# Sixième partie

# Complément sur IP : la fragmentation

# Chapitre 36

# Fragmentation pour la couche réseau : le cas de IPv4

Nous avons vu au chapitre 33 comment envoyer un paquet IP ordinaire depuis un ordinateur hôte en supposant qu'il n'avait pas besoin dêtre fragmenté. Nous avons également vu au chapitre 23 comment il était reçu par un ordinateur hôte en supposant qu'il n'avait pas été fragmenté au cours de sa route. Nous allons étudier dans ce chapitre comment fragmenter et réassembler un paquet.

## 36.1 Fragmentation

Comme nous l'avons vu au chapitre 33, la fonction de fragmentation `ip_fragment()` est éventuellement appelée par la fonction `ip_output()` en spécifiant deux arguments : un descripteur de tampon de socket et la fonction `ip_finish_output()`.

Code Linux 2.6.10

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

```
420 /*
421  *      Ce datagramme IP est trop gros pour etre envoye en un seul morceau. Coupons-le en
422  *      morceaux plus petits (chacun de taille egale a [celle de] l'en-tete IP plus
423  *      un bloc de donnees de la partie des donnees IP originelle) qui conviennent a une seule
424  *      trame du peripherique, et placons une telle trame dans la file d'attente d'envoi.
425  */
426
427 int ip_fragment(struct sk_buff *skb, int (*output)(struct sk_buff*))
428 {
429         struct iphdr *iph;
430         int raw = 0;
431         int ptr;
432         struct net_device *dev;
433         struct sk_buff *skb2;
434         unsigned int mtu, hlen, left, len, ll_rs;
435         int offset;
436         int not_last_frag;
437         struct rtable *rt = (struct rtable*)skb->dst;
438         int err = 0;
439
440         dev = rt->u.dst.dev;
441
442         /*
443          *      Pointer sur l'en-tete du datagramme IP.
444          */
445
446         iph = skb->nh.iph;
447
448         if (unlikely((iph->frag_off & htons(IP_DF)) && !skb->local_df)) {
449                 icmp_send(skb, ICMP_DEST_UNREACH, ICMP_FRAG_NEEDED,
450                           htonl(dst_pmtu(&rt->u.dst)));
451                 kfree_skb(skb);
452                 return -EMSGSIZE;
453         }
454
455         /*
456          *      Initialiser les valeurs de depart.
457          */
458
459         hlen = iph->ihl * 4;
460         mtu = dst_pmtu(&rt->u.dst) - hlen;      /* Taille de l'espace des donnees */
461
462         /* Lorsque frag_list est donnee, l'utiliser. Verifier d'abord sa validite :
463          * certaines transformations pourraient creer de fausses frag_list ou couper des
464          * listes existantes, ce qui n'est pas interdit. Dans ce cas ne pas copier.
465          *
466          * PLUS TARD : Cette etape peut etre fusionnee avec la generation reelle de fragments,
467          * nous pouvons passer a la copie lorsque nous voyons le premier fragment mauvais.
468          */
469         if (skb_shinfo(skb)->frag_list) {
470                 struct sk_buff *frag;
471                 int first_len = skb_pagelen(skb);
472
473                 if (first_len - hlen > mtu ||
474                     ((first_len - hlen) & 7) ||
```

```
475                     (iph->frag_off & htons(IP_MF|IP_OFFSET)) ||
476                     skb_cloned(skb))
477                         goto slow_path;
478
479                 for (frag = skb_shinfo(skb)->frag_list; frag; frag = frag->next) {
480                         /* Geometrie correcte. */
481                         if (frag->len > mtu ||
482                             ((frag->len & 7) && frag->next) ||
483                             skb_headroom(frag) < hlen)
484                             goto slow_path;
485
486                         /* skb partiellement clone ? */
487                         if (skb_shared(frag))
488                                 goto slow_path;
489                 }
490
491                 /* Chaque chose est OK. Generer ! */
492
493                 err = 0;
494                 offset = 0;
495                 frag = skb_shinfo(skb)->frag_list;
496                 skb_shinfo(skb)->frag_list = NULL;
497                 skb->data_len = first_len - skb_headlen(skb);
498                 skb->len = first_len;
499                 iph->tot_len = htons(first_len);
500                 iph->frag_off |= htons(IP_MF);
501                 ip_send_check(iph);
502
503                 for (;;) {
504                         /* Preparer l'en-tete de la trame suivante,
505                          * avant de laisser tomber la precedente. */
506                         if (frag) {
507                                 frag->h.raw = frag->data;
508                                 frag->nh.raw = __skb_push(frag, hlen);
509                                 memcpy(frag->nh.raw, iph, hlen);
510                                 iph = frag->nh.iph;
511                                 iph->tot_len = htons(frag->len);
512                                 ip_copy_metadata(frag, skb);
513                                 if (offset == 0)
514                                         ip_options_fragment(frag);
515                                 offset += skb->len - hlen;
516                                 iph->frag_off = htons(offset>>3);
517                                 if (frag->next != NULL)
518                                         iph->frag_off |= htons(IP_MF);
519                                 /* Pret, completer la somme de controle */
520                                 ip_send_check(iph);
521                         }
522
523                         err = output(skb);
524
525                         if (err || !frag)
526                                 break;
527
528                         skb = frag;
529                         frag = skb->next;
530                         skb->next = NULL;
531                 }
532
533                 if (err == 0) {
534                         IP_INC_STATS(IPSTATS_MIB_FRAGOKS);
535                         return 0;
536                 }
```

```
537
538                 while (frag) {
539                         skb = frag->next;
540                         kfree_skb(frag);
541                         frag = skb;
542                 }
543                 IP_INC_STATS(IPSTATS_MIB_FRAGFAILS);
544                 return err;
545         }
546
547 slow_path:
548         left = skb->len - hlen;         /* Espace par trame */
549         ptr = raw + hlen;               /* La ou on demarre */
550
551 #ifdef CONFIG_BRIDGE_NETFILTER
552         /* Pour un trafic IP sur pont encapsule par exemple dans un en-tete vlan,
553          * nous avons besoin de faire de la place pour l'en-tete encapsulant */
554         ll_rs = LL_RESERVED_SPACE_EXTRA(rt->u.dst.dev, nf_bridge_pad(skb));
555         mtu -= nf_bridge_pad(skb);
556 #else
557         ll_rs = LL_RESERVED_SPACE(rt->u.dst.dev);
558 #endif
559         /*
560          *      Fragmenter le datagramme.
561          */
562
563         offset = (ntohs(iph->frag_off) & IP_OFFSET) << 3;
564         not_last_frag = iph->frag_off & htons(IP_MF);
565
566         /*
567          *      Conserver une copie des donnees jusqu'a ce qu'on ait fini.
568          */
569
570         while(left > 0) {
571                 len = left;
572                 /* SI : il ne convient pas, utiliser 'mtu' - l'espace restant des donnees */
573                 if (len > mtu)
574                         len = mtu;
575                 /* SI : nous ne sommes pas en train d'envoyer la fin du paquet
576                    alors aligner le prochain depart sur une frontiere de huit octets */
577                 if (len < left) {
578                         len &= ~7;
579                 }
580                 /*
581                  *      Allouer un tampon.
582                  */
583
584                 if ((skb2 = alloc_skb(len+hlen+ll_rs, GFP_ATOMIC)) == NULL) {
585                         NETDEBUG(printk(KERN_INFO "IP: frag: no memory for new fragment!\n"));
586                         err = -ENOMEM;
587                         goto fail;
588                 }
589
590                 /*
591                  *      Mettre les donnees dans le paquet
592                  */
593
594                 ip_copy_metadata(skb2, skb);
595                 skb_reserve(skb2, ll_rs);
596                 skb_put(skb2, len + hlen);
597                 skb2->nh.raw = skb2->data;
598                 skb2->h.raw = skb2->data + hlen;
```

```
599
600                 /*
601                  *      Charger la memoire pour le fragment a tout proprietaire
602                  *      qui pourrait le posseder
603                  */
604
605                 if (skb->sk)
606                         skb_set_owner_w(skb2, skb->sk);
607
608                 /*
609                  *      Copier l'en-tete de paquet dans le nouveau tampon.
610                  */
611
612                 memcpy(skb2->nh.raw, skb->data, hlen);
613
614                 /*
615                  *      Copier un bloc du datagramme IP.
616                  */
617                 if (skb_copy_bits(skb, ptr, skb2->h.raw, len))
618                         BUG();
619                 left -= len;
620
621                 /*
622                  *      Renseigner les champs du nouvel en-tete.
623                  */
624                 iph = skb2->nh.iph;
625                 iph->frag_off = htons((offset >> 3));
626
627                 /* ANK : pas beau mais truc effectif. Mise a jour des options seulement si
628                  * le segment a etre fragmente etait LE PREMIER (autrement,
629                  * les options ont deja ete rectifiees) et le faire UNE FOIS
630                  * sur le skb initial, de telle facon que les fragments suivants
631                  * heriteront des options rectifiees.
632                  */
633                 if (offset == 0)
634                         ip_options_fragment(skb);
635
636                 /*
637                  *      Ajoute AC : Si nous sommes en train de fragmenter un fragment qui
638                  *                  n'est pas le dernier fragment alors conserver MF sur
                                       chaque bit
639                  */
640                 if (left > 0 || not_last_frag)
641                         iph->frag_off |= htons(IP_MF);
642                 ptr += len;
643                 offset += len;
644
645                 /*
646                  *      Mettre ce fragment dans la file d'attente en envoi.
647                  */
648
649                 IP_INC_STATS(IPSTATS_MIB_FRAGCREATES);
650
651                 iph->tot_len = htons(len + hlen);
652
653                 ip_send_check(iph);
654
655                 err = output(skb2);
656                 if (err)
657                         goto fail;
658         }
659         kfree_skb(skb);
```

```
660         IP_INC_STATS(IPSTATS_MIB_FRAGOKS);
661         return err;
662
663 fail:
664         kfree_skb(skb);
665         IP_INC_STATS(IPSTATS_MIB_FRAGFAILS);
666         return err;
667 }
```

