Les sockets

1. Concept Client-Serveur

Quand deux tâches veulent communiquer, elles doivent être actives toutes les deux :

- Une tâche est en permanence à l'écoute des clients, c'est la tâche qui offre (sert) un service. Elle est située sur le PC Serveur.
- La tâche qui demande le service est sur le PC Client.

Pour créer une application de communication, il faut programmer le serveur et le client en utilisant les sockets. Il y a diverses sortes de sockets, nous traitons des sockets Internet.

1.1. Le Serveur

La tâche serveur est lancée au démarrage de la machine. Elle est en sommeil et se réveille quand elle est sollicitée par un client, elle ne prend pas l'initiative d'une communication. En mode connecté, on dit que la connexion est passive :

- L'adresse IP et le numéro de port du serveur doivent être connus de tous les clients potentiels.
- Le serveur ne connaît pas les adresses IP des clients et n'a pas besoin de les connaître avant leur connexion.

1.2. Le Client

C'est une application située sur une machine distante du serveur. Elle peut envoyer une demande de connexion (mode connecté) ou directement une requête (mode non-connecté).

Elle doit connaître l'adresse IP et le numéro de port de l'application serveur ainsi que les services que cette application propose. L'adresse IP du client ne sert à l'application serveur que pour lui répondre.

1.3. Choix d'un service avec ou sans connexion

On choisit UDP dans les cas suivants :

- Requête du client et réponse très courtes,
- Application très spécifique qui ne peut supporter ni le temps perdu, ni la charge du réseau imposés par la connexion.
- l'application prend en charge les services rendus par TCP (erreur, contrôle de flux, …).

On choisit TCP tous les autres cas.

2. Les sockets

2.1. Définition

Ce sont des points de communication par lesquels un processus peut émettre ou recevoir.

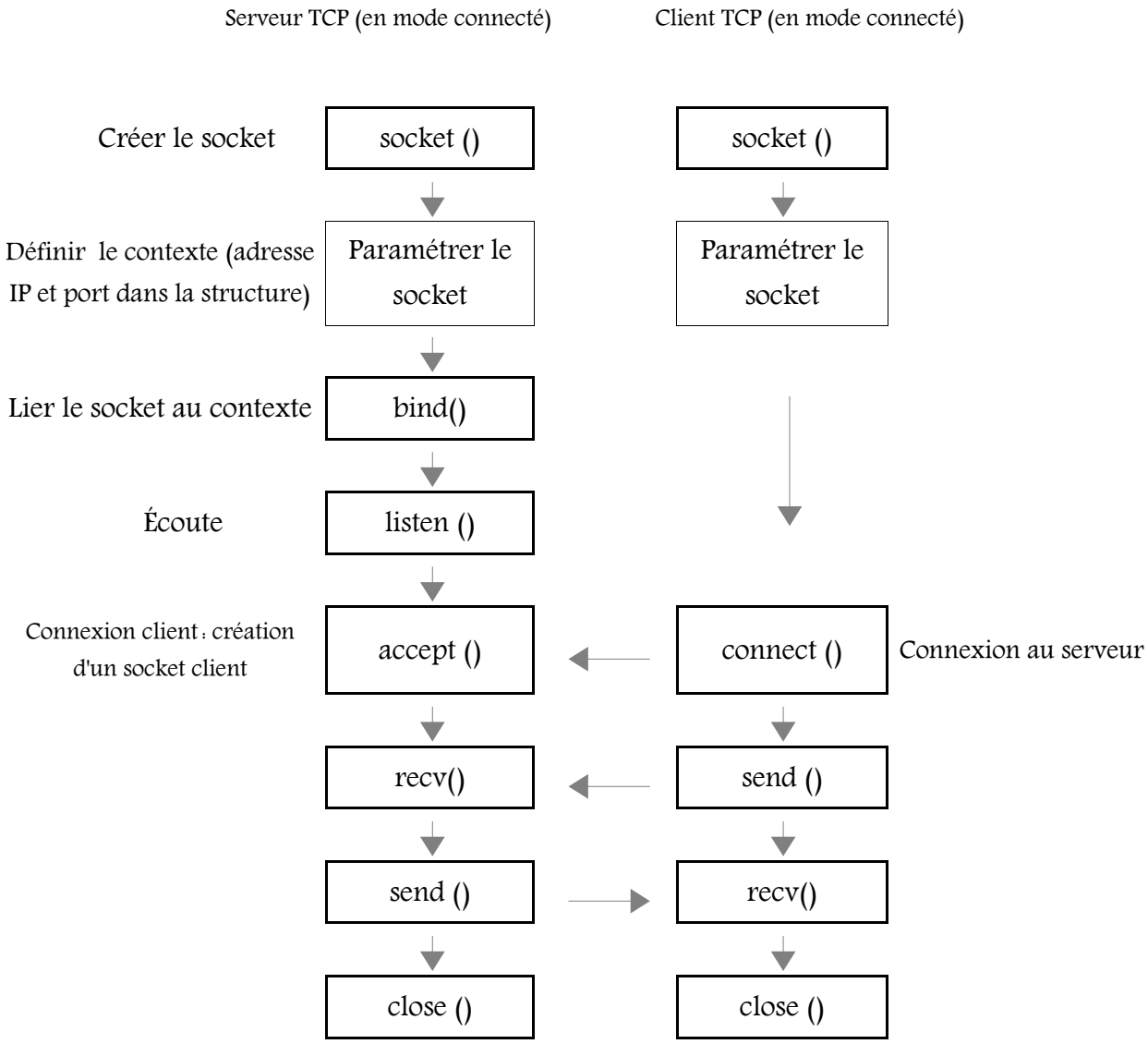
Les types de sockets sont :

