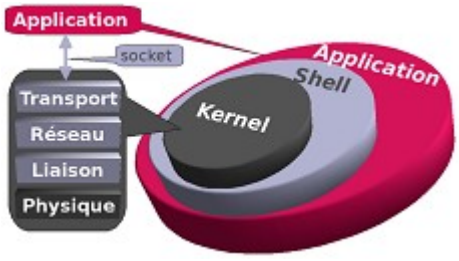
Les sockets

Table des matières

Les sockets sont des flux de données, permettant à des machines locales ou distantes de communiquer entre elles via des protocoles. Les différents protocoles sont TCP qui est un protocole dit "connecté", et UDP qui est un protocole dit "non connecté".



1. Introduction

Les sockets ont été mises au point en 1984, lors de la création des distributions BSD (Berkeley Software Distribution). Apparues pour la première fois dans les systèmes UNIX, les sockets sont des points de terminaison mis à l'écoute sur le réseau, afin de faire transiter des données logicielles.

Celles-ci sont associées à un numéro de port.

Les ports sont des numéros allant de 0 à 216-1 inclus (soit 65535). Chacun de ces ports est associé à une application (à savoir que les 1024 premiers ports sont réservé à des utilisations bien précises).

Les sockets sont aussi associées à un protocole (UDP/IP et TCP/IP). Nous utiliserons ici uniquement le protocole TCP/IP.

Les sockets servent à établir une transmission de flux de données (octets) entre deux machines ou applications.

2. Fonctionnement des sockets

Les sockets ne s'utilisent pas de manière identique selon les différents systèmes d'exploitation.

2.1. Sur MS-Windows

Presque tout ce qui touche aux sockets MS-Windows se trouve dans le fichier "winsock2.h". Il faut l'inclure dans le programme comme suit :

```
#include <winsock2.h>
```

On peut remarquer que le type socklen_t qui existe sous Linux, n'est pas défini sous MS-Windows. Ce type sert à stocker la taille d'une structures de type sockaddr_in. Ça n'est rien d'autre qu'un entier mais il nous évitera des problèmes éventuels de compilation sous GNU/Linux par la suite. Il va donc falloir le définir nous même à l'aide du mot clef typedef comme il suit :

```
typedef int socklen_t;
```

De plus, il faut ajouter, dans le début de la fonction main, le code suivant pour pouvoir utiliser les sockets sous MS-Windows :

WSADATA WSAData;

WSAStartup(MAKEWORD(2,2), &WSAData);

La fonction WSAStartup sert à initialiser la bibliothèque WinSock. La macro MAKEWORD transforme les deux entiers (d'un octet) qui lui sont passés en paramètres en un seul entier (de 2 octets) qu'elle retourne. Cet entier sert à renseigner la bibliothèque sur la version que l'utilisateur souhaite utiliser. Elle retourne la valeur 0 si tout s'est bien passé.

Puis à la fin, appelez cette fonction va simplement libérer les ressources allouées par la fonction WSAStartup().

WSACleanup();

2.2. Sur GNU/Linux

Sur GNU/Linux, les fichiers à inclure ne sont pas les mêmes...

Pour combler l'écart entre M$-Windows et GNU/Linux, nous utiliserons des définitions et des

typedef.

Commençons par inclure les fichiers nécessaires :

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
```

Un premier problème se pose :

Dans le fichier "socket.h" de GNU/Linux, la fonction qui sert à fermer une socket (que nous verrons par la suite) se nomme close alors que dans le fichier "winsock2.h" de MS-Windows la fonction se nomme closesocket... Pour éviter de faire deux codes sources pour deux OS différents, nous utiliserons une définition comme il suit :

```
#define closesocket(param) close(param)
```

Ainsi dans le code la fonction closesocket() sera remplacée par la fonction close() qui pourra ensuite être exécutée.

Le deuxième problème vient du fait qu'il "manque" deux définitions et trois typedef qui peuvent nous être utile dans le fichier "socket.h" de GNU/Linux par rapport au fichier "winsock2.h" de MS-Windows.

Voila donc le contenu du fichier pour le moment :

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>

#define INVALID_SOCKET -1
#define SOCKET_ERROR -1
#define closesocket(param) close(param)

typedef int SOCKET;
typedef struct sockaddr_in SOCKADDR_IN;
typedef struct sockaddr SOCKADDR;
```

2.3. Un code portable

Pour pouvoir avoir un code un peu plus portable, nous utiliserons les définitions WIN32 et GNU/linux.

Cette méthode indiquera à votre compilateur le code à compiler en fonction de l'OS.

