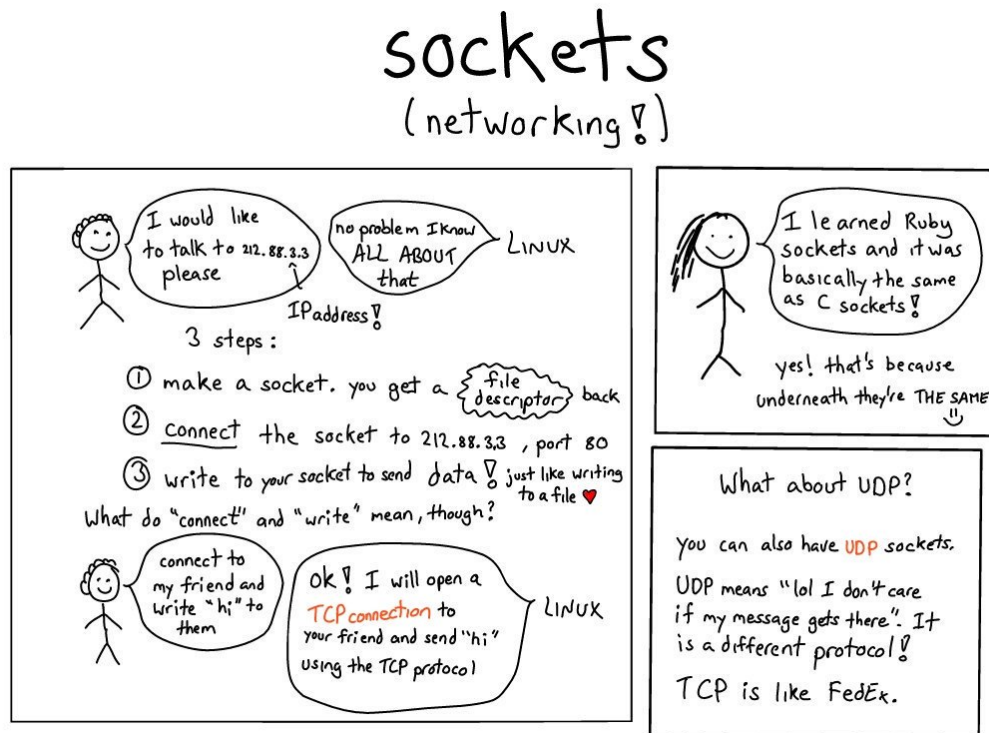


Auteur : Pascal Fougeray

Source : <https://twitter.com/b0rk>

## 1 Introduction

Lorsque l'on désire réaliser des applications client-serveur, il est nécessaire d'utiliser les Socket<sup>1</sup> et dans votre domaine de prédilection, l'informatique, vous devez avoir quelques notions sur la programmation d'une application client-serveur.

Dans ce chapitre, nous allons juste voir les grandes lignes et appréhender le développement d'une simple application.

En effet, il existe une multitude de solutions selon que l'on désire créer un serveur **itératif** ou un serveur **parallèle** en utilisant une communication en **mode connecté** ou **non connecté**.

Ce cours est loin d'être exhaustif et je vous invite à lire différents ouvrages à ce sujet, notamment celui de Jean-Marie Rifflet : **La communication sous UNIX, Applications réparties**, dans lequel j'ai pioché quelques exemples.

Il existe de nombreux sites qui l'expliquent surement même mieux que moi.

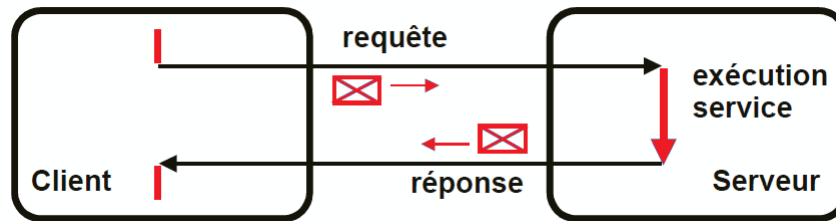
Vous pourrez faire le TP en python ou un autre langage plus moderne, java par exemple ©si vous le voulez.

## 2 Rappels

Quelques rappels sur la notion client/serveur et le modèle en couche ?

1. La traduction littérale est : prise ou point de communication

## 2.1 Un Appel Synchrone Requête-Réponse !



## 2.2 Serveur, Client et Service

Dans une application répartie, les processus utilisés sont divisés en 2 catégories : un ou plusieurs processus serveurs et des processus clients. Ces différents processus échangent des messages : le client adresse à un serveur une requête par l'intermédiaire d'un message et le serveur transmet au client un message de réponse, après avoir satisfait la requête du client. On distingue ainsi :

- le **service** : C'est une tâche particulière dont on peut demander la réalisation ;
- Sur Linux, on connaît les **services** avec le **N° de port** associé, en lisant le fichier **/etc/services**.

Dont voici un extrait :

```

msp      18/tcp      # message send protocol
msp      18/udp
chargen  19/tcp      ttytst source
chargen  19/udp      ttytst source
ftp-data 20/tcp
ftp       21/tcp
fsp       21/udp      fspd
ssh       22/tcp      # SSH Remote Login Protocol
telnet    23/tcp
smtp      25/tcp      mail
time      37/tcp      timserver
time      37/udp      timserver
rlp       39/udp      resource
nameserver 42/tcp      name
whois     43/tcp      nickname
tacacs    49/tcp      # Login Host Protocol (TACACS)
  
```

- le **serveur** : C'est une machine sur laquelle un service est réalisé ou un **processus (thread)** qui rend ce service ;
- le **client** : C'est une machine faisant appel à une machine serveur ou un **processus (thread)** sollicitant un processus serveur.

**Question** : Quel est le seul protocole où il n'y a qu'un seul client et plein de serveurs ?

Réponse : **SNMP : Simple Network Management Protocol**

C'est un protocole utilisé par les différents éléments actifs d'un réseau (Ordinateurs, switches, routeurs etc..) sont interrogés régulièrement par un client (le manager) pour connaître leur état à chacun.

## 2.3 Les ports

Il y a 65536 ports différents, le codage sur 16 bits donne  $2^{16}$  possibilités.

Tous les ports n'ont pas les mêmes caractéristiques.

À chaque N° de port peut correspondre un service différent, mais il y a beaucoup plus de N° de ports que de services.

- **Le port 0** : Il n'est pas utilisable pour une application, c'est un "**joker**" indiquant au système que c'est à lui de compléter automatiquement le numéro entre **49152** et **65535**.
- De 1 à 1023 : Pour utiliser cette zone il faut avoir les droits du root, sinon, à l'exécution le **bind** retourne une erreur.  
Les serveurs "classiques" (**ftp**, **smtp**, **telnet**, **ssh**...) utilisent cette plage de N° de ports.  
si si vous vous souvenez le TP où l'on parlait le SMTP avec telnet : telnet @Ip N°port
- De 1024 à 49151 est la zone des services enregistrés par l'IANA<sup>2</sup> et qui fonctionnent avec des droits ordinaires.

2. Internet Assigned Numbers Authority : [www.iana.org](http://www.iana.org)



- Port de **49152** à 65535 est la zone d'attribution automatique des ports, pour la partie cliente des connexions (si le protocole n'impose pas une valeur particulière) et pour les tests de serveurs locaux.

