Chapitre 30

Réception d'un datagramme par une socket

Nous avons laissé au chapitre 25 le datagramme arrivé sur la carte réseau dans la file d'attente de réception UDP. Nous allons voir dans ce chapitre comment l'application récupère celui-ci.

Rappelons qu'une application récupère des données d'une connexion non connectée grâce à l'appel système `recvfrom()`, comme le montre le tableau vu au chapitre 7 précisant les appels systèmes concernant l'envoi et la réception des données pour une socket :

Écriture	Lecture	Fonctionnalités ajoutées
write()	read()	Fonction de base
send()	recv()	Ajoute un argument de drapeaux
sendto()	recvfrom()	Ajoute une adresse de socket et une longueur
writev()	readv()	Ni socket ni drapeaux, mais un vecteur
sendmsg()	recvmsg()	Drapeaux, socket, vecteur et données ancillaires.

Nous allons commencer par une section sur l'implémentation des types liés aux fonctions d'entrée-sortie, qui interviennent même dans les cas simples.

30.1 Types liés aux fonctions d'entrée-sortie

30.1.1 Vecteurs d'entrée-sortie

Nous avons vu que les fonctions de lecture et d'écriture `readv()` et `writev()` de UNIX ont comme paramètre un vecteur d'entrée-sortie. Le type `struct iovec` des éléments des vecteurs d'entrée-sortie est défini dans le fichier en-tête `linux/include/linux/uio.h`:

Code Linux 2.6.10

```
7 /*
8  *      Structures UIO style Berkeley   -       Alan Cox 1994.
9  *
[...]
17 /* Un mot d'avertissement : notre structure uio est incompatible avec celle de la
      bibliotheque C (qui est maintenant obsolete). Supprimer celle de la bibliotheque
18    C situee dans sys/uio.h si vous avez un tres vieil ensemble de bibliotheques */
19
20 struct iovec
21 {
22         void __user *iov_base;   /* BSD utilise caddr_t (1003.1g necessite void *) */
23         __kernel_size_t iov_len; /* Doit etre size_t (1003.1g) */
24 };
```

30.1.2 En-tête de message

Nous avons vu que les fonctions de lecture et d'écriture `recvmsg()` et `sendmsg()` de UNIX ont comme paramètre un en-tête de message de type `struct msghdr`. Celui-ci est défini dans le fichier en-tête `linux/include/linux/socket.h`:

Code Linux 2.6.10

```
47 /*
48  *      Lorsque nous passons un message 4.4BSD nous utilisons un systeme de passage de
49  *      message 4.4BSD, pas 4.3. Donc msg_accrights(len) manque maintenant. Il
50  *      appartient a une obscure emulation de libc ou a bin.
51  */
52
53 struct msghdr {
54         void    *       msg_name;       /* Nom de la socket             */
55         int             msg_namelen;    /* Longueur du nom              */
56         struct iovec *  msg_iov;        /* Blocs de donnees             */
57         __kernel_size_t msg_iovlen;     /* Nombre de blocs              */
58         void    *       msg_control;    /* Pour la magie du protocole (par exemple le
                                              passage de descripteurs de fichier BSD) */
59         __kernel_size_t msg_controllen; /* Longueur de la liste cmsg     */
60         unsigned        msg_flags;
61 };
```

30.1.3 En-tête de contrôle

Nous venons de voir que les en-têtes de message font référence à des messages de contrôle. Ceux-ci possédent des en-têtes du type `struct cmsghdr`. Celui-ci est défini dans le même fichier en-tête:

Code Linux 2.6.10

```
63 /*
64  *      POSIX 1003.1g - informations sur les objets de donnees ancillaires
65  *      Les donnees ancillaires consistent en une suite de couples
66  *      (cmsghdr, cmsg_data[])
67  */
68
69 struct cmsghdr {
70         __kernel_size_t cmsg_len;       /* compteur d'octets des donnees, y compris hdr */
71         int             cmsg_level;     /* protocole d'origine */
```

```
72         int             cmsg_type;      /* type specifique au protocole */
73 };
```

30.2 Implémentation générale de la réception des datagrammes

30.2.1 Fonction d'appel

La fonction d'appel `sys_recvfrom()` est définie dans le fichier `linux/net/socket.c`: Code Linux 2.6.10

```
1553 /*
1554  *      Recoit une trame d'une socket et optionnellement enregistre l'adresse de
1555  *      l'emetteur. Nous verifions qu'on peut ecrire sur les tampons et, si necessaire,
1556  *      nous deplacons l'adresse de l'emetteur du noyau a l'espace utilisateur.
1557  */
1558
1559 asmlinkage long sys_recvfrom(int fd, void __user * ubuf, size_t size, unsigned flags,
1560                              struct sockaddr __user *addr, int __user *addr_len)
1561 {
1562         struct socket *sock;
1563         struct iovec iov;
1564         struct msghdr msg;
1565         char address[MAX_SOCK_ADDR];
1566         int err,err2;
1567
1568         sock = sockfd_lookup(fd, &err);
1569         if (!sock)
1570                 goto out;
1571
1572         msg.msg_control=NULL;
1573         msg.msg_controllen=0;
1574         msg.msg_iovlen=1;
1575         msg.msg_iov=&iov;
1576         iov.iov_len=size;
1577         iov.iov_base=ubuf;
1578         msg.msg_name=address;
1579         msg.msg_namelen=MAX_SOCK_ADDR;
1580         if (sock->file->f_flags & O_NONBLOCK)
1581                 flags |= MSG_DONTWAIT;
1582         err=sock_recvmsg(sock, &msg, size, flags);
1583
1584         if(err >= 0 && addr != NULL)
1585         {
1586                 err2=move_addr_to_user(address, msg.msg_namelen, addr, addr_len);
1587                 if(err2<0)
1588                         err=err2;
1589         }
1590         sockfd_put(sock);
1591 out:
1592         return err;
1593 }
```

