TP UNIX N°6

ISI − 2ème année

Les sockets

Introduction

Les *sockets* sont des points de connexion pour une communication. En première approximation, les modèles de communication qui sont accessibles à travers les *sockets* sont tout à fait analogues à deux outils de la vie courante : le courrier et le téléphone. Dans tous les cas de communication, il y a au moins deux entités qui communiquent : il faut donc au moins deux points d'entrée.

− Le courrier

L'analogie de la *socket* est ici une boîte aux lettres. Le message est une lettre qui porte une adresse. L'expéditeur dépose la lettre dans la boîte aux lettres du service postal. L'adresse permet aux services postaux de déposer le message dans la boîte aux lettres du destinataire.

− Le téléphone

Dans ce modèle de communication, les deux entités établissent une connexion directe entre leurs postes téléphoniques respectifs. Le numéro du poste de l'appelé doit être connu pour que l'initiateur de la connexion puisse le composer. Le modèle de messages échangé est un flot bidirectionnel : il n'y a pas de notion de frontières entre des messages. L'analogie de la *socket* est ici un poste téléphonique.

1. Caractéristiques d'une socket

1.1 Propriétés d'une communication

a) Fiabilité

b) Préservation de l'ordre des données

c) Non duplication des données

d) Communication en mode connecté

e) Préservation des limites de messages

f) Possibilités d'envoi de messages "urgents"

1.2 Le domaine d'une socket

Les différents domaines

- AF_UNIX : sockets locales au système (à la machine),
- AF_INET : domaine Internet,
- AF_NS : protocoles XEROX NS,
- AF_CCITT : protocoles CCITT X25, ...
- AF_SNA : IBM SNA,
- AF_DECnet,

- AF_APPLETALK.

1.3 Les types disponibles

Le type **SOCK_DGRAM :** Mode non connecté, envoi de datagrammes de taille bornée, préservation des limites de messages (propriétée). Dans le domaine Internet, le protocole sous−jacent est UDP (c.f. `man udp`).

Le type **SOCK_STREAM** : Communications fiables (propriétés a, b, et c) en mode connecté (propriété d). Eventuellement messages "hors−bande" (propriété f). Dans le domaine Internet, le protocole sous−jacent est TCP (c.f. `man tcp`).

Le type **SOCK_RAW** : Permet l'accès aux protocoles de plus bas niveau : par exemple, le protocole IP dans le domaine Internet.

Le type **SOCK_SEQPACKET** : Possède les propriétés a, b, c, d, et e. N'est pas disponible pour toutes les familles.

1.4 Création, identification et destruction d'une socket

La création d'une socket se fait grâce à l'appel de la fonction *socket()* :

```
SYNOPSIS
 cc [ flag ... ] file ... -lsocket -lnsl [ library ... ]

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

 int socket(int domain, int type, int protocol);
```

Cette fonction rend un (petit) entier qui servira à identifier la socket créée. Cet entier est un index dans la table des descripteurs de fichiers du processus. Il s'agit donc d'une ressource qui est du même genre que ce qui est allouée par la fonction *open()*. De ce fait, la destruction de la socket se fait avec le même appel que celui de la fermeture d'un fichier : *close()*.

Remarque : L'appel système *open()* d'une part alloue une ressource (le descripteur de fichier dans l'espace du processus) et en fournit l'identificateur, et d'autre part associe ce descripteur avec la ressource externe : le fichier. Pour celà, un des arguments de la fonction *open()* est le descripteur externe du fichier, à savoir son nom. Par contre, l'appel *socket()* ne fait que l'allocation de la ressource dans le processus, et ne fait pas la liaison avec le nom externe. Cette liaison est faite par l'appel *bind()*.

1.5 Attachement d'une socket à une adresse

L'attachement d'une adresse à une socket se fait grâce à l'appel de la fonction *bind()* :

```
SYNOPSIS
 cc [ flag ... ] file ... -lsocket -lnsl [ library ... ]

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

 int bind(int s, const struct sockaddr *name, int namelen);
```

1.6.1 Les adresses selon le domaine

Structure générique

La structure générique pour les adresses (extrait du fichier */usr/include/sys/socket.h*) :

```
struct in_addr {
 u_long S_addr;
 };
```

```
struct sockaddr {
 u_short sa_family; /* address family */
 char sa_data[14]; /* up to 14 bytes of direct address */
 };
```

Domaine AF_UNIX

La structure spécifique au domaine **AF_UNIX** (extrait du fichier */usr/include/sys/un.h*) :

```
struct sockaddr_un {
 short sun_family; /* AF_UNIX */
 char sun_path[108]; /* path name */
 };
```