## 2.4 Modes connecté et non-connecté

Dans le domaine des réseaux il existe 2 modes principaux de communication :

- Dans le mode **connecté**, la communication est réalisée sur un canal dont l'établissement a été réalisé, il y a négociation entre les 2 entités entre lesquelles la communication est établie. La rupture de la connexion peut également être négociée. Cette communication est fiable.

On parle aussi de **transmission par flux, Stream : Téléphone , TCP**.

L'avantage de l'utilisation d'un protocole comme TCP/IP est la fiabilité !

La couche **transport** effectue elle même

- le "**checksum**",
  - la ré-émission de morceaux de messages perdus,
  - l'élimination des morceaux dupliqués,
  - l'adaptation du débit.
- Dans le mode **non-connecté** chaque message est transmis individuellement sans relation particulière avec aucun message antérieur.

Ce mode de communication est **non fiable** et ne donne aucune garantie d'arrivée, et de respect de l'ordre des messages n'est assurée.

On parle aussi de **transmission par paquets, Data-grams : Courrier, UDP**.

Un protocole comme UDP/IP n'effectue pas ces vérifications qui doivent alors être faites par le protocole de communication de **niveau applicatif**.

Pour cette raison, la programmation de **clients ou serveurs en mode non connecté est plus complexe** qu'en mode connecté.

**Cependant, en réseau local (même réseau !!!) où le transport est fiable, il est avantageux d'utiliser UDP car ce mode de communication demande moins d'opérations que TCP.**

Ce qui est le cas quand on n'a pas de nombreux sauts de routage (couche 3) à réaliser.

- Le 3 ème mode dont on parle peu et pas plus difficile à programmer (quoi que...) est le mode **RAW** un mode qui n'est ni TCP, ni UDP... mais qu'est-ce qu'il y a sur la couche 4 du modèle OSI, vous savez la couche... Transport... ?

Et bien c'est le **ping** !

Je ne vais pas en parler d'avantage car le temps est compté et surtout qu'il faut être root pour faire les TP...

## 3 Les sockets

Mécanisme de communication permettant d'utiliser l'interface de transport TCP-UDP donc la couche 4 du modèle OSI et s'appuyant sur IP.

Ce mécanisme fut introduit dans Unix dans les années 80, c'est un standard aujourd'hui !

### 3.1 Définition

Un (ou une...) *socket* est un **point de contact** dans une communication.

Elle permet à 2 entités, généralement un client et un serveur de se retrouver en relation et de pouvoir communiquer, un peu comme les tubes que nous avons étudiés précédemment mais avec une puissance plus conséquente.

Pour faire beaucoup plus simple, une *socket* est soit un téléphone, soit une boîte aux lettres...

**ATTENTION :**

**Je ne veux pas entendre qu'un socket c'est une IP et un N° de port, ce n'est pas que ça!!! et la couche 4 vous en faites quoi ?**

Et puis des sockets il y en a plein sur une machine!!!

La preuve : la commande **find -type s | wc -l** sur ma VM me renvoie 44 donc 44 sockets

Voici le résultat, ceux qui sont soulignés doivent vous parler...



```

root@debian95-Rx-Sys-Fougeray:~# find -type s
./tmp/.ICE-unix/1101
./tmp/ssh-wLgg2UM1XGrT/agent.1055
./tmp/.X11-unix/X1
./tmp/.X11-unix/X0
./var/lib/courier/sqwebmail.sock
./root/L3/TD_TP/socket/chapitre_14/local/socket_serveur
./root/L3/TD_TP/socket/local/socket_serveur
./run/openvswitch/ovs-vswitchd.723.ctl
./run/openvswitch/ovsdb-server.687.ctl
./run/openvswitch/db.sock
./run/docker/libnetwork/b917ad3e95e4fbc61658e58db48f52a7f4ec43112c39cd00c1eeaf7aeld2ela.sock
./run/docker/metrics.sock
./run/docker/containerd/docker-containerd.sock
./run/docker/containerd/docker-containerd-debug.sock
./run/mysqld/mysqld.sock
./run/uuid/request
./run/docker.sock
./run/avahi-daemon/socket
./run/dbus/system_bus_socket
./run/cups/cups.sock
./run/user/0/bus
./run/user/0/gnupg/S.gpg-agent.ssh

```

```

./run/user/0/gnupg/S.gpg-agent.extra
./run/user/0/gnupg/S.gpg-agent.browser
./run/user/0/gnupg/S.gpg-agent
./run/user/0/systemd/private
./run/user/0/systemd/notify
./run/user/114/gnupg/S.gpg-agent
./run/user/114/gnupg/S.gpg-agent.ssh
./run/user/114/gnupg/S.gpg-agent.extra
./run/user/114/gnupg/S.gpg-agent.browser
./run/user/114/bus
./run/user/114/systemd/private
./run/user/114/systemd/notify
./run/systemd/fsck.progress
./run/systemd/journal/syslog
./run/systemd/journal/socket
./run/systemd/journal/stdout
./run/systemd/journal/dev-log
./run/systemd/private
./run/systemd/cgroups-agent
./run/systemd/notify
./run/systemd/inaccessible.sock
./run/udev/control
root@debian95-Rx-Sys-Fougeray:~#

```

Voyons ce que sont celles qui sont dans `/root/L3/TD_TP/socket` juste par curiosité !

Un `ls -l /root/L3/TD_TP/socket/local/socket_serveur`  
et un `file /root/L3/TD_TP/socket/local/socket_serveur`  
renvoient

```

./run/udev/control
root@debian95-Rx-Sys-Fougeray:~# find -type s | wc -l
44
root@debian95-Rx-Sys-Fougeray:~# ls -l /root/L3/TD_TP/socket/local/socket_serveur
-rwxr-xr-x 1 root root 0 nov.  8 00:17 /root/L3/TD_TP/socket/local/socket_serveur
root@debian95-Rx-Sys-Fougeray:~# file /root/L3/TD_TP/socket/local/socket_serveur
/root/L3/TD_TP/socket/local/socket_serveur: socket
root@debian95-Rx-Sys-Fougeray:~#

```

**S comme Socket**

**UN ou UNE SOCKET c'est donc un fichier sous Linux mais pas que ça!!!!**

## 3.2 Principe de fonctionnement

Il y a 3 phases !

1. le serveur crée une "socket serveur" associée à un port et se met en attente, il écoute (*Listen*)
2. le client se connecte à la socket serveur ;  
deux sockets sont alors créés :
  - (a) une "**socket client**", côté client,
  - et
  - (a) une "**socket service client**" côté serveur.

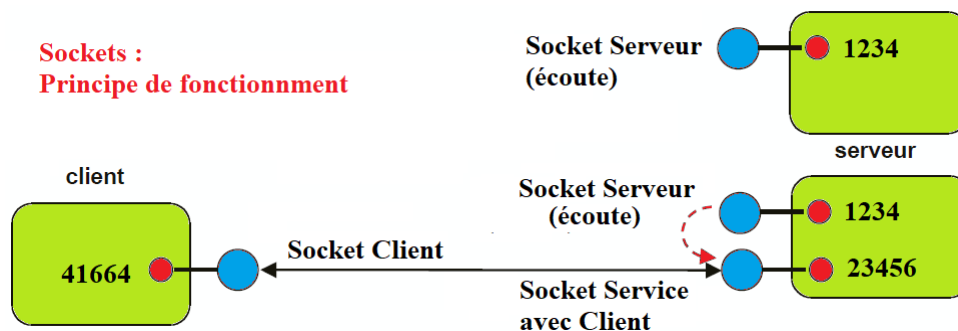
**Ces sockets sont connectées entre elles**

3. Le client et le serveur communiquent par les sockets.

L'interface est celle des fichiers (**read**, **write**). Car tout est fichiers!!!

