Tutoriel : Les sockets

Version en ligne (../../../www.siteduzero.com/informatique/tutoriels/les-sockets)

Table des matières

Les sockets

Dans ce tutoriel, nous allons apprendre à utiliser les sockets avec le protocole TCP/IP, une communication logique entre des systèmes reliés au réseau (http://fr.wikipedia.org/wiki/Réseau_informatique) internet, ou entre des applications en local...
Il faudra comme pré-requis connaître les deux premières parties du cours de M@teo21 sur le langage C (http://www.siteduzero.com/tuto-3-8-0-apprenez-a-programmer-en-c-c.html).

> Malheureusement, comme je ne connais pas Mac OS, je ne pourrai aider que les utilisateurs de Linux et Windows tout au long de ce tutoriel. Cependant, le fonctionnement des sockets Mac doit être le même que sous Linux, car ces deux systèmes ont une même base : UNIX (http://fr.wikipedia.org/wiki/UNIX).

Histoire et fonctionnement

Dans ce chapitre vous apprendrez presque tout sur l'histoire des sockets.
Vous verrez comment elles fonctionnent mais aussi quand et pourquoi elles ont été créées.
Nous nous attaquerons ensuite à la partie préprocesseur du code qui nous sera indispensable par la suite dans tous les projets avec des sockets.

Un peu d'histoire

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 (http://fr.wikipedia.org/wiki/UNIX), les sockets sont des points de terminaison mis à l'écoute sur le réseau (http://fr.wikipedia.org/wiki/Réseau_informatique), afin de faire transiter des données logicielles.
Celles-ci sont associées à un numéro de port (http://fr.wikipedia.org/wiki/Port_(logiciel)).
Les ports sont des numéros allant de 0 à 216-1 inclus (soit 65535 :p). 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 (http://fr.wikipedia.org/wiki/Protocole_de_communication). Vous avez sûrement déjà entendu parler des protocoles UDP/IP (http://fr.wikipedia.org/wiki/User_Datagram_Protocol) et TCP/IP (http://fr.wikipedia.org/wiki/Transmission_Control_Protocol), sinon renseignez-vous ;) . Dans notre cas nous utiliserons le protocole TCP/IP.
Les sockets servent à établir une transmission de flux de données (octets (http://fr.wikipedia.org/wiki/Octets)) entre deux machines ou applications.

> *C'est bien tout ça, mais à quoi ça sert exactement :p ?*

Eh bien en C, vous avez sûrement plus d'une fois eu besoin d'un moyen de communication entre deux programmes, non ?
Dans ce cas on utilise parfois des fichiers qui servent de "passerelle" ^^ , mais on passe souvent à côté des sockets qui peuvent le faire aussi bien ;) (je dirai même mieux :p).
Le principal atout des sockets est que les informations sont transmises directement au programme voulu en plus d'être plus sécurisées que les fichiers.
Elles servent bien plus qu'on ne le pense...
Par exemple, le langage PHP (http://www.siteduzero.com/tutoriel-3-14668-un-site-dynamique-avec-php.html#part_14667) illustre très bien les sockets, car il utilise ce principe "Client / Serveur".
Ou bien même, quand vous naviguez sur Internet dans une page XHTML (http://www.siteduzero.com/tutoriel-3-13666-apprenez-a-creer-votre-site-web.html#part_13665), votre navigateur va utiliser les sockets pour demander au serveur le code source (http://fr.wikipedia.org/wiki/Code_source) de la page, pour pouvoir afficher ce qu'elle contient :

> Ce schéma est TRES simplifié ! Il n'est pas complet, mais permet d'avoir une vision simplifiée du principe. Pour le moment nous nous contenterons de cela.

Elles servent aussi pour tout ce qui touche au réseau (http://fr.wikipedia.org/wiki/Réseau_informatique).
Vous voyez donc que les exemples sont multiples; je ne les citerai pas tous :p .

Leurs fonctionnement

Les sockets ne s'utilisent pas de manière identique selon les différents systèmes d'exploitation : je vais donc vous guider tout le long de ce chapitre pour que vous ne quittiez pas le bon chemin :p .

Les inclusions et les ressources

Sur Windows

Tout d'abord, n'oubliez pas, dans **chaque projet** que vous créez, d'ajouter le fichier "ws2_32.lib" (pour le compilateur Visual C++) ou "libws2_32.a" (pour les autres) dans votre éditeur de liens. Vous trouverez ce fichier dans le dossier "lib" de votre IDE. J'insiste un peu, car on oublie très souvent de le faire :p .

Il faut savoir que presque tout ce qui touche aux sockets Windows se trouve dans le fichier "winsock2.h", dans le dossier *header* de votre IDE. Celui-ci est un fichier standard de Windows, il n'y a pas besoin de le télécharger ;) .
Nous allons donc tout de suite l'inclure dans notre premier programme comme suit :

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

En général, vous aurez besoin des fichiers standards "stdio.h" et "stdlib.h".
Nous allons donc aussi les inclure :

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

On peut remarquer que le type socklen_t qui existe sous Linux, n'est pas défini sous 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 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, vous devrez ajouter, dans le début de votre fonction *main*, le code suivant pour pouvoir utiliser les sockets sous Windows :

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

La fonction WSAStartup (http://msdn2.microsoft.com/en-us/library/ms742213.aspx) 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 (ici la version 2,0). Elle retourne la valeur 0 si tout s'est bien passé.
Puis à la fin, placez celui-ci :

```
WSACleanup();
```

Cette fonction va simplement libérer les ressources allouées par la fonction WSAStartup().

Sur Linux

Sur Linux, c'est un peu différent puisque les fichiers à inclure ne sont pas les mêmes...
Pour combler l'écart entre Windows et 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 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 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 Linux par rapport au fichier "winsock2.h" de Windows.
Voila donc le contenu de notre 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;
```

Sachez qu'il y a beaucoup de fichiers à inclure par rapport à Windows mais qu'ils sont tous utiles.

Un code portable

Pour pouvoir avoir un code un peu plus portable, nous utiliserons les définitions WIN32 et linux.
Cette méthode indiquera à votre compilateur le code à compiler en fonction de votre OS.

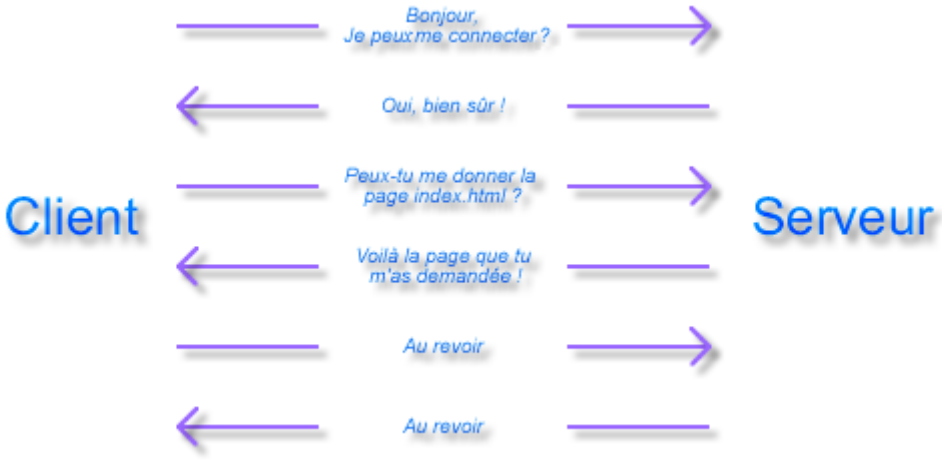
```
//Si nous sommes sous Windows
#if defined (WIN32)

