Chapitre 23

Réception des paquets ordinaires sous IPv4

Les trames entrant dans l'ordinateur *via* la carte réseau et destinées à l'ordinateur stockent, comme décrit au chapitre 19, le paquet correspondant dans la file d'attente d'entrée du processeur actif. Une fois le protocole réseau déterminé dans la couche liaison (`ETH_P_IP` dans le cas qui nous intéresse), le paquet est transmis à la fonction `ip_rcv()`. Nous allons étudier l'itinéraire de ce paquet dans ce chapitre.

23.1 Première étape : tri et contrôle de l'intégrité du paquet

Nous avons vu au chapitre 19 que, lorsqu'une trame est reçue, elle est traitée au niveau de la couche de liaison puis passée à la couche supérieure, c'est-à-dire à la couche réseau. Ceci signifie qu'un tampon de socket est créé et qu'une fonction est appelée, `ip_rcv()` dans le cas de IPv4 comme l'a montré `ip_packet_type`, avec en particulier le descripteur de ce tampon de socket comme paramètre.

23.1.1 Tri et contrôle

La fonction `ip_rcv(skb, dev, pkt_type)` rejette, en premier lieu, les paquets qui ne sont pas destinés à cet ordinateur. Elle vérifie ensuite les critères d'exactitude fondamentaux du paquet : comporte-t-il au moins la taille d'un en-tête IP ? s'agit-il de la version 4 du protocole Internet ? la somme de contrôle est-elle correcte ? la longueur du paquet est-elle erronée ? Elle passe la main, enfin, à l'étape suivante.

La fonction renvoie un entier, correspondant à l'une des constantes symboliques suivantes, déjà vues au chapitre 19 : `NET_RX_SUCCESS`, `NET_RX_DROP`, `NET_RX_CN_LOW`, `NET_RX_CN_MOD`, `NET_RX_CN_HIGH` ou `NET_RX_BAD`.

Code Linux 2.6.10

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

```
357 /*
358  *      Routine principale de reception IP.
359  */
360 int ip_rcv(struct sk_buff *skb, struct net_device *dev, struct packet_type *pt)
361 {
362         struct iphdr *iph;
363
364         /* Lorsque l'interface est en mode promiscuite, ecarter tous les dechets
365          * qu'elle recoit, ne pas essayer de les analyser.
366          */
367         if (skb->pkt_type == PACKET_OTHERHOST)
368                 goto drop;
369
370         IP_INC_STATS_BH(IPSTATS_MIB_INRECEIVES);
371
372         if ((skb = skb_share_check(skb, GFP_ATOMIC)) == NULL) {
373                 IP_INC_STATS_BH(IPSTATS_MIB_INDISCARDS);
374                 goto out;
375         }
376
377         if (!pskb_may_pull(skb, sizeof(struct iphdr)))
378                 goto inhdr_error;
379
380         iph = skb->nh.iph;
381
382         /*
383          *      RFC1122: 3.1.2.2 DOIT ecarter sans faire de bruit toutes les trames IP dont la
                        somme de controle n'est pas bonne.
384          *
385          *      Le datagramme est-il acceptable ?
386          *
387          *      1.      Longueur d'au moins la taille d'un en-tete ip
388          *      2.      Version 4
389          *      3.      Somme de controle correcte. [Accelerer l'optimisation pour plus tard,
                            passer le controle pour le loopback]
390          *      4.      N'a pas une longueur bidon
```

```
391          */
392
393         if (iph->ihl < 5 || iph->version != 4)
394                 goto inhdr_error;
395
396         if (!pskb_may_pull(skb, iph->ihl*4))
397                 goto inhdr_error;
398
399         iph = skb->nh.iph;
400
401         if (ip_fast_csum((u8 *)iph, iph->ihl) != 0)
402                 goto inhdr_error;
403
404         {
405                 __u32 len = ntohs(iph->tot_len);
406                 if (skb->len < len || len < (iph->ihl<<2))
407                         goto inhdr_error;
408
409                 /* On peut avoir rembourre le tampon de notre support de transport. Maintenant
410                  * que nous savons qu'il s'agit d'IP, nous pouvons tailler a la vraie
                    longueur de la trame.
411                  * Noter que ceci signifie maintenant que skb->len contient
412                    ntohs(iph->tot_len).*/
413                 if (skb->len > len) {
414                         __pskb_trim(skb, len);
415                         if (skb->ip_summed == CHECKSUM_HW)
416                                 skb->ip_summed = CHECKSUM_NONE;
417                 }
418         }
419
420         return NF_HOOK(PF_INET, NF_IP_PRE_ROUTING, skb, dev, NULL,
421                        ip_rcv_finish);
422
423 inhdr_error:
424         IP_INC_STATS_BH(IPSTATS_MIB_INHDRERRORS);
425 drop:
426         kfree_skb(skb);
427 out:
428         return NET_RX_DROP;
429 }
```

Rappelons que dans le cas de IPv4, le paramètre `pt` a la valeur `ip_packet_type`. Décrivons maintenant le corps de la fonction :

- On déclare une adresse d'en-tête IP.
- S'il s'agit d'un paquet IP non destiné à cet ordinateur (le mode promiscuité l'a laissé passé mais il est destiné à un renifleur ; il ne doit pas être traité comme un paquet ordinaire), on libère le descripteur de tampon passé en argument et on renvoie l'état `NET_RX_DROP`. Il sera traité par une autre fonction de réception (liée au reniflage).
- On incrémente le nombre de paquets reçus dans les informations statistiques relatives à la couche IP en réception.

  La macro `IP_INC_STATS_BH()` est définie dans le fichier `linux/include/net/ip.h`: Code Linux 2.6.10

```
155 #define IP_INC_STATS_BH(field)          SNMP_INC_STATS_BH(ip_statistics, field)
```

  La variable `ip_statistics` est déclarée dans le fichier `linux/net/ipv4/ip_input.c`: Code Linux 2.6.10

```
149 /*
150  *      Gestion des statistiques SNMP
151  */
152
153 DEFINE_SNMP_STAT(struct ipstats_mib, ip_statistics);
```

Code Linux 2.6.10

La structure `ipstats_mib` et la macro `DEFINE_SNMP_STAT()` sont définies dans le fichier `linux/include/net/snmp.h`:

```
1  /*
2   *
3   *              Entrees MIB SNMP pour le sous-systeme IP.
4   *
5   *              Alan Cox <gw4pts@gw4pts.ampr.org>
6   *
7   *              Nous ne choisissons pas d'implementer SNMP dans le noyau (Ceci serait
8   *              idiot puisque SNMP est dur a l'heure actuelle). Nous avons
9   *              cependant besoin de collecter les statistiques MIB et de les exporter
10  *              hors de /proc (a un certain moment)
[...]
71 /* IPstats */
72 #define IPSTATS_MIB_MAX __IPSTATS_MIB_MAX
73 struct ipstats_mib {
74         unsigned long   mibs[IPSTATS_MIB_MAX];
75 } __SNMP_MIB_ALIGN__;
[...]
122 #define DEFINE_SNMP_STAT(type, name)    \
123          __typeof__(type) *name[2]
```

Code Linux 2.6.10

Les constantes `__IPSTATS_MIB_MAX`, `IPSTATS_MIB_INRECEIVES`, `IPSTATS_MIB_INDISCARDS` et autres sont définies dans le fichier `linux/include/linux/snmp.h`:

```
10 /* Definitions mib ipstats */
11 /*
12  * RFC 1213 :  MIB-II
13  * RFC 2011 (met a jour 1213) :  SNMPv2-MIB-IP
14  * RFC 2863 :  Interfaces Group MIB
15  * RFC 2465 :  IPv6 MIB : General Group
16  * draft-ietf-ipv6-rfc2011-update-10.txt: MIB for IP: IP Statistics Tables
17  */
18 enum
19 {
20         IPSTATS_MIB_NUM = 0,
21         IPSTATS_MIB_INRECEIVES,                 /* InReceives */
22         IPSTATS_MIB_INHDRERRORS,                /* InHdrErrors */
23         IPSTATS_MIB_INTOOBIGERRORS,             /* InTooBigErrors */
24         IPSTATS_MIB_INNOROUTES,                 /* InNoRoutes */
25         IPSTATS_MIB_INADDRERRORS,               /* InAddrErrors */
26         IPSTATS_MIB_INUNKNOWNPROTOS,            /* InUnknownProtos */
27         IPSTATS_MIB_INTRUNCATEDPKTS,            /* InTruncatedPkts */
28         IPSTATS_MIB_INDISCARDS,                 /* InDiscards */
29         IPSTATS_MIB_INDELIVERS,                 /* InDelivers */
30         IPSTATS_MIB_OUTFORWDATAGRAMS,           /* OutForwDatagrams */
31         IPSTATS_MIB_OUTREQUESTS,                /* OutRequests */
32         IPSTATS_MIB_OUTDISCARDS,                /* OutDiscards */
33         IPSTATS_MIB_OUTNOROUTES,                /* OutNoRoutes */
34         IPSTATS_MIB_REASMTIMEOUT,               /* ReasmTimeout */
35         IPSTATS_MIB_REASMREQDS,                 /* ReasmReqds */
36         IPSTATS_MIB_REASMOKS,                   /* ReasmOKs */
37         IPSTATS_MIB_REASMFAILS,                 /* ReasmFails */
38         IPSTATS_MIB_FRAGOKS,                    /* FragOKs */
39         IPSTATS_MIB_FRAGFAILS,                  /* FragFails */
40         IPSTATS_MIB_FRAGCREATES,                /* FragCreates */
41         IPSTATS_MIB_INMCASTPKTS,                /* InMcastPkts */
42         IPSTATS_MIB_OUTMCASTPKTS,               /* OutMcastPkts */
43         __IPSTATS_MIB_MAX
44 };
```