La socket serveur peut accepter de nouvelles connexions

En 1 dessin cela donne : **1234** etc... sont les **N° de ports**, **remarquez** que dans notre cas le client a très soif ☺



## 3.3 Deux réalisations possibles du client-serveur avec sockets

1. Mode **connecté** (TCP)
  - Ouverture d'une liaison, suite d'échanges, fermeture de la liaison



- Le serveur préserve son état entre deux requêtes
- Garanties de TCP : ordre, contrôle de flux, fiabilité
- Adapté aux échanges ayant une certaine durée (plusieurs messages)

## 2. Mode **non connecté** (UDP)

- Les requêtes successives sont **indépendantes**
- Pas de préservation de l'état entre les requêtes
- Le **client doit indiquer son adresse à chaque requête** (pas de liaison permanente)
- Pas de garanties particulières (UDP)
- Adapté aux échanges brefs (réponse en 1 message)
- Points communs
  - Le client a l'initiative de la communication ; le serveur doit être à l'écoute
  - Le client doit connaître la référence du serveur [adresse IP, n° de port]  
il peut la trouver dans un annuaire si le serveur l'y a enregistrée au préalable, ou la connaître par convention (**/etc/services**) : n°socket de port pré-affectés
  - Le serveur peut servir plusieurs clients (1 Processus/thread unique ou 1 Processus/thread par client)

## 3.4 Créer, attacher et supprimer une *Socket*

- La **création** d'une *socket* est réalisée par la primitive **socket** dont la valeur de retour est un descripteur, tout comme un descripteur de fichiers, sur lequel il est possible de réaliser des opérations de lecture et d'écriture puisque contrairement aux tubes une communication par *socket* est bidirectionnelle.
- L'**attachement** : Il existe une différence fondamentale entre une *socket* et un fichier, c'est que le nommage des *sockets* est une opération distincte de leur création. Bien sûr, ouverture et création ne peuvent être dissociées comme pour les tubes, mais il est ensuite possible d'**attacher une adresse** de son domaine à l'objet créé. Pour cela on utilise la primitive **bind**.  
Sans ce principe de nommage, le processus qui héritent du descripteur peuvent lire et/ou écrire sur la *socket*, mais personne ne peut envoyer de données. car contrairement à un tube, la *socket* n'est qu'un point<sup>3</sup> de communication qui n'est relié à rien et nécessite d'être désignée de l'extérieur pour pouvoir être contactée. On lui donne pour cela une adresse.
- La **suppression** : une *socket* est supprimée à la réalisation effective de la fermeture du dernier descripteur permettant d'y accéder. On utilise pour cela les primitives **close** ou **shutdown**.

Le code source dessous est celui d'une fonction permettant de créer et d'attacher une *socket*.  
Quelques explications...

- La ligne 11 donne le type de structure utilisée pour l'adresse (voir plus loin).
- La ligne 17 crée la *socket* en utilisant la primitive **socket**.
- La ligne 25 attache la *socket*.
- Les lignes 21 à 23 permettent de remplir la structure contenant l'adresse.
- La ligne 27 montre la fermeture du fichier descripteur
- La ligne 30 permet d'obtenir le nom de la *socket*.
- La ligne 31 donne en retour la valeur du descripteur de fichier avec lequel nous aurons accès à la *socket*.

---

3. Et pour faire une droite, il faut 2 points...

```

1 // Permet de créer une socket
2 // que l'on compilera avec
3 // gcc -c creer_socket.c -o creer_socket.o
4 #include <stdio.h>
5 #include <stdlib.h>
6 #include <sys/types.h>
7 #include <sys/socket.h>
8 #include <unistd.h>
9 #include <netinet/in.h>
10
11 static struct sockaddr_in adresse;
12
13 int creer_socket(int type, int *ptr_port, struct sockaddr_in *ptr_adresse){
14     int desc; // Descripteur de la socket
15     int longueur = sizeof(struct sockaddr_in); // Taille de l'adresse
16     // Création de la socket
17     if((desc=socket(AF_INET, type, 0)) == -1) {
18         perror("Création de socket impossible");
19         return -1;
20     }
21     /* Préparation de l'adresse d'attachement */
22     adresse.sin_family = AF_INET;
23     adresse.sin_addr.s_addr = htonl(INADDR_ANY);
24     adresse.sin_port= htons(*ptr_port); // n° de port au format réseau
25     /* Demande d'attachement de la socket */
26     if(bind(desc, &adresse, longueur) == -1) {
27         perror("Attachement de la socket impossible");
28         close(desc); // Donc suppression de la socket
29         return -1;
30     }
31     if(ptr_adresse != NULL)
32         getsockname(desc, ptr_adresse, &longueur);
33     return desc;

```

### 3.4.1 Le domaine d'une socket

Une socket est identifiée localement dans un processus par un **descripteur**, cette identification n'est valable que dans le contexte du processus!!!

Comme le montre le programme suivant !

```

1 // programme 3-sockets.c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
6 #include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>

int main() {
11     int sockfd_serveur_TCP, sockfd_serveur_UDP; // descripteurs pour les 3 sockets
    int sockfd_serveur_RAW;

    // On cree les sockets serveurs en écoute
    sockfd_serveur_TCP = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP);
16     sockfd_serveur_UDP = socket(AF_INET, SOCK_DGRAM, IPPROTO_UDP);
    sockfd_serveur_RAW = socket(AF_INET, SOCK_RAW, IPPROTO_RAW);

    printf("Valeur du descripteur de la socket TCP : %d \n", sockfd_serveur_TCP);
    printf("Valeur du descripteur de la socket UDP : %d \n", sockfd_serveur_UDP);
21     printf("Valeur du descripteur de la socket RAW : %d \n", sockfd_serveur_RAW);

    while(1) {} // bon et bien plus qu'à le tuer ^^

```





}

Dont le résultat renvoie :

```
root@debian95-Rx-Sys-Fougeray:~/L3/TD_TP/socket# gcc 3-sockets.c -o 3-sockets
root@debian95-Rx-Sys-Fougeray:~/L3/TD_TP/socket# ./3-sockets &
[3] 1705
root@debian95-Rx-Sys-Fougeray:~/L3/TD_TP/socket#
    Valeur du descripteur de la socket TCP : 3
    Valeur du descripteur de la socket UDP : 4
    Valeur du descripteur de la socket RAW : 5
```

■

**Question** : Pourquoi les valeurs 3, 4 et 5 et pas 0,1 et 2

**Réponse** : Voir le cours précédent sur les tubes

**Si on désire que d'autres processus (sans lien de parenté) transmettent des données à destination de la socket, il est nécessaire que ceux-ci disposent d'un moyen externe de nommer cette socket.**

**Le domaine d'une socket définit le format des adresses possibles, les sockets avec lesquelles elle pourra communiquer, ainsi qu'une famille de protocoles utilisables.**

Quelques domaines :

Nom symbolique	domaine	format adresses
AF_UNIX	local	sockaddr_un
AF_INET	Internet	sockaddr_in
AF_APPLETALK	Apple talk	???