    #include <winsock2.h>

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

// Sinon, si nous sommes sous 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 Windows
    #if defined (WIN32)
        WSADATA WSAData;
        WSAStartup(MAKEWORD(2,2), &WSAData);
    #endif

    // ICI on mettra notre code sur les sockets

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

    return EXIT_SUCCESS;
}
```

Manipulation de sockets

Dans cette partie, nous verrons qu'il n'est pas si difficile de manipuler les sockets en C, et que leur utilisation peut s'avérer très pratique dans bon nombre de cas. ^^

Partie 1 : L'application serveur

Pour pouvoir utiliser pleinement les sockets, nous allons suivre une démarche précise ^^ :
Tout d'abord, nous allons **créer une socket** pour pouvoir configurer la connexion qu'elle va établir.
Ensuite, nous allons **la paramétrer** pour **communiquer avec le client**.
Enfin, nous allons **fermer la connexion** précédemment établie.
Je ne sais pas si vous vous rappelez du schéma que j'avais fait dans le chapitre précédent, si c'est le cas oubliez-le et sinon tant mieux :p .
Voici comment cela va se passer vraiment si l'on reprend l'ancien schéma :



(Notez que ce schéma est toujours simplifié car le client ne va pas dire "Bonjour" et le serveur ne va pas répondre "oui, bien sûr" :p ... tout se fait par données.)
Chaque action est associée à une fonction que nous allons voir dans ce chapitre ;) .

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 (http://www.gnu.org/software/libc/manual/html_node/Creating-a-Socket.html#Creating-a-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 mais ils nous seront inutiles).
- 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);
```

Paramétrer une socket

Après avoir déclaré et créé la socket, nous allons 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 (en savoir plus (http://www.gnu.org/software/libc/manual/html_node/Address-Formats.html#Address-Formats)).
- 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);
```

Établir une connexion avec le client

Enfin, 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 (en savoir plus (http://www.gnu.org/software/libc/manual/html_node/Setting-Address.html#Setting-Address)).
- 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 (notre structure SOCKADDR_IN), nous utiliserons donc *sizeof*;) (si vous ne vous rappelez plus du cours de m@teo21, je vous conseil de relire le cours sur l'allocation dynamique (http://www.siteduzero.com/tuto-3-4830-1-l-allocation-dynamique.html#ss_part_1) :p).

Donc, nous ferons toujours ainsi :

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

Voilà ! Maintenant que toutes les informations sont données, il va falloir mettre la socket dans un état d'écoute (établir la connexion, si vous préférez :p).
Pour cela, nous allons utiliser la fonction listen (http://www.gnu.org/software/libc/manual/html_node/Listening.html#Listening). 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 notre 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 (http://www.gnu.org/software/libc/manual/html_node/Accepting-Connections.html#Accepting-Connections) 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. Pour le moment cela ne nous gène pas puisque nous sommes sous la console, mais après, quand nous ferrons des applications fenêtrées, il va falloir gérer les threads (http://fr.wikipedia.org/wiki/Processus_léger). Vous devez juste retenir qu'ils servent à faire plusieurs choses en parallèle dans une même application ;) . Ne vous inquiétez pas j'aborderai les threads dans la suite du cours.

Fermer la connexion

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

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

Son prototype est très simple, je pense donc que la fonction se passe de commentaires :-° .

On récapitule

Nous allons réaliser une application qui va attendre qu'un client se connecte à celle-ci. Bien sûr, comme nous n'avons pas encore fait l'application "client", notre application (qui jouera le rôle de serveur) ne pourra pas établir de connexion... Nous verrons ensuite les fonctions relatives au "client", ce qui nous permettra de réaliser une vraie connexion.
Sachez cependant que la partie serveur était la plus difficile et la plus longue ;) .

Réfléchissez un peu à l'ordre d'utilisation des fonctions, passez un peu de temps dessus.
Voir la solution directement ne vous aidera pas ^^ .

> Il ne faut pas oublier que chaque client et serveur contient une socket et un contexte d'adressage pour lui seul. Notez qui est possible d'avoir plusieurs sockets serveur sur pour une même application.

```
#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>
#define PORT 23



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

/* Socket et contexte d'adressage du serveur */
SOCKADDR_IN sin;
SOCKET sock;
socklen_t recsize = sizeof(sin);

/* Socket et contexte d'adressage du client */
SOCKADDR_IN csin;
SOCKET csock;
socklen_t crecsize = sizeof(csin);

int sock_err;


if(!erreur)
{
    /* Création d'une 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 */
    sin.sin_addr.s_addr = htonl(INADDR_ANY);  /* Adresse IP automatique */
    sin.sin_family = AF_INET;                 /* Protocole familial (IP) */
    sin.sin_port = htons(PORT);               /* Listage du port */
    sock_err = bind(sock, (SOCKADDR*)&sin, recsize);

    /* Si la socket fonctionne */
    if(sock_err != SOCKET_ERROR)
    {
        /* Démarrage du listage (mode server) */
        sock_err = listen(sock, 5);
        printf("Listage 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);
            csock = accept(sock, (SOCKADDR*)&csin, &crecsize);
            printf("Un client se connecte avec la socket %d de %s:%d\n", csock, inet_ntoa(csin.sin_addr), htons(csin.sin_port));
        }
        else
```

```
                perror("listen");
            }
            else
                perror("bind");

            /* Fermeture de la socket client et de la socket serveur */
            printf("Fermeture de la socket client\n");
            closesocket(csock);
            printf("Fermeture de la socket serveur\n");
            closesocket(sock);
            printf("Fermeture du serveur terminée\n");
        }
        else
            perror("socket");

