Cinquième partie

Envoi

Chapitre 31

Envoi d'un datagramme par une socket

Ce chapitre est le pendant du chapitre 30, traitant l'envoi des données et non plus leur réception. Son but est d'étudier l'implémentation de l'envoi de données par une socket dans le cas d'un type de communication non connecté, ce qui correspond à un datagramme et donc à UDP dans le cas de TCP/IP.

Nous avons vu au chapitre 6 que l'envoi de données de façon non connectée sur une socket s'effectue grâce à la fonction :

```
int sendto(int s, const void *buf, int len, unsigned int flags, const struct sockaddr *to,
           int tolen);
```

possèdant six arguments :

- le numéro de socket sur laquelle il faut écrire ;
- l'adresse du tampon d'écriture ;
- la longueur du tampon d'écriture ;
- les drapeaux d'option ;
- un pointeur à une adresse générique de socket, indiquant où doit être envoyé le datagramme ;
- la longueur de cette adresse.

Nous allons étudier l'implémentation de cette fonction dans ce chapitre.

31.1 Niveau implémentation des sockets

31.1.1 Fonction d'appel

Code Linux 2.6.10

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

```
1499 /*
1500  *      Envoie un datagramme a une adresse donnee. Nous deplacons l'adresse dans l'espace
1501  *      noyau et nous verifions que l'on peut lire la zone des donnees utilisateur
1502  *      avant de faire appel au protocole.
1503  */
1504
1505 asmlinkage long sys_sendto(int fd, void __user * buff, size_t len, unsigned flags,
1506                            struct sockaddr __user *addr, int addr_len)
1507 {
1508         struct socket *sock;
1509         char address[MAX_SOCK_ADDR];
1510         int err;
1511         struct msghdr msg;
1512         struct iovec iov;
1513
1514         sock = sockfd_lookup(fd, &err);
1515         if (!sock)
1516                 goto out;
1517         iov.iov_base=buff;
1518         iov.iov_len=len;
1519         msg.msg_name=NULL;
1520         msg.msg_iov=&iov;
1521         msg.msg_iovlen=1;
1522         msg.msg_control=NULL;
1523         msg.msg_controllen=0;
1524         msg.msg_namelen=0;
1525         if(addr)
1526         {
1527                 err = move_addr_to_kernel(addr, addr_len, address);
1528                 if (err < 0)
1529                         goto out_put;
1530                 msg.msg_name=address;
1531                 msg.msg_namelen=addr_len;
1532         }
1533         if (sock->file->f_flags & O_NONBLOCK)
1534                 flags |= MSG_DONTWAIT;
1535         msg.msg_flags = flags;
1536         err = sock_sendmsg(sock, &msg, len);
1537
1538 out_put:
1539         sockfd_put(sock);
1540 out:
1541         return err;
1542 }
```

Autrement dit :

- On déclare un descripteur de socket, une adresse, un code de retour, un en-tête de message et un élément de vecteur d'entrée-sortie.
- 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()`.

  Rappelons que si nous avons un numéro de fichier de socket valide, son descripteur de socket a été instantié de la façon vue au chapitre 28 grâce à la fonction `sock_create()`.

- On initialise le vecteur d'entrée-sortie avec l'adresse du tampon de mémoire vive et la taille passées en argument.
- On initialise l'en-tête de message : pas d'adresse (et donc de longueur nulle) dans une première étape, un seul élément de vecteur d'entrée-sortie (celui initialisé ci-dessus) puisque les données ne sont pas dispersées, pas de données ancillaires.
- Si une adresse est transmise en argument, on essaie de la rapatrier dans l'espace noyau. Si on n'y parvient pas, on libère le descripteur de socket et on renvoie le code fourni par la fonction `move_addr_to_kernel()`. Si on y parvient, ce que l'on espère, on renseigne les deux champs consacrés à l'adresse de l'en-tête de message avec cette adresse.
- Si le champ drapeaux de la socket spécifie que l'écriture ne doit pas être bloquante, on le répercute sur le champ vecteur de drapeaux de l'en-tête de message.
- On fait appel à la fonction `sock_sendmsg()` étudiée ci-dessous, on libère le descripteur de socket et on renvoie le code de retour qu'elle a fourni.

31.1.2 Envoi de message de socket

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

```
548 int sock_sendmsg(struct socket *sock, struct msghdr *msg, size_t size)
549 {
550         struct kiocb iocb;
551         struct sock_iocb siocb;
552         int ret;
553
554         init_sync_kiocb(&iocb, NULL);
555         iocb.private = &siocb;
556         ret = __sock_sendmsg(&iocb, sock, msg, size);
557         if (-EIOCBQUEUED == ret)
558                 ret = wait_on_sync_kiocb(&iocb);
559         return ret;
560 }
```

Autrement dit :

- On déclare un bloc de contrôle d'entrée-sortie noyau, un bloc de contrôle d'entrée-sortie de socket et un code de retour.
- On initialise le bloc de contrôle d'entrée-sortie noyau.
- On renseigne le champ partie privée de ce bloc d'entrée-sortie noyau avec l'adresse du bloc de contrôle d'entrée-sortie de socket, de façon à ce que celui-ci ne fasse pas référence sur rien.
- On fait appel à la fonction interne d'envoi de message `__sock_sendmsg()`, étudiée ci-dessous.
- Si le code de retour de cette dernière fonction est `-EIOCBQUEUED`, on attend la fin de la synchronisation.
- On renvoie le code fourni par l'une ou l'autre des fonctions appelées.

