1 Introduction

1.1 Définition

La notion de **sockets** a été introduite dans les distributions de Berkeley (un des premiers UNIX), c'est pourquoi on parle parfois de **sockets BSD** (Berkeley Software Distribution). Les sockets utilisent un modèle permettant la communication inter processus (IPC - Inter Process Communication) afin de permettre à divers processus de communiquer entre eux aussi bien sur une même machine qu'à travers un réseau TCP/IP.

La communication par socket utilise deux modes de communication :

- Le **mode connecté** (comme une communication téléphonique), qui utilise le protocole **TCP**. Une connexion durable est établie entre les deux processus, de telle façon que l'adresse de destination n'est pas nécessaire à chaque envoi de données.
- Le **mode non connecté** (comme un envoi par courrier), utilisant le protocole **UDP**. Ce mode nécessite l'adresse de destination à chaque envoi, et aucun accusé de réception n'est donné.

1.2 Position des sockets dans le modèle OSI

Les sockets se situent dans les couches supérieures au-dessus de TCP ou UDP :

Modèle des sockets	Modèle OSI
Application utilisant les sockets	Application
	Présentation
	Session
UDP/TCP	Transport
IP/ARP	Réseau
Ethernet, X25, ...	Liaison
	Physique

FIGURE 1 – Position des sockets dans le modèle OSI.

1.3 Principe d'un échange d'une communication en mode connecté

Voici un schéma montrant les différentes étape d'une communication en mode connecté :

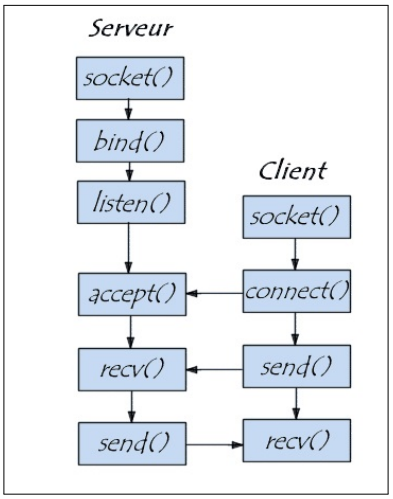

FIGURE 2 – Modèle de communication en mode connecté.

Le point important est que le programme serveur doit être lancé en premier. En effet celui-ci attend une

demande connexion de la part d'un programme client. Il faut donc qu'il soit prêt au moment où le programme client effectue sa demande de connexion, sinon la demande échoue.

Les deux cotés doivent d'abord créer une socket **(primitive socket)**, ensuite le serveur doit se mettre en attente en liant **(primitive bind)** sa socket à une (ou toutes) de ses interfaces réseaux pour indiquer sur quelle interface le serveur va attendre une connexion. De plus il indiquera le numéro de port TCP sur lequel il attend. Ensuite l'appel **listen** permet d'indiquer qu'il va se mettre en attente d'une demande de connexion. Enfin **la primitive accept** qui en général est bloquante permet de se mettre en attente de connexion.

De son coté le programme client exécute **la primitive connect** qui (si elle aboutit) permet ensuite les échanges avec **send** et **recv**. Enfin l**a primitive close permet de cloturer le canal de communication, ce qui libère des ressources et ferme aussi la connexion TCP.**

1.4 Principe d'un échange d'une communication en mode non connecté

Voici un schéma montrant les différentes étape d'une communication en mode non connecté :

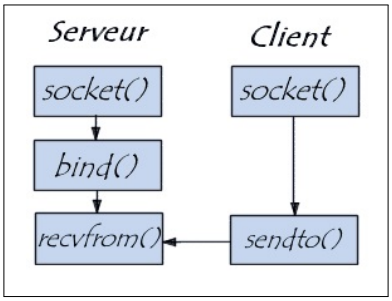

FIGURE 3 – Modèle de communication en mode connecté.