Autrement dit :

- On déclare un descripteur de socket, un élément de vecteur d'entrée-sortie, un en-tête de message, une adresse et deux codes de retour.
- On essaie d'instantier le descripteur de socket avec celui associé au numéro de fichier passé en argument. Si on n'y parvient pas, on renvoie le code fourni par la fonction `sockfd_lookup()`.

- On initialise l'en-tête de message: pas de données ancillaires, un seul élément de vecteur d'entrée-sortie (celui déclaré ci-dessus) et d'adresse celle passée en argument, de longueur celle passée en argument.
- On initialise le vecteur d'entrée-sortie avec l'adresse de tampon et la taille passées en argument.
- On modifie éventuellement le vecteur de drapeaux passé en argument.
- On fait appel à la fonction `sock_recvmsg()` étudiée ci-dessous, on libère le descripteur de socket et on renvoie le code qu'elle fournit.

30.2.2 Réception de message de socket

Code Linux 2.6.0

La fonction `sock_recvmsg()` est également définie dans le fichier `linux/net/socket.c`:

```
599 int sock_recvmsg(struct socket *sock, struct msghdr *msg,
600                   size_t size, int flags)
601 {
602         struct kiocb iocb;
603         struct sock_iocb siocb;
604         int ret;
605
606         init_sync_kiocb(&iocb, NULL);
607         iocb.private = &siocb;
608         ret = __sock_recvmsg(&iocb, sock, msg, size, flags);
609         if (-EIOCBQUEUED == ret)
610                 ret = wait_on_sync_kiocb(&iocb);
611         return ret;
612 }
```

Autrement dit:

- On déclare un bloc de contrôle d'entrée-sortie noyau.

  Code Linux 2.6.10

  Le type général `struct kiocb` des blocs d'entrée-sortie noyau est défini dans le fichier `linux/include/linux/aio.h`:

```
46 struct kiocb {
47         struct list_head        ki_run_list;
48         long                    ki_flags;
49         int                     ki_users;
50         unsigned                ki_key;         /* id de cette requete */
51
52         struct file             *ki_filp;
53         struct kioctx           *ki_ctx;        /* peut etre NULL pour les ops sync */
54         int                     (*ki_cancel)(struct kiocb *, struct io_event *);
55         ssize_t                 (*ki_retry)(struct kiocb *);
56         void                    (*ki_dtor)(struct kiocb *);
57
58         struct list_head        ki_list;        /* le coeur de aio l'utilise
60                                                    * pour l'annulation */
60
61         union {
62                 void __user             *user;
63                 struct task_struct      *tsk;
64         } ki_obj;
65         __u64                   ki_user_data;   /* donnees de l'utilisateur a
                                                      completer */
66         loff_t                  ki_pos;
67         /* Etat que nous nous rappelons etre capable de redemarrer/recuperer */
68         unsigned short          ki_opcode;
69         size_t                  ki_nbytes;      /* copie de iocb->aio_nbytes */
```

```
70         char                    __user *ki_buf; /* reste de iocb->aio_buf */
71         size_t                  ki_left;        /* octets restants */
72         wait_queue_t            ki_wait;
73         long                    ki_retried;     /* juste pour test */
74         long                    ki_kicked;      /* juste pour test */
75         long                    ki_queued;      /* juste pour test */
76
77         void                    *private;
78 };
```

– On déclare un code de retour.
– On initialise le bloc de contrôle d'entrée-sortie noyau.

La macro générale `init_sync_kiocb()` est définie dans le fichier `linux/include/linux/aio.h`: Code Linux 2.6.10

```
81 #define init_sync_kiocb(x, filp)                        \
82         do {                                            \
83                 struct task_struct *tsk = current;      \
84                 (x)->ki_flags = 0;                      \
85                 (x)->ki_users = 1;                      \
86                 (x)->ki_key = KIOCB_SYNC_KEY;           \
87                 (x)->ki_filp = (filp);                  \
88                 (x)->ki_ctx = &tsk->active_mm->default_kioctx;  \
89                 (x)->ki_cancel = NULL;                  \
90                 (x)->ki_dtor = NULL;                    \
91                 (x)->ki_obj.tsk = tsk;                  \
92                 (x)->ki_user_data = 0;                  \
93                 init_wait((&(x)->ki_wait));             \
94         } while (0)
```

– On fait appel à la fonction `__sock_recvmsg()` interne de réception de socket, que nous étudierons dans la sous-section suivante.
– Si le code de retour de cette dernière fonction est `-EIOCBQUEUED`, on attend la fin de la synchronisation.