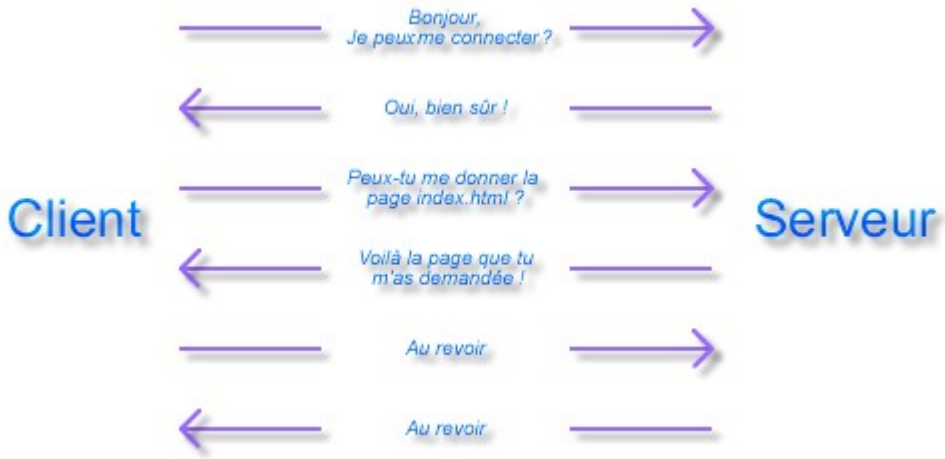
```
//Si nous sommes sous MS-Windows
#if defined (WIN32)
    #include <winsock2.h>

    // typedef, qui nous serviront par la suite
    typedef int socklen_t;
```

```
// Sinon, si nous sommes sous GNU/Linux
#elif defined (linux)

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

    // Define, qui nous serviront par la suite
    #define INVALID_SOCKET -1
    #define SOCKET_ERROR -1
    #define closesocket(s) close (s)

    // De même
    typedef int SOCKET;
    typedef struct sockaddr_in SOCKADDR_IN;
    typedef struct sockaddr SOCKADDR;

#endif

// On inclut les fichiers standards
#include <stdio.h>
#include <stdlib.h>

int main(void)
{
    // Si la plateforme est MS-Windows
    #if defined (WIN32)
        WSADATA WSAData;
        WSAStartup(MAKEWORD(2,2), &WSAData);
    #endif

    // ICI on mettra le code sur les sockets

    // Si la plateforme est MS-Windows
    #if defined (WIN32)
        WSACleanup();
    #endif

    return EXIT_SUCCESS;
}
```

3. Manipulation de sockets

1. Tout d'abord, il faut créer une socket pour pouvoir configurer la connexion qu'elle va établir.
2. Ensuite, la paramétrer pour communiquer avec le client.
3. Enfin, fermer la connexion précédemment établie.

Exemple de la mise en œuvre du protocole :



NB : chaque action est associée à une fonction.

3.1. Créer une socket

Pour utiliser une socket, il va nous falloir le déclarer avec le type SOCKET :

SOCKET sock;

Pour la créer, il nous faudra utiliser la fonction socket avec le prototype suivant :

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

La fonction retourne une socket créée à partir des paramètres qui suivent.

- Le paramètre domain représente la famille de protocoles utilisée.

  Il prend la valeur AF_INET pour le protocole TCP/IP.

  Sinon, il prend la valeur AF_UNIX pour les communications UNIX en local sur une même machine.
- Le type indique le type de service, il peut avoir les valeurs suivantes :
  - SOCK_STREAM, si on utilise le protocole TCP/IP.
  - SOCK_DGRAM, si on utilise le protocole UDP/IP.

  Nous utiliserons donc la première (notez qu'il en existe d'autres comme SOCK_RAW).
- Dans le cas de la suite TCP/IP, le paramètre protocol n'est pas utile, on le mettra ainsi toujours à 0.

Comme dans notre cas nous utiliserons le protocole TCP/IP, notre fonction sera toujours :

```
sock = socket(AF_INET, SOCK_STREAM, 0);
```

3.2. Paramétrer une socket

Après avoir déclaré et créé la socket, il faut la paramétrer.

Pour cela, nous allons déclarer une structure de type SOCKADDR_IN qui va nous permettre de configurer la connexion. On l'appelle contexte d'adressage. Cette structure est définie de la façon suivante :

```
struct sockaddr_in
{
```

```
    short               sin_family;
    unsigned short      sin_port;
    struct in_addr      sin_addr;
    char                sin_zero[8];
};
```

Notez que la structure in_addr ne contient qu'un seul et unique champ nommé s_addr dont le type importe peu car nous n'y touchons pas directement (de plus celui-ci varie plus ou moins d'un système d'exploitation à un autre).

- sin.sin_addr.s_addr sera l'IP donnée automatiquement au serveur. Pour le connaître nous utiliserons la fonction htonl avec comme seul paramètre la valeur INADDR_ANY.

  Si vous voulez spécifier une adresse IP precise à utiliser, il est possible d'utiliser la fonction inet_addr avec comme seul paramètre l'IP dans une chaine de caractères :

  inet_addr("127.0.0.1");
- sin.sin_family sera toujours égal à AF_INET dans notre cas.
- Et sin.sin_port sera égal à la valeur retournée par la fonction htons, avec comme paramètre le port utilisé.
- Le champ sin_zero ne sera pas utilisé.

Nous allons la déclarer et l'initialiser comme ceci :

```
SOCKADDR_IN sin;

sin.sin_addr.s_addr = htonl(INADDR_ANY);
sin.sin_family = AF_INET;

sin.sin_port = htons(23);
```

3.3. Établir une connexion avec le client

Pour associer à la socket ces informations, nous allons utiliser la fonction :

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

La fonction retourne SOCKET_ERROR en cas d'erreur.

- Le paramètre socket désigne la socket du serveur avec laquelle on va associer les informations.
- Le paramètre addr est un pointeur de structure sockaddr du serveur.

