

Cours

Sockets TCP en C

Safa YAHY

[safa.yahi@univ-amu.fr](mailto:safa.yahi@univ-amu.fr)

année 2023-2024

# Programmation répartie

Mécanismes de communication entre applications réparties :

- **Sockets (bas niveau)**
- RPC (Remote Procedure Call)
- RMI (Remote Method Invocation)
- Services Web

## Rappels - Modèles de communication

La plupart des communications réseau se font selon le modèle **client-serveur** :

- Le serveur (prestataire de services) se lance en premier et se met à l'écoute des requêtes des clients.
- Le client initie la communication avec le serveur en lui envoyant une requête : demander une page Web, envoyer un email, récupérer un email, transférer un fichier, etc.

## Rappels - Modèles de communication

- Certaines communications s'effectuent selon le modèle pair à pair (P2P peer to peer en anglais) où un programme peut jouer à la fois le rôle de client et de serveur.
- Nous nous intéresserons ici au modèle client-serveur.

## Rappels - Ports de communication

- Une même machine (identifiée par une adresse IP) peut offrir plusieurs services à la fois (web, email, fichier, etc).
- L'adresse IP n'est pas suffisante => utilisation de **ports** (ports logiques).
- Un port est un numéro qui va de 1 à 65535.
- Les ports de 1 à 1023 sont attribués à des services courants : http(80), smtp(25), daytime(13), echo(7), etc.
- Sur Linux, le fichier /etc/services donne pour chaque service le port associé.

## Rappels - Protocole TCP

- Un protocole orienté connexion
- Même principe qu'une conversation téléphonique :
  - Établir une connexion
  - Envoi de données
- Fiabilité : garantie de la non perte des données et de leur récupération dans le bon ordre.

## Rappels - Protocole UDP

- Utilise un protocole sans connexion
- Peut être vu comme un service postal : deux applications peuvent s'envoyer des messages sans établir une connexion entre elles.
- Protocole non fiable : l'arrivée n'est pas vérifiée par l'émetteur
- Convient à des applications en temps réel, comme la visioconférence
- Il convient aussi pour des applications où l'échange entre client et serveur est assez concis (requête / réponse) et donc la gestion de la fiabilité peut être déléguée à la couche application.

## Rappels - Protocole UDP

➤ UDP permet, et ce contrairement à TCP, des communications en multicast :

envoyer un même message à un groupe de destinataires d'un seul coup.



# Sockets

- Les **sockets** sont un mécanisme de bas niveau d'IPC (Inter-Process Communication) permettant l'échange de données entre applications sur une même machine ou sur des machines distantes connectées via un réseau.
- Types de sockets :
  - Sockets du domaine UNIX
  - Sockets du domaine Internet (IPv4 et Ipv6) : mode TCP, mode UDP, etc.

## Sockets en mode TCP

- Le serveur crée une socket passive liée à une adresse IP (de la machine sur laquelle il s'exécute) et un port.
- Le serveur attend, via cette socket d'écoute, les requêtes des clients (qui doivent connaître son adresse IP et son numéro de port).
- Le client tente de se connecter au serveur.
- Si le serveur accepte la connexion de ce client :
  - une nouvelle socket active est créée au niveau serveur
  - une nouvelle socket active est créée au niveau client

## Sockets en mode TCP

- La socket du client est liée localement à son adresse IP et un port généralement assigné par le SE.
- La nouvelle socket du serveur est liée au même port local que sa socket d'écoute. Le point de terminaison (@ IP, port) distant de cette nouvelle socket comprend l'adresse et le port du client.
- Une fois la connexion établie, le client et le serveur peuvent communiquer via des E/S sur les nouvelles sockets.
- Le serveur continue à écouter d'autres requêtes sur la première socket d'écoute.