Autrement dit :

- On déclare un en-tête IP, un index brut, un pointeur, un descripteur de périphérique réseau, un deuxième descripteur de tampon de socket, une MTU, une longueur d'en-tête, le nombre de fragments traités, une longueur, un décalage, un test indiquant que ce n'est pas le dernier fragment, une entrée de table de routage et un code de retour.
- On instantie l'entrée de table de routage avec celle associée au descripteur de tampon passé en argument.
- On instantie le descripteur de périphérique réseau avec celui associé à cette entrée de table de routage.
- On instantie l'en-tête IP avec celle associée au descripteur de tampon passé en argument.
- Si l'en-tête IP spécifie que l'on ne doit pas fragmenter le paquet, on envoie un message ICMP à l'émetteur indiquant que l'on doit fragmenter le paquet, on libère le descripteur de tampon passé en argument et on renvoie l'opposé du code d'erreur `EMSGSIZE`.
- On initialise la taille de l'en-tête IP en octets, c'est-à-dire quatre fois celle indiquée par l'en-tête lui-même (exprimée en mots).
- On détermine la taille des données dans un fragment, à savoir l'unité de transfert du réseau moins la longueur de l'en-tête.
- Si une liste de fragmentation existe déjà dans le descripteur de tampon :
  - On déclare un descripteur de tampon pour un fragment.
  - On déclare une longueur initiale du descripteur de tampon passé en argument, que l'on instantie avec la longueur totale des fragments de celui-ci.

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

Code Linux 2.6.10

```
665 static inline int skb_pagelen(const struct sk_buff *skb)
666 {
667         int i, len = 0;
668
669         for (i = (int)skb_shinfo(skb)->nr_frags - 1; i >= 0; i--)
670                 len += skb_shinfo(skb)->frags[i].size;
671         return len + skb_headlen(skb);
672 }
```

  - Si cette longueur est supérieure à l'unité de transfert ou si le descripteur de socket passé en argument est cloné, on ne peut pas effectuer le traitement rapide ci-dessous.
  - On vérifie chacun des fragments de la liste :
    - Si la longueur d'un tel fragment est supérieure à l'unité de transfert ou si la taille de l'espace de tête du tampon dont le descripteur est passé en argument est inférieure à celle de l'en-tête, on ne peut pas effectuer le traitement rapide ci-dessous.
    - Si aucun descripteur de couche réseau n'est associé au fragment ou au descripteur de tampon passé en argument, on ne peut pas effectuer le traitement rapide ci-dessous.

  - Si le fragment est partagé, on ne peut pas effectuer le traitement rapide ci-dessous.

- Sinon on peut passer à un traitement rapide :

  - Le code de retour et le décalage sont initialisés à zéro.
  - Le descripteur de tampon de fragment est instantié avec le premier élément de la liste des fragments du descripteur de tampon passé en argument et on indique dans ce dernier descripteur qu'on l'a récupéré.
  - On spécifie comme longueur des données du descripteur de tampon passé en argument la longueur initiale moins la longueur de l'en-tête.
  - On spécifie comme longueur totale du descripteur de tampon passé en argument la longueur initiale.
  - On spécifie comme longueur totale du paquet dans l'en-tête IP la longueur initiale.
  - On spécifie dans l'en-tête IP que ce paquet est un fragment suivi d'autres fragments.
  - On ajoute la somme de contrôle à l'en-tête IP.
  - On entre dans une boucle (pseudo-)infinie pour traiter tous les fragments, en fait tant que le descripteur de tampon de fragment est non nul :

    - On fait pointer l'en-tête matériel brut du descripteur de fragment sur le début des données du fragment.
    - On fait pointer l'en-tête réseau brut du descripteur de fragment à l'emplacement adéquat et on instantie cet en-tête avec l'en-tête IP défini auparavant.
    - On fait pointer l'adresse de l'en-tête IP à l'adresse de l'en-tête réseau brut du descripteur de fragment.
    - On initialise la longueur totale du paquet dans cet en-tête avec la longueur du fragment.
    - On copie les méta-données du descripteur de tampon dans le fragment.

