Les sockets

Les sockets sont à la fois une bibiothèque de communication et un identifiant unique représentant une adresse sur le réseau. Des processus peuvent se connecter à cette adresse pour envoyer ou recevoir des données. Les primitives utilisées dans cette technique s'appuient sur le protocole TCP-IP.

7	
6	
5	SOCKET
4	TCP / UDP
3	IP
2	ETHERNET
1	ETHERNET

L'API (Application Programming Interface, interface d'utilisation) de ces protocoles repose principalement sur l'utilisation de socket de communication. Une socket est un port de communication depuis un processus vers l'extérieur. Un processus possédant une socket peut communiquer avec un autre processus possédant une socket à condition de mettre les deux sockets en rapport.

Il existe deux modes de communication avec les sockets.

- Un mode qui est dit "connecté" et qui utilise le protocole TCP, dans ce cas deux sockets établissent une relation durable qui permet de ne pas préciser la socket destinatrice à chaque envoi. On peut établir une analogie (relative) entre le mode connecté et l'utilisation de téléphones. Dans ce cas, la socket est un téléphone (appareil), il faut lui faire attribuer un numéro pour pouvoir l'appeler. Une fois la connexion établie il n'y a pas besoin de refaire le numéro chaque fois qu'on dit quelquechose.
- L'autre mode est dit "non-connecté", il utilise le protocole UDP, dans ce cas le destinataire de la socket est précisé à chaque envoi. L'analogie se fait avec des boîtes aux lettres. Pour pouvoir recevoir du courier il faut mettre un nom dessus, ensuite à chaque lettre on précise le destinataire.

On crée une socket grâce à la fonction :

```
descSock = socket(domain, type,0);
int domain, type;
```

où :

descSock est un int contenant un descripteur de socket au retour de la fonction. Ce descripteur nous sert de référence à chaque fois que nous devons utiliser la socket. Si la socket ne peut être crée *descSock* = −1.

domain est le domaine d'utilisation de la socket. Il y a deux domaines possibles AF_UNIX et AF_INET. Le domaine AF_UNIX est limité à une seule machine, le domaine AF_INET permet la communication entre machines sur le réseau Internet.

Pour le protocole TCP on met type à SOCK_STREAM et pour le protocole UDP, on met type à SOCK_DGRAM

Includes nécessaires :

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

Pour terminer un programme proprement il faut penser, avant de quitter à fermer les sockets qui ont été ouvertes en utilisant la fonction :

```
err = close( descSock);
int err, descSock;
```

où

descSock est le descripteur de la socket que l'on veut détruire,

err un code d'erreur en cas de problème, si $err == -1$ il y a une erreur.

1 Les adresses

Pour identifier la socket avec laquelle on désire se mettre en communication, il faut lui donner une adresse (ou un numéro de tel).

Le destinataire d'un message est identifié à partir d'une adresse sock_addr.

On donne une adresse à la socket grâce à la fonction :

```
err = bind(descSock, addr, addrLen)
int err, descSock;
struct sockaddr *addr;
int addrLen;
```

où :

err est un code d'erreur permettant de savoir comment ça c'est passé. Pb si $err = -1$.

descSock est le descripteur de la socket à laquelle on veut associer une adresse,

addr l'adresse qui doit être associée à la socket,

addrLen la taille de la structure d'adresse, parce que les adresses n'ont pas la même taille dans les domaines AF_UNIX et AF_INET. On utilisera sizeof(addr).

Il y a deux types d'adresses qui correspondent à deux types de domaines :

- **Les adresses AF_UNIX** Les adresses AF_UNIX permettent uniquement la communication en local sur une machine (accède au même système de fichier). L'adresse d'une socket dans le domaine AF_UNIX est liée à un nom de fichier. Lorsqu'on donne une adresse à une socket un fichier est créé (penser à supprimer le fichier lorsqu'il n'est plus utilisé). L'adresse est de type sockaddr_un :

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

struct  sockaddr_un {
        short   sun_family;     /* AF_UNIX */
        char    sun_path[108]; /* nom de socket (chaine de car)*/
};
```

- **Les adresses AF_INET**

Dans le domaine AF_UNIX, l'appel à la fonction bind à pour effet de créer un fichier sous le répertoire courant (sauf si on donne un path explicite). On peut le voir en exécutant la commande `ls -l` après le bind (type s comme socket). A la fin du programme il faut détruire ce fichier pour pouvoir le recréer lorsqu'on relance le programme (sinon bind fait une erreur). Ceci se fait par la fonction unlink ou en utilisant rm après la fin du programme.