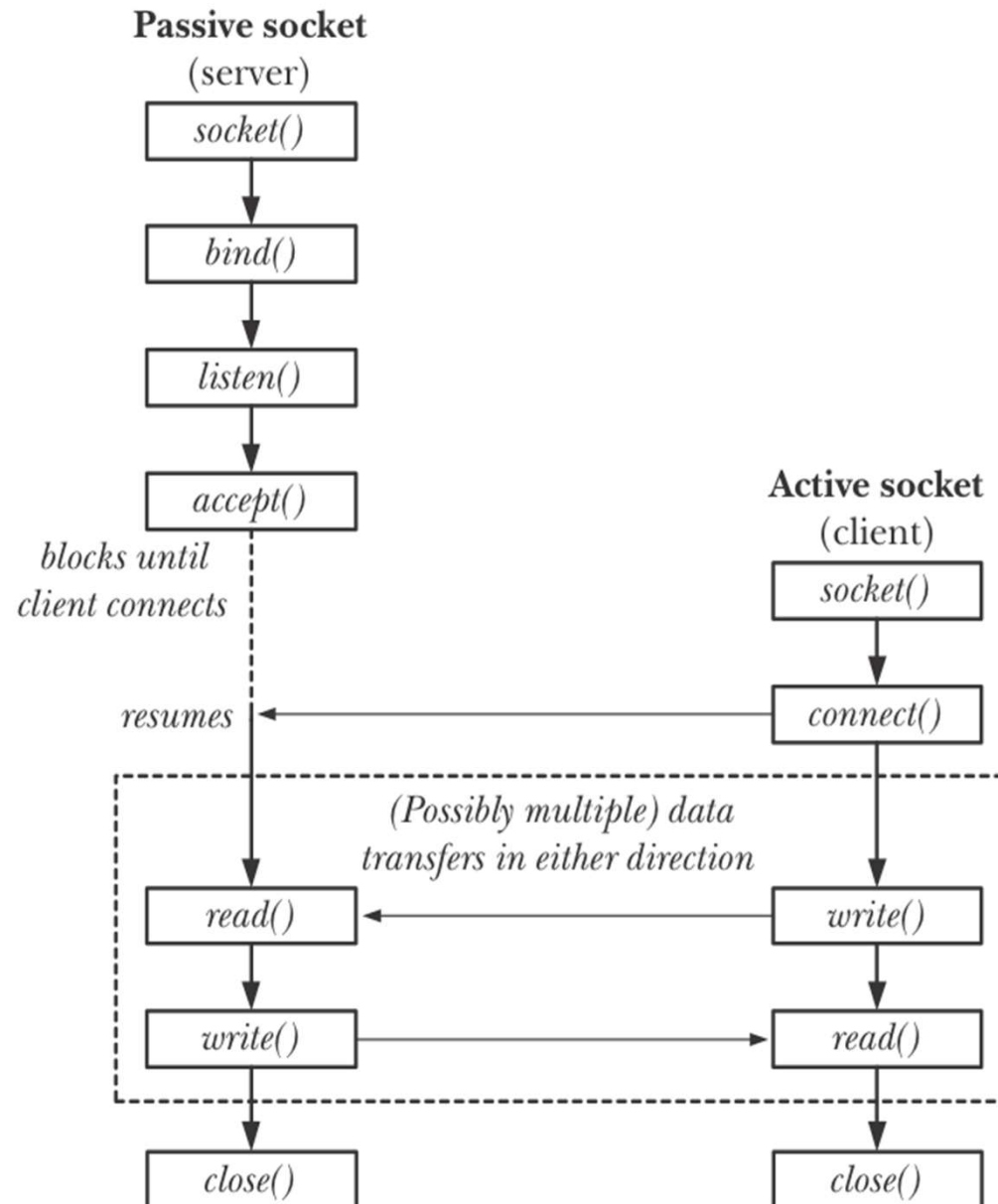
## Démo sur Sockets en mode TCP

- Visualiser la socket d'écoute d'un serveur FTP installé sur une VM avec “netstat -ltn”
- Connecter un client (gFTP, par exemple) depuis une autre VM et voir les nouvelles sockets d'écoute créées sur la machine du client et sur le serveur avec “netstat -atn”.
- Connecter un autre client FTP (via telnet @serveur 21) et constater que le port local du client a changé.
- Deviner la nouvelle socket créée côté serveur.

## Implémentation des sockets

- La première implémentation de l'API des sockets remonte à 1983 avec 4.2BSD (pour Berkeley Software Distribution) qui est une version d'UNIX.
- Plusieurs API de sockets
- API en C : la plus proche du SE.