- Si le tampon est partagé, on essaie de le cloner et on travaillera désormais sur le clone. Si on ne parvient pas à le cloner, on incrémente les informations statistiques concernant les paquets détruits lors de la réception et on renvoie `NET_RX_DROP`.
- Si le tampon est de taille inférieure à celle d'un en-tête IP, on incrémente l'information statistique concernant le nombre d'erreurs sur l'en-tête des paquets IP entrants, on libère le descripteur de tampon et on renvoie `NET_RX_DROP`. Sinon on déplace le début de la zone des données du tampon d'une taille convenant à un en-tête IP. La zone des données du tampon de socket comportera donc maintenant un datagramme et non plus un paquet.
- On instantie l'adresse d'en-tête IP déclarée avec celle du champ adéquat du descripteur de tampon passé en paramètre, ce qui nous permet de nous positionner au début de l'en-tête IP du paquet contenu dans le tampon.
- En conformité avec [RFC 1122], si la longueur de cet en-tête IP est inférieure à 5 mots (soit 20 octets) ou si la version n'est pas IPv4, on incrémente l'information statistique concernant le nombre d'erreurs sur l'en-tête des paquets entrants, on libère le descripteur de tampon et on renvoie `NET_RX_DROP`.
- On vérifie que la longueur du tampon est supérieure à la longueur indiquée dans l'en-tête IP. Si ce n'est pas le cas, on incrémente l'information statistique concernant le nombre d'erreurs sur l'en-tête des paquets entrants, on libère le descripteur de tampon et on renvoie `NET_RX_DROP`.
- On déplace à nouveau le début de la zone des données du tampon pour tenir compte de la taille réelle de l'en-tête IP (options comprises) à partir de ce qui est indiqué dans l'en-tête IP.
- On vérifie la somme de contrôle de l'en-tête. Si elle ne cononcorde avec ce qui est indiqué dans l'en-tête, on incrémente l'information statistique concernant le nombre d'erreurs sur l'en-tête des paquets entrants, on libère le descripteur de tampon et on renvoie `NET_RX_DROP`.
- On déclare une longueur que l'on instantie avec la longueur totale du paquet. Si la taille du tampon est inférieure à la longueur du paquet, on incrémente l'information statistique concernant le nombre d'erreurs sur l'en-tête des paquets entrants, on libère le descripteur de tampon et on renvoie `NET_RX_DROP`.
- Si la taille du tampon est supérieure à la longueur du paquet, c'est que le paquet contenait un champ de remplissage. On déplace donc la fin de la zones des données du tampon pour retirer celle-ci du datagramme et on change le mode de vérification de la somme de contrôle.
- Le point d'ancrage `NF_IP_PRE_ROUTING` est appelé, pour traitement éventuel, avec `ip_rcv_finish()` comme fonction pour continuer après ce traitement éventuel.

  Nous allons parler des points d'ancrage dans la sous-section suivante.

23.1.2 Étape optionnelle : les points d'ancrage

L'option de compilation `NETFILTER` du noyau permet d'étendre le déroulement de certains protocoles, si cela est souhaité. Les **points d'ancrage** (*hook* en anglais) Netfilter sont placés à des points stratégiques de l'implémentation de certains protocoles et sont déployés par exemple pour les fonctions pare-feu, qualité de service ou transformation d'adresse. Un point d'ancrage Netfilter est appelé par la macro `NF_HOOK()` à laquelle sera transmise la fonction à venir après traitement de l'extension Netfilter sous la forme d'un pointeur de fonction.

Si Netfilter n'a pas été configuré, la macro veille à ce que cette fonction à venir soit immédiatement démarrée, comme le montre les lignes 147–148 de la définition de la macro `NF_HOOK()`, définie dans le fichier `linux/include/linux/netfilter.h`:

Code Linux 2.6.10

```
121 /* Activation d'un point d'ancrage ; on appelle okfn ou kfree_skb, a moins qu'un point
```

```
122    d'ancrage ne renvoie NF_STOLEN (dans un tel cas, c'est au point d'ancrage de reagir avec
123    les consequences possibles).
124
125    Renvoie -ERRNO si le paquet est ecarte. Zero signifie mise en file d'attente, vole ou
126    accepte.
127 */
128
129 /* RR :
130    > Je ne veux pas que nf_hook renvoie quelque chose parce que les gens pourraient l'oublier
131    > avec async et compter sur le fait qu'une valeur de retour signifie "le paquet etait ok".
132
133    AK :
134    Il suffit de le documenter clairement, vous pouvez alors esperer un peu de bon sens
135    de la part des codeurs du noyau :)
136 */
137
138 /* Ceci est fruste, mais les fonctions en ligne ne l'arreteront pas pour eviter l'appel de
139    fonction en routage rapide : gcc ne traite pas les fonctions en ligne (necessite des
       valeurs de pistage ?). --RR */
140 #ifdef CONFIG_NETFILTER_DEBUG
141 #define NF_HOOK(pf, hook, skb, indev, outdev, okfn)                    \
142  nf_hook_slow((pf), (hook), (skb), (indev), (outdev), (okfn), INT_MIN)
143 #define NF_HOOK_THRESH nf_hook_slow
144 #else
145 #define NF_HOOK(pf, hook, skb, indev, outdev, okfn)                    \
146 (list_empty(&nf_hooks[(pf)][(hook)])                                   \
147  ? (okfn)(skb)                                                         \
148  : nf_hook_slow((pf), (hook), (skb), (indev), (outdev), (okfn), INT_MIN))
149 #define NF_HOOK_THRESH(pf, hook, skb, indev, outdev, okfn, thresh)     \
150 (list_empty(&nf_hooks[(pf)][(hook)])                                   \
151  ? (okfn)(skb)                                                         \
152  : nf_hook_slow((pf), (hook), (skb), (indev), (outdev), (okfn), (thresh)))
153 #endif
154
155 int nf_hook_slow(int pf, unsigned int hook, struct sk_buff *skb,
156                  struct net_device *indev, struct net_device *outdev,
157                  int (*okfn)(struct sk_buff *), int thresh);
```

C'est tout ce qui nous intéressera sur les points d'ancrage pour l'instant.

23.2 Deuxième étape : routage et traitement des options

23.2.1 Vue d'ensemble

Nous avons vu ci-dessus que la fonction à venir dans le cas de notre point d'ancrage est `ip_rcv_finish()`. Celle-ci commence par déterminer la route du paquet, c'est-à-dire s'il doit être délivré localement, s'il s'agit d'une diffusion individuelle ou d'une multidiffusion, ou s'il doit être acheminé vers un autre ordinateur. Puis elle vérifie si des options sont indiquées dans l'en-tête IP et les traite le cas échéant.

