

Utilisation des sockets pour des communications TCP/UDP de type client/serveur

Objectifs

- Comprendre le principe de la communication client/serveur.
- Etudier une application client/serveur (en utilisant les sockets) :
 - 1- En mode Connecté : utilisation du protocole TCP.
 - 2- En mode non Connecté : utilisation du protocole UDP.
- Développer un serveur traitant des requêtes de plusieurs clients.

NB.

- Tous les programmes seront écrits en langage C.
- L'environnement de travail est Linux Suse 9.3.

1. Modèle client/serveur

De nombreuses applications fonctionnent selon en environnement client/serveur, cela signifie que des machines clientes contactent un serveur, une machine généralement très puissante en terme de capacités d'entrées-sorties, qui leur fournit des services.

Les services sont exploités par des programmes appelés programmes clients, s'exécutant sur les machines clientes. On parle de client FTP, client de messagerie,... lorsque l'on désigne un programme, tournant sur une machine cliente, capable de traiter des informations qu'il récupère auprès du serveur (par exemple dans le cas du client FTP, il s'agit de fichiers).

Le fonctionnement d'un système client/serveur est comme suit :

- Le client émet une requête vers le serveur grâce à son adresse et le port, qui désigne un service particulier du serveur
- Le serveur reçoit la demande et répond à l'aide de l'adresse de la machine client et son port.

Le modèle client/serveur est particulièrement recommandé pour des réseaux nécessitant un grand niveau de fiabilité, ses principaux avantages sont :

- **Des ressources centralisées** : étant donné que le serveur est au centre du réseau, il peut gérer des ressources communes à tous les utilisateurs, comme par exemple une base de données centralisées, afin d'éviter les problèmes de redondance et de contradiction.
- **Une meilleure sécurité** : car le nombre de points d'entrée permettant l'accès aux données est moins important.
- **Une administration au niveau serveur** : les clients ayant peu d'importance dans ce modèle, ont moins besoin d'être administrés.
- **Un réseau évolutif** : grâce à cette architecture il est possible de supprimer ou rajouter des clients sans perturber le fonctionnement du réseau et sans modifications majeures.

Une connexion est identifiée de façon unique par la donnée de deux couples : une adresse IP et un numéro de port, un pour le client et un autre pour le serveur.

Il est important de noter qu'un dialogue client/serveur n'a pas forcément lieu via un réseau. Il est en effet possible de faire communiquer un client et un serveur sur une même machine, via ce qu'on appelle l'interface de loopback, représentée par convention par l'adresse IP "127.0.0.1" ou par "localhost".

Les sockets permettent, entre autres, de construire des applications réparties selon le modèle client/serveur.

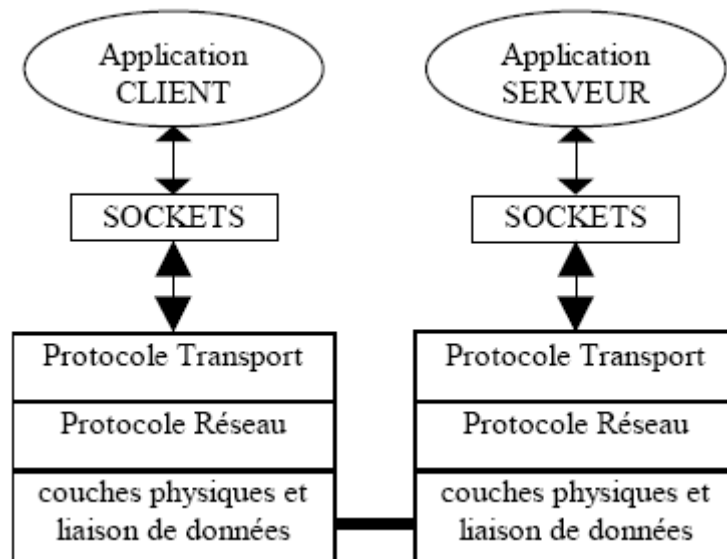


Figure 1 : Modèle client/serveur et sockets

Une socket est une notion très répandue en programmation réseau, en particulier sous UNIX. Il s'agit en fait d'un point de communication entre un processus et un réseau.

