Les sockets en C

par Benjamin Roux (Retour aux articles)

Date de publication : 27/01/2008

Dernière mise à jour : 06/03/2009

Cet article vous présentera les sockets en langage C, du TCP à l'UDP de Windows à Linux.

1 - Introduction

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".

Nous allons voir par la suite comment réaliser diverses applications telles qu'un client/serveur TCP ou même un client/serveur UDP.
Nous passerons aussi sur les sockets asynchrones, ainsi que les différentes bibliothèques disponibles pour nous faciliter la tâche.

2 - Les différentes implémentations

2-1 - Les sockets sous Linux

2-1-1 - Les structures

2-1-1-a - struct sockaddr_in

struct sockaddr_in
```
struct sockaddr_in {
   uint8_t          sin_len;       /* longueur totale      */
   sa_family_t      sin_family;    /* famille : AF_INET     */
   in_port_t        sin_port;      /* le numéro de port    */
   struct in_addr   sin_addr;      /* l'adresse internet   */
   unsigned char    sin_zero[8];   /* un champ de 8 zéros  */
};
```

Nous verrons l'utilisation du champs *sin_addr* et *sin_port* un peu plus tard.
Les champs *sin_len* et *sin_zero* ne sont pas utilisé par le développeur.
Vous vous dites, oui mais dans les fonctions on demande une structure de type **sockaddr**, et vous avez raison, seulement cette structure sert juste de référence générique pour les appels systèmes. Il vous faudra donc simplement caster votre **sockaddr_in** en **sockaddr**.

2-1-1-b - struct sockaddr et struct in_addr

struct sockaddr
```
struct sockaddr {
   unsigned char    sa_len;         /* longueur totale   */
   sa_family_t      sa_family;      /* famille d'adresse  */
   char             sa_data[14];    /* valeur de l'adresse */
};
```

Le champ *sin_zero* de la structure **sockaddr_in** existe uniquement dans le but que la taille de la structure **sockaddr_in** soit la même que celle de la structure **sockaddr**.
Voir plus **loin** pour l'utilisation de ces structures.

struct in_addr
```
struct in_addr {
   in_addr_t s_addr;
};
```

Les définitions de **sockaddr** et de **in_addr** ne sont données qu'à titre purement indicatif.

2-1-1-c - struct hostent

```
struct hostent {
   char    *h_name;       /* Nom officiel de l'hôte.    */
   char   **h_aliases;    /* Liste d'alias.             */
   int      h_addrtype;   /* Type d'adresse de l'hôte.  */
   int      h_length;     /* Longueur de l'adresse.     */
   char   **h_addr_list;  /* Liste d'adresses.          */
}
#define h_addr  h_addr_list[0] /* pour compatibilité.   */
};
```

2-1-2 - Les fonctions

Sous Linux, le Langage C nous fournit un certain nombre de fonctions et structures pour manipuler les sockets.
Voici une liste non exhaustive.

2-1-2-a - socket

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

Cette fonction crée un socket. On remarque qu'un socket est donc de type **int**.
Cette fonction renvoie le descripteur de votre socket nouvellement créé.

Pour de plus amples informations : **socket()**

2-1-2-b - close

close
```
int close(int fd);
```

Cette fonction ferme le descripteur *fd*, dans notre cas, elle fermera simplement notre socket.

Pour de plus amples informations : **close()**

2-1-2-c - send et sendto

send et sendto
```
int send(int s, const void *msg, size_t len, int flags);
int sendto(int s, const void *msg, size_t len, int flags, const struct sockaddr *to, socklen_t tolen);
```

send est à utiliser pour le mode connecté. Elle envoie sur le socket *s*, les données pointées par *msg*, pour une taille de *len* octets.
sendto est à utiliser pour le mode non connecté. Pareil que la précédente sauf que l'on spécifie notre structure **sockaddr** (voir plus bas).
Ces 2 fonctions renvoient le nombre d'octets envoyés.

Pour de plus amples informations : **send() et sendto()**

2-1-2-d - recv et recvfrom

recv et recvfrom
```
int recv(int s, void *buf, int len, unsigned int flags);
int recvfrom(int s, void *buf, int len, unsigned int flags struct sockaddr *from, socklen_t *fromlen);
```

recv est à utiliser pour le mode connecté. Elle reçoit sur le socket *s*, des données qu'elle stockera à l'endroit pointé par *buf*, pour une taille maximale de *len* octets.
recvfrom est à utiliser pour le mode non connecté. Pareil que la précédente sauf que l'on spécifie notre structure **sockaddr** (voir plus bas).
Ces 2 fonctions renvoient le nombre d'octets reçus.

Pour de plus amples informations : **recv() et recvfrom()**

2-1-2-e - bind

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

bind lie un socket avec une structure sockaddr.

Pour de plus amples informations : **bind()**

2-1-2-f - connect

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

Cette fonction connecte votre socket à l'adresse spécifiée dans la structure **sockaddr**.
Il s'agit donc d'une fonction à utiliser dans un client.

Pour de plus amples informations : **connect()**

2-1-2-g - listen

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

Cette fonction définie la taille de la file de connexions en attente pour votre socket *s*.

Pour de plus amples informations : **listen()**

2-1-2-h - accept

accept
```
int accept(int sock, struct sockaddr *adresse, socklen_t *longueur);
```

Cette fonction accepte la connexion d'un socket sur le socket *sock*. Le socket aura été préalablement lié avec un port avec la fonction **bind**
L'argument *adresse* sera remplie avec les informations du client qui s'est connecté.
Cette fonction retourne un nouveau socket, qui devra être utilisé pour communiquer avec le client.

Pour de plus amples informations : **accept()**

Voilà c'est tout pour les fonctions basiques et les structures utilisées. Nous verrons leur utilisation plus tard dans cet article.
Passons maintenant aux sockets sous Windows.

2-2 - Les sockets sous Windows

Sous Windows l'implémentation de sockets n'est pas très différente de celle des systèmes UNIX.
Il faudra aussi rajouter cette option à votre éditeur de lien (une option dans votre éditeur) : -lws2_32 pour utiliser la bibliothèque des sockets.

2-2-1 - Les fonctions

Sous Windows, les fonctions sont les mêmes que sous les systèmes Unix. En revanche il faut appeler 2 fonctions supplémentaires en début de programme et en fin de programme pour respectivement initialiser une DLL permettant d'utiliser les sockets et pour libérer cette même DLL.

WSAStartup
```
int WSAStartup(
  __in          WORD wVersionRequested,
  __out         LPWSADATA lpWSAData
);
```

WSAStartup sur la MSDN
Vous vous dites, ohla ça a l'air compliqué. Et bien pas du tout, voilà comment on utilise cette fonction.

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

Tout simplement.

WSACleanup
```
int WSACleanup(void);
```

WSACleanup sur la MSDN
Cette fonction libère la DLL qui permet d'utiliser les sockets sous Windows.

