INFO-F-201 – OS
TP8 : Programmation réseau

Yannick Molinghen, Alexis Reynouard

22 novembre 2021

ULB

1 Sockets

Dans le monde Unix, un socket est une interface de communication inter-processus bidirectionnelle. Cependant, on parle généralement de socket dans le cadre de la programmation réseau (généralement sur des machines différentes) qui est un cas particulier de comminucation inter-processus.

Les sockets sont des canaux de communication bidirectionels et sont indépendants du protocole utilisé (TCP, UDP, RFCOMM (bluetooth), L2CAP (bluetooth), ...). Ils servent uniquement à assurer l'envoi et la réception de messages.

Dans le cadre de ce cours, nous nous intéresserons uniquement aux sockets réseaux avec le protocole TCP/IP.

1.1 Procédure de connexion

Afin d'établir une connexion entre deux hôtes, il y a une procédure à suivre comme illustré dans la figure 1. Plus de détails sont donnés sur les différentes étapes et leur utilité dans les sections suivantes.

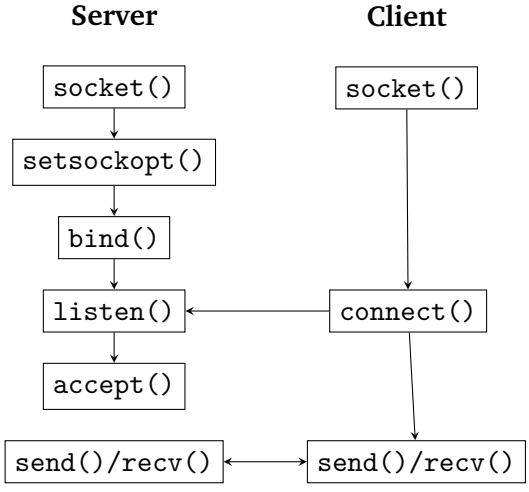


FIGURE 1 – TCP socket setup

Ce processus est illustré dans les listings 1 et 2. Attention, les erreurs n'y sont pas vérifiées !

```c
int server_fd = socket(AF_INET, SOCK_STREAM, 0); // file descriptor
int opt = 1;
// Permettre la réutilisation du port/de l'adresse
setsockopt(server_fd, SOL_SOCKET, SO_REUSEADDR | SO_REUSEPORT, &opt, sizeof(opt));
struct sockaddr_in address;
address.sin_family = AF_INET;
address.sin_addr.s_addr = INADDR_ANY;
address.sin_port = htons(8080);
bind(server_fd, (struct sockaddr *)&address, sizeof(address));
listen(server_fd, 3); // maximum 3 connexions en attente
size_t addrlen = sizeof(address);
int new_socket = accept(server_fd, (struct sockaddr *)&address, (socklen_t *)&addrlen);
char buffer[1024];
read(new_socket, buffer, 1024);
printf("Message reçu:%s\n", buffer);
close(new_sock);
close(server_fd);
```

Listing 1 – Exemple de serveur dans une comunication via socket.

```c
int sock = socket(AF_INET, SOCK_STREAM, 0);
struct sockaddr_in serv_addr;
serv_addr.sin_family = AF_INET;
serv_addr.sin_port = htons(8080);
// Conversion de string vers IPv4 ou IPv6 en binaire
inet_pton(AF_INET, "127.0.0.1", &serv_addr.sin_addr);
connect(sock, (struct sockaddr *)&serv_addr, sizeof(serv_addr));
char message[] = "Salut à tous";
write(sock, message, strlen(message));
close(sock);
```

Listing 2 – Exemple de client dans une communication via socket

1.1.1 Créer un socket

La fonction suivante permet de créer un socket et de récupérer son descripteur de fichier.