**3.4.2 Le type d'une socket** : détermine la **sémantique** des communications qu'elle permet de réaliser.

- **SOCK\_RAW** : *socket* orientée vers l'échange de data-grammes à un niveau bas de protocole (IP par exemple).  
Elle est notamment utilisée par la commande **ping**... Ce mode n'est accessible qu'au super-utilisateur.
- **SOCK\_DGRAM** : *socket* orientée vers la transmission de data-grammes structurées en mode non-connecté avec une fiabilité minimale : **UDP**
- **SOCK\_STREAM** : *socket* orientée vers l'échange de séquences continues de caractères, en mode connecté avec garantie du maximum de fiabilité : **TCP**.

**3.4.3 Le protocole d'une socket** : L'argument protocole dans l'appel système socket est généralement mis à 0. Dans certaines applications spécifiques, il faut cependant spécifier la valeur du protocole.

**3.4.4 Le format de l'adresse**

#### AF\_UNIX

Une *socket* est désignée dans le domaine UNIX comme un fichier, elle a un i-nœud.

Une telle adresse correspond à la structure suivante :

```
#include <sys/un.h>
struct sockaddr_un {
    short sun_family; /* AF_UNIX */
    char sun_path[108]; /*reference */
};
```

L'attachement d'une *socket* est faite seulement si la référence n'existe pas encore.

Cette référence peut être supprimée soit à l'aide de la commande externe **rm** ou bien dans le programme avec la primitive **unlink**.

#### AF\_INET

Une *socket* est désignée dans le domaine INET non pas comme un fichier, mais possède une adresse construite en utilisant les structures suivantes :

Une telle adresse correspond à la structure suivante :

```
#include <netinet/in.h>
/* adresse Internet d'une machine */
struct in_addr {
    u_long s_addr; /* 32 bits non signé */
```



```
};
/* adresse Internet d'une socket */
struct sockaddr_in {
    short sin_family;      /* AF_INET */
    u_short sin_port;      /* N° port associé */
    struct in_addr sin_addr; /* @ IP machine */
    char sin_zero[8];      /* Champ de 8 car nuls */
};
```

- Le champ **sin\_addr.sin\_addr** peut avoir la valeur **INADDR\_ANY** permettant ainsi d'associer la socket à toutes les adresses possibles de la machine.
- Le champ **sin\_port** peut avoir une valeur particulière, afin de permettre au client d'avoir accès à un service particulier. (Voir le fichier */etc/services*).

**Remarque** : si un processus crée un *socket* et commence à émettre sans un attachement de cette *socket*, le système fera un **attachement sur un port quelconque généralement compris entre 49152 et 65535 !!!**

### 3.5 Exemple d'utilisation

Le source suivant montre comment utiliser la fonction **créer\_socket** dont le code est donné précédemment.

```
1 // Exemple d'utilisation de la fonction créer_socket
2
3 #include <unistd.h>
4 #include <stdio.h>
5 #include <stdlib.h>
6 #include <sys/types.h>
7 #include <sys/socket.h>
8 #include <netinet/in.h>
9
10 int créer_socket(int, int *, struct sockaddr_in *);
11
12 int main(){
13     struct sockaddr_in adresse_udp, adresse_tcp;
14     short port_udp, port_tcp;
15     int socket_udp, socket_tcp;
16
17     port_udp=0;
18     if((socket_udp=créer_socket(SOCK_DGRAM, &port_udp, &adresse_udp))!=-1)
19         printf("Socket UDP attaché au port : %d\n",ntohs(adresse_udp.sin_port));
20
21     port_tcp=0;
22     if((socket_tcp=créer_socket(SOCK_STREAM, &port_tcp, &adresse_tcp))!=-1)
23         printf("Socket TCP attachée au port : %d\n",ntohs(adresse_tcp.sin_port));
24
25     sleep(60);
26     return 0;
27 }
```

## 4 La communication en local

Les *sockets* sont principalement utilisées dans le domaine des réseaux afin de permettre à un serveur et un client se trouvant sur 2 machines distinctes de communiquer. On peut néanmoins les utiliser pour faire communiquer 2 processus en local.

On utilise pour cela les *sockets* dans le domaine local : **AF\_UNIX**. Cela va nous permettre de voir rapidement le fonctionnement et les mécanismes mis en jeu et nous aurons l'occasion de voir cela plus en détails lors du TP.





```

1 // Le serveur local
2
3 #include <stdio.h>
4 #include <stdlib.h>
5 #include <unistd.h>
6
7 #include <sys/types.h>
8 #include <sys/socket.h>
9 #include <sys/un.h>
10
11 int main() {
12
13     int sockfd_serveur, sockfd_client; // descripteurs pour les 2 sockets
14     struct sockaddr_un adresse_serveur; // la structure pour le socket
serveur
15     struct sockaddr_un adresse_client; // la structure pour le socket
client
16     char ch;
17     int long_client;
18
19     // On supprime un éventuel socket déjà existant...
20     unlink("socket_serveur");
21
22     // On crée le socket pour le client
23     sockfd_serveur = socket(AF_UNIX, SOCK_STREAM, 0);
24
25     // On nomme le socket
26     adresse_serveur.sun_family = AF_UNIX;
27     strcpy(adresse_serveur.sun_path, "socket_serveur");
28
29     // On attache le socket
30     if(bind(sockfd_serveur, (struct sockaddr *)&adresse_serveur, si-
sizeof(adresse_serveur))!=-1){
31         perror("pb bind : Serveur");
32         exit(1);
33     }
34
35     // On crée une file d'attente de connexion et on attend le client
36     listen(sockfd_serveur, 5);
37     while(1){
38         printf("Le serveur attend \n");
39
40         // Le serveur accepte la connexion
41         long_client = sizeof(adresse_client);
42         if(sockfd_client = accept(sockfd_serveur, (struct sockaddr
*)&adresse_client, &long_client))!=-1){
43             perror("pb accept : Serveur");
44             exit(1);
45         }
46
47         // On peut maintenant lire et écrire via sockfd
48         read(sockfd_client, &ch, 1);
49         printf("Le serveur a lu : %c\n", ch);
50         ch++;
51         printf("Le serveur écrit : %c\n", ch);
52         write(sockfd_client, &ch, 1);
53
54         //on ferme le socket avec le client
55         close(sockfd_client);
56     }
57
58     return 0;}
1 // Le client local
2
3 #include <stdio.h>
4 #include <stdlib.h>
5 #include <unistd.h>
6
7 #include <sys/types.h>
8 #include <sys/socket.h>
9 #include <sys/un.h>
10
11 int main() {
12
13     int sockfd; // le descripteur pour le socket
14     struct sockaddr_un adresse; // la structure pour le socket
15     char ch = 'A';
16
17     // On crée la socket pour le client
18     sockfd = socket(AF_UNIX, SOCK_STREAM, 0);
19
20     // On nomme le socket en "osmose" avec le serveur
21     adresse.sun_family = AF_UNIX;
22     strcpy(adresse.sun_path, "socket_serveur");
23
24     // On connect le socket au serveur
25     if((connect(sockfd, (struct sockaddr *)&adresse, si-
sizeof(adresse))!=-1)){
26         perror("pb connect : Client");
27         exit(1);
28     }
29
30     // On peut maintenant écrire et lire via sockfd
31     printf("Le client a écrit : %c \n", ch);
32     write(sockfd, &ch, 1); // le client écrit
33     read(sockfd, &ch, 1); // le client lit
34     printf("Le serveur a renvoyé : %c\n", ch);
35
36     //on ferme le socket qui n'est plus utile
37     close(sockfd);
38     return 0;
39 }

```

## 5 La communication en mode connecté

C'est le mode de communication associé aux *sockets* de type **SOCK\_STREAM**. Il permet à des applications réparties de s'échanger des séquences de caractères **continues** et **non structurées en messages**. Il est adapté à l'implantation d'un mécanisme de connexion à distance tel **telnet**.

