244         long                    sk_sndtimeo;
245         struct sk_filter        *sk_filter;
246         void                    *sk_protinfo;
247         kmem_cache_t            *sk_slab;
248         struct timer_list       sk_timer;
249         struct timeval          sk_stamp;
250         struct socket           *sk_socket;
251         void                    *sk_user_data;
252         struct module           *sk_owner;
253         struct page             *sk_sndmsg_page;
254         __u32                   sk_sndmsg_off;
255         struct sk_buff          *sk_send_head;
256         int                     sk_write_pending;
257         void                    *sk_security;
258         __u8                    sk_queue_shrunk;
259         /* three bytes hole, try to pack */
260         void                    (*sk_state_change)(struct sock *sk);
261         void                    (*sk_data_ready)(struct sock *sk, int bytes);
262         void                    (*sk_write_space)(struct sock *sk);
263         void                    (*sk_error_report)(struct sock *sk);
264         int                     (*sk_backlog_rcv)(struct sock *sk,
265                                                   struct sk_buff *skb);
266         void                    (*sk_destruct)(struct sock *sk);
267 };
```

24.1.2 Attributs communs

Le premier champ, `__sk_common`, regroupe les attributs qui sont communs avec une autre structure, `struct tcp_tw_bucket`, concernant le protocole TCP. Le type `struct sock_common` de la partie commune est défini une peu plus haut dans le même fichier en-tête :

Code Linux 2.6.10

```
92  /**
93    *     struct sock_common - representation minimale de la couche reseau des sockets
94    *     @skc_family - famille d'adresses reseau
95    *     @skc_state - Etat de connexion
96    *     @skc_reuse - positionnement de %SO_REUSEADDR
97    *     @skc_bound_dev_if - index du peripherique lie si != 0
98    *     @skc_node - principal lien de hachage pour les tables de consultation de divers
                         protocoles
99    *     @skc_bind_node - lien de hachage bind pour les tables de consultation de divers
                           protocoles
100    *     @skc_refcnt - compteur des references
101   *
102   *     Ceci est la representation minimale pour la couche reseau des sockets, l'en-tete
103   *     de struct sock et de struct tcp_tw_bucket.
104   */
105 struct sock_common {
```

```
106         unsigned short          skc_family;
107         volatile unsigned char  skc_state;
108         unsigned char           skc_reuse;
109         int                     skc_bound_dev_if;
110         struct hlist_node       skc_node;
111         struct hlist_node       skc_bind_node;
112         atomic_t                skc_refcnt;
113 };
```

Les champs sont :

- Le numéro de la famille de protocoles (`AF_INET` pour ce qui nous intéresse).
- L'état de la connexion, les valeurs dépendant du type de communication choisi.

  Les états d'une connexion sont représentés par des constantes symboliques (de préfixe `TCP_` même s'il s'agit de UDP) définies dans le fichier `linux/include/linux/tcp.h`: Code Linux 2.6.10

```
59 enum {
60   TCP_ESTABLISHED = 1,
61   TCP_SYN_SENT,
62   TCP_SYN_RECV,
63   TCP_FIN_WAIT1,
64   TCP_FIN_WAIT2,
65   TCP_TIME_WAIT,
66   TCP_CLOSE,
67   TCP_CLOSE_WAIT,
68   TCP_LAST_ACK,
69   TCP_LISTEN,
70   TCP_CLOSING,   /* maintenant un etat valide */
71
72   TCP_MAX_STATES /* Laisser a la fin ! */
73 };
```

  Une connexion connaît plusieurs états durant sa durée de vie. Les états définis par [RFC 793] pour TCP sont les suivants :

  - `LISTEN` : la connexion reste en ettente d'une requête de connexion externe par une socket TCP distante. Cet état est atteint après une demande de connexion passive.
  - `SYN-SENT` : la connexion se met en attente d'une requête de connexion, après avoir elle-même envoyé une requête à un destinataire.
  - `SYN-RECEIVED` : les deux requêtes de connexion se sont croisées ; la connexion attend confirmation de son établissement.
  - `ESTABLISHED` : la connexion a été confirmée de part et d'autre et les données peuvent transiter sur la voie de communication. C'est l'état stable actif de la connexion.
  - `FIN-WAIT-1` : sur requête de déconnexion émise par l'application, la connexion demande la confirmation d'une requête de déconnexion qu'elle a elle-même émise vers la socket distante.
  - `FIN-WAIT-2` : la connexion se met en attente d'une requête de déconnexion par la socket distante, une fois reçue la confirmation de sa propre requête.
  - `CLOSE-WAIT` : la connexion se met en attente d'une requête de déconnexion émise par l'application.
  - `CLOSING` : la connexion attend la confirmation de sa requête de déconnexion par la socket TCP distante, laquelle avait auparavant émis sa propre requête de déconnexion.
  - `LAST-ACK` : la connexion attend la confirmation de sa requête de déconnexion, émise suite à une requête similaire à l'initiative de la socket distante.

  - `TIME-WAIT`: la connexion connaît un temps de latence avant fermeture complète du canal, pour s'assurer que toutes les confirmations ont bien été reçues.
  - `CLOSED`: la connexion n'existe plus. Il s'agit d'un pseudo-état dans la mesure où la connexion n'existe plus.

- Un paramètre permettant de spécifier si on réutilise le descripteur (utilisé dans la fonction `setsockopt()`).
- L'index de l'interface physique du périphérique.
- Une liste de hachage.
- Une seconde liste de hachage, utile pour certains protocoles.
- Un compteur de références, qui dénombre les entités qui utilisent le descripteur. Le descripteur ne pourra être libéré que si ce compteur est égal à zéro.

24.1.3 Attributs de contrôle

Les attributs de contrôle sont les suivants :

- Le champ `sk_zapped` est utile pour les suites de protocoles AX25 et IPX mais pas pour TCP/IP.
- Le champ `sk_shutdown` indique si on en train d'arrêter la connexion.
- Le champ `sk_use_write_queue` indique si on doit appeler `sk->sk_write_space`, utile pour la fonction `sock_vfree()`.
- Le champ `sk_userlocks` permet de spécifier ce qu'il faut synchroniser.
- Le champ `sk_lock` permet de synchroniser.

Code Linux 2.6.10

Le type `socket_lock_t` est défini dans le fichier `linux/include/net/sock.h`:

```
73 /* On a un verrou par socket. Ce verrou tournant fournit une synchronisation
74  * entre les contextes utilisateur et les processus d'interruptions logicielles, alors
75  * que le mini-semaphore synchronise les utilisateurs multiple entre eux.
76  */
77 struct sock_iocb;
78 typedef struct {
79         spinlock_t              slock;
80         struct sock_iocb        *owner;
81         wait_queue_head_t       wq;
82 } socket_lock_t;
```

- Le champ `sk_rcvbuf` indique la taille en octets du tampon de réception.
- Le champ `sk_sleep` spécifie l'adresse de la file d'attente des entités en attente d'opération sur le descripteur de couche transport.

Code Linux 2.6.10

Le type général `wait_queue_head_t` est défini dans le fichier `linux/include/linux/wait.h`:

```
51 struct __wait_queue_head {
52         spinlock_t lock;
53         struct list_head task_list;
54 };
55 typedef struct __wait_queue_head wait_queue_head_t;
```

- Le champ `sk_dst_cache` spécifie une entrée dans le cache de destination.
- Le verrou `sk_dst_lock` indique si on peut utiliser ce cache.

– Le champ sk_policy[] indique la politique pour chacun des deux flux d'entrée et de sortie.

Le type struct xfrm_policy, qui ne nous intéressera pas ici, est défini dans le fichier linux/include/net/xfrm.h:

Code Linux 2.6.10

```
262 struct xfrm_policy
263 {
264         struct xfrm_policy      *next;
265         struct list_head        list;
266
267         /* Ce verrou affecte seulement les elements extirpes pour l'entree. */
268         rwlock_t                lock;
269         atomic_t                refcnt;
270         struct timer_list       timer;
271
272         u32                     priority;
273         u32                     index;
274         struct xfrm_selector    selector;
275         struct xfrm_lifetime_cfg lft;
276         struct xfrm_lifetime_cur curlft;
277         struct dst_entry       *bundles;
278         __u16                   family;
279         __u8                    action;
280         __u8                    flags;
281         __u8                    dead;
282         __u8                    xfrm_nr;
283         struct xfrm_tmpl        xfrm_vec[XFRM_MAX_DEPTH];
284 };
```

– Le champ sk_rmem_alloc permet d'indiquer que des données arrivent.
– Le champ sk_receive_queue spécifie la file d'attente des paquets en réception.
– Le champ sk_wmem_alloc permet d'indiquer que des données doivent être transmises.
– Le champ sk_write_queue spécifie la file d'attente des paquets qui attendent d'être envoyés.
– Le champ sk_omem_alloc permet d'indiquer que l'on attend quelque chose d'autre.
– Le champ sk_wmem_queued spécifie la taille de la file d'attente des paquets à envoyer.
– Le champ sk_forward_alloc spécifie l'espace mémoire alloué pour la retransmission.
– Le champ sk_allocation spécifie le mode d'allocation.
– Le champ sk_sndbuf spécifie la taille du tampon d'envoi, en octets.
– Le champ sk_no_check spécifie si on doit vérifier les paquets ou non.
– Le champ sk_debug spécifie si on doit déboguer.
– Le champ sk_rcvtstamp spécifie si on doit utiliser une estampille temporelle ou non.
– Le champ sk_no_largesend spécifie si on peut envoyer de grands segments ou non.
– Le champ sk_route_caps spécifie les capacités de routage.
– Le champ sk_hashent spécifie l'entrée dans la table de hachage.
– Le champ sk_backlog est un champ spécial.
– Le champ sk_callback_lock permet de synchroniser.
– Le champ sk_error_queue spécifie la file d'attente des erreurs. Il est rarement utilisé.
– Le champ sk_prot spécifie l'ensemble des opérations faisant intervenir ce type de descripteur pour la famille de protocoles et le type de communication de la socket.
– Le champ sk_err spécifie la dernière erreur intervenue.
– Le champ sk_err_soft spécifie le dernier avertissement.
– Le champ sk_ack_backlog spécifie le numéro de client en cours.
– Le champ sk_max_ack_backlog spécifie le nombre maximal de clients qui peuvent se connecter simultanémment.

– Le champ `sk_type` spécifie le type de communication (UDP pour ce qui nous intéresse).
– Le champ `sk_protocol` spécifie le protocole pour la famille d'adresses et le type de communication (toujours égal à 0 pour `AF_INET`).
– Le champ `sk_filter` spécifie les instructions de filtrage.

Code Linux 2.6.10

Le type `struct sk_filter` est défini dans le fichier `linux/include/linux/filter.h`:

```
21 /*
22  *      On essaie de garder ces valeurs et structures semblables a BSD, en particulier
23  *      pour les definitions du code BPF qui ont besoin de concorder, ainsi pouvez-vous
        partager les filtres
24  */
25
26 struct sock_filter      /* Bloc de filtre */
27 {
28         __u16   code;   /* Code de filtre actuel */
29         __u8    jt;     /* saut si vrai */
30         __u8    jf;     /* saut si faux */
31         __u32   k;      /* Champ multiusage generique */
32 };
[...]
40 #ifdef __KERNEL__
41 struct sk_filter
42 {
43         atomic_t                refcnt;
44         unsigned int            len;    /* Nombre de blocs de filtre */
45         struct sock_filter      insns[0];
46 };
```

– Le champ `sk_protinfo` est spécifique à la famille de protocole, utile lorsqu'on n'utilise pas le champ suivant.
– Le champ `sk_slab` spécifie où se trouve le cache.
– Le champ `sk_timer` spécifie le minuteur de nettoyage de ce descripteur.
– Le champ `sk_stamp` spécifie l'heure d'arrivée du dernier paquet.
– Le champ `sk_socket` spécifie le descripteur de socket associé à ce descripteur de couche de transport.
– Le champ `sk_user_data` spécifie l'emplacement des données privées de la couche RPC.
– Le champ `sk_owner` spécifie le module qui possède cette socket.

24.1.4 Attributs associés aux options des sockets

Nous avons vu que l'utilisateur pouvait choisir des options pour les sockets. L'état de celles-ci est reporté dans le descripteur de couche transport. Il s'agit des champs suivants :

– Le champ `sk_flags` est le vecteur des drapeaux, les valeurs étant celles utilisées par la fonction utilisateur `setsockopt()`.
– Le champ `sk_lingertime` spécifie si on a positionné `SO_LINGER` ou non.
– Le champ `sk_priority` spécifie si `SO_PRIORITY` est positionné ou non.
– Le champ `sk_localroute` spécifie si `SO_DONTROUTE` est positionné ou non.
– Le champ `sk_peercred` spécifie si `SO_PEERCRED` est positionné ou non.
– Le champ `sk_rcvlowat` spécifie si `SO_RCVLOWAT` est positionné ou non.
– Le champ `sk_rcvtimeo` spécifie si `SO_RCVTIMEO` est positionné ou non.
– Le champ `sk_sndtimeo` spécifie si `SO_SNDTIMEO` est positionné ou non.

24.1.5 Fonctions membre

Rappelons que Linux est conçu comme un programme orienté objet dont les structures comprennent des attributs et des méthodes. Les méthodes sont les suivantes:

- la fonction `sk_state_change()` permet d'indiquer un changement d'état;
- la fonction `sk_data_ready()` permet de savoir s'il y a des données à traiter;
- la fonction `sk_write_space()` permet de savoir s'il y a de la place dans la file d'attente d'envoi;
- la fonction `sk_error_report()` permet d'indiquer les erreurs;
- la fonction `sk_backlog_rcv()` permet de traiter les clients en attente;
- la fonction `sk_destruct()` permet de détruire le descripteur.

24.2 Allocation et libération

Les descripteurs de couche transport sont instantiés grâce à la fonction `sk_alloc()`. Chaque descripteur maintient un compteur de référence: les fonctions `sock_hold()` et `sock_put()` permettent d'incrémenter et de décrémenter celui-ci. Lorsque que le compteur de référence atteint zéro, le descripteur peut être libéré grâce à la fonction `sk_free()`.

24.2.1 Allocation

La fonction `sk_alloc()` est définie dans le fichier `linux/net/core/sock.c`: Code Linux 2.6.10

```
606 static kmem_cache_t *sk_cachep;
607
608 /**
609  *      sk_alloc - Tous les objets socket sont alloues ici
610  *      @family - famille de protocoles
611  *      @priority - pour l'allocation (%GFP_KERNEL, %GFP_ATOMIC, etc)
612  *      @zero_it - met a zero le descripteur alloue
613  *      @slab - slab alternatif
614  *
615  *      Tous les objets socket sont alloues ici. Si @zero_it est non nul,
616  *      elle doit contenir la taille de ce qui doit etre mis a zero, puisque les
617  *      caches slab prives ont des tailles differentes de la struct sock generique.
618  *      1 a ete garde comme une facon de dire sizeof(struct sock).
619  */
620 struct sock *sk_alloc(int family, int priority, int zero_it, kmem_cache_t *slab)
621 {
622         struct sock *sk = NULL;
623
624         if (!slab)
625                 slab = sk_cachep;
626         sk = kmem_cache_alloc(slab, priority);
627         if (sk) {
628                 if (zero_it) {
629                         memset(sk, 0,
630                                zero_it == 1 ? sizeof(struct sock) : zero_it);
631                         sk->sk_family = family;
632                         sock_lock_init(sk);
633                 }
634                 sk->sk_slab = slab;
635
636                 if (security_sk_alloc(sk, family, priority)) {
637                         kmem_cache_free(slab, sk);
638                         sk = NULL;
```

```
639                 }
640         }
641         return sk;
642 }
```

Autrement dit :

- On déclare une adresse de descripteur de couche transport, que l'on initialise à NULL.
- Si l'emplacement de l'antémémoire n'est pas spécifié, on choisit celle par défaut, correspondant à l'emplacement de la variable sk_cachep, déclarée juste avant la définition de la fonction.

  Cette variable est initialisée dans la fonction sk_init(), définie dans le fichier linux/net/core/sock.c:

Code Linux 2.6.10

```
669 void __init sk_init(void)
670 {
671         sk_cachep = kmem_cache_create("sock", sizeof(struct sock), 0,
672                                       SLAB_HWCACHE_ALIGN, NULL, NULL);
673         if (!sk_cachep)
674                 printk(KERN_CRIT "sk_init: Cannot create sock SLAB cache!");
675
676         if (num_physpages <= 4096) {
677                 sysctl_wmem_max = 32767;
678                 sysctl_rmem_max = 32767;
679                 sysctl_wmem_default = 32767;
680                 sysctl_rmem_default = 32767;
681         } else if (num_physpages >= 131072) {
682                 sysctl_wmem_max = 131071;
683                 sysctl_rmem_max = 131071;
684         }
685 }
```

  Les variables sysctl_wmem_max, sysctl_rmem_max, sysctl_rmem_default et sysctl_wmem_default sont définies plus haut dans le même fichier :

Code Linux 2.6.10

```
130 /* Prendre en consideration la taille de l'en-tete de struct sk_buff dans la
131  * determination de ces valeurs, puisque ceci n'est pas constant suivant la
132  * plate-forme. Ceci fait que le comportement de la mise en file d'attente des sockets
133  * et la performance de dependent pas de telles differences.
134  */
135 #define _SK_MEM_PACKETS         256
136 #define _SK_MEM_OVERHEAD        (sizeof(struct sk_buff) + 256)
137 #define SK_WMEM_MAX             (_SK_MEM_OVERHEAD * _SK_MEM_PACKETS)
138 #define SK_RMEM_MAX             (_SK_MEM_OVERHEAD * _SK_MEM_PACKETS)
139
140 /* Parametres ajustable en execution. */
141 __u32 sysctl_wmem_max     = SK_WMEM_MAX;
142 __u32 sysctl_rmem_max     = SK_RMEM_MAX;
143 __u32 sysctl_wmem_default = SK_WMEM_MAX;
144 __u32 sysctl_rmem_default = SK_RMEM_MAX;
```

  dont l'initialisation ici ne sert à rien puisque nous venons de voir que les valeurs des variables sont surchargées dans la fonction sk_init().
- On essaie de réserver de la place en mémoire vive pour le descripteur de couche transport. Si on n'y parvient pas, on renvoie la valeur NULL.
- Si on a spécifié en argument de mettre à zéro :
  - On initialise cet emplacement mémoire à zéro.
  - On renseigne le champ famille de protocoles de ce descripteur, grâce à la valeur passée en paramètre (UDP pour ce qui nous intéresse).

– On verrouille le descripteur pour qu'il ne puisse plus être initialisé à nouveau.

La macro `sock_lock_init()` est définie dans le fichier `linux/include/net/sock.h`: Code Linux 2.6.10

```
84 #define sock_lock_init(__sk) \
85 do {    spin_lock_init(&((__sk)->sk_lock.slock)); \
86         (__sk)->sk_lock.owner = NULL; \
87         init_waitqueue_head(&((__sk)->sk_lock.wq)); \
88 } while(0)
```

– On renseigne le champ du descripteur spécifiant l'emplacement de l'antémémoire.
– On renvoie l'adresse du descripteur ainsi créé.

24.2.2 Gestion du compteur de référence

24.2.2.1 Incrémentation

La fonction `sock_hold()` est définie dans le fichier `linux/include/net/sock.h`: Code Linux 2.6.10

```
318 /* Met la main sur le compteur de reference d'une socket. Cette operation n'est valide que
319    lorsque sk a DEJA ete agrippe, par exemple elle est trouvee dans une table de hachage
320    ou dans une liste et la consultation est faite sous verrouillage empechant les
321    modifications de la table de hachage.
322  */
323
324 static inline void sock_hold(struct sock *sk)
325 {
326         atomic_inc(&sk->sk_refcnt);
327 }
[...]
871 /*
872  * Postulats de denombrement des references a une socket.
873  *
874  * * Chaque utilisateur d'une socket DOIT detenir un compteur de reference.
875  * * Chaque point d'acces a une socket (une chaine d'une table de hachage, une reference a
       une liste,
876  *   un minuteur en fonctionnement, un skb en vol) DOIT detenir un compteur de reference.
877  * * Lorsque le compteur de reference atteint 0, ceci signifie qu'il ne s'accroitra plus.
878  * * Lorsque le compteur de reference atteint 0, ceci signifie qu'aucune references a
879  *   la socket provenant de l'exterieur n'existe et que le processus en cours sur le
880  *   microprocesseur en cours en est le dernier utilisateur et qu'il peut/doit detruire cette
       socket.
881  * * sk_free est appelee depuis les contextes suivants : processus, BH, IRQ. Lorsqu'elle
882  *   est appelee, la socket n'a aucune reference provenant de l'exterieur -> sk_free
883  *   peut liberer les ressources descendantes allouees par la socket, mais
884  *   au moment ou elle est appelee, la socket N'est referencee par aucune
885  *   table de hachage, liste etc.
886  * * Les paquets provenant de l'exterieur (par le reseau ou provenant d'un autre processus)
887  *   et mis dans la file d'attente en reception/erreur NE DEVRAIENT PAS mettre la main
       sur le compteur de reference
888  *   lorsqu'ils s'inserent dans la file. Autrement les paquets s'echapperont par le trou
889  *   lorsque la socket sera consultee par une cpu et que la decrementation est faite par une
       autre CPU.
890  *   Ceci est vrai pour udp/raw, netlink (fuite pour les files d'attente en reception et en
       erreur), tcp
891  *   (fuite en nombre de clients). Une socket de paquet effectue toutes ses taches dans
892  *   BR_NETPROTO_LOCK, aussi n'a-t-elle pas cette condition de course. Les sockets UNIX
893  *   utilisent un verrou SMP independant, aussi en sont-elles egalement sujettes.
894  */
```

Autrement dit on incrémente de 1 le compteur de référence du descripteur de couche réseau en prenant les précautions pour que l'opération soit atomique.

24.2.2.2 Décrémentation

Code Linux 2.6.10

La fonction `sock_put()` est définie dans le fichier `linux/include/net/sock.h`:

```
896 /* Se detacher d'une socket et la detruire, si c'etait la derniere reference. */
897 static inline void sock_put(struct sock *sk)
898 {
899         if (atomic_dec_and_test(&sk->sk_refcnt))
900                 sk_free(sk);
901 }
```

Autrement dit on décrémente de 1 le compteur de référence du descripteur de couche réseau.

Nous allons étudier la fonction `sk_free()` dans la sous-section suivante.

24.2.3 Libération

24.2.3.1 Fonction de libération

Code Linux 2.6.10

La fonction `sk_free()` est définie dans le fichier `linux/net/core/sock.c`:

```
644 void sk_free(struct sock *sk)
645 {
646         struct sk_filter *filter;
647         struct module *owner = sk->sk_owner;
648
649         if (sk->sk_destruct)
650                 sk->sk_destruct(sk);
651
652         filter = sk->sk_filter;
653         if (filter) {
654                 sk_filter_release(sk, filter);
655                 sk->sk_filter = NULL;
656         }
657
658         sock_disable_timestamp(sk);
659
660         if (atomic_read(&sk->sk_omem_alloc))
661                 printk(KERN_DEBUG "%s: optmem leakage (%d bytes) detected.\n",
662                        __FUNCTION__, atomic_read(&sk->sk_omem_alloc));
663
664         security_sk_free(sk);
665         kmem_cache_free(sk->sk_slab, sk);
666         module_put(owner);
667 }
```

Autrement dit :

- On déclare un filtre de socket.
- On déclare un propriétaire du module, que l'on initialise avec le propriétaire du descripteur de couche transport passé en argument.
- Si une fonction spécifique de destruction a été déclarée dans le descripteur de couche transport, on fait appel à elle.
- On instantie le filtre avec celui du descripteur de couche transport passé en argument. S'il est non nul, on le libère grâce à la fonction auxiliaire `sk_filter_release()` et on met à zéro le champ filtre du descripteur.

  La fonction en ligne `sk_filter_release()` est définie dans le fichier `linux/include/net/sock.h`:

Code Linux 2.6.10

```
847 /**
```

```
848  *      sk_filter_release : supprime un filtre de socket
849  *      @sk : socket
850  *      @fp : filtre a supprimer
851  *
852  *      Supprime un filtre d'une socket et libere ses ressources.
853  */
854
855 static inline void sk_filter_release(struct sock *sk, struct sk_filter *fp)
856 {
857         unsigned int size = sk_filter_len(fp);
858
859         atomic_sub(size, &sk->sk_omem_alloc);
860
861         if (atomic_dec_and_test(&fp->refcnt))
862                 kfree(fp);
863 }
```

La fonction en ligne `sk_filter_len()` est définie dans le fichier `linux/include/linux/filter.h`: Code Linux 2.6.10

```
48 static inline unsigned int sk_filter_len(struct sk_filter *fp)
49 {
50         return fp->len*sizeof(struct sock_filter) + sizeof(*fp);
51 }
```

– On inhibe le minuteur du descripteur de couche transport.

La fonction `sock_disable_timestamp()` est définie dans le fichier `linux/net/core/sock.c`: Code Linux 2.6.10

```
178 static void sock_disable_timestamp(struct sock *sk)
179 {
180         if (sock_flag(sk, SOCK_TIMESTAMP)) {
181                 sock_reset_flag(sk, SOCK_TIMESTAMP);
182                 net_disable_timestamp();
183         }
184 }
```

Les fonctions en ligne `sock_flag()` et `sock_reset_flag()` sont définies dans le fichier `linux/include/net/sock.h`: Code Linux 2.6.10

```
401 static inline void sock_reset_flag(struct sock *sk, enum sock_flags flag)
402 {
403         __clear_bit(flag, &sk->sk_flags);
404 }
405
406 static inline int sock_flag(struct sock *sk, enum sock_flags flag)
407 {
408         return test_bit(flag, &sk->sk_flags);
409 }
```

La fonction `net_disable_timestamp()` est définie dans le fichier `linux/net/core/dev.c`: Code Linux 2.6.10

```
1004 /* Lorsque > 0 il y a des consommateurs d'estampillles temporelle skb rx */
1005 static atomic_t netstamp_needed = ATOMIC_INIT(0);
[...]
1012 void net_disable_timestamp(void)
1013 {
1014         atomic_dec(&netstamp_needed);
1015 }
```

– Si le champ `sk_omem_alloc` du descripteur de couche transport est non nul, on ne devrait pas libérer le descripteur. On affiche donc un message noyau d'erreur.

- On fait éventuellement appel à la fonction de sécurité de libération du descripteur de couche transport. La fonction `security_sk_free()` ne nous intéressera pas dans cet ouvrage.
- On libère la mémoire vive qui était occupée par le descripteur.
- On décrémente le nombre d'utilisateurs du module propriétaire, pour qu'il puisse être libéré si besoin est.

24.2.3.2 Macro de libération

Code Linux 2.6.10

La fonction `release_sock()` est définie dans le fichier `linux/net/core/sock.c`:

```
1217 void fastcall release_sock(struct sock *sk)
1218 {
1219         spin_lock_bh(&(sk->sk_lock.slock));
1220         if (sk->sk_backlog.tail)
1221                 __release_sock(sk);
1222         sk->sk_lock.owner = NULL;
1223         if (waitqueue_active(&(sk->sk_lock.wq)))
1224                 wake_up(&(sk->sk_lock.wq));
1225         spin_unlock_bh(&(sk->sk_lock.slock));
1226 }
```

Code Linux 2.6.10

qui renvoie à la fonction `__release_sock()` définie dans le même fichier :

```
932 static void __release_sock(struct sock *sk)
933 {
934         struct sk_buff *skb = sk->sk_backlog.head;
935
936         do {
937                 sk->sk_backlog.head = sk->sk_backlog.tail = NULL;
938                 bh_unlock_sock(sk);
939
940                 do {
941                         struct sk_buff *next = skb->next;
942
943                         skb->next = NULL;
944                         sk->sk_backlog_rcv(sk, skb);
945                         skb = next;
946                 } while (skb != NULL);
947
948                 bh_lock_sock(sk);
949         } while((skb = sk->sk_backlog.head) != NULL);
950 }
```

24.3 Gestion des données associées

24.3.1 Initialisation des données

Code Linux 2.6.10

La fonction `sock_init_data()` est définie dans le fichier `linux/net/core/sock.c`:

```
1153 void sock_init_data(struct socket *sock, struct sock *sk)
1154 {
1155         skb_queue_head_init(&sk->sk_receive_queue);
1156         skb_queue_head_init(&sk->sk_write_queue);
1157         skb_queue_head_init(&sk->sk_error_queue);
1158
1159         sk->sk_send_head        =       NULL;
1160
1161         init_timer(&sk->sk_timer);
1162
1163         sk->sk_allocation       =       GFP_KERNEL;
```

```
1164        sk->sk_rcvbuf           =       sysctl_rmem_default;
1165        sk->sk_sndbuf           =       sysctl_wmem_default;
1166        sk->sk_state            =       TCP_CLOSE;
1167        sk->sk_zapped           =       1;
1168        sk->sk_socket           =       sock;
1169
1170        if(sock)
1171        {
1172                sk->sk_type     =       sock->type;
1173                sk->sk_sleep    =       &sock->wait;
1174                sock->sk        =       sk;
1175        } else
1176                sk->sk_sleep    =       NULL;
1177
1178        rwlock_init(&sk->sk_dst_lock);
1179        rwlock_init(&sk->sk_callback_lock);
1180
1181        sk->sk_state_change     =       sock_def_wakeup;
1182        sk->sk_data_ready       =       sock_def_readable;
1183        sk->sk_write_space      =       sock_def_write_space;
1184        sk->sk_error_report     =       sock_def_error_report;
1185        sk->sk_destruct         =       sock_def_destruct;
1186
1187        sk->sk_sndmsg_page      =       NULL;
1188        sk->sk_sndmsg_off       =       0;
1189
1190        sk->sk_peercred.pid     =       0;
1191        sk->sk_peercred.uid     =       -1;
1192        sk->sk_peercred.gid     =       -1;
1193        sk->sk_write_pending    =       0;
1194        sk->sk_rcvlowat         =       1;
1195        sk->sk_rcvtimeo         =       MAX_SCHEDULE_TIMEOUT;
1196        sk->sk_sndtimeo         =       MAX_SCHEDULE_TIMEOUT;
1197        sk->sk_owner            =       NULL;
1198
1199        sk->sk_stamp.tv_sec     = -1L;
1200        sk->sk_stamp.tv_usec    = -1L;
1201
1202        atomic_set(&sk->sk_refcnt, 1);
1203 }
```

Autrement dit :

- On initialise les trois champs files d'attente du descripteur de couche transport passé en argument.
- On renseigne le champ `sk_send_head` avec l'adresse nulle.
- On initialise le champ liste des minuteurs.
- On indique que ce descripteur doit se trouver dans l'espace mémoire noyau.
- On indique où se trouveront les données du tampon, en réception et en émission.
- On spécifie l'état de la socket, qui est fermée pour l'instant.
- On spécifie que le descripteur est zappé (rappelons que ce champ n'a d'intérêt que pour les familles de protocoles AX25 et IPX).
- On spécifie comme descripteur de socket associé au descripteur de couche transport celui dont l'adresse est passée en paramètre :
  - Si cette adresse est non nulle :
    - On renseigne le champ type du descripteur de couche transport avec celui du descripteur de socket.

- La file des entités en attente d'opérations sur le descripteur de couche transport est instantiée avec celle associée au descripteur de socket.
- On indique que le descripteur de couche transport associé au descripteur de socket est celui que l'on est en train de renseigner.

– sinon le champ `sk_sleep` du descripteur de couche transport est initialisé à zéro.

– On initialise les autres champs du descripteur de couche transport.

La constante symbolique générale `MAX_SCHEDULE_TIMEOUT` est définie dans le fichier `linux/include/linux/sched.h`:

Code Linux 2.6.10

```
180 #define MAX_SCHEDULE_TIMEOUT    LONG_MAX
```

– On initialise le compteur de référence du descripteur de couche réseau à 1.

24.3.2 Verrouillage

La fonction `lock_sock()` est définie dans le fichier `linux/net/core/sock.c`:

Code Linux 2.6.10

```
1205 void fastcall lock_sock(struct sock *sk)
1206 {
1207         might_sleep();
1208         spin_lock_bh(&(sk->sk_lock.slock));
1209         if (sk->sk_lock.owner)
1210                 __lock_sock(sk);
1211         sk->sk_lock.owner = (void *)1;
1212         spin_unlock_bh(&(sk->sk_lock.slock));
1213 }
```

précédemment définie dans le fichier `linux/include/net/sock.h`, précédée d'un commentaire:

Code Linux 2.6.0

```
529 /* Utilise par les processus pour "verrouiller" un etat de socket de facon a ce que
530  * les routines d'interruptions et de partie basse ne puissent pas le changer
531  * a notre insu. Il bloque essentiellement tous les paquets entrants
532  * de facon a ce que nous n'obtenions pas de nouvelles donnees ou de
533  * nouveaux paquets qui changent l'etat de la socket.
534  *
535  * Tant qu'il est verrouille, le processus BH ajoutera les nouveaux paquets a
536  * la file d'attente de nouveaux clients. Cette file d'attente est manipulee par le
537  * proprietaire de la socket verrouillee juste avant qu'elle ne soit liberee.
538  *
539  * Depuis ~2.3.5 elle est egalement verrouillee pour les acces en serie
540  * dans le contexte des processus utilisateur.
541  */
```

Autrement dit:

– On indique que cette activitée peut être assoupie.
– On verrouille l'état de la socket.
– Si le module propriétaire n'est pas verrouillé, on fait appel à la fonction auxiliaire `__lock_sock()`.
– On verrouille le module.
– On déverrouille l'état de la socket.

La fonction auxiliaire `__lock_sock()` est définie plus haut dans le même fichier:

Code Linux 2.6.10

```
916 static void __lock_sock(struct sock *sk)
917 {
918         DEFINE_WAIT(wait);
```

```
919
920         for(;;) {
921                 prepare_to_wait_exclusive(&sk->sk_lock.wq, &wait,
922                                           TASK_UNINTERRUPTIBLE);
923                 spin_unlock_bh(&sk->sk_lock.slock);
924                 schedule();
925                 spin_lock_bh(&sk->sk_lock.slock);
926                 if(!sock_owned_by_user(sk))
927                         break;
928         }
929         finish_wait(&sk->sk_lock.wq, &wait);
930 }
```

qui fait essentiellement appel à l'ordonnanceur.

24.3.3 Initialisation du délai

La fonction `sock_sndtimeo()` est définie dans le fichier `linux/include/net/sock.h`: Code Linux 2.6.10

```
1228 static inline long sock_sndtimeo(const struct sock *sk, int noblock)
1229 {
1230         return noblock ? 0 : sk->sk_sndtimeo;
1231 }
```

Chapitre 25

Réception des datagrammes UDP

Après une étude générale de l'implémentation Linux de la couche de transport UDP (en-tête et calcul de la somme de contrôle), on étudie la réception des datagrammes dans la couche de transport UDP.

25.1 Implémentation Linux générale d'UDP

25.1.1 Implémentation Linux de l'en-tête UDP

L'en-tête UDP est une entité du type `struct udphdr`. Celui-ci est défini dans le fichier en-tête `linux/include/linux/udp.h`:

Code Linux 2.6.10

```
6   *              Definitions pour le protocole UDP.
7   *
8   * Version :    @(#)udp.h       1.0.2   04/28/93
9   *
10  * Auteur :     Fred N. van Kempen, <waltje@uWalt.NL.Mugnet.ORG>
[...]
17 #ifndef _LINUX_UDP_H
18 #define _LINUX_UDP_H
19
20 #include <linux/types.h>
21
22 struct udphdr {
23         __u16   source;
24         __u16   dest;
25         __u16   len;
26         __u16   check;
27 };
```

25.1.2 Calcul de la somme de contrôle

25.1.2.1 Calcul partiel de la somme de contrôle d'un bloc de mémoire

La fonction `csum_partial()` permet de calculer la somme de contrôle d'un bloc de mémoire. Elle dépend du microprocesseur. Elle est déclarée dans le fichier `linux/include/asm-i386/checksum.h` dans le cas des microprocesseurs Intel :

Code Linux 2.6.10

```
6  /*
7   * Calcule la somme de controle d'un bloc de memoire d'un buff, de longueur len,
8   * et l'ajoute a "sum" (32-bit)
9   *
10  * Renvoie un nombre de 32 bits adequat en lui-meme
11  * ou pour csum_tcpudp_magic
12  *
13  * Cette fonction doit etre appelee avec des longueurs paires, sauf pour
14  * le dernier fragment, dont la longueur peut etre impaire
15  *
16  * C'est mieux d'avoir des buff alignes sur 32 bits
17  */
18 asmlinkage unsigned int csum_partial(const unsigned char * buff, int len,
                                       unsigned int sum);
```

La définition de cette fonction, en langage d'assemblage, se trouve dans le fichier `linux/arch/i386/lib/checksum.S`:

Code Linux 2.6.10

```
6    *              Routines de calcul de la somme de controle pour IP/TCP/UDP
7    *
8    * Auteurs :    Jorge Cwik, <jorge@laser.satlink.net>
9    *              Arnt Gulbrandsen, <agulbra@nvg.unit.no>
10   *              Tom May, <ftom@netcom.com>
11   *              Routines Pentium Pro/II :
12   *              Alexander Kjeldaas <astor@guardian.no>
13   *              Finn Arne Gangstad <finnag@guardian.no>
14   *              Beaucoup de code a ete deplace depuis tcp.c et ip.c ; voir ces fichiers
15   *              pour plus de noms.
16   *
17   * Changements : Ingo Molnar, a converti csum_partial_copy() en routine d'exception
18   *                           2.1.
19   *              Andi Kleen, a ajoute la mise a zero en cas d'erreur
20   *                   a converti en langage d'assemblage pur
[...]
28  #include <linux/config.h>
29  #include <asm/errno.h>
30
31  /*
32   * calcule une somme de controle partielle, par exemple pour des fragments TCP/UDP
33   */
34
35   /*
36  unsigned int csum_partial(const unsigned char * buff, int len, unsigned int sum)
37   */
38
39  .text
40  .align 4
41  .globl csum_partial
42
43  #ifndef CONFIG_X86_USE_PPRO_CHECKSUM
44
45            /*
46             * Les experiences avec les connexions Ethernet et SLIP montrent que buff
47             * est aligne sur une frontiere 2 octets ou 4 octets. Nous obtiendrons au
48             * moins une acceleration double sur 486 et Pentium si c'est aligne sur 4 octets.
```

```
49              * Heureusement, il est facile de convertir un alignement 2 octets en un
50              * alignement 4 octets pour la boucle non enroulee.
51              */
52  csum_partial:
53          pushl %esi
54          pushl %ebx
55          movl 20(%esp),%eax      # Arg de fonction : unsigned int sum
56          movl 16(%esp),%ecx      # Arg de fonction : int len
57          movl 12(%esp),%esi      # Arg de fonction : unsigned char *buff
58          testl $3, %esi          # Verifie l'alignement.
59          jz 2f                   # Saut si l'alignement est ok.
60          testl $1, %esi          # Verifie l'alignement.
61          jz 10f                  # Saut si l'alignement est une frontiere de 2 octets.
62
63          # buf est impair
64          dec %ecx
65          jl 8f
66          movzbl (%esi), %ebx
67          adcl %ebx, %eax
68          roll $8, %eax
69          inc %esi
70          testl $2, %esi
71          jz 2f
72  10:
73          subl $2, %ecx           # L'alignement utilise jusqu'a deux octets.
74          jae 1f                  # Saut si nous avons au moins deux octets.
75          addl $2, %ecx           # ecx etait < 2. Faire avec ca.
76          jmp 4f
77  1:      movw (%esi), %bx
78          addl $2, %esi
79          addw %bx, %ax
80          adcl $0, %eax
81  2:
82          movl %ecx, %edx
83          shrl $5, %ecx
84          jz 2f
85          testl %esi, %esi
86  1:      movl (%esi), %ebx
87          adcl %ebx, %eax
88          movl 4(%esi), %ebx
89          adcl %ebx, %eax
90          movl 8(%esi), %ebx
91          adcl %ebx, %eax
92          movl 12(%esi), %ebx
93          adcl %ebx, %eax
94          movl 16(%esi), %ebx
95          adcl %ebx, %eax
96          movl 20(%esi), %ebx
97          adcl %ebx, %eax
98          movl 24(%esi), %ebx
99          adcl %ebx, %eax
100         movl 28(%esi), %ebx
101         adcl %ebx, %eax
102         lea 32(%esi), %esi
103         dec %ecx
104         jne 1b
105         adcl $0, %eax
106 2:      movl %edx, %ecx
107         andl $0x1c, %edx
108         je 4f
109         shrl $2, %edx           # Ceci met CF a zero
110 3:      adcl (%esi), %eax
```

```
111         lea 4(%esi), %esi
112         dec %edx
113         jne 3b
114         adcl $0, %eax
115 4:      andl $3, %ecx
116         jz 7f
117         cmpl $2, %ecx
118         jb 5f
119         movw (%esi),%cx
120         leal 2(%esi),%esi
121         je 6f
122         shll $16,%ecx
123 5:      movb (%esi),%cl
124 6:      addl %ecx,%eax
125         adcl $0, %eax
126 7:
127         testl $1, 12(%esp)
128         jz 8f
129         roll $8, %eax
130 8:
131         popl %ebx
132         popl %esi
133         ret
134
135 #else
136
137 /* Version pour PentiumII/PPro */
138
139 csum_partial:
140         pushl %esi
141         pushl %ebx
142         movl 20(%esp),%eax      # Arg de fonction : unsigned int sum
143         movl 16(%esp),%ecx      # Arg de fonction : int len
144         movl 12(%esp),%esi      # Arg de fonction : const unsigned char *buf
145
146         testl $3, %esi
147         jnz 25f
148 10:
149         movl %ecx, %edx
150         movl %ecx, %ebx
151         andl $0x7c, %ebx
152         shrl $7, %ecx
153         addl %ebx,%esi
154         shrl $2, %ebx
155         negl %ebx
156         lea 45f(%ebx,%ebx,2), %ebx
157         testl %esi, %esi
158         jmp *%ebx
159
160         # Manipule les regions alignees sur 2 octets
161 20:      addw (%esi), %ax
162         lea 2(%esi), %esi
163         adcl $0, %eax
164         jmp 10b
165 25:
166         testl $1, %esi
167         jz 30f
168         # buf is odd
169         dec %ecx
170         jl 90f
171         movzbl (%esi), %ebx
172         addl %ebx, %eax
```

```
173         adcl $0, %eax
174         roll $8, %eax
175         inc %esi
176         testl $2, %esi
177         jz 10b
178
179 30:     subl $2, %ecx
180         ja 20b
181         je 32f
182         addl $2, %ecx
183         jz 80f
184         movzbl (%esi),%ebx      # calcule la somme 1 octet, 2-aligne
185         addl %ebx, %eax
186         adcl $0, %eax
187         jmp 80f
188 32:
189         addw (%esi), %ax        # calcule la somme 2 octets, 2-aligne
190         adcl $0, %eax
191         jmp 80f
192
193 40:
194         addl -128(%esi), %eax
195         adcl -124(%esi), %eax
196         adcl -120(%esi), %eax
197         adcl -116(%esi), %eax
198         adcl -112(%esi), %eax
199         adcl -108(%esi), %eax
200         adcl -104(%esi), %eax
201         adcl -100(%esi), %eax
202         adcl -96(%esi), %eax
203         adcl -92(%esi), %eax
204         adcl -88(%esi), %eax
205         adcl -84(%esi), %eax
206         adcl -80(%esi), %eax
207         adcl -76(%esi), %eax
208         adcl -72(%esi), %eax
209         adcl -68(%esi), %eax
210         adcl -64(%esi), %eax
211         adcl -60(%esi), %eax
212         adcl -56(%esi), %eax
213         adcl -52(%esi), %eax
214         adcl -48(%esi), %eax
215         adcl -44(%esi), %eax
216         adcl -40(%esi), %eax
217         adcl -36(%esi), %eax
218         adcl -32(%esi), %eax
219         adcl -28(%esi), %eax
220         adcl -24(%esi), %eax
221         adcl -20(%esi), %eax
222         adcl -16(%esi), %eax
223         adcl -12(%esi), %eax
224         adcl -8(%esi), %eax
225         adcl -4(%esi), %eax
226 45:
227         lea 128(%esi), %esi
228         adcl $0, %eax
229         dec %ecx
230         jge 40b
231         movl %edx, %ecx
232 50:     andl $3, %ecx
233         jz 80f
234
```

```
235         # Manipule les derniers 1-3 octets sans saut
236         notl %ecx               # 1->2, 2->1, 3->0, les bits plus hauts sont masques
237         movl $0xffffff,%ebx     # par les instructions shll et shrl
238         shll $3,%ecx
239         shrl %cl,%ebx
240         andl -128(%esi),%ebx    # esi est 4-aligne donc devrait etre ok
241         addl %ebx,%eax
242         adcl $0,%eax
243 80:
244         testl $1, 12(%esp)
245         jz 90f
246         roll $8, %eax
247 90:
248         popl %ebx
249         popl %esi
250         ret
251
252 #endif
```

25.1.2.2 Calcul complet de la somme de contrôle

La fonction `skb_checksum(skb, offset, len, csum)` calcule la somme de contrôle `csum` de la partie des données du tampon `skb` qui commence au décalage `offset` et de longueur `len`. Elle est définie dans le fichier `linux/net/core/skbuff.c`:

Code Linux 2.6.10

```
996  /* Calcul de la somme de controle des donnees skb. */
997
998  unsigned int skb_checksum(const struct sk_buff *skb, int offset,
999                            int len, unsigned int csum)
1000 {
1001         int start = skb_headlen(skb);
1002         int i, copy = start - offset;
1003         int pos = 0;
1004
1005         /* Calcul de la somme de controle de l'en-tete. */
1006         if (copy > 0) {
1007                 if (copy > len)
1008                         copy = len;
1009                 csum = csum_partial(skb->data + offset, copy, csum);
1010                 if ((len -= copy) == 0)
1011                         return csum;
1012                 offset += copy;
1013                 pos     = copy;
1014         }
1015
1016         for (i = 0; i < skb_shinfo(skb)->nr_frags; i++) {
1017                 int end;
1018
1019                 BUG_TRAP(start <= offset + len);
1020
1021                 end = start + skb_shinfo(skb)->frags[i].size;
1022                 if ((copy = end - offset) > 0) {
1023                         unsigned int csum2;
1024                         u8 *vaddr;
1025                         skb_frag_t *frag = &skb_shinfo(skb)->frags[i];
1026
1027                         if (copy > len)
1028                                 copy = len;
1029                         vaddr = kmap_skb_frag(frag);
1030                         csum2 = csum_partial(vaddr + frag->page_offset +
1031                                              offset - start, copy, 0);
1032                         kunmap_skb_frag(vaddr);
```

```
1033                        csum = csum_block_add(csum, csum2, pos);
1034                        if (!(len -= copy))
1035                                return csum;
1036                        offset += copy;
1037                        pos    += copy;
1038                }
1039                start = end;
1040        }
1041
1042        if (skb_shinfo(skb)->frag_list) {
1043                struct sk_buff *list = skb_shinfo(skb)->frag_list;
1044
1045                for (; list; list = list->next) {
1046                        int end;
1047
1048                        BUG_TRAP(start <= offset + len);
1049
1050                        end = start + list->len;
1051                        if ((copy = end - offset) > 0) {
1052                                unsigned int csum2;
1053                                if (copy > len)
1054                                        copy = len;
1055                                csum2 = skb_checksum(list, offset - start,
1056                                                     copy, 0);
1057                                csum = csum_block_add(csum, csum2, pos);
1058                                if ((len -= copy) == 0)
1059                                        return csum;
1060                                offset += copy;
1061                                pos    += copy;
1062                        }
1063                        start = end;
1064                }
1065        }
1066        if (len)
1067                BUG();
1068
1069        return csum;
1070 }
```

Autrement dit on calcule la somme de contrôle partielle de l'en-tête à laquelle on ajoute les sommes de contrôle partielles des fragments du tableau des fragments puis celles des tampons de la liste des fragments.

Les fonctions auxiliaires `csum_add()` et `csum_block_add()` sont définies dans le fichier `linux/include/net/checksum.h`:

Code Linux 2.6.10

```
60 static inline unsigned int csum_add(unsigned int csum, unsigned int addend)
61 {
62         csum += addend;
63         return csum + (csum < addend);
64 }
[...]
71 static inline unsigned int
72 csum_block_add(unsigned int csum, unsigned int csum2, int offset)
73 {
74         if (offset&1)
75                 csum2 = ((csum2&0xFF00FF)<<8)+((csum2>>8)&0xFF00FF);
76         return csum_add(csum, csum2);
77 }
```

25.2 Réception des datagrammes UDP

Nous avons vu au chapitre 23 que le paquet reçu par la couche IP est transmis à la couche transport, en traitant le descripteur de tampon grâce à la méthode `handler()` de l'instance de structure `inet_protocol` associée au type de communication. Nous avons vu aussi que dans le cas d'UDP, cette instance s'appelle `udp_protocol` et le nom de la fonction `udp_rcv()`.

25.2.1 Fonction de traitement principale

Code Linux 2.6.10

La fonction `udp_rcv()` est définie dans le fichier `linux/net/ipv4/udp.c`:

```
1110 /*
1111  *      Tout ce que nous avons besoin de faire est d'obtenir la socket, et alors
              d'effectuer une verification de la somme de controle.
1112  */
1113
1114 int udp_rcv(struct sk_buff *skb)
1115 {
1116         struct sock *sk;
1117         struct udphdr *uh;
1118         unsigned short ulen;
1119         struct rtable *rt = (struct rtable*)skb->dst;
1120         u32 saddr = skb->nh.iph->saddr;
1121         u32 daddr = skb->nh.iph->daddr;
1122         int len = skb->len;
1123
1124         /*
1125          *      Valider le paquet et la longueur UDP.
1126          */
1127         if (!pskb_may_pull(skb, sizeof(struct udphdr)))
1128                 goto no_header;
1129
1130         uh = skb->h.uh;
1131
1132         ulen = ntohs(uh->len);
1133
1134         if (ulen > len || ulen < sizeof(*uh))
1135                 goto short_packet;
1136
1137         if (pskb_trim(skb, ulen))
1138                 goto short_packet;
1139
1140         if (udp_checksum_init(skb, uh, ulen, saddr, daddr) < 0)
1141                 goto csum_error;
1142
1143         if(rt->rt_flags & (RTCF_BROADCAST|RTCF_MULTICAST))
1144                 return udp_v4_mcast_deliver(skb, uh, saddr, daddr);
1145
1146         sk = udp_v4_lookup(saddr, uh->source, daddr, uh->dest, skb->dev->ifindex);
1147
1148         if (sk != NULL) {
1149                 int ret = udp_queue_rcv_skb(sk, skb);
1150                 sock_put(sk);
1151
1152                 /* une valeur de retour > 0 signifie de soumettre a nouveau l'entree,
1153                  * mais il faut que le retour soit -protocol ou 0
1154                  */
1155                 if (ret > 0)
1156                         return -ret;
1157                 return 0;
1158         }
```

```
1159
1160        if (!xfrm4_policy_check(NULL, XFRM_POLICY_IN, skb))
1161                goto drop;
1162
1163        /* Pas de socket. Ecarter le paquet sans faire de bruit si la somme de controle
               est fausse */
1164        if (udp_checksum_complete(skb))
1165                goto csum_error;
1166
1167        UDP_INC_STATS_BH(UDP_MIB_NOPORTS);
1168        icmp_send(skb, ICMP_DEST_UNREACH, ICMP_PORT_UNREACH, 0);
1169
1170        /*
1171         * Hum. Nous obtenons un paquet UDP pour un port qui n'est pas a
1172         * l'ecoute. Ignorons-le.
1173         */
1174        kfree_skb(skb);
1175        return(0);
1176
1177 short_packet:
1178        NETDEBUG(if (net_ratelimit())
1179                printk(KERN_DEBUG "UDP: short packet: From %u.%u.%u.%u:%u %d/%d
                                                          to %u.%u.%u.%u:%u\n",
1180                       NIPQUAD(saddr),
1181                       ntohs(uh->source),
1182                       ulen,
1183                       len,
1184                       NIPQUAD(daddr),
1185                       ntohs(uh->dest)));
1186 no_header:
1187        UDP_INC_STATS_BH(UDP_MIB_INERRORS);
1188        kfree_skb(skb);
1189        return(0);
1190
1191 csum_error:
1192        /*
1193         * RFC1122 : OK. Ecarter le mauvais paquet sans faire de bruit (pour autant que
1194         * le reseau est concerne) comme pour 4.1.3.4 (MUST).
1195         */
1196        NETDEBUG(if (net_ratelimit())
1197                 printk(KERN_DEBUG "UDP: bad checksum. From %d.%d.%d.%d:%d
                                                           to %d.%d.%d.%d:%d ulen %d\n",
1198                       NIPQUAD(saddr),
1199                       ntohs(uh->source),
1200                       NIPQUAD(daddr),
1201                       ntohs(uh->dest),
1202                       ulen));
1203 drop:
1204        UDP_INC_STATS_BH(UDP_MIB_INERRORS);
1205        kfree_skb(skb);
1206        return(0);
1207 }
```

Autrement dit :

- On déclare une adresse de descripteur de couche transport, un en-tête UDP et une longueur de datagramme UDP.
- On déclare une entrée de cache de routage, que l'on instantie avec celle associée au descripteur de tampon de socket passé en argument.

  Nous avons vu au chapitre 23 comment ce champ entrée de cache de routage a été renseigné.

- On déclare une adresse IP source, que l'on instantie avec celle associée à l'en-tête IP du descripteur de tampon passé en argument.
- On déclare une adresse IP destination, que l'on instantie avec celle associée à l'en-tête IP du descripteur de tampon passé en argument.
- On déclare une longueur, que l'on instantie avec celle associée au descripteur de tampon passé en argument.
- On vérifie la longueur du datagramme :
  - Si la longueur du datagramme est inférieure à la taille minimum d'un en-tête UDP, on incrémente le nombre d'erreurs sur les datagrammes UDP entrants, on libère le descripteur de tampon et on renvoie 0.

    Les informations statistiques sur UDP sont détenues dans une variable du type `udp_mib`. Celui-ci est défini dans le fichier en-tête `linux/include/net/snmp.h` :

Code Linux 2.6.10

```
97  /* UDP */
98  #define UDP_MIB_MAX     __UDP_MIB_MAX
99  struct udp_mib {
100         unsigned long   mibs[UDP_MIB_MAX];
101 } __SNMP_MIB_ALIGN__;
```

    La macro `UDP_INC_STATS_BH()` est définie dans le fichier en-tête `linux/include/net/udp.h` :

Code Linux 2.6.10

```
77 DECLARE_SNMP_STAT(struct udp_mib, udp_statistics);
78 #define UDP_INC_STATS(field)            SNMP_INC_STATS(udp_statistics, field)
79 #define UDP_INC_STATS_BH(field)         SNMP_INC_STATS_BH(udp_statistics, field)
80 #define UDP_INC_STATS_USER(field)       SNMP_INC_STATS_USER(udp_statistics, field)
```

  - On instantie l'adresse d'en-tête UDP avec celle du champ correspondant du descripteur de tampon passé en argument.
  - On initialise la longueur de datagramme UDP avec celle indiquée dans l'en-tête UDP.
  - Si cette longueur de datagramme UDP est plus grande que la longueur du datagramme décrit par le descripteur de tampon passé en argument ou est plus petite que la taille minimale d'un en-tête UDP, on affiche un message noyau, on incrémente le nombre d'erreurs sur les datagrammes UDP entrants, on libère le descripteur de tampon et on renvoie 0.
- On vérifie la somme de contrôle, grâce à la fonction `udp_checksum_init()` étudiée ci-après. Si elle ne concorde pas avec celle indiquée dans l'en-tête UDP, on affiche un message noyau, on incrémente le nombre d'erreurs sur les datagrammes UDP entrants, on libère le descripteur de tampon et on renvoie 0.
- Les datagrammes de multidiffusion et de diffusion générale sont transmis à la fonction `udp_v4_mcast_deliver()`, qui ne nous intéressera pas ici.
- On essaie d'instantier le descripteur de couche transport grâce à la fonction `udp_v4_lookup()` de consultation de la table `udp_hash`, qui sera étudiée ci-dessous.
- Si on y parvient :
  - On place ce descripteur dans la file d'attente UDP en réception dans laquelle il sera récupéré par l'utilisateur, grâce à la fonction `udp_queue_rcv_skb()` qui sera étudiée ci-dessous.
  - On décrémente le compteur de référence de ce descripteur de couche de transport.
  - Si une erreur est intervenue, on renvoie l'opposé du code d'erreur fourni par la fonction précédente. Sinon on renvoie 0.

- Si on n'est pas parvenu à instantier le descripteur de couche transport :
  - Si c'est dû à un problème de politique, on détruit le datagramme: on incrémente le nombre d'erreurs sur les datagrammes UDP entrants, on libère le descripteur de tampon et on renvoie 0.
  - Si ceci est dû à un problème de somme de contrôle, vérifié grâce à la fonction `udp_checksum_complete()`, on affiche un message noyau, on incrémente le nombre d'erreurs sur les datagrammes UDP entrants, on libère le descripteur de tampon et on renvoie 0.
  - Dans les autres cas, c'est que le port n'est pas accessible: on incrémente le nombre d'erreurs sur les datagrammes UDP entrants pour lesquels le port n'est pas accessible, on envoie un message ICMP, on libère le descripteur de tampon et on renvoie 0.

25.2.2 Vérification rapide de la somme de contrôle

La fonction `udp_checksum_init()` est définie dans le fichier `linux/net/ipv4/udp.c`: Code Linux 2.6.10

```
1085 /* Initialise la verification de la somme de controle UDP. Si on la quitte avec
        la valeur zero (succes),
1086  * CHECKSUM_UNNECESSARY signifie qu'on n'a pas besoin d'autres verifications.
1087  * Sinon la verification complete necessite le corps du paquet,
1088  * incluant l'en-tete udp et l'incorporant dans skb->csum.
1089  */
1090 static int udp_checksum_init(struct sk_buff *skb, struct udphdr *uh,
1091                              unsigned short ulen, u32 saddr, u32 daddr)
1092 {
1093         if (uh->check == 0) {
1094                 skb->ip_summed = CHECKSUM_UNNECESSARY;
1095         } else if (skb->ip_summed == CHECKSUM_HW) {
1096                 skb->ip_summed = CHECKSUM_UNNECESSARY;
1097                 if (!udp_check(uh, ulen, saddr, daddr, skb->csum))
1098                         return 0;
1099                 NETDEBUG(if (net_ratelimit()) printk(KERN_DEBUG "udp v4 hw csum
                                                        failure.\n"));
1100                 skb->ip_summed = CHECKSUM_NONE;
1101         }
1102         if (skb->ip_summed != CHECKSUM_UNNECESSARY)
1103                 skb->csum = csum_tcpudp_nofold(saddr, daddr, ulen, IPPROTO_UDP, 0);
1104         /* Nous devrions probablement verifier l'en-tete udp (il devrait etre dans le cache
1105          * en tout cas) et les donnees dans les petits paquets (< rx copybreak).
1106          */
1107         return 0;
1108 }
```

Autrement dit :

- Si la valeur de la somme de contrôle de l'en-tête UDP est nulle, c'est que l'émetteur n'a pas envoyé de somme de contrôle. On positionne alors le champ `ip_summed` du descripteur de tampon passé en argument à la valeur `CHECKSUM_UNNECESSARY`.
- Si la valeur de la somme de contrôle de l'en-tête UDP est non nulle et que la valeur du champ `ip_summed` du descripteur de tampon passé en argument est `CHECKSUM_HW`:
  - On positionne également le champ `ip_summed` du descripteur de tampon passé en argument à la valeur `CHECKSUM_UNNECESSARY`.
  - On vérifie qu'il s'agit de la bonne somme de contrôle grâce à la fonction `udp_check()` et on renvoie 0 en cas de succès.

Code Linux 2.6.10

La fonction udp_check() est définie dans le fichier linux/net/ipv4/udp.c:

```
475 static unsigned short udp_check(struct udphdr *uh, int len, unsigned long saddr,
                                    unsigned long daddr, unsigned long base)
476 {
477         return(csum_tcpudp_magic(saddr, daddr, len, IPPROTO_UDP, base));
478 }
```

qui renvoie à la fonction csum_tcpudp_magic(), qui dépend du microprocesseur. Celle-ci est définie dans le fichier linux/include/asm-i386/checksum.h pour les microprocesseurs Intel:

Code Linux 2.6.10

```
93  /*
94   *      Plie une somme de controle partielle
95   */
96
97  static inline unsigned int csum_fold(unsigned int sum)
98  {
99          __asm__(
100                 "addl %1, %0            ;\n"
101                 "adcl $0xffff, %0       ;\n"
102                 : "=r" (sum)
103                 : "r" (sum << 16), "" (sum & 0xffff0000)
104         );
105         return (~sum) >> 16;
106 }
107
108 static inline unsigned long csum_tcpudp_nofold(unsigned long saddr,
109                                                unsigned long daddr,
110                                                unsigned short len,
111                                                unsigned short proto,
112                                                unsigned int sum)
113 {
114     __asm__(
115         "addl %1, %0    ;\n"
116         "adcl %2, %0    ;\n"
117         "adcl %3, %0    ;\n"
118         "adcl $0, %0    ;\n"
119         : "=r" (sum)
120         : "g" (daddr), "g"(saddr), "g"((ntohs(len)<<16)+proto*256), ""(sum));
121     return sum;
122 }
123
124 /*
125  * calcule la somme de controle du pseudo en-tete TCP/UDP
126  * renvoie une somme de controle sur 16 bits, deja complementee
127  */
128 static inline unsigned short int csum_tcpudp_magic(unsigned long saddr,
129                                                unsigned long daddr,
130                                                unsigned short len,
131                                                unsigned short proto,
132                                                unsigned int sum)
133 {
134         return csum_fold(csum_tcpudp_nofold(saddr,daddr,len,proto,sum));
135 }
```

  - Sinon on affiche un message noyau annonçant l'échec de la vérification et on positionne le champ ip_summed du descripteur de tampon passé en argument à la valeur CHECKSUM_NONE.

- Si la valeur du champ ip_summed du descripteur de tampon passé en argument est CHECKSUM_UNNECESSARY, on calcule la somme de contrôle que l'on place dans le champ csum du descripteur de tampon passé en argument et on renvoie 0.

25.2.3 Consultation de la table de hachage UDP

25.2.3.1 Appel de la consultation

La fonction en ligne udp_v4_lookup() est définie dans le fichier linux/net/ipv4/udp.c: Code Linux 2.6.10

```
266 __inline__ struct sock *udp_v4_lookup(u32 saddr, u16 sport, u32 daddr, u16 dport, int dif)
267 {
268         struct sock *sk;
269
270         read_lock(&udp_hash_lock);
271         sk = udp_v4_lookup_longway(saddr, sport, daddr, dport, dif);
272         if (sk)
273                 sock_hold(sk);
274         read_unlock(&udp_hash_lock);
275         return sk;
276 }
```

Autrement dit :

- On déclare un descripteur de couche transport.
- On verrouille en lecture la table de hachage UDP.
  Le verrou udp_hash_lock est défini plus haut dans le même fichier : Code Linux 2.6.10

  ```
  118 rwlock_t udp_hash_lock = RW_LOCK_UNLOCKED;
  ```

- On essaie d'instantier le descripteur de couche transport grâce à la fonction udp_v4_lookup_longway() étudiée ci-dessous.
- Si on y parvient, on incrémente son compteur de référence.
- On déverrouille la lecture dans la table de hachage UDP et on renvoie l'adresse du descripteur (NULL si on n'est pas parvenue à l'instantier).

25.2.3.2 Options Internet

Nous allons avoir besoin d'une structure de données supplémentaires. La structure inet_opt est définie dans le fichier linux/include/linux/ip.h: Code Linux 2.6.10

```
110 struct inet_opt {
111         /* Comparaisons de demultiplexage de socket sur les paquets entrants. */
112         __u32                   daddr;          /* Adresse IPv4 etrangere */
113         __u32                   rcv_saddr;      /* Adresse IPv4 liee localement */
114         __u16                   dport;          /* Port de destination */
115         __u16                   num;            /* Port local */
116         __u32                   saddr;          /* Source d'envoi */
117         int                     uc_ttl;         /* Unicast TTL */
118         int                     tos;            /* TOS */
119         unsigned                cmsg_flags;
120         struct ip_options       *opt;
121         __u16                   sport;          /* Port source */
122         unsigned char           hdrincl;        /* En-tetes incluses ? */
123         __u8                    mc_ttl;         /* Multicasting TTL */
124         __u8                    mc_loop;        /* Loopback */
125         __u8                    pmtudisc;
126         __u16                   id;             /* Compteur d'IDentification pour les
                                                       paquets DF */
127         unsigned                recverr : 1,
128                                 freebind : 1;
129         int                     mc_index;       /* Multicast device index */
130         __u32                   mc_addr;
131         struct ip_mc_socklist   *mc_list;       /* Tableau de groupe */
```

```
132         /*
133          * Les membres suivants sont utilises pour detenir les informations necessaires
134          * a la construction d'un en-tete ip pour chaque fragment ip lorsque la socket
               a un bouchon.
135          */
136         struct {
137                 unsigned int            flags;
138                 unsigned int            fragsize;
139                 struct ip_options       *opt;
140                 struct rtable           *rt;
141                 int                     length; /* Longueur totale de toutes les trames */
142                 u32                     addr;
143                 struct flowi            fl;
144         } cork;
145 };
```

25.2.3.3 Consultation proprement dite

Code Linux 2.6.10

La fonction udp_v4_lookup_longway() est définie dans le fichier linux/net/ipv4/udp.c:

```
219 /* UDP est presque toujours en dehors du wazoo, il est inutile d'essayer d'etre
220  * plus dur que ca. -DaveM
221  */
222 struct sock *udp_v4_lookup_longway(u32 saddr, u16 sport, u32 daddr, u16 dport, int dif)
223 {
224         struct sock *sk, *result = NULL;
225         struct hlist_node *node;
226         unsigned short hnum = ntohs(dport);
227         int badness = -1;
228
229         sk_for_each(sk, node, &udp_hash[hnum & (UDP_HTABLE_SIZE - 1)]) {
230                 struct inet_opt *inet = inet_sk(sk);
231
232                 if (inet->num == hnum && !ipv6_only_sock(sk)) {
233                         int score = (sk->sk_family == PF_INET ? 1 : 0);
234                         if (inet->rcv_saddr) {
235                                 if (inet->rcv_saddr != daddr)
236                                         continue;
237                                 score+=2;
238                         }
239                         if (inet->daddr) {
240                                 if (inet->daddr != saddr)
241                                         continue;
242                                 score+=2;
243                         }
244                         if (inet->dport) {
245                                 if (inet->dport != sport)
246                                         continue;
247                                 score+=2;
248                         }
249                         if (sk->sk_bound_dev_if) {
250                                 if (sk->sk_bound_dev_if != dif)
251                                         continue;
252                                 score+=2;
253                         }
254                         if(score == 9) {
255                                 result = sk;
256                                 break;
257                         } else if(score > badness) {
258                                 result = sk;
259                                 badness = score;
260                         }
```

```
261                 }
262         }
263         return result;
264 }
```

Autrement dit :

- On déclare deux descripteurs de couche transport : un servant d'index dans la table de hachage UDP et l'autre pour recueillir le résultat, ce dernier étant initialisé à `NULL`.
- On déclare une liste de hachage de nœuds.
- On déclare un numéro de hachage, que l'on initialise avec le numéro de port de destination passé en argument.
- On déclare un entier de test, que l'on initialise à -1.
- On parcourt la table de hachage UDP.

  La table `udp_hash[]` de hachage UDP est définie dans le fichier `linux/net/ipv4/-udp.c` : Code Linux 2.6.10

```
117 struct hlist_head udp_hash[UDP_HTABLE_SIZE];
```

  La taille `UDP_HTABLE_SIZE` de la table de hachage UDP est définie dans le fichier `linux/include/net/udp.h` : Code Linux 2.6.10

```
32 #define UDP_HTABLE_SIZE         128
```

  La macro `sk_for_each()` est définie dans le fichier `linux/include/net/sock.h` : Code Linux 2.6.10

```
371 #define sk_for_each(__sk, node, list) \
372         hlist_for_each_entry(__sk, node, list, sk_node)
```

- Pour chaque élément, on déclare une option Internet, que l'on instantie avec le champ correspondant du descripteur de couche transport en train d'être analysé.

  La fonction en ligne `inet_sk()` est définie dans le fichier `linux/include/linux/ip.h` : Code Linux 2.6.10

```
151 /* ATTENTION : ne pas changer la place des  membres de inet_sock ! */
152 struct inet_sock {
153         struct sock       sk;
154 #if defined(CONFIG_IPV6) || defined(CONFIG_IPV6_MODULE)
155         struct ipv6_pinfo *pinet6;
156 #endif
157         struct inet_opt   inet;
158 };
159
160 static inline struct inet_opt * inet_sk(const struct sock *__sk)
161 {
162         return &((struct inet_sock *)__sk)->inet;
163 }
```

  La macro `ipv6_only_sock()`, qui ne nous concerne pas vraiment ici, est définie dans le fichier `linux/include/linux/ipv6.h` : Code Linux 2.6.10

```
292 #if defined(CONFIG_IPV6) || defined(CONFIG_IPV6_MODULE)
293 #define __ipv6_only_sock(sk)    (inet6_sk(sk)->ipv6only)
294 #define ipv6_only_sock(sk)      ((sk)->sk_family == PF_INET6 && __ipv6_only_sock(sk))
295 #else
296 #define __ipv6_only_sock(sk)    0
297 #define ipv6_only_sock(sk)      0
298 #endif
```

- Si le champ numéro de l'option Internet est égal au numéro de hachage et s'il ne s'agit pas d'un descripteur de couche de transport spécifique à IPv6 :
  - On déclare un score, que l'on initialise à 1 ou à 0 suivant que la famille de protocoles indiquée dans le descripteur de couche de transport est IPv4 ou non.
  - Si l'adresse source reçue est définie dans l'option Internet : si elle est différente de l'adresse de destination passée en argument, on sort de la boucle ; sinon on augmente le score de 2.
  - Si l'adresse de destination est définie dans l'option Internet : si elle est différente de l'adresse source passée en argument, on sort de la boucle ; sinon on augmente le score de 2.
  - Si le numéro de port de destination est défini dans l'option Internet : s'il est différent du numéro de port source passé en argument, on sort de la boucle ; sinon on augmente le score de 2.
  - Si l'index d'interface réseau est défini dans le descripteur de transport en train d'être analysé : s'il est différent de l'index d'interface de destination passé en argument, on sort de la boucle ; sinon on augmente le score de 2.
  - Si le score est égal à 9, on a trouvé le descripteur de couche de transport qu'il fallait. On en renvoie l'adresse.
  - Sinon, si le score est supérieur à la valeur de test, le résultat prend (provisoirement) la valeur du descripteur de couche de transport en train d'être analysé et la valeur de test ce score. Ceci permettra de renvoyer le descripteur de couche de transport qui correspond au mieux aux critères.
- Si on n'a pas trouvé le descripteur de couche de transport qui répond à tous les critères, on renvoie celui qui en remplit le plus.

25.2.4 Traitement du datagramme UDP

Code Linux 2.6.10

La fonction `udp_queue_rcv_skb()` est définie dans le fichier `linux/net/ipv4/udp.c` :

```
978  /* renvoie :
979   *  -1 : erreur
980   *   0 : succes
981   *  >0 : on soumet a nouveau le protocole "udp encap"
982   *
983   * Noter qu'en cas de succes ou d'erreur, le skb est suppose avoir
984   * ete place dans la file d'attente ou libere.
985   */
986  static int udp_queue_rcv_skb(struct sock * sk, struct sk_buff *skb)
987  {
988          struct udp_opt *up = udp_sk(sk);
989
990          /*
991           *      Charger-le a la socket, ecarter-le si la file d'attente est pleine.
992           */
993          if (!xfrm4_policy_check(sk, XFRM_POLICY_IN, skb)) {
994                  kfree_skb(skb);
995                  return -1;
996          }
997
998          if (up->encap_type) {
999                  /*
1000                  * C'est une socket d'encapsulation, regardons donc si c'est
1001                  * un paquet encapsule.
```

```
1002                 * S'il s'agit d'un paquet a garder en vie, contentons-nous de le manger.
1003                 * S'il s'agit d'un paquet encapsule, passons-le en entree
1004                 * xfrm IPsec et renvoyons la reponse
1005                 * appropriee. Sinon, passons a travers et
1006                 * passons-le a la socket UDP.
1007                 */
1008                int ret;
1009
1010                ret = udp_encap_rcv(sk, skb);
1011                if (ret == 0) {
1012                        /* Mangeons le paquet ... */
1013                        kfree_skb(skb);
1014                        return 0;
1015                }
1016                if (ret < 0) {
1017                        /* traitons le paquet ESP */
1018                        ret = xfrm4_rcv_encap(skb, up->encap_type);
1019                        UDP_INC_STATS_BH(UDP_MIB_INDATAGRAMS);
1020                        return -ret;
1021                }
1022                /* PAS PASSE A TRAVERS -- c'est un paquet UDP */
1023        }
1024
1025        if (sk->sk_filter && skb->ip_summed != CHECKSUM_UNNECESSARY) {
1026                if (__udp_checksum_complete(skb)) {
1027                        UDP_INC_STATS_BH(UDP_MIB_INERRORS);
1028                        kfree_skb(skb);
1029                        return -1;
1030                }
1031                skb->ip_summed = CHECKSUM_UNNECESSARY;
1032        }
1033
1034        if (sock_queue_rcv_skb(sk,skb)<0) {
1035                UDP_INC_STATS_BH(UDP_MIB_INERRORS);
1036                kfree_skb(skb);
1037                return -1;
1038        }
1039        UDP_INC_STATS_BH(UDP_MIB_INDATAGRAMS);
1040        return 0;
1041 }
```

Autrement dit :

– On déclare une adresse d'option UDP, que l'on instantie avec l'adresse du champ correspondant du descripteur de couche transport passé en argument.

La structure udp_opt est définie dans le fichier `linux/include/linux/udp.h`: Code Linux 2.6.10

```
43 struct udp_opt {
44         int             pending;        /* Y a-t-il des trames en attente ?      */
45         unsigned int    corkflag;       /* Un bouchon est necessaire             */
46         __u16           encap_type;     /* Est-ce une socket d'encapsulation ?   */
47         /*
48          * Le membre suivant contient les infomations pour creer un en-tete UDP
49          * lorsque la socket a un bouchon.
50          */
51         __u16           len;            /* longueur totale des trames en attente */
52 };
```

La fonction en ligne udp_sk() est définie dans le fichier `linux/include/linux/udp.h`: Code Linux 2.6.10

```
54 /* ATTENTION ! ne pas changer la disposition des membres dans udp_sock */
55 struct udp_sock {
```

```
56         struct sock       sk;
57 #if defined(CONFIG_IPV6) || defined(CONFIG_IPV6_MODULE)
58         struct ipv6_pinfo *pinet6;
59 #endif
60         struct inet_opt   inet;
61         struct udp_opt    udp;
62 };
63
64 static inline struct udp_opt * udp_sk(const struct sock *__sk)
65 {
66         return &((struct udp_sock *)__sk)->udp;
67 }
```

- Si la file d'attente est pleine, on libère le descripteur de tampon de socket passé en argument et on renvoie -1.
- S'il s'agit d'une socket d'encapsulation, on la traite comme indiqué dans le commentaire, ce qui ne nous intéresse pas vraiment dans cet ouvrage.
- S'il s'agit d'un filtre et si le champ `ip_summed` du descripteur de tampon passé en argument ne spécifie pas qu'il n'est pas nécessaire de vérifier la somme de contrôle:

  - On calcule la somme de contrôle. Si elle ne concorde pas, on incrémente l'information statistique sur le nombre d'erreurs des datagrammes UDP entrants, on libère le descripteur de tampon passé en argument et on renvoie -1.

    La fonction en ligne `__udp_checksum_complete()` est définie dans le fichier `linux/net/ipv4/udp.c`:

Code Linux 2.6.10

```
757 static __inline__ int __udp_checksum_complete(struct sk_buff *skb)
758 {
759         return (unsigned short)csum_fold(skb_checksum(skb, 0, skb->len,
                        skb->csum));
760 }
```

  - Sinon on modifie le champ `ip_summed` du descripteur de tampon passé en argument en spécifiant maintenant qu'il n'est pas nécessaire de vérifier la somme de contrôle.

- Sinon on essaie de placer le datagramme dans la file d'attente des datagrammes reçus grâce à la fonction `sock_queue_rcv_skb()`.

  C'est le cas qui nous intéresse.
- Si on n'y parvient pas, on incrémente l'information statistique sur le nombre d'erreurs des datagrammes UDP entrants, on libère le descripteur de tampon passé en argument et on renvoie -1.
- Sinon on incrémente l'information statistique sur le nombre de datagrammes UDP reçus avec succès, on libère le descripteur de tampon passé en argument et on renvoie 0.

25.2.5 Mise dans la file d'attente de réception UDP

La fonction en ligne `sock_queue_rcv_skb()` est définie dans le fichier `linux/include/net/sock.h`:

Code Linux 2.6.10

```
1070 static inline int sock_queue_rcv_skb(struct sock *sk, struct sk_buff *skb)
1071 {
1072         int err = 0;
1073         int skb_len;
1074
1075         /* Convertit skb->rcvbuf en unsigned... C'est sans importance mais ca reduit
1076            le nombre d'avertissements lorsqu'on compile avec -W --ANK
```