2-2-2 - Les types

Microsoft a aussi redéfini certains types via des typages personnalisés (typedef), voici quelques uns des nouveaux types. Ces types sont définis dans <winsock2.h>

typedef Microsoft
- SOCKET pour l'utilisation de socket (à la place de int)
- SOCKADDR_IN pour struct sockaddr_in
- SOCKADDR pour struct sockaddr
- IN_ADDR pour struct in_addr

winsock2.h définit aussi quelques constantes non définies sous UNIX

Constantes Microsoft
- INVALID_SOCKET -1
- SOCKET_ERROR -1

Les autres fonctions s'utilisent de la même manière.
Bon vous voyez qu'il n'y a pas trop de différences, donc nous allons voir comment créer nos propres sockets portables dans la prochaine partie.

2-3 - Les sockets portables Windows/Linux

Pour utiliser des sockets portables, nous allons donc user de directives de compilation conditionnelle.

Voici un fichier d'en-tête standard pour une application voulant utiliser les sockets de manière portable.

```
#ifdef WIN32 /* si vous êtes sous Windows */

#include <winsock2.h>

#elif defined (linux) /* si vous êtes sous Linux */

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

```
#include <netinet/in.h>
#include <arpa/inet.h>
#include <unistd.h> /* close */
#include <netdb.h> /* gethostbyname */
#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;
typedef struct in_addr IN_ADDR;

#else /* sinon vous êtes sur une plateforme non supportée */

#error not defined for this platform

#endif
```

Comme vous pouvez le voir, selon la plateforme on inclut les fichiers d'en-têtes de la-dite plateforme, de plus si on est sous un système Linux on définit les types et constantes utilisés sous Windows.

N'oublions pas les 2 fonctions à appeler sous Windows. Pour cela je préconise d'utiliser 2 fonctions : ***init*** et ***end***.

```
static void init(void)
{
#ifdef WIN32
    WSADATA wsa;
    int err = WSAStartup(MAKEWORD(2, 2), &wsa);
    if(err < 0)
    {
        puts("WSAStartup failed !");
        exit(EXIT_FAILURE);
    }
#endif
}

static void end(void)
{
#ifdef WIN32
    WSACleanup();
#endif
}
```

La fonction ***init*** est a appeler en tout premier dans le **main**, la fonction ***end*** juste avant le **return** du **main**.

Voilà nous avons désormais des sockets portables Windows/Linux. Voyons comment utiliser les sockets ainsi que les fonctions à présent.

Dans la suite de l'article, je me sers de errno, en cas d'erreur sur une fonction. Or utiliser errno sous Windows ne fonctionne pas et est donc inutile.
*Une solution portable serait donc de définir une fonction **sock_error** qui retourne errno sous Linux et WSAGetLastError sous Windows, et une fonction **sock_err_message** qui appelle strerror sous Linux et FormatMessage sous Windows (merci à Melem).*

3 - Les protocoles TCP et UDP

3-1 - Présentation du protocole TCP

Le protocole TCP est un protocole dit connecté. Il contrôle si le paquet est arrivé à destination si ce n'est pas le cas il le renvoie.

Pour plus d'infos sur ce protocole je vous renvoie **ici** ou **là** ou même encore **ici**, ou bien encore et toujours au **man**.

3-2 - Utilisation du protocole TCP

L'utilisation du protocole TCP en C et avec les sockets est assez aisée. Il suffit simplement d'utiliser les fonctions détaillées plus haut.

3-2-1 - Côté client

Dans cette partie nous allons nous placer côté client, c'est-à-dire que nous allons nous connecter à un serveur. Nous allons voir comment faire.

3-2-1-a - Création du socket

Avant toute chose, il nous faut créer notre socket.

```
SOCKET sock = socket(AF_INET, SOCK_STREAM, 0);
if(sock == INVALID_SOCKET)
{
    perror("socket()");
    exit(errno);
}
```

C'est tout pour la création du socket. Pour les paramètres de la fonction **socket**, le **man** nous donnera plus de détails.
Je voudrais cependant apporter une petite précision, d'après POSIX.1 PF_INET et AF_INET sont confondus, c'est pourquoi j'utilise AF_INET alors que le man indique PF_INET (qui est aussi correct).

3-2-1-b - Connexion au serveur

Maintenant nous allons nous connecter à notre serveur, en remplissant au préalable la structure contenant les informations du serveur (port et adresse IP ou nom d'hôte).

```
struct hostent *hostinfo = NULL;
SOCKADDR_IN sin = { 0 }; /* initialise la structure avec des 0 */
const char *hostname = "www.developpez.com";

hostinfo = gethostbyname(hostname); /
* on récupère les informations de l'hôte auquel on veut se connecter */
if (hostinfo == NULL) /* l'hôte n'existe pas */
{
    fprintf (stderr, "Unknown host %s.\n", hostname);
    exit(EXIT_FAILURE);
}

sin.sin_addr = *(IN_ADDR *) hostinfo->h_addr; /
* l'adresse se trouve dans le champ h_addr de la structure hostinfo */
sin.sin_port = htons(PORT); /* on utilise htons pour le port */
```

```
sin.sin_family = AF_INET;

if(connect(sock,(SOCKADDR *) &sin, sizeof(SOCKADDR)) == SOCKET_ERROR)
{
    perror("connect()");
    exit(errno);
}
```

Alors ici, tout d'abord nous récupérons l'adresse IP de l'hôte qui a pour nom *hostname* (de type char *), *hostname* peut être directement une adresse IP (IPv4 ou IPv6 si le standard d'écriture est conservé).

Pour la structure *hostent*, je vous renvoie (encore) au man de la fonction **gethostbyname**. Le champ *h_addr* de la structure **hostinfo** contient l'adresse IP de l'hôte. Il faut donc le caster pour le mettre dans le champ *sin.sin_addr* qui n'est pas du même type. Ensuite nous remplissons notre structure **SOCKADDR_IN**, avec tout d'abord l'adresse IP, puis le port (*PORT* peut être défini par un #define par exemple), et enfin la famille. AF_INET spécifie que l'on utilise le protocole IP.
Une fois la structure remplie, on utilise la fonction **connect** pour se connecter.

3-2-1-c - Envoi et réception de données

Maintenant pour communiquer avec notre serveur (envoyer et recevoir des données), nous allons utiliser les fonctions **send** et **recv**.

Envoi de données

```
SOCKET sock;
char buffer[1024];
[...]
if(send(sock, buffer, strlen(buffer), 0) < 0)
{
    perror("send()");
    exit(errno);
}
```

Dans cet exemple, nous envoyons une chaîne de caractères, mais nous pourrions très bien envoyer autre chose comme une structure ou un int par exemple.

Réception de données