Un processus client et un processus serveur, lorsqu'ils communiquent, ouvrent chacun une socket. C'est le système d'exploitation qui alloue ces sockets sur demande d'une application. Les sockets sont ce qu'on appelle une API ("Application Programming Interface") c'est à dire une interface entre les programmes d'applications et la couche transport, par exemple TCP ou UDP.

A chaque socket est associé un port de connexion. Les numéros de ports sont uniques sur un système donné.

2. Présentation des Socket

Une socket est un simple canal de communication à travers lequel 2 programmes peuvent communiquer sur un réseau. Une socket supporte une communication bi-directionnelle entre un client et un serveur selon un protocole donné.

Les sockets peuvent être utilisés avec un grand nombre de protocoles réseau mais sont utilisés le plus fréquemment pour accéder aux réseaux **TPC/IP**.

Dans cette famille de protocole, les principales caractéristiques des socket sont : les « **stream socket** » qui font appel au protocole TCP et les « **datagram socket** » qui font référence au protocole UDP.

Une socket utilisant la famille TCP/IP est identifiée uniquement par une **adresse IP**, le type de protocole **TCP** ou **UDP** et un **numéro de port**.

Quand un processus ouvre un fichier (open), le système place un pointeur sur les structures internes de données correspondantes dans la **table des descripteurs** ouverts de ce processus et renvoie l'indice utilisé dans cette table. Par la suite, l'utilisateur manipule ce fichier uniquement par l'intermédiaire de l'indice, aussi nommé descripteur de fichier.

Comme pour un fichier, chaque socket active est identifiée par un entier appelé descripteur de socket. Unix place ce descripteur dans la même table que les descripteurs de fichiers, ainsi une application ne peut pas avoir un descripteur de fichier et un descripteur de socket identiques.

Remarques

Les entrées/sorties sur réseau mettent en jeu plus de mécanismes que les entrées/sorties sur un système de fichiers. En effet, il faut considérer les points suivants :

- Dans une relation du type client/serveur, les relations ne sont pas symétriques. Démarrer une telle relation suppose que le programme sait quel rôle il doit jouer.
- Une connexion réseau peut être du type connecté ou non. Dans le premier cas, une fois la connexion établie le processus émetteur discute uniquement avec le processus destinataire. Dans le cas d'un mode non connecté, un même processus peut envoyer plusieurs datagrammes à plusieurs autres processus sur des machines différentes.
- Une connexion est définie par un quintuplet qui est beaucoup plus compliqué qu'un simple nom de fichier.

En conclusion, on peut dire que le terme socket désigne, d'une part un ensemble de primitives, on parle des sockets de Berkeley, et d'autre part l'extrémité d'un canal de communication (point de communication) par lequel un processus peut émettre ou recevoir des données. Ce point de communication est représenté par une variable entière, similaire à un descripteur de fichier.

2.1. Principes de base

Les sockets permettent l'échange de messages entre 2 processus, situés sur des machines différentes :

- 1- Chaque machine crée une socket,
- 2- chaque socket sera associée à un port de sa machine hôte,
- 3- les deux sockets seront explicitement connectées si on utilise un protocole en mode connecté,
- 4- chaque machine lit et/ou écrit dans sa socket,
- 5- les données vont d'une socket à une autre à travers le réseau,
- 6- une fois terminé chaque machine ferme sa socket.

2.2. Etude des primitives

Création d'une socket : socket()

La création d'une socket se fait par l'appel système **socket**.

```
#include <sys/types.h> /* Pour toutes les primitives */
#include <sys/socket.h> /* de ce chapitre il faut */
#include <netinet/in.h> /* inclure ces fichiers. */
int socket(int PF, int TYPE, int PROTOCOL);
```

PF : Spécifie la famille de protocole ("Protocol Family") à utiliser avec la socket. Sur tout système qui permet l'usage des sockets, on trouve les familles :

```
PF_INET : Pour les sockets IPv4
PF_INET6 : Pour les sockets IPv6
PF_CCITT : Pour l'interface avec X25
PF_LOCAL : Pour rester en mode local (pipe). . .
PF_UNIX : Idem AF LOCAL
PF_ROUTE : Accès à la table de routage
```

Le préfixe PF est la contraction de "Protocol Family". On peut également utiliser le préfixe **AF**, pour "Address Family". Les deux nommages sont possibles ; l'équivalence est définie dans le fichier d'entête socket.h.