```
1077          */
1078         if (atomic_read(&sk->sk_rmem_alloc) + skb->truesize >=
1079             (unsigned)sk->sk_rcvbuf) {
1080                 err = -ENOMEM;
1081                 goto out;
1082         }
1083
1084         /* Ceci devrait etre un blocage si sock_queue_rcv_skb est utilise
1085            avec un verrou de socket ! Nous supposons que les utilisateurs de cette
1086            fonction n'utilisent pas de verrou.
1087         */
1088         err = sk_filter(sk, skb, 1);
1089         if (err)
1090                 goto out;
1091
1092         skb->dev = NULL;
1093         skb_set_owner_r(skb, sk);
1094
1095         /* Mise en cache de la longueur du SKB avant que nous mettions dans la file
1096          * d'attente en reception. Une fois qu'il a ete ajoute, il ne nous appartient
1097          * plus et il peut etre libere par d'autres threads de controle
1098          * de depilement des paquets de la file d'attente.
1099          */
1100         skb_len = skb->len;
1101
1102         skb_queue_tail(&sk->sk_receive_queue, skb);
1103
1104         if (!sock_flag(sk, SOCK_DEAD))
1105                 sk->sk_data_ready(sk, skb_len);
1106 out:
1107         return err;
1108 }
```

Chapitre 26

Installation de la famille de protocoles IPv4

Nous avons vu jusqu'ici comment le datagramme UDP d'une trame entrant dans un ordinateur est déposé dans la file d'attente UDP. Avant de voir comment l'application le récupère, il faut voir comment est installée l'API des sockets. Ceci est l'objet des deux chapitres qui suivent.

Commençons par voir comment est mis en place la famille de protocoles TCP/IP. Les familles de protocoles sont mises en place lors du démarrage du système d'exploitation Linux. Nous allons voir comment elles sont mises en place de façon générale, puis ce qu'il en est de la suite TCP/IP pour IPv4.

26.1 Manipulation des familles de protocoles

Nous allons voir quelles sont les familles de protocoles implémentées sous Linux et les fonctions qui permettent de les manipuler. Nous étudierons l'implémentation de ces fonctions dans la section suivante.

26.1.1 Familles de protocoles

26.1.1.1 Numéro des famille de protocoles

Il existe un certain nombre de familles de protocoles prises en charge par Linux. Chacune d'elles est repérée par un numéro. Des constantes symboliques représentant les numéros des familles de protocoles implémentées sont définies dans le fichier en-tête `linux/include/linux/socket.h`:

Code Linux 2.6.10

```
154 /* Familles d'adresses supportees. */
155 #define AF_UNSPEC       0
156 #define AF_UNIX         1       /* Sockets du domaine Unix         */
157 #define AF_LOCAL        1       /* Nom POSIX pour ci-dessus        */
158 #define AF_INET         2       /* Protocole Internet IP           */
159 #define AF_AX25         3       /* Radio Amateur AX.25             */
160 #define AF_IPX          4       /* Novell IPX                      */
161 #define AF_APPLETALK    5       /* DDP AppleTalk DDP               */
162 #define AF_NETROM       6       /* Radio Amateur NET/ROM           */
163 #define AF_BRIDGE       7       /* Pont multiprotocole             */
164 #define AF_ATMPVC       8       /* PVC ATM                         */
165 #define AF_X25          9       /* Reserve au projet X.25          */
166 #define AF_INET6        10      /* IP version 6                    */
167 #define AF_ROSE         11      /* Radio Amateur X.25 PLP          */
168 #define AF_DECnet       12      /* Reserve au projet DECnet        */
169 #define AF_NETBEUI      13      /* Reserve au projet 802.2LLC      */
170 #define AF_SECURITY     14      /* Pseudo AF de retour de securite */
171 #define AF_KEY          15      /* API de gestion de cle PF_KEY    */
172 #define AF_NETLINK      16
173 #define AF_ROUTE        AF_NETLINK /* Alias pour emuler 4.4BSD     */
174 #define AF_PACKET       17      /* Famille de paquets              */
175 #define AF_ASH          18      /* Ash                             */
176 #define AF_ECONET       19      /* Acorn Econet                    */
177 #define AF_ATMSVC       20      /* SVC ATM                         */
178 #define AF_SNA          22      /* Projet Linux SNA (cingles !)    */
179 #define AF_IRDA         23      /* sockets IRDA                    */
180 #define AF_PPPOX        24      /* sockets PPPoX                   */
181 #define AF_WANPIPE      25      /* Sockets de l'API Wanpipe        */
182 #define AF_LLC          26      /* LLC Linux                       */
183 #define AF_BLUETOOTH    31      /* Sockets Bluetooth               */
184 #define AF_MAX          32      /* Pour l'instant...               */
185
186 /* Familles de protocoles, la meme chose que les familles d'adresses. */
187 #define PF_UNSPEC       AF_UNSPEC
188 #define PF_UNIX         AF_UNIX
189 #define PF_LOCAL        AF_LOCAL
190 #define PF_INET         AF_INET
191 #define PF_AX25         AF_AX25
192 #define PF_IPX          AF_IPX
193 #define PF_APPLETALK    AF_APPLETALK
194 #define PF_NETROM       AF_NETROM
195 #define PF_BRIDGE       AF_BRIDGE
196 #define PF_ATMPVC       AF_ATMPVC
197 #define PF_X25          AF_X25
198 #define PF_INET6        AF_INET6
199 #define PF_ROSE         AF_ROSE
```

```
200 #define PF_DECnet       AF_DECnet
201 #define PF_NETBEUI      AF_NETBEUI
202 #define PF_SECURITY     AF_SECURITY
203 #define PF_KEY          AF_KEY
204 #define PF_NETLINK      AF_NETLINK
205 #define PF_ROUTE        AF_ROUTE
206 #define PF_PACKET       AF_PACKET
207 #define PF_ASH          AF_ASH
208 #define PF_ECONET       AF_ECONET
209 #define PF_ATMSVC       AF_ATMSVC
210 #define PF_SNA          AF_SNA
211 #define PF_IRDA         AF_IRDA
212 #define PF_PPPOX        AF_PPPOX
213 #define PF_WANPIPE      AF_WANPIPE
214 #define PF_LLC          AF_LLC
215 #define PF_BLUETOOTH    AF_BLUETOOTH
216 #define PF_MAX          AF_MAX
```

dont seuls AF_INET (et donc PF_INET), et éventuellement AF_LOCAL (et donc PF_LOCAL, AF_UNIX et PF_UNIX) et AF_NETLINK (et donc PF_NETLINK), nous intéresseront dans ce livre. On remarquera qu'un certain nombre d'implémentations sont en fait en projet.

26.1.1.2 Nombre de familles de protocoles prises en charge

Le nombre de familles de protocoles prises en charge par Linux, à savoir 32 dans la version 2.6.10, est repéré par la constante symbolique NPROTO dans le fichier en-tête linux/include/linux/net.h:

Code Linux 2.6.10

```
29 #define NPROTO          32              /* devrait suffire pour l'instant...   */
```

26.1.1.3 Nom de la fonction de création de socket

L'appel système socket() permet la création d'une socket, comme nous l'avons vu au chapitre 6. Son paramètre famille de protocoles permet de renvoyer à une fonction spécifique de création. Le nom de cette fonction de création est fourni par une entité du type net_proto_family. Celui-ci est défini dans le fichier en-tête linux/include/linux/net.h:

Code Linux 2.6.10

```
162 struct net_proto_family {
163         int             family;
164         int             (*create)(struct socket *sock, int protocol);
165         /* Ce sont des compteurs pour le nombre de methodes differentes
166            de celles que nous avons mis en place */
167         short           authentication;
168         short           encryption;
169         short           encrypt_net;
170         struct module   *owner;
171 };
```

Par exemple, pour IPv4, cette entité s'appelle inet_family_ops. Elle est définie dans le fichier linux/net/ipv4/af_inet.c:

Code Linux 2.6.10

```
849 static struct net_proto_family inet_family_ops = {
850         .family = PF_INET,
851         .create = inet_create,
852         .owner  = THIS_MODULE,
853 };
```

la fonction inet_create(), sur laquelle nous reviendrons plus tard, étant définie dans le même fichier.

26.1.1.4 Tableau des familles de protocoles

Les noms des fonctions de créations des sockets, ou plus exactement des entités du type ci-dessus, sont placés dans un tableau `net_families[]` indexé par le numéro de la famille de protocoles.

Code Linux 2.6.10

Ce tableau est déclaré dans le fichier `linux/net/socket.c`:

```
139 /*
140  *      La liste des protocoles. Chaque protocole y est enregistre.
141  */
142
143 static struct net_proto_family *net_families[NPROTO];
```

Il est d'abord initialisé à vide lors du démarrage du système. On le remplit au fur et à mesure pour les familles de protocoles dont on a besoin. Cela s'appelle **installer une famille de protocoles**. On doit désinstaller (retirer) une famille lorsqu'on n'en a plus besoin (ne serait-ce que pour faire de la place en mémoire vide).

L'enregistrement, ou installation, d'une famille de protocoles s'effectue grâce à la fonction :

```
int sock_register(struct net_proto_family *ops);
```

Le retrait d'une famille de protocoles s'effectue grâce à la fonction :

```
int sock_unregister(int family);
```

L'installation de la famille de protocoles IPv4 par exemple, celle qui nous intéresse, est située ligne 1049 de la fonction `inet_init()` définie dans le fichier `linux/net/ipv4/af_inet.c`, et appelée lors du démarrage du système :

Code Linux 2.6.10

```
1017 static int __init inet_init(void)
1018 {
[...]
1045         /*
1046          *      Dire aux SOCKETS que nous sommes vivant...
1047          */
1048
1049         (void)sock_register(&inet_family_ops);
[...]
1122 }
```

26.1.2 Les types de communication

Nous venons de voir, dans la première section, comment est identifié le premier argument de l'appel système de création d'une socket. Étudions maintenant ce qu'il en est du second argument, le type de communication, tout au moins dans le cas qui nous intéresse, celui de IPv4, lorsque nous avons besoin de préciser.

26.1.2.1 Numéro des types de communication

Chaque type de communication possède un numéro. Ceux-ci sont repérés par des constantes symboliques, définies dans le fichier en-tête `linux/include/linux/net.h`:

Code Linux 2.6.10

```
65 #ifndef ARCH_HAS_SOCKET_TYPES
66 /** sock_type - Types de socket
67  *
68  * Lorsqu'on ajoute un nouveau type de socket, s'il vous plait
69  * grep ARCH_HAS_SOCKET_TYPE include/asm-* /socket.h, au moins MIPS
70  * surcharger cette enum pour des raisons de compatibilite binaire.
71  *
72  * @SOCK_STREAM - socket de flux (connexion)
```

```
73  * @SOCK_DGRAM - socket de datagramme (sans conn.)
74  * @SOCK_RAW - socket brute
75  * @SOCK_RDM - pour Reliably-Delivered Message
76  * @SOCK_SEQPACKET - socket de paquet sequentiel
77  * @SOCK_PACKET - facon specifique a linux d'obtenir des paquets du niveau peripherique.
78  *                Pour ecrire rarp et d'autres choses semblales au niveau utilisateur.
79  */
80 enum sock_type {
81         SOCK_STREAM     = 1,
82         SOCK_DGRAM      = 2,
83         SOCK_RAW        = 3,
84         SOCK_RDM        = 4,
85         SOCK_SEQPACKET  = 5,
86         SOCK_PACKET     = 10,
87 };
88
89 #define SOCK_MAX (SOCK_PACKET + 1)
90
91 #endif /* ARCH_HAS_SOCKET_TYPES */
```

26.1.2.2 Descripteur de type de communication IP

Chaque type de communication est décrit par une entité du type `struct inet_protosw` (avec `sw` pour *Socket Communication*). Ce type est défini dans le fichier en-tête `linux/include/net/-protocol.h`:

Code Linux 2.6.10

```
59 /* Ceci est utilise pour enregistrer les interfaces socket pour les protocoles IP.  */
60 struct inet_protosw {
61         struct list_head list;
62
63         /* Ces deux champs forment la cle de consultation.  */
64         unsigned short   type;     /* C'est le 2nd argument de socket(2).      */
65         int              protocol; /* C'est le numero de protocole L4.         */
66
67         struct proto     *prot;
68         struct proto_ops *ops;
69
70         int              capability; /* De quelle capacite (s'il y en a une)
71                                       * avons-nous besoin pour utiliser cette
72                                       * interface de socket ?
73                                       */
74         char             no_check;   /* somme de controle en rcv/xmit/aucune ? */
75         unsigned char    flags;      /* Voir INET_PROTOSW_* ci-dessous.        */
76 };
```

dont la signification des champs est la suivante:

- Ces entités sont placées dans une liste chaînée, d'où l'intérêt du premier champ.
- L'entité est associée à un protocole de la suite TCP/IP et à un type de communication. On l'obtient à partir de ces deux paramètres. On a cependant quelquefois besoin de revenir en arrière, c'est-à-dire de retrouver ceux-ci, d'où l'intérêt des champs `protocol` et `type`. Le protocole est l'une des constantes symboliques définies dans le fichier `linux/include/linux/in.h`:

  Code Linux 2.6.10

```
24 /* Protocoles IP bien definis par des standards.  */
25 enum {
26   IPPROTO_IP   = 0,    /* Protocole par defaut pour TCP                              */
27   IPPROTO_ICMP = 1,    /* Internet Control Message Protocol                          */
28   IPPROTO_IGMP = 2,    /* Internet Group Management Protocol                         */
29   IPPROTO_IPIP = 4,    /* Tunnels IPIP (les tunnels KA9Q, plus anciens, utilisent 94) */
```

```
30   IPPROTO_TCP  = 6,    /* Transmission Control Protocol                    */
31   IPPROTO_EGP  = 8,    /* Exterior Gateway Protocol                        */
32   IPPROTO_PUP  = 12,   /* Protocole PUP                                    */
33   IPPROTO_UDP  = 17,   /* User Datagram Protocol                           */
34   IPPROTO_IDP  = 22,   /* Protocole IDP XNS                                */
35   IPPROTO_RSVP = 46,   /* Protocole RSVP                                   */
36   IPPROTO_GRE  = 47,   /* Tunnels GRE Cisco (rfc 1701,1702)                */
37
38   IPPROTO_IPV6 = 41,   /* Tunnels IPv6-dans-IPv4                           */
39
40   IPPROTO_ESP  = 50,   /* Protocole Encapsulation Security Payload         */
41   IPPROTO_AH   = 51,   /* Protocole Authentication Header                  */
42   IPPROTO_PIM  = 103,  /* Protocol Independent Multicast                   */
43
44   IPPROTO_COMP = 108,  /* Compression Header protocol                      */
45   IPPROTO_SCTP = 132,  /* Stream Control Transport Protocol                */
46
47   IPPROTO_RAW  = 255,  /* Paquets IP bruts                                 */
48   IPPROTO_MAX
49 };
```

dont nous avons déjà rencontré IPPROTO_UDP et IPPROTO_TCP lors de la programmation des sockets.

- prot spécifie l'ensemble des opérations sur les descripteurs de couche de transport (paramètre principal de type sock) dont nous étudions le type ci-dessous.
- ops spécifie l'ensemble des opérations sur les descripteurs de socket (paramètre principal de type socket).
- capability prend la valeur - 1 ou celle de l'une des constantes symboliques définie dans le fichier linux/include/linux/capability.h:

Code Linux 2.6.10

```
141 /* Permet la liaison aux sockets TCP/UDP en-dessous de 1024 */
142 /* Permet la liaison aux VCI ATM en dessous de 32 */
143
144 #define CAP_NET_BIND_SERVICE 10
145
146 /* Permet la diffusion generale, d'ecouter en multidiffusion */
147
148 #define CAP_NET_BROADCAST    11
149
150 /* Permet la configuration de l'interface */
151 /* Permet l'administraion des pare-feu IP, le masquage et la comptabilite */
152 /* Permet de positionner l'option de debogage sur les sockets */
153 /* Permet la modification des tables de routage */
154 /* Permet le positionnement de proprietaire de processus quelconque / groupe de
155    processus aux sockets */
156 /* Permet la liaison a toute adresse pour le proxy transparent */
157 /* Permet le positionnement des TOS (type de service) */
158 /* Permet le positionnement du mode de promiscuite */
159 /* Permet de mettre a zero les statistiques du peripherique */
160 /* Permet la multidiffusion */
161 /* Permet la lecture/ecriture sur les registres specifiques au peripherique */
162 /* Permet l'activation des sockets de controle ATM */
163
164 #define CAP_NET_ADMIN        12
165
166 /* Permet l'utilisation des sockets RAW */
167 /* Permet l'utilisation des sockets PACKET */
168
169 #define CAP_NET_RAW          13
```

- no_check spécifie éventuellement la somme de contrôle obtenue lors de la réception.

– `flags` indique s'il s'agit d'une entité permanente ou non. Les valeurs possibles sont définies juste après dans le même fichier :

Code Linux 2.6.10

```
77 #define INET_PROTOSW_REUSE     0x01  /* Les ports sont-ils automatiquement
                                        reutilisables ? */
78 #define INET_PROTOSW_PERMANENT 0x02  /* On ne peut pas retirer les protocoles
                                        permanents.     */
```

On trouve, par exemple pour UDP dans le cas de IPv4, dans le fichier `linux/net/ipv4/af_inet.c`:

Code Linux 2.6.10

```
874         {
875                 .type =       SOCK_DGRAM,
876                 .protocol =   IPPROTO_UDP,
877                 .prot =       &udp_prot,
878                 .ops =        &inet_dgram_ops,
879                 .capability = -1,
880                 .no_check =   UDP_CSUM_DEFAULT,
881                 .flags =      INET_PROTOSW_PERMANENT,
882         },
```

26.1.2.3 Opérations sur les sockets pour la couche transport

Le type `struct proto` est défini dans le fichier en-tête `linux/include/net/sock.h`:

Code Linux 2.6.10

```
496 /* Blocs de protocole IP que nous attachons aux sockets.
497  * interface couche socket -> couche transport
498  * l'interface transport -> reseau est definie par struct inet_proto
499  */
500 struct proto {
501         void                    (*close)(struct sock *sk,
502                                         long timeout);
503         int                     (*connect)(struct sock *sk,
504                                         struct sockaddr *uaddr,
505                                         int addr_len);
506         int                     (*disconnect)(struct sock *sk, int flags);
507
508         struct sock *           (*accept) (struct sock *sk, int flags, int *err);
509
510         int                     (*ioctl)(struct sock *sk, int cmd,
511                                          unsigned long arg);
512         int                     (*init)(struct sock *sk);
513         int                     (*destroy)(struct sock *sk);
514         void                    (*shutdown)(struct sock *sk, int how);
515         int                     (*setsockopt)(struct sock *sk, int level,
516                                         int optname, char __user *optval,
517                                         int optlen);
518         int                     (*getsockopt)(struct sock *sk, int level,
519                                         int optname, char __user *optval,
520                                         int __user *option);
521         int                     (*sendmsg)(struct kiocb *iocb, struct sock *sk,
522                                            struct msghdr *msg, size_t len);
523         int                     (*recvmsg)(struct kiocb *iocb, struct sock *sk,
524                                            struct msghdr *msg,
525                                         size_t len, int noblock, int flags,
526                                         int *addr_len);
527         int                     (*sendpage)(struct sock *sk, struct page *page,
528                                         int offset, size_t size, int flags);
529         int                     (*bind)(struct sock *sk,
530                                         struct sockaddr *uaddr, int addr_len);
531
532         int                     (*backlog_rcv) (struct sock *sk,
```

```
533                                             struct sk_buff *skb);
534
535         /* Methodes pour garder une trace de sk, le consulter et pour selectionner un port. */
536         void                    (*hash)(struct sock *sk);
537         void                    (*unhash)(struct sock *sk);
538         int                     (*get_port)(struct sock *sk, unsigned short snum);
539
540         /* Pression sur la memoire */
541         void                    (*enter_memory_pressure)(void);
542         atomic_t                *memory_allocated;      /* Memoire allouee actuellement. */
543         atomic_t                *sockets_allocated;     /* Nombre actuel de sockets. */
544         /*
545          * Drapeaux de pression : essaie de stagner.
546          * Note technique : c'est utilise dans de nombreux contextes non atomiquement.
547          * Tout le sk_stream_mem_schedule() est de cette nature : la comptabilite
548          * est stricte, les actions sont consultatives et ont la meme latence.
549          */
550         int                     *memory_pressure;
551         int                     *sysctl_mem;
552         int                     *sysctl_wmem;
553         int                     *sysctl_rmem;
554         int                     max_header;
555
556         kmem_cache_t            *slab;
557         int                     slab_obj_size;
558
559         struct module           *owner;
560
561         char                    name[32];
562
563         struct {
564                 int inuse;
565                 u8  __pad[SMP_CACHE_BYTES - sizeof(int)];
566         } stats[NR_CPUS];
567 };
```

On retrouve les opérations sur les sockets vues au niveau de l'API mais avec un paramètre du type descripteur de couche transport (et non un numéro de socket), ainsi que des opérations relatives au hachage.

26.1.2.4 Tableau des types de communication pour IPv4

Les types de communication pour IPv4 sont définis dans le tableau `inetsw_array[]`. Celui-ci est déclaré et initialisé dans le fichier `linux/net/ipv4/af_inet.c`:

Code Linux 2.6.10

```
859 /* Nous inserons au demarrage tous les elements de inetsw_array[] dans
860  * la liste chainee inetsw.
861  */
862 static struct inet_protosw inetsw_array[] =
863 {
864         {
865                 .type =       SOCK_STREAM,
866                 .protocol =   IPPROTO_TCP,
867                 .prot =       &tcp_prot,
868                 .ops =        &inet_stream_ops,
869                 .capability = -1,
870                 .no_check =   0,
871                 .flags =      INET_PROTOSW_PERMANENT,
872         },
873
874         {
875                 .type =       SOCK_DGRAM,
```

```
876                 .protocol =   IPPROTO_UDP,
877                 .prot =       &udp_prot,
878                 .ops =        &inet_dgram_ops,
879                 .capability = -1,
880                 .no_check =   UDP_CSUM_DEFAULT,
881                 .flags =      INET_PROTOSW_PERMANENT,
882         },
883
884
885         {
886                 .type =       SOCK_RAW,
887                 .protocol =   IPPROTO_IP,        /* wild card */
888                 .prot =       &raw_prot,
889                 .ops =        &inet_sockraw_ops,
890                 .capability = CAP_NET_RAW,
891                 .no_check =   UDP_CSUM_DEFAULT,
892                 .flags =      INET_PROTOSW_REUSE,
893         }
894 };
```

On enregistre un type de communication grâce à la fonction :

```
void inet_register_protosw(struct inet_protosw *p)
```

26.1.3 Les protocoles de la suite TCP/IPv4

Les protocoles de cette suite sont définis dans le fichier `linux/net/ipv4/af_inet.c`: Code Linux 2.6.10

```
969 #ifdef CONFIG_IP_MULTICAST
970 static struct net_protocol igmp_protocol = {
971         .handler =      igmp_rcv,
972 };
973 #endif
974
975 static struct net_protocol tcp_protocol = {
976         .handler =      tcp_v4_rcv,
977         .err_handler =  tcp_v4_err,
978         .no_policy =    1,
979 };
980
981 static struct net_protocol udp_protocol = {
982         .handler =      udp_rcv,
983         .err_handler =  udp_err,
984         .no_policy =    1,
985 };
986
987 static struct net_protocol icmp_protocol = {
988         .handler =      icmp_rcv,
989 };
```

26.2 Implémentation

26.2.1 Tableau des familles de protocoles

26.2.1.1 Première initialisation

Le tableau `net_families[]` est initialisé à nul, lors du démarrage du système, lignes 2036–2041 de la fonction `sock_init()` définie à la fin du fichier `linux/net/socket.c`: Code Linux 2.6.10

```
2032 void __init sock_init(void)
```

```
2033 {
2034         int i;
2035
2036         /*
2037          *      Initialise toutes les familles d'addresses (de protocoles).
2038          */
2039
2040         for (i = 0; i < NPROTO; i++)
2041                 net_families[i] = NULL;
2042
2043         /*
2044          *      Initialise le cache SLAB des sock.
2045          */
2046
2047         sk_init();
2048
2049 #ifdef SLAB_SKB
2050         /*
2051          *      Initialise le cache SLAB des skbuff
2052          */
2053         skb_init();
2054 #endif
2055
2056         /*
2057          *      Initialise le module des protocoles.
2058          */
2059
2060         init_inodecache();
2061         register_filesystem(&sock_fs_type);
2062         sock_mnt = kern_mount(&sock_fs_type);
2063         /* L'initialisation reelle des protocoles est effectuee lorsque
2064          *  do_initcalls tourne.
2065          */
2066
2067 #ifdef CONFIG_NETFILTER
2068         netfilter_init();
2069 #endif
2070 }
```

26.2.1.2 Installation d'une famille de protocoles

Code Linux 2.6.10

La fonction `sock_register()` est définie dans le fichier `linux/net/socket.c`:

```
1984 /*
1985  *      Cette fonction est appelee par une routine de protocole qui veut
1986  *      faire de la publicite pour sa famille d'adresses, et qui l'a liee dans le
1987  *      module SOCKET.
1988  */
1989
1990 int sock_register(struct net_proto_family *ops)
1991 {
1992         int err;
1993
1994         if (ops->family >= NPROTO) {
1995                 printk(KERN_CRIT "protocol %d >= NPROTO(%d)\n", ops->family, NPROTO);
1996                 return -ENOBUFS;
1997         }
1998         net_family_write_lock();
1999         err = -EEXIST;
2000         if (net_families[ops->family] == NULL) {
2001                 net_families[ops->family]=ops;
2002                 err = 0;
```

```
2003         }
2004         net_family_write_unlock();
2005         printk(KERN_INFO "NET: Registered protocol family %d\n",
2006                ops->family);
2007         return err;
2008 }
```

Autrement dit :

- On vérifie que le numéro de la famille de protocoles à installer est inférieur au nombre déclaré de familles de protocoles. Si ce n'est pas le cas, on affiche un message noyau d'erreur et on renvoie l'opposé du code d'erreur `ENOBUFS`.
- On verrouille toute autre tentative d'installation (n'oublions pas que les processus se déroulent en parallèle), grâce à la fonction auxiliaire `net_family_write_lock()` étudiée ci-dessous.
- Si rien n'était installé pour ce numéro de famille de protocoles, on l'installe, on déverrouille les autres tentatives d'installation, on affiche un message du noyau et on renvoie 0. Sinon on déverrouille pour permette d'autres tentatives et on renvoie l'opposé du code d'erreur `EEXIST`.

26.2.1.3 Verrouillage et déverrouillage de l'installation

La fonction `net_family_write_lock()` est définie dans le même fichier : Code Linux 2.6.10

```
145 #if defined(CONFIG_SMP) || defined(CONFIG_PREEMPT)
146 static atomic_t net_family_lockct = ATOMIC_INIT(0);
147 static spinlock_t net_family_lock = SPIN_LOCK_UNLOCKED;
148
149 /* La strategie est : les modifications du vecteur net_family sont courtes, ne pas
150    assoupir, et tres rares, mais l'acces en lecture devrait etre libre de tout verrou
151    exclusif.
152  */
153
154 static void net_family_write_lock(void)
155 {
156         spin_lock(&net_family_lock);
157         while (atomic_read(&net_family_lockct) != 0) {
158                 spin_unlock(&net_family_lock);
159
160                 yield();
161
162                 spin_lock(&net_family_lock);
163         }
164 }
```

ainsi que la fonction en ligne `net_family_write_unlock()` : Code Linux 2.6.10

```
166 static __inline__ void net_family_write_unlock(void)
167 {
168         spin_unlock(&net_family_lock);
169 }
```

26.2.1.4 Retrait d'une famille de protocoles

La fonction `sock_unregister()` est définie dans le fichier `linux/net/socket.c`: Code Linux 2.6.10

```
2010 /*
2011  *      Cette fonction est appelee par une routine de protocole qui veut
2012  *      supprimer sa famille d'adresses, et qui a retire son lien au
```

```
2013  *      module SOCKET.
2014  */
2015
2016 int sock_unregister(int family)
2017 {
2018         if (family < 0 || family >= NPROTO)
2019                 return -1;
2020
2021         net_family_write_lock();
2022         net_families[family]=NULL;
2023         net_family_write_unlock();
2024         printk(KERN_INFO "NET: Unregistered protocol family %d\n",
2025                family);
2026         return 0;
2027 }
```

Autrement dit :

- On vérifie que le numéro de famille de protocoles est bien situé dans l'intervalle adéquat. Si ce n'est pas le cas, on renvoie - 1.
- On verrouille pour interdire toute autre tentative de retrait.
- On met à zéro l'adresse de la fonction de création de socket.
- On déverrouille pour permettre les autres tentatives de retrait, on affiche un message noyau et on renvoie 0.

26.2.2 Enregistrement d'un type de communication IPv4

Code Linux 2.6.10

La fonction `inet_register_protosw()` est définie dans le fichier `linux/net/ipv4/af_inet.c` :

```
898 void inet_register_protosw(struct inet_protosw *p)
899 {
900         struct list_head *lh;
901         struct inet_protosw *answer;
902         int protocol = p->protocol;
903         struct list_head *last_perm;
904
905         spin_lock_bh(&inetsw_lock);
906
907         if (p->type >= SOCK_MAX)
908                 goto out_illegal;
909
910         /* Si nous essayons de surcharger un protocole permanent, sous caution. */
911         answer = NULL;
912         last_perm = &inetsw[p->type];
913         list_for_each(lh, &inetsw[p->type]) {
914                 answer = list_entry(lh, struct inet_protosw, list);
915
916                 /* Verifier uniquement la concordance non sauvage. */
917                 if (INET_PROTOSW_PERMANENT & answer->flags) {
918                         if (protocol == answer->protocol)
919                                 break;
920                         last_perm = lh;
921                 }
922
923                 answer = NULL;
924         }
925         if (answer)
926                 goto out_permanent;
927
```

```
928         /* Ajoute la nouvelle entree apres la derniere entree permanente si elle existe,
929          * de facon a ce que cette nouvelle entree ne surcharge pas une entree permanente
930          * lorsqu'elle concorde avec un protocole de carte sauvage. Mais il est permis de
931          * surcharger une entree non permanente existante. Ceci signifie que lorsque nous
932          * supprimons cette entree, le systeme renvoie automatiquement a l'ancien
                comportement.
933          */
934         list_add_rcu(&p->list, last_perm);
935 out:
936         spin_unlock_bh(&inetsw_lock);
937
938         synchronize_net();
939
940         return;
941
942 out_permanent:
943         printk(KERN_ERR "Attempt to override permanent protocol %d.\n",
944                protocol);
945         goto out;
946
947 out_illegal:
948         printk(KERN_ERR
949                "Ignoring attempt to register invalid socket type %d.\n",
950                p->type);
951         goto out;
952 }
```

dont le code se comprend assez aisément.

Le verrou `inetsw_lock` est défini dans le même fichier : Code Linux 2.6.10

```
128 static spinlock_t inetsw_lock = SPIN_LOCK_UNLOCKED;
```

La fonction `synchronize_net()` est définie dans le fichier `linux/net/core/dev.c` : Code Linux 2.6.10

```
2987 /* Synchronisation avec le processus de reception des paquets. */
2988 void synchronize_net(void)
2989 {
2990         might_sleep();
2991         synchronize_kernel();
2992 }
```

26.2.3 Installation de la suite de protocoles TCP/IPv4

La famille de protocole IPv4 est installée lors du démarrage de Linux, grâce à la fin du fichier `linux/net/ipv4/af_inet.c` : Code Linux 2.6.10

```
1124 module_init(inet_init);
```

La fonction `inet_init()` est définie dans ce même fichier : Code Linux 2.6.10

```
1017 static int __init inet_init(void)
1018 {
1019         struct sk_buff *dummy_skb;
1020         struct inet_protosw *q;
1021         struct list_head *r;
1022         int rc = -EINVAL;
1023
1024         if (sizeof(struct inet_skb_parm) > sizeof(dummy_skb->cb)) {
1025                 printk(KERN_CRIT "%s: panic\n", __FUNCTION__);
1026                 goto out;
1027         }
1028
1029         rc = sk_alloc_slab(&tcp_prot, "tcp_sock");
```

```
1030         if (rc) {
1031                 sk_alloc_slab_error(&tcp_prot);
1032                 goto out;
1033         }
1034         rc = sk_alloc_slab(&udp_prot, "udp_sock");
1035         if (rc) {
1036                 sk_alloc_slab_error(&udp_prot);
1037                 goto out_tcp_free_slab;
1038         }
1039         rc = sk_alloc_slab(&raw_prot, "raw_sock");
1040         if (rc) {
1041                 sk_alloc_slab_error(&raw_prot);
1042                 goto out_udp_free_slab;
1043         }
1044
1045         /*
1046          *      Dire aux SOCKETS que nous sommes en vie...
1047          */
1048
1049         (void)sock_register(&inet_family_ops);
1050
1051         /*
1052          *      Ajouter tous les protocoles de base.
1053          */
1054
1055         if (inet_add_protocol(&icmp_protocol, IPPROTO_ICMP) < 0)
1056                 printk(KERN_CRIT "inet_init: Cannot add ICMP protocol\n");
1057         if (inet_add_protocol(&udp_protocol, IPPROTO_UDP) < 0)
1058                 printk(KERN_CRIT "inet_init: Cannot add UDP protocol\n");
1059         if (inet_add_protocol(&tcp_protocol, IPPROTO_TCP) < 0)
1060                 printk(KERN_CRIT "inet_init: Cannot add TCP protocol\n");
1061 #ifdef CONFIG_IP_MULTICAST
1062         if (inet_add_protocol(&igmp_protocol, IPPROTO_IGMP) < 0)
1063                 printk(KERN_CRIT "inet_init: Cannot add IGMP protocol\n");
1064 #endif
1065
1066         /* Enregistrer l'information du cote des sockets pour inet_create. */
1067         for (r = &inetsw[0]; r < &inetsw[SOCK_MAX]; ++r)
1068                 INIT_LIST_HEAD(r);
1069
1070         for (q = inetsw_array; q < &inetsw_array[INETSW_ARRAY_LEN]; ++q)
1071                 inet_register_protosw(q);
1072
1073         /*
1074          *      Activer le module ARP
1075          */
1076
1077         arp_init();
1078
1079         /*
1080          *      Activer le module IP
1081          */
1082
1083         ip_init();
1084
1085         tcp_v4_init(&inet_family_ops);
1086
1087         /* Positionner le cache slab TCP pour l'ouverture des requetes. */
1088         tcp_init();
1089
1090
1091         /*
```

```
1092          *      Activer la couche ICMP
1093          */
1094
1095         icmp_init(&inet_family_ops);
1096
1097         /*
1098          *      Initialiser le routeur multidiffusion
1099          */
1100 #if defined(CONFIG_IP_MROUTE)
1101         ip_mr_init();
1102 #endif
1103         /*
1104          *      Initialiser un mib ipv4 par cpu
1105          */
1106
1107         if(init_ipv4_mibs())
1108                 printk(KERN_CRIT "inet_init: Cannot init ipv4 mibs\n"); ;
1109
1110         ipv4_proc_init();
1111
1112         ipfrag_init();
1113
1114         rc = 0;
1115 out:
1116         return rc;
1117 out_tcp_free_slab:
1118         sk_free_slab(&tcp_prot);
1119 out_udp_free_slab:
1120         sk_free_slab(&udp_prot);
1121         goto out;
1122 }
```

Autrement dit :

- On déclare un descripteur de tampon de socket, un descripteur de type de communication, une liste et un code de retour.
- Si la taille d'une entité de `inet_skb_parm` est supérieure à celle du bloc de contrôle d'un descripteur de tampon de socket, c'est que la famille de protocoles est mal implémentée. On affiche donc un message d'erreur sur l'écran des erreurs et on renvoie l'opposé du code d'erreur `EINVAL`.
- On essaie d'instantier l'antémémoire de tampon pour TCP, UDP et le mode brut. Si on ne parvient pas à instantier l'une de ces antémémoires, on affiche un message noyau et on renvoie le code d'erreur fourni par la fonction `sk_alloc_slab()`.

  La fonction `sk_alloc_slab()` est définie dans le fichier `linux/net/core/sock.c`: Code Linux 2.6.10

```
1338 int sk_alloc_slab(struct proto *prot, char *name)
1339 {
1340         prot->slab = kmem_cache_create(name,
1341                                        prot->slab_obj_size, 0,
1342                                        SLAB_HWCACHE_ALIGN, NULL, NULL);
1343
1344         return prot->slab != NULL ? 0 : -ENOBUFS;
1345 }
```

  La fonction en ligne `sk_alloc_slab_error()` est définie dans le fichier `linux/include/net/sock.h`: Code Linux 2.6.10

```
572 static inline void sk_alloc_slab_error(struct proto *proto)
573 {
574         printk(KERN_CRIT "%s: Can't create sock SLAB cache!\n", proto->name);
575 }
```

– On installe la famille de protocole IPv4.
– On essaie d'installer les protocoles ICMP, UDP, TCP et, éventuellement, IGMP. Si on ne parvient pas à installer l'un de ceux-ci, on affiche un message noyau.
– On initialise les listes des types de communication qui serviront lors de la création des sockets. On n'a besoin que des adresses de ses éléments pour cette initialisation.

Code Linux 2.6.10

Le tableau `inetsw[]` est déclaré dans le fichier `linux/net/ipv4/af_inet.c`:

```
124 /* La table inetsw contient toute chose dont inet_create a besoin pour
125  * construire une nouvelle socket.
126  */
127 static struct list_head inetsw[SOCK_MAX];
```

– On enregistre les protocoles de communication.

Code Linux 2.6.10

La constante `INETSW_ARRAY_LEN` est définie dans le fichier `linux/net/ipv4/af_inet.c`:

```
896 #define INETSW_ARRAY_LEN (sizeof(inetsw_array) / sizeof(struct inet_protosw))
```

– On active le protocole ARP.

Code Linux 2.6.10

La fonction `arp_init()` est définie dans le fichier `linux/net/ipv4/arp.c`:

```
1235 void __init arp_init(void)
1236 {
1237         neigh_table_init(&arp_tbl);
1238
1239         dev_add_pack(&arp_packet_type);
1240         arp_proc_init();
1241 #ifdef CONFIG_SYSCTL
1242         neigh_sysctl_register(NULL, &arp_tbl.parms, NET_IPV4,
1243                               NET_IPV4_NEIGH, "ipv4", NULL);
1244 #endif
1245         register_netdevice_notifier(&arp_netdev_notifier);
1246 }
```

– On active le protocole IP.

Code Linux 2.6.10

La fonction `ip_init()` est définie dans le fichier `linux/net/ipv4/ip_output.c`:

```
1336 void __init ip_init(void)
1337 {
1338         dev_add_pack(&ip_packet_type);
1339
1340         ip_rt_init();
1341         inet_initpeers();
1342
1343 #if defined(CONFIG_IP_MULTICAST) && defined(CONFIG_PROC_FS)
1344         igmp_mc_proc_init();
1345 #endif
1346 }
```

– On active le protocole TCP grâce aux fonctions `tcp_v4_init()` et `tcp_init()`, qui ne nous intéresserons pas dans ce volume.
– On active le protocole ICMP grâce à la fonction `icmp_init()`, qui ne nous intéressera pas dans ce volume.
– On initialise éventuellement le routeur de multidiffusion grâce à la fonction `ip_mr_init()`, ce qui ne nous intéressera pas ici.
– On initialise le service `MIBS`.

Code Linux 2.6.10

La fonction `init_ipv4_mibs()` est définie dans le fichier `linux/net/ipv4/af_inet.c`:

```
991  static int __init init_ipv4_mibs(void)
```

```
992  {
993          net_statistics[0] = alloc_percpu(struct linux_mib);
994          net_statistics[1] = alloc_percpu(struct linux_mib);
995          ip_statistics[0] = alloc_percpu(struct ipstats_mib);
996          ip_statistics[1] = alloc_percpu(struct ipstats_mib);
997          icmp_statistics[0] = alloc_percpu(struct icmp_mib);
998          icmp_statistics[1] = alloc_percpu(struct icmp_mib);
999          tcp_statistics[0] = alloc_percpu(struct tcp_mib);
1000         tcp_statistics[1] = alloc_percpu(struct tcp_mib);
1001         udp_statistics[0] = alloc_percpu(struct udp_mib);
1002         udp_statistics[1] = alloc_percpu(struct udp_mib);
1003         if (!
1004             (net_statistics[0] && net_statistics[1] && ip_statistics[0]
1005              && ip_statistics[1] && tcp_statistics[0] && tcp_statistics[1]
1006              && udp_statistics[0] && udp_statistics[1]))
1007                 return -ENOMEM;
1008
1009         (void) tcp_mib_init();
1010
1011         return 0;
1012 }
```

– On crée les entrées du répertoire /proc liées au réseau.
La fonction `ipv4_proc_init()` est définie dans le fichier `linux/net/ipv4/af_inet.c`: Code Linux 2.6.10

```
1139 int __init ipv4_proc_init(void)
1140 {
1141         int rc = 0;
1142
1143         if (raw_proc_init())
1144                 goto out_raw;
1145         if (tcp4_proc_init())
1146                 goto out_tcp;
1147         if (udp4_proc_init())
1148                 goto out_udp;
1149         if (fib_proc_init())
1150                 goto out_fib;
1151         if (ip_misc_proc_init())
1152                 goto out_misc;
1153 out:
1154         return rc;
1155 out_misc:
1156         fib_proc_exit();
1157 out_fib:
1158         udp4_proc_exit();
1159 out_udp:
1160         tcp4_proc_exit();
1161 out_tcp:
1162         raw_proc_exit();
1163 out_raw:
1164         rc = -ENOMEM;
1165         goto out;
1166 }
```

– On initialise la fragmentation des paquets IP, grâce à la fonction `ipfrag_init()`, et on renvoie 0.

Chapitre 27

Implémentation des fichiers de type socket

Nous avons vu au chapitre 6 que la programmation des réseaux se fait à travers un type de fichiers particulier, appelés sockets. Nous allons étudier l'implémentation de ce type de fichiers dans ce chapitre.

De même que les chapitres 13 et 24 portent plus sur la gestion de la mémoire que sur le sous-système réseau, ce chapitre porte sur la gestion des fichiers.

27.1 Implémentation générale des fichiers

Comme nous venons de le rappeler, sous UNIX tout est fichier, ce qui est donc également le cas des sockets, appartenant au type de fichiers qui nous intéresse. Commençons par donner une vue générale de l'implémentation des fichiers sous Linux.

27.1.1 Système de fichiers virtuel

Une des clés du succès de Linux est sa capacité à coexister aisément avec des systèmes de fichiers autres qui ceux qui sont natifs pour lui. Il est possible de monter, en toute transparence, des disques ou des partitions qui hébergent des formats de fichiers utilisés par Windows, d'autres systèmes UNIX ou des systèmes à faible part de marché comme Amiga.

27.1.1.1 Modèle de systèmes de fichiers commun

Linux gère le support de ces types multiples types de systèmes de fichiers au moyen d'un concept appelé **système de fichiers virtuel** (ou **VFS** pour l'anglais *Virtual File Sytem*), introduit par Kleiman en 1986 ([KLE–86]), avec une implémentation qui lui est propre.

Le concept majeur du `VFS` consiste à présenter un *modèle de fichier commun* capable de représenter tous les systèmes de fichiers pris en charge. L'idée du système de fichiers virtuel est que les entités internes représentant les fichiers et les systèmes de fichiers, situées dans la mémoire du noyau, renferment une vaste gamme d'informations de telle façon que toute opération fournie par un système de fichiers réel (compatible avec Linux) soit prise en charge par un champ du système virtuel. Pour toute fonction de lecture, d'écriture ou autre qui est appelée, le noyau substitue la fonction réelle adéquate.

Chaque implémentation d'un système de fichiers spécifique devra traduire son organisation physique dans le modèle de fichier commun du `VFS`.

27.1.1.2 Implémentation orientée objet

On peut considérer que le modèle de fichier commun est orienté objet, où un *objet* est une structure logicielle qui définit à la fois une structure de données et les méthodes qui opèrent à son niveau. Pour des raisons d'efficacité, Linux n'est pas programmé à l'aide d'un langage orienté objet tel que C++. Les objets sont implémentés comme des structures de données dont certains champs pointent sur les fonctions correspondant aux méthodes de l'objet.

27.1.1.3 Les composants du modèle de fichier commun

Le modèle de fichier commun se compose de quatre types d'objets, ceux-ci oncernant les fichiers (deux types), les répertoires et le disque :

Fichier. Les systèmes UNIX distinguent traditionnellement deux types de structures de données pour un fichier :

Nœud d'information et descripteur. Un nœud d'information (*inode* en anglais) sur le disque enregistre les informations générales sur un fichier donné (telles que le propriétaire du fichier et les droits d'accès). À ce nœud d'information, situé sur le disque pour les systèmes de fichiers natifs, correspond un **descripteur de nœud d'information**, situé en mémoire vive dans l'espace noyau, pour chaque fichier utilisé dans le système.

Descripteur de fichier. Les informations sur l'interaction entre un fichier ouvert et un processus sont stockées dans un descripteur de fichier[1]. Il s'agit des attributs du fichier, tel que le mode dans lequel le fichier peut être utilisé (lecture, écriture, lecture-écriture), ou la position en cours de la prochaine opération d'entrées-sorties. Ces informations existent seulement dans la mémoire du noyau et au cours de la période durant laquelle un processus accède à un fichier.

Entrée de répertoire. Un tel objet (en anglais *dentry* pour *Directory ENTRY*) stocke des informations sur la correspondance entre une entrée de répertoire et le fichier correspondant. Chaque système de fichiers sur disque enregistre ces informations sur un disque selon sa manière propre.

Super-bloc. Un tel objet enregistre des informations concernant un système de fichier monté, normalement une partition de disque dur ou un CD-ROM mais également une socket comme nous allons le voir.

Nous allons maintenant décrire les structures de données utilisées par Linux pour ces quatre types d'objets.

27.1.2 Super-bloc

27.1.2.1 Descripteur

Les descripteurs de super-bloc sont des entités du type `struct super_block`, défini dans le fichier `linux/include/linux/fs.h`:

Code Linux 2.6.10

```
754 struct super_block {
755         struct list_head        s_list;         /* Garder ceci en premier */
756         dev_t                   s_dev;          /* index de recherche ; _pas_ kdev_t */
757         unsigned long           s_blocksize;
758         unsigned long           s_old_blocksize;
759         unsigned char           s_blocksize_bits;
760         unsigned char           s_dirt;
761         unsigned long long      s_maxbytes;     /* Taille max du fichier */
762         struct file_system_type *s_type;
763         struct super_operations *s_op;
764         struct dquot_operations *dq_op;
765         struct quotactl_ops     *s_qcop;
766         struct export_operations *s_export_op;
767         unsigned long           s_flags;
768         unsigned long           s_magic;
769         struct dentry           *s_root;
770         struct rw_semaphore     s_umount;
771         struct semaphore        s_lock;
772         int                     s_count;
773         int                     s_syncing;
774         int                     s_need_sync_fs;
775         atomic_t                s_active;
776         void                    *s_security;
777         struct xattr_handler    **s_xattr;
778
779         struct list_head        s_dirty;        /* noeud d'information sale */
780         struct list_head        s_io;           /* en reserve d'ecriture */
781         struct hlist_head       s_anon;         /* dentries anonymes a exporter (nfs) */
782         struct list_head        s_files;
783
784         struct block_device     *s_bdev;
```

1. Nous appellerons **numéro de fichier** ce qui est quelquefois appelé descripteur de fichier.

```
785         struct list_head        s_instances;
786         struct quota_info       s_dquot;        /* Options relatives au quota du disque */
787
788         int                     s_frozen;
789         wait_queue_head_t       s_wait_unfrozen;
790
791         char s_id[32];                          /* Nom donnant de l'information */
792
793         void                    *s_fs_info;     /* Infos privees du systeme de fichiers */
794
795         /*
796          * Le champ suivant interesse *exclusivement* VFS. Aucun systeme de fichiers n'a
797          * la possibilite de le verrouiller. Vous etes avertis.
798          */
799         struct semaphore s_vfs_rename_sem;      /* Kludge */
800 };
```

Nous nous permettons de renvoyer à notre livre [CEG–03] pour une description détaillée des champs dans le cas, plus simple, du noyau 0.01.

27.1.2.2 Opérations sur les super-blocs

Code Linux 2.6.10

Les opérations permises sur un super-bloc sont précisées par le champ de la ligne 763, à savoir `s_op` de type `super_operations`. Ce dernier est défini dans le même fichier en-tête :

```
969 /*
970  * NOTE : write_inode, delete_inode, clear_inode, put_inode peuvent etre appeles
971  * sans que le verrou du noyau ne soit detenu dans certains systemes de fichiers.
972  */
973 struct super_operations {
974         struct inode *(*alloc_inode)(struct super_block *sb);
975         void (*destroy_inode)(struct inode *);
976
977         void (*read_inode) (struct inode *);
978
979         void (*dirty_inode) (struct inode *);
980         int (*write_inode) (struct inode *, int);
981         void (*put_inode) (struct inode *);
982         void (*drop_inode) (struct inode *);
983         void (*delete_inode) (struct inode *);
984         void (*put_super) (struct super_block *);
985         void (*write_super) (struct super_block *);
986         int (*sync_fs)(struct super_block *sb, int wait);
987         void (*write_super_lockfs) (struct super_block *);
988         void (*unlockfs) (struct super_block *);
989         int (*statfs) (struct super_block *, struct kstatfs *);
990         int (*remount_fs) (struct super_block *, int *, char *);
991         void (*clear_inode) (struct inode *);
992         void (*umount_begin) (struct super_block *);
993
994         int (*show_options)(struct seq_file *, struct vfsmount *);
995 };
```

27.1.2.3 Opération sur le statut

La fonction `statfs()`, l'une des trois seules qui nous intéresseront dans le cas des sockets comme nous le verrons, doit fournir le statut du système de fichiers. Celui-ci est décrit dans une entité du type `struct statfs`. Ce type dépend de l'architecture du microprocesseur. Il est défini, en particulier pour les microprocesseurs Intel, dans le fichier en-tête `linux/include/asm-generic/statfs.h`:

Code Linux 2.6.10

```
1  #ifndef _GENERIC_STATFS_H
2  #define _GENERIC_STATFS_H
3
4  #ifndef __KERNEL_STRICT_NAMES
5  # include <linux/types.h>
6  typedef __kernel_fsid_t fsid_t;
7  #endif
8
9  struct statfs {
10         __u32 f_type;
11         __u32 f_bsize;
12         __u32 f_blocks;
13         __u32 f_bfree;
14         __u32 f_bavail;
15         __u32 f_files;
16         __u32 f_ffree;
17         __kernel_fsid_t f_fsid;
18         __u32 f_namelen;
19         __u32 f_frsize;
20         __u32 f_spare[5];
21 };
```

27.1.3 Nœud d'information

27.1.3.1 Descripteur

Les descripteurs de nœud d'information ont une structure, appelée `inode`, définie dans le fichier `linux/include/linux/fs.h`:

Code Linux 2.6.10

```
429 struct inode {
430         struct hlist_node       i_hash;
431         struct list_head        i_list;
432         struct list_head        i_dentry;
433         unsigned long           i_ino;
434         atomic_t                i_count;
435         umode_t                 i_mode;
436         unsigned int            i_nlink;
437         uid_t                   i_uid;
438         gid_t                   i_gid;
439         dev_t                   i_rdev;
440         loff_t                  i_size;
441         struct timespec         i_atime;
442         struct timespec         i_mtime;
443         struct timespec         i_ctime;
444         unsigned int            i_blkbits;
445         unsigned long           i_blksize;
446         unsigned long           i_version;
447         unsigned long           i_blocks;
448         unsigned short          i_bytes;
449         unsigned char           i_sock;
450         spinlock_t              i_lock; /* i_blocks, i_bytes, peut-etre i_size */
451         struct semaphore        i_sem;
452         struct rw_semaphore     i_alloc_sem;
```

```
453         struct inode_operations *i_op;
454         struct file_operations  *i_fop; /* precedemment ->i_op->default_file_ops */
455         struct super_block      *i_sb;
456         struct file_lock        *i_flock;
457         struct address_space    *i_mapping;
458         struct address_space    i_data;
459 #ifdef CONFIG_QUOTA
460         struct dquot            *i_dquot[MAXQUOTAS];
461 #endif
462         /* Ces trois champs devraient probablement former une union */
463         struct list_head        i_devices;
464         struct pipe_inode_info  *i_pipe;
465         struct block_device     *i_bdev;
466         struct cdev             *i_cdev;
467         int                     i_cindex;
468
469         __u32                   i_generation;
470
471 #ifdef CONFIG_DNOTIFY
472         unsigned long           i_dnotify_mask; /* Evenements de notification repertoire */
473         struct dnotify_struct   *i_dnotify;     /* pour les notifications repertoire     */
474 #endif
475
476         unsigned long           i_state;
477         unsigned long           dirtied_when;   /* heure exacte (en jiffies) de
                                                       la premiere salissure */
478
479         unsigned int            i_flags;
480
481         atomic_t                i_writecount;
482         void                    *i_security;
483         union {
484                 void            *generic_ip;
485         } u;
486 #ifdef __NEED_I_SIZE_ORDERED
487         seqcount_t              i_size_seqcount;
488 #endif
489 };
```

Nous renvoyons à notre livre [CEG–03] pour la description de la structure associée aux descripteurs de nœud d'information dans le cas, plus simple mais suffisant pour avoir une idée générale, du noyau Linux 0.01.

Les deux champs qui nous intéresseront plus particulièrement sont le champ `i_fop` de la ligne 454 concernant les opérations possibles sur le type de fichier et le champ `u` de la ligne 485, spécifique aux sockets.

27.1.3.2 Opérations

Code Linux 2.6.10

Le type `struct file_operations` du champ `i_fop` est défini dans le fichier en-tête `linux/include/linux/fs.h`:

```
903 /*
904  * NOTE :
905  * read, write, poll, fsync, readv, writev peuvent etre appeles
906  *   sans que le verrou noyau ne soit detenu pour certains systemes de fichiers.
907  */
908 struct file_operations {
909         struct module *owner;
910         loff_t (*llseek) (struct file *, loff_t, int);
911         ssize_t (*read) (struct file *, char __user *, size_t, loff_t *);
912         ssize_t (*aio_read) (struct kiocb *, char __user *, size_t, loff_t);
```

```
913         ssize_t (*write) (struct file *, const char __user *, size_t, loff_t *);
914         ssize_t (*aio_write) (struct kiocb *, const char __user *, size_t, loff_t);
915         int (*readdir) (struct file *, void *, filldir_t);
916         unsigned int (*poll) (struct file *, struct poll_table_struct *);
917         int (*ioctl) (struct inode *, struct file *, unsigned int, unsigned long);
918         int (*mmap) (struct file *, struct vm_area_struct *);
919         int (*open) (struct inode *, struct file *);
920         int (*flush) (struct file *);
921         int (*release) (struct inode *, struct file *);
922         int (*fsync) (struct file *, struct dentry *, int datasync);
923         int (*aio_fsync) (struct kiocb *, int datasync);
924         int (*fasync) (int, struct file *, int);
925         int (*lock) (struct file *, int, struct file_lock *);
926         ssize_t (*readv) (struct file *, const struct iovec *, unsigned long, loff_t *);
927         ssize_t (*writev) (struct file *, const struct iovec *, unsigned long, loff_t *);
928         ssize_t (*sendfile) (struct file *, loff_t *, size_t, read_actor_t, void *);
929         ssize_t (*sendpage) (struct file *, struct page *, int, size_t, loff_t *, int);
930         unsigned long (*get_unmapped_area)(struct file *, unsigned long, unsigned long,
                                               unsigned long, unsigned long);
931         int (*check_flags)(int);
932         int (*dir_notify)(struct file *filp, unsigned long arg);
933         int (*flock) (struct file *, int, struct file_lock *);
934 };
```

27.1.4 Descripteur de fichier

Les descripteurs de fichier ont une structure, appelée `file`, définie dans le fichier `linux/include/linux/fs.h`:

Code Linux 2.6.10

```
577 struct file {
578         struct list_head        f_list;
579         struct dentry           *f_dentry;
580         struct vfsmount         *f_vfsmnt;
581         struct file_operations  *f_op;
582         atomic_t                f_count;
583         unsigned int            f_flags;
584         mode_t                  f_mode;
585         int                     f_error;
586         loff_t                  f_pos;
587         struct fown_struct      f_owner;
588         unsigned int            f_uid, f_gid;
589         struct file_ra_state    f_ra;
590
591         unsigned long           f_version;
592         void                    *f_security;
593
594         /* necessaire pour les peripheriques tty et peut-etre d'autres */
595         void                    *private_data;
596
597 #ifdef CONFIG_EPOLL
598         /* Utilise par fs/eventpoll.c pour lier toutes les ancres a ce fichier */
599         struct list_head        f_ep_links;
600         spinlock_t              f_ep_lock;
601 #endif /* #ifdef CONFIG_EPOLL */
602         struct address_space    *f_mapping;
603 };
```

Là encore nous renvoyons à notre livre [CEG–03] pour une étude de cette structure dans le cas plus simple du noyau Linux 0.01. Le seul champ qui nous intéressera, dans le cas des sockets, est celui de la ligne 581, à savoir `f_op` concernant sur les opérations permises et dont le type a été défini ci-dessus.

27.1.5 Répertoire

27.1.5.1 Descripteur

Code Linux 2.6.10

Les descripteurs d'entrées de répertoire sont des entités du type `struct dentry` défini dans le fichier en-tête `linux/include/linux/dcache.h`:

```
83  struct dentry {
84          atomic_t d_count;
85          unsigned int d_flags;           /* protege par d_lock */
86          spinlock_t d_lock;              /* un verrou par dentry */
87          struct inode *d_inode;          /* La a qui le nom appartient - NULL est
88                                           * negatif */
89          /*
90           * Les trois champs suivants sont affectes par __d_lookup. Les placer ici
91           * pour qu'ils soient tous dans une amplitude de 16 octets, avec un alignement
               de 16 octets.
92           */
93          struct dentry *d_parent;        /* repertoire parent */
94          struct qstr d_name;
95
96          struct list_head d_lru;         /* Liste LRU */
97          struct list_head d_child;       /* fils de la liste parent */
98          struct list_head d_subdirs;     /* nos enfants */
99          struct list_head d_alias;       /* liste d'alias des inode */
100         unsigned long d_time;           /* utilise par d_revalidate */
101         struct dentry_operations *d_op;
102         struct super_block *d_sb;       /* La racine de l'arbre du dentry */
103         void *d_fsdata;                 /* donnees specifiques au fs */
104         struct rcu_head d_rcu;
105         struct dcookie_struct *d_cookie; /* cafteur, s'il y en a */
106         struct hlist_node d_hash;       /* liste de hachage de consultation */
107         int d_mounted;
108         unsigned char d_iname[DNAME_INLINE_LEN_MIN];    /* petits noms */
109 };
```

27.1.5.2 Opérations

Code Linux 2.6.10

Les fonctions permises sur les entrées de répertoire sont définies par le champ `d_op` de la ligne 101, du type `struct dentry_operations`. Celui-ci est défini dans le même fichier en-tête:

```
111 struct dentry_operations {
112         int (*d_revalidate)(struct dentry *, struct nameidata *);
113         int (*d_hash) (struct dentry *, struct qstr *);
114         int (*d_compare) (struct dentry *, struct qstr *, struct qstr *);
115         int (*d_delete)(struct dentry *);
116         void (*d_release)(struct dentry *);
117         void (*d_iput)(struct dentry *, struct inode *);
118 };
```

27.1.6 Types de fichiers

Code Linux 2.6.10

Les types de fichiers acceptés par Linux sont définis dans le fichier `linux/include/linux/-fs.h`:

```
844 /*
845  * Types de fichiers
846  *
847  * NOTE : Ceci correspond aux bits 12..15 de stat.st_mode
848  * (par exemple "(i_mode >> 12) & 15").
849  */
```

```
850 #define DT_UNKNOWN      0
851 #define DT_FIFO         1
852 #define DT_CHR          2
853 #define DT_DIR          4
854 #define DT_BLK          6
855 #define DT_REG          8
856 #define DT_LNK          10
857 #define DT_SOCK         12
858 #define DT_WHT          14
```

à savoir les types inconnu, tube de communication, périphérique caractère, répertoire, périphérique bloc, régulier, lien, socket et blanc. Seul le type socket nous intéressera dans ce livre.

27.1.7 Déclaration d'un système de fichiers

Un système de fichier est caractérisé par une entité du type `struct file_system_type`, défini dans le fichier en-tête `linux/include/linux/fs.h`:

Code Linux 2.6.10

```
1137 struct file_system_type {
1138         const char *name;
1139         int fs_flags;
1140         struct super_block *(*get_sb) (struct file_system_type *, int,
1141                                        const char *, void *);
1142         void (*kill_sb) (struct super_block *);
1143         struct module *owner;
1144         struct file_system_type * next;
1145         struct list_head fs_supers;
1146 };
```

qui spécifie en particulier le nom du système de fichiers et les fonctions permettant d'obtenir le super-bloc d'une instance de celui-ci et de libérer celle-ci.

27.1.8 Enregistrement d'un système de fichiers

L'enregistrement d'un (type de) système(s) de fichiers est effectué grâce à la fonction `register_filesystem`, définie dans le fichier `linux/fs/filesystems.c`:

Code Linux 2.6.10

```
53 /**
54  *      register_filesystem - enregistre un nouveau systeme de fichiers
55  *      @fs : la structure systeme de fichiers
56  *
57  *      Ajoute le systeme de fichiers passe a la liste des systemes de fichiers dont le noyau
58  *      est averti pour le montage et autres appels systeme. Renvoie 0 en cas de succes
59  *      ou un code errno negatif en cas d'erreur.
60  *
61  *      La &struct file_system_type qui est passee est liee aux structures du
62  *      noyau et ne doit pas etre liberee avant que le systeme de fichiers n'ait ete
63  *      retire.
64  */
65
66 int register_filesystem(struct file_system_type * fs)
67 {
68         int res = 0;
69         struct file_system_type ** p;
70
71         if (!fs)
72                 return -EINVAL;
73         if (fs->next)
74                 return -EBUSY;
75         INIT_LIST_HEAD(&fs->fs_supers);
76         write_lock(&file_systems_lock);
```

```
77          p = find_filesystem(fs->name);
78          if (*p)
79                  res = -EBUSY;
80          else
81                  *p = fs;
82          write_unlock(&file_systems_lock);
83          return res;
84  }
```

27.2 Descripteur de socket

Linux place les informations générales sur une socket, celles qui sont indépendantes de tout protocole, dans un **descripteur de socket**, entité du type `struct socket`.

27.2.1 Définition du type

Code Linux 2.6.10

Le type `struct socket` est défini dans le fichier `linux/include/linux/net.h`:

```
93  /**
94    *  struct socket - socket BSD generale
95    *  @state - etat de la socket (%SS_CONNECTED, etc)
96    *  @flags - drapeaux de la socket (%SOCK_ASYNC_NOSPACE, etc)
97    *  @ops - operations sur la socket specifiques au protocole
98    *  @fasync_list - liste d'eveil asynchrone
99    *  @file - pointeur de retour au fichier pour le gc
100   *  @sk - representation de la socket agnostique au protocole de reseau interne
101   *  @wait - attend une file d'attente pour plusieurs usages
102   *  @type - type de socket (%SOCK_STREAM, etc)
103   *  @passcred - pieces d'identite (utilisees seulement pour les sockets Unix (ou PF_LOCAL))
104   */
105 struct socket {
106         socket_state            state;
107         unsigned long           flags;
108         struct proto_ops        *ops;
109         struct fasync_struct    *fasync_list;
110         struct file             *file;
111         struct sock             *sk;
112         wait_queue_head_t       wait;
113         short                   type;
114         unsigned char           passcred;
115 };
```

Décrivons rapidement chacun des champs avant de revenir plus en détail sur certains d'entre eux:

- l'état de la socket, sur lequel nous allons revenir, est précisé par le champ `state`;
- un vecteur de drapeaux `flags`, sur lequel nous allons également revenir;
- les opérations `ops` permises par la famille de protocoles utilisée, que nous détaillerons ci-dessous;
- la liste `fasync_list` des fichiers à réveiller, le type général `struct fasync_struct` étant défini dans le fichier `linux/include/linux/fs.h`;
- le descripteur de fichier `file` associé, ce qui sert pour le ramasse-miettes; il s'agit d'une référence inverse puisque c'est normalement le descripteur de fichier qui fait référence au descripteur de socket;
- un descripteur de couche transport `sk`;
- la file d'attente `wait` associée à la socket;

- le type de communication `type` tel qu'il est passé en argument lors de la création d'une socket ;
- les pièces d'identité `passcred` de la socket, utiles lors de l'utilisation de données ancillaires.

27.2.2 Les états d'une socket

Le type `socket_state`, défini dans le fichier en-tête `linux/include/linux/net.h`: Code Linux 2.6.10

```
49 typedef enum {
50         SS_FREE = 0,                    /* non allouee                    */
51         SS_UNCONNECTED,                 /* non connectee a une autre socket */
52         SS_CONNECTING,                  /* en train d'etre connectee      */
53         SS_CONNECTED,                   /* connectee a une autre socket   */
54         SS_DISCONNECTING                /* en train d'etre deconnectee    */
55 } socket_state;
```

précise les états possibles :

- `SS_FREE` : descripteur de socket non alloué ;
- `SS_UNCONNECTED` : socket non connectée à une autre socket ;
- `SS_CONNECTING` : socket en train de se connecter à une autre socket ;
- `SS_CONNECTED` : socket connectée à une autre socket ;
- `SS_DISCONNECTING` : socket en train de se déconnecter.

27.2.3 Les drapeaux

Le trois valeurs possibles des drapeaux sont définies dans le fichier en-tête `linux/include/linux/net.h`: Code Linux 2.6.10

```
61 #define SOCK_ASYNC_NOSPACE      0
62 #define SOCK_ASYNC_WAITDATA     1
63 #define SOCK_NOSPACE            2
```

27.2.4 Le type opérations sur une socket

Le type `struct proto_ops` est défini dans le fichier en-tête `linux/include/linux/net.h`: Code Linux 2.6.10

```
124 struct proto_ops {
125         int             family;
126         struct module   *owner;
127         int             (*release)   (struct socket *sock);
128         int             (*bind)      (struct socket *sock,
129                                       struct sockaddr *myaddr,
130                                       int sockaddr_len);
131         int             (*connect)   (struct socket *sock,
132                                       struct sockaddr *vaddr,
133                                       int sockaddr_len, int flags);
134         int             (*socketpair)(struct socket *sock1,
135                                       struct socket *sock2);
136         int             (*accept)    (struct socket *sock,
137                                       struct socket *newsock, int flags);
138         int             (*getname)   (struct socket *sock,
139                                       struct sockaddr *addr,
140                                       int *sockaddr_len, int peer);
141         unsigned int    (*poll)      (struct file *file, struct socket *sock,
142                                       struct poll_table_struct *wait);
143         int             (*ioctl)     (struct socket *sock, unsigned int cmd,
144                                       unsigned long arg);
```

```
145         int             (*listen)    (struct socket *sock, int len);
146         int             (*shutdown)  (struct socket *sock, int flags);
147         int             (*setsockopt)(struct socket *sock, int level,
148                                       int optname, char __user *optval, int optlen);
149         int             (*getsockopt)(struct socket *sock, int level,
150                                       int optname, char __user *optval, int __user *optlen);
151         int             (*sendmsg)   (struct kiocb *iocb, struct socket *sock,
152                                       struct msghdr *m, size_t total_len);
153         int             (*recvmsg)   (struct kiocb *iocb, struct socket *sock,
154                                       struct msghdr *m, size_t total_len,
155                                       int flags);
156         int             (*mmap)      (struct file *file, struct socket *sock,
157                                       struct vm_area_struct * vma);
158         ssize_t         (*sendpage)  (struct socket *sock, struct page *page,
159                                       int offset, size_t size, int flags);
160 };
```

On reconnaît les noms des appels système concernant les sockets.

27.3 Déclaration du système de fichiers des sockets

27.3.1 La structure

Code Linux 2.6.10

Le nom, `sockfs`, du système de fichiers des sockets et le type associé, `sock_fs_type`, sont définis dans le fichier `linux/net/socket.c`:

```
335 static struct file_system_type sock_fs_type = {
336         .name =         "sockfs",
337         .get_sb =       sockfs_get_sb,
338         .kill_sb =      kill_anon_super,
339 };
```

Les fonctions d'obtention et de libération d'un super-bloc d'un système de fichiers de sockets s'appellent donc `sockfs_get_sb()` et `kill_anon_super()`. Nous allons les décrire dans les sous-sections suivantes.

Code Linux 2.6.10

Le nombre magique `SOCKFS_MAGIC` associé à ce système de fichiers est défini dans le même fichier:

```
272 #define SOCKFS_MAGIC 0x534F434B
```

27.3.2 Enregistrement

Code Linux 2.6.10

L'enregistrement du sytème de fichiers des sockets est effectué ligne 2061 de la définition de la fonction `sock_init()`, déjà rencontrée au chapitre 26:

```
2032 void __init sock_init(void)
2033 {
[...]
2056         /*
2057          *      Initialise le modules des protocoles.
2058          */
2059
2060         init_inodecache();
2061         register_filesystem(&sock_fs_type);
2062         sock_mnt = kern_mount(&sock_fs_type);
2063         /* L'initialisation reelle des protocoles est effectuee lorsque
2064          *  do_initcalls est en action.
2065          */
[...]
2070 }
```

27.4 Système de fichiers de sockets

Une socket est un système de fichiers de sockets à elle toute seule. Un système de fichiers de sockets possède donc un super-bloc dont peu de champs sont significatifs et un seul répertoire (outre les deux fichiers point et point-point obligatoires) contenant un seul fichier.

La structure `sock_fs_type` définie ci-dessus précise le nom des deux fonctions d'obtention et de libération d'un super-bloc dans le cas des sockets.

27.4.1 Obtention du super-bloc

27.4.1.1 Fonction spécifique

Nous venons de voir que la fonction de lecture du super-bloc d'un système de fichiers de type socket s'appelle `sockfs_get_sb()`. Elle est définie dans le fichier `linux/net/socket.c`: Code Linux 2.6.10

```
327 static struct super_block *sockfs_get_sb(struct file_system_type *fs_type,
328         int flags, const char *dev_name, void *data)
329 {
330         return get_sb_pseudo(fs_type, "socket:", &sockfs_ops, SOCKFS_MAGIC);
331 }
```

et renvoie à la fonction `get_sb_pseudo()` qui concerne les pseudo-systèmes de fichiers, ceux qui ne sont pas en mémoire de masse.

On remarquera que les trois derniers paramètres ne sont pas pris en compte (ils ont un intérêt pour les systèmes de fichiers situés en mémoire de masse mais pas pour celui des sockets). Le premier paramètre est nécessairement `sock_fs_type`, aussi cette fonction ne possède-t-elle aucun paramètre utile. C'est tout à fait normal pour une socket, le système de fichiers étant plutôt une abstraction.

27.4.1.2 Cas des pseudo-systèmes de fichiers

La fonction générale `get_sb_pseudo()` est définie dans le fichier `linux/fs/libfs.c`: Code Linux 2.6.10

```
192 /*
193  * Routine d'aide commune aux pseudo-systemes de fichiers (sockfs, pipefs, bdev - choses qui
194  * ne seront jamais montees)
195  */
196 struct super_block *
197 get_sb_pseudo(struct file_system_type *fs_type, char *name,
198         struct super_operations *ops, unsigned long magic)
199 {
200         struct super_block *s = sget(fs_type, NULL, set_anon_super, NULL);
201         static struct super_operations default_ops = {.statfs = simple_statfs};
202         struct dentry *dentry;
203         struct inode *root;
204         struct qstr d_name = {.name = name, .len = strlen(name)};
205
206         if (IS_ERR(s))
207                 return s;
208
209         s->s_flags = MS_NOUSER;
210         s->s_maxbytes = ~0ULL;
211         s->s_blocksize = 1024;
212         s->s_blocksize_bits = 10;
213         s->s_magic = magic;
214         s->s_op = ops ? ops : &default_ops;
215         root = new_inode(s);
216         if (!root)
217                 goto Enomem;
```

```
218         root->i_mode = S_IFDIR | S_IRUSR | S_IWUSR;
219         root->i_uid = root->i_gid = 0;
220         root->i_atime = root->i_mtime = root->i_ctime = CURRENT_TIME;
221         dentry = d_alloc(NULL, &d_name);
222         if (!dentry) {
223                 iput(root);
224                 goto Enomem;
225         }
226         dentry->d_sb = s;
227         dentry->d_parent = dentry;
228         d_instantiate(dentry, root);
229         s->s_root = dentry;
230         s->s_flags |= MS_ACTIVE;
231         return s;
232
233 Enomem:
234         up_write(&s->s_umount);
235         deactivate_super(s);
236         return ERR_PTR(-ENOMEM);
237 }
```

Autrement dit :

– Un descripteur de super-bloc est déclaré, que l'on essaie d'instantier. Si on n'y parvient pas, on renvoie le code fourni par la fonction `sget()`.

Code Linux 2.6.10

La fonction générale `sget()` est définie dans le fichier `linux/fs/super.c`:

```
264 /**
265  *      sget    -       trouver ou creer un super bloc
266  *      @type : type de systemes de fichiers auquel le super bloc doit appartenir
267  *      @test : rappel de comparaison
268  *      @set :  rappel d'initialisation
269  *      @data : argument pour chacun d'eux
270  */
271 struct super_block *sget(struct file_system_type *type,
272                          int (*test)(struct super_block *,void *),
273                          int (*set)(struct super_block *,void *),
274                          void *data)
275 {
276         struct super_block *s = NULL;
277         struct list_head *p;
278         int err;
279
280 retry:
281         spin_lock(&sb_lock);
282         if (test) list_for_each(p, &type->fs_supers) {
283                 struct super_block *old;
284                 old = list_entry(p, struct super_block, s_instances);
285                 if (!test(old, data))
286                         continue;
287                 if (!grab_super(old))
288                         goto retry;
289                 if (s)
290                         destroy_super(s);
291                 return old;
292         }
293         if (!s) {
294                 spin_unlock(&sb_lock);
295                 s = alloc_super();
296                 if (!s)
297                         return ERR_PTR(-ENOMEM);
298                 goto retry;
```

```
299         }
300
301         err = set(s, data);
302         if (err) {
303                 spin_unlock(&sb_lock);
304                 destroy_super(s);
305                 return ERR_PTR(err);
306         }
307         s->s_type = type;
308         strlcpy(s->s_id, type->name, sizeof(s->s_id));
309         list_add_tail(&s->s_list, &super_blocks);
310         list_add(&s->s_instances, &type->fs_supers);
311         spin_unlock(&sb_lock);
312         get_filesystem(type);
313         return s;
314 }
```

Nous ne décrirons pas en détail cette fonction puisqu'elle n'est pas spécifique au type de fichiers des sockets. Elle recherche un descripteur de super-bloc libre dans la liste des descripteurs du système de fichiers en question et elle en renvoie l'adresse, sinon elle en ajoute un et en renvoie l'adresse.

- On déclare un ensemble d'opérations par défaut sur les super-blocs, qui ne comprend que celle concernant le statut.
- On déclare une entrée de répertoire, un descripteur de nœud d'information (qui sera celui de la racine du système de fichiers monté) et un métanom.

  Le type général `struct qstr` des métanoms est défini dans le fichier `linux/include/linux/dcache.h`:

Code Linux 2.6.10

```
27 /*
28  * "quick string" -- facilite le passage des parametres, mais encore plus important,
29  * sauvegarde les "metadonnees" sur la chaine de caracteres (c'est-a-dire la longueur et
      le hachage).
30  *
31  * hash vient en premier, donc il est contre d_parent dans
32  * dentry.
33  */
34 struct qstr {
35         unsigned int hash;
36         unsigned int len;
37         const unsigned char *name;
38 };
```

- Si la valeur du pointeur de super-bloc n'est pas adéquate, on la renvoie.
- On initialise quelques champs du descripteur de super-bloc, en particulier la taille d'un bloc (1 024 octets), le nombre de bits pour cette taille (10 puisque $2^{10} = 1024$), le nombre magique de ce type de fichier (d'après la valeur passée en argument) et enfin l'ensemble des opérations sur les super-blocs de ce type de fichiers (d'après la valeur passée en argument ; `sockfs_ops` dans notre cas, étudié ci-dessous).
- On essaie d'instantier le descripteur de nœud d'information déclaré au début avec un descripteur qui n'est pas utilisé. Si on n'y parvient pas, on renvoie l'opposé du code d'erreur `ENOMEM`. Sinon on indique le mode pour un nœud d'information de socket (pas de répertoire, lecture et écriture pour l'utilisateur), l'identificateur de son propriétaire et l'identificateur du groupe d'utilisateurs (tous les deux égaux à 0, donc se rapportant au super-utilisateur) ainsi que les différentes estampilles de temps (toutes initialisées à l'heure en cours).
- On essaie d'allouer une entrée de répertoire avec le nom passé en paramètre ("socket:" dans notre cas). Si on n'y parvient pas, on libère le descripteur de nœud d'information et on renvoie l'opposé du code d'erreur `ENOMEM`.

– On initialise ce répertoire avec les fichiers obligatoires point '.' et point-point '..'.
– On associe cette entrée de répertoire au descripteur de nœud d'information.
– On associe cette entrée de répertoire au descripteur de super-bloc et on renvoie l'adresse de celui-ci.

27.4.2 Libération d'un super-bloc

Nous avons vu ci-dessus que la fonction de libération du super-bloc d'un système de fichiers de type socket s'appelle `kill_anon_super()`. Cette fonction générale est définie dans le fichier `linux/fs/super.c`:

Code Linux 2.6.10

```
622 void kill_anon_super(struct super_block *sb)
623 {
624         int slot = MINOR(sb->s_dev);
625
626         generic_shutdown_super(sb);
627         spin_lock(&unnamed_dev_lock);
628         idr_remove(&unnamed_dev_idr, slot);
629         spin_unlock(&unnamed_dev_lock);
630 }
```

qui fait essentiellement appel à la fonction `generic_shutdown_super()` chargée de la partie de la libération indépendante du système de fichiers. Celle-ci est définie dans le même fichier :

Code Linux 2.6.10