```
char buffer[1024]
int n = 0;

if((n = recv(sock, buffer, sizeof buffer - 1, 0)) < 0)
{
    perror("recv()");
    exit(errno);
}

buffer[n] = '\0';
```

Pour la réception de données, il faut bien s'assurer de placer le \0 final à notre chaîne de caractères (d'où le -1 dans le recv).
Dans le cas de la réception d'une structure il ne faudrait bien évidemment pas mettre ce -1.

Pour recevoir un flux de données (plusieurs données, un flux de grosse taille...), on doit bien entendu entourer notre **recv** avec un *while*. La fonction **recv** est une fonction bloquante, votre application sera donc bloquée tant qu'il n'y a rien à lire sur le socket. Dans le cas du *while* vous serez donc bloqué en attendant quelque chose à lire. C'est la raison pour laquelle la plupart des protocoles définissent un caractère ou une marque de fin d'envoi. Le serveur envoie son flux de données, puis pour finir sa marque de fin. Lorsque le client reçoit cette marque de fin il sort du *while*.
Pour la réception d'un fichier par exemple, le client et le serveur définissent une taille de bloc à envoyer (4096 octets par exemple), le serveur envoie donc le fichier bloc par bloc de 4096 octets, et le dernier bloc, lui, de la taille restante. Par exemple si votre fichier fait 5000 octets, vous aurez 2 blocs à envoyer un de 4096 octets et un de 904 octets. Le client, lui, lit tant que la taille reçue est égale à 4096, lorsqu'elle est différente c'est qu'il s'agit normalement du

dernier bloc, on peut donc sortir du *while*. Sinon (s'il n'y a pas de marqueur de fin ou de taille de bloc définis) il faut utiliser (par exemple) **select**.

3-2-1-d - Fermeture du socket

Et bien entendu on n'oublie pas de fermer notre socket

```
SOCKET sock;
[...]
closesocket(sock);
```

Voilà, vous savez comment vous connecter, envoyer et recevoir des données en C avec le protocole TCP. Nous verrons plus tard un exemple d'un client TCP.

Maintenant voyons comment cela se passe du côté serveur.

3-2-2 - Côté serveur

Côté serveur, c'est légèrement différent du côté client. Voyons cela de plus près.

3-2-2-a - Création du socket

La création du socket reste identique.

```
SOCKET sock = socket(AF_INET, SOCK_STREAM, 0);
if(sock == INVALID_SOCKET)
{
    perror("socket()");
    exit(errno);
}
```

3-2-2-b - Création de l'interface

Maintenant nous allons créer l'interface sur laquelle notre socket va écouter pour lier cette interface à notre socket. Pour cela nous allons encore utiliser la structure **SOCKADDR_IN**, puis la fonction **bind**.

```
SOCKADDR_IN sin = { 0 };

sin.sin_addr.s_addr = htonl(INADDR_ANY); /
* nous sommes un serveur, nous acceptons n'importe quelle adresse */

sin.sin_family = AF_INET;

sin.sin_port = htons(PORT);

if(bind (sock, (SOCKADDR *) &sin, sizeof sin) == SOCKET_ERROR)
{
    perror("bind()");
    exit(errno);
}
```

Tout d'abord on remplit la structure **SOCKADDR_IN**. Pour l'adresse nous utilisons **htonl** ici notre serveur écoutera sur le port *PORT* (#define).
Puis on lie notre socket à notre interface de connexion.

3-2-2-c - Ecoute et connexion des clients

Mais ce n'est pas encore fini, il reste encore à écouter et à accepter les connexions entrantes.

```
if(listen(sock, 5) == SOCKET_ERROR)
{
    perror("listen()");
    exit(errno);
}
```

Nous définissons une limite pour notre file d'entrée à 5 connexions simultanées.

```
SOCKET sock;
[...]
SOCKADDR_IN csin = { 0 };
SOCKET csock;

int sinsize = sizeof csin;

csock = accept(sock, (SOCKADDR *)&csin, &sinsize);

if(csock == INVALID_SOCKET)
{
    perror("accept()");
    exit(errno);
}
```

Je pense qu'ici quelques explications s'imposent.
La fonction **accept** renvoie un **nouveau** socket, il faut donc le stocker dans une **nouvelle** variable. Désormais, c'est ce socket que nous utiliserons pour communiquer avec notre client. Nous l'utiliserons pour faire des **read** ou des **send**.
La fonction **accept** remplit aussi une structure **SOCKADDR_IN** (2ème argument), avec les informations du client (adresse IP et port côté client).

3-2-2-d - Fermeture du socket

Et bien entendu on n'oublie pas de fermer notre socket mais aussi ceux de nos clients.

```
SOCKET sock;
[...]
closesocket(sock);
closesocket(csock);
```

Voilà pour le côté serveur. Bien entendu ici nous ne gérons pas de système multi-client, nous verrons cela plus tard dans un exemple de serveur utilisant le protocole TCP.

Voilà c'est tout pour le protocole TCP, je pense que certaines choses sont encore un peu floues, mais tout sera éclairci dans la partie 5, avec la réalisation d'une application client/serveur en TCP. Vous verrez donc comment utiliser tous les bouts de code que nous venons de voir.

Intéressons-nous maintenant au protocole UDP.

3-3 - Présentation du protocole UDP

A la différence de TCP, UDP est un protocole en mode non-connecté, il ne vérifie pas si le paquet est arrivé à destination.
Pour plus d'informations, vous pouvez aller voir **ici** ou **là**, ou encore sur le  **man**.

3-4 - Utilisation du protocole UDP

3-4-1 - Côté client

Dans cette partie nous allons nous placer du côté client, c'est à dire que nous allons envoyer des données à un serveur.

3-4-1-a - Création du socket

La création du socket est identique par rapport au protocole TCP, il y a seulement un paramètre qui change.

```
SOCKET sock = socket(AF_INET, SOCK_DGRAM, 0);
if(sock == INVALID_SOCKET)
{
    perror("socket()");
    exit(errno);
}
```

SOCK_STREAM est simplement devenu *SOCK_DGRAM*.

3-4-1-b - Envoi et réception de données

Et oui on passe directement à l'envoi et la réception de données, parce que comme dit plus haut UDP est un protocole non-connecté, donc pas de connexion au serveur.

Envoi et réception de données

