Chapitre 32

Envoi de datagrammes ordinaires sous UDP

Nous avons vu au chapitre 31 comment débute l'envoi d'un datagramme par une socket pour la famille de protocoles IPv4 et le type de communication non connecté, c'est-à-dire pour le protocole UDP de couche de transport, ce qui constituait l'interface application/transport. Nous allons étudier dans ce chapitre la suite du cheminement, plus précisément ce qui concerne la couche transport, en illustrant nos propos par UDP.

32.1 Fonction d'envoi spécifique à UDP

32.1.1 Description

Nous avons vu que, pour l'envoi de données par une socket dans le cas de la famille de protocoles IPv4 et d'un type de communication non connecté, il est fait appel à la fonction :

```
int udp_sendmsg(struct kiocb *iocb, struct sock *sk, struct msghdr *msg,
                size_t len)
```

dont les arguments sont :

- un(e adresse de) bloc de contrôle de données d'entrée-sortie, spécifiant où se trouvent les données ;
- un(e adresse de) descripteur de couche transport, spécifiant entre autre les adresses et numéros de port source et destination ;
- un(e adresse d')en-tête de message, spécifiant comment transmettre les données ;
- la taille des données à envoyer.

La fonction renvoie le nombre d'octets effectivement envoyés.

32.1.2 Implémentation

Code Linux 2.6.10

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

```
480 int udp_sendmsg(struct kiocb *iocb, struct sock *sk, struct msghdr *msg,
481                 size_t len)
482 {
483         struct inet_opt *inet = inet_sk(sk);
484         struct udp_opt *up = udp_sk(sk);
485         int ulen = len;
486         struct ipcm_cookie ipc;
487         struct rtable *rt = NULL;
488         int free = 0;
489         int connected = 0;
490         u32 daddr, faddr, saddr;
491         u16 dport;
492         u8  tos;
493         int err;
494         int corkreq = up->corkflag || msg->msg_flags&MSG_MORE;
495
496         if (len > 0xFFFF)
497                 return -EMSGSIZE;
498
499         /*
500          *      Verifier les drapeaux.
501          */
502
503         if (msg->msg_flags&MSG_OOB)     /* Compatibilite avec les messages d'erreur BSD */
504                 return -EOPNOTSUPP;
505
506         ipc.opt = NULL;
507
508         if (up->pending) {
509                 /*
510                  * Il y a des trames en attente.
511                  * Le verrou de la socket doit etre detenu pendant qu'on lui met un bouchon.
512                  */
513                 lock_sock(sk);
```

```
514                 if (likely(up->pending)) {
515                         if (unlikely(up->pending != AF_INET)) {
516                                 release_sock(sk);
517                                 return -EINVAL;
518                         }
519                         goto do_append_data;
520                 }
521                 release_sock(sk);
522         }
523         ulen += sizeof(struct udphdr);
524 
525         /*
526          *      Recupere et verifie l'adresse.
527          */
528         if (msg->msg_name) {
529                 struct sockaddr_in * usin = (struct sockaddr_in*)msg->msg_name;
530                 if (msg->msg_namelen < sizeof(*usin))
531                         return -EINVAL;
532                 if (usin->sin_family != AF_INET) {
533                         if (usin->sin_family != AF_UNSPEC)
534                                 return -EAFNOSUPPORT;
535                 }
536 
537                 daddr = usin->sin_addr.s_addr;
538                 dport = usin->sin_port;
539                 if (dport == 0)
540                         return -EINVAL;
541         } else {
542                 if (sk->sk_state != TCP_ESTABLISHED)
543                         return -EDESTADDRREQ;
544                 daddr = inet->daddr;
545                 dport = inet->dport;
546                 /* Ouvre un chemin rapide pour la socket connectee.
547                    La route ne sera pas utilisee si au moins une option est positionnee.
548                  */
549                 connected = 1;
550         }
551         ipc.addr = inet->saddr;
552 
553         ipc.oif = sk->sk_bound_dev_if;
554         if (msg->msg_controllen) {
555                 err = ip_cmsg_send(msg, &ipc);
556                 if (err)
557                         return err;
558                 if (ipc.opt)
559                         free = 1;
560                 connected = 0;
561         }
562         if (!ipc.opt)
563                 ipc.opt = inet->opt;
564 
565         saddr = ipc.addr;
566         ipc.addr = faddr = daddr;
567 
568         if (ipc.opt && ipc.opt->srr) {
569                 if (!daddr)
570                         return -EINVAL;
571                 faddr = ipc.opt->faddr;
572                 connected = 0;
573         }
574         tos = RT_TOS(inet->tos);
575         if (sk->sk_localroute || (msg->msg_flags & MSG_DONTROUTE) ||
```

```
576             (ipc.opt && ipc.opt->is_strictroute)) {
577                 tos |= RTO_ONLINK;
578                 connected = 0;
579         }
580 
581         if (MULTICAST(daddr)) {
582                 if (!ipc.oif)
583                         ipc.oif = inet->mc_index;
584                 if (!saddr)
585                         saddr = inet->mc_addr;
586                 connected = 0;
587         }
588 
589         if (connected)
590                 rt = (struct rtable*)sk_dst_check(sk, 0);
591 
592         if (rt == NULL) {
593                 struct flowi fl = { .oif = ipc.oif,
594                                     .nl_u = { .ip4_u =
595                                               { .daddr = faddr,
596                                                 .saddr = saddr,
597                                                 .tos = tos } },
598                                     .proto = IPPROTO_UDP,
599                                     .uli_u = { .ports =
600                                                { .sport = inet->sport,
601                                                  .dport = dport } } };
602                 err = ip_route_output_flow(&rt, &fl, sk, !(msg->msg_flags&MSG_DONTWAIT));
603                 if (err)
604                         goto out;
605 
606                 err = -EACCES;
607                 if ((rt->rt_flags & RTCF_BROADCAST) &&
608                     !sock_flag(sk, SOCK_BROADCAST))
609                         goto out;
610                 if (connected)
611                         sk_dst_set(sk, dst_clone(&rt->u.dst));
612         }
613 
614         if (msg->msg_flags&MSG_CONFIRM)
615                 goto do_confirm;
616 back_from_confirm:
617 
618         saddr = rt->rt_src;
619         if (!ipc.addr)
620                 daddr = ipc.addr = rt->rt_dst;
621 
622         lock_sock(sk);
623         if (unlikely(up->pending)) {
624                 /* On a deja mis le bouchon a la socket lorsqu'elle a ete preparee. */
625                 /* ... Ce qui est un bogue d'application evident. --ANK */
626                 release_sock(sk);
627 
628                 NETDEBUG(if (net_ratelimit()) printk(KERN_DEBUG "udp cork app bug 2\n"));
629                 err = -EINVAL;
630                 goto out;
631         }
632         /*
633          *      On met maintenant le bouchon a la socket pour les donnees en souffrance.
634          */
635         inet->cork.fl.fl4_dst = daddr;
636         inet->cork.fl.fl_ip_dport = dport;
637         inet->cork.fl.fl4_src = saddr;
```

```
638         inet->cork.fl.fl_ip_sport = inet->sport;
639         up->pending = AF_INET;
640
641 do_append_data:
642         up->len += ulen;
643         err = ip_append_data(sk, ip_generic_getfrag, msg->msg_iov, ulen,
644                         sizeof(struct udphdr), &ipc, rt,
645                         corkreq ? msg->msg_flags|MSG_MORE : msg->msg_flags);
646         if (err)
647                 udp_flush_pending_frames(sk);
648         else if (!corkreq)
649                 err = udp_push_pending_frames(sk, up);
650         release_sock(sk);
651
652 out:
653         ip_rt_put(rt);
654         if (free)
655                 kfree(ipc.opt);
656         if (!err) {
657                 UDP_INC_STATS_USER(UDP_MIB_OUTDATAGRAMS);
658                 return len;
659         }
660         return err;
661
662 do_confirm:
663         dst_confirm(&rt->u.dst);
664         if (!(msg->msg_flags&MSG_PROBE) || len)
665                 goto back_from_confirm;
666         err = 0;
667         goto out;
668 }
```

