Chapitre 22

Cache de routage

Bien que les structures de données de la FIB autorisent une interrogation relativement rapide, le coût qu'impliquerait l'utilisation de ce genre d'interrogation pour chaque paquet IP serait globalement très élevé. C'est la raison pour laquelle le noyau Linux comporte, parallèlement à la FIB, un cadre supplémentaire chargé de mémoriser le résultat des dernières (en général 256) demandes de redirection utilisées offrant l'avantage d'être très rapide d'accès: le **cache de routage**. Lors d'une redirection, le cache de routage est toujours consulté en premier. Dans un deuxième temps, mais uniquement s'il n'existe pas d'entrée correspondant à la demande, une requête est envoyée à la FIB. Une nouvelle entrée de cache est alors immédiatement générée avec le résultat de la requête FIB.

22.1 Structure du cache de routage

Le **cache de routage** contient des informations, indépendantes de tout protocole, sur l'itinéraire du paquet, entre autres la carte réseau sur laquelle le paquet doit quitter l'ordinateur (dans le cas où il y en a plusieurs) et éventuellement un renvoi sur un en-tête MAC déjà stocké dans le cache d'en-tête physique.

22.1.1 Cache de destination

22.1.1.1 Entrée du cache de destination

Une entrée du cache de destination est une entité du type `dst_entry`. Celui-ci est défini dans le fichier en-tête `linux/include/net/dst.h`:

Code Linux 2.6.10

```
1 /*
2  * net/dst.h    Definitions du cache de destination independant du protocole.
3  *
4  * Auteurs :    Alexey Kuznetsov, <kuznet@ms2.inr.ac.ru>
5  *
6  */
7
8 #ifndef _NET_DST_H
9 #define _NET_DST_H
[...]
29 /* Chaque dst_entry a un compteur de reference et est situee dans une (des) liste(s) parent.
30  * Lorsqu'elle est retiree d'une liste parent, elle est "liberee" (dst_free).
31  * Apres cela, elle entre dans un etat letal (dst->obsolete > 0) et si son refcnt
32  * est nul, elle peut etre detruite immediatement, sinon elle est ajoutee a
33  * la liste gc et le glaneur de cellules verifie periodiquement le refcnt.
34  */
35
36 struct sk_buff;
37
38 struct dst_entry
39 {
40         struct dst_entry        *next;
41         atomic_t                __refcnt;       /* references client    */
42         int                     __use;
43         struct dst_entry        *child;
44         struct net_device       *dev;
45         int                     obsolete;
46         int                     flags;
47 #define DST_HOST                1
48 #define DST_NOXFRM              2
49 #define DST_NOPOLICY            4
50 #define DST_NOHASH              8
51         unsigned long           lastuse;
52         unsigned long           expires;
53
54         unsigned short          header_len;     /* plus d'espace est requis en tete */
55         unsigned short          trailer_len;    /* espace a reserver en queue       */
56
57         u32                     metrics[RTAX_MAX];
58         struct dst_entry        *path;
59
60         unsigned long           rate_last;      /* taux limitant pour ICMP */
61         unsigned long           rate_tokens;
62
63         int                     error;
64
65         struct neighbour        *neighbour;
```

```
66          struct hh_cache         *hh;
67          struct xfrm_state       *xfrm;
68
69          int                     (*input)(struct sk_buff*);
70          int                     (*output)(struct sk_buff*);
71
72 #ifdef CONFIG_NET_CLS_ROUTE
73          __u32                   tclassid;
74 #endif
75
76          struct  dst_ops         *ops;
77          struct rcu_head         rcu_head;
78
79          char                    info[0];
80 };
```

Il s'agit d'un mélange de champs constituant les données proprement dites et de champs de contrôle (qui correspondraient à un descripteur en d'autres circonstances). Les champs constituant les données sont les suivants :

- Le champ `dev` spécifie l'interface de sortie vers le routeur.
- Les champs `header_len` et `trailer_len` spécifient la taille de l'en-tête et du suffixe d'une trame (pour cette interface).
- L'instance de voisinage `neighbour` spécifie le prochain routeur ou l'hôte de destination dans le cas d'un sysème à diffusion. Dans le cas d'un système point à point, le champ `dev` suffit.