Pour une communication en mode non connecté, celui qui veut emettre n'a pas besoin de faire appel à **la primitive bind**. Il suffit de créer la socket avec l**a primitive socket**, puis d'envoyer au vis à vis avec **la primitive sendto**.

Par contre l'extrémité qui veut recevoir doit indiquer avec **la primitive bind** sur quel port UDP il attend la trame. La réception se fait avec **la primitive recvfrom**.

2 Programmation des sockets

2.1 Utilisation de la primitive socket

Pour créer une socket, on utilisera l'appel système socket() qui crée le point de communication. Voici le prototype :

```
#include <sys/types.h>
#include <sys/socket.h>
int socket(int domain, int type, int protocol);
```

Paramètres :
- domain : famille
  - AF_INET (ou PF_INET) famille TCP/IP version 4
  - AF_INET6 famille TCP/IP version 6
  - AF_UNIX famille Unix semblable aux tubes
- type indique le type de service (orienté connexion ou non).
  - SOCK_STREAM : service orienté connexion (TCP)
  - SOCK_DGRAM : service sans connexion, utilisation de datagrammes UDP.
- protocole : permet de préciser un protocole particuler pour certains types de domaines. Mettre 0 pour les sockets TCP/IP.

Valeur de retour : Cet appel système renvoie un descripteur référençant la socket créée s'il réussit. S'il échoue, il renvoie -1 et errno contient le code d'erreur.

Donc pour une communication en mode connecté :

```
int fdSocket=socket(AF_INET,SOCK_STREAM,0);
if(fdSocket== -1)
        {
        printf("Erreur de création de socket\n");
        exit(0);
        }
```

Donc pour une communication en mode non connecté :

```
int fdSocket=socket(AF_INET,SOCK_DGRAM,0);
if(fdSocket== -1)
        {
        printf("Erreur de création de socket\n");
        exit(0);
        }
```

2.2 Les structures d'adresse utilisées

Les appels systèmes bind(), accept() et connect() utilisent une structure permettant de préciser :
- une adresse IPV4
- un numéro de port (TCP ou UDP)

En fait il existe une structure générique permettant d'être utilisée dans tous les cas (quel que soit le type de socket), ensuite une structure spécifique aux sockets de type TCP/IP nécessitent l'utilisation d'une autre structure comportant les paramètres ci-dessus.

La structure générique s'appelle **sockaddr** dont voici la définition (elle est faite dans le fichier sys/socket.h) :

```
 struct sockaddr
{
unsigned short int sa_family; //au choix
unsigned char sa_data[14]; //en fonction de la famille
};
```

Pour la famille AF_INET on utiise les structures suivantes (définies dans netinet/in.h) :

```
 struct in_addr { unsigned int s_addr; }; // une adresse Ipv4 (32 bits)
struct sockaddr_in
{
unsigned short int sin_family; // <- PF_INET
unsigned short int sin_port; // <- numéro de port
struct in_addr sin_addr; // <- adresse IPv4
unsigned char sin_zero[8]; // ajustement pour être compatible avec sockaddr
};
```

2.3 Les fonctions permettant d'écrire dans la structure sockaddr_in

Pour inscrire les valeurs numériques nécessaires dans la structure sockaddr_in il faut prendre garde à l'ordre des octets. Les fonctions réseau nécessitent que les octets (par exemple les adresses IP sur 4 octets, ou les numéros de port sur 2 octets) soient rangés par ordre de poids forts d'abord. C'est ce qu'on appelle ordre big endian. L'ordre de rangement en mémoire dépend du processuer et peut être big endian ou little endian :

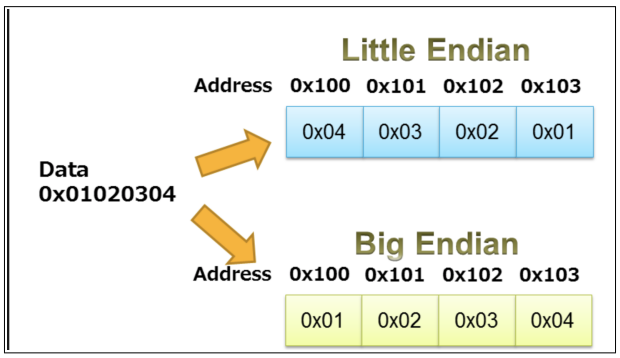

FIGURE 4 – Big endian / little endian.