Autrement dit :

- On déclare une option Internet, que l'on instantie avec celle associée au descripteur de couche transport passé en argument.
- On déclare une adresse d'option UDP, que l'on initialise avec celle du champ adéquat du descripteur de couche transport passé en argument.
- On déclare une longueur de datagramme UDP, que l'on commence par initialiser pour l'instant avec la taille des données passée en argument.
- On déclare un cafteur IPCM, qui ne va pas nous intéresser ici.

  Le type `ipcm_cookie` est défini dans le fichier `linux/include/linux/udp.h` :

Code Linux 2.6.10

```
50 struct ipcm_cookie
51 {
52         u32                     addr;
53         int                     oif;
54         struct ip_options       *opt;
55 };
```

- On déclare une adresse d'entrée du cache de routage, que l'on initialise à `NULL`.
- On déclare un test de bouchon, que l'on initialise suivant les valeurs de l'option UDP et de l'en-tête de message.
- Si la taille des données passée en argument est supérieure à 65 535 octets, celles-ci ne peuvent pas tenir sur un datagramme UDP. On renvoie donc l'opposé du code d'erreur `EMSGSIZE`.
- Puisque l'envoi des données urgentes (`MSG_OOB`) n'est pas prévu pour UDP, dans le cas d'une telle tentative, on renvoie l'opposé du code d'erreur `EOPNOTSUPP`.

- Si l'option UDP indique qu'il reste des données à envoyer en souffrance, on verrouille le descripteur de couche transport. Si elles ne correspondent pas à la famille de protocoles `AF_INET`, il y a une erreur ; on libère alors le descripteur de couche transport et on renvoie l'opposé du code d'erreur `EINVAL`. Sinon on essaie d'ajouter les données aux données nouvelles à envoyer de la façon que nous verrons ci-dessous.
- S'il n'y a pas de données à envoyer en souffrance, on ajoute à la longueur la taille d'un en-tête UDP pour obtenir la longueur du datagramme UDP.
- On récupère et on vérifie l'adresse de destination passée en argument dans l'en-tête de message :
  - Si l'adresse de l'adresse (*sic*) de l'en-tête de message est nulle, il s'agit d'une convention TCP pour ne pas redonner l'adresse à chaque fois dans le cas connecté. Si l'état indiqué dans le descripteur de couche transport est différent de `TCP_ESTABLISHED`, il y a un problème ; on renvoie l'opposé du code d'erreur `EDSTADDRREQ`. Sinon on récupère l'adresse et le numéro de port de destination à partir de l'option Internet et on indique qu'on est déjà connecté.

    Ce cas ne nous intéresse pas ici puisque nous étudions le cas non connecté.
  - Si la longueur d'adresse indiquée dans l'en-tête de message est inférieure à celle d'une adresse Internet ou si la famille de protocoles spécifiée dans l'adresse est `AF_UNSPEC`, il y a également un problème ; on renvoie donc l'opposé du code d'erreur `EINVAL`.
  - Sinon on récupère l'adresse et le numéro de port de destination à partir de cette adresse.
  - Si le numéro de port est nul, on renvoie l'opposé du code d'erreur `EINVAL`.

    Ce cas ne devrait pas se produire puisque nous avons vu au chapitre 31 que nous avons attribué un numéro de port automatiquement.
- Les champs adresse et interface physique de sortie du cafteur sont renseignés avec l'adresse source et l'interface fournie par le descripteur de couche transport.
- Si l'en-tête de message fait apparaître des données ancillaires, on fait appel à la fonction `ip_cmsg_send()`, qui ne nous intéressera pas ici. Si une erreur se produit durant cette phase, on renvoie le code fourni par cette fonction ; sinon on indique qu'on n'est pas connecté.
- Si le champ option Internet du cafteur IPCM n'est pas renseigné, on lui affecte la valeur de l'option déclarée au début.
- On sauvegarde l'adresse source dans la variable `saddr` et on renseigne le champ adresse du cafteur avec l'adresse de destination.
- Si l'option du cafteur n'est pas affectée ou si la source n'est pas renseignée : on renvoie l'opposé du code d'erreur `EINVAL` si l'adresse de destination n'est pas déterminée ; sinon on sauvegarde l'adresse du premier routeur auquel il faut envoyer le datagramme et on indique qu'on n'est pas connecté.
- Le type de service (*tos*) est obtenu à partir de l'option IP.

  Code Linux 2.6.10

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

```
30 #define RT_TOS(tos)     ((tos)&IPTOS_TOS_MASK)
```

- Si le descripteur de couche transport, le vecteur des drapeaux de l'en-tête de message ou l'option du cafteur indiquent que l'on ne doit pas s'éloigner de la route spécifiée, on modifie le type de service en conséquence et on indique qu'on n'est pas connecté.
- Dans le cas d'une multidiffusion, on agit en conséquence, ce qui ne nous intéressera pas pour l'instant.

– Si l'on est connecté, on essaie d'initialiser l'entrée de cache de routage.

Ce cas ne nous intéresse pas puisque nous n'étudions pour l'instant que le cas non connecté.