TYPE : Cet argument spécifie le type de communication désiré. En fait avec la famille PF_INET, le type permet de faire le choix entre un mode connecté, un mode non connecté ou une intervention directe dans la couche IP :

```
SOCK_STREAM : Mode connecté, fiable et séquencée, composée de flux de bytes.
SOCK_DGRAM : Mode non connecté, non fiable, composée de datagrammes de taille
              maximum fixée (les messages sont en général de petite taille).
SOCK_RAW : Dialogue direct avec la couche IP
```

PROTOCOL : Ce troisième argument permet de spécifier le protocole à utiliser.

Il peut être du type UDP ou TCP sous Unix.

```
IPPROTO_TCP : TCP
IPPROTO_UDP : UDP
IPPROTO_RAW, IPPROTO_ICMP : uniquement avec SOCK RAW
```

PROTOCOL est typiquement mis à zéro car l'association de la famille de protocole et du type de communication définit explicitement le protocole de transport :

```
PF_INET + SOCK_STREAM ==> TCP = IPPROTO_TCP
PF_INET + SOCK_DGRAM ==> UDP = IPPROTO_UDP
```

La primitive socket retourne un entier qui est le descripteur de la socket nouvellement créée par cet appel (et sinon -1 en cas d'erreur). Par rapport à la connexion future cette primitive ne fait que donner le premier élément du quintuplet :

{ **protocole**, port local, adresse locale, port éloigné, adresse éloignée }

En cas de succès, cette primitive retourne le descripteur de la socket.

Sockets-C les plus courantes :

- Internet - TCP : **socket(AF_INET,SOCK_STREAM,0) //0: choix auto du protocole**
- Internet - UDP : **socket(AF_INET,SOCK_DGRAM,0) //0: choix auto du protocole**

Remarque importante :

Comme pour les entrées/sorties sur fichiers, un appel système **fork** duplique la table des descripteurs de fichiers ouverts du processus père dans le processus fils. Ainsi les descripteurs de sockets sont également transmis.

Le bon usage du descripteur de socket partagé entre les deux processus incombe donc à la responsabilité du programmeur.

2.2.1. Association d'une socket à un port : bind()

Il faut remarquer qu'une socket est créée sans adresse d'émetteur ni celle du destinataire.

Pour les protocoles de l'Internet cela signifie que la création d'une socket est indépendante d'un numéro de port.

Dans beaucoup d'applications, surtout des clients, on n'a pas besoin de s'inquiéter du numéro de port, le protocole sous-jacent s'occupe de choisir un numéro de port pour l'application.

Cependant un serveur qui fonctionne suivant un numéro de port "bien connu" doit être capable de spécifier un numéro de port bien précis.

Une fois que la socket est créée, la primitive bind permet d'effectuer ce travail, c'est à dire d'associer une adresse IP et un numéro de port à une socket :

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

socket Est le descripteur de la socket, renvoyé par socket.

myaddr Est une structure qui spécifie l'adresse locale avec laquelle bind doit travailler.

addrlen Est un entier qui donne la longueur de l'adresse, en octets.

La description de l'adresse change d'une famille de protocole à une autre, c'est pourquoi les concepteurs ont choisi de passer en argument une structure plutôt qu'un entier. Cette structure générique, est nommée généralement sockaddr.

Elle commence généralement par deux octets qui rappellent la famille de protocole, suivis de 14 octets qui définissent l'adresse en elle-même :

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

Pour la famille PF_INET cette structure se nomme sockaddr_in, elle est définie de la façon suivante :

```
struct sockaddr_in {  
    short sin_family ; /* AF_INET */  
    u_short sin_port ; /* Port */  
    struct in_addr sin_addr ; /* Adresse IP */  
    char sin_zero[8] ;  
};
```

```
struct in_addr {  
    unsigned long s_addr ; /* 32b Internet */  
};
```

```
} ;
```

La primitive `bind` ne permet pas toutes les associations de numéros de port, par exemple si un numéro de port est déjà utilisé par un autre processus, ou si l'adresse Internet est invalide.