- Les sockets de type "flots de données" (**stream sockets**), qui utilisent TCP. Ex Telnet, HTTP
- Les sockets de type "datagramme" (**datagram sockets**), qui utilisent UDP. Ex TFTP
- Les sockets de type "accès direct aux couches basses" (**raw sockets**) qui utilisent directement IP, ICMP (ex : ping)

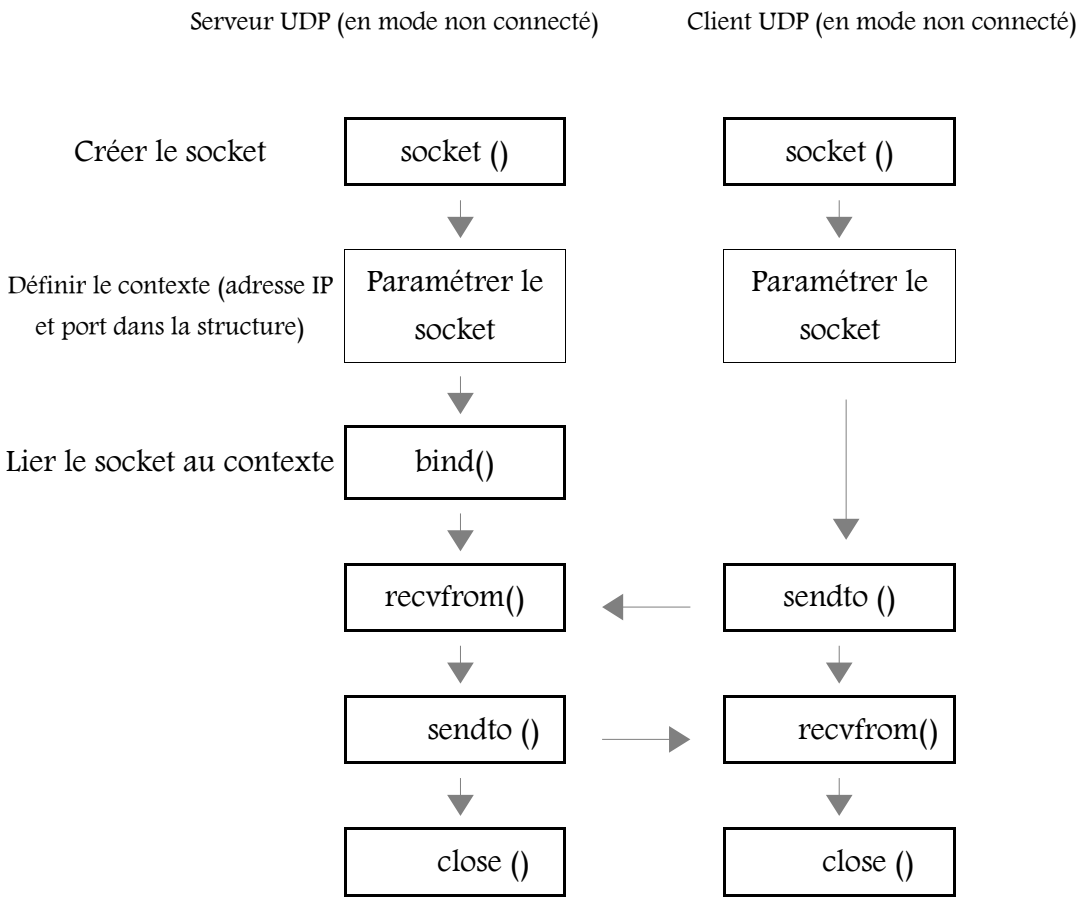
Un socket est défini par la paire (adresse IP, numéro de port). La combinaison de 2 sockets (une au niveau serveur, l'autre au niveau client) définit complètement une connexion TCP ou un échange UDP.