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

    return EXIT_SUCCESS;
}
```

Le code n'est pas complexe quand on connaît les fonctions qu'il utilise ^^ .

> Vous devez être en mode administrateur (root sous linux) pour faire fonctionner un programme serveur sinon cela risque de ne pas fonctionner.

Partie 2 : L'application client

Normalement, tout devrait bien se passer puisque cette partie est plus simple :p .
Sachez que les inclusions et les définitions se conservent, et donc qu'il n'y aura qu'une partie de la fonction principale qui changera ;) . Maintenant, nous allons réaliser l'application qui va jouer le rôle du client. Pour cela, créez un nouveau projet.

Eh bien : récapitulons ce que nous savons faire :

- Créer une socket.
- Associer une Socket à un point de terminaison local.
- Mettre une Socket en état d'écoute.
- Accepter un appel de connexion avec un client.
- Fermer la connexion Socket, et libérer toutes les ressources associées.

C'est bien d'accepter un appel, mais faut déjà commencer par faire une requête :p . Et oui, nous ne savons pas établir une connexion du côté client ! Pour cela nous allons utiliser la fonction connect (http://www.gnu.org/software/libc/manual/html_node/Connecting.html#Connecting).
Son prototype est le suivant :

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

- La fonction retourne 0 si la connexion s'est bien déroulée, sinon -1.
- Le paramètre **socket** représente la socket à utiliser (ça n'a toujours pas changé :p).
- Le paramètre **addr** représente l'adresse de l'hôte à contacter. On va faire un cast comme avec la fonction *accept*.
- Le dernier paramètre, **addrlen**, représente la taille de l'adresse de l'appelant (un sizeof suffira ^^).

On va appeler notre fonction comme cela :

```
connect(sock, (SOCKADDR*)&sin, sizeof(sin))
```

Avec la structure **sin** précédemment déclarée et initialisée.

> Sachez que pour l'application client, il n'y aura besoin d'utiliser ni la fonction *bind* puisqu'elle est comprise dans la fonction *connect*, ni *listen* puisqu'il n'y a pas de sockets à mettre à l'écoute, ni encore *accept* puisque l'application joue le rôle de client.

Et voila c'est finie pour les nouvelles fonctions de notre application client ^^ . Je vous avez dit que cette partie était plus simple :p .
Maintenant que vous savez tout ce dont vous avez besoin, vous pouvez commencer à réfléchir au code.

```
#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>
#define PORT 23



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

SOCKET sock;
SOCKADDR_IN sin;

if(!erreur)
{
    /* Création de la socket */
    sock = socket(AF_INET, SOCK_STREAM, 0);

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

    /* Si le client arrive à se connecter */
    if(connect(sock, (SOCKADDR*)&sin, sizeof(sin)) != SOCKET_ERROR)
        printf("Connexion à %s sur le port %d\n", inet_ntoa(sin.sin_addr), htons(sin.sin_port));
    else
        printf("Impossible de se connecter\n");

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

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

    return EXIT_SUCCESS;
}
```

Ce code affiche cela si tout se passe bien :

```
Connexion à 127.0.0.1 sur le port 23
```

Ce code va créer une socket et va essayer de se connecter sur une application serveur en local, une seule fois, puis va se fermer.

Vous pouvez maintenant tester votre programme serveur :D .
Lancez votre programme serveur en premier, puis votre programme client.
En effet : si vous faites le contraire, votre programme client va essayer de se connecter au programme serveur alors que celui-ci n'est pas lancé... Comme il ne se connecte qu'une seule fois, le programme client se fermera alors que le programme serveur, lui, attendra une connexion du client.

Votre programme serveur affiche donc ça avant d'être fermé :

```
La socket xxxx est maintenant ouverte en mode TCP/IP
Listage du port 23...
Patientez pendant que le client se connecte sur le port 23...
Un client se connecte avec la socket xxxx de 127.0.0.1:xxxx
Fermeture de la socket
Fermeture du serveur terminée
```

Si ce n'est pas le cas, vérifiez que vous ne vous êtes pas trompés précédemment dans le code ;) .

Exercice

Maintenant, vous pouvez réaliser une connexion du client en boucle. Puis, une fois connectés, vous fermez les deux programmes.

Ajoutez simplement une boucle au code qui précède :p !

Ce chapitre est dur à comprendre, et long. Si vous avez du mal, n'hésitez pas à le relire plusieurs fois ;) .

Dans le chapitre suivant, nous verrons comment transmettre des variables et des chaînes de caractères du serveur au client (on pourra faire un chat :p).

La transmission de flux

> Dans ce chapitre, nous allons apprendre à transmettre des flux d'octets du serveur au client.
> Vous verrez que ce que l'on va apprendre est le coeur d'une communication entre le Client et le Serveur. Pour cela, nous allons devoir découvrir d'autres fonctions :p .

Transmission d'une chaîne de caractères

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

Pour cela, nous allons utiliser trois fonctions :

- La fonction send (http://www.gnu.org/software/libc/manual/html_node/Sending-Data.html#Sending-Data), qui va envoyer les données (sous forme de tableau de char).
- La fonction recv (http://www.gnu.org/software/libc/manual/html_node/Receiving-Data.html#Receiving-Data), qui va recevoir ce qu'a envoyé la fonction *send* (sous forme de tableau de char).
- La fonction shutdown (http://www.gnu.org/software/libc/manual/html_node/Closing-a-Socket.html#Closing-a-Socket), qui va désactiver les envois et les réceptions sur la socket.

Premièrement, nous allons revenir sur le code source du programme serveur, pour étudier le fonctionnement de la fonction *send*.

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 connaitre la taille de la chaine (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).

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 comme cela par exemple :

```
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. Sinon, on risque de voir le programme boguer :p . Par exemple, pour les chaînes de caractères nous devrions envoyer d'abord un entier qui spécifie la taille de la chaine puis envoyer la chaine elle-même (sans le caractère '\0'). Une autre méthode consiste à lire les octets de la chaine 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 ;) .

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);
```

Maintenant que nous avons tout, nous allons faire le point et améliorer nos deux applications.

Faisons le point

Nous allons améliorer nos deux applications pour qu'elles se transmettent des données : pour cela, nous allons nous servir des trois fonctions précédemment apprises...

Nous allons du côté serveur envoyer un "bonjour" quand un client se connecte, puis fermer l'application.

Du côté client, nous allons recevoir la chaîne de caractères, et l'afficher à l'écran.

Vous pouvez vous lancer maintenant, ce n'est pas difficile ;) . Je suis même persuadé que vous pouvez le faire :p .

Codons ... ...
... ...

C'est fini ! Voilà la correction ^^ :

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>
#define PORT 23



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

    SOCKET sock;
    SOCKADDR_IN sin;
    SOCKET csock;
    SOCKADDR_IN csin;
    char buffer[32] = "Bonjour !";
    socklen_t recsize = sizeof(csin);
    int sock_err;

    /* Si les sockets Windows fonctionnent */
    if(!erreur)
    {
        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 */
sin.sin_addr.s_addr    = htonl(INADDR_ANY);   /* Adresse IP automatique */
sin.sin_family         = AF_INET;             /* Protocole familial (IP) */
sin.sin_port           = htons(PORT);         /* Listage du port */
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("Listage 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);

        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 envoyée : %s\n", buffer);
```

```
                    else
                        printf("Erreur de transmission\n");

                    /* Il ne faut pas oublier de fermer la connexion (fermée 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
    }

    /* On attend que l'utilisateur tape sur une touche, puis on ferme */
    getchar();

    return EXIT_SUCCESS;
}
```