Il y a trois utilisations de la primitive :

1. En général les serveurs ont des numéros de port bien connus du système (cf `/etc/services`). `Bind` dit au système "c'est mon adresse, tout message reçu à cette adresse doit m'être renvoyé". Que cela soit en mode connecté ou non, les serveurs ont besoin de le dire avant de pouvoir accepter les connexions.
2. Un client peut préciser sa propre adresse, en mode connecté ou non.
3. Un client en mode non connecté a besoin que le système lui assigne une adresse particulière, ce qui autorise l'usage des primitives `read` et `write` traditionnellement dédiées au mode connecté.

`Bind` retourne 0 si tout va bien, -1 si une erreur est intervenue. Dans ce cas la variable globale `errno` est positionnée à la bonne valeur.

Cet appel système complète l'adresse locale et le numéro de port du quintuplet qui qualifie une connexion. Avec `bind+socket` on a la moitié d'une connexion, à savoir un protocole, un numéro de port et une adresse IP :

{protocole, port local, adresse locale, port éloigné, adresse éloignée}

2.2.2. Ecoute des demandes de connexion : `listen()`

La fonction `listen()` permet de mettre un socket en attente de connexion.

NB : La fonction `listen()` ne s'utilise qu'en mode connecté (donc avec le protocole TCP)

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

-**socket** représente le socket précédemment ouvert

-**backlog** représente le nombre maximal de connexions pouvant être mises en attente

La fonction `listen()` retourne la valeur `SOCKET_ERROR` en cas de problème, sinon elle retourne 0.

Exemple : Utilisation de la fonction `listen()` :

```
if (listen(socket,10) == SOCKET_ERROR)
{
// traitement de l'erreur
}
```

2.2.3. Acceptation de la connexion : `accept()`

La fonction `accept()` permet la connexion en acceptant un appel :

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

-socket représente le socket précédemment ouvert (le socket local)

-addr représente un tampon destiné à stocker l'adresse de l'appelant

-addrlen représente la taille de l'adresse de l'appelant

La fonction *accept()* retourne un identificateur du socket de réponse. Si une erreur intervient la fonction *accept()* retourne la valeur *INVALID_SOCKET*.

Exemple : Utilisation de la fonction *accept()* :

```
Socketaddr_in Appelant;
/* structure destinée à recueillir les renseignements sur l'appelant
Appelantlen = sizeof(from);
accept(socket_local, (struct sockaddr*)&Appelant, &Appelantlen);
```

2.2.4. Demande de connexion : connect()

La fonction *connect()* permet d'établir une connexion avec un serveur :

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

-socket représente le socket précédemment ouvert (le socket à utiliser)

-addr représente l'adresse de l'hôte à contacter. Pour établir une connexion, le client ne nécessite pas de faire un *bind()*

-addrlen représente la taille de l'adresse de l'hôte à contacter

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

Exemple : Utilisation de la fonction *connect()*, qui connecte le socket "s" du client sur le port *port* de l'hôte portant le nom *serveur* :

```
toinfo = gethostbyname(serveur);
toaddr = (u_long *)toinfo.h_addr_list[0];
/* Protocole internet */
to.sin_family = AF_INET;
/* Toutes les adresses IP de la station */
to.sin_addr.s_addr = toaddr;
/* port d'écoute par défaut au-dessus des ports réservés */
to.sin_port = htons(port);
if (connect(socket, (struct sockaddr*)to, sizeof(to)) == -1)
{
// Traitement de l'erreur;
}
```

Le quintuplet est maintenant complet :

{protocole, port local, adresse locale, port éloigné, adresse éloignée}

2.2.5. Réception des données : `recv()`

La fonction `recv()` permet de lire dans un socket en mode connecté (TCP) :

```
int recv(int socket, char * buffer, int len, int flags)
```

-**socket** représente le socket précédemment ouvert

-**buffer** représente un tampon qui recevra les octets en provenance du client

-**len** indique le nombre d'octets à lire

-**flags** correspond au type de lecture à adopter :

1. Le flag `MSG_PEEK` indiquera que les données lues ne sont pas retirées de la queue de réception
2. Le flag `MSG_OOB` indiquera que les données urgentes (*Out Of Band*) doivent être lues
3. Le flag `0` indique une lecture normale

La fonction `recv()` renvoie le nombre d'octets lus. De plus cette fonction bloque le processus jusqu'à ce qu'elle reçoive des données.

