

Cours 1

Sockets TCP en C

Safa YAHY
safa.yahi@univ-amu.fr

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 la date, une page Web, envoyer un email, récupérer un email, envoyer un fichier, etc.

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.
- ♦ Dans ce cours, nous nous intéresserons au modèle client-serveur.

Ports de communication

- Une même machine (identifiée par une adresse IP) peut offrir plusieurs services à la fois (web, date, email, 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.

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 arrivée dans le bon ordre.

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'ordre d'arrivée n'est pas garanti, l'arrivée non plus.
- UDP plus rapide que TCP.
- UDP permet, et ce contrairement à TCP, des communications en multicast : envoyer un même message à un groupe de destinataire d'un seul coup.

Sockets

- ♦ Les **sockets** sont un mécanisme d'IPC permettant l'échange de données entre applications sur une même machine ou sur des machines distantes connectées par un réseau.
- ♦ Types de sockets :
 - ♦ Sockets du domaine UNIX
 - ♦ Sockets du domaine Internet (IPv4 et IPv6)
- ♦ La première implémentation de l'API des sockets remonte à 1983 avec 4.2BSD (pour Berkeley Software Distribution) qui est une version d'UNIX.

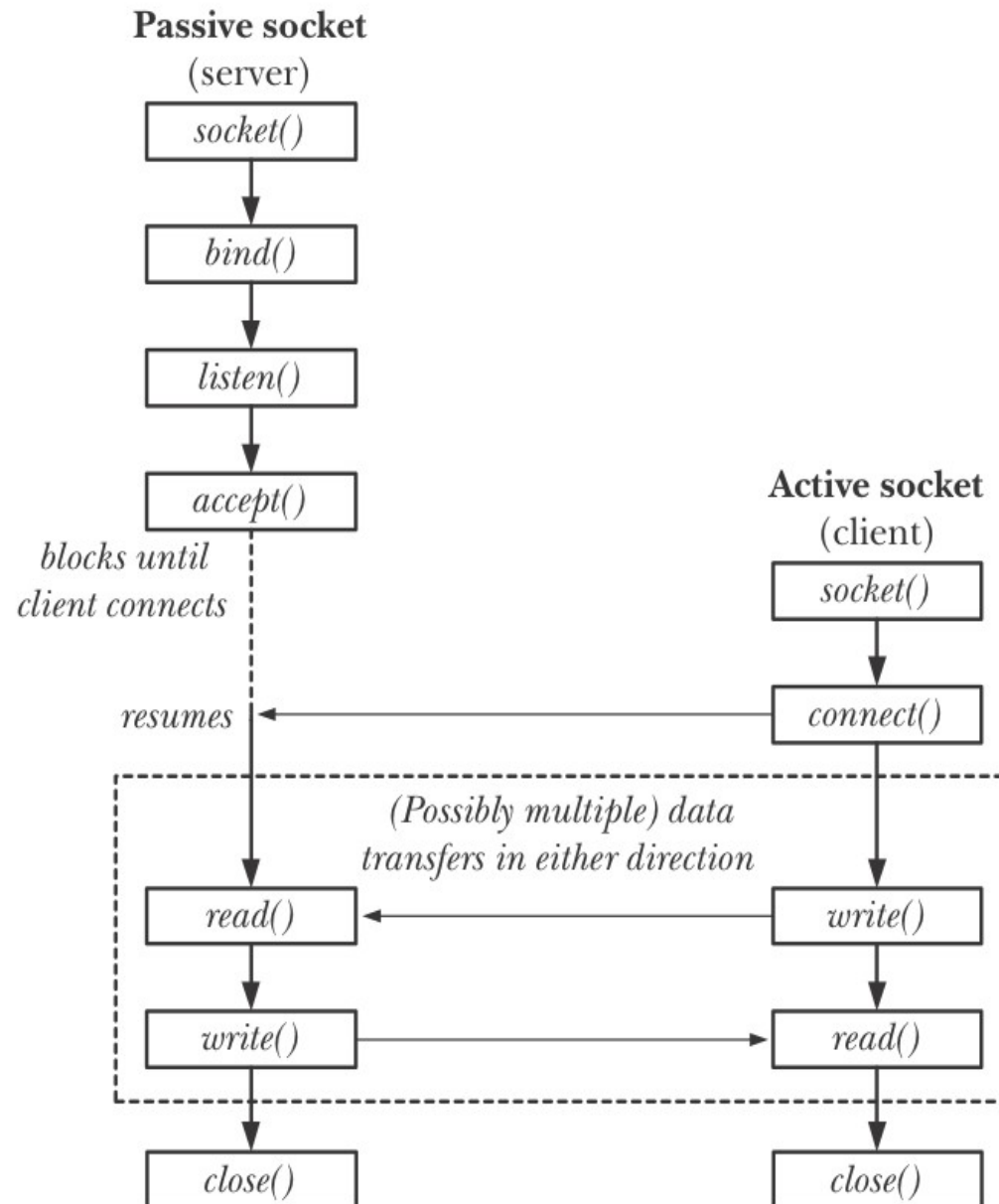
Sockets en mode TCP

- ♦ Le serveur crée une socket d'écoute 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 essaye de se connecter au serveur.
- ♦ Si le serveur accepte la connexion de ce client :
 - ♦ une nouvelle socket est créée au niveau du serveur
 - ♦ une nouvelle socket est créée au niveau client

Sockets en mode TCP

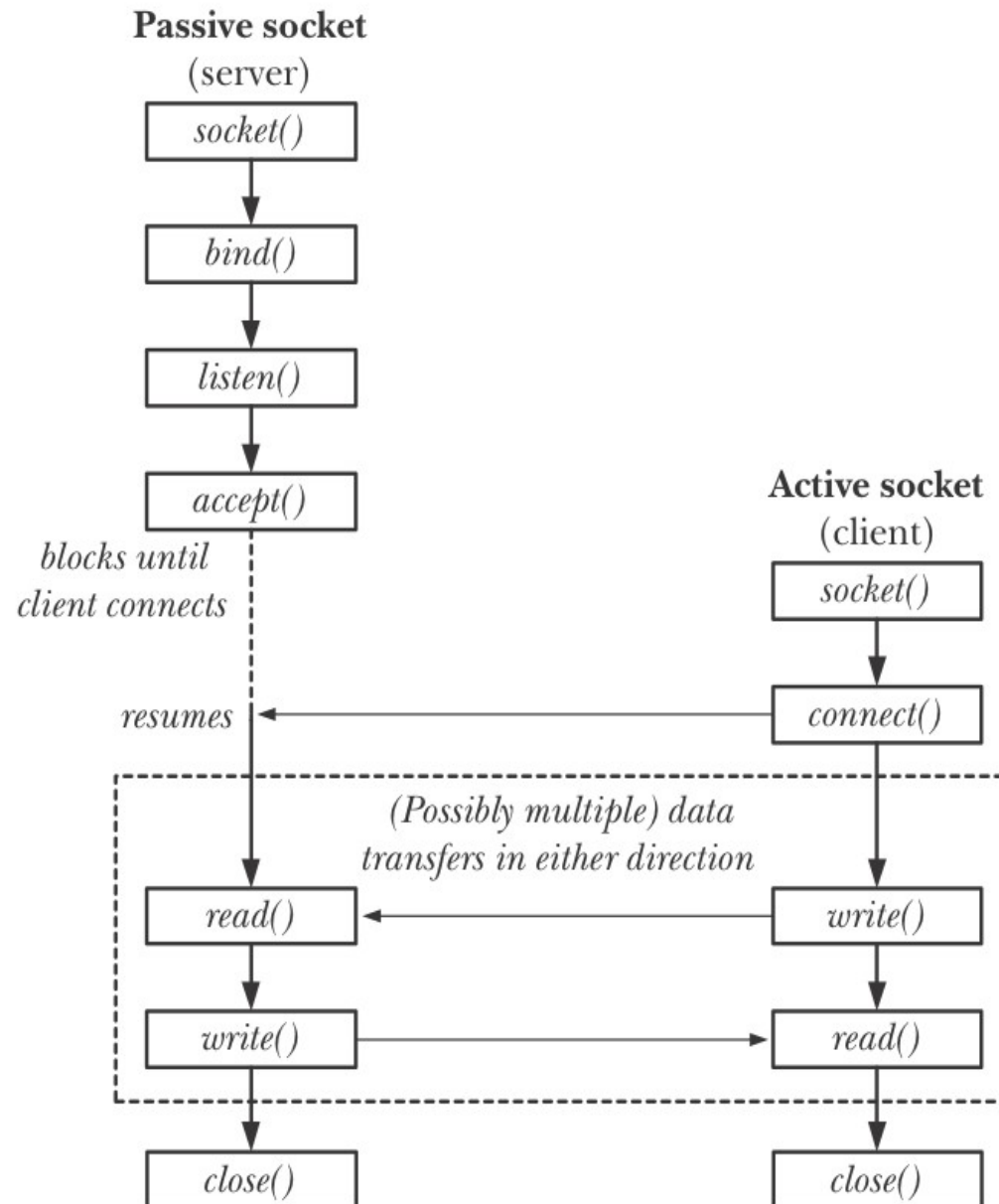
- La nouvelle socket du serveur est liée au même port local que sa socket d'écoute. Le point de terminaison (@ IP, port) distant de cette nouvelle socket comprend l'adresse et le port du client.
- La socket du client est liée localement à son adresse IP et un port généralement assigné par le système.
- 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.