  Nous n'avons pas besoin de connaître le type `neighbour` pour l'instant puisque seule l'adresse intervient.
- L'entrée du cache matériel `hh` comporte un en-tête de trame qu'il ne reste plus qu'à copier, comme nous l'avons déjà vu au chapitre 14.
- Les fonctions `input()` et `output()` sont chargées de spécifier comment poursuivre le traitement respectivement lors de la réception et de l'émission d'un paquet IP adapté à l'entrée du cache.

  Dans le cas des paquets IP entrants, le pointeur `input()` fera référence à `ip_local_deliver()` pour un paquet à diffusion individuelle à adresser localement, à `ip_mr_input()` pour un paquet de multidiffusion et à `ip_error()` lorsque la redirection est impossible, la destination n'étant pas accessible. Le pointeur `output()` fera référence à `ip_output()` pour un paquet à diffusion individuelle et à `ip_mc_output()` pour un paquet de multidiffusion.
- Nous reviendrons ci-dessous sur les opérations sur les entrées du cache de destination.

Les champs de contrôle sont les suivants :

- Les entrées du cache de routage appartiennent à une liste chaînée, d'où l'intérêt du champ `next`.
- Le compteur de references `__refcnt` permet de détruire l'entrée de cache lorsque celui-ci vaut zéro.
- Le compteur d'utilisation `__use` est incrémenté chaque fois que l'entrée de cache est utilisée.
- Le champ `child` permet l'utilisation d'une autre liste chaînée.
- Le champ `obsolete` est une autre sécurité évitant de détruire l'entrée de cache par inadvertance. On ne doit la détruire que si ce champ est strictement positif.
- Le champ `flags` est un champ de bits dont quatre seulement sont utilisés, représentés par les constantes symboliques `DST_HOST`, `DST_NOXFRM`, `DST_NOPOLICY` et `DST_NOHASH`.

- Le champ `lastuse` spécifie le moment, en jiffies, auquel l'entrée de cache a été utilisée pour la dernière fois.
- Le champ `expires` spécifie le moment, en jiffies, auquel l'entrée de cache ne sera plus valable.
- La liste d'entrée de cache `path` forme le chemin.
- Les champs `rate_last` et `rate_tokens` concernent ICMP, qui ne nous intéresse pas dans cet ouvrage.
- Le champ `rcu_head` permet un appel pour une lecture–mise à jour.
- Le champ `info[]` ne représente rien, il indique simplement que les informations suivent.

22.1.1.2 Opérations sur les entrées de cache de destination

Les opérations sur les entrées du cache de destination, le champ `ops` de la structure précédente, dépendent de la famille de protocoles. Le type `struct dst_ops` est defini dans le fichier `linux/-include/net/dst.h`:

Code Linux 2.6.10

```
83 struct dst_ops
84 {
85         unsigned short          family;
86         unsigned short          protocol;
87         unsigned                gc_thresh;
88
89         int                     (*gc)(void);
90         struct dst_entry *      (*check)(struct dst_entry *, __u32 cookie);
91         void                    (*destroy)(struct dst_entry *);
92         void                    (*ifdown)(struct dst_entry *, int how);
93         struct dst_entry *      (*negative_advice)(struct dst_entry *);
94         void                    (*link_failure)(struct sk_buff *);
95         void                    (*update_pmtu)(struct dst_entry *dst, u32 mtu);
96         int                     (*get_mss)(struct dst_entry *dst, u32 mtu);
97         int                     entry_size;
98
99         atomic_t                entries;
100         kmem_cache_t            *kmem_cachep;
101 };
```

Il s'agit toujours, pour les entrées IPv4, de la structure `ipv4_dst_ops` définie globalement. Celle-ci comporte un certain nombre de fonctions par l'intermédiaire desquelles les utilisateurs d'entrée de cache de routage, tels que TCP ou ARP, peuvent rendre compte du routage, par exemple au sujet de connexions annulées. Elle est définie dans le fichier `linux/net/ipv4/route.c`:

Code Linux 2.6.0

```
148 static struct dst_ops ipv4_dst_ops = {
149         .family =               AF_INET,
150         .protocol =             __constant_htons(ETH_P_IP),
151         .gc =                   rt_garbage_collect,
152         .check =                ipv4_dst_check,
153         .destroy =              ipv4_dst_destroy,
154         .ifdown =               ipv4_dst_ifdown,
155         .negative_advice =      ipv4_negative_advice,
156         .link_failure =         ipv4_link_failure,
157         .update_pmtu =          ip_rt_update_pmtu,
158         .entry_size =           sizeof(struct rtable),
159 };
```

Nous étudierons ces fonctions au fur et à mesure des besoins.

22.1.2 Entrée de cache de routage

Une entrée de cache de routage est une entité du type `struct rtable`, défini dans le fichier `linux/include/net/route.h`:

Code Linux 2.6.10

```
6   *              Definitions pour les routeurs IP.
7   *
8   * Version :    @(#)route.h     1.0.4   05/27/93
9   *
10  * Auteurs :    Ross Biro, <bir7@leland.Stanford.Edu>
11  *              Fred N. van Kempen, <waltje@uWalt.NL.Mugnet.ORG>
12  * Rectifications :
13  *              Alan Cox         :      A reformate. A ajoute ip_rt_local()
14  *              Alan Cox         :      Support pour les parametres TCP.
15  *              Alexey Kuznetsov :      Changement majeur dans le code de routage.
16  *              Mike McLagan     :      Routage par la source
17  *              Robert Olsson    :      Ajout des statistiques rt_cache
[...]
50 struct rtable
51 {
52          union
53          {
54                  struct dst_entry        dst;
55                  struct rtable           *rt_next;
56          } u;
57
58          struct in_device        *idev;
59
60          unsigned                rt_flags;
61          unsigned                rt_type;
62
63          __u32                   rt_dst; /* Destination du circuit */
64          __u32                   rt_src; /* Source du circuit      */
65          int                     rt_iif;
66
67          /* Info sur la station voisine */
68          __u32                   rt_gateway;
69
70          /* Cles de consultation du cache */
71          struct flowi            fl;
72
73          /* Informations cachees diverses */
74          __u32                   rt_spec_dst; /* Destination specifique RFC1122 */
75          struct inet_peer        *peer;       /* info sur le partenaire durable */
76 };
```

dont la signification de certains des champs est précisée ci-dessous :

- Le champ `u` permet de référencer l'élément suivant de la table, soit globalement, soit une partie seulement.
- Les champs `rt_src` et `rt_dst` spécifient l'adresse source et l'adresse de destination des paquets IP traités à l'aide de l'entrée.
- L'identifiant d'interface physique `iif` désigne soit l'interface de sortie, soit l'interface d'entrée.
- Le champ `ret_gateway` comporte soit l'adresse de destination, si l'ordinateur de destination est situé dans le réseau local dans lequel nous nous trouvons, soit du routeur le plus proche de celui-ci sinon (soit situé dans le réseau local, soit en lien direct).
- Le champ `key` est utilisé comme clé lors de la recherche dans le cadre d'un routage.