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>
#define PORT 23



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

SOCKET sock;
SOCKADDR_IN sin;
char buffer[32] = "";

/* Si les sockets Windows fonctionnent */
if(!erreur)
{
    /* Création de la socket */
    sock = socket(AF_INET, SOCK_STREAM, 0);

    /* Configuration de la connexion */
    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 à %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 */
        if(recv(sock, buffer, 32, 0) != SOCKET_ERROR)
            printf("Recu : %s\n", buffer);
    }
    /* sinon, on affiche "Impossible de se connecter" */
    else
    {
        printf("Impossible de se connecter\n");
    }

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

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

/* On attend que l'utilisateur tape sur une touche, puis on ferme */
getchar();

return EXIT_SUCCESS;
}
```

```
#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>
#define PORT 23



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

    SOCKET sock;
    SOCKADDR_IN sin;
    SOCKET csock;
    SOCKADDR_IN csin;
    char buffer[32] = "Bonjour !";
    socklen_t recsize = sizeof(csin);
    int sock_err;

    /* Si les sockets Windows fonctionnent */
    if(!erreur)
    {
        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 */
sin.sin_addr.s_addr    = htonl(INADDR_ANY);   /* Adresse IP automatique */
sin.sin_family         = AF_INET;             /* Protocole familial (IP) */
sin.sin_port           = htons(PORT);         /* Listage du port */
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("Listage 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);

        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 envoyée : %s\n", buffer);
```

```
                    else
                        printf("Erreur de transmission\n");

                    /* Il ne faut pas oublier de fermer la connexion (fermée 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
    }

    /* On attend que l'utilisateur tape sur une touche, puis on ferme */
    getchar();

    return EXIT_SUCCESS;
}
```

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>
#define PORT 23



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

SOCKET sock;
SOCKADDR_IN sin;
char buffer[32] = "";

/* Si les sockets Windows fonctionnent */
if(!erreur)
{
    /* Création de la socket */
    sock = socket(AF_INET, SOCK_STREAM, 0);

    /* Configuration de la connexion */
    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 à %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 */
        if(recv(sock, buffer, 32, 0) != SOCKET_ERROR)
            printf("Recu : %s\n", buffer);
    }
    /* sinon, on affiche "Impossible de se connecter" */
    else
    {
        printf("Impossible de se connecter\n");
    }

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

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

/* On attend que l'utilisateur tape sur une touche, puis on ferme */
getchar();

return EXIT_SUCCESS;
}
```

```
#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>
#define PORT 23



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

SOCKET sock;
SOCKADDR_IN sin;
char buffer[32] = "";

/* Si les sockets Windows fonctionnent */
if(!erreur)
{
    /* Création de la socket */
    sock = socket(AF_INET, SOCK_STREAM, 0);

    /* Configuration de la connexion */
    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 à %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 */
        if(recv(sock, buffer, 32, 0) != SOCKET_ERROR)
            printf("Recu : %s\n", buffer);
    }
    /* sinon, on affiche "Impossible de se connecter" */
    else
    {
        printf("Impossible de se connecter\n");
    }

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

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

/* On attend que l'utilisateur tape sur une touche, puis on ferme */
getchar();

return EXIT_SUCCESS;
}
```

> Au lieu d'envoyer 32 octets de chaine de caractères (dans le cas ou sizeof(char)=1) nous aurions pu n'envoyer que 10 octets (c'est la taille de la chaine "Bonjour !"). Mais le problème qui se serait posé est que l'application client ne connait pas la taille de la chaine envoyée par le serveur. Il y a plusieurs solution pour régler ce problème comme envoyer un nombre qui précède la chaine spécifiant sa taille ou de lire la chaine de caractère reçu bits à bits jusqu'à se que l'on trouve le caractère '\0'. Ici, pour faire simple, nous n'allons pas déployer ces principes.

Sachez que vous pouvez créer un fichier *header.h* qui contient les inclusions, *typedef* et déclarations. Je ne l'ai pas fait pour éviter de vous perdre, avec les fichiers (déjà pas facile, les sockets... avec deux applications :p). C'est en temps normal vivement recommandé ^^ .

Transmission d'une structure

> La transmission de structures, d'entiers (de type int, short, long...), de nombres décimaux (double, float...), etc. Bref, tout ce qui n'est pas une chaine de caractères ne fonctionne qu'en local. La transmission d'autres choses que des chaines de caractères est expliquée à la fin de ce chapitre.

Comme nous l'avons vu, les fonctions *send* et *recv* permettent la transmission de nombres et de tableaux (dont les chaînes de caractères). Mais cela ne s'arrête pas là. **Ces deux fonctions permettent aussi l'envoi de structures et même plus** :D ! Il suffira juste de donner un pointeur de la structure aux deux fonctions et la taille que prend cette structure en mémoire ^^ . Bien sûr, il faudra que les déclarations des structures soient présentes dans les deux projets ;) .

Prenons par exemple une structure Point contenant deux entiers correspondants aux coordonnées X et Y du point. Eh bien au lieu de passer les données par un tableau il vous suffira de passer l'adresse de cette structure (à l'aide du symbole '&'). Dans cet exemple, il n'est pas plus difficile de passer les variables X et Y dans un tableau de 2 cases, mais dans le cas d'une grosse structure cela peux vraiment vous aider. Prenons maintenant une structure déclarée comme ci-dessous :

```
struct Joueur
{
    int ID;
    char nom[256];
    char prenom[256];
};
```

Eh bien dans ce cas, l'envoi d'un tableau peut paraître délicat alors qu'en envoyant une structure vous pourrez accéder facilement aux données qu'elle comporte.

Passons donc maintenant à la pratique :) .
Nous allons créer une structure Personnage qui contiendra le nom, le prénom et l'âge de celui-ci :

```
struct Personnage
{
    char nom[256];
    char prenom[256];
    int age;
};
typedef struct Personnage Personnage;
```

Rien de bien compliqué ^^ .
Comme dit plus haut, ce code devra être écrit dans les deux projets.
Ensuite nous allons déclarer une variable de type Personnage et remplir ses composantes :

```
Personnage monPersonnage;
[...]
sprintf(monPersonnage.nom, "Matin");
sprintf(monPersonnage.prenom, "Dupont");
monPersonnage.age = 29;
```

Mon personnage se nommera donc Matin Dupont et aura 29 ans :p .