Avec ces *sockets* et la communication en mode connecté, la dissymétrie entre le client et le serveur est réelle au niveau de l'établissement de la connexion : le serveur est en attente passive de demande de connexion que lui adressera le client. Ce dernier prend l'initiative de la demande de connexion, il est actif.

### 5.1 L'établissement de la connexion

#### 5.1.1 Le serveur : Un serveur est passif dans l'établissement d'une connexion.

De son côté, il crée la *socket* (**socket**), l'attache (**bind**), prévient le système auquel il appartient qu'il est prêt à accepter les demandes de connexion des clients (**listen**). Il se met en attente de demande de connexion.



Il dispose d'une **socket d'écoute** attachée au port correspondant au service et supposé connu des clients. Lorsqu'une demande de connexion arrive, le système crée une **nouvelle socket** dédiée à cette nouvelle connexion que l'on nomme **socket de service**, ce qui permet de multiplexer sur la même **socket** plusieurs connexions.

Le processus serveur prend connaissance de l'existence d'une nouvelle connexion par un appel à la primitive **accept** : au retour de cet appel, le processus reçoit un **descripteur** lui permettant d'accéder à cette **socket de service**.

Après avoir accepté une connexion le serveur qui est le père crée un fils (**fork**) ou une activité (**pthread\_create**) et le dialogue se fait alors entre le fils et un client. Cela permet au serveur de pouvoir prendre plus rapidement en compte les nouvelles demandes de connexion. C'est ce mécanisme qui est utilisé lorsque vous êtes plusieurs à vouloir vous connecter simultanément sur un serveur Web, FTP, telnet etc...

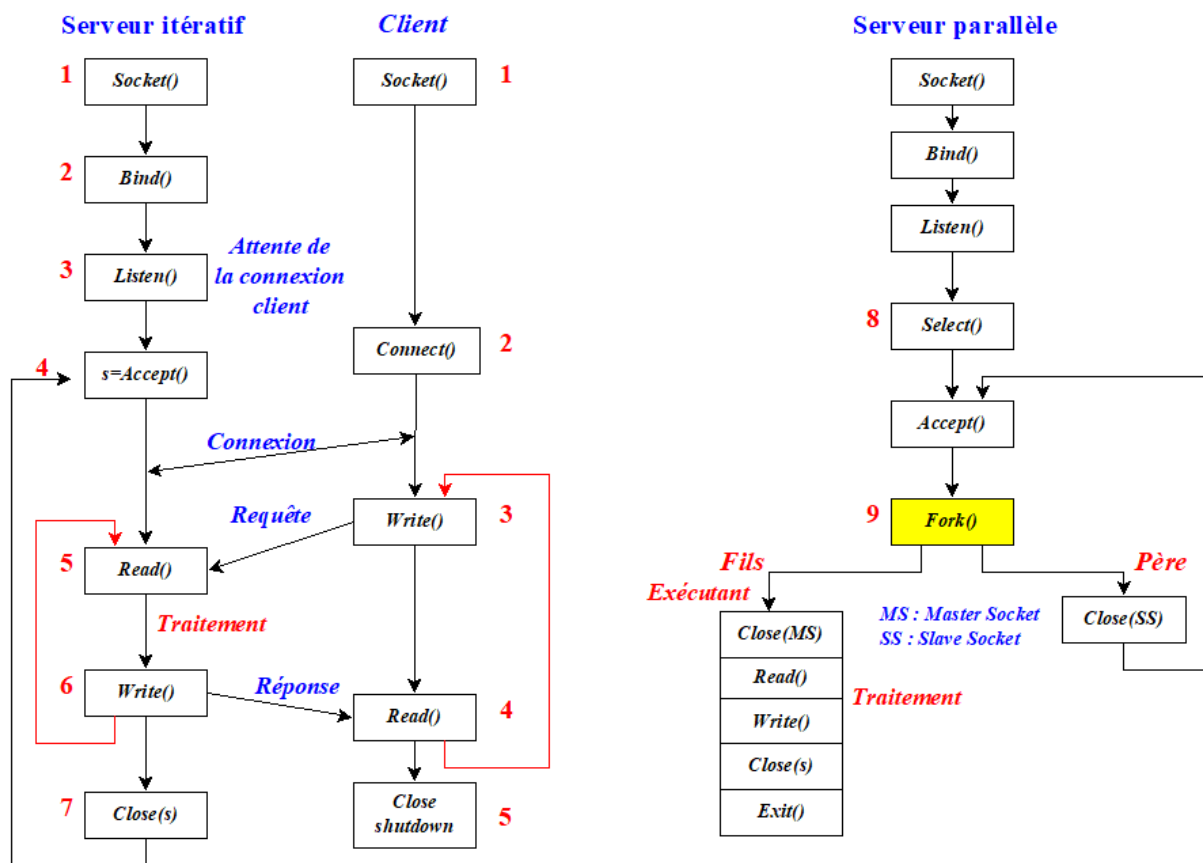
Une connexion TCP est identifiée par un 4-uple :

**(adresse\_machine\_1, port\_machine\_1, adresse\_machine\_2, port\_machine\_2)**

**5.1.2 Le client** : est l'entité active<sup>4</sup> dans le processus d'établissement d'une connexion avec un serveur

C'est lui qui prend l'initiative. Cette demande de connexion est réalisée à l'aide de la primitive **connect**. Il n'est pas nécessaire, pour appeler la primitive **connect**, que la **socket** locale ait été au préalable attachée à une adresse : si un appel à la primitive **bind** n'a pas été fait avant la demande de connexion, un attachement sur un port quelconque sera automatiquement réalisé à l'appel de la primitive **connect**.

L'image qui suit donne le déroulement d'une communication entre un serveur **itératif** TCP et un client TCP, ainsi que la communication entre un serveur **parallèle** TCP et un client TCP.



#### Côté serveur

1. Création de la socket serveur
2. Récupération de l'adresse IP et du numéro de port du serveur donc le lien de la socket à l'adresse du serveur
3. Mise en **mode passif** de la socket : elle est prête à accepter les requêtes des clients
4. Un peu comme les étudiants qui sont actifs et le prof qui est passif et attend leurs questions afin d'y répondre :)



4. (opération bloquante) : acceptation d'une connexion d'un client et création d'une socket service client, dont l'identité est rendue en retour
5. Lecture,  
**traitement** et
6. Écriture (selon algorithme du service)
7. Fermeture de la socket et remise en attente

#### Côté client

1. Création de la socket
2. Connexion de la socket au serveur
  - (a) choix d'un port libre pour la socket par la couche TCP
  - (b) attachement automatique de la socket à l'adresse (IP machine locale + n° de port)
  - (c) connexion de la socket au serveur en passant en paramètre l'adresse IP et le n° de port du serveur
3. Écriture donc dialogue avec le serveur (selon algorithme du service)  
**traitement** et
4. Lecture
5. Fermeture de la connexion avec le serveur

## 6 La communication en mode non connecté

**Un serveur en mode non connecté est comme un étudiant en amphi sur son téléphone portable, il n'écoute pas et n'accepte pas l'information donné par le prof!!! donc pas de listen() ni d'accept()!!!**

C'est le mode de communication associé aux *sockets* de type **SOCK\_DGRAM**.