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

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

Le paramètre `domain` définit le domaine (famille de protocole) du socket. Dans notre cas, ce sera `AF_INET` (protocol family internet) pour permettre les communications sur le réseau.
Le paramètre `type` définit le type de socket, dans le cadre de ce cours, nous utiliserons des sockets de type `SOCK_STREAM`, permettant une communication bidirectionnelle, sûre (pas de perte de paquet ni de duplication) et séquencée (l'ordre des envois est respecté).
Le paramètre `protocol` définit le type de protocole de communication qui sera utilisé sur le socket. Lorsque cette valeur est mise à 0, le protocole par défaut (défini en fonction des deux premiers arguments) est utilisé. Dans notre cas, le protocole par défaut est TCP.

Il est utile de préciser certaines options pour un socket telles que `SO_REUSEADDR` et `SO_REUSEPORT` qui permettent de réutiliser l'adresse/le port en question s'il a été utilisé récemment. L'ommission de cette option a pour conséquence que le programme ne pourra réutiliser la même adresse/port que lorsque l'OS aura constaté par lui-même que cette addresse/port n'est plus utilisée, ce qui peut prendre quelques minutes.
Avant la fermeture du programme, il faut prendre soin de bien fermer les sockets ouverts (comme pour tout descripteur de fichier) avec la fonction `close(int fd)`.

1.1.2 Lier un socket à une adresse et à un port

Comme vu précédemment, un socket est identifié sur le système par son descripteur de fichier. Or, pour pouvoir envoyer et recevoir des messages, il faut lui assigner un port, contenu dans une structure d'adresse de socket. Le lien entre les deux est fait à l'aide de la fonction `bind`.

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

int bind(int sockfd, struct sockaddr *my_addr, int addrlen);
```

Le premier paramètre est les descripteur de fichier identifiant le socket, le second est un pointeur sur la structure d'adresse contenant les informations à lier au socket et le dernier contient la taille de la structure.

Lorsque le port n'a pas besoin d'être fixe, le champ `sin_port` de la structure d'adresse à lier peut être mis à 0, auquel cas, c'est l'OS qui choisira de manière arbitraire un port disponible. Notez également que les ports 1 à 1024 sont réservés par le système et ne peuvent pas être utilisés. De plus, pour être valide, le port affecté ne doit pas non plus être déjà utilisé par un autre processus.

Il est important de comprendre que lorsqu'un processus veut recevoir des connexions et/ou des messages entrant, il est sensé lier le descripteur de fichier représentant le socket sur lequel il veut écouter avec une structure d'adresse de socket définissant une adresse (couple adresse IP et port dans notre cas). Comme mentionné l'appel à `bind` permet de faire cette correspondance. Cependant, il est souvent utile de lier un descripteur de fichier relatif à un socket à toutes les interfaces disponibles sur la machine, pour ce faire, on peut affecter au champ `sin_addr.s_addr` la valeur `INADDR_ANY` (on utilisera alors la fonction `htonl` pour réaliser la conversion en ordre gros-boutiste).

1.1.3 Attendre une connexion (côté serveur)

La phase de connexion est initialisée côté serveur par l'appel à la fonction suivante :

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

int listen(int sockfd, int backlog);
```

Cette fonction permet d'écouter sur le socket passé en paramètres des demandes de connexion provenant de l'extérieur. Le second paramètre défini le nombre maximal de demandes de connexions à mettre dans la file, en attente d'une acceptation.

Lorsque la fonction `listen` se termine, c'est qu'il y a un candidat à la connexion. C'est la fonction `accept` qui permet d'accepter cette nouvelle connexion.

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

int accept(int s, struct sockaddr *addr, socklen_t *addrlen);
```

La fonction `accept` renvoie un nouveau file descriptor qui correspond au socket qui permet de communiquer avec le client qui vient de se connecter. Ce socket est différent de celui qui a été utilisé pour l'appel à `bind` et `listen` car le serveur peut toujours recevoir de nouvelles connexions sur ce socket.

Le premier paramètre est le descripteur de fichier du socket sur lequel a été effectué l'appel à `listen`. Le second paramètre est l'adresse d'une structure d'adresse de socket qui contiendra les informations relatives au client qui s'est connecté (port du socket et adresse IP de la machine). Le dernier est la taille de la structure d'adresse de socket.

1.1.4 Se connecter (côté client)

La demande de connexion d'un client à un serveur se fait via l'utilisation de la fonction suivante :

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

int connect(int sockfd, struct sockaddr *serv_addr, int addrlen);
```