Maintenant, nous allons envoyer cette structure à un client en tant que serveur:

```
send(sock, &monPersonnage, sizeof(monPersonnage), 0);
```

Et le client va recevoir cette structure avec la fonction *recv* :

```
recv(sock, &monPersonnage, sizeof(monPersonnage), 0)
```

> Du côté du client, la structure devra être déclarée. Elle sera automatiquement remplacée par la structure reçue.

> *Et maintenant ... Comment peut-on accéder à la structure reçue du côté du client ?*

Eh bien tous simplement comme n'importe quelle structure ^^ . Faîtes un petit *printf* avec comme paramètre **monPersonnage.age**, vous verrez que la console vous affichera 29 (à part si vous avez mis autre chose :p). Il en est de même pour les deux autres paramètres.

Vous l'avez peu être compris, vous pouvez transmettre tous les types de données que vous voulez (int, char, long, structures, etc.) avec les fonctions *send* et *recv* à condition que ce type soit déclaré dans le code serveur et aussi dans le code client.

> Vous remarquerez que les deux chaînes de caractères qui étaient contenues dans nos structures disposaient d'une taille fixe (256 caractères avec le caractère '\0' inclus). Si nous voulons ne pas avoir cette limitation, il faut utiliser des pointeurs à la place et les allouer dynamiquement. Mais cela pose un problème car en envoyant la structure contenant les pointeurs on ne fait qu'envoyer la valeur des pointeurs (là ou il pointe) mais pas les variables sur lesquels ils pointent. Il faut donc envoyer aussi les données pointées par ces pointeurs ;) .

Un problème de portabilité

Jusque là, nous avons transmis des données d'un ordinateur à un autre sans faire attention aux types de données que nous transmettions. Mais cela pose de sérieux problèmes de portabilité.
En effet, après avoir sûrement lu le cours de M@téo21 sur l'allocation dynamique (http://www.siteduzero.com/tutoriel-3-14061-l-allocation-dynamique.html#ss_part_1) vous avez appris que, par exemple, la taille d'une variable de type int peut varier d'un ordinateur à un autre.
Donc, si nous envoyons une variable de type int à partir d'un ordinateur où le type int fait 8 octets sur un autre ordinateur où le type int fait 2 octets cela posera irrémédiablement problème : Pendant que l'ordinateur source envoi une seule variable de type int codée sur 8 octets, l'ordinateur de destination se verra recevoir 4 variables de type int :-° !
Ce problème pourrait être réglé en utilisant un type de données qui garde le même nombre d'octets d'un ordinateur à autre...
Mais voilà qu'il y à un second problème : 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 (http://fr.wikipedia.org/wiki/Petit-boutiste) (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 résumer...

Si votre application communique avec une autre application qui se trouve sur le même ordinateur, il n'y aura donc pas de problème car la taille des types de données ne change pas d'une application à une autre, de même pour l'ordre des octets d'une variable en mémoire.
A contrario, pour réaliser une communication portable entre deux ordinateurs, il faut absolument transmettre les octets de nos variables un à un selon un ordre donné qui est le même pour les deux applications distantes. Des fonctions existent pour réaliser ces manipulations, nous allons donc commencer par les étudier puis nous allons les utiliser dans le cadre d'un exemple pour bien comprendre leur fonctionnement.

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.

Un exemple de fonctionnement des fonctions htonl et ntohl

Supposons que vous vouliez 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)
{
    // Tableau d'octet qui sera ensuite envoyé
    char dataSend[4];

    // On décompose l'entier 'data' de 4 octets en 4 parties de 1 octet
    dataSend[0] = (data >> 24) & 0xFF;  // On sélectionne l'octet de poids fort de 'data' que l'on met dans la première case du tableau d'octet
 'dataSend'
    dataSend[1] = (data >> 16) & 0xFF;  // De même avec l'octet qui suit
    dataSend[2] = (data >> 8) & 0xFF;   // De même avec l'octet qui suit
    dataSend[3] = (data >> 0) & 0xFF;   // On sélectionne l'octet de poids faible de 'data' que l'on met dans la dernière case du tableau d'octe
t 'dataSend'

    // On envoi les 4 octets dans un ordre qui ne change jamais quelque soit la machine
    send(sock, dataSend, 4, 0);
}

void read4(int sock, unsigned long* data)
{
    char dataRecv[4];

    // On reçoit une suite de 4 octets, le premier octet reçu est toujours l'octet de poids fort
    recv(sock, dataRecv, 4, 0);

    // On rassemble les 4 octets séparé en une seul variable de 4 octets
    unsigned long temp = 0;
    temp |= dataRecv[0] << 24;
    temp |= dataRecv[1] << 16;
    temp |= dataRecv[2] << 8;
    temp |= dataRecv[3] << 0;

    // On fini par copier le résultat dans 'data'
    *data = temp;
}
```

Ces deux fonctions sont un peu lourdes donc je vous conseil d'utiliser plutôt les fonctions de conversion spécifiées plus haut. Voici les même fonctions send4 et read4 implémentées en utilisant les fonctions htonl et ntohl :