Exemple : Utilisation de la fonction `recv()` :

```
retour = recv(socket, Buffer, sizeof(Buffer), 0 );  
if (retour == SOCKET_ERROR) { // traitement de l'erreur  
}
```

2.2.6. Envoi des données : `send()`

La fonction `send()` permet d'écrire dans une socket (envoyer des données) en mode connecté (TCP) :

```
int send(int socket, char * buffer, int len, int flags)
```

-**socket** représente le socket précédemment ouvert

-**buffer** représente un tampon contenant les octets à envoyer au client

-**len** indique le nombre d'octets à envoyer

-**flags** correspond au type d'envoi à adopter :

1. le flag `MSG_DONTROUTE` indiquera que les données ne routeront pas
2. le flag `MSG_OOB` indiquera que les données urgentes (*Out Of Band*) doivent être envoyées
3. le flag `0` indique un envoi normal

La fonction `send()` renvoie le nombre d'octets effectivement envoyés.

Exemple : Utilisation de la fonction `send()` :

```
retour = send(socket, Buffer, sizeof(Buffer), 0 );  
if (retour == SOCKET_ERROR)  
{  
    // traitement de l'erreur  
}
```

2.2.7. Fermeture d'une socket : `close()` et `shutdown()`

La fonction `close()` permet la fermeture d'un socket en permettant au système d'envoyer les données restantes (pour TCP) :

```
int close(int socket)
```

La fonction `shutdown()` permet la fermeture d'un socket dans un des deux sens (pour une connexion full-duplex) :

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

Si `how` est égal à 0, le socket est fermé en réception

Si `how` est égal à 1, le socket est fermé en émission

Si `how` est égal à 2, le socket est fermé dans les deux sens

`close()` comme `shutdown()` retournent -1 en cas d'erreur, 0 si la fermeture se déroule bien.

Si un processus ayant ouvert des sockets vient à s'interrompre pour une raison quelconque, en interne la socket est fermée et si plus aucun processus n'a de descripteur ouvert sur elle, le noyau la supprime.

3. Application

3.1. Les Sockets : Mode connecté

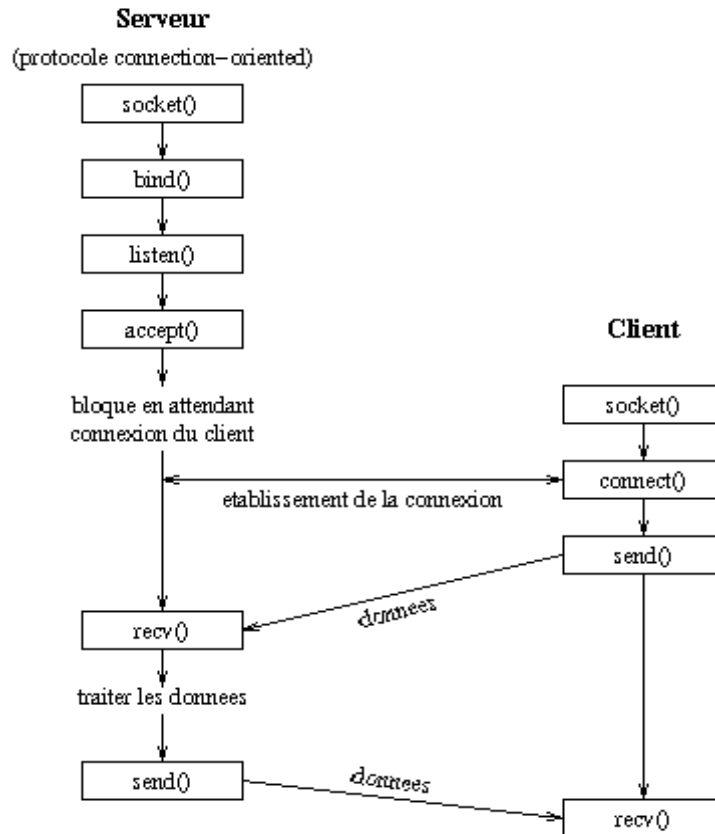
1.3.1. Etapes de connexion client-serveur en TCP

