Chapitre 7

API des sockets : niveau avancé

Dans le premier chapitre consacré à l'API des sockets, nous avons étudié l'essentiel pour concevoir un programme qui fonctionne. Dans ce chapitre nous passons en revue d'autres appels système de cette API. Nous n'en avons pas vraiment besoin dans ce volume mais nous devrons y faire référence dans l'implémentation Linux, y compris pour le seul cas qui nous intéresse, c'est-à-dire l'envoi de datagrammes UDP.

7.1 Données dispersées

7.1.1 Les entrées-sorties sur socket

Nous avons vu que l'on peut utiliser les fonctions `write()` et `read()` sur les sockets dans le cas de communications connectées et les fonctions `sendto()` et `recvfrom()` dans le cas de communications non connectées. En fait il existe cinq types de fonctions, indiquées ci-dessous dans l'ordre de complexité de mise en œuvre pour l'utilisateur :

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 étudier le cas des données dispersées dans cette section. Les ordres de lecture-écriture de données dispersées (*scatter* en anglais) permettent de manipuler des données éparpillées en plusieurs endroits. Nous avons besoin de la notion de vecteur d'entrée-sortie pour travailler avec ces fonctions.

7.1.2 Vecteur d'entrée-sortie

Un **vecteur d'entrée-sortie** est un tableau d'entités du type `struct iovec`:

```
#include <sys/uio.h>

struct iovec
  {
  ptr_t   iov_base;     /* adresse de debut */
  size_t  iov_len;      /* taille en octets */
  };
```

chaque instance spécifiant un tampon de données, grâce à son adresse de base et sa taille (en octets).

7.1.3 Syntaxe des fonctions

Les fonctions d'entrée-sortie de données dispersées :

```
#include >sys/uio.h>

int readv(int fd, const struct iovec *vector, int count);
int writev(int fd, const struct iovec *vector, int count);
```

possèdent trois arguments :

- le numéro de fichier sur lequel lire ou écrire ;
- le vecteur d'entrée-sortie ;
- le nombre d'éléments de ce vecteur.

La valeur de retour est le nombre d'octets lus ou écrits ou -1 en cas d'erreur.

7.1.4 Exemple

Le programme suivant permet d'écrire des données qui sont dispersées en trois endroits différents :

```
/* scatter.c */

#include <sys/uio.h>
#include <string.h>

void main(void)
{
  char part1[] = "Essai d'";
  char part2[] = "ecriture ";
  char part3[] = "dispersee\n";
  struct iovec iov[3];

  iov[0].iov_base = part1;
  iov[0].iov_len = strlen(part1);

  iov[1].iov_base = part2;
  iov[1].iov_len = strlen(part2);

  iov[2].iov_base = part3;
  iov[2].iov_len = strlen(part3);

  writev(1, iov, 3);
}
```

7.2 Données ancillaires

7.2.1 Notion

Supposons que l'un des utilisateurs de votre système UNIX soit chargé du serveur Web mais, que pour des raisons de sécurité, vous ne voulez pas lui confier le mot de passe du super-utilisateur. Il y a un petit problème car UNIX réserve tous les numéros de port inférieurs à 1 024, et donc le port 80, au super-utilisateur.

Heureusement le problème peut être réglé en utilisant les **données ancillaires** (*ancillary data* en anglais, c'est-à-dire relatives aux servantes), qui sont des données supplémentaires ou auxiliaires envoyées en même temps que d'autres données, dites données normales. On les appelle aussi **données auxiliaires** ou **données de contrôle**.

La solution au problème énoncé ci-dessus se trouve dans la réalisation d'un serveur de sockets. Il s'agit d'un serveur local possédant les droits du super-utilisateur (et donc pouvant accéder au port 80). Le client local, par exemple celui qui doit s'occuper du serveur Web, lui envoie un numéro de port. Le serveur vérifie son identité pour savoir s'il est bien accrédité ; si c'est le cas, il crée une socket pour le port demandé et lui envoie le descripteur de fichier associé à celle-ci.

Les données ancillaires doivent donc porter sur au moins deux types de données : les descripteurs de fichier et les identificateurs. On utilise en fait un regroupement d'identificateurs appelé pièces d'identité (*credentials* en anglais). Les pièces d'identités sont fournies par le noyau Linux lui-même, jamais par l'utilisateur. Elles peuvent donc être considérées comme sûres sur un système local (mais pas pour un hôte distant, puisque le noyau distant peut être manipulé). Les données ancillaires ne sont implémentées que dans le cas d'un système local.

7.2.2 Les fonctions de transmission de messages

La dernière forme de fonctions d'entrée-sortie sur socket utilise une nouvelle entité, dite en-tête de message, qui permet la transmission d'une adresse de socket, de données dispersées et de données ancillaires.