22.1.3 Cache de routage

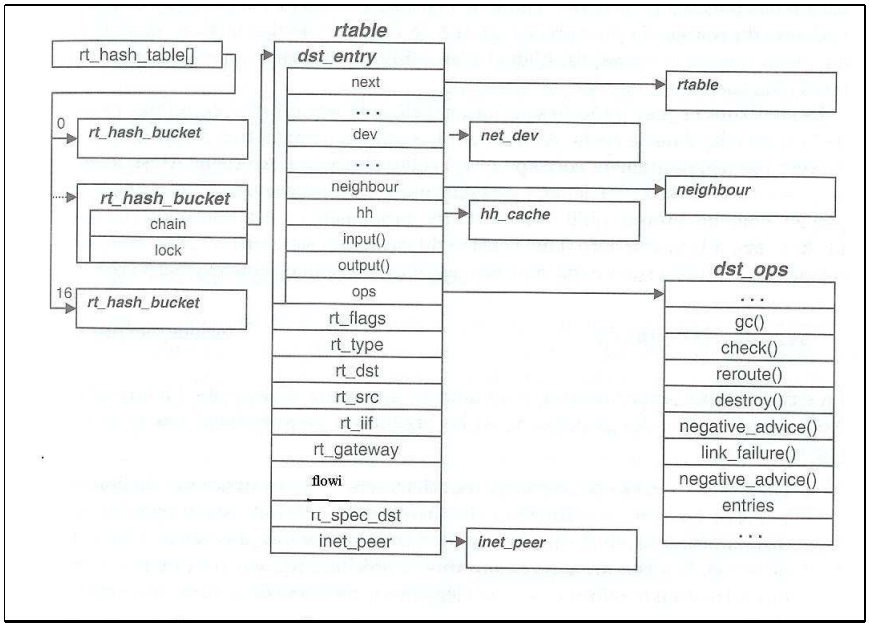

FIG. 22.1 – *Structure du cache de routage*

Comme le montre la figure 22.1, le cache de routage `rt_hash_table[]` est une table de hachage unique d'entrées de cache, présentées dans une suite linéaire pour une valeur de hachage identique.

Code Linux 2.6.10

Elle est déclarée dans le fichier `linux/net/ipv4/route.c`:

```
197 struct rt_hash_bucket {
198         struct rtable   *chain;
199         spinlock_t      lock;
200 } __attribute__((__aligned__(8)));
201
202 static struct rt_hash_bucket    *rt_hash_table;
```

Dans chaque seau de hachage, de type `rt_hash_bucket`, le champ `chain` forme le point d'ancrage d'une liste de structures `rtable` représentant les entrées du cache. L'accès à cette liste est régenté par le verrou tournant `lock`.

Code Linux 2.6.10

La dimension `rhash_entries` de ce tableau est déclarée à la fin du même fichier :

```
2696 static __initdata unsigned long rhash_entries;
```

22.2 Initialisation du cache de routage IP

L'initialisation du cache de routage IP est effectuée par la fonction `ip_rt_init()`, définie dans le fichier `linux/net/ipv4/route.c`:

Code Linux 2.6.10

```
2706 int __init ip_rt_init(void)
2707 {
2708         int i, order, goal, rc = 0;
2709
2710         rt_hash_rnd = (int) ((num_physpages ^ (num_physpages>>8)) ^
```

```
2711                            (jiffies ^ (jiffies >> 7)));
2712
2713 #ifdef CONFIG_NET_CLS_ROUTE
2714         for (order = 0;
2715              (PAGE_SIZE << order) < 256 * sizeof(struct ip_rt_acct) * NR_CPUS; order++)
2716                 /* RIEN */;
2717         ip_rt_acct = (struct ip_rt_acct *)__get_free_pages(GFP_KERNEL, order);
2718         if (!ip_rt_acct)
2719                 panic("IP: failed to allocate ip_rt_acct\n");
2720         memset(ip_rt_acct, 0, PAGE_SIZE << order);
2721 #endif
2722
2723         ipv4_dst_ops.kmem_cachep = kmem_cache_create("ip_dst_cache",
2724                                                      sizeof(struct rtable),
2725                                                      0, SLAB_HWCACHE_ALIGN,
2726                                                      NULL, NULL);
2727
2728         if (!ipv4_dst_ops.kmem_cachep)
2729                 panic("IP: failed to allocate ip_dst_cache\n");
2730
2731         goal = num_physpages >> (26 - PAGE_SHIFT);
2732         if (rhash_entries)
2733                 goal = (rhash_entries * sizeof(struct rt_hash_bucket)) >> PAGE_SHIFT;
2734         for (order = 0; (1UL << order) < goal; order++)
2735                 /* RIEN */;
2736
2737         do {
2738                 rt_hash_mask = (1UL << order) * PAGE_SIZE /
2739                         sizeof(struct rt_hash_bucket);
2740                 while (rt_hash_mask & (rt_hash_mask - 1))
2741                         rt_hash_mask--;
2742                 rt_hash_table = (struct rt_hash_bucket *)
2743                         __get_free_pages(GFP_ATOMIC, order);
2744         } while (rt_hash_table == NULL && --order > 0);
2745
2746         if (!rt_hash_table)
2747                 panic("Failed to allocate IP route cache hash table\n");
2748
2749         printk(KERN_INFO "IP: routing cache hash table of %u buckets, %ldKbytes\n",
2750                rt_hash_mask,
2751                (long) (rt_hash_mask * sizeof(struct rt_hash_bucket)) / 1024);
2752
2753         for (rt_hash_log = 0; (1 << rt_hash_log) != rt_hash_mask; rt_hash_log++)
2754                 /* RIEN */;
2755
2756         rt_hash_mask--;
2757         for (i = 0; i <= rt_hash_mask; i++) {
2758                 spin_lock_init(&rt_hash_table[i].lock);
2759                 rt_hash_table[i].chain = NULL;
2760         }
2761
2762         ipv4_dst_ops.gc_thresh = (rt_hash_mask + 1);
2763         ip_rt_max_size = (rt_hash_mask + 1) * 16;
2764
2765         rt_cache_stat = alloc_percpu(struct rt_cache_stat);
2766         if (!rt_cache_stat)
2767                 return -ENOMEM;
2768
2769         devinet_init();
2770         ip_fib_init();
2771
2772         init_timer(&rt_flush_timer);
```

```
2773         rt_flush_timer.function = rt_run_flush;
2774         init_timer(&rt_periodic_timer);
2775         rt_periodic_timer.function = rt_check_expire;
2776         init_timer(&rt_secret_timer);
2777         rt_secret_timer.function = rt_secret_rebuild;
2778 
2779         /* Tous les minuteurs, demarres au depart du systeme, tendent
2780            a se synchroniser. Perturbons-les un peu.
2781          */
2782         rt_periodic_timer.expires = jiffies + net_random() % ip_rt_gc_interval +
2783                                     ip_rt_gc_interval;
2784         add_timer(&rt_periodic_timer);
2785 
2786         rt_secret_timer.expires = jiffies + net_random() % ip_rt_secret_interval +
2787                 ip_rt_secret_interval;
2788         add_timer(&rt_secret_timer);
2789 
2790 #ifdef CONFIG_PROC_FS
2791         {
2792         struct proc_dir_entry *rtstat_pde = NULL; /* keep gcc happy */
2793         if (!proc_net_fops_create("rt_cache", S_IRUGO, &rt_cache_seq_fops) ||
2794             !(rtstat_pde = create_proc_entry("rt_cache", S_IRUGO,
2795                                              proc_net_stat))) {
2796                 free_percpu(rt_cache_stat);
2797                 return -ENOMEM;
2798         }
2799         rtstat_pde->proc_fops = &rt_cpu_seq_fops;
2800         }
2801 #ifdef CONFIG_NET_CLS_ROUTE
2802         create_proc_read_entry("rt_acct", 0, proc_net, ip_rt_acct_read, NULL);
2803 #endif
2804 #endif
2805 #ifdef CONFIG_XFRM
2806         xfrm_init();
2807         xfrm4_init();
2808 #endif
2809         return rc;
2810 }
```

Autrement dit :

- On initialise la variable `rt_hash_rnd` (pour *RaNDom*), qui sert à pertuber les minuteurs pour qu'ils ne démarrent pas tous exactement au même moment, de façon plus ou moins aléatoire (ligne 2710–2711).

Code Linux 2.6.10

  Cette variable est déclarée au début du même fichier :

```
205 static unsigned int            rt_hash_rnd;
```

- On essaie de renseigner le champ de début du cache de la structure d'opérations `ipv4_dst_ops`. Si on n'y parvient pas, on affiche un message de panique (lignes 2723–2729).
- On initialise la variable `goal` (lignes 2731–2733) et on détermine à la main le logarithme binaire `order` de celle-ci (lignes 2734–2735).
- On essaie de déterminer le masque de hachage de la table de hachage et d'obtenir de l'emplacement mémoire pour la table de hachage. Si on n'y parvient pas, on affiche un message de panique (lignes 2737–2747).