La fonction générale `wait_on_sync_kiocb()` est définie dans le fichier `linux/fs/aio.c`: Code Linux 2.6.10

```
311 /* wait_on_sync_kiocb :
312  *      attend que le kiocb synchronise donne ait termine.
313  */
314 ssize_t fastcall wait_on_sync_kiocb(struct kiocb *iocb)
315 {
316         while (iocb->ki_users) {
317                 set_current_state(TASK_UNINTERRUPTIBLE);
318                 if (!iocb->ki_users)
319                         break;
320                 schedule();
321         }
322         __set_current_state(TASK_RUNNING);
323         return iocb->ki_user_data;
324 }
```

– On renvoie le code fourni par l'une ou l'autre des fonctions appelées.

30.2.3 Passage espace noyau/espace utilisateur

La fonction `move_addr_to_user()` est définie dans le fichier `linux/net/socket.c`: Code Linux 2.6.10

```
232 /**
233  *      move_addr_to_user       -       copie une adresse dans l'espace utilisateur
234  *      @kaddr : adresse de l'espace noyau
```

```
235  *      @klen : longueur de l'adresse dans le noyau
236  *      @uaddr : adresse de l'espace utilisateur
237  *      @ulen : pointeur sur le champ longueur de l'utilisateur
238  *
239  *      La valeur pointee par ulen en entree est la longueur du tampon disponible.
240  *      Ceci est surcharge par l'espace tampon utilise. -EINVAL est renvoye
241  *      si un tampon trop long est specifie et en cas de taille de tampon negative. -EFAULT
242  *      est renvoye lorsque le tampon ou le champ longueur ne sont pas
243  *      accessibles.
244  *      Apres avoir copier les donnees jusqu'a la limite specifiee par l'utilisateur, la
245  *      vraie longueur des donnees est ecrite sur la limite de longueur specifiee
246  *      par l'utilisateur. Zero est renvoye en cas de succes.
247  */
248
249 int move_addr_to_user(void *kaddr, int klen, void __user *uaddr, int __user *ulen)
250 {
251         int err;
252         int len;
253
254         if((err=get_user(len, ulen)))
255                 return err;
256         if(len>klen)
257                 len=klen;
258         if(len<0 || len> MAX_SOCK_ADDR)
259                 return -EINVAL;
260         if(len)
261         {
262                 if(copy_to_user(uaddr,kaddr,len))
263                         return -EFAULT;
264         }
265         /*
266          *      "fromlen referera a la valeur avant troncature..."
267          *                      1003.1g
268          */
269         return __put_user(klen, ulen);
270 }
```

Le code se comprend aisément.

30.2.4 Fonction interne de réception d'un message

Code Linux 2.6.10

La fonction `__sock_recvmsg()` est définie dans le fichier `linux/net/socket.c`:

```
580 static inline int __sock_recvmsg(struct kiocb *iocb, struct socket *sock,
581                                   struct msghdr *msg, size_t size, int flags)
582 {
583         int err;
584         struct sock_iocb *si = kiocb_to_siocb(iocb);
585
586         si->sock = sock;
587         si->scm = NULL;
588         si->msg = msg;
589         si->size = size;
590         si->flags = flags;
591
592         err = security_socket_recvmsg(sock, msg, size, flags);
593         if (err)
594                 return err;
595
596         return sock->ops->recvmsg(iocb, sock, msg, size, flags);
597 }
```

Autrement dit :

- On déclare une adresse de bloc de contrôle d'entrée-sortie de socket.

  Le type `struct sock_iocb` est défini dans le fichier en-tête `linux/include/net/sock.h` :

  Code Linux 2.6.10

```
620 /* sock_iocb : utilise pour demarrer le processus async des E/S d'une socket */
621 struct sock_iocb {
622         struct list_head        list;
623
624         int                     flags;
625         int                     size;
626         struct socket           *sock;
627         struct sock             *sk;
628         struct scm_cookie       *scm;
629         struct msghdr           *msg, async_msg;
630         struct iovec            async_iov;
631         struct kiocb            *kiocb;
632 };
```

- On initialise cette adresse avec celle de la partie spécifique du bloc de contrôle d'entrée-sortie noyau passé en argument.

  La fonction en ligne `kiocb_to_siocb()` est définie dans le fichier en-tête `linux/include/net/sock.h` :

  Code Linux 2.6.10

```
634 static inline struct sock_iocb *kiocb_to_siocb(struct kiocb *iocb)
635 {
636         return (struct sock_iocb *)iocb->private;
637 }
```

- On renseigne les champs de ce bloc de contrôle d'entrée-sortie de socket avec les valeurs passées en argument, l'adresse du cafteur (*cookie* en anglais) étant initialisée à `NULL`.
- On fait appel à la fonction de sécurité `security_socket_recvmsg()` qui ne nous intéresse pas dans cet ouvrage. Si le droit d'accès n'est pas accordé, on renvoie le code d'erreur fourni par cette fonction.
- On fait appel à la fonction de réception de message spécifique à la famille de protocoles utilisée pour la lecture proprement dite et on renvoie le code fourni par celle-ci.