7.2.2.1 Les en-têtes de message

Un en-tête de message est une entité du type `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;
  int           msg_flags;
  };
```

comprenant :

- l'adresse de la socket et sa longueur. Ces deux premiers arguments ne sont nécessaires que dans le cas des communications non connectées (UDP par exemple). Puisque le type du champ de l'adresse est `(void *)`, on n'a pas besoin de convertir celle-ci au type `(struct sockaddr *)`.
- l'adresse et la dimension du vecteur des données dispersées.
- le tampon des données ancillaires et sa taille (en octets).
- le vecteur des drapeaux. Ce dernier champ n'est utile que pour la fonction de lecture. Les coordonnées possibles sont les suivantes :
  - `MSG_EOR` (pour *End Of Record*) : la fin de l'enregistrement a été reçue ; utile pour le type de communication `SOCK_SEQPACKET` ;
  - `MSG_TRUNC` : la fin du datagramme a été tronquée car le tampon de réception était trop petit pour le recevoir en entier ;
  - `MSG_CTRUNC` : les données ancillaires (dites aussi de contrôle, rappelons-le) ont été tronquées car le tampon de réception était trop petit pour les recevoir en entier ;
  - `MSG_OOB` : des données urgentes ont été reçues ;
  - `MSG_ERRQUEUE` : aucune donnée n'a été reçue mais une erreur a été renvoyée.

Les données ancillaires doivent répondre à une syntaxe sur laquelle nous reviendrons ci-dessous, après avoir vu les fonctions qui permettent de les transmettre.

7.2.2.2 Les fonctions

Les fonctions de lecture-écriture permettant les données ancillaires sont les suivantes :

```
#include <sys/types.h>
#include <sys/socket.h>

int sendmsg(int s, const struct msghdr *msg, unsigned int flags);
int recvmsg(int s, const struct msghdr *msg, unsigned int flags);
```

possédant trois arguments :

- le numéro du fichier de socket auquel envoyer ou duquel recevoir le message ;

– l'adresse de l'en-tête de message ;
– le vecteur des drapeaux dont nous avons vu les valeurs possibles à propos des données urgentes.

La valeur de retour indique le nombre d'octets envoyés ou reçus et -1 en cas d'erreur.

7.2.3 Structure des données ancillaires

7.2.3.1 Structure générale

Le champ des données ancillaires peut comprendre 0, une, deux ou plusieurs données. Chacune d'entre elles comprend un en-tête de contrôle, éventuellement des octets de remplissage, suivi des données elles-mêmes, elles-mêmes suivies éventuellement d'octets de remplissage, comme l'indique la figure ci-dessous ([GRA–00], p. 436) :

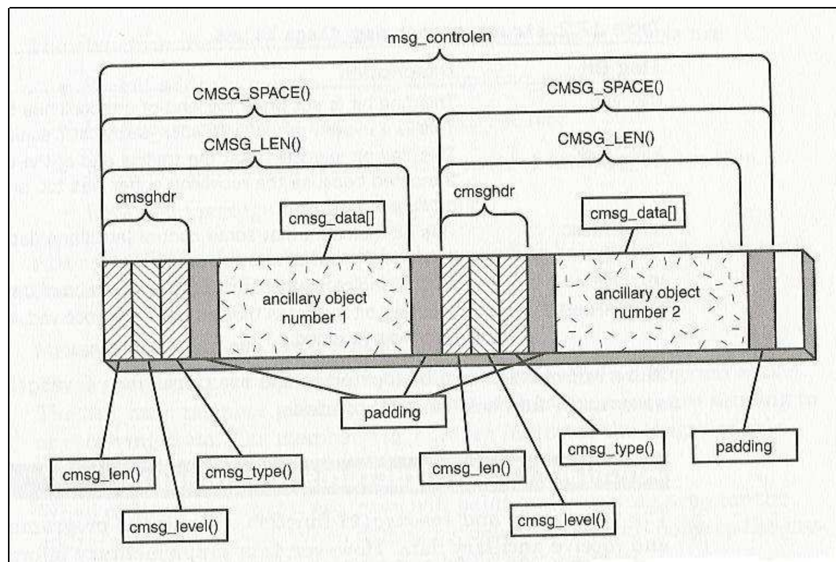

FIG. 7.1 – *Données ancillaires*

7.2.3.2 En-tête de contrôle

L'en-tête de contrôle est une entité du type `struct cmsghdr` (pour *Control MeSsaGe HeaDeR*) :

```
struct cmsghdr
    {
    socklen_t cmsg_len;
    int       cmsg_level;
    int       cmsg_type;
/*  u_char    smsg_data[]; */
    };
```

comprenant trois champs :

– la taille, en nombre d'octets, de la donnée ancillaire, incluant l'en-tête ;
– le niveau de protocole d'origine, seul `SOL_SOCKET` nous intéressant ici ;

– le type de message de contrôle ; les deux types possibles pour `SOL_SOCKET` sont :
  – `SCM_RIGHTS` : la donnée ancillaire est un descripteur de fichier ;
  – `SCM_CREDENTIALS` : la donnée ancillaire est une structure contenant des pièces d'identité.

Le champ `cmsg_data` n'existe pas vraiment ; il est là pour indiquer où se trouvent placées physiquement les données.

On peut obtenir plus d'informations avec l'entrée `cmsg` du manuel en ligne et la [RFC 2292].

7.2.3.3 Les macros concernant les données ancillaires

Du fait de la complexité de la structuration des données ancillaires, un certain nombre de macros C sont fournies pour en faciliter l'utilisation. En voici les protocoles :