Domaine AF_INET

La structure spécifique au domaine **AF_INET** (extrait du fichier */usr/include/netinet/in.h*) :

```
struct sockaddr_in {
 short sin_family; /* AF_INET */
 u_short sin_port; /* le numéro de port */
 struct in_addr sin_addr; /* l'adresse Internet */
 char sin_zero[8]; /* un champ de 8 zéros */
 };
```

Dans le domain AF_INET, le premier champ doit contenir `AF_INET`. Le troisième champ identifie la machine par son numéro IP. La machine émetrice doit remplir ce champ avec le numéro de la machine à laquelle elle veut envoyer des données. On peut spécifier ce numéro "à la main", si on le connaît, ou utiliser le résultat de *gethostbyname()* si on connaît la machine destinatrice par son nom.

```
SYNOPSIS
 cc [ flag ... ] file ... -lnsl [ library ... ]

 #include <sys/types.h>
 #include <sys/socket.h>
 #include <netinet/in.h>
 #include <arpa/inet.h>
 #include <netdb.h>

 struct hostent *gethostbyname(const char *name);
```

Enfin, le deuxième champ doit contenir le numéro de port voulu sur la machine destinatrice. Là encore, on peut fixer ce numéro "à la main" ou utiliser le résultat de la fonction *getservbyname()* :

```
SYNOPSIS
 cc [ flag ... ] file ... -lsocket -lnsl [ library ... ]

 #include <netdb.h>

 struct servent *getservbyname(const char *name, const char
 *proto);
```

Cette fonction utilise le fichier */etc/services* qui décrit les numéros de port universels.

On peut aussi du côté du récepteur laisser le système choisir un numéro en initialisant `sin_port` à zéro.

Du côté du récepteur, il faut obligatoirement appeler la fonction *bind()* pour que d'autres processus puissent contacter le récepteur à travers la socket ouverte. Cette fonction attend comme deuxième argument l'adresse de la machine réceptrice. Il faut effectivement y spécifier un numéro de port dans le deuxième champ. Par contre, pour avoir un programme portable d'une part, et qui fonctionne sur les machine reliées à plusieurs réseaux, dans le troisième champ (celui qui devrait contenir un numéro IP), on utilise la constante `INADDR_ANY` définie dans le fichier */usr/include/netinet/in.h*.

On peut retrouver l'adresse attachée à une socket dont on ne connaît que l'identificateur grâce à la fonction *getsockname()*.

```
SYNOPSIS
```

```
 cc [ flag ... ] file ... -lsocket -lnsl [ library ... ]

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

 int getsockname(int s, struct sockaddr *name, int *namelen);
```

2 Communication par datagrammes

2.1 Principe général

Les communications par datagrammes permettent d'échanger des messages analogues à des lettres :

- les frontières entre les messages sont conservées,
- il n'y a pas de connexion entre les deux entités communiquantes,
- chaque message doit contenir l'adresse de son destinataire (sauf cas de pseudo−connexion).

Il n'y a pas de garantie autre que la qualité du contenu du message. Cela signifie que :

- l'expéditeur n'est pas assuré de la délivrance de son message,
- il ne peut pas avoir d'information sur l'arrivée de son message,
- un même message peut être délivré plusieurs fois,
- deux messages émis dans un certain ordre n'arrivent pas forcément dans le même ordre.

Un processus, après avoir ouvert *une* socket, peut l'utiliser pour communiquer avec plusieurs partenaires, aussi bien en émission qu'en réception.

2.2 Exemple d'échange de données en mode datagramme

Côté émetteur

Une fois la socket créée, il suffit de l'utiliser avec la fonction *sendto()* :

```
SYNOPSIS
 cc [ flag ... ] file ... -lsocket -lnsl [ library ... ]

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

 int sendto(int s, const char *msg, int len, int flags,
 const struct sockaddr *to, int tolen);
```

Cette fonction retourne le nombre d'octets effectivement transmis et −1 en cas d'échec. Toutefois, les conditions d'échec ne sont que locales, en particulier, si on envoie un message à une machine et un numéro de port où aucun processus n'écoute, il n'y aura pas de compte rendu d'erreur, et le message sera perdu sans que l'émetteur en soit avisé. Par contre, la validité du descripteur de socket et de l'adresse sont vérifiés.

Côté récepteur

Après l'ouverture de la socket et son attachement, le récepteur attend des messages grâce à la fonction *recvfrom()*

```
SYNOPSIS
 cc [ flag ... ] file ... -lsocket -lnsl [ library ... ]

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

 int recvfrom(int s, char *buf, int len, int flags,
 struct sockaddr *from, int *fromlen);
```

Pour pouvoir récupérer un message avec cette fonction, elle doit être appelée après l'attachement de la socket, et après qu'un émetteur ait envoyé un message après cet attachement. Un message complet sera extrait de la file, même si sa longueur est plus grande que `len`, auquel cas la fin du message sera irrémédiablement perdu. Cette fonction retourne le nombre d'octets reçus, et −1 en cas d'erreur. C'est une fonction bloquante. Le récepteur peut connaître l'adresse de l'émetteur grâce à l'argument `from`, à condition de bien passer le nombre d'octets alloués pour `from` dans `fromlen`, la longueur effective de l'adresse se trouvant dans `fromlen` au retour.

