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) :

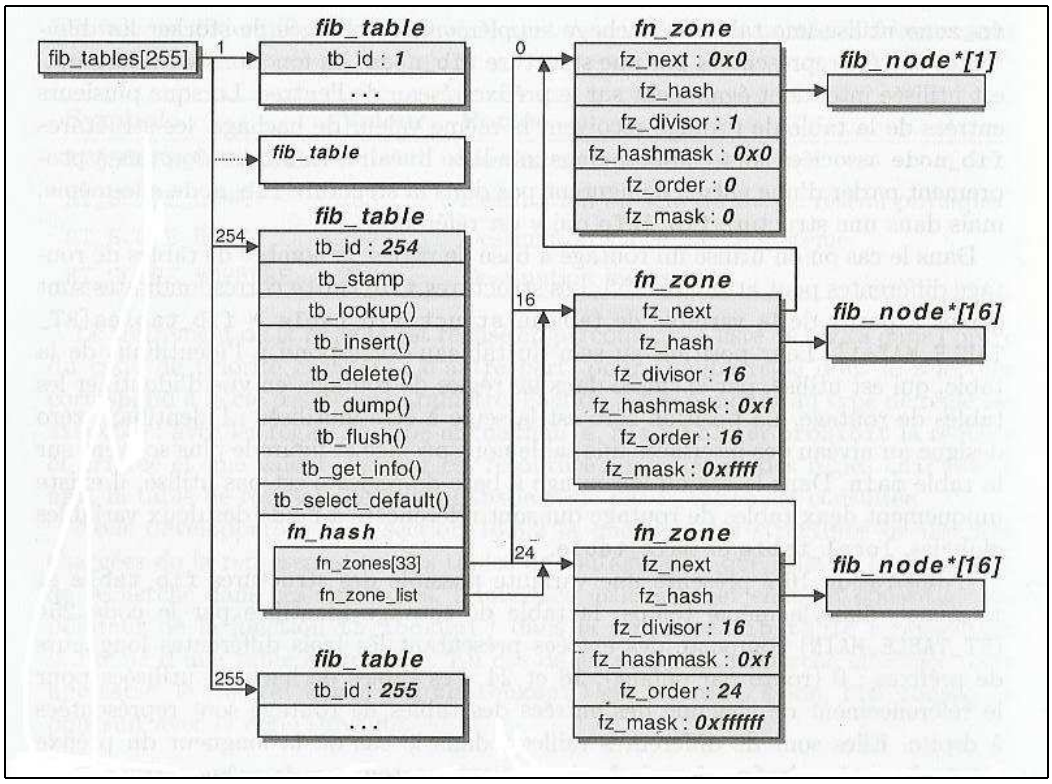

FIG. 21.1 – *Table de routage*

- 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.

Code Linux 2.6.10

Les macros `RTA_PAYLOAD()`, `RTA_DATA()` et `RTA_OK()` sont définies dans le fichier `linux/include/linux/rtnetlink.h`:

```
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.