Il permet à des applications réparties de s'échanger des données.

### 6.1 L'établissement de la connexion :) **ATTENTION!!!**

On ne peut pas faire le même plan qu'avec TCP, car nous sommes en mode non connecté donc il n'y a pas d'établissement de la connexion...

Les mécanismes sont beaucoup plus simples que pour le mode connecté vu précédemment :

Un processus désirant communiquer avec une *socket* de type **SOCK\_DGRAM** doit donc réaliser les opérations suivantes :

- Demander la création d'une *socket* du domaine **AF\_UNIX** pour les communications locales ou du domaine **AF\_INET** pour les communications distantes en utilisant le protocole UDP.
- Demander éventuellement l'attachement de cette *socket* sur un port convenu s'il est le serveur ou un port quelconque s'il est le client prenant l'initiative d'interroger le serveur.
- Construire l'adresse de son interlocuteur en mémoire.
  - Un client qui s'adresse à un serveur doit en connaître l'adresse et donc la préparer en mémoire, il peut pour cela consulter la base de données des services et utiliser la primitive **gethostbyname** pour obtenir l'adresse IP du serveur.
  - Si c'est un serveur, il recevra avec chaque message l'adresse du client qu'il utilisera pour renvoyer la réponse.
- Procéder à des émissions et des réceptions de messages.

Chaque demande d'envoi de datagramme est accompagné de la spécification complète de l'adresse de son destinataire et de même, chaque réception est accompagné de la spécification complète de l'adresse de son destinataire l'émetteur.

**Remarque** : il est possible avec des *sockets* de type **SOCK\_DGRAM** de réaliser des pseudo-connexions, mais nous n'aurons pas le temps de les étudier alors je ne vais pas en parler !

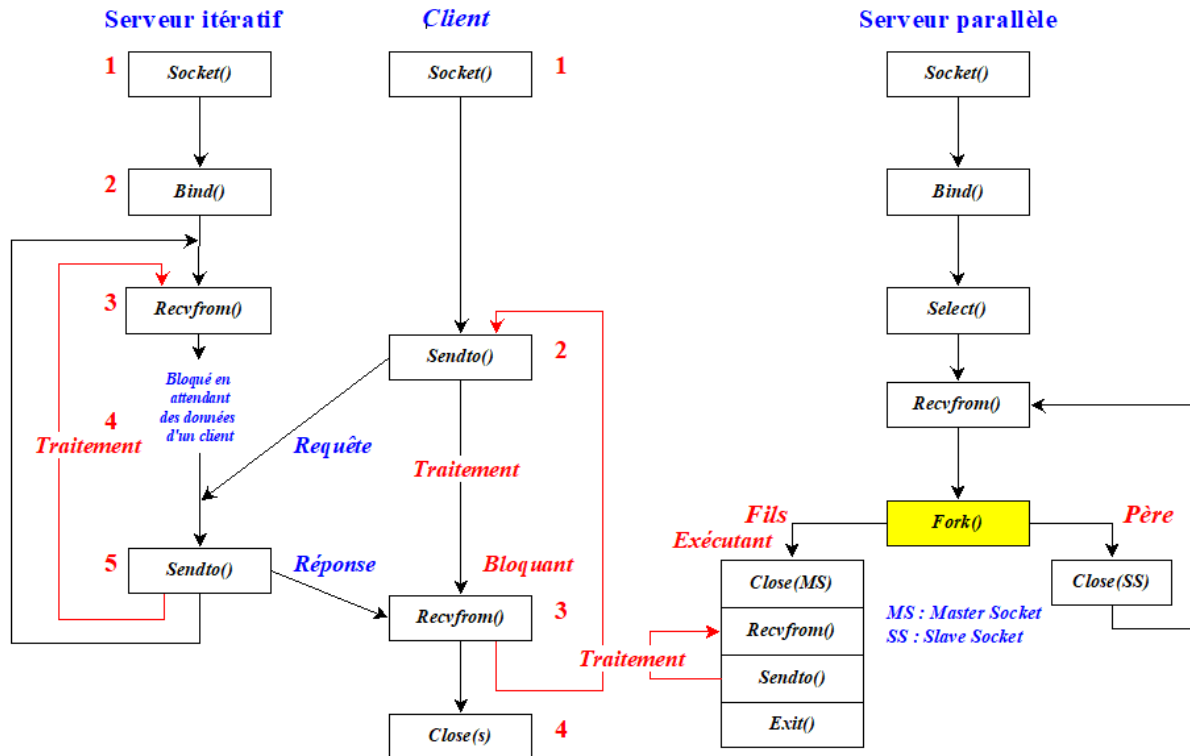
**Attention**, on rappelle que dans le domaine **AF\_INET**, la communication s'appuie sur le protocole UDP.

Il n'y a donc pas de service permettant à l'émetteur de savoir si son message est arrivé à destination ou non.

De plus, si plusieurs messages sont envoyés au même destinataire, l'émetteur n'est pas assuré que ses messages seront délivrés dans le bon ordre.



L'image qui suit donne le déroulement d'une communication entre un serveur **itératif** UDP et un client UDP, ainsi que la communication entre un serveur **parallèle** UDP et un client UDP.



### Côté serveur

1. Création de la socket serveur
2. Récupération de l'adresse IP et du numéro de port du serveur donc le lien de la socket à l'adresse du serveur
3. Réception d'une requête de client
4. Traitement de la requête ; préparation de la réponse
5. Réponse à la requête en utilisant la socket et l'adresse du client obtenues par **recvfrom** ; retour pour attente d'une nouvelle requête

### Côté client

1. Création de la socket avec l'association à une adresse locale [adresse IP + n° port] est faite automatiquement lors de l'envoi de la requête.
2. Envoi d'une requête au serveur en spécifiant son adresse dans l'appel **traitement** et
3. Réception de la réponse à la requête
4. Fermeture de la socket

**Exemples** de 2 codes de clients pour se connecter au serveur Daytime.

```

1 // Client TCP
  // Usage : Client @IP

#include <stdio.h>
#include <stdlib.h>
6 #include <string.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/socket.h>
11 #include <sys/param.h>