Shéma général client / serveur TCP en C



A copier :)

Shéma général client / serveur TCP en C



A copier :)

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	struct sockaddr_un
AF_INET	adresse IPv4 de 32 bits + un port sur 16 bits	struct sockaddr_in
AF_INET6	adresse IPv6 sur 128 bits + un port sur 16 bits	struct sockaddr_in6

socket()

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

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

- Paramètre type : type de la socket

Exemple : 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.

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

- 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`

=> Ecrire un serveur daytime TCP en C qui écoute, par exemple, sur le port 50013 et l'adresse IP 192.168.1.76.
On suppose que la machine du serveur a pour adresse IP 192.168.1.76.

Exemple : serveur daytime TCP

```
#include <sys/socket.h>
int main()

{

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

// ...

return 0;
}
```

bind()

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

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

- bind() affecte l'adresse spécifiée par addr à la socket 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'echec.
- La fonction bind() (comme d'autres fonctions de socket) est commune à tous les domaines de communication des sockets => elle doit accepter les différents types d'adresses de sockets vus précédemment.

struct sockaddr : une structure d'adresse générique

- La structure **sockaddr** est définie par

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

- La structure sockaddr permet d'utiliser différentes structures d'adresse (via un transtypage).
- 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.

Structure sockaddr_in

La structure **sockaddr_in** est définie par

```
struct sockaddr_in
```

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

```
  in_port_t sin_port;      /* Numéro port */
```

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

```
  ...
```

```
};
```

in_port_t est un type d'entiers non signés de 16 bits.

Structure in_addr

La structure **in_addr** est définie par :

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

où in_addr_t est un type d'entiers non signés de 32 bits

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

- `in_addr myaddr;`
- `inet_aton("127.0.0.1", &myaddr);`
- `=> myaddr.s_addr` contient 00000001 00000000 00000000 01111111

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

- `in_addr myaddr;`
- `inet_aton("127.0.0.1", &myaddr);`
- `=> myaddr.s_addr` contient 00000001 00000000 00000000 01111111

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.`
- `=> buf contient : "127.0.0.1"`

Constantes d'adresses

- La constante **INADDR_ANY** représente l'adresse wildcard 0.0.0.0

sockaddr_server.sin_addr.s_addr = **INADDR_ANY**

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

- La constante **INADDR_LOOPBACK** représente l'adresse loopback (127.0.0.1).

Exemple : 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("192.168.1.76", &sockaddr_serveur.sin_addr);

bind(sock_serveur, (struct sockaddr *) &sockaddr_serveur, sizeof(struct
sockaddr_in ));
// ...
return 0;
}
```

Network byte order

- 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), l'octet de poids inférieur est rangé à l'adresse suivante et ainsi de suite.
 - **Little Endian** : on commence par l'octet de poids le plus faible et on le range en premier.
- Pour échanger des entiers (tels que les ports et les adresses IP) au sein d'un réseau, il faut utiliser le même ordre : il s'agit du **Network byte order** du protocole TCP/IP et qui correspond au Big Endian.

Network byte order

Plusieurs fonctions peuvent être utilisées pour convertir des entiers non signés de 16 ou 32 bits de l'ordre machine vers l'ordre réseau ou inversement :