  Il spécifie l'IP à laquelle on se connecte... Comme la fonction a besoin d'un pointeur sur structure sockaddr, et que nous disposons que d'une structure SOCKADDR_IN, nous allons faire un cast, pour éviter que le compilateur nous retourne une erreur lors de la compilation.
- Le paramètre addrlen sera la taille mémoire occupée par le contexte d'adressage du serveur (la structure SOCKADDR_IN).

Donc, nous ferons toujours ainsi :

```
bind(sock, (SOCKADDR*)&sin, sizeof(sin));
```

Maintenant, il va falloir mettre la socket dans un état d'écoute.

Pour cela, nous allons utiliser la fonction listen. Voici son prototype :

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

- La fonction retourne SOCKET_ERROR si une erreur est survenue.
- Le paramètre socket désigne la socket qui va être utilisée.
- Le paramètre backlog représente le nombre maximal de connexions pouvant être mises en attente.

Nous utiliserons donc la fonction ainsi :

listen(sock, 5);

En général, on met le nombre maximal de connexions pouvant être mises en attente à 5 (comme les clients FTP).

Enfin, on termine avec la fonction accept avec le prototype suivant :

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

Cette fonction permet la connexion entre le client et le serveur en acceptant un appel de connexion.

La fonction retourne la valeur INVALID_SOCKET en cas d'échec. Sinon, elle retourne la socket du client.

- Le paramètre socket est, comme dans les autre fonctions, la socket serveur utilisée.
- Le paramètre addr est un pointeur sur le contexte d'adressage du client.
- Le paramètre addrlen ne s'utilise pas comme dans la fonction bind ; ici, il faut créer une variable taille de type socklen_t (qui n'est rien d'autre qu'un entier), égale à la taille du contexte d'adressage du client. Ensuite, il faudra passer l'adresse de cette variable en paramètre.

On utilisera donc la fonction comme cela :

```
socklen_t taille = sizeof(csin);
csock = accept(sock, (SOCKADDR*)&csin, &taille);
```

Avec csock représentant la socket client et csin son contexte d'adressage.

Note : La fonction accept demande un type socklen_t* comme 3ème paramètre donc la variable taille doit être de type socklen_t.

La fonction accept est une fonction bloquante qui se termine que si un client se connecte.

3.4. Fermer la connexion

Finalement nous terminerons par la fonction closesocket qui permet de fermer une socket.

```
int closesocket(int sock);
```

4. Transmission d'une chaîne de caractères

Pour pouvoir réaliser une transmission de données, un programme serveur va devoir envoyer des données, et un programme client les recevoir.

Pour cela, nous allons utiliser trois fonctions :

1. La fonction send, qui va envoyer les données (sous forme de tableau de char).
2. La fonction recv, qui va recevoir ce qu'a envoyé la fonction send (sous forme de tableau de char).

3. La fonction shutdown, qui va désactiver les envois et les réceptions sur la socket.

4.1. La fonction send

Voici son prototype :

```
int send(int socket, void* buffer, size_t len, int flags);
```

La fonction retourne SOCKET_ERROR en cas d'erreur, sinon elle retourne le nombre d'octets envoyés.

- Le premier paramètre représente la socket destinée à recevoir le message.
- Le deuxième représente un pointeur (comme par exemple un tableau) dans lequel figureront nos informations à transmettre.
- Le paramètre len indique le nombre d'octets à lire.
- Le dernier correspond au type d'envoi ; il nous est inutile, nous le mettrons donc à 0 pour avoir un envoi normal.

La fonction est très simple : send(sock, buffer, sizeof(buffer), 0);

sizeof(buffer) n'est pas toujours la bonne valeur à mettre pour le troisième paramètre. Par exemple, pour les chaînes de caractères, on peut utiliser la fonction strlen pour connaître la taille de la chaîne (en lui ajoutant 1 pour le caractère '\0'). De même, pour un tableau, il faut passer en paramètre la taille totale du que prend le tableau (nombre de cases * taille d'une case).

4.2. La fonction recv

Maintenant, nous allons nous pencher sur l'application client. Pour pouvoir étudier maintenant la fonction recv.

Cette fonction est aussi simple que la fonction send, et son fonctionnement le même :

```
int recv(int socket, void* buffer, size_t len, int flags);
```

La fonction retourne SOCKET_ERROR en cas d'erreur, sinon elle retourne le nombre d'octets lus.

- Le premier paramètre représente la socket destinée à attendre un message.
- Le deuxième représente un pointeur (un tableau, par exemple) dans lequel résideront les informations à recevoir.
- Le paramètre len indique le nombre d'octets à lire.
- De même, le dernier correspond au type d'envoi : il nous est également inutile, nous le mettrons donc aussi à 0.

Nous recevrons les données envoyées : recv(sock, buffer, sizeof(buffer), 0);

Tout comme la fonction send, sizeof(buffer) n'est pas toujours la bonne taille à mettre pour le troisième paramètre. Pour cette fonction il ne faut pas mettre une valeur plus grande que la taille du tableau elle-même. Par exemple, pour les chaînes de caractères nous devrions envoyer d'abord un entier qui spécifie la taille de la chaîne puis envoyer la chaîne elle-même (sans le caractère '\0'). Une autre méthode consiste à lire les octets de la chaîne un à un jusqu'à se que le caractère '\0' soit trouvé mais cette méthode est moins performante que la précédente bien qu'elle réduise la taille des données envoyées.

