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

Code Linux 2.6.10

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` :

```
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

La fonction `inet_register_protosw()` est définie dans le fichier `linux/net/ipv4/af_inet.c` :

Code Linux 2.6.10

```
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.