Code Linux 2.6.10

Cette fonction est définie dans le fichier `linux/net/ipv4/ip_input.c` :

```
285 static inline int ip_rcv_finish(struct sk_buff *skb)
286 {
287         struct net_device *dev = skb->dev;
288         struct iphdr *iph = skb->nh.iph;
289
290         /*
291          *      Initialise le cache de circuit virtuel pour le paquet. Decrit
292          *      comment le paquet voyage dans le reseau Linux.
293          */
```

```
294         if (skb->dst == NULL) {
295                 if (ip_route_input(skb, iph->daddr, iph->saddr, iph->tos, dev))
296                         goto drop;
297         }
298
299 #ifdef CONFIG_NET_CLS_ROUTE
300         if (skb->dst->tclassid) {
301                 struct ip_rt_acct *st = ip_rt_acct + 256*smp_processor_id();
302                 u32 idx = skb->dst->tclassid;
303                 st[idx&0xFF].o_packets++;
304                 st[idx&0xFF].o_bytes+=skb->len;
305                 st[(idx>>16)&0xFF].i_packets++;
306                 st[(idx>>16)&0xFF].i_bytes+=skb->len;
307         }
308 #endif
309
310         if (iph->ihl > 5) {
311                 struct ip_options *opt;
312
313                 /* Il semble surestime, puisque toutes les options IP
314                    n'exigent pas une mutilation.
315                    Mais c'est plus facile pour l'instant, en particulier en prenant
316                    en compte le fait que cette combinaison des options IP
317                    et l'execution d'un renifleur est une condition extremement rare.
318                                                 --ANK (980813)
319                 */
320
321                 if (skb_cow(skb, skb_headroom(skb))) {
322                         IP_INC_STATS_BH(IPSTATS_MIB_INDISCARDS);
323                         goto drop;
324                 }
325                 iph = skb->nh.iph;
326
327                 if (ip_options_compile(NULL, skb))
328                         goto inhdr_error;
329
330                 opt = &(IPCB(skb)->opt);
331                 if (opt->srr) {
332                         struct in_device *in_dev = in_dev_get(dev);
333                         if (in_dev) {
334                                 if (!IN_DEV_SOURCE_ROUTE(in_dev)) {
335                                         if (IN_DEV_LOG_MARTIANS(in_dev) && net_ratelimit())
336                                                 printk(KERN_INFO
                        "source route option %u.%u.%u.%u -> %u.%u.%u.%u\n",
337                                                        NIPQUAD(iph->saddr),
                                                       NIPQUAD(iph->daddr));
338                                         in_dev_put(in_dev);
339                                         goto drop;
340                                 }
341                                 in_dev_put(in_dev);
342                         }
343                         if (ip_options_rcv_srr(skb))
344                                 goto drop;
345                 }
346         }
347
348         return dst_input(skb);
349
350 inhdr_error:
351         IP_INC_STATS_BH(IPSTATS_MIB_INHDRERRORS);
352 drop:
353         kfree_skb(skb);
```

```
354         return NET_RX_DROP;
355 }
```

Autrement dit :

- On déclare un descripteur de périphérique réseau, que l'on instantie avec celui associé au descripteur de tampon passé en argument.
- On déclare un en-tête IP, que l'on instantie avec celui associé au descripteur de tampon passé en argument.
- Si l'entrée de cache de destination `skb->dst` associée au descripteur de tampon passé en argument n'est pas définie, on essaie de la déterminer à l'aide de la fonction de redirection `ip_route_input()`, étudiée dans la sous-section suivante. Si ceci ne permet pas de lui en attribuer une, on libère le descripteur de tampon et on renvoie `NET_RX_DROP`.

  Cette entrée de cache n'est pas définie dans le cas d'un paquet provenant d'une trame, aussi fait-on appel à la fonction de redirection.
- Si la longueur de l'en-tête IP, telle qu'annoncée par celui-ci, est strictement supérieure à 5 mots, c'est que l'en-tête contient des options. On traite alors celles-ci (lignes 310–346), ce qui ne nous intéressera pas ici.
- On fait appel à la fonction chargée de poursuivre le traitement du paquet *via* la fonction `dst_input()`, étudiée dans une sous-section suivante, et on renvoie le code d'erreur fourni par celle-ci.

23.2.2 Fonction de redirection en entrée

La requête de redirection dans le traitement des paquets IP entrants s'effectue grâce à la fonction `ip_route_input()`, appelée par la fonction `ip_rcv_finish()` pour chaque paquet entrant. Elle est chargée d'identifier une entrée du cache de destination afin de déterminer la suite du parcours du paquet.

23.2.2.1 Ca de l'existence d'une entrée de cache

La fonction `ip_route_input()` prend en argument l'adresse du descripteur de tampon de socket du paquet ainsi que les adresses de destination et source du paquet, le type de service et l'adresse du descripteur de périphérique réseau chargé de la réception. Les quatre derniers arguments peuvent être déterminés à partir du premier. Cependant, comme ils doivent être à nouveau passés séparément, il est plus rapide qu'ils apparaissent en tant qu'arguments de la fonction `ip_route_input()`. Ceci permet de plus de traiter la structure `sk_buff` comme une "boîte noire" ou presque. Elle détermine l'entrée de cache adéquate si celle-ci apparaît dans le cache IP ou la suite du traitement avec `ip_route_input_mc()` ou `ip_route_input_slow()` sinon.

Code Linux 2.6.10

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

```
1816 int ip_route_input(struct sk_buff *skb, u32 daddr, u32 saddr,
1817                     u8 tos, struct net_device *dev)
1818 {
1819         struct rtable * rth;
1820         unsigned        hash;
1821         int iif = dev->ifindex;
1822
1823         tos &= IPTOS_RT_MASK;
1824         hash = rt_hash_code(daddr, saddr ^ (iif << 5), tos);
1825
1826         rcu_read_lock();
1827         for (rth = rcu_dereference(rt_hash_table[hash].chain); rth;
```

```
1828             rth = rcu_dereference(rth->u.rt_next)) {
1829                if (rth->fl.fl4_dst == daddr &&
1830                    rth->fl.fl4_src == saddr &&
1831                    rth->fl.iif == iif &&
1832                    rth->fl.oif == 0 &&
1833 #ifdef CONFIG_IP_ROUTE_FWMARK
1834                    rth->fl.fl4_fwmark == skb->nfmark &&
1835 #endif
1836                    rth->fl.fl4_tos == tos) {
1837                        rth->u.dst.lastuse = jiffies;
1838                        dst_hold(&rth->u.dst);
1839                        rth->u.dst.__use++;
1840                        RT_CACHE_STAT_INC(in_hit);
1841                        rcu_read_unlock();
1842                        skb->dst = (struct dst_entry*)rth;
1843                        return 0;
1844                }
1845                RT_CACHE_STAT_INC(in_hlist_search);
1846        }
1847        rcu_read_unlock();
1848
1849        /* La logique de reconnaissance de la multidiffusion est deplacee du cache
              de routage a ici.
1850           Le probleme etait que beaucoup trop de cartes Ethernet possedent des filtres
1851           materiels de multidiffusion mutiles/perdus :-( Il en resulte que l'hote du
1852           reseau de multidiffusion recoit beaucoup d'entrees de cache de routage
1853           inutiles, sortes de messages SDR provenant du monde entier. Nous essayons
              de nous en debarasser des maintenant.
1854           De fait, le logiciel IP de filtrage de multidiffussion ainsi fourni est
1855           organise raisonnablement. Il n'en resulte pas de ralentissement
1856           compare avec les entrees du cache de routage rejetees.
1857           Noter que les routeurs de multidiffusion ne sont pas leses puisque
1858           l'entree de cache de routage finit par etre creee.
1859         */
1860        if (MULTICAST(daddr)) {
1861                struct in_device *in_dev;
1862
1863                rcu_read_lock();
1864                if ((in_dev = __in_dev_get(dev)) != NULL) {
1865                        int our = ip_check_mc(in_dev, daddr, saddr,
1866                                skb->nh.iph->protocol);
1867                        if (our
1868 #ifdef CONFIG_IP_MROUTE
1869                            || (!LOCAL_MCAST(daddr) && IN_DEV_MFORWARD(in_dev))
1870 #endif
1871                            ) {
1872                                rcu_read_unlock();
1873                                return ip_route_input_mc(skb, daddr, saddr,
1874                                                         tos, dev, our);
1875                        }
1876                }
1877                rcu_read_unlock();
1878                return -EINVAL;
1879        }
1880        return ip_route_input_slow(skb, daddr, saddr, tos, dev);
1881 }
```

Autrement dit :

- On déclare une entrée de cache de routage et un numéro de hachage.
- On déclare un index de périphérique réseau d'entrée, que l'on instantie avec l'index de périphérique réseau du descripteur de périphérique passé en argument.

– On ne garde que les deux premier bits du type de service passé en argument.

Le masque `IPTOS_RT_MASK` pour ce faire est défini dans le fichier `linux/include/net/route.h`:

Code Linux 2.6.10

```
137 #define IPTOS_RT_MASK   (IPTOS_TOS_MASK & ~3)
```

– Un appel à la fonction `rt_hash_code()` permet de calculer, à partir des adresses et du type de service, l'index dans la table de hachage du cache de routage.

– On verrouille en écriture le dispositif de mise à jour/écriture.