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

```
999  static inline struct dst_entry *
1000 sk_dst_check(struct sock *sk, u32 cookie)
1001 {
1002         struct dst_entry *dst = sk_dst_get(sk);
1003
1004         if (dst && dst->obsolete && dst->ops->check(dst, cookie) == NULL) {
1005                 sk_dst_reset(sk);
1006                 return NULL;
1007         }
1008
1009         return dst;
1010 }
```

La fonction `sk_dst_get()` est définie un peu plus haut dans le même fichier : Code Linux 2.6.10

```
937 static inline struct dst_entry *
938 sk_dst_get(struct sock *sk)
939 {
940         struct dst_entry *dst;
941
942         read_lock(&sk->sk_dst_lock);
943         dst = sk->sk_dst_cache;
944         if (dst)
945                 dst_hold(dst);
946         read_unlock(&sk->sk_dst_lock);
947         return dst;
948 }
```

– Si, comme dans le cas UDP, on n'a même pas essayé d'initialiser l'entrée de cache de routage ou si, comme dans le cas TCP, on n'y est pas parvenu (autrement dit si l'adresse de destination ne se trouve pas, en ce moment, dans le cache de routage) :

  – On déclare une structure de flux Internet générique, que l'on instantie avec les données que l'on possède.
  – On fait appel à la fonction `ip_route_output_flow()` pour déterminer une route. Si on n'en obtient pas, on renvoie le code d'erreur fourni par cette fonction.

    La fonction `ip_route_output_flow()` sera étudiée à sa place naturelle au chapitre 33. Nous verrons alors que si la table de routage possède une adresse par défaut, on est sûr d'obtenir une route.
  – Dans le cas d'une diffusion générale, on renvoie l'opposé du code d'erreur `EACCES`.
  – Si on est connecté, ce qui n'est pas le cas pour UDP, on renseigne le champ entrée de cache de destination du descripteur de couche transport pour les envois suivants.

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

```
950 static inline void
951 __sk_dst_set(struct sock *sk, struct dst_entry *dst)
952 {
953         struct dst_entry *old_dst;
954
955         old_dst = sk->sk_dst_cache;
956         sk->sk_dst_cache = dst;
957         dst_release(old_dst);
```

```
958 }
959
960 static inline void
961 sk_dst_set(struct sock *sk, struct dst_entry *dst)
962 {
963         write_lock(&sk->sk_dst_lock);
964         __sk_dst_set(sk, dst);
965         write_unlock(&sk->sk_dst_lock);
966 }
```

- Si le vecteur des drapeaux de l'en-tête de message spécifie qu'il faut confirmer :
  - On fait appel à la fonction `dst_confirm()` définie dans le fichier `linux/include/net/dst.h` :

Code Linux 2.6.10

```
188 static inline void dst_confirm(struct dst_entry *dst)
189 {
190         if (dst)
191                 neigh_confirm(dst->neighbour);
192 }
```

    La fonction `neigh_confirm()` est définie dans le fichier `linux/include/net/neighbour.h` :

Code Linux 2.6.10

```
316 static inline void neigh_confirm(struct neighbour *neigh)
317 {
318         if (neigh)
319                 neigh->confirmed = jiffies;
320 }
```

  - Si le vecteurs des drapeaux de l'en-tête de message indique qu'il ne faut pas sonder et si la longueur des données à envoyer est nulle, on renvoie 0.

- On détermine l'adresse source et l'adresse de destination.
- On verrouille le descripteur de couche transport.
- Si on a déjà mis un bouchon à la socket alors qu'il restait des trames en souffrance, il y a visiblement une erreur. On libère donc le descripteur de tampon, on indique qu'il y a une erreur et on renvoie l'opposé du code d'erreur `EINVAL`.
- On bouche la socket pour permettre d'envoyer les données (lignes 635–638).
- On ajoute la longueur des données à envoyer à celles des données en souffrance dans le champ adéquat de l'option UDP.
- On ajoute les données en souffrance en faisant appel à la fonction `ip_append_data()`, qui sera étudiée dans la section suivante.
- Si cette fonction nous indique qu'on peut faire partir les données, on fait appel à la fonction `udp_flush_pending_frames()` pour constituer le datagramme ; sinon on fait appel à la fonction `udp_push_pending_frames()` de constitution de données UDP en souffrance.
- On libère le descripteur de couche transport.
- On libère l'entrée de cache de routage.

  La fonction en ligne `ip_rt_put()` est définie dans le fichier `linux/include/net/route.h` :

Code Linux 2.6.10

```
131 static inline void ip_rt_put(struct rtable * rt)
132 {
133         if (rt)
134                 dst_release(&rt->u.dst);
135 }
```

- On libère l'option du cafteur.
- En cas d'erreur, on renvoie le code d'erreur.
- Sinon on incrémente l'information statistique concernant le nombre de datagrammes UDP envoyés avec succès et on renvoie le nombre d'octets envoyés.

32.2 Regroupement de données UDP

Nous venons de voir que la fonction `udp_sendmsg()` fait appel à la fonction `ip_append_data()` pour regrouper les données afin de constituer un datagramme de dimension optimum. Cette fonction a un nom qui fait penser *a priori* à la couche réseau. En fait celui-ci est dû au fait que ça ressemble à de la fragmentation, mais cette fonction n'est utilisée que par UDP.