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

Code Linux 2.6.10

```
386 static void ip_copy_metadata(struct sk_buff *to, struct sk_buff *from)
387 {
388         to->pkt_type = from->pkt_type;
389         to->priority = from->priority;
390         to->protocol = from->protocol;
391         to->security = from->security;
392         to->dst = dst_clone(from->dst);
393         to->dev = from->dev;
394
395         /* Copie les drapeaux de chaque fragment. */
396         IPCB(to)->flags = IPCB(from)->flags;
397
398 #ifdef CONFIG_NET_SCHED
399         to->tc_index = from->tc_index;
400 #endif
401 #ifdef CONFIG_NETFILTER
402         to->nfmark = from->nfmark;
403         to->nfcache = from->nfcache;
404         /* L'association de connexion est la meme que dans le paquet
               pre-frag */
405         nf_conntrack_put(to->nfct);
406         to->nfct = from->nfct;
407         nf_conntrack_get(to->nfct);
```

```
408         to->nfctinfo = from->nfctinfo;
409 #ifdef CONFIG_BRIDGE_NETFILTER
410         nf_bridge_put(to->nf_bridge);
411         to->nf_bridge = from->nf_bridge;
412         nf_bridge_get(to->nf_bridge);
413 #endif
414 #ifdef CONFIG_NETFILTER_DEBUG
415         to->nf_debug = from->nf_debug;
416 #endif
417 #endif
418 }
```

La macro `IPCB()` de copie du bloc de contrôle est définie dans le fichier `linux/include/net/ip.h`:

Code Linux 2.6.10

```
57 #define IPCB(skb) ((struct inet_skb_parm*)((skb)->cb))
```

– Si le décalage du fragment est nul, il s'agit du premier fragment. On remplit les renseignements sur les options grâce à la fonction `ip_options_fragment()` définie dans le fichier `linux/net/ipv4/ip_options.c`:

Code Linux 2.6.10

```
202 /*
203  *      "fragmenter" les options, juste remplir les options non
204  *      permises dans les fragments avec NOOP.
205  *      Simple et stupide 8), mais la facon la plus efficace.
206  */
207
208 void ip_options_fragment(struct sk_buff * skb)
209 {
210         unsigned char * optptr = skb->nh.raw;
211         struct ip_options * opt = &(IPCB(skb)->opt);
212         int  l = opt->optlen;
213         int  optlen;
214
215         while (l > 0) {
216                 switch (*optptr) {
217                 case IPOPT_END:
218                         return;
219                 case IPOPT_NOOP:
220                         l--;
221                         optptr++;
222                         continue;
223                 }
224                 optlen = optptr[1];
225                 if (optlen<2 || optlen>l)
226                   return;
227                 if (!IPOPT_COPIED(*optptr))
228                         memset(optptr, IPOPT_NOOP, optlen);
229                 l -= optlen;
230                 optptr += optlen;
231         }
232         opt->ts = 0;
233         opt->rr = 0;
234         opt->rr_needaddr = 0;
235         opt->ts_needaddr = 0;
236         opt->ts_needtime = 0;
237         return;
238 }
```

– On met à jour le décalage en ajoutant la longueur du descripteur de tampon moins la longueur de l'en-tête et on place la valeur obtenue dans l'en-tête IP.

        - Si le fragment est suivi d'un autre fragment, on positionne le drapeau `MF` (plus de fragments) dans l'en-tête IP.
        - On place la somme de contrôle dans l'en-tête IP.

      - On passe à la fonction `ip_finish_output()` passée en argument et étudiée dans le chapitre 33. Si celle-ci renvoie une erreur ou s'il n'y a plus de fragment, on sort de la boucle (pseudo-)infinie.
      - Sinon on passe au fragment suivant : le descripteur de tampon prend la valeur du fragment, le tampon prend la valeur du segment suivant, on spécifie que le descripteur de tampon n'a pas de suivant.

    - Si le code de retour est nul, on incrémente les informations statistiques spécifiant que la fragmentation s'est bien passée et on renvoie 0.
    - Sinon on passe les fragments, on incrémente les informations statistiques spécifiant que la fragmentation a échouée et on renvoie le dernier code d'erreur obtenu.

- On a vu que dans un certain nombre de cas, on ne peut pas effectuer le traitement rapide ci-dessus. Dans ces cas-là :

    - On initialise l'espace des données restant, c'est-à-dire la longueur totale du paquet moins la longueur de l'en-tête IP.
    - On initialise le pointeur au début des données brutes dont on passe l'en-tête IP.
    - Tant qu'on n'a pas parcouru toute la zone des données :

        - On initialise la longueur avec le minimum entre le nombre d'octets restants dans la zone des données et l'unité de transfert.
        - S'il ne s'agit pas du dernier fragment, on aligne le prochain départ sur une frontière de huit octets.
        - On essaie d'allouer le deuxième descripteur de tampon. Si on n'y parvient pas, on affiche un message noyau et on renvoie l'opposé du code d'erreur `ENOMEM`.
        - On copie les méta-données du descripteur de tampon passé en argument dans le deuxième descripteur de tampon.
        - On copie le nombre adéquat d'octets des données du descripteur de tampon passé en argument dans celles du deuxième descripteur de tampon.
        - On renseigne les champs de l'en-tête du nouveau fragment.
        - On incrémente les informations statistiques sur le nombre de fragments IP créés.
        - On passe à la fonction `ip_finish_output()` passée en argument et étudiée dans le chapitre 33.
        - Si celle-ci renvoie une erreur, on libère le descripteur de tampon passé en argument, on incrémente les informations statistiques sur les échecs de fragmentation et on renvoie le code d'erreur.
        - Sinon on libère le descripteur de tampon passé en argument, on incrémente les informations statistiques sur les fragmentations réussies et on renvoie le code de retour.

## 36.2 Réassemblage

Le réassemblage, c'est-à-dire la collecte et l'assemblage de paquets IP fragmentés, n'a lieu que dans le système d'extrémité. Pour ce faire, comme nous l'avons vu au chapitre 23, tous les paquets IP fragmentés sont transmis à la fonction de réassemblage `ip_defrag()` dans la fonction `ip_local_deliver()` en passant un descripteur de tampon de socket en argument.

### 36.2.1 Cache de fragment

Les paquets IP fragmentés sont gérés dans un **cache de fragment** jusqu'à ce que tous les fragments d'un paquet soient arrivés et que le paquet puisse être distribué localement ou jusqu'à ce que le temps d'attente maximal des fragments d'un paquet se soit écoulé et que le paquet soit rejeté.