```
SOCKET sock;
char buffer[1024];
[...]
struct hostent *hostinfo = NULL;
SOCKADDR_IN to = { 0 };
const char *hostname = "www.developpez.com";
int tosize = sizeof to;

hostinfo = gethostbyname(hostname);
if (hostinfo == NULL)
{
    fprintf (stderr, "Unknown host %s.\n", hostname);
    exit(EXIT_FAILURE);
}

to.sin_addr = *(IN_ADDR *) hostinfo->h_addr;
to.sin_port = htons(PORT);
to.sin_family = AF_INET;

if(sendto(sock, buffer, strlen(buffer), 0, (SOCKADDR *)&to, tosize) < 0)
{
    perror("sendto()");
    exit(errno);
}

if((n = recvfrom(sock, buffer, sizeof buffer - 1, 0, (SOCKADDR *)&to, &tosize)) < 0)
{
    perror("recvfrom()");
    exit(errno);
}

buffer[n] = '\0';
```

C'est donc ici que l'on renseigne la structure avec les informations du serveur.

Même remarque que pour le TCP. Ici nous envoyons une chaîne de caractères, mais nous pourrions très bien envoyer autre chose comme une structure ou un int par exemple.

Pour que le serveur puisse connaître vos informations (IP et port) pour savoir où envoyer des données, c'est à vous de commencer à parler au serveur.
Un **recvfrom** doit donc être précédé d'au moins un **sendto**.
Vous envoyez via le **sendto**, le serveur connaît désormais vos informations, il peut donc vous envoyer des données que vous recevrez dans un **recvfrom**.

Même remarque que pour le TCP, pour la réception de données, il faut bien s'assurer de placer le \0 final à notre chaîne de caractères (d'où le -1 dans le recv).
Dans le cas de la réception d'une structure ou d'un entier par exemple il ne faudrait bien évidemment pas mettre ce -1.

Toujours pareil qu'en TCP, pour lire un flux de données, il faut utiliser un *while*.

3-4-1-c - Fermeture du socket

Et bien entendu on n'oublie pas de fermer notre socket

```
SOCKET sock;
[...]
closesocket(sock);
```

Voyons maintenant du côté du serveur.

3-4-2 - Côté serveur

Côté serveur, ce n'est pas non plus très différent.

3-4-2-a - Création du socket

Là c'est pareil, ça ne change pas.

```
SOCKET sock = socket(AF_INET, SOCK_DGRAM, 0);
if(sock == INVALID_SOCKET)
{
    perror("socket()");
    exit(errno);
}
```

Maintenant nous allons créer notre interface.

3-4-2-b - Création de l'interface

A l'instar du protocole TCP il faut ici aussi créer son interface, c'est-à-dire configurer l'adresse IP et le PORT, sur lequel notre serveur va recevoir des données.

```
SOCKADDR_IN sin = { 0 };

sin.sin_addr.s_addr = htonl(INADDR_ANY);

sin.sin_family = AF_INET;

sin.sin_port = htons(PORT);

if(bind (sock, (SOCKADDR *) &sin, sizeof sin) == SOCKET_ERROR)
```

```
{
    perror("bind()");
    exit(errno);
}
```

Comme vous le voyez, aucun changement.

Et voilà c'est tout, rien d'autre. Pas de **accept** ni de **listen**. Votre client est prêt à recevoir des données.
Maintenant nous n'allons utiliser que des **sendto** et des **recvfrom**.
Leur utilisation est légèrement différente par rapport au côté client.

3-4-2-c - Envoi et réception de données

Votre serveur doit tout d'abord commencer par un **recvfrom** pour pouvoir obtenir les informations du client, pour savoir où répondre. Après seulement il pourra lui envoyer des données.

Envoi et réception de données

```
SOCKET sock;
char buffer[1024];
[...]
struct hostent *hostinfo = NULL;
SOCKADDR_IN from = { 0 };
int fromsize = sizeof from;

if((n = recvfrom(sock, buffer, sizeof buffer - 1, 0, (SOCKADDR *)&from, &fromsize)) < 0)
{
    perror("recvfrom()");
    exit(errno);
}

buffer[n] = '\0';

/* traitement */
[...]

if(sendto(sock, buffer, strlen(buffer), 0, (SOCKADDR *)&from, fromsize) < 0)
{
    perror("sendto()");
    exit(errno);
}
```

Une fois que le client a envoyé des données, la structure *from* est remplie avec ses informations (adresse IP et port), pour qu'on puisse lui répondre. L'entier *fromsize* est lui aussi rempli avec la taille effective de la structure.

Toujours pareil qu'en TCP, pour lire un flux de données, il faut utiliser un *while*.

3-4-2-d - Fermeture du socket

Et bien entendu on n'oublie pas de fermer notre socket

```
SOCKET sock;
[...]
closesocket(sock);
```

Voilà c'est tout pour le protocole UDP, je pense que certaines choses sont encore un peu floues, mais tout sera éclairci dans la partie 5, avec la réalisation d'une application client/serveur en UDP. Vous verrez donc comment utiliser tous les bouts de code que nous venons de voir.

*Je voudrais également rajouter que tout plein d'options sont utilisables avec les fonctions **recvfrom** et **sendto** et les sockets en UDP en général. Je ne les ai pas détaillées mais*

certaines sont d'une grande utilité. Il est possible de faire des sockets non bloquants, du multicast, de l'unicast... bref tout plein de choses intéressantes.

3-5 - Récapitulatif

Alors quoi utiliser quand et dans quel ordre ?

TCP côté client

- socket
- connect
- send
- recv
- close

TCP côté serveur

- socket
- bind
- listen
- accept
- send
- recv
- close

Pour les **send** et **recv** l'ordre n'a pas d'importance, on peut appeler l'un avant l'autre ou vice-versa. Ils sont par ailleurs facultatifs.

UDP côté client

- socket
- sendto
- recvfrom
- close

UDP côté serveur

- socket
- bind
- recvfrom
- sendto
- close

L'ordre des **sendto** et **recvfrom** a ici une importance, en revanche ils sont tout de même facultatifs (bien que l'intérêt d'une application réseau qui ne communique pas soit assez limitée).

4 - Les sockets avancés

4-1 - select

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

La fonction select attend un changement d'état des descripteurs contenus dans différents ensembles. Le paramètre *timeout* permet de placer une limite de temps à l'attente de **select**.
Le premier paramètre *n* correspond à la valeur du plus grand descripteur de fichiers de vos ensembles + 1.

Les ensembles

- readfds : les descripteurs seront surveillés pour voir si des données en lecture sont disponibles, un appel à recv par exemple, ne sera donc pas bloquant
- writefds : les descripteurs seront surveillés en écriture pour voir s'il y a de l'espace afin d'écrire les données, un appel à write ne sera donc pas bloquant.
- exceptfds : voir **man** pour une explication détaillée

Lorsque l'état de l'un des descripteurs change **select** retourne une valeur > 0, et les ensembles sont modifiés. Il faut par la suite vérifier l'état de chacun des descripteurs des ensembles via des macros présentées plus bas.

De plus amples informations : **select** mais aussi un tutoriel sur **l'utilisation de select dans le man**

Afin de manipuler les ensembles des descripteurs, plusieurs macros sont à notre disposition.

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

- FD_CLR supprime le descripteur *fd* de l'ensemble *set*.
- FD_ISSET vérifie si le descripteur *fd* est contenu dans l'ensemble *set* après l'appel à **select**
- FD_SET ajoute le descripteur *fd* à l'ensemble *set*
- FD_ZERO vide l'ensemble *set*

4-1-1 - Utilisation

Tout d'abord il faut remplir les différents ensembles. En général on n'utilise que l'ensemble des descripteurs en lecture.