# Schéma général client / serveur TCP en C



A copier sur feuille :)

## socket()

```
#include <sys/socket.h>  
  
int socket(int domain, int type, int protocol);
```

- crée une socket.
- retourne :
  - un descripteur de fichier qui identifie la socket créée en cas de succès
  - -1 en cas d'erreur

## socket()

```
#include <sys/socket.h>  
  
int socket(int domain, int type, int protocol);
```

Paramètre domain : domaine de communication

• .AF\_UNIX (ou AF\_LOCAL) : communication locale

• .AF\_INET : communication IPv4

• .AF\_INET6 : communication IPv6

•



## socket()

Pour chaque domaine de communication, on a un type d'adresses de sockets :

Domaine	Format d'adresse	Type d'adresse
AF_UNIX	pathname	<b>struct sockaddr_un</b>
AF_INET	adresse IPv4 de 32 bits + un port sur 16 bits	<b>struct sockaddr_in</b>
AF_INET6	adresse IPv6 sur 128 bits + un port sur 16 bits	<b>struct sockaddr_in6</b>

## socket()

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

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

Pour le domaine AF\_INET

.SOCK\_STREAM : mode TCP (pour le domaine Internet)

.SOCK\_DGRAM : mode UDP (pour le domaine Internet)

.SOCK\_RAW : permet de communiquer directement avec la couche IP (exemple : ICMP).

## socket()

```
#include <sys/socket.h>
int socket(int domain, int type, int protocol);
```

Paramètre protocol : indique le protocol de communication.  
En général, **protocol = 0**

## Exemple : serveur daytime TCP

- Ecrire un serveur daytime TCP en C qui écoute, par exemple, sur le port 50013 et l'adresse IP 192.168.1.1.
- Le protocole **daytime** en TCP est très simple :
  - le client se connecte au serveur
  - le serveur lui envoie un msg contenant la date et l'heure
  - le serveur ferme la connexion avec ce client.
- Par défaut, daytime utilise le port 13 (accès root)
- Voir démo sur un terminal : `netstat -ltn` ensuite avec `telnet adresse_serveur port_serveur`

## Exemple : serveur daytime TCP

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

```
int main()
```

```
{
```

```
int sock_serveur = socket(AF_INET, SOCK_STREAM, 0);
```

```
// ...
```

```
return 0;
```

```
}
```

Copier le code sur une feuille au fur et à mesure !

## bind()

Mais c'est quoi sockaddr ?

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

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

• `bind()` affecte l'adresse de socket spécifiée par addr à la socket référencée par sockfd.

• addrlen donne la taille, en octets, de la structure d'adresse pointée par `addr`.

• Retourne 0 en cas de succès, -1 en cas d'échec.

## bind()

La fonction `bind()` (comme d'autres fonctions de sockets) est commune à tous les domaines de communication des sockets (`AF_INET`, `AF_INET6`, `AF_UNIX`).

=> elle doit accepter les différents types d'adresses de sockets : `sockaddr_in`, `sockaddr_in6`, `sockaddr_un`.

=> d'où sa définition avec le type **générique struct sockaddr**

## struct sockaddr : une structure d'adresse générique

• La structure **sockaddr** est définie par

```
• struct sockaddr {  
•     sa_family_t sa_family;  
•     char        sa_data[]; }
```



Juste pour info

• La structure d'adresse effectivement passée dans addr au moment de l'appel dépend du domaine de communication : **sockaddr\_in**, **sockaddr\_in6** ou **sockaddr\_un**.

• Ajouter un transtypage



## Structure sockaddr\_in

```
struct sockaddr_in
```

```
    { sa_family_t    sin_family;          /*vaut AF_INET*/
```

```
        uint16_t      sin_port;           /* port dans l'ordre  
réseau */
```

```
        struct in_addr sin_addr;         /* Adresse IP */
```

```
};
```

## Structure in\_addr

La structure **in\_addr** est définie par :

```
struct in_addr  
{  uint32_t s_addr;  // adresse IP dans l'ordre réseau };
```

## String vers in\_addr

```
int inet_aton(const char *_cp, struct in_addr *_inp);
```

La fonction `inet_aton()` récupère la représentation binaire de l'adresse IPv4 donnée par la chaîne `cp` et la stocke dans la structure `in_addr` pointée par `inp`.