- 1- Le serveur et le client **ouvrent** chacun une « socket »
- 2- le serveur la **nomme** (il l'attache à un de ses ports (un port précis)) (le client n'est pas obligé de la nommer (elle sera attachée automatiquement à un port lors de la connexion))
- 3- le serveur **écoute** sa socket nommée
- 4- le serveur **attend** des demandes de connexion
- 5- le client **connecte** sa socket au serveur et à un de ses ports (précis)
- 6- le serveur **détecte** la demande de connexion (une nouvelle socket est ouverte automatiquement)
 - a. le nouveau processus continue le dialogue sur la nouvelle socket
 - b. le serveur attend en parallèle de nouvelles demandes de connexions
- 7- le serveur crée un processus pour **dialoguer** avec le client
- 8- finalement toutes les sockets doivent être **fermées**

Détection de fin de connexion

- Protocole de maintien de connexion
- Si le processus d'une extrémité meurt l'autre en est averti
- Si le client meurt la nouvelle socket automatiquement ouverte sur le serveur est détruite.

La figure suivante résume les étapes mentionnées ci-dessus.



Les sockets de type `SOCK_STREAM` ont la capacité de mettre les requêtes de connexion en files d'attente, ce qui ressemble au téléphone qui sonne en attendant que l'on réponde. Si c'est occupé, la connexion va attendre que la ligne soit libérée. La fonction `listen()` est utilisée pour donner le nombre maximum de requêtes en attente (généralement jusqu'à 5 maximum) avant de refuser les connexions.

1.3.2. Le programme serveur

```

#include <stdio.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>

void error(char *msg)
{
    perror(msg);
    exit(1);
}

int main(int argc, char *argv[])
{
    int sockfd, newsockfd, portno, clien;
    char buffer[256];
    struct sockaddr_in serv_addr, cli_addr;
    int n;
    if (argc < 2) {
        fprintf(stderr, "Erreur, Spécifiez le Port\n");
    }
  
```

```

    exit(1);
}
sockfd = socket(AF_INET, SOCK_STREAM, 0);
if (sockfd < 0)
    error("Erreur d'ouverture du socket");
bzero((char *) &serv_addr, sizeof(serv_addr));
portno = atoi(argv[1]);
serv_addr.sin_family = AF_INET;
serv_addr.sin_addr.s_addr = INADDR_ANY;
serv_addr.sin_port = htons(portno);
if (bind(sockfd, (struct sockaddr *) &serv_addr,
    sizeof(serv_addr)) < 0)
    error("Erreur d'association au port");
listen(sockfd,5);
clilen = sizeof(cli_addr);
newsockfd = accept(sockfd,
    (struct sockaddr *) &cli_addr,
    &clilen);
if (newsockfd < 0)
    error("Erreur d'acceptation");
bzero(buffer,256);
n = read(newsockfd,buffer,255);
if (n < 0) error("Erreur de lecture du socket");
printf("Voici le message que j'ai reçu : %s\n",buffer);
n = write(newsockfd,"J'ai reçu votre message\n",24);
if (n < 0) error("Erreur d'écriture dans le socket");
return 0;
}

```

1.3.3. Le programme client

```

#include <stdio.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>

void error(char *msg)
{
    perror(msg);
    exit(0);
}

int main(int argc, char *argv[])
{
    int sockfd, portno, n;
    struct sockaddr_in serv_addr;
    struct hostent *server;

    char buffer[256];
    if (argc < 3) {
        fprintf(stderr,"Spécifiez les arguments SVP %s nom_hôte No_Port\n", argv[0]);
        exit(0);
    }
}