32.2.1 Fonction principale

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

```
697 /*
698  *      ip_append_data() et ip_append_page() peuvent faire un grand datagramme IP
699  *      a partir de plusieurs pieces de donnees. Chaque piece sera contenue dans la socket
700  *      jusqu'a ce que ip_push_pending_frames() soit appelee. Chaque piece peut etre une page
701  *      ou des donnees moindre qu'une page.
702  *
703  *      Pas seulement UDP, d'autres protocoles de transport - par exemple les sockets
704  *      brutes - peuvent potentiellement utiliser cette interface.
705  *
706  *      PLUS TARD : la longueur doit etre ajustee avec du remplissage en queue lorsque
        necessaire.
707  */
708 int ip_append_data(struct sock *sk,
709                    int getfrag(void *from, char *to, int offset, int len,
710                                int odd, struct sk_buff *skb),
711                    void *from, int length, int transhdrlen,
712                    struct ipcm_cookie *ipc, struct rtable *rt,
713                    unsigned int flags)
714 {
715         struct inet_opt *inet = inet_sk(sk);
716         struct sk_buff *skb;
717
718         struct ip_options *opt = NULL;
719         int hh_len;
720         int exthdrlen;
721         int mtu;
722         int copy;
723         int err;
724         int offset = 0;
725         unsigned int maxfraglen, fragheaderlen;
726         int csummode = CHECKSUM_NONE;
727
728         if (flags&MSG_PROBE)
729                 return 0;
730
731         if (skb_queue_empty(&sk->sk_write_queue)) {
732                 /*
733                  * se preparer a mettre un bouchon.
734                  */
735                 opt = ipc->opt;
736                 if (opt) {
737                         if (inet->cork.opt == NULL) {
```

```
738                                 inet->cork.opt = kmalloc(sizeof(struct ip_options) + 40,
                                                             sk->sk_allocation);
739                                 if (unlikely(inet->cork.opt == NULL))
740                                         return -ENOBUFS;
741                         }
742                         memcpy(inet->cork.opt, opt, sizeof(struct ip_options)+opt->optlen);
743                         inet->cork.flags |= IPCORK_OPT;
744                         inet->cork.addr = ipc->addr;
745                 }
746                 dst_hold(&rt->u.dst);
747                 inet->cork.fragsize = mtu = dst_pmtu(&rt->u.dst);
748                 inet->cork.rt = rt;
749                 inet->cork.length = 0;
750                 sk->sk_sndmsg_page = NULL;
751                 sk->sk_sndmsg_off = 0;
752                 if ((exthdrlen = rt->u.dst.header_len) != 0) {
753                         length += exthdrlen;
754                         transhdrlen += exthdrlen;
755                 }
756         } else {
757                 rt = inet->cork.rt;
758                 if (inet->cork.flags & IPCORK_OPT)
759                         opt = inet->cork.opt;
760
761                 transhdrlen = 0;
762                 exthdrlen = 0;
763                 mtu = inet->cork.fragsize;
764         }
765         hh_len = LL_RESERVED_SPACE(rt->u.dst.dev);
766
767         fragheaderlen = sizeof(struct iphdr) + (opt ? opt->optlen : 0);
768         maxfraglen = ((mtu - fragheaderlen) & ~7) + fragheaderlen;
769
770         if (inet->cork.length + length > 0xFFFF - fragheaderlen) {
771                 ip_local_error(sk, EMSGSIZE, rt->rt_dst, inet->dport, mtu-exthdrlen);
772                 return -EMSGSIZE;
773         }
774
775         /*
776          * transhdrlen > 0 signifie que c'est le premier fragment et que nous desirons
777          * qu'il ne soit pas fragmente dans le futur.
778          */
779         if (transhdrlen &&
780             length + fragheaderlen <= mtu &&
781             rt->u.dst.dev->features&(NETIF_F_IP_CSUM|NETIF_F_NO_CSUM|NETIF_F_HW_CSUM) &&
782             !exthdrlen)
783                 csummode = CHECKSUM_HW;
784
785         inet->cork.length += length;
786
787         /* Qu'est-ce qui va se passer dans la boucle ci-dessous ?
788          *
789          * Nous utilisons la longueur de fragment calculee pour generer des skb enchaines,
790          * chaque segment est un fragment IP pret a l'envoi sur le reseau apres
791          * l'ajout de l'en-tete IP approprie.
792          */
793
794         if ((skb = skb_peek_tail(&sk->sk_write_queue)) == NULL)
795                 goto alloc_new_skb;
796
797         while (length > 0) {
798                 /* Verifier si les donnees restantes tiennent dans le paquet en cours. */
```

```
799                 copy = mtu - skb->len;
800                 if (copy < length)
801                         copy = maxfraglen - skb->len;
802                 if (copy <= 0) {
803                         char *data;
804                         unsigned int datalen;
805                         unsigned int fraglen;
806                         unsigned int fraggap;
807                         unsigned int alloclen;
808                         struct sk_buff *skb_prev;
809 alloc_new_skb:
810                         skb_prev = skb;
811                         if (skb_prev)
812                                 fraggap = skb_prev->len - maxfraglen;
813                         else
814                                 fraggap = 0;
815
816                         /*
817                          * Si les donnees restantes depassent la mtu,
818                          * nous savons que nous avons besoin de plus de fragments.
819                          */
820                         datalen = length + fraggap;
821                         if (datalen > mtu - fragheaderlen)
822                                 datalen = maxfraglen - fragheaderlen;
823                         fraglen = datalen + fragheaderlen;
824
825                         if ((flags & MSG_MORE) &&
826                             !(rt->u.dst.dev->features&NETIF_F_SG))
827                                 alloclen = mtu;
828                         else
829                                 alloclen = datalen + fragheaderlen;
830
831                         /* Le dernier fragment obtient de l'espace supplementaire a la fin.
832                          * Noter qu'avec MSG_MORE nous surallouons les fragments,
833                          * parce que nous n'avons aucune idee de quel fragment sera
834                          * le dernier.
835                          */
836                         if (datalen == length)
837                                 alloclen += rt->u.dst.trailer_len;
838
839                         if (transhdrlen) {
840                                 skb = sock_alloc_send_skb(sk,
841                                                 alloclen + hh_len + 15,
842                                                 (flags & MSG_DONTWAIT), &err);
843                         } else {
844                                 skb = NULL;
845                                 if (atomic_read(&sk->sk_wmem_alloc) <=
846                                     2 * sk->sk_sndbuf)
847                                         skb = sock_wmalloc(sk,
848                                                            alloclen + hh_len + 15, 1,
849                                                            sk->sk_allocation);
850                                 if (unlikely(skb == NULL))
851                                         err = -ENOBUFS;
852                         }
853                         if (skb == NULL)
854                                 goto error;
855
856                         /*
857                          *      Remplir les structures de controle
858                          */
859                         skb->ip_summed = csummode;
860                         skb->csum = 0;
```

```
861                     skb_reserve(skb, hh_len);
862
863                     /*
864                      *      Trouver ou est le debut des octets a placer.
865                      */
866                     data = skb_put(skb, fraglen);
867                     skb->nh.raw = data + exthdrlen;
868                     data += fragheaderlen;
869                     skb->h.raw = data + exthdrlen;
870
871                     if (fraggap) {
872                             skb->csum = skb_copy_and_csum_bits(
873                                     skb_prev, maxfraglen,
874                                     data + transhdrlen, fraggap, 0);
875                             skb_prev->csum = csum_sub(skb_prev->csum,
876                                                       skb->csum);
877                             data += fraggap;
878                             skb_trim(skb_prev, maxfraglen);
879                     }
880
881                     copy = datalen - transhdrlen - fraggap;
882                     if (copy > 0 && getfrag(from, data + transhdrlen, offset, copy,
                                                fraggap, skb) < 0) {
883                             err = -EFAULT;
884                             kfree_skb(skb);
885                             goto error;
886                     }
887
888                     offset += copy;
889                     length -= datalen - fraggap;
890                     transhdrlen = 0;
891                     exthdrlen = 0;
892                     csummode = CHECKSUM_NONE;
893
894                     /*
895                      * Mettre le paquet dans la file d'attente en souffrance.
896                      */
897                     __skb_queue_tail(&sk->sk_write_queue, skb);
898                     continue;
899             }
900
901             if (copy > length)
902                     copy = length;
903
904             if (!(rt->u.dst.dev->features&NETIF_F_SG)) {
905                     unsigned int off;
906
907                     off = skb->len;
908                     if (getfrag(from, skb_put(skb, copy),
909                                     offset, copy, off, skb) < 0) {
910                             __skb_trim(skb, off);
911                             err = -EFAULT;
912                             goto error;
913                     }
914             } else {
915                     int i = skb_shinfo(skb)->nr_frags;
916                     skb_frag_t *frag = &skb_shinfo(skb)->frags[i-1];
917                     struct page *page = sk->sk_sndmsg_page;
918                     int off = sk->sk_sndmsg_off;
919                     unsigned int left;
920
921                     if (page && (left = PAGE_SIZE - off) > 0) {
```

```
922                                 if (copy >= left)
923                                         copy = left;
924                                 if (page != frag->page) {
925                                         if (i == MAX_SKB_FRAGS) {
926                                                 err = -EMSGSIZE;
927                                                 goto error;
928                                         }
929                                         get_page(page);
930                                         skb_fill_page_desc(skb, i, page,
                                                                   sk->sk_sndmsg_off, 0);
931                                         frag = &skb_shinfo(skb)->frags[i];
932                                 }
933                         } else if (i < MAX_SKB_FRAGS) {
934                                 if (copy > PAGE_SIZE)
935                                         copy = PAGE_SIZE;
936                                 page = alloc_pages(sk->sk_allocation, 0);
937                                 if (page == NULL)  {
938                                         err = -ENOMEM;
939                                         goto error;
940                                 }
941                                 sk->sk_sndmsg_page = page;
942                                 sk->sk_sndmsg_off = 0;
943
944                                 skb_fill_page_desc(skb, i, page, 0, 0);
945                                 frag = &skb_shinfo(skb)->frags[i];
946                                 skb->truesize += PAGE_SIZE;
947                                 atomic_add(PAGE_SIZE, &sk->sk_wmem_alloc);
948                         } else {
949                                 err = -EMSGSIZE;
950                                 goto error;
951                         }
952                         if (getfrag(from, page_address(frag->page)+frag->page_offset
                             +frag->size, offset, copy, skb->len, skb) < 0) {
953                                 err = -EFAULT;
954                                 goto error;
955                         }
956                         sk->sk_sndmsg_off += copy;
957                         frag->size += copy;
958                         skb->len += copy;
959                         skb->data_len += copy;
960                 }
961                 offset += copy;
962                 length -= copy;
963         }
964
965         return 0;
966
967 error:
968         inet->cork.length -= length;
969         IP_INC_STATS(IPSTATS_MIB_OUTDISCARDS);
970         return err;
971 }
```