```
err = unlink( name);
char *name;
```

où name est la chaîne de caractère qui identifie le fichier, ce qui est donné par sun_path dans l'adresse.

Rque : en AF_UNIX on ne peut pas avoir une adresse pas défaut.

```
#include <netinet/in.h>
/* Adresse Internet d'une machine */
struct in_addr {
   u_long s_addr;
};
/* Adresse Internet d'une socket */
struct sockaddr_in{
   short   sin_family;        /* AF_INET */
   u_short sin_port;          /* numero du port associe */
   struct  in_addr sin_addr; /* adresse internet de la machine */
   char    sin_zero[8];       /* un champ de 8 caracteres nuls */
};
```

2 Les modes de communication

Suivant le protocole de communication choisi, on utilise différents modes de communication.

2.1 Le mode non-connecté

Principe de la boîte aux lettres.

Les messages envoyés sur les sockets sont identifiés à partir de l'adresse mémoire à laquelle ils se trouvent et de leur taille.

On envoie un message grâce à la fonction :

```
nb = sendto(descSock, msg, msgLen, 0, to, toLen)

int nb;                 /* nombre octets envoyes ou code d'erreur */
int descSock;           /* descripteur de la socket locale */
char *msg;              /* pointeur sur le message à envoyer */
int msgLen;             /* taille du message */



struct sockaddr *to;  /* adresse (AF_UNIX) du destinataire */
int toLen;            /* taille de l'adresse, comme dans bind */
```

Rque : ici la technique utilisée pour identifier une zone mémoire consiste à prendre l'adresse du début de la zone et la taille de la zone. Le système ne se soucie pas de ce qui se trouve (type) dans la zone.

On reçoit un message grâce à la fonction :

```
err =recvfrom(sockDesc, buf, bufLen, 0, from, fromLen)

int err;                /* nombre octets recus ou code d'erreur */
int sockDesc;           /* descripteur de la socket de recep */
char *buf;              /* pointeur sur le buffer de recep */
int bufLen;             /* taille du buffer, >= taille msg */
struct sockaddr *from; /* adresse de l'envoyeur s'il en a */
int *fromlen;           /* taille de cette adresse */
```

L'appel est bloquant tant que des données ne sont pas recues, pas forcément la totalité. Attention, si buffer est trop petit on peut perdre du message.

2.2 Le mode connecté

Dans le mode connecté on relie temporaiarement les deux descripteurs. Principe du téléphone. il n'est donc plus nécéssaire de préciser l'adresse à chaque échange. Par contre cela implique une petite gymnastique préliminaire. On distingue dans ce cas un client d'un serveur. Le client se limite à demander sa connexion au serveur grâce à la fonction :

```
err = connect(s, name, nameLen)
int err;                /* code d'erreur */
struct sockaddr *name; /* adresse de la socket a laquelle on se
                          connect */
int nameLen;           /* taille de cette adresse */
```

Cela suppose que le serveur ait donné une adresse réseau à sa socket. Le système envoie alors une demande de connexion au serveur. Cet appel est bloquant tant que la connexion n'a pas été acceptée ou refusée.

Pour que le serveur ne mélange pas les demandes de connexion et les données qui sont recues sur sa socket, il est nécéssaire de déclarer une socket comme réservée aux demandes de connexion grâce à la fonction :

```
err = listen(sockDesc, backlog)
int err;       /* code d'erreur */
int sockDesc; /* descripteur de la socket a transformer */
int backlog;  /* nombre max de demande de connexion misent en
                 attente */
```

Cette fonction a également pour effet de créer, associée à la socket, une file d'attente de demandes de connexion (max 8). L'appel à `listen` est non-bloquant.

On se met en attente de connexion grâce à la fonction :

```
socData = accept(socConex, addr, addrlen)
int socData;            /* nouveau desc ou code erreur */
int socConex;           /* socket connexion */
struct sockaddr *addr; /* adrsocket demandant la connexion */
int *addrlen;           /* pointeur sur la taile de l'adresse */
```

La fonction `accept` a pour effet de créer une nouvelle socket dont le descripteur est `sockData`, connectée avec celle qui a demandé la connexion. C'est donc cette nouvelle socket qui sert pour échanger des données avec le client. La socket `socConnex` reste affectée à la réception de demandes de connexion. Cet appel est bloquant tant qu'on a pas reçu une demande de connexion, permet d'attendre uen demande de connexion.

On envoie un message grâce à la fonction :

```
nb = send(socDesc, msg, msgLen, 0)

int nb;        /* nombre octets envoyes ou code d'erreur */
int socDesc;   /* descripteur de la socket utilisee en emission */
char *msg;     /* pointeur sur le message */
int msgLen;    /* longueur du message */
```

On reçoit un message grâce à la fonction :