```

```

}
portno = atoi(argv[2]);
sockfd = socket(AF_INET, SOCK_STREAM, 0);
if (sockfd < 0)
    error("Erreur dans l'ouverture de socket");
server = gethostbyname(argv[1]);
if (server == NULL) {
    fprintf(stderr, "Erreur, Nom d'hôte incorrect\n");
    exit(0);
}
bzero((char *) &serv_addr, sizeof(serv_addr));
serv_addr.sin_family = AF_INET;
bcopy((char *)server->h_addr,
      (char *)&serv_addr.sin_addr.s_addr,
      server->h_length);
serv_addr.sin_port = htons(portno);
if (connect(sockfd,&serv_addr,sizeof(serv_addr)) < 0)
    error("Erreur de connexion");
printf("Entrez le message à envoyer : ");
bzero(buffer,256);
fgets(buffer,255,stdin);
n = write(sockfd,buffer,strlen(buffer));
if (n < 0)
    error("Erreur d'écriture dans le socket");
bzero(buffer,256);
n = read(sockfd,buffer,255);
if (n < 0)
    error("Erreur de lecture du socket");
printf("%s\n",buffer);
return 0;
}

```

1.3.4. Compilation et exécution

Pour compiler l'un de ces programmes,

- On se situe au répertoire où ils se trouvent, par exemple

```
cd /usr/socket/tcp
```

- La commande de compilation est

```
cc server.c -o server
```

l'option `-o` permet de nommer l'output de la compilation (fichier exécutable `server`)

- L'exécution du serveur :

```
./server no_port
```

Dans un autre terminal, compiler le client et lancer-le (n'oublier pas les paramètres nécessaires).

Tester l'envoi et la réception de message.

3.2. Les Sockets : Mode non connecté

2.3.1. Etapes d'une connexion client-serveur en UDP

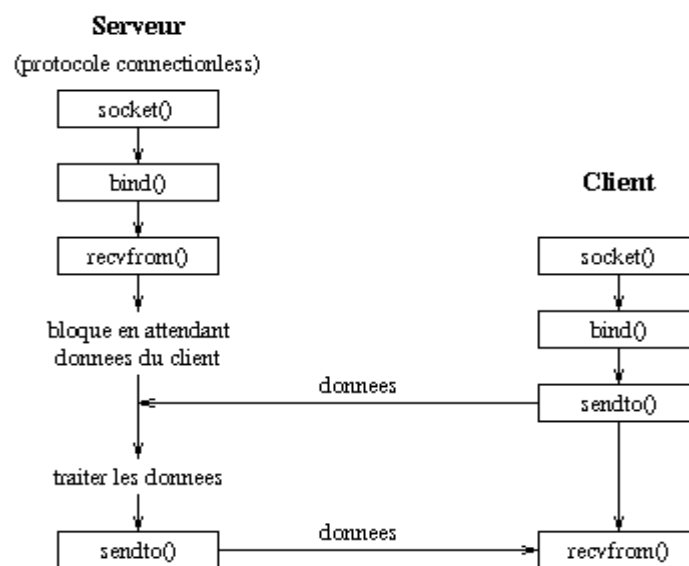
1- le serveur et le client ouvrent chacun une « socket »

- 2- le serveur la nomme (il l'attache à un de ses ports (un port précis)) (le client ne nomme pas sa socket (elle sera attachée automatiquement à un port lors de l'émission))
- 3- le client et le serveur dialogue : **sendto(...)** et **recvfrom(...)**
 - le client n'établit pas de connexion avec le serveur mais émet un datagramme (sendto) vers le serveur
 - Le serveur n'accepte pas de connexion, mais attend un datagramme d'un client par recvfrom qui transmet le datagramme à l'application ainsi que l'adresse client
- 4- finalement toutes les sockets doivent être refermées

Les deux extrémités n'établissent pas une connexion

- elles ne mettent pas en oeuvre un protocole de maintien de connexion
- si le processus d'une extrémité meurt l'autre n'en sait rien !

La figure suivante résume l'établissement de sockets en mode non connecté.