3 Communication en mode connecté

3.1 Principe général

Dans le domaine Internet, ce mode de communication s'appuie sur le protocole TCP. Il accroit donc le volume d'octets transférés sur le réseau mais apporte en contre partie la fiabilité de la communication.

Un circuit virtuel est établi entre les deux entités communiquantes. Une fois la connexion établie, les deux entités jouent un rôle symétrique. Avant la connexion, l'une des deux entités doit attendre une connexion, et l'autre la demander ; elles n'ont donc pas un rôle symétrique.

Outre la fiabilité apportée par la communication avec le protocole TCP, un autre aspect de ce mode de communication est l'aspect continu de l'information : il n'y a pas de frontières de messages.

3.2 L'attente et l'établissement de la connexion

Une première socket d'écoute est ouverte par le récepteur. La fonction *listen()* installe une file d'attente sur cette socket. Ensuite l'appel de la fonction *accept()* bloque le processus. Lorsqu'une demande de connexion arrive sur la première socket, l'appel de *accept()* est débloqué, le processus réveillé, et une nouvelle socket est créée et est retournée par la fonction *accept()*. C'est cette deuxième socket qui est connectée à l'émetteur.

Pour réaliser un véritable serveur, il faut créer un nouveau processus dès qu'une demande de connexion arrive et débloque la fonction *accept()*. Le processus fils traitera alors le dialogue avec le client, pendant que le père attendra de nouvelles demandes de connexion sur la première socket qu'il a créée. Si on ne procède pas de cette façon, le serveur ne peut alors traiter qu'une seule demande à la fois, et toutes les autres demandes provenant d'autres clients sont mises en attente jusqu'à ce que le serveur ait terminé la transaction avec le client en cours de traitement.

3.3 La demande de connexion par le client

La demande de connexion par le client est réalisé par l'appel de la fonction *connect()*. Cette fonction réussit à condition qu'il y ait bien un processus en attente de connexion (fonction *listen()*) sur la machine et le port désignés par l'adresse passée à la fonction *connect()* et que la file des demandes de connexions ne soit pas pleine.

3.4 Le dialogue serveur/client

L'envoi de données sur la connexion établie par *connect()* et *accept()*, est bidirectionnel. Chacune des deux entités peut alors envoyer des octets dans ce tuyau. L'ordre des octets est conservé. Cependant un appel d'un côté ne correspond pas nécessairement à un appel de l'autre côté. C'est−à−dire qu'il peut y avoir fragmentation ou assemblage par le destinataire des blocs émis par l'émetteur.

Les émissions de données peuvent se faire avec *send()* ou *write()*. Ces appels sont bloquants lorsque le tampon de réception de la socket distante et le tampon d'émission de la socket locale sont pleins. La primitive *send()* permet en outre de pouvoir envoyer des messages "hors−bande" avec le flag `MSG_OOB`, ce que l'on ne peut pas faire avec *write()*. Par contre, on peut utiliser au dessus de *write()* les fonctions de la librairie standard d'entrée−sortie, par exemple *fprintf()*.

La réception de données peut se faire avec *recv()* ou *read()*.

Ces appels sont bloquants si aucun caractère n'est disponible pour la lecture. Le processus sera réveillé dès qu'un caractère sera disponible. La primitive *recv()* permet en outre de recevoir les messages "hors−bande" avec le flag `MSG_OOB` et la consultation sans extraction avec le flag `MSG_PEEK`. Par contre, on peut utiliser au dessus de *read()* les fonctions de la librairie standard d'entrée−sortie, par exemple *fgets()*.

3.5 Autres capacités

3.5.1 Les options sur les sockets

- *getsockopt()*
- *setsockopt()*

3.5.2 La déconnexion

- *close()*
- *shutdown()*

3.5.3 Le multiplexage des entrées−sorties

- *select()*

4 Pages de manuel à consulter AVANT de commencer à programmer !

Obligatoire :

- `socket`
- `connect`
- `send`
- `bind`
- `listen`
- `accept`
- `recv`

Conseillé :

- `ifconfig`
- `iostat`
- `netstat`
- `ping`
- `vmstat`

- hosts Le fichier */etc/hosts*.
- networks Le fichier */etc/networks*.
- protocols Le fichier */etc/protocols*.
- services Le fichier */etc/services*.

5 Exercice(s)

Ecrire une communication par socket entre 2 processus dans chacun des modes proposés.