Code Linux 2.6.10

  La variable `rt_hash_mask` est déclarée au début du même fichier :

```
203 static unsigned                rt_hash_mask;
```

- On affiche un message système spécifiant les caractéristiques du cache de routage IP.

– On calcule à la main le logarithme binaire du masque de hachage (lignes 2753–2754).
La variable `rt_hash_log` est déclarée au début du même fichier : Code Linux 2.6.10

```
204 static int                    rt_hash_log;
```

– On initialise la table de hachage (lignes 2756–2760).
– On renseigne le champ `gc_thresh` concernant le glaneur de cellules de `ipv4_dst_ops`.
– On initialise la taille maximum `ip_rt_max_size` du cache de routage.
Cette variable est définie dans le fichier `linux/net/ipv4/route.c` : Code Linux 2.6.10

```
113 int ip_rt_max_size;
```

– On essaie d'allouer de la place en mémoire vive pour les statistiques concernant le cache de routage sur chaque microprocesseur. Si on n'y parvient pas, on renvoie l'opposé du code d'erreur `ENOMEM` (lignes 2765–2767).
Les statistiques concernant le cache de routage sont détenues dans une structure du type `rt_cache_stat`, définie dans le fichier `linux/include/net/route.h` : Code Linux 2.6.10

```
86  struct rt_cache_stat
87  {
88          unsigned int in_hit;
89          unsigned int in_slow_tot;
90          unsigned int in_slow_mc;
91          unsigned int in_no_route;
92          unsigned int in_brd;
93          unsigned int in_martian_dst;
94          unsigned int in_martian_src;
95          unsigned int out_hit;
96          unsigned int out_slow_tot;
97          unsigned int out_slow_mc;
98          unsigned int gc_total;
99          unsigned int gc_ignored;
100         unsigned int gc_goal_miss;
101         unsigned int gc_dst_overflow;
102         unsigned int in_hlist_search;
103         unsigned int out_hlist_search;
104 };
```

Une variable du même nom est définie dans le fichier `linux/net/ipv4/route.c` : Code Linux 2.6.10

```
207 struct rt_cache_stat *rt_cache_stat;
```

– On initialise ce qui concerne les périphériques réseau pour IPv4.
La fonction `devinet_init()` est définie dans le fichier `linux/net/ipv4/devinet.c` : Code Linux 2.6.10

```
1482 void __init devinet_init(void)
1483 {
1484         register_gifconf(PF_INET, inet_gifconf);
1485         register_netdevice_notifier(&ip_netdev_notifier);
1486         rtnetlink_links[PF_INET] = inet_rtnetlink_table;
1487 #ifdef CONFIG_SYSCTL
1488         devinet_sysctl.sysctl_header =
1489                 register_sysctl_table(devinet_sysctl.devinet_root_dir, 0);
1490         devinet_sysctl_register(NULL, &ipv4_devconf_dflt);
1491 #endif
1492 }
```

– On initialise les deux tables de routage statiques, de la façon vue au chapitre 21.

– On initialise les minuteurs du cache de routage et les fonctions associées. On pertube un peu les minuteurs pour qu'ils ne ssoient pas synchronisés.

Les minuteurs `rt_flush_timer`, `rt_periodic_timer` et `rt_secret_timer` sont définis dans le fichier `linux/net/ipv4/route.c`:

Code Linux 2.6.10

```
131 static struct timer_list rt_flush_timer;
132 static struct timer_list rt_periodic_timer;
133 static struct timer_list rt_secret_timer;
```

Les fonctions `rt_run_flush()`, `rt_check_expire()` et `rt_secret_rebuild()` seront étudiées au fur et à mesure des besoins.

– On renvoie le code de retour, initialisé à 0.