2.3.2. Le programme serveur

```

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

#define MYPORT 4950 // the port users will be connecting to

#define MAXBUFLLEN 100

int main(void)
{
    int sockfd;
  
```

```

struct sockaddr_in my_addr; // my address information
struct sockaddr_in their_addr; // connector's address information
int addr_len, numbytes;
char buf[MAXBUFLen];

if ((sockfd = socket(AF_INET, SOCK_DGRAM, 0)) == -1) {
    perror("socket");
    exit(1);
}

my_addr.sin_family = AF_INET; // host byte order
my_addr.sin_port = htons(MYPORT); // short, network byte order
my_addr.sin_addr.s_addr = INADDR_ANY; // automatically fill with my IP
memset(&(my_addr.sin_zero), '\0', 8); // zero the rest of the struct

if (bind(sockfd, (struct sockaddr *)&my_addr,
        sizeof(struct sockaddr)) == -1) {
    perror("bind");
    exit(1);
}

addr_len = sizeof(struct sockaddr);
if ((numbytes=recvfrom(sockfd, buf, MAXBUFLen-1, 0,
        (struct sockaddr *)&their_addr, &addr_len)) == -1) {
    perror("recvfrom");
    exit(1);
}

printf("got packet from %s\n",inet_ntoa(their_addr.sin_addr));
printf("packet is %d bytes long\n",numbytes);
buf[numbytes] = '\0';
printf("packet contains \"%s\"\n",buf);

close(sockfd);

return 0;
}

```

2.3.3. Le programme client

```

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

```

```
#include <netdb.h>

#define MYPORT 4950 // the port users will be connecting to

int main(int argc, char *argv[])
{
    int sockfd;
    struct sockaddr_in their_addr; // connector's address information
    struct hostent *he;
    int numbytes;

    if (argc != 3) {
        fprintf(stderr, "usage: talker hostname message\n");
        exit(1);
    }

    if ((he=gethostbyname(argv[1])) == NULL) { // get the host info
        perror("gethostbyname");
        exit(1);
    }

    if ((sockfd = socket(AF_INET, SOCK_DGRAM, 0)) == -1) {
        perror("socket");
        exit(1);
    }

    their_addr.sin_family = AF_INET; // host byte order
    their_addr.sin_port = htons(MYPORT); // short, network byte order
    their_addr.sin_addr = *((struct in_addr *)he->h_addr);
    memset(&(their_addr.sin_zero), '\0', 8); // zero the rest of the struct

    if ((numbytes=sendto(sockfd, argv[2], strlen(argv[2]), 0,
                        (struct sockaddr *)&their_addr, sizeof(struct sockaddr))) == -1) {
        perror("sendto");
        exit(1);
    }

    printf("sent %d bytes to %s\n", numbytes, inet_ntoa(their_addr.sin_addr));

    close(sockfd);

    return 0;
}
```

3.3. Interactivité entre le client et le serveur

Retournons aux programmes client et serveur en mode connecté.

On propose de rendre la communication entre le client et le serveur plus interactif (question/réponse).

Le serveur envoie la chaîne "Veuillez entrer votre nom: " au client à travers un tampon de chaîne. Le client imprime la chaîne de caractères, lit le nom comme une chaîne en provenance du clavier et retourne la chaîne à travers le tampon. Le serveur demande alors l'année de naissance. Quand le client la récupère sous forme de chaîne de caractères, le serveur la convertit en nombre et la soustrait de 2005. Il renvoie alors l'âge approximatif résultant au client. C'est fini maintenant mais compte tenu du fait que le client attend une entrée clavier avant d'effectuer le retour de fonction, le serveur demande qu'un "q" soit tapé. Un codage plus sophistiqué pourrait éliminer cet inconvénient. Ce modèle client/serveur illustre le passage de chaîne de caractères entre le serveur et le client, les requêtes et les réponses et l'utilisation d'arithmétique.

Exemple d'exécution :

```
Veuillez entrer votre nom: ali
Bonjour, ali
Veuillez entrer votre année de naissance: 1983
Votre âge approximatif est 22.
Tapez q pour quitter: q
```