La fonction htons() (host to network short) permet d'affecter un mot de 16 bits dans l'ordre big endian. On l'utilisera pour affecter le numéro de port dans la structure. La fonction htonl() (host to network long) permet d'affecter un mot de 32 bits dans l'ordre big endian.

Les adresses IP sont en général connues sous forme de chaine de caractère. La fonction inet_aton() sert à inscrire l'adresse dans l'ordre réseau :

```
int inet_aton (const char *cp, struct in_addr *inp);
```

cp est l'adresse IP en ascii (notation décimale pointée) et inp est l'adresse d'une structure de type in_addr où inscrire la valeur. La fonction retourne 0 en cas d'échec (adresse cp non valide).

2.4 L'appel système bind()

Cette primitive est utilisée pour définir sur quelle interface et quel numéro de port on veut attendre soit une connexion en mode connecté, soit un paquet en réception en mode non connecté. On dit qu'on nomme la socket. Son prototype est :

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

int bind(int sockfd, const struct sockaddr *addr, socklen_t addrlen);
```

Paramètres :
- sockfd : le descripteur de la socket à nommer
- sockaddr : adresse de la structure (attention bind attend l'adresse d'une structure générique)
- addrlen : taille en nombre d'octets de la strucure utilisée

Valeur de retour : L'appel renvoie 0 s'il réussit, ou -1 s'il échoue, auquel cas errno contient le code d'erreur.

Voici un exemple de nommage pour un serveur en mode connecté :

```
struct sockaddr_in pointDeRencontreLocal;
int longueurAdresse;
// préparer la structure sockaddr_in
longueurAdresse = sizeof(struct sockaddr_in);
memset(&pointDeRencontreLocal, 0x00, longueurAdresse);
pointDeRencontreLocal.sin_family = AF_INET;
pointDeRencontreLocal.sin_addr.s_addr = htonl(INADDR_ANY);
// toutes les interfaces locales disponibles
pointDeRencontreLocal.sin_port = htons(6000);
//permet d'attendre la connexion sur le port 6000
if(bind(fdSocket,(sockaddr*)&pointDeRencontreLocal,longueurAdresse) == -1)
        {
        printf("Erreur de nommage \n");
        exit(0);
        }
```

La manière de renseigner l'adresse IP du serveur indique que celui-ci attend une connexion sur n'importe laquelle de ses interfaces.

Si on veut préciser sur quelle interface unique attendre la connexion on va utiliser la fonction inet_aton().

Par exemple si le serveur a une interface réseau d'adresse 192.168.0.1 et qu'on attend une connexion dessus :

```
inet\_aton("192.168.0.1",&pointLocal.sin_addr);
```

2.5 La primitive listen()

La primitive listen() permet au serveur de se mettre en attente de connexion. Si plusieurs demandes de connexions ont lieu, le programme serveur va les stocker dans une file d'attente dont la profondeur est définie par liste(). Son prototype :

```
#include <sys/types.h>
#include <sys/socket.h>
int listen(int sockfd, int backlog);
```

Paramètres :
- sockfd : le descripteur de la socket
- backlog : définit une longueur maximale jusqu'à laquelle la file des connexions en attente pour sockfd peut croître

Valeur de retour : L'appel renvoie 0 s'il réussit, ou -1 s'il échoue, auquel cas errno contient le code d'erreur.

Voici la façon typique d'utiliser listen() :