Le premier paramètre contient le descripteur de fichier relatif au socket (local) utilisé pour la communication avec une autre machine. Le second est une structure d'adresse contenant les informations relatives à un socket **distant**, elle contient donc l'adresse IP de la machine distante et le port sur lequel est sensé écouter cette dernière. Enfin, comme pour `bind`, le dernier argument contient la taille de la structure de données.
Il est important de noter qu'un appel à `connect` n'est pas obligatoirement précédé d'un appel à `bind`. Le cas échéant, un numéro de port est arbitrairement associé au socket par le système d'exploitation, comme dans l'exemple du Listing 2.

1.1.5 Fermer un socket

Pour rappel un socket peut être fermé en utilisant la fonction `close`, comme n'importe quel file descriptor.

```
#include <unistd.h>

int close(int fd);
```

Fermer un socket empêche lecture et écriture sur le socket et libère le port qui a été lié au socket.

Exercice 1. *Ecrivez un echo client et un echo server de sorte que le client demande d'entrer du texte à l'utilisateur puis l'envoie au serveur. Ensuite, le serveur renvoie ce qu'il a reçu au client.*
N'oubliez pas de vérifier la valeur de retour des fonctions.

Exercice 2. *Reprenez le code de l'exercice précédent et incluez-le dans une boucle de sorte que le client lise sur `stdin` jusqu'à ce que l'utilisateur appuie sur `CTRL+D`, et que le serveur lui réponde à chaque fois.*
Comment le serveur sait-il quand le client se déconnecte ?

2 Protocole de communication

Vous ne l'aurez peut-être pas remarqué, mais en résolvant l'exercice précédent, vous avez implémenté un protocole de communication selon lequel les messages transitent d'abord du client vers le serveur, puis du serveur vers le client, et ainsi de suite.

Ce protocole, aussi simple soit-il, illustre qu'il est important que le client et le serveur s'accordent sur la manière dont les informations sont structurées. Dans le cas où les informations envoyées sont de taille variable et très grandes, il peut par exemple être nécessaire d'envoyer comme premiers bytes la taille du message, afin que le destinataire sache combien de bytes il doit lire, ce qui implique de potentiellement effectuer plusieurs `read` successifs.
En effet, si les deux hôtes souhaitent par exemple effecuter une lecture sur le socket simultanément alors qu'il n'y a rien à lire, nous arrivons dans un deadlock.

Exercice 3. *Ecrivez un echo client et un echo serveur qui envoient comme première information de chaque message le nombre de bytes à lire sous forme d'un `size_t`.*

3 Plusieurs clients simultanés

Il est souvent nécessaire pour un serveur d'accepter plusieurs connexions à la fois, comme le ferait un serveur web. Il existe plusieurs manières de gérer des clients simultanées :

1. Gérer chaque client dans un processus
2. Gérer les clients dans un thread
3. Utiliser l'appel système `select`

Si les deux premières options peuvent être de bonnes options dans certains cas, elles n'en sont pas moins lourdes à gérer pour l'OS. Nous allons dans ce cadre-ci explorer la troisième option qui permet de gérer plusieurs connexions simultanément sans utiliser de multi-processing ni de multi-threading.

```
#include <sys/select.h>
int select(int nfds, fd_set *readfds, fd_set *writefds, fd_set *exceptfds,
           struct timeval *timeout);

void FD_CLR(int fd, fd_set *set);
int  FD_ISSET(int fd, fd_set *set);
void FD_SET(int fd, fd_set *set);
void FD_ZERO(fd_set *set);
```

Listing 3 – Appel système `select` et ses quelques macros

L'appel système `select` se base sur le principe de `fd_set`, pour *file descriptor set* ou ensemble de *file descriptors*. L'idée est de mettre dans des `fd_set` tous les *file descriptors* que l'on veut gérer, et de les donner comme entrée à `select` qui sera blocante jusqu'à ce qu'au moins un des *file descriptors* dans les `fd_set` soit prêt.