Autrement dit :

- On déclare une option Internet, que l'on instantie avec celle associée au descripteur de couche transport passé en argument.
- On déclare un descripteur de tampon de socket.
- On déclare une adresse de descripteur d'options IP, que l'on initialise à `NULL`.
- Si le vecteur de drapeaux passé en paramètre spécifie que l'on doit sonder, on renvoie 0.

- Si la file d'attente en émission du descripteur de couche transport est vide, on met un bouchon à la socket (lignes 732–755). Sinon l'entrée de cache de routage est initialisée avec celle associée à l'option IP.
- La longueur de l'en-tête physique est déterminée à partir de cette entrée de cache de routage.

  La macro `LL_RESERVED_SPACE()` est définie dans le fichier `linux/include/linux/net-device.h`:

Code Linux 2.6.10

```
213 /* Reserve HH_DATA_MOD octets aligne a hard_header_len, c'est-a-dire au moins ca.
214  * Une alternative est :
215  *   dev->hard_header_len ? (dev->hard_header_len +
216  *                           (HH_DATA_MOD - 1)) & ~(HH_DATA_MOD - 1) : 0
217  *
218  * Nous pourrions utiliser d'autres valeurs d'alignement, mais nous devons maintenir
219  * la relation : alignement HH <= alignement LL.
220  */
221 #define LL_RESERVED_SPACE(dev) \
222         (((dev)->hard_header_len&~(HH_DATA_MOD - 1)) + HH_DATA_MOD)
223 #define LL_RESERVED_SPACE_EXTRA(dev,extra) \
224         ((((dev)->hard_header_len+extra)&~(HH_DATA_MOD - 1)) + HH_DATA_MOD)
```

- La longueur de l'en-tête de fragmentation et la longueur maximum des fragments sont calculées.
- Si la longueur est trop importante, on fait appel à la fonction `ip_local_error()`, qui ne nous intéressera pas ici, et on renvoie l'opposé du code d'erreur `EMSGSIZE`.
- S'il s'agit du premier fragment remplissant quelques conditions on spécifie que le calcul de la somme de contrôle sera effectuée par le périphérique.
- On met à jour le champ longueur du bouchon de l'option Internet.
- On essaie d'instantier le descripteur de tampon de socket avec le premier élément de la file d'attente en écriture du descripteur de couche transport.
- Tant que la longueur des données à copier est strictement positive:
  - On calcule le nombre d'octets copiés.
  - Si le nombre d'octets copiés est négatif:
    - On détermine la longueur des données. Si celle-ci est supérieure à la longueur maximale, on tronque à celle-ci.
    - On détermine la longueur totale du fragment, c'est-à-dire longueur des données et taille de l'en-tête.
    - S'il s'agit du dernier fragment, il faut un peu plus d'espace à la fin.
    - Si on n'a pas tronqué la longueur, on instantie le descripteur de tampon de socket grâce à la fonction `sock_alloc_send_skb()`, que nous étudierons à la section suivante.
    - Sinon on essaie d'instantier ce descripteur grâce à la fonction `sock_wamalloc()`. Si on n'y parvient pas, on renvoie l'opposé du code d'erreur `ENOBUFS`.

      La fonction `sock_wamalloc()` est définie dans le fichier `linux/net/core/-sock.c`:

Code Linux 2.6.10

```
737 /*
738  * Alloue un skb du tampon d'envoi des sockets.
739  */
740 struct sk_buff *sock_wmalloc(struct sock *sk, unsigned long size, int force,
                                 int priority)
```

```
741 {
742         if (force || atomic_read(&sk->sk_wmem_alloc) < sk->sk_sndbuf) {
743                 struct sk_buff * skb = alloc_skb(size, priority);
744                 if (skb) {
745                         skb_set_owner_w(skb, sk);
746                         return skb;
747                 }
748         }
749         return NULL;
750 }
```

- Si on n'est pas parvenu à instantier le descripteur de tampon de socket, on met à jour la longueur du bouchon dans l'option Internet, on incrémente le nombre de paquets IP écartés lors de l'envoi et on renvoie le code d'erreur.
- On renseigne les champs concernant la somme de contrôle du descripteur de tampon de socket.
- On déplace le début des données du tampon de socket pour accueillir l'en-tête physique.
- On se place au début des données à copier et depuis lesquelles copier.
- S'il y a un trou dans la fragmentation, on arrange ça.
- On utilise la fonction `getfrag()` passée en argument pour copier les données. Si on n'y parvient pas, on libère le descripteur de tampon, on met à jour la longueur du bouchon dans l'option Internet, on incrémente le nombre de paquets IP écartés lors de l'envoi et on renvoie l'opposé du code d'erreur `EFAULT`.