```
SOCKET sock;
[...];
fd_set readfs;

while(1)
{
   int ret = 0;
   FD_ZERO(&readfs);
   FD_SET(sock, &readfs);

   if((ret = select(sock + 1, &readfs, NULL, NULL, NULL)) < 0)
   {
      perror("select()");
      exit(errno);
   }

   /*
   if(ret == 0)
   {
      ici le code si la temporisation (dernier argument) est écoulée (il faut bien évidemment avoir mis quelque c
```

```
    }
    */

    if(FD_ISSET(sock, readfs))
    {
       /* des données sont disponibles sur le socket */
       /* traitement des données */
    }
}
```

Après chaque appel à select on vérifie les différents descripteurs de nos ensembles, une fois les données traitées, il faut remettre nos descripteurs dans nos ensembles.

*Sous Linux **stdin** est considéré comme un descripteur, il peut donc être ajouté à un ensemble et être utilisé avec select. En revanche sous Windows cela ne fonctionne pas. Il ne faut donc pas utiliser STDIN_FILENO et select sous Windows.*

5 - Un client/serveur

5-1 - Présentation

Dans ce paragraphe, nous allons voir la création d'un petit client/serveur, de type chat.

5-2 - Fonctionnement général

Le serveur attend la connexion des clients. Quand un client se connecte et envoie un message, le message est transmis à tous les clients.

*Les sockets utilisés dans le code source sont parfaitement portables, en revanche pour ne pas alourdir le code et parce qu'il convient parfaitement ici, j'ai utilisé **select** à la place d'utiliser des thread. Pour la saisie clavier j'ai donc ajouter STDIN_FILENO à **select**, or le comportement n'est pas celui attendu sous Windows, en conséquence le code ne fonctionne pas sous ce dernier. Pour pallier à ça tout en gardant le plus de code possible, il est possible de remplacer **select** par des threads portables, type **pthread** ou bien de trouver une autre solution pour la saisie de message sur le client (interface graphique...).*

5-3 - Version TCP

Voici donc le code source de la version TCP

5-3-1 - Client

5-3-1-a - Code source

client.h

```
#ifndef CLIENT_H
#define CLIENT_H

#ifdef WIN32

#include <winsock2.h>

#elif defined (linux)

#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <unistd.h> /* close */
#include <netdb.h> /* gethostbyname */
#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;
typedef struct in_addr IN_ADDR;

#else

#error not defined for this platform

#endif

#define CRLF  "\r\n"
```

client.h

```c
#define PORT  1977

#define BUF_SIZE 1024

static void init(void);
static void end(void);
static void app(const char *address, const char *name);
static int init_connection(const char *address);
static void end_connection(int sock);
static int read_server(SOCKET sock, char *buffer);
static void write_server(SOCKET sock, const char *buffer);

#endif /* guard */
```

main.c

```c
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#include <string.h>

#include "client.h"

static void init(void)
{
#ifdef WIN32
   WSADATA wsa;
   int err = WSAStartup(MAKEWORD(2, 2), &wsa);
   if(err < 0)
   {
      puts("WSAStartup failed !");
      exit(EXIT_FAILURE);
   }
#endif
}

static void end(void)
{
#ifdef WIN32
   WSACleanup();
#endif
}

static void app(const char *address, const char *name)
{
   SOCKET sock = init_connection(address);
   char buffer[BUF_SIZE];

   fd_set rdfs;

   /* send our name */
   write_server(sock, name);

   while(1)
   {
      FD_ZERO(&rdfs);

      /* add STDIN_FILENO */
      FD_SET(STDIN_FILENO, &rdfs);

      /* add the socket */
      FD_SET(sock, &rdfs);

      if(select(sock + 1, &rdfs, NULL, NULL, NULL) == -1)
      {
         perror("select()");
         exit(errno);
      }

      /* something from standard input : i.e keyboard */
      if(FD_ISSET(STDIN_FILENO, &rdfs))
```

main.c

```
        {
            fgets(buffer, BUF_SIZE - 1, stdin);
            {
                char *p = NULL;
                p = strstr(buffer, "\n");
                if(p != NULL)
                {
                    *p = 0;
                }
                else
                {
                    /* fclean */
                    buffer[BUF_SIZE - 1] = 0;
                }
            }
            write_server(sock, buffer);
        }
        else if(FD_ISSET(sock, &rdfs))
        {
            int n = read_server(sock, buffer);
            /* server down */
            if(n == 0)
            {
                printf("Server disconnected !\n");
                break;
            }
            puts(buffer);
        }
    }

    end_connection(sock);
}

static int init_connection(const char *address)
{
    SOCKET sock = socket(AF_INET, SOCK_STREAM, 0);
    SOCKADDR_IN sin = { 0 };
    struct hostent *hostinfo;

    if(sock == INVALID_SOCKET)
    {
        perror("socket()");
        exit(errno);
    }

    hostinfo = gethostbyname(address);
    if (hostinfo == NULL)
    {
        fprintf (stderr, "Unknown host %s.\n", address);
        exit(EXIT_FAILURE);
    }

    sin.sin_addr = *(IN_ADDR *) hostinfo->h_addr;
    sin.sin_port = htons(PORT);
    sin.sin_family = AF_INET;

    if(connect(sock,(SOCKADDR *) &sin, sizeof(SOCKADDR)) == SOCKET_ERROR)
    {
        perror("connect()");
        exit(errno);
    }

    return sock;
}

static void end_connection(int sock)
{
    closesocket(sock);
}

static int read_server(SOCKET sock, char *buffer)
```

main.c
```
{
   int n = 0;

   if((n = recv(sock, buffer, BUF_SIZE - 1, 0)) < 0)
   {
      perror("recv()");
      exit(errno);
   }

   buffer[n] = 0;

   return n;
}

static void write_server(SOCKET sock, const char *buffer)
{
   if(send(sock, buffer, strlen(buffer), 0) < 0)
   {
      perror("send()");
      exit(errno);
   }
}

int main(int argc, char **argv)
{
   if(argc < 2)
   {
      printf("Usage : %s [address] [pseudo]\n", argv[0]);
      return EXIT_FAILURE;
   }

   init();

   app(argv[1], argv[2]);

   end();

   return EXIT_SUCCESS;
}
```

5-3-1-b - Fonctionnement

Tout d'abord nous créons notre socket, puis nous nous connectons au serveur. Une fois la connexion effectuée nous envoyons notre pseudo puis nous attendons l'un des événements suivants grâce à select: réception d'un message du serveur ou saisie d'un message par le client.
Si un message est reçu, nous l'affichons et nous repartons dans l'attente d'un événement.
Si le client saisit un message, nous l'envoyons au serveur puis nous nous remettons en attente d'un événement.

*Comme déjà dit, **select** avec STDIN_FILENO ne fonctionne pas sous Windows, il faudrait donc trouver une alternative pour la saisie de message (khbit(), interface graphique...).*

5-3-2 - Serveur

5-3-2-a - Code source

server.h
```
#ifndef SERVER_H
#define SERVER_H

#ifdef WIN32

#include <winsock2.h>
```

server.h

```
#elif defined (linux)