```



```

#include <netinet/in.h>
#include <arpa/inet.h>

16 #define USAGE "Usage:%s adresse_du_serveur\n"
#define MAXMSG 1024
#define NPORT 13

int main(int argc, char *argv[]) {
21 int n, sfd ;
char buf[MAXMSG] ;
struct sockaddr_in saddr ;
if (argc != 2) {
    (void)fprintf(stderr,USAGE,argv[0]) ;
26 exit(EX_USAGE) ;
}
if ((sfd = socket(PF_INET,SOCK_STREAM,IPPROTO_TCP)) < 0) {
    perror("socket") ;
    exit(EX_OSERR) ;
31 }
saddr.sin_family = AF_INET ;
saddr.sin_port = htons(NPORT) ; /* Attention au NBO ! */
saddr.sin_addr.s_addr = inet_addr(argv[1]) ;
if (connect(sfd,(struct sockaddr *)&saddr,sizeof saddr) < 0) {
36 perror("connect") ;
    exit(EX_OSERR) ;
}
if ((n = read(sfd, buf,MAXMSG-1)) < 0) {
    perror("read") ;
41 exit(EX_OSERR) ;
}
buf[n] = '\0' ;
(void)printf("Date(%s) = %s\n",argv[1],buf) ;
exit(EX_OK) ; /* close(sfd) implicite */
46 }

// Client UDP
// Usage : Client @IP

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

#define USAGE "Usage:%s adresse_du_serveur\n"
#define MAXMSG 1024
#define NPORT 13
19
int main(int argc, char *argv[]) {
    int n, sfd ;
    char buf[MAXMSG] ;
    struct sockaddr_in saddr ;
24 if (argc != 2) {
        (void)fprintf(stderr,USAGE,argv[0]) ;
        exit(EX_USAGE) ;
    }
    if ((sfd = socket(PF_INET,SOCK_DGRAM,IPPROTO_UDP)) < 0) {

```



```

29     perror("socket") ;
        exit(EX_OSERR) ;
    }
    saddr.sin_family = AF_INET ;
    saddr.sin_port = htons(NPORT) ; /* Attention au NBO ! */
34    saddr.sin_addr.s_addr = inet_addr(argv[1]) ;
    if (sendto(sfd, buf, 1, 0, (struct sockaddr *)&saddr, sizeof saddr) !=1) {
        perror("sendto");
        exit(EX_OSERR);
    }
39    if ((n = recv(sfd, buf, MAXMSG-1, 0)) < 0) {
        perror("recv") ;
        exit(EX_OSERR) ;
    }
    buf[n] = '\0' ;
44    (void)printf("Date(%s) = %s\n", argv[1], buf) ;
    exit(EX_OK) ; /* close(sfd) implicite */
}

```

## 7 Les primitives

Dans cette partie nous allons voir comment utiliser les primitives de base pour programmer des applications réseau. Certaines de ces primitives sont utilisées pour les modes connecté et non connecté, d'autres ne fonctionnent qu'avec le mode connecté et bien sûr d'autres que pour le mode non connecté.

Il faudra donc bien faire attention lors de la programmation d'un client et/ou un serveur selon qu'il soit destiné au mode oui/non connecté.

**Remarque** : Tout n'est pas expliqué, si vous voulez une documentation plus exhaustive, je vous autorise à aller consulter les *man* :)

### 7.1 Créer une Socket

#### En modes connecté et non connecté

La primitive **socket** permet de créer une **int socket(**  
**socket** ayant un domaine, un type et un **int domaine,**  
protocole. **int type**  
La valeur de retour est un descripteur sur **int protocole**  
la **socket**. )

```

/* AF_UNIX, AF_INET*/
/* SOCK_DGRAM, SOCK_STREAM*/
/* 0 : protocole par défaut */

```

Pour le protocole, on peut utiliser les constantes IPPROTO\_UDP et IPPROTO\_TCP prédéfinies dans le fichier d'en tête <netinet/in.h>

### 7.2 attacher une Socket

#### En modes connecté et non connecté

La primitive **bind** permet d'attacher la **int bind(**  
**socket** de descripteur **descripteur** à **int descripteur,** /\* descripteur socket \*/  
l'adresse **\*adresse**. Le paramètre **longueur\_adresse** est égale à la taille, en octets, de cette adresse. **struct sockadr \*adresse,** /\* adresse \*/  
**int longueur\_adresse** /\* longueur zone adresse \*/  
**)**

### 7.3 Connexion à une adresse distante

#### En mode connecté seulement



La primitive **connect** permet d'associer la *socket* locale identifiée par le descripteur **descripteur** à la *socket* d'adresse **\*adresse**.  
Tout nouvel appel annule la demande précédente

```
int connect(
    int descripteur,           /* descripteur socket */
    struct sockadr *adresse, /* adresse de réception */
    int *longueur_adresse     /* taille zone réservée */
) /* 0 */
/* adresse émetteur */
/* pointeur sur longueur zone
   adresse */
```

## 7.4 Spécifier une file d'attente

### En mode connecté seulement

La primitive **listen** permet à un processus de déclarer un service ouvert auprès de son système local. Après cet appel, l'entité TCP commence à accepter les connexions.

```
int listen(
    int descripteur,           /* descripteur socket */
    int nb_pendantes,         /* nb max de connexions pendantes
                               acceptables */
)
```

## 7.5 Accepter une connexion

### En mode connecté seulement

La primitive **accept** permet à un processus de prendre connaissance de l'existence d'une nouvelle connexion, extraite de la liste des connexions pendantes. Un **descripteur** sur une **socket de service** dédiée à cette nouvelle connexion est renvoyé.

```
int accept(
    int descripteur,           /* descripteur socket */
    struct sockadr *adresse_client, /* adresse du client */
    int *longueur_adresse      /* longueur adresse destinataire */
)
```

Ce nouveau descripteur permet d'identifier localement la connexion. Voir le 4-uple de la partie mode connecté.

La *socket*, dite **socket de service** de ce descripteur renvoyé permet d'envoyer et de recevoir des caractères sur la connexion. Mais **elle ne permet pas d'accepter de nouvelles connexions!!!**

L'adresse de la *socket* du client est alors écrite en mémoire à l'adresse **\*adresse\_client**.

## 7.6 Envoyer des données

### En mode connecté

La primitive **write** permet de d'envoyer sur la *socket* **nb\_caractères** caractères qui seront lus de la mémoire à l'adresse **message**.  
La fonction renvoie le nombre de caractères envoyés.

```
int write(
    int descripteur,           /* descripteur socket */
    void *message,            /* adresse de réception */
    int nb_caractères,        /* quantité octets */
)
```

### En mode non connecté

La primitive **sendto** permet d'envoyer, via la *socket* identifiée localement par le **descripteur** donné, le **message** de **longueur** spécifiée à destination de la *socket* dont l'adresse est pointée par **adresse** et est de longueur **longueur\_adresse**.  
La valeur de retour est le nombre de caractères effectivement envoyés.

```
int sendto(
    int descripteur,           /* descripteur socket */
    void *message,            /* message à envoyer */
    int longueur,             /* longueur de message */
    int option,               /* 0 */
    struct sockadr *adresse, /* adresse destinataire */
    int longueur_adresse     /* longueur adresse destinataire */
)
```

## 7.7 Recevoir des données

### En mode connecté

La primitive **read** permet de lire sur la *socket* d'au plus **taille** caractères qui seront écrits en mémoire à l'adresse **message**.  
La fonction renvoie le nombre de caractères lus.

```
int read(
    int descripteur,           /* descripteur socket */
    void *message,            /* adresse de réception */
    int taille,               /* quantité octets */
)
```