30.3 Cas de IPv4

Nous venons de voir que la fonction interne de réception d'un message fait appel à une fonction de réception de message spécifique à la famille de protocoles. Comme on le voit à propos de `inet_dgram_ops` pour UDP, le nom de la fonction de réception d'un message spécifique à IPv4 est `sock_common_recvmsg()`. Cette fonction est définie dans le fichier `linux/net/core/sock.c` :

Code Linux 2.6.10

```
1268 int sock_common_recvmsg(struct kiocb *iocb, struct socket *sock,
1269                         struct msghdr *msg, size_t size, int flags)
1270 {
1271         struct sock *sk = sock->sk;
1272         int addr_len = 0;
1273         int err;
1274
1275         err = sk->sk_prot->recvmsg(iocb, sk, msg, size, flags & MSG_DONTWAIT,
1276                                    flags & ~MSG_DONTWAIT, &addr_len);
1277         if (err >= 0)
1278                 msg->msg_namelen = addr_len;
1279         return err;
1280 }
```

Autrement dit :

- on déclare un descripteur de couche transport, que l'on instantie avec celui associé au descripteur de socket passé en argument ;
- on déclare une longueur d'adresse, que l'on initialise à zéro ;
- on fait appel à la fonction spécifique au protocole de transport ;
- si celle-ci se déroule sans problème, on met à jour la longueur de l'adresse dans l'en-tête de message passé en argument ;
- on renvoie le code fourni par cette fonction spécifique.

30.4 Cas de UDP

Code Linux 2.6.10

La fonction spécifique de réception d'un message dans le cas de UDP, comme le montre `udp_prot`, est `udp_recvmsg()`. Celle-ci est définie dans le fichier `linux/net/ipv4/udp.c` :

```
768 /*
769  *      Ceci devrait etre facile : s'il y a quelque chose le renvoyer,
770  *      sinon nous bloquons.
771  */
772
773 int udp_recvmsg(struct kiocb *iocb, struct sock *sk, struct msghdr *msg,
774                 size_t len, int noblock, int flags, int *addr_len)
775 {
776         struct inet_opt *inet = inet_sk(sk);
777         struct sockaddr_in *sin = (struct sockaddr_in *)msg->msg_name;
778         struct sk_buff *skb;
779         int copied, err;
780
781         /*
782          *      Verifier toutes les adresses passees
783          */
784         if (addr_len)
785                 *addr_len=sizeof(*sin);
786
787         if (flags & MSG_ERRQUEUE)
788                 return ip_recv_error(sk, msg, len);
789
790 try_again:
791         skb = skb_recv_datagram(sk, flags, noblock, &err);
792         if (!skb)
793                 goto out;
794
795         copied = skb->len - sizeof(struct udphdr);
796         if (copied > len) {
797                 copied = len;
798                 msg->msg_flags |= MSG_TRUNC;
799         }
800
801         if (skb->ip_summed==CHECKSUM_UNNECESSARY) {
802                 err = skb_copy_datagram_iovec(skb, sizeof(struct udphdr), msg->msg_iov,
803                                               copied);
804         } else if (msg->msg_flags&MSG_TRUNC) {
805                 if (__udp_checksum_complete(skb))
806                         goto csum_copy_err;
807                 err = skb_copy_datagram_iovec(skb, sizeof(struct udphdr), msg->msg_iov,
808                                               copied);
809         } else {
810                 err = skb_copy_and_csum_datagram_iovec(skb, sizeof(struct udphdr),
                                                       msg->msg_iov);
```

```
811
812                 if (err == -EINVAL)
813                         goto csum_copy_err;
814         }
815
816         if (err)
817                 goto out_free;
818
819         sock_recv_timestamp(msg, sk, skb);
820
821         /* Copier les adresses. */
822         if (sin)
823         {
824                 sin->sin_family = AF_INET;
825                 sin->sin_port = skb->h.uh->source;
826                 sin->sin_addr.s_addr = skb->nh.iph->saddr;
827                 memset(sin->sin_zero, 0, sizeof(sin->sin_zero));
828         }
829         if (inet->cmsg_flags)
830                 ip_cmsg_recv(msg, skb);
831
832         err = copied;
833         if (flags & MSG_TRUNC)
834                 err = skb->len - sizeof(struct udphdr);
835
836 out_free:
837         skb_free_datagram(sk, skb);
838 out:
839         return err;
840
841 csum_copy_err:
842         UDP_INC_STATS_BH(UDP_MIB_INERRORS);
843
844         /* Vider la file d'attente. */
845         if (flags&MSG_PEEK) {
846                 int clear = 0;
847                 spin_lock_irq(&sk->sk_receive_queue.lock);
848                 if (skb == skb_peek(&sk->sk_receive_queue)) {
849                         __skb_unlink(skb, &sk->sk_receive_queue);
850                         clear = 1;
851                 }
852                 spin_unlock_irq(&sk->sk_receive_queue.lock);
853                 if (clear)
854                         kfree_skb(skb);
855         }
856
857         skb_free_datagram(sk, skb);
858
859         if (noblock)
860                 return -EAGAIN;
861         goto try_again;
862 }
```

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 Internet, que l'on instantie avec celle contenue dans l'en-tête de message passé en argument.
- On déclare un descripteur de tampon de socket, une variable pour décompter le nombre d'octets copiés et un code de retour.