#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <unistd.h> /* close */
#include <netdb.h> /* gethostbyname */
#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;
typedef struct in_addr IN_ADDR;

#else

#error not defined for this platform

#endif

#define CRLF  "\r\n"
#define PORT   1977
#define MAX_CLIENTS  100

#define BUF_SIZE 1024

#include "client.h"

static void init(void);
static void end(void);
static void app(void);
static int init_connection(void);
static void end_connection(int sock);
static int read_client(SOCKET sock, char *buffer);
static void write_client(SOCKET sock, const char *buffer);
static void send_message_to_all_clients(Client *clients, Client client, int
 actual, const char *buffer, char from_server);
static void remove_client(Client *clients, int to_remove, int *actual);
static void clear_clients(Client *clients, int actual);

#endif /* guard */
```

client.h

```
#ifndef CLIENT_H
#define CLIENT_H

#include "server.h"

typedef struct
{
   SOCKET sock;
   char name[BUF_SIZE];
}Client;

#endif /* guard */
```

main.c

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

#include "server.h"
#include "client.h"

static void init(void)
{
```

main.c

```
#ifdef WIN32
   WSADATA wsa;
   int err = WSAStartup(MAKEWORD(2, 2), &wsa);
   if(err < 0)
   {
      puts("WSAStartup failed !");
      exit(EXIT_FAILURE);
   }
#endif
}

static void end(void)
{
#ifdef WIN32
   WSACleanup();
#endif
}

static void app(void)
{
   SOCKET sock = init_connection();
   char buffer[BUF_SIZE];
   /* the index for the array */
   int actual = 0;
   int max = sock;
   /* an array for all clients */
   Client clients[MAX_CLIENTS];

   fd_set rdfs;

   while(1)
   {
      int i = 0;
      FD_ZERO(&rdfs);

      /* add STDIN_FILENO */
      FD_SET(STDIN_FILENO, &rdfs);

      /* add the connection socket */
      FD_SET(sock, &rdfs);

      /* add socket of each client */
      for(i = 0; i < actual; i++)
      {
         FD_SET(clients[i].sock, &rdfs);
      }

      if(select(max + 1, &rdfs, NULL, NULL, NULL) == -1)
      {
         perror("select()");
         exit(errno);
      }

      /* something from standard input : i.e keyboard */
      if(FD_ISSET(STDIN_FILENO, &rdfs))
      {
         /* stop process when type on keyboard */
         break;
      }
      else if(FD_ISSET(sock, &rdfs))
      {
         /* new client */
         SOCKADDR_IN csin = { 0 };
         size_t sinsize = sizeof csin;
         int csock = accept(sock, (SOCKADDR *)&csin, &sinsize);
         if(csock == SOCKET_ERROR)
         {
            perror("accept()");
            continue;
         }
```

main.c

```
         /* after connecting the client sends its name */
         if(read_client(csock, buffer) == -1)
         {
            /* disconnected */
            continue;
         }

         /* what is the new maximum fd ? */
         max = csock > max ? csock : max;

         FD_SET(csock, &rdfs);

         Client c = { csock };
         strncpy(c.name, buffer, BUF_SIZE - 1);
         clients[actual] = c;
         actual++;
      }
      else
      {
         int i = 0;
         for(i = 0; i < actual; i++)
         {
            /* a client is talking */
            if(FD_ISSET(clients[i].sock, &rdfs))
            {
               Client client = clients[i];
               int c = read_client(clients[i].sock, buffer);
               /* client disconnected */
               if(c == 0)
               {
                  closesocket(clients[i].sock);
                  remove_client(clients, i, &actual);
                  strncpy(buffer, client.name, BUF_SIZE - 1);
                  strncat(buffer, " disconnected !", BUF_SIZE - strlen(buffer) - 1);
                  send_message_to_all_clients(clients, client, actual, buffer, 1);
               }
               else
               {
                  send_message_to_all_clients(clients, client, actual, buffer, 0);
               }
               break;
            }
         }
      }
   }

   clear_clients(clients, actual);
   end_connection(sock);
}

static void clear_clients(Client *clients, int actual)
{
   int i = 0;
   for(i = 0; i < actual; i++)
   {
      closesocket(clients[i].sock);
   }
}

static void remove_client(Client *clients, int to_remove, int *actual)
{
   /* we remove the client in the array */
   memmove(clients + to_remove, clients + to_remove + 1, (*actual - to_remove - 1) * sizeof(Client));
   /* number client - 1 */
   (*actual)--;
}

static void send_message_to_all_clients(Client *clients, Client sender, int
 actual, const char *buffer, char from_server)
{
   int i = 0;
```

main.c

```
    char message[BUF_SIZE];
    message[0] = 0;
    for(i = 0; i < actual; i++)
    {
       /* we don't send message to the sender */
       if(sender.sock != clients[i].sock)
       {
          if(from_server == 0)
          {
             strncpy(message, sender.name, BUF_SIZE - 1);
             strncat(message, " : ", sizeof message - strlen(message) - 1);
          }
          strncat(message, buffer, sizeof message - strlen(message) - 1);
          write_client(clients[i].sock, message);
       }
    }
}

static int init_connection(void)
{
   SOCKET sock = socket(AF_INET, SOCK_STREAM, 0);
   SOCKADDR_IN sin = { 0 };

   if(sock == INVALID_SOCKET)
   {
      perror("socket()");
      exit(errno);
   }

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

   if(bind(sock,(SOCKADDR *) &sin, sizeof sin) == SOCKET_ERROR)
   {
      perror("bind()");
      exit(errno);
   }

   if(listen(sock, MAX_CLIENTS) == SOCKET_ERROR)
   {
      perror("listen()");
      exit(errno);
   }

   return sock;
}

static void end_connection(int sock)
{
   closesocket(sock);
}

static int read_client(SOCKET sock, char *buffer)
{
   int n = 0;

   if((n = recv(sock, buffer, BUF_SIZE - 1, 0)) < 0)
   {
      perror("recv()");
      /* if recv error we disonnect the client */
      n = 0;
   }

   buffer[n] = 0;

   return n;
}

static void write_client(SOCKET sock, const char *buffer)
{
```

main.c
```
    if(send(sock, buffer, strlen(buffer), 0) < 0)
    {
       perror("send()");
       exit(errno);
    }
}

int main(int argc, char **argv)
{
   init();

   app();

   end();

   return EXIT_SUCCESS;
}
```

5-3-2-b - Fonctionnement

Tout d'abord nous créons notre socket et notre interface de connexion, une fois fait, nous attendons l'un des événements suivants : saisie au clavier, connexion d'un client, envoi d'un message par un client.
Si quelque chose est saisit au clavier nous arrêtons le programme (moyen propre au lieu de faire Ctrl+C).
Si un client se connecte, nous acceptons la connexion et nous attendons la réception de son pseudo, une fois reçu, nous créons un objet de type Client (voir source), contenant le socket pour discuter avec lui ainsi que son pseudo. Nous ajoutons maintenant notre client à notre tableau de clients.
Si un client envoie un message, nous le diffusons à tous les autres clients. Lorsqu'un client se déconnecte (Ctrl+C), **recv** retourne 0, nous gérons donc ce cas et donc nous supprimons notre client de notre tableau tout en notifiant aux autres clients sa déconnexion.


select *servant ici à couper proprement le serveur, et ce dernier ne fonctionnant pas sous Windows, il est tout à fait possible de ne pas ajouter STDIN_FILENO à* ***select****, et donc de couper le serveur via un CTRL+C.*

5-4 - Version UDP

Maintenant voici la version UDP.