### En mode non connecté





La primitive **recvfrom** permet de lire, **int recvfrom(**  
sur la **socket** de **descripteur** spécifié, **int descripteur,** /\* descripteur *socket* \*/  
le message et d'en récupérer le contenu **void \*message,** /\* adresse de réception \*/  
à l'adresse **message**. Le paramètre **longueur**, **int longueur,** /\* taille zone réservée \*/  
**int option,** /\* 0 \*/  
**struct sockadr \*adresse,** /\* adresse émetteur \*/  
**int \*longueur\_adresse** /\* pointeur sur longueur zone  
) adresse \*/

## 7.8 Terminer une connexion

### En mode connecté seulement

Elle n'est nécessaire que pour le mode connecté, il existe 2 primitives pour cela :

1. La première **close** entraîne la suppression de la *socket*, si elle est faite sur le dernier descripteur. Dans le cas d'une *socket* de type `SOCK_STREAM`, s'il y a encore des données dans le tampon, le système continu de les acheminer.

```
#include <unistd.h>
int close(int fildes);
```

2. La seconde **shutdown**, elle permet de rendre la communication entre 2 *sockets* non *full-duplex*.

Si le champ **how**, qui définit le sens vaut

- 0 ou `SHUT_RD` alors la *socket* n'accepte plus de lecture, un appel *read* ou *recv* renverra la valeur 0 ;
- 1 ou `SHUT_WR` alors la *socket* n'accepte plus d'écriture, un appel *write* ou *send* provoquera, comme pour les tubes, la génération d'un exemplaire du signal `SIGPIPE` ;
- 2 ou `SHUT_RDWR` alors la *socket* n'accepte plus ni lecture, ni écriture...

```
#include <sys/socket.h>
int shutdown(int socket, int how);
```

## 8 Conclusion

Ce n'est qu'un début... et les exemples sont en langage C, si vous voulez programmer une application dans d'autres langages, à quelques exceptions près vous n'aurez qu'à faire du transcodage...

Il faudra surtout veiller à respecter les consignes données dans le déroulement d'une communication entre un serveur et un client ou plusieurs clients en fonction des protocoles etc...

### Exemple en python !

```
# -*- coding: utf-8 -*-
# Le serveur.py
import socket

4 hote = ''
  port = 1664

connexion_principale = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
connexion_principale.bind((hote, port))
9 connexion_principale.listen(5)

print()
print()
print("_____Le serveur écoute à présent sur le port_{}".format(port))
14 print("_____Je vous sers quoi_:_?")
   print()
   print()

connexion_avec_client, infos_connexion = connexion_principale.accept()
19 msg_recu = b""
while msg_recu != b"fin":
    msg_recu = connexion_avec_client.recv(1024)
```



```

24  # L'instruction ci-dessous peut lever une exception si le message
    # Réceptionné comporte des accents
    print("Le_client_commande_:" + msg_recu.decode())
    print()
    connexion_avec_client.send(b"vous_m'avez_demand\xc3\xa9:" + msg_recu + b"_bien_recu_5_/_5")
29
    print("Fermeture_de_la_connexion")
    connexion_avec_client.close()
    connexion_principale.close()

    #- coding: utf-8 -#
    # Le client.py
3  import socket

    hote = "localhost"
    port = 1664


    connexion_avec_serveur = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    connexion_avec_serveur.connect((hote, port))
    print()
    print()
    print("Connexion_établie_avec_le_serveur_sur_le_port_{}".format(port))
13  print()
    print()

    msg_a_envoyer = b""
    while msg_a_envoyer != b"fin":
18        msg_a_envoyer = input(">")
        # Peut planter si vous tapez des caractères spéciaux
        msg_a_envoyer = msg_a_envoyer.encode()
        # On envoie le message
        connexion_avec_serveur.send(msg_a_envoyer)
23        msg_recu = connexion_avec_serveur.recv(1024)
        print(msg_recu.decode()) # Là encore, peut planter s'il y a des accents

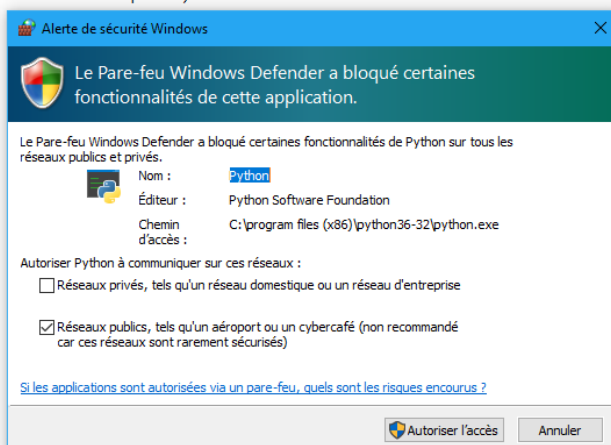
    print("Fermeture_de_la_connexion")
    connexion_avec_serveur.close()

```

Cela donne

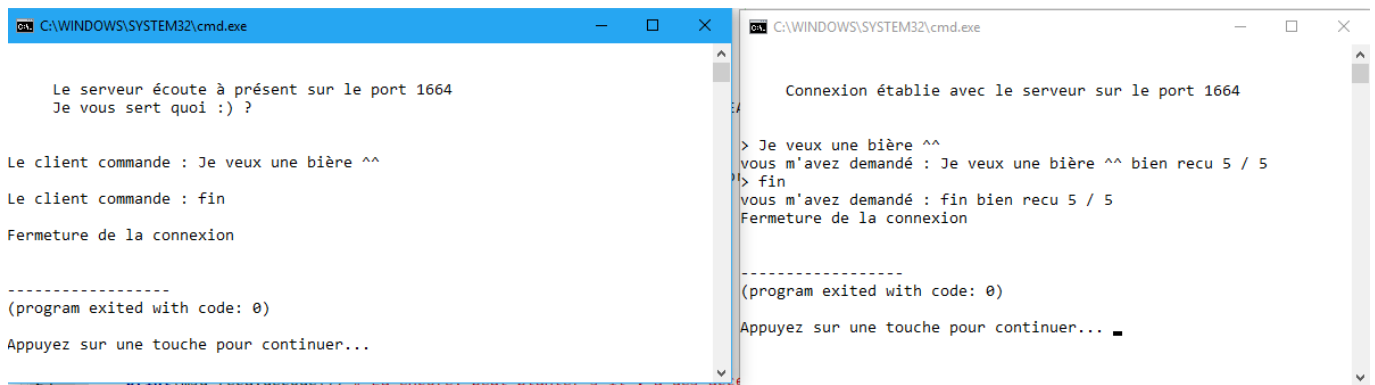
 C:\WINDOWS\SYSTEM32\cmd.exe

Le serveur écoute à présent sur le port 1664  
Je vous sert quoi :) ?



et





```
C:\WINDOWS\SYSTEM32\cmd.exe
Le serveur écoute à présent sur le port 1664
Je vous sert quoi :) ?

Le client commande : Je veux une bière ^^
Le client commande : fin
Fermeture de la connexion

-----
(program exited with code: 0)
Appuyez sur une touche pour continuer...
```

```
C:\WINDOWS\SYSTEM32\cmd.exe
Connexion établie avec le serveur sur le port 1664

> Je veux une bière ^^
vous m'avez demandé : Je veux une bière ^^ bien reçu 5 / 5
> fin
vous m'avez demandé : fin bien reçu 5 / 5
Fermeture de la connexion

-----
(program exited with code: 0)
Appuyez sur une touche pour continuer...
```