Code Linux 2.6.10

Le temps maximal d'attente (environ 30 secondes) est représenté par la constante `IP_FRAG_TIME` définie dans le fichier `linux/include/net/ip.h`:

```
75 #define IP_FRAG_TIME    (30 * HZ)               /* Duree de vie d'un fragment */
```

Le cache de fragment est constitué d'une table de hachage comportant des entités du type `struct ipq`, comme le montre la figure 36.1 ([WPRMB–02], p. 281). Chacune de ces entités représente un paquet IP fragmenté.

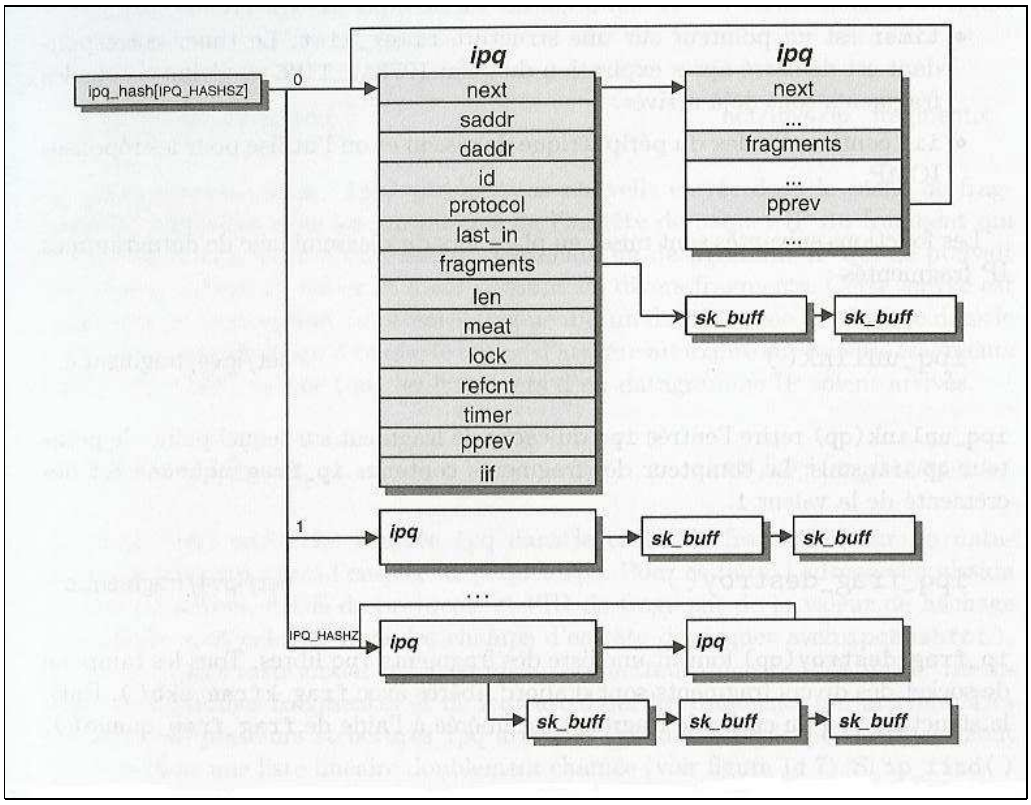


FIG. 36.1 – *Cache de fragment*

Le type `struct ipq` est défini dans le fichier `linux/net/ipv4/ip_fragment.c`: Code Linux 2.6.10

```
72 /* Decrit une entree dans la file d'attente des "datagrammes incomplets". */
73 struct ipq {
74         struct ipq      *next;          /* pointeurs pour la liste chainee              */
75         struct list_head lru_list;      /* membre de liste lru                          */
76         u32             saddr;
77         u32             daddr;
78         u16             id;
79         u8              protocol;
80         u8              last_in;
81 #define COMPLETE                4
82 #define FIRST_IN                2
83 #define LAST_IN                 1
84
85         struct sk_buff  *fragments;     /* liste chainee des fragments recus            */
86         int             len;            /* longueur totale du datagramme originel       */
87         int             meat;
88         spinlock_t      lock;
89         atomic_t        refcnt;
90         struct timer_list timer;        /* quand cette file d'attente expirera-t-elle ? */
91         struct ipq      **pprev;
92         int             iif;
93         struct timeval  stamp;
94 };
```

dont la signification de certains des champs est la suivante :

- `next` et `pprev` servent à la liaison des structures `ipq` dans une ligne de hachage. Il s'agit donc d'une liste doublement chaînée et d'une résolution de collision linéaire dans la table de hachage.
- Les membres `saddr`, `daddr`, `id` et `protocol` forment la clé destinée à la fonction de hachage et à l'attribution de fragments entrants à leurs paquets IP.
- `last_in` mémorise un drapeau indiquant si le premier fragment, le dernier fragment ou tous les fragments sont arrivés.
- `fragments` est une liste de descripteurs de tampon de socket dans laquelle tous les fragments arrivés sont stockés en fonction de la clé vue ci-dessus, dans l'ordre où ils seront ultérieurement reconstitués en tant que paquet complet.
- `len` contient la longueur totale du paquet IP originel.
- `meat` indique combien d'octets sont déjà placés dans le cache de fragment. Lorsque `meat` atteint la valeur `len`, tous les fragments du paquet sont arrivés. Il peut alors être reconstitué.
- `lock` sert pour se protéger d'opérations parallèles sur l'instance de structure de données `ipq`.
- `timer` est un pointeur sur une liste de minuteurs. Le minuteur correspondant est démarré lors de l'arrivée du premier fragment. Après expiration du délai `IP_FRAG_TIME`, on vérifie que tous les fragments sont arrivés.
- `iif` contient l'index du périphérique réseau. On l'utilise pour les réponses ICMP.

La table de hachage, appelée `ipq_hash[]` est définie dans le même fichier : Code Linux 2.6.10

```
96  /* Table de hachage. */
97
98  #define IPQ_HASHSZ      64
99
100 /* Un verrou par seau est facile a ajouter maintenant. */
101 static struct ipq *ipq_hash[IPQ_HASHSZ];
```

### 36.2.2 Traitement du réassemblage

#### 36.2.2.1 Vue d'ensemble

Code Linux 2.6.10

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

```
644 /* Traite un fragment de datagramme IP arrivant. */
645 struct sk_buff *ip_defrag(struct sk_buff *skb)
646 {
647         struct iphdr *iph = skb->nh.iph;
648         struct ipq *qp;
649         struct net_device *dev;
650
651         IP_INC_STATS_BH(IPSTATS_MIB_REASMREQDS);
652
653         /* Commencer par nettoyer la memoire. */
654         if (atomic_read(&ip_frag_mem) > sysctl_ipfrag_high_thresh)
655                 ip_evictor();
656
657         dev = skb->dev;
658
659         /* Consultation (ou creation) d'en-tete de file d'attente */
660         if ((qp = ip_find(iph)) != NULL) {
661                 struct sk_buff *ret = NULL;
662
663                 spin_lock(&qp->lock);
664
665                 ip_frag_queue(qp, skb);
666
667                 if (qp->last_in == (FIRST_IN|LAST_IN) &&
668                     qp->meat == qp->len)
669                         ret = ip_frag_reasm(qp, dev);
670
671                 spin_unlock(&qp->lock);
672                 ipq_put(qp, NULL);
673                 return ret;
674         }
675
676         IP_INC_STATS_BH(IPSTATS_MIB_REASMFAILS);
677         kfree_skb(skb);
678         return NULL;
679 }
```

Autrement dit :

- On déclare un en-tête IP, que l'on instantie avec celui associé au descripteur de tampon passé en argument.
- On déclare une entrée de cache de fragment et un descripteur de périphérique réseau.
- On incrémente les informations statistiques sur le nombre de requêtes de réassemblage.
- On commence par faire de la place dans la partie de la mémoire vive réservée au réassemblage, grâce à la fonction `ip_evictor()` étudiée ci-après, si la fragmentation des paquets utilise trop de mémoire.

