Programmation réseau 7

Juliusz Chroboczek

5 mars 2019

1 Entrées-sorties vectorisées

Les appels système `read` et `write` (pour les entrées-sorties par flots) et `sendto` et `recvfrom` (pour les entrées-sorties par paquets) prennent en paramètre des tampons qui sont contigus en mémoire. Il arrive parfois que les données ne soient pas contigües : par exemple, lors de l'écriture d'une réponse HTTP, il faut écrire les entêtes puis les données d'application, et ces deux parties ne sont pas forcément contigües.

En apparence, il existe deux façons de contourner ce problème : en faisant plusieurs appels systèmes, ou en recopiant les données dans un tampon contigü. Cependant, en UDP, il n'est pas possible de faire plusieurs appels système, car UDP préserve les frontières de messages — faire deux petits `sendto` n'est pas équivalent à faire un seul `sendto` avec les données concaténées. Même en TCP, il n'est généralement pas désirable de faire plusieurs appels à `write`, ce qui prend du temps et interagit de façon indésirable avec l'algorithme de Nagle (voir poly 3).

La technique généralement employée consiste donc à recopierg les données à envoyer dans un tampon contigu. Le temps nécessaire pour faire cette copie est généralement négligeable par rapport au reste des activités d'un pair, sauf dans le cas d'un serveur qui ne fait pratiquement rien d'autre qu'envoyer les données sur le réseau, par exemple un serveur de *streaming* audio ou vidéo.

1.1 Entrées-sorties vectorisées par flots

L'appel système `writev` permet d'envoyer des données non contigües comme si elles avaient été concaténées. Il prend en paramètre un tableau de `struct iovec` qui décrivent les données à envoyer :

```
struct iovec {
    void  *iov_base;
    size_t iov_len;
};

ssize_t writev(int fd, const struct iovec *iov, int iovcnt);
```

Une exécution de `writev` est équivalente à une exécution de `write`, où les données passées à `write` consisteraient de la concaténation des `iovcnt` tampons décrits par les entrées du tableau

iov ; writev retourne le nombre total d'octets écrits. Le dual de writev est readv, qui remplit les iovcnt tampons passés en paramètre dans l'ordre :

```
ssize_t readv(int fd, const struct iovec *iov, int iovcnt);
```

1.2 Entrées-sorties vectorisées par paquets

L'équivalent vectorisé de sendto s'appelle sendmsg, et prend ses paramètre encapsulés dans une struct msghdr :

```
struct msghdr {
    void         *msg_name;
    socklen_t     msg_namelen;
    struct iovec *msg_iov;
    size_t        msg_iovlen;
    void         *msg_control;
    size_t        msg_controllen;
    /* autres champs omis */
};

ssize_t sendmsg(int sockfd, const struct msghdr *msg, int flags);
```

Les champs msg_name et msg_namelen sont mal nommés : ils indiquent l'adresse destination, et correspondent aux deux derniers paramètres de sendto. Les champs msg_iov et msg_iovlen contiennent la liste de tampons d'émission. Nous verrons l'utilité des champs msg_control et msg_controllen au paragraphe suivant.

Le dual de sendmsg s'appelle recvmsg :

```
ssize_t recvmsg(int sockfd, struct msghdr *msg, int flags);
```

2 Messages ancillaires

Le modèle d'entrées-sorties Unix est un modèle de flots d'octets — un fichier, un tube, un tube nommé implémentent tous ce modèle. Comme nous l'avons vu, ce modèle n'est pas suffisant pour décrire la communication UDP : c'est pour cela que nous avons dû utiliser sendto et recvfrom au lieu de write et read.

Il s'avère que sendto et recvfrom eux-mêmes ne sont pas suffisants pour décrire la communcation à travers le réseau, qui recquiert deux fonctionnalités supplémentaires :

- la possibilité de passer des paramètres supplémentaires lors d'une écriture, ou de recevoir des informations supplémentaires lors d'une lecture ;
- la possibilité de recevoir des indications asynchrones, et pas seulement au moment d'une lecture.

Les champs msg_control et msg_controllen de la struct msghdr décrivent un tampon qui contient une suite de *messages ancillaires* (ou *messages de contrôle*) qui indiquent des paramètres supplémentaires passés à sendmsg ou des indications retournées par recvmsg.

Dans la suite de ce paragraphe, nous voyons comment utiliser les messages ancillaires dans un serveur UDP pour recevoir l'adresse destination d'un paquet reçu et fixer l'adresse source d'un paquet émis.