4.3. La fonction shutdown

Voici le prototype de la dernière fonction : elle servira à fermer la transmission de données entre le serveur et le client.

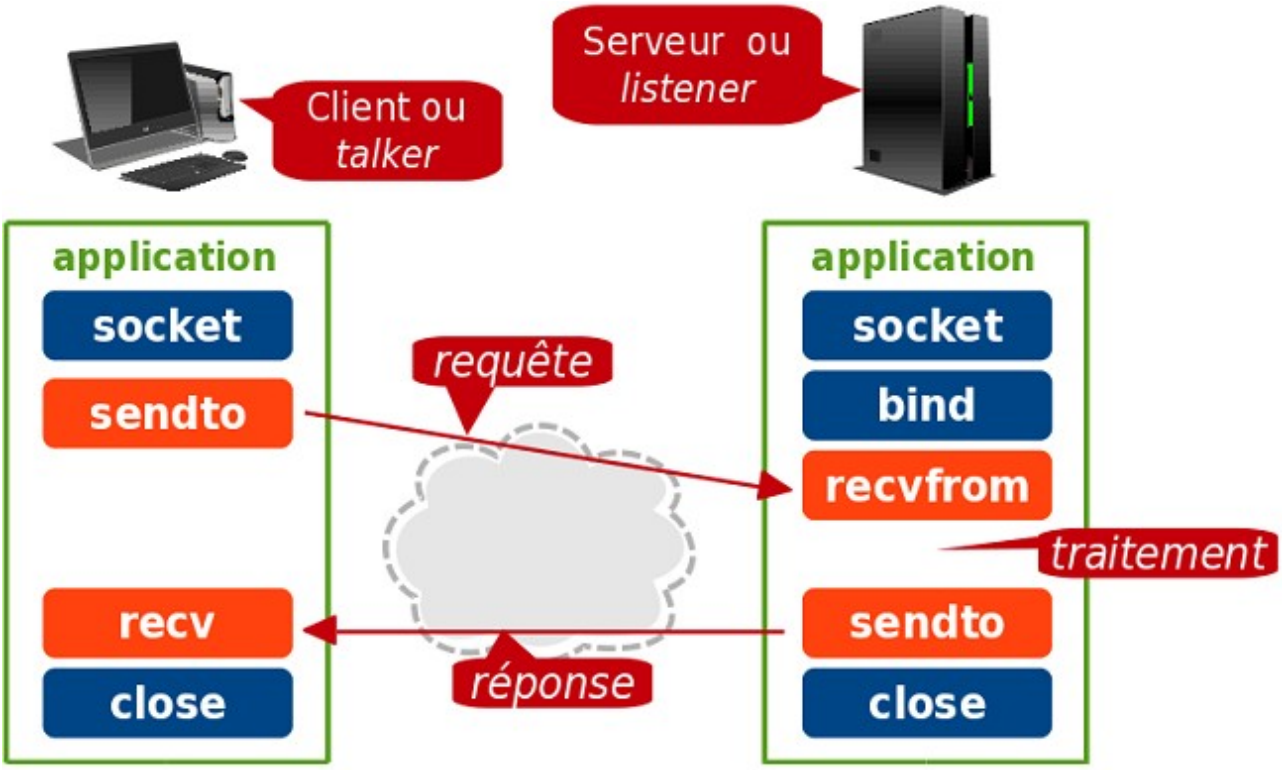
```
int shutdown(int socket, int how);
```

La fonction retourne la valeur -1 en cas d'erreur, sinon elle retourne la valeur 0.

- Le premier paramètre désigne sur quel socket on doit fermer la connection.
- Le deuxième paramètre définit où va se fermer la transition. Il peut prendre trois valeurs : 0, pour fermer la socket en réception, 1, en émission, 2 dans les deux sens.

Nous l'utiliserons ainsi, si l'on se place du côté du serveur : shutdown(sock, 2);

4.4. Exemple d'application Client/serveur



4.4.1. L'application CLIENT

```
#if defined (WIN32)
    #include <winsock2.h>
    typedef int socklen_t;
#elif defined (linux)
    #include <sys/types.h>
    #include <sys/socket.h>
    #include <netinet/in.h>
    #include <arpa/inet.h>
    #include <unistd.h>
    #define INVALID_SOCKET -1
    #define SOCKET_ERROR -1
    #define closesocket(s) close(s)
    typedef int SOCKET;
    typedef struct sockaddr_in SOCKADDR_IN;
    typedef struct sockaddr SOCKADDR;
```

```
#endif

#include <stdio.h>
#include <stdlib.h>

int main(void)
{
    const unsigned int PORT = 23;

    #if defined (WIN32)
        WSADATA WSAData;
        int erreur = WSAStartup(MAKEWORD(2,2), &WSAData);
    #else
        int erreur = 0;
    #endif

    /* test si les sockets Windows fonctionnent */
    if ( erreur )
        printf("Impossible de se connecter\n");
    else {
        /* Création de la socket */
        SOCKET sock = socket(AF_INET, SOCK_STREAM, 0);

        /* Configuration de la connexion */
        SOCKADDR_IN sin;
        sin.sin_addr.s_addr = inet_addr("127.0.0.1");
        sin.sin_family = AF_INET;
        sin.sin_port = htons(PORT);

        /* Si l'on a réussi à se connecter */
        if ( connect(sock, (SOCKADDR*)&sin, sizeof(sin)) != SOCKET_ERROR ) {
            printf("Connection a %s sur le port %d\n", inet_ntoa(sin.sin_addr),
htons(sin.sin_port));

            /* Si l'on reçoit des informations : on les affiche à l'écran */
            char buffer[128] = "";
            if ( recv(sock, buffer, 32, 0) != SOCKET_ERROR )
                printf("Recu : %s\n", buffer);
            }

        /* On ferme la socket */
        closesocket(sock);

        #if defined (WIN32)
            WSACleanup();
        #endif
        }

    return EXIT_SUCCESS;
}
```