On distingue trois `fd_set`

— Ceux en lecture qui débloqueront `select` dès que l'un des sockets a des données à lire

— Cenx en écriture qui débloqueront `select` dès qu'il est possible d'écrire dans l'un d'eux

— Ceux susceptibles de générer des erreurs qui débloqueront le `select` dès qu'une erreur s'y produit.

Ainsi, un serveur peut mettre dans son `fd_set` tous les sockets de ses clients, et attendre que des données soient disponibles sur n'importe lequel des sockets (donc une requête) pour ensuite lui répondre.

Les macros présentes dans le Listing 3 sont des utilitaires pour gérer les structures `fd_set`, comme montré dans le Listing 4.

```
// appel à socket et bind déjà faits
fd_set readfds;
int max = 0;
// On ajoute tous les sockets au fd_set
FD_ZERO(&readfds);
for (int i=0; i < nb_sockets; i++) {
    FD_SET(sockets[i], &readfds);
    max = (sockets[i] > max) ? sockets[i] : max;
}
// Select avec uniquement un fd_set en lecture
select(max + 1, &readfds, NULL, NULL, NULL);
for (int i = 0; i < nb_sockets; i++) {
    if (FD_ISSET(sockets[i], &readfds)) {
        // On a reçu un message sur sockets[i]
        char buffer[1024];
        read(sockets[i], buffer, 1024);
        // traitement nécessaire, pourquoi pas dans un thread
    }
}
```

Listing 4 – Exemple de serveur utilisant `select`

Exercice 4. *Modifiez l'exercice précédent pour que votre serveur accepte plusieurs connexions simultanées avec `select`. Le client reste identique.*

A Annexes

A.1 Structure d'adresse de socket internet

Cette structure de données est utilisée pour stocker les informations d'un socket (local ou distant). Dans le cadre des communications internet, un socket est défini par :

1. une adresse IP,
2. un port.

La structure est donc définie comme suit :

```
struct sockaddr_in {
    short int          sin_family;  // Famille d'adresse
    unsigned short int sin_port;    // Numero de port
    struct in_addr     sin_addr;    // Adresse internet
    unsigned char      sin_zero[8]; // Espace vide
};
```

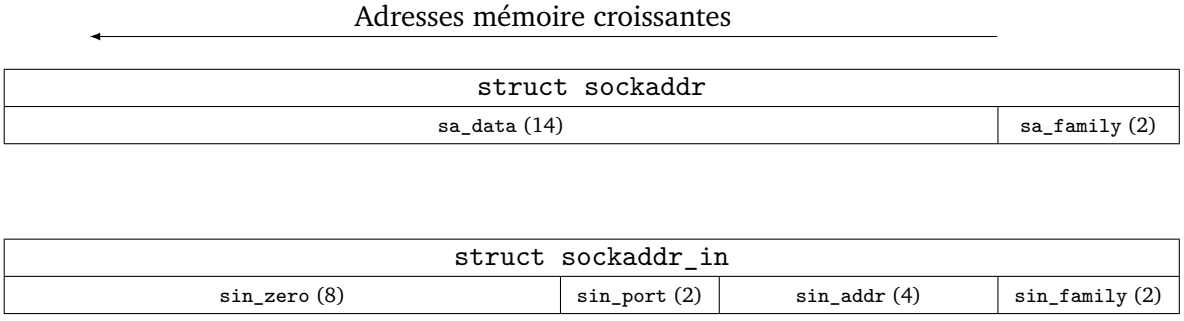
Avec :

```
struct in_addr {
    uint32_t long s_addr; // Adresse IPv4 sur 4 octets
};
```