- Si la longueur d'adresse passée en argument est nulle, on la positionne à celle d'une adresse Internet.
- Si le vecteur des drapeaux spécifie que cela concerne les erreurs intervenues dans la couche IP, on fait appel à la fonction `ip_recv_error()`, que nous devrions étudier à propos d'ICMP.
- On essaie d'instantier le tampon de socket grâce à la fonction `skb_recv_datagram()`, étudiée dans la section suivante. Si on n'y parvient pas, on renvoie le code fourni par cette fonction.
- On calcule le nombre de caractères qui doivent être lus (copiés). Si ce nombre est supérieur à la taille `len` passée en argument, on tronque à cette valeur et on l'indique dans le vecteur des drapeaux de l'en-tête de message.
- La lecture proprement dite dépend du comportement à l'égard de la somme de contrôle IP :
  - S'il n'est pas nécessaire de s'en préoccuper, on fait appel à la fonction `skb_copy_datagram_iovec()`, que nous étudierons dans la section suivante.
  - Sinon, si les données à lire sont tronquées, on vérifie la somme de contrôle en faisant appel à la fonction `__udp_checksum_complete()` :
    - Si elle ne concorde pas :
      - On incrémente l'information statistique sur le nombre d'erreurs intervenues en entrées UDP.
      - On vide la file d'attente.
      - On libère le descripteur de socket.
      - Et on renvoie l'opposé du code d'erreur `EAGAIN`.
    - Sinon on fait appel à la fonction `skb_copy_datagram_iovec()`, comme dans le premier cas.
  - Si on doit tenir compte de la somme de contrôle IP et que les données ne sont pas tronquées, on fait appel à la fonction `skb_copy_and_csum_datagram_iovec()`, que nous étudierons dans la section suivante. Si cette fonction renvoie l'opposé du code d'erreur `EINVAL`, on effectue ce qui a été dit ci-dessus lorsque la somme de contrôle ne concorde pas.
- Si l'une des deux fonctions renvoie un code d'erreur, on libère le descripteur de socket et on renvoie le code fourni.
- On indique le moment auquel on a reçu les données en faisant appel à la fonction `sock_recv_timestamp()`, définie dans le fichier `linux/include/net/sock.h` :

Code Linux 2.6.10

```
1246 static __inline__ void
1247 sock_recv_timestamp(struct msghdr *msg, struct sock *sk, struct sk_buff *skb)
1248 {
1249         struct timeval *stamp = &skb->stamp;
1250         if (sk->sk_rcvtstamp) {
1251                 /* Course intervenue entre l'etablissement de l'estampille temporelle
1252                    et la reception du paquet. Remplir avec l'heure exacte actuelle. */
1253                 if (stamp->tv_sec == 0)
1254                         do_gettimeofday(stamp);
1255                 put_cmsg(msg, SOL_SOCKET, SO_TIMESTAMP, sizeof(struct timeval),
1256                          stamp);
1257         } else
1258                 sk->sk_stamp = *stamp;
1259 }
```

dont le code se comprend aisément.
- Si l'adresse est nulle, on renseigne les champs à partir de ceux du descripteur de socket.
- S'il y des données ancillaires, on les récupère grâce à la fonction `ip_cmsg_recv()`, qui ne nous intéressera pas ici.
- On libère le descripteur de socket et on renvoie le nombre de caractères effectivement lus.

30.5 Lien avec les tampons de socket

Nous venons de voir que la fonction `udp_recvmsg()` fait appel aux fonctions :

- `skb_recv_datagram()` d'instantiation de descripteur de tampon de socket en réception ;
- `skb_copy_datagram_iovec()` de copie de datagramme ;
- `skb_copy_and_csum_datagram_iovec()` de copie de datagramme si l'on doit tenir compte de la somme de contrôle ;
- `skb_free_datagram()` pour libérer une socket de datagramme.

Nous allons étudier l'implémentation de ces fonctions maintenant.

30.5.1 Instantiation de descripteur de tampon en réception

30.5.1.1 Fonction générale

La fonction `skb_recv_datagram()` est définie dans le fichier `linux/net/core/datagram.c` : Code Linux 2.6.10

```
116 /**
117  *      skb_recv_datagram - Recoit un skbuff de datagramme
118  *      @sk - socket
119  *      @flags - drapeaux MSG_
120  *      @noblock - operation bloquante ?
121  *      @err - code d'erreur renvoye
122  *
123  *      Recupere un skbuff de datagramme, a savoir le depilement, les reveils non bloquants
124  *      et les concurrences possibles. Ceci remplace le code identique des paquets, bruts et
125  *      udp, aussi bien que IPX AX.25 et Appletalk. Il rectifie egalement de facon definitive
126  *      le long depilement et la concurrence pour les sockets de datagramme. Si vous
127  *      changer cette routine, n'oubliez pas qu'elle doit etre re-entrante.
128  *
129  *      Cette fonction verrouillera la socket si un skb est renvoye, aussi l'appelant
130  *      doit-il deverrouiller la socket dans ce cas (normalement en appelant
131  *      skb_free_datagram)
132  *
133  *      * Elle ne verrouille plus la socket depuis aujourd'hui. Cette fonction n'a
134  *      * plus a s'occuper des problemes de concurrence. Cette mesure doit/peut ameliorer
135  *      * significativement les latences des sockets de datagramme a haute charge,
136  *      * lorsque la copie des donnees dans l'espace utilisateur prend beaucoup de temps.
137  *      * (BTW J'ai juste tue le dernier cli() dans IP/IPv6/core/netlink/packet
138  *      *  8) Gros gain.)
139  *      *                                          --ANK (980729)
140  *
141  *      L'ordre des tests lorsque nous ne trouvons pas de donnees en attente est specifie
142  *      presque explicitment par POSIX 1003.1g. Ne pas le changer sans consulter
143  *      le standard s'il vous plait.
144  */
145 struct sk_buff *skb_recv_datagram(struct sock *sk, unsigned flags,
146                                   int noblock, int *err)
147 {
148         struct sk_buff *skb;
```

```
149         long timeo;
150         /*
151          * Il est permis que l'appelant ne verifie pas sk->sk_err avant skb_recv_datagram()
152          */
153         int error = sock_error(sk);
154
155         if (error)
156                 goto no_packet;
157
158         timeo = sock_rcvtimeo(sk, noblock);
159
160         do {
161                 /* Une fois encore seul le code au niveau utilisateur appelle cette fonction,
162                  * donc rien du niveau interruption ne mangera soudainenement la
                    receive_queue.
163                  *
164                  * Regarder quand meme le client nfs en cours...
165                  * Cependant cette fonction etait correcte dans tous les cas. 8)
166                  */
167                 if (flags & MSG_PEEK) {
168                         unsigned long cpu_flags;
169
170                         spin_lock_irqsave(&sk->sk_receive_queue.lock,
171                                           cpu_flags);
172                         skb = skb_peek(&sk->sk_receive_queue);
173                         if (skb)
174                                 atomic_inc(&skb->users);
175                         spin_unlock_irqrestore(&sk->sk_receive_queue.lock,
176                                                cpu_flags);
177                 } else
178                         skb = skb_dequeue(&sk->sk_receive_queue);
179
180                 if (skb)
181                         return skb;
182
183                 /* L'utilisateur ne veut pas attendre */
184                 error = -EAGAIN;
185                 if (!timeo)
186                         goto no_packet;
187
188         } while (!wait_for_packet(sk, err, &timeo));
189
190         return NULL;
191
192 no_packet:
193         *err = error;
194         return NULL;
195 }
```