  Dans l'appel de la fonction `udp_sendmsg()`, le pointeur de fonction `getfrag()` portait sur `ip_generic_getfrag()`, que nous étudierons dans l'une des sous-sections suivantes.
- On place le paquet dans la file d'attente des paquets à émettre du descripteur de couche transport.

- Si le nombre d'octets à copier est supérieur à la longueur maximale, on tronque à cette dernière longueur.
- On effectue des opérations analogues s'il s'agit d'une page complète.

- Si tout s'est bien déroulé, on renvoie 0.

32.2.2 Instantiation d'un tampon de socket pour l'émission

Nous venons de voir que la fonction `ip_append_data()` fait appel à la fonction `sock_alloc_send_skb()` d'instantiation de descripteur de tampon de socket en émission. Celle-ci est définie dans le fichier `linux/net/core/sock.c`:

Code Linux 2.6.10

```
910 struct sk_buff *sock_alloc_send_skb(struct sock *sk, unsigned long size,
911                                       int noblock, int *errcode)
912 {
913         return sock_alloc_send_pskb(sk, size, 0, noblock, errcode);
914 }
```

qui renvoie à la fonction `sock_alloc_send_pskb()` définie dans le même fichier :

Code Linux 2.6.10

```
824 /*
825  *      Manipulateurs de tampon d'emission/reception generiques
826  */
827
828 struct sk_buff *sock_alloc_send_pskb(struct sock *sk, unsigned long header_len,
```

```
829                                     unsigned long data_len, int noblock, int *errcode)
830 {
831         struct sk_buff *skb;
832         unsigned int gfp_mask;
833         long timeo;
834         int err;
835 
836         gfp_mask = sk->sk_allocation;
837         if (gfp_mask & __GFP_WAIT)
838                 gfp_mask |= __GFP_REPEAT;
839 
840         timeo = sock_sndtimeo(sk, noblock);
841         while (1) {
842                 err = sock_error(sk);
843                 if (err != 0)
844                         goto failure;
845 
846                 err = -EPIPE;
847                 if (sk->sk_shutdown & SEND_SHUTDOWN)
848                         goto failure;
849 
850                 if (atomic_read(&sk->sk_wmem_alloc) < sk->sk_sndbuf) {
851                         skb = alloc_skb(header_len, sk->sk_allocation);
852                         if (skb) {
853                                 int npages;
854                                 int i;
855 
856                                 /* Pas de page, c'est fait... */
857                                 if (!data_len)
858                                         break;
859 
860                                 npages = (data_len + (PAGE_SIZE - 1)) >> PAGE_SHIFT;
861                                 skb->truesize += data_len;
862                                 skb_shinfo(skb)->nr_frags = npages;
863                                 for (i = 0; i < npages; i++) {
864                                         struct page *page;
865                                         skb_frag_t *frag;
866 
867                                         page = alloc_pages(sk->sk_allocation, 0);
868                                         if (!page) {
869                                                 err = -ENOBUFS;
870                                                 skb_shinfo(skb)->nr_frags = i;
871                                                 kfree_skb(skb);
872                                                 goto failure;
873                                         }
874 
875                                         frag = &skb_shinfo(skb)->frags[i];
876                                         frag->page = page;
877                                         frag->page_offset = 0;
878                                         frag->size = (data_len >= PAGE_SIZE ?
879                                                       PAGE_SIZE :
880                                                       data_len);
881                                         data_len -= PAGE_SIZE;
882                                 }
883 
884                                 /* Succes complet... */
885                                 break;
886                         }
887                         err = -ENOBUFS;
888                         goto failure;
889                 }
890                 set_bit(SOCK_ASYNC_NOSPACE, &sk->sk_socket->flags);
```

```
891                 set_bit(SOCK_NOSPACE, &sk->sk_socket->flags);
892                 err = -EAGAIN;
893                 if (!timeo)
894                         goto failure;
895                 if (signal_pending(current))
896                         goto interrupted;
897                 timeo = sock_wait_for_wmem(sk, timeo);
898         }
899
900         skb_set_owner_w(skb, sk);
901         return skb;
902
903 interrupted:
904         err = sock_intr_errno(timeo);
905 failure:
906         *errcode = err;
907         return NULL;
908 }
```

Autrement dit :

- On déclare un descripteur de tampon de socket, un masque, un délai et un code de retour.
- On instantie le masque avec celui du descripteur de couche transport passé en argument, que l'on modifie éventuellement.
- On instantie le délai à partir de celui du descripteur de couche transport passé en argument.
- Si le descripteur de couche transport spécifie une erreur, on reporte le code d'erreur dans l'argument passé par adresse et on renvoie `NULL`.
- Si le descripteur de couche transport spécifie que la socket est fermée, on place l'opposé du code d'erreur `EPIPE` dans l'argument passé par adresse et on renvoie `NULL`.
- Si la taille du tampon d'envoi, telle que spécifiée par le descripteur de couche transport passé en argument, est supérieure à ce qui a été alloué au tampon :
  - On essaie d'allouer de la place pour le descripteur de tampon. Si on n'y parvient pas, on place l'opposé du code d'erreur `ENOBUFS` dans l'argument passé par adresse et on renvoie `NULL`.
  - On détermine le nombre de pages de mémoire vive dont on va avoir besoin, que l'on reporte dans le champ adéquat de l'appendice du descripteur de tampon.
  - Pour chacune de ces pages :
    - On déclare une page et une adresse de descripteur de fragment de tampon de socket.
    - On essaie d'instantier cette page. Si on n'y parvient pas, on rectifie le nombre de pages dans le champ adéquat de l'appendice du descripteur de tampon, on libère le descripteur de tampon, on place l'opposé du code d'erreur `ENOBUFS` dans l'argument passé par adresse et on renvoie `NULL`.
    - On initialise l'adresse de descripteur de fragment de tampon, on renseigne les champs de ce descripteur et on diminue la longueur des données restant à placer dans le tampon.
- Sinon :
  - On indique qu'il n'y pas assez de place pour le moment dans les drapeaux.
  - Si le délai est écoulé, on place l'opposé du code d'erreur `EAGAIN` dans l'argument passé par adresse et on renvoie `NULL`.

  - Si un signal nous dit de nous arrêter, on place l'opposé du code d'erreur fourni par ce signal dans l'argument passé par adresse et on renvoie `NULL`.
  - On met à jour le délai pour permettre d'attendre de la mémoire vive en écriture et on recommence.