```
nb =  recv(socDesc, buf, bufLen, 0)
int nb;       /* nombre octets recus ou code d'erreur */
int socDesc;  /* descripteur de la socket utilisee en reception */
char *buf;    /* adresse du buffer de reception */
int bufLen;   /* longueur du buffeur */
```

Ces deux appels sont bloquants, send si on émet trop vite, recv si rien n'est envoyé.

On ferme une connexion grâce à la fonction :

```
err = shutdown(descSock, how)
int err;       /* code d'erreur */
int descSock; /* descripteur de la socket */
int how;       /* 0 ferme en reception
                  1 ferme en envoi
                  2 ferme tout */
```

Au lieu d'utiliser *send* et *recv* on peut utiliser l'interface POSIX standard *read* et *write*

Envoi de caractères : write

nb = write(socDesc, msg, msgLen)

Réception de caractères : read

nb = read(socDesc, msg, msgLen)

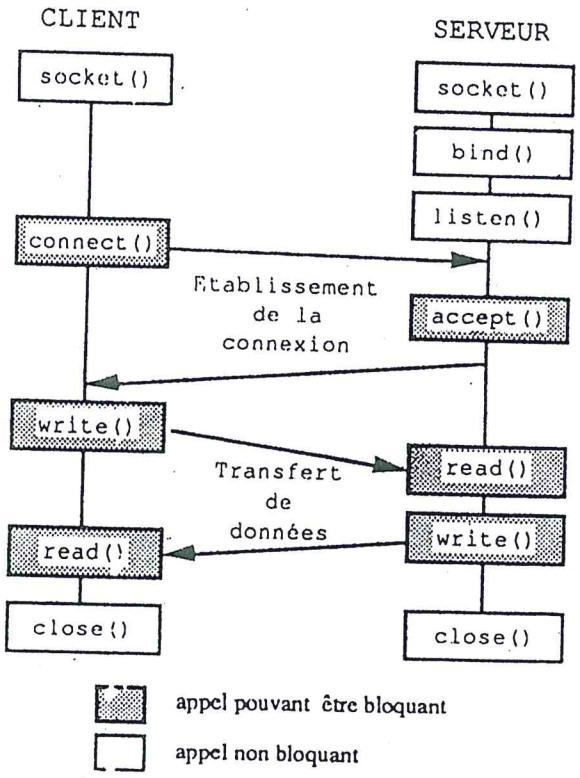
Remarques :

Les fonctions `recvfrom` et `accept` retournent dans *from* l'adresse de l'émetteur s'il a fait un *bind*

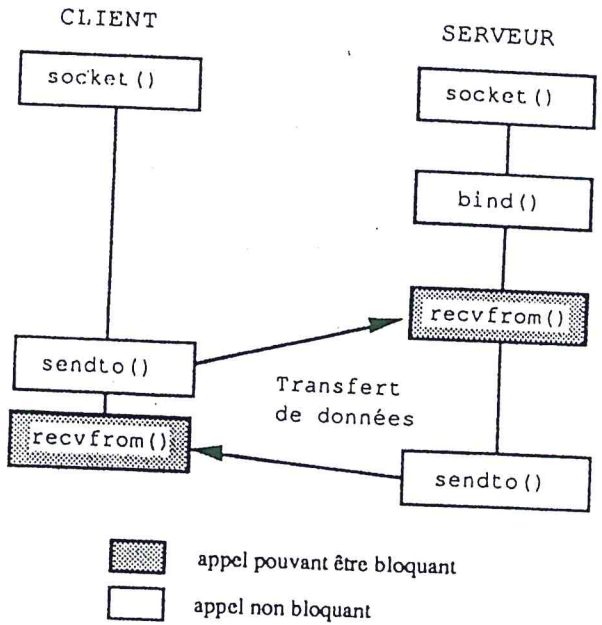
Les appels `read`, `recv`, `recvfrom`, `accept` sont bloquants

On peut faire afficher les messages d'erreur de 3 façons différentes. Dans tous les cas il faut écrire `extern int errno;` et `#include <errno.h>`

- `fprintf(stderr, " erreur \n" )`
- `void perror(const char *)`, affiche un message d'erreur en fonction de *errno* et la chaîne donnée en paramètre,
- `char *strerror ( int errno)` rend une chaîne qui contient le message fonction de *errno*.



- Utilisation des sockets en mode connecté. TCP



- Utilisation des sockets en mode datagramme. (non connecté)

UDP