# SAÉ C3-T3N
# Prog. Système & réseau
# Sockets

François Rousselle
Isabelle Delignières

# Contexte

- Cette SAÉ = compétence 3 : Administrer des systèmes informatiques communicants complexes
  → une note C3-PN dans la SAÉ S3.01 (moyenne de plusieurs notes)
- Certainement note C6 : travail en équipe projet
- Couvre les apprentissages critiques :
  - AC23.01 | Concevoir et développer des applications communicantes
  - AC23.02 | Utiliser des serveurs et des services réseaux virtualisés
- Développe la composante essentielle :
  - CE 1 : En maîtrisant l'architecture des systèmes et des réseaux ».

# SAÉ C3 ?

- SAÉ dev. S4 → version réseau du jeu vidéo
- SAÉ C3 = initiation communication client-serveur
- Sockets TCP/IP
- CM environ 1h :
  - intro. sockets
  - exemple socket en C
  - présentation SAÉ

# Langage

- Langage C obligatoire !
- Passage de paramètre par adresse
- Passage d’arguments en ligne de commande

# Intro. sockets

- Introduction = tout n'est pas vu, ni expliqué !
- Références :
  - Cours Dominique Dussart, R3.05 BUT2 APP
  - Manpages Linux section 2 « system calls » :
    - Commande : man 2 xxxxxxx
    - ou https://www.kernel.org/doc/man-pages/
  - Initiation à la prog. réseau sous Windows (Jessee Edouard)
  - Berkeley sockets (wikipedia)

# Définitions

- Wikipedia : « Berkeley Sockets Interface ou simplement sockets, est un ensemble normalisé de fonctions de communication lancé par l'université de Berkeley au début des années 1980 pour leur Berkeley Software Distribution (abr. BSD). »
- « 30 ans après son lancement, cette interface de programmation est proposée dans quasiment tous les langages de programmation populaires (Java, C#, C++, …). »
- Linux, MacOS : API socket
- Windows : API Winsock