- On associe le descripteur de tampon et le descripteur de couche transport et on renvoie l'adresse du descripteur de tampon, alloué avec succès si on parvient à cette instruction.

32.2.3 Récupération des fragments

Code Linux 2.6.10

Nous venons de voir que la fonction `ip_append_data()` fait appel à la fonction `ip_generic_getfrag()`. Celle-ci est définie dans le fichier `linux/net/ipv4/ip_output.c`:

```
669 int
670 ip_generic_getfrag(void *from, char *to, int offset, int len, int odd, struct sk_buff *skb)
671 {
672         struct iovec *iov = from;
673
674         if (skb->ip_summed == CHECKSUM_HW) {
675                 if (memcpy_fromiovecend(to, iov, offset, len) < 0)
676                         return -EFAULT;
677         } else {
678                 unsigned int csum = 0;
679                 if (csum_partial_copy_fromiovecend(to, iov, offset, len, &csum) < 0)
680                         return -EFAULT;
681                 skb->csum = csum_block_add(skb->csum, csum, odd);
682         }
683         return 0;
684 }
```

qui renvoie à la fonction `memcpy_fromiovecend()` ou à la fonction `csum_partial_copy_fromiovecend()` suivant le cas.

Code Linux 2.6.10

Ces fonctions sont définies dans le fichier `linux/net/core/iovec.c`:

```
147 /*
148  *      Pour utilisation avec ip_build_xmit
149  */
150 int memcpy_fromiovecend(unsigned char *kdata, struct iovec *iov, int offset,
151                         int len)
152 {
153         /* Passer sur les iovec termines */
154         while (offset >= iov->iov_len) {
155                 offset -= iov->iov_len;
156                 iov++;
157         }
158
159         while (len > 0) {
160                 u8 __user *base = iov->iov_base + offset;
161                 int copy = min_t(unsigned int, len, iov->iov_len - offset);
162
163                 offset = 0;
164                 if (copy_from_user(kdata, base, copy))
165                         return -EFAULT;
166                 len -= copy;
167                 kdata += copy;
168                 iov++;
169         }
170
171         return 0;
172 }
```

```
173
174 /*
175  *      Et maintenant pour le tout en un : copie et calcule la somme de controle
176  *      directement d'un iovec utilisateur dans un datagramme
177  *      Appels a csum_partial mais le dernier doit etre un multiple de 32 bits
178  *
179  *      ip_build_xmit doit s'assurer lorsqu'on fragmente que seul le dernier
180  *      appel a cette fonction sera sur un non aligne.
181  */
182 int csum_partial_copy_fromiovecend(unsigned char *kdata, struct iovec *iov,
183                                    int offset, unsigned int len, int *csump)
184 {
185         int csum = *csump;
186         int partial_cnt = 0, err = 0;
187
188         /* Passer sur les iovec termines */
189         while (offset >= iov->iov_len) {
190                 offset -= iov->iov_len;
191                 iov++;
192         }
193
194         while (len > 0) {
195                 u8 __user *base = iov->iov_base + offset;
196                 int copy = min_t(unsigned int, len, iov->iov_len - offset);
197
198                 offset = 0;
199
200                 /* Il y a un reste du precedent iov. */
201                 if (partial_cnt) {
202                         int par_len = 4 - partial_cnt;
203
204                         /* le composant iov est trop court... */
205                         if (par_len > copy) {
206                                 if (copy_from_user(kdata, base, copy))
207                                         goto out_fault;
208                                 kdata += copy;
209                                 base += copy;
210                                 partial_cnt += copy;
211                                 len -= copy;
212                                 iov++;
213                                 if (len)
214                                         continue;
215                                 *csump = csum_partial(kdata - partial_cnt,
216                                                       partial_cnt, csum);
217                                 goto out;
218                         }
219                         if (copy_from_user(kdata, base, par_len))
220                                 goto out_fault;
221                         csum = csum_partial(kdata - partial_cnt, 4, csum);
222                         kdata += par_len;
223                         base  += par_len;
224                         copy  -= par_len;
225                         len   -= par_len;
226                         partial_cnt = 0;
227                 }
228
229                 if (len > copy) {
230                         partial_cnt = copy % 4;
231                         if (partial_cnt) {
232                                 copy -= partial_cnt;
233                                 if (copy_from_user(kdata + copy, base + copy,
234                                                    partial_cnt))
```

```
235                                         goto out_fault;
236                         }
237                 }
238
239                 if (copy) {
240                         csum = csum_and_copy_from_user(base, kdata, copy,
241                                                        csum, &err);
242                         if (err)
243                                 goto out;
244                 }
245                 len   -= copy + partial_cnt;
246                 kdata += copy + partial_cnt;
247                 iov++;
248         }
249         *csump = csum;
250 out:
251         return err;
252
253 out_fault:
254         err = -EFAULT;
255         goto out;
256 }
```

sur lesquelles il n'y a pas grand chose à dire.

32.3 Constitution du datagramme

Code Linux 2.6.10

Une fois que les données sont placées dans le tampon de socket, il faut constituer le datagramme, c'est-à-dire essentiellement renseigner les champs de l'en-tête UDP. Ceci est l'objet de la fonction `udp_push_pending_frames()`, définie dans le fichier `linux/net/ipv4/udp.c`:

```
396 /*
397  * Pousse au dehors toutes les donnees en suspens en tant que datagramme UDP. La socket
       est verrouillee.
398  */
399 static int udp_push_pending_frames(struct sock *sk, struct udp_opt *up)
400 {
401         struct inet_opt *inet = inet_sk(sk);
402         struct flowi *fl = &inet->cork.fl;
403         struct sk_buff *skb;
404         struct udphdr *uh;
405         int err = 0;
406
407         /* Attrape le skbuff la ou l'espace de l'en-tete UDP existe. */
408         if ((skb = skb_peek(&sk->sk_write_queue)) == NULL)
409                 goto out;
410
411         /*
412          * Cree un en-tete UDP
413          */
414         uh = skb->h.uh;
415         uh->source = fl->fl_ip_sport;
416         uh->dest = fl->fl_ip_dport;
417         uh->len = htons(up->len);
418         uh->check = 0;
419
420         if (sk->sk_no_check == UDP_CSUM_NOXMIT) {
421                 skb->ip_summed = CHECKSUM_NONE;
422                 goto send;
423         }
424
```

```
425         if (skb_queue_len(&sk->sk_write_queue) == 1) {
426                 /*
427                  * Seulement un fragment dans la socket.
428                  */
429                 if (skb->ip_summed == CHECKSUM_HW) {
430                         skb->csum = offsetof(struct udphdr, check);
431                         uh->check = ~csum_tcpudp_magic(fl->fl4_src, fl->fl4_dst,
432                                         up->len, IPPROTO_UDP, 0);
433                 } else {
434                         skb->csum = csum_partial((char *)uh,
435                                         sizeof(struct udphdr), skb->csum);
436                         uh->check = csum_tcpudp_magic(fl->fl4_src, fl->fl4_dst,
437                                         up->len, IPPROTO_UDP, skb->csum);
438                         if (uh->check == 0)
439                                 uh->check = -1;
440                 }
441         } else {
442                 unsigned int csum = 0;
443                 /*
444                  * HW-checksum ne marche pas lorsqu'il y a deux segments ou plus
445                  * dans la socket puisque toutes les csums de sk_buffs
446                  * doivent etre ensemble.
447                  */
448                 if (skb->ip_summed == CHECKSUM_HW) {
449                         int offset = (unsigned char *)uh - skb->data;
450                         skb->csum = skb_checksum(skb, offset, skb->len - offset, 0);
451
452                         skb->ip_summed = CHECKSUM_NONE;
453                 } else {
454                         skb->csum = csum_partial((char *)uh,
455                                         sizeof(struct udphdr), skb->csum);
456                 }
457
458                 skb_queue_walk(&sk->sk_write_queue, skb) {
459                         csum = csum_add(csum, skb->csum);
460                 }
461                 uh->check = csum_tcpudp_magic(fl->fl4_src, fl->fl4_dst,
462                                 up->len, IPPROTO_UDP, csum);
463                 if (uh->check == 0)
464                         uh->check = -1;
465         }
466 send:
467         err = ip_push_pending_frames(sk);
468 out:
469         up->len = 0;
470         up->pending = 0;
471         return err;
472 }
```