Code Linux 2.6.10

La variable `ip_frag_mem` est définie dans le fichier `linux/net/ipv4/ip_fragment.c`:

```
169 atomic_t ip_frag_mem = ATOMIC_INIT(0);  /* Memoire utilisee pour les fragments */
```

Code Linux 2.6.10

La variable `sysctl_ipfrag_high_thresh` est définie dans le même fichier :

```
46 /* NOTE. La logique de la defragmentation IP est maintenant parallele au code IPv6
47  * correspondant. Si vous changez quelque chose ici, mettez egalement a jour _S'IL VOUS
48  * PLAIT_ ipv6/reassembly.c. Ou notifiez-le moi, au minimum. --ANK
```

```
49  */
50
51 /* Limites du cache de fragment. Nous ferons 256Ko en une fois. Si nous devions
52  * depasser cette limite, nous l'elaguerions a 192Ko. Ceci nous permettrait de nous en
53  * tirer meme avec les cas les plus extremes sans permettre a un attaquant de nuire
54  * considerablement aux performances de la machine.
55  */
56 int sysctl_ipfrag_high_thresh = 256*1024;
57 int sysctl_ipfrag_low_thresh  = 192*1024;
```

- On instantie le descripteur de périphérique réseau avec celui associé au descripteur de tampon passé en argument.
- On essaie de rechercher l'entrée de cache de fragment associée aux clés du fragment, en en créant une si besoin est, grâce à la fonction `ip_find()` étudiée ci-dessous. Si on n'y parvient pas, on incrémente les informations statistiques sur le nombre d'échec de réassemblage, on libère le descripteur de tampon passé en argument et on renvoie `NULL`.
- Sinon on déclare un descripteur de tampon de retour, que l'on initialise à `NULL`.
- On verrouille le réassemblage pour cette entrée de cache.
- On place le fragment nouvellement arrivé au bon endroit dans la file d'attente des fragments du paquet, grâce à la fonction `ip_frag_queue()` étudiée ci-après.
- Si le premier ou le dernier fragment est parvenu et si on a obtenu le nombre d'octets du paquet originel, on réassemble les fragments grâce à la fonction `ip_frag_reasm()` étudiée ci-après.
- On déverrouille l'entrée de cache de fragment.
- On libère l'entrée de cache de fragment, grâce à la fonction `ipq_put()` étudiée ci-après.
- On renvoie la valeur `NULL` ou la valeur de retour fournie par la fonction `ip_frag_reasm()`.

#### 36.2.2.2 Première étape : nettoyage de la mémoire

La fonction `ip_evictor()` est appelée par `ip_defrag()` lorsque la fragmentation des paquets utilise trop de mémoire. La valeur de seuil pour la mémoire maximale du cache de fragment est fixée à 256 Ko. Toutes les lignes de hachage du cache de fragment sont vérifiées les unes après les autres et d'éventuelles entrées supprimées jusqu'à ce qu'on atteigne la valeur minimale de 192 Ko.

Elle est définie dans le fichier `linux/net/ipv4/ip_fragment.c` : Code Linux 2.6.10

```
243 /* Limitation de la memoire pour les fragments. Evictor met a la poubelle le plus vieux
244  * de la file d'attente jusqu'a ce que nous tombions en dessous du seuil minimal.
245  */
246 static void __ip_evictor(int threshold)
247 {
248         struct ipq *qp;
249         struct list_head *tmp;
250         int work;
251
252         work = atomic_read(&ip_frag_mem) - threshold;
253         if (work <= 0)
254                 return;
255
256         while (work > 0) {
257                 read_lock(&ipfrag_lock);
258                 if (list_empty(&ipq_lru_list)) {
259                         read_unlock(&ipfrag_lock);
260                         return;
261                 }
262                 tmp = ipq_lru_list.next;
```

```
263                 qp = list_entry(tmp, struct ipq, lru_list);
264                 atomic_inc(&qp->refcnt);
265                 read_unlock(&ipfrag_lock);
266
267                 spin_lock(&qp->lock);
268                 if (!(qp->last_in&COMPLETE))
269                         ipq_kill(qp);
270                 spin_unlock(&qp->lock);
271
272                 ipq_put(qp, &work);
273                 IP_INC_STATS_BH(IPSTATS_MIB_REASMFAILS);
274         }
275 }
276
277 static inline void ip_evictor(void)
278 {
279         __ip_evictor(sysctl_ipfrag_low_thresh);
280 }
```

#### 36.2.2.3 Deuxième étape : consultation du cache de fragment

La fonction `ip_find()` cherche une l'entrée de cache correspondant aux quatre clés (identificateur, adresse source, adresse de destination et protocole) contenues dans l'en-tête IP passé en argument. La valeur de hachage de l'entrée est calculée avec `ipqhashfn()`. Les collisions sont résolues dans une liste doublement chaînée. Si on ne trouve pas d'entrée convenant au fragment, une nouvelle entrée est créée dans le cache de fragment grâce à la fonction `ip_frag_create()` et c'est celle-ci qui est renvoyée.