2.2. Les routines mises en œuvre

2.2.1. Routines en mode connecté : TCP


Serveur TCP (en mode connecté)
Client TCP (en mode connecté)
Créer le socket
socket ()
socket ()
Définir le contexte (adresse IP et port dans la structure)
Paramétrer le socket
Paramétrer le socket
Lier le socket au contexte
bind()
Écoute
listen ()
Connexion client : création d'un socket client
accept ()
connect ()
Connexion au serveur
recv()
send ()
send ()
recv()
close ()
close ()

2.2.2. Routines en mode non connecté : UDP



2.3. Initialisation des Sockets

Les sockets ne s'utilisent pas de manière identique selon les différents systèmes d'exploitation. Dans ce cours on verra le cas de Windows (Linux) et le langage C/C++

2.3.1. Librairies

Dans chaque projet crée, il faut ajouter dans votre éditeur de liens l'indication de la librairie utilisée :

- Le fichier "ws2_32.lib" pour le compilateur Visual C++
- Le fichier "libws2_32.a" pour les autres compilateurs.

Ces fichiers sont dans le dossier "lib" de votre IDE (Integrated Development Environment = Environnement de développement intégré ou environnement de développement logiciel)

Et déclarer les sockets Windows par le fichier "winsock2.h", dans le dossier header de votre IDE (fichier standard de Windows). On l'inclut dans le programme comme suit :

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

2.3.2. Fonctions WSAStartup et WSACleanup

WSAStartup sert à initialiser la bibliothèque WinSock.

Code en C :

```
WSADATA WSAData;

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

La macro MAKEWORD transforme ses deux paramètres (deux entiers d'un octet chacun) 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).

WSAStartup retourne la valeur 0 si tout s'est bien passé.

WSACleanup sert à libérer les ressources allouées par la fonction WSAStartup(). Elle doit être placée en fin de programme.

Code en C :

```
WSACleanup();
```

2.4. Fonctions de conversion

```
unsigned long htonl(unsigned long hostlong);
```

Convertit l'entier de 4 octets hostlong depuis l'ordre des octets de l'hôte vers celui du réseau.

```
unsigned short htons(unsigned short hostshort);
```

Convertit l'entier de 2 octets hostshort depuis l'ordre des octets de l'hôte vers celui du réseau.

```
unsigned long ntohl(unsigned long netlong);
```

Convertit l'entier de 4 octets netlong depuis l'ordre des octets du réseau vers celui de l'hôte.

```
unsigned short ntohs(unsigned short netshort);
```

Convertit l'entier de 2 octets `netshort` depuis l'ordre des octets du réseau vers celui de l'hôte.

```
inet_addr("192.168.10.1")
```

La fonction **inet_addr()** convertit l'adresse Internet de l'hôte depuis la notation décimale pointée en une donnée binaire dans l'ordre des octets du réseau.

Si l'adresse est invalide, INADDR_NONE (généralement -1) est renvoyé. Attention, si l'adresse IP vaut 255.255.255.255 le résultat renvoyé est aussi -1.

```
inet_ntoa (sin.sin_addr)
```

La fonction **inet_ntoa()** convertit l'adresse Internet de l'hôte donnée dans l'ordre des octets du réseau en une chaîne de caractères dans la notation décimale pointée. La chaîne est renvoyée dans un buffer alloué statiquement, qui est donc écrasé à chaque appel.

Cette fonction est utile si l'on veut par exemple afficher (ou imprimer) l'adresse IP.

La structure *in_addr* utilisée dans **inet_ntoa()**, **inet_makeaddr()**, **inet_lnoaf()** et **inet_netof()** est :

```
struct in_addr {
unsigned long int s_addr;
}
```

> *Notez que l'ordre des octets des i80x86 est LSB (poids faible en premier), alors que l'ordre des octets sur l'Internet est MSB (poids fort en premier).*

3. L'application serveur

3.1. Créer un socket : fonction socket

Elle permet de créer un socket en déclarant son type d'adresse et son protocole. La création consiste en l'affectation d'un numéro au socket crée.

Il faut déclarer le socket avec le type SOCKET et la créer avec la fonction socket

Prototype de la fonction :

```
SOCKET ssock;

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

La fonction retourne un identifiant du socket créé, en cas d'erreur elle retourne le code :

INVALID_SOCKET.