### Exemple

```
.struct in_addr myaddr;  
.inet_aton("10.20.30.1", &myaddr);
```

## in\_addr vers string

```
char * inet_ntoa(struct in_addr in);
```

Retourne la chaîne de caractères qui correspond à la notation décimale pointée de l'adresse IP définie par la structure in\_addr in.

### Exemple

```
char * buf = inet_ntoa(myaddr); // myaddr de l'exemple précédent.
```

## Constantes d'adresses

• La constante **INADDR\_LOOPBACK** représente l'adresse loopback (127.0.0.1).

• La constante **INADDR\_ANY** représente l'adresse wildcard  
0.0.0.0

• `sockaddr_server.sin_addr.s_addr = htonl(INADDR_ANY)`

=> le serveur écoute sur toutes les interfaces réseau de sa machine.

```
int main()

{

int sock_serveur = socket(AF_INET, SOCK_STREAM,0);


struct sockaddr_in sockaddr_serveur;


sockaddr_serveur.sin_family = AF_INET;


sockaddr_serveur.sin_port = htons(50013);


inet_aton("192.168.1.1", &sockaddr_serveur.sin_addr);
// sockaddr_serveur.sin_addr contient l'adresse IP 192.168.1.1

bind (sock_serveur, (struct sockaddr *) &sockaddr_serveur, sizeof(struct
sockaddr_in ));
```

## Network byte order / Host byte order

- Les champs *sin\_port* et *s\_addr* doivent être dans l'**ordre réseau** qui correspond au Big Endian.
- Il existe plusieurs ordres pour ranger un entier en mémoire selon l'architecture de la machine en question :
  - **Big Endian** : l'octet de poids le plus fort est rangé en premier (à la petite adresse mémoire)
  - **Little Endian** : on commence par l'octet de poids faible.

## Network byte order / Host byte order

Convertir des entiers non signés de l'ordre machine vers l'ordre réseau :

```
.uint16_t htons(uint16_t hostshort); // host to network "short"  
.uint32_t htonl(uint32_t hostlong); // host to network "long"
```



## Network byte order / Host byte order

Convertir des entiers non signés de l'ordre réseau vers l'ordre machine :

```
.uint16_t ntohs(uint16_t netshort); //network to host "short"  
.uint32_t ntohl(uint32_t netlong); // network to host "long"
```

## listen()

```
#include <sys/socket.h>  
  
int listen(int sockfd, int backlog);
```

- La fonction `listen()` met la socket sockfd en mode écoute: elle sera utilisée pour attendre des clients (via `accept()`).
- backlog est la taille maximale de la file d'attente des connexions entrantes.
- Si `backlog` dépasse la valeur spécifiée dans </proc/sys/net/core/somaxonn> alors il est réduit à cette valeur (**128** par défaut).

## accept()

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

```
int accept(int sockfd, struct sockaddr *addr, socklen_t *addrlen);
```

• S'il n'y a aucune demande de connexion en attente, l'appel `accept()` bloque, par défaut, le serveur jusqu'à nouvelle demande de connexion.

• S'il y a une demande de connexion, le serveur crée une nouvelle socket dont le descripteur est retourné par `accept()`. La nouvelle socket est une socket active et servira pour communiquer avec le client.

• `accept()` retourne -1 en cas d'erreur.

## accept()

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

```
int accept(int sockfd, struct sockaddr *addr, socklen_t *addrlen);
```

- En acceptant une demande de connexion, la structure pointée par addr reçoit l'adresse de la socket du client.
- Avant d'appeler `accept()`, le serveur initialise addrlen par la taille (en octets) de la structure pointée par `addr`.
- addrlen est renseigné au retour par la longueur réelle (en octets) de l'adresse remplie.

## accept()

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

```
int accept(int sockfd, struct sockaddr *addr, socklen_t *addrlen);
```

- Si on ne veut pas voir l'adresse du client, `addr` et `addrlen` peuvent être initialisés à `NULL`.
- On pourra les récupérer si besoin par la suite par `getpeername()`.