4.4.1. L'application SERVEUR

```
#if defined (WIN32)
    #include <winsock2.h>
    typedef int socklen_t;
#elif defined (linux)
    #include <sys/types.h>
    #include <sys/socket.h>
    #include <netinet/in.h>
    #include <arpa/inet.h>
    #include <unistd.h>
    #define INVALID_SOCKET -1
    #define SOCKET_ERROR -1
    #define closesocket(s) close(s)
```

```
    typedef int SOCKET;
    typedef struct sockaddr_in SOCKADDR_IN;
    typedef struct sockaddr SOCKADDR;
#endif

#include <stdio.h>
#include <stdlib.h>
#include <string.h>


int main(void)
{
    const unsigned int PORT = 23;
    const char buffer[] = "Bonjour !";

    #if defined (WIN32)
        WSADATA WSAData;
        int erreur = WSAStartup(MAKEWORD(2,2), &WSAData);
    #else
        int erreur = 0;
    #endif

    /* test si les sockets Windows fonctionnent */
    if ( erreur )
        printf("Impossible de se connecter\n");
    else {
        SOCKET sock = socket(AF_INET, SOCK_STREAM, 0);

        /* Si la socket est valide */
        if ( sock != INVALID_SOCKET ) {
            printf("La socket %d est maintenant ouverte en mode TCP/IP\n", sock);

            /* Configuration */
            SOCKADDR_IN sin;
            sin.sin_addr.s_addr    = htonl(INADDR_ANY);   /* Adresse IP automatique */
            sin.sin_family         = AF_INET;             /* Protocole familial (IP) */
            sin.sin_port           = htons(PORT);         /* Ecoute du port */
            int sock_err = bind(sock, (SOCKADDR*)&sin, sizeof(sin));

            /* Si la socket fonctionne */
            if ( sock_err != SOCKET_ERROR ) {
                /* Démarrage du listage (mode server) */
                sock_err = listen(sock, 5);
                printf("Ecoute du port %d...\n", PORT);

                /* Si la socket fonctionne */
                if ( sock_err != SOCKET_ERROR ) {
                    /* Attente pendant laquelle le client se connecte */
                    printf("Patientez pendant que le client se connecte sur le port
%d...\n", PORT);

                    SOCKADDR_IN csin;
                    socklen_t recsize = sizeof(csin);

                    SOCKET csock = accept(sock, (SOCKADDR*)&csin, &recsize);
                    printf("Un client se connecte avec la socket %d de %s:%d\n", csock,
inet_ntoa(csin.sin_addr), htons(csin.sin_port));

                    sock_err = send(csock, buffer, 32, 0);

                    if ( sock_err != SOCKET_ERROR )
                        printf("Chaine envoyee : %s\n", buffer);
                    else
                        printf("Erreur de transmission\n");
```

```
                    /* fermeture de la connexion (fermee dans les deux sens) */
                    shutdown(csock, 2);
                    }
                }

            /* Fermeture de la socket */
            printf("Fermeture de la socket...\n");
            closesocket(sock);
            printf("Fermeture du serveur terminee\n");
            }

        #if defined (WIN32)
            WSACleanup();
        #endif
        }

    return EXIT_SUCCESS;
}
```

5. Un problème de portabilité

La taille d'une variable de type int peut varier d'un ordinateur à un autre. De même, l'ordre des octets d'une variable codée sur plusieurs octets n'est pas toujours le même non plus.

Il y a plusieurs façon de représenter un groupe d'octets en mémoire, on peut commencer par l'octet de poids fort ou par l'octet de poids faible par exemple, cela s'appelle l'Endianness (ou boutisme en français). Si un groupe d'octet commence par l'octet de poids fort on dit que sont orientation est big-endian, s'il commence par l'octet de poids faible on dit que sont orientation est little-endian. La façon d'organiser un groupe d'octet en mémoire dépend de l'architecture de la machine.

Pour réaliser une communication portable entre deux ordinateurs, il faut absolument transmettre les octets des variables un à un selon un ordre donné qui est le même pour les deux applications distantes.

5.1. Fonctions de conversion

Deux groupes de fonctions de conversion de l'ordre des octets existe. Le premier groupe de fonctions a pour but de convertir un entier qui à l'endianness de votre ordinateur en un entier qui à l'endianness du réseau qui est toujours en big-endian (octet de poids fort en première position). Le second groupe de fonctions a pour but de faire la même opération mais dans le sens opposé.