Remarquons que le paramètre `sk` est un descripteur de couche transport et non un descripteur de socket comme pourrait le laisser entendre le commentaire. Commentons maintenant le code source :

- On déclare un descripteur de tampon de socket et un délai.
- On vérifie si le champ erreur du descripteur de couche de transport passé en argument est positionné. Si c'est le cas, on positionne le code d'erreur de retour (passé en argument) et on renvoie `NULL`.

Code Linux 2.6.10

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

```
1125 /*
1126  *      Recupere un rapport d'erreur et l'efface atomiquement
```

```
1127  */
1128
1129 static inline int sock_error(struct sock *sk)
1130 {
1131         int err = xchg(&sk->sk_err, 0);
1132         return -err;
1133 }
```

– On récupère le délai indiqué dans le descripteur de couche transport.
La fonction `sock_rcvtimeo()` est définie dans le fichier `linux/include/net/sock.h`: Code Linux 2.6.10

```
1223 static inline long sock_rcvtimeo(const struct sock *sk, int noblock)
1224 {
1225         return noblock ? 0 : sk->sk_rcvtimeo;
1226 }
```

– S'il faut récupérer les données sans les enlever de la file d'attente (cas de `MSG_PEEK`) :
  – on verrouille la file d'attente en réception ;
  – on fait appel à la fonction `skb_peek()` ;
  – si on obtient un tampon de socket grâce à cette fonction, on incrémente le nombre d'utilisateurs de celui-ci ;
  – on déverrouille la file d'attente en réception.
– Sinon on fait appel à la fonction `skb_dequeue()`.
– Si on obtient un tampon de socket grâce à l'une de ces deux méthodes, on renvoie son adresse. Sinon, si le délai est dépassé, on positionne le code d'erreur à l'opposé de `EAGAIN` et on renvoie `NULL`. Si le délai n'est pas dépassé, on recommence jusqu'à ce qu'un paquet arrive.
La fonction auxiliaire `wait_for_packet()` est étudiée ci-dessous.

30.5.1.2 Attente d'un paquet

La fonction `wait_for_packet()` est définie dans le fichier `linux/net/core/datagram.c`: Code Linux 2.6.10

```
66 /*
67  * Attend un paquet...
68  */
69 static int wait_for_packet(struct sock *sk, int *err, long *timeo_p)
70 {
71         int error;
72         DEFINE_WAIT(wait);
73
74         prepare_to_wait_exclusive(sk->sk_sleep, &wait, TASK_INTERRUPTIBLE);
75
76         /* Erreurs de socket */
77         error = sock_error(sk);
78         if (error)
79                 goto out_err;
80
81         if (!skb_queue_empty(&sk->sk_receive_queue))
82                 goto out;
83
84         /* Socket fermee ? */
85         if (sk->sk_shutdown & RCV_SHUTDOWN)
86                 goto out_noerr;
87
88         /* Les paquets en sequence peuvent venir deconnecter.
```

```
89          * S'il en est ainsi, on fait un rapport sur le probleme
90          */
91         error = -ENOTCONN;
92         if (connection_based(sk) &&
93             !(sk->sk_state == TCP_ESTABLISHED || sk->sk_state == TCP_LISTEN))
94                 goto out_err;
95
96         /* signaux de manipulation */
97         if (signal_pending(current))
98                 goto interrupted;
99
100        error = 0;
101        *timeo_p = schedule_timeout(*timeo_p);
102 out:
103        finish_wait(sk->sk_sleep, &wait);
104        return error;
105 interrupted:
106        error = sock_intr_errno(*timeo_p);
107 out_err:
108        *err = error;
109        goto out;
110 out_noerr:
111        *err = 0;
112        error = 1;
113        goto out;
114 }
```

Autrement dit :

– On définit une file d'attente de nom `wait`.