## Example : serveur daytime TCP (suite)

```
// section include

int main()
{
    int sock_serveur = socket(AF_INET, SOCK_STREAM, 0);

    struct sockaddr_in sockaddr_serveur;

    sockaddr_serveur.sin_family = AF_INET;
    sockaddr_serveur.sin_port = htons (50013);
    inet_aton("192.168.1.1", &sockaddr_serveur.sin_addr);

    bind (sock_serveur, (struct sockaddr *) &sockaddr_serveur, sizeof(struct sockaddr_in ));

    listen(sock_serveur, 128);

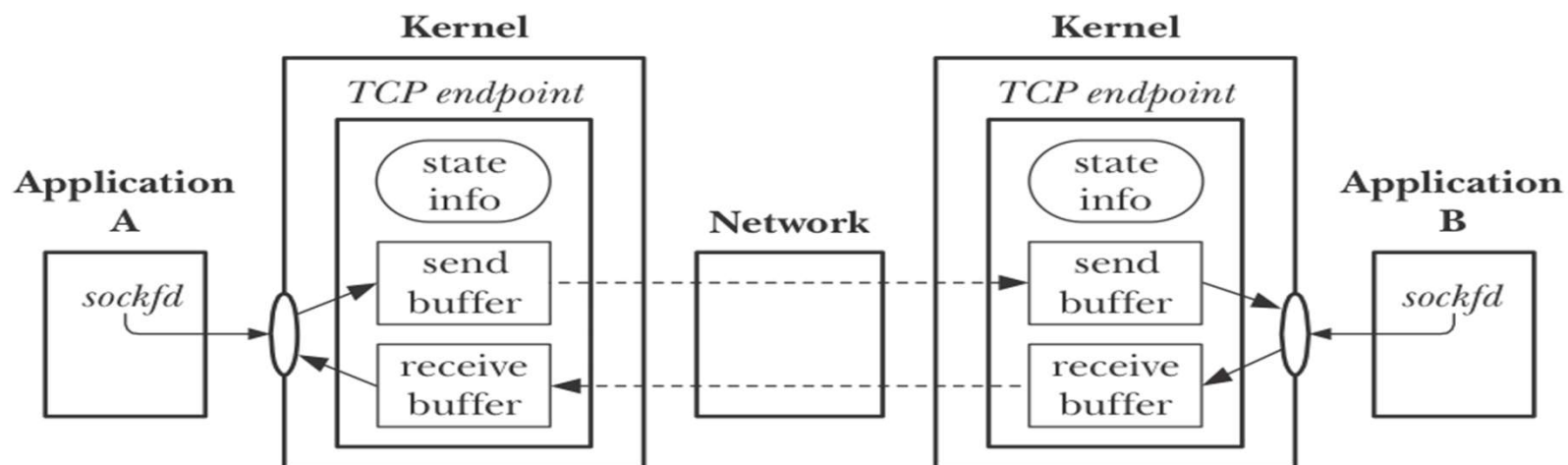
    int sock_client;

    sock_client = accept(sock_serveur, NULL, NULL);

    return 0;}
```

## Les I/O sur les sockets connectées

Une paire de sockets stream connectées fournit un canal de communication bidirectionnel entre le client et le serveur => chaque application peut lire ou écrire sur via sa socket.



## read()

```
#include <unistd.h>
```

```
ssize_t read(int sockfd, void *buf, size_t len);
```

.sockfd est le descripteur de la socket receptrice

.buf contient le message reçu

.len est la taille maximale du message reçu

.read() retourne :

- . le nombre d'octets lus,
- . 0 sur EOF (Deconnexion)
- . ou -1 en cas d'erreur.

.Si aucun message n'est disponible dans le buffer, read() se bloque par défaut.



## write()

```
#include <unistd.h>
```

```
ssize_t write(int sockfd, const void *buf, size_t len);
```

• sockfd est le descripteur de la socket émetrice

• buf contient le message à envoyer

• len est la taille du message à envoyer

• write() retourne -1 en cas d'échec, et le nombre d'octets envoyés en cas de succès.