Dans le cadre de ce cours, le champ `sin_family` devra tout le temps prendre la valeur `AF_INET` (address family internet), à savoir l'identifiant des familles d'adresses internet.
Un point d'attention peut être porté au champs `sin_zero`. Comme expliqué précédemment, les sockets servent dans un grand nombre de protocoles et souvent à des fins bien différentes, les informations nécessaires pour un bon fonctionnement peuvent alors être totalement différentes (on voit mal à quoi pourrait servir une adresse IP lors de communications locales par exemple). Afin d'offrir une interface de programmation unifiée (mêmes fonctions pour la manipulation de sockets), il a été introduit une structure générique pour les adresses de socket dont la définition est la suivante :

```
struct sockaddr {
    unsigned short int sa_family;    // Famille d'adresse, AF_xxx
    char               sa_data[14];  // 14 octets pour les adresses
};
```

Le champ `sa_data` est donc un espace mémoire générique 14 octets qui peut alors être découpé comme souhaité pour décliner les protocoles. Le découpage de cet espace mémoire peut alors être connu à l'aide du champ `sa_family`. La généricité d'une telle structure repose également sur sa taille, voici la représentation mémoire des deux structures d'adresse introduites :



Il est alors possible de passer d'une structure à l'autre en utilisant un cast.

A.2 Adresses IP

Une adresse IP est régulièrement écrite sous sa représentation A.B.C.D, ce qui correspond en réalité à quatre bytes, donc à un nombre sur 32bits qui est en réalité la représentatin utilisée par le système. Il existe quelques fonctions utilitaires pour transformer des adresses IP d'un format à l'autre.

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

// aton = address to number
int inet_aton(const char *cp, struct in_addr *inp);
// ntoa = number to address
char *inet_ntoa(struct in_addr in);
```

La fonction `inet_aton` retourne 0 en cas d'erreur et 1 si la chaîne a été correctement interprétée. La fonction `char *inet_ntoa (struct in_addr in)`, permet de faire la conversion inverse.

A.3 Représentation mémoire et boutisme (Endianness)

Dans la grande majorité des architectures actuelles, un nombre entier est représenté en mémoire sur plusieurs octets, cependant ces dernières peuvent différer sur l'agencement en mémoire de ces octets. Prenons l'exemple d'un entier codé sur 16 bits soit deux octets, Il existe alors deux représentations possibles :

Adresses mémoire croissantes ←

Octet de poids fort								Octet de poids faible							
2^{15}	2^{14}	2^{13}	2^{12}	2^{11}	2^{10}	2^{9}	2^{8}	2^{7}	2^{6}	2^{5}	2^{4}	2^{3}	2^{2}	2^{1}	2^{0}

Petit boutiste (Little Endian)

Octet de poids faible								Octet de poids fort							
2^{7}	2^{6}	2^{5}	2^{4}	2^{3}	2^{2}	2^{1}	2^{0}	2^{15}	2^{14}	2^{13}	2^{12}	2^{11}	2^{10}	2^{9}	2^{8}

Gros boutiste (Big Endian)

Naturellement sur un réseau, des machines aux architectures différentes doivent pouvoir communiquer peu importe leur boutisme. C'est pourquoi les protocoles de communication définissent une norme de boutisme. Dans le cas des protocoles internet, c'est l'ordre gros-boutiste.
Comme vu précédemment, le port utilisé dans les structures d'adresse de socket est un `unisgned short int`, il est donc représenté (au minimum) sur deux octets. Il faut donc s'assurer que ce dernier (ainsi que tout entier transitant sur le réseau) soit bien stocké sous sa forme gros-boutiste. Pour réaliser ces conversions, différentes fonctions existent :

```
#include <arpa/inet.h>

// host to network long
uint32_t htonl(uint32_t hostlong);

// host to network short
```

```
uint16_t htons(uint16_t hostshort);

// network to host long
uint32_t ntohl(uint32_t netlong);

// network to host short
uint16_t ntohs(uint16_t netshort);
```