```
213 /**
214  *      generic_shutdown_super  -       routine d'aide commune pour ->kill_sb()
215  *      @sb : super bloc a detruire
216  *
217  *      generic_shutdown_super() fait tout le travail independant du fs en fermeture de
218  *      super-bloc. Une ->kill_sb() typique devrait amasser tous les objets specifiques au
219  *      fs que necessite la destruction du super bloc, appeler generic_shutdown_super()
220  *      puis liberer les objets mentionnes ci-dessus. Note : on _doit_ prendre soin des
221  *      dentries et des inodes et ne pas avoir besoin de routine specifique.
222  */
223 void generic_shutdown_super(struct super_block *sb)
224 {
225         struct dentry *root = sb->s_root;
226         struct super_operations *sop = sb->s_op;
227
228         if (root) {
229                 sb->s_root = NULL;
230                 shrink_dcache_parent(root);
231                 shrink_dcache_anon(&sb->s_anon);
232                 dput(root);
233                 fsync_super(sb);
234                 lock_super(sb);
235                 lock_kernel();
236                 sb->s_flags &= ~MS_ACTIVE;
237                 /* mauvais nom - ce devrait etre evict_inodes() */
238                 invalidate_inodes(sb);
239
240                 if (sop->write_super && sb->s_dirt)
241                         sop->write_super(sb);
242                 if (sop->put_super)
243                         sop->put_super(sb);
244
245                 /* Oublier tous les inodes restants */
246                 if (invalidate_inodes(sb)) {
247                         printk("VFS: Busy inodes after unmount. "
248                            "Self-destruct in 5 seconds.  Have a nice day...\n");
249                 }
```

```
250
251                 unlock_kernel();
252                 unlock_super(sb);
253         }
254         spin_lock(&sb_lock);
255         /* devrait etre initialise pour __put_super_and_need_restart() */
256         list_del_init(&sb->s_list);
257         list_del(&sb->s_instances);
258         spin_unlock(&sb_lock);
259         up_write(&sb->s_umount);
260 }
```

27.5 Opérations sur les super-blocs

27.5.1 Ensemble des opérations

Nous avons vu que, pour obtenir un super-bloc de socket, on doit passer en argument l'ensemble des opérations sur les super-blocs. Celui-ci, `sockfs_ops`, est défini dans le fichier `linux/net/socket.c`:

Code Linux 2.6.10

```
321 static struct super_operations sockfs_ops = {
322         .alloc_inode =  sock_alloc_inode,
323         .destroy_inode =sock_destroy_inode,
324         .statfs =       simple_statfs,
325 };
```

Seules trois opérations sont donc nécessaires pour les super-blocs de socket sur les dix-huit prévues dans le cas général (comme vu ci-dessus) : celle qui alloue un descripteur de nœud d'information, celle qui le libère et celle qui renvoie le statut du système de fichiers.

27.5.2 Statut du système de fichiers

La fonction générale `simple_statfs()` est définie dans le fichier `linux/fs/libfs.c`:

Code Linux 2.6.10

```
21 int simple_statfs(struct super_block *sb, struct kstatfs *buf)
22 {
23         buf->f_type = sb->s_magic;
24         buf->f_bsize = PAGE_CACHE_SIZE;
25         buf->f_namelen = NAME_MAX;
26         return 0;
27 }
```

Autrement dit elle renseigne les champs nombre magique avec `SOCKFS_MAGIC`, taille d'un bloc avec 1 024 (octets) et nombre maximum de caractères avec 255 pour le nom d'une socket dans un tampon du type `struct kstatfs`.

Celui-ci est défini dans le fichier `linux/include/linux/statfs.h` que nous reproduisons intégralement :

Code Linux 2.6.10

```
1  #ifndef _LINUX_STATFS_H
2  #define _LINUX_STATFS_H
3
4  #include <linux/types.h>
5
6  #include <asm/statfs.h>
7
8  struct kstatfs {
9          long            f_type;
10         long            f_bsize;
11         sector_t        f_blocks;
```

```
12         sector_t        f_bfree;
13         sector_t        f_bavail;
14         sector_t        f_files;
15         sector_t        f_ffree;
16         __kernel_fsid_t f_fsid;
17         long            f_namelen;
18         long            f_frsize;
19         long            f_spare[5];
20 };
21
22 #endif
```

27.5.3 Allocation d'un descripteur de nœud d'information

Code Linux 2.6.10

La fonction `sock_alloc_inode()` est définie dans le fichier `linux/net/socket.c`:

```
276 static struct inode *sock_alloc_inode(struct super_block *sb)
277 {
278         struct socket_alloc *ei;
279         ei = (struct socket_alloc *)kmem_cache_alloc(sock_inode_cachep, SLAB_KERNEL);
280         if (!ei)
281                 return NULL;
282         init_waitqueue_head(&ei->socket.wait);
283
284         ei->socket.fasync_list = NULL;
285         ei->socket.state = SS_UNCONNECTED;
286         ei->socket.flags = 0;
287         ei->socket.ops = NULL;
288         ei->socket.sk = NULL;
289         ei->socket.file = NULL;
290         ei->socket.passcred = 0;
291
292         return &ei->vfs_inode;
293 }
```

Autrement dit:

- On déclare une entité du type `struct socket_alloc`.

  Code Linux 2.6.10

  Ce type est défini dans le fichier `linux/include/net/sock.h`:

  ```
  644 struct socket_alloc {
  645         struct socket socket;
  646         struct inode vfs_inode;
  647 };
  ```

  et comprend deux champs: un descripteur de socket et un descripteur de nœud d'information.
- On essaie d'instantier cette entité en lui réservant de la place mémoire dans l'espace noyau. Si on n'y parvient pas, on renvoie `NULL`.

  Code Linux 2.6.10

  La variable `sock_inode_cachep` est définie dans le fichier `linux/net/sockect.c`:

  ```
  274 static kmem_cache_t * sock_inode_cachep;
  ```

- On renseigne les champs du descripteur de socket.
- On renvoie l'adresse du descripteur de nœud d'information.

Astuce

Remarquons que l'on utilise une astuce (est-ce bien venu?): le descripteur de socket associé à ce descripteur de nœud d'information est déterminé par son emplacement dans l'espace noyau (un nombre déterminé d'octets avant l'emplacement du descripteur de nœud d'information) au lieu d'un champ spécifique.

27.5.4 Libération d'un descripteur de nœud d'information

La fonction `sock_destroy_inode()` est définie dans le fichier `linux/net/socket.c`: Code Linux 2.6.10

```
295 static void sock_destroy_inode(struct inode *inode)
296 {
297         kmem_cache_free(sock_inode_cachep,
298                         container_of(inode, struct socket_alloc, vfs_inode));
299 }
```

Il n'y a pas grand chose à en dire, sinon l'utilisation de l'astuce notée ci-dessus, mais maintenant à travers l'utilisation de la macro générale `container_of()`, définie dans le fichier `linux/include/linux/kernel.h`: Code Linux 2.6.10

```
232 /**
233  * container_of - convertit un membre d'une structure en dehors de la structure qui le
      contient
234  *
235  * @ptr :       le pointeur au membre.
236  * @type :      le type de la struct conteneur dans laquelle il est contenu.
237  * @member :    le nom du membre dans la struct.
238  *
239  */
240 #define container_of(ptr, type, member) ({                      \
241         const typeof( ((type *)0)->member ) *__mptr = (ptr);    \
242         (type *)( (char *)__mptr - offsetof(type,member) );})
```

27.6 Opérations sur les répertoires de socket

Nous avons vu ci-dessus que, pour chaque type de fichiers, il faut définir l'ensemble des opérations permises sur les répertoires de ce type, utilisé par le système de fichier virtuel. Dans le cas des sockets cette structure, `sockfs_dentry_operations`, est définie dans le fichier `linux/net/socket.c`: Code Linux 2.6.10

```
344 static struct dentry_operations sockfs_dentry_operations = {
345         .d_delete =     sockfs_delete_dentry,
346 };
```

Une seule opération est donc nécessaire pour les répertoires de socket sur les six prévues dans le cas général, celle qui concerne la destruction d'une entrée de répertoire.

Cette fonction `sockfs_delete_dentry()` est définie dans le même fichier: Code Linux 2.6.10

```
340 static int sockfs_delete_dentry(struct dentry *dentry)
341 {
342         return 1;
343 }
```

Elle se contente de renvoyer qu'il y a une erreur (puisque la notion de répertoire n'a pas de sens pour les sockets).

27.7 Opérations sur les fichiers de type socket

Nous avons vu ci-dessus que, pour chaque type de fichiers, il faut définir une structure des opérations sur les fichiers de ce type, utilisée par le système de fichier virtuel. Cette structure, `socket_file_ops`, est définie dans le fichier `linux/net/socket.c` dans le cas des sockets, précédée de la liste des fichiers inclus nécessaires et des prototypes des douze fonctions nécessaires pour définir cette structure: Code Linux 2.6.10

```
61  #include <linux/config.h>
62  #include <linux/mm.h>
63  #include <linux/smp_lock.h>
64  #include <linux/socket.h>
65  #include <linux/file.h>
66  #include <linux/net.h>
67  #include <linux/interrupt.h>
68  #include <linux/netdevice.h>
69  #include <linux/proc_fs.h>
70  #include <linux/seq_file.h>
71  #include <linux/wanrouter.h>
72  #include <linux/if_bridge.h>
73  #include <linux/init.h>
74  #include <linux/poll.h>
75  #include <linux/cache.h>
76  #include <linux/module.h>
77  #include <linux/highmem.h>
78  #include <linux/divert.h>
79  #include <linux/mount.h>
80  #include <linux/security.h>
81  #include <linux/syscalls.h>
82  #include <linux/compat.h>
83  #include <linux/kmod.h>
84
85  #ifdef CONFIG_NET_RADIO
86  #include <linux/wireless.h>             /* Note : WIRELESS_EXT a definir */
87  #endif  /* CONFIG_NET_RADIO */
88
89  #include <asm/uaccess.h>
90  #include <asm/unistd.h>
91
92  #include <net/compat.h>
93
94  #include <net/sock.h>
95  #include <linux/netfilter.h>
96
97  static int sock_no_open(struct inode *irrelevant, struct file *dontcare);
98  static ssize_t sock_aio_read(struct kiocb *iocb, char __user *buf,
99                               size_t size, loff_t pos);
100 static ssize_t sock_aio_write(struct kiocb *iocb, const char __user *buf,
101                               size_t size, loff_t pos);
102 static int sock_mmap(struct file *file, struct vm_area_struct * vma);
103
104 static int sock_close(struct inode *inode, struct file *file);
105 static unsigned int sock_poll(struct file *file,
106                               struct poll_table_struct *wait);
107 static int sock_ioctl(struct inode *inode, struct file *file,
108                       unsigned int cmd, unsigned long arg);
109 static int sock_fasync(int fd, struct file *filp, int on);
110 static ssize_t sock_readv(struct file *file, const struct iovec *vector,
111                           unsigned long count, loff_t *ppos);
112 static ssize_t sock_writev(struct file *file, const struct iovec *vector,
113                           unsigned long count, loff_t *ppos);
114 static ssize_t sock_sendpage(struct file *file, struct page *page,
115                              int offset, size_t size, loff_t *ppos, int more);
116
117
118 /*
119  *      Les fichiers socket possedent un ensemble d'operations 'speciales' ainsi que
        des operations de fichier generique. Celles-ci n'apparaissent pas
120  *      dans la structure des operations mais sont effectuees directement a travers le
        multiplexeur socketcall().
```

```
121  */
122
123 static struct file_operations socket_file_ops = {
124         .owner =        THIS_MODULE,
125         .llseek =       no_llseek,
126         .aio_read =     sock_aio_read,
127         .aio_write =    sock_aio_write,
128         .poll =         sock_poll,
129         .ioctl =        sock_ioctl,
130         .mmap =         sock_mmap,
131         .open =         sock_no_open,   /* code open special pour refuser open via /proc */
132         .release =      sock_close,
133         .fasync =       sock_fasync,
134         .readv =        sock_readv,
135         .writev =       sock_writev,
136         .sendpage =     sock_sendpage
137 };
```

On remarquera que seules douze des vingt et une fonctions prévues dans le cas général sont nécessaires pour les sockets.

27.8 Premières implémentations de fonctions

Un certain nombre de fonctions sur les fichiers de type socket sont implémentées mais n'ont pas vraiment d'intérêt. Voyons-les à la fin de ce chapitre, que l'on peut passer sans problème.

27.8.1 Implémentation du positionnement

La fonction générale `no_llseek()` est définie dans le fichier `linux/fs/read_write.c`: Code Linux 2.6.10

```
81 loff_t no_llseek(struct file *file, loff_t offset, int origin)
82 {
83         return -ESPIPE;
84 }
```

Elle se contente de renvoyer l'opposé d'un code d'erreur puisqu'il n'y a pas d'accès direct pour les tubes de communication, plus exactement pour les sockets dans notre cas.

27.8.2 Implémentation de la scrutation

27.8.2.1 Fonction principale

La fonction `sock_poll()`, qui permet de déterminer s'il existe des données disponibles sur la socket, est définie dans le fichier `linux/net/socket.c`: Code Linux 2.6.10

```
932 /* Pas de verrou noyau detenu - parfait */
933 static unsigned int sock_poll(struct file *file, poll_table * wait)
934 {
935         struct socket *sock;
936
937         /*
938          *      Nous ne devons pas renvoyer d'erreurs a poll, aussi est-ce oui ou non.
939          */
940         sock = SOCKET_I(file->f_dentry->d_inode);
941         return sock->ops->poll(file, sock, wait);
942 }
```

Autrement dit :

- on charge le descripteur de socket associé au descripteur de fichier passé en paramètre, grâce à la fonction auxiliaire `SOCKET_I()` étudiée ci-après ;
- on utilise la fonction de scrutation spécifique à la famille de protocoles et on renvoie ce qui est transmis par celle-ci.

27.8.2.2 Descripteur de socket associé à un nœud d'information de socket

La fonction :

```
struct socket *SOCKET_I(struct inode *inode);
```

permet de connaître le descripteur de socket auquel un nœud d'information (de socket) est associé. Bien entendu s'il ne s'agit pas d'un nœud d'information de socket, `NULL` est renvoyé.

Cette fonction en ligne est définie, d'une façon on ne peut plus simple, dans le fichier `linux/include/net/sock.h` :

Code Linux 2.6.10

```
649 static inline struct socket *SOCKET_I(struct inode *inode)
650 {
651         return &container_of(inode, struct socket_alloc, vfs_inode)->socket;
652 }
```

27.8.2.3 Cas de IPv4/UDP

Nous avons vu au chapitre 26 que, dans le cas de IPv4/UDP, l'ensemble des opérations sur les descripteurs de socket s'appelle `inet_dgram_ops`. Cet ensemble est défini dans le fichier `linux/net/ipv4/af_inet.c` :

Code Linux 2.6.10

```
803 struct proto_ops inet_dgram_ops = {
804         .family =       PF_INET,
805         .owner =        THIS_MODULE,
806         .release =      inet_release,
807         .bind =         inet_bind,
808         .connect =      inet_dgram_connect,
809         .socketpair =   sock_no_socketpair,
810         .accept =       sock_no_accept,
811         .getname =      inet_getname,
812         .poll =         udp_poll,
813         .ioctl =        inet_ioctl,
814         .listen =       sock_no_listen,
815         .shutdown =     inet_shutdown,
816         .setsockopt =   sock_common_setsockopt,
817         .getsockopt =   sock_common_getsockopt,
818         .sendmsg =      inet_sendmsg,
819         .recvmsg =      sock_common_recvmsg,
820         .mmap =         sock_no_mmap,
821         .sendpage =     inet_sendpage,
822 };
```

Le nom de la fonction spécifique de la fonction de scrutation dans le cas du couple IPv4/UDP est donc `datagram_poll()`, mais cela ne nous intéressera pas ici.

27.8.3 Implémentation du non mappage en mémoire vive

La fonction `sock_no_mmap()` est définie dans le fichier `linux/net/core/sock.c` :

Code Linux 2.6.10

```
1055 int sock_no_mmap(struct file *file, struct socket *sock, struct vm_area_struct *vma)
1056 {
```

```
1057         /* Code d'erreur de la methode mmap manquant */
1058         return -ENODEV;
1059 }
```

qui renvoie l'opposé du code d'erreur `ENODEV`. En effet, cette fonction est prévue pour copier le contenu intégral d'un fichier en mémoire vive pour accélérer les opérations sur celui-ci, ce qui n'est pas possible pour une socket.

27.8.4 Implémentation de l'ouverture de fichier

La fonction `sock_no_open()` est définie dans le fichier `linux/net/socket.c`: Code Linux 2.6.10

```
483 /*
484  *      En theorie vous ne pouvez pas ouvrir cet inode, mais /proc fournit
485  *      une porte derobee. Rappelons qu'il faut la garder fermee, sinon vous allez y
486  *      deposer des betes rampantes.
487  */
488
489 static int sock_no_open(struct inode *irrelevant, struct file *dontcare)
490 {
491         return -ENXIO;
492 }
```

Elle se contente de renvoyer l'opposé du code d'erreur `ENXIO`, puisqu'on ne peut pas réouvrir une socket. Elle est ouverte lors de sa création, un point c'est tout.

27.8.5 Implémentation des événements asynchrones

La fonction `sock_fasync()` est définie dans le fichier `linux/net/socket.c`: Code Linux 2.6.10

```
968  /*
969   *      Met a jour la liste async de la socket
970   *
971   *      Strategie de verrouillage de fasync_list.
972   *
973   *      1. fasync_list ne peut etre modifiee que par un processus ayant le verrou de
974   *         la socket c'est-a-dire avec semaphore.
975   *      2. fasync_list est utilise avec read_lock(&sk->sk_callback_lock)
976   *         ou le verrou de la socket.
977   *      3. fasync_list peut etre utilisee dans un contexte softirq, donc cette
978   *         modification avec verrou de socket doit etre effectuee avec
979   *         write_lock_bh(&sk->sk_callback_lock).
980   *                                                      --ANK (990710)
981   */
982
983  static int sock_fasync(int fd, struct file *filp, int on)
984  {
985          struct fasync_struct *fa, *fna=NULL, **prev;
986          struct socket *sock;
987          struct sock *sk;
988
989          if (on)
990          {
991                  fna=(struct fasync_struct *)kmalloc(sizeof(struct fasync_struct),
                        GFP_KERNEL);
992                  if(fna==NULL)
993                          return -ENOMEM;
994          }
995
996          sock = SOCKET_I(filp->f_dentry->d_inode);
997
```

```
998         if ((sk=sock->sk) == NULL) {
999                 if (fna)
1000                        kfree(fna);
1001                return -EINVAL;
1002        }
1003
1004        lock_sock(sk);
1005
1006        prev=&(sock->fasync_list);
1007
1008        for (fa=*prev; fa!=NULL; prev=&fa->fa_next,fa=*prev)
1009                if (fa->fa_file==filp)
1010                        break;
1011
1012        if(on)
1013        {
1014                if(fa!=NULL)
1015                {
1016                        write_lock_bh(&sk->sk_callback_lock);
1017                        fa->fa_fd=fd;
1018                        write_unlock_bh(&sk->sk_callback_lock);
1019
1020                        kfree(fna);
1021                        goto out;
1022                }
1023                fna->fa_file=filp;
1024                fna->fa_fd=fd;
1025                fna->magic=FASYNC_MAGIC;
1026                fna->fa_next=sock->fasync_list;
1027                write_lock_bh(&sk->sk_callback_lock);
1028                sock->fasync_list=fna;
1029                write_unlock_bh(&sk->sk_callback_lock);
1030        }
1031        else
1032        {
1033                if (fa!=NULL)
1034                {
1035                        write_lock_bh(&sk->sk_callback_lock);
1036                        *prev=fa->fa_next;
1037                        write_unlock_bh(&sk->sk_callback_lock);
1038                        kfree(fa);
1039                }
1040        }
1041
1042 out:
1043        release_sock(sock->sk);
1044        return 0;
1045 }
```

27.8.6 Implémentation de la libération de fichier

27.8.6.1 Fonction principale

Code Linux 2.6.10

La fonction `sock_close()` est définie dans le fichier `linux/net/socket.c`:

```
951 int sock_close(struct inode *inode, struct file *filp)
952 {
953         /*
954          *      Il est possible que le inode soit NULL. Nous serions alors
955          *      en train de fermer une socket indefinie.
956          */
957
```

```
958         if (!inode)
959         {
960                 printk(KERN_DEBUG "sock_close: NULL inode\n");
961                 return 0;
962         }
963         sock_fasync(-1, filp, 0);
964         sock_release(SOCKET_I(inode));
965         return 0;
966 }
```

Autrement dit on libère les ressources adéquates:

- si aucun descripteur de nœud d'information n'est affecté à la socket, c'est que celle-ci a déjà été fermée; on affiche donc un message d'erreur et on renvoie 0;
- sinon on libère le descripteur de fichier, le descripteur de socket et on renvoie 0.

27.8.6.2 Libération d'un descripteur de socket

La fonction `sock_release()`, fonction interne de libération d'un descripteur de socket, est définie dans le fichier `linux/net/socket.c`: Code Linux 2.6.10

```
499 /**
500  *      sock_release    -       ferme une socket
501  *      @sock : socket a fermer
502  *
503  *      La socket est enlevee de la pile de protocoles si elle a un rappel de liberation,
504  *      puis l'inode est alors libere si la socket est liee a un
505  *      inode et non a un fichier.
506  */
507
508 void sock_release(struct socket *sock)
509 {
510         if (sock->ops) {
511                 struct module *owner = sock->ops->owner;
512
513                 sock->ops->release(sock);
514                 sock->ops = NULL;
515                 module_put(owner);
516         }
517
518         if (sock->fasync_list)
519                 printk(KERN_ERR "sock_release: fasync list not empty!\n");
520
521         get_cpu_var(sockets_in_use)--;
522         put_cpu_var(sockets_in_use);
523         if (!sock->file) {
524                 iput(SOCK_INODE(sock));
525                 return;
526         }
527         sock->file=NULL;
528 }
```

Autrement dit:

- s'il existe une opération spécifique de libération de socket associée à la famille de protocole, on fait appel à elle;
- si la liste `fasync` n'est pas vide, on affiche un message d'erreur;
- on décrémente le nombre de sockets en utilisation sur le microprocesseur concerné;
- on libère le nœud d'information associé à la socket.

27.8.6.3 Cas de IPv4

Comme nous l'avons vu à propos de `inet_dgram_ops`, le nom de la fonction spécifique de libération d'un descripteur de socket dans le cas de IPv4, que ce soit pour TCP ou pour UDP, est `inet_release()`. Elle est définie dans le fichier `linux/net/ipv4/af_inet.c`:

Code Linux 2.6.10

```
353 /*
354  *      La socket paire devrait toujours etre NULL (ou autre). Lorsque nous appelons cette
355  *      fonction, nous sommes en train de detruire l'objet et donc plus personne ne
356  *      peut lui faire reference.
357  */
358 int inet_release(struct socket *sock)
359 {
360         struct sock *sk = sock->sk;
361
362         if (sk) {
363                 long timeout;
364
365                 /* Les applications oublient de quitter les groupes avant de sortir */
366                 ip_mc_drop_socket(sk);
367
368                 /* Si linger est positionne, nous ne renverrons rien avant que la fermeture
369                  * soit complete. Sinon nous renvoyons immediatement. Actuellement la
370                  * fermeture est faite de la meme facon.
371                  *
372                  * Si la fermeture est due a la mort du processus, nous n'avons
373                  * jamais linger...
374                  */
375                 timeout = 0;
376                 if (sock_flag(sk, SOCK_LINGER) &&
377                     !(current->flags & PF_EXITING))
378                         timeout = sk->sk_lingertime;
379                 sock->sk = NULL;
380                 sk->sk_prot->close(sk, timeout);
381         }
382         return 0;
383 }
```

Chapitre 28

Création et fermeture des sockets

Nous avons vu au chapitre 6 un exemple de programmation avec l'API des sockets dans le cas d'une communication non connectée (ce qui correspond à UDP dans le cas de TCP/IP). Les opérations les plus importantes sont évidemment l'envoi et la réception de datagrammes mais avant cela il faut créer une socket, éventuellement initialiser dans le cas d'un serveur et après cela, fermer la socket. Nous allons étudier l'implémentation des appels systèmes de création, de semi-arrêt et de fermeture des sockets dans ce chapitre.

L'implémentation de chacun de ces appels système comprend une partie générale, indépendante de tout protocole, une partie dépendant de la famille d'adresses (ce qui correspond au protocole de la couche réseau) et une partie dépendant du type de communication (ce qui correspond au protocole de la couche transport). Nous choisirons toujours, à titre d'illustration, le cas de IPv4 pour la couche réseau et la cas UDP pour la couche transport.

28.1 Création d'une socket

La fonction utilisateur `socket()` est définie, comme pour tout appel système, grâce à la fonction d'appel `sys_socket()`. Celle-ci fait appel à la fonction générale `sock_create()` qui renvoie à une fonction de création d'une socket spécifique à la famille de protocoles et au type de communication indiqués.

28.1.1 Traitement général

28.1.1.1 Fonction d'appel

Code Linux 2.6.10

La fonction d'appel `sys_socket()` est définie dans le fichier `linux/net/socket.c`:

```
1187 asmlinkage long sys_socket(int family, int type, int protocol)
1188 {
1189         int retval;
1190         struct socket *sock;
1191
1192         retval = sock_create(family, type, protocol, &sock);
1193         if (retval < 0)
1194                 goto out;
1195
1196         retval = sock_map_fd(sock);
1197         if (retval < 0)
1198                 goto out_release;
1199
1200 out:
1201         /* Il peut y avoir deja un autre descripteur 8) Ce n'est pas le probleme du
               noyau. */
1202         return retval;
1203
1204 out_release:
1205         sock_release(sock);
1206         return retval;
1207 }
```

Autrement dit :

- On déclare une valeur de retour et un descripteur de socket.
- On fait appel à la fonction `sock_create()`, étudiée ci-après, qui a pour fonction d'instantier le descripteur de socket et de renvoyer 0 ou l'opposé d'un code d'erreur. S'il y a un problème, on renvoie l'opposé du code d'erreur fourni par cette fonction.
- On instantie un descripteur de fichier (de socket), grâce à la fonction `sock_map_fd()` étudiée ci-après, associé au descripteur de socket instantié ci-dessus. S'il y a un problème, on relâche le descripteur de socket et on renvoie l'opposé du code d'erreur adéquat. Sinon on renvoie le numéro de fichier.

28.1.1.2 Instantiation d'une socket

La fonction :

```
int sock_create(int family, int type, int protocol, struct socket **res);
```

vérifie que la famille de protocole `family` et le type de communication `type` sont implémentés, fait appel à la fonction spécifique de création de socket pour ceux-ci et place le descripteur de socket ainsi créé dans `res`. Elle renvoie 0 en cas de succès, l'opposé d'un code d'erreur sinon.

La fonction `int sock_create()` est définie dans le fichier `linux/net/socket.c`: Code Linux 2.6.10

```
1074 static int __sock_create(int family, int type, int protocol, struct socket **res, int kern)
1075 {
1076         int i;
1077         int err;
1078         struct socket *sock;
1079
1080         /*
1081          *      Verifie si le protocole est dans l'intervalle
1082          */
1083         if (family < 0 || family >= NPROTO)
1084                 return -EAFNOSUPPORT;
1085         if (type < 0 || type >= SOCK_MAX)
1086                 return -EINVAL;
1087
1088         /* Compatibilite.
1089
1090            Ce truc affreux a ete deplace de la couche INET jusqu'ici pour eviter
1091             une impasse lors du chargement du module.
1092          */
1093         if (family == PF_INET && type == SOCK_PACKET) {
1094                 static int warned;
1095                 if (!warned) {
1096                         warned = 1;
1097                         printk(KERN_INFO "%s uses obsolete (PF_INET,SOCK_PACKET)\n",
                                        current->comm);
1098                 }
1099                 family = PF_PACKET;
1100         }
1101
1102         err = security_socket_create(family, type, protocol, kern);
1103         if (err)
1104                 return err;
1105
1106 #if defined(CONFIG_KMOD)
1107         /* Essaie de charger un module de protocole si la recherche a echouee.
1008          *
1009          * 12/09/1996 Marcin : Mais ! ceci a REELLEMENT un sens si l'utilisateur
1010          * demande un support de reseau bien caracterise et reel lors de la configuration.
1011          * Sinon le support du module se brisera !
1012          */
1013         if (net_families[family]==NULL)
1014         {
1015                 request_module("net-pf-%d",family);
1016         }
1017 #endif
1018
1019         net_family_read_lock();
1020         if (net_families[family] == NULL) {
1021                 i = -EAFNOSUPPORT;
1022                 goto out;
1023         }
1024
1025 /*
1026  *      Alloue la socket et permet a la famille d'activer des choses. Si
1027  *      le protocole est 0, la famille a pour instruction de selectionner une
1028  *      valeur par defaut appropriee.
1029  */
1030
1031         if (!(sock = sock_alloc()))
1032         {
1033                 printk(KERN_WARNING "socket: no more sockets\n");
```

```
1134                i = -ENFILE;            /* Ne concorde pas exactement, mais c'est la
1135                                           chose posix la plus proche */
1136                goto out;
1137        }
1138
1139        sock->type  = type;
1140
1141        /*
1142         * Nous appellerons la fonction ->create, qui est certainement dans un module
1143         * chargeable, aussi avons-nous a tamponner le refcnt du module chargeable d'abord.
1144         */
1145        i = -EAFNOSUPPORT;
1146        if (!try_module_get(net_families[family]->owner))
1147                goto out_release;
1148
1149        if ((i = net_families[family]->create(sock, protocol)) < 0)
1150                goto out_module_put;
1151        /*
1152         * Tamponner maintenant le refcnt du module [chargeable] qui possede cette
1153         * socket. Au moment de sock_release, nous decrementons ce refcnt.
1154         */
1155        if (!try_module_get(sock->ops->owner)) {
1156                sock->ops = NULL;
1157                goto out_module_put;
1158        }
1159        /*
1160         * Maintenant que nous avons fini avec la fonction ->create, le module
1161         * [chargeable] peut avoir son refcnt decremente
1162         */
1163        module_put(net_families[family]->owner);
1164        *res = sock;
1165        security_socket_post_create(sock, family, type, protocol, kern);
1166
1167 out:
1168        net_family_read_unlock();
1169        return i;
1170 out_module_put:
1171        module_put(net_families[family]->owner);
1172 out_release:
1173        sock_release(sock);
1174        goto out;
1175 }
1176
1177 int sock_create(int family, int type, int protocol, struct socket **res)
1178 {
1179        return __sock_create(family, type, protocol, res, 0);
1180 }
```

Autrement dit :

- On déclare un descripteur de socket.
- Si le type de famille de protocoles n'appartient pas à l'intervalle adéquat, on renvoie l'opposé du code d'erreur `EAFNOSUPPORT`.
- Si le type de communication n'appartient pas à l'intervalle adéquat, on renvoie l'opposé du code d'erreur `EINVAL`.
- L'interface ayant évoluée, un petit message est indiqué si on utilise le couple "PF_INET" et "SOCK_PACKET" au lieu du couple "PF_PACQUET" et "SOCK_PACKET", puis le type de famille de protocoles voulu est postionné.
- On fait appel à la fonction `security_socket_create()`, sur laquelle nous allons revenir ci-dessous, et on s'arrête éventuellement en renvoyant le code d'erreur fourni par celle-ci.

- Si la famille de protocoles n'est pas installée, on essaie de charger le module correspondant si l'on permet les modules.
- Si la famille de protocoles n'est (toujours) pas installée, on renvoie l'opposé du code d'erreur EAFNOSUPPORT.

Les fonctions en ligne net_family_read_lock() et net_family_read_unlock() sont définies dans le fichier linux/net/socket.c:

Code Linux 2.6.10

```
171 static __inline__ void net_family_read_lock(void)
172 {
173         atomic_inc(&net_family_lockct);
174         spin_unlock_wait(&net_family_lock);
175 }
176
177 static __inline__ void net_family_read_unlock(void)
178 {
179         atomic_dec(&net_family_lockct);
180 }
181
182 #else
183 #define net_family_write_lock() do { } while(0)
184 #define net_family_write_unlock() do { } while(0)
185 #define net_family_read_lock() do { } while(0)
186 #define net_family_read_unlock() do { } while(0)
187 #endif
```

- On essaie d'instantier le descripteur de socket, grâce à la fonction sock_alloc() étudiée ci-après. Si on n'y parvient pas, on affiche un message d'erreur et on renvoie l'opposé du code d'erreur ENFILE.
- On renseigne le champ type de communication du descripteur de socket avec celui passé en argument.
- Si la famille de protocoles a ete chargée depuis un module, on essaie d'incrémenter le compteur de références de celui-ci. Si on n'y parvient pas, on libère le descripteur de socket, on déverrouille la famille et on renvoie l'opposé du code d'erreur EAFNOSUPPORT.

  La fonction en ligne générale try_module_get() est définie dans le fichier linux/include/linux/module.h.
- On essaie de renseigner les champs du descripteur de socket en utilisant la fonction de création spécifique à la famille de protocoles. Si on n'y parvient pas, on décrémente le compteur de référence du module de la famille de protocoles, on libère le descripteur de socket et on renvoie l'opposé du code d'erreur fourni par la fonction spécifique de création.

  La fonction en ligne générale module_put() est définie dans le fichier linux/include/linux/module.h.
- On essaie d'incrémenter le compteur de références du module correspondant à l'ensemble des opérations. Si on n'y parvient pas, on positionne à NULL le champ ops du descripteur de socket, on décrémente le compteur de référence au module de la famille de protocoles, on libère le descripteur de socket, on déverrouille la famille et on renvoie 0.
- On décrémente le compteur de référence du module de la famille de protocoles.
- Le résultat est renseigné avec le descripteur de socket obtenu.
- On fait appel à la fonction de sécurité security_socket_post_create().
- On déverrouille la lecture pour la famille de protocoles et on renvoie 0.

28.1.1.3 Instantiation d'un descripteur de socket

Code Linux 2.6.10

La fonction `sock_alloc()` permet d'obtenir un nouveau descripteur de socket, paramétré grâce à la variable globale `sock_mnt`. Elle est définie dans le fichier `linux/net/socket.c`:

```
454 /**
455  *      sock_alloc      -       alloue une socket
456  *
457  *      Alloue un nouveau inode et un nouveau objet socket. Les deux sont lies ensemble
458  *      et initialises. La socket est alors renvoyee. Si nous sommes a court d'inodes,
459  *      NULL est renvoye.
460  */
461
462 static struct socket *sock_alloc(void)
463 {
464         struct inode * inode;
465         struct socket * sock;
466
467         inode = new_inode(sock_mnt->mnt_sb);
468         if (!inode)
469                 return NULL;
470
471         sock = SOCKET_I(inode);
472
473         inode->i_mode = S_IFSOCK|S_IRWXUGO;
474         inode->i_sock = 1;
475         inode->i_uid = current->fsuid;
476         inode->i_gid = current->fsgid;
477
478         get_cpu_var(sockets_in_use)++;
479         put_cpu_var(sockets_in_use);
480         return sock;
481 }
```

Autrement dit :

- Un descripteur de nœud d'information et un descripteur de socket sont déclarés.
- On essaie d'instantier le descripteur de nœud d'information et de positionner son point de montage, initialisé grâce à la variable `sock_mnt`. Si on n'y parvient pas, on renvoie `NULL`.

  Code Linux 2.6.10

  La variable `sock_mnt` est définie un peu plus haut dans le même fichier :

  ```
  333 static struct vfsmount *sock_mnt;
  ```

- L'adresse du descripteur de socket est initialisée avec celle du descripteur de socket associé au descripteur de nœud d'information.
- On renseigne un certain nombre de champs du descripteur de nœud d'information (son mode d'accès, le fait qu'il soit un nœud d'information de socket, l'utilisateur et le groupe d'utilisateur associés).
- On incrémente le nombre de sockets en utilisation sur le microprocesseur.

  Code Linux 2.6.10

  La variable `sockets_in_use`, une par microprocesseur, est définie plus haut dans le même fichier :

  ```
  190 /*
  191  *      Compteurs de statistiques des listes de sockets
  192  */
  193
  194 static DEFINE_PER_CPU(int, sockets_in_use) = 0;
  ```

- On renvoie l'adresse du descripteur de socket.

28.1.1.4 Points d'ancrage des opérations sécurisées

Nous allons rencontrer plusieurs fois des points d'ancrage à des opérations sécurisées dans la version 2.6.10 de Linux. Il s'agit d'une option de configuration du noyau. Par défaut ces opérations ne font rien. Nous ne nous intéresseront pas à elles dans une première étape. Nous nous contenterons de donner quelques indications.

Une variable `security_ops` est définie dans le fichier `linux/security/security.c`: Code Linux 2.6.10

```
27 struct security_operations *security_ops;      /* Initialise a NULL */
```

Le type `struct security_operations` est défini dans le fichier `linux/include/linux/security.h`: Code Linux 2.6.10

```
92  /**
93   * struct security_operations - structure principale pour la securite
94   *
[...]
650  * Points d'ancrage de securite pour le protocole reseau Unix.
651  *
652  * @unix_stream_connect :
653  *      Verifie les droits d'acces avant d'etablir une connexion de flux du domaine Unix
654  *      entre @sock et @other.
655  *      @sock contient la structure de socket.
656  *      @other contient la structure de socket paire.
657  *      Renvoie 0 si le droit d'acces est accorde.
658  * @unix_may_send :
659  *      Verifie les droits d'acces avant une connexion ou l'envoi de  datagrammes de @sock
660  *      a @other.
661  *      @sock contient la structure de socket.
662  *      @sock contient la structure de socket paire.
663  *      Renvoie 0 si le droit d'acces est accorde.
664  *
665  * Les points d'ancrage @unix_stream_connect et @unix_may_send sont necessaires puisque
666  * Linux fournit une alternative a l'espace de noms de fichier classique pour les
667  * sockets du domaine Unix. Bien que la liaison et la connexion des sockets dans l'espace des
668  * noms de fichier se fasse a travers les droits d'acces de fichiers typiques (et attrapes par
669  * mknod et les points d'ancrage de droits d'acces dans inode_security_ops), la liaison et la
670  * connexion aux sockets dans l'espace de noms abstrait se fait sans intermediaire aucun.
671  * Un controle suffisant des sockets du domaine Unix dans l'espace des noms abstrait n'est pas
672  * possible en utilisant seulement les points d'ancrage de la couche des sockets, puisque nous
673  * avons besoin de connaitre la socket cible, qui n'est pas consultee jusqu'a ce que nous
       soyons dans le code af_unix.
674  *
675  * Points d'ancrage de securite pour les operations sur les sockets.
676  *
677  * @socket_create :
678  *      Verifie les droits d'acces avant de creer une nouvelle socket.
679  *      @family contient la famille de protocoles requise.
680  *      @type contient le type de communication requis.
681  *      @protocol contient le protocole requis.
682  *      @kern positionne a 1 si c'est une socket noyau.
683  *      Renvoie 0 si le droit d'acces est accorde.
684  * @socket_post_create :
685  *      Ce point d'ancrage permet a un module de mettre a jour ou d'allouer une structure de
686  *      securite par socket. Noter que le champ de securite n'est pas ajoute directement a la
687  *      structure de socket, mais que l'information de securite de la socket est stockee
688  *      dans le inode associe. Typiquement le point d'ancrage d'inode alloc_security
689  *      allouera et attachera une information de securite a
690  *      sock->inode->i_security. Ce point d'ancrage peut etre utilise pour mettre a jour le
691  *      champ sock->inode->i_security avec des informations supplementaires qui ne sont pas
692  *      disponible lorsque l'inode a ete alloue.
693  *      @sock contient la structure de la socket nouvellement creee.
```

```
694  *      @family contient la famille de protocoles requise.
695  *      @type contient le type de communication requis.
696  *      @protocol contient le protocole requis.
697  *      @kern positionne a 1 si c'est une socket noyau.
698  * @socket_bind :
699  *      Verifies les droits d'acces avant que l'operation de liaison de la couche de protcole
700  *      des sockets soit effectuee et que la socket @sock soit liee a l'adresse specifiee dans
701  *      le parametre @address.
702  *      @sock contient la structure de socket.
703  *      @address contient l'adresse a laquelle lier.
704  *      @addrlen contient la longueur de l'adresse.
705  *      Renvoie 0 si le droit d'acces est accorde.
706  * @socket_connect :
707  *      Verifie les droits d'acces avant que l'operation de connexion de la couche protocole
708  *      des sockets essaie de connecter la socket @sock a une adresse distante, @address.
709  *      @sock contient la structure de socket.
710  *      @address contient l'adresse du point distant.
711  *      @addrlen contient la longueur de l'adresse.
712  *      Renvoie 0 si le droit d'acces est accorde.
713  * @socket_listen :
714  *      Verifie les droits d'acces avant l'operation d'ecoute de la couche protocole des
         sockets.
715  *      @sock contient la structure de socket.
716  *      @backlog contient la longueur maximum de la file d'attente de connection.
717  *      Renvoie 0 si le droit d'acces est accorde.
718  * @socket_accept :
719  *      Verifie les droits d'acces avant d'accepter une nouvelle connexion. Noter que la
720  *      nouvelle socket, @newsock, a ete creee et quelques informations copiees dans
721  *      celle-ci, mais l'operation d'acceptation n'a pas encore ete effectuee.
722  *      @sock contient la structure de socket d'ecoute.
723  *      @newsock contient la socket serveur nouvellement creee pour la connexion.
724  *      Renvoie 0 si le droit d'acces est accorde.
725  * @socket_post_accept :
726  *      Ce point d'ancrage permet a un module de securite de copier des informations
727  *      de securite dans l'inode de socket nouvellement cree.
728  *      @sock contient la structure de socket d'ecoute.
729  *      @newsock contient la socket serveur nouvellement creee pour la connexion.
730  * @socket_sendmsg :
731  *      Verifie les droits d'acces avant de transmettre un message a une autre socket.
732  *      @sock contient la structure de socket.
733  *      @msg contient le message a transmettre.
734  *      @size contient la taille du message.
735  *      Renvoie 0 si le droit d'acces est accorde.
736  * @socket_recvmsg :
737  *      Verifie les droits d'acces avant de recevoir un message d'une socket.
738  *      @sock contient la structure de socket.
739  *      @msg contient la structure de message.
740  *      @size contient la taille de la structure de message.
741  *      @flags contient les drapeaux operationnels.
742  *      Renvoie 0 si le droit d'acces est accorde.
743  * @socket_getsockname :
744  *      Verifie les droits d'acces avant que l'adresse locale (nom) de l'objet socket
745  *      @sock soit recuperee.
746  *      @sock contient la structure de socket.
747  *      Renvoie 0 si le droit d'acces est accorde.
748  * @socket_getpeername :
749  *      Verifie les droits d'acces avant que l'adresse distante (nom) de l'objet socket
750  *      @sock soit recuperee.
751  *      @sock contient la structure de socket.
752  *      Renvoie 0 si le droit d'acces est accorde.
753  * @socket_getsockopt :
754  *      Verifie les droits d'acces avant de recuperer les options associees a la socket
```

```
755  *      @sock.
756  *      @sock contient la structure de socket.
757  *      @level contient le niveau de protocole duquel recuperer l'option.
758  *      @optname contient le nom de l'option a recuperer.
759  *      Renvoie 0 si le droit d'acces est accorde.
760  * @socket_setsockopt :
761  *      Verifie les droits d'acces avant de positionner les options associees a une socket
762  *      @sock.
763  *      @sock contient la structure de socket.
764  *      @level contient le niveau de protocole auquel positionner les options.
765  *      @optname contient le nom de l'option a positionner.
766  *      Renvoie 0 si le droit d'acces est accorde.
767  * @socket_shutdown :
768  *      Verifie les droits d'acces avant que tout ou partie d'une connexion a la socket
769  *      @sock soit fermee.
770  *      @sock contient la structure de socket.
771  *      @how contient le drapeau indiquant comment les envois et reception futurs seront
       manipules.
772  *      Renvoie 0 si le droit d'acces est accorde.
773  * @socket_sock_rcv_skb :
774  *      Verifie les droits d'acces sur les paquets reseau entrants. Ce point d'ancrage
775  *      est different des points d'ancrages d'entree IP de Netfilter puisque c'est la premiere
776  *      fois que le sk_buff entrant @skb est associe a une socket particuliere, @sk.
777  *      @sk contient le sock (non socket) associe au sk_buff entrant.
778  *      @skb contient les donnees reseau entrantes.
[...]
1018 * C'est la structure de securite principale.
1019 */
1020 struct security_operations {
[...]
1203 #ifdef CONFIG_SECURITY_NETWORK
1204         int (*unix_stream_connect) (struct socket * sock,
1205                                     struct socket * other, struct sock * newsk);
1206         int (*unix_may_send) (struct socket * sock, struct socket * other);
1207
1208         int (*socket_create) (int family, int type, int protocol, int kern);
1209         void (*socket_post_create) (struct socket * sock, int family,
1210                                     int type, int protocol, int kern);
1211         int (*socket_bind) (struct socket * sock,
1212                             struct sockaddr * address, int addrlen);
1213         int (*socket_connect) (struct socket * sock,
1214                                struct sockaddr * address, int addrlen);
1215         int (*socket_listen) (struct socket * sock, int backlog);
1216         int (*socket_accept) (struct socket * sock, struct socket * newsock);
1217         void (*socket_post_accept) (struct socket * sock,
1218                                     struct socket * newsock);
1219         int (*socket_sendmsg) (struct socket * sock,
1220                                struct msghdr * msg, int size);
1221         int (*socket_recvmsg) (struct socket * sock,
1222                                struct msghdr * msg, int size, int flags);
1223         int (*socket_getsockname) (struct socket * sock);
1224         int (*socket_getpeername) (struct socket * sock);
1225         int (*socket_getsockopt) (struct socket * sock, int level, int optname);
1226         int (*socket_setsockopt) (struct socket * sock, int level, int optname);
1227         int (*socket_shutdown) (struct socket * sock, int how);
1228         int (*socket_sock_rcv_skb) (struct sock * sk, struct sk_buff * skb);
1229         int (*socket_getpeersec) (struct socket *sock, char __user *optval,
                                      int __user *optlen, unsigned len);
1230         int (*sk_alloc_security) (struct sock *sk, int family, int priority);
1231         void (*sk_free_security) (struct sock *sk);
1232 #endif  /* CONFIG_SECURITY_NETWORK */
1233 };
```

28.1.1.5 Association d'un descripteur de fichier à un descripteur de socket

La fonction `sock_map_fd()` permet d'obtenir un nouveau descripteur de fichier (de type socket) et de l'associer au descripteur de socket entré en paramètre. Elle est définie dans le fichier `linux/net/socket.c`:

Code Linux 2.6.10

```
348 /*
349  *      Obtient le premier descripteur de fichier disponible et l'active pour l'utiliser.
350  *
351  *      Cette fonction cree une structure file et l'associe a l'espace des fd
352  *      du processus en cours. En cas de succes elle renvoie de descripteur de fichier
353  *      et la structure file est implicitement stockee dans sock->file.
354  *      Noter qu'un autre thread peut fermer le descripteur de fichier avant que nous
355  *      ne sortions de cette fonction. Nous utilisons le fait que nous ne faisons pas
356  *      reference a la socket immediatement apres l'association. Si un jour nous en avons
357  *      besoin, cette fonction incrementera le compteur de ref. du fichier de 1.
358  *
359  *      En tout cas le fd renvoye PEUT NE PAS ETRE valide !
360  *      Cette condition de course est inevitable
361  *      avec les espaces de fd partages, nous ne pouvons pas le resoudre dans le noyau,
362  *      mais nous prenons encore soin de la coherence interne.
363  */
364
365 int sock_map_fd(struct socket *sock)
366 {
367         int fd;
368         struct qstr this;
369         char name[32];
370
371         /*
372          *      Trouver un descripteur de fichier adequet a renvoyer a l'utilisateur.
373          */
374
375         fd = get_unused_fd();
376         if (fd >= 0) {
377                 struct file *file = get_empty_filp();
378
379                 if (!file) {
380                         put_unused_fd(fd);
381                         fd = -ENFILE;
382                         goto out;
383                 }
384
385                 sprintf(name, "[%lu]", SOCK_INODE(sock)->i_ino);
386                 this.name = name;
387                 this.len = strlen(name);
388                 this.hash = SOCK_INODE(sock)->i_ino;
389
390                 file->f_dentry = d_alloc(sock_mnt->mnt_sb->s_root, &this);
391                 if (!file->f_dentry) {
392                         put_filp(file);
393                         put_unused_fd(fd);
394                         fd = -ENOMEM;
395                         goto out;
396                 }
397                 file->f_dentry->d_op = &sockfs_dentry_operations;
398                 d_add(file->f_dentry, SOCK_INODE(sock));
399                 file->f_vfsmnt = mntget(sock_mnt);
400                 file->f_mapping = file->f_dentry->d_inode->i_mapping;
401
402                 sock->file = file;
403                 file->f_op = SOCK_INODE(sock)->i_fop = &socket_file_ops;
404                 file->f_mode = FMODE_READ | FMODE_WRITE;
```

```
405                 file->f_flags = O_RDWR;
406                 file->f_pos = 0;
407                 fd_install(fd, file);
408         }
409
410 out:
411         return fd;
412 }
```

Autrement dit :

- On déclare un numéro de fichier `fd`, un métanom `this` et une chaîne de caractères `name`.
- On essaie de trouver un numéro de fichier non utilisé. S'il n'y en a pas, on renvoie le code d'erreur fourni par la fonction générale `get_unused_fd()`.
- On essaie de trouver un descripteur de fichier (vide) non utilisé. Si on n'y parvient pas, on libère le numéro de fichier et on renvoie l'opposé du code d'erreur `ENFILE`.
- On instantie `name` avec le nunméro du descripteur de socket.

  La fonction en ligne `SOCK_INODE()` est définie dans le fichier `include/net/sock.h`: Code Linux 2.6.10

  ```
  654 static inline struct inode *SOCK_INODE(struct socket *socket)
  655 {
  656         return &container_of(socket, struct socket_alloc, socket)->vfs_inode;
  657 }
  ```

  Elle renvoie l'adresse de `socket->vfs_inode`.
- On renseigne les champs du métanom `this`.
- On essaie d'allouer une entrée de répertoire au descripteur de fichier. Si on n'y parvient pas, on libère le descripteur de fichier, on libère le numéro de fichier et on renvoie l'opposé du code d'erreur `ENOMEM`.
- On associe comme ensemble d'opérations sur les entrées de répertoire l'ensemble `sockfs_dentry_operations` vu au chapitre 27.
- On place comme descripteur de nœud d'information associé à l'entrée de répertoire initialisée ci-dessus le descripteur de nœud d'information associé au descripteur de socket.
- Le point de montage du descripteur de fichier est `sock_mnt`.
- Le descripteur de fichier associé au descripteur de socket sera celui que l'on vient d'initialiser.
- On initialise quelques champs du descripteur de fichier et on associe celui-ci au numéro de fichier `fd` que l'on renvoie.

28.1.2 Cas de la famille de protocoles IPv4

Nous avons ci-dessus que la fonction `sock_create()` fait appel à une fonction spécifique de création, dépendant de la famille de protocoles, pour renseigner les champs du descripteur de socket. L'instance `inet_family_ops` de `net_proto_family` montre que, dans le cas de IPv4, la fonction spécifique de création de socket s'appelle `inet_create()`.

28.1.2.1 Création d'une socket

La fonction `inet_create()` est définie dans le fichier `linux/net/ipv4/af_inet.c`: Code Linux 2.6.10

```
226 /*
227  *      Cree une socket inet.
228  */
229
```

```
230 static int inet_create(struct socket *sock, int protocol)
231 {
232         struct sock *sk;
233         struct list_head *p;
234         struct inet_protosw *answer;
235         struct inet_opt *inet;
236         struct proto *answer_prot;
237         unsigned char answer_flags;
238         char answer_no_check;
239         int err;
240
241         sock->state = SS_UNCONNECTED;
242
243         /* Cherche le couple type/protocole requis. */
244         answer = NULL;
245         rcu_read_lock();
246         list_for_each_rcu(p, &inetsw[sock->type]) {
247                 answer = list_entry(p, struct inet_protosw, list);
248
249                 /* Verifie la concordance non-sauvage. */
250                 if (protocol == answer->protocol) {
251                         if (protocol != IPPROTO_IP)
252                                 break;
253                 } else {
254                         /* Verification pour les deux cas sauvages. */
255                         if (IPPROTO_IP == protocol) {
256                                 protocol = answer->protocol;
257                                 break;
258                         }
259                         if (IPPROTO_IP == answer->protocol)
260                                 break;
261                 }
262                 answer = NULL;
263         }
264
265         err = -ESOCKTNOSUPPORT;
266         if (!answer)
267                 goto out_rcu_unlock;
268         err = -EPERM;
269         if (answer->capability > 0 && !capable(answer->capability))
270                 goto out_rcu_unlock;
271         err = -EPROTONOSUPPORT;
272         if (!protocol)
273                 goto out_rcu_unlock;
274
275         sock->ops = answer->ops;
276         answer_prot = answer->prot;
277         answer_no_check = answer->no_check;
278         answer_flags = answer->flags;
279         rcu_read_unlock();
280
281         BUG_TRAP(answer_prot->slab != NULL);
282
283         err = -ENOBUFS;
284         sk = sk_alloc(PF_INET, GFP_KERNEL,
285                       answer_prot->slab_obj_size,
286                       answer_prot->slab);
287         if (sk == NULL)
288                 goto out;
289
290         err = 0;
291         sk->sk_prot = answer_prot;
```

```
292         sk->sk_no_check = answer_no_check;
293         if (INET_PROTOSW_REUSE & answer_flags)
294                 sk->sk_reuse = 1;
295
296         inet = inet_sk(sk);
297
298         if (SOCK_RAW == sock->type) {
299                 inet->num = protocol;
300                 if (IPPROTO_RAW == protocol)
301                         inet->hdrincl = 1;
302         }
303
304         if (ipv4_config.no_pmtu_disc)
305                 inet->pmtudisc = IP_PMTUDISC_DONT;
306         else
307                 inet->pmtudisc = IP_PMTUDISC_WANT;
308
309         inet->id = 0;
310
311         sock_init_data(sock, sk);
312         sk_set_owner(sk, sk->sk_prot->owner);
313
314         sk->sk_destruct    = inet_sock_destruct;
315         sk->sk_family      = PF_INET;
316         sk->sk_protocol    = protocol;
317         sk->sk_backlog_rcv = sk->sk_prot->backlog_rcv;
318
319         inet->uc_ttl    = -1;
320         inet->mc_loop   = 1;
321         inet->mc_ttl    = 1;
322         inet->mc_index  = 0;
323         inet->mc_list   = NULL;
324
325 #ifdef INET_REFCNT_DEBUG
326         atomic_inc(&inet_sock_nr);
327 #endif
328
329         if (inet->num) {
330                 /* On suppose que tout protocole qui permet a
331                  * l'utilisateur d'assigner un numero a une socket
332                  * partage automatiquement le temps de la
333                  * creation.
334                  */
335                 inet->sport = htons(inet->num);
336                 /* Ajoute aux chaines de hachage du protocole. */
337                 sk->sk_prot->hash(sk);
338         }
339
340         if (sk->sk_prot->init) {
341                 err = sk->sk_prot->init(sk);
342                 if (err)
343                         sk_common_release(sk);
344         }
345 out:
346         return err;
347 out_rcu_unlock:
348         rcu_read_unlock();
349         goto out;
350 }
```

Autrement dit :

– On déclare un descripteur de couche transport `sk`, une tête de liste, un protocole de com-

munication réponse, une option Internet, un protocole réponse, des drapeaux, une réponse de non vérification et un code de retour.

– On initialise l'état du descripteur de socket à non connecté.
– On essaie d'instantier le descripteur de type de communication en vérifiant que le couple famille de protocoles – type de communication est valide (lignes 243–263). Si on n'y parvient pas, on renvoie l'opposé du code d'erreur `ESOCKNOTSUPPORT`.

Code Linux 2.6.10

La macro générale `list_for_each_rcu()` est définie dans le fichier `linux/include/linux/list.h`:

```
427 /**
428  * list_for_each_rcu    -       itere sur une liste rcu-protegee
429  * @pos :       la &struct list_head a utiliser comme compteur de boucle.
430  * @head :      la tete de votre liste.
431  *
432  * Cette primitive de parcours de liste peut de facon sure s'effectuer en meme temps
433  * que des primitives de mutation de listes _rcu telles que list_add_rcu()
434  * tant que le parcours est protege par rcu_read_lock().
435  */
436 #define list_for_each_rcu(pos, head) \
437         for (pos = (head)->next, prefetch(pos->next); pos != (head); \
438                 pos = rcu_dereference(pos->next), prefetch(pos->next))
```

Code Linux 2.6.10

La macro générale `list_entry()` est définie dans le même fichier général:

```
314 /**
315  * list_entry - recupere la struct pour cette entree
316  * @ptr :       le pointeur &struct list_head.
317  * @type :      le type de la struct dans laquelle ceci est immerge.
318  * @member :    le nom de la list_struct dans la struct.
319  */
320 #define list_entry(ptr, type, member) \
321         container_of(ptr, type, member)
```

– S'il y a un problème de capacité, on renvoie l'opposé du code d'erreur `EPERM`.
– S'il y a un problème avec la famille de protocoles, on renvoie l'opposé du code d'erreur `EPROTONOTSUPPORT`.
– On renseigne le champ ensemble des opérations du descripteur de socket.
– On essaie d'instantier le descripteur de couche transport. Si on n'y parvient pas, on renvoie l'opposé du code d'erreur `ENOBUFS`.
– On renseigne les champs protocole, pas de vérification et réutilisation du descripteur de couche transport.
– On instantie l'option Internet.
– On renseigne certains champs de cette option Internet : le numéro de port local et le fait que les en-têtes sont incluses si le type de la couche transport est le type brut ; le champ `pmtudisc` et le compteur d'identification.

Code Linux 2.6.10

La structure `ipv4_config` est définie dans le fichier `linux/include/net/ip.h`:

```
145 struct ipv4_config
146 {
147         int     log_martians;
148         int     autoconfig;
149         int     no_pmtu_disc;
150 };
```

Code Linux 2.6.10

La variable `ipv4_config` est définie dans le fichier `linux/net/ipv4/sysctl_net_ipv4.c`:

```
57 struct ipv4_config ipv4_config;
```

Les constantes symboliques IP_PMTUDISC_DONT et IP_PMTUDISC_WANT sont définies dans le fichier linux/include/linux/ip.h: Code Linux 2.6.10

```
78 /* Valeurs de IP_MTU_DISCOVER */
79 #define IP_PMTUDISC_DONT               0       /* Ne jamais envoyer de trames DF */
80 #define IP_PMTUDISC_WANT               1       /* Indications de route par
                                                     l'utilisateur */
81 #define IP_PMTUDISC_DO                 2       /* Toujours DF           */
```

- On associe les descripteurs de socket et de couche de transport.
- On s'approprie le descripteur de couche transport.

  La fonction en ligne sk_set_owner() est définie dans le fichier linux/include/net/sock.h: Code Linux 2.6.10

```
577 static __inline__ void sk_set_owner(struct sock *sk, struct module *owner)
578 {
579         /*
580          * On ne devrait utiliser sk_set_owner qu'une seule fois, apres la creation de
               struct sock,
581          * faites-le rapidement apres sk_alloc ou apres une fonction qui renvoie une
582          * nouvelle struct sock (et ceci abaisse la chaine d'appel appelee sk_alloc),
583          * par exemple les modules IPv4 et IPv6 partagent tcp_create_openreq_child, donc
584          * si tcp_create_openreq_child appelait sk_set_owner IPv6 devrait changer
585          * le proprietaire de cette struct sock, le remplacant par un qui n'est pas
586          * necessairement un appel sk_set_owner ephemere.
587          */
588         BUG_ON(sk->sk_owner != NULL);
589
590         sk->sk_owner = owner;
591         __module_get(owner);
592 }
```

- On renseigne des champs du descripteur de couche transport et de l'option Internet.
- Si le numéro de port local de l'option Internet est non nul, on le place comme numéro de port source de celle-ci et on ajoute le descripteur de couche de transport aux chaînes de hachage du protocole.
- Si une fonction d'initialisation est associée au descripteur de couche transport, on fait appel à celle-ci. Si tout ne se passe pas correctement, on libère le descripteur de couche de transport, grâce à la fonction sk_common_release() étudiée ci-dessous. Sinon on renvoie 0.

Les actions essentielles consistent donc à spécifier les deux ensembles d'opérations et à associer un descripteur de couche transport au descripteur général de socket.

28.1.2.2 Libération d'un descripteur de couche de transport

La fonction sk_common_release() est définie dans le fichier linux/net/core/sock.c: Code Linux 2.6.10

```
1297 void sk_common_release(struct sock *sk)
1298 {
1299         if (sk->sk_prot->destroy)
1300                 sk->sk_prot->destroy(sk);
1301
1302         /*
1303          * Observation : lorsque sock_common_release est appelee, les processus n'ont
1304          * pas acces a la socket. Mais le reseau l'a encore.
1305          * Etape un, la detacher du reseau :
1306          *
1307          * A. Retirer des tables de hachage.
```

```
1308          */
1309
1310         sk->sk_prot->unhash(sk);
1311
1312         /*
1313          * A ce point, la socket ne peut plus recevoir de nouveaux paquets, mais il est
1314          * possible que des paquets soient en route puisque certains CPU effectuent des
1315          * receptions et ont consulte la table de hachage avant que nous n'en retirions
1316          * la socket. Ils termineront la file d'attente de reception et seront purges
                par le destructeur de la socket.
1317          *
1318          * Ainsi nous avons encore des paquets en suspens sur la file d'attente en
1319          * reception et probablement nos propres paquets attendront dans les files
                d'attente des peripheriques. sock_destroy drainera
1320          * la file d'attente en reception, mais les paquets transmis retarderont la
1321          * destruction de la socket jusqu'a ce que la derniere reference soit liberee.
1322          */
1323
1324         sock_orphan(sk);
1325
1326         xfrm_sk_free_policy(sk);
1327
1328 #ifdef INET_REFCNT_DEBUG
1329         if (atomic_read(&sk->sk_refcnt) != 1)
1330                 printk(KERN_DEBUG "Destruction of the socket %p delayed, c=%d\n",
1331                        sk, atomic_read(&sk->sk_refcnt));
1332 #endif
1333         sock_put(sk);
1334 }
```

Code Linux 2.6.10

La fonction `sock_orphan()` est définie dans le fichier `linux/include/net/sock.h`:

```
903 /* Detache la socket du constexte processus.
904  * Annonce la mort de la socket, la detache de la file d'attente et du noeud d'information.
905  * Noter que le noeud d'information parent tient la compteur de reference sur cette struct
906  * sock, nous ne devons pas la liberer dans cette fonction, puisque le protocole
907  * veut probablemnt des nettoyages supplementaires ou meme continuer
908  * a travailler avec cette socket (TCP).
909  */
910 static inline void sock_orphan(struct sock *sk)
911 {
912         write_lock_bh(&sk->sk_callback_lock);
913         sock_set_flag(sk, SOCK_DEAD);
914         sk->sk_socket = NULL;
915         sk->sk_sleep  = NULL;
916         write_unlock_bh(&sk->sk_callback_lock);
917 }
```

28.2 Semi-arrêt d'une socket

28.2.1 Traitement général

28.2.1.1 Fonction d'appel

Code Linux 2.6.10

La fonction `sys_shutdown()` est définie dans le fichier `linux/net/socket.c`:

```
1663 /*
1664  *      Ferme une socket.
1665  */
1666
1667 asmlinkage long sys_shutdown(int fd, int how)
```

```
1668 {
1669         int err;
1670         struct socket *sock;
1671
1672         if ((sock = sockfd_lookup(fd, &err))!=NULL)
1673         {
1674                 err = security_socket_shutdown(sock, how);
1675                 if (err) {
1676                         sockfd_put(sock);
1677                         return err;
1678                 }
1679
1680                 err=sock->ops->shutdown(sock, how);
1681                 sockfd_put(sock);
1682         }
1683         return err;
1684 }
```

Autrement dit :

- On déclare un code d'erreur et un descripteur de socket.
- On essaie de d'instantier le descripteur de socket avec celui associé au numéro de fichier (de socket) passé en argument. Si on n'y parvient pas, on renvoie l'opposé du code d'erreur fourni par la fonction `sockfd_lookup()`, étudiée ci-dessous.
- On fait appel à la fonction de sécurité `security_socket_shutdown()`, dont nous ne nous occuperons pas ici. Si elle n'accorde pas le droit d'accès, on libère le descripteur de fichier associé à la socket et on renvoie le code d'erreur fourni par cette fonction.
- On fait appel à la fonction d'arrêt spécifique à la famille d'adresses. On libère le descripteur de fichier associé à la socket et on renvoie 0 si tout s'est bien passé ou l'opposé du code d'erreur fourni par cette fonction spécifique sinon.

28.2.1.2 Descripteur de socket associé à un numéro de fichier

La fonction `sockfd_lookup()` permet d'obtenir le descripteur de socket associé à un numéro de fichier (de socket). Elle est définie dans le fichier `linux/net/socket.c`: Code Linux 2.6.10

```
414 /**
415  *      sockfd_lookup   -       passe d'un numero de fichier a son emplacement de socket
416  *      @fd  : numero de fichier
417  *      @err : pointeur sur un renvoi de code d'erreur
418  *
419  *      Le numero de fichier passe en argument est verrouille et la socket a laquelle il
420  *      est lie est renvoyee. Si une erreur apparait, le pointeur err est surcharge
421  *      avec un code errno negatif et NULL est renvoye. La fonction verifie
422  *      a la fois les numeros non valides et le passage d'un numero qui n'est pas une socket.
423  *
424  *      En cas de succes, le pointeur de l'objet socket est renvoye.
425  */
426
427 struct socket *sockfd_lookup(int fd, int *err)
428 {
429         struct file *file;
430         struct inode *inode;
431         struct socket *sock;
432
433         if (!(file = fget(fd)))
434         {
435                 *err = -EBADF;
436                 return NULL;
```

```
437         }
438
439         inode = file->f_dentry->d_inode;
440         if (!inode->i_sock || !(sock = SOCKET_I(inode)))
441         {
442                 *err = -ENOTSOCK;
443                 fput(file);
444                 return NULL;
445         }
446
447         if (sock->file != file) {
448                 printk(KERN_ERR "socki_lookup: socket file changed!\n");
449                 sock->file = file;
450         }
451         return sock;
452 }
```

Autrement dit :

- On déclare un descripteur de fichier, un descripteur de nœud d'information et un descripteur de socket.
- On essaie d'instantier le descripteur de fichier avec celui associé au numéro de fichier passé en argument. Si on n'y parvient pas, on positionne le code de retour à l'opposé du code d'erreur `EBADF` et on renvoie `NULL`.
- On instantie le descripteur de nœud d'information avec celui associé au descripteur de fichier.
- On essaie d'instantier le descripteur de socket avec celui associé à ce descripteur de nœud d'information. Si le descripteur de fichier n'est pas du type socket ou si l'on ne parvient pas à instantier le descripteur de socket, on libère le descripteur de fichier et on positionne le code de retour avec l'opposé du code d'erreur `ENOTSOCK` et on renvoie `NULL`.
- Si le descripteur de fichier associé au descripteur de socket n'est pas le descripteur de fichier instantié ci-dessus, on affiche un message d'erreur noyau et on le modifie.
- On renvoie l'adresse du descripteur de socket.

28.2.2 Cas de IPv4

Nous avons vu au chapitre 27, à propos de `inet_dgram_ops` pour UDP, que la fonction spécifique à IPv4 d'arrêt de socket s'appelle `inet_shutdown()`. Cette fonction est définie dans le fichier `linux/net/ipv4/af_inet.c` :

Code Linux 2.6.10

```
676 int inet_shutdown(struct socket *sock, int how)
677 {
678         struct sock *sk = sock->sk;
679         int err = 0;
680
681         /* On doit vraiment verifier pour s'assurer que
682          * la socket est une socket TCP . (POURQUOI AC...)
683          */
684         how++; /* transforme 0->1 a l'avantage de faire du bit 1 receptions et
685                               1->2 du bit 2 envois.
686                               2->3 */
687         if ((how & ~SHUTDOWN_MASK) || !how)     /* MAXINT->0 */
688                 return -EINVAL;
689
690         lock_sock(sk);
691         if (sock->state == SS_CONNECTING) {
692                 if ((1 << sk->sk_state) &
```

```
693                     (TCPF_SYN_SENT | TCPF_SYN_RECV | TCPF_CLOSE))
694                         sock->state = SS_DISCONNECTING;
695                 else
696                         sock->state = SS_CONNECTED;
697         }
698
699         switch (sk->sk_state) {
700         case TCP_CLOSE:
701                 err = -ENOTCONN;
702                 /* Force a reveiller les autres qui sont a l'ecoute, qui peuvent elire
703                    POLLHUP, meme par exemple pour les sockets UDP non connectees -- RR */
704         default:
705                 sk->sk_shutdown |= how;
706                 if (sk->sk_prot->shutdown)
707                         sk->sk_prot->shutdown(sk, how);
708                 break;
709
710         /* Les deux branches restantes sont des solutions temporaires pour le close()
711          * manquant dans un environnement multithread. Ce n'est _pas_ une bonne idee,
712          * mais nous n'avons pas le choix jusqu'a ce que close() soit repare au niveau VFS.
713          */
714         case TCP_LISTEN:
715                 if (!(how & RCV_SHUTDOWN))
716                         break;
717                 /* Passer a travers */
718         case TCP_SYN_SENT:
719                 err = sk->sk_prot->disconnect(sk, O_NONBLOCK);
720                 sock->state = err ? SS_DISCONNECTING : SS_UNCONNECTED;
721                 break;
722         }
723
724         /* Reveiller tout ce qui est assoupi pour l'election. */
725         sk->sk_state_change(sk);
726         release_sock(sk);
727         return err;
728 }
```

Autrement dit :

- On déclare un descripteur de couche transport, que l'on instantie avec celui associé au descripteur de socket passé en argument.
- Si la valeur de l'argument `how` n'est pas située dans le bon intervalle, on renvoie l'opposé du code d'erreur `EINVAL`.
- On verrouille le descripteur de couche transport.
- On change l'état du descripteur de socket si la socket était en train de se connecter.
- Si la socket était déjà fermée, on renvoie l'opposé du code d'erreur `ENOTCONN`.
- Si la socket avait envoyé un message de synchronisation pour se connecter, on se déconnecte grâce à la fonction spécifique à la couche transport et on change d'état.
- On change l'état du descripteur de couche de transport et on libère celui-ci.

28.2.3 Fonction de déconnexion spécifique à UDP

Nous avons vu que, dans le cas de IPv4/UDP, l'ensemble des opérations au niveau de la couche de transport s'appelle `udp_prot`. Cet ensemble est défini dans le fichier `linux/net/ipv4/udp.c`: Code Linux 2.6.10

```
1353 struct proto udp_prot = {
1354         .name =         "UDP",
1355         .owner =        THIS_MODULE,
```

```
1356         .close =        udp_close,
1357         .connect =      ip4_datagram_connect,
1358         .disconnect =   udp_disconnect,
1359         .ioctl =        udp_ioctl,
1360         .destroy =      udp_destroy_sock,
1361         .setsockopt =   udp_setsockopt,
1362         .getsockopt =   udp_getsockopt,
1363         .sendmsg =      udp_sendmsg,
1364         .recvmsg =      udp_recvmsg,
1365         .sendpage =     udp_sendpage,
1366         .backlog_rcv =  udp_queue_rcv_skb,
1367         .hash =         udp_v4_hash,
1368         .unhash =       udp_v4_unhash,
1369         .get_port =     udp_v4_get_port,
1370         .slab_obj_size = sizeof(struct udp_sock),
1371 };
```

Cette structure `udp_prot` indique que la fonction de déconnexion spécifique à UDP s'appelle `udp_disconnect()` et qu'il n'y a pas de fonction spécifique `shutdown()`.

28.2.3.1 Fonction de déconnexion

Code Linux 2.6.10

La fonction `udp_disconnect()` de déconnexion spécifique à UDP est définie dans le fichier `linux/net/ipv4/udp.c`:

```
865 int udp_disconnect(struct sock *sk, int flags)
866 {
867         struct inet_opt *inet = inet_sk(sk);
868         /*
869          *      1003.1g - briser l'association.
870          */
871
872         sk->sk_state = TCP_CLOSE;
873         inet->daddr = 0;
874         inet->dport = 0;
875         sk->sk_bound_dev_if = 0;
876         if (!(sk->sk_userlocks & SOCK_BINDADDR_LOCK))
877                 inet_reset_saddr(sk);
878
879         if (!(sk->sk_userlocks & SOCK_BINDPORT_LOCK)) {
880                 sk->sk_prot->unhash(sk);
881                 inet->sport = 0;
882         }
883         sk_dst_reset(sk);
884         return 0;
885 }
```

Autrement dit:

- On déclare une option Internet, que l'on instantie avec celle associée au descripteur de couche de transport passé en argument.
- On indique, grâce à son descripteur de couche de transport, que l'état de la socket est fermé.
- On positionne l'adresse et le numéro de port de destination de l'option Internet à zéro.
- On positionne le numéro d'interface physique du descripteur de couche de transport à zéro.
- Si on n'est pas verrouillé en utilisation, on met à zéro l'adresse et le port source de l'option Internet.

Les drapeaux verrou SOCK_BINDADDR_LOCK et SOCK_BINDPORT_LOCK sont définis dans le fichier linux/include/net/sock.h: Code Linux 2.6.10

```
615 #define SOCK_SNDBUF_LOCK        1
616 #define SOCK_RCVBUF_LOCK        2
617 #define SOCK_BINDADDR_LOCK      4
618 #define SOCK_BINDPORT_LOCK      8
```

La fonction en ligne inet_reset_saddr() est définie dans le fichier linux/include/net/ip.h: Code Linux 2.6.10

```
236 static __inline__ void inet_reset_saddr(struct sock *sk)
237 {
238         inet_sk(sk)->rcv_saddr = inet_sk(sk)->saddr = 0;
239 #if defined(CONFIG_IPV6) || defined(CONFIG_IPV6_MODULE)
240         if (sk->sk_family == PF_INET6) {
241                 struct ipv6_pinfo *np = inet6_sk(sk);
242
243                 memset(&np->saddr, 0, sizeof(np->saddr));
244                 memset(&np->rcv_saddr, 0, sizeof(np->rcv_saddr));
245         }
246 #endif
247 }
```

La fonction unhash() spécifique à UDP s'appelle udp_v4_unhash() comme le montre udp_prot. Cette fonction est définie dans le fichier linux/net/ipv4/udp.c: Code Linux 2.6.10

```
209 static void udp_v4_unhash(struct sock *sk)
210 {
211         write_lock_bh(&udp_hash_lock);
212         if (sk_del_node_init(sk)) {
213                 inet_sk(sk)->num = 0;
214                 sock_prot_dec_use(sk->sk_prot);
215         }
216         write_unlock_bh(&udp_hash_lock);
217 }
```

– On réinitialise le cache de destination, grâce à la fonction sk_dst_reset() étudiée ci-dessous, et on renvoie 0.

28.2.3.2 Réinitialisation de l'entrée de cache de destination

La fonction sk_dst_reset() est définie dans le fichier linux/include/net/sock.h: Code Linux 2.6.10

```
968 static inline void
969 __sk_dst_reset(struct sock *sk)
970 {
971         struct dst_entry *old_dst;
972
973         old_dst = sk->sk_dst_cache;
974         sk->sk_dst_cache = NULL;
975         dst_release(old_dst);
976 }
977
978 static inline void
979 sk_dst_reset(struct sock *sk)
980 {
981         write_lock(&sk->sk_dst_lock);
982         __sk_dst_reset(sk);
983         write_unlock(&sk->sk_dst_lock);
984 }
```

Autrement dit :

- on verrouille le champ entrée de cache de destination du descripteur de couche de transport en écriture, pour que personne d'autre ne puisse y accéder ;
- on fait appel à la fonction `__sk_dst_reset()` ; celle-ci met à `NULL` le champ entrée de cache de destination et libère l'ancienne entrée de cache ;
- on déverrouille le champ entrée de cache en écriture, pour qu'on puisse y accéder à nouveau.

28.3 Fermeture d'une socket

Code Linux 2.6.10

La fonction d'appel générale `sys_close()` est définie dans le fichier `linux/fs/open.c` :

```
1018 /*
1019  * Soyez soigneux ici ! Nous testons si le pointeur de fichier est NULL avant
1020  * de liberer le fd. Ceci nous assure qu'une tache clone ne peut pas liberer
1021  * un fd pendant qu'un autre clone est en train de l'ouvrir.
1022  */
1023 asmlinkage long sys_close(unsigned int fd)
1024 {
1025         struct file * filp;
1026         struct files_struct *files = current->files;
1027
1028         spin_lock(&files->file_lock);
1029         if (fd >= files->max_fds)
1030                 goto out_unlock;
1031         filp = files->fd[fd];
1032         if (!filp)
1033                 goto out_unlock;
1034         files->fd[fd] = NULL;
1035         FD_CLR(fd, files->close_on_exec);
1036         __put_unused_fd(files, fd);
1037         spin_unlock(&files->file_lock);
1038         return filp_close(filp, files);
1039
1040 out_unlock:
1041         spin_unlock(&files->file_lock);
1042         return -EBADF;
1043 }
```

Autrement dit :

- On déclare une adresse de descripteur de fichier.
- On déclare une table des fichiers ouverts, que l'on instantie avec celle associée au processus en cours.

  Le type général `struct files_struct` est défini dans le fichier `linux/include/linux/file.h` :

Code Linux 2.6.10

```
19 /*
20  * Structure table des fichiers ouverts
21  */
22 struct files_struct {
23         atomic_t count;
24         spinlock_t file_lock;     /* Protege tous les membres ci-dessous. Se niche
                                       dans tsk->alloc_lock */
25         int max_fds;
26         int max_fdset;
27         int next_fd;
28         struct file ** fd;        /* tableau des fd en cours */
29         fd_set *close_on_exec;
```

```
30         fd_set *open_fds;
31         fd_set close_on_exec_init;
32         fd_set open_fds_init;
33         struct file * fd_array[NR_OPEN_DEFAULT];
34 };
```

- On verrouille la table des fichiers ouverts.
- Si le numéro de fichier passé en argument est supérieur au nombre maximum de fichiers ouverts, on déverrouille la table et on renvoie l'opposé du code d'erreur `EBADF`.
- On essaie d'instantier le descripteur de fichier avec celui associé au numéro de fichier passé en argument. Si on n'y parvient pas, on déverrouille la table et on renvoie l'opposé du code d'erreur `EBADF`.
- On indique dans la table que le descripteur de fichier associé au numéro de fichier est nul.
- On indique qu'on n'aura plus à fermer le fichier associé au numéro à la fin de l'exécution du processus en cours.
- On déverrouille la table des fichiers ouverts.
- On fait appel à la fonction `filp_close()` et on renvoie le code d'erreur fourni.

Cette fonction générale est définie un peu plus haut dans le même fichier : Code Linux 2.6.10