Autrement dit :

- On déclare une option Internet, que l'on instantantie avec celle associée au descripteur de couche transport passé en argument.
- On déclare un flux Internet, que l'on instantie avec celui associé à cette option Internet.
- On déclare un descripteur de tampon de socket, une adresse d'en-tête UDP et un code d'erreur.
- On essaie d'instantier le descripteur de tampon avec le premier élément de la file d'attente en écriture du descripteur de couche transport passé en argument, sans retirer celui-ci de la file d'attente. Si on n'y parvient pas, on renseigne quelques champs de l'option UDP dont l'adresse est passée en argument (longueur nulle et pas de de données en attente) et

on renvoie 0.

- On instantie l'adresse d'en-tête UDP avec celle du champ de même type du descripteur de tampon de socket.
- On renseigne les champs de l'en-tête UDP : adresse source et adresse de destination à partir des données du flux Internet, longueur des données à partir des options UDP passées en argument et pas de somme de contrôle.
- Si la valeur du champ `sk_no_check` du descripteur de couche transport passé en argument est `UDP_CSUM_NOXMIT`, on renseigne le champ `ip_summed` du descripteur de tampon avec `CHECKSUM_NONE`, on fait appel à la fonction `ip_push_pending_frames()` étudiée au chapitre 33, on renseigne les champs de l'option UDP passée en argument et on renvoie le code fourni par cette fonction.

  Les constantes `UDP_CSUM_NOXMIT`, `UDP_CSUM_NORCV` et `UDP_CSUM_DEFAULT` sont définies dans le fichier `linux/include/net/udp.h` :

Code Linux 2.6.10

```
54 /* Note : ceci doit concorder avec 'valbool' dans sock_setsockopt */
55 #define UDP_CSUM_NOXMIT         1
56
57 /* Utilise par la couche SunRPC/xprt. */
58 #define UDP_CSUM_NORCV          2
59
60 /* La valeur par defaut, comme dans la RFC, demande toujours de faire le calcul. */
61 #define UDP_CSUM_DEFAULT        0
```

- Si la taille de la file d'attente est égale à 1, c'est qu'il y a seulement un fragment pour le datagramme :

  - Si la valeur du champ `ip_summed` du descripteur de tampon est `CHECKSUM_HW`, on renseigne le champ `csum` de ce descripteur avec la valeur du champ `check` de l'en-tête UDP et le champ `check` de l'en-tête UDP en utilisant la fonction `csum_tcpudp_magic()`.

    La macro générale `offsetof()` est définie dans le fichier `linux/include/linux/stddef.h` :

Code Linux 2.6.10

```
13 #undef offsetof
14 #ifdef __compiler_offsetof
15 #define offsetof(TYPE,MEMBER) __compiler_offsetof(TYPE,MEMBER)
16 #else
17 #define offsetof(TYPE, MEMBER) ((size_t) &((TYPE *)0)->MEMBER)
18 #endif
```

  - Sinon on renseigne le champ `csum` du descripteur de tampon grâce au calcul partiel et le champ `check` de l'en-tête UDP en utilisant la même fonction que ci-dessus. Si la valeur obtenue par calcul est nulle, on indique -1 puisque 0 a une signification particulière.

- Si la taille de la file d'attente est supérieure à deux :

  - Si la valeur du champ `ip_summed` du descripteur de tampon est égale à `CHECKSUM_HW` : on calcule le décalage de l'en-tête UDP ; on calcule la valeur du champ `csum` du descripteur de tampon ; on positionne la valeur du champ `ip_summed` du descripteur de tampon à `CHECKSUM_NONE`.
  - Sinon on se contente de calculer la valeur du champ `csum` du descripteur de tampon.
  - On parcourt la file d'attente en écriture afin de compléter le calcul de `csum`.

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

Code Linux 2.6.10

```
1073 #define skb_queue_walk(queue, skb) \
1074                 for (skb = (queue)->next, prefetch(skb->next);  \
1075                      (skb != (struct sk_buff *)(queue));        \
1076                      skb = skb->next, prefetch(skb->next))
1077
```

  - On calcule le champ `check` de l'en-tête UDP.

- On fait appel à la fonction `ip_push_pending_frame()`, on renseigne les champs de l'option UDP passée en argument et on renvoie le code fourni par cette fonction.

32.4 Envoi du datagramme à la couche réseau

Nous venons de voir que la fonction `udp_push_pending_frames()` de constitution du datagramme fait appel à la fonction `udp_flush_pending_frames()` ou directement à la fonction `ip_push_pending_frames()` pour envoyer le datagramme à la couche réseau IP. Nous allons étudier la première fonction dans cette section. Nous étudierons la seconde fonction dans le chapitre 33

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

```
382 /*
383  * Parcourt toutes les donnees en suspens et annule la mise du bouchon. La socket est
       verrouillee.
384  */
385 static void udp_flush_pending_frames(struct sock *sk)
386 {
387         struct udp_opt *up = udp_sk(sk);
388
389         if (up->pending) {
390                 up->len = 0;
391                 up->pending = 0;
392                 ip_flush_pending_frames(sk);
393         }
394 }
```

Autrement dit on fait directement appel à la fonction `ip_push_pending_frame()`.