- `unsigned long htonl(unsigned long hostlong);`
- `unsigned short htons(unsigned short hostshort);`
- `unsigned long ntohl(unsigned long netlong);`
- `unsigned short ntohs(unsigned short netshort);`

La fonction htonl convertit l'entier de 4 octets hostlong depuis l'ordre des octets de l'hôte vers celui du réseau.

La fonction htons convertit l'entier de 2 octets hostshort depuis l'ordre des octets de l'hôte vers celui du réseau.

La fonction ntohl convertit l'entier de 4 octets netlong depuis l'ordre des octets du réseau vers celui de l'hôte.

La fonction ntohs convertit l'entier de 2 octets netshort depuis l'ordre des octets du réseau vers celui de l'hôte.

Exemple : on veut transmettre un entier codée sur 4 octets à un autre ordinateur de manière portable. Il va falloir décomposer l'entier en 4 parties et envoyer chaque octet un à un.

De même pour la réception sauf qu'il va juste falloir faire l'opération inverse.

```
void send4(int sock, unsigned long data)
{
    // On convertit data en entier big-endian
    long dataSend = htonl(data);

    // On envoie l'entier convertit
    send(sock, (char*)&dataSend, 4, 0);
}

void read4(int sock, unsigned long* data)
{
    long dataRecv;

    // On récupère l'entier en big-endian
    recv(sock, (char*)&dataRecv, 4, 0);

    // On convertit l'entier récupéré en little-endian si l'ordinateur
    // stock les entiers en mémoire en little-endian, sinon s'il les
    // stock en big-endian l'entier est convertit en big-endian
    *data = ntohl(dataRecv);
}
```

Remarque : pour envoyer un entier de 2 octets le fonctionnement est exactement le même.

Pour ce qui est de la transmission de structures, il faudra les convertir en chaînes de caractères.

6. La sélection de sockets

La sélection de sockets permet de manipuler plusieurs sockets avec un thread.

6.1. Threads

Un même processus peut se décomposer en plusieurs parties, qui vont s'exécuter simultanément en partageant les mêmes données en mémoire. Ces parties se nomment threads.

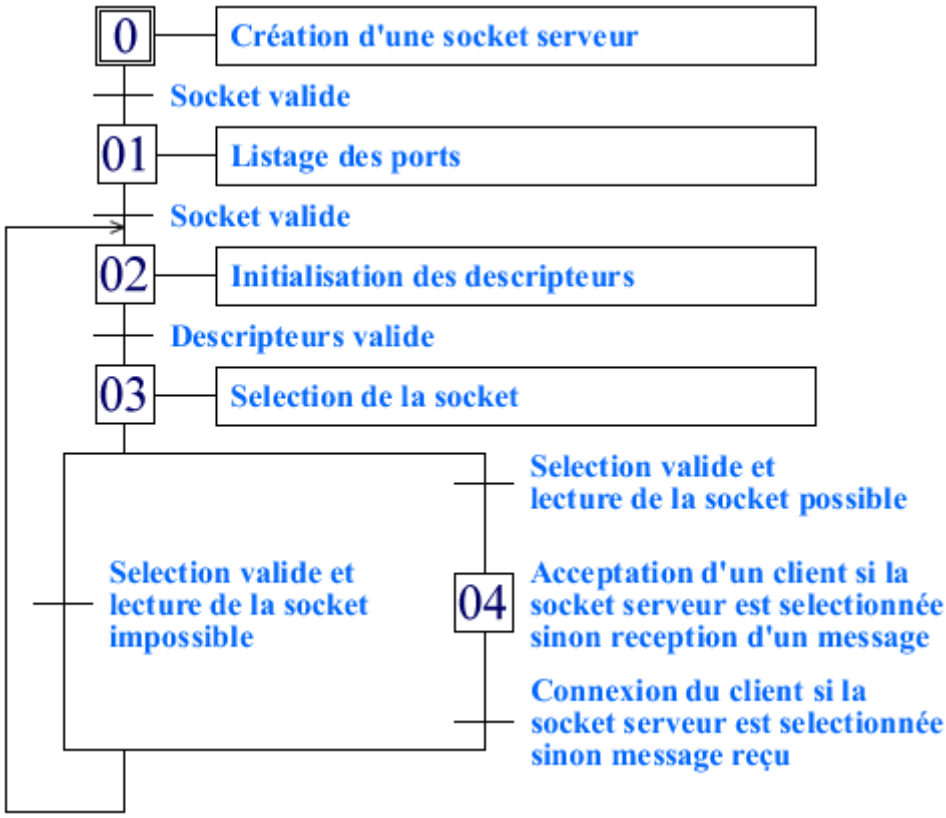
Du point de vue de l'utilisateur, les threads semblent se dérouler en parallèle.

Lorsqu'une fonction bloque par exemple un programme (comme la fonction recv), si celui-ci dispose d'une interface graphique, il sera inactif tant que la fonction le bloquera.

6.2. Le fonctionnement global

La sélection de sockets s'inscrit dans un fonctionnement événementiel, c'est à dire que tout se fait dans un seul thread et dans un seul et même processus.