```
986 /*
987  * "id" est l'ID du thread POSIX. Nous utilisons le
988  * pointeur de fichiers a la place...
989  */
990 int filp_close(struct file *filp, fl_owner_t id)
991 {
992         int retval;
993
994         /* Reporter et effacer les erreurs exceptionnelles */
995         retval = filp->f_error;
996         if (retval)
997                 filp->f_error = 0;
998
999         if (!file_count(filp)) {
1000                printk(KERN_ERR "VFS: Close: file count is 0\n");
1001                return retval;
1002        }
1003
1004        if (filp->f_op && filp->f_op->flush) {
1005                int err = filp->f_op->flush(filp);
1006                if (!retval)
1007                        retval = err;
1008        }
1009
1010        dnotify_flush(filp, id);
1011        locks_remove_posix(filp, id);
1012        fput(filp);
1013        return retval;
1014 }
```

Remarquons que rien de spécifique aux sockets n'intervient dans le cas de la fermeture.

Chapitre 29

Initialisation d'un serveur

Nous avons vu au chapitre 6 que l'initialisation d'un serveur s'effectue grâce à l'appel système :

```
bind(socket, adresse, taille);
```

Nous allons étudier l'implémentation de cet appel système dans ce chapitre, dans sa partie générale, dans le cas de la couche réseau IPv4 et de la couche de transport UDP.

29.1 Partie générale

29.1.1 Fonction d'appel

Code Linux 2.6.10

La fonction d'appel `sys_bind()` est définie dans le fichier `linux/net/socket.c`:

```
1275 /*
1276  *      Lie un nom a une socket. Rien de plus a faire ici puisque c'est de
1277  *      la responsabilite du protocole de traiter l'adresse locale.
1278  *
1279  *      Nous deplacons l'adresse de la socket depuis l'espace noyau avant d'appeler
1280  *      la couche de protocole (ayant egalement verifie que l'adresse est ok).
1281  */
1282
1283 asmlinkage long sys_bind(int fd, struct sockaddr __user *umyaddr, int addrlen)
1284 {
1285         struct socket *sock;
1286         char address[MAX_SOCK_ADDR];
1287         int err;
1288
1289         if((sock = sockfd_lookup(fd,&err))!=NULL)
1290         {
1291                 if((err=move_addr_to_kernel(umyaddr,addrlen,address))>=0) {
1292                         err = security_socket_bind(sock, (struct sockaddr *)address,
                                                           addrlen);
1293                         if (err) {
1294                                 sockfd_put(sock);
1295                                 return err;
1296                         }
1297                         err = sock->ops->bind(sock, (struct sockaddr *)address, addrlen);
1298                 }
1299                 sockfd_put(sock);
1300         }
1301         return err;
1302 }
```

Autrement dit :

- On déclare un descripteur de socket, une adresse et une variable pour la valeur renvoyée.
- On essaie d'instantier le descripteur de socket avec celui associé au numéro de fichier (de socket) passé en argument. Si on n'y parvient pas, on renvoie l'opposé du code d'erreur fourni par la fonction `sockfd_lookup()`.
- L'adresse du serveur se trouvant dans l'espace utilisateur, on essaie de la rapatrier dans l'espace noyau, grâce à la fonction `move_addr_to_kernel()` étudiée ci-après. Si on n'y parvient pas, on libère le descripteur de socket et on renvoie l'opposé du code d'erreur fourni par cette dernière fonction.
- On fait appel à la fonction de sécurité `security_socket_bind()`, dont nous ne nous occuperons pas ici. Si elle n'accorde pas le droit d'accès, on libère le descripteur de fichier associé à la socket et on renvoie le code d'erreur fourni par cette fonction.
- On fait appel à la fonction spécifique associée à la famille de protocoles, on libère le descripteur de fichier associé au descripteur de socket et on renvoie le code fourni par la fonction spécifique.

29.1.2 Passage espace utilisateur/espace noyau

Code Linux 2.6.10

La fonction `move_addr_to_kernel()` est définie dans le fichier `linux/net/socket.c`:

```
210 /**
```

```
211  *      move_addr_to_kernel     -       copie une adresse de socket dans l'espace noyau
212  *      @uaddr : Adresse dans l'espace utilisateur
213  *      @kaddr : Adresse dans l'espace noyau
214  *      @ulen  : Longueur dans l'espace utilisateur
215  *
216  *      L'adresse est copiee dans l'espace noyau. Si l'adresse fournie est
217  *      trop longue, un code d'erreur de -EINVAL est renvoye. Si la copie donne
218  *      des adresses non valides, -EFAULT est renvoye. En cas de succes, 0 est renvoye.
219  */
220
221 int move_addr_to_kernel(void __user *uaddr, int ulen, void *kaddr)
222 {
223         if(ulen<0||ulen>MAX_SOCK_ADDR)
224                 return -EINVAL;
225         if(ulen==0)
226                 return 0;
227         if(copy_from_user(kaddr,uaddr,ulen))
228                 return -EFAULT;
229         return 0;
230 }
```

Le code se comprend aisément.

29.2 Cas de IPv4

29.2.1 Fonction spécifique d'initialisation du serveur

Nous avons vu, à propos de `inet_dgram_ops` pour UDP, que la fonction d'initialisation du serveur spécifique à IPv4 s'appelle `inet_bind()`. Cette fonction est définie dans le fichier `linux/net/ipv4/af_inet.c`:

Code Linux 2.6.10

```
385 /* Elle est desactivee par defaut, voir ci-dessous. */
386 int sysctl_ip_nonlocal_bind;
387
388 int inet_bind(struct socket *sock, struct sockaddr *uaddr, int addr_len)
389 {
390         struct sockaddr_in *addr = (struct sockaddr_in *)uaddr;
391         struct sock *sk = sock->sk;
392         struct inet_opt *inet = inet_sk(sk);
393         unsigned short snum;
394         int chk_addr_ret;
395         int err;
396
397         /* Si la socket a sa propre fonction bind, utilisons-la. (RAW) */
398         if (sk->sk_prot->bind) {
399                 err = sk->sk_prot->bind(sk, uaddr, addr_len);
400                 goto out;
401         }
402         err = -EINVAL;
403         if (addr_len < sizeof(struct sockaddr_in))
404                 goto out;
405
406         chk_addr_ret = inet_addr_type(addr->sin_addr.s_addr);
407
408         /* Specifie par aucun standard en soi, cependant elle brise beaucoup trop
409          * d'applications lorsque supprime. C'est dommage puisque
410          * permettre aux applications d'effectuer une liaison non locale resout
411          * certains problemes avec les systemes utilisant un adressage dynamique.
412          * (c'est-a-dire que vos serveurs demarreront meme si votre lien ISDN
413          *  est temporairement inactif)
414          */
```

```
415         err = -EADDRNOTAVAIL;
416         if (!sysctl_ip_nonlocal_bind &&
417             !inet->freebind &&
418             addr->sin_addr.s_addr != INADDR_ANY &&
419             chk_addr_ret != RTN_LOCAL &&
420             chk_addr_ret != RTN_MULTICAST &&
421             chk_addr_ret != RTN_BROADCAST)
422                 goto out;
423
424         snum = ntohs(addr->sin_port);
425         err = -EACCES;
426         if (snum && snum < PROT_SOCK && !capable(CAP_NET_BIND_SERVICE))
427                 goto out;
428
429         /*      Nous gardons une paire d'adresses. rcv_saddr est celle utilisee par
430          *      les consultations de hachage et saddr est utilisee pour la transmission.
431          *
432          *      Dans l'API BSD, ce sont les memes sauf lorsqu'il
433          *      serait illegal de les utiliser (multicast/broadcast), auquel
434          *      cas l'adresse du peripherique d'envoi est utilise.
435          */
436         lock_sock(sk);
437
438         /* Verifie ces erreurs (socket active, double liaison). */
439         err = -EINVAL;
440         if (sk->sk_state != TCP_CLOSE || inet->num)
441                 goto out_release_sock;
442
443         inet->rcv_saddr = inet->saddr = addr->sin_addr.s_addr;
444         if (chk_addr_ret == RTN_MULTICAST || chk_addr_ret == RTN_BROADCAST)
445                 inet->saddr = 0;  /* Utiliser le peripherique */
446
447         /* S'assurer que nous avons le droit de se relier ici. */
448         if (sk->sk_prot->get_port(sk, snum)) {
449                 inet->saddr = inet->rcv_saddr = 0;
450                 err = -EADDRINUSE;
451                 goto out_release_sock;
452         }
453
454         if (inet->rcv_saddr)
455                 sk->sk_userlocks |= SOCK_BINDADDR_LOCK;
456         if (snum)
457                 sk->sk_userlocks |= SOCK_BINDPORT_LOCK;
458         inet->sport = htons(inet->num);
459         inet->daddr = 0;
460         inet->dport = 0;
461         sk_dst_reset(sk);
462         err = 0;
463 out_release_sock:
464         release_sock(sk);
465 out:
466         return err;
467 }
```

Autrement dit :

- On déclare une adresse Internet, que l'on instantie avec celle passée en argument.
- On déclare un descripteur de couche transport, que l'on instantie avec celui associé au descripteur de socket passé en argument.
- On déclare une option Internet, que l'on instantie avec celle associée au descripteur de transport.

- Si la socket possède sa propre fonction d'initialisation (ce qui est le cas du mode brut mais pas de UDP, comme le montre `udp_prot`, ni de TCP, comme le montre `tcp_prot`), on fait appel à celle-ci, on libère le descripteur de couche transport et on renvoie le code fourni par cette fonction.
- Si la longueur de l'adresse, telle qu'entrée en argument, est inférieure à la longueur d'une adresse standard d'Internet, on renvoie l'opposé du code d'erreur `EINVAL`.
- On détermine le type d'adresse grâce à la fonction auxiliaire `inet_addr_type()`, qui sera étudiée ci-dessous. La valeur de retour est l'une des trois constantes symboliques `RTN_UNICAST`, `RTN_BROADCAST`, `RTN_MULTICAST` ou `RTN_LOCAL`.
- Dans un certain nombre de cas choisis par Linux, on renvoie l'opposé du code d'erreur `EADDRNOTAVAIL`.
- On détermine le numéro de port source `snum` à partir de l'adresse spécifiée.
- Si le numéro de port spécifié ne convient pas, par exemple s'il est inférieur à 1 024 alors que le processus n'appartient pas au super-utilisateur, on renvoie l'opposé du code d'erreur `EACCES`.

  La constante `PROT_SOCK` est définie dans le fichier en-tête `linux/include/net/sock.h`: Code Linux 2.6.10

```
608 /* Les sockets 0-1023 ne peuvent pas etre liees a moins que vous ne soyez le
       super-utilisateur */
609 #define PROT_SOCK       1024
```

- On verrouille le descripteur de couche transport de façon à ce que personne d'autre n'intervienne sur lui.
- Si la socket est déjà active ou si l'on essaie d'y connecter un second serveur, on libère le descripteur de couche transport et on renvoie l'opposé du code d'erreur `EINVAL`.
- On initialise l'adresse de réception de l'option internet associée au descripteur de couche transport, ce qui est le but principal de cette fonction.
- Dans le cas d'une multidiffusion ou de la diffusion générale, on positionne cette adresse à zéro, ce qui permet d'indiquer au périphérique réseau que l'on veut un tel type de diffusion.
- On vérifie que l'on peut utiliser le numéro de port spécifié. Si ce n'est pas le cas, on remet à zéro les adresses précédentes, on libère le descripteur de couche transport et on renvoie l'opposé du code d'erreur `EADDRINUSE`. On utilise pour cela la fonction `get_port()` spécifique au protocole de transport.
- On renseigne encore quelques champs du descripteur de couche transport et de l'option internet, on libère le descripteur et on renvoie 0.

29.2.2 Détermination du type d'adresse

La fonction `inet_addr_type()` est définie dans le fichier `linux/net/ipv4/fib_frontend.c`: Code Linux 2.6.10

```
127 unsigned inet_addr_type(u32 addr)
128 {
129         struct flowi            fl = { .nl_u = { .ip4_u = { .daddr = addr } } };
130         struct fib_result       res;
131         unsigned ret = RTN_BROADCAST;
132
133         if (ZERONET(addr) || BADCLASS(addr))
134                 return RTN_BROADCAST;
135         if (MULTICAST(addr))
136                 return RTN_MULTICAST;
137
138 #ifdef CONFIG_IP_MULTIPLE_TABLES
139         res.r = NULL;
```

```
140 #endif
141
142         if (ip_fib_local_table) {
143                 ret = RTN_UNICAST;
144                 if (!ip_fib_local_table->tb_lookup(ip_fib_local_table,
145                                                    &fl, &res)) {
146                         ret = res.type;
147                         fib_res_put(&res);
148                 }
149         }
150         return ret;
151 }
```

Autrement dit :

- On déclare un flux Internet, que l'on instantie.
- On déclare un résultat de consultation de table de routage et une valeur de retour.
- Si l'adresse passée en argument est nulle ou correspond à une mauvaise classe d'adresses, on renvoie `RTN_BROADCAST`.
- S'il s'agit d'une adresse de multidiffusion, on renvoie `RTN_MULTICAST`.
- Si la table de routage locale n'est pas définie, on renvoie `RTN_BROADCAST`.
- Si la table de routage locale est définie :
  - On la consulte. Si on obtient un résultat, on place la réponse et on renvoie le type fourni par la fonction de consultation.

    La fonction en ligne `fib_res_put()` est définie dans le fichier en-tête `linux/include/net/ip_fib.h` :

Code Linux 2.6.10

```
260 static inline void fib_res_put(struct fib_result *res)
261 {
262         if (res->fi)
263                 fib_info_put(res->fi);
264 #ifdef CONFIG_IP_MULTIPLE_TABLES
265         if (res->r)
266                 fib_rule_put(res->r);
267 #endif
268 }
```

  - Sinon on renvoie la valeur `RTN_UNICAST`.

29.3 Obtention et vérification du port dans le cas UDP

Rappelons que le numéro de port en cas d'émission par un client est attribué dynamiquement pour savoir à quel processus remettre le paquet répondant à un paquet envoyé. Rappelons également que les numéros 0 à 1 023 sont réservés aux services connus.

Le tableau `sysctl_local_port_range[]` permet de savoir dans quel intervalle le système peut attribuer ces numéros. Il est défini dans le fichier `linux/net/ipv4/tcp_ipv4.c` :

Code Linux 2.6.10

```
99  /*
100  * Ce tableau contient le premier et le dernier numeros de port locaux.
101  * Pour des systemes a tres grand usage, changer ce sysctl par
102  * 32768-61000
103  */
104 int sysctl_local_port_range[2] = { 1024, 4999 };
```

La fonction `get_port()` d'obtention et de vérification du numéro de port spécifique à IPv4/-UDP s'appelle `udp_v4_get_port()`, comme le montre `udp_prot`. Elle est définie dans le fichier `linux/net/ipv4/udp.c`:

Code Linux 2.6.10

```
120 /* Partage par udp v4/v6. */
121 int udp_port_rover;
122
123 static int udp_v4_get_port(struct sock *sk, unsigned short snum)
124 {
125         struct hlist_node *node;
126         struct sock *sk2;
127         struct inet_opt *inet = inet_sk(sk);
128
129         write_lock_bh(&udp_hash_lock);
130         if (snum == 0) {
131                 int best_size_so_far, best, result, i;
132
133                 if (udp_port_rover > sysctl_local_port_range[1] ||
134                     udp_port_rover < sysctl_local_port_range[0])
135                         udp_port_rover = sysctl_local_port_range[0];
136                 best_size_so_far = 32767;
137                 best = result = udp_port_rover;
138                 for (i = 0; i < UDP_HTABLE_SIZE; i++, result++) {
139                         struct hlist_head *list;
140                         int size;
141
142                         list = &udp_hash[result & (UDP_HTABLE_SIZE - 1)];
143                         if (hlist_empty(list)) {
144                                 if (result > sysctl_local_port_range[1])
145                                         result = sysctl_local_port_range[0] +
146                                                 ((result - sysctl_local_port_range[0]) &
147                                                  (UDP_HTABLE_SIZE - 1));
148                                 goto gotit;
149                         }
150                         size = 0;
151                         sk_for_each(sk2, node, list)
152                                 if (++size >= best_size_so_far)
153                                         goto next;
154                         best_size_so_far = size;
155                         best = result;
156                 next:;
157                 }
158                 result = best;
159                 for(i = 0; i < (1 << 16) / UDP_HTABLE_SIZE; i++, result += UDP_HTABLE_SIZE) {
160                         if (result > sysctl_local_port_range[1])
161                                 result = sysctl_local_port_range[0]
162                                         + ((result - sysctl_local_port_range[0]) &
163                                            (UDP_HTABLE_SIZE - 1));
164                         if (!udp_lport_inuse(result))
165                                 break;
166                 }
167                 if (i >= (1 << 16) / UDP_HTABLE_SIZE)
168                         goto fail;
169 gotit:
170                 udp_port_rover = snum = result;
171         } else {
172                 sk_for_each(sk2, node,
173                             &udp_hash[snum & (UDP_HTABLE_SIZE - 1)]) {
174                         struct inet_opt *inet2 = inet_sk(sk2);
175
176                         if (inet2->num == snum &&
177                             sk2 != sk &&
178                             !ipv6_only_sock(sk2) &&
```

```
179                             (!sk2->sk_bound_dev_if ||
180                              !sk->sk_bound_dev_if ||
181                              sk2->sk_bound_dev_if == sk->sk_bound_dev_if) &&
182                             (!inet2->rcv_saddr ||
183                              !inet->rcv_saddr ||
184                              inet2->rcv_saddr == inet->rcv_saddr) &&
185                             (!sk2->sk_reuse || !sk->sk_reuse))
186                                 goto fail;
187                 }
188         }
189         inet->num = snum;
190         if (sk_unhashed(sk)) {
191                 struct hlist_head *h = &udp_hash[snum & (UDP_HTABLE_SIZE - 1)];
192
193                 sk_add_node(sk, h);
194                 sock_prot_inc_use(sk->sk_prot);
195         }
196         write_unlock_bh(&udp_hash_lock);
197         return 0;
198
199 fail:
200         write_unlock_bh(&udp_hash_lock);
201         return 1;
202 }
```

Autrement dit :

- La variable globale `udp_port_rover`, définie à la ligne 121, permet de savoir où on en est dans la distribution des numéros de port locaux.
- On déclare une liste de descripteurs de nœuds d'information.
- On déclare un descripteur de couche transport distant `sk2`.
- On déclare une option internet, que l'on instantie avec celle associée au descripteur de couche transport local (celui passé en argument).
- On verrouille en écriture.
- Si le numéro de port source passé en argument est nul, c'est qu'il faut attribuer un numéro de port dynamiquement. Dans ce cas :
  - Si la variable `udp_port_rover` est sortie de l'intervalle dans lequel elle doit prendre ses valeurs, on lui donne comme valeur l'origine de cet intervalle.
  - On donne provisoirement comme numéro de port la valeur de cette variable, puis on utilise la procédure de hachage pour lui donner sa valeur définitive à partir de celle-ci, que l'on attribue également comme nouvelle valeur à `udp_port_rover`.

    La fonction en ligne `udp_lport_inuse()` permet de savoir si un numéro de port est déjà utilisé. Elle est définie dans le fichier `linux/include/net/udp.h` :

Code Linux 2.6.10

```
43 static inline int udp_lport_inuse(u16 num)
44 {
45         struct sock *sk;
46         struct hlist_node *node;
47
48         sk_for_each(sk, node, &udp_hash[num & (UDP_HTABLE_SIZE - 1)])
49                 if (inet_sk(sk)->num == num)
50                         return 1;
51         return 0;
52 }
```

- Sinon on essaie de lui attribuer le numéro passé en argument après vérification. Si on n'y parvient pas, on renvoie 1.

– Dans les deux cas, on indique ce numéro de port dans l'option Internet.

La fonction en ligne `sk_unhashed()` est définie dans le fichier `linux/include/net/sock.h`:

Code Linux 2.6.10

```
288 static inline int sk_unhashed(struct sock *sk)
289 {
290         return hlist_unhashed(&sk->sk_node);
291 }
```

La fonction en ligne `sk_add_node()` est définie dans le fichier `linux/include/net/sock.h`:

Code Linux 2.6.10

```
349 static __inline__ void __sk_add_node(struct sock *sk, struct hlist_head *list)
350 {
351         hlist_add_head(&sk->sk_node, list);
352 }
353
354 static __inline__ void sk_add_node(struct sock *sk, struct hlist_head *list)
355 {
356         sock_hold(sk);
357         __sk_add_node(sk, list);
358 }
```

La fonction en ligne `sock_prot_inc_use()` est définie dans le fichier `linux/include/net/sock.h`:

Code Linux 2.6.10

```
594 /* Appelee avec les bh locales inhibees */
595 static __inline__ void sock_prot_inc_use(struct proto *prot)
596 {
597         prot->stats[smp_processor_id()].inuse++;
598 }
```

– On déverrouille en écriture et on renvoie 0.

Chapitre 30

Réception d'un datagramme par une socket

Nous avons laissé au chapitre 25 le datagramme arrivé sur la carte réseau dans la file d'attente de réception UDP. Nous allons voir dans ce chapitre comment l'application récupère celui-ci.

Rappelons qu'une application récupère des données d'une connexion non connectée grâce à l'appel système `recvfrom()`, comme le montre le tableau vu au chapitre 7 précisant les appels systèmes concernant l'envoi et la réception des données pour une socket :

Écriture	Lecture	Fonctionnalités ajoutées
write()	read()	Fonction de base
send()	recv()	Ajoute un argument de drapeaux
sendto()	recvfrom()	Ajoute une adresse de socket et une longueur
writev()	readv()	Ni socket ni drapeaux, mais un vecteur
sendmsg()	recvmsg()	Drapeaux, socket, vecteur et données ancillaires.

Nous allons commencer par une section sur l'implémentation des types liés aux fonctions d'entrée-sortie, qui interviennent même dans les cas simples.

30.1 Types liés aux fonctions d'entrée-sortie

30.1.1 Vecteurs d'entrée-sortie

Nous avons vu que les fonctions de lecture et d'écriture `readv()` et `writev()` de UNIX ont comme paramètre un vecteur d'entrée-sortie. Le type `struct iovec` des éléments des vecteurs d'entrée-sortie est défini dans le fichier en-tête `linux/include/linux/uio.h`:

Code Linux 2.6.10

```
7 /*
8  *      Structures UIO style Berkeley   -       Alan Cox 1994.
9  *
[...]
17 /* Un mot d'avertissement : notre structure uio est incompatible avec celle de la
      bibliotheque C (qui est maintenant obsolete). Supprimer celle de la bibliotheque
18    C situee dans sys/uio.h si vous avez un tres vieil ensemble de bibliotheques */
19
20 struct iovec
21 {
22         void __user *iov_base;   /* BSD utilise caddr_t (1003.1g necessite void *) */
23         __kernel_size_t iov_len; /* Doit etre size_t (1003.1g) */
24 };
```

30.1.2 En-tête de message

Nous avons vu que les fonctions de lecture et d'écriture `recvmsg()` et `sendmsg()` de UNIX ont comme paramètre un en-tête de message de type `struct msghdr`. Celui-ci est défini dans le fichier en-tête `linux/include/linux/socket.h`:

Code Linux 2.6.10

```
47 /*
48  *      Lorsque nous passons un message 4.4BSD nous utilisons un systeme de passage de
49  *      message 4.4BSD, pas 4.3. Donc msg_accrights(len) manque maintenant. Il
50  *      appartient a une obscure emulation de libc ou a bin.
51  */
52
53 struct msghdr {
54         void    *       msg_name;       /* Nom de la socket             */
55         int             msg_namelen;    /* Longueur du nom              */
56         struct iovec *  msg_iov;        /* Blocs de donnees             */
57         __kernel_size_t msg_iovlen;     /* Nombre de blocs              */
58         void    *       msg_control;    /* Pour la magie du protocole (par exemple le
                                              passage de descripteurs de fichier BSD) */
59         __kernel_size_t msg_controllen; /* Longueur de la liste cmsg     */
60         unsigned        msg_flags;
61 };
```

30.1.3 En-tête de contrôle

Nous venons de voir que les en-têtes de message font référence à des messages de contrôle. Ceux-ci possédent des en-têtes du type `struct cmsghdr`. Celui-ci est défini dans le même fichier en-tête :

Code Linux 2.6.10

```
63 /*
64  *      POSIX 1003.1g - informations sur les objets de donnees ancillaires
65  *      Les donnees ancillaires consistent en une suite de couples
66  *      (cmsghdr, cmsg_data[])
67  */
68
69 struct cmsghdr {
70         __kernel_size_t cmsg_len;       /* compteur d'octets des donnees, y compris hdr */
71         int             cmsg_level;     /* protocole d'origine */
```

```
72        int             cmsg_type;      /* type specifique au protocole */
73 };
```

30.2 Implémentation générale de la réception des datagrammes

30.2.1 Fonction d'appel

La fonction d'appel `sys_recvfrom()` est définie dans le fichier `linux/net/socket.c`: Code Linux 2.6.10

```
1553 /*
1554  *      Recoit une trame d'une socket et optionnellement enregistre l'adresse de
1555  *      l'emetteur. Nous verifions qu'on peut ecrire sur les tampons et, si necessaire,
1556  *      nous deplacons l'adresse de l'emetteur du noyau a l'espace utilisateur.
1557  */
1558
1559 asmlinkage long sys_recvfrom(int fd, void __user * ubuf, size_t size, unsigned flags,
1560                              struct sockaddr __user *addr, int __user *addr_len)
1561 {
1562         struct socket *sock;
1563         struct iovec iov;
1564         struct msghdr msg;
1565         char address[MAX_SOCK_ADDR];
1566         int err,err2;
1567
1568         sock = sockfd_lookup(fd, &err);
1569         if (!sock)
1570                 goto out;
1571
1572         msg.msg_control=NULL;
1573         msg.msg_controllen=0;
1574         msg.msg_iovlen=1;
1575         msg.msg_iov=&iov;
1576         iov.iov_len=size;
1577         iov.iov_base=ubuf;
1578         msg.msg_name=address;
1579         msg.msg_namelen=MAX_SOCK_ADDR;
1580         if (sock->file->f_flags & O_NONBLOCK)
1581                 flags |= MSG_DONTWAIT;
1582         err=sock_recvmsg(sock, &msg, size, flags);
1583
1584         if(err >= 0 && addr != NULL)
1585         {
1586                 err2=move_addr_to_user(address, msg.msg_namelen, addr, addr_len);
1587                 if(err2<0)
1588                         err=err2;
1589         }
1590         sockfd_put(sock);
1591 out:
1592         return err;
1593 }
```

Autrement dit :

- On déclare un descripteur de socket, un élément de vecteur d'entrée-sortie, un en-tête de message, une adresse et deux codes de retour.
- On essaie d'instantier le descripteur de socket avec celui associé au numéro de fichier passé en argument. Si on n'y parvient pas, on renvoie le code fourni par la fonction `sockfd_lookup()`.

- On initialise l'en-tête de message : pas de données ancillaires, un seul élément de vecteur d'entrée-sortie (celui déclaré ci-dessus) et d'adresse celle passée en argument, de longueur celle passée en argument.
- On initialise le vecteur d'entrée-sortie avec l'adresse de tampon et la taille passées en argument.
- On modifie éventuellement le vecteur de drapeaux passé en argument.
- On fait appel à la fonction `sock_recvmsg()` étudiée ci-dessous, on libère le descripteur de socket et on renvoie le code qu'elle fournit.

30.2.2 Réception de message de socket

Code Linux 2.6.0

La fonction `sock_recvmsg()` est également définie dans le fichier `linux/net/socket.c`:

```
599 int sock_recvmsg(struct socket *sock, struct msghdr *msg,
600                 size_t size, int flags)
601 {
602         struct kiocb iocb;
603         struct sock_iocb siocb;
604         int ret;
605
606         init_sync_kiocb(&iocb, NULL);
607         iocb.private = &siocb;
608         ret = __sock_recvmsg(&iocb, sock, msg, size, flags);
609         if (-EIOCBQUEUED == ret)
610                 ret = wait_on_sync_kiocb(&iocb);
611         return ret;
612 }
```

Autrement dit :

- On déclare un bloc de contrôle d'entrée-sortie noyau.

  Code Linux 2.6.10

  Le type général `struct kiocb` des blocs d'entrée-sortie noyau est défini dans le fichier `linux/include/linux/aio.h`:

```
46 struct kiocb {
47         struct list_head        ki_run_list;
48         long                    ki_flags;
49         int                     ki_users;
50         unsigned                ki_key;         /* id de cette requete */
51
52         struct file             *ki_filp;
53         struct kioctx           *ki_ctx;        /* peut etre NULL pour les ops sync */
54         int                     (*ki_cancel)(struct kiocb *, struct io_event *);
55         ssize_t                 (*ki_retry)(struct kiocb *);
56         void                    (*ki_dtor)(struct kiocb *);
57
58         struct list_head        ki_list;        /* le coeur de aio l'utilise
60                                                    * pour l'annulation */
60
61         union {
62                 void __user             *user;
63                 struct task_struct      *tsk;
64         } ki_obj;
65         __u64                   ki_user_data;   /* donnees de l'utilisateur a
                                                      completer */
66         loff_t                  ki_pos;
67         /* Etat que nous nous rappelons etre capable de redemarrer/recuperer */
68         unsigned short          ki_opcode;
69         size_t                  ki_nbytes;      /* copie de iocb->aio_nbytes */
```

```
70        char                    __user *ki_buf; /* reste de iocb->aio_buf */
71        size_t                  ki_left;        /* octets restants */
72        wait_queue_t            ki_wait;
73        long                    ki_retried;     /* juste pour test */
74        long                    ki_kicked;      /* juste pour test */
75        long                    ki_queued;      /* juste pour test */
76
77        void                    *private;
78 };
```

– On déclare un code de retour.
– On initialise le bloc de contrôle d'entrée-sortie noyau.

La macro générale `init_sync_kiocb()` est définie dans le fichier `linux/include/linux/aio.h`: Code Linux 2.6.10

```
81 #define init_sync_kiocb(x, filp)                      \
82        do {                                           \
83                struct task_struct *tsk = current;     \
84                (x)->ki_flags = 0;                     \
85                (x)->ki_users = 1;                     \
86                (x)->ki_key = KIOCB_SYNC_KEY;          \
87                (x)->ki_filp = (filp);                 \
88                (x)->ki_ctx = &tsk->active_mm->default_kioctx;  \
89                (x)->ki_cancel = NULL;                 \
90                (x)->ki_dtor = NULL;                   \
91                (x)->ki_obj.tsk = tsk;                 \
92                (x)->ki_user_data = 0;                 \
93                init_wait((&(x)->ki_wait));            \
94        } while (0)
```

– On fait appel à la fonction `__sock_recvmsg()` interne de réception de socket, que nous étudierons dans la sous-section suivante.
– Si le code de retour de cette dernière fonction est `-EIOCBQUEUED`, on attend la fin de la synchronisation.

La fonction générale `wait_on_sync_kiocb()` est définie dans le fichier `linux/fs/aio.c`: Code Linux 2.6.10

```
311 /* wait_on_sync_kiocb :
312  *      attend que le kiocb synchronise donne ait termine.
313  */
314 ssize_t fastcall wait_on_sync_kiocb(struct kiocb *iocb)
315 {
316        while (iocb->ki_users) {
317                set_current_state(TASK_UNINTERRUPTIBLE);
318                if (!iocb->ki_users)
319                        break;
320                schedule();
321        }
322        __set_current_state(TASK_RUNNING);
323        return iocb->ki_user_data;
324 }
```

– On renvoie le code fourni par l'une ou l'autre des fonctions appelées.

30.2.3 Passage espace noyau/espace utilisateur

La fonction `move_addr_to_user()` est définie dans le fichier `linux/net/socket.c`: Code Linux 2.6.10

```
232 /**
233  *      move_addr_to_user       -       copie une adresse dans l'espace utilisateur
234  *      @kaddr : adresse de l'espace noyau
```

```
235  *      @klen : longueur de l'adresse dans le noyau
236  *      @uaddr : adresse de l'espace utilisateur
237  *      @ulen : pointeur sur le champ longueur de l'utilisateur
238  *
239  *      La valeur pointee par ulen en entree est la longueur du tampon disponible.
240  *      Ceci est surcharge par l'espace tampon utilise. -EINVAL est renvoye
241  *      si un tampon trop long est specifie et en cas de taille de tampon negative. -EFAULT
242  *      est renvoye lorsque le tampon ou le champ longueur ne sont pas
243  *      accessibles.
244  *      Apres avoir copier les donnees jusqu'a la limite specifiee par l'utilisateur, la
245  *      vraie longueur des donnees est ecrite sur la limite de longueur specifiee
246  *      par l'utilisateur. Zero est renvoye en cas de succes.
247  */
248
249 int move_addr_to_user(void *kaddr, int klen, void __user *uaddr, int __user *ulen)
250 {
251         int err;
252         int len;
253
254         if((err=get_user(len, ulen)))
255                 return err;
256         if(len>klen)
257                 len=klen;
258         if(len<0 || len> MAX_SOCK_ADDR)
259                 return -EINVAL;
260         if(len)
261         {
262                 if(copy_to_user(uaddr,kaddr,len))
263                         return -EFAULT;
264         }
265         /*
266          *      "fromlen referera a la valeur avant troncature..."
267          *                      1003.1g
268          */
269         return __put_user(klen, ulen);
270 }
```

Le code se comprend aisément.

30.2.4 Fonction interne de réception d'un message

Code Linux 2.6.10

La fonction `__sock_recvmsg()` est définie dans le fichier `linux/net/socket.c`:

```
580 static inline int __sock_recvmsg(struct kiocb *iocb, struct socket *sock,
581                                   struct msghdr *msg, size_t size, int flags)
582 {
583         int err;
584         struct sock_iocb *si = kiocb_to_siocb(iocb);
585
586         si->sock = sock;
587         si->scm = NULL;
588         si->msg = msg;
589         si->size = size;
590         si->flags = flags;
591
592         err = security_socket_recvmsg(sock, msg, size, flags);
593         if (err)
594                 return err;
595
596         return sock->ops->recvmsg(iocb, sock, msg, size, flags);
597 }
```

Autrement dit :

- On déclare une adresse de bloc de contrôle d'entrée-sortie de socket.

  Le type `struct sock_iocb` est défini dans le fichier en-tête `linux/include/net/sock.h`: Code Linux 2.6.10

```
620 /* sock_iocb : utilise pour demarrer le processus async des E/S d'une socket */
621 struct sock_iocb {
622         struct list_head        list;
623
624         int                     flags;
625         int                     size;
626         struct socket           *sock;
627         struct sock             *sk;
628         struct scm_cookie       *scm;
629         struct msghdr           *msg, async_msg;
630         struct iovec            async_iov;
631         struct kiocb            *kiocb;
632 };
```

- On initialise cette adresse avec celle de la partie spécifique du bloc de contrôle d'entrée-sortie noyau passé en argument.

  La fonction en ligne `kiocb_to_siocb()` est définie dans le fichier en-tête `linux/include/net/sock.h`: Code Linux 2.6.10

```
634 static inline struct sock_iocb *kiocb_to_siocb(struct kiocb *iocb)
635 {
636         return (struct sock_iocb *)iocb->private;
637 }
```

- On renseigne les champs de ce bloc de contrôle d'entrée-sortie de socket avec les valeurs passées en argument, l'adresse du cafteur (*cookie* en anglais) étant initialisée à `NULL`.
- On fait appel à la fonction de sécurité `security_socket_recvmsg()` qui ne nous intéresse pas dans cet ouvrage. Si le droit d'accès n'est pas accordé, on renvoie le code d'erreur fourni par cette fonction.
- On fait appel à la fonction de réception de message spécifique à la famille de protocoles utilisée pour la lecture proprement dite et on renvoie le code fourni par celle-ci.

30.3 Cas de IPv4

Nous venons de voir que la fonction interne de réception d'un message fait appel à une fonction de réception de message spécifique à la famille de protocoles. Comme on le voit à propos de `inet_dgram_ops` pour UDP, le nom de la fonction de réception d'un message spécifique à IPv4 est `sock_common_recvmsg()`. Cette fonction est définie dans le fichier `linux/net/core/sock.c`: Code Linux 2.6.10

```
1268 int sock_common_recvmsg(struct kiocb *iocb, struct socket *sock,
1269                           struct msghdr *msg, size_t size, int flags)
1270 {
1271         struct sock *sk = sock->sk;
1272         int addr_len = 0;
1273         int err;
1274
1275         err = sk->sk_prot->recvmsg(iocb, sk, msg, size, flags & MSG_DONTWAIT,
1276                                    flags & ~MSG_DONTWAIT, &addr_len);
1277         if (err >= 0)
1278                 msg->msg_namelen = addr_len;
1279         return err;
1280 }
```

Autrement dit :

- on déclare un descripteur de couche transport, que l'on instantie avec celui associé au descripteur de socket passé en argument ;
- on déclare une longueur d'adresse, que l'on initialise à zéro ;
- on fait appel à la fonction spécifique au protocole de transport ;
- si celle-ci se déroule sans problème, on met à jour la longueur de l'adresse dans l'en-tête de message passé en argument ;
- on renvoie le code fourni par cette fonction spécifique.

30.4 Cas de UDP

Code Linux 2.6.10

La fonction spécifique de réception d'un message dans le cas de UDP, comme le montre `udp_prot`, est `udp_recvmsg()`. Celle-ci est définie dans le fichier `linux/net/ipv4/udp.c` :

```
768 /*
769  *      Ceci devrait etre facile : s'il y a quelque chose le renvoyer,
770  *      sinon nous bloquons.
771  */
772
773 int udp_recvmsg(struct kiocb *iocb, struct sock *sk, struct msghdr *msg,
774                 size_t len, int noblock, int flags, int *addr_len)
775 {
776         struct inet_opt *inet = inet_sk(sk);
777         struct sockaddr_in *sin = (struct sockaddr_in *)msg->msg_name;
778         struct sk_buff *skb;
779         int copied, err;
780
781         /*
782          *      Verifier toutes les adresses passees
783          */
784         if (addr_len)
785                 *addr_len=sizeof(*sin);
786
787         if (flags & MSG_ERRQUEUE)
788                 return ip_recv_error(sk, msg, len);
789
790 try_again:
791         skb = skb_recv_datagram(sk, flags, noblock, &err);
792         if (!skb)
793                 goto out;
794
795         copied = skb->len - sizeof(struct udphdr);
796         if (copied > len) {
797                 copied = len;
798                 msg->msg_flags |= MSG_TRUNC;
799         }
800
801         if (skb->ip_summed==CHECKSUM_UNNECESSARY) {
802                 err = skb_copy_datagram_iovec(skb, sizeof(struct udphdr), msg->msg_iov,
803                                               copied);
804         } else if (msg->msg_flags&MSG_TRUNC) {
805                 if (__udp_checksum_complete(skb))
806                         goto csum_copy_err;
807                 err = skb_copy_datagram_iovec(skb, sizeof(struct udphdr), msg->msg_iov,
808                                               copied);
809         } else {
810                 err = skb_copy_and_csum_datagram_iovec(skb, sizeof(struct udphdr),
                                                       msg->msg_iov);
```

```
811
812                 if (err == -EINVAL)
813                         goto csum_copy_err;
814         }
815
816         if (err)
817                 goto out_free;
818
819         sock_recv_timestamp(msg, sk, skb);
820
821         /* Copier les adresses. */
822         if (sin)
823         {
824                 sin->sin_family = AF_INET;
825                 sin->sin_port = skb->h.uh->source;
826                 sin->sin_addr.s_addr = skb->nh.iph->saddr;
827                 memset(sin->sin_zero, 0, sizeof(sin->sin_zero));
828         }
829         if (inet->cmsg_flags)
830                 ip_cmsg_recv(msg, skb);
831
832         err = copied;
833         if (flags & MSG_TRUNC)
834                 err = skb->len - sizeof(struct udphdr);
835
836 out_free:
837         skb_free_datagram(sk, skb);
838 out:
839         return err;
840
841 csum_copy_err:
842         UDP_INC_STATS_BH(UDP_MIB_INERRORS);
843
844         /* Vider la file d'attente. */
845         if (flags&MSG_PEEK) {
846                 int clear = 0;
847                 spin_lock_irq(&sk->sk_receive_queue.lock);
848                 if (skb == skb_peek(&sk->sk_receive_queue)) {
849                         __skb_unlink(skb, &sk->sk_receive_queue);
850                         clear = 1;
851                 }
852                 spin_unlock_irq(&sk->sk_receive_queue.lock);
853                 if (clear)
854                         kfree_skb(skb);
855         }
856
857         skb_free_datagram(sk, skb);
858
859         if (noblock)
860                 return -EAGAIN;
861         goto try_again;
862 }
```

Autrement dit :

- On déclare une option Internet, que l'on instantie avec celle associée au descripteur de couche transport passé en argument.
- On déclare une adresse Internet, que l'on instantie avec celle contenue dans l'en-tête de message passé en argument.
- On déclare un descripteur de tampon de socket, une variable pour décompter le nombre d'octets copiés et un code de retour.

- Si la longueur d'adresse passée en argument est nulle, on la positionne à celle d'une adresse Internet.
- Si le vecteur des drapeaux spécifie que cela concerne les erreurs intervenues dans la couche IP, on fait appel à la fonction `ip_recv_error()`, que nous devrions étudier à propos d'ICMP.
- On essaie d'instantier le tampon de socket grâce à la fonction `skb_recv_datagram()`, étudiée dans la section suivante. Si on n'y parvient pas, on renvoie le code fourni par cette fonction.
- On calcule le nombre de caractères qui doivent être lus (copiés). Si ce nombre est supérieur à la taille `len` passée en argument, on tronque à cette valeur et on l'indique dans le vecteur des drapeaux de l'en-tête de message.
- La lecture proprement dite dépend du comportement à l'égard de la somme de contrôle IP :
  - S'il n'est pas nécessaire de s'en préoccuper, on fait appel à la fonction `skb_copy_datagram_iovec()`, que nous étudierons dans la section suivante.
  - Sinon, si les données à lire sont tronquées, on vérifie la somme de contrôle en faisant appel à la fonction `__udp_checksum_complete()` :
    - Si elle ne concorde pas :
      - On incrémente l'information statistique sur le nombre d'erreurs intervenues en entrées UDP.
      - On vide la file d'attente.
      - On libère le descripteur de socket.
      - Et on renvoie l'opposé du code d'erreur `EAGAIN`.
    - Sinon on fait appel à la fonction `skb_copy_datagram_iovec()`, comme dans le premier cas.
  - Si on doit tenir compte de la somme de contrôle IP et que les données ne sont pas tronquées, on fait appel à la fonction `skb_copy_and_csum_datagram_iovec()`, que nous étudierons dans la section suivante. Si cette fonction renvoie l'opposé du code d'erreur `EINVAL`, on effectue ce qui a été dit ci-dessus lorsque la somme de contrôle ne concorde pas.
- Si l'une des deux fonctions renvoie un code d'erreur, on libère le descripteur de socket et on renvoie le code fourni.
- On indique le moment auquel on a reçu les données en faisant appel à la fonction `sock_recv_timestamp()`, définie dans le fichier `linux/include/net/sock.h` :

Code Linux 2.6.10

```
1246 static __inline__ void
1247 sock_recv_timestamp(struct msghdr *msg, struct sock *sk, struct sk_buff *skb)
1248 {
1249         struct timeval *stamp = &skb->stamp;
1250         if (sk->sk_rcvtstamp) {
1251                 /* Course intervenue entre l'etablissement de l'estampille temporelle
1252                    et la reception du paquet. Remplir avec l'heure exacte actuelle. */
1253                 if (stamp->tv_sec == 0)
1254                         do_gettimeofday(stamp);
1255                 put_cmsg(msg, SOL_SOCKET, SO_TIMESTAMP, sizeof(struct timeval),
1256                          stamp);
1257         } else
1258                 sk->sk_stamp = *stamp;
1259 }
```

dont le code se comprend aisément.
- Si l'adresse est nulle, on renseigne les champs à partir de ceux du descripteur de socket.
- S'il y des données ancillaires, on les récupère grâce à la fonction `ip_cmsg_recv()`, qui ne nous intéressera pas ici.
- On libère le descripteur de socket et on renvoie le nombre de caractères effectivement lus.

30.5 Lien avec les tampons de socket

Nous venons de voir que la fonction `udp_recvmsg()` fait appel aux fonctions :

- `skb_recv_datagram()` d'instantiation de descripteur de tampon de socket en réception ;
- `skb_copy_datagram_iovec()` de copie de datagramme ;
- `skb_copy_and_csum_datagram_iovec()` de copie de datagramme si l'on doit tenir compte de la somme de contrôle ;
- `skb_free_datagram()` pour libérer une socket de datagramme.

Nous allons étudier l'implémentation de ces fonctions maintenant.

30.5.1 Instantiation de descripteur de tampon en réception

30.5.1.1 Fonction générale

La fonction `skb_recv_datagram()` est définie dans le fichier `linux/net/core/datagram.c` : Code Linux 2.6.10

```
116 /**
117  *      skb_recv_datagram - Recoit un skbuff de datagramme
118  *      @sk - socket
119  *      @flags - drapeaux MSG_
120  *      @noblock - operation bloquante ?
121  *      @err - code d'erreur renvoye
122  *
123  *      Recupere un skbuff de datagramme, a savoir le depilement, les reveils non bloquants
124  *      et les concurrences possibles. Ceci remplace le code identique des paquets, bruts et
125  *      udp, aussi bien que IPX AX.25 et Appletalk. Il rectifie egalement de facon definitive
126  *      le long depilement et la concurrence pour les sockets de datagramme. Si vous
127  *      changer cette routine, n'oubliez pas qu'elle doit etre re-entrante.
128  *
129  *      Cette fonction verrouillera la socket si un skb est renvoye, aussi l'appelant
130  *      doit-il deverrouiller la socket dans ce cas (normalement en appelant
131  *      skb_free_datagram)
132  *
133  *      * Elle ne verrouille plus la socket depuis aujourd'hui. Cette fonction n'a
134  *      * plus a s'occuper des problemes de concurrence. Cette mesure doit/peut ameliorer
135  *      * significativement les latences des sockets de datagramme a haute charge,
136  *      * lorsque la copie des donnees dans l'espace utilisateur prend beaucoup de temps.
137  *      * (BTW J'ai juste tue le dernier cli() dans IP/IPv6/core/netlink/packet
138  *      *  8) Gros gain.)
139  *      *                                          --ANK (980729)
140  *
141  *      L'ordre des tests lorsque nous ne trouvons pas de donnees en attente est specifie
142  *      presque explicitment par POSIX 1003.1g. Ne pas le changer sans consulter
143  *      le standard s'il vous plait.
144  */
145 struct sk_buff *skb_recv_datagram(struct sock *sk, unsigned flags,
146                                   int noblock, int *err)
147 {
148         struct sk_buff *skb;
```

```
149         long timeo;
150         /*
151          * Il est permis que l'appelant ne verifie pas sk->sk_err avant skb_recv_datagram()
152          */
153         int error = sock_error(sk);
154
155         if (error)
156                 goto no_packet;
157
158         timeo = sock_rcvtimeo(sk, noblock);
159
160         do {
161                 /* Une fois encore seul le code au niveau utilisateur appelle cette fonction,
162                  * donc rien du niveau interruption ne mangera soudainement la
                    receive_queue.
163                  *
164                  * Regarder quand meme le client nfs en cours...
165                  * Cependant cette fonction etait correcte dans tous les cas. 8)
166                  */
167                 if (flags & MSG_PEEK) {
168                         unsigned long cpu_flags;
169
170                         spin_lock_irqsave(&sk->sk_receive_queue.lock,
171                                           cpu_flags);
172                         skb = skb_peek(&sk->sk_receive_queue);
173                         if (skb)
174                                 atomic_inc(&skb->users);
175                         spin_unlock_irqrestore(&sk->sk_receive_queue.lock,
176                                                cpu_flags);
177                 } else
178                         skb = skb_dequeue(&sk->sk_receive_queue);
179
180                 if (skb)
181                         return skb;
182
183                 /* L'utilisateur ne veut pas attendre */
184                 error = -EAGAIN;
185                 if (!timeo)
186                         goto no_packet;
187
188         } while (!wait_for_packet(sk, err, &timeo));
189
190         return NULL;
191
192 no_packet:
193         *err = error;
194         return NULL;
195 }
```

Remarquons que le paramètre `sk` est un descripteur de couche transport et non un descripteur de socket comme pourrait le laisser entendre le commentaire. Commentons maintenant le code source :

- On déclare un descripteur de tampon de socket et un délai.
- On vérifie si le champ erreur du descripteur de couche de transport passé en argument est positionné. Si c'est le cas, on positionne le code d'erreur de retour (passé en argument) et on renvoie `NULL`.

Code Linux 2.6.10

La fonction `sock_error()` est définie dans le fichier `linux/include/net/sock.h`:

```
1125 /*
1126  *      Recupere un rapport d'erreur et l'efface atomiquement
```

```
1127  */
1128
1129 static inline int sock_error(struct sock *sk)
1130 {
1131         int err = xchg(&sk->sk_err, 0);
1132         return -err;
1133 }
```

– On récupère le délai indiqué dans le descripteur de couche transport.
La fonction `sock_rcvtimeo()` est définie dans le fichier `linux/include/net/sock.h`: Code Linux 2.6.10

```
1223 static inline long sock_rcvtimeo(const struct sock *sk, int noblock)
1224 {
1225         return noblock ? 0 : sk->sk_rcvtimeo;
1226 }
```

– S'il faut récupérer les données sans les enlever de la file d'attente (cas de `MSG_PEEK`):
  – on verrouille la file d'attente en réception;
  – on fait appel à la fonction `skb_peek()`;
  – si on obtient un tampon de socket grâce à cette fonction, on incrémente le nombre d'utilisateurs de celui-ci;
  – on déverrouille la file d'attente en réception.
– Sinon on fait appel à la fonction `skb_dequeue()`.
– Si on obtient un tampon de socket grâce à l'une de ces deux méthodes, on renvoie son adresse. Sinon, si le délai est dépassé, on positionne le code d'erreur à l'opposé de `EAGAIN` et on renvoie `NULL`. Si le délai n'est pas dépassé, on recommence jusqu'à ce qu'un paquet arrive.
La fonction auxiliaire `wait_for_packet()` est étudiée ci-dessous.

30.5.1.2 Attente d'un paquet

La fonction `wait_for_packet()` est définie dans le fichier `linux/net/core/datagram.c`: Code Linux 2.6.10

```
66 /*
67  * Attend un paquet...
68  */
69 static int wait_for_packet(struct sock *sk, int *err, long *timeo_p)
70 {
71         int error;
72         DEFINE_WAIT(wait);
73
74         prepare_to_wait_exclusive(sk->sk_sleep, &wait, TASK_INTERRUPTIBLE);
75
76         /* Erreurs de socket */
77         error = sock_error(sk);
78         if (error)
79                 goto out_err;
80
81         if (!skb_queue_empty(&sk->sk_receive_queue))
82                 goto out;
83
84         /* Socket fermee ? */
85         if (sk->sk_shutdown & RCV_SHUTDOWN)
86                 goto out_noerr;
87
88         /* Les paquets en sequence peuvent venir deconnecter.
```

```
89          * S'il en est ainsi, on fait un rapport sur le probleme
90          */
91         error = -ENOTCONN;
92         if (connection_based(sk) &&
93             !(sk->sk_state == TCP_ESTABLISHED || sk->sk_state == TCP_LISTEN))
94                 goto out_err;
95
96         /* signaux de manipulation */
97         if (signal_pending(current))
98                 goto interrupted;
99
100        error = 0;
101        *timeo_p = schedule_timeout(*timeo_p);
102 out:
103        finish_wait(sk->sk_sleep, &wait);
104        return error;
105 interrupted:
106        error = sock_intr_errno(*timeo_p);
107 out_err:
108        *err = error;
109        goto out;
110 out_noerr:
111        *err = 0;
112        error = 1;
113        goto out;
114 }
```

Autrement dit :

– On définit une file d'attente de nom `wait`.

Code Linux 2.6.10

La macro générale `DEFINE_WAIT()` est définie dans le fichier `linux/include/linux/wait.h`:

```
325 #define DEFINE_WAIT(name)                                               \
326         wait_queue_t name = {                                           \
327                 .task           = current,                              \
328                 .func           = autoremove_wake_function,             \
329                 .task_list      = {     .next = &(name).task_list,      \
330                                         .prev = &(name).task_list,      \
331                                 },                                      \
332         }
```

– On se prépare à attendre.

Code Linux 2.6.10

La fonction générale `prepare_to_wait_exclusive()` est définie dans le fichier `linux/kernel/wait.c`:

```
78 void fastcall
79 prepare_to_wait_exclusive(wait_queue_head_t *q, wait_queue_t *wait, int state)
80 {
81         unsigned long flags;
82
83         wait->flags |= WQ_FLAG_EXCLUSIVE;
84         spin_lock_irqsave(&q->lock, flags);
85         if (list_empty(&wait->task_list))
86                 __add_wait_queue_tail(q, wait);
87         /*
88          * Ne pas alterer l'etat de la tache si ceci est juste aller a la
89          * file d'attente sur un appel de file d'attente async
90          */
91         if (is_sync_wait(wait))
92                 set_current_state(state);
93         spin_unlock_irqrestore(&q->lock, flags);
94 }
```

- Si le descripteur de couche transport laisse apparaître une erreur, on positionne celle-ci (en argument), on indique qu'on n'attend plus et on renvoie le code d'erreur.
- Si la file d'attente est non vide, on indique qu'on n'attend plus et on renvoie 0.
- Si la socket était en train de s'arrêter, on positionne le code d'erreur (en argument) à 0, on indique qu'on n'attend plus et on renvoie 1.
- Si la socket est orientée connexion (ce qui n'est pas le cas de UDP) et si elle est déconnectée, on positionne le code d'erreur à l'opposé de ENOTCOMM, on indique qu'on n'attend plus et on renvoie le code d'erreur.

  La fonction en ligne `connection_based()` est définie dans le fichier `linux/net/core/-datagram.c`:

Code Linux 2.6.10

```
58 /*
59  *      Une socket est-elle 'orientee connexion' ?
60  */
61 static inline int connection_based(struct sock *sk)
62 {
63         return sk->sk_type == SOCK_SEQPACKET || sk->sk_type == SOCK_STREAM;
64 }
```

- Si un signal nous prévient qu'il faut interrompre l'attente, on positionne le code d'erreur à celui fourni par la fonction `sock_intr_errno()`, on indique qu'on n'attend plus et on renvoie ce code.

  La fonction en ligne `sock_intr_errno()` est définie dans le fichier `linux/include/-net/sock.h`:

Code Linux 2.6.10

```
1238 /* Helas, avec le delai, les operations socket ne sont pas redemarrables.
1239  * Comparer ceci a poll().
1240  */
1241 static inline int sock_intr_errno(long timeo)
1242 {
1243         return timeo == MAX_SCHEDULE_TIMEOUT ? -ERESTARTSYS : -EINTR;
1244 }
```

- Sinon, on attend que le délai se soit écoulé, on indique qu'on n'attend plus et on renvoie 0.

30.5.2 Copie d'un datagramme dans l'espace utilisateur

30.5.2.1 Sans tenir compte de la somme de contrôle

La fonction `skb_copy_datagram_iovec()` est définie dans le fichier `linux/net/core/datagram.c`:

Code Linux 2.6.10

```
215 /**
216  *      skb_copy_datagram_iovec - Copie un datagramme vers un iovec.
217  *      @skb - tampon a  copier
218  *      @offset - decalage dans le tampon a partir duquel debute la copie
219  *      @iovec - vecteur d'entree-sortie vers lequel copier
220  *      @len - quantite de donnees a copier du tampn vers iovec
221  *
222  *      Note : le iovec est modifie durant la copie.
223  */
224 int skb_copy_datagram_iovec(const struct sk_buff *skb, int offset,
225                             struct iovec *to, int len)
226 {
227         int start = skb_headlen(skb);
228         int i, copy = start - offset;
229
230         /* Copie l'en-tete. */
```

```
231         if (copy > 0) {
232                 if (copy > len)
233                         copy = len;
234                 if (memcpy_toiovec(to, skb->data + offset, copy))
235                         goto fault;
236                 if ((len -= copy) == 0)
237                         return 0;
238                 offset += copy;
239         }
240
241         /* Copie l'appendice en page. Hum... pourquoi ceci semble-t-il si complique ? */
242         for (i = 0; i < skb_shinfo(skb)->nr_frags; i++) {
243                 int end;
244
245                 BUG_TRAP(start <= offset + len);
246
247                 end = start + skb_shinfo(skb)->frags[i].size;
248                 if ((copy = end - offset) > 0) {
249                         int err;
250                         u8  *vaddr;
251                         skb_frag_t *frag = &skb_shinfo(skb)->frags[i];
252                         struct page *page = frag->page;
253
254                         if (copy > len)
255                                 copy = len;
256                         vaddr = kmap(page);
257                         err = memcpy_toiovec(to, vaddr + frag->page_offset +
258                                              offset - start, copy);
259                         kunmap(page);
260                         if (err)
261                                 goto fault;
262                         if (!(len -= copy))
263                                 return 0;
264                         offset += copy;
265                 }
266                 start = end;
267         }
268
269         if (skb_shinfo(skb)->frag_list) {
270                 struct sk_buff *list = skb_shinfo(skb)->frag_list;
271
272                 for (; list; list = list->next) {
273                         int end;
274
275                         BUG_TRAP(start <= offset + len);
276
277                         end = start + list->len;
278                         if ((copy = end - offset) > 0) {
279                                 if (copy > len)
280                                         copy = len;
281                                 if (skb_copy_datagram_iovec(list,
282                                                             offset - start,
283                                                             to, copy))
284                                         goto fault;
285                                 if ((len -= copy) == 0)
286                                         return 0;
287                                 offset += copy;
288                         }
289                         start = end;
290                 }
291         }
292         if (!len)
```

```
293                 return 0;
294
295 fault:
296         return -EFAULT;
297 }
```

Autrement dit :

- On se place au début des données du tampon spécifié par le descripteur de tampon de socket passé en argument.
- On calcule le nombre copy d'octets à copier.
- On copie l'en-tête.

  La fonction memcpy_toiovec() est définie dans le fichier linux/net/core/iovec.c: Code Linux 2.6.10

```
77 /*
78  *      Copie depuis le noyau vers un iovec. Renvoie -EFAULT en cas d'erreur.
79  *
80  *      Note : ceci modifie le iovec originel.
81  */
82
83 int memcpy_toiovec(struct iovec *iov, unsigned char *kdata, int len)
84 {
85         while (len > 0) {
86                 if (iov->iov_len) {
87                         int copy = min_t(unsigned int, iov->iov_len, len);
88                         if (copy_to_user(iov->iov_base, kdata, copy))
89                                 return -EFAULT;
90                         kdata += copy;
91                         len -= copy;
92                         iov->iov_len -= copy;
93                         iov->iov_base += copy;
94                 }
95                 iov++;
96         }
97
98         return 0;
99 }
```

- On copie les pages.
- On copie les listes.
- On renvoie 0 si tout s'est bien passé ou l'opposé du code d'erreur EFAULT sinon.

30.5.2.2 Copie de datagramme en tenant compte de la somme de contrôle

La fonction skb_copy_and_csum_datagram_iovec() est définie dans le fichier linux/net/core/datagram.c: Code Linux 2.6.10

```
390 /**
391  *      skb_copy_and_csum_datagram_iovec - Copie en verifiant la somme de controle
                                               un skb vers un iovec utilisateur.
392  *      @skb - skbuff
393  *      @hlen - longueur physique
394  *      @iovec - vecteur d'entree-sortie
395  *
396  *      L'appelant _doit_ verifier que le skb sera adapte a cet iovec.
397  *
398  *      Renvoie : 0       - en cas de succes.
399  *                -EINVAL - echec du controle de la somme de controle.
400  *                -EFAULT - faute durant la copie. Faites attentions que dans ce cas
401  *                          iovec peut etre modifie !
```

```
402  */
403 int skb_copy_and_csum_datagram_iovec(const struct sk_buff *skb,
404                                      int hlen, struct iovec *iov)
405 {
406         unsigned int csum;
407         int chunk = skb->len - hlen;
408
409         /* Saute les elements remplis.
410          * Elegant, voir aussi memcpy_toiovec 8)
411          */
412         while (!iov->iov_len)
413                 iov++;
414
415         if (iov->iov_len < chunk) {
416                 if ((unsigned short)csum_fold(skb_checksum(skb, 0, chunk + hlen,
417                                                            skb->csum)))
418                         goto csum_error;
419                 if (skb_copy_datagram_iovec(skb, hlen, iov, chunk))
420                         goto fault;
421         } else {
422                 csum = csum_partial(skb->data, hlen, skb->csum);
423                 if (skb_copy_and_csum_datagram(skb, hlen, iov->iov_base,
424                                                chunk, &csum))
425                         goto fault;
426                 if ((unsigned short)csum_fold(csum))
427                         goto csum_error;
428                 iov->iov_len -= chunk;
429                 iov->iov_base += chunk;
430         }
431         return 0;
432 csum_error:
433         return -EINVAL;
434 fault:
435         return -EFAULT;
436 }
```

Autrement dit :

- On calcule la taille des données à copier.
- On recherche le premier élément non vide du vecteur d'entrée-sortie.
- Si la taille de l'élément de vecteur d'entrée-sortie est inférieure au nombre d'octets à copier :
  - On vérifie la somme de contrôle. Si elle ne concorde pas, on renvoie l'opposé du code d'erreur `EINVAL`.
  - On fait appel à la fonction `skb_copy_datagram_iovec()` étudiée dans la sous-section ci-dessus pour la copie proprement dite. Si une erreur intervient, on renvoie l'opposé du code d'erreur `EFAULT`.
- Sinon :
  - On calcule la somme de contrôle partielle.
  - On fait appel à la fonction elle-même (récursivité) pour essayer de copier en vérifiant la somme de contrôle. Si une erreur intervient, on renvoie l'opposé du code d'erreur `EFAULT`.
  - On vérifie la somme de contrôle. Si elle ne concorde pas, on renvoie l'opposé du code d'erreur `EINVAL`.
  - On met à jour l'adresse de base et la longueur de l'élément de vecteur d'entrée-sortie.
- Dans les deux cas, si tout s'est bien passé, on renvoie 0.

30.5.3 Libération d'un tampon de datagramme

La fonction `skb_free_datagram()` est définie dans le fichier `linux/net/core/datagram.c`: Code Linux 2.6.10

```
197 void skb_free_datagram(struct sock *sk, struct sk_buff *skb)
198 {
199         kfree_skb(skb);
200 }
```

Cinquième partie

Envoi

Chapitre 31

Envoi d'un datagramme par une socket

Ce chapitre est le pendant du chapitre 30, traitant l'envoi des données et non plus leur réception. Son but est d'étudier l'implémentation de l'envoi de données par une socket dans le cas d'un type de communication non connecté, ce qui correspond à un datagramme et donc à UDP dans le cas de TCP/IP.

Nous avons vu au chapitre 6 que l'envoi de données de façon non connectée sur une socket s'effectue grâce à la fonction :

```
int sendto(int s, const void *buf, int len, unsigned int flags, const struct sockaddr *to,
           int tolen);
```

possèdant six arguments :

- le numéro de socket sur laquelle il faut écrire ;
- l'adresse du tampon d'écriture ;
- la longueur du tampon d'écriture ;
- les drapeaux d'option ;
- un pointeur à une adresse générique de socket, indiquant où doit être envoyé le datagramme ;
- la longueur de cette adresse.

Nous allons étudier l'implémentation de cette fonction dans ce chapitre.

31.1 Niveau implémentation des sockets

31.1.1 Fonction d'appel

Code Linux 2.6.10

La fonction d'appel `sys_sendto()` est définie dans le fichier `linux/net/socket.c`:

```
1499 /*
1500  *      Envoie un datagramme a une adresse donnee. Nous deplacons l'adresse dans l'espace
1501  *      noyau et nous verifions que l'on peut lire la zone des donnees utilisateur
1502  *      avant de faire appel au protocole.
1503  */
1504
1505 asmlinkage long sys_sendto(int fd, void __user * buff, size_t len, unsigned flags,
1506                            struct sockaddr __user *addr, int addr_len)
1507 {
1508         struct socket *sock;
1509         char address[MAX_SOCK_ADDR];
1510         int err;
1511         struct msghdr msg;
1512         struct iovec iov;
1513
1514         sock = sockfd_lookup(fd, &err);
1515         if (!sock)
1516                 goto out;
1517         iov.iov_base=buff;
1518         iov.iov_len=len;
1519         msg.msg_name=NULL;
1520         msg.msg_iov=&iov;
1521         msg.msg_iovlen=1;
1522         msg.msg_control=NULL;
1523         msg.msg_controllen=0;
1524         msg.msg_namelen=0;
1525         if(addr)
1526         {
1527                 err = move_addr_to_kernel(addr, addr_len, address);
1528                 if (err < 0)
1529                         goto out_put;
1530                 msg.msg_name=address;
1531                 msg.msg_namelen=addr_len;
1532         }
1533         if (sock->file->f_flags & O_NONBLOCK)
1534                 flags |= MSG_DONTWAIT;
1535         msg.msg_flags = flags;
1536         err = sock_sendmsg(sock, &msg, len);
1537
1538 out_put:
1539         sockfd_put(sock);
1540 out:
1541         return err;
1542 }
```

Autrement dit:

- On déclare un descripteur de socket, une adresse, un code de retour, un en-tête de message et un élément de vecteur d'entrée-sortie.
- On essaie d'instantier le descripteur de socket avec celui associé au numéro de fichier passé en argument. Si on n'y parvient pas, on renvoie le code fourni par la fonction `sockfd_lookup()`.

  Rappelons que si nous avons un numéro de fichier de socket valide, son descripteur de socket a été instantié de la façon vue au chapitre 28 grâce à la fonction `sock_create()`.

- On initialise le vecteur d'entrée-sortie avec l'adresse du tampon de mémoire vive et la taille passées en argument.
- On initialise l'en-tête de message : pas d'adresse (et donc de longueur nulle) dans une première étape, un seul élément de vecteur d'entrée-sortie (celui initialisé ci-dessus) puisque les données ne sont pas dispersées, pas de données ancillaires.
- Si une adresse est transmise en argument, on essaie de la rapatrier dans l'espace noyau. Si on n'y parvient pas, on libère le descripteur de socket et on renvoie le code fourni par la fonction `move_addr_to_kernel()`. Si on y parvient, ce que l'on espère, on renseigne les deux champs consacrés à l'adresse de l'en-tête de message avec cette adresse.
- Si le champ drapeaux de la socket spécifie que l'écriture ne doit pas être bloquante, on le répercute sur le champ vecteur de drapeaux de l'en-tête de message.
- On fait appel à la fonction `sock_sendmsg()` étudiée ci-dessous, on libère le descripteur de socket et on renvoie le code de retour qu'elle a fourni.

31.1.2 Envoi de message de socket

La fonction `sock_sendmsg()` est également définie dans le fichier `linux/net/socket.c`: Code Linux 2.6.10

```
548 int sock_sendmsg(struct socket *sock, struct msghdr *msg, size_t size)
549 {
550         struct kiocb iocb;
551         struct sock_iocb siocb;
552         int ret;
553
554         init_sync_kiocb(&iocb, NULL);
555         iocb.private = &siocb;
556         ret = __sock_sendmsg(&iocb, sock, msg, size);
557         if (-EIOCBQUEUED == ret)
558                 ret = wait_on_sync_kiocb(&iocb);
559         return ret;
560 }
```

Autrement dit :

- On déclare un bloc de contrôle d'entrée-sortie noyau, un bloc de contrôle d'entrée-sortie de socket et un code de retour.
- On initialise le bloc de contrôle d'entrée-sortie noyau.
- On renseigne le champ partie privée de ce bloc d'entrée-sortie noyau avec l'adresse du bloc de contrôle d'entrée-sortie de socket, de façon à ce que celui-ci ne fasse pas référence sur rien.
- On fait appel à la fonction interne d'envoi de message `__sock_sendmsg()`, étudiée ci-dessous.
- Si le code de retour de cette dernière fonction est `-EIOCBQUEUED`, on attend la fin de la synchronisation.
- On renvoie le code fourni par l'une ou l'autre des fonctions appelées.

31.1.3 Fonction interne d'envoi de message

La fonction `__sock_sendmsg()` est définie, elle aussi, dans le même fichier : Code Linux 2.6.10

```
530 static inline int __sock_sendmsg(struct kiocb *iocb, struct socket *sock,
531                                    struct msghdr *msg, size_t size)
532 {
533         struct sock_iocb *si = kiocb_to_siocb(iocb);
534         int err;
```

```
535
536         si->sock = sock;
537         si->scm = NULL;
538         si->msg = msg;
539         si->size = size;
540
541         err = security_socket_sendmsg(sock, msg, size);
542         if (err)
543                 return err;
544
545         return sock->ops->sendmsg(iocb, sock, msg, size);
546 }
```

Autrement dit :

- On déclare une adresse de bloc de contrôle d'entrée-sortie de socket, que l'on initialise avec celle de la partie privée du bloc de contrôle d'entrée-sortie noyau passé en argument.
- On renseigne les champs de cette entité avec les entités passées en argument, le cafteur étant initialisé à zéro.
- On fait appel à la fonction de sécurité `security_socket_sendmsg()` qui ne nous intéresse pas dans cet ouvrage. Si les droits d'accès ne sont pas accordés, on renvoie le code d'erreur fourni par cette fonction.
- On fait appel à la fonction d'envoi de message spécifique à la famille de protocoles utilisée pour l'écriture proprement dite et on renvoie le code fourni par celle-ci.

  Rappelons que le descripteur de socket passé en argument provient d'un des arguments de la fonction `sock_sendmsg()`, qui lui-même est associé au numéro de fichier passé en argument dans la fonction `sys_sendto()`. Celui-ci a été instantié dans la fonction `sock_create()` étudiée au chapitre 28. Ses champs ont été renseigné en faisant appel à la fonction de création spécifique à la famille de protocoles. Dans le cas de IPv4, la fonction spécifique `inet_create()` renseigne le champ `ops` suivant le type de communication.

31.2 Cas de IPv4

Dans le cas IPv4/UDP, l'ensemble des opérations associé à l'instance de `struct inet_protosw` s'appelle `inet_dgram_ops`. La fonction spécifique d'envoi d'un message s'appelle `inet_sendmsg()`. En fait c'est la même dans le cas de IPv4/TCP, ce qui montre que cette fonction concerne IPv4.

31.2.1 Appel à la fonction spécifique à la couche de transport

Code Linux 2.6.10

La fonction `inet_sendmsg()` est définie dans le fichier `linux/net/ipv4/af_inet.c` :

```
649 int inet_sendmsg(struct kiocb *iocb, struct socket *sock, struct msghdr *msg,
650                 size_t size)
651 {
652         struct sock *sk = sock->sk;
653
654         /* Nous pouvons avoir besoin de lier la socket. */
655         if (!inet_sk(sk)->num && inet_autobind(sk))
656                 return -EAGAIN;
657
658         return sk->sk_prot->sendmsg(iocb, sk, msg, size);
659 }
```

Autrement dit :

- On charge le descripteur de couche transport associé au descripteur de socket passé en argument.

  Rappelons que la fonction `inet_create()` a instantié ce descripteur de couche transport et renseigné ses champs, en particulier le champ `prot` avec `udp_prot` dans le cas de UDP mais pas de numéro de port.
- Si ce descripteur de couche de transport ne spécifie pas de numéro de port, on essaie de lui en attribuer un automatiquement (grâce à la fonction `inet_autobind()` étudiée ci-dessous). Si on n'y parvient pas, on renvoie l'opposé du code d'erreur `EAGAIN`.
- On fait appel à la fonction d'envoi de message spécifique au protocole de la couche de transport (TCP ou UDP).

  Dans le cas de UDP, l'instance `udp_prot` montre qu'il s'agit de la fonction `udp_sendmsg()`, que nous étudierons au chapitre 32.

31.2.2 Attribution automatique de numéro de port

La fonction `inet_autobind()` est définie dans le fichier `linux/net/ipv4/af_inet.c`: Code Linux 2.6.10

```
170 /*
171  *      Lie automatiquement une socket non liee.
172  */
173
174 static int inet_autobind(struct sock *sk)
175 {
176         struct inet_opt *inet;
177         /* Nous pouvons avoir besoin de lier la socket. */
178         lock_sock(sk);
179         inet = inet_sk(sk);
180         if (!inet->num) {
181                 if (sk->sk_prot->get_port(sk, 0)) {
182                         release_sock(sk);
183                         return -EAGAIN;
184                 }
185                 inet->sport = htons(inet->num);
186         }
187         release_sock(sk);
188         return 0;
189 }
```

Autrement dit :

- On déclare une option Internet.
- On verrouille le descripteur de couche transport passé en argument.
- On instantie l'option Internet avec celle associé au descripteur de couche transport passé en argument.
- Si le champ numéro de port de cette option Internet n'est pas renseigné, on essaie d'obtenir un numéro de port libre grâce à la fonction `get_port()` spécifique au protocole de transport. Si on n'y parvient pas, on libère le descripteur de couche transport et on renvoie l'opposé du code d'erreur `EAGAIN`. Sinon, on renseigne le champ port source `sport` de l'option Internet avec le numéro que l'on vient d'obtenir, on libère le descripteur de couche transport et on renvoie 0.

  Dans le cas d'un envoi, le numéro n'a pas été renseigné. Rappelons que la fonction spécifique pour UDP a déjà été étudiée au chapitre 30.

Chapitre 32

Envoi de datagrammes ordinaires sous UDP

Nous avons vu au chapitre 31 comment débute l'envoi d'un datagramme par une socket pour la famille de protocoles IPv4 et le type de communication non connecté, c'est-à-dire pour le protocole UDP de couche de transport, ce qui constituait l'interface application/transport. Nous allons étudier dans ce chapitre la suite du cheminement, plus précisément ce qui concerne la couche transport, en illustrant nos propos par UDP.

32.1 Fonction d'envoi spécifique à UDP

32.1.1 Description

Nous avons vu que, pour l'envoi de données par une socket dans le cas de la famille de protocoles IPv4 et d'un type de communication non connecté, il est fait appel à la fonction :

```
int udp_sendmsg(struct kiocb *iocb, struct sock *sk, struct msghdr *msg,
                size_t len)
```

dont les arguments sont :

- un(e adresse de) bloc de contrôle de données d'entrée-sortie, spécifiant où se trouvent les données ;
- un(e adresse de) descripteur de couche transport, spécifiant entre autre les adresses et numéros de port source et destination ;
- un(e adresse d')en-tête de message, spécifiant comment transmettre les données ;
- la taille des données à envoyer.

La fonction renvoie le nombre d'octets effectivement envoyés.

32.1.2 Implémentation

Code Linux 2.6.10

La fonction `udp_sendmsg()` est définie dans le fichier `linux/net/ipv4/udp.c` :

```
480 int udp_sendmsg(struct kiocb *iocb, struct sock *sk, struct msghdr *msg,
481                 size_t len)
482 {
483         struct inet_opt *inet = inet_sk(sk);
484         struct udp_opt *up = udp_sk(sk);
485         int ulen = len;
486         struct ipcm_cookie ipc;
487         struct rtable *rt = NULL;
488         int free = 0;
489         int connected = 0;
490         u32 daddr, faddr, saddr;
491         u16 dport;
492         u8  tos;
493         int err;
494         int corkreq = up->corkflag || msg->msg_flags&MSG_MORE;
495
496         if (len > 0xFFFF)
497                 return -EMSGSIZE;
498
499         /*
500          *      Verifier les drapeaux.
501          */
502
503         if (msg->msg_flags&MSG_OOB)     /* Compatibilite avec les messages d'erreur BSD */
504                 return -EOPNOTSUPP;
505
506         ipc.opt = NULL;
507
508         if (up->pending) {
509                 /*
510                  * Il y a des trames en attente.
511                  * Le verrou de la socket doit etre detenu pendant qu'on lui met un bouchon.
512                  */
513                 lock_sock(sk);
```

```
514                 if (likely(up->pending)) {
515                         if (unlikely(up->pending != AF_INET)) {
516                                 release_sock(sk);
517                                 return -EINVAL;
518                         }
519                         goto do_append_data;
520                 }
521                 release_sock(sk);
522         }
523         ulen += sizeof(struct udphdr);
524 
525         /*
526          *      Recupere et verifie l'adresse.
527          */
528         if (msg->msg_name) {
529                 struct sockaddr_in * usin = (struct sockaddr_in*)msg->msg_name;
530                 if (msg->msg_namelen < sizeof(*usin))
531                         return -EINVAL;
532                 if (usin->sin_family != AF_INET) {
533                         if (usin->sin_family != AF_UNSPEC)
534                                 return -EAFNOSUPPORT;
535                 }
536 
537                 daddr = usin->sin_addr.s_addr;
538                 dport = usin->sin_port;
539                 if (dport == 0)
540                         return -EINVAL;
541         } else {
542                 if (sk->sk_state != TCP_ESTABLISHED)
543                         return -EDESTADDRREQ;
544                 daddr = inet->daddr;
545                 dport = inet->dport;
546                 /* Ouvre un chemin rapide pour la socket connectee.
547                    La route ne sera pas utilisee si au moins une option est positionnee.
548                  */
549                 connected = 1;
550         }
551         ipc.addr = inet->saddr;
552 
553         ipc.oif = sk->sk_bound_dev_if;
554         if (msg->msg_controllen) {
555                 err = ip_cmsg_send(msg, &ipc);
556                 if (err)
557                         return err;
558                 if (ipc.opt)
559                         free = 1;
560                 connected = 0;
561         }
562         if (!ipc.opt)
563                 ipc.opt = inet->opt;
564 
565         saddr = ipc.addr;
566         ipc.addr = faddr = daddr;
567 
568         if (ipc.opt && ipc.opt->srr) {
569                 if (!daddr)
570                         return -EINVAL;
571                 faddr = ipc.opt->faddr;
572                 connected = 0;
573         }
574         tos = RT_TOS(inet->tos);
575         if (sk->sk_localroute || (msg->msg_flags & MSG_DONTROUTE) ||
```