- **addr_family** représente la famille de protocoles utilisée.
  - AF_INET pour IPv4
  - AF_INET16 pour IPv6
  - ...
- **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
  - …

À titre d'exemple voici les définitions extraites de winsock2.h :

```
#define SOCK_STREAM 1
#define SOCK_DGRAM  2
#define SOCK_RAW    3
```

- **protocol** définit le protocole à utiliser (dépend de addr_family et de type).
  - IPPROTO_TCP pour TCP

- IPPROTO_UDP pour UDP
- 0 est une valeur par défaut (qui permet un choix automatique de TCP ou UDP). C'est ce qu'on utilisera en général.

Voici la définition extraite de winsock2.h :

```
#define IPPROTO_UDP 17
#define IPPROTO_TCP 6
```

Exemple de Code en C :

```
SOCKET ssock;

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

Ce qui équivaut à :

```
SOCKET ssock;

ssock = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP);
```

3.2. Paramétrer un socket

Paramétrer le socket c'est créer une structure de type SOCKADDR_IN et configurer ses éléments par les données de la connexion (adresse IP, Port, ...). On l'appelle contexte d'adressage. Cette structure est définie dans les fichiers d'en-tête « winsock2.h », de la manière suivante (Vous n'avez donc plus à la définir dans votre source) :

```
struct sockaddr_in {
        short   sin_family;
        u_short         sin_port;
        struct  in_addr sin_addr;
        char    sin_zero[8];
};
```

Ou :

```
struct sockaddr {
        u_short sa_family;
        char    sa_data[14];
};
```

Notez que la structure ***sin_addr*** *ne contient qu'un seul et unique champ nommé s_addr dont le type importe peu (varie plus ou moins d'un système d'exploitation à un autre).*

SOCKADDR et SOCKADDR_IN sont déclarés dans winsock2.h de la façon suivante :

typedef struct sockaddr SOCKADDR;

```
typedef struct sockaddr_in SOCKADDR_IN;
```

```
SOCKADDR_IN ssin;
```

sin.sin_addr.s_addr = l'adresse IP. Dans le cas d'un serveur, elle est fournie par la fonction htonl avec comme seul paramètre la valeur INADDR_ANY.

Pour spécifier une adresse IP précise (cas du client), on utilise la fonction inet_addr avec comme seul paramètre l'adresse IP dans une chaîne de caractères :
Code en C : inet_addr("127.0.0.1");

sin.sin_family = AF_INET (cas de TCP/IP)
sin.sin_port = la valeur retournée par la fonction htons, avec comme paramètre le port utilisé
sin_zero : non utilisé

Exemple de Code en C

```
SOCKADDR_IN ssin;

ssin.sin_addr.s_addr = htonl(INADDR_ANY);

ssin.sin_family = AF_INET;

ssin.sin_port = htons(2023);
```

3.3. Établir un lien avec le contexte : fonction bind

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

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

La fonction retourne **SOCKET_ERROR** en cas d'erreur.

socket désigne le socket du serveur avec laquelle on va associer les informations.
addr est un pointeur de la structure sockaddr du serveur qui spécifie l'IP à laquelle on se connecte.

sockaddr est une structure «castée» à sockaddr_in donc on peut utiliser l'une ou l'autre.