Avec la sélection de sockets, nous avons un schéma similaire à celui-ci :



Notez que le schéma ci-dessus est un grafcet :

- Chaque rectangle désigne donc une action (étape) repérée par un nombre unique.
- Chaque barre entre les actions désigne la condition pour que l'action suivante se réalise.
- Les étapes se déroulent dans l'ordre (l'étape 3 se déroule après l'étape 2 et ainsi de suite).
- On commence toujours par l'étape initiale (celle qui porte le numéro d'étape 0 et qui est encadré dans deux rectangles).

On crée une socket serveur, on liste les ports, puis on initialise les descripteurs. Ensuite, on sélectionne la ou les socket(s) voulue(s) et pour chaque socket sélectionnée, on regarde dans quel état elle se trouve (y a t-il des données à lire ? à écrire ? etc.). Le tout ce fait dans un seul thread et dans un seul processus.

Notez que la sélection de sockets est bloquante pendant un temps que vous spécifiez ou non, c'est à dire que tant que l'état des descripteurs ne change pas ou tant que le temps donné n'est pas dépassé, la sélection reste bloquante. Si vous ne spécifiez pas de temps alors seul un changement d'état des descripteurs débloquera la sélection.

6.3. Le descripteur de socket

Un descripteur de socket est tout simplement une variable (un entier) qui nous servira à manipuler la socket. L'état de cet entier peut nous permettre de connaître si des données ont été reçues ou envoyées sur la socket. Le type de variable SOCKET est lui même un type de descripteur de socket. Le type SOCKET n'est donc qu'un entier (int), néanmoins on préfère utiliser le type SOCKET pour mieux comprendre les choses et respecter les normes.

Pour initialiser les descripteurs, nous allons utiliser des fonctions. Ces fonctions nous permettront

de lier une ou plusieurs sockets à des ensembles. Par exemple, si nous voulons que la sélection d'une socket cliente soit bloquante jusqu'à se qu'elle reçoive des données en lecture, alors nous allons initialiser un ensemble de lecture et nous allons lui ajouter cette socket. Si l'ensemble en lecture est vide cela voudra dire qu'il n'y a rien à lire sur la socket. A l'inverse, si l'ensemble n'est pas vide cela signifie que la socket a reçu des données et que nous pouvons les lire.

De même, si nous voulons par exemple que deux sockets clientes bloquent la sélection jusqu'à ce qu'elles reçoivent des données en lecture, il suffira d'ajouter ces deux sockets à un même ensemble de lecture.

Un ensemble est un type de variable permettant de connaître l'état du descripteur de socket. Il en existe trois :

- L'ensemble de lecture readfds, il permet de savoir si le client a envoyé des données sur la socket sélectionnée. Un appel à recv ne sera donc pas bloquant.
- L'ensemble de écriture writefds, il permet de savoir si le client a reçu les données sur la socket sélectionnée. Un appel à send ne sera donc pas bloquant.
- L'ensemble d'exception exceptfds, il permet de gérer les exceptions.

6.3.1. L'initialisation des descripteurs

Pour faire cela nous utiliserons à quatre fonctions présenté ci-dessous.

- FD_SET(int fd, fd_set* set);

  Cette fonction ajoute le descripteur fd à l'ensemble set.

  Le descripteur fd n'est rien d'autre qu'une socket mais comme dit plus haut, une socket est avant tout un type int.

- FD_ISSET(int fd, fd_set* set);

  Cette fonction vérifie si le descripteur fd est contenu dans l'ensemble set après l'appel à select.

  Par exemple, si l'ensemble set est un ensemble de lecture la fonction servira à savoir si la socket fd a reçu des données.

- FD_CLR(int fd, fd_set *set);

  Cette fonction supprime le descripteur fd de l'ensemble set.

  Cette fonction est beaucoup moins utilisé que les trois autre mais n'en n'est pas pour au temps inutile.

- FD_ZERO(fd_set *set);
- Cette fonction vide l'ensemble set. Cela revient à supprimer tout les descripteurs ajouté précédemment à l'ensemble.

6.3.2. La sélection de la socket

La sélection de sockets se fait via la fonction select qui détient le prototype suivant :

```
int select(int fdmax, fd_set *readfds, fd_set *writefds, fd_set *exceptfds,
struct timeval *timeout);
```

En cas de réussite la fonction retourne le nombre de descripteurs dans les ensembles. Si la fonction rend la main à l'application car le timeout a expiré alors elle retourne 0, sinon en cas d'erreur la

fonction retourne -1.

- Le paramètre fdmax correspond au descripteur de socket le plus grand auquel on ajoute un.

  Une fois que vous avez ajouté des descripteurs de sockets au ensembles vous allez chercher le descripteur le plus grand (vous savez maintenant que les descripteurs de sockets sont de simples entiers avant tout ^^) et le passer en paramètre à la fonction select tout en lui ajoutant un.
- Le paramètre readfds correspond à l'ensemble de lecture. Si on ne veut pas recevoir des données sur aucune des sockets sélectionnées, on peut mettre ce paramètre à NULL.
- Le paramètre writefds correspond à l'ensemble d'écriture. Si on ne veut pas envoyer des données sur aucune des sockets sélectionnées, on peut mettre ce paramètre à NULL.
- Le paramètre exceptfds correspond à l'ensemble d'exception. Nous le mettrons à NULL car en général, nous ne l'utiliserons pas.
- Le paramètre timeout est une structure qui contient le temps limite d'attente de blocage de la fonction. En général, nous le mettrons à NULL ce paramètre pour que la fonction reste bloquante tant qu'elle ne reçoit pas de changements d'états des descripteurs.