```
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);
}
```

Notez que si vous voulez envoyer un entier de 2 octets le fonctionnement est exactement le même :) .
Pour ce qui est de la transmission de structures, il faudra envoyer chaque éléments qui la compose un à un pour que le code reste portable.

Ce chapitre est la base de la transmission de flux entre une application serveur et application cliente. Il vous est maintenant possible de transmettre tout ce que vous voulez entre les deux. Il est donc essentiel.
Notez que nous n'avons pas parler du cas des nombres réels (float, double, etc.) car c'est assez difficile de transmettre ces nombres entre deux applications tout en restant portable. Mais rien ne peut vous empêcher de convertir ces nombres en chaines de caractères ou encore en nombres entiers après les avoir multipliés par 1000 par exemple ;) .

Les threads et les mutex

Dans ce chapitre, nous allons parler des threads, des processus et des mutex. Il est relativement important car il constitue la partie théorique du cours sur les threads et permet d'installer une bibliothèque nommée Pthread ;) .

Quelques définitions

Les processus

Un processus est une tâche qui est en train de s'exécuter.
Par exemple, quand vous lancez un de vos programmes que vous avez développé, votre OS (http://fr.wikipedia.org/wiki/OS) crée un nouveau processus et celui-ci exécutera une suite d'instructions sur votre ordinateur (le code de votre programme compilé).

Citation : Wikipédia

> Un processus est défini par :
>
> - Un ensemble d'instructions à exécuter
> - Un espace mémoire pour les données de travail
> - Éventuellement, d'autres ressources, comme des descripteurs de fichiers, des ports réseaux, etc...

Si vous être sous Windows, vous pouvez accéder à la liste des processus via un gestionnaire des tâches (en appuyant simultanément sur les touches CTRL+ALT+SUPPR).
Sous linux, vous pouvez accéder à la liste des processus via un indicateur de performance.
Plus d'informations : ici (http://fr.wikipedia.org/wiki/Processus_(informatique)).

Les 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. Les threads nous permettront de régler ce problème.
Plus d'informations : ici (http://fr.wikipedia.org/wiki/Processus_léger).

Les mutex

Il est parfois nécessaire d'interdire momentanément certaines opérations d'un ou plusieurs threads : par exemple, si plusieurs threads sont amenés à lire une variable, faire des calculs avec puis la modifier en fonction du résultat de ces calculs, il ne faut pas qu'ils le fassent en même temps, sinon cela risque tous simplement de boguer.
Les mutex permettent donc d'éviter ces problèmes de synchronisation :) .
Plus d'informations : ici (http://fr.wikipedia.org/wiki/Exclusion_mutuelle).

Pourquoi choisir la bibliothèque pthread ?

Le terme Pthread est une abréviation de "POSIX Threads".
POSIX est lui un acronyme de "Portable Operating System Interface for UniX".

J'ai donc choisi pthread car c'est une très bonne bibliothèque **portable** permettant de manipuler les threads, les processus et les mutex assez facilement ;) .

Installation de pthread

Sous Windows

Sous Windows, cette bibliothèque n'est pas installée par défaut, il va donc falloir le faire.
Nous allons dans un premier temps télécharger la bibliothèque puis l'installer et ensuite nous pourrons l'utiliser dans nos programmes.

Vous pouvez obtenir la bibliothèque pthread en cliquant sur le lien qui suit :
Pthread - Win32 - Version 2.8.0 (ftp://sourceware.org/pub/pthreads-win32/pthreads-w32-2-8-0-release.exe)

Commencez par ouvrir l'exécutable et cliquez sur le bouton "Extract".
Ainsi, trois dossiers sont créés dans le répertoire de l'exécutable.
Le dossier "pthreads.2" contient les sources de la bibliothèque. Elles vous seront utiles dans le cas où vous devriez compiler vous même la bibliothèque.
Le dossier "Pre-built.2" contient les fichiers .h à inclure et les fichiers .lib à linker.
C'est donc ce dossier qui va nous intéresser, ouvrez le ;) .

Mettez les fichiers .lib/.a dans le dossier "lib" de votre compilateur et les headers (les fichiers .h) dans le dossier "include".

Petit rappel :
Pour ceux qui utilisent VC, prenez les fichiers ayant les extensions .lib.
Pour ceux qui utilisent Code::Blocks ou Dev-C++, prenez les fichiers ayant l'extension .a.

Une fois la bibliothèque pthread installée, nous allons linker les fichiers .lib/.a à notre projet et inclure les headers dans notre code. En fonction de votre IDE et du langage que vous avez choisi d'utiliser, le fichier linké ne sera pas le même :
Si vous faîtes du C, linkez le fichier "pthreadVC2.lib" pour VC et le fichier "libpthreadGCc.a" pour Code::Blocks et Dev-C++.
Sinon si vous faîtes du C++, linkez le fichier "pthreadVCE2.lib" pour VC et le fichier "libpthreadGCEc.a" pour Code::Blocks et Dev-C++.

Il en est de même pour les DLL ;) :
Si vous faîtes du C, prenez la DLL "pthreadVC2.dll" pour VC et la DLL "pthreadGC2.dll" pour Code::Blocks et Dev-C++.
Sinon si vous faîtes du C++, prenez la DLL "pthreadVCE2.dll" pour VC et la DLL "pthreadGCE2.dll" pour Code::Blocks et Dev-C++.
La DLL doit être mise dans le répertoire de votre projet.

Note : *Vous pouvez mettre la DLL dans le répertoire "X:\WINDOWS\system32\" (ou X est le nom du disque dur contenant le dossier Windows). Ainsi, la DLL n'a plus besoin d'être dans le répertoire de vos projets :) sur votre ordinateur.*

Nous allons ensuite inclure pthread comme ceci :

```
#include <pthread.h>
```

Sous Linux

Sous linux la bibliothèque est déjà installée normalement, vous n'aurez donc pas besoin de le faire :) .
Toutefois sachez que les sources utilisant les threads nécessitent une édition de lien avec la librairie pthread :

```
gcc nomSource.c -lpthread -o nomExecutable
```

Les threads

Dans cette partie nous allons voir comment utiliser les threads avec la bibliothèque pthread.

Déclarer un thread

Pour pouvoir utiliser notre thread, nous allons tout d'abord déclarer une variable de type pthread_t comme il suit.

```
pthread_t thread;
```

Créer un thread

Une fois que notre thread est déclaré, il va falloir le lier à une fonction de notre choix, la fonction désignée se déroulera ensuite en parallèle avec le reste de l'application. Pour réaliser cela nous allons utiliser la fonction *pthread_create* dont le prototype est donné ci-dessous.

```
int pthread_create(pthread_t* thread, pthread_attr_t* attr, void*(*start_routine)(void*), void* arg);
```

- En cas de succès la fonction renvoie 0. En cas d'erreur, la fonction renvoie un code d'erreur non nul.
- L'argument **thread** correspond au thread qui va exécuter la fonction.
- L'argument **attr** indique les attributs du thread, ce paramètre ne nous intéresse pas, nous mettrons donc celui-ci à NULL pour que les attributs par défaut soient utilisés.
- L'argument **start_routine** correspond à la fonction à exécuter.
- L'argument **arg** est un pointeur sur void qui sera passé à la fonction à exécuter. Si vous n'avez aucun paramètre à passer, mettez ce paramètre à NULL.

Toutefois, pour éviter des erreurs de compilation, la fonction exécutée par le thread créé devra toujours avoir le prototype suivant :

```
void* ma_fonction(void* data);
```

Dans l'exemple qui suit, le thread **thread** va exécuter la fonction *ma_fonction* en parallèle avec comme paramètre l'entier **valeur**.