La macro générale `rcu_dereference()` eet définie dans le fichier `linux/include/linux/rcupdate.h`:

Code Linux 2.6.10

```
225 /**
226  * rcu_dereference - charge un pointeur RCU-protege dans une
227  * section critique du cote lecture RCU. Ce pointeur peut etre
228  * dereference de facon sure plus tard.
229  *
230  * Insere des barrieres de memoire sur les architectures qui l'exigent
231  * (a l'heure actuelle seulement les Alpha) et, plus important, documente
232  * exactement les pointeurs qui sont proteges par RCU.
233  */
234
235 #define rcu_dereference(p)     ({ \
236                                 typeof(p) _________p1 = p; \
237                                 smp_read_barrier_depends(); \
238                                 (_________p1); \
239                                 })
```

– On parcourt la liste chaînée associée à cet index à la recherche d'une concordance avec les cinq paramètres de la clé (adresse de destination, adresse source, carte réseau d'entrée, carte réseau de sortie égale à zéro et type de service). Si on trouve une telle entrée de cache de routage:

  – On modifie quelques-uns de ses champs, en particulier l'heure de dernier accès et son nombre de références.

  – On incrémente les informations statistiques sur le nombre de fois qu'on est entré dans le cache.

  La macro `RT_CACHE_STAT_INC()` eet définie dans le fichier `linux/include/net/route.h`:

  Code Linux 2.6.10

```
107 #define RT_CACHE_STAT_INC(field)                                        \
108                 (per_cpu_ptr(rt_cache_stat, smp_processor_id())->field++)
```

  – On déverrouille en écriture le dispositif de mise à jour/écriture.

  – On renseigne le champ entrée de cache de destination du descripteur de périphérique passé en argument avec le début de cette entrée de cache de routage, ce qui était le but recherché, et on renvoie 0.

– Dans le cas où aucune entrée de cache de routage qui concorde n'est trouvée, ce qui arrive nécessairement pour le premier paquet de la session, la suite du traitement est assurée par l'une des deux fonctions suivantes:

  – La fonction `ip_route_imput_mc()` est appelée dans le cas d'une adresse de destination de multidiffusion, ce qui ne nous intéressera pas pour l'instant.

  La macro `MULTICAST()` est définie dans le fichier `linux/include/linux/in.h`:

  Code Linux 2.6.10

```
246 #define MULTICAST(x)    (((x) & htonl(0xf0000000)) == htonl(0xe0000000))
```

– La fonction `ip_route_input_slow()` est chargée de la gestion des adresses de destination individuelle. Ceci est le cas du premier paquet entrant en particulier. Nous allons étudier ce cas dans la sous-section suivante.

23.2.2.2 Structure de données pour la redirection

Rappelons qu'une entité du type `in_device` permet de détenir les paramètres propres à Internet pour un descripteur de périphérique. Le type est défini dans le fichier `linux/include/linux/inetdevice.h`: Code Linux 2.6.10

```
37 struct in_device
38 {
39         struct net_device       *dev;
40         atomic_t                refcnt;
41         int                     dead;
42         struct in_ifaddr        *ifa_list;      /* chaine ifaddr IP                  */
43         rwlock_t                mc_list_lock;
44         struct ip_mc_list       *mc_list;       /* chaine de filtres de
                                                      multidiffusion IP                 */
45         spinlock_t              mc_tomb_lock;
46         struct ip_mc_list       *mc_tomb;
47         unsigned long           mr_v1_seen;
48         unsigned long           mr_v2_seen;
49         unsigned long           mr_maxdelay;
50         unsigned char           mr_qrv;
51         unsigned char           mr_gq_running;
52         unsigned char           mr_ifc_count;
53         struct timer_list       mr_gq_timer;    /* minuteur general de requete       */
54         struct timer_list       mr_ifc_timer;   /* minuteur de changement d'interface */
55
56         struct neigh_parms      *arp_parms;
57         struct ipv4_devconf     cnf;
58         struct rcu_head         rcu_head;
59 };
```

dont les champs ont la signification suivante :

– Le champ `dev` est une référence arrière permettant de savoir à quel descripteur de périphérique est associée cette entité.
– Le champ `refcnt` est le compteur de références habituel. On modifie ce champ essentiellement à l'aide des fonctions `in_dev_get()` et `in_dev_put()` définies dans le même fichier.
– Le champ `dead` indique si le périphérique réseau IP est encore valide ou non.
– Le champ `ifa_list` renvoie à une liste de structures `in_ifaddr` contenant l'enregistrement des adresses IP de ce périphérique réseau. En effet Linux permet d'affecter plusieurs adresses IP à un périphérique réseau (*alias*).

  Le type `struct in_ifaddr` est défini dans le même fichier : Code Linux 2.6.10

```
85 struct in_ifaddr
86 {
87         struct in_ifaddr        *ifa_next;
88         struct in_device        *ifa_dev;
89         struct rcu_head         rcu_head;
90         u32                     ifa_local;
91         u32                     ifa_address;
92         u32                     ifa_mask;
93         u32                     ifa_broadcast;
94         u32                     ifa_anycast;
95         unsigned char           ifa_scope;
96         unsigned char           ifa_flags;
```

```
97         unsigned char           ifa_prefixlen;
98         char                    ifa_label[IFNAMSIZ];
99 };
```

Outre l'adresse IP (`ifa_address`), la structure `in_ifaddr` contient l'enregistrement de plusieurs autres paramètres, comme le masque de sous-réseau (`ifa_mask`), l'adresse de diffusion générale (`ifa_broadcast`), etc.

- Le champ `mc_list` est une liste constituée d'entités du type `ip_mc_list`. Chaque élément de cette liste mémorise les informations d'un groupe de multidiffusion IP. La multidiffusion ne nous intéressera pas pour l'instant.
- `arp_parms` renvoie sur une structure du type `neigh_parms` dans laquelle les principaux paramètres du protocole ARP sont mémorisés.