```
576             (ipc.opt && ipc.opt->is_strictroute)) {
577                 tos |= RTO_ONLINK;
578                 connected = 0;
579         }
580
581         if (MULTICAST(daddr)) {
582                 if (!ipc.oif)
583                         ipc.oif = inet->mc_index;
584                 if (!saddr)
585                         saddr = inet->mc_addr;
586                 connected = 0;
587         }
588
589         if (connected)
590                 rt = (struct rtable*)sk_dst_check(sk, 0);
591
592         if (rt == NULL) {
593                 struct flowi fl = { .oif = ipc.oif,
594                                     .nl_u = { .ip4_u =
595                                               { .daddr = faddr,
596                                                 .saddr = saddr,
597                                                 .tos = tos } },
598                                     .proto = IPPROTO_UDP,
599                                     .uli_u = { .ports =
600                                                { .sport = inet->sport,
601                                                  .dport = dport } } };
602                 err = ip_route_output_flow(&rt, &fl, sk, !(msg->msg_flags&MSG_DONTWAIT));
603                 if (err)
604                         goto out;
605
606                 err = -EACCES;
607                 if ((rt->rt_flags & RTCF_BROADCAST) &&
608                     !sock_flag(sk, SOCK_BROADCAST))
609                         goto out;
610                 if (connected)
611                         sk_dst_set(sk, dst_clone(&rt->u.dst));
612         }
613
614         if (msg->msg_flags&MSG_CONFIRM)
615                 goto do_confirm;
616 back_from_confirm:
617
618         saddr = rt->rt_src;
619         if (!ipc.addr)
620                 daddr = ipc.addr = rt->rt_dst;
621
622         lock_sock(sk);
623         if (unlikely(up->pending)) {
624                 /* On a deja mis le bouchon a la socket lorsqu'elle a ete preparee. */
625                 /* ... Ce qui est un bogue d'application evident. --ANK */
626                 release_sock(sk);
627
628                 NETDEBUG(if (net_ratelimit()) printk(KERN_DEBUG "udp cork app bug 2\n"));
629                 err = -EINVAL;
630                 goto out;
631         }
632         /*
633          *      On met maintenant le bouchon a la socket pour les donnees en souffrance.
634          */
635         inet->cork.fl.fl4_dst = daddr;
636         inet->cork.fl.fl_ip_dport = dport;
637         inet->cork.fl.fl4_src = saddr;
```

```
638         inet->cork.fl.fl_ip_sport = inet->sport;
639         up->pending = AF_INET;
640
641 do_append_data:
642         up->len += ulen;
643         err = ip_append_data(sk, ip_generic_getfrag, msg->msg_iov, ulen,
644                         sizeof(struct udphdr), &ipc, rt,
645                         corkreq ? msg->msg_flags|MSG_MORE : msg->msg_flags);
646         if (err)
647                 udp_flush_pending_frames(sk);
648         else if (!corkreq)
649                 err = udp_push_pending_frames(sk, up);
650         release_sock(sk);
651
652 out:
653         ip_rt_put(rt);
654         if (free)
655                 kfree(ipc.opt);
656         if (!err) {
657                 UDP_INC_STATS_USER(UDP_MIB_OUTDATAGRAMS);
658                 return len;
659         }
660         return err;
661
662 do_confirm:
663         dst_confirm(&rt->u.dst);
664         if (!(msg->msg_flags&MSG_PROBE) || len)
665                 goto back_from_confirm;
666         err = 0;
667         goto out;
668 }
```

Autrement dit :

- On déclare une option Internet, que l'on instantie avec celle associée au descripteur de couche transport passé en argument.
- On déclare une adresse d'option UDP, que l'on initialise avec celle du champ adéquat du descripteur de couche transport passé en argument.
- On déclare une longueur de datagramme UDP, que l'on commence par initialiser pour l'instant avec la taille des données passée en argument.
- On déclare un cafteur IPCM, qui ne va pas nous intéresser ici.
  Le type `ipcm_cookie` est défini dans le fichier `linux/include/linux/udp.h`: Code Linux 2.6.10

```
50 struct ipcm_cookie
51 {
52         u32                     addr;
53         int                     oif;
54         struct ip_options       *opt;
55 };
```

- On déclare une adresse d'entrée du cache de routage, que l'on initialise à `NULL`.
- On déclare un test de bouchon, que l'on initialise suivant les valeurs de l'option UDP et de l'en-tête de message.
- Si la taille des données passée en argument est supérieure à 65 535 octets, celles-ci ne peuvent pas tenir sur un datagramme UDP. On renvoie donc l'opposé du code d'erreur `EMSGSIZE`.
- Puisque l'envoi des données urgentes (`MSG_OOB`) n'est pas prévu pour UDP, dans le cas d'une telle tentative, on renvoie l'opposé du code d'erreur `EOPNOTSUPP`.

- Si l'option UDP indique qu'il reste des données à envoyer en souffrance, on verrouille le descripteur de couche transport. Si elles ne correspondent pas à la famille de protocoles `AF_INET`, il y a une erreur ; on libère alors le descripteur de couche transport et on renvoie l'opposé du code d'erreur `EINVAL`. Sinon on essaie d'ajouter les données aux données nouvelles à envoyer de la façon que nous verrons ci-dessous.
- S'il n'y a pas de données à envoyer en souffrance, on ajoute à la longueur la taille d'un en-tête UDP pour obtenir la longueur du datagramme UDP.
- On récupère et on vérifie l'adresse de destination passée en argument dans l'en-tête de message :
  - Si l'adresse de l'adresse (*sic*) de l'en-tête de message est nulle, il s'agit d'une convention TCP pour ne pas redonner l'adresse à chaque fois dans le cas connecté. Si l'état indiqué dans le descripteur de couche transport est différent de `TCP_ESTABLISHED`, il y a un problème ; on renvoie l'opposé du code d'erreur `EDSTADDRREQ`. Sinon on récupère l'adresse et le numéro de port de destination à partir de l'option Internet et on indique qu'on est déjà connecté.

    Ce cas ne nous intéresse pas ici puisque nous étudions le cas non connecté.
  - Si la longueur d'adresse indiquée dans l'en-tête de message est inférieure à celle d'une adresse Internet ou si la famille de protocoles spécifiée dans l'adresse est `AF_UNSPEC`, il y a également un problème ; on renvoie donc l'opposé du code d'erreur `EINVAL`.
  - Sinon on récupère l'adresse et le numéro de port de destination à partir de cette adresse.
  - Si le numéro de port est nul, on renvoie l'opposé du code d'erreur `EINVAL`.

    Ce cas ne devrait pas se produire puisque nous avons vu au chapitre 31 que nous avons attribué un numéro de port automatiquement.
- Les champs adresse et interface physique de sortie du cafteur sont renseignés avec l'adresse source et l'interface fournie par le descripteur de couche transport.
- Si l'en-tête de message fait apparaître des données ancillaires, on fait appel à la fonction `ip_cmsg_send()`, qui ne nous intéressera pas ici. Si une erreur se produit durant cette phase, on renvoie le code fourni par cette fonction ; sinon on indique qu'on n'est pas connecté.
- Si le champ option Internet du cafteur IPCM n'est pas renseigné, on lui affecte la valeur de l'option déclarée au début.
- On sauvegarde l'adresse source dans la variable `saddr` et on renseigne le champ adresse du cafteur avec l'adresse de destination.
- Si l'option du cafteur n'est pas affectée ou si la source n'est pas renseignée : on renvoie l'opposé du code d'erreur `EINVAL` si l'adresse de destination n'est pas déterminée ; sinon on sauvegarde l'adresse du premier routeur auquel il faut envoyer le datagramme et on indique qu'on n'est pas connecté.
- Le type de service (*tos*) est obtenu à partir de l'option IP.

  Code Linux 2.6.10

  La macro `RT_TOS()` est définie dans le fichier `linux/include/linux/in_route.h` :

```
30 #define RT_TOS(tos)     ((tos)&IPTOS_TOS_MASK)
```

- Si le descripteur de couche transport, le vecteur des drapeaux de l'en-tête de message ou l'option du cafteur indiquent que l'on ne doit pas s'éloigner de la route spécifiée, on modifie le type de service en conséquence et on indique qu'on n'est pas connecté.
- Dans le cas d'une multidiffusion, on agit en conséquence, ce qui ne nous intéressera pas pour l'instant.

– Si l'on est connecté, on essaie d'initialiser l'entrée de cache de routage.

Ce cas ne nous intéresse pas puisque nous n'étudions pour l'instant que le cas non connecté.

La fonction `sk_dst_check()` est définie dans le fichier `linux/include/net/sock.h`: Code Linux 2.6.10

```
999  static inline struct dst_entry *
1000 sk_dst_check(struct sock *sk, u32 cookie)
1001 {
1002         struct dst_entry *dst = sk_dst_get(sk);
1003
1004         if (dst && dst->obsolete && dst->ops->check(dst, cookie) == NULL) {
1005                 sk_dst_reset(sk);
1006                 return NULL;
1007         }
1008
1009         return dst;
1010 }
```

La fonction `sk_dst_get()` est définie un peu plus haut dans le même fichier: Code Linux 2.6.10

```
937 static inline struct dst_entry *
938 sk_dst_get(struct sock *sk)
939 {
940         struct dst_entry *dst;
941
942         read_lock(&sk->sk_dst_lock);
943         dst = sk->sk_dst_cache;
944         if (dst)
945                 dst_hold(dst);
946         read_unlock(&sk->sk_dst_lock);
947         return dst;
948 }
```

– Si, comme dans le cas UDP, on n'a même pas essayé d'initialiser l'entrée de cache de routage ou si, comme dans le cas TCP, on n'y est pas parvenu (autrement dit si l'adresse de destination ne se trouve pas, en ce moment, dans le cache de routage):

  – On déclare une structure de flux Internet générique, que l'on instantie avec les données que l'on possède.
  – On fait appel à la fonction `ip_route_output_flow()` pour déterminer une route. Si on n'en obtient pas, on renvoie le code d'erreur fourni par cette fonction.

    La fonction `ip_route_output_flow()` sera étudiée à sa place naturelle au chapitre 33. Nous verrons alors que si la table de routage possède une adresse par défaut, on est sûr d'obtenir une route.
  – Dans le cas d'une diffusion générale, on renvoie l'opposé du code d'erreur `EACCES`.
  – Si on est connecté, ce qui n'est pas le cas pour UDP, on renseigne le champ entrée de cache de destination du descripteur de couche transport pour les envois suivants.

    La fonction en ligne `sk_dst_set()` est définie dans le fichier `linux/include/net/sock.h`: Code Linux 2.6.10

```
950 static inline void
951 __sk_dst_set(struct sock *sk, struct dst_entry *dst)
952 {
953         struct dst_entry *old_dst;
954
955         old_dst = sk->sk_dst_cache;
956         sk->sk_dst_cache = dst;
957         dst_release(old_dst);
```

```
958 }
959
960 static inline void
961 sk_dst_set(struct sock *sk, struct dst_entry *dst)
962 {
963         write_lock(&sk->sk_dst_lock);
964         __sk_dst_set(sk, dst);
965         write_unlock(&sk->sk_dst_lock);
966 }
```

- Si le vecteur des drapeaux de l'en-tête de message spécifie qu'il faut confirmer :
  - On fait appel à la fonction `dst_confirm()` définie dans le fichier `linux/include/net/dst.h` :

Code Linux 2.6.10

```
188 static inline void dst_confirm(struct dst_entry *dst)
189 {
190         if (dst)
191                 neigh_confirm(dst->neighbour);
192 }
```

    La fonction `neigh_confirm()` est définie dans le fichier `linux/include/net/neighbour.h` :

Code Linux 2.6.10

```
316 static inline void neigh_confirm(struct neighbour *neigh)
317 {
318         if (neigh)
319                 neigh->confirmed = jiffies;
320 }
```

  - Si le vecteurs des drapeaux de l'en-tête de message indique qu'il ne faut pas sonder et si la longueur des données à envoyer est nulle, on renvoie 0.
- On détermine l'adresse source et l'adresse de destination.
- On verrouille le descripteur de couche transport.
- Si on a déjà mis un bouchon à la socket alors qu'il restait des trames en souffrance, il y a visiblement une erreur. On libère donc le descripteur de tampon, on indique qu'il y a une erreur et on renvoie l'opposé du code d'erreur `EINVAL`.
- On bouche la socket pour permettre d'envoyer les données (lignes 635–638).
- On ajoute la longueur des données à envoyer à celles des données en souffrance dans le champ adéquat de l'option UDP.
- On ajoute les données en souffrance en faisant appel à la fonction `ip_append_data()`, qui sera étudiée dans la section suivante.
- Si cette fonction nous indique qu'on peut faire partir les données, on fait appel à la fonction `udp_flush_pending_frames()` pour constituer le datagramme ; sinon on fait appel à la fonction `udp_push_pending_frames()` de constitution de données UDP en souffrance.
- On libère le descripteur de couche transport.
- On libère l'entrée de cache de routage.

  La fonction en ligne `ip_rt_put()` est définie dans le fichier `linux/include/net/route.h` :

Code Linux 2.6.10

```
131 static inline void ip_rt_put(struct rtable * rt)
132 {
133         if (rt)
134                 dst_release(&rt->u.dst);
135 }
```

- On libère l'option du cafteur.
- En cas d'erreur, on renvoie le code d'erreur.
- Sinon on incrémente l'information statistique concernant le nombre de datagrammes UDP envoyés avec succès et on renvoie le nombre d'octets envoyés.

32.2 Regroupement de données UDP

Nous venons de voir que la fonction `udp_sendmsg()` fait appel à la fonction `ip_append_data()` pour regrouper les données afin de constituer un datagramme de dimension optimum. Cette fonction a un nom qui fait penser *a priori* à la couche réseau. En fait celui-ci est dû au fait que ça ressemble à de la fragmentation, mais cette fonction n'est utilisée que par UDP.

32.2.1 Fonction principale

La fonction `ip_append_data()` est définie dans le fichier `linux/net/ipv4/ip_output.c`: Code Linux 2.6.10

```
697 /*
698  *      ip_append_data() et ip_append_page() peuvent faire un grand datagramme IP
699  *      a partir de plusieurs pieces de donnees. Chaque piece sera contenue dans la socket
700  *      jusqu'a ce que ip_push_pending_frames() soit appelee. Chaque piece peut etre une page
701  *      ou des donnees moindre qu'une page.
702  *
703  *      Pas seulement UDP, d'autres protocoles de transport - par exemple les sockets
704  *      brutes - peuvent potentiellement utiliser cette interface.
705  *
706  *      PLUS TARD : la longueur doit etre ajustee avec du remplissage en queue lorsque
        necessaire.
707  */
708 int ip_append_data(struct sock *sk,
709                    int getfrag(void *from, char *to, int offset, int len,
710                                int odd, struct sk_buff *skb),
711                    void *from, int length, int transhdrlen,
712                    struct ipcm_cookie *ipc, struct rtable *rt,
713                    unsigned int flags)
714 {
715         struct inet_opt *inet = inet_sk(sk);
716         struct sk_buff *skb;
717
718         struct ip_options *opt = NULL;
719         int hh_len;
720         int exthdrlen;
721         int mtu;
722         int copy;
723         int err;
724         int offset = 0;
725         unsigned int maxfraglen, fragheaderlen;
726         int csummode = CHECKSUM_NONE;
727
728         if (flags&MSG_PROBE)
729                 return 0;
730
731         if (skb_queue_empty(&sk->sk_write_queue)) {
732                 /*
733                  * se preparer a mettre un bouchon.
734                  */
735                 opt = ipc->opt;
736                 if (opt) {
737                         if (inet->cork.opt == NULL) {
```

```
738                                 inet->cork.opt = kmalloc(sizeof(struct ip_options) + 40,
                                                             sk->sk_allocation);
739                                 if (unlikely(inet->cork.opt == NULL))
740                                         return -ENOBUFS;
741                         }
742                         memcpy(inet->cork.opt, opt, sizeof(struct ip_options)+opt->optlen);
743                         inet->cork.flags |= IPCORK_OPT;
744                         inet->cork.addr = ipc->addr;
745                 }
746                 dst_hold(&rt->u.dst);
747                 inet->cork.fragsize = mtu = dst_pmtu(&rt->u.dst);
748                 inet->cork.rt = rt;
749                 inet->cork.length = 0;
750                 sk->sk_sndmsg_page = NULL;
751                 sk->sk_sndmsg_off = 0;
752                 if ((exthdrlen = rt->u.dst.header_len) != 0) {
753                         length += exthdrlen;
754                         transhdrlen += exthdrlen;
755                 }
756         } else {
757                 rt = inet->cork.rt;
758                 if (inet->cork.flags & IPCORK_OPT)
759                         opt = inet->cork.opt;
760
761                 transhdrlen = 0;
762                 exthdrlen = 0;
763                 mtu = inet->cork.fragsize;
764         }
765         hh_len = LL_RESERVED_SPACE(rt->u.dst.dev);
766
767         fragheaderlen = sizeof(struct iphdr) + (opt ? opt->optlen : 0);
768         maxfraglen = ((mtu - fragheaderlen) & ~7) + fragheaderlen;
769
770         if (inet->cork.length + length > 0xFFFF - fragheaderlen) {
771                 ip_local_error(sk, EMSGSIZE, rt->rt_dst, inet->dport, mtu-exthdrlen);
772                 return -EMSGSIZE;
773         }
774
775         /*
776          * transhdrlen > 0 signifie que c'est le premier fragment et que nous desirons
777          * qu'il ne soit pas fragmente dans le futur.
778          */
779         if (transhdrlen &&
780             length + fragheaderlen <= mtu &&
781             rt->u.dst.dev->features&(NETIF_F_IP_CSUM|NETIF_F_NO_CSUM|NETIF_F_HW_CSUM) &&
782             !exthdrlen)
783                 csummode = CHECKSUM_HW;
784
785         inet->cork.length += length;
786
787         /* Qu'est-ce qui va se passer dans la boucle ci-dessous ?
788          *
789          * Nous utilisons la longueur de fragment calculee pour generer des skb enchaines,
790          * chaque segment est un fragment IP pret a l'envoi sur le reseau apres
791          * l'ajout de l'en-tete IP approprie.
792          */
793
794         if ((skb = skb_peek_tail(&sk->sk_write_queue)) == NULL)
795                 goto alloc_new_skb;
796
797         while (length > 0) {
798                 /* Verifier si les donnees restantes tiennent dans le paquet en cours. */
```

```
799                 copy = mtu - skb->len;
800                 if (copy < length)
801                         copy = maxfraglen - skb->len;
802                 if (copy <= 0) {
803                         char *data;
804                         unsigned int datalen;
805                         unsigned int fraglen;
806                         unsigned int fraggap;
807                         unsigned int alloclen;
808                         struct sk_buff *skb_prev;
809 alloc_new_skb:
810                         skb_prev = skb;
811                         if (skb_prev)
812                                 fraggap = skb_prev->len - maxfraglen;
813                         else
814                                 fraggap = 0;
815
816                         /*
817                          * Si les donnees restantes depassent la mtu,
818                          * nous savons que nous avons besoin de plus de fragments.
819                          */
820                         datalen = length + fraggap;
821                         if (datalen > mtu - fragheaderlen)
822                                 datalen = maxfraglen - fragheaderlen;
823                         fraglen = datalen + fragheaderlen;
824
825                         if ((flags & MSG_MORE) &&
826                             !(rt->u.dst.dev->features&NETIF_F_SG))
827                                 alloclen = mtu;
828                         else
829                                 alloclen = datalen + fragheaderlen;
830
831                         /* Le dernier fragment obtient de l'espace supplementaire a la fin.
832                          * Noter qu'avec MSG_MORE nous surallouons les fragments,
833                          * parce que nous n'avons aucune idee de quel fragment sera
834                          * le dernier.
835                          */
836                         if (datalen == length)
837                                 alloclen += rt->u.dst.trailer_len;
838
839                         if (transhdrlen) {
840                                 skb = sock_alloc_send_skb(sk,
841                                                 alloclen + hh_len + 15,
842                                                 (flags & MSG_DONTWAIT), &err);
843                         } else {
844                                 skb = NULL;
845                                 if (atomic_read(&sk->sk_wmem_alloc) <=
846                                     2 * sk->sk_sndbuf)
847                                         skb = sock_wmalloc(sk,
848                                                            alloclen + hh_len + 15, 1,
849                                                            sk->sk_allocation);
850                                 if (unlikely(skb == NULL))
851                                         err = -ENOBUFS;
852                         }
853                         if (skb == NULL)
854                                 goto error;
855
856                         /*
857                          *      Remplir les structures de controle
858                          */
859                         skb->ip_summed = csummode;
860                         skb->csum = 0;
```

```
861                     skb_reserve(skb, hh_len);
862
863                     /*
864                      *      Trouver ou est le debut des octets a placer.
865                      */
866                     data = skb_put(skb, fraglen);
867                     skb->nh.raw = data + exthdrlen;
868                     data += fragheaderlen;
869                     skb->h.raw = data + exthdrlen;
870
871                     if (fraggap) {
872                             skb->csum = skb_copy_and_csum_bits(
873                                     skb_prev, maxfraglen,
874                                     data + transhdrlen, fraggap, 0);
875                             skb_prev->csum = csum_sub(skb_prev->csum,
876                                                       skb->csum);
877                             data += fraggap;
878                             skb_trim(skb_prev, maxfraglen);
879                     }
880
881                     copy = datalen - transhdrlen - fraggap;
882                     if (copy > 0 && getfrag(from, data + transhdrlen, offset, copy,
                                                fraggap, skb) < 0) {
883                             err = -EFAULT;
884                             kfree_skb(skb);
885                             goto error;
886                     }
887
888                     offset += copy;
889                     length -= datalen - fraggap;
890                     transhdrlen = 0;
891                     exthdrlen = 0;
892                     csummode = CHECKSUM_NONE;
893
894                     /*
895                      * Mettre le paquet dans la file d'attente en souffrance.
896                      */
897                     __skb_queue_tail(&sk->sk_write_queue, skb);
898                     continue;
899             }
900
901             if (copy > length)
902                     copy = length;
903
904             if (!(rt->u.dst.dev->features&NETIF_F_SG)) {
905                     unsigned int off;
906
907                     off = skb->len;
908                     if (getfrag(from, skb_put(skb, copy),
909                                     offset, copy, off, skb) < 0) {
910                             __skb_trim(skb, off);
911                             err = -EFAULT;
912                             goto error;
913                     }
914             } else {
915                     int i = skb_shinfo(skb)->nr_frags;
916                     skb_frag_t *frag = &skb_shinfo(skb)->frags[i-1];
917                     struct page *page = sk->sk_sndmsg_page;
918                     int off = sk->sk_sndmsg_off;
919                     unsigned int left;
920
921                     if (page && (left = PAGE_SIZE - off) > 0) {
```

```
922                                 if (copy >= left)
923                                         copy = left;
924                                 if (page != frag->page) {
925                                         if (i == MAX_SKB_FRAGS) {
926                                                 err = -EMSGSIZE;
927                                                 goto error;
928                                         }
929                                         get_page(page);
930                                         skb_fill_page_desc(skb, i, page,
                                                                   sk->sk_sndmsg_off, 0);
931                                         frag = &skb_shinfo(skb)->frags[i];
932                                 }
933                         } else if (i < MAX_SKB_FRAGS) {
934                                 if (copy > PAGE_SIZE)
935                                         copy = PAGE_SIZE;
936                                 page = alloc_pages(sk->sk_allocation, 0);
937                                 if (page == NULL)  {
938                                         err = -ENOMEM;
939                                         goto error;
940                                 }
941                                 sk->sk_sndmsg_page = page;
942                                 sk->sk_sndmsg_off = 0;
943
944                                 skb_fill_page_desc(skb, i, page, 0, 0);
945                                 frag = &skb_shinfo(skb)->frags[i];
946                                 skb->truesize += PAGE_SIZE;
947                                 atomic_add(PAGE_SIZE, &sk->sk_wmem_alloc);
948                         } else {
949                                 err = -EMSGSIZE;
950                                 goto error;
951                         }
952                         if (getfrag(from, page_address(frag->page)+frag->page_offset
                            +frag->size, offset, copy, skb->len, skb) < 0) {
953                                 err = -EFAULT;
954                                 goto error;
955                         }
956                         sk->sk_sndmsg_off += copy;
957                         frag->size += copy;
958                         skb->len += copy;
959                         skb->data_len += copy;
960                 }
961                 offset += copy;
962                 length -= copy;
963         }
964
965         return 0;
966
967 error:
968         inet->cork.length -= length;
969         IP_INC_STATS(IPSTATS_MIB_OUTDISCARDS);
970         return err;
971 }
```

Autrement dit :

- On déclare une option Internet, que l'on instantie avec celle associée au descripteur de couche transport passé en argument.
- On déclare un descripteur de tampon de socket.
- On déclare une adresse de descripteur d'options IP, que l'on initialise à `NULL`.
- Si le vecteur de drapeaux passé en paramètre spécifie que l'on doit sonder, on renvoie 0.

– Si la file d'attente en émission du descripteur de couche transport est vide, on met un bouchon à la socket (lignes 732–755). Sinon l'entrée de cache de routage est initialisée avec celle associée à l'option IP.
– La longueur de l'en-tête physique est déterminée à partir de cette entrée de cache de routage.

La macro `LL_RESERVED_SPACE()` est définie dans le fichier `linux/include/linux/net-device.h`:

Code Linux 2.6.10

```
213 /* Reserve HH_DATA_MOD octets aligne a hard_header_len, c'est-a-dire au moins ca.
214  * Une alternative est :
215  *   dev->hard_header_len ? (dev->hard_header_len +
216  *                           (HH_DATA_MOD - 1)) & ~(HH_DATA_MOD - 1) : 0
217  *
218  * Nous pourrions utiliser d'autres valeurs d'alignement, mais nous devons maintenir
219  * la relation : alignement HH <= alignement LL.
220  */
221 #define LL_RESERVED_SPACE(dev) \
222         (((dev)->hard_header_len&~(HH_DATA_MOD - 1)) + HH_DATA_MOD)
223 #define LL_RESERVED_SPACE_EXTRA(dev,extra) \
224         ((((dev)->hard_header_len+extra)&~(HH_DATA_MOD - 1)) + HH_DATA_MOD)
```

– La longueur de l'en-tête de fragmentation et la longueur maximum des fragments sont calculées.
– Si la longueur est trop importante, on fait appel à la fonction `ip_local_error()`, qui ne nous intéressera pas ici, et on renvoie l'opposé du code d'erreur `EMSGSIZE`.
– S'il s'agit du premier fragment remplissant quelques conditions on spécifie que le calcul de la somme de contrôle sera effectuée par le périphérique.
– On met à jour le champ longueur du bouchon de l'option Internet.
– On essaie d'instantier le descripteur de tampon de socket avec le premier élément de la file d'attente en écriture du descripteur de couche transport.
– Tant que la longueur des données à copier est strictement positive:
  – On calcule le nombre d'octets copiés.
  – Si le nombre d'octets copiés est négatif:
    – On détermine la longueur des données. Si celle-ci est supérieure à la longueur maximale, on tronque à celle-ci.
    – On détermine la longueur totale du fragment, c'est-à-dire longueur des données et taille de l'en-tête.
    – S'il s'agit du dernier fragment, il faut un peu plus d'espace à la fin.
    – Si on n'a pas tronqué la longueur, on instantie le descripteur de tampon de socket grâce à la fonction `sock_alloc_send_skb()`, que nous étudierons à la section suivante.
    – Sinon on essaie d'instantier ce descripteur grâce à la fonction `sock_wamalloc()`. Si on n'y parvient pas, on renvoie l'opposé du code d'erreur `ENOBUFS`.

      La fonction `sock_wamalloc()` est définie dans le fichier `linux/net/core/-sock.c`:

Code Linux 2.6.10

```
737 /*
738  * Alloue un skb du tampon d'envoi des sockets.
739  */
740 struct sk_buff *sock_wmalloc(struct sock *sk, unsigned long size, int force,
                                 int priority)
```

```
741 {
742         if (force || atomic_read(&sk->sk_wmem_alloc) < sk->sk_sndbuf) {
743                 struct sk_buff * skb = alloc_skb(size, priority);
744                 if (skb) {
745                         skb_set_owner_w(skb, sk);
746                         return skb;
747                 }
748         }
749         return NULL;
750 }
```

- Si on n'est pas parvenu à instantier le descripteur de tampon de socket, on met à jour la longueur du bouchon dans l'option Internet, on incrémente le nombre de paquets IP écartés lors de l'envoi et on renvoie le code d'erreur.
- On renseigne les champs concernant la somme de contrôle du descripteur de tampon de socket.
- On déplace le début des données du tampon de socket pour accueillir l'en-tête physique.
- On se place au début des données à copier et depuis lesquelles copier.
- S'il y a un trou dans la fragmentation, on arrange ça.
- On utilise la fonction `getfrag()` passée en argument pour copier les données. Si on n'y parvient pas, on libère le descripteur de tampon, on met à jour la longueur du bouchon dans l'option Internet, on incrémente le nombre de paquets IP écartés lors de l'envoi et on renvoie l'opposé du code d'erreur `EFAULT`.

  Dans l'appel de la fonction `udp_sendmsg()`, le pointeur de fonction `getfrag()` portait sur `ip_generic_getfrag()`, que nous étudierons dans l'une des sous-sections suivantes.
- On place le paquet dans la file d'attente des paquets à émettre du descripteur de couche transport.

- Si le nombre d'octets à copier est supérieur à la longueur maximale, on tronque à cette dernière longueur.
- On effectue des opérations analogues s'il s'agit d'une page complète.

- Si tout s'est bien déroulé, on renvoie 0.

32.2.2 Instantiation d'un tampon de socket pour l'émission

Nous venons de voir que la fonction `ip_append_data()` fait appel à la fonction `sock_alloc_send_skb()` d'instantiation de descripteur de tampon de socket en émission. Celle-ci est définie dans le fichier `linux/net/core/sock.c`:

Code Linux 2.6.10

```
910 struct sk_buff *sock_alloc_send_skb(struct sock *sk, unsigned long size,
911                                        int noblock, int *errcode)
912 {
913         return sock_alloc_send_pskb(sk, size, 0, noblock, errcode);
914 }
```

qui renvoie à la fonction `sock_alloc_send_pskb()` définie dans le même fichier:

Code Linux 2.6.10

```
824 /*
825  *      Manipulateurs de tampon d'emission/reception generiques
826  */
827
828 struct sk_buff *sock_alloc_send_pskb(struct sock *sk, unsigned long header_len,
```

```
829                                     unsigned long data_len, int noblock, int *errcode)
830 {
831         struct sk_buff *skb;
832         unsigned int gfp_mask;
833         long timeo;
834         int err;
835
836         gfp_mask = sk->sk_allocation;
837         if (gfp_mask & __GFP_WAIT)
838                 gfp_mask |= __GFP_REPEAT;
839
840         timeo = sock_sndtimeo(sk, noblock);
841         while (1) {
842                 err = sock_error(sk);
843                 if (err != 0)
844                         goto failure;
845
846                 err = -EPIPE;
847                 if (sk->sk_shutdown & SEND_SHUTDOWN)
848                         goto failure;
849
850                 if (atomic_read(&sk->sk_wmem_alloc) < sk->sk_sndbuf) {
851                         skb = alloc_skb(header_len, sk->sk_allocation);
852                         if (skb) {
853                                 int npages;
854                                 int i;
855
856                                 /* Pas de page, c'est fait... */
857                                 if (!data_len)
858                                         break;
859
860                                 npages = (data_len + (PAGE_SIZE - 1)) >> PAGE_SHIFT;
861                                 skb->truesize += data_len;
862                                 skb_shinfo(skb)->nr_frags = npages;
863                                 for (i = 0; i < npages; i++) {
864                                         struct page *page;
865                                         skb_frag_t *frag;
866
867                                         page = alloc_pages(sk->sk_allocation, 0);
868                                         if (!page) {
869                                                 err = -ENOBUFS;
870                                                 skb_shinfo(skb)->nr_frags = i;
871                                                 kfree_skb(skb);
872                                                 goto failure;
873                                         }
874
875                                         frag = &skb_shinfo(skb)->frags[i];
876                                         frag->page = page;
877                                         frag->page_offset = 0;
878                                         frag->size = (data_len >= PAGE_SIZE ?
879                                                       PAGE_SIZE :
880                                                       data_len);
881                                         data_len -= PAGE_SIZE;
882                                 }
883
884                                 /* Succes complet... */
885                                 break;
886                         }
887                         err = -ENOBUFS;
888                         goto failure;
889                 }
890                 set_bit(SOCK_ASYNC_NOSPACE, &sk->sk_socket->flags);
```

```
891                 set_bit(SOCK_NOSPACE, &sk->sk_socket->flags);
892                 err = -EAGAIN;
893                 if (!timeo)
894                         goto failure;
895                 if (signal_pending(current))
896                         goto interrupted;
897                 timeo = sock_wait_for_wmem(sk, timeo);
898         }
899
900         skb_set_owner_w(skb, sk);
901         return skb;
902
903 interrupted:
904         err = sock_intr_errno(timeo);
905 failure:
906         *errcode = err;
907         return NULL;
908 }
```

Autrement dit :

- On déclare un descripteur de tampon de socket, un masque, un délai et un code de retour.
- On instantie le masque avec celui du descripteur de couche transport passé en argument, que l'on modifie éventuellement.
- On instantie le délai à partir de celui du descripteur de couche transport passé en argument.
- Si le descripteur de couche transport spécifie une erreur, on reporte le code d'erreur dans l'argument passé par adresse et on renvoie `NULL`.
- Si le descripteur de couche transport spécifie que la socket est fermée, on place l'opposé du code d'erreur `EPIPE` dans l'argument passé par adresse et on renvoie `NULL`.
- Si la taille du tampon d'envoi, telle que spécifiée par le descripteur de couche transport passé en argument, est supérieure à ce qui a été alloué au tampon :
  - On essaie d'allouer de la place pour le descripteur de tampon. Si on n'y parvient pas, on place l'opposé du code d'erreur `ENOBUFS` dans l'argument passé par adresse et on renvoie `NULL`.
  - On détermine le nombre de pages de mémoire vive dont on va avoir besoin, que l'on reporte dans le champ adéquat de l'appendice du descripteur de tampon.
  - Pour chacune de ces pages :
    - On déclare une page et une adresse de descripteur de fragment de tampon de socket.
    - On essaie d'instantier cette page. Si on n'y parvient pas, on rectifie le nombre de pages dans le champ adéquat de l'appendice du descripteur de tampon, on libère le descripteur de tampon, on place l'opposé du code d'erreur `ENOBUFS` dans l'argument passé par adresse et on renvoie `NULL`.
    - On initialise l'adresse de descripteur de fragment de tampon, on renseigne les champs de ce descripteur et on diminue la longueur des données restant à placer dans le tampon.
- Sinon :
  - On indique qu'il n'y pas assez de place pour le moment dans les drapeaux.
  - Si le délai est écoulé, on place l'opposé du code d'erreur `EAGAIN` dans l'argument passé par adresse et on renvoie `NULL`.

  - Si un signal nous dit de nous arrêter, on place l'opposé du code d'erreur fourni par ce signal dans l'argument passé par adresse et on renvoie NULL.
  - On met à jour le délai pour permettre d'attendre de la mémoire vive en écriture et on recommence.

- On associe le descripteur de tampon et le descripteur de couche transport et on renvoie l'adresse du descripteur de tampon, alloué avec succès si on parvient à cette instruction.

32.2.3 Récupération des fragments

Code Linux 2.6.10

Nous venons de voir que la fonction `ip_append_data()` fait appel à la fonction `ip_generic_getfrag()`. Celle-ci est définie dans le fichier `linux/net/ipv4/ip_output.c`:

```
669 int
670 ip_generic_getfrag(void *from, char *to, int offset, int len, int odd, struct sk_buff *skb)
671 {
672         struct iovec *iov = from;
673
674         if (skb->ip_summed == CHECKSUM_HW) {
675                 if (memcpy_fromiovecend(to, iov, offset, len) < 0)
676                         return -EFAULT;
677         } else {
678                 unsigned int csum = 0;
679                 if (csum_partial_copy_fromiovecend(to, iov, offset, len, &csum) < 0)
680                         return -EFAULT;
681                 skb->csum = csum_block_add(skb->csum, csum, odd);
682         }
683         return 0;
684 }
```

qui renvoie à la fonction `memcpy_fromiovecend()` ou à la fonction `csum_partial_copy_fromiovecend()` suivant le cas.

Code Linux 2.6.10

Ces fonctions sont définies dans le fichier `linux/net/core/iovec.c`:

```
147 /*
148  *      Pour utilisation avec ip_build_xmit
149  */
150 int memcpy_fromiovecend(unsigned char *kdata, struct iovec *iov, int offset,
151                         int len)
152 {
153         /* Passer sur les iovec termines */
154         while (offset >= iov->iov_len) {
155                 offset -= iov->iov_len;
156                 iov++;
157         }
158
159         while (len > 0) {
160                 u8 __user *base = iov->iov_base + offset;
161                 int copy = min_t(unsigned int, len, iov->iov_len - offset);
162
163                 offset = 0;
164                 if (copy_from_user(kdata, base, copy))
165                         return -EFAULT;
166                 len -= copy;
167                 kdata += copy;
168                 iov++;
169         }
170
171         return 0;
172 }
```

```
173
174 /*
175  *      Et maintenant pour le tout en un : copie et calcule la somme de controle
176  *      directement d'un iovec utilisateur dans un datagramme
177  *      Appels a csum_partial mais le dernier doit etre un multiple de 32 bits
178  *
179  *      ip_build_xmit doit s'assurer lorsqu'on fragmente que seul le dernier
180  *      appel a cette fonction sera sur un non aligne.
181  */
182 int csum_partial_copy_fromiovecend(unsigned char *kdata, struct iovec *iov,
183                                    int offset, unsigned int len, int *csump)
184 {
185         int csum = *csump;
186         int partial_cnt = 0, err = 0;
187
188         /* Passer sur les iovec termines */
189         while (offset >= iov->iov_len) {
190                 offset -= iov->iov_len;
191                 iov++;
192         }
193
194         while (len > 0) {
195                 u8 __user *base = iov->iov_base + offset;
196                 int copy = min_t(unsigned int, len, iov->iov_len - offset);
197
198                 offset = 0;
199
200                 /* Il y a un reste du precedent iov. */
201                 if (partial_cnt) {
202                         int par_len = 4 - partial_cnt;
203
204                         /* le composant iov est trop court... */
205                         if (par_len > copy) {
206                                 if (copy_from_user(kdata, base, copy))
207                                         goto out_fault;
208                                 kdata += copy;
209                                 base += copy;
210                                 partial_cnt += copy;
211                                 len -= copy;
212                                 iov++;
213                                 if (len)
214                                         continue;
215                                 *csump = csum_partial(kdata - partial_cnt,
216                                                       partial_cnt, csum);
217                                 goto out;
218                         }
219                         if (copy_from_user(kdata, base, par_len))
220                                 goto out_fault;
221                         csum = csum_partial(kdata - partial_cnt, 4, csum);
222                         kdata += par_len;
223                         base  += par_len;
224                         copy  -= par_len;
225                         len   -= par_len;
226                         partial_cnt = 0;
227                 }
228
229                 if (len > copy) {
230                         partial_cnt = copy % 4;
231                         if (partial_cnt) {
232                                 copy -= partial_cnt;
233                                 if (copy_from_user(kdata + copy, base + copy,
234                                                    partial_cnt))
```

```
235                                         goto out_fault;
236                         }
237                 }
238
239                 if (copy) {
240                         csum = csum_and_copy_from_user(base, kdata, copy,
241                                                        csum, &err);
242                         if (err)
243                                 goto out;
244                 }
245                 len   -= copy + partial_cnt;
246                 kdata += copy + partial_cnt;
247                 iov++;
248         }
249         *csump = csum;
250 out:
251         return err;
252
253 out_fault:
254         err = -EFAULT;
255         goto out;
256 }
```

sur lesquelles il n'y a pas grand chose à dire.

32.3 Constitution du datagramme

Code Linux 2.6.10

Une fois que les données sont placées dans le tampon de socket, il faut constituer le datagramme, c'est-à-dire essentiellement renseigner les champs de l'en-tête UDP. Ceci est l'objet de la fonction `udp_push_pending_frames()`, définie dans le fichier `linux/net/ipv4/udp.c`:

```
396 /*
397  * Pousse au dehors toutes les donnees en suspens en tant que datagramme UDP. La socket
       est verrouillee.
398  */
399 static int udp_push_pending_frames(struct sock *sk, struct udp_opt *up)
400 {
401         struct inet_opt *inet = inet_sk(sk);
402         struct flowi *fl = &inet->cork.fl;
403         struct sk_buff *skb;
404         struct udphdr *uh;
405         int err = 0;
406
407         /* Attrape le skbuff la ou l'espace de l'en-tete UDP existe. */
408         if ((skb = skb_peek(&sk->sk_write_queue)) == NULL)
409                 goto out;
410
411         /*
412          * Cree un en-tete UDP
413          */
414         uh = skb->h.uh;
415         uh->source = fl->fl_ip_sport;
416         uh->dest = fl->fl_ip_dport;
417         uh->len = htons(up->len);
418         uh->check = 0;
419
420         if (sk->sk_no_check == UDP_CSUM_NOXMIT) {
421                 skb->ip_summed = CHECKSUM_NONE;
422                 goto send;
423         }
424
```

```
425         if (skb_queue_len(&sk->sk_write_queue) == 1) {
426                 /*
427                  * Seulement un fragment dans la socket.
428                  */
429                 if (skb->ip_summed == CHECKSUM_HW) {
430                         skb->csum = offsetof(struct udphdr, check);
431                         uh->check = ~csum_tcpudp_magic(fl->fl4_src, fl->fl4_dst,
432                                         up->len, IPPROTO_UDP, 0);
433                 } else {
434                         skb->csum = csum_partial((char *)uh,
435                                         sizeof(struct udphdr), skb->csum);
436                         uh->check = csum_tcpudp_magic(fl->fl4_src, fl->fl4_dst,
437                                         up->len, IPPROTO_UDP, skb->csum);
438                         if (uh->check == 0)
439                                 uh->check = -1;
440                 }
441         } else {
442                 unsigned int csum = 0;
443                 /*
444                  * HW-checksum ne marche pas lorsqu'il y a deux segments ou plus
445                  * dans la socket puisque toutes les csums de sk_buffs
446                  * doivent etre ensemble.
447                  */
448                 if (skb->ip_summed == CHECKSUM_HW) {
449                         int offset = (unsigned char *)uh - skb->data;
450                         skb->csum = skb_checksum(skb, offset, skb->len - offset, 0);
451
452                         skb->ip_summed = CHECKSUM_NONE;
453                 } else {
454                         skb->csum = csum_partial((char *)uh,
455                                         sizeof(struct udphdr), skb->csum);
456                 }
457
458                 skb_queue_walk(&sk->sk_write_queue, skb) {
459                         csum = csum_add(csum, skb->csum);
460                 }
461                 uh->check = csum_tcpudp_magic(fl->fl4_src, fl->fl4_dst,
462                                 up->len, IPPROTO_UDP, csum);
463                 if (uh->check == 0)
464                         uh->check = -1;
465         }
466 send:
467         err = ip_push_pending_frames(sk);
468 out:
469         up->len = 0;
470         up->pending = 0;
471         return err;
472 }
```

Autrement dit :

- On déclare une option Internet, que l'on instantantie avec celle associée au descripteur de couche transport passé en argument.
- On déclare un flux Internet, que l'on instantie avec celui associé à cette option Internet.
- On déclare un descripteur de tampon de socket, une adresse d'en-tête UDP et un code d'erreur.
- On essaie d'instantier le descripteur de tampon avec le premier élément de la file d'attente en écriture du descripteur de couche transport passé en argument, sans retirer celui-ci de la file d'attente. Si on n'y parvient pas, on renseigne quelques champs de l'option UDP dont l'adresse est passée en argument (longueur nulle et pas de de données en attente) et

on renvoie 0.

- On instantie l'adresse d'en-tête UDP avec celle du champ de même type du descripteur de tampon de socket.
- On renseigne les champs de l'en-tête UDP : adresse source et adresse de destination à partir des données du flux Internet, longueur des données à partir des options UDP passées en argument et pas de somme de contrôle.
- Si la valeur du champ `sk_no_check` du descripteur de couche transport passé en argument est `UDP_CSUM_NOXMIT`, on renseigne le champ `ip_summed` du descripteur de tampon avec `CHECKSUM_NONE`, on fait appel à la fonction `ip_push_pending_frames()` étudiée au chapitre 33, on renseigne les champs de l'option UDP passée en argument et on renvoie le code fourni par cette fonction.

  Les constantes `UDP_CSUM_NOXMIT`, `UDP_CSUM_NORCV` et `UDP_CSUM_DEFAULT` sont définies dans le fichier `linux/include/net/udp.h` :

Code Linux 2.6.10

```
54 /* Note : ceci doit concorder avec 'valbool' dans sock_setsockopt */
55 #define UDP_CSUM_NOXMIT         1
56
57 /* Utilise par la couche SunRPC/xprt. */
58 #define UDP_CSUM_NORCV          2
59
60 /* La valeur par defaut, comme dans la RFC, demande toujours de faire le calcul. */
61 #define UDP_CSUM_DEFAULT        0
```

- Si la taille de la file d'attente est égale à 1, c'est qu'il y a seulement un fragment pour le datagramme :

  - Si la valeur du champ `ip_summed` du descripteur de tampon est `CHECKSUM_HW`, on renseigne le champ `csum` de ce descripteur avec la valeur du champ `check` de l'en-tête UDP et le champ `check` de l'en-tête UDP en utilisant la fonction `csum_tcpudp_magic()`.

    La macro générale `offsetof()` est définie dans le fichier `linux/include/linux/stddef.h` :

Code Linux 2.6.10

```
13 #undef offsetof
14 #ifdef __compiler_offsetof
15 #define offsetof(TYPE,MEMBER) __compiler_offsetof(TYPE,MEMBER)
16 #else
17 #define offsetof(TYPE, MEMBER) ((size_t) &((TYPE *)0)->MEMBER)
18 #endif
```

  - Sinon on renseigne le champ `csum` du descripteur de tampon grâce au calcul partiel et le champ `check` de l'en-tête UDP en utilisant la même fonction que ci-dessus. Si la valeur obtenue par calcul est nulle, on indique -1 puisque 0 a une signification particulière.

- Si la taille de la file d'attente est supérieure à deux :

  - Si la valeur du champ `ip_summed` du descripteur de tampon est égale à `CHECKSUM_HW` : on calcule le décalage de l'en-tête UDP ; on calcule la valeur du champ `csum` du descripteur de tampon ; on positionne la valeur du champ `ip_summed` du descripteur de tampon à `CHECKSUM_NONE`.
  - Sinon on se contente de calculer la valeur du champ `csum` du descripteur de tampon.
  - On parcourt la file d'attente en écriture afin de compléter le calcul de `csum`.

    La macro `skb_queue_walk()` est définie dans le fichier `linux/include/linux/skbuff.h` :

Code Linux 2.6.10

```
1073 #define skb_queue_walk(queue, skb) \
1074                 for (skb = (queue)->next, prefetch(skb->next);  \
1075                      (skb != (struct sk_buff *)(queue));        \
1076                      skb = skb->next, prefetch(skb->next))
1077
```

  - On calcule le champ `check` de l'en-tête UDP.

- On fait appel à la fonction `ip_push_pending_frame()`, on renseigne les champs de l'option UDP passée en argument et on renvoie le code fourni par cette fonction.

32.4 Envoi du datagramme à la couche réseau

Nous venons de voir que la fonction `udp_push_pending_frames()` de constitution du datagramme fait appel à la fonction `udp_flush_pending_frames()` ou directement à la fonction `ip_push_pending_frames()` pour envoyer le datagramme à la couche réseau IP. Nous allons étudier la première fonction dans cette section. Nous étudierons la seconde fonction dans le chapitre 33

La fonction `udp_flush_pending_frames()` est définie dans le fichier `linux/net/ipv4/udp.c`: Code Linux 2.6.10

```
382 /*
383  * Parcourt toutes les donnees en suspens et annule la mise du bouchon. La socket est
       verrouillee.
384  */
385 static void udp_flush_pending_frames(struct sock *sk)
386 {
387         struct udp_opt *up = udp_sk(sk);
388
389         if (up->pending) {
390                 up->len = 0;
391                 up->pending = 0;
392                 ip_flush_pending_frames(sk);
393         }
394 }
```

Autrement dit on fait directement appel à la fonction `ip_push_pending_frame()`.

Chapitre 33

Envoi de paquets ordinaires sous IPv4

Nous avons vu au chapitre 32 que la fonction UDP d'envoi de datagramme fait appel aux fonctions `ip_route_output_flow()` de redirection en sortie et `ip_push_pending_frames()` de constitution du paquet IP et d'envoi de celui-ci. Nous allons étudier ces fonctions dans ce chapitre.

33.1 Première étape : routage d'un paquet ordinaire

Comme dans le cas des paquets entrants, la redirection d'un paquet sortant se fait d'abord par recherche dans le cache de routage et, en cas d'échec dans celui-ci, directement par requête à la FIB.

33.1.1 Recherche dans le cache de routage

Code Linux 2.6.10

La fonction `ip_route_output_flow()` est définie dans le fichier `linux/net/ipv4/route.c`:

```
2214 int ip_route_output_flow(struct rtable **rp, struct flowi *flp, struct sock *sk, int flags)
2215 {
2216         int err;
2217
2218         if ((err = __ip_route_output_key(rp, flp)) != 0)
2219                 return err;
2220
2221         if (flp->proto) {
2222                 if (!flp->fl4_src)
2223                         flp->fl4_src = (*rp)->rt_src;
2224                 if (!flp->fl4_dst)
2225                         flp->fl4_dst = (*rp)->rt_dst;
2226                 return xfrm_lookup((struct dst_entry **)rp, flp, sk, flags);
2227         }
2228
2229         return 0;
2230 }
```

Elle fait essentiellement appel à la fonction `__ip_route_output_key()`.

La fonction `__ip_route_output_key()` est chargée de fournir une entrée de cache de routage à partir du flux Internet générique passé en paramètre. La procédure correspond tout d'abord à celle utilisée dans la fonction `ip_route_input()` : en d'autres termes, la fonction recherche dans le cache de routage s'il existe une entrée correspondante. Si aucune entrée n'a pu être trouvée, la procédure passe la main à `ip_route_output_slow()` pour effectuer une requête FIB.

Code Linux 2.6.10

Elle est définie dans le fichier `linux/net/ipv4/route.c`:

```
2180 int __ip_route_output_key(struct rtable **rp, const struct flowi *flp)
2181 {
2182         unsigned hash;
2183         struct rtable *rth;
2184
2185         hash = rt_hash_code(flp->fl4_dst, flp->fl4_src ^ (flp->oif << 5), flp->fl4_tos);
2186
2187         rcu_read_lock_bh();
2188         for (rth = rcu_dereference(rt_hash_table[hash].chain); rth;
2189                 rth = rcu_dereference(rth->u.rt_next)) {
2190                 if (rth->fl.fl4_dst == flp->fl4_dst &&
2191                     rth->fl.fl4_src == flp->fl4_src &&
2192                     rth->fl.iif == 0 &&
2193                     rth->fl.oif == flp->oif &&
2194 #ifdef CONFIG_IP_ROUTE_FWMARK
2195                     rth->fl.fl4_fwmark == flp->fl4_fwmark &&
2196 #endif
2197                     !((rth->fl.fl4_tos ^ flp->fl4_tos) &
2198                             (IPTOS_RT_MASK | RTO_ONLINK))) {
2199                         rth->u.dst.lastuse = jiffies;
2200                         dst_hold(&rth->u.dst);
2201                         rth->u.dst.__use++;
2202                         RT_CACHE_STAT_INC(out_hit);
```

```
2203                        rcu_read_unlock_bh();
2204                        *rp = rth;
2205                        return 0;
2206                }
2207                RT_CACHE_STAT_INC(out_hlist_search);
2208        }
2209        rcu_read_unlock_bh();
2210
2211        return ip_route_output_slow(rp, flp);
2212 }
```

Autrement dit :

- On déclare un index de hachage et une entrée de cache de routage.
- On calcule l'index de hachage à partir des clés fournies par le flux Internet passé en argument.
- On verrouille le dispositif d'écriture/mise à jour.
- On recherche dans la liste chaînée associée à l'index de hachage si on trouve une entrée de cache de routage dont les clès correspondent.

  La constante symbolique `RTO_ONLINK` est définie dans le fichier `linux/include/net/route.h` : Code Linux 2.6.10

  ```
  41 #define RTO_ONLINK      0x01
  ```

- Si on en trouve une :
  - On met à jour l'heure précise de dernière utilisation de cette entrée avec l'heure en cours.
  - On incrémente le nombre de références à cette entrée de cache de routage.
  - On met à jour une information statistique à ce propos.
  - On déverrouille le dispositif d'écriture/mise à jour.
  - L'argument adresse d'entrée de cache de routage prend comme valeur l'adresse de cette entrée et on renvoie 0.
- Si on n'en touve pas :
  - On met à jour une information statistique à ce propos.
  - On déverrouille le dispositif d'écriture/mise à jour.
  - On fait appel à la fonction `ip_route_output_slow()` pour consulter la FIB et on renvoie le code fourni par celle-ci.

33.1.2 Recherche étendue à la FIB

Lorsque le cache de routage ne comporte aucune entrée pour la destination d'un paquet IP créé localement, la fonction `ip_route_output_slow()` est appelée. Cette fonction soumet une requête FIB et un certain nombre de cas particuliers, pour lesquels la fonction `fib_lookup()` est insuffisante, sont traités. Si on trouve un résultat, celui-ci est consigné dans le cache de routage et on retourne la nouvelle entrée (à travers un argument).

La fonction `ip_route_output_slow()` est définie dans le fichier `linux/net/ipv4/route.c` : Code Linux 2.6.10

```
1883 /*
1884  * Routine majeure de resolution de route.
1885  */
1886
```

```
1887 static int ip_route_output_slow(struct rtable **rp, const struct flowi *oldflp)
1888 {
1889         u32 tos = oldflp->fl4_tos & (IPTOS_RT_MASK | RTO_ONLINK);
1890         struct flowi fl = { .nl_u = { .ip4_u =
1891                                       { .daddr = oldflp->fl4_dst,
1892                                         .saddr = oldflp->fl4_src,
1893                                         .tos = tos & IPTOS_RT_MASK,
1894                                         .scope = ((tos & RTO_ONLINK) ?
1895                                                   RT_SCOPE_LINK :
1896                                                   RT_SCOPE_UNIVERSE),
1897 #ifdef CONFIG_IP_ROUTE_FWMARK
1898                                         .fwmark = oldflp->fl4_fwmark
1899 #endif
1900                                       } },
1901                             .iif = loopback_dev.ifindex,
1902                             .oif = oldflp->oif };
1903         struct fib_result res;
1904         unsigned flags = 0;
1905         struct rtable *rth;
1906         struct net_device *dev_out = NULL;
1907         struct in_device *in_dev = NULL;
1908         unsigned hash;
1909         int free_res = 0;
1910         int err;
1911
1912         res.fi          = NULL;
1913 #ifdef CONFIG_IP_MULTIPLE_TABLES
1914         res.r           = NULL;
1915 #endif
1916
1917         if (oldflp->fl4_src) {
1918                 err = -EINVAL;
1919                 if (MULTICAST(oldflp->fl4_src) ||
1920                     BADCLASS(oldflp->fl4_src) ||
1921                     ZERONET(oldflp->fl4_src))
1922                         goto out;
1923
1924                 /* C'est equivalent a inet_addr_type(saddr) == RTN_LOCAL */
1925                 dev_out = ip_dev_find(oldflp->fl4_src);
1926                 if (dev_out == NULL)
1927                         goto out;
1928
1929                 /* J'ai enleve la verification de == dev_out->oif ici.
1930                    C'etait faux pour deux raisons :
1931                    1. 1. ip_dev_find(saddr) peut renvoyer une fausse iface si saddr est
1932                       assignee a de multiples interfaces.
1933                    2. De plus, nous avons le droit d'envoyer des paquets avec saddr
1934                       d'une autre iface. --ANK
1935                  */
1936
1937                 if (oldflp->oif == 0
1938                     && (MULTICAST(oldflp->fl4_dst) || oldflp->fl4_dst == 0xFFFFFFFF)) {
1939                         /* Truc special : l'utilisateur peut diriger des multidiffusions
1940                            et des diffusions generales limitees via l'interface necessire
1941                            sans le chanter sur les toits avec IP_MULTICAST_IF ou IP_PKTINFO.
1942                            Ce truc ne sert pas qu'a s'amuser, il permet a
1943                            vic, vat et ses amis de marcher.
1944                            Ils lient les sockets a loopback, positionne ttl a zero
1945                            et s'attend a ce que ca marche.
1946                            Du point de vue du cache de routage ils ont disparus,
1947                            parce que nous n'avons pas le droit de construire un chemin de
1948                            multidiffusion avec une adr source loopback, le cache de
```

```
1949                          routage ne peut pas savoir que ttl est nul, donc ce paquet
1950                          ne quittera pas cet hote et la route est valide).
1951                          Heureusement ce truc est du bon travail.
1952                        */
1953
1954                       fl.oif = dev_out->ifindex;
1955                       goto make_route;
1956               }
1957               if (dev_out)
1958                       dev_put(dev_out);
1959               dev_out = NULL;
1960       }
1961       if (oldflp->oif) {
1962               dev_out = dev_get_by_index(oldflp->oif);
1963               err = -ENODEV;
1964               if (dev_out == NULL)
1965                       goto out;
1966               if (__in_dev_get(dev_out) == NULL) {
1967                       dev_put(dev_out);
1968                       goto out;       /* Faux code d'erreur */
1969               }
1970
1971               if (LOCAL_MCAST(oldflp->fl4_dst) || oldflp->fl4_dst == 0xFFFFFFFF) {
1972                       if (!fl.fl4_src)
1973                               fl.fl4_src = inet_select_addr(dev_out, 0,
1974                                                             RT_SCOPE_LINK);
1975                       goto make_route;
1976               }
1977               if (!fl.fl4_src) {
1978                       if (MULTICAST(oldflp->fl4_dst))
1979                               fl.fl4_src = inet_select_addr(dev_out, 0,
1980                                                             fl.fl4_scope);
1981                       else if (!oldflp->fl4_dst)
1982                               fl.fl4_src = inet_select_addr(dev_out, 0,
1983                                                             RT_SCOPE_HOST);
1984               }
1985       }
1986
1987       if (!fl.fl4_dst) {
1988               fl.fl4_dst = fl.fl4_src;
1989               if (!fl.fl4_dst)
1990                       fl.fl4_dst = fl.fl4_src = htonl(INADDR_LOOPBACK);
1991               if (dev_out)
1992                       dev_put(dev_out);
1993               dev_out = &loopback_dev;
1994               dev_hold(dev_out);
1995               fl.oif = loopback_dev.ifindex;
1996               res.type = RTN_LOCAL;
1997               flags |= RTCF_LOCAL;
1998               goto make_route;
1999       }
2000
2001       if (fib_lookup(&fl, &res)) {
2002               res.fi = NULL;
2003               if (oldflp->oif) {
2004                       /* Apparemment les tables de routage sont fausses. Supposons
2005                          que la destination est sur le lien.
2006
2007                          POURQUOI ? DW.
2008                          Parce que nous avons le droit d'envoyer a iface
2009                          meme s'il N'y a PAS de routes et PAS d'adresses
2010                          assignees. Lorsque oif est specifiee, les tables de routage
```

```
2011                            sont consultees dans un seul but :
2012                            voir si la destination passe par une passerelle plutot que
2013                            d'etre directe. De plus, si MSG_DONTROUTE est positionne,
2014                            nous envoyons le paquet, en ignorant et les tables de routages
2015                            et l'etat ifaddr. --ANK
2016
2017
2018                            Nous pourrions le faire meme si oif est inconnue,
2019                            comme dans IPv6, mais nous ne le faisons pas.
2020                          */
2021
2022                         if (fl.fl4_src == 0)
2023                                 fl.fl4_src = inet_select_addr(dev_out, 0,
2024                                                               RT_SCOPE_LINK);
2025                         res.type = RTN_UNICAST;
2026                         goto make_route;
2027                 }
2028                 if (dev_out)
2029                         dev_put(dev_out);
2030                 err = -ENETUNREACH;
2031                 goto out;
2032         }
2033         free_res = 1;
2034
2035         if (res.type == RTN_LOCAL) {
2036                 if (!fl.fl4_src)
2037                         fl.fl4_src = fl.fl4_dst;
2038                 if (dev_out)
2039                         dev_put(dev_out);
2040                 dev_out = &loopback_dev;
2041                 dev_hold(dev_out);
2042                 fl.oif = dev_out->ifindex;
2043                 if (res.fi)
2044                         fib_info_put(res.fi);
2045                 res.fi = NULL;
2046                 flags |= RTCF_LOCAL;
2047                 goto make_route;
2048         }
2049
2050 #ifdef CONFIG_IP_ROUTE_MULTIPATH
2051         if (res.fi->fib_nhs > 1 && fl.oif == 0)
2052                 fib_select_multipath(&fl, &res);
2053         else
2054 #endif
2055         if (!res.prefixlen && res.type == RTN_UNICAST && !fl.oif)
2056                 fib_select_default(&fl, &res);
2057
2058         if (!fl.fl4_src)
2059                 fl.fl4_src = FIB_RES_PREFSRC(res);
2060
2061         if (dev_out)
2062                 dev_put(dev_out);
2063         dev_out = FIB_RES_DEV(res);
2064         dev_hold(dev_out);
2065         fl.oif = dev_out->ifindex;
2066
2067 make_route:
2068         if (LOOPBACK(fl.fl4_src) && !(dev_out->flags&IFF_LOOPBACK))
2069                 goto e_inval;
2070
2071         if (fl.fl4_dst == 0xFFFFFFFF)
2072                 res.type = RTN_BROADCAST;
```

```
2073        else if (MULTICAST(fl.fl4_dst))
2074                res.type = RTN_MULTICAST;
2075        else if (BADCLASS(fl.fl4_dst) || ZERONET(fl.fl4_dst))
2076                goto e_inval;
2077
2078        if (dev_out->flags & IFF_LOOPBACK)
2079                flags |= RTCF_LOCAL;
2080
2081        in_dev = in_dev_get(dev_out);
2082        if (!in_dev)
2083                goto e_inval;
2084
2085        if (res.type == RTN_BROADCAST) {
2086                flags |= RTCF_BROADCAST | RTCF_LOCAL;
2087                if (res.fi) {
2088                        fib_info_put(res.fi);
2089                        res.fi = NULL;
2090                }
2091        } else if (res.type == RTN_MULTICAST) {
2092                flags |= RTCF_MULTICAST|RTCF_LOCAL;
2093                if (!ip_check_mc(in_dev, oldflp->fl4_dst, oldflp->fl4_src, oldflp->proto))
2094                        flags &= ~RTCF_LOCAL;
2095                /* Si la route de multidiffusion n'existe pas, utiliser
2096                   celle par defaut mais pas de passerelle dans ce cas.
2097                   Oui, c'est un truc.
2098                 */
2099                if (res.fi && res.prefixlen < 4) {
2100                        fib_info_put(res.fi);
2101                        res.fi = NULL;
2102                }
2103        }
2104
2105        rth = dst_alloc(&ipv4_dst_ops);
2106        if (!rth)
2107                goto e_nobufs;
2108
2109        atomic_set(&rth->u.dst.__refcnt, 1);
2110        rth->u.dst.flags= DST_HOST;
2111        if (in_dev->cnf.no_xfrm)
2112                rth->u.dst.flags |= DST_NOXFRM;
2113        if (in_dev->cnf.no_policy)
2114                rth->u.dst.flags |= DST_NOPOLICY;
2115        rth->fl.fl4_dst = oldflp->fl4_dst;
2116        rth->fl.fl4_tos = tos;
2117        rth->fl.fl4_src = oldflp->fl4_src;
2118        rth->fl.oif     = oldflp->oif;
2119 #ifdef CONFIG_IP_ROUTE_FWMARK
2120        rth->fl.fl4_fwmark= oldflp->fl4_fwmark;
2121 #endif
2122        rth->rt_dst     = fl.fl4_dst;
2123        rth->rt_src     = fl.fl4_src;
2124        rth->rt_iif     = oldflp->oif ? : dev_out->ifindex;
2125        rth->u.dst.dev  = dev_out;
2126        dev_hold(dev_out);
2127        rth->idev       = in_dev_get(dev_out);
2128        rth->rt_gateway = fl.fl4_dst;
2129        rth->rt_spec_dst= fl.fl4_src;
2130
2131        rth->u.dst.output=ip_output;
2132
2133        RT_CACHE_STAT_INC(out_slow_tot);
2134
```

```
2135         if (flags & RTCF_LOCAL) {
2136                 rth->u.dst.input = ip_local_deliver;
2137                 rth->rt_spec_dst = fl.fl4_dst;
2138         }
2139         if (flags & (RTCF_BROADCAST | RTCF_MULTICAST)) {
2140                 rth->rt_spec_dst = fl.fl4_src;
2141                 if (flags & RTCF_LOCAL && !(dev_out->flags & IFF_LOOPBACK)) {
2142                         rth->u.dst.output = ip_mc_output;
2143                         RT_CACHE_STAT_INC(out_slow_mc);
2144                 }
2145 #ifdef CONFIG_IP_MROUTE
2146                 if (res.type == RTN_MULTICAST) {
2147                         if (IN_DEV_MFORWARD(in_dev) &&
2148                             !LOCAL_MCAST(oldflp->fl4_dst)) {
2149                                 rth->u.dst.input = ip_mr_input;
2150                                 rth->u.dst.output = ip_mc_output;
2151                         }
2152                 }
2153 #endif
2154         }
2155
2156         rt_set_nexthop(rth, &res, 0);
2157
2158
2159         rth->rt_flags = flags;
2160
2161         hash = rt_hash_code(oldflp->fl4_dst, oldflp->fl4_src ^ (oldflp->oif << 5), tos);
2162         err = rt_intern_hash(hash, rth, rp);
2163 done:
2164         if (free_res)
2165                 fib_res_put(&res);
2166         if (dev_out)
2167                 dev_put(dev_out);
2168         if (in_dev)
2169                 in_dev_put(in_dev);
2170 out:    return err;
2171
2172 e_inval:
2173         err = -EINVAL;
2174         goto done;
2175 e_nobufs:
2176         err = -ENOBUFS;
2177         goto done;
2178 }
```

Commentons le corps de cette fonction dans le cas d'un paquet ordinaire (ni multidiffusion ou diffusion générale ou autre). Autrement dit :

- Comme pour la fonction précédente le seul paramètre d'entrée est un flux Internet. Le résultat est une adresse d'entrée de cache de routage (apparaissant donc comme paramètre) et un code d'erreur (dont on espère qu'il sera nul pour indiquer qu'on a bien trouvé une entrée).
- On déclare un type de service, que l'on instantie à partir de celui fourni par le flux Internet passé en argument.
- On déclare un flux Internet, que l'on renseigne avec les champs du flux Internet passé en argument. Cependant on ne recopie pas les champs `iif` et `scope:` pour le premier, on suppose toujours que le périphérique *loopback* est utilisé ; pour le second, le champ est défini en fonction du drapeau `RTO_ONLINK` du champ `tos` du flux passé en argument et prend la valeur `RT_SCOPE_LINK` ou `RT_SCOPE_UNIVERSE`.

Ce flux Internet peut ensuite être modifié sans que les données passées en paramètre ne soient perdues.

- On déclare un résultat de consultation de la FIB, dont on renseigne le champ information `fi` à `NULL`.
- On examine, dans un premier temps, les paramètres d'entrée afin de déterminer s'ils comportent des erreurs ou s'ils correspondent à des cas justifiant un traitement particulier (lignes 1917–1999) :
  - Si l'adresse source est une adresse de multidiffusion, une mauvaise adresse ou une adresse nulle, on renvoie l'opposé du code d'erreur `EINVAL`.
  - Avec une adresse de destination de multidiffusion, on crée immédiatement une route sans effectuer de requête FIB lorsqu'une adresse source valide est spécifiée, à partir de laquelle un périphérique de sortie exploitable peut être identifié. Grâce à ce traitement spécifique, l'envoi des paquets de multidiffusion est simplifié.

    La fonction `ip_dev_find()` est définie dans le fichier `linux/net/ipv4/fib_frontend.c` :

Code Linux 2.6.10

```
99  /*
100  *      Trouver le premier peripherique avec une adresse source donnee.
101  */
102
103 struct net_device * ip_dev_find(u32 addr)
104 {
105         struct flowi fl = { .nl_u = { .ip4_u = { .daddr = addr } } };
106         struct fib_result res;
107         struct net_device *dev = NULL;
108
109 #ifdef CONFIG_IP_MULTIPLE_TABLES
110         res.r = NULL;
111 #endif
112
113         if (!ip_fib_local_table ||
114             ip_fib_local_table->tb_lookup(ip_fib_local_table, &fl, &res))
115                 return NULL;
116         if (res.type != RTN_LOCAL)
117                 goto out;
118         dev = FIB_RES_DEV(res);
119
120         if (dev)
121                 dev_hold(dev);
122 out:
123         fib_res_put(&res);
124         return dev;
125 }
```

  - Si le flux Internet passé en argument spécifie une interface de sortie, on détermine une adresse source associée et, faute de spécifier une adresse de destination, on utilise l'adresse 127.0.0.1.

- Au terme de toutes ces opérations préparatoires, la requête FIB est exécutée (ligne 2001).

  Rappelons que celle-ci donnera toujours un résultat s'il existe une adresse par défaut.
- Si celle-ci ne donne aucun résultat, il est quand même possible de poursuivre dans certains cas : lorsqu'une interface de sortie est spécifiée dans le flux Internet passé en argument, celle-ci est utilisée, en supposant que la destination se trouve au sein du réseau auquel est connecté l'hôte, en spécifiant le type passerelle ou route directe.

– Le résultat de la requête permet de distinguer les adresses de destination locales de celles qui ne le sont pas. Dans le premier type d'adresses, on utilise toujours l'interface *loopback* tandis que dans le cas des adresses non locales, il peut être nécessaire d'utiliser la fonction `fib_select_multipath()` (ligne 2052) afin de choisir entre plusieurs routes, ou d'utiliser la fonction `fib_select_default()` (ligne 2056) pour choisir entre plusieurs routes par défaut.

La fonction en ligne `fib_select_default()` est définie dans le fichier `linux/include/net/ip_fib.h`:

Code Linux 2.6.10

```
166 static inline void fib_select_default(const struct flowi *flp, struct fib_result *res)
167 {
168         if (FIB_RES_GW(*res) && FIB_RES_NH(*res).nh_scope == RT_SCOPE_LINK)
169                 ip_fib_main_table->tb_select_default(ip_fib_main_table, flp, res);
170 }
```

– Les opérations s'achèvent avec le remplissage d'une nouvelle entrée de table de routage (à partir de la ligne 2105). Le pointeur `output()` prend comme valeur l'adresse de la fonction `ip_output()` (ligne 2131) et, dans le cas où l'adresse de destination se trouve sur le système local, le pointeur `input()` prend comme valeur l'adresse de la fonction `ip_local_deliver()` (ligne 2136).

– On fait appel à la fonction `rt_set_nexthop()` (ligne 2156), définie dans le fichier `linux/net/ipv4/route.c`:

Code Linux 2.6.10

```
1413 static void rt_set_nexthop(struct rtable *rt, struct fib_result *res, u32 itag)
1414 {
1415         struct fib_info *fi = res->fi;
1416
1417         if (fi) {
1418                 if (FIB_RES_GW(*res) &&
1419                     FIB_RES_NH(*res).nh_scope == RT_SCOPE_LINK)
1420                         rt->rt_gateway = FIB_RES_GW(*res);
1421                 memcpy(rt->u.dst.metrics, fi->fib_metrics,
1422                        sizeof(rt->u.dst.metrics));
1423                 if (fi->fib_mtu == 0) {
1424                         rt->u.dst.metrics[RTAX_MTU-1] = rt->u.dst.dev->mtu;
1425                         if (rt->u.dst.metrics[RTAX_LOCK-1] & (1 << RTAX_MTU) &&
1426                             rt->rt_gateway != rt->rt_dst &&
1427                             rt->u.dst.dev->mtu > 576)
1428                                 rt->u.dst.metrics[RTAX_MTU-1] = 576;
1429                 }
1430 #ifdef CONFIG_NET_CLS_ROUTE
1431                 rt->u.dst.tclassid = FIB_RES_NH(*res).nh_tclassid;
1432 #endif
1433         } else
1434                 rt->u.dst.metrics[RTAX_MTU-1]= rt->u.dst.dev->mtu;
1435
1436         if (rt->u.dst.metrics[RTAX_HOPLIMIT-1] == 0)
1437                 rt->u.dst.metrics[RTAX_HOPLIMIT-1] = sysctl_ip_default_ttl;
1438         if (rt->u.dst.metrics[RTAX_MTU-1] > IP_MAX_MTU)
1439                 rt->u.dst.metrics[RTAX_MTU-1] = IP_MAX_MTU;
1440         if (rt->u.dst.metrics[RTAX_ADVMSS-1] == 0)
1441                 rt->u.dst.metrics[RTAX_ADVMSS-1] =
                            max_t(unsigned int, rt->u.dst.dev->mtu - 40,
1442                                                ip_rt_min_advmss);
1443         if (rt->u.dst.metrics[RTAX_ADVMSS-1] > 65535 - 40)
1444                 rt->u.dst.metrics[RTAX_ADVMSS-1] = 65535 - 40;
1445
1446 #ifdef CONFIG_NET_CLS_ROUTE
1447 #ifdef CONFIG_IP_MULTIPLE_TABLES
1448         set_class_tag(rt, fib_rules_tclass(res));
1449 #endif
```

```
1450         set_class_tag(rt, itag);
1451 #endif
1452         rt->rt_type = res->type;
1453 }
```

– La fonction `rt_intern_hash()` est utilisée (ligne 2162) pour placer l'entrée du cache dans la table de hachage. Par ailleurs, cette entrée forme la valeur renvoyée par la fonction.

33.2 Seconde étape : constitution du paquet IP

33.2.1 Implémentation des options IP

Avant d'étudier la fonction `ip_push_pending_frames()`, nous avons besoin de connaître une structure de données supplémentaires, celle correspondant aux options IP. L'envoi d'un paquet IP suppose en principe que toutes les indications nécessaires soient contenues dans l'en-tête IP du paquet. Toutefois, il arrive que des paquets contenant les informations supplémentaires exigées dans l'en-tête de protocole soient émis occasionnellement à fins de diagnostic, ou lorsque l'itinéraire d'un paquet est déjà fixé avant l'émission. Pour ce faire on peut ajouter, comme nous l'avons déjà vu lors de l'analyse de l'en-tête IP, un champ d'option de longueur variable à chaque en-tête IP. L'implémentation Linux utilise une structure de données, `struct ip_options`, définie dans le fichier `linux/include/linux/ip.h`:

Code Linux 2.6.10

```
88  struct ip_options {
89    __u32         faddr;               /* Adresse du premier saut sauvegardee              */
90    unsigned char optlen;
91    unsigned char srr;
92    unsigned char rr;
93    unsigned char ts;
94    unsigned char is_setbyuser:1,      /* Positionne par setsockopt ?                      */
95                  is_data:1,           /* Options dans __data, plutot que dans skb         */
96                  is_strictroute:1,    /* Route source stricte                             */
97                  srr_is_hit:1,        /* L'adresse de destination du paquet etait la notre */
98                  is_changed:1,        /* La somme de controle IP n'est plus valide        */
99                  rr_needaddr:1,       /* Necessite d'enregistrer l'adresse du peripherique
                                            d'entree  */
100                 ts_needtime:1,       /* Necessite d'enregistrer l'estampille temporelle  */
101                 ts_needaddr:1;       /* Necessite d'enregistrer l'adresse du peripherique
                                            de sortie */
102   unsigned char router_alert;
103   unsigned char __pad1;
104   unsigned char __pad2;
105   unsigned char __data[0];
106 };
```

33.2.2 Récupération des fragments

La fonction `ip_push_pending_frames()` est définie dans le fichier `linux/net/ipv4/ip_output.c`:

Code Linux 2.6.10

```
1107 /*
1108  *      Combine tous les fragments IP en suspens sur la socket en tant que datagramme IP
1109  *      et les pousse dehors.
1110  */
1111 int ip_push_pending_frames(struct sock *sk)
1112 {
1113         struct sk_buff *skb, *tmp_skb;
1114         struct sk_buff **tail_skb;
1115         struct inet_opt *inet = inet_sk(sk);
```