Code Linux 2.6.10

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

```
325 #define DEFINE_WAIT(name)                                               \
326         wait_queue_t name = {                                           \
327                 .task           = current,                              \
328                 .func           = autoremove_wake_function,             \
329                 .task_list      = {     .next = &(name).task_list,      \
330                                         .prev = &(name).task_list,      \
331                                 },                                      \
332         }
```

– On se prépare à attendre.

Code Linux 2.6.10

La fonction générale `prepare_to_wait_exclusive()` est définie dans le fichier `linux/kernel/wait.c` :

```
78 void fastcall
79 prepare_to_wait_exclusive(wait_queue_head_t *q, wait_queue_t *wait, int state)
80 {
81         unsigned long flags;
82
83         wait->flags |= WQ_FLAG_EXCLUSIVE;
84         spin_lock_irqsave(&q->lock, flags);
85         if (list_empty(&wait->task_list))
86                 __add_wait_queue_tail(q, wait);
87         /*
88          * Ne pas alterer l'etat de la tache si ceci est juste aller a la
89          * file d'attente sur un appel de file d'attente async
90          */
91         if (is_sync_wait(wait))
92                 set_current_state(state);
93         spin_unlock_irqrestore(&q->lock, flags);
94 }
```

- Si le descripteur de couche transport laisse apparaître une erreur, on positionne celle-ci (en argument), on indique qu'on n'attend plus et on renvoie le code d'erreur.
- Si la file d'attente est non vide, on indique qu'on n'attend plus et on renvoie 0.
- Si la socket était en train de s'arrêter, on positionne le code d'erreur (en argument) à 0, on indique qu'on n'attend plus et on renvoie 1.
- Si la socket est orientée connexion (ce qui n'est pas le cas de UDP) et si elle est déconnectée, on positionne le code d'erreur à l'opposé de `ENOTCOMM`, on indique qu'on n'attend plus et on renvoie le code d'erreur.

  La fonction en ligne `connection_based()` est définie dans le fichier `linux/net/core/-datagram.c`:

Code Linux 2.6.10

```
58 /*
59  *      Une socket est-elle 'orientee connexion' ?
60  */
61 static inline int connection_based(struct sock *sk)
62 {
63         return sk->sk_type == SOCK_SEQPACKET || sk->sk_type == SOCK_STREAM;
64 }
```

- Si un signal nous prévient qu'il faut interrompre l'attente, on positionne le code d'erreur à celui fourni par la fonction `sock_intr_errno()`, on indique qu'on n'attend plus et on renvoie ce code.

  La fonction en ligne `sock_intr_errno()` est définie dans le fichier `linux/include/-net/sock.h`:

Code Linux 2.6.10

```
1238 /* Helas, avec le delai, les operations socket ne sont pas redemarrables.
1239  * Comparer ceci a poll().
1240  */
1241 static inline int sock_intr_errno(long timeo)
1242 {
1243         return timeo == MAX_SCHEDULE_TIMEOUT ? -ERESTARTSYS : -EINTR;
1244 }
```

- Sinon, on attend que le délai se soit écoulé, on indique qu'on n'attend plus et on renvoie 0.

30.5.2 Copie d'un datagramme dans l'espace utilisateur

30.5.2.1 Sans tenir compte de la somme de contrôle

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

Code Linux 2.6.10

```
215 /**
216  *      skb_copy_datagram_iovec - Copie un datagramme vers un iovec.
217  *      @skb - tampon a  copier
218  *      @offset - decalage dans le tampon a partir duquel debute la copie
219  *      @iovec - vecteur d'entree-sortie vers lequel copier
220  *      @len - quantite de donnees a copier du tampn vers iovec
221  *
222  *      Note : le iovec est modifie durant la copie.
223  */
224 int skb_copy_datagram_iovec(const struct sk_buff *skb, int offset,
225                             struct iovec *to, int len)
226 {
227         int start = skb_headlen(skb);
228         int i, copy = start - offset;
229
230         /* Copie l'en-tete. */
```

```
231         if (copy > 0) {
232                 if (copy > len)
233                         copy = len;
234                 if (memcpy_toiovec(to, skb->data + offset, copy))
235                         goto fault;
236                 if ((len -= copy) == 0)
237                         return 0;
238                 offset += copy;
239         }
240 
241         /* Copie l'appendice en page. Hum... pourquoi ceci semble-t-il si complique ? */
242         for (i = 0; i < skb_shinfo(skb)->nr_frags; i++) {
243                 int end;
244 
245                 BUG_TRAP(start <= offset + len);
246 
247                 end = start + skb_shinfo(skb)->frags[i].size;
248                 if ((copy = end - offset) > 0) {
249                         int err;
250                         u8  *vaddr;
251                         skb_frag_t *frag = &skb_shinfo(skb)->frags[i];
252                         struct page *page = frag->page;
253 
254                         if (copy > len)
255                                 copy = len;
256                         vaddr = kmap(page);
257                         err = memcpy_toiovec(to, vaddr + frag->page_offset +
258                                              offset - start, copy);
259                         kunmap(page);
260                         if (err)
261                                 goto fault;
262                         if (!(len -= copy))
263                                 return 0;
264                         offset += copy;
265                 }
266                 start = end;
267         }
268 
269         if (skb_shinfo(skb)->frag_list) {
270                 struct sk_buff *list = skb_shinfo(skb)->frag_list;
271 
272                 for (; list; list = list->next) {
273                         int end;
274 
275                         BUG_TRAP(start <= offset + len);
276 
277                         end = start + list->len;
278                         if ((copy = end - offset) > 0) {
279                                 if (copy > len)
280                                         copy = len;
281                                 if (skb_copy_datagram_iovec(list,
282                                                             offset - start,
283                                                             to, copy))
284                                         goto fault;
285                                 if ((len -= copy) == 0)
286                                         return 0;
287                                 offset += copy;
288                         }
289                         start = end;
290                 }
291         }
292         if (!len)
```

```
293                 return 0;
294
295 fault:
296         return -EFAULT;
297 }
```

Autrement dit :

- On se place au début des données du tampon spécifié par le descripteur de tampon de socket passé en argument.
- On calcule le nombre `copy` d'octets à copier.
- On copie l'en-tête.