```
Struct sockaddr {
        sa_family_t sa_family;      /* famille d'adresse */
        char sa_data[14];           /* valeur de l'adresse */
};
```

addrlen = taille mémoire occupée par le contexte d'adressage du serveur.

Le type socklen_t est simplement un alias de « int » utilisé pour être compatible avec d'autres OS. Il peut être défini de la minière suivante :

```
typedef int socklen_t;
```

Code en C :

```
bind(ssock, (SOCKADDR*)&ssin, sizeof(ssin));
```

3.4. État d'écoute : fonction listen

Prototype, Code en C :

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

La fonction retourne **SOCKET_ERROR** si une erreur est survenue.

socket = le socket qui va être utilisée.
backlog = le nombre maximal de connexions pouvant être mises en attente.

Ex : Code en C :

```
listen(ssock, 5);
```

5 est le cas de FTP par exemple.

3.5. Acceptation d'un appel « Client » : fonction accept

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

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

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

Utilisation :

```
typedef int socklen_t

socklen_t taille;

taille = sizeof(csin);

csock = accept(ssock, (SOCKADDR*) &csin, &taille);
```

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

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

3.6. Fermer la connexion : fonction closesocket

```
int closesocket(int ssock);
```

4. L'application client

Pour l'application client, il n'y aura besoin :

- Ni de la fonction bind puisqu'elle est comprise dans la fonction connect
- Ni de listen puisqu'il n'y a pas de sockets à mettre à l'écoute

- Ni d'accept puisque l'application joue le rôle de client.

4.1. Création du socket

Cette étape est exactement identique à l'application serveur.

4.2. Paramétrage du socket

Le paramétrage est identique à l'application serveur sauf pour l'adresse IP. Le client doit désigner l'adresse précise du serveur auquel il veut se connecter d'où l'exemple suivant :

```
SOCKADDR_IN csin;

csin.sin_addr.s_addr = inet_addr("192.168.10.1");

csin.sin_family = AF_INET;

csin.sin_port = htons(2023);
```

4.3. La fonction connect

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

La fonction retourne 0 si la connexion s'est bien déroulée, sinon -1.

socket = le socket à utiliser
addr = pointeur sur le contexte (structure) donnant l'adresse de l'hôte à contacter.
Addrlen = taille de l'adresse de l'appelant (on peut utiliser un sizeof).

```
connect (csock, (SOCKADDR*)&csin, sizeof(csin))
```

csin étant la structure précédemment déclarée et initialisée dans l'application client.

5. La transmission des données

Pour pouvoir réaliser une transmission de données, on utilise trois fonctions :

- La fonction send, pour envoyer les données.
- La fonction recv, pour recevoir ce qu'envoie la fonction send.
- La fonction shutdown, pour désactiver les envois et les réceptions sur le socket.

5.1. La fonction send

```
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.

socket = le socket destinée à recevoir le message.
buffer = un pointeur (comme un tableau) dans lequel figureront nos informations à transmettre.
len = indique le nombre d'octets à lire.
flags = type d'envoi ; (0 pour avoir un envoi normal).

```
send(sock, buffer, sizeof(buffer), 0);
```

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

5.2. La fonction recv

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

Elle retourne **SOCKET_ERROR** en cas d'erreur, sinon elle retourne le nombre d'octets lus.

Socket = le socket destinée à attendre un message.
buffer = un pointeur (un tableau, par exemple) dans lequel seront stockées les informations à recevoir.
len = nombre d'octets à lire.
flags = type d'envoi (0 pour un envoi normal).

```
recv (sock, buffer, sizeof(buffer), 0);
```

5.3. Fonction sendto

Pour le mode non connecté, il faut utiliser sendto pour émettre des données.

```
int sendto(int socket, const char *buffer, int bufferlen,
```

```
int flags, const struct SOCKADDR *sin, int sinlen);
```

Elle retourne **SOCKET_ERROR** en cas d'erreur, sinon elle retourne le nombre d'octets envoyés.

5.4. Fonction recvfrom

Pour le mode non connecté, il faut utiliser recvfrom pour recevoir des données.

```
int recvfrom(int socket, const char *buffer, int bufferlen,
int flags, struct SOCKADDR *sin, int *sinlen);
```

Elle retourne **SOCKET_ERROR** en cas d'erreur, sinon elle retourne le nombre d'octets lus.

int socket : le socket utilisé
const char *buffer : pointeur sur le buffer d'émission ou réception
int bufferlen : longueur du buffer
int flags : mettre à 0
const struct sockaddr *sin : pointeur sur la structure contenant les paramètres de contexte (adresse IP, port, ...)
int sinlen : longueur de la structure
int *sinlen : pointeur sur la longueur de la structure.

5.5. La fonction shutdown

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

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

Socket = socket sur lequel on doit fermer la connexion.
how = comment va se fermer la connexion. Il peut prendre trois valeurs :

- 0 pour fermer le socket en réception
- 1 en émission,
- 2 dans les deux sens.

```
shutdown(sock, 2);
```