```
if(listen(fdSocket,5)== -1)
        {
        printf("Erreur de listen \n");
        exit(0);
        }
```

2.6 La primitive accept()

L'appel système accept() est employé avec les sockets utilisant un protocole en mode connecté (SOCK_STREAM). Il extrait la première connexion de la file des connexions en attente de la socket sockfd à l'écoute, crée une nouvelle socket connectée, et renvoie un nouveau descripteur de fichier qui fait référence à cette socket qui servira aux connexions ultérieures. Le prototype est :

```
#include <sys/types.h>
#include <sys/socket.h>
int accept(int sockfd, struct sockaddr *adresse, int *longueur);
```

Paramètres :
- sockfd : descripteur de la socket qui est en écoute
- adresse : structure d'adresse du point de connexion distant où la primitive va noter les coordonnées du client qui se connecte.
- adresse d'un entier qui doit contenir la taille de la structure d'adresse.

Valeur de retour : S'il réussit, accept() renvoie un entier non négatif, constituant **un descripteur pour la nouvelle socket**. S'il échoue, l'appel renvoie -1 et errno contient le code d'erreur.

Remarque : la structure d'adresse du client (point distant du serveur) n'a pas besoin d'être préparée, seule la variable contenant la taille de la structure doit l'être.

Voici un exemple :

```
struct sockaddr_in PointDistant;
int longueurAdresse=sizeof(PointDistant);
int fdSocketCommunication=accept(fdSocket,(sockaddr*)&PointDistant,&longueurAdresse);
if(fdSocketCommunication == -1)
        {
        printf("Erreur d'accept \n");
        exit(0);
        }
```

2.7 La primitive connect()

Cette primitive permet à un client en mode connecté de débuter une connexion sur une socket. Voici le prototype :

```
#include <sys/types.h>
#include <sys/socket.h>
int connect(int sockfd, const struct sockaddr *serv_addr,int addrlen);
```

Paramètres :
- sockfd : descripteur de la socket du client
- serv_addr : adresse de la structure d'adresse décrivant le serveur et devant être renseignée avant l'appel de connect ;
- addrlen : taille de la structure d'adresse

Valeur de retour : connect() renvoie 0 s'il réussit, ou -1 s'il échoue, auquel cas errno contient le code d'erreur.

Voici un exemple pour lequel l'adresse IP du serveur vaut 192.168.0.1 et le port serveur 6000 :

```
struct sockaddr_in pointDistant;
pointDistant.sin_family=AF_INET;
inet_aton("192.168.0.1",&pointDistant.sin_addr);
pointDistant.sin_port=htons(6000);
if(connect(fdSocket,(struct sockaddr*)&pointDistant,sizeof(pointDistant)) == -1)
 {
 printf("Erreur de connect \n");
 exit(0);
 }
```

2.8 La primitive send()

Cette primitive permet d'envoyer un bloc de données sur la socket du client ou la socket de communication du serveur. Son prototype est :

```
#include <sys/types.h>
#include <sys/socket.h>
int send(int s, const void *buf, int len, int flags);
```

Paramètres :
- s : descripteur de la socket ;
- buf : adresse du buffer (tableau d'octets) qui contient les données à envoyer ;
- len : nombre d'octets à transmettre
- flags : options (mettre 0)

Valeur de retour : si send réussit il renvoie le nombre de caractères émis. S'il échoue, il renvoient -1 et errno contient le code d'erreur.

2.9 La primitive recv()

Cette primitive permet de recevoir un message sur la socket du client ou la socket de communication du serveur. Son prototype est :

```
#include <sys/types.h>
#include <sys/socket.h>
int recv(int s, const void *buf, int len, int flags);
```

Paramètres :
- s : descripteur de la socket ;
- buf : adresse du buffer (tableau d'octets) qui va contenir les données à recevoir ;
- len : nombre maximum d'octets à recevoir
- flags : options (mettre 0)

Valeur de retour : si recv réussit il renvoie le nombre de caractères effectivement reçus. S'il échoue, il renvoient -1 et errno contient le code d'erreur.