```
#include <sys/socket.h>

struct cmsghdr *CMSG_FIRSTHDR(struct msghdr *msgh);
struct cmsghdr *CMSG_NXTHDR(struct msghdr *msgh, struct cmsghdr *cmsg);
size_t CMSG_ALIGN(size_t length);
size_t CMSG_SPACE(size_t length);
size_t CMSG_LEN(size_t length);
void *CMSG_DATA(struct cmsghdr *cmsg);
```

– La macro `CMSG_LEN()` accepte en paramètre la taille de l'objet que l'on veut placer dans le tampon des données ancillaires. Elle renvoie la valeur à placer dans le champ `cmsg_len` de l'en-tête.
– La macro `CMSG_SPACE()` est utilisée pour calculer l'espace total requis par la donnée ancillaire et son en-tête, sans inclure les octets de remplissage de données (par contre ceux concernant l'en-tête sont inclus). Elle est utile pour déterminer la taille du tampon.
– La macro `CMSG_DATA()` prend un pointeur sur un en-tête de message de contrôle en entrée. La valeur de pointeur renvoyée est l'adresse du premier octet de donnée ancillaire qui suit cet en-tête (et éventuellement les octets de remplissage).
– La macro `CMSG_ALIGN()` est une extension de Linux qui ne fait pas partie du standard Posix.1g. Elle prend une longueur en octets en entrée et renvoie une longueur qui tient compte des octets de remplissage pour que l'alignement soit maintenu.
– La macro `CMSG_FIRSTHDR()` prend en entrée un pointeur sur un en-tête de message. Elle évalue les champs de cet en-tête pour savoir si des données ancillaires existent et, le cas échéant, renvoye le pointeur de l'en-tête de message de contrôle de la première donnée ancillaire du tampon, la valeur `NULL` sinon.
– La macro `CMSG_NXTHDR()` prend en entrée un pointeur sur un en-tête de message et un pointeur sur l'en-tête de message de contrôle en cours. Elle renvoie le pointeur de l'en-tête de message de contrôle suivant s'il existe, la valeur `NULL` sinon.

7.2.3.4 Pièces d'identité d'un utilisateur

Une pièce d'identité est une entité du type `struct ucred` (pour *User CREDentials*) :

```
struct ucred {
        __u32   pid;
        __u32   uid;
        __u32   gid;
};
```

qui comprend trois champs : un identificateur de processus, un identificateur d'utilisateur et un identificateur de groupe d'utilisateurs.

7.3 Les options

Pour récupérer et positionner les options d'une socket, on utilise les fonctions suivantes :

```
#include <sys/types.h>
#include <sys/socket.h>

int getsockopt(int s,
               int level,
               int optname,
               void *opval,
               socklen_t *optel);
int setsockopt(int s,
               int level,
               int optname,
               void *opval,
               socklen_t optel);
```

dont la signification de chacun des cinq arguments est la suivante :

- `s` est le numéro de fichier duquel inspecter (auquel positionner) l'option.
- `level` est le niveau de protocole, qui prend l'une des deux valeurs suivantes :
  - `SOL_SOCKET` pour le niveau utilisateur ;
  - `SOL_TCP` pour le niveau transport TCP.
- `optname` est le nom de l'option. Dans le cas du niveau `SOL_SOCKET`, il s'agit de l'une des valeurs suivantes :
  - `SO_REUSEADDR` permet à un serveur d'accepter un nouveau client voulant utiliser la même adresse (même adresse IP et même numéro de port) qu'un client déjà actif. Ceci permet, par exemple, à deux processus distants d'accéder à l'ordinateur *via* telnet ou le Web.
  - `SO_KEEPALIVE` spécifie à un serveur d'envoyer un message à une socket distante si elle est inactive depuis assez longtemps (en général deux heures).
  - `SO_LINGER` permet de contrôler comment une socket est arrêtée lors de l'appel de `close()`, pour les types de communication orientés connexion (tel que TCP) uniquement.
  - `SO_BROADCAST` permet la diffusion générale avec un type de communication non connecté, UDP par exemple.
  - `SO_OOBINLINE` permet l'envoi de données urgentes.
  - `SO_SNDBUF` pour la taille du tampon d'envoi, la valeur étant un entier (le nombre d'octets).
  - `SO_RCVBUF` pour la taille du tampon de réception, la valeur étant un entier (le nombres d'octets).
  - `SO_TYPE` pour le type de socket (récupération uniquement), dont les deux valeurs entières principales sont `SOCK_STREAM` et `SOCK_DGRAM`.
  - et enfin `SO_ERROR`.
- `optval` est un pointeur sur le tampon de réception (ou d'envoi) de la valeur de l'option.
- `optlen` est (un pointeur, dans le cas de la récupération, car il s'agit d'un passage par adresse sur) la longueur de l'option.