31.1.3 Fonction interne d'envoi de message

La fonction `__sock_sendmsg()` est définie, elle aussi, dans le même fichier : Code Linux 2.6.10

```
530 static inline int __sock_sendmsg(struct kiocb *iocb, struct socket *sock,
531                                    struct msghdr *msg, size_t size)
532 {
533         struct sock_iocb *si = kiocb_to_siocb(iocb);
534         int err;
```

```
535
536         si->sock = sock;
537         si->scm = NULL;
538         si->msg = msg;
539         si->size = size;
540
541         err = security_socket_sendmsg(sock, msg, size);
542         if (err)
543                 return err;
544
545         return sock->ops->sendmsg(iocb, sock, msg, size);
546 }
```

Autrement dit :

- On déclare une adresse de bloc de contrôle d'entrée-sortie de socket, que l'on initialise avec celle de la partie privée du bloc de contrôle d'entrée-sortie noyau passé en argument.
- On renseigne les champs de cette entité avec les entités passées en argument, le cafteur étant initialisé à zéro.
- On fait appel à la fonction de sécurité `security_socket_sendmsg()` qui ne nous intéresse pas dans cet ouvrage. Si les droits d'accès ne sont pas accordés, on renvoie le code d'erreur fourni par cette fonction.
- On fait appel à la fonction d'envoi de message spécifique à la famille de protocoles utilisée pour l'écriture proprement dite et on renvoie le code fourni par celle-ci.

  Rappelons que le descripteur de socket passé en argument provient d'un des arguments de la fonction `sock_sendmsg()`, qui lui-même est associé au numéro de fichier passé en argument dans la fonction `sys_sendto()`. Celui-ci a été instantié dans la fonction `sock_create()` étudiée au chapitre 28. Ses champs ont été renseigné en faisant appel à la fonction de création spécifique à la famille de protocoles. Dans le cas de IPv4, la fonction spécifique `inet_create()` renseigne le champ `ops` suivant le type de communication.

31.2 Cas de IPv4

Dans le cas IPv4/UDP, l'ensemble des opérations associé à l'instance de `struct inet_protosw` s'appelle `inet_dgram_ops`. La fonction spécifique d'envoi d'un message s'appelle `inet_sendmsg()`. En fait c'est la même dans le cas de IPv4/TCP, ce qui montre que cette fonction concerne IPv4.

31.2.1 Appel à la fonction spécifique à la couche de transport

Code Linux 2.6.10

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

```
649 int inet_sendmsg(struct kiocb *iocb, struct socket *sock, struct msghdr *msg,
650                 size_t size)
651 {
652         struct sock *sk = sock->sk;
653
654         /* Nous pouvons avoir besoin de lier la socket. */
655         if (!inet_sk(sk)->num && inet_autobind(sk))
656                 return -EAGAIN;
657
658         return sk->sk_prot->sendmsg(iocb, sk, msg, size);
659 }
```

Autrement dit :

- On charge le descripteur de couche transport associé au descripteur de socket passé en argument.

  Rappelons que la fonction `inet_create()` a instantié ce descripteur de couche transport et renseigné ses champs, en particulier le champ `prot` avec `udp_prot` dans le cas de UDP mais pas de numéro de port.
- Si ce descripteur de couche de transport ne spécifie pas de numéro de port, on essaie de lui en attribuer un automatiquement (grâce à la fonction `inet_autobind()` étudiée ci-dessous). Si on n'y parvient pas, on renvoie l'opposé du code d'erreur `EAGAIN`.
- On fait appel à la fonction d'envoi de message spécifique au protocole de la couche de transport (TCP ou UDP).

  Dans le cas de UDP, l'instance `udp_prot` montre qu'il s'agit de la fonction `udp_sendmsg()`, que nous étudierons au chapitre 32.

31.2.2 Attribution automatique de numéro de port

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

```
170 /*
171  *      Lie automatiquement une socket non liee.
172  */
173
174 static int inet_autobind(struct sock *sk)
175 {
176         struct inet_opt *inet;
177         /* Nous pouvons avoir besoin de lier la socket. */
178         lock_sock(sk);
179         inet = inet_sk(sk);
180         if (!inet->num) {
181                 if (sk->sk_prot->get_port(sk, 0)) {
182                         release_sock(sk);
183                         return -EAGAIN;
184                 }
185                 inet->sport = htons(inet->num);
186         }
187         release_sock(sk);
188         return 0;
189 }
```

Autrement dit :

- On déclare une option Internet.
- On verrouille le descripteur de couche transport passé en argument.
- On instantie l'option Internet avec celle associé au descripteur de couche transport passé en argument.
- Si le champ numéro de port de cette option Internet n'est pas renseigné, on essaie d'obtenir un numéro de port libre grâce à la fonction `get_port()` spécifique au protocole de transport. Si on n'y parvient pas, on libère le descripteur de couche transport et on renvoie l'opposé du code d'erreur `EAGAIN`. Sinon, on renseigne le champ port source `sport` de l'option Internet avec le numéro que l'on vient d'obtenir, on libère le descripteur de couche transport et on renvoie 0.

  Dans le cas d'un envoi, le numéro n'a pas été renseigné. Rappelons que la fonction spécifique pour UDP a déjà été étudiée au chapitre 30.