5-4-1 - Client

5-4-1-a - Code source

client.h
```
#ifndef CLIENT_H
#define CLIENT_H

#ifdef WIN32

#include <winsock2.h>

#elif defined (linux)

#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <unistd.h> /* close */
#include <netdb.h> /* gethostbyname */
#define INVALID_SOCKET -1
#define SOCKET_ERROR -1
```

client.h

```
#define closesocket(s) close(s)
typedef int SOCKET;
typedef struct sockaddr_in SOCKADDR_IN;
typedef struct sockaddr SOCKADDR;
typedef struct in_addr IN_ADDR;

#else

#error not defined for this platform

#endif

#define CRLF  "\r\n"
#define PORT  1977

#define BUF_SIZE 1024

static void init(void);
static void end(void);
static void app(const char *address, const char *name);
static int init_connection(const char *address, SOCKADDR_IN *sin);
static void end_connection(int sock);
static int read_server(SOCKET sock, SOCKADDR_IN *sin, char *buffer);
static void write_server(SOCKET sock, SOCKADDR_IN *sin, const char *buffer);

#endif /* guard */
```

main.c

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

#include "client.h"

static void init(void)
{
#ifdef WIN32
   WSADATA wsa;
   int err = WSAStartup(MAKEWORD(2, 2), &wsa);
   if(err < 0)
   {
      puts("WSAStartup failed !");
      exit(EXIT_FAILURE);
   }
#endif
}

static void end(void)
{
#ifdef WIN32
   WSACleanup();
#endif
}

static void app(const char *address, const char *name)
{
   SOCKADDR_IN sin = { 0 };
   SOCKET sock = init_connection(address, &sin);
   char buffer[BUF_SIZE];

   fd_set rdfs;

   /* send our name */
   write_server(sock, &sin, name);

   while(1)
   {
      FD_ZERO(&rdfs);
```

main.c

```
        /* add STDIN_FILENO */
        FD_SET(STDIN_FILENO, &rdfs);

        /* add the socket */
        FD_SET(sock, &rdfs);

        if(select(sock + 1, &rdfs, NULL, NULL, NULL) == -1)
        {
           perror("select()");
           exit(errno);
        }

        /* something from standard input : i.e keyboard */
        if(FD_ISSET(STDIN_FILENO, &rdfs))
        {
           fgets(buffer, BUF_SIZE - 1, stdin);
           {
              char *p = NULL;
              p = strstr(buffer, "\n");
              if(p != NULL)
              {
                 *p = 0;
              }
              else
              {
                 /* fclean */
                 buffer[BUF_SIZE - 1] = 0;
              }
           }
           write_server(sock, &sin, buffer);
        }
        else if(FD_ISSET(sock, &rdfs))
        {
           int n = read_server(sock, &sin, buffer);
           /* server down */
           if(n == 0)
           {
              printf("Server disconnected !\n");
              break;
           }
           puts(buffer);
        }
     }

     end_connection(sock);
}

static int init_connection(const char *address, SOCKADDR_IN *sin)
{
   /* UDP so SOCK_DGRAM */
   SOCKET sock = socket(AF_INET, SOCK_DGRAM, 0);
   struct hostent *hostinfo;

   if(sock == INVALID_SOCKET)
   {
      perror("socket()");
      exit(errno);
   }

   hostinfo = gethostbyname(address);
   if (hostinfo == NULL)
   {
      fprintf (stderr, "Unknown host %s.\n", address);
      exit(EXIT_FAILURE);
   }

   sin->sin_addr = *(IN_ADDR *) hostinfo->h_addr;
   sin->sin_port = htons(PORT);
   sin->sin_family = AF_INET;

   return sock;
```

main.c
```
}

static void end_connection(int sock)
{
   closesocket(sock);
}

static int read_server(SOCKET sock, SOCKADDR_IN *sin, char *buffer)
{
   int n = 0;
   size_t sinsize = sizeof *sin;

   if((n = recvfrom(sock, buffer, BUF_SIZE - 1, 0, (SOCKADDR *) sin, &sinsize)) < 0)
   {
      perror("recvfrom()");
      exit(errno);
   }

   buffer[n] = 0;

   return n;
}

static void write_server(SOCKET sock, SOCKADDR_IN *sin, const char *buffer)
{
   if(sendto(sock, buffer, strlen(buffer), 0, (SOCKADDR *) sin, sizeof *sin) < 0)
   {
      perror("sendto()");
      exit(errno);
   }
}

int main(int argc, char **argv)
{
   if(argc < 2)
   {
      printf("Usage : %s [address] [pseudo]\n", argv[0]);
      return EXIT_FAILURE;
   }

   init();

   app(argv[1], argv[2]);

   end();

   return EXIT_SUCCESS;
}
```

5-4-1-b - Fonctionnement

Le fonctionnement est plus ou moins le même que la version TCP, à la différence près qu'il n'y a pas la partie connexion au serveur.

5-4-2 - Serveur

5-4-2-a - Code source

server.h
```
#ifndef SERVER_H
#define SERVER_H

#ifdef WIN32

#include <winsock2.h>
```

server.h

```
#elif defined (linux)

#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <unistd.h> /* close */
#include <netdb.h> /* gethostbyname */
#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;
typedef struct in_addr IN_ADDR;

#else

#error not defined for this platform

#endif

#define CRLF  "\r\n"
#define PORT   1977
#define MAX_CLIENTS  100

#define BUF_SIZE 1024

#include "client.h"

static void init(void);
static void end(void);
static void app(void);
static int init_connection(void);
static void end_connection(int sock);
static int read_client(SOCKET sock, SOCKADDR_IN *csin, char *buffer);
static void write_client(SOCKET sock, SOCKADDR_IN *csin, const char *buffer);
static void send_message_to_all_clients(int sock, Client *clients, Client *client, int
 actual, const char *buffer, char from_server);
static void remove_client(Client *clients, int to_remove, int *actual);
static int check_if_client_exists(Client *clients, SOCKADDR_IN *csin, int actual);
static Client* get_client(Client *clients, SOCKADDR_IN *csin, int actual);

#endif /* guard */
```

client.h

```
#ifndef CLIENT_H
#define CLIENT_H

#include "server.h"

typedef struct
{
   SOCKADDR_IN sin;
   char name[BUF_SIZE];
}Client;