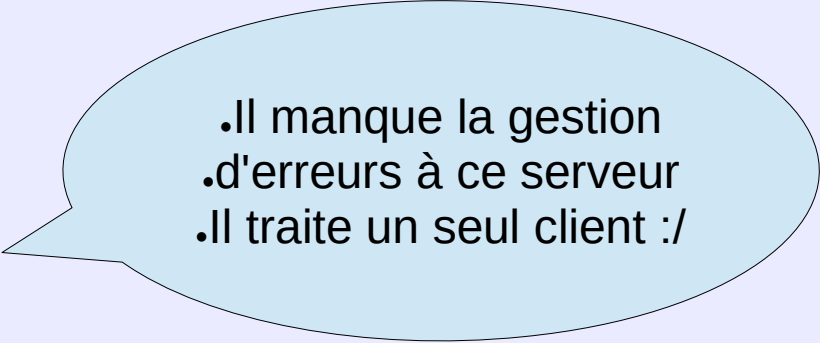
## D'autres fonctions pour l'échange de données

- Il existe des fonctions I/O spécifiques aux sockets.
- Pour envoyer des données, on peut utiliser aussi :
  - `send()`
  - `sendto()`
- Pour recevoir des données, on peut utiliser aussi :
  - `recv()`
  - `recvfrom()`
- Ces fonctions offrent plus d'options. Par exemple, elles permettent de manipuler des données urgentes.

```
int main(){
int sock_serveur = socket(AF_INET, SOCK_STREAM,0);
struct sockaddr_in sockaddr_serveur;
sockaddr_serveur.sin_family = AF_INET;
sockaddr_serveur.sin_port = htons (50013);
inet_aton("192.168.1.1", &sockaddr_serveur.sin_addr);
bind (sock_serveur, (struct sockaddr *) &sockaddr_serveur, sizeof(struct sockaddr_in ));
listen(sock_serveur, 128);
int sock_client ;
sock_client = accept(sock_serveur, NULL, NULL);

char *msg; time_t date ; date = time(NULL); msg = ctime(&date);
// msg contient la date et l'heure.
write(sock_client, msg, strlen(msg));
close(sock_client);
close(sock_serveur);

return 0 : }
```



.Il manque la gestion  
.d'erreurs à ce serveur  
.Il traite un seul client :/

## Serveur itératif / Serveur concurrent

• Un **serveur itératif** traite plusieurs clients mais un client à la fois. Cela convient quand les requêtes des clients peuvent être traitées rapidement.

• => Dans notre exemple de serveur daytime, il suffit de rajouter une boucle à partir de l'appel `accept()` jusqu'à la fermeture de la socket du client.

• Un **serveur concurrent** peut traiter plusieurs clients “en même temps”. Deux possibilités :

- créer un processus fils pour chaque client (via `fork()`)
- créer un thread pour chaque client (via `pthread_create()`).

## connect()

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

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

- connect() permet de connecter la socket sockfd à la socket désignée par l'adresse addr.
- L'argument addrlen indique la taille de addr.
- Elle retourne 0 en cas de succès, -1 en cas d'erreur.
- Comme bind(), elle utilise le type générique struct sockaddr => transtypage.

## close()

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

- Ferme la socket sockfd et libère ses ressources système.
- Retourne 0 en cas de succès, -1 en cas d'erreur.
- Une fois une socket est fermée :
  - si on tente de lire avec la socket qui lui était connectée , alors read() retourne 0 (après avoir récupéré toutes les données bufferisées).
  - Si on tente d'écrire, on recevra le signal SIGPIPE (souvent ignoré) et l'appel d'écriture échoue.

## Exercice

Ecrire un client daytime TCP en C.

## Recevoir un message en TCP

• En mode TCP, un message peut arriver chez le destinataire par plusieurs segments (morceaux) (même si l'émetteur a effectué un seul `write()`).

• => Un seul appel de `read()` récupère les données actuellement disponibles dans le buffer (mais pas les données qui arriveraient par la suite).

• Pour récupérer l'intégralité du message envoyé à une étape du dialogue, on fait une boucle de `read()` jusqu'à satisfaire une condition décrite par le protocole applicatif en question, par exemple rencontrer `'\n'`, EOF ou atteindre une certaine taille.