Notez que la recherche du descripteur de socket le plus grand n'est pas toujours très rapide sur des serveurs qui peuvent avoir des centaines ou même milliers de clients. On préférera alors faire la recherche du plus grand descripteur seulement quand un client quitte le serveur ou qu'un autre se connecte au lieu de le calculer à chaque fois avant l'utilisation de la fonction select.

Notez aussi que la fonction select peut modifier les ensembles qui lui sont passés en paramètres. Nous redéfinirons alors à chaque fois les descripteurs associés au ensembles avant l'utilisation de la fonction select

6.3.3. Exemple

Voici un exemple de serveur multi-clients utilisant la sélection de sockets.

Le client se connecte au serveur puis est immédiatement déconnecté de celui-ci :

```
#if defined (WIN32)
    #include <winsock2.h>
    typedef int socklen_t;
#elif defined (linux)
    #include <sys/types.h>
    #include <sys/socket.h>
    #include <netinet/in.h>
    #include <arpa/inet.h>
    #include <unistd.h>
    #define INVALID_SOCKET -1
    #define SOCKET_ERROR -1
    #define closesocket(s) close (s)
    typedef int SOCKET;
    typedef struct sockaddr_in SOCKADDR_IN;
    typedef struct sockaddr SOCKADDR;
#endif

#include <stdio.h>
#include <stdlib.h>
#include <errno.h>

int main(void)
{
    const unsigned int PORT = 23;
```

```
    #if defined (WIN32)
        WSADATA WSAData;
        int erreur = WSAStartup(MAKEWORD(2,2), &WSAData);
    #else
        int erreur = 0;
    #endif

    if ( !erreur ) {
        SOCKET sock = socket(AF_INET, SOCK_STREAM, 0);

        if ( sock != INVALID_SOCKET ) {
            printf("La socket %d est maintenant ouverte en mode TCP/IP\n", sock);

            SOCKADDR_IN sin;
            int recsize = sizeof sin;

            sin.sin_addr.s_addr = htonl(INADDR_ANY);
            sin.sin_family = AF_INET;
            sin.sin_port = htons(PORT);
            int sock_err = bind(sock, (SOCKADDR*) &sin, recsize);

            if ( sock_err != SOCKET_ERROR ) {
                sock_err = listen(sock, 5);
                printf("Ecoute du port %d...\n", PORT);

                if ( sock_err != SOCKET_ERROR ) {
                    /* Création de l'ensemble de lecture */
                    fd_set readfs;

                    for (;;) {
                        /* On vide l'ensemble de lecture et on lui ajoute
                        la socket serveur */
                        FD_ZERO(&readfs);
                        FD_SET(sock, &readfs);

                        /* Si une erreur est survenue au niveau du select */
                        if ( select(sock + 1, &readfs, NULL, NULL, NULL) < 0 ) {
                            perror("select()");
                            exit(errno);
                            }

                        /* On regarde si la socket serveur contient des
                        informations à lire */
                        if ( FD_ISSET(sock, &readfs) ) {
                            /* Ici comme c'est la socket du serveur cela signifie
                            forcement qu'un client veut se connecter au serveur.
                            Dans le cas d'une socket cliente c'est juste des
                            données qui serons reçues ici*/

                            SOCKADDR_IN csin;
                            int crecsize = sizeof csin;

                            /* Juste pour l'exemple nous acceptons le client puis
                            nous refermons immédiatement après la connexion */
                            SOCKET csock = accept(sock, (SOCKADDR *) &csin, &crecsize);
                            closesocket(csock);

                            printf("Un client s'est connecte\n");
                            }
                        }   // end for ever
                    }   // endif SOCKET_ERROR
                }
            }   // endif INVALID_SOCKET
        }   // endif erreur
```

```
    #if defined (WIN32)
        WSACleanup();
    #endif

    return EXIT_SUCCESS;
}
```

Notez qu'une socket serveur reçoit des données en lecture que quand un client se connecte à celui-ci. Bien que, les fonctions recv et accept soit bloquantes en temps normale, ici, elles ne le sont plus car on les appellent lorsqu'il le faut (par exemple, on sait que la fonction recv ne sera pas bloquante si des données viennent d'être reçues).

La sélection de sockets est présentée, ici, pour une application serveur mais sachez que le principe fonctionne aussi avec les applications clientes.

7. Bibliothèques

Il existe un certain nombre de bibliothèques pour faciliter la gestion des sockets. Voici les plus connues.

- libcurl : libcurl est une bibliothèque de transfert de fichier multiprotocole. Elle inclut entre autres les protocoles HTTP, FTP, HTTPS, SCP, TELNET... La gestion des sockets est faite en interne.
- GNet : GNet est une bibliothèque écrite en C, orientée objet et basée sur la GLib. Elle inclut entre autres un système de client/serveur TCP, des sockets multicasts UDP, des sockets asynchrones...
- SDL_net : SDL_net est une bibliothèque permettant d'accéder au réseau de manière portable. Elle inclut les protocoles UDP et TCP avec des systèmes de client/serveur.