La fonction de recherche `ip_find()` est définie dans le fichier `linux/net/ipv4/ip_fragment.c` :

Code Linux 2.6.10

```
386 /* Cherche l'entree correcte dans la file d'attente des "datagrammes incomplets" pour
387  * ce datagramme IP. On en cree une nouvelle si rien n'est trouve.
388  */
389 static inline struct ipq *ip_find(struct iphdr *iph)
390 {
391         __u16 id = iph->id;
392         __u32 saddr = iph->saddr;
393         __u32 daddr = iph->daddr;
394         __u8 protocol = iph->protocol;
395         unsigned int hash = ipqhashfn(id, saddr, daddr, protocol);
396         struct ipq *qp;
397
398         read_lock(&ipfrag_lock);
399         for(qp = ipq_hash[hash]; qp; qp = qp->next) {
400                 if(qp->id == id         &&
401                    qp->saddr == saddr   &&
402                    qp->daddr == daddr   &&
403                    qp->protocol == protocol) {
404                         atomic_inc(&qp->refcnt);
405                         read_unlock(&ipfrag_lock);
406                         return qp;
407                 }
408         }
409         read_unlock(&ipfrag_lock);
410
411         return ip_frag_create(hash, iph);
412 }
```

La fonction de hachage `ipqhashfn()` est définie dans le même fichier :

Code Linux 2.6.10

```
103 static u32 ipfrag_hash_rnd;
```

```
[...]
123 static unsigned int ipqhashfn(u16 id, u32 saddr, u32 daddr, u8 prot)
124 {
125         return jhash_3words((u32)id << 16 | prot, saddr, daddr,
126                             ipfrag_hash_rnd) & (IPQ_HASHSZ - 1);
127 }
```

La fonction de création d'une nouvelle entrée `ip_frag_create()` est également définie dans le même fichier :

Code Linux 2.6.10

```
354 /* Ajoute une entree a la file d'attente 'ipq' pour un datagramme IP nouvellement recu. */
355 static struct ipq *ip_frag_create(unsigned hash, struct iphdr *iph)
356 {
357         struct ipq *qp;
358
359         if ((qp = frag_alloc_queue()) == NULL)
360                 goto out_nomem;
361
362         qp->protocol = iph->protocol;
363         qp->last_in = 0;
364         qp->id = iph->id;
365         qp->saddr = iph->saddr;
366         qp->daddr = iph->daddr;
367         qp->len = 0;
368         qp->meat = 0;
369         qp->fragments = NULL;
370         qp->iif = 0;
371
372         /* Initialise un minuteur pour cette entree. */
373         init_timer(&qp->timer);
374         qp->timer.data = (unsigned long) qp;    /* pointeur sur la file d'attente */
375         qp->timer.function = ip_expire;         /* routine d'expiration           */
376         spin_lock_init(&qp->lock);
377         atomic_set(&qp->refcnt, 1);
378
379         return ip_frag_intern(hash, qp);
380
381 out_nomem:
382         NETDEBUG(if (net_ratelimit()) printk(KERN_ERR "ip_frag_create: no memory left !\n"));
383         return NULL;
384 }
```

La fonction en ligne `frag_alloc_queue()` d'allocation d'une nouvelle entrée de file d'attente est également définie dans le même fichier :

Code Linux 2.6.10

```
188 static __inline__ struct ipq *frag_alloc_queue(void)
189 {
190         struct ipq *qp = kmalloc(sizeof(struct ipq), GFP_ATOMIC);
191
192         if(!qp)
193                 return NULL;
194         atomic_add(sizeof(struct ipq), &ip_frag_mem);
195         return qp;
196 }
```

Le traitement lors de l'expiration du délai, contenu dans la fonction `ip_expire()`, sera étudié plus bas.

La fonction `ip_frag_intern()` d'initialisation d'une nouvelle entrée de file d'attente est également définie dans le même fichier :

Code Linux 2.6.10

```
312 /* Primitives de creation. */
313
```

```
314 static struct ipq *ip_frag_intern(unsigned int hash, struct ipq *qp_in)
315 {
316         struct ipq *qp;
317
318         write_lock(&ipfrag_lock);
319 #ifdef CONFIG_SMP
320         /* Avec la course SMP nous avons a verifier a nouveau la table de hachage, puisque
321          * une telle entree aurait pu etre creee sur un autre cpu, bien que le
322          * verrou d'ecriture etait positionne en verrou d'ecriture.
323          */
324         for(qp = ipq_hash[hash]; qp; qp = qp->next) {
325                 if(qp->id == qp_in->id          &&
326                    qp->saddr == qp_in->saddr    &&
327                    qp->daddr == qp_in->daddr    &&
328                    qp->protocol == qp_in->protocol) {
329                         atomic_inc(&qp->refcnt);
330                         write_unlock(&ipfrag_lock);
331                         qp_in->last_in |= COMPLETE;
332                         ipq_put(qp_in, NULL);
333                         return qp;
334                 }
335         }
336 #endif
337         qp = qp_in;
338
339         if (!mod_timer(&qp->timer, jiffies + sysctl_ipfrag_time))
340                 atomic_inc(&qp->refcnt);
341
342         atomic_inc(&qp->refcnt);
343         if((qp->next = ipq_hash[hash]) != NULL)
344                 qp->next->pprev = &qp->next;
345         ipq_hash[hash] = qp;
346         qp->pprev = &ipq_hash[hash];
347         INIT_LIST_HEAD(&qp->lru_list);
348         list_add_tail(&qp->lru_list, &ipq_lru_list);
349         ip_frag_nqueues++;
350         write_unlock(&ipfrag_lock);
351         return qp;
352 }
```

#### 36.2.2.4 Troisième étape : insertion d'un fragment

La fonction `ip_frag_queue()` insère un fragment nouvellement arrivé dans la file d'attente des fragments d'un datagramme IP, représenté à l'aide d'un descripteur de fragment. On vérifie d'abord si le paquet n'est pas déjà complet et si le fragment ne serait pas un simple doublon. Si tel n'est pas le cas, les positions (`offset` et `end`) du fragment dans le paquet d'origine sont calculées grâce à l'en-tête IP. On vérifie ensuite, au moyen du drapeau `MF` s'il s'agit du dernier fragment du paquet et dans ce cas on positionne `LAST_IN`. Puis on recherche la position correcte du fragment dans la liste des fragments reçus jusqu'ici et le tampon de socket est placé à cette position. Le paramètre `meat` du descripteur de fragment du paquet est augmenté de la longueur du fragment qui vient d'être inséré.