```
pthread_create(&thread, NULL, ma_fonction, (void*)&valeur);
```

Pour le dernier argument, **&valeur** sera un `int*`, mais pour le transformer en `void*` nous ferons un simple cast :) .

Attendre la fin de l'exécution d'un thread

Une fois que notre thread est exécuté, il se peut que nous ayons besoin de savoir quand il se termine.
La fonction *pthread_join* va permettre d'attendre la fin du thread c'est à dire la fin de l'exécution de la fonction exécutée par celui-ci. Voici le prototype:

```
int pthread_join(pthread_t thread, void **thread_return);
```

- En cas de succès, la fonction renvoie 0. En cas d'erreur, la fonction renvoie un code d'erreur non nul.
- L'argument **thread** correspond au thread à attendre.
- L'argument **thread_return** est un pointeur sur la valeur de retour du thread.

Terminer le thread courant

Dans certain cas, il est possible que la fonction principale d'un thread ne se termine jamais, notamment lorsque l'on utilise une boucle infinie. Dans ce cas, on doit forcer la fin de du thread avec la fonction *pthread_exit* :

```
void pthread_exit(void *retval);
```

- La fonction ne renvoie rien.
- L'argument **retval** est un pointeur sur void, il correspond à la valeur de retour du thread qui exécute la fonction (si cette fonction est utilisée dans la fonction principale du thread, cela équivaut à faire : *"return retval;"*).

Terminer un thread

Dans de nombreux cas, il est possible que vous ayez à terminer un thread depuis un autre. Par exemple, lorsque vous utiliserez les threads avec les sockets, la fonction ci-dessous pourrait être utilisée pour expulser des clients (associés à un thread) depuis le thread principal.

```
int pthread_cancel(pthread_t thread);
```

- En cas de succès, la fonction renvoie 0. En cas d'erreur, la fonction renvoie un code d'erreur non nul.
- L'argument **thread** correspond au thread à terminer.

> La mémoire de tous les threads d'un même processus peut être partagée ou non. Dans le cas de la mémoire partagée, cela signifie que si on crée dynamiquement une variable dans un thread, elle peut être lue dans un autre (c'est le cas des variables globales par exemple). Dans le cas de la mémoire privée, les variables crées dans un thread ne peuvent pas être lue dans un autre (c'est le cas des objets locaux par exemple). Il faut donc bien surveiller la mémoire qui est allouée dans un thread car elle n'est pas automatiquement libérée à la fin du thread. Pensez donc à libérer la mémoire qui a été allouée dynamiquement au cours du thread avant sa destruction, si besoin est ^^ . Il faut aussi veiller à ne pas désallouer la mémoire qui sera utilisée par un autre thread durant la suite de l'exécution programme.

Un petit exemple :

```c
// On inclue les fichiers standard et le header pthread.h
#include <stdlib.h>
#include <stdio.h>
#include <pthread.h>

void* maFonction(void* data);



int main()
{
    int i;

    // On crée un thread
    pthread_t thread;

    // Permet d'exécuter le fonction maFonction en parallèle
    pthread_create(&thread, NULL, maFonction, NULL);

    // Affiche 50 fois 1
    for(i=0 ; i<50 ; i++)
        printf("1");

    // Attend la fin du thread créé
    pthread_join(thread, NULL);

    return 0;
}


void* maFonction(void* data)
{
    int i;

    // Affiche 50 fois 2
    for(i=0 ; i<50 ; i++)
        printf("2");

    return NULL;
}
```

Le résultat donné est le suivant :