23.2.2.3 Cas d'une adresse de destination individuelle

Code Linux 2.6.10

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

```
1536 /*
1537  *      NOTE. Nous ecartons tous les paquets qui ont des adresses source
1538  *      locales, puisque tout paquet envoye en boucle de facon propre
1539  *      doit deja avoir une destination correcte attachee par la routine de sortie.
1540  *
1541  *      Un tel point de vue resoud deux gros problemes :
1542  *      1. Les peripheriques non simplexes sont traites correctement.
1543  *      2. Les essais de parodie IP sont filtre avec 100% de garantie.
1544  */
1545
1546 static int ip_route_input_slow(struct sk_buff *skb, u32 daddr, u32 saddr,
1547                         u8 tos, struct net_device *dev)
1548 {
1549         struct fib_result res;
1550         struct in_device *in_dev = in_dev_get(dev);
1551         struct in_device *out_dev = NULL;
1552         struct flowi fl = { .nl_u = { .ip4_u =
1553                                       { .daddr = daddr,
1554                                         .saddr = saddr,
1555                                         .tos = tos,
1556                                         .scope = RT_SCOPE_UNIVERSE,
1557 #ifdef CONFIG_IP_ROUTE_FWMARK
1558                                         .fwmark = skb->nfmark
1559 #endif
1560                                       } },
1561                             .iif = dev->ifindex };
1562         unsigned        flags = 0;
1563         u32             itag = 0;
1564         struct rtable * rth;
1565         unsigned        hash;
1566         u32             spec_dst;
1567         int             err = -EINVAL;
1568         int             free_res = 0;
1569
1570         /* IP est inhibee sur le peripherique. */
1571
1572         if (!in_dev)
1573                 goto out;
1574
1575         hash = rt_hash_code(daddr, saddr ^ (fl.iif << 5), tos);
1576
1577         /* Verification pour la plupart des martiens mysterieux, qui peuvent ne
1578            pas etre detectes par fib_lookup.
```

```
1579          */
1580
1581         if (MULTICAST(saddr) || BADCLASS(saddr) || LOOPBACK(saddr))
1582                 goto martian_source;
1583
1584         if (daddr == 0xFFFFFFFF || (saddr == 0 && daddr == 0))
1585                 goto brd_input;
1586
1587         /* Accepte l'adresse zero uniquement pour la diffusion generale ;
1588          * je ne sais meme pas le rectifier. J'attends les reclamations :-)
1589          */
1590         if (ZERONET(saddr))
1591                 goto martian_source;
1592
1593         if (BADCLASS(daddr) || ZERONET(daddr) || LOOPBACK(daddr))
1594                 goto martian_destination;
1595
1596         /*
1597          *      Maintenant nous sommes prets a router le paquet.
1598          */
1599         if ((err = fib_lookup(&fl, &res)) != 0) {
1600                 if (!IN_DEV_FORWARD(in_dev))
1601                         goto e_inval;
1602                 goto no_route;
1603         }
1604         free_res = 1;
1605
1606         RT_CACHE_STAT_INC(in_slow_tot);
1607
1608         if (res.type == RTN_BROADCAST)
1609                 goto brd_input;
1610
1611         if (res.type == RTN_LOCAL) {
1612                 int result;
1613                 result = fib_validate_source(saddr, daddr, tos,
1614                                              loopback_dev.ifindex,
1615                                              dev, &spec_dst, &itag);
1616                 if (result < 0)
1617                         goto martian_source;
1618                 if (result)
1619                         flags |= RTCF_DIRECTSRC;
1620                 spec_dst = daddr;
1621                 goto local_input;
1622         }
1623
1624         if (!IN_DEV_FORWARD(in_dev))
1625                 goto e_inval;
1626         if (res.type != RTN_UNICAST)
1627                 goto martian_destination;
1628
1629 #ifdef CONFIG_IP_ROUTE_MULTIPATH
1630         if (res.fi->fib_nhs > 1 && fl.oif == 0)
1631                 fib_select_multipath(&fl, &res);
1632 #endif
1633         out_dev = in_dev_get(FIB_RES_DEV(res));
1634         if (out_dev == NULL) {
1635                 if (net_ratelimit())
1636                         printk(KERN_CRIT "Bug in ip_route_input_slow(). "
1637                                          "Please, report\n");
1638                 goto e_inval;
1639         }
1640
```

```
1641         err = fib_validate_source(saddr, daddr, tos, FIB_RES_OIF(res), dev,
1642                                   &spec_dst, &itag);
1643         if (err < 0)
1644                 goto martian_source;
1645
1646         if (err)
1647                 flags |= RTCF_DIRECTSRC;
1648
1649         if (out_dev == in_dev && err && !(flags & (RTCF_NAT | RTCF_MASQ)) &&
1650             (IN_DEV_SHARED_MEDIA(out_dev) ||
1651              inet_addr_onlink(out_dev, saddr, FIB_RES_GW(res))))
1652                 flags |= RTCF_DOREDIRECT;
1653
1654         if (skb->protocol != htons(ETH_P_IP)) {
1655                 /* Non IP (par exemple ARP). Ne pas creer de route si ceci n'est pas
1656                  * valide pour le proxy arp. Les routes DNAT sont toujours valides.
1657                  */
1658                 if (out_dev == in_dev && !(flags & RTCF_DNAT))
1659                         goto e_inval;
1660         }
1661
1662         rth = dst_alloc(&ipv4_dst_ops);
1663         if (!rth)
1664                 goto e_nobufs;
1665
1666         atomic_set(&rth->u.dst.__refcnt, 1);
1667         rth->u.dst.flags= DST_HOST;
1668         if (in_dev->cnf.no_policy)
1669                 rth->u.dst.flags |= DST_NOPOLICY;
1670         if (in_dev->cnf.no_xfrm)
1671                 rth->u.dst.flags |= DST_NOXFRM;
1672         rth->fl.fl4_dst = daddr;
1673         rth->rt_dst     = daddr;
1674         rth->fl.fl4_tos = tos;
1675 #ifdef CONFIG_IP_ROUTE_FWMARK
1676         rth->fl.fl4_fwmark= skb->nfmark;
1677 #endif
1678         rth->fl.fl4_src = saddr;
1679         rth->rt_src     = saddr;
1680         rth->rt_gateway = daddr;
1681         rth->rt_iif     =
1682         rth->fl.iif     = dev->ifindex;
1683         rth->u.dst.dev  = out_dev->dev;
1684         dev_hold(rth->u.dst.dev);
1685         rth->idev       = in_dev_get(rth->u.dst.dev);
1686         rth->fl.oif     = 0;
1687         rth->rt_spec_dst= spec_dst;
1688
1689         rth->u.dst.input = ip_forward;
1690         rth->u.dst.output = ip_output;
1691
1692         rt_set_nexthop(rth, &res, itag);
1693
1694         rth->rt_flags = flags;
1695
1696 intern:
1697         err = rt_intern_hash(hash, rth, (struct rtable**)&skb->dst);
1698 done:
1699         in_dev_put(in_dev);
1700         if (out_dev)
1701                 in_dev_put(out_dev);
1702         if (free_res)
```

```
1703                 fib_res_put(&res);
1704 out:    return err;
1705
1706 brd_input:
1707         if (skb->protocol != htons(ETH_P_IP))
1708                 goto e_inval;
1709
1710         if (ZERONET(saddr))
1711                 spec_dst = inet_select_addr(dev, 0, RT_SCOPE_LINK);
1712         else {
1713                 err = fib_validate_source(saddr, 0, tos, 0, dev, &spec_dst,
1714                                           &itag);
1715                 if (err < 0)
1716                         goto martian_source;
1717                 if (err)
1718                         flags |= RTCF_DIRECTSRC;
1719         }
1720         flags |= RTCF_BROADCAST;
1721         res.type = RTN_BROADCAST;
1722         RT_CACHE_STAT_INC(in_brd);
1723
1724 local_input:
1725         rth = dst_alloc(&ipv4_dst_ops);
1726         if (!rth)
1727                 goto e_nobufs;
1728
1729         rth->u.dst.output= ip_rt_bug;
1730
1731         atomic_set(&rth->u.dst.__refcnt, 1);
1732         rth->u.dst.flags= DST_HOST;
1733         if (in_dev->cnf.no_policy)
1734                 rth->u.dst.flags |= DST_NOPOLICY;
1735         rth->fl.fl4_dst = daddr;
1736         rth->rt_dst     = daddr;
1737         rth->fl.fl4_tos = tos;
1738 #ifdef CONFIG_IP_ROUTE_FWMARK
1739         rth->fl.fl4_fwmark= skb->nfmark;
1740 #endif
1741         rth->fl.fl4_src = saddr;
1742         rth->rt_src     = saddr;
1743 #ifdef CONFIG_NET_CLS_ROUTE
1744         rth->u.dst.tclassid = itag;
1745 #endif
1746         rth->rt_iif     =
1747         rth->fl.iif     = dev->ifindex;
1748         rth->u.dst.dev  = &loopback_dev;
1749         dev_hold(rth->u.dst.dev);
1750         rth->idev       = in_dev_get(rth->u.dst.dev);
1751         rth->rt_gateway = daddr;
1752         rth->rt_spec_dst= spec_dst;
1753         rth->u.dst.input= ip_local_deliver;
1754         rth->rt_flags   = flags|RTCF_LOCAL;
1755         if (res.type == RTN_UNREACHABLE) {
1756                 rth->u.dst.input= ip_error;
1757                 rth->u.dst.error= -err;
1758                 rth->rt_flags   &= ~RTCF_LOCAL;
1759         }
1760         rth->rt_type    = res.type;
1761         goto intern;
1762
1763 no_route:
1764         RT_CACHE_STAT_INC(in_no_route);
```

```
1765         spec_dst = inet_select_addr(dev, 0, RT_SCOPE_UNIVERSE);
1766         res.type = RTN_UNREACHABLE;
1767         goto local_input;
1768
1769         /*
1770          *      Ne pas mettre les adresses de martiens en cache : elles devraient etre
                    loguees (RFC1812)
1771          */
1772 martian_destination:
1773         RT_CACHE_STAT_INC(in_martian_dst);
1774 #ifdef CONFIG_IP_ROUTE_VERBOSE
1775         if (IN_DEV_LOG_MARTIANS(in_dev) && net_ratelimit())
1776                 printk(KERN_WARNING "martian destination %u.%u.%u.%u from "
1777                         "%u.%u.%u.%u, dev %s\n",
1778                         NIPQUAD(daddr), NIPQUAD(saddr), dev->name);
1779 #endif
1780 e_inval:
1781         err = -EINVAL;
1782         goto done;
1783
1784 e_nobufs:
1785         err = -ENOBUFS;
1786         goto done;
1787
1788 martian_source:
1789
1790         RT_CACHE_STAT_INC(in_martian_src);
1791 #ifdef CONFIG_IP_ROUTE_VERBOSE
1792         if (IN_DEV_LOG_MARTIANS(in_dev) && net_ratelimit()) {
1793                 /*
1794                  *      Recommandation RFC1812 : si la source vient de Mars,
1795                  *      la seule indication est l'en-tete MAC.
1796                  */
1797                 printk(KERN_WARNING "martian source %u.%u.%u.%u from "
1798                         "%u.%u.%u.%u, on dev %s\n",
1799                         NIPQUAD(daddr), NIPQUAD(saddr), dev->name);
1800                 if (dev->hard_header_len) {
1801                         int i;
1802                         unsigned char *p = skb->mac.raw;
1803                         printk(KERN_WARNING "ll header: ");
1804                         for (i = 0; i < dev->hard_header_len; i++, p++) {
1805                                 printk("%02x", *p);
1806                                 if (i < (dev->hard_header_len - 1))
1807                                         printk(":");
1808                         }
1809                         printk("\n");
1810                 }
1811         }
1812 #endif
1813         goto e_inval;
1814 }
```

Autrement dit :

- On déclare un résultat de consultation des tables de routage.
- On déclare une adresse de structure de périphérique Internet entrant, que l'on instantie avec celle du champ correspondant du descripteur de périphérique passé en argument. Si on ne parvient pas à l'instantier, on renvoie l'opposé du code d'erreur `EINVAL` (lignes 1572–1573).