2.1 Réception de l'adresse destination d'un paquet

Comme d'habitude, on commence par créer une *socket* UDP et la lier à un numéro de port :

```
s = socket(PF_INET6, SOCK_DGRAM, 0);
if(s < 0) ...
memset(&sin6, 0, sizeof(sin6));
sin6.sin6_family = AF_INET6;
sin6.sin6_port = htons(4242);
rc = bind(s, (struct sockaddr*)&sin6, sizeof(sin6));
if(rc < 0) ...
```

On demande ensuite au système qu'il nous remette, à chaque réception de paquet, des messages ancillaires de type `IPV6_PKTINFO` :

```
int one = 1;
rc = setsockopt(s, IPPROTO_IPV6, IPV6_RECVPKTINFO,
                &one, sizeof(one));
if(rc < 0) ...
```

Dans la boucle principale du serveur (le `while(1)` habituel), on utilise `recvmsg` avec un tampon de taille suffisante pour le message de contrôle :

```
unsigned char buf[4096];
unsigned char cmsgbuf[CMSG_SPACE(sizeof(struct in6_pktinfo))];
struct iovec iov[1];
struct msghdr msg;
struct sockaddr_in6 addr;

iov[0].iov_base = buf;
iov[0].iov_len = 4096;
memset(&msg, 0, sizeof(msg));

msg.msg_name = &from;
msg.msg_namelen = sizeof(from);
msg.msg_iov = iov;
msg.msg_iovlen = 1;
msg.msg_control = (struct cmsghdr*)cmsgbuf;
msg.msg_controllen = sizeof(cmsgbuf);

rc = recvmsg(flood_socket, &msg, 0);
if(rc < 0) ...
```

Après une réception réussie, le tampon `cmsgbuf` devrait contenir une `struct in6_pktinfo` contenant l'adresse destination du paquet reçu :

```
struct in6_pktinfo {
    struct in6_addr ipi6_addr;
    unsigned int ipi6_ifindex;
};
```

Comme le tampon peut, en général, contenir plusieurs messages ancillaires, on le parcourt à l'aide d'une boucle et on sauvegarde l'adresse dès qu'on la trouve :

```
struct sockaddr_in6 myaddr;
struct cmsghdr *cmsg;
struct in6_pktinfo *info = NULL;

cmsg = CMSG_FIRSTHDR(&msg);
while(cmsg != NULL) {
    if ((cmsg->cmsg_level == IPPROTO_IPV6) &&
        (cmsg->cmsg_type == IPV6_PKTINFO)) {
        info = (struct in6_pktinfo*)CMSG_DATA(cmsg);
        break;
    }
    cmsg = CMSG_NXTHDR(&msg, cmsg);
}

if(info == NULL) {
    /* ce cas ne devrait pas arriver */
    fprintf(stderr, "IPV6_PKTINFO non trouvé\n");
    continue;
}
myaddr = info->ipi6_addr;
```

2.2 Spécification de l'adresse source d'un paquet émis

Maintenant qu'on a obtenu l'adresse destination de la requête, on peut s'en servir lors de l'émission de la réponse. On commence par construire les paramètres de la réponse :

```
iov[0].iov_base = buf;
iov[0].iov_len = buflen;
memset(&msg, 0, sizeof(msg));
msg.msg_name = addr;
msg.msg_namelen = sizeof(addr);
msg.msg_iov = iov;
msg.msg_iovlen = 1;
```

On construit le message ancillaire contenant l'adresse source :

```
struct in6_pktinfo info;
memset(&info, 0, sizeof(info));
info.ipi6_addr = myaddr;

unsigned char cmsgbuf[CMSG_SPACE(sizeof(struct in6_pktinfo))];
memset(cmsgbuf, 0, sizeof(cmsgbuf));
msg.msg_control = cmsgbuf;
msg.msg_controllen = sizeof(cmsgbuf);
cmsg = CMSG_FIRSTHDR(&msg);
cmsg->cmsg_level = IPPROTO_IPV6;
cmsg->cmsg_type = IPV6_PKTINFO;
cmsg->cmsg_len = CMSG_LEN(sizeof(struct in6_pktinfo));
memcpy(CMSG_DATA(cmsg), &info, sizeof(struct in6_pktinfo));
```

Enfin, on envoie la réponse :

```
rc = sendmsg(s, &msg, 0);
if(rc < 0) ...
```

2.3 Alignement des messages ancillaires

Sur certains systèmes, il se peut que les messages ancillaires doivent être alignés à une adresse qui est un multiple de 4 ou de 8. Il existe deux façons d'obtenir cet alignement en C. La plus élémentaires est d'allouer le tampon (le `cmsgbuf` ci-dessus) à l'aide de `malloc` plutôt que sur la pile. Il ne faudra bien entendu pas oublier de le libérer à l'aide de `free`.

L'autre technique consiste à l'allouer au sein d'une union. Au lieu de déclarer

```
unsigned char cmsgbuf[CMSG_SPACE(sizeof(struct in6_pktinfo))];
```

on alloue une union :

```
union {
    struct cmsghdr hdr;
    unsigned char cmsgbuf[CMSG_SPACE(sizeof(struct in6_pktinfo))];
} u;
```

et on utilise `u.cmsgbuf` au lieu de `cmsgbuf` :

```
msg.msg_control = u.cmsgbuf;
```