22.3 Interface du cache de routage vers les fonctions de redirection

L'implémentation du cache de routage se trouve presque entièrement contenue dans le fichier `linux/net/ipv4/route.c`. Elle n'est pratiquement pas encapsulée. Ainsi il n'existe pas de fonction indépendante chargée de rechercher/créer une entrée du cache: dans les deux fonctions de l'interface principale concernant le traitement des paquets IP (à savoir `ip_route_input()` et `ip_route_output()`), cette recherche/création est réalisée directement sur la structure de données.

22.3.1 Calcul de l'index dans la table de routage

La fonction `rt_hash_code()` permet de calculer, à partir des adresses source et de destination et du type de service, l'index dans la table de hachage du cache de routage. Elle permet également de parcourir, le cas échéant, la liste ancrée dans le membre `chain`, ce qui permet de trouver éventuellement une entrée de cache identique au niveau des adresses, du type de service et éventuellement de l'identifiant `fwmark`.

Code Linux 2.6.10

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

```
212 static unsigned int rt_hash_code(u32 daddr, u32 saddr, u8 tos)
213 {
214         return (jhash_3words(daddr, saddr, (u32) tos, rt_hash_rnd)
215                 & rt_hash_mask);
216 }
```

22.3.2 Insertion dans la table de hachage

Les fonctions de redirection font appel à la fonction `rt_intern_hash()` afin d'insérer dans la table de hachage une structure `rtable` déjà remplie pour l'essentiel. Les nouvelles entrées sont toujours insérées en première position dans une liste de résolution de collisions éventuelles. Dans le cas où une entrée utilisant une clé identique figure déjà dans la liste, celle-ci passe en début de liste et la nouvelle entrée est rejetée.

Outre l'opération d'insertion, la fonction `rt_intern_hash()` se charge également d'obtenir le pointeur contenu dans la structure `rtable` sur une structure `neighbour` correspondant au routeur le plus proche ou au système de destination.