```
1116        struct ip_options *opt = NULL;
1117        struct rtable *rt = inet->cork.rt;
1118        struct iphdr *iph;
1119        int df = 0;
1120        __u8 ttl;
1121        int err = 0;
1122
1123        if ((skb = __skb_dequeue(&sk->sk_write_queue)) == NULL)
1124                goto out;
1125        tail_skb = &(skb_shinfo(skb)->frag_list);
1126
1127        /* deplace skb->data a l'en-tete ip depuis l'en-tete ext */
1128        if (skb->data < skb->nh.raw)
1129                __skb_pull(skb, skb->nh.raw - skb->data);
1130        while ((tmp_skb = __skb_dequeue(&sk->sk_write_queue)) != NULL) {
1131                __skb_pull(tmp_skb, skb->h.raw - skb->nh.raw);
1132                *tail_skb = tmp_skb;
1133                tail_skb = &(tmp_skb->next);
1134                skb->len += tmp_skb->len;
1135                skb->data_len += tmp_skb->len;
1136                skb->truesize += tmp_skb->truesize;
1137                __sock_put(tmp_skb->sk);
1138                tmp_skb->destructor = NULL;
1139                tmp_skb->sk = NULL;
1140        }
1141
1142        /* A moins que l'utilisateur ait demande une decouverte de pmtu reelle
1143         * (IP_PMTUDISC_DO), nous permettons la fragmentation de la trame generee ici.
1144         * Qu'importe que la transformation change la taille du paquet, il sera expulse.
1145         */
1146        if (inet->pmtudisc != IP_PMTUDISC_DO)
1147                skb->local_df = 1;
1148
1149        /* Le bit DF est positionne lorsque nous voulons voir DF sur les trames sortantes.
1150         * Si local_df est egalement positionne, nous permettons encore de fragmenter cette
1151         * trame localement. */
1152        if (inet->pmtudisc == IP_PMTUDISC_DO ||
1153            (!skb_shinfo(skb)->frag_list && ip_dont_fragment(sk, &rt->u.dst)))
1154                df = htons(IP_DF);
1155
1156        if (inet->cork.flags & IPCORK_OPT)
1157                opt = inet->cork.opt;
1158
1159        if (rt->rt_type == RTN_MULTICAST)
1160                ttl = inet->mc_ttl;
1161        else
1162                ttl = ip_select_ttl(inet, &rt->u.dst);
1163
1164        iph = (struct iphdr *)skb->data;
1165        iph->version = 4;
1166        iph->ihl = 5;
1167        if (opt) {
1168                iph->ihl += opt->optlen>>2;
1169                ip_options_build(skb, opt, inet->cork.addr, rt, 0);
1170        }
1171        iph->tos = inet->tos;
1172        iph->tot_len = htons(skb->len);
1173        iph->frag_off = df;
1174        if (!df) {
1175                __ip_select_ident(iph, &rt->u.dst, 0);
1176        } else {
1177                iph->id = htons(inet->id++);
```

```
1178         }
1179         iph->ttl = ttl;
1180         iph->protocol = sk->sk_protocol;
1181         iph->saddr = rt->rt_src;
1182         iph->daddr = rt->rt_dst;
1183         ip_send_check(iph);
1184
1185         skb->priority = sk->sk_priority;
1186         skb->dst = dst_clone(&rt->u.dst);
1187
1188         /* Netfilter recupere tout le skb non fragmente. */
1189         err = NF_HOOK(PF_INET, NF_IP_LOCAL_OUT, skb, NULL,
1190                       skb->dst->dev, dst_output);
1191         if (err) {
1192                 if (err > 0)
1193                         err = inet->recverr ? net_xmit_errno(err) : 0;
1194                 if (err)
1195                         goto error;
1196         }
1197
1198 out:
1199         inet->cork.flags &= ~IPCORK_OPT;
1200         if (inet->cork.opt) {
1201                 kfree(inet->cork.opt);
1202                 inet->cork.opt = NULL;
1203         }
1204         if (inet->cork.rt) {
1205                 ip_rt_put(inet->cork.rt);
1206                 inet->cork.rt = NULL;
1207         }
1208         return err;
1209
1210 error:
1211         IP_INC_STATS(IPSTATS_MIB_OUTDISCARDS);
1212         goto out;
1213 }
```

Autrement dit :

- On déclare un descripteur de tampon de socket, un descripteur de tampon de socket provisoire et une liste de descripteurs de tampon de socket.
- On déclare une option Internet, que l'on initialise avec le champ correspondant du descripteur de couche transport passé en argument.
- On déclare une adresse d'options IP, que l'on initialise à `NULL`.
- On déclare une entrée de cache de routage, que l'on initialise avec celle associée au bouchon de l'option Internet.
- On déclare un en-tête IP, un test pour DF (initialisé à 0), une durée de vie et un code d'erreur (initialisé à 0).
- On essaie de récupérer le premier descripteur de tampon de la file d'attente en écriture du descripteur de couche transport passé en argument. S'il n'y en a pas, c'est qu'il n'y a pas de données à envoyer ; on retire alors le drapeau `IPCORK_OPT` du champ bouchon de l'option Internet, on libère éventuellement les options IP et l'entrée de cache de routage du bouchon et on renvoie 0.

  La constante symbolique `IPCORK_OPT` est définie dans le fichier `linux/include/linux/ip.h` :

Code Linux 2.6.10

```
147 #define IPCORK_OPT      1       /* Des options IP etaient detenues dans ipcork.opt */
```

- On initialise l'adresse de la liste des descripteurs de tampon de socket avec celle du champ liste des fragments du descripteur de tampon que l'on vient de récupérer.
- On change éventuellement l'adresse du début des données du descripteur de tampon pour tenir compte de l'en-tête IP.
- On récupère un à un les descripteurs de tampon de la file d'attente en écriture du descripteur de couche transport passé en argument et on les place dans la liste des fragments (lignes 1130–1140).
- À moins que l'utilisateur n'ait demandé une découverte de pmtu réelle (drapeau IP_PMTUDISC_DO), nous permettons la fragmentation de la trame générée ici.
- Le bit DF est positionné lorsque nous voulons voir DF sur les trames sortantes. Si `local_df` est également positionné, nous permettons encore de fragmenter cette trame localement.

Code Linux 2.6.10

La fonction en ligne `ip_dont_fragment()` est définie dans le fichier `linux/include/net/ip.h`:

```
179 static inline
180 int ip_dont_fragment(struct sock *sk, struct dst_entry *dst)
181 {
182         return (inet_sk(sk)->pmtudisc == IP_PMTUDISC_DO ||
183                 (inet_sk(sk)->pmtudisc == IP_PMTUDISC_WANT &&
184                  !(dst_metric(dst, RTAX_LOCK)&(1<<RTAX_MTU))));
185 }
```

- Si le drapeau `IPCORK_OPT` est positionné parmi les drapeaux du bouchon de l'option Internet, on instantie de descripteur d'options IP avec les options du bouchon de l'option Internet.
- On initialise la durée de vie.

Code Linux 2.6.10

La fonction en ligne `ip_select_ttl()` est définie dans le fichier `linux/net/ipv4/ip_output.c`:

```
118 static inline int ip_select_ttl(struct inet_opt *inet, struct dst_entry *dst)
119 {
120         int ttl = inet->uc_ttl;
121
122         if (ttl < 0)
123                 ttl = dst_metric(dst, RTAX_HOPLIMIT);
124         return ttl;
125 }
```

- On instantie l'adresse de l'en-tête IP avec celle du début des données du descripteur de tampon.
- On renseigne les champs de cet en-tête : la version est IPv4, la longueur de l'en-tête est de 5 mots, on place éventuellement les options (ce qui ne nous intéresse pas pour le moment), le type de service est celui indiqué dans l'option Internet, la longueur totale du paquet est celle indiquée dans le descripteur de tampon, le décalage de fragment est positionné, la durée de vie est celle déterminée ci-dessus, le protocole est celui spécifié par le descripteur de couche transport, les adresses source et de destination sont fournies par l'entrée de cache de routage (lignes 1165–1182).

Code Linux 2.6.10

La fonction `__ip_select_ident()` est définie dans le fichier `linux/net/ipv4/route.c`:

```
939 void __ip_select_ident(struct iphdr *iph, struct dst_entry *dst, int more)
940 {
941         struct rtable *rt = (struct rtable *) dst;
942
943         if (rt) {
944                 if (rt->peer == NULL)
```

```
945                         rt_bind_peer(rt, 1);
946
947                 /* Si peer est attache a destination, il n'est jamais detache,
948                    aussi n'avons-nous pas besoin d'attraper un verrou pour le
                      dereferencer.
949                  */
950                 if (rt->peer) {
951                         iph->id = htons(inet_getid(rt->peer, more));
952                         return;
953                 }
954         } else
955                 printk(KERN_DEBUG "rt_bind_peer(0) @%p\n", NET_CALLER(iph));
956
957         ip_select_fb_ident(iph);
958 }
```

– On calcule la somme de contrôle de l'en-tête IP et on la place dans celui-ci.

La fonction en ligne `ip_send_check()` est définie dans le fichier `linux/net/ipv4/ip_output.c`:

Code Linux 2.6.10

```
95  /* Genere une somme de controle pour un datagramme IP sortant. */
96  __inline__ void ip_send_check(struct iphdr *iph)
97  {
98          iph->check = 0;
99          iph->check = ip_fast_csum((unsigned char *)iph, iph->ihl);
100 }
```

– On renseigne les champs priorité et destination du descripteur de tampon.
– On permet un point d'ancrage `Netfilter` dont la fonction suivante, l'étape qui nous intéresse, est `dst_output()`, que nous allons étudier ci-dessous.

33.3 Troisième étape : transfert de la couche réseau à la couche inférieure

33.3.1 Première sous-étape : détermination de la fonction de transfert

La fonction `dst_output()` est définie dans le fichier `linux/include/net/dst.h`:

Code Linux 2.6.10

```
219 /* Paquet sortant de reseau a transport.  */
220 static inline int dst_output(struct sk_buff *skb)
221 {
222         int err;
223
224         for (;;) {
225                 err = skb->dst->output(skb);
226
227                 if (likely(err == 0))
228                         return err;
229                 if (unlikely(err != NET_XMIT_BYPASS))
230                         return err;
231         }
232 }
```

En général, le pointeur `dst->output` a été initialisé à `ip_output()`, dans le cas d'un paquet, lors de l'étape de redirection, comme nous l'avons vu ci-dessus.

33.3.2 Deuxième sous-étape : fragmentation éventuelle

La fonction `ip_output(skb)` décide si le paquet doit être transmis immédiatement à la fonction `ip_finish()` ou s'il faut au préalable que la fonction `ip_fragment()` l'adapte à la taille d'une trame de la couche liaison.

Code Linux 2.6.10

Cette fonction est définie dans le fichier `linux/net/ipv4/ip_output.c` :

```
287 int ip_output(struct sk_buff *skb)
288 {
289         IP_INC_STATS(IPSTATS_MIB_OUTREQUESTS);
290
291         if (skb->len > dst_pmtu(skb->dst) && !skb_shinfo(skb)->tso_size)
292                 return ip_fragment(skb, ip_finish_output);
293         else
294                 return ip_finish_output(skb);
295 }
```

Nous étudierons la fonction `ip_fragment()` au chapitre 36. Étudions donc la fonction `ip_finish_output()`.

33.3.3 Troisième sous-étape : positionnement du type du paquet

Code Linux 2.6.10

La fonction `ip_finish_output()` est définie dans le fichier `linux/net/ipv4/ip_output.c` :

```
216 int ip_finish_output(struct sk_buff *skb)
217 {
218         struct net_device *dev = skb->dst->dev;
219
220         skb->dev = dev;
221         skb->protocol = htons(ETH_P_IP);
222
223         return NF_HOOK(PF_INET, NF_IP_POST_ROUTING, skb, NULL, dev,
224                        ip_finish_output2);
225 }
```

Autrement dit :

- On déclare un descripteur de périphérique réseau, que l'on instantie avec celui associé à l'entrée de cache du descripteur de tampon de socket passé en argument.
- Ce descripteur est transmis comme valeur au champ descripteur de périphérique réseau du descripteur de tampon.
- Le type de paquet de la sous-couche 2b, à savoir `ETH_P_IP`, est indiqué dans le champ protocole du descripteur de tampon.
- Le point d'ancrage `NF_IP_POST_ROUTING` est éventuellement traité, renvoyant ensuite à la fonction `ip_finish_output2()`.

33.3.4 Quatrième sous-étape : passage à la couche inférieure

Code Linux 2.6.10

La fonction `ip_finish_output2()` est définie dans le fichier `linux/net/ipv4/ip_output.c` :

```
172 static inline int ip_finish_output2(struct sk_buff *skb)
173 {
174         struct dst_entry *dst = skb->dst;
175         struct hh_cache *hh = dst->hh;
176         struct net_device *dev = dst->dev;
177         int hh_len = LL_RESERVED_SPACE(dev);
178
179         /* Soyons plutot paranoiaque que trop intelligent. */
```

```
180         if (unlikely(skb_headroom(skb) < hh_len && dev->hard_header)) {
181                 struct sk_buff *skb2;
182
183                 skb2 = skb_realloc_headroom(skb, LL_RESERVED_SPACE(dev));
184                 if (skb2 == NULL) {
185                         kfree_skb(skb);
186                         return -ENOMEM;
187                 }
188                 if (skb->sk)
189                         skb_set_owner_w(skb2, skb->sk);
190                 kfree_skb(skb);
191                 skb = skb2;
192         }
193
194 #ifdef CONFIG_NETFILTER_DEBUG
195         nf_debug_ip_finish_output2(skb);
196 #endif /*CONFIG_NETFILTER_DEBUG*/
197
198         if (hh) {
199                 int hh_alen;
200
201                 read_lock_bh(&hh->hh_lock);
202                 hh_alen = HH_DATA_ALIGN(hh->hh_len);
203                 memcpy(skb->data - hh_alen, hh->hh_data, hh_alen);
204                 read_unlock_bh(&hh->hh_lock);
205                 skb_push(skb, hh->hh_len);
206                 return hh->hh_output(skb);
207         } else if (dst->neighbour)
208                 return dst->neighbour->output(skb);
209
210         if (net_ratelimit())
211                 printk(KERN_DEBUG "ip_finish_output2: No header cache and no neighbour!\n");
212         kfree_skb(skb);
213         return -EINVAL;
214 }
```

Autrement dit :

- On déclare une entrée de cache de destination, que l'on instantie avec celle associée au descripteur de tampon de socket passé en argument.
- On déclare une entrée du cache matériel, que l'on instantie avec celle associée à l'entrée de cache de destination.
- On déclare un descripteur de périphérique réseau, que l'on instantie avec celui associé à l'entrée de cache de destination.
- On déclare une longueur d'en-tête matériel, que l'on instantie avec celle que l'on déduit du descripteur de périphérique réseau.
- Si l'espace de tête a une taille inférieure à cette longueur et que la méthode `hard_header()` existe pour le périphérique réseau (rappelons que c'est le cas pour les cartes Ethernet avec la fonction `eth_header()`) :
  - On déclare un nouveau descripteur de tampon de socket, que l'on essaie d'instantier avec le descripteur passé en argument mais avec suffisamment de place dans son espace de tête.
  - Si on ne parvient pas à l'instantier, on libère le descripteur de tampon passé en argument et on renvoie l'opposé du code d'erreur `ENOMEM`.
  - Si un descripteur de couche transport est associé au descripteur de tampon passé en argument, on l'associe également au nouveau descripteur de tampon.

Code Linux 2.6.10

La fonction en ligne `skb_set_owner_w()` est définie dans le fichier `linux/include/netsock.h`:

```
1050 static inline void skb_set_owner_w(struct sk_buff *skb, struct sock *sk)
1051 {
1052         sock_hold(sk);
1053         skb->sk = sk;
1054         skb->destructor = sock_wfree;
1055         atomic_add(skb->truesize, &sk->sk_wmem_alloc);
1056 }
```

- On libère le descripteur de tampon passé en argument que l'on remplace par le nouveau descripteur de tampon.

- Si l'entrée de cache matériel existe:
  - On déclare une longueur d'en-tête matériel alignée.
  - On verrouille l'entrée de cache matériel.
  - On instantie la longueur d'en-tête matériel alignée avec celle déterminée ci-dessus.
  - On copie les données de l'entrée de cache à l'emplacement adéquat du descripteur de tampon.
  - On déverrouille l'entrée de cache matériel.
  - On déplace le début de la zone des données du descripteur de tampon.
  - On fait appel à la fonction d'envoi de trame spécifiée par cette entrée de cache et on renvoie le code fourni par cette fonction.

    Il s'agit de la fonction `dev_queue_xmit()` d'émission d'une trame, étudiée au chapitre 34, comme le montre les instances de `struct neigh_ops` définies dans le fichier `linux/net/ipv4/arp.c`:

Code Linux 2.6.10

```
138 static struct neigh_ops arp_generic_ops = {
139         .family =               AF_INET,
140         .solicit =              arp_solicit,
141         .error_report =         arp_error_report,
142         .output =               neigh_resolve_output,
143         .connected_output =     neigh_connected_output,
144         .hh_output =            dev_queue_xmit,
145         .queue_xmit =           dev_queue_xmit,
146 };
147
148 static struct neigh_ops arp_hh_ops = {
149         .family =               AF_INET,
150         .solicit =              arp_solicit,
151         .error_report =         arp_error_report,
152         .output =               neigh_resolve_output,
153         .connected_output =     neigh_resolve_output,
154         .hh_output =            dev_queue_xmit,
155         .queue_xmit =           dev_queue_xmit,
156 };
```

- Si l'entrée de cache matériel n'est pas définie dans l'entrée de cache de destination mais qu'une station de voisinage y est spécifiée, on fait appel à la fonction d'envoi de trame spécifiée par celle-ci et on renvoie le code fourni par cette fonction.
- Sinon on affiche un message noyau, on libère le descripteur de tampon et on renvoie l'opposé du code d'erreur `EINVAL`.

Chapitre 34

Envoi de trames

Comme nous venons de le voir dans le chapitre 33, les paquets IP sont transmis à la couche liaison de données pour être envoyés par un périphérique réseau grâce à la fonction `dev_queue_xmit(skb)`, où `skb` désigne un descripteur de tampon de socket dont la partie données comprend un paquet de la couche réseau.

Nous allons étudier dans ce chapitre comment les trames sont envoyées en prenant comme exemple la carte Ethernet 3Com 501.

Une fois que l'interface est activée, pour l'émission d'une trame le noyau se contente d'indiquer à la carte réseau qu'il y a des données à envoyer et leur emplacement en mémoire centrale. La carte s'occupe du reste, utilisant l'accès direct à la mémoire (DMA) pour récupérer les données. À la fin de l'émission, la carte lève une interruption pour prévenir le noyau que la trame a été envoyée avec succès ou que le délai s'est écoulé.

Les données consistent en un paquet de la couche réseau. La carte s'occupe d'encapsuler celui-ci pour obtenir une trame.

34.1 Transmission à la carte réseau : partie générale

La fonction `dev_queue_xmit()` traite la partie générale (c'est-à-dire non liée à un périphérique particulier) de l'envoi d'une trame. Elle appelle soit une fonction de traitement de la file d'attente des trames en partance, soit directement une fonction spécifique à la carte réseau utilisée. Autrement dit elle distingue deux cas :

- Si une méthode de gestion des file d'attente (`dev->enqueue != NULL`) est associée au périphérique, elle place le tampon de socket dans la file d'attente des trames en partance (grâce à la méthode `dev->qdisc->enqueue()`) puis elle fait appel à la méthode `qdisc_run()` pour déclencher la poursuite du traitement des paquets à envoyer.
- Sinon elle fait appel à l'émission immédiat du paquet (grâce à la fonction spécifique `dev->hard_start_xmit()`). Ce cas correspond aux périphériques réseau logiques comme le périphérique en boucle ou aux périphériques de tunnelisation.

34.1.1 Première étape : mise en file d'attente

34.1.1.1 Fonction de mise en file d'attente

Code Linux 2.6.10

La fonction `dev_queue_xmit()` est définie dans le fichier `linux/net/core/dev.c` :

```
1205 /**
1206  *      dev_queue_xmit - fait partir un tampon
1207  *      @skb : tampon a faire partir
1208  *
1209  *      Met en file d'attente un tampon pour qu'il soit transmis a un peripherique reseau.
1210  *      L'appelant doit avoir positionne le peripherique, la priorite et construit le
1211  *      tampon avant d'appeler cette fonction. La fonction peut etre appelee depuis une
        interruption.
1212  *
1213  *      Un code errno negatif est renvoye en cas d'echec. Un succes ne garantit pas
1214  *      que la trame sera transmise car elle peut etre detruite en cas de
1215  *      congestion ou de formation du trafic.
1216  */
1217
1218 int dev_queue_xmit(struct sk_buff *skb)
1219 {
1220         struct net_device *dev = skb->dev;
1221         struct Qdisc *q;
1222         int rc = -ENOMEM;
1223
1224         if (skb_shinfo(skb)->frag_list &&
1225             !(dev->features & NETIF_F_FRAGLIST) &&
1226             __skb_linearize(skb, GFP_ATOMIC))
1227                 goto out_kfree_skb;
1228
1229         /* Un skb fragmente est linearise si le peripherique ne supporte pas SG,
1230          * ou si au moins un des fragments est dans highmem et que le peripherique
1231          * ne supporte pas la DMA pour elle.
1232          */
1233         if (skb_shinfo(skb)->nr_frags &&
1234             (!(dev->features & NETIF_F_SG) || illegal_highdma(dev, skb)) &&
1235             __skb_linearize(skb, GFP_ATOMIC))
1236                 goto out_kfree_skb;
1237
1238         /* Si la somme de controle du paquet n'est pas calculee et que le peripherique
1239          * ne supporte pas le calcul pour ce protocole, l'effectuer ici.
1240          */
1241         if (skb->ip_summed == CHECKSUM_HW &&
```

```
1242            (!(dev->features & (NETIF_F_HW_CSUM | NETIF_F_NO_CSUM)) &&
1243             (!(dev->features & NETIF_F_IP_CSUM) ||
1244              skb->protocol != htons(ETH_P_IP))))
1245                if (skb_checksum_help(skb, 0))
1246                        goto out_kfree_skb;
1247
1248        /* Annihiler les irqs logicielles pour divers verrous ci-dessous. Stopper
1249         * egalement la preemption pour RCU.
1250         */
1251        local_bh_disable();
1252
1253        /* Les mises a jour de qdisc sont serialisees par queue_lock.
1254         * La struct Qdisc qui est pointee par qdisc est maintenant une
1255         * structure rcu - on peut y acceder sans acquerir
1256         * de verrou (mais la structure peut ne pas etre a jour). La liberation du
1257         * qdisc sera retardee jusqu'a ce qu'il soit connu qu'il n'y a plus
1258         * aucune reference a lui.
1259         *
1260         * Si le qdisc a une fonction enqueue, nous aurons encore besoin de
1261         * detenir queue_lock avant de l'appeler, puisque queue_lock
1262         * serialise aussi l'acces a la file d'attente des peripheriques.
1263         */
1264
1265        q = rcu_dereference(dev->qdisc);
1266 #ifdef CONFIG_NET_CLS_ACT
1267        skb->tc_verd = SET_TC_AT(skb->tc_verd,AT_EGRESS);
1268 #endif
1269        if (q->enqueue) {
1270                /* S'emparer de la file d'attente du peripherique */
1271                spin_lock(&dev->queue_lock);
1272
1273                rc = q->enqueue(skb, q);
1274
1275                qdisc_run(dev);
1276
1277                spin_unlock(&dev->queue_lock);
1278                rc = rc == NET_XMIT_BYPASS ? NET_XMIT_SUCCESS : rc;
1279                goto out;
1280        }
1281
1282        /* Le peripherique n'a pas de file d'attente. Cas frequent des peripheriques
1283           logiques : loopback, toutes sortes de tunnels...
1284
1285           C'est malheureusement pour cela que la protection xmit_lock est necessaire ici.
1286           (par exemple loopback et les tunnels IP sont propres, ignorant les
1287           compteurs de statistiques.)
1288           Il est possible cependant qu'ils reposent sur la protection
1289           effectuee par nous ici.
1290
1291           Verifier ceci et tirer le verrou. Ceci ne conduit pas a une impasse.
1292           Soit tirer le qdisc noqueue, ce qui est meme plus simple 8)
1293         */
1294        if (dev->flags & IFF_UP) {
1295                int cpu = smp_processor_id(); /* ok because BHs are off */
1296
1297                if (dev->xmit_lock_owner != cpu) {
1298
1299                        HARD_TX_LOCK(dev, cpu);
1300
1301                        if (!netif_queue_stopped(dev)) {
1302                                if (netdev_nit)
1303                                        dev_queue_xmit_nit(skb, dev);
```

```
1304
1305                                rc = 0;
1306                                if (!dev->hard_start_xmit(skb, dev)) {
1307                                        HARD_TX_UNLOCK(dev);
1308                                        goto out;
1309                                }
1310                        }
1311                        HARD_TX_UNLOCK(dev);
1312                        if (net_ratelimit())
1313                                printk(KERN_CRIT "Virtual device %s asks to "
1314                                       "queue packet!\n", dev->name);
1315                } else {
1316                        /* Une recursion est detectee ! C'est possible,
1317                         * malheureusement */
1318                        if (net_ratelimit())
1319                                printk(KERN_CRIT "Dead loop on virtual device "
1320                                       "%s, fix it urgently!\n", dev->name);
1321                }
1322        }
1323
1324        rc = -ENETDOWN;
1325        local_bh_enable();
1326
1327 out_kfree_skb:
1328        kfree_skb(skb);
1329        return rc;
1330 out:
1331        local_bh_enable();
1332        return rc;
1333 }
```

Autrement dit :

- On déclare un descripteur de périphérique réseau, que l'on initialise avec celui associé au descripteur de tampon passé en argument.
- On déclare une stratégie de mise en file d'attente et un code de retour.
- On essaie de linéariser le tampon dans les cas indiqués en commentaire. Si on n'y parvient pas, on libère le descripteur de tampon et on renvoie l'opposé du code d'erreur ENOMEM.

Code Linux 2.6.10

La fonction `__skb_linearize()` est définie dans le fichier `linux/net/core/dev.c`:

```
1133 /* Garder la meme tete : remplacer les donnees */
1134 int __skb_linearize(struct sk_buff *skb, int gfp_mask)
1135 {
1136        unsigned int size;
1137        u8 *data;
1138        long offset;
1139        struct skb_shared_info *ninfo;
1140        int headerlen = skb->data - skb->head;
1141        int expand = (skb->tail + skb->data_len) - skb->end;
1142
1143        if (skb_shared(skb))
1144                BUG();
1145
1146        if (expand <= 0)
1147                expand = 0;
1148
1149        size = skb->end - skb->head + expand;
1150        size = SKB_DATA_ALIGN(size);
1151        data = kmalloc(size + sizeof(struct skb_shared_info), gfp_mask);
1152        if (!data)
1153                return -ENOMEM;
```

```
1154
1155        /* Copier la chose en entier */
1156        if (skb_copy_bits(skb, -headerlen, data, headerlen + skb->len))
1157                BUG();
1158
1159        /* Initialiser shinfo */
1160        ninfo = (struct skb_shared_info*)(data + size);
1161        atomic_set(&ninfo->dataref, 1);
1162        ninfo->tso_size = skb_shinfo(skb)->tso_size;
1163        ninfo->tso_segs = skb_shinfo(skb)->tso_segs;
1164        ninfo->nr_frags = 0;
1165        ninfo->frag_list = NULL;
1166
1167        /* Le decalage entre les deux, en octets */
1168        offset = data - skb->head;
1169
1170        /* Liberer les vieilles donnees. */
1171        skb_release_data(skb);
1172
1173        skb->head = data;
1174        skb->end  = data + size;
1175
1176        /* Initialiser les nouveaux pointeurs */
1177        skb->h.raw   += offset;
1178        skb->nh.raw  += offset;
1179        skb->mac.raw += offset;
1180        skb->tail    += offset;
1181        skb->data    += offset;
1182
1183        /* Nous ne sommes plus un clone, meme si nous l'etions. */
1184        skb->cloned    = 0;
1185
1186        skb->tail     += skb->data_len;
1187        skb->data_len  = 0;
1188        return 0;
1189 }
```

La fonction en ligne `illegal_highdma()` est définie dans le fichier `linux/net/core/-dev.c`:

Code Linux 2.6.10

```
1108 #ifdef CONFIG_HIGHMEM
1109 /* En fait nous devons eliminer cette verification des que nous savons que :
1110  * 1. IOMMU est present et permet d'adresser toute la memoire.
1111  * 2. Aucune memoire haute n'existe sur cette machine.
1112  */
1113
1114 static inline int illegal_highdma(struct net_device *dev, struct sk_buff *skb)
1115 {
1116         int i;
1117
1118         if (dev->features & NETIF_F_HIGHDMA)
1119                 return 0;
1120
1121         for (i = 0; i < skb_shinfo(skb)->nr_frags; i++)
1122                 if (skb_shinfo(skb)->frags[i].page >= highmem_start_page)
1123                         return 1;
1124
1125         return 0;
1126 }
1127 #else
1128 #define illegal_highdma(dev, skb)       (0)
1129 #endif
```

– Si la somme de contrôle du paquet n'a pas été calculée et que le périphérique ne supporte pas le calcul de celle-ci pour ce protocole, on la calcule. Si on n'y parvient pas, on libère le descripteur de tampon et on renvoie l'opposé du code d'erreur ENOMEM.

Code Linux 2.6.10

La fonction skb_checksum_help() est définie dans le fichier linux/net/core/dev.c:

```
1072 /*
1073  * Invalide la somme de controle du materiel lorsque le paquet doit etre estropie, et
1074  * calcule la somme complete en chemin.
1075  */
1076 int skb_checksum_help(struct sk_buff *skb, int inward)
1077 {
1078         unsigned int csum;
1079         int ret = 0, offset = skb->h.raw - skb->data;
1080
1081         if (inward) {
1082                 skb->ip_summed = CHECKSUM_NONE;
1083                 goto out;
1084         }
1085
1086         if (skb_cloned(skb)) {
1087                 ret = pskb_expand_head(skb, 0, 0, GFP_ATOMIC);
1088                 if (ret)
1089                         goto out;
1090         }
1091
1092         if (offset > (int)skb->len)
1093                 BUG();
1094         csum = skb_checksum(skb, offset, skb->len-offset, 0);
1095
1096         offset = skb->tail - skb->h.raw;
1097         if (offset <= 0)
1098                 BUG();
1099         if (skb->csum + 2 > offset)
1100                 BUG();
1101
1102         *(u16*)(skb->h.raw + skb->csum) = csum_fold(csum);
1103         skb->ip_summed = CHECKSUM_NONE;
1104 out:
1105         return ret;
1106 }
```

– On annihile les interruptions logicielles.

– Si la stratégie de mise en file d'attente associée au descripteur de périphérique possède une fonction d'empilement : on verrouille la file d'attente du périphérique, on place le descripteur de tampon dans la file d'attente grâce à la fonction enqueue() de cette stratégie, on fait appel à la fonction qdisc_run() étudiée ci-après, on déverrouille la file d'attente et on renvoie l'une des constantes symboliques NET_XMIT_BYPASS ou NET_XMIT_SUCCESS.

Code Linux 2.6.10

Celles-ci sont définies dans le fichier linux/include/linux/netdevice.h:

```
54 #define NET_XMIT_SUCCESS        0
55 #define NET_XMIT_DROP           1       /* skb ecarte                          */
56 #define NET_XMIT_CN             2       /* notification de congestion          */
57 #define NET_XMIT_POLICED        3       /* skb est abattu par la police        */
58 #define NET_XMIT_BYPASS         4       /* le paquet ne quitte pas via dequeue ;
59                                            (usage TC seulement - dev_queue_xmit
60                                            renvoie ceci comme NET_XMIT_SUCCESS) */
```

– S'il n'y a pas de fonction d'empilement mais que le périphérique est actif, on lui indique qu'on vient de placer de nouvelles données en faisant appel tout de suite à la fonction d'émission dev->hard_start_xmit(skb, dev) spécifique à la carte.

34.1.1.2 Empilement

La fonction précédente fait référence à la méthode `enqueue()` de la stratégie de mise en file d'attente. Nous avons vu au chapitre 18 que `enqueue()` peut prendre l'une des deux valeurs `noop_enqueue()` ou `pfifo_fast_enqueue()`.

Cas de la stratégie qui ne fait rien

La fonction `noop_enqueue()` est définie dans le fichier `linux/net/sched/sch_generic.c`: Code Linux 2.6.10

```
240 /* Ordonnanceur "NOOP" : le meilleur ordonnanceur, recommande pour toutes les interfaces
241    et en toute circonstance. Il est difficile d'inventer une chose plus rapide
242    ou moins chere.
243  */
244
245 static int
246 noop_enqueue(struct sk_buff *skb, struct Qdisc * qdisc)
247 {
248         kfree_skb(skb);
249         return NET_XMIT_CN;
250 }
```

Elle libère le descripteur de tampon passé en argument et indique que la file d'attente est congestionnée.

Cas de la stratégie rapide

Plus sérieusement, la fonction `pfifo_fast_enqueue()` est définie dans le fichier `linux/net/sched/sch_generic.c`: Code Linux 2.6.10

```
306 static const u8 prio2band[TC_PRIO_MAX+1] =
307         { 1, 2, 2, 2, 1, 2, 0, 0 , 1, 1, 1, 1, 1, 1, 1, 1 };
308
309 /* Une file d'attente FIFO a 3 bandes : style ancien mais devrait etre un petit peu
310    plus rapide que la combinaison prio+fifo generique.
311  */
312
313 static int
314 pfifo_fast_enqueue(struct sk_buff *skb, struct Qdisc* qdisc)
315 {
316         struct sk_buff_head *list = qdisc_priv(qdisc);
317
318         list += prio2band[skb->priority&TC_PRIO_MAX];
319
320         if (list->qlen < qdisc->dev->tx_queue_len) {
321                 __skb_queue_tail(list, skb);
322                 qdisc->q.qlen++;
323                 qdisc->bstats.bytes += skb->len;
324                 qdisc->bstats.packets++;
325                 return 0;
326         }
327         qdisc->qstats.drops++;
328         kfree_skb(skb);
329         return NET_XMIT_DROP;
330 }
```

La constante symbolique `TC_PRIO_MAX` est définie dans le fichier `linux/include/linux/pkt_sched.h`: Code Linux 2.6.10

```
4  /* Bandes de priorite logique ne dependant pas d'un ordonnanceur de paquets particulier.
5     Chaque ordonnanceur les appliquera aux classes de trafic reelles s'il n'a pas
6     de mecanisme plus precis pour classifier les paquets.
7
8     Ces nombres n'ont pas de signification speciale, bien que leur coincidence
```

```
9     avec les valeurs IPv6 obsoletes ne soit pas accidentelle :-). Les nouvelles esquisses
10    de IPv6 ont prefere l'anarchie totale inspiree par differents groupes.
11
12    Note : TC_PRIO_BESTEFFORT ne signifie pas qu'il s'agisse de la classe la plus malheureuse
13    actuellement, en regle generale elle sera manipulee avec plus de soin qu'un
14    entonnoir ou de meme grosseur.
15  */
16
17 #define TC_PRIO_BESTEFFORT              0
18 #define TC_PRIO_FILLER                  1
19 #define TC_PRIO_BULK                    2
20 #define TC_PRIO_INTERACTIVE_BULK        4
21 #define TC_PRIO_INTERACTIVE             6
22 #define TC_PRIO_CONTROL                 7
23
24 #define TC_PRIO_MAX                     15
```

34.1.2 Deuxième étape : récupération des paquets

34.1.2.1 Fonction principale

La fonction `qdisc_run()` comporte peu de fonctionnalités. Elle appelle seulement la fonction `qdisc_restart()` tant que celle-ci retourne une valeur de renvoi supérieure ou égale à zéro, c'est-à-dire jusqu'à ce qu'il n'y ait plus de paquets dans la file d'attente, ou que le périphérique réseau n'accepte plus de paquet (ce que l'on sait grâce à la fonction `netif_queue_stopped(dev)`).

Code Linux 2.6.10

La fonction `qdisc_run()` est définie dans le fichier `linux/include/net/pkt_sched.h`:

```
231 static inline void qdisc_run(struct net_device *dev)
232 {
233         while (!netif_queue_stopped(dev) && qdisc_restart(dev) < 0)
234                 /* RIEN */;
235 }
```

Code Linux 2.6.10

La fonction `netif_queue_stopped()` est définie dans le fichier `linux/include/linux/net-device.h`:

```
630 static inline int netif_queue_stopped(const struct net_device *dev)
631 {
632         return test_bit(__LINK_STATE_XOFF, &dev->state);
633 }
```

34.1.2.2 Récupération et émission d'un paquet

La fonction `qdisc_restart()` est chargée de récupérer le paquet suivant dans la file d'attente en émission du périphérique réseau et de l'envoyer. La valeur de retour :

- est nulle lorsque la file d'attente est vide.
- est strictement positive lorsque la file d'attente n'est pas vide mais que, conformément à la stratégie de mise en file d'attente (`dev->qdisc`), aucun paquet ne peut être envoyé, par exemple parce qu'il n'a pas encore atteint son moment de transmission désiré (mise en forme du trafic).
- est strictement négative lorsque la file d'attente n'est pas vide mais que le périphérique réseau ne peut plus en ce moment accepter de paquet parce que tous les emplacements du tampon sont occupés.

Code Linux 2.6.10

La fonction `qdisc_restart()` est définie dans le fichier `linux/net/sched/sch_generic.c`:

```
82  /* Donne un coup de pied au peripherique.
```

```
83      Noter que cette procedure peut etre appelee par un minuteur chien de garde, et
84      donc que nous ne verifions pas le drapeau dev->tbusy ici.
85
86      Renvoie :  0  - la file d'attente est vide.
87                >0  - la file d'attente n'est pas vide mais ralentie.
88                <0  - la file d'attente n'est pas vide. Le peripherique est ralenti
89                      si dev->tbusy != 0.
90
91      NOTE : Appelee sous dev->queue_lock avec BH localement inhibee.
92   */
93  int qdisc_restart(struct net_device *dev)
94  {
95          struct Qdisc *q = dev->qdisc;
96          struct sk_buff *skb;
97
98          /* Retire le paquet de la file d'attente */
99          if ((skb = q->dequeue(q)) != NULL) {
100                 unsigned nolock = (dev->features & NETIF_F_LLTX);
101                 /*
102                  * Lorsque le pilote a LLTX positionne, il fait son propre verrouillage
103                  * dans start_xmit. Pas besoin d'ajouter encore des verrouillages
104                  * supplementaires. Ces verifications sont de valeur puisque
105                  * des verrous meme non congestionnes peuvent etre tres onereux.
106                  * Le pilote peut faire des trylocks comme ici aussi, en cas
107                  * de congestion de verrou, il renverra -1 et le paquet
108                  * sera replace dans la file d'attente.
109                  */
110                 if (!nolock) {
111                         if (!spin_trylock(&dev->xmit_lock)) {
112                         collision:
113                                 /* Ainsi quelqu'un a accapare le pilote. */
114
115                                 /* Cela peut etre une erreur de configuration transitoire,
116                                    lorsque hard_start_xmit() est recursif. Nous le detectons
117                                    en verifiant le proprietaire en emission et nous ecartons
118                                    le paquet lorsque le deadloop est detecte.
119                                 */
120                                 if (dev->xmit_lock_owner == smp_processor_id()) {
121                                         kfree_skb(skb);
122                                         if (net_ratelimit())
123                                                 printk(KERN_DEBUG "Dead loop on netdevice %s,
                                                               fix it urgently!\n", dev->name);
124                                         return -1;
125                                 }
126                                 __get_cpu_var(netdev_rx_stat).cpu_collision++;
127                                 goto requeue;
128                         }
129                         /* Rappelons que nous avons accapare le pilote. */
130                         dev->xmit_lock_owner = smp_processor_id();
131                 }
132
133                 {
134                         /* Et relachons la file d'attente */
135                         spin_unlock(&dev->queue_lock);
136
137                         if (!netif_queue_stopped(dev)) {
138                                 int ret;
139                                 if (netdev_nit)
140                                         dev_queue_xmit_nit(skb, dev);
141
142                                 ret = dev->hard_start_xmit(skb, dev);
```

```
143                                 if (ret == NETDEV_TX_OK) {
144                                         if (!nolock) {
145                                                 dev->xmit_lock_owner = -1;
146                                                 spin_unlock(&dev->xmit_lock);
147                                         }
148                                         spin_lock(&dev->queue_lock);
149                                         return -1;
150                                 }
151                                 if (ret == NETDEV_TX_LOCKED && nolock) {
152                                         spin_lock(&dev->queue_lock);
153                                         goto collision;
154                                 }
155                         }
156
157                         /* NETDEV_TX_BUSY - nous avons besoin de le replacer dans la file
                               d'attente */
158                         /* Liberons le pilote */
159                         if (!nolock) {
160                                 dev->xmit_lock_owner = -1;
161                                 spin_unlock(&dev->xmit_lock);
162                         }
163                         spin_lock(&dev->queue_lock);
164                         q = dev->qdisc;
165                 }
166
167                 /* Le peripherique nous donne un coup de pied :(
168                    Ceci est possible dans trois cas :
169
170                    0. le peripherique est verrouille
171                    1. fastroute est active
172                    2. le peripherique ne peut pas determiner un etat occupe
173                       avant le debut de la transmission (par exemple par telephone)
174                    3. le peripherique est bogue (ppp)
175                  */
176
177 requeue:
178                 q->ops->requeue(skb, q);
179                 netif_schedule(dev);
180                 return 1;
181         }
182         return q->q.qlen;
183 }
```

Remarquons que la référence à `dev->tbusy` dans le commentaire n'a plus lieu d'etre puisque ce champ n'existe plus depuis la version 2.6.0 du noyau.
Autrement dit :

- On déclare une stratégie de mise en file d'attente, que l'on initialise avec celle associée au descripteur de périphérique réseau passé en argument.
- On déclare un descripteur de tampon de socket.
- On essaie de récupérer un descripteur de tampon dans la file d'attente.
- Si on y parvient et si le pilote n'a pas déjà verrouillé, on essaie de positionner le verrou rotatif en émission du descripteur de périphérique. Si on n'y parvient pas, c'est qu'un microprocesseur diffuse un autre paquet *via* ce périphérique de réseau :
  - S'il s'agit du même microprocesseur, on rejette le paquet : on libère le descripteur de tampon, on affiche un message noyau et on renvoie -1.
  - S'il s'agit d'un autre microprocesseur, on incrémente le nombre de collisions liées à des problèmes de microprocesseurs, le tampon est replacé dans la file d'attente, l'envoi

du paquet est différé, ce qui est effectué en faisant appel à la fonction `netif_schedule(dev)`, étudiée ci-dessous et on renvoie 1.

La variable `netdev_rx_stat` est déclarée dans le fichier `linux/net/core/dev.c`: Code Linux 2.6.10

```
1351 DEFINE_PER_CPU(struct netif_rx_stats, netdev_rx_stat) = { 0, };
```

Le type `netif_rx_stats` est déclaré dans le fichier `linux/include/linux/netdevice.h`: Code Linux 2.6.10

```
162 struct netif_rx_stats
163 {
164         unsigned total;
165         unsigned dropped;
166         unsigned time_squeeze;
167         unsigned throttled;
168         unsigned fastroute_hit;
169         unsigned fastroute_success;
170         unsigned fastroute_defer;
171         unsigned fastroute_deferred_out;
172         unsigned fastroute_latency_reduction;
173         unsigned cpu_collision;
174 };
```

Si on y parvient, on spécifie au descripteur de périphérique le microprocesseur qui a verrouilé la file d'attente.

- On déverrouille la file d'attente du descripteur de périphérique.
- Si le périphérique réseau accepte un paquet :
  - Si la variable `netdev_nit` est positionnée, on fait appel à la fonction `dev_queue_xmit_nit(skb, dev)`, étudiée ci-après, d'envoi à toutes les cartes.

    La variable `netdev_nit` est déclarée dans le fichier `linux/net/core/dev.c`: Code Linux 2.6.10

    ```
    236 /*
    237  *      Pour l'efficacite
    238  */
    239
    240 int netdev_nit;
    ```

  - On fait appel à la fonction spécifique à la carte réseau `dev->hard_start_xmit(skb, dev)` pour envoyer le paquet.

    Les constantes symboliques `NETDEV_TX_OK` et `NETDEV_TX_LOCKED` sont définies dans le fichier `linux/include/linux/netdevice.h`: Code Linux 2.6.10

    ```
    76 /* Codes de retour pour l'emission de la part d'un pilote */
    77 #define NETDEV_TX_OK 0          /* Le pilote a pris soin du paquet */
    78 #define NETDEV_TX_BUSY 1        /* Le chamin en tx du pilote etait occupe */
    79 #define NETDEV_TX_LOCKED -1     /* Le verrou en tx du pilote etait deja tire */
    ```

  - Si le pilote a transmis le paquet au périphérique, on indique éventuellement qu'aucun microprocesseur n'a accaparé le périphérique et on déverrouille en émission, on verrouille la file d'attente et on renvoie -1.
  - Si le verrou en émission du pilote était déjà tiré, on verrouille la file d'attente et on continue comme dans le cas d'une collision avec un autre microprocesseur.
- Si le périphérique réseau n'accepte pas de nouvelle trame pour le moment, on indique éventuellement qu'aucun microprocesseur n'a accaparé le périphérique et on déverrouille en émission, on verrouille la file d'attente et on initialise la stratégie de mise en file d'attente avec celle du descripteur de périphérique passé en argument.

– Si on ne parvient pas à récupérer de paquet dans la file d'attente, on renvoie la longueur de la file d'attente.

En conclusion, dans le cas d'un envoi normal, il est fait appel à la fonction spécifique à la carte réseau pour envoyer le paquet et on se prépare à envoyer le paquet suivant.

34.1.2.3 Fonction de dépilement

La fonction précédente fait référence à la méthode `dequeue()` de la stratégie de file d'attente. Nous avons vu au chapitre 18 que `enqueue()` peut prendre l'une des deux valeurs `noop_dequeue()` ou `pfifo_fast_dequeue()`.

Stratégie qui ne fait rien

Code Linux 2.6.10

La fonction `noop_dequeue()` est définie dans le fichier `linux/net/sched/sch_generic.c`:

```
252 static struct sk_buff *
253 noop_dequeue(struct Qdisc * qdisc)
254 {
255         return NULL;
256 }
```

Stratégie rapide

Code Linux 2.6.10

La fonction `pfifo_fast_dequeue()` est définie dans le fichier `linux/net/sched/sch_generic.c`:

```
332 static struct sk_buff *
333 pfifo_fast_dequeue(struct Qdisc* qdisc)
334 {
335         int prio;
336         struct sk_buff_head *list = qdisc_priv(qdisc);
337         struct sk_buff *skb;
338
339         for (prio = 0; prio < 3; prio++, list++) {
340                 skb = __skb_dequeue(list);
341                 if (skb) {
342                         qdisc->q.qlen--;
343                         return skb;
344                 }
345         }
346         return NULL;
347 }
```

34.1.2.4 Fonction de réempilement

La fonction précédente fait également référence à la méthode `requeue()` de la stratégie de file d'attente. Nous avons vu au chapitre 18 que `requeue()` peut prendre l'une des deux valeurs `noop_requeue()` ou `pfifo_fast_requeue()`.

Stratégie qui ne fait rien

Code Linux 2.6.10

La fonction `noop_requeue()` est définie dans le fichier `linux/net/sched/sch_generic.c`:

```
258 static int
259 noop_requeue(struct sk_buff *skb, struct Qdisc* qdisc)
260 {
261         if (net_ratelimit())
262                 printk(KERN_DEBUG "%s deferred output. It is buggy.\n", skb->dev->name);
263         kfree_skb(skb);
264         return NET_XMIT_CN;
265 }
```

Stratégie rapide

La fonction `pfifo_fast_requeue()` est définie dans le fichier `linux/net/sched/sch_generic.c` :

Code Linux 2.6.10

```
349 static int
350 pfifo_fast_requeue(struct sk_buff *skb, struct Qdisc* qdisc)
351 {
352         struct sk_buff_head *list = qdisc_priv(qdisc);
353
354         list += prio2band[skb->priority&TC_PRIO_MAX];
355
356         __skb_queue_head(list, skb);
357         qdisc->q.qlen++;
358         qdisc->qstats.requeues++;
359         return 0;
360 }
```

34.1.3 Troisième étape : envoi

Outre l'envoi direct au périphérique, nous avons vu que l'étape précédente fait appel à l'envoi différé et à l'envoi à toutes les cartes.

34.1.3.1 Envoi différé

La fonction `netif_schedule()` est définie dans le fichier `linux/include/linux/netdevice.h` :

Code Linux 2.6.10

```
585 static inline void __netif_schedule(struct net_device *dev)
586 {
587         if (!test_and_set_bit(__LINK_STATE_SCHED, &dev->state)) {
588                 unsigned long flags;
589                 struct softnet_data *sd;
590
591                 local_irq_save(flags);
592                 sd = &__get_cpu_var(softnet_data);
593                 dev->next_sched = sd->output_queue;
594                 sd->output_queue = dev;
595                 raise_softirq_irqoff(NET_TX_SOFTIRQ);
596                 local_irq_restore(flags);
597         }
598 }
599
600 static inline void netif_schedule(struct net_device *dev)
601 {
602         if (!test_bit(__LINK_STATE_XOFF, &dev->state))
603                 __netif_schedule(dev);
604 }
```

Autrement dit si le périphérique n'est pas désactivé, on fait appel à la fonction `__netif_schedule()` définie juste avant dans le même fichier. Celle-ci :

- ne fait rien si le périphérique est déjà dans l'état "ordonnancement" ;
- sinon on déclare un vecteur de drapeaux et une file d'attente en émission de microprocesseur ;
- on inhibe les interruptions ;
- on instantie la file d'attente avec celle du microprocesseur en cours ;
- on ajoute le descripteur de périphérique passé en argument à cette file d'attente ;
- on lève `NET_TX_SOFTIRQ` ;
- et on restaure les interruptions matérielles.

34.1.3.2 Envoi à toutes les cartes

Code Linux 2.6.10

La fonction `dev_queue_xmit_nit()` est définie dans le fichier `linux/net/core/dev.c`:

```
1027 /*
1028  *      Routine de support. Envoie les trames de sortie a toutes les cartes
1029  *      reseau actuellement actives.
1030  */
1031
1032 void dev_queue_xmit_nit(struct sk_buff *skb, struct net_device *dev)
1033 {
1034         struct packet_type *ptype;
1035         net_timestamp(&skb->stamp);
1036
1037         rcu_read_lock();
1038         list_for_each_entry_rcu(ptype, &ptype_all, list) {
1039                 /* Ne jamais renvoyer les paquets a la socket
1040                  * dont ils proviennent - MvS (miquels@drinkel.ow.org)
1041                  */
1042                 if ((ptype->dev == dev || !ptype->dev) &&
1043                     (ptype->af_packet_priv == NULL ||
1044                      (struct sock *)ptype->af_packet_priv != skb->sk)) {
1045                         struct sk_buff *skb2= skb_clone(skb, GFP_ATOMIC);
1046                         if (!skb2)
1047                                 break;
1048
1049                         /* skb->nh doit etre correctement positionne
1050                            par l'emetteur, ce second enonce est donc
1051                            juste une protection contre les protocoles bogues.
1052                          */
1053                         skb2->mac.raw = skb2->data;
1054
1055                         if (skb2->nh.raw < skb2->data ||
1056                             skb2->nh.raw > skb2->tail) {
1057                                 if (net_ratelimit())
1058                                         printk(KERN_CRIT "protocol %04x is "
1059                                                "buggy, dev %s\n",
1060                                                skb2->protocol, dev->name);
1061                                 skb2->nh.raw = skb2->data;
1062                         }
1063
1064                         skb2->h.raw = skb2->nh.raw;
1065                         skb2->pkt_type = PACKET_OUTGOING;
1066                         ptype->func(skb2, skb->dev, ptype);
1067                 }
1068         }
1069         rcu_read_unlock();
1070 }
```

Autrement dit :

- On déclare un type de paquets.
- On renseigne le champ estampille du descripteur de tampon passé en argument avec l'heure en cours.

Code Linux 2.6.10

La fonction `net_timestamp()` est définie dans le fichier `linux/net/core/dev.c`:

```
1017 static inline void net_timestamp(struct timeval *stamp)
1018 {
1019         if (atomic_read(&netstamp_needed))
1020                 do_gettimeofday(stamp);
1021         else {
1022                 stamp->tv_sec = 0;
```

```
1023                    stamp->tv_usec = 0;
1024            }
1025 }
```

- On verrouille la mise à jour en écriture.
- On parcourt la liste de tous les types de paquets à la recherche des types dont le champ descripteur de périphérique est égal à celui passé en argument ou non spécifié et dont le champ type de paquet n'est pas spécifié ou correspond à celui du descripteur de tampon passé en argument. Pour chacun de ces types de paquet :
  - On essaie de cloner le descripteur de tampon passé en argument. Si on n'y parvient pas, on passe au type suivant.
  - On renseigne certains champs de ce descripteur de tampon : le champ en-tête matériel (brut) avec le début des données ; si le champ en-tête réseau brut n'est pas bien positionné (ce qui ne devrait pas être le cas comme l'indique le commentaire), on renseigne également ce champ avec le début des données ; le champ en-tête transport (brut) avec l'en-tête réseau ; le champ type de paquet avec "paquet sortant".
  - On fait appel à la fonction spécifique de ce type de paquet pour l'envoi proprement dit.
- On déverrouille la mise à jour en écriture.

34.2 Transmission d'un paquet à la carte : partie spécifique à la carte

34.2.1 Étude générale

Les fonctions précédentes font appel à la fonction `dev->hard_start_xmit(skb, dev)` spécifique à la carte réseau. Celle-ci est chargée de copier la zone des données du tampon de socket dans un emplacement interne du périphérique et de commander le début de la transmission. Si la copie réussit, on part également du principe que l'émission réussira. Dans ce cas, `hard_start_xmit()` doit revenir à l'état antérieur avec la valeur de renvoi 0. Dans le cas contraire, un 1 sera envoyé pour que le noyau sache que l'émission du paquet n'a pas pu aboutir.

Lors de la remise des paquets, on peut distinguer deux techniques de copie entre système d'exploitation et adaptateur réseau :

- Les cartes les plus anciennes, comme la carte 3c501 de 3Com, comportent une mémoire tampon interne en prévision de l'émission de paquets sur l'adaptateur. Ainsi, le noyau ne peut-il remettre qu'un seul paquet à l'adaptateur. Il copie alors le paquet dans un tampon libre à l'identique de l'adaptateur et le noyau peut supprimer le tampon de socket correspondant.
- Les adaptateurs réseau plus récents suivent une autre voie. Le pilote gère un tampon circulaire de 16 à 64 pointeurs de tampon de socket. Lorsqu'il s'agit d'émettre un paquet, le tampon de socket correspondant est placé dans cet anneau et un pointeur sur la zone des données est transmis à l'adaptateur de réseau. Le tampon de socket reste ensuite dans le tampon circulaire jusqu'à ce que l'adaptateur ait signalé son émission par une interruption. Le tampon de socket est alors retiré et libéré du tampon circulaire.

  En cas de succès, le tampon de socket n'est plus nécessaire et peut être libéré. En cas d'échec, il ne faut pas toucher au tampon de socket parce que le noyau tentera probablement de l'émettre à nouveau.

34.2.2 Cas de la carte 3Com 501 : Mise en file d'attente

Nous avons vu au chapitre 17 que la fonction `hard_start_xmit()` spécifique à la carte Ethernet 3Com 501 s'appelle `el_start_xmit()`. Celle-ci est définie dans le fichier `linux/drivers/net/3c501.c`:

Code Linux 2.6.10

```
390 /**
391  * el_start_xmit :
392  * @skb : Le paquet qui est mis en file d'attente pour etre envoye
393  * @dev : La carte 3c501 dans laquelle nous voulons le deposer
394  *
395  * Essaie d'envoyer un paquet a une carte 3c501. Il y a certaines entourloupettes
396  * interessantes ici puisque la 3c501 est extrement vieille et donc une piece de
397  * technologie stupide.
398  *
399  * Si nous traitons une interruption sur l'autre CPU, nous ne pouvons pas charger un paquet
400  * alors que nous pouvons encore essayer de recuperer le dernier tampon de paquet en RX.
401  *
402  * Lorsqu'une emission a depasse le delai, nous vidons la carte du mode de controle
403  * et nous redemarrons. Il arrive souvent que ce ne soit pas pire.
404  *
405  * Nous evitons de detenir les verrous tournants lorsque le paquet est en train de
406  * passer dans la carte. Le peripherique est tres lent, et son mode DMA est encore plus
407  * lent. Si nous detenions le verrou lors du transfert des 1500 octets dans le controleur,
408  * nous ecarterions beaucoup trop de caracteres du port serie. Ceci exige que nous
409  * fassions un verrouillage supplementaire, mais nous n'avons pas reellement le choix.
410  */
411
412 static int el_start_xmit(struct sk_buff *skb, struct net_device *dev)
413 {
414         struct net_local *lp = netdev_priv(dev);
415         int ioaddr = dev->base_addr;
416         unsigned long flags;
417
418         /*
419          *      Eviter que ne surviennent des interruptions entre nos petites emissions et le
420          *      mode ou le peripherique suppose que la transmission est un indicateur fidele
421          *      de l'etat de la carte
422          */
423
424         spin_lock_irqsave(&lp->lock, flags);
425
426         /*
427          *      Eviter les conflits de retransmission fondes sur le delai.
428          */
429
430         netif_stop_queue(dev);
431
432         do
433         {
434                 int len = skb->len;
435                 int pad = 0;
436                 int gp_start;
437                 unsigned char *buf = skb->data;
438
439                 if (len < ETH_ZLEN)
440                         pad = ETH_ZLEN - len;
441
442                 gp_start = 0x800 - ( len + pad );
443
444                 lp->tx_pkt_start = gp_start;
445                 lp->collisions = 0;
446
```

```
447                 lp->stats.tx_bytes += skb->len;
448
449                 /*
450                  *      Le mode commande avec le statut efface devrait [en theorie]
451                  *      signifier qu'il n'y a plus d'interruptions en attente sur la carte.
452                  */
453
454                 outb_p(AX_SYS, AX_CMD);
455                 inb_p(RX_STATUS);
456                 inb_p(TX_STATUS);
457
458                 lp->loading = 1;
459                 lp->txing = 1;
460
461                 /*
462                  *      Retourne aux interruptions cependant que nous passons une apres-midi
463                  *      charmante en train de charger des octets dans la carte
464                  */
465
466                 spin_unlock_irqrestore(&lp->lock, flags);
467
468                 outw(0x00, RX_BUF_CLR);         /* Met la zone du paquet en rx a 0. */
469                 outw(gp_start, GP_LOW);         /* but - le paquet sera charge au
                                                       debut du tampon */
470                 outsb(DATAPORT,buf,len);        /* charge le tampon (chose usuelle, chaque
                                                       octets incremente le pointeur) */
471                 if (pad) {
472                         while(pad--)            /* Fin du tampon rempli avec des zeros */
473                                 outb(0, DATAPORT);
474                 }
475                 outw(gp_start, GP_LOW);         /* la carte reutilise le meme registre */
476
477                 if(lp->loading != 2)
478                 {
479                         outb(AX_XMIT, AX_CMD);          /* feu... envoyer l'emission.  */
480                         lp->loading=0;
481                         dev->trans_start = jiffies;
482                         if (el_debug > 2)
483                                 printk(KERN_DEBUG " queued xmit.\n");
484                         dev_kfree_skb (skb);
485                         return 0;
486                 }
487                 /* Une reception recoit notre chargement en depit de nos meilleurs efforts */
488                 if(el_debug>2)
489                         printk(KERN_DEBUG "%s: burped during tx load.\n", dev->name);
490                 spin_lock_irqsave(&lp->lock, flags);
491         }
492         while(1);
493
494 }
```

Autrement dit :

- On déclare une structure de réseau locale, que l'on initialise avec la partie privée du descripteur de périphérique réseau passé en argument.
- On déclare un port d'entrée-sortie, que l'on initialise avec celui fourni par le descripteur de périphérique réseau passé en argument.
- On déclare un vecteur de drapeaux.
- On sauvegarde les registres IRQ et on inhibe les interruptions.
- On évite les conflits de retransmission en arrêtant la file d'attente du périphérique.

Code Linux 2.6.10

La fonction `netif_stop_queue()` est définie dans le fichier `linux/include/linux/netdevice.h`:

```
621 static inline void netif_stop_queue(struct net_device *dev)
622 {
623 #ifdef CONFIG_NETPOLL_TRAP
624         if (netpoll_trap())
625                 return;
626 #endif
627         set_bit(__LINK_STATE_XOFF, &dev->state);
628 }
```

- On entre dans une boucle (*a priori* infinie) de scrutation:
  - On déclare une taille, que l'on initialise avec la taille du tampon de socket passé en argument.
  - On déclare une valeur de rembourrage, que l'on initialise à zéro.
  - On déclare une valeur entière `gp_start`, qui est la valeur du pointeur `GP` de la carte.
  - On déclare une adresse de chaîne de caractères, que l'on initialise avec celle des données du tampon de socket passé en argument.
  - Si la taille est strictement inférieure à celle d'une trame Ethernet minimum, on calcule le nombre de caractères de remplissage.
  - On calcule la valeur de `GP` à transmettre à la carte.
  - On initialise la structure locale de réseau: valeur de `GP`, nombre de collisions nul, le nombre d'octets qui restent à transmettre.
  - On réinitialise le mode de commande de la carte.
  - On indique à la structure de réseau locale que l'on est en train de charger des données en vue d'une émission.
  - On remet les interruptions en service.
  - On réinitialise le pointeur de la zone de réception de la carte réseau en la mettant à zéro.
  - On indique à la carte l'emplacement de mémoire vive depuis lequel charger le paquet à transmettre.
  - On transfère le paquet de la mémoire vive vers la carte réseau, en ajoutant si besoin est des caractères de remplissage.
  - On positionne le pointeur `GP` de la carte.
  - Si le niveau de chargement est différent de `2`:
    - On envoie à la carte l'ordre d'émettre.
    - On passe au niveau de chargement 0.
    - On indique au descripteur de périphérique l'heure de la dernière transmission.
    - Si le niveau de débogage est strictement supérieur à 2, on affiche un message noyau.
    - On libère le descripteur de tampon.
    - On renvoie 0 (et donc on sort de la boucle infinie).
  - Si on n'est pas parvenu à émettre: si le niveau de débogage est strictement supérieur à 2, on affiche un message noyau, on verrouille l'interruption et on recommence.

34.3 Réaction à l'envoi

Une fois que le paquet a été transmis à la file d'attente en émission de la carte, on attend une réaction à cet envoi : soit une interruption spécifiant que la trame a été envoyée avec succès, soit la fin du délai.

34.3.1 Réponse du gestionnaire d'interruption

Le gestionnaire d'interruption dépend de la carte réseau. Nous allons nous intéresser, comme d'habitude, à celui de la carte 3Com 501. Nous avons commencer à étudier cette routine dans le chapitre consacré à la réception. Reprenons notre commentaire de la routine d'interruption dans le cas de l'émission :

Code Linux 2.6.10

```
519 static irqreturn_t el_interrupt(int irq, void *dev_id, struct pt_regs *regs)
520 {
[...]
548         if (lp->txing)
549         {
550
551                 /*
552                  *      La carte est en mode emission. Peut etre en train de charger.
553                  *      Si nous sommes en train de charger, nous devrions avoir obtenu ceci.
554                  */
555
556                 int txsr = inb(TX_STATUS);
557
558                 if(lp->loading==1)
559                 {
560                         if(el_debug > 2)
561                         {
562                                 printk(KERN_DEBUG "%s: Interrupt while loading [", dev->name);
563                                 printk(KERN_DEBUG " txsr=%02x gp=%04x rp=%04x]\n", txsr,
                                        inw(GP_LOW),inw(RX_LOW));
564                         }
565                         lp->loading=2;          /* Force a charger a nouveau */
566                         spin_unlock(&lp->lock);
567                         goto out;
568                 }
569
570                 if (el_debug > 6)
571                         printk(KERN_DEBUG " txsr=%02x gp=%04x rp=%04x", txsr,
                               inw(GP_LOW),inw(RX_LOW));
572
573                 if ((axsr & 0x80) && (txsr & TX_READY) == 0)
574                 {
575                         /*
576                          *      A RECTIFIER : y a-t-il une logique si on garde sur essai ou
577                          *      si on reinitialise immediatement ?
578                          */
579                         if(el_debug>1)
580                                 printk(KERN_DEBUG "%s: Unusual interrupt during Tx,
                                        txsr=%02x axsr=%02x"
581                                         " gp=%03x rp=%03x.\n", dev->name, txsr, axsr,
582                         inw(ioaddr + EL1_DATAPTR), inw(ioaddr + EL1_RXPTR));
583                         lp->txing = 0;
584                         netif_wake_queue(dev);
585                 }
586                 else if (txsr & TX_16COLLISIONS)
587                 {
588                         /*
```

```
589                          *      Delai depasse
590                          */
591                         if (el_debug)
592                                 printk (KERN_DEBUG "%s: Transmit failed 16 times,
                                            Ethernet jammed?\n",dev->name);
593                         outb(AX_SYS, AX_CMD);
594                         lp->txing = 0;
595                         lp->stats.tx_aborted_errors++;
596                         netif_wake_queue(dev);
597                 }
598                 else if (txsr & TX_COLLISION)
599                 {
600                         /*
601                          *      Reenclencher l'emission.
602                          */
603
604                         if (el_debug > 6)
605                                 printk(KERN_DEBUG " retransmitting after a collision.\n");
606                         /*
607                          *      Pauvre petite puce qui ne peut pas reinitialiser son propre
                                    pointeur de debut
608                          */
609
610                         outb(AX_SYS, AX_CMD);
611                         outw(lp->tx_pkt_start, GP_LOW);
612                         outb(AX_XMIT, AX_CMD);
613                         lp->stats.collisions++;
614                         spin_unlock(&lp->lock);
615                         goto out;
616                 }
617                 else
618                 {
619                         /*
620                          *      Ca a marche... nous allons maintenant passer au mode reception
621                          */
622                         lp->stats.tx_packets++;
623                         if (el_debug > 6)
624                                 printk(KERN_DEBUG " Tx succeeded %s\n",
625                                         (txsr & TX_RDY) ? "." : "but tx is busy!");
626                         /*
627                          *      Ceci est sur car l'interruption est atomique a l'egard
                                    d'elle-meme.
628                          */
629
630                         lp->txing = 0;
631                         netif_wake_queue(dev);  /* Au cas ou il y a encore a emettre */
632                 }
633         }
[...]
686 }
```

Autrement dit :

- Le gestionnaire d'interruption récupère le contenu du registre de statut d'émission.
- Si la structure locale de réseau indique qu'on est en train de recevoir des données : si le niveau de débogage est strictement supérieur à 2, on affiche un message noyau ; on force la continuation de la réception des données, on déverrouille la structure locale de réseau et on renvoie `IRQ_HANDLED`.
- Si le niveau de débogage est strictement supérieur à 6, on affiche un message noyau sur le contenu des registres.

– Si l'un des quatre bits forts du registre ASR est non nul et si le périphérique est prêt à transmettre : si le niveau de débogage est strictement supérieur à 1, on affiche un message noyau spécifiant qu'il s'agit d'une interruption non usuelle ; on change le champ de la structure locale de réseau et on réveille la file d'attente.

La fonction netif_wake_queue() est définie dans le fichier linux/include/linux/-netdevice.h: Code Linux 2.6.10

```
611 static inline void netif_wake_queue(struct net_device *dev)
612 {
613 #ifdef CONFIG_NETPOLL_TRAP
614         if (netpoll_trap())
615                 return;
616 #endif
617         if (test_and_clear_bit(__LINK_STATE_XOFF, &dev->state))
618                 __netif_schedule(dev);
619 }
```

– Si le nombre de 16 collisions est atteint : si le niveau de débogage est non nul, on affiche un message noyau ; on indique à la carte de ne plus réessayer de transmettre les données ; on spécifie à la structure locale de réseau qu'on n'est plus dans l'état d'émission ; on incrémente le nombre d'abandons en transmission dans les informations statistiques de la structure locale de réseau et on réveille la file d'attente.
– Lorsqu'il y a une collision : si le niveau de débogage est strictement supérieur à 6, on affiche un message noyau ; on indique à la carte du périphérique réseau de retransmettre les données ; on met à jour les statistiques de la structure locale de réseau sur le nombre de collisions ; on déverrouille la structure locale de réseau et on renvoie IRQ_HANDLED.
– Sinon, c'est qu'on a réussi à transmettre les données : on met à jour les statistiques de la structure locale de réseau sur le nombre de paquets émis ; si le niveau de débogage est strictement supérieur à 6, on affiche un message noyau ; on spécifie à la structure locale de réseau qu'on n'est plus en train de transmettre et on réveille la file d'attente.

34.3.2 Action lorsque le délai est écoulé

34.3.2.1 Fonction indépendante de la carte

Lorsque l'heure système dépasse l'heure trans_start du périphérique de la durée du délai watchdog_timeo, la couche réseau fait appel à la méthode tx_timeout() du pilote de périphérique, comme indiqué par la méthode dev_watchdog(), définie dans le fichier linux/net/sched/-sch_generic.c: Code Linux 2.6.10

```
185 static void dev_watchdog(unsigned long arg)
186 {
187         struct net_device *dev = (struct net_device *)arg;
188
189         spin_lock(&dev->xmit_lock);
190         if (dev->qdisc != &noop_qdisc) {
191                 if (netif_device_present(dev) &&
192                     netif_running(dev) &&
193                     netif_carrier_ok(dev)) {
194                         if (netif_queue_stopped(dev) &&
195                             (jiffies - dev->trans_start) > dev->watchdog_timeo) {
196                                 printk(KERN_INFO "NETDEV WATCHDOG: %s: transmit timed out\n",
                                               dev->name);
197                                 dev->tx_timeout(dev);
198                         }
199                         if (!mod_timer(&dev->watchdog_timer, jiffies + dev->watchdog_timeo))
200                                 dev_hold(dev);
```

```
201                 }
202         }
203         spin_unlock(&dev->xmit_lock);
204
205         dev_put(dev);
206 }
```

34.3.2.2 Cas de la carte 3Com 501

Code Linux 2.6.10

Comme nous l'avons vu au chapitre 17, la fonction `tx_timeout()` spécifique à la carte 3Com 501 s'appelle `el_timeout()`. Celle-ci est définie dans le fichier `linux/drivers/net/3c501.c`:

```
363 /**
364  * el_timeout :
365  * @dev : La carte 3c501 dont le delai est ecoule
366  *
367  * Essaie de redemarrer la carte. Fondamentalement, ceci est un melange d'extreme
368  * violence et de priere
369  *
370  */
371
372 static void el_timeout(struct net_device *dev)
373 {
374         struct net_local *lp = netdev_priv(dev);
375         int ioaddr = dev->base_addr;
376
377         if (el_debug)
378                 printk (KERN_DEBUG "%s: transmit timed out, txsr %#2x axsr=%02x rxsr=%02x.\n",
379                         dev->name, inb(TX_STATUS), inb(AX_STATUS), inb(RX_STATUS));
380         lp->stats.tx_errors++;
381         outb(TX_NORM, TX_CMD);
382         outb(RX_NORM, RX_CMD);
383         outb(AX_OFF, AX_CMD);   /* Juste declencher une fausse interruption. */
384         outb(AX_RX, AX_CMD);    /* Controle aux : irq et reception activees  */
385         lp->txing = 0;          /* Revenir a la RX                           */
386         netif_wake_queue(dev);
387 }
```

Autrement dit :

- on déclare une structure de réseau locale, que l'on initialise avec la partie privée du descripteur de périphérique passé en argument ;
- on déclare une adresse d'entrée-sortie, que l'on initialise avec celle fournie par le descripteur de périphérique passé en argument ;
- si `el_debug` est positionné, on affiche un message noyau ;
- on incrémente le nombre d'erreurs dans les informations statistiques de la structure locale de réseau ;
- on redémarre la carte, ce qui est extrêmement violent comme indiqué en commentaire ;
- on réveille la file d'attente.

Chapitre 35

Désactivation et retrait d'un pilote de périphérique réseau

Nous avons retardé jusqu'à ce chapitre l'étude de l'implémentation de la désactivation et du retrait d'un pilote de périphérique non pas tant par esprit de symétrie mais pour ne pas surcharger les chapitres 18 et 15.

35.1 Désactivation

La désactivation d'un périphérique réseau s'effectue grâce à la fonction `dev_close()`, qui fait appel à une fonction spécifique à la carte réseau.

35.1.1 Cas général

35.1.1.1 Fonction de fermeture

Code Linux 2.6.10

La fonction `dev_close()` est définie dans le fichier `linux/net/core/dev.c`:

```
875 /**
876  *      dev_close - desactive une interface.
877  *      @dev : peripherique a desactiver
878  *
879  *      Cette fonction fait passer un peripherique de l'etat actif a l'etat non actif. Un
880  *      %NETDEV_GOING_DOWN est envoye a la chaine de notification netdev. Le peripherique
881  *      est alors desactive puis un %NETDEV_DOWN est envoye a la chaine de
882  *      notification.
883  */
884 int dev_close(struct net_device *dev)
885 {
886         if (!(dev->flags & IFF_UP))
887                 return 0;
888
889         /*
890          *      Dire aux gens que nous sommes desactives, qu'ils peuvent se
891          *      preparer a notre mort, lorsque le peripherique est encore en train d'agir.
892          */
893         notifier_call_chain(&netdev_chain, NETDEV_GOING_DOWN, dev);
894
895         dev_deactivate(dev);
896
897         clear_bit(__LINK_STATE_START, &dev->state);
898
899         /* Synchroniser l'election dans l'ordonnanceur. Nous ne pouvons pas atteindre la liste
900          * d'election, elle peut meme etre sur une cpu differente. Donc juste effacer
901          * netif_running(), et attendre que l'election arrive reellement. Actuellement,
902          * la meilleure place pour ceci est a l'interieur de dev->stop() apres que le
903          * peripherique ait arrete sa machine irq, mais ceci necessite plusieurs
            changements dans les peripheriques. */
904
905         smp_mb__after_clear_bit(); /* Commit netif_running(). */
906         while (test_bit(__LINK_STATE_RX_SCHED, &dev->state)) {
907                 /* Ne pas se precipiter. */
908                 current->state = TASK_INTERRUPTIBLE;
909                 schedule_timeout(1);
910         }
911
912         /*
913          *      Appeler la fonction close specifique au peripherique. Ceci n'echouera pas.
914          *      Seulement si le peripherique est actif.
915          *
916          *      Nous permettons qu'elle soit appelee meme apres un evenement
917          *      de DETACHEMENT a chaud.
918          */
919         if (dev->stop)
920                 dev->stop(dev);
921
922         /*
923          *      Le peripherique est maintenant desactive.
924          */
```

```
925
926         dev->flags &= ~IFF_UP;
927
928         /*
929          * Dire aux gens que nous sommes desactive
930          */
931         notifier_call_chain(&netdev_chain, NETDEV_DOWN, dev);
932
933         return 0;
934 }
```

Autrement dit :

- Si le périphérique n'est pas ouvert, on renvoie 0.
- Toutes les méthodes de la chaîne de notification `netdev_chain` sont informées de la désactivation imminente du périphérique et peuvent agir en conséquence.
- On désactive le périphérique.
- On met à zéro le bit `__LINK_STATE_START` de l'état du descripteur de périphérique.
- On synchronise la liste d'élection et on change l'état du processus en cours.

  La fonction générale `schedule_timeout()` est définie dans le fichier `linux/kernel/-timer.c` :

Code Linux 2.6.10

```
1087 /**
1088  * schedule_timeout - s'assoupir jusqu'a la fin du delai
1089  * @timeout : valeur du delai en jiffies
1090  *
1091  * Fait que la tache en cours s'assoupisse jusqu'a ce que @timeout jiffies se soient
1092  * ecoulees. La routine rendra la main immediatement a moins que
1093  * l'etat de la tache en cours ait ete positionne (voir set_current_state()).
1094  *
1095  * Vous pouvez positionner l'etat de la tache comme suit -
1096  *
1097  * %TASK_UNINTERRUPTIBLE - on est sur qu'au moins @timeout jiffies s'ecoulent
1098  * avant que la routine rende la main. La routine renverra 0.
1099  *
1100  * %TASK_INTERRUPTIBLE - la routine peut rendre la main plus tot si un signal est
1101  * delivre a la tache en cours. Dans ce cas, le temps restant
1102  * en jiffies sera renvoye ou 0 si le minuteur a termine.
1103  *
1104  * On est sur que l'etat de la tache en cours est TASK_RUNNING lorsque cette
1105  * routine rend la main.
1106  *
1107  * Specifier une valeur @timeout de %MAX_SCHEDULE_TIMEOUT appellera l'ordonnanceur
1108  * du CPU sans limite sur le delai. Dans ce cas, la valeur
1109  * renvoyee sera %MAX_SCHEDULE_TIMEOUT.
1110  *
1111  * Dans tous les cas, on est sur que la valeur renvoyee est positive ou nulle.
1112  */
1113 fastcall signed long __sched schedule_timeout(signed long timeout)
```

- On fait appel à la fonction de fermeture associée au périphérique.

  Nous étudierons ci-dessous le cas de la fonction correspondant à la carte 3Com 501.
- On change le vecteur de drapeaux du descripteur de périphérique pour indiquer que le périphérique est désactivé.
- L'arrêt du périphérique est communiqué à tous les protocoles concernés et on renvoie 0.

35.1.1.2 Désactivation de la stratégie de mise en file d'attente

La fonction `dev_deactivate()` de désactivation de la stratégie de mise en file d'attente associée au descripteur de périphérique passé en argument est définie dans le fichier `linux/net/sched/sch_generic.c`:

Code Linux 2.6.10

```
551 void dev_deactivate(struct net_device *dev)
552 {
553         struct Qdisc *qdisc;
554
555         spin_lock_bh(&dev->queue_lock);
556         qdisc = dev->qdisc;
557         dev->qdisc = &noop_qdisc;
558
559         qdisc_reset(qdisc);
560
561         spin_unlock_bh(&dev->queue_lock);
562
563         dev_watchdog_down(dev);
564
565         while (test_bit(__LINK_STATE_SCHED, &dev->state))
566                 yield();
567
568         spin_unlock_wait(&dev->xmit_lock);
569 }
```

Code Linux 2.6.10

La fonction `qdisc_reset()` est définie dans le fichier `linux/net/sched/sch_generic.c`:

```
449 /* Avec dev->queue_lock et BH ! */
450
451 void qdisc_reset(struct Qdisc *qdisc)
452 {
453         struct Qdisc_ops *ops = qdisc->ops;
454
455         if (ops->reset)
456                 ops->reset(qdisc);
457 }
```

35.1.1.3 Arrêt du minuteur d'identification de problème en émission

La fonction `dev_watchdog_down()` arrête le minuteur dédié à l'identification de problèmes d'émission. Elle est définie dans le fichier `linux/net/sched/sch_generic.c`:

Code Linux 2.6.10

```
232 static void dev_watchdog_down(struct net_device *dev)
233 {
234         spin_lock_bh(&dev->xmit_lock);
235         if (del_timer(&dev->watchdog_timer))
236                 __dev_put(dev);
237         spin_unlock_bh(&dev->xmit_lock);
238 }
```

Code Linux 2.6.10

La macro `__dev_put()` est définie dans le fichier `linux/include/linux/netdevice.h`:

```
695 #define __dev_put(dev) atomic_dec(&(dev)->refcnt)
```

35.1.2 Cas de la carte 3Com 501

Nous avons vu au chapitre 17 que la fonction spécifique de fermeture associée à la carte 3Com 501 s'appelle `el1_close()`. Celle-ci est définie dans le fichier `linux/drivers/net/3c501.c`:

Code Linux 2.6.10

```
788 /**
789  * el1_close :
```

```
790  * @dev : carte 3c501 a fermer
791  *
792  * Ferme une carte 3c501. Le drapeau IFF_UP a ete efface par l'utilisateur via
793  * le ioctl SIOCSIFFLAGS. Nous arretons toute transmission encore presente dans la
794  * file d'attente et inhibons alors les interruptions. Nous reinitialisons finalement la puce.
795  * Les effets du reste seront nettoyes par #el1_open. Renvoie toujours 0, indiquant
796  * un succes.
797  */
798
799 static int el1_close(struct net_device *dev)
800 {
801         int ioaddr = dev->base_addr;
802
803         if (el_debug > 2)
804                 printk(KERN_INFO "%s: Shutting down Ethernet card at %#x.\n",
                               dev->name, ioaddr);
805
806         netif_stop_queue(dev);
807
808         /*
809          *      Libere et inhibe l'IRQ.
810          */
811
812         free_irq(dev->irq, dev);
813         outb(AX_RESET, AX_CMD);         /* Reinitialise la puce */
814
815         return 0;
816 }
```

Autrement dit :

- On déclare un port d'entrée-sortie, que l'on initialise avec celui fourni par le descripteur de périphérique passé en argument.
- Si le niveau de débogage est strictement supérieur à 2, on affiche un message noyau indiquant que l'on est en train de fermer la carte.
- On indique qu'il n'est plus possible de transmettre grâce à la fonction `netif_stop_queue()`.
- On libère l'IRQ associée au périphérique.
- On réinitialise le contrôleur du périphérique et on renvoie 0.

35.2 Retrait d'un périphérique réseau

Le retrait d'un périphérique réseau est effectué grâce à la fonction `unregister_netdev()` définie dans le fichier `linux/drivers/net/net_init.c` :

Code Linux 2.6.10

```
144 void unregister_netdev(struct net_device *dev)
145 {
146         rtnl_lock();
147         unregister_netdevice(dev);
148         rtnl_unlock();
149 }
```

qui fait appel à la fonction `unregister_netdevice()`, définie dans le fichier `linux/net/core/dev.c` :

Code Linux 2.6.10

```
2994 /**
2995  *      unregister_netdevice - supprime un peripherique du noyau
2996  *      @dev : peripherique
2997  *
2998  *      Cette fonction arrete une interface de peripherique et la supprime
```

```
2999  *      des tables du noyau. En cas de succes 0 est renvoye, en cas d'echec
3000  *      un code errno negatif est renvoye.
3001  *
3002  *      Les appelants doivent detenir le semaphore rtnl. Voir le commentaire a la
3003  *      fin de Space.c pour les details sur le verrouillage. Vous pouvez vouloir
3004  *      unregister_netdev() au lieu de cette fonction.
3005  */
3006
3007 int unregister_netdevice(struct net_device *dev)
3008 {
3009         struct net_device *d, **dp;
3010
3011         BUG_ON(dev_boot_phase);
3012         ASSERT_RTNL();
3013
3014         /* Certains peripheriques appellent sans s'enregistrer pour derouler
               l'initialisation. */
3015         if (dev->reg_state == NETREG_UNINITIALIZED) {
3016                 printk(KERN_DEBUG "unregister_netdevice: device %s/%p never "
3017                                   "was registered\n", dev->name, dev);
3018                 return -ENODEV;
3019         }
3020
3021         BUG_ON(dev->reg_state != NETREG_REGISTERED);
3022
3023         /* Si le peripherique est en marche, commencons par le fermer. */
3024         if (dev->flags & IFF_UP)
3025                 dev_close(dev);
3026
3027         /* Et enlevons-le de la chaine des peripheriques. */
3028         for (dp = &dev_base; (d = *dp) != NULL; dp = &d->next) {
3029                 if (d == dev) {
3030                         write_lock_bh(&dev_base_lock);
3031                         hlist_del(&dev->name_hlist);
3032                         hlist_del(&dev->index_hlist);
3033                         if (dev_tail == &dev->next)
3034                                 dev_tail = dp;
3035                         *dp = d->next;
3036                         write_unlock_bh(&dev_base_lock);
3037                         break;
3038                 }
3039         }
3040         if (!d) {
3041                 printk(KERN_ERR "unregister net_device: '%s' not found\n",
3042                        dev->name);
3043                 return -ENODEV;
3044         }
3045
3046         dev->reg_state = NETREG_UNREGISTERING;
3047
3048         synchronize_net();
3049
3050         /* Arret de la discipline de mise en file d'attente. */
3051         dev_shutdown(dev);
3052
3053
3054         /* Notifions aux protocoles que nous sommes en train de detruire
3055            ce peripherique. Ils doivent nettoyer toute chose.
3056         */
3057         notifier_call_chain(&netdev_chain, NETDEV_UNREGISTER, dev);
3058
3059         /*
```

```
3060          *      Vidons la chaine de multiffusion
3061          */
3062         dev_mc_discard(dev);
3063
3064         if (dev->uninit)
3065                 dev->uninit(dev);
3066
3067         /* La chaine de notification DOIT nous detacher du peripherique maitre. */
3068         BUG_TRAP(!dev->master);
3069
3070         free_divert_blk(dev);
3071
3072         /* Finir le traitement de retrait apres le deverrouillage */
3073         net_set_todo(dev);
3074
3075         synchronize_net();
3076
3077         dev_put(dev);
3078         return 0;
3079 }
```

Autrement dit :

- On déclare un descripteur de périphérique réseau et une liste de tels descripteurs.
- Si nous sommes en phase de démarrage, on reporte une erreur.
- On attend de pouvoir s'approprier le sémaphore RTNL.
- Si le descripteur de périphérique passé en argument n'est pas initialisé, on affiche un message noyau et on renvoie l'opposé du code d'erreur `ENODEV`.
- Si le descripteur de périphérique passé en argument n'a pas été installé, on reporte une erreur.
- Si le périphérique est en activité, on commence par le désactiver.
- On essaie de retirer le descripteur passé en argument de la liste des périphériques réseau. Si on ne le trouve pas dans cette liste, on affiche un message noyau et on renvoie l'opposé du code d'erreur `ENODEV`.
- On change son état d'enregistrement en "en train d'être retiré".
- On synchronise le réseau.
- On arrête la stratégie de mise en file d'attente du descripteur de périphérique passé en argument.

La fonction `dev_shutdown()` est définie dans le fichier `linux/net/sched/sch_generic.c`:

Code Linux 2.6.10

```
582 void dev_shutdown(struct net_device *dev)
583 {
584         struct Qdisc *qdisc;
585
586         qdisc_lock_tree(dev);
587         qdisc = dev->qdisc_sleeping;
588         dev->qdisc = &noop_qdisc;
589         dev->qdisc_sleeping = &noop_qdisc;
590         qdisc_destroy(qdisc);
591 #if defined(CONFIG_NET_SCH_INGRESS) || defined(CONFIG_NET_SCH_INGRESS_MODULE)
592         if ((qdisc = dev->qdisc_ingress) != NULL) {
593                 dev->qdisc_ingress = NULL;
594                 qdisc_destroy(qdisc);
595         }
596 #endif
597         BUG_TRAP(!timer_pending(&dev->watchdog_timer));
```

```
598         qdisc_unlock_tree(dev);
599 }
```

- On notifie aux protocoles que nous sommes en train de retirer ce périphérique.
- On force la chaîne de multidiffusion.

  La multidiffusion ne nous intéresse pas dans ce livre. La fonction `dev_mc_discard()`, qui vide une liste, est définie dans le fichier `linux/net/core/dev_mcast.c`:

Code Linux 2.6.10

```
202 /*
203  *      Detruit la liste de multidiffusion lorsqu'un peripherique est arrete
204  */
205
206 void dev_mc_discard(struct net_device *dev)
207 {
208         spin_lock_bh(&dev->xmit_lock);
209
210         while (dev->mc_list != NULL) {
211                 struct dev_mc_list *tmp = dev->mc_list;
212                 dev->mc_list = tmp->next;
213                 if (tmp->dmi_users > tmp->dmi_gusers)
214                         printk("dev_mc_discard: multicast leakage! dmi_users=%d\n",
                                tmp->dmi_users);
215                 kfree(tmp);
216         }
217         dev->mc_count = 0;
218
219         spin_unlock_bh(&dev->xmit_lock);
220 }
```

- Si le descripteur de périphérique fait apparaître une méthode `uninit()`, ce qui n'est le cas d'aucun descripteur de périphérique à l'heure actuelle, on fait appel à elle.
- Si la chaîne de notification ne nous a pas détachée du périphérique maître, on reporte une erreur.
- On libère la trame de diversion éventuelle.

  La fonction `free_divert_blk()` est définie dans le fichier `linux/net/core/dv.c`:

Code Linux 2.6.10

```
78 /*
79  * Libere un divert_blk alloue par la fonction ci-dessus, si il a ete
80  * alloue sur ce peripherique.
81  */
82 void free_divert_blk(struct net_device *dev)
83 {
84         if (dev->divert) {
85                 kfree(dev->divert);
86                 dev->divert=NULL;
87                 dev_put(dev);
88                 printk(KERN_DEBUG "divert: freeing divert_blk for %s\n",
89                         dev->name);
90         } else {
91                 printk(KERN_DEBUG "divert: no divert_blk to free, %s not ethernet\n",
92                         dev->name);
93         }
94 }
```

- On termine le traitement du retrait en ajoutant le champ `dev->todo_list` du descripteur de périphérique réseau à la liste `net_todo_list`.
- On décrémente le compteur d'utilisation de ce descripteur de périphérique.

  La fonction `dev_put()` est définie dans le fichier `linux/include/linux/netdevice.h`:

Code Linux 2.6.10

```
690 static inline void dev_put(struct net_device *dev)
```

```
691 {
692         atomic_dec(&dev->refcnt);
693 }
```

35.2.1 Démontage du système de fichier

La fonction `netdev_run_todo()`, étudiée au chapitre 15, montre que dans le cas d'un retrait on démonte le système de fichiers associé grâce à la fonction `netdev_unregister_sysfs()` étudiée ci-après, on change l'état en "désinstallé", on attend jusqu'à ce que toutes les références aient terminées (en faisant appel à la fonction `netdev_wait_allrefs()` étudiée ci-dessous), on vérifie quelques points, puis on fait appel à la fonction spécifique de destruction pour ce périphérique si elle existe.

Dans le cas de la carte 3Com 501, nous avons vu qu'il n'existe pas de fonction de destrucion spécifique.

35.2.2 Démontage du système de fichiers

La fonction `netdev_unregister_sysfs()` est définie dans le fichier `linux/net/core/net-sysfs.c`: Code Linux 2.6.0

```
386 void netdev_unregister_sysfs(struct net_device * net)
387 {
388         struct class_device * class_dev = &(net->class_dev);
389
390         if (net->get_stats)
391                 sysfs_remove_group(&class_dev->kobj, &netstat_group);
392
393 #ifdef WIRELESS_EXT
394         if (net->get_wireless_stats)
395                 sysfs_remove_group(&class_dev->kobj, &wireless_group);
396 #endif
397         class_device_del(class_dev);
398
399 }
```

35.2.3 Attente des références

La fonction `netdev_wait_allrefs()` est définie dans le fichier `linux/net/core/dev.c`: Code Linux 2.6.0

```
2730 /*
2731  * netdev_wait_allrefs - attend jusqu'a ce que toutes les references s'en sont allees.
2732  *
2733  * Ceci est appele lors de l'enregistrement des peripheriques reseau.
2734  *
2735  * Tout protocole ou peripherique qui a une a reference doit s'enregistrer
2736  * pour la notification des peripheriques reseau, puis nettoyer et retirer la
2737  * reference si elle recoit un evenement UNREGISTER.
2738  * Nous pouvons avoir des pretentieux ici si des protocoles bogues n'appellent pas
2739  * correctement dev_put.
2740  */
2741 static void netdev_wait_allrefs(struct net_device *dev)
2742 {
2743         unsigned long rebroadcast_time, warning_time;
2744
2745         rebroadcast_time = warning_time = jiffies;
2746         while (atomic_read(&dev->refcnt) != 0) {
2747                 if (time_after(jiffies, rebroadcast_time + 1 * HZ)) {
2748                         rtnl_shlock();
```

```
2749                        rtnl_exlock();
2750
2751                        /* Renvoyons la notification de retrait */
2752                        notifier_call_chain(&netdev_chain,
2753                                            NETDEV_UNREGISTER, dev);
2754
2755                        if (test_bit(__LINK_STATE_LINKWATCH_PENDING,
2756                                     &dev->state)) {
2757                                /* Nous ne devons pas avoir d'evenements linkwatch
2758                                 * en attente de retrait. Si cela
2759                                 * arrive, nous ne passons pas la file d'attente
2760                                 * a l'ordonnanceur, d'ou un resultat de noop
2761                                 * pour ce peripherique.
2762                                 */
2763                                linkwatch_run_queue();
2764                        }
2765
2766                        rtnl_exunlock();
2767                        rtnl_shunlock();
2768
2769                        rebroadcast_time = jiffies;
2770                }
2771
2772                current->state = TASK_INTERRUPTIBLE;
2773                schedule_timeout(HZ / 4);
2774
2775                if (time_after(jiffies, warning_time + 10 * HZ)) {
2776                        printk(KERN_EMERG "unregister_netdevice: "
2777                               "waiting for %s to become free. Usage "
2778                               "count = %d\n",
2779                               dev->name, atomic_read(&dev->refcnt));
2780                        warning_time = jiffies;
2781                }
2782        }
2783 }
```

Sixième partie

Complément sur IP : la fragmentation

Chapitre 36

Fragmentation pour la couche réseau : le cas de IPv4

Nous avons vu au chapitre 33 comment envoyer un paquet IP ordinaire depuis un ordinateur hôte en supposant qu'il n'avait pas besoin dêtre fragmenté. Nous avons également vu au chapitre 23 comment il était reçu par un ordinateur hôte en supposant qu'il n'avait pas été fragmenté au cours de sa route. Nous allons étudier dans ce chapitre comment fragmenter et réassembler un paquet.

36.1 Fragmentation

Comme nous l'avons vu au chapitre 33, la fonction de fragmentation `ip_fragment()` est éventuellement appelée par la fonction `ip_output()` en spécifiant deux arguments : un descripteur de tampon de socket et la fonction `ip_finish_output()`.

Code Linux 2.6.10

La fonction `ip_fragment()` est définie dans le fichier `linux/net/ipv4/ip_output.c`:

```
420 /*
421  *      Ce datagramme IP est trop gros pour etre envoye en un seul morceau. Coupons-le en
422  *      morceaux plus petits (chacun de taille egale a [celle de] l'en-tete IP plus
423  *      un bloc de donnees de la partie des donnees IP originelle) qui conviennent a une seule
424  *      trame du peripherique, et placons une telle trame dans la file d'attente d'envoi.
425  */
426
427 int ip_fragment(struct sk_buff *skb, int (*output)(struct sk_buff*))
428 {
429         struct iphdr *iph;
430         int raw = 0;
431         int ptr;
432         struct net_device *dev;
433         struct sk_buff *skb2;
434         unsigned int mtu, hlen, left, len, ll_rs;
435         int offset;
436         int not_last_frag;
437         struct rtable *rt = (struct rtable*)skb->dst;
438         int err = 0;
439
440         dev = rt->u.dst.dev;
441
442         /*
443          *      Pointer sur l'en-tete du datagramme IP.
444          */
445
446         iph = skb->nh.iph;
447
448         if (unlikely((iph->frag_off & htons(IP_DF)) && !skb->local_df)) {
449                 icmp_send(skb, ICMP_DEST_UNREACH, ICMP_FRAG_NEEDED,
450                           htonl(dst_pmtu(&rt->u.dst)));
451                 kfree_skb(skb);
452                 return -EMSGSIZE;
453         }
454
455         /*
456          *      Initialiser les valeurs de depart.
457          */
458
459         hlen = iph->ihl * 4;
460         mtu = dst_pmtu(&rt->u.dst) - hlen;      /* Taille de l'espace des donnees */
461
462         /* Lorsque frag_list est donnee, l'utiliser. Verifier d'abord sa validite :
463          * certaines transformations pourraient creer de fausses frag_list ou couper des
464          * listes existantes, ce qui n'est pas interdit. Dans ce cas ne pas copier.
465          *
466          * PLUS TARD : Cette etape peut etre fusionnee avec la generation reelle de fragments,
467          * nous pouvons passer a la copie lorsque nous voyons le premier fragment mauvais.
468          */
469         if (skb_shinfo(skb)->frag_list) {
470                 struct sk_buff *frag;
471                 int first_len = skb_pagelen(skb);
472
473                 if (first_len - hlen > mtu ||
474                     ((first_len - hlen) & 7) ||
```

```
475                 (iph->frag_off & htons(IP_MF|IP_OFFSET)) ||
476                 skb_cloned(skb))
477                     goto slow_path;
478
479             for (frag = skb_shinfo(skb)->frag_list; frag; frag = frag->next) {
480                     /* Geometrie correcte. */
481                     if (frag->len > mtu ||
482                         ((frag->len & 7) && frag->next) ||
483                         skb_headroom(frag) < hlen)
484                         goto slow_path;
485
486                     /* skb partiellement clone ? */
487                     if (skb_shared(frag))
488                             goto slow_path;
489             }
490
491             /* Chaque chose est OK. Generer ! */
492
493             err = 0;
494             offset = 0;
495             frag = skb_shinfo(skb)->frag_list;
496             skb_shinfo(skb)->frag_list = NULL;
497             skb->data_len = first_len - skb_headlen(skb);
498             skb->len = first_len;
499             iph->tot_len = htons(first_len);
500             iph->frag_off |= htons(IP_MF);
501             ip_send_check(iph);
502
503             for (;;) {
504                     /* Preparer l'en-tete de la trame suivante,
505                      * avant de laisser tomber la precedente. */
506                     if (frag) {
507                             frag->h.raw = frag->data;
508                             frag->nh.raw = __skb_push(frag, hlen);
509                             memcpy(frag->nh.raw, iph, hlen);
510                             iph = frag->nh.iph;
511                             iph->tot_len = htons(frag->len);
512                             ip_copy_metadata(frag, skb);
513                             if (offset == 0)
514                                     ip_options_fragment(frag);
515                             offset += skb->len - hlen;
516                             iph->frag_off = htons(offset>>3);
517                             if (frag->next != NULL)
518                                     iph->frag_off |= htons(IP_MF);
519                             /* Pret, completer la somme de controle */
520                             ip_send_check(iph);
521                     }
522
523                     err = output(skb);
524
525                     if (err || !frag)
526                             break;
527
528                     skb = frag;
529                     frag = skb->next;
530                     skb->next = NULL;
531             }
532
533             if (err == 0) {
534                     IP_INC_STATS(IPSTATS_MIB_FRAGOKS);
535                     return 0;
536             }
```

```
537
538                 while (frag) {
539                         skb = frag->next;
540                         kfree_skb(frag);
541                         frag = skb;
542                 }
543                 IP_INC_STATS(IPSTATS_MIB_FRAGFAILS);
544                 return err;
545         }
546
547 slow_path:
548         left = skb->len - hlen;         /* Espace par trame */
549         ptr = raw + hlen;               /* La ou on demarre */
550
551 #ifdef CONFIG_BRIDGE_NETFILTER
552         /* Pour un trafic IP sur pont encapsule par exemple dans un en-tete vlan,
553          * nous avons besoin de faire de la place pour l'en-tete encapsulant */
554         ll_rs = LL_RESERVED_SPACE_EXTRA(rt->u.dst.dev, nf_bridge_pad(skb));
555         mtu -= nf_bridge_pad(skb);
556 #else
557         ll_rs = LL_RESERVED_SPACE(rt->u.dst.dev);
558 #endif
559         /*
560          *      Fragmenter le datagramme.
561          */
562
563         offset = (ntohs(iph->frag_off) & IP_OFFSET) << 3;
564         not_last_frag = iph->frag_off & htons(IP_MF);
565
566         /*
567          *      Conserver une copie des donnees jusqu'a ce qu'on ait fini.
568          */
569
570         while(left > 0) {
571                 len = left;
572                 /* SI : il ne convient pas, utiliser 'mtu' - l'espace restant des donnees */
573                 if (len > mtu)
574                         len = mtu;
575                 /* SI : nous ne sommes pas en train d'envoyer la fin du paquet
576                    alors aligner le prochain depart sur une frontiere de huit octets */
577                 if (len < left) {
578                         len &= ~7;
579                 }
580                 /*
581                  *      Allouer un tampon.
582                  */
583
584                 if ((skb2 = alloc_skb(len+hlen+ll_rs, GFP_ATOMIC)) == NULL) {
585                         NETDEBUG(printk(KERN_INFO "IP: frag: no memory for new fragment!\n"));
586                         err = -ENOMEM;
587                         goto fail;
588                 }
589
590                 /*
591                  *      Mettre les donnees dans le paquet
592                  */
593
594                 ip_copy_metadata(skb2, skb);
595                 skb_reserve(skb2, ll_rs);
596                 skb_put(skb2, len + hlen);
597                 skb2->nh.raw = skb2->data;
598                 skb2->h.raw = skb2->data + hlen;
```

```
599
600                 /*
601                  *      Charger la memoire pour le fragment a tout proprietaire
602                  *      qui pourrait le posseder
603                  */
604
605                 if (skb->sk)
606                         skb_set_owner_w(skb2, skb->sk);
607
608                 /*
609                  *      Copier l'en-tete de paquet dans le nouveau tampon.
610                  */
611
612                 memcpy(skb2->nh.raw, skb->data, hlen);
613
614                 /*
615                  *      Copier un bloc du datagramme IP.
616                  */
617                 if (skb_copy_bits(skb, ptr, skb2->h.raw, len))
618                         BUG();
619                 left -= len;
620
621                 /*
622                  *      Renseigner les champs du nouvel en-tete.
623                  */
624                 iph = skb2->nh.iph;
625                 iph->frag_off = htons((offset >> 3));
626
627                 /* ANK : pas beau mais truc effectif. Mise a jour des options seulement si
628                  * le segment a etre fragmente etait LE PREMIER (autrement,
629                  * les options ont deja ete rectifiees) et le faire UNE FOIS
630                  * sur le skb initial, de telle facon que les fragments suivants
631                  * heriteront des options rectifiees.
632                  */
633                 if (offset == 0)
634                         ip_options_fragment(skb);
635
636                 /*
637                  *      Ajoute AC : Si nous sommes en train de fragmenter un fragment qui
638                  *                  n'est pas le dernier fragment alors conserver MF sur
                                       chaque bit
639                  */
640                 if (left > 0 || not_last_frag)
641                         iph->frag_off |= htons(IP_MF);
642                 ptr += len;
643                 offset += len;
644
645                 /*
646                  *      Mettre ce fragment dans la file d'attente en envoi.
647                  */
648
649                 IP_INC_STATS(IPSTATS_MIB_FRAGCREATES);
650
651                 iph->tot_len = htons(len + hlen);
652
653                 ip_send_check(iph);
654
655                 err = output(skb2);
656                 if (err)
657                         goto fail;
658         }
659         kfree_skb(skb);
```

```
660         IP_INC_STATS(IPSTATS_MIB_FRAGOKS);
661         return err;
662
663 fail:
664         kfree_skb(skb);
665         IP_INC_STATS(IPSTATS_MIB_FRAGFAILS);
666         return err;
667 }
```

Autrement dit :

- On déclare un en-tête IP, un index brut, un pointeur, un descripteur de périphérique réseau, un deuxième descripteur de tampon de socket, une MTU, une longueur d'en-tête, le nombre de fragments traités, une longueur, un décalage, un test indiquant que ce n'est pas le dernier fragment, une entrée de table de routage et un code de retour.
- On instantie l'entrée de table de routage avec celle associée au descripteur de tampon passé en argument.
- On instantie le descripteur de périphérique réseau avec celui associé à cette entrée de table de routage.
- On instantie l'en-tête IP avec celle associée au descripteur de tampon passé en argument.
- Si l'en-tête IP spécifie que l'on ne doit pas fragmenter le paquet, on envoie un message ICMP à l'émetteur indiquant que l'on doit fragmenter le paquet, on libère le descripteur de tampon passé en argument et on renvoie l'opposé du code d'erreur `EMSGSIZE`.
- On initialise la taille de l'en-tête IP en octets, c'est-à-dire quatre fois celle indiquée par l'en-tête lui-même (exprimée en mots).
- On détermine la taille des données dans un fragment, à savoir l'unité de transfert du réseau moins la longueur de l'en-tête.
- Si une liste de fragmentation existe déjà dans le descripteur de tampon :
  - On déclare un descripteur de tampon pour un fragment.
  - On déclare une longueur initiale du descripteur de tampon passé en argument, que l'on instantie avec la longueur totale des fragments de celui-ci.

    La fonction en ligne `skb_pagelen()` est définie dans le fichier `linux/include/linux/skbuff.h` :

Code Linux 2.6.10

```
665 static inline int skb_pagelen(const struct sk_buff *skb)
666 {
667         int i, len = 0;
668
669         for (i = (int)skb_shinfo(skb)->nr_frags - 1; i >= 0; i--)
670                 len += skb_shinfo(skb)->frags[i].size;
671         return len + skb_headlen(skb);
672 }
```

  - Si cette longueur est supérieure à l'unité de transfert ou si le descripteur de socket passé en argument est cloné, on ne peut pas effectuer le traitement rapide ci-dessous.
  - On vérifie chacun des fragments de la liste :
    - Si la longueur d'un tel fragment est supérieure à l'unité de transfert ou si la taille de l'espace de tête du tampon dont le descripteur est passé en argument est inférieure à celle de l'en-tête, on ne peut pas effectuer le traitement rapide ci-dessous.
    - Si aucun descripteur de couche réseau n'est associé au fragment ou au descripteur de tampon passé en argument, on ne peut pas effectuer le traitement rapide ci-dessous.

  - Si le fragment est partagé, on ne peut pas effectuer le traitement rapide ci-dessous.

- Sinon on peut passer à un traitement rapide :

  - Le code de retour et le décalage sont initialisés à zéro.
  - Le descripteur de tampon de fragment est instantié avec le premier élément de la liste des fragments du descripteur de tampon passé en argument et on indique dans ce dernier descripteur qu'on l'a récupéré.
  - On spécifie comme longueur des données du descripteur de tampon passé en argument la longueur initiale moins la longueur de l'en-tête.
  - On spécifie comme longueur totale du descripteur de tampon passé en argument la longueur initiale.
  - On spécifie comme longueur totale du paquet dans l'en-tête IP la longueur initiale.
  - On spécifie dans l'en-tête IP que ce paquet est un fragment suivi d'autres fragments.
  - On ajoute la somme de contrôle à l'en-tête IP.
  - On entre dans une boucle (pseudo-)infinie pour traiter tous les fragments, en fait tant que le descripteur de tampon de fragment est non nul :

    - On fait pointer l'en-tête matériel brut du descripteur de fragment sur le début des données du fragment.
    - On fait pointer l'en-tête réseau brut du descripteur de fragment à l'emplacement adéquat et on instantie cet en-tête avec l'en-tête IP défini auparavant.
    - On fait pointer l'adresse de l'en-tête IP à l'adresse de l'en-tête réseau brut du descripteur de fragment.
    - On initialise la longueur totale du paquet dans cet en-tête avec la longueur du fragment.
    - On copie les méta-données du descripteur de tampon dans le fragment.

      La fonction `ip_copy_metadata()` est définie dans le fichier `linux/net/ipv4/ip_output.c` :

Code Linux 2.6.10

```
386 static void ip_copy_metadata(struct sk_buff *to, struct sk_buff *from)
387 {
388         to->pkt_type = from->pkt_type;
389         to->priority = from->priority;
390         to->protocol = from->protocol;
391         to->security = from->security;
392         to->dst = dst_clone(from->dst);
393         to->dev = from->dev;
394
395         /* Copie les drapeaux de chaque fragment. */
396         IPCB(to)->flags = IPCB(from)->flags;
397
398 #ifdef CONFIG_NET_SCHED
399         to->tc_index = from->tc_index;
400 #endif
401 #ifdef CONFIG_NETFILTER
402         to->nfmark = from->nfmark;
403         to->nfcache = from->nfcache;
404         /* L'association de connexion est la meme que dans le paquet
               pre-frag */
405         nf_conntrack_put(to->nfct);
406         to->nfct = from->nfct;
407         nf_conntrack_get(to->nfct);
```

```
408         to->nfctinfo = from->nfctinfo;
409 #ifdef CONFIG_BRIDGE_NETFILTER
410         nf_bridge_put(to->nf_bridge);
411         to->nf_bridge = from->nf_bridge;
412         nf_bridge_get(to->nf_bridge);
413 #endif
414 #ifdef CONFIG_NETFILTER_DEBUG
415         to->nf_debug = from->nf_debug;
416 #endif
417 #endif
418 }
```

Code Linux 2.6.10

La macro `IPCB()` de copie du bloc de contrôle est définie dans le fichier `linux/include/net/ip.h`:

```
57 #define IPCB(skb) ((struct inet_skb_parm*)((skb)->cb))
```

– Si le décalage du fragment est nul, il s'agit du premier fragment. On remplit les renseignements sur les options grâce à la fonction `ip_options_fragment()` définie dans le fichier `linux/net/ipv4/ip_options.c`:

Code Linux 2.6.10

```
202 /*
203  *      "fragmenter" les options, juste remplir les options non
204  *      permises dans les fragments avec NOOP.
205  *      Simple et stupide 8), mais la facon la plus efficace.
206  */
207
208 void ip_options_fragment(struct sk_buff * skb)
209 {
210         unsigned char * optptr = skb->nh.raw;
211         struct ip_options * opt = &(IPCB(skb)->opt);
212         int  l = opt->optlen;
213         int  optlen;
214
215         while (l > 0) {
216                 switch (*optptr) {
217                 case IPOPT_END:
218                         return;
219                 case IPOPT_NOOP:
220                         l--;
221                         optptr++;
222                         continue;
223                 }
224                 optlen = optptr[1];
225                 if (optlen<2 || optlen>l)
226                   return;
227                 if (!IPOPT_COPIED(*optptr))
228                         memset(optptr, IPOPT_NOOP, optlen);
229                 l -= optlen;
230                 optptr += optlen;
231         }
232         opt->ts = 0;
233         opt->rr = 0;
234         opt->rr_needaddr = 0;
235         opt->ts_needaddr = 0;
236         opt->ts_needtime = 0;
237         return;
238 }
```

– On met à jour le décalage en ajoutant la longueur du descripteur de tampon moins la longueur de l'en-tête et on place la valeur obtenue dans l'en-tête IP.

        - Si le fragment est suivi d'un autre fragment, on positionne le drapeau `MF` (plus de fragments) dans l'en-tête IP.
        - On place la somme de contrôle dans l'en-tête IP.

    - On passe à la fonction `ip_finish_output()` passée en argument et étudiée dans le chapitre 33. Si celle-ci renvoie une erreur ou s'il n'y a plus de fragment, on sort de la boucle (pseudo-)infinie.
    - Sinon on passe au fragment suivant : le descripteur de tampon prend la valeur du fragment, le tampon prend la valeur du segment suivant, on spécifie que le descripteur de tampon n'a pas de suivant.

  - Si le code de retour est nul, on incrémente les informations statistiques spécifiant que la fragmentation s'est bien passée et on renvoie 0.
  - Sinon on passe les fragments, on incrémente les informations statistiques spécifiant que la fragmentation a échouée et on renvoie le dernier code d'erreur obtenu.

- On a vu que dans un certain nombre de cas, on ne peut pas effectuer le traitement rapide ci-dessus. Dans ces cas-là :

  - On initialise l'espace des données restant, c'est-à-dire la longueur totale du paquet moins la longueur de l'en-tête IP.
  - On initialise le pointeur au début des données brutes dont on passe l'en-tête IP.
  - Tant qu'on n'a pas parcouru toute la zone des données :

    - On initialise la longueur avec le minimum entre le nombre d'octets restants dans la zone des données et l'unité de transfert.
    - S'il ne s'agit pas du dernier fragment, on aligne le prochain départ sur une frontière de huit octets.
    - On essaie d'allouer le deuxième descripteur de tampon. Si on n'y parvient pas, on affiche un message noyau et on renvoie l'opposé du code d'erreur `ENOMEM`.
    - On copie les méta-données du descripteur de tampon passé en argument dans le deuxième descripteur de tampon.
    - On copie le nombre adéquat d'octets des données du descripteur de tampon passé en argument dans celles du deuxième descripteur de tampon.
    - On renseigne les champs de l'en-tête du nouveau fragment.
    - On incrémente les informations statistiques sur le nombre de fragments IP créés.
    - On passe à la fonction `ip_finish_output()` passée en argument et étudiée dans le chapitre 33.
    - Si celle-ci renvoie une erreur, on libère le descripteur de tampon passé en argument, on incrémente les informations statistiques sur les échecs de fragmentation et on renvoie le code d'erreur.
    - Sinon on libère le descripteur de tampon passé en argument, on incrémente les informations statistiques sur les fragmentations réussies et on renvoie le code de retour.

36.2 Réassemblage

Le réassemblage, c'est-à-dire la collecte et l'assemblage de paquets IP fragmentés, n'a lieu que dans le système d'extrémité. Pour ce faire, comme nous l'avons vu au chapitre 23, tous les paquets IP fragmentés sont transmis à la fonction de réassemblage `ip_defrag()` dans la fonction `ip_local_deliver()` en passant un descripteur de tampon de socket en argument.

36.2.1 Cache de fragment

Les paquets IP fragmentés sont gérés dans un **cache de fragment** jusqu'à ce que tous les fragments d'un paquet soient arrivés et que le paquet puisse être distribué localement ou jusqu'à ce que le temps d'attente maximal des fragments d'un paquet se soit écoulé et que le paquet soit rejeté.

Le temps maximal d'attente (environ 30 secondes) est représenté par la constante `IP_FRAG_TIME` définie dans le fichier `linux/include/net/ip.h`:

Code Linux 2.6.10

```
75 #define IP_FRAG_TIME    (30 * HZ)               /* Duree de vie d'un fragment */
```

Le cache de fragment est constitué d'une table de hachage comportant des entités du type `struct ipq`, comme le montre la figure 36.1 ([WPRMB–02], p. 281). Chacune de ces entités représente un paquet IP fragmenté.

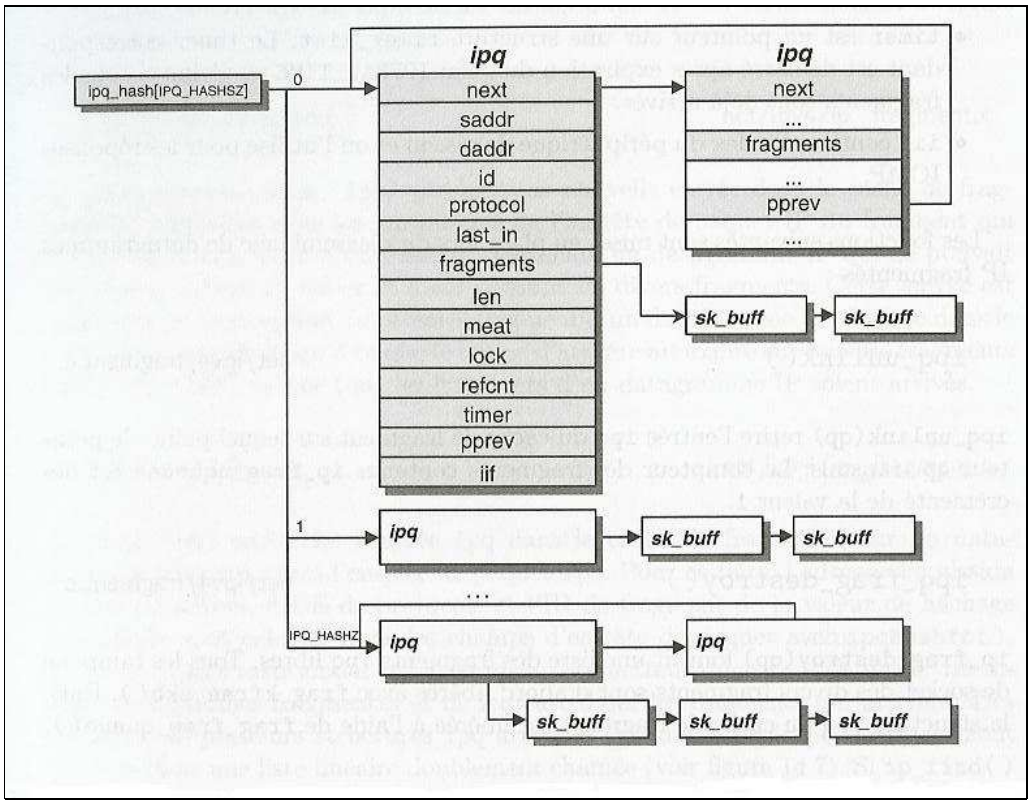


FIG. 36.1 – *Cache de fragment*

Le type `struct ipq` est défini dans le fichier `linux/net/ipv4/ip_fragment.c`: Code Linux 2.6.10

```
72 /* Decrit une entree dans la file d'attente des "datagrammes incomplets". */
73 struct ipq {
74         struct ipq      *next;          /* pointeurs pour la liste chainee              */
75         struct list_head lru_list;      /* membre de liste lru                          */
76         u32             saddr;
77         u32             daddr;
78         u16             id;
79         u8              protocol;
80         u8              last_in;
81 #define COMPLETE                4
82 #define FIRST_IN                2
83 #define LAST_IN                 1
84
85         struct sk_buff  *fragments;     /* liste chainee des fragments recus            */
86         int             len;            /* longueur totale du datagramme originel       */
87         int             meat;
88         spinlock_t      lock;
89         atomic_t        refcnt;
90         struct timer_list timer;        /* quand cette file d'attente expirera-t-elle ? */
91         struct ipq      **pprev;
92         int             iif;
93         struct timeval  stamp;
94 };
```

dont la signification de certains des champs est la suivante :

- `next` et `pprev` servent à la liaison des structures `ipq` dans une ligne de hachage. Il s'agit donc d'une liste doublement chaînée et d'une résolution de collision linéaire dans la table de hachage.
- Les membres `saddr`, `daddr`, `id` et `protocol` forment la clé destinée à la fonction de hachage et à l'attribution de fragments entrants à leurs paquets IP.
- `last_in` mémorise un drapeau indiquant si le premier fragment, le dernier fragment ou tous les fragments sont arrivés.
- `fragments` est une liste de descripteurs de tampon de socket dans laquelle tous les fragments arrivés sont stockés en fonction de la clé vue ci-dessus, dans l'ordre où ils seront ultérieurement reconstitués en tant que paquet complet.
- `len` contient la longueur totale du paquet IP originel.
- `meat` indique combien d'octets sont déjà placés dans le cache de fragment. Lorsque `meat` atteint la valeur `len`, tous les fragments du paquet sont arrivés. Il peut alors être reconstitué.
- `lock` sert pour se protéger d'opérations parallèles sur l'instance de structure de données `ipq`.
- `timer` est un pointeur sur une liste de minuteurs. Le minuteur correspondant est démarré lors de l'arrivée du premier fragment. Après expiration du délai `IP_FRAG_TIME`, on vérifie que tous les fragments sont arrivés.
- `iif` contient l'index du périphérique réseau. On l'utilise pour les réponses ICMP.

La table de hachage, appelée `ipq_hash[]` est définie dans le même fichier : Code Linux 2.6.10

```
96  /* Table de hachage. */
97
98  #define IPQ_HASHSZ      64
99
100 /* Un verrou par seau est facile a ajouter maintenant. */
101 static struct ipq *ipq_hash[IPQ_HASHSZ];
```

36.2.2 Traitement du réassemblage

36.2.2.1 Vue d'ensemble

Code Linux 2.6.10

La fonction `ip_defrag()` est définie dans le fichier `linux/net/ipv4/fragment.c`:

```
644 /* Traite un fragment de datagramme IP arrivant. */
645 struct sk_buff *ip_defrag(struct sk_buff *skb)
646 {
647         struct iphdr *iph = skb->nh.iph;
648         struct ipq *qp;
649         struct net_device *dev;
650
651         IP_INC_STATS_BH(IPSTATS_MIB_REASMREQDS);
652
653         /* Commencer par nettoyer la memoire. */
654         if (atomic_read(&ip_frag_mem) > sysctl_ipfrag_high_thresh)
655                 ip_evictor();
656
657         dev = skb->dev;
658
659         /* Consultation (ou creation) d'en-tete de file d'attente */
660         if ((qp = ip_find(iph)) != NULL) {
661                 struct sk_buff *ret = NULL;
662
663                 spin_lock(&qp->lock);
664
665                 ip_frag_queue(qp, skb);
666
667                 if (qp->last_in == (FIRST_IN|LAST_IN) &&
668                     qp->meat == qp->len)
669                         ret = ip_frag_reasm(qp, dev);
670
671                 spin_unlock(&qp->lock);
672                 ipq_put(qp, NULL);
673                 return ret;
674         }
675
676         IP_INC_STATS_BH(IPSTATS_MIB_REASMFAILS);
677         kfree_skb(skb);
678         return NULL;
679 }
```

Autrement dit :

- On déclare un en-tête IP, que l'on instantie avec celui associé au descripteur de tampon passé en argument.
- On déclare une entrée de cache de fragment et un descripteur de périphérique réseau.
- On incrémente les informations statistiques sur le nombre de requêtes de réassemblage.
- On commence par faire de la place dans la partie de la mémoire vive réservée au réassemblage, grâce à la fonction `ip_evictor()` étudiée ci-après, si la fragmentation des paquets utilise trop de mémoire.

Code Linux 2.6.10

La variable `ip_frag_mem` est définie dans le fichier `linux/net/ipv4/ip_fragment.c`:

```
169 atomic_t ip_frag_mem = ATOMIC_INIT(0);  /* Memoire utilisee pour les fragments */
```

Code Linux 2.6.10

La variable `sysctl_ipfrag_high_thresh` est définie dans le même fichier :

```
46 /* NOTE. La logique de la defragmentation IP est maintenant parallele au code IPv6
47  * correspondant. Si vous changez quelque chose ici, mettez egalement a jour _S'IL VOUS
48  * PLAIT_ ipv6/reassembly.c. Ou notifiez-le moi, au minimum. --ANK
```

```
49  */
50
51 /* Limites du cache de fragment. Nous ferons 256Ko en une fois. Si nous devions
52  * depasser cette limite, nous l'elaguerions a 192Ko. Ceci nous permettrait de nous en
53  * tirer meme avec les cas les plus extremes sans permettre a un attaquant de nuire
54  * considerablement aux performances de la machine.
55  */
56 int sysctl_ipfrag_high_thresh = 256*1024;
57 int sysctl_ipfrag_low_thresh  = 192*1024;
```

- On instantie le descripteur de périphérique réseau avec celui associé au descripteur de tampon passé en argument.
- On essaie de rechercher l'entrée de cache de fragment associée aux clés du fragment, en en créant une si besoin est, grâce à la fonction `ip_find()` étudiée ci-dessous. Si on n'y parvient pas, on incrémente les informations statistiques sur le nombre d'échec de réassemblage, on libère le descripteur de tampon passé en argument et on renvoie `NULL`.
- Sinon on déclare un descripteur de tampon de retour, que l'on initialise à `NULL`.
- On verrouille le réassemblage pour cette entrée de cache.
- On place le fragment nouvellement arrivé au bon endroit dans la file d'attente des fragments du paquet, grâce à la fonction `ip_frag_queue()` étudiée ci-après.
- Si le premier ou le dernier fragment est parvenu et si on a obtenu le nombre d'octets du paquet originel, on réassemble les fragments grâce à la fonction `ip_frag_reasm()` étudiée ci-après.
- On déverrouille l'entrée de cache de fragment.
- On libère l'entrée de cache de fragment, grâce à la fonction `ipq_put()` étudiée ci-après.
- On renvoie la valeur `NULL` ou la valeur de retour fournie par la fonction `ip_frag_reasm()`.

36.2.2.2 Première étape : nettoyage de la mémoire

La fonction `ip_evictor()` est appelée par `ip_defrag()` lorsque la fragmentation des paquets utilise trop de mémoire. La valeur de seuil pour la mémoire maximale du cache de fragment est fixée à 256 Ko. Toutes les lignes de hachage du cache de fragment sont vérifiées les unes après les autres et d'éventuelles entrées supprimées jusqu'à ce qu'on atteigne la valeur minimale de 192 Ko.

Elle est définie dans le fichier `linux/net/ipv4/ip_fragment.c` : Code Linux 2.6.10

```
243 /* Limitation de la memoire pour les fragments. Evictor met a la poubelle le plus vieux
244  * de la file d'attente jusqu'a ce que nous tombions en dessous du seuil minimal.
245  */
246 static void __ip_evictor(int threshold)
247 {
248         struct ipq *qp;
249         struct list_head *tmp;
250         int work;
251
252         work = atomic_read(&ip_frag_mem) - threshold;
253         if (work <= 0)
254                 return;
255
256         while (work > 0) {
257                 read_lock(&ipfrag_lock);
258                 if (list_empty(&ipq_lru_list)) {
259                         read_unlock(&ipfrag_lock);
260                         return;
261                 }
262                 tmp = ipq_lru_list.next;
```

```
263                 qp = list_entry(tmp, struct ipq, lru_list);
264                 atomic_inc(&qp->refcnt);
265                 read_unlock(&ipfrag_lock);
266
267                 spin_lock(&qp->lock);
268                 if (!(qp->last_in&COMPLETE))
269                         ipq_kill(qp);
270                 spin_unlock(&qp->lock);
271
272                 ipq_put(qp, &work);
273                 IP_INC_STATS_BH(IPSTATS_MIB_REASMFAILS);
274         }
275 }
276
277 static inline void ip_evictor(void)
278 {
279         __ip_evictor(sysctl_ipfrag_low_thresh);
280 }
```

36.2.2.3 Deuxième étape : consultation du cache de fragment

La fonction `ip_find()` cherche une l'entrée de cache correspondant aux quatre clés (identificateur, adresse source, adresse de destination et protocole) contenues dans l'en-tête IP passé en argument. La valeur de hachage de l'entrée est calculée avec `ipqhashfn()`. Les collisions sont résolues dans une liste doublement chaînée. Si on ne trouve pas d'entrée convenant au fragment, une nouvelle entrée est créée dans le cache de fragment grâce à la fonction `ip_frag_create()` et c'est celle-ci qui est renvoyée.

La fonction de recherche `ip_find()` est définie dans le fichier `linux/net/ipv4/ip_fragment.c`:

Code Linux 2.6.10

```
386 /* Cherche l'entree correcte dans la file d'attente des "datagrammes incomplets" pour
387  * ce datagramme IP. On en cree une nouvelle si rien n'est trouve.
388  */
389 static inline struct ipq *ip_find(struct iphdr *iph)
390 {
391         __u16 id = iph->id;
392         __u32 saddr = iph->saddr;
393         __u32 daddr = iph->daddr;
394         __u8 protocol = iph->protocol;
395         unsigned int hash = ipqhashfn(id, saddr, daddr, protocol);
396         struct ipq *qp;
397
398         read_lock(&ipfrag_lock);
399         for(qp = ipq_hash[hash]; qp; qp = qp->next) {
400                 if(qp->id == id         &&
401                    qp->saddr == saddr   &&
402                    qp->daddr == daddr   &&
403                    qp->protocol == protocol) {
404                         atomic_inc(&qp->refcnt);
405                         read_unlock(&ipfrag_lock);
406                         return qp;
407                 }
408         }
409         read_unlock(&ipfrag_lock);
410
411         return ip_frag_create(hash, iph);
412 }
```

La fonction de hachage `ipqhashfn()` est définie dans le même fichier :

Code Linux 2.6.10

```
103 static u32 ipfrag_hash_rnd;
```

```
[...]
123 static unsigned int ipqhashfn(u16 id, u32 saddr, u32 daddr, u8 prot)
124 {
125         return jhash_3words((u32)id << 16 | prot, saddr, daddr,
126                             ipfrag_hash_rnd) & (IPQ_HASHSZ - 1);
127 }
```

La fonction de création d'une nouvelle entrée `ip_frag_create()` est également définie dans le même fichier :

Code Linux 2.6.10

```
354 /* Ajoute une entree a la file d'attente 'ipq' pour un datagramme IP nouvellement recu. */
355 static struct ipq *ip_frag_create(unsigned hash, struct iphdr *iph)
356 {
357         struct ipq *qp;
358
359         if ((qp = frag_alloc_queue()) == NULL)
360                 goto out_nomem;
361
362         qp->protocol = iph->protocol;
363         qp->last_in = 0;
364         qp->id = iph->id;
365         qp->saddr = iph->saddr;
366         qp->daddr = iph->daddr;
367         qp->len = 0;
368         qp->meat = 0;
369         qp->fragments = NULL;
370         qp->iif = 0;
371
372         /* Initialise un minuteur pour cette entree. */
373         init_timer(&qp->timer);
374         qp->timer.data = (unsigned long) qp;    /* pointeur sur la file d'attente */
375         qp->timer.function = ip_expire;         /* routine d'expiration           */
376         spin_lock_init(&qp->lock);
377         atomic_set(&qp->refcnt, 1);
378
379         return ip_frag_intern(hash, qp);
380
381 out_nomem:
382         NETDEBUG(if (net_ratelimit()) printk(KERN_ERR "ip_frag_create: no memory left !\n"));
383         return NULL;
384 }
```

La fonction en ligne `frag_alloc_queue()` d'allocation d'une nouvelle entrée de file d'attente est également définie dans le même fichier :

Code Linux 2.6.10

```
188 static __inline__ struct ipq *frag_alloc_queue(void)
189 {
190         struct ipq *qp = kmalloc(sizeof(struct ipq), GFP_ATOMIC);
191
192         if(!qp)
193                 return NULL;
194         atomic_add(sizeof(struct ipq), &ip_frag_mem);
195         return qp;
196 }
```

Le traitement lors de l'expiration du délai, contenu dans la fonction `ip_expire()`, sera étudié plus bas.

La fonction `ip_frag_intern()` d'initialisation d'une nouvelle entrée de file d'attente est également définie dans le même fichier :

Code Linux 2.6.10

```
312 /* Primitives de creation. */
313
```

```
314 static struct ipq *ip_frag_intern(unsigned int hash, struct ipq *qp_in)
315 {
316         struct ipq *qp;
317
318         write_lock(&ipfrag_lock);
319 #ifdef CONFIG_SMP
320         /* Avec la course SMP nous avons a verifier a nouveau la table de hachage, puisque
321          * une telle entree aurait pu etre creee sur un autre cpu, bien que le
322          * verrou d'ecriture etait positionne en verrou d'ecriture.
323          */
324         for(qp = ipq_hash[hash]; qp; qp = qp->next) {
325                 if(qp->id == qp_in->id          &&
326                    qp->saddr == qp_in->saddr    &&
327                    qp->daddr == qp_in->daddr    &&
328                    qp->protocol == qp_in->protocol) {
329                         atomic_inc(&qp->refcnt);
330                         write_unlock(&ipfrag_lock);
331                         qp_in->last_in |= COMPLETE;
332                         ipq_put(qp_in, NULL);
333                         return qp;
334                 }
335         }
336 #endif
337         qp = qp_in;
338
339         if (!mod_timer(&qp->timer, jiffies + sysctl_ipfrag_time))
340                 atomic_inc(&qp->refcnt);
341
342         atomic_inc(&qp->refcnt);
343         if((qp->next = ipq_hash[hash]) != NULL)
344                 qp->next->pprev = &qp->next;
345         ipq_hash[hash] = qp;
346         qp->pprev = &ipq_hash[hash];
347         INIT_LIST_HEAD(&qp->lru_list);
348         list_add_tail(&qp->lru_list, &ipq_lru_list);
349         ip_frag_nqueues++;
350         write_unlock(&ipfrag_lock);
351         return qp;
352 }
```

36.2.2.4 Troisième étape : insertion d'un fragment

La fonction `ip_frag_queue()` insère un fragment nouvellement arrivé dans la file d'attente des fragments d'un datagramme IP, représenté à l'aide d'un descripteur de fragment. On vérifie d'abord si le paquet n'est pas déjà complet et si le fragment ne serait pas un simple doublon. Si tel n'est pas le cas, les positions (`offset` et `end`) du fragment dans le paquet d'origine sont calculées grâce à l'en-tête IP. On vérifie ensuite, au moyen du drapeau `MF` s'il s'agit du dernier fragment du paquet et dans ce cas on positionne `LAST_IN`. Puis on recherche la position correcte du fragment dans la liste des fragments reçus jusqu'ici et le tampon de socket est placé à cette position. Le paramètre `meat` du descripteur de fragment du paquet est augmenté de la longueur du fragment qui vient d'être inséré.

Code Linux 2.6.10

La fonction `ip_frag_queue()` est définie dans le fichier `linux/net/ipv4/ip_fragment.c`:

```
414 /* Ajoute un nouveau segment a une file d'attente existante. */
415 static void ip_frag_queue(struct ipq *qp, struct sk_buff *skb)
416 {
417         struct sk_buff *prev, *next;
418         int flags, offset;
419         int ihl, end;
```

```
420
421         if (qp->last_in & COMPLETE)
422                 goto err;
423
424         offset = ntohs(skb->nh.iph->frag_off);
425         flags = offset & ~IP_OFFSET;
426         offset &= IP_OFFSET;
427         offset <<= 3;           /* le deplacement est dans l'emplacement 8-octet */
428         ihl = skb->nh.iph->ihl * 4;
429
430         /* Determine la position ce ce fragment. */
431         end = offset + skb->len - ihl;
432
433         /* Est-ce le dernier fragment ? */
434         if ((flags & IP_MF) == 0) {
435                 /* Si nous avons deja quelques bits au-dela de la fin
436                  * ou si nous avons une fin differente, le segment est corrompu.
437                  */
438                 if (end < qp->len ||
439                     ((qp->last_in & LAST_IN) && end != qp->len))
440                         goto err;
441                 qp->last_in |= LAST_IN;
442                 qp->len = end;
443         } else {
444                 if (end&7) {
445                         end &= ~7;
446                         if (skb->ip_summed != CHECKSUM_UNNECESSARY)
447                                 skb->ip_summed = CHECKSUM_NONE;
448                 }
449                 if (end > qp->len) {
450                         /* Quelques bits au-dela de la fin -> corrompu. */
451                         if (qp->last_in & LAST_IN)
452                                 goto err;
453                         qp->len = end;
454                 }
455         }
456         if (end == offset)
457                 goto err;
458
459         if (pskb_pull(skb, ihl) == NULL)
460                 goto err;
461         if (pskb_trim(skb, end-offset))
462                 goto err;
463
464         /* Trouver quels fragments sont avant nous et lesquels sont apres nous
465          * dans la chaine des fragments. Nous devons savoir ou inserer
466          * ce fragment, n'est-ce pas ?
467          */
468         prev = NULL;
469         for(next = qp->fragments; next != NULL; next = next->next) {
470                 if (FRAG_CB(next)->offset >= offset)
471                         break;  /* bingo! */
472                 prev = next;
473         }
474
475         /* Nous avons trouve ou l'inserer. Verifions le chevauchement avec
476          * le fragment precedent, et, si necessaire, alignons les choses pour que
477          * tout chevauchement soit elimine.
478          */
479         if (prev) {
480                 int i = (FRAG_CB(prev)->offset + prev->len) - offset;
481
```

```
482                 if (i > 0) {
483                         offset += i;
484                         if (end <= offset)
485                                 goto err;
486                         if (!pskb_pull(skb, i))
487                                 goto err;
488                         if (skb->ip_summed != CHECKSUM_UNNECESSARY)
489                                 skb->ip_summed = CHECKSUM_NONE;
490                 }
491         }
492
493         while (next && FRAG_CB(next)->offset < end) {
494                 int i = end - FRAG_CB(next)->offset; /* chevauchement de 'i' octets */
495
496                 if (i < next->len) {
497                         /* Mangeons le debut du fragment suivant chevauche et conservons
498                          * le reste. Les suivants ne peuvent pas etre chevauches.
499                          */
500                         if (!pskb_pull(next, i))
501                                 goto err;
502                         FRAG_CB(next)->offset += i;
503                         qp->meat -= i;
504                         if (next->ip_summed != CHECKSUM_UNNECESSARY)
505                                 next->ip_summed = CHECKSUM_NONE;
506                         break;
507                 } else {
508                         struct sk_buff *free_it = next;
509
510                         /* Le vieux fragment est completement chevauche par
511                          * le nouveau. Ecartons-le.
512                          */
513                         next = next->next;
514
515                         if (prev)
516                                 prev->next = next;
517                         else
518                                 qp->fragments = next;
519
520                         qp->meat -= free_it->len;
521                         frag_kfree_skb(free_it, NULL);
522                 }
523         }
524
525         FRAG_CB(skb)->offset = offset;
526
527         /* Insere ce fragment dans la chaine des fragments. */
528         skb->next = next;
529         if (prev)
530                 prev->next = skb;
531         else
532                 qp->fragments = skb;
533
534         if (skb->dev)
535                 qp->iif = skb->dev->ifindex;
536         skb->dev = NULL;
537         qp->stamp = skb->stamp;
538         qp->meat += skb->len;
539         atomic_add(skb->truesize, &ip_frag_mem);
540         if (offset == 0)
541                 qp->last_in |= FIRST_IN;
542
543         write_lock(&ipfrag_lock);
```

```
544         list_move_tail(&qp->lru_list, &ipq_lru_list);
545         write_unlock(&ipfrag_lock);
546
547         return;
548
549 err:
550         kfree_skb(skb);
551 }
```

La macro FRAG_CB() est définie dans le même fichier : Code Linux 2.6.10

```
64 struct ipfrag_skb_cb
65 {
66         struct inet_skb_parm    h;
67         int                     offset;
68 };
69
70 #define FRAG_CB(skb)    ((struct ipfrag_skb_cb*)((skb)->cb))
```

36.2.2.5 Quatrième étape : réassemblage des fragments

La fonction ip_frag_reasm() permet de reconstituer le paquet lorsque tous les fragments sont arrivés. Pour ce faire, un nouveau descripteur de tampon de socket est d'abord créé avec une zone de données longue de qp->len octets et initialisée avec l'en-tête du paquet IP. Puis les données IP utiles des divers fragments sont copiées dans la zone des données du descripteur de tampon qui vient d'être créé.

La fonction ip_frag_reasm() est définie dans le fichier linux/net/ipv4/ip_fragment.c: Code Linux 2.6.10

```
554 /* Construit un nouveau datagramme IP a partir de tous ses fragments. */
555
556 static struct sk_buff *ip_frag_reasm(struct ipq *qp, struct net_device *dev)
557 {
558         struct iphdr *iph;
559         struct sk_buff *fp, *head = qp->fragments;
560         int len;
561         int ihlen;
562
563         ipq_kill(qp);
564
565         BUG_TRAP(head != NULL);
566         BUG_TRAP(FRAG_CB(head)->offset == 0);
567
568         /* Alloue un nouveau tampon pour le datagramme. */
569         ihlen = head->nh.iph->ihl*4;
570         len = ihlen + qp->len;
571
572         if(len > 65535)
573                 goto out_oversize;
574
575         /* La tete de la liste ne doit pas etre clonee. */
576         if (skb_cloned(head) && pskb_expand_head(head, 0, 0, GFP_ATOMIC))
577                 goto out_nomem;
578
579         /* Si le premier fragment est lui-meme fragmente, nous le divisons
580          * en deux morceaux : le premier avec les donnees et la partie pagee
581          * et le second contenant seulement des fragments. */
582         if (skb_shinfo(head)->frag_list) {
583                 struct sk_buff *clone;
584                 int i, plen = 0;
585
586                 if ((clone = alloc_skb(0, GFP_ATOMIC)) == NULL)
```

```
587                     goto out_nomem;
588             clone->next = head->next;
589             head->next = clone;
590             skb_shinfo(clone)->frag_list = skb_shinfo(head)->frag_list;
591             skb_shinfo(head)->frag_list = NULL;
592             for (i=0; i<skb_shinfo(head)->nr_frags; i++)
593                     plen += skb_shinfo(head)->frags[i].size;
594             clone->len = clone->data_len = head->data_len - plen;
595             head->data_len -= clone->len;
596             head->len -= clone->len;
597             clone->csum = 0;
598             clone->ip_summed = head->ip_summed;
599             atomic_add(clone->truesize, &ip_frag_mem);
600     }
601
602     skb_shinfo(head)->frag_list = head->next;
603     skb_push(head, head->data - head->nh.raw);
604     atomic_sub(head->truesize, &ip_frag_mem);
605
606     for (fp=head->next; fp; fp = fp->next) {
607             head->data_len += fp->len;
608             head->len += fp->len;
609             if (head->ip_summed != fp->ip_summed)
610                     head->ip_summed = CHECKSUM_NONE;
611             else if (head->ip_summed == CHECKSUM_HW)
612                     head->csum = csum_add(head->csum, fp->csum);
613             head->truesize += fp->truesize;
614             atomic_sub(fp->truesize, &ip_frag_mem);
615     }
616
617     head->next = NULL;
618     head->dev = dev;
619     head->stamp = qp->stamp;
620
621     iph = head->nh.iph;
622     iph->frag_off = 0;
623     iph->tot_len = htons(len);
624     IP_INC_STATS_BH(IPSTATS_MIB_REASMOKS);
625     qp->fragments = NULL;
626     return head;
627
628 out_nomem:
629     NETDEBUG(if (net_ratelimit())
630              printk(KERN_ERR
631                     "IP: queue_glue: no memory for gluing queue %p\n",
632                     qp));
633     goto out_fail;
634 out_oversize:
635     if (net_ratelimit())
636             printk(KERN_INFO
637                     "Oversized IP packet from %d.%d.%d.%d.\n",
638                     NIPQUAD(qp->saddr));
639 out_fail:
640     IP_INC_STATS_BH(IPSTATS_MIB_REASMFAILS);
641     return NULL;
642 }
```

36.2.2.6 Cinquième étape : libération de l'entrée du cache de fragment

Code Linux 2.6.10

La fonction en ligne `ipq_put()` est définie dans le fichier `linux/net/ipv4/ip_fragment.c`:

```
222 static __inline__ void ipq_put(struct ipq *ipq, int *work)
```

```
223 {
224         if (atomic_dec_and_test(&ipq->refcnt))
225                 ip_frag_destroy(ipq, work);
226 }
```

qui renvoie à la fonction `ip_frag_destroy()`, définie dans le même fichier : Code Linux 2.6.10

```
199 /* Primitives de destruction. */
200
201 /* Destruction complete de ipq. */
202 static void ip_frag_destroy(struct ipq *qp, int *work)
203 {
204         struct sk_buff *fp;
205
206         BUG_TRAP(qp->last_in&COMPLETE);
207         BUG_TRAP(del_timer(&qp->timer) == 0);
208
209         /* Libere toutes les donnees du fragment. */
210         fp = qp->fragments;
211         while (fp) {
212                 struct sk_buff *xp = fp->next;
213
214                 frag_kfree_skb(fp, work);
215                 fp = xp;
216         }
217
218         /* Enfin libere le descripteur de file d'attente lui-meme. */
219         frag_free_queue(qp, work);
220 }
```

La fonction en ligne `frag_kfree_skb()` est définie dans le même fichier : Code Linux 2.6.10

```
171 /* Fonctions de tracage de la memoire. */
172 static __inline__ void frag_kfree_skb(struct sk_buff *skb, int *work)
173 {
174         if (work)
175                 *work -= skb->truesize;
176         atomic_sub(skb->truesize, &ip_frag_mem);
177         kfree_skb(skb);
178 }
```

ainsi que la fonction en ligne `frag_free_queue()` : Code Linux 2.6.10

```
180 static __inline__ void frag_free_queue(struct ipq *qp, int *work)
181 {
182         if (work)
183                 *work -= sizeof(struct ipq);
184         atomic_sub(sizeof(struct ipq), &ip_frag_mem);
185         kfree(qp);
186 }
```

36.2.3 Traitement lors de l'expiration du délai

La fonction `ip_expire()` est la routine de traitement à l'expiration du délai du minuteur démarré au début du réassemblage des fragments d'un paquet IP. Si tous les fragments ne sont pas arrivés lors de l'expiration de ce minuteur, l'entrée est supprimée dans le cache de fragment. Si tous les fragments ont été reçus, rien n'est entrepris dans cette fonction. Lorsque tous les fragments ne sont pas encore reçus mais que le premier au moins est disponible, un message d'erreur ICMP du type `ICMP_TIME_EXCEEDED/ICMP_EXC_FRAGTIME` est expédié.

Code Linux 2.6.10

La fonction `ip_expire()` est définie dans le fichier `linux/net/ipv4/ip_fragment.c`:

```
282 /*
283  * Oups, une file d'attente de fragments a depasse son temps. Tuons-la et envoyons une
      reponse ICMP.
284  */
285 static void ip_expire(unsigned long arg)
286 {
287         struct ipq *qp = (struct ipq *) arg;
288
289         spin_lock(&qp->lock);
290
291         if (qp->last_in & COMPLETE)
292                 goto out;
293
294         ipq_kill(qp);
295
296         IP_INC_STATS_BH(IPSTATS_MIB_REASMTIMEOUT);
297         IP_INC_STATS_BH(IPSTATS_MIB_REASMFAILS);
298
299         if ((qp->last_in&FIRST_IN) && qp->fragments != NULL) {
300                 struct sk_buff *head = qp->fragments;
301                 /* Envoie un message ICMP "Depassement de duree de reassemblage de
                       fragments". */
302                 if ((head->dev = dev_get_by_index(qp->iif)) != NULL) {
303                         icmp_send(head, ICMP_TIME_EXCEEDED, ICMP_EXC_FRAGTIME, 0);
304                         dev_put(head->dev);
305                 }
306         }
307 out:
308         spin_unlock(&qp->lock);
309         ipq_put(qp, NULL);
310 }
```

Code Linux 2.6.10

La fonction `ipq_kill()` est définie dans le fichier `linux/net/ipv4/ip_fragment.c`:

```
228 /* Tue une entree ipq. Elle n'est pas detruite immediatement,
229  * puisque l'appelant (et quelques autres) y fait encore reference.
230  */
231 static void ipq_kill(struct ipq *ipq)
232 {
233         if (del_timer(&ipq->timer))
234                 atomic_dec(&ipq->refcnt);
235
236         if (!(ipq->last_in & COMPLETE)) {
237                 ipq_unlink(ipq);
238                 atomic_dec(&ipq->refcnt);
239                 ipq->last_in |= COMPLETE;
240         }
241 }
```

La fonction `ipq_unlink()` retire l'entrée du cache de fragment sur lequel pointe le descripteur transmis. Le compteur des fragments `ip_frag_nqueues` est décrémenté. Elle est définie dans le

Code Linux 2.6.10

fichier `linux/net/ipv4/ip_fragment.c`:

```
105 int ip_frag_nqueues = 0;
106
107 static __inline__ void __ipq_unlink(struct ipq *qp)
108 {
109         if(qp->next)
110                 qp->next->pprev = qp->pprev;
111         *qp->pprev = qp->next;
112         list_del(&qp->lru_list);
```

```
113         ip_frag_nqueues--;
114 }
115
116 static __inline__ void ipq_unlink(struct ipq *ipq)
117 {
118         write_lock(&ipfrag_lock);
119         __ipq_unlink(ipq);
120         write_unlock(&ipfrag_lock);
121 }
```

36.3 Initialisation de la fragmentation

La fonction `ipfrag_init()` est définie dans le fichier `linux/net/ipv4/ip_fragment.c`: Code Linux 2.6.10