Code Linux 2.6.10

La fonction en ligne `in_dev_get()` est définie dans le même fichier :

```
143 static __inline__ struct in_device *
144 in_dev_get(const struct net_device *dev)
```

```
145 {
146         struct in_device *in_dev;
147
148         rcu_read_lock();
149         in_dev = dev->ip_ptr;
150         if (in_dev)
151                 atomic_inc(&in_dev->refcnt);
152         rcu_read_unlock();
153         return in_dev;
154 }
```

- On déclare une adresse de structure de périphérique Internet sortant, que l'on initialise à `NULL` puisqu'on est en train de s'occuper d'un paquet entrant.
- On déclare un descripteur de flux Internet `fl` que les paramètres passés à la fonction permettent d'instantier (lignes 1552–1561).
- On détermine l'index de hachage (ligne 1575).
- On vérifie, avant de procéder à une requête FIB, que les adresses ne comportent pas de valeurs non autorisées telles que des adresses source de multidiffusion ou des adresses dont le premier octet du préfixe réseau est nul (lignes 1581–1594). Ces paquets sont détruits et, dans le cas où la compilation du noyau est configurée avec le paramètre `CONFIG_IP_ROUTE_VERBOSE`, ceci est consigné dans le journal système. Le seul cas où le zéro peut être utilisé est celui de la diffusion générale, lorsqu'on utilise 0.0.0.0 comme adresse source et adresse de destination.

  Les macros `BADCLASS()`, `LOOPBACK()` et `ZERONET()` sont définies dans le fichier `linux/include/linux/in.h` :

Code Linux 2.6.10

```
244 /* Quelques cas speciaux utiles dans le noyau. */
245 #define LOOPBACK(x)     (((x) & htonl(0xff000000)) == htonl(0x7f000000))
246 #define MULTICAST(x)    (((x) & htonl(0xf0000000)) == htonl(0xe0000000))
247 #define BADCLASS(x)     (((x) & htonl(0xf0000000)) == htonl(0xf0000000))
248 #define ZERONET(x)      (((x) & htonl(0xff000000)) == htonl(0x00000000))
249 #define LOCAL_MCAST(x)  (((x) & htonl(0xFFFFFF00)) == htonl(0xE0000000))
```

- On effectue une requête FIB (ligne 1599). Dans le cas où aucune entrée appropriée n'est trouvée, la fonction est interrompue. Elle retourne alors un code d'erreur, ce qui donne lieu ensuite au rejet du paquet, dans la fonction `ip_rcv_finish()` à partir de laquelle cette fonction avait été appelée.

  Dans le cas où le paquet est bien destiné à l'ordinateur hôte, une entrée, correspondant à celle de l'ordinateur, est bien trouvée dans la table locale, la première consultée.

  La macro `IN_DEV_FORWARD()` est définie dans le fichier `linux/include/linux/netdevice.h` :

Code Linux 2.6.10

```
61 #define IN_DEV_FORWARD(in_dev)          ((in_dev)->cnf.forwarding)
```

- On spécifie qu'on pourra libérer le résultat.
- On met à jour les informations statistiques sur le nombre total de requêtes effectuées avec succès.
- À partir du résultat de la requête FIB, il est possible de voir si l'adresse de destination est une adresse locale (`RTN_LOCAL`), et donc si le paquet correspondant est destiné au système local. Nous ne nous intéresserons pas ici, par exemple, aux paquets destinés à la diffusion générale (ligne 1608).
- Dans le cas où le résultat de la requête FIB indique que la destination est locale (ligne 1611) :
  - On effectue un contrôle de l'adresse source, à l'aide de la fonction `fib_validate_source()` étudiée dans la sous-section suivante. Si cette fonction a renvoyé une erreur,

on incrémente les informations statistiques à propos des sources “martiennes” et on renvoie l'opposé du code d'erreur `EINVAL`.

- L'adresse IP de destination spécifique prend comme valeur l'adresse de destination, c'est-à-dire une de celles de l'ordinateur hôte (il peut en posséder plusieurs).
- On essaie d'allouer une nouvelle entrée de cache de destination avec `ipv4_dst_ops` comme ensemble d'opérations (ligne 1725). Si on n'y parvient pas, on renvoie l'opposé du code d'erreur `ENOBUFS`.

Code Linux 2.6.10

La fonction `dst_alloc()` est définie dans le fichier `linux/net/core/dst.c`:

```
115 void * dst_alloc(struct dst_ops * ops)
116 {
117         struct dst_entry * dst;
118
119         if (ops->gc && atomic_read(&ops->entries) > ops->gc_thresh) {
120                 if (ops->gc())
121                         return NULL;
122         }
123         dst = kmem_cache_alloc(ops->kmem_cachep, SLAB_ATOMIC);
124         if (!dst)
125                 return NULL;
126         memset(dst, 0, ops->entry_size);
127         atomic_set(&dst->__refcnt, 0);
128         dst->ops = ops;
129         dst->lastuse = jiffies;
130         dst->path = dst;
131         dst->input = dst_discard_in;
132         dst->output = dst_discard_out;
133 #if RT_CACHE_DEBUG >= 2
134         atomic_inc(&dst_total);
135 #endif
136         atomic_inc(&ops->entries);
137         return dst;
138 }
```

- On renseigne les champs de cette nouvelle entrée de cache (lignes 1729–1760), dont seuls quelques-uns demandent une explication:
  - On attribue comme pointeur de fonction `output()` l'adresse de la fonction `ip_rt_bug()` afin que le paquet ne puisse pas sortir du système.

    Code Linux 2.6.10

    Cette fonction `ip_rt_bug()` est définie dans le fichier `linux/net/ipv4/route.c`:

```
1369 static int ip_rt_bug(struct sk_buff *skb)
1370 {
1371         printk(KERN_DEBUG "ip_rt_bug: %u.%u.%u.%u -> %u.%u.%u.%u, %s\n",
1372                 NIPQUAD(skb->nh.iph->saddr), NIPQUAD(skb->nh.iph->daddr),
1373                 skb->dev ? skb->dev->name : "?");
1374         kfree_skb(skb);
1375         return 0;
1376 }
```

  - Rappelons que le champ `rt_gateway` concerne l'adresse de la passerelle ou celle du destinataire dans le réseau local à diffusion dans lequel nous nous trouvons. Ce champ prend donc dans notre cas l'adresse de destination, qui est une des adresses IP de l'ordinateur sur lequel nous nous trouvons (ligne 1751).
  - On attribue comme valeur du pointeur `input()` l'adresse de la fonction `ip_local_deliver()` (ligne 1753) afin de permettre au paquet d'être livré localement.

– La structure `rtable`, remplie dans son ensemble, est intégrée à la table de hachage du cache de routage (ligne 1697).
– On décrémente le compteur de référence des ressources utilisées jusque-là et le code d'erreur fourni par la fonction `rt_intern_hash()` est renvoyé, ce qui termine l'action de la fonction étudiée dans le cas d'une adresse de destination locale.