Code Linux 2.6.10

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

```
414 /* Ajoute un nouveau segment a une file d'attente existante. */
415 static void ip_frag_queue(struct ipq *qp, struct sk_buff *skb)
416 {
417         struct sk_buff *prev, *next;
418         int flags, offset;
419         int ihl, end;
```

```
420
421         if (qp->last_in & COMPLETE)
422                 goto err;
423
424         offset = ntohs(skb->nh.iph->frag_off);
425         flags = offset & ~IP_OFFSET;
426         offset &= IP_OFFSET;
427         offset <<= 3;           /* le deplacement est dans l'emplacement 8-octet */
428         ihl = skb->nh.iph->ihl * 4;
429
430         /* Determine la position ce ce fragment. */
431         end = offset + skb->len - ihl;
432
433         /* Est-ce le dernier fragment ? */
434         if ((flags & IP_MF) == 0) {
435                 /* Si nous avons deja quelques bits au-dela de la fin
436                  * ou si nous avons une fin differente, le segment est corrompu.
437                  */
438                 if (end < qp->len ||
439                     ((qp->last_in & LAST_IN) && end != qp->len))
440                         goto err;
441                 qp->last_in |= LAST_IN;
442                 qp->len = end;
443         } else {
444                 if (end&7) {
445                         end &= ~7;
446                         if (skb->ip_summed != CHECKSUM_UNNECESSARY)
447                                 skb->ip_summed = CHECKSUM_NONE;
448                 }
449                 if (end > qp->len) {
450                         /* Quelques bits au-dela de la fin -> corrompu. */
451                         if (qp->last_in & LAST_IN)
452                                 goto err;
453                         qp->len = end;
454                 }
455         }
456         if (end == offset)
457                 goto err;
458
459         if (pskb_pull(skb, ihl) == NULL)
460                 goto err;
461         if (pskb_trim(skb, end-offset))
462                 goto err;
463
464         /* Trouver quels fragments sont avant nous et lesquels sont apres nous
465          * dans la chaine des fragments. Nous devons savoir ou inserer
466          * ce fragment, n'est-ce pas ?
467          */
468         prev = NULL;
469         for(next = qp->fragments; next != NULL; next = next->next) {
470                 if (FRAG_CB(next)->offset >= offset)
471                         break;  /* bingo! */
472                 prev = next;
473         }
474
475         /* Nous avons trouve ou l'inserer. Verifions le chevauchement avec
476          * le fragment precedent, et, si necessaire, alignons les choses pour que
477          * tout chevauchement soit elimine.
478          */
479         if (prev) {
480                 int i = (FRAG_CB(prev)->offset + prev->len) - offset;
481
```

```
482                 if (i > 0) {
483                         offset += i;
484                         if (end <= offset)
485                                 goto err;
486                         if (!pskb_pull(skb, i))
487                                 goto err;
488                         if (skb->ip_summed != CHECKSUM_UNNECESSARY)
489                                 skb->ip_summed = CHECKSUM_NONE;
490                 }
491         }
492 
493         while (next && FRAG_CB(next)->offset < end) {
494                 int i = end - FRAG_CB(next)->offset; /* chevauchement de 'i' octets */
495 
496                 if (i < next->len) {
497                         /* Mangeons le debut du fragment suivant chevauche et conservons
498                          * le reste. Les suivants ne peuvent pas etre chevauches.
499                          */
500                         if (!pskb_pull(next, i))
501                                 goto err;
502                         FRAG_CB(next)->offset += i;
503                         qp->meat -= i;
504                         if (next->ip_summed != CHECKSUM_UNNECESSARY)
505                                 next->ip_summed = CHECKSUM_NONE;
506                         break;
507                 } else {
508                         struct sk_buff *free_it = next;
509 
510                         /* Le vieux fragment est completement chevauche par
511                          * le nouveau. Ecartons-le.
512                          */
513                         next = next->next;
514 
515                         if (prev)
516                                 prev->next = next;
517                         else
518                                 qp->fragments = next;
519 
520                         qp->meat -= free_it->len;
521                         frag_kfree_skb(free_it, NULL);
522                 }
523         }
524 
525         FRAG_CB(skb)->offset = offset;
526 
527         /* Insere ce fragment dans la chaine des fragments. */
528         skb->next = next;
529         if (prev)
530                 prev->next = skb;
531         else
532                 qp->fragments = skb;
533 
534         if (skb->dev)
535                 qp->iif = skb->dev->ifindex;
536         skb->dev = NULL;
537         qp->stamp = skb->stamp;
538         qp->meat += skb->len;
539         atomic_add(skb->truesize, &ip_frag_mem);
540         if (offset == 0)
541                 qp->last_in |= FIRST_IN;
542 
543         write_lock(&ipfrag_lock);
```

```
544         list_move_tail(&qp->lru_list, &ipq_lru_list);
545         write_unlock(&ipfrag_lock);
546
547         return;
548
549 err:
550         kfree_skb(skb);
551 }
```

La macro `FRAG_CB()` est définie dans le même fichier : Code Linux 2.6.10

```
64 struct ipfrag_skb_cb
65 {
66         struct inet_skb_parm    h;
67         int                     offset;
68 };
69
70 #define FRAG_CB(skb)    ((struct ipfrag_skb_cb*)((skb)->cb))
```

#### 36.2.2.5 Quatrième étape : réassemblage des fragments

La fonction `ip_frag_reasm()` permet de reconstituer le paquet lorsque tous les fragments sont arrivés. Pour ce faire, un nouveau descripteur de tampon de socket est d'abord créé avec une zone de données longue de `qp->len` octets et initialisée avec l'en-tête du paquet IP. Puis les données IP utiles des divers fragments sont copiées dans la zone des données du descripteur de tampon qui vient d'être créé.

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

```
554 /* Construit un nouveau datagramme IP a partir de tous ses fragments. */
555
556 static struct sk_buff *ip_frag_reasm(struct ipq *qp, struct net_device *dev)
557 {
558         struct iphdr *iph;
559         struct sk_buff *fp, *head = qp->fragments;
560         int len;
561         int ihlen;
562
563         ipq_kill(qp);
564
565         BUG_TRAP(head != NULL);
566         BUG_TRAP(FRAG_CB(head)->offset == 0);
567
568         /* Alloue un nouveau tampon pour le datagramme. */
569         ihlen = head->nh.iph->ihl*4;
570         len = ihlen + qp->len;
571
572         if(len > 65535)
573                 goto out_oversize;
574
575         /* La tete de la liste ne doit pas etre clonee. */
576         if (skb_cloned(head) && pskb_expand_head(head, 0, 0, GFP_ATOMIC))
577                 goto out_nomem;
578
579         /* Si le premier fragment est lui-meme fragmente, nous le divisons
580          * en deux morceaux : le premier avec les donnees et la partie pagee
581          * et le second contenant seulement des fragments. */
582         if (skb_shinfo(head)->frag_list) {
583                 struct sk_buff *clone;
584                 int i, plen = 0;
585
586                 if ((clone = alloc_skb(0, GFP_ATOMIC)) == NULL)
```

```
587                     goto out_nomem;
588             clone->next = head->next;
589             head->next = clone;
590             skb_shinfo(clone)->frag_list = skb_shinfo(head)->frag_list;
591             skb_shinfo(head)->frag_list = NULL;
592             for (i=0; i<skb_shinfo(head)->nr_frags; i++)
593                     plen += skb_shinfo(head)->frags[i].size;
594             clone->len = clone->data_len = head->data_len - plen;
595             head->data_len -= clone->len;
596             head->len -= clone->len;
597             clone->csum = 0;
598             clone->ip_summed = head->ip_summed;
599             atomic_add(clone->truesize, &ip_frag_mem);
600     }
601
602     skb_shinfo(head)->frag_list = head->next;
603     skb_push(head, head->data - head->nh.raw);
604     atomic_sub(head->truesize, &ip_frag_mem);
605
606     for (fp=head->next; fp; fp = fp->next) {
607             head->data_len += fp->len;
608             head->len += fp->len;
609             if (head->ip_summed != fp->ip_summed)
610                     head->ip_summed = CHECKSUM_NONE;
611             else if (head->ip_summed == CHECKSUM_HW)
612                     head->csum = csum_add(head->csum, fp->csum);
613             head->truesize += fp->truesize;
614             atomic_sub(fp->truesize, &ip_frag_mem);
615     }
616
617     head->next = NULL;
618     head->dev = dev;
619     head->stamp = qp->stamp;
620
621     iph = head->nh.iph;
622     iph->frag_off = 0;
623     iph->tot_len = htons(len);
624     IP_INC_STATS_BH(IPSTATS_MIB_REASMOKS);
625     qp->fragments = NULL;
626     return head;
627
628 out_nomem:
629     NETDEBUG(if (net_ratelimit())
630              printk(KERN_ERR
631                     "IP: queue_glue: no memory for gluing queue %p\n",
632                     qp));
633     goto out_fail;
634 out_oversize:
635     if (net_ratelimit())
636             printk(KERN_INFO
637                     "Oversized IP packet from %d.%d.%d.%d.\n",
638                     NIPQUAD(qp->saddr));
639 out_fail:
640     IP_INC_STATS_BH(IPSTATS_MIB_REASMFAILS);
641     return NULL;
642 }
```