```
11111111112222222222222222222222222222222211111111111111111111111111111111112222222222222222211111
```

Comme vous pouvez le voir, l'utilisation des threads permet d'exécuter une fonction en parallèle.

> *Mais pourquoi les nombres s'affichent par paquets ? Si les fonctions se lançaient en parallèle, je devrais obtenir 12121212...*

Si les nombres s'affichent par paquet, c'est à cause de votre système d'exploitation et de votre processeur.
En effet, avec un processeur mono-coeur, la fonction principale (*main*) et la fonction exécuté par le thread créé (*maFonction*) ne se lance pas tout à fait en parallèle car votre OS va attribuer un temps d'exécution pour chaque thread.
Votre OS va, par exemple, exécuter le code du thread n°1 de votre processus pendant 8 ms, puis va exécuter le code du thread n°2 du même processus aussi pendant 8 ms, etc. Il en est de même pour le fonctionnement des processus ;) .
Ainsi, comme chaque thread est exécuté à tour de rôle pendant un temps très court, vous avez l'impression qu'ils se déroulent en parallèle ^^ . On appel ça l'ordonnancement des thread.

Si vous avez un processeur multi-coeurs cela n'est plus vraiment valable car les deux threads peuvent réellement se dérouler en parallèle, et c'est tout l'avantage des threads :) .
Par exemple, si vous disposez d'un processeur qui dispose de deux coeurs ou plus et que votre application lance deux threads simultanément, votre application sera théoriquement jusqu'à deux fois plus rapide. De même avec quatre coeurs ou plus et quatre threads votre application sera théoriquement jusqu'à quatre fois plus rapide, etc. Par contre, avec deux coeurs et quatre threads votre application sera jusqu'à deux fois plus rapide et non pas quatre en théorie :p .

Ces dernières années, le nombre de coeurs dans les processeurs ne cesse de croître et donc il est de plus en plus intéressant de mettre en place des solutions parallèles efficaces pour accélérer nos programmes.

Vous avez surement observé que les thread et les mutex ont un fonctionnement est assez simple à assimiler. Cependant, selon l'utilisation que l'on en fait, cela peut très vite devenir une horreur. Il faut donc être assez rigoureux quand on met en place des threads et surtout particulièrement lorsque l'on utilise des mutex pour éviter par exemple les problèmes inter-blocage ;) .

La selection de sockets

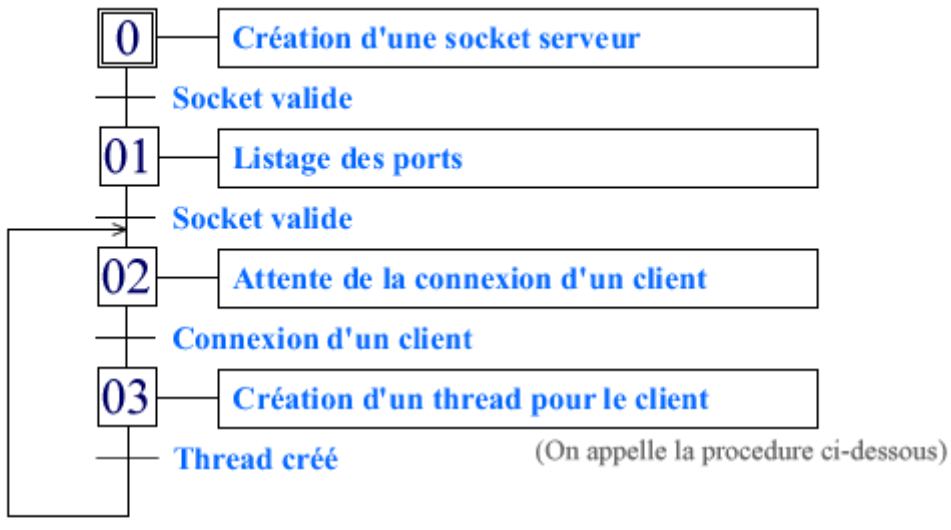
> Bien que les threads sont beaucoup utilisés dans le domaine du réseau, on utilise aussi un autre moyen pour manipuler plusieurs sockets : la sélection de sockets.
> La sélection de socket est un principe un peu plus simple à comprendre que l'utilisation de threads. Mais, ne vous faites pas d'illusions car dans de nombreux cas vous aurez à utiliser les threads en plus de la sélection de sockets. Dans cette partie, mon but sera de vous expliquer quels sont les avantages et inconvénients de ces deux méthodes ;) .

Le fonctionnement

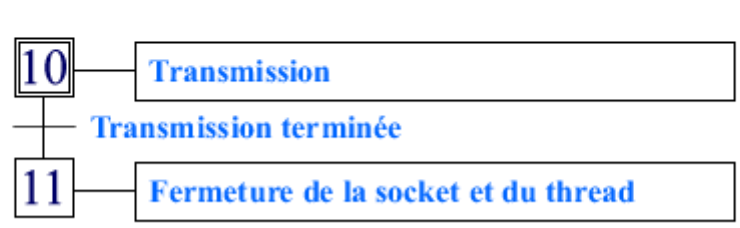
Le fonctionnement global

La sélection de sockets s'inscrit dans un fonctionnement évènementiel, c'est à dire que tout se fait dans un seul thread et dans un seul et même processus. Elle présente une alternative puissante à l'utilisation des threads. Si vous avez lu le tutoriel de m@teo21 sur la SDL ou si vous connaissez, par exemple, l'API Windows, ce principe vous est déjà un peu familier :) .

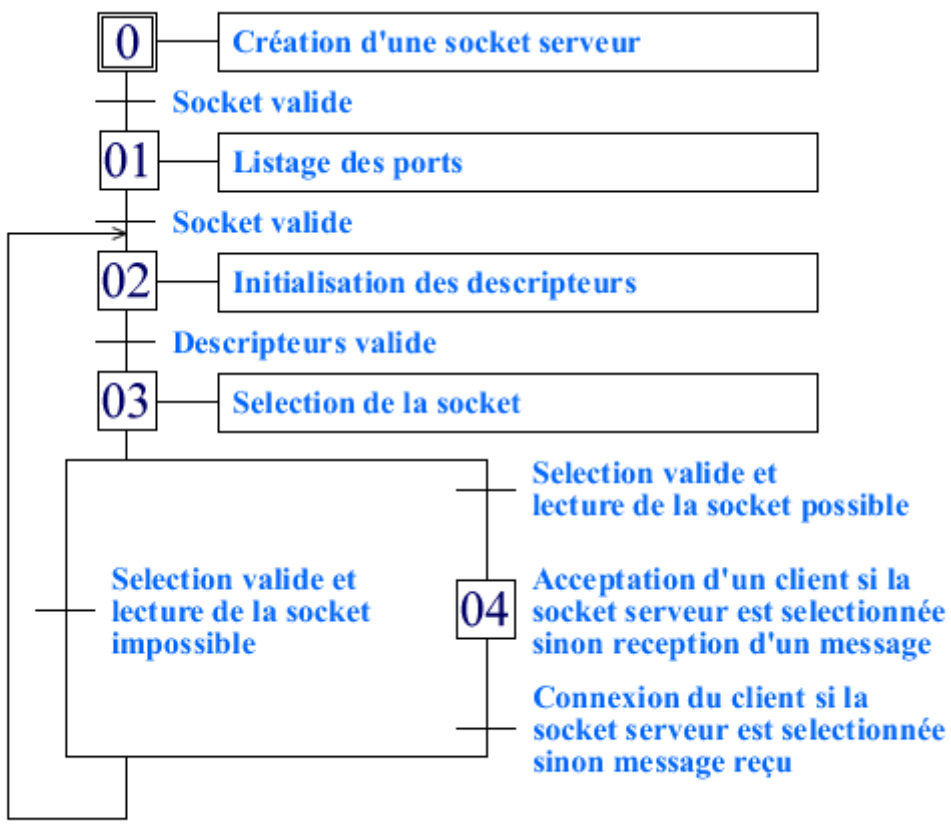
Avant, avec l'utilisation des threads sans la sélection de sockets, nous avions un schéma similaire à celui-ci :



Thread appelé :



Et maintenant en utilisant la sélection de socket, nous avons ce schéma :



Notez que les schémas ci-dessus sont des grafcets (http://fr.wikipedia.org/wiki/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).

Dans le cas des threads, on crée une socket serveur, on liste les ports, puis pour chaque clients qui se connecte on crée un thread qui lui est approprié dans lequel la transmission entre le client et le serveur se déroulera.

Dans le cas de la sélection de sockets, 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.

> *Qu'est ce qu'un 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. Vous ne le savez peut être pas jusque là mais 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.

> *Qu'est ce qu'un ensemble ?*

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 mais nous ne nous en servirons pas dans ce chapitre.

Ce qu'il faut donc retenir

Vous pouvez choisir si la sélection de sockets sera bloquante ou non quand tel ou tel événement se produit en fonction des descripteurs que vous lui transmettez. Prenons le cas ou vous spécifiez la sélection d'une socket client avec un descripteur en lecture seulement (on cherche à savoir si l'on peut lire des données sur la socket, si c'est le cas cela signifie que l'on a reçu des données sur cette socket ;)) et un temps limite de 50 ms : La sélection de la socket cliente est bloquante tant qu'elle ne reçoit pas de données jusqu'à ce que 50 ms se soit écoulé, après, la sélection rend la main (elle ne devient plus bloquante). La valeur qu'elle retourne spécifie l'évènement qui a mis fin au blocage (ici, le temps ou des données reçues peuvent mettre fin au blocage).

Un peu de pratique

Je m'en serais douté, vous avez surement eu du mal à comprendre ce qui a été spécifié ci-dessus et je vous comprends car ce n'est pas très simple :-° .
J'espère donc que la partie pratique vous sera plus parlante :) .

L'initialisation des descripteurs

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 ;) .
Pour faire cela nous somme face à quatre fonctions présenté ci-dessous.

FD_SET

```
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

```
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

```
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_ZERO(fd_set *set);
```

Cette fonction vide l'ensemble **set**. Cela revient à supprimer tout les descripteurs ajouté précédemment à l'ensemble.

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

Un 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>

#define PORT 23



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

    SOCKADDR_IN sin;
    SOCKET sock;
    int recsize = sizeof sin;

    int sock_err;

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

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

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

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

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

                    while(1)
                    {
                        /* 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");
                    }
                }
            }
        }
    }
}

#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.

La suite arrive bientôt :) ...

Merci à Trist@n21 pour avoir corrigé une partie de ce tutoriel et merci à -ed- pour avoir corrigé les points sombres du tutoriel :) .