La fonction `in_dev_put()` est définie dans le fichier `linux/include/linux/netdevice.h` :

Code Linux 2.6.10

```
164 static inline void in_dev_put(struct in_device *idev)
165 {
166         if (atomic_dec_and_test(&idev->refcnt))
167                 in_dev_finish_destroy(idev);
168 }
```

23.2.2.4 Validation de l'adresse source

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

Code Linux 2.6.10

```
153 /* Etant donnes (source du paquet source, interface d'entree) et optionnellement
       (dst, oif, tos) :
154    - (principalement) verifier que la source est valide, c'est-a-dire qu'il ne
155      s'agit pas d'une adresse de diffusion generale ou de notre adresse locale.
156    - determiner depuis quelle interface "logique" ce paquet est arrive
157      et calculer l'adresse de "destination specifique".
158    - verifier que ce paquet provient d'une interface physique attendue.
159  */
160
161 int fib_validate_source(u32 src, u32 dst, u8 tos, int oif,
162                         struct net_device *dev, u32 *spec_dst, u32 *itag)
163 {
164         struct in_device *in_dev;
165         struct flowi fl = { .nl_u = { .ip4_u =
166                                       { .daddr = src,
167                                         .saddr = dst,
168                                         .tos = tos } },
169                             .iif = oif };
170         struct fib_result res;
171         int no_addr, rpf;
172         int ret;
173
174         no_addr = rpf = 0;
175         read_lock(&inetdev_lock);
176         in_dev = __in_dev_get(dev);
177         if (in_dev) {
178                 no_addr = in_dev->ifa_list == NULL;
179                 rpf = IN_DEV_RPFILTER(in_dev);
180         }
181         read_unlock(&inetdev_lock);
182
183         if (in_dev == NULL)
184                 goto e_inval;
185
186         if (fib_lookup(&fl, &res))
187                 goto last_resort;
188         if (res.type != RTN_UNICAST)
189                 goto e_inval_res;
190         *spec_dst = FIB_RES_PREFSRC(res);
191         fib_combine_itag(itag, &res);
192 #ifdef CONFIG_IP_ROUTE_MULTIPATH
```

```
193         if (FIB_RES_DEV(res) == dev || res.fi->fib_nhs > 1)
194 #else
195         if (FIB_RES_DEV(res) == dev)
196 #endif
197         {
198                 ret = FIB_RES_NH(res).nh_scope >= RT_SCOPE_HOST;
199                 fib_res_put(&res);
200                 return ret;
201         }
202         fib_res_put(&res);
203         if (no_addr)
204                 goto last_resort;
205         if (rpf)
206                 goto e_inval;
207         fl.oif = dev->ifindex;
208
209         ret = 0;
210         if (fib_lookup(&fl, &res) == 0) {
211                 if (res.type == RTN_UNICAST) {
212                         *spec_dst = FIB_RES_PREFSRC(res);
213                         ret = FIB_RES_NH(res).nh_scope >= RT_SCOPE_HOST;
214                 }
215                 fib_res_put(&res);
216         }
217         return ret;
218
219 last_resort:
220         if (rpf)
221                 goto e_inval;
222         *spec_dst = inet_select_addr(dev, 0, RT_SCOPE_UNIVERSE);
223         *itag = 0;
224         return 0;
225
226 e_inval_res:
227         fib_res_put(&res);
228 e_inval:
229         return -EINVAL;
230 }
```

Autrement dit :

- On déclare l'adresse d'une structure de périphérique Internet, que l'on essaie d'instantier avec l'adresse du champ associé du descripteur de périphérique passé en argument. Si on n'y parvient pas, on renvoie l'opposé du code d'erreur `EINVAL`.

  La fonction en ligne `__in_dev_get()` est définie dans le fichier `linux/include/linux/-netdevice.h` :

Code Linux 2.6.10

```
156 static __inline__ struct in_device *
157 __in_dev_get(const struct net_device *dev)
158 {
159         return (struct in_device*)dev->ip_ptr;
160 }
```

- On déclare un flux Internet, dont on renseigne les champs grace aux valeurs passées en argument.
- On effectue une requête FIB (ligne 186). Si celle-ci réussit, on renseigne l'argument (passé en adresse) `spec_dst` et on renvoie 0.

23.2.3 Choix de la fonction de traitement du paquet

La fonction `dst_input()` est définie dans le fichier `linux/include/net/dst.h`: Code Linux 2.6.10

```
234 /* Paquet entrant de reseau a transport.  */
235 static inline int dst_input(struct sk_buff *skb)
236 {
237         int err;
238
239         for (;;) {
240                 err = skb->dst->input(skb);
241
242                 if (likely(err == 0))
243                         return err;
244                 /* Oh, Jamal... Je ne vous pardonnerai pas cette pagaille. :-) */
245                 if (unlikely(err != NET_XMIT_BYPASS))
246                         return err;
247         }
248 }
```

Elle se contente de faire appel à la fonction `skb->dst->input()` associée au descripteur de tampon et de renvoyer le code d'erreur fourni par celle-ci.

La fonction pointée, comme nous venons de le voir est l'une des trois suivantes:

- `ip_local_deliver()` pour les paquets locaux à diffusion individuelle.
- `ip_forward()` en cas d'acheminement d'un paquet à diffusion individuelle.
- `ip_mr_input()` pour les paquets de multidiffusion.

Dans la suite de ce chapitre, nous ne nous intéressons qu'aux paquets à remettre localement. Les deux autres cas seront traités plus tard.

23.3 Remise locale des paquets

23.3.1 Première étape : réassemblage des fragments

La fonction `ip_local_deliver()` est définie dans le fichier `linux/net/ipv4/ip_input.c`: Code Linux 2.6.10

```
266 /*
267  *      Livre les paquets IP aux couches superieures de protocole.
268  */
269 int ip_local_deliver(struct sk_buff *skb)
270 {
271         /*
272          *      Reassemble les fragments IP.
273          */
274
275         if (skb->nh.iph->frag_off & htons(IP_MF|IP_OFFSET)) {
276                 skb = ip_defrag(skb);
277                 if (!skb)
278                         return 0;
279         }
280
281         return NF_HOOK(PF_INET, NF_IP_LOCAL_IN, skb, skb->dev, NULL,
282                        ip_local_deliver_finish);
283 }
```

Autrement dit la première tâche consiste à réassembler d'éventuels fragments du paquet à l'aide de la fonction `ip_defrag()`, ce qui sera étudié dans au chapitre 36 consacré à la fragmentation IP. Nous supposerons dans une première étape que ceci n'est pas nécessaire. Suit un appel d'un point d'ancrage `NF_IP_LOCAL_IN`, la fonction à venir étant `ip_local_deliver_finish()`.

23.3.2 Gestion des protocoles de transport dans la couche IP

23.3.2.1 Descripteur de protocole de transport

Sous Linux, chaque protocole de couche de transport de la suite TCP/IP est enregistré à travers un **descripteur de protocole** (*protocol handler* en anglais), entité du type `struct net_protocol`. Ce type est défini dans le fichier en-tête `linux/include/net/protocol.h`:

Code Linux 2.6.10

```
36 /* Ceci est utilise pour enregistrer les protocoles. */
37 struct net_protocol {
38         int                     (*handler)(struct sk_buff *skb);
39         void                    (*err_handler)(struct sk_buff *skb, u32 info);
40         int                     no_policy;
41 };
```

Il comprend essentiellement les fonction de réception des tampons de socket et de traitement des messages d'erreurs ICMP transmis par la couche IP.

Par exemple, l'entité `udp_protocol` pour UDP est définie dans le fichier `linux/net/ipv4/af_inet.c`:

Code Linux 2.6.10

```
981 static struct net_protocol udp_protocol = {
982          .handler =      udp_rcv,
983          .err_handler =  udp_err,
984          .no_policy =    1,
985 };
```

23.3.2.2 Table de hachage des protocoles de transport

La couche IP gère les descripteurs de protocole de transport grâce à une table de hachage appelée `inet_protos[]`, déclarée dans le fichier `linux/net/ipv4/protocol.c`:

Code Linux 2.6.10

```
51 struct net_protocol *inet_protos[MAX_INET_PROTOS];
```

Le nombre maximum de protocoles de couche de transport est spécifié par la constante `MAX_INET_PROTOS`, définie dans le fichier `linux/include/net/protocol.h`:

Code Linux 2.6.10

```
33 #define MAX_INET_PROTOS 256             /* Doit etre une puissance de 2         */
```

23.3.2.3 Ajout d'un protocole de transport

L'ajout d'un descripteur de protocole à la table de hachage s'effectue grâce à la fonction `inet_add_protocol()`, définie dans le fichier `include/net/ipv4/protocol.c`:

Code Linux 2.6.10

```
54 /*
55  *      Ajoute un descripteur de protocole aux tables de hachage
56  */
57
58 int inet_add_protocol(struct inet_protocol *prot, unsigned char protocol)
59 {
60         int hash, ret;
61
62         hash = protocol & (MAX_INET_PROTOS - 1);
63
64         spin_lock_bh(&inet_proto_lock);
65         if (inet_protos[hash]) {
66                 ret = -1;
67         } else {
68                 inet_protos[hash] = prot;
69                 ret = 0;
70         }
71         spin_unlock_bh(&inet_proto_lock);
```

```
72
73         return ret;
74 }
```

23.3.2.4 Initialisation de la table de hachage des protocoles de transport

La table de hachage des protocoles de transport est initialisée dans le corps de la fonction `inet_init()`, déjà rencontrée, définie dans le fichier `linux/net/ipv4/af_inet.c` :