```
681 void ipfrag_init(void)
682 {
683         ipfrag_hash_rnd = (u32) ((num_physpages ^ (num_physpages>>7)) ^
684                                  (jiffies ^ (jiffies >> 6)));
685
686         init_timer(&ipfrag_secret_timer);
687         ipfrag_secret_timer.function = ipfrag_secret_rebuild;
688         ipfrag_secret_timer.expires = jiffies + sysctl_ipfrag_secret_interval;
689         add_timer(&ipfrag_secret_timer);
690 }
```

Septième partie

Appendices

Bibliographie

[3Com501] 3COM CORPORATION, **Etherlink (3C501) Adapter Technical Reference**, Manual Part 6405-00, 1988, 21 p.

[ABR–70] ABRAMSON, N., *The Aloha System-Another Alternative for Computer Communications*, Compte rendu de la conférence **Fall Joint Computer**, conférence AFIPS, p.37, 1970.

[ABR–85] ABRAMSON, N., *Development of the Alohanet*, **IEEE Transactions on Information Theory**, vol.IT–31, March 1985, pp.119–123.

[BAR–60] BARAN, Paul, *Reliable Digital Communications Systems Using Unreliable Network Repeater Nodes*, **RAND Corporation Mathematics Division Report** n^o P-1995, 27 mai 1960.

[BAR–64] BARAN, Paul, *On Distributed Communications Networks*, **IEEE Transactions on Information Theory**, 1er mars 1964.

[BEC–96] BECK, Michael & BÖHME, Harald & DZIADZKA, Mirko & KUNITZ, Ulrich & MAGNUS, Robert & VERWORNER, Dirk, **Linux-Kernel-Programmierung**, Addison-Wesley (Deutschland); traduction anglaise **Linux Kernel Internals**, Addison-Wesley, 1996; second edition, 1998, XVI + 480 p. + CD-ROM; third edition, Addison-Wesley, 2002, XIV + 471 p. + CD-ROM.

[BOV–01] BOVET, Daniel & CESATI, Marco, **Understanding the Linux Kernel: from I/O ports to process management**, O'Reilly, 2001, XVI + 684 p.; traduction française **Le noyau Linux**, O'Reilly, 2001, XVI + 673 p.

[Le plus détaillé des livres sur l'implémentation de Linux avant [CEG–03], le premier à aborder les points concernant l'architecture du microprocesseur (en l'occurrence le `80x86` d'Intel). Le choix du dernier noyau de l'époque, le noyau 2.2, ne permet pas de commenter tous les points essentiels d'un système d'exploitation, ce qui est dommage vu la qualité de ce livre.]

[CAR–98] CARD, Rémy & DUMAS, Éric & MÉVEL, Franck, **Programmation Linux 2.0 : API système et fonctionnement du noyau**, Eyrolles, 1998, XIII + 520 p. + CD-ROM, ISBN 2-212-08932-5; traduction anglaise **The Linux Kernel Book**, Wiley, 1998.

[Comme son nom l'indique, son but est d'aider le programmeur Linux. Il donne cependant des notes sur l'implémentation, en décrivant surtout les structures utilisées (pour le noyau 2.0).]

[CEG–03] CÉGIELSKI, Patrick, **Conception des systèmes d'exploitation : le cas Linux**, Eyrolles, 2003, XIII + 595 p., ISBN 2-212-11360-9; deuxième édition, XIII + 680 p., septembre 2004.

[Prend comme exemple de système d'exploitation le noyau Linux 0.01 dont il commente le code complet.]

[CK–74] CERF, Vincent & KAHN, R., *A Protocol for Packet Network Interconnection*, **IEEE Transactions on Communications Technology**, vol. COM-22, no. 5, 1974, pp. 627–641.

[C-P–02] CROWCROFT, Jon & PHILLIPS, Iain, **TCP/IP and Linux Protocol Implementation**, Wiley, 2002, L + 925 p., ISBN 0-471-40882-4.

[Extrait de code ordonné avec assez peu de commentaires.]

[DAV–00] DAVIS, Martin, **The Universal Computer**, Norton, 2000, XII + 257 p.

[GRA–00] GRAY, Warren W., **Linux Socket Programming By Example**, QUE, 2000, XV + 558 p., ISBN 0-7897-2241-0.

[HER–00] HERRIN, David, **Linux IP Networking: A Guide to the Implementation and Modification of the Linux Protocol Stack**, disponible en ligne :

`http://kernelnewbies.org/documents/ipnetwoorking/linuxipnetworking.html`

[HL–96] HAFNER, Katie & LYON, Matthew, **Where Wizards stay up late**, Simon & Schuster, 1996; traduction française **Les sorciers du Net : les origines de l'Internet**, Calmann-Lévy, 1999, 347 p.

[HUU–03] HUURDEMAN, Anton, **The Worldwide History of Telecommunications**, Wiley, 2003, XX+638 p.

[Très bon travail de synthèse sur les télécommunications avant l'apparition des réseaux informatiques. Ne comprend pas que le système de commutation est intrinsèquement différent pour les réseaux informatique et que ces derniers vont détrôner les premiers (avec la technique de la voix sur IP).]

[IEEE-802] **IEEE Standards for Local and Metropolitan Area Networks: Overview and Architecture**, 1990, 31 p., ISBN 1-55937-052-1. Téléchargeable à :

`http://standards.ieee.org/catalog/olis/802-1990.pdf`

[IEEE-802.2] **IEEE Standards for Local and Metropolitan Area Networks. Part 2: Logical Link Control**, 1998, 253 p., aussi ISO/IEC 8802.2:1998. Téléchargeable à :

`http://grouper.ieee.org/groups/scc32/dsrc/ip/ip_images/802.2-1998.pdf`

[IEEE-802.3] **IEEE Standards for Local and Metropolitan Area Networks. Part 3: Carrier sense multiple access with collision detection (CSMA/CD) access method and physical layer specifications**, 2002, 1 538 p. Téléchargeable à :

`http://standards.ieee.org/getieee802/download/802.3-2002.pdf`

[K-D–01] KIRCH, Olaf & DAWSON, Terry, **Administration réseau sous Linux**, O'Reilly, seconde édition, janvier 2001, 544 p., ISBN : 2-84177-125-3.

[K-R–01] KUROSE, James F. & ROSS, Keith W., **Computer Networking: A Top Down Approach Featuring the Internet**, Addison-Weslay, 2001, XXIV + 712 p., ISBN 0-201-47711-4, third edition, 2004; traduction française de la deuxième édition **Analyse structurée des réseaux**, Pearson, 2003, 900 p, ISBN: 2-7440-7000-9.

[ISO 7498-1] **ISO/IEC 7498 Part 1: The Basic Model**

[La seconde version, celle de 1994, est disponible auprès du site web de ISO :

`http://www.iso.org/iso/en/CatalogueDetailPage.CatalogueDetail?CSNUMBER=20269`

en anglais ou en français, sous forme électronique ou papier, au prix de 160 CHF. Une version texte antérieure peut être chargée gratuitement sur le site web de l'ACM :

`http://www.acm.org/sigs/sigcomm/standards/iso_stds/OSI_MODEL/ISO_IEC_7498-1.TXT`

]

[ISO 7498-2] **ISO/IEC 7498 Part 2: Security Architecture**

[La version de 1989 est disponible auprès du site web de ISO :

`http://www.iso.org/iso/en/CatalogueDetailPage.CatalogueDetail?CSNUMBER=14256`

en anglais ou en français, sous forme électronique ou papier, au prix de 116 CHF.]

[ISO 7498-3] **ISO/IEC 7498 Part 3: Naming and Addressing**

[La version de 1997 est disponible auprès du site web de ISO :

`http://www.iso.org/iso/en/CatalogueDetailPage.CatalogueDetail?CSNUMBER=25022`

en anglais ou en français, sous forme électronique ou papier, au prix de 97 CHF. Une version texte antérieure peut être chargée gratuitement sur le site web de l'ACM :

`http://www.acm.org/sigs/sigcomm/standards/iso_stds/OSI_MODEL/ISO_IEC_7498-3.TXT`

]

[ISO 7498-4] **ISO/IEC 7498 Part 4: Management Framework**

[La version de 1989 est disponible auprès du site web de ISO :

`http://www.iso.org/iso/en/CatalogueDetailPage.CatalogueDetail?CSNUMBER=14258`

en anglais ou en français, sous forme électronique ou papier, au prix de 61 CHF.]

[KLE–61] KLEINROCK, Leonard, *Information Flow in Large Communication Networks*, **RLE Quaterly Progress Reports**, juillet 1961.

[KLE–86] KLEIMAN, S. *Vnodes: An Architecture for Multiple File System Types in Sun Unix*, **Proccedings of the Summer USENIX Conference**, June 1986, pp. 260–269.

[KLE–64] KLEINROCK, Leonard, **Communication Nets: Stochastic Message Flow and Delay**, McGraw-Hill, 1964.

[LEF–89] LEFFLER, Samuel J. & MCKUSICK, Marshall Kirk & KARELS, Michael J. & QUARTERMAN, John S., **The Design and Implementation of the 4.3 BSD UNIX Operating System**, Addison-Wesley, 1989 (reprinted with corrections on October, 1990), 471 p., ISBN 0-201-06196-1.

[MAN–01] MANCILL, Tony, **Linux Routers: A Primer for Network Administrators**, Prentice Hall, 2001.

[McK–96] MCKUSICK, Keith Bostic, Marshall Kirk & KARELS, Michael J. & QUARTERMAN, John S., **The Design and Implementation of the 4.4 BSD UNIX Operating System**, Addison-Wesley, 1996, ISBN 0-201-54979-4; traduction française, Vuibert, 1997, 576 p, 2-84180-142-X.

[M-R–76] METCALFE, R. M. et BOGGS, D. R., *Ethernet: Distributed Packet Switching for Local Computer Networks*, **Communications of the Association for Computing Machinery**, vol. 19, July 1976, pp.395–404.

[PAR–96] PARKER, Timothy, **Teach Yourself TCP/IP in 14 days**, Sams, 1996, ISBN 0-672-30885-1; traduction française **TCP/IP**, Simon & Schuster Macmillan, 1996, VII + 450 p.

[PUJ–04] PUJOLLE, Guy, **Réseaux**, Eyrolles, cinquième édition, 2004, 1094 pages, ISBN : 2-212-11437-0.

[RFC 1] CROCKER, S., **Host Software**, 7 April 1969, 11 p., RFC 1.

[RFC 768] POSTEL, Jon, **User Datagram Protocol**, IETF, 29 August 1980, RFC 768, 3 p. Traduction française par F.G. Fremaux.

[RFC 791] POSTEL, Jon, **Internet Protocol: Darpa Internet Program Protocol Specification**, September 1981, 45 p., RFC 791. Traduction française par F.G. Fremaux.

[RFC 792] POSTEL, Jon, **Internet Control Message Protocol**, September 1981, 21 p., RFC 792. Traduction française.

[RFC 793] POSTEL, Jon, **Transmission Control Protocol**, September 1981, 85 p., RFC 793. Traduction française par F.G. Fremaux.

[RFC 826] PLUMER, David C., **An Ethernet Address Resolution Protocol: Converting Network Protocol Addresses to 48.bit Ethernet Address for Transmission on Ethernet Hardware**, November 1982, RFC 826.

[RFC 1071] BRADEN, R., BORMAN, D. & PARTRIDGE, C., **Computing the Internet Checksum**, IETF, September 1988, 24 p., RFC 1071.

[RFC 1122] BRADEN, R., **Requirements for Internet Hosts – Communication Layers**, October 1989, 116 p., RFC 1122.

[RFC 1141] MALLORY, T. et KULLBERG, A., **Incremental Updating of the Internet Checksum**, IETF, January 1990, 2 p., RFC 1141.

[RFC 1310] CHAPIN, Lyman, **The Internet Standards Process**, March 1992, 23 p., RFC 1310.

[RFC 1518] REKHTER, Yakov et LI, Tony, **An Architecture for IP Address Allocation with CIDR**, IETF, September 1993, 27 p., RFC 1518.

[RFC 1519] FULLER, Vince, LI, Tony, YU, Jessica & VARADHAN, Kannan, **Classless Inter-Domain Routing (CIDR): an Address Assignment and Aggregation Strategy**, IETF, September 1993, 24 p., RFC 1519.

[RFC 1624] RIJSINGHANI, Anil, **Computation of the Internet Checksum via Incremental Update**, IETF, May 1994, 6 p., RFC 1624.

[RFC 1700] REYNOLDS, J & POSTEL, Jon, **Assigned Numbers**, October 1994, 230 p., RFC 1700.

[RFC 1812] BAKER, Fred, **Requirements for IP Version 4 Routers**, June 1995, 175 p., RFC 1812.

[RFC 1889] SCHULZRINNE, H., FREDERIK, R. & JACOBSON, V., **RTP: A Transport Protocol for Real-Time Applications**, January 1996, 75 p., RFC 1889.

[RFC 2292] STEVENS, W. & THOMAS, M., **Advanced Sockets API for IPv6**, February 1998, 67 p., RFC 2292.

[RFC 2474] NICHOLS, K. & BLAKE, S. & BAKER, F. & BLACK, D. **Definition of the Differentiated Services Field (DS Field) in the IPv4 and IPv6 Headers**, December 1998, 20 p., RFC 2474.

[SAT–00] SATCHELL, Stephen T., **Linux IP Stacks Commentary**, Coriolis Open Press, 2000.

[STE–90] STEVENS, W. Richard, **Unix Network Programming**, Prentice-Hall, 1990, XI + 772 p., second edition in two volumes in 1999, third edition in two volumes.

[TAN–81] TANENBAUM, Andrew, **Computer Networks**, Prentice-Hall, 1981, second edition, 1988, fourth edition, 2002; traduction française de la quatrième édition **Réseaux**, Pearson, 2003, XI + 908 p., ISBN : 2-7440-7001-7.

[TJ–97] TISCHER, Michael & JENNRICH, Bruno, **Internet Bible**, Data Becker, 1997; traduction française **La Bible Internet : expertise et programmation**, Micro Application, 1997, 1545 p. + CD-ROM, ISBN 2-7429-0762-9.

[UFF–87] UFFENBECK, John, **The 80x86 Family: Design, Programming, and Interfacing**, Prentice-Hall, 1987, 1998, third edition 2002, IX + 678 p. + CD-ROM.

[WPRMB–02] WEHRLE, Klaus & PHLKE, Frank & RITTER, Hartmut & MÜLLER, Daniel & BECHLER, Marc, **Linux Netzwerkarchitektur**, Addison-Wesley Verlag, 2002; traduction française **Architecture réseau Linux : conception et implémentation des protocoles réseau du noyau Linux**, Vuibert Informatique, 2003, XV + 726 p.

[Le but de l'ouvrage est de documenter l'API des réseaux de Linux pour implémenter de nouveaux protocoles et de nouveaux périphériques. Ce faisant, un éclairage fort intéressant sur la conception, sans entrer dans le détail du code, est fourni.]

Index

_SK_MEM_OVERHEAD, 412
_SK_MEM_PACKETS, 412
__BIG_ENDIAN, 227
__BIG_ENDIAN_BITFIELD, 227
__LINK_STATE_LINKWATCH_PENDING, 207
__LINK_STATE_NOCARRIER, 207
__LINK_STATE_PRESENT, 207
__LINK_STATE_RX_SCHED, 207
__LINK_STATE_SCHED, 207
__LINK_STATE_START, 207
__LINK_STATE_XOFF, 207
__LITTLE_ENDIAN, 227
__LITTLE_ENDIAN_BITFIELD, 227
___pskb_trim(), 195
__dev_alloc_skb(), 197
__dev_get_by_index(), 247
__dev_get_by_name(), 237
__dev_put(), 614
__dev_remove_pack(), 308
__in_dev_get(), 396
__init, 139
__initcall(), 140
__ip_evictor(), 635
__ip_route_output_key(), 572
__ip_select_ident(), 584
__kernel_caddr_t, 122
__kernel_clock_t, 122
__kernel_daddr_t, 122
__kernel_gid16_t, 122
__kernel_gid_32_t, 122
__kernel_gid_t, 122
__kernel_ino_t, 122
__kernel_ipc_pid_t, 122
__kernel_loff_t, 122
__kernel_mode_t, 122
__kernel_nlink_t, 122
__kernel_off_t, 122
__kernel_old_dev_t, 122
__kernel_old_gid_t, 122
__kernel_old_uid_t, 122
__kernel_pid_t, 122
__kernel_ptrdiff_t, 122
__kernel_size_t, 122
__kernel_ssize_t, 122
__kernel_suseconds_t, 122
__kernel_time_t, 122
__kernel_uid16_t, 122
__kernel_uid32_t, 122
__kernel_uid_t, 122
__kfree_skb(), 164
__lock_sock(), 418
__netdev_watchdog_up(), 285
__netif_rx_schedule(), 309
__netif_schedule(), 601
__pskb_pull(), 190
__pskb_pull_tail(), 191
__pskb_trim(), 195
__raise_softirq_irqoff(), 310
__release_sock(), 416
__s16, 123
__s32, 123
__s64, 123
__s8, 123
__sk_add_node(), 517
__sk_dst_reset(), 505
__sk_dst_set(), 553
__sk_head(), 401
__skb_append(), 179
__skb_dequeue(), 176
__skb_dequeue_tail(), 177
__skb_insert(), 178
__skb_linearize(), 592
__skb_pull(), 169
__skb_queue_head(), 174
__skb_queue_purge(), 178
__skb_queue_tail(), 175
__skb_trim(), 195
__skb_unlink(), 177
__sock_create(), 487
__sock_recvmsg(), 524
__sock_sendmsg(), 543

__u16, 123
__u32, 123
__u64, 123
__u8, 123
__udp_checksum_complete(), 438
10Base-F, 46
10Base2, 46
10Base5, 46
3C501.h
    l.54, 270
    l.61–70, 270
    l.72–79, 270
    l.80, 270
    l.82–88, 270
    l.90–92, 270
3Com, 33
3Com 501, 93
3c501.c
    l.1–100, 268
    l.139–201, 271
    l.203–325, 274
    l.327–361, 286
    l.363–387, 610
    l.390–494, 604
    l.496–686, 292
    l.548–633, 607
    l.635–673, 294
    l.689–754, 296
    l.756–786, 287
    l.788–816, 614
    l.818–833, 318
    l.885–889, 278
    l.891–898, 232
3c501.h
    l.102, 277
    l.103–107, 277
    l.18, 273
    l.20–24, 277
    l.26–38, 277
    l.41–53, 269
    l.56–59, 269
802/, 119

1822
    protocole, 16

abonné
    au téléphone, 4
accept(), 76
ACK, 39
acknowledge, 39
activité, 129
adaptateur réseau, 93
add_timer(), 128
adjacentes (machines), 35
adresse
    de sous-réseau, 106
    générique, 72
    Internet, 61
        par défaut, 73
    IP, 59, 61
        default, 326
    MAC, 100
    Unix, 72
AF_APPLETALK, 67, 442
AF_ASH, 442
AF_ATMPVC, 67, 442
AF_ATMSVC, 442
AF_AX25, 67, 442
AF_BLUETOOTH, 442
AF_BRIDGE, 442
AF_DECnet, 442
AF_ECONET, 442
AF_INET, 67, 442
af_inet.c
    l.1017–1122, 453
    l.1045–1049, 444
    l.1051–1064, 399
    l.1124, 453
    l.1139–1166, 457
    l.124–127, 456
    l.128, 453
    l.170–189, 545
    l.226–350, 495
    l.353–383, 484
    l.385–467, 511
    l.649–659, 544
    l.676–728, 502
    l.803–822, 480
    l.849–853, 443
    l.859–894, 448
    l.874–882, 447
    l.896, 456
    l.898–952, 452
    l.969–989, 449
    l.981–985, 398
    l.991–1012, 456
AF_INET6, 67, 442
AF_IPX, 67, 442
AF_IRDA, 442

AF_KEY, 442
AF_LLC, 442
AF_LOCAL, 67, 442
AF_MAX, 442
AF_NETBEUI, 442
AF_NETLINK, 67, 442
AF_NETROM, 442
AF_PACKET, 67, 442
AF_PPPOX, 442
AF_ROSE, 442
AF_ROUTE, 442
AF_SECURITY, 442
AF_SNA, 442
AF_UNIX, 67, 442
AF_UNSPEC, 442
AF_WANPIPE, 442
AF_X25, 67, 442
aio.c
    l.311–324, 523
aio.h
    l.46-78, 522
    l.81–94, 523
alloc_divert_blk(), 245
alloc_etherdev(), 262
alloc_netdev(), 263
alloc_skbuff(), 161
ALOHANET, 32
ancillary data, 87
anneau à jeton, 48
API
    réseau, v
appel système, 129
appletalk/, 119
application layer, 10
arch/, 118
architecture
    hybride, 11
    Internet, 55
    réseau, 7
    TCP/IP, 55
ARCNET, 33
ARP, 60
arp.c
    l.1235–1246, 456
    l.138–156, 588
arp_generic_ops, 588
arp_hh_ops, 588
arp_init(), 456
ARPA, 15
ARPANET, 15
arphdr (struct), 154
ARPHRD_CSLIP, 216
ARPHRD_ETHER, 216
ARPHRD_IEEE802, 216
ARPHRD_LOOPBACK, 216
ARPHRD_PPP, 216
ARPHRD_SLIP, 216
ARPHRD_VOID, 216
ASSERT_RTNL(), 243
AT&T, 14
ATM, 59
atm/, 119
atomic.h
    l.19–24, 133
    l.26–58, 134
    l.8–11, 133
atomic_add(), 134
atomic_dec(), 134
atomic_inc(), 134
ATOMIC_INIT(), 134
atomic_read(), 134
atomic_set(), 134
atomic_sub(), 134
atomic_t, 133
atomicité, 132
ax25.h, 73
ax25/, 119
AX_CMD, 269
AX_LOOP, 270
AX_OFF, 270
AX_RESET, 270
AX_RX, 270
AX_STATUS, 269
AX_SYS, 270
AX_XMIT, 270

backbone, 25, 29
backlog, 301
backplane, 30
BADCLASS(), 393
bande passante, 42
Baran, Paul, 6, 14
base de données
    de redirection, 327
BBN, 16
Bell Telephone Company, 4
Bell, Alexander Graham, 3
BH, 131
big_endian.h
    l.1–9, 227

bind(), 76
Biro, Ross, 144
bit
    d'arrêt, 23
    de début, 23
BITNET, 30
bitops.h, 133
BITS_PER_LONG, 123
Bluetooth, 25, 48
bluetooth/, 119
bottom half, 131
bps, 23
bridge/, 119
broadcast, 26, 100
BSD, 56
BUG(), 165
bug.h
    l.21–23, 168
    l.25–32, 166
    l.6–20, 165
BUG_ON(), 168
BUG_TRAP(), 186
bus, 29
bus à jeton, 48
BUS_ID_SIZE, 252
byteorder.h/asm-i386
    l.1–59, 227

câble
    de descente, 30
    Ethernet
      fin, 28
    transcepteur, 47
cache
    d'en-têtes matériel, 199
    de fragment, 632
    de routage, 363
cafteur, 525
canal, 42
    à accès aléatoire, 43
    à accès multiple, 43
    à contention, 42
    bruité, 35
    parfait, 35
CAP_NET_ADMIN, 446
CAP_NET_BIND_SERVICE, 446
CAP_NET_BROADCAST, 446
CAP_NET_RAW, 446
capability.h
    l.141–169, 446
carte
    Ethernet, 47
    réseau, 47, 93
central téléphonique, 4
Cerf, Vincent, 33, 56
chaîne
    de notification, 239
change_bit(), 132
change_nexthops(), 345
checksum, 36
checksum.h
    l.54–91, 320
    l.6–18, 422
    l.93–135, 432
checksum.h/net
    l.60–64, 427
    l.71–77, 427
checksum.S
    l.6–252, 422
CHECKSUM_HW, 155
CHECKSUM_NONE, 155
CHECKSUM_UNNECESSARY, 155
CIDR, 107
circuit, 4
circuit switching, 4
clé, 343
class_device (struct), 222
class_device_create_file(), 252
class_device_register(), 252
class_device_unregister(), 252
close(), 69
CMSG_ALIGN(), 90
CMSG_DATA(), 90
CMSG_FIRSTHDR(), 90
CMSG_LEN(), 90
CMSG_NXTHDR(), 90
CMSG_SPACE(), 90
cmsghdr (struct), 89, 520
codage
    binaire simple, 24
    Manchester, 24
collision, 42
commande
    ifconfig, 97, 206, 215, 218
    ip, 324, 327
    route, 324
communication
    connectée, 10, 66
    non connectée, 66
commutation

automatique, 4
de circuits, 4
de message, 5
manuelle, 4
par paquets, 5
compare_keys(), 375
compiler.h
l.54–61, 166
COMPLETE, 633
concentrateur, 30
CONFIG_INET, 327
CONFIG_IP_ADVANCED_ROUTER, 327
CONFIG_IP_MULTIPLE_TABLES, 327
CONFIG_IP_ROUTE_VERBOSE, 393
CONFIG_NETLINK, 327
CONFIG_RTNETLINK, 327
confirmation, 39
connect(), 74
connecteur
BNC, 28, 46
connection_based(), 533
connexion, 62
à distance, 60
distribuée, 60
container_of(), 477
contrôle
de flux
avec retour d'information, 41
basé sur le débit, 41
contrôle des flux, 35
cookie, 525
copy_from_user(), 126
copy_skb_header(), 187
copy_to_user(), 126
cordon
AUI, 47
core/, 118
couche, 7
(sous-) 2a, 48
(sous-) 2b, 48
(sous-) LLC, 48
(sous-) MAC, 48
(sous-) de contrôle d'accès au canal, 48
d'application, 10
de liaison, 9
de présentation, 10
de réseau, 9
de session, 10
de transport, 9
inférieure, 9
Internet, 11
physique, 9
supérieure, 9
courrier électronique, 61
Cox, Alan, 145
CRC, 101
credentials, 87
Crocker, 57
crypto/, 118
CSMA, 43
CSNET, 30
CSP, 43
csum_add(), 427
csum_block_add(), 427
csum_fold(), 432
csum_partial(), 422
csum_partial_copy_fromiovecend(), 564
csum_tcpudp_magic(), 432
csum_tcpudp_nofold(), 432
CTSS, 13
current_text_addr(), 168

débit, 23
délai, 40
démon
réseau, 74
démultiplexage, 10
DARPA, 16
data
rate, 23
data link layer, 9
datagram.c
l.116–195, 529
l.197–200, 537
l.215–297, 533
l.390–436, 535
l.58–64, 533
l.66–114, 531
datagram_poll(), 480
datagramme, 58
DATAPORT, 269
Davies, Donald Watts, 14
daytime, 74
dcache.h
l.111–118, 466
l.27–38, 473
l.83–109, 466
DECLARE_MUTEX(), 138
decnet/, 119
default_rebuild_header(), 244

DEFINE_SNMP_STAT(), 380
DEFINE_WAIT(), 532
del_timer(), 128
deliver_skb(), 317
dentry, 461
dentry (struct), 466
dentry_operations (struct), 466
descripteur
    de couche transport, 403
    de fichier, 461
    de nœud d'information, 460
    de protocole, 398
    de socket, 468
    de tampon de socket, 147
dev.c
    l.1004–1015, 415
    l.1017–1025, 602
    l.1027–1070, 602
    l.1072–1106, 594
    l.1108–1129, 593
    l.1133–1189, 592
    l.1218–1333, 590
    l.129–159, 305
    l.1340, 301
    l.1341, 302
    l.1351, 599
    l.1405–1481, 302
    l.1498–1506, 316
    l.1554–1559, 317
    l.1625–1714, 314
    l.166–184, 234
    l.1716–1762, 312
    l.1764-1805, 311
    l.185, 200
    l.187, 234
    l.192–200, 238
    l.207–211, 239
    l.2151–2161, 219
    l.2176–2188, 218
    l.236–240, 599
    l.242–284, 307
    l.2656–2672, 246
    l.2674, 243
    l.2676–2677, 250
    l.2678–2685, 245
    l.2687–2816, 241
    l.2730–2783, 619
    l.2887–2892, 453
    l.2892, 250
    l.2893–2958, 249
    l.290–344, 308
    l.2960–2985, 274
    l.2994–3079, 615
    l.3153–3155, 307
    l.3163–3181, 301
    l.3190–3191, 310
    l.346–353, 260
    l.382–407, 273
    l.410–439, 258
    l.476–498, 237
    l.523–545, 247
    l.640–653, 244
    l.655–718, 236
    l.799–805, 244
    l.808–873, 281
    l.875–934, 612
    l.900–918, 232
dev_3c501, 232
dev_activate(), 283
dev_add_pack(), 307
dev_alloc_name(), 236
dev_alloc_skb(), 197
dev_base, 200
dev_base_lock, 234
dev_boot_phase, 243
dev_boot_setup[], 260
dev_close(), 612
dev_deactivate(), 614
dev_get_index(), 208
dev_hold(), 245
dev_index_head[], 238
dev_init_scheduler(), 247
dev_kfree_skb(), 166
dev_mc_discard(), 618
dev_mc_list (struct), 218
dev_mcast.c
    l.202–220, 618
dev_name_hash(), 238
dev_name_head[], 238
dev_new_index(), 246
dev_open(), 281
dev_put(), 618
dev_queue_xmit(), 590
dev_queue_xmit_nit(), 602
dev_remove_pack(), 308
dev_set_allmulti(), 218
dev_set_promiscuity(), 219
dev_shutdown(), 617
dev_valid_name(), 244
dev_watchdog(), 609

dev_watchdog_down(), 614
dev_watchdog_init(), 249
dev_watchdog_up(), 285
device.h
    l.28, 252
device_initcall(), 230
devinet.c
    l.1482–1492, 371
devinet_init(), 371
devprobe2 (struct), 261
diffusion
    générale, 100
    individuelle, 27
    multidestinataire, 100
    multidiffusion, 100
    restreinte, 100
divert.h
    l.111–130, 317
    l.14–34, 222
divert_blk (struct), 222
divert_frame(), 317
DIVERT_PROTO_ICMP, 222
DIVERT_PROTO_IP, 222
DIVERT_PROTO_NONE, 222
DIVERT_PROTO_TCP, 222
DIVERT_PROTO_UDP, 222
DNS, 61
do_softirq(), 131
Documentation/, 118
donnée
    ancillaire, 87
    auxiliaire, 87
    de contrôle, 87
données
    de niveau n, 7
dorsale, 25, 29, 30
down(), 138
drivers/, 118
drop, 30
DRV_NAME, 277
dst.c
    l.115–138, 394
dst.h
    l.1–9, 364
    l.137–143, 183
    l.145–152, 165
    l.188–192, 554
    l.219–232, 585
    l.234–248, 397
    l.29–80, 364
    l.83–101, 366
dst_alloc(), 394
dst_clone(), 183
dst_confirm(), 554
dst_entry (struct), 364
DST_HOST, 365
dst_input(), 397
DST_NOHASH, 365
DST_NOPOLICY, 365
DST_NOXFRM, 365
dst_ops (struct), 366
dst_output(), 585
dst_release(), 165
DT_SOCK, 466
durée
    de vie, 103
dv.c
    l.48–76, 245
    l.6–12, 245
    l.78–94, 618

EACCES, 68, 513, 553
EADDRINUSE, 513
EADDRNOTAVAIL, 513
EAFNOSUPPORT, 71, 489
EAGAIN, 277, 528, 531, 545, 563
EBADF, 76, 84, 502, 507
ECMA, 3
econet/, 119
EDSTADDRREQ, 552
EEXIST, 244, 344, 346, 451
EFAULT, 186, 535, 536, 561
EINVAL, 68, 71, 237, 244, 343, 351, 362, 392, 394, 396, 455, 488, 503, 513, 536, 552, 554, 579, 588
EIO, 244
EIOCBQUEUED, 523, 543
el1_close(), 614
EL1_DATAPORT, 269
EL1_DATAPTR, 269
el1_get_stats(), 318
EL1_IO_EXTENT, 273
el1_probe(), 271
el1_probe1(), 274
EL1_RXPTR, 269
EL1_SAPROM, 269
el_debug, 277
el_interrupt(), 292
el_open(), 286
el_receive(), 296

el_reset(), 287
el_start_xmit(), 604
el_timeout(), 610
EMFILE, 68
EMSGSIZE, 551, 560, 628
en-queue
    de trame, 36
en-tête
    de message, 88, 520
    de protocole, 7, 12
    de trame, 36
    matériel, 199
    physique, 199
    pseudo, 112
endfor_nexthops(), 345
ENETUNREACH, 362
ENFILE, 68, 71, 237, 489, 495
Engelbart, Doug, 58
ENOBUFS, 343, 346, 354, 394, 451, 498, 560, 563
ENODEV, 208, 260, 273, 277, 283, 481, 617
ENOENT, 346
ENOMEM, 68, 237, 246, 272, 371, 473, 495, 587, 592, 594, 631
ENOTCOMM, 533
ENOTCONN, 84, 503
ENOTSOCK, 76, 84, 502
ENXIO, 273, 481
EOPNOTSUPP, 76
EPERM, 498
EPIPE, 563
EPROTONOSUPPORT, 68
EPROTONOTSUPPORT, 498
err.h
    l.21–29, 233
    l.8–19, 272
ERR_PTR(), 272
ESOCKNOTSUPPORT, 498
ESPIPE, 479
eth.c
    l.153–209, 298
    l.266–288, 261
    l.291–307, 262
ETH_ALEN, 256
ETH_DATA_LEN, 256
ETH_FRAME_LEN, 256
eth_hdr(), 299
ETH_HLEN, 256
ETH_P_ALL, 307
ETH_P_ARP, 257
ETH_P_IP, 257
ETH_P_LOOP, 257
eth_type_trans(), 298
ETH_ZLEN, 256
ether_setup(), 261
Ethernet, 32, 59
    épais, 46
    fin, 46
    rapide, 28
ethernet/, 119
ethhdr (struct), 257
ethif_probe2(), 258
ethtool.h
    l.16–30, 266
    l.264–354, 264
ethtool_cmd (struct), 266
ethtool_ops (struct), 264

fanion
    de signalisation, 38
Fast Ethernet, 28
fasync_struct (struct), 468
fc_setup(), 261
FDDI, 261
fddi_setup(), 261
fddi_type_trans(), 298
FDM, 42
fdopen(), 77
feed-back based flow control, 41
fenêtre glissante, 40
FIB, 327
fib_alias, 343
fib_create_info(), 348
fib_find_alias(), 344
fib_find_node(), 343
fib_frontend.c
    l.127–151, 513
    l.153–230, 395
    l.50–55, 332
    l.527–586, 333
    l.588–607, 333
    l.99–125, 579
fib_hash.c
    l.143–188, 354
    l.201–241, 347
    l.243–273, 359
    l.384–390, 347
    l.392–405, 343
    l.407–425, 344
    l.427–576, 340

l.48–49, 335
l.51–55, 330
l.57–73, 330
l.579–656, 355
l.75–78, 329
l.799–824, 347
l.826–860, 335
l.90–93, 343
l.95, 346
l.97, 343
fib_hash_alloc(), 352
fib_hash_free(), 352
fib_hash_init(), 335
fib_hash_lock, 346
fib_hash_move(), 352
fib_hash_size, 352
fib_inetaddr_event(), 333
fib_inetaddr_notifier, 333
fib_info (struct), 331
fib_info_cnt, 352
fib_info_lock, 344
fib_info_put(), 345
fib_insert_node(), 347
fib_lookup(), 362
fib_lookup.c
l.8–15, 343
fib_netdev_event(), 333
fib_netdev_notifier, 333
fib_nh (struct), 330
fib_node (struct), 330
fib_props[], 351
fib_release_info(), 344
fib_res_put(), 514
fib_result, 358
fib_rules.c, 327
fib_select_default(), 580
fib_semantic_match(), 360
fib_semantics.c
l.141–156, 345
l.158–174, 344
l.50, 344
l.503–570, 352
l.53–54, 352
l.572–763, 348
l.72–82, 345
l.765–832, 360
l.85–138, 351
fib_table (struct), 328
fib_validate_source(), 395
fichier
descripteur, 461
numéro, 461
file (struct), 465
file.h
l.19–34, 506
file_operations (struct), 464
file_system_type (struct), 467
files_struct (struct), 506
filesystems.c
l.53–84, 467
filp_close(), 507
filter.h
l.21–32, 410
l.40–46, 410
l.48–51, 415
FIRST_IN, 633
flowi (struct), 357
fn_alias_kmem, 335
fn_hash (struct), 329
fn_hash_delete(), 355
fn_hash_insert(), 340
fn_hash_kmem, 335
fn_hash_lookup(), 359
fn_new_zone(), 347
fn_rehash_zone(), 354
fn_zone (struct), 330
forwarding, 105
frag_alloc_queue(), 637
FRAG_CB(), 641
frag_free_queue(), 643
frag_kfree_skb(), 643
fragment, 104
de tampon
liste, 149
tableau, 149
fragment.c
l.644–679, 634
fragmentation, 9, 104
des données, 102
frame, 36, 58
Frame Relay, 59
free_divert_blk(), 618
free_fib_info(), 345
free_irq(), 130
free_netdev(), 274
fs.h, 468
l.1137–1146, 467
l.429–489, 463
l.577–603, 465
l.754–800, 461

l.844–858, 466
l.903–934, 464
l.969–995, 462
fs/, 118
FTP, 56
full_name_hash(), 238
FZ_HASHMASK(), 330
fz_key(), 343
FZ_MASK(), 330
FZ_MASK_DIVISOR, 343

gateway, 26
gen_stats.h
l.16–53, 212
generic.h
l.1–172, 224
generic_shutdown_super(), 474
get_free_pages(), 125
get_sb_pseudo(), 471
get_stats()/loopback.c, 231
getsockopt(), 91
GFP_ATOMIC, 125
GFP_DMA, 125
GFP_KERNEL, 125
GFP_USER, 125
gnet_stats_basic (struct), 212
gnet_stats_queue (struct), 212
gnet_stats_rate_est (struct), 212
GP_HIGH, 269
GP_LOW, 269

handle_bridge(), 317
handle_diverter(), 317
header, 36
headroom, 148
hh_cache (struct), 222
HH_DATA_ALIGN(), 222
HH_DATA_MOD, 222
HH_DATA_OFF(), 222
HIPPI, 156
hippi_setup(), 261
hlist_entry(), 238
hlist_for_each(), 238
hlist_node (struct), 214
hook, 381
hop, 103
htonl(), 73
htons(), 73
HTTP, 61
hub, 30

HW-IRQ, 129

IAB, 58
IANA, 62, 110
ICCB, 58
ICMP, 59, 103
numéro de protocole IP, 110
ICMP_EXC_FRAGTIME, 643
icmp_protocol, 449
icmp_send(), 402
ICMP_TIME_EXCEEDED, 643
identifiant
de système terminal, 106
IEEE, 48
IETF, 58
if.h
l.26, 206
l.29–50, 214
l.52–81, 215
l.84–103, 259
if_arp.h, 154
l.28–89, 216
if_ether.h
l.108–111, 299
l.35–93, 256
l.6–33, 256
l.95–104, 257
IF_GET_IFACE, 215
IF_GET_PROTO, 215
IF_IFACE_E1, 215
IF_IFACE_SYNC_SERIAL, 215
IF_IFACE_T1, 215
IF_IFACE_V24, 215
IF_IFACE_V35, 215
IF_IFACE_X21, 215
if_packet.h
l.22–31, 155
IF_PORT_100BASEFX, 206
IF_PORT_100BASET, 206
IF_PORT_100BASETX, 206
IF_PORT_10BASE2, 206
IF_PORT_10BASET, 206
IF_PORT_AUI, 206
IF_PORT_UNKNOWN, 206
IF_PROTO_CISCO, 215
IF_PROTO_FR, 215
IF_PROTO_FR_ADD_ETH_PVC, 215
IF_PROTO_FR_ADD_PVC, 215
IF_PROTO_FR_DEL_ETH_PVC, 215
IF_PROTO_FR_DEL_PVC, 215

IF_PROTO_FR_ETH_PVC, 215
IF_PROTO_FR_PVC, 215
IF_PROTO_HDLC, 215
IF_PROTO_HDLC_ETH, 215
IF_PROTO_PPP, 215
IF_PROTO_X25, 215
ifconfig, 97, 279
IFF_802_1Q_VLAN, 215
IFF_ALLMULTI, 215
IFF_AUTOMEDIA, 215
IFF_BROADCAST, 215
IFF_DEBUG, 215
IFF_DYNAMIC, 215
IFF_EBRIDGE, 215
IFF_LOOPBACK, 215
IFF_MASTER, 215
IFF_MULTICAST, 215
IFF_NOARP, 215
IFF_NOTRAILERS, 215
IFF_POINTOPOINT, 215
IFF_PORTSEL, 215
IFF_PROMISC, 215
IFF_RUNNING, 215
IFF_SLAVE, 215
IFF_UP, 214
IFF_VOLATILE, 214
ifmap (struct), 259
IFNAMESIZ, 206
igmhdr (struct), 153
IGMP, 60
igmp_protocol, 449
illegal_highdma(), 593
IMP, 15
in-band, 78
in.h, 72, 73
    l.24–49, 445
    l.244–249, 393
    l.246, 386
IN_DEV_FORWARD(), 393
in_dev_get(), 392
in_dev_put(), 395
in_device (struct), 387
in_ifaddr (struct), 387
in_route.h
    l.30, 552
INADDR_ANY, 73
INADDR_LOOPBACK, 73
include/, 118
index
    de périphérique réseau, 200
inet_add_protocol(), 398
inet_addr_type(), 513
inet_autobind(), 545
inet_bind(), 511
inet_create(), 495
inet_dgram_ops, 480
inet_family_ops, 443
inet_init(), 399, 444, 453
inet_opt (struct), 433
inet_protos[], 398
inet_protosw (struct), 445
INET_PROTOSW_PERMANENT, 447
INET_PROTOSW_REUSE, 447
inet_register_protosw(), 452
inet_release(), 484
inet_reset_saddr(), 505
inet_sendmsg(), 544
inet_shutdown(), 502
inet_sk(), 435
inet_sock (struct), 435
inetd, 74
inetdevice.h
    l.143–154, 392
    l.37–59, 387
    l.85–99, 387
inetsw[], 456
inetsw_array[], 448
INETSW_ARRAY_LEN, 456
inetsw_lock, 453
infirmation, 39
init.h
    l.148–156, 140
    l.7–40, 139
    l.78–87, 140
    l.94, 140
    l.97, 140
init/, 118
init_ipv4_mibs(), 456
INIT_LIST_HEAD(), 248
init_module(), 232
init_sync_kiocb(), 523
init_timer(), 128
inode, 460
inode (struct), 463
int16_t, 123
int32_t, 123
int64_t, 123
int8_t, 123
interface, 7
    Netlink, 327

Internet, 26, 33
    layer, 11
internet, 26
Internet Society, 58
interréseau, 26
interrupt.h
    l.108, 310
    l.16–34, 294
    l.79–93, 131
interruption
    matérielle, 129
      lente, 130
      rapide, 130
ioctl(), 215, 221
iovec (struct), 86, 520
iovec.c
    l.147–256, 564
    l.77–99, 535
IP, 59
ip
    commande, 327
ip.h
    l.110–145, 433
    l.147, 583
    l.151–163, 435
    l.155, 379
    l.179–185, 584
    l.236–247, 505
    l.314–321, 498
    l.41, 573
    l.57, 630
    l.75, 632
    l.78–81, 499
    l.88–106, 581
ip.h/linux
    l.167–187, 319
    l.21–37, 320
    l.6–19, 319
ip.h/net
    l.69–73, 320
ip_append_data(), 555
IP_CE, 320
ip_cmsg_recv(), 529
ip_cmsg_send(), 552
ip_copy_metadata(), 629
ip_defrag(), 634
ip_dev_find(), 579
IP_DF, 320
ip_dont_fragment(), 584
ip_evictor(), 635
ip_expire()(), 644
ip_fast_csum(), 320
ip_fib.h, 327
    l.122–148, 328
    l.158–164, 362
    l.166–170, 580
    l.23–40, 336
    l.254–258, 345
    l.260–268, 514
    l.44–59, 330
    l.61–86, 331
    l.93–102, 358
ip_fib_init(), 333
ip_fib_local_table, 332
ip_fib_main_table, 332
ip_find(), 636
ip_finish_output(), 586
ip_finish_output2(), 586
ip_forward(), 397
ip_frag_create(), 637
ip_frag_destroy(), 643
ip_frag_intern(), 637
ip_frag_mem, 634
ip_frag_nqueues, 644
ip_frag_queue(), 638
ip_frag_reasm(), 641
IP_FRAG_TIME, 632
ip_fragment(), 624
ip_fragment.c
    l.103, 636
    l.105–121, 644
    l.123–127, 636
    l.169, 634
    l.171–178, 643
    l.180–186, 643
    l.188–196, 637
    l.199–220, 643
    l.222–226, 642
    l.228–241, 644
    l.243–280, 635
    l.282–310, 644
    l.312–352, 637
    l.354–384, 637
    l.386–412, 636
    l.414–551, 638
    l.46–57, 634
    l.554–642, 641
    l.64–70, 641
    l.681–690, 645
    l.72–94, 633

l.96–101, 633
ip_generic_getfrag(), 564
IP_INC_STATS_BH(), 379
ip_init(), 456
ip_input.c
l.149–153, 379
l.199–264, 399
l.266–283, 397
l.285–355, 382
l.357–429, 378
ip_local_deliver(), 397
ip_local_deliver_finish(), 399
ip_local_error(), 560
IP_MF, 320
ip_mr_input(), 397
IP_OFFSET, 320
ip_options (struct), 581
ip_options.c
l.202–238, 630
ip_options_fragment(), 630
ip_output(), 586
ip_output.c
l.1107–1213, 581
l.118–125, 584
l.1323–1330, 305
l.1336–1346, 456
l.172–214, 586
l.216–225, 586
l.287–295, 586
l.386–418, 629
l.420–667, 624
l.669–684, 564
l.697–971, 555
l.95–100, 585
ip_packet_type, 305
IP_PMTUDISC_DO, 499
IP_PMTUDISC_DONT, 499
IP_PMTUDISC_WANT, 499
ip_push_pending_frames(), 581
ip_rcv(), 378
ip_rcv_finish(), 382
ip_recv_error(), 528
ip_route_input(), 384
ip_route_input_slow(), 388
ip_route_output_flow(), 572
ip_route_output_slow(), 573
ip_rt_bug(), 394
ip_rt_init(), 368
ip_rt_max_size, 371
ip_rt_put(), 554
ip_select_ttl(), 584
ip_send_check(), 585
ip_statistics, 379
ipc/, 118
IPCB(), 630
ipcm_cookie (struct), 551
IPCORK_OPT, 583
ipfrag_hash_rnd, 636
ipfrag_init(), 645
ipfrag_skb_cb, 641
iphdr (struct), 319
IPng, 60
IPPROTO_AH, 445
IPPROTO_COMP, 445
IPPROTO_EGP, 445
IPPROTO_ESP, 445
IPPROTO_GRE, 445
IPPROTO_ICMP, 445
IPPROTO_IDP, 445
IPPROTO_IGMP, 445
IPPROTO_IP, 445
IPPROTO_IPIP, 445
IPPROTO_IPV6, 445
IPPROTO_MAX, 445
IPPROTO_PIM, 445
IPPROTO_PUP, 445
IPPROTO_RAW, 445
IPPROTO_RSVP, 445
IPPROTO_SCTP, 445
IPPROTO_TCP, 71, 445
IPPROTO_UDP, 71, 445
ipq (struct), 633
ipq_hash[], 633
IPQ_HASHZ, 633
ipq_kill(), 644
ipq_put(), 642
ipq_unlink(), 644
ipqhashfn(), 636
IPsec, 60
ipstats_mib (struct), 380
IPSTATS_MIB_INDISCARDS, 380
IPSTATS_MIB_INRECEIVES, 380
IPTOS_LOWDELAY, 320
IPTOS_RT_MASK, 386
IPTOS_THROUGHPUT, 320
IPTOS_TOS(), 320
IPTOS_TOS_MASK, 320
IPv4, 60
ipv4/, 118
ipv4_config, 498

ipv4_config (struct), 498
ipv4_dst_ops, 366
ipv4_proc_init(), 457
IPv6, 60
ipv6.h
    l.292–298, 435
ipv6/, 119
ipv6_only_sock(), 435
ipv6hdr (struct), 153
ipx.h, 73
ipx/, 119
irda/, 119
IRQ_HANDLED, 294
IRQ_NONE, 294
IRQ_RETVAL(), 294
irqreturn_t, 294
IRTF, 58
IS_ERR(), 233
isa_probes[], 261
ISO, 9

Kahn, 56
kern_rta (struct), 336
kernel.h
    l.232–242, 477
kernel/, 118
key/, 119
kfree(), 126
kfree_skb(), 164
kfree_skbmem(), 162
kill_anon_super(), 474
kiocb (struct), 522
kiocb_to_siocb(), 525
Kleinrock, Leonard, 14
kmalloc(), 125
kmap_skb_frag(), 186
kmem_cache_alloc(), 127
kmem_cache_create(), 126
kmem_cache_free(), 127
kmem_cache_t, 127
kstatfs (struct), 475
kunmap_skb_frag(), 186
Kuznetsov, Alexey, 327

LAN, 25
lapb/, 119
LAST_IN, 633
lib/, 118
libfs.c
    l.192–237, 471
    l.21–27, 475
Licklider, Joseph Carl Robnett, 15
likely(), 166
list.h
    l.18–30, 210
    l.298–306, 250
    l.314–321, 498
    l.37–39, 248
    l.427–438, 498
    l.503–505, 214
    l.631–635, 238
    l.70–81, 310
list_add_tail(), 310
list_entry(), 498
list_for_each_rcu(), 498
list_head, 210
list_splice_init(), 250
listen(), 76
little_endian.h
    l.1–9, 227
LL_MAX_HEADER, 223
LL_RESERVED_SPACE(), 560
llc/, 119
llc_rcv(), 299
lock, 134
lock_sock(), 418
loff_t, 125
logiciel, 2
loopback, 73, 230
LOOPBACK(), 393
loopback.c
    l.161–185, 231
    l.198–215, 230
    l.217–231, 231
loopback_dev, 230
loopback_init(), 231
lpr, 72
lsocket.c
    l.276–293, 476
    l.295–299, 477
ltalk_setup(), 261
LXR, 121
lxr, vii

métrique, 281
MAN, 25
man
    byteorder, 73
    cmsg, 90
    ddp(7), 67

ifconfig, 280
ip(4), 73
ip(7), 67
netlink(7), 67
packet(7), 67
recv, 80
sendmsg, 79
unix(7), 67
x25(7), 67
manage.c
l.218–232, 130
l.279–310, 129
Marill, Tom, 2, 13
marque, 23
marqueur
de début, 22
de fin, 22
masque
réseau, 108
matériel, 2
MAX_ADDR_LEN, 217
MAX_DIVERT_PORTS, 222
MAX_INET_PROTOS, 398
MAX_SCHEDULE_TIMEOUT, 418
MAX_SKB_FRAGS, 149
media, 9
medium, 2, 21
memcpy_fromiovecend(), 564
memcpy_toiovec(), 535
message, 58
message switching, 5
Metcalfe, Bob, 32
metric, 281
minuteur
de réassemblage, 104
mm/, 118
modèle
en couches, 7
OSI, 8, 9
réseau, 7
mode
de promiscuité, 142
modpost.h
l.71–81, 139
module, 138
module (struct), 139
module.h
l.302–315, 489
l.317–327, 489
module_init(), 140
module_put(), 489
mot, 108
move_addr_to_kernel(), 510
move_addr_to_user(), 523
MSG_CONFIRM, 79
MSG_CTRUNC, 88
MSG_DONTROUTE, 79
MSG_DONTWAIT, 79
MSG_EOR, 88
MSG_ERRQUEUE, 80, 88
MSG_NOSIGNAL, 79, 80
MSG_OOB, 79, 88
MSG_PEEK, 79
MSG_TRUNC, 80, 88
MSG_WAITALL, 80
msghdr (struct), 88, 520
MTU, 216
multicast, 60
MULTICAST(), 386
MULTICS, 13
multihoming, 108
multiplexage, 10
mutex, 138

NAK, 39
NCP, 13
neigh_confirm(), 554
neigh_ops (struct), 588
neighbour.h
l.316–236, 554
Net-1, 144
Net-2, 145
Net-3, 145
Net-4, 145
net-sysfs.c
l.188–201, 252
l.258–283, 253
l.286–289, 253
l.386–399, 619
l.389–395, 252
net-tools, 279
net.h
l.124–160, 469
l.162–171, 443
l.29, 443
l.49–55, 469
l.61–63, 469
l.65–91, 444
l.93–115, 468
net/, 118

Net/one, 33
NET_BH, 131
net_class, 252
net_class_attributes[], 252
net_dev_init(), 301, 307, 310
net_device (struct), 201
net_device_stats (struct), 209
net_disable_timestamp(), 415
net_families[], 444
net_family_lock, 451
net_family_lockct, 451
net_family_read_lock(), 489
net_family_read_unlock(), 489
net_family_write_lock(), 451
net_family_write_unlock(), 451
net_init.c, 261
    l.108–142, 233
    l.144–149, 615
    l.55–70, 200
    l.73–105, 263
net_local (struct), 210, 277
net_olddevs_init(), 230
net_proto_family (struct), 443
net_protocol (struct), 398
net_rx_action(), 311
NET_RX_BAD, 300
NET_RX_CN_HIGH, 300
NET_RX_CN_LOW, 300
NET_RX_CN_MOD, 300
NET_RX_DROP, 300
NET_RX_SOFTIRQ, 131
NET_RX_SUCCESS, 300
net_set_todo(), 245
net_timestamp(), 602
net_todo_list, 245
net_todo_list_lock, 250
net_todo_run_mutex, 250
NET_TX, 208
NET_TX_SOFTIRQ, 131
NET_XMIT_BYPASS, 594
NET_XMIT_SUCCESS, 594
NetBIOS, 55
NETDEV_ALIGN_CONST, 263
netdev_boot_base(), 258
netdev_boot_setup (struct), 259
netdev_boot_setup_add(), 260
netdev_boot_setup_check(), 273
NETDEV_BOOT_SETUP_MAX, 259
netdev_chain, 239
NETDEV_CHANGE, 240
NETDEV_CHANGEADDR, 240
NETDEV_CHANGEMTU, 240
NETDEV_CHANGENAME, 240
NETDEV_DOWN, 240
netdev_ethtool_ops, 278
netdev_get_drvinfo(), 278
netdev_get_msglevel(), 278
NETDEV_HASHBITS, 238
netdev_max_backlog, 301
netdev_nit, 599
netdev_priv(), 264
NETDEV_REGISTER, 240
netdev_register_sysfs(), 251
netdev_run_todo(), 249
netdev_rx_stat, 599
netdev_set_msglevel(), 278
netdev_state_t, 207
NETDEV_TX_BUSY, 599
NETDEV_TX_LOCKED, 599
NETDEV_TX_OK, 599
NETDEV_UNREGISTER, 240
netdev_unregister_sysfs(), 619
NETDEV_UP, 240
netdev_wait_allrefs(), 619
netdevice.h
    l.103–139, 209
    l.142–151, 206
    l.156–160, 396
    l.162–174, 599
    l.164–168, 395
    l.179–190, 218
    l.192–211, 222
    l.213–224, 560
    l.226–240, 207
    l.236–240, 599
    l.243–251, 259
    l.254–491, 201
    l.493–494, 263
    l.496–501, 264
    l.503–506, 273
    l.509–516, 304
    l.54–60, 594
    l.563–579, 300
    l.581, 301
    l.585–604, 601
    l.6–26, 201
    l.606–609, 287
    l.61, 393
    l.611–619, 609
    l.62–68, 300

l.621–628, 606
l.630–633, 596
l.635–638, 309
l.690–693, 618
l.695, 614
l.696, 245
l.726–730, 283
l.74, 217
l.799–805, 309
l.807–824, 309
l.81–94, 223
l.826–832, 309
l.879–882, 313
Netfilter, 381
netfilter.h
l.121–157, 381
netif_device_present(), 283
NETIF_F_HW_CSUM, 220
NETIF_F_IP_CSUM, 220
NETIF_F_NO_CSUM, 220
netif_poll_enable(), 313
netif_queue_stopped(), 207, 596
netif_receive_skb(), 314
netif_running(), 207, 309
netif_rx(), 302
netif_rx_schedule(), 309
netif_rx_schedule_prep(), 309
netif_rx_stats, 599
netif_schedule(), 601
netif_schedule(dev), 599
netif_start_queue(), 208, 287
netif_stop_queue(), 207, 606
netif_wake_queue(), 208, 609
Netlink, 327
netlink, 234
netlink.c
l.54–59, 234
l.61–66, 235
netlink.h
l.800, 235
l.803–806, 235
netlink/, 118
NETLINK_ROUTE, 327
netlink_skb_parms (struct), 340
NETREG_REGISTERED, 214
NETREG_REGISTERING, 214
NETREG_RELEASED, 214
NETREG_UNINITIALIZED, 214
NETREG_UNREGISTERED, 214
NETREG_UNREGISTERING, 214
netrom/, 119
netstamp_needed, 415
netstat_attrs, 253
netstat_group, 253
network, 2
layer, 9
nf_debug_ip_local_deliver(), 400
NF_HOOK(), 381
NF_IP_LOCAL_IN, 397
NF_IP_POST_ROUTING, 586
nf_reset(), 400
NIC, 58, 61
niveau
administrateur, v
conception du noyau réseau, v
programmation réseau, v
utilisateur, v
NLM_F_ACK, 339
NLM_F_APPEND, 339
NLM_F_ATOMIC, 339
NLM_F_CREATE, 339
NLM_F_DUMP, 339
NLM_F_ECHO, 339
NLM_F_EXCL, 339
NLM_F_MATCH, 339
NLM_F_MULTI, 339
NLM_F_REPLACE, 339
NLM_F_REQUEST, 339
NLM_F_ROOT, 339
nlmsghdr (struct), 339
NLS, 58
no_llseek(), 479
non-acknowledge, 39
noop_dequeue(), 600
noop_enqueue(), 595
noop_qdisc, 248
noop_qdisc_ops, 248
noop_requeue(), 600
noqueue_qdisc, 285
noqueue_qdisc_ops, 285
norme
IEEE 802.3, 45
notifier-chain, 239
notifier.h
l.1–8, 239
l.14–19, 239
l.28–35, 241
l.45–58, 240
notifier_block (struct), 239
notifier_call_chain(), 240

NOTIFY_DONE, 241
NOTIFY_STOP_MASK, 241
Novell, 33
NPROTO, 443
ntohl(), 73
ntohs(), 73
numéro
    de fichier, 461
    de socket, 67
NWG, 57
nœud d'information, 460

offsetof(), 568
opération
    atomique, 132
open.c
    l.1018–1043, 506
    l.986–1014, 507
option
    IP, 103
ordre
    des octets
        grand boutien, 73
        petit boutien, 73
        réseau, 73
OSI, 8
OUI, 100
out-of-band, 78

périphérique
    activation, 279
    désactivation, 279
    en boucle, 230
    enregistrement, 279
    fermeture, 279
    ouverture, 279
    réseau
        en boucle, 200
    retrait, 279
packet, 36
packet switching, 5
packet/, 118
PACKET_BROADCAST, 155
PACKET_FASTROUTE, 155
PACKET_HOST, 155
PACKET_LOOPBACK, 155
PACKET_MULTICAST, 155
PACKET_OTHERHOST, 155
packet_type (struct), 304
PAN, 25
paquet, 14, 36
    Internet, 108
    IP, 108
partie
    basse, 131
    haute, 130
passerelle, 26, 103
PCI, 12
PDU, 12
peer, 62
peer-to-peer, 28
PF_APPLETALK, 67, 442
PF_ASH, 442
PF_ATMPVC, 67, 442
PF_ATMSVC, 442
PF_AX25, 67, 442
PF_BLUETOOTH, 442
PF_BRIDGE, 442
PF_DECnet, 442
PF_ECONET, 442
PF_INET, 67, 442
PF_INET6, 67, 442
PF_IPX, 67, 442
PF_IRDA, 442
PF_KEY, 442
PF_LLC, 442
PF_LOCAL, 67, 442
PF_MAX, 442
PF_NETBEUI, 442
PF_NETLINK, 67, 327, 442
PF_NETROM, 442
PF_PACKET, 67, 442
PF_PPPOX, 442
PF_ROSE, 442
PF_ROUTE, 442
PF_SECURITY, 442
PF_SNA, 442
PF_UNIX, 67, 442
PF_UNSPEC, 442
PF_WANPIPE, 442
PF_X25, 67, 442
pfifo_fast_dequeue(), 600
pfifo_fast_enqueue(), 595
pfifo_fast_ops, 284
pfifo_fast_requeue(), 601
physical layer, 9
pile
    réseau, 8
ping, 59
pkt_sched.h

l.16–17, 285
l.231–235, 596
l.4–24, 595
plaque
de cache, 126
point
de synchronisation, 10
point d'ancrage, 381
polling, 80
port, 30, 62
Posix.1g, 90
posix_types.h
l.1–39, 122
posix_types.h/linux, 121
PPP, 59
préfixe
réseau, 107
prepare_to_wait_exclusive(), 532
presentation layer, 10
prise
RJ45, 46
probe_list2(), 260
proc/, 457
process_backlog(), 312
processor.h
l.36–40, 168
processus, 129
PROT_SOCK, 513
proto (struct), 447
proto_ops (struct), 469
protocol
handler, 398
protocol.c
l.51, 398
l.54–74, 398
protocol.h
l.33, 398
l.36–41, 398
l.59–76, 445
l.77–78, 447
protocole, 2, 13
à détection de porteuse, 43
à diffusion
restreinte, 60
unique, 60
1822, 16
en-tête, 7
suffixe, 7
pskb_copy(), 188
pskb_expand_head(), 193
pskb_may_pull(), 190
pskb_pull(), 190
pskb_trim(), 195
pt_regs (struct), 293
PTR_ERR(), 233
ptrace.h
l.23–42, 293
ptype_all, 305
ptype_base[], 305
ptype_lock, 305

Qdisc (struct), 211
QDISC_ALIGN_CONST, 285
Qdisc_class_ops (struct), 213
qdisc_create_dflt(), 284
qdisc_lock_tree(), 247
Qdisc_ops (struct), 213
qdisc_reset(), 614
qdisc_restart(), 596
qdisc_run(), 596
qdisc_tree_lock, 247
qdisc_unlock_tree(), 248
qstr (struct), 473

réassemblage, 104
régulation des flux, 35
réseau
étendu, 25
à diffusion, 26
démon, 74
distribué, 6
en étoile, 30
en anneau, 29
en bus, 29
familial, 25
hiérarchisé, 6
local, 25
longue distance, 25
métropolitain, 25
machine à machine, 28
maillé, 6
personnel, 25
point-à-point, 27
réparti, 6
rate based flow control, 41
raw.c
l.30–31, 401
RAW4_HTABLE_SIZE, 401
raw_v4_htable[], 401
raw_v4_input(), 401

rcu, 212
rcu_dereference(), 386
rcu_head (struct), 212
rcu_read_lock(), 316
rcu_read_unlock(), 316
rcupdate.h
    l.157–193, 316
    l.225–239, 386
    l.45–53, 212
read(), 69
read_lock(), 137
read_unlock(), 137
read_write.c
    l.81–84, 479
readv(), 86
recv(), 79
recvfrom(), 81
recvmsg(), 88
redirection, 105
register_filesystem, 467
register_netdev(), 233
register_netdevice(), 241
release_sock(), 416
remote
    connexion, 60
    socket, 66
    telephone, 66
renifleur, 142
représentation
    canonique, 10
requête
    de redirection, 357
    FIB, 357
request_irq(), 129
resynchronisation, 38
RFC, 57
    768, 111
    791, 108
    1071, 113
    1122, 10, 381
    1141, 113
    1310, 58
    1518, 107
    1519, 107
    1624, 113
    1700, 62, 110, 216
    1889, 60
    2292, 90
    2474, 109
rhash_entries, 368
Roberts, Larry, 13, 15, 43
rose/, 119
routage, 5, 9, 102, 105
    dynamique, 324
    statique, 324
    strict, 111
    table de, 14
    tolérant, 111
route, 9, 27
route.c
    l.113, 371
    l.131–133, 372
    l.1369–1376, 394
    l.1413–1453, 580
    l.148–159, 366
    l.1546–1814, 388
    l.1816–1881, 384
    l.1887-2178, 573
    l.197–202, 368
    l.203, 370
    l.204, 371
    l.205, 370
    l.207, 371
    l.212–216, 372
    l.2180-2212, 572
    l.2215–2230, 572
    l.2696, 368
    l.2706–2810, 368
    l.452–456, 375
    l.585–625, 345
    l.769–774, 375
    l.776–900, 373
    l.939–958, 584
route.h
    l.107–108, 386
    l.131–135, 554
    l.137, 386
    l.50–76, 367
    l.6–17, 367
    l.86–104, 371
router, 105
routeur, 5, 105
routing, 105
    loose, 111
rt_cache_flush(), 345
rt_cache_stat, 371
rt_cache_stat (struct), 371
RT_CACHE_STAT_INC(), 386
rt_check_expire(), 372
rt_drop(), 375

rt_flush_lock, 345
rt_flush_timer, 372
rt_hash_bucket (struct), 368
rt_hash_code(), 372
rt_hash_log, 371
rt_hash_mask, 370
rt_hash_rnd, 370
rt_hash_table[], 368
rt_intern_hash(), 373
rt_periodic_timer, 372
rt_run_flush(), 372
RT_SCOPE_HOST, 338
RT_SCOPE_LINK, 338
RT_SCOPE_NOWHERE, 338
RT_SCOPE_SITE, 338
RT_SCOPE_UNIVERSE, 338
rt_secret_rebuild(), 372
rt_secret_timer, 372
rt_set_nexthop(), 580
RT_TABLE_DEFAULT, 334
RT_TABLE_LOCAL, 334
RT_TABLE_MAIN, 334
RT_TABLE_MAX, 334
RT_TABLE_MIN, 332
RT_TABLE_UNSPEC, 334
RT_TOS(), 552
RTA_ALIGNTO(), 354
rta_cacheinfo (struct), 336
RTA_DATA(), 354
RTA_LENGTH(), 354
RTA_NEXT(), 354
RTA_OK(), 354
RTA_PAYLOAD(), 354
rta_session (struct), 337
RTA_SPACE(), 354
rtable (struct), 367
rtattr (struct), 336
RTAX_ADVMSS, 332
RTAX_MAX, 332
RTAX_MTU, 332
RTAX_RTT, 332
RTAX_WINDOW, 332
rtmsg (struct), 337
rtmsg_fib(), 347
RTN_ANYCAST, 338
RTN_BLACKHOLE, 338
RTN_BROADCAST, 338
RTN_LOCAL, 338
RTN_MAX, 338
RTN_MULTICAST, 338
RTN_NAT, 338
RTN_PROHIBIT, 338
RTN_THROW, 338
RTN_UNICAST, 338
RTN_UNREACHABLE, 338
RTN_UNSPEC, 338
RTN_XRESOLVE, 338
rtnetlink.c
    l.81, 235
rtnetlink.h
    l.105–113, 340
    l.108–120, 354
    l.125–142, 337
    l.144–165, 338
    l.168–190, 337
    l.192-211, 338
    l.220–231, 334
    l.295–309, 336
    l.311–344, 332
    l.33–68, 339
    l.350–368, 337
    l.812–819, 243
    l.821–826, 186
    l.96–106, 336
rtnl, 235
rtnl_lock(), 234
rtnl_sem, 234
rtnl_shlock(), 235
rtnl_shunlock(), 235
rtnl_unlock(), 235
RTO_ONLINK, 573
RTP, 60
RTPROT_BOOT, 337
RTPROT_KERNEL, 337
RTPROT_REDIRECT, 337
RTPROT_STATIC, 337
RTPROT_UNSPEC, 337
RW_LOCK_UNLOCKED, 137
rwlock_init(), 137
RWLOCK_MAGIC, 136
rwlock_t, 136
RX_BUF_CLR, 269
RX_CMD, 269
RX_GOOD, 270
RX_HIGH, 269
RX_LOW, 269
RX_MISSED, 270
RX_MULT, 270
RX_NORM, 270
RX_PROM, 270

RX_RUNT, 270
RX_STATUS, 269
rxrpc/, 119

s16, 123
s32, 123
s64, 123
s8, 123
sémaphore, 138
sa_family_t, 72
SA_INTERRUPT, 129, 130
SA_SAMPLE
    RANDOM, 129
SA_SHIRQ, 129
SAP, 12
SAPROM, 269
saut, 103
sch_generic.c
    l.185–206, 609
    l.208–213, 249
    l.215–230, 285
    l.232–238, 614
    l.240–250, 595
    l.252–256, 600
    l.258–265, 600
    l.267–284, 248
    l.286–303, 285
    l.306–330, 595
    l.332–347, 600
    l.349–360, 601
    l.399–447, 284
    l.40–63, 247
    l.449–457, 614
    l.515–549, 283
    l.551–569, 614
    l.571–580, 247
    l.582–599, 617
    l.65–69, 248
    l.82–183, 596
sch_generic.h
    l.27–56, 211
    l.58–107, 213
sched.h
    l.180, 418
sched/, 118
schedule_timeout(), 613
SCM_CREDENTIALS, 90
SCM_RIGHTS, 90
SCP, 61
scripts/, 118
scrutation, 80
sctp/, 119
SDU, 12
security.c
    l.27, 491
security.h
    l.1018–1020, 491
    l.1203–1233, 491
    l.650–778, 491
    l.92–94, 491
security/, 118
security_operations (struct), 491
security_ops, 491
security_sk_free(), 416
security_socket_bind(), 510
security_socket_create(), 488
security_socket_post_create(), 489
security_socket_recvmsg(), 525
security_socket_sendmsg(), 544
security_socket_shutdown(), 501
segment, 58
semaphore, 138
semaphore.h
    l.44–48, 138
send(), 79
sendmsg(), 88
sendto(), 81
serveur, 70
session layer, 10
set_bit(), 132
SET_MODULE_OWNER(), 273
setsockopt(), 91, 408, 410
SFD, 102
SFTP, 61
sget(), 472
SHUT_RD, 84
SHUT_RDWR, 84
SHUT_WR, 84
simple_statfs(), 475
size_t, 125
sk_add_node(), 517
sk_alloc(), 411
sk_alloc_slab(), 455
sk_alloc_slab_error(), 455
sk_buff (struct), 149
sk_buff_head (struct), 157
sk_cachep, 411
sk_common_release(), 499
sk_dst_check(), 553
sk_dst_get(), 553

sk_dst_reset(), 505
sk_dst_set(), 553
sk_dst_set.h
    l.950–966, 553
sk_filter (struct), 410
sk_filter_len(), 415
sk_filter_release(), 414
sk_for_each(), 435
sk_free(), 414
sk_head(), 401
sk_init(), 412
SK_RMEM_MAX, 412
sk_set_owner(), 499
sk_unhashed(), 517
SK_WMEM_MAX, 412
skb_append(), 179
skb_bond(), 316
skb_checksum(), 426
skb_checksum_help(), 594
skb_clone(), 181
skb_clone_fraglist(), 194
skb_cloned(), 170
skb_copy(), 183
skb_copy_and_csum_datagram_iovec(), 535
skb_copy_bits(), 185
skb_copy_datagram_iovec(), 533
skb_copy_expand(), 189
skb_cow(), 196
SKB_DATA_ALIGN(), 162
skb_dequeue(), 175
skb_dequeue_tail(), 176
skb_drop_fraglist(), 163
skb_frag_struct (struct), 148
skb_frag_t, 148
skb_free_datagram(), 537
skb_get(), 170
skb_headlen(), 186
skb_headroom(), 167
skb_init(), 181
skb_insert(), 178
skb_is_nonlinear(), 166
SKB_LINEAR_ASSERT(), 168
skb_over_panic(), 168
skb_pagelen(), 628
skb_peek(), 180
skb_peek_tail(), 180
skb_pull(), 169
skb_push(), 168
skb_put(), 167
skb_queue_empty(), 173
skb_queue_head(), 173
skb_queue_head_init(), 172
skb_queue_len(), 173
skb_queue_purge(), 177
skb_queue_tail(), 174
skb_queue_walk(), 568
skb_realloc_headroom(), 194
skb_recv_datagram(), 529
skb_release_data(), 163
skb_reserve(), 167
skb_set_owner_w(), 588
skb_share_check(), 171
skb_shared(), 170
skb_shared_info (struct), 148
skb_shinfo(), 162
skb_tailroom(), 166
skb_trim(), 195
skb_under_panic(), 169
skb_unlink(), 179
skb_unshare(), 171
skbuff.c
    l.1–39, 160
    l.110–164, 161
    l.1181–1199, 175
    l.1201–1218, 176
    l.1220–1233, 177
    l.1235–1253, 173
    l.1255–1273, 174
    l.1274–1298, 179
    l.1301–1318, 179
    l.1321–1338, 178
    l.1428–1437, 181
    l.167–178, 163
    l.180–183, 194
    l.188–203, 163
    l.205–212, 162
    l.214–257, 164
    l.259–348, 181
    l.350–399, 187
    l.401–441, 183
    l.444–497, 188
    l.499–559, 193
    l.561–579, 194
    l.582–636, 189
    l.667–714, 195
    l.71, 161
    l.716–853, 191
    l.79–92, 168
    l.855–930, 185
    l.94–108, 169

l.998–1070, 426
skbuff.h
l.1–15, 149
l.1055–1071, 186
l.1073–1077, 568
l.115–122, 157
l.1151–1158, 400
l.126–127, 149
l.129–135, 148
l.137–147, 148
l.149–283, 149
l.307, 166
l.313–314, 162
l.316–325, 173
l.327–338, 170
l.340–356, 164
l.36–38, 155
l.365–376, 170
l.378–388, 170
l.390–412, 171
l.40–41, 162
l.414–443, 171
l.445–464, 180
l.466–485, 180
l.48–71, 155
l.487–496, 173
l.498–503, 172
l.505–535, 174
l.537–560, 175
l.563–589, 176
l.592–605, 178
l.607–614, 179
l.616–632, 177
l.637–652, 177
l.655–658, 166
l.660–663, 186
l.665–672, 628
l.687, 168
l.701–719, 167
l.728–744, 168
l.73–84, 220
l.746–766, 169
l.768–791, 190
l.793–802, 167
l.804–813, 166
l.815–827, 167
l.853–892, 195
l.910–924, 178
l.926–962, 197
l.964–986, 196
skbuff_head_cache, 161
slab, 126
slab.c
l.2406–2426, 125
SLAB_CACHE_DMA, 127
SLAB_HWCACHE_ALIGN, 127
SLAB_NO_REAP, 127
sliding window, 40
SLIP, 59
slot, 43
SMP, 128
SMTP, 61
SNA, 13
snmp.h
l.1–10, 380
l.122–123, 380
l.71–75, 380
l.97–101, 430
snmp.h/linux
l.10–44, 380
SO_BROADCAST, 91
SO_DONTROUTE, 410
SO_ERROR, 91
SO_KEEPALIVE, 91
SO_LINGER, 91, 410
SO_OOBINLINE, 91
SO_PEERCRED, 410
SO_PRIORITY, 410
SO_RCVBUF, 91
SO_RCVLOWAT, 410
SO_RCVTIMEO, 410
SO_REUSEADDR, 91
SO_SNDBUF, 91
SO_SNDTIMEO, 410
SO_TYPE, 91
sock (struct), 404
sock.c
l.1153–1203, 416
l.1205–1213, 418
l.1217–1226, 416
l.1268–1280, 525
l.1297–1334, 499
l.130–144, 412
l.1338–1345, 455
l.178–184, 415
l.606–642, 411
l.644–667, 414
l.669–685, 412
l.737–750, 560
l.824–908, 561

l.84–88, 413
l.910–914, 561
l.916–930, 418
l.932–950, 416
sock.h
l.1–80, 357
l.10238–1244, 533
l.1050–1056, 588
l.1070–1108, 438
l.1125–1133, 530
l.115-267, 404
l.1223–1226, 531
l.1228–1231, 419
l.1246–1259, 528
l.269–280, 401
l.288–291, 517
l.318–327, 413
l.349–358, 517
l.371–372, 435
l.401–409, 415
l.496–567, 447
l.529–541, 418
l.572–575, 455
l.577–592, 499
l.594–598, 517
l.608–609, 513
l.615–618, 505
l.620–632, 525
l.634–637, 525
l.644-647, 476
l.649–652, 480
l.654–657, 495
l.73–82, 408
l.847–863, 414
l.871–894, 413
l.896–901, 414
l.903–917, 500
l.92-113, 406
l.937–948, 553
l.968–984, 505
l.999–1010, 553
sock_alloc(), 490
sock_alloc_inode(), 476
sock_alloc_send_pskb(), 561
sock_alloc_send_skb(), 561
SOCK_ASYNC_NOSPACE, 469
SOCK_ASYNC_WAITDATA, 469
SOCK_BINDADDR_LOCK, 505
SOCK_BINDPORT_LOCK, 505
sock_close(), 482
sock_common (struct), 406
sock_common_recvmsg(), 525
sock_create(), 487
sock_destroy_inode(), 477
SOCK_DGRAM, 68, 444
sock_disable_timestamp(), 415
sock_error(), 530
sock_fasync(), 481
sock_filter (struct), 410
sock_flag(), 415
sock_fs_type, 470
sock_hold(), 413
sock_init(), 449, 470
sock_init_data(), 416
SOCK_INODE(), 495
sock_inode_cachep, 476
sock_intr_errno(), 533
sock_iocb (struct), 525
sock_lock_init(), 413
sock_map_fd(), 494
SOCK_MAX, 444
sock_mnt, 490
sock_no_mmap(), 480
sock_no_open(), 481
SOCK_NOSPACE, 469
sock_orphan(), 500
SOCK_PACKET, 68, 444
sock_poll(), 479
sock_prot_inc_use(), 517
sock_put(), 414
sock_queue_rcv_skb(), 438
SOCK_RAW, 68, 444
sock_rcvtimeo(), 531
SOCK_RDM, 68, 444
sock_recv_timestamp(), 528
sock_recvmsg(), 522
sock_register(), 450
sock_release(), 483
sock_reset_flag(), 415
sock_sendmsg(), 543
SOCK_SEQPACKET, 68, 444
sock_sndtimeo(), 419
SOCK_STREAM, 68, 444
sock_unregister(), 451
sock_vfree(), 408
sock_wamalloc(), 560
sockaddr (struct), 72
sockaddr_ax25 (struct), 73
sockaddr_in (struct), 72
sockaddr_in6 (struct), 73

sockaddr_ipx (struct), 73
sockaddr_un (struct), 72
sockect.c
    l.274, 476
socket, 66
    buffer, 147
    client, 62, 71
    distante, 66
    locale, 66
    numéro, 67
    remote, 66
    serveur, 62, 70
    structure, 468
socket (struct), 468
socket(), 71
socket.c
    l.1–59, 119
    l.1055–1059, 480
    l.1074–1180, 487
    l.1187–1207, 486
    l.1275–1302, 510
    l.139–143, 444
    l.145–164, 451
    l.1499–1542, 542
    l.1553–1593, 521
    l.166–169, 451
    l.1663–1684, 500
    l.171–187, 489
    l.190–194, 490
    l.1984–2008, 450
    l.2010–2027, 451
    l.2032–2070, 449
    l.210–230, 510
    l.232–270, 523
    l.272, 470
    l.321–325, 475
    l.327–331, 471
    l.333, 490
    l.335–339, 470
    l.340–343, 477
    l.344–346, 477
    l.348–412, 494
    l.414–452, 501
    l.454–481, 490
    l.483–492, 481
    l.499–528, 483
    l.530–546, 543
    l.548–560, 543
    l.580–597, 524
    l.599–612, 522
    l.61–137, 477
    l.932–942, 479
    l.951–966, 482
    l.968–1045, 481
socket.h
    l.148–152, 339
    l.154–216, 442
    l.47–61, 520
    l.63–73, 520
socket_alloc (struct), 476
socket_file_ops, 477
SOCKET_I(), 480
socket_lock_t, 408
socket_state, 469
socketpair(), 67
sockets_in_use, 490
sockfd_lookup(), 501
sockfs, 470
sockfs_delete_dentry(), 477
sockfs_dentry_operations (struct), 477
sockfs_get_sb(), 471
SOCKFS_MAGIC, 470
sockfs_ops, 475
sofnet_data (struct), 208
softnet_data (struct), 300
SOL_SOCKET, 89, 91
SOL_TCP, 91
somme de contrôle, 36
    IP, 103
sound/, 118
sous-paquet, 109
sous-réseau, 25
sous-système
    réseau, v
Space.c, 229
    l.106–109, 261
    l.111–124, 260
    l.165–259, 261
    l.315–333, 258
    l.389–421, 230
    l.423, 230
spin_is_locked(), 136
spin_lock(), 135
spin_lock_bh(), 136
spin_lock_irq(), 136
spin_lock_irqsave(), 135
SPIN_LOCK_UNLOCKED, 135
spin_trylock(), 136
spin_unlock(), 136
spin_unlock_bh(), 136

spin_unlock_irq(), 136
spin_unlock_irqrestore(), 136
spin_unlock_wait(), 136
spinlock.h
    l.154–171, 136
    l.173–181, 137
    l.89–97, 135
    l.98–108, 135
spinlock_init(), 135
spinlock_t, 135
SS_CONNECTED, 469
SS_CONNECTING, 469
SS_DISCONNECTING, 469
SS_FREE, 469
SS_UNCONNECTED, 469
SSH, 60
ssize_t, 125
standard
    DIX, 45
start bit, 23
statfs (struct), 463
statfs.h, 475
statfs.h/asm-generic
    l.1–21, 463
stddef.h
    l.13–18, 568
stop bit, 23
store-and-forward, 5
Strowger, Almon, 4
SUD, 12
suffixe
    de protocole, 7
suite, 7
    de protocoles, 2
    TCP/IP, 8
sunrpc/, 119
super-bloc, 461
super.c
    l.213–260, 474
    l.264–314, 472
    l.622–630, 474
super_block (struct), 461
super_operations (struct), 462
support physique, 2
suseseconds_t, 127
synchronize_net(), 453
sys-fs.c
    l.412–456, 251
sys.c
    l.149–180, 240
sys_bind(), 510
sys_close(), 506
sys_recvfrom(), 521
sys_sendto(), 542
sys_shutdown(), 500
sys_socket(), 486
sysctl_ip_nonlocal_bind, 511
sysctl_ipfrag_high_thresh, 634
sysctl_ipfrag_low_thresh, 634
sysctl_local_port_range[], 514
sysctl_net_ipv4.c
    l.57, 498
sysctl_rmem_default, 412
sysctl_rmem_max, 412
sysctl_wmem_default, 412
sysctl_wmem_max, 412
sysfs_create_group(), 253
système
    informatique, 2
    ouvert, 3
    propriétaire, 3
système de fichiers
    virtuel, 460

tailroom, 148
tampon
    de socket, 147
TASK_INTERRUPTIBLE, 613
TASK_UNINTERRUPTIBLE, 613
tasklet, 131
tasklet_schedule(), 131
TASKLET_SOFTIRQ, 131
tasklet_struct (struct), 131
taux de transmission, 23
Taylor, Robert, 15
TC_PRIO_MAX, 595
TCP, 60
    numéro de protocole IP, 110
tcp.h
    l.59–73, 407
TCP/IP, 8
    architecture, 55
    implémentation, 55
    modèle, 55
TCP_CLOSE, 407
TCP_CLOSE_WAIT, 407
TCP_CLOSING, 407
TCP_ESTABLISHED, 407
TCP_FIN_WAIT1, 407
TCP_FIN_WAIT2, 407

tcp_ipv4.c
l.99–104, 514
TCP_LAST_ACK, 407
TCP_LISTEN, 407
TCP_MAX_STATES, 407
tcp_protocol, 449
TCP_SYN_RECV, 407
TCP_SYN_SENT, 407
TCP_TIME_WAIT, 407
TCQ_F_BUILTIN, 211
TCQ_F_INGRES, 211
TCQ_F_THROTTLED, 211
TDM, 42
technique
de remplissage, 39
technologie
de transmission, 26
telnet, 60
terminaison, 29
test_and_change_bit(), 132
test_and_clear_bit(), 132
test_and_set_bit(), 132
test_bit(), 132
Thin Ethernet, 28
time out, 40
time-sharing, 13
time.h
l.18–21, 127
time_t, 127
timer.c
l.1087–113, 613
timer.h
l.11–22, 128
l.35–47, 128
timer_list (struct), 128
timeval (struct), 127
TLI, 65
token bus, 48
Token Ring, 59
token ring, 48
top half, 130
topologie
d'un réseau, 27
TOS, 108
tr_configure(), 261
tr_type_trans(), 298
trailer, 36
trame, 36, 58
Ethernet
802.3, 102
DIX, 100
transceiver, 29, 46
transcepteur, 29, 46
transfert
de fichier, 61
transmetteur/récepteur, 29
transmission
différée, 5
filaire, 22
sans fil, 22
transport layer, 9
try_module_get(), 489
TTL, 103
TX_16COLLISIONS, 270
TX_CMD, 269
TX_COLLISION, 270
TX_NORM, 270
TX_RDY, 270
TX_READY, 270
TX_STATUS, 269
type
de paquet, 304
type de servive, 108
types.h
l.1–5, 124
l.1–50, 123
l.13–16, 124
l.29, 127
l.54–70, 125
l.77–80, 127
l.92–124, 124

u16, 123
u32, 123
u64, 123
u8, 123
u_char, 123
u_int, 123
u_int32_t, 123
u_int64_t, 123
u_int8_t, 123
u_int_16_t, 123
u_long, 123
u_short, 123
UCB, 56
ucred (struct), 90, 339
ud_mib (struct), 430
UDP, 60
numéro de protocole IP, 110
udp.c

l.1085–1108, 431
l.1110–1207, 428
l.117, 435
l.118, 433
l.120–202, 515
l.1353–1371, 503
l.209–217, 505
l.219–264, 434
l.266–276, 433
l.382–394, 569
l.396–472, 566
l.475–478, 432
l.480–668, 548
l.757–760, 438
l.768–862, 526
l.865–885, 504
l.978–1041, 436
udp.h
l.32, 435
l.43–52, 437, 516
l.50–55, 551
l.54–61, 568
l.54–67, 437
l.6–27, 421
l.77–80, 430
udp_check(), 432
udp_checksum_complete(), 431
udp_checksum_init(), 431
UDP_CSUM_DEFAULT, 568
UDP_CSUM_NORCV, 568
UDP_CSUM_NOXMIT, 568
udp_disconnect(), 504
udp_flush_pending_frames(), 569
udp_hash[], 435
udp_hash_lock, 433
UDP_HTABLE_SIZE, 435
UDP_INC_STATS(), 430
UDP_INC_STATS_BH(), 430
UDP_INC_STATS_USER(), 430
udp_lport_inuse(), 516
udp_opt (struct), 437
udp_port_rover, 515, 516
udp_prot, 503
udp_protocol, 398, 449
udp_push_pending_frames(), 566
udp_queue_rcv_skb(), 436
udp_rcv(), 428
udp_recvmsg(), 526
udp_sendmsg(), 548
udp_sk(), 437
udp_sock (struct), 437
udp_statistics, 430
udp_v4_get_port(), 515
udp_v4_lookup(), 433
udp_v4_lookup_longway(), 434
udp_v4_mcast_deliver(), 430
udp_v4_unhash(), 505
udphdr (struct), 421
uint, 123
uint16_t, 123
uint32_t, 123
uint64_t, 123
uint8_t, 123
uio.h
l.17–24, 520
l.7–9, 520
ulong, 123
un.h, 72
unchar, 123
unicast, 27, 60
unistd.h, 69
unité
de fragmentation, 110
unix/, 119
unlikely(), 166
unregister_netdev(), 615
unregister_netdevice(), 615
up(), 138
ushort, 123
usr/, 118
utilitaire
de Berkeley, 56
UUCP, 55

van Kempen, Fred, 145
verrou rotatif, 134
version, 277
VFS, 460
vlan/, 119
VTAM, 13

wait.c
l.78–94, 532
wait.h
l.325–332, 532
l.51–55, 408
wait_for_packet(), 531
wait_on_sync_kiocb(), 523
wait_queue_head_t, 408
WAN, 25

wanrouter/, 119
WARN_ON(), 166
web, 61
weight_p, 302
Wi-Fi, 25, 48
wild address, 73
write(), 69
write_lock(), 137
writev(), 86

X.25, 59
x25/, 119
xfrm.h
    l.262-284, 409
xfrm/, 119
xfrm_policy (struct), 409
XNS, 55

ZERONET(), 393
zone
    critique, 132