#### 36.2.2.6 Cinquième étape : libération de l'entrée du cache de fragment

Code Linux 2.6.10

La fonction en ligne `ipq_put()` est définie dans le fichier `linux/net/ipv4/ip_fragment.c` :

```
222 static __inline__ void ipq_put(struct ipq *ipq, int *work)
```

```
223 {
224         if (atomic_dec_and_test(&ipq->refcnt))
225                 ip_frag_destroy(ipq, work);
226 }
```

qui renvoie à la fonction `ip_frag_destroy()`, définie dans le même fichier : Code Linux 2.6.10

```
199 /* Primitives de destruction. */
200
201 /* Destruction complete de ipq. */
202 static void ip_frag_destroy(struct ipq *qp, int *work)
203 {
204         struct sk_buff *fp;
205
206         BUG_TRAP(qp->last_in&COMPLETE);
207         BUG_TRAP(del_timer(&qp->timer) == 0);
208
209         /* Libere toutes les donnees du fragment. */
210         fp = qp->fragments;
211         while (fp) {
212                 struct sk_buff *xp = fp->next;
213
214                 frag_kfree_skb(fp, work);
215                 fp = xp;
216         }
217
218         /* Enfin libere le descripteur de file d'attente lui-meme. */
219         frag_free_queue(qp, work);
220 }
```

La fonction en ligne `frag_kfree_skb()` est définie dans le même fichier : Code Linux 2.6.10

```
171 /* Fonctions de tracage de la memoire. */
172 static __inline__ void frag_kfree_skb(struct sk_buff *skb, int *work)
173 {
174         if (work)
175                 *work -= skb->truesize;
176         atomic_sub(skb->truesize, &ip_frag_mem);
177         kfree_skb(skb);
178 }
```

ainsi que la fonction en ligne `frag_free_queue()` : Code Linux 2.6.10

```
180 static __inline__ void frag_free_queue(struct ipq *qp, int *work)
181 {
182         if (work)
183                 *work -= sizeof(struct ipq);
184         atomic_sub(sizeof(struct ipq), &ip_frag_mem);
185         kfree(qp);
186 }
```

### 36.2.3 Traitement lors de l'expiration du délai

La fonction `ip_expire()` est la routine de traitement à l'expiration du délai du minuteur démarré au début du réassemblage des fragments d'un paquet IP. Si tous les fragments ne sont pas arrivés lors de l'expiration de ce minuteur, l'entrée est supprimée dans le cache de fragment. Si tous les fragments ont été reçus, rien n'est entrepris dans cette fonction. Lorsque tous les fragments ne sont pas encore reçus mais que le premier au moins est disponible, un message d'erreur ICMP du type `ICMP_TIME_EXCEEDED/ICMP_EXC_FRAGTIME` est expédié.

Code Linux 2.6.10

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

```
282 /*
283  * Oups, une file d'attente de fragments a depasse son temps. Tuons-la et envoyons une
       reponse ICMP.
284  */
285 static void ip_expire(unsigned long arg)
286 {
287         struct ipq *qp = (struct ipq *) arg;
288
289         spin_lock(&qp->lock);
290
291         if (qp->last_in & COMPLETE)
292                 goto out;
293
294         ipq_kill(qp);
295
296         IP_INC_STATS_BH(IPSTATS_MIB_REASMTIMEOUT);
297         IP_INC_STATS_BH(IPSTATS_MIB_REASMFAILS);
298
299         if ((qp->last_in&FIRST_IN) && qp->fragments != NULL) {
300                 struct sk_buff *head = qp->fragments;
301                 /* Envoie un message ICMP "Depassement de duree de reassemblage de
                       fragments". */
302                 if ((head->dev = dev_get_by_index(qp->iif)) != NULL) {
303                         icmp_send(head, ICMP_TIME_EXCEEDED, ICMP_EXC_FRAGTIME, 0);
304                         dev_put(head->dev);
305                 }
306         }
307 out:
308         spin_unlock(&qp->lock);
309         ipq_put(qp, NULL);
310 }
```

Code Linux 2.6.10

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

```
228 /* Tue une entree ipq. Elle n'est pas detruite immediatement,
229  * puisque l'appelant (et quelques autres) y fait encore reference.
230  */
231 static void ipq_kill(struct ipq *ipq)
232 {
233         if (del_timer(&ipq->timer))
234                 atomic_dec(&ipq->refcnt);
235
236         if (!(ipq->last_in & COMPLETE)) {
237                 ipq_unlink(ipq);
238                 atomic_dec(&ipq->refcnt);
239                 ipq->last_in |= COMPLETE;
240         }
241 }
```

Code Linux 2.6.10

La fonction `ipq_unlink()` retire l'entrée du cache de fragment sur lequel pointe le descripteur transmis. Le compteur des fragments `ip_frag_nqueues` est décrémenté. Elle est définie dans le fichier `linux/net/ipv4/ip_fragment.c` :

```
105 int ip_frag_nqueues = 0;
106
107 static __inline__ void __ipq_unlink(struct ipq *qp)
108 {
109         if(qp->next)
110                 qp->next->pprev = qp->pprev;
111         *qp->pprev = qp->next;
112         list_del(&qp->lru_list);
```

```
113         ip_frag_nqueues--;
114 }
115
116 static __inline__ void ipq_unlink(struct ipq *ipq)
117 {
118         write_lock(&ipfrag_lock);
119         __ipq_unlink(ipq);
120         write_unlock(&ipfrag_lock);
121 }
```

## 36.3 Initialisation de la fragmentation

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

```
681 void ipfrag_init(void)
682 {
683         ipfrag_hash_rnd = (u32) ((num_physpages ^ (num_physpages>>7)) ^
684                                  (jiffies ^ (jiffies >> 6)));
685
686         init_timer(&ipfrag_secret_timer);
687         ipfrag_secret_timer.function = ipfrag_secret_rebuild;
688         ipfrag_secret_timer.expires = jiffies + sysctl_ipfrag_secret_interval;
689         add_timer(&ipfrag_secret_timer);
690 }
```