Code Linux 2.6.10

```
1017 static int __init inet_init(void)
1018 {
[...]
1051         /*
1052          *      Ajouter tous les protocoles de base.
1053          */
1054
1055         if (inet_add_protocol(&icmp_protocol, IPPROTO_ICMP) < 0)
1056                 printk(KERN_CRIT "inet_init: Cannot add ICMP protocol\n");
1057         if (inet_add_protocol(&udp_protocol, IPPROTO_UDP) < 0)
1058                 printk(KERN_CRIT "inet_init: Cannot add UDP protocol\n");
1059         if (inet_add_protocol(&tcp_protocol, IPPROTO_TCP) < 0)
1060                 printk(KERN_CRIT "inet_init: Cannot add TCP protocol\n");
1061 #ifdef CONFIG_IP_MULTICAST
1062         if (inet_add_protocol(&igmp_protocol, IPPROTO_IGMP) < 0)
1063                 printk(KERN_CRIT "inet_init: Cannot add IGMP protocol\n");
1064 #endif
[...]
1122 }
```

23.3.3 Deuxième étape : démultiplexage de la couche de transport

La fonction `ip_local_deliver_finish()` est chargée du démultiplexage, c'est-à-dire de déterminer à quel protocole de couche de transport envoyer la partie données du paquet, c'est-à-dire le datagramme dans le cas UDP. Elle vérifie d'abord si le paquet est brut, c'est-à-dire s'il ne correspond à aucun des protocoles enregistrés. Dans le cas contraire, le protocole de transport est déterminé et la partie données du paquet (le tampon de socket dans le cas de Linux) est passé à ce protocole.

Cette fonction est définie dans le fichier `linux/net/ipv4/ip_input.c` :

Code Linux 2.6.10

```
199 static inline int ip_local_deliver_finish(struct sk_buff *skb)
200 {
201         int ihl = skb->nh.iph->ihl*4;
202
203 #ifdef CONFIG_NETFILTER_DEBUG
204         nf_debug_ip_local_deliver(skb);
205 #endif /*CONFIG_NETFILTER_DEBUG*/
206
207         __skb_pull(skb, ihl);
208
209         /* Reference libre auparavant : nous n'en avons pas besoin plus longtemps,
210            et elle peut contenir le module ip_conntrack charge indefiniment. */
211         nf_reset(skb);
212
213         /* Pointe sur le datagramme IP, juste passer l'en-tete. */
214         skb->h.raw = skb->data;
215
216         rcu_read_lock();
217         {
218                 /* Note : voir raw.c et net/raw.h, RAWV4_HTABLE_SIZE==MAX_INET_PROTOS */
219                 int protocol = skb->nh.iph->protocol;
```

```
220                 int hash;
221                 struct sock *raw_sk;
222                 struct net_protocol *ipprot;
223
224         resubmit:
225                 hash = protocol & (MAX_INET_PROTOS - 1);
226                 raw_sk = sk_head(&raw_v4_htable[hash]);
227
228                 /* S'il peut y avoir une socket brute, nous devons verifier -
229                  * sinon inutile de prendre des precautions
230                  */
231                 if (raw_sk)
232                         raw_v4_input(skb, skb->nh.iph, hash);
233
234                 if ((ipprot = rcu_dereference(inet_protos[hash])) != NULL) {
235                         int ret;
236
237                         if (!ipprot->no_policy &&
238                             !xfrm4_policy_check(NULL, XFRM_POLICY_IN, skb)) {
239                                 kfree_skb(skb);
240                                 goto out;
241                         }
242                         ret = ipprot->handler(skb);
243                         if (ret < 0) {
244                                 protocol = -ret;
245                                 goto resubmit;
246                         }
247                         IP_INC_STATS_BH(IPSTATS_MIB_INDELIVERS);
248                 } else {
249                         if (!raw_sk) {
250                                 if (xfrm4_policy_check(NULL, XFRM_POLICY_IN, skb)) {
251                                         IP_INC_STATS_BH(IPSTATS_MIB_INUNKNOWNPROTOS);
252                                         icmp_send(skb, ICMP_DEST_UNREACH,
253                                                   ICMP_PROT_UNREACH, 0);
254                                 }
255                         } else
256                                 IP_INC_STATS_BH(IPSTATS_MIB_INDELIVERS);
257                         kfree_skb(skb);
258                 }
259         }
260  out:
261         rcu_read_unlock();
262
263         return 0;
264 }
```

Autrement dit :

- On déclare une longueur d'en-tête IP, que l'on instantie avec celle de l'en-tête IP associé au descripteur de tampon passé en argument.
- Si on a choisi l'option de compilation du noyau concernant le débogage du filtrage réseau, ce qui ne nous intéressera pas ici, on fait appel à la fonction `nf_debug_ip_local_deliver()`.
- On ajuste la zone des données du tampon de socket pour qu'elle puisse contenir l'en-tête.
- On positionne le compteur de références au filtrage à `NULL`.

La fonction en ligne `nf_reset()` est définie dans le fichier `linux/include/linux/skbuff.h` :

Code Linux 2.6.10

```
1151 static inline void nf_reset(struct sk_buff *skb)
1152 {
1153         nf_conntrack_put(skb->nfct);
```

```
1154         skb->nfct = NULL;
1155 #ifdef CONFIG_NETFILTER_DEBUG
1156         skb->nf_debug = 0;
1157 #endif
1158 }
```

- On passe l'en-tête IP en faisant pointer l'en-tête pour un paquet brut au début de la zone des données du paquet.
- On positionne le verrou d'écriture pour le mécanisme de mise à jour de l'écriture-copie.
- On déclare un numéro de protocole de la couche transport, que l'on instantie avec celui associé à l'en-tête IP, lui-même associé au descripteur de tampon passé en argument.
- On déclare un numéro de hachage, un descripteur de couche de transport (non vraiment utilisé) et une adresse de descripteur de protocole.
- On détermine le numéro de hachage du protocole de la couche transport.
- On essaie d'instantier le descripteur de couche transport à partir de ce numéro de hachage en cherchant d'abord dans la table des paquets bruts.

  La table `raw_v4_htable[]` est déclarée dans le fichier `linux/net/ipv4/raw.c`: Code Linux 2.6.10

```
30 #define RAWV4_HTABLE_SIZE       MAX_INET_PROTOS
31 extern struct hlist_head raw_v4_htable[RAWV4_HTABLE_SIZE];
```

  La fonction en ligne `sk_head()` est définie dans le fichier `linux/include/net/sock.h`: Code Linux 2.6.10

```
269 /*
270  * Routines d'aide pour les listes de hachage
271  */
272 static inline struct sock *__sk_head(struct hlist_head *head)
273 {
274         return hlist_entry(head->first, struct sock, sk_node);
275 }
276
277 static inline struct sock *sk_head(struct hlist_head *head)
278 {
279         return hlist_empty(head) ? NULL : __sk_head(head);
280 }
```

- Si on a obtenu ainsi un descripteur de couche de transport de paquet brut, on fait appel à la fonction `raw_v4_input()`, que nous devrions étudier à propos des paquets bruts, mais qui ne nous intéressera pas dans cet ouvrage.
- On essaie d'obtenir le descripteur de protocole grâce au numéro de hachage.
  - Si on en obtient un :
    - On déclare une valeur de retour.
    - Dans le cas où le descripteur de protocole exige une police mais qu'on ne peut pas obtenir celle-ci, on libère le descripteur de tampon, on déverrouille le dispositif de mise à jour de l'écriture-copie et on renvoie 0.
    - On fait appel à la routine de traitement correspondante de la couche de transport, ce qui est le but principal de cette fonction.

      Nous avons vu, par exemple, qu'il s'agit de `udp_rcv()` dans le cas de UDP, c'est-à-dire d'un numéro de protocole égal à 17.
    - Si une erreur s'est produite lors de l'appel à cette routine, on essaie à nouveau.
    - On incrémente l'information statistique sur le nombre de paquets IP reçus et livrés à la couche transport correspondante.

- Si on n'obtient pas de descripteur de protocole de couche de transport :
  - S'il s'agit d'un paquet brut, on incrémente l'information statistique sur le nombre de paquets IP reçus et livrés à la couche transport correspondante.
  - Sinon on incrémente l'information statistique sur le nombre de protocoles inconnus et on envoie un message ICMP "Destination non atteignable" à l'expéditeur du paquet, grâce à la fonction `icmp_send()`.
- Dans les deux cas, on libère le descripteur de tampon de socket.

- On déverrouille le dispositif de mise à jour de l'écriture-copie et on renvoie 0.