#endif /* guard */
```

main.c

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

#include "server.h"
#include "client.h"
```

main.c

```
static void init(void)
{
#ifdef WIN32
   WSADATA wsa;
   int err = WSAStartup(MAKEWORD(2, 2), &wsa);
   if(err < 0)
   {
      puts("WSAStartup failed !");
      exit(EXIT_FAILURE);
   }
#endif
}

static void end(void)
{
#ifdef WIN32
   WSACleanup();
#endif
}

static void app(void)
{
   SOCKET sock = init_connection();
   char buffer[BUF_SIZE];
   /* the index for the array */
   int actual = 0;
   int max = sock;
   /* an array for all clients */
   Client clients[MAX_CLIENTS];

   fd_set rdfs;

   while(1)
   {
      FD_ZERO(&rdfs);

      /* add STDIN_FILENO */
      FD_SET(STDIN_FILENO, &rdfs);

      /* add the connection socket */
      FD_SET(sock, &rdfs);

      if(select(max + 1, &rdfs, NULL, NULL, NULL) == -1)
      {
  perror("select()");
  exit(errno);
      }

      /* something from standard input : i.e keyboard */
      if(FD_ISSET(STDIN_FILENO, &rdfs))
      {
  /* stop process when type on keyboard */
  break;
      }
      else if(FD_ISSET(sock, &rdfs))
      {
  /* new client */
  SOCKADDR_IN csin = { 0 };

  /* a client is talking */
  read_client(sock, &csin, buffer);

  if(check_if_client_exists(clients, &csin, actual) == 0)
  {
     if(actual != MAX_CLIENTS)
     {
        Client c = { csin };
        strncpy(c.name, buffer, BUF_SIZE - 1);
        clients[actual] = c;
        actual++;
     }
```

main.c

```
   }
   else
   {
      Client *client = get_client(clients, &csin, actual);
      if(client == NULL) continue;
      send_message_to_all_clients(sock, clients, client, actual, buffer, 0);
   }
       }
    }

    end_connection(sock);
}

static int check_if_client_exists(Client *clients, SOCKADDR_IN *csin, int actual)
{
   int i = 0;
   for(i = 0; i < actual; i++)
   {
      if(clients[i].sin.sin_addr.s_addr == csin->sin_addr.s_addr
     && clients[i].sin.sin_port == csin->sin_port)
      {
  return 1;
      }
   }

   return 0;
}

static Client* get_client(Client *clients, SOCKADDR_IN *csin, int actual)
{
   int i = 0;
   for(i = 0; i < actual; i++)
   {
      if(clients[i].sin.sin_addr.s_addr == csin->sin_addr.s_addr
     && clients[i].sin.sin_port == csin->sin_port)
      {
  return &clients[i];
      }
   }

   return NULL;
}

static void remove_client(Client *clients, int to_remove, int *actual)
{
   /* we remove the client in the array */
   memmove(clients + to_remove, clients + to_remove + 1, (*actual - to_remove) * sizeof(Client));
   /* number client - 1 */
   (*actual)--;
}

static void send_message_to_all_clients(int sock, Client *clients, Client *sender, int
 actual, const char *buffer, char from_server)
{
   int i = 0;
   char message[BUF_SIZE];
   message[0] = 0;
   for(i = 0; i < actual; i++)
   {
      /* we don't send message to the sender */
      if(sender != &clients[i])
      {
  if(from_server == 0)
  {
     strncpy(message, sender->name, BUF_SIZE - 1);
     strncat(message, " : ", sizeof message - strlen(message) - 1);
  }
  strncat(message, buffer, sizeof message - strlen(message) - 1);
  write_client(sock, &clients[i].sin, message);
      }
```

main.c

```
    }
}

static int init_connection(void)
{
   /* UDP so SOCK_DGRAM */
   SOCKET sock = socket(AF_INET, SOCK_DGRAM, 0);
   SOCKADDR_IN sin = { 0 };

   if(sock == INVALID_SOCKET)
   {
      perror("socket()");
      exit(errno);
   }

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

   if(bind(sock,(SOCKADDR *) &sin, sizeof sin) == SOCKET_ERROR)
   {
      perror("bind()");
      exit(errno);
   }

   return sock;
}

static void end_connection(int sock)
{
   closesocket(sock);
}

static int read_client(SOCKET sock, SOCKADDR_IN *sin, char *buffer)
{
   int n = 0;
   size_t sinsize = sizeof *sin;

   if((n = recvfrom(sock, buffer, BUF_SIZE - 1, 0, (SOCKADDR *) sin, &sinsize)) < 0)
   {
      perror("recvfrom()");
      exit(errno);
   }

   buffer[n] = 0;

   return n;
}

static void write_client(SOCKET sock, SOCKADDR_IN *sin, const char *buffer)
{
   if(sendto(sock, buffer, strlen(buffer), 0, (SOCKADDR *) sin, sizeof *sin) < 0)
   {
      perror("send()");
      exit(errno);
   }
}

int main(int argc, char **argv)
{
   init();

   app();

   end();

   return EXIT_SUCCESS;
}
```

5-4-2-b - Fonctionnement

Le fonctionnement est ici aussi similaire à la version TCP, à la différence qu'il n'y a pas la partie acceptation de la connexion.
Lorsqu'un client parle recvfrom remplit la structure **SOCKADDR_IN** avec ses infos. On vérifie ensuite si un client avec les mêmes infos (adresse IP et port) existe déjà dans notre tableau de clients, si oui on envoie le message à tous les autres clients, si non, cela signifie que c'est la première fois que le client parle, on crée donc un objet de type Client (différent de la version TCP / voir code source) et on l'ajoute à notre tableau de clients. L'égalité de 2 clients en UDP est basée sur l'adresse IP et le port qui ne peuvent pas être tous les 2 égaux si les clients sont différents.

*Toujours pareil pour le **select** sous Windows.*
Le problème ici est de détecter la déconnexion d'un client puisqu'il n'y a pas de connexion. Lorsqu'un client est ajouté à notre tableau il n'est jamais supprimé. Pour pallier à ça voici quelques solutions qui peuvent être mises en place. Première solution, envoi d'une commande type pour quitter (/quit par exemple) mais reste le problème du client qui se déconnecte sans envoyer /quit (Ctrl+C, arrêt du processus...).
Seconde solution, le serveur garde tous les messages et ne sauvegarde aucun client, c'est le client lui-même, qui, régulièrement demande au serveur si il y a de nouveaux messages. UDP n'est pas très adapté pour ce genre de situation.

6 - Les 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.

7 - Liens

Voici un lien d'utilisation assez concrète des sockets, avec le protocole SMTP.

http://broux.developpez.com/articles/protocoles/smtp/

8 - Remerciements

Tout d'abord merci à l'équipe C pour m'avoir donné leurs avis, et plus particulièrement **buchs** pour toutes ses critiques et son avis.
Merci ensuite à ma petite pomme dauphine (**LineLe**) pour sa relecture et ses corrections orthographiques.