Cette fonction est définie dans le fichier `linux/net/ipv4/route.c`: Code Linux 2.6.10

```
776 static int rt_intern_hash(unsigned hash, struct rtable *rt, struct rtable **rp)
777 {
778         struct rtable   *rth, **rthp;
779         unsigned long   now;
780         struct rtable   *cand, **candp;
781         u32             min_score;
782         int             chain_length;
783         int attempts = !in_softirq();
784
785 restart:
786         chain_length = 0;
787         min_score = ~(u32)0;
788         cand = NULL;
789         candp = NULL;
790         now = jiffies;
791
792         rthp = &rt_hash_table[hash].chain;
793
794         spin_lock_bh(&rt_hash_table[hash].lock);
795         while ((rth = *rthp) != NULL) {
796                 if (compare_keys(&rth->fl, &rt->fl)) {
797                         /* Mettons-la en premier */
798                         *rthp = rth->u.rt_next;
799                         /*
800                          * Puisque la consultation s'effectue sans verrouillage, le retrait
801                          * doit etre visible par une autre CPU faiblement ordonnee avant
802                          * l'insertion au debut de la chaine de hachage.
803                          */
804                         rcu_assign_pointer(rth->u.rt_next,
805                                            rt_hash_table[hash].chain);
806                         /*
807                          * Puisque la consultation s'effectue sans verrouillage, la mise a
808                          * jour (ecriture) doit etre ordonnee pour la consistance sur SMP.
809                          */
810                         rcu_assign_pointer(rt_hash_table[hash].chain, rth);
811
812                         rth->u.dst.__use++;
813                         dst_hold(&rth->u.dst);
814                         rth->u.dst.lastuse = now;
815                         spin_unlock_bh(&rt_hash_table[hash].lock);
816
817                         rt_drop(rt);
818                         *rp = rth;
819                         return 0;
820                 }
821
822                 if (!atomic_read(&rth->u.dst.__refcnt)) {
823                         u32 score = rt_score(rth);
824
825                         if (score <= min_score) {
826                                 cand = rth;
827                                 candp = rthp;
828                                 min_score = score;
829                         }
830                 }
831
832                 chain_length++;
833
834                 rthp = &rth->u.rt_next;
835         }
836
```

```
837         if (cand) {
838                 /* ip_rt_gc_elasticity utilise comme longueur moyenne la longueur
839                  * de chaine. Lorsqu'elle est depassee le gc devient reellement agressif.
840                  *
841                  * La seconde limite est moins certaine. Pour l'instant elle permet
842                  * seulement 2 entrees par chaine. Nous verrons.
843                  */
844                 if (chain_length > ip_rt_gc_elasticity) {
845                         *candp = cand->u.rt_next;
846                         rt_free(cand);
847                 }
848         }
849 
850         /* Essayer de lier la route a arp seulement si c'est une route de
851            sortie ou un chemin de reacheminement individuel.
852          */
853         if (rt->rt_type == RTN_UNICAST || rt->fl.iif == 0) {
854                 int err = arp_bind_neighbour(&rt->u.dst);
855                 if (err) {
856                         spin_unlock_bh(&rt_hash_table[hash].lock);
857 
858                         if (err != -ENOBUFS) {
859                                 rt_drop(rt);
860                                 return err;
861                         }
862 
863                         /* Les tables de voisins sont pleines et rien ne peut etre
864                            libere. Essayer de reduire le cache de routage, il est plus
865                            plausible qu'il detienne des enregistrements de voisins.
866                          */
867                         if (attempts-- > 0) {
868                                 int saved_elasticity = ip_rt_gc_elasticity;
869                                 int saved_int = ip_rt_gc_min_interval;
870                                 ip_rt_gc_elasticity     = 1;
871                                 ip_rt_gc_min_interval   = 0;
872                                 rt_garbage_collect();
873                                 ip_rt_gc_min_interval   = saved_int;
874                                 ip_rt_gc_elasticity     = saved_elasticity;
875                                 goto restart;
876                         }
877 
878                         if (net_ratelimit())
879                                 printk(KERN_WARNING "Neighbour table overflow.\n");
880                         rt_drop(rt);
881                         return -ENOBUFS;
882                 }
883         }
884 
885         rt->u.rt_next = rt_hash_table[hash].chain;
886 #if RT_CACHE_DEBUG >= 2
887         if (rt->u.rt_next) {
888                 struct rtable *trt;
889                 printk(KERN_DEBUG "rt_cache @%02x: %u.%u.%u.%u", hash,
890                        NIPQUAD(rt->rt_dst));
891                 for (trt = rt->u.rt_next; trt; trt = trt->u.rt_next)
892                         printk(" . %u.%u.%u.%u", NIPQUAD(trt->rt_dst));
893                 printk("\n");
894         }
895 #endif
896         rt_hash_table[hash].chain = rt;
897         spin_unlock_bh(&rt_hash_table[hash].lock);
898         *rp = rt;
```

```
899         return 0;
900 }
```

Autrement dit :

- On déclare une liste d'entrées de cache de routage, que l'on initialise avec la liste chaînée correspondant à l'index de hachage passé en argument (ligne 783).
- On verrouille cette liste chaînée.
- On parcourt cette liste chaînée à la recherche d'une entrée dont les clés concordent avec l'entrée passé en argument. Si on en trouve une (en fait elle est unique), on la place en premier dans la liste, on incrémente son nombre d'utilisateurs, on renseigne l'heure de dernière utilisation avec l'heure en cours, on déverrouille la liste chaînée, on décrémente le compteur de référence de l'entrée passée en argument et on renvoie 0 (lignes 786–827).

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

```
769 static inline int compare_keys(struct flowi *fl1, struct flowi *fl2)
770 {
771         return memcmp(&fl1->nl_u.ip4_u, &fl2->nl_u.ip4_u, sizeof(fl1->nl_u.ip4_u)) == 0
                &&
772                fl1->oif     == fl2->oif &&
773                fl1->iif     == fl2->iif;
774 }
```

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

```
452 static __inline__ void rt_drop(struct rtable *rt)
453 {
454         ip_rt_put(rt);
455         call_rcu_bh(&rt->u.dst.rcu_head, dst_rcu_free);
456 }
```

- Si on ne l'a pas trouvée dans la liste, si le type est diffusion individuelle et si l'interface d'entrée n'est pas spécifiée, on essaie de trouver un voisin (ligne 846)
-
-