```
#include <arpa/inet.h>
```

- `uint16_t htons(uint16_t hostshort);` // host to network “short”
- `uint32_t htonl(uint32_t hostlong);` // host to network “long”
- `uint32_t ntohl(uint32_t netlong);`
- `uint16_t ntohs(uint16_t netshort);`

listen()

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

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

- La fonction `listen()` met la socket sockfd en mode écoute: elle devient une socket passive et sera utilisée pour attendre des clients via `accept()`.
- backlog est la taille maximale de la file des connexions en attente. Par défaut, il vaut 128.

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 donné par la valeur de retour de 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 `accept()`, le serveur initialise le paramètre 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.
- Si on n'est pas intéressé explicitement par l'adresse de l'interlocuteur, `addr` et `addrlen` peuvent être initialisés à `NULL`.
- On pourra les récupérer si besoin par `getpeername()`.

Exemple : 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("192.168.1.76", &sockaddr_serveur.sin_addr);

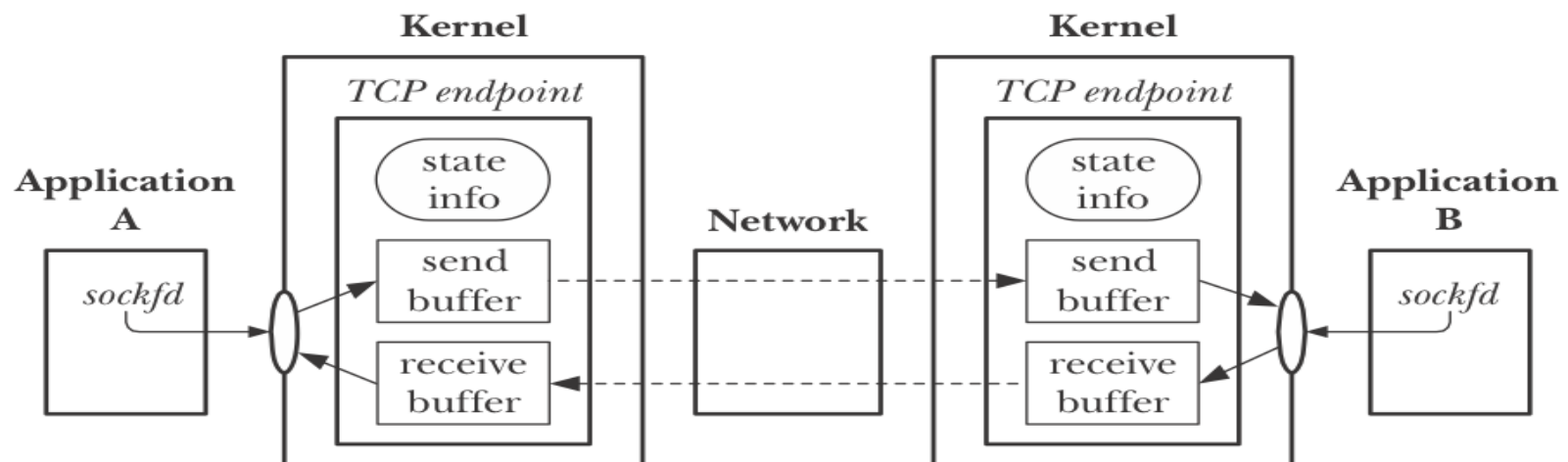
bind(sock_serveur, (struct sockaddr *) &sockaddr_serveur, sizeof(struct sockaddr_in));

listen(sock_serveur, 10);
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
 - 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.

Exemple : serveur daytime TCP (suite)

```
#include <sys/socket.h>
#include <arpa/inet.h>
#include <unistd.h>
#include <stdio.h>
#include <time.h>
#include <string.h>
```

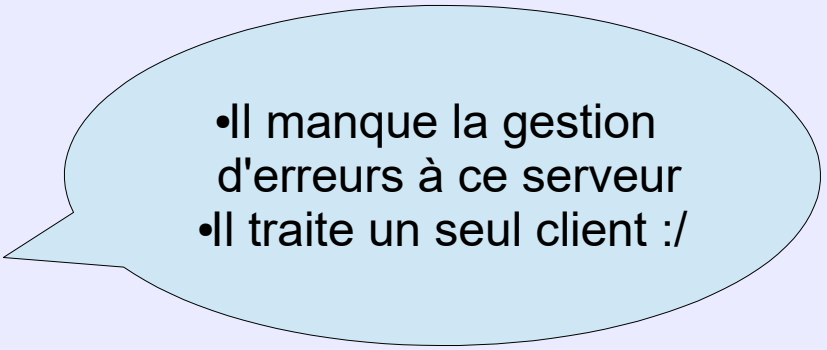
```
int main(){
int sock_serveur = socket(AF_INET, SOCK_STREAM,0);
struct sockaddr_in sockaddr_serveur;
sockaddr_serveur.sin_family = AF_INET;
sockaddr_serveur.sin_port = htons (50013);
inet_aton("192.168.1.76", &sockaddr_serveur.sin_addr);
bind (sock_serveur, (struct sockaddr *) &sockaddr_serveur, sizeof(struct sockaddr_in ));
listen(sock_serveur, 10);
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.

close()

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

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

- Permet de fermer une socket.
- Une fois, une socket fermé par une application, si l'autre applicaiton tente de lire via sa socket, alors elle rencontrera EOF (une fois elle reccupère toutes les données bufferisées). Si elle tente d'écrire, elle recevra le signal SIGPIPE (souvent ignoré) et l'appel d'écriture échoue.

Exercice 1

Ecrire un client daytime TCP en C.