  La fonction `memcpy_toiovec()` est définie dans le fichier `linux/net/core/iovec.c` : Code Linux 2.6.10

```
77 /*
78  *      Copie depuis le noyau vers un iovec. Renvoie -EFAULT en cas d'erreur.
79  *
80  *      Note : ceci modifie le iovec originel.
81  */
82
83 int memcpy_toiovec(struct iovec *iov, unsigned char *kdata, int len)
84 {
85         while (len > 0) {
86                 if (iov->iov_len) {
87                         int copy = min_t(unsigned int, iov->iov_len, len);
88                         if (copy_to_user(iov->iov_base, kdata, copy))
89                                 return -EFAULT;
90                         kdata += copy;
91                         len -= copy;
92                         iov->iov_len -= copy;
93                         iov->iov_base += copy;
94                 }
95                 iov++;
96         }
97
98         return 0;
99 }
```

- On copie les pages.
- On copie les listes.
- On renvoie 0 si tout s'est bien passé ou l'opposé du code d'erreur `EFAULT` sinon.

30.5.2.2 Copie de datagramme en tenant compte de la somme de contrôle

La fonction `skb_copy_and_csum_datagram_iovec()` est définie dans le fichier `linux/net/core/datagram.c` : Code Linux 2.6.10

```
390 /**
391  *      skb_copy_and_csum_datagram_iovec - Copie en verifiant la somme de controle
                                               un skb vers un iovec utilisateur.
392  *      @skb - skbuff
393  *      @hlen - longueur physique
394  *      @iovec - vecteur d'entree-sortie
395  *
396  *      L'appelant _doit_ verifier que le skb sera adapte a cet iovec.
397  *
398  *      Renvoie : 0       - en cas de succes.
399  *                -EINVAL - echec du controle de la somme de controle.
400  *                -EFAULT - faute durant la copie. Faites attentions que dans ce cas
401  *                          iovec peut etre modifie !
```

```
402  */
403 int skb_copy_and_csum_datagram_iovec(const struct sk_buff *skb,
404                                      int hlen, struct iovec *iov)
405 {
406         unsigned int csum;
407         int chunk = skb->len - hlen;
408
409         /* Saute les elements remplis.
410          * Elegant, voir aussi memcpy_toiovec 8)
411          */
412         while (!iov->iov_len)
413                 iov++;
414
415         if (iov->iov_len < chunk) {
416                 if ((unsigned short)csum_fold(skb_checksum(skb, 0, chunk + hlen,
417                                                            skb->csum)))
418                         goto csum_error;
419                 if (skb_copy_datagram_iovec(skb, hlen, iov, chunk))
420                         goto fault;
421         } else {
422                 csum = csum_partial(skb->data, hlen, skb->csum);
423                 if (skb_copy_and_csum_datagram(skb, hlen, iov->iov_base,
424                                                chunk, &csum))
425                         goto fault;
426                 if ((unsigned short)csum_fold(csum))
427                         goto csum_error;
428                 iov->iov_len -= chunk;
429                 iov->iov_base += chunk;
430         }
431         return 0;
432 csum_error:
433         return -EINVAL;
434 fault:
435         return -EFAULT;
436 }
```

Autrement dit :

- On calcule la taille des données à copier.
- On recherche le premier élément non vide du vecteur d'entrée-sortie.
- Si la taille de l'élément de vecteur d'entrée-sortie est inférieure au nombre d'octets à copier :
  - On vérifie la somme de contrôle. Si elle ne concorde pas, on renvoie l'opposé du code d'erreur `EINVAL`.
  - On fait appel à la fonction `skb_copy_datagram_iovec()` étudiée dans la sous-section ci-dessus pour la copie proprement dite. Si une erreur intervient, on renvoie l'opposé du code d'erreur `EFAULT`.
- Sinon :
  - On calcule la somme de contrôle partielle.
  - On fait appel à la fonction elle-même (récursivité) pour essayer de copier en vérifiant la somme de contrôle. Si une erreur intervient, on renvoie l'opposé du code d'erreur `EFAULT`.
  - On vérifie la somme de contrôle. Si elle ne concorde pas, on renvoie l'opposé du code d'erreur `EINVAL`.
  - On met à jour l'adresse de base et la longueur de l'élément de vecteur d'entrée-sortie.
- Dans les deux cas, si tout s'est bien passé, on renvoie 0.

30.5.3 Libération d'un tampon de datagramme

La fonction `skb_free_datagram()` est définie dans le fichier `linux/net/core/datagram.c`: Code Linux 2.6.10

```
197 void skb_free_datagram(struct sock *sk, struct sk_buff *skb)
198 {
199         kfree_skb(skb);
200 }
```