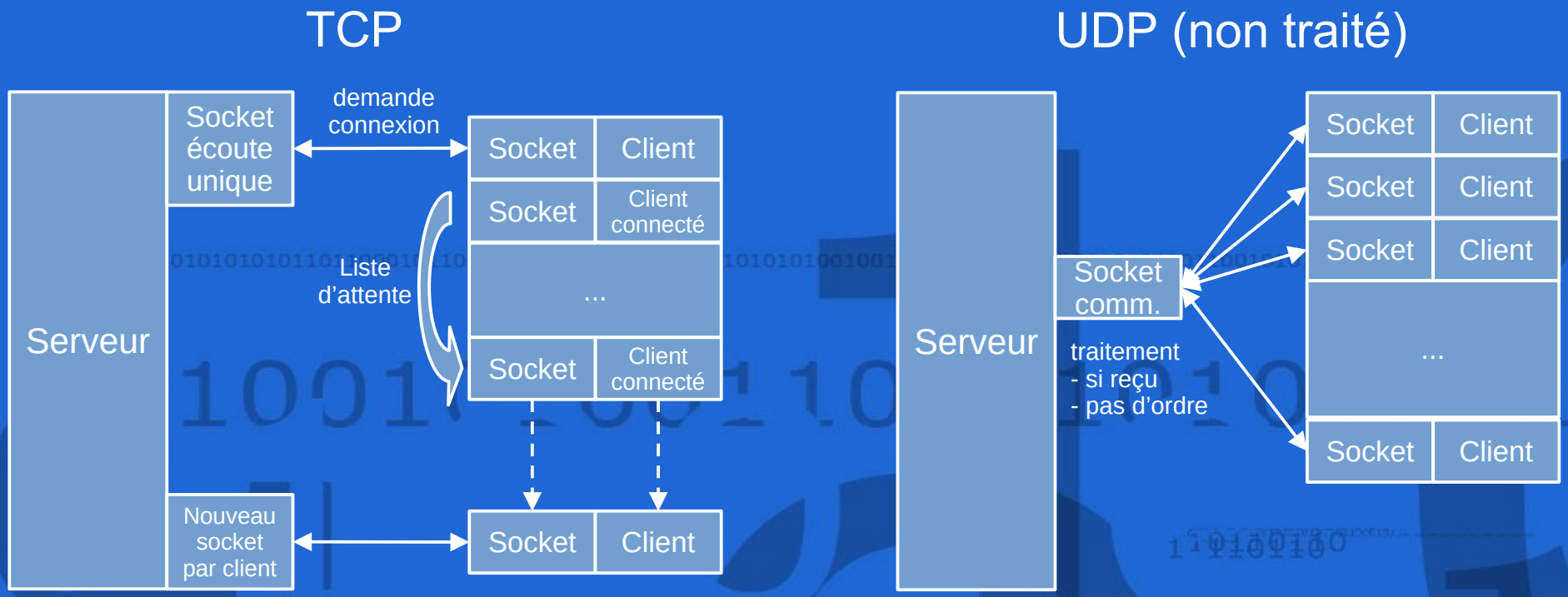
# Socket

- Point de communication bidirectionnel
- La communication peut être :
  - entre deux processus situés sur deux machines.
  - entre deux processus de la même machine
  - entre un processus et le noyau...
- Une connexion réseau :
  - 2 sockets = les deux points terminaux de la connexion

# Socket

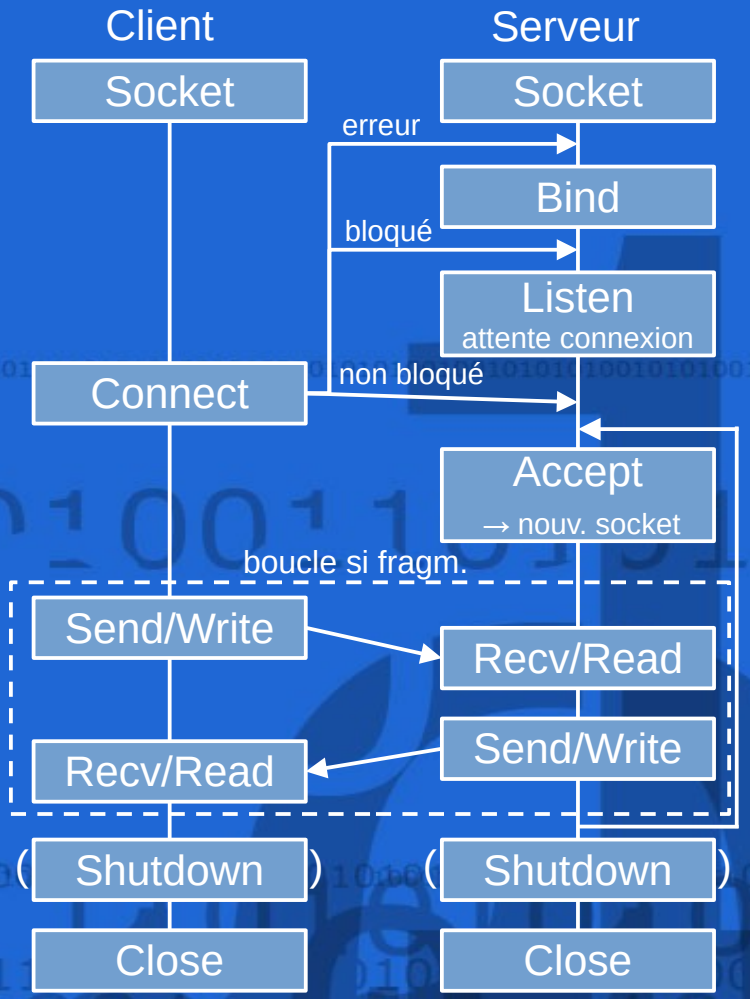
- Domaine du socket :
  - Unix = 2 processus sur le même OS
  - Internet = TCP/IP par exemple
  - ...
- Types du socket :
  - *stream* : avec connexion – sûre – possibilité de fragmentation des paquets (sur internet = TCP)
  - *datagram* : sans connexion – non sûre (sur internet = UDP)
  - ...

# Socket internet *stream/datagram*




- Traitement des clients en parallèle possible en forkant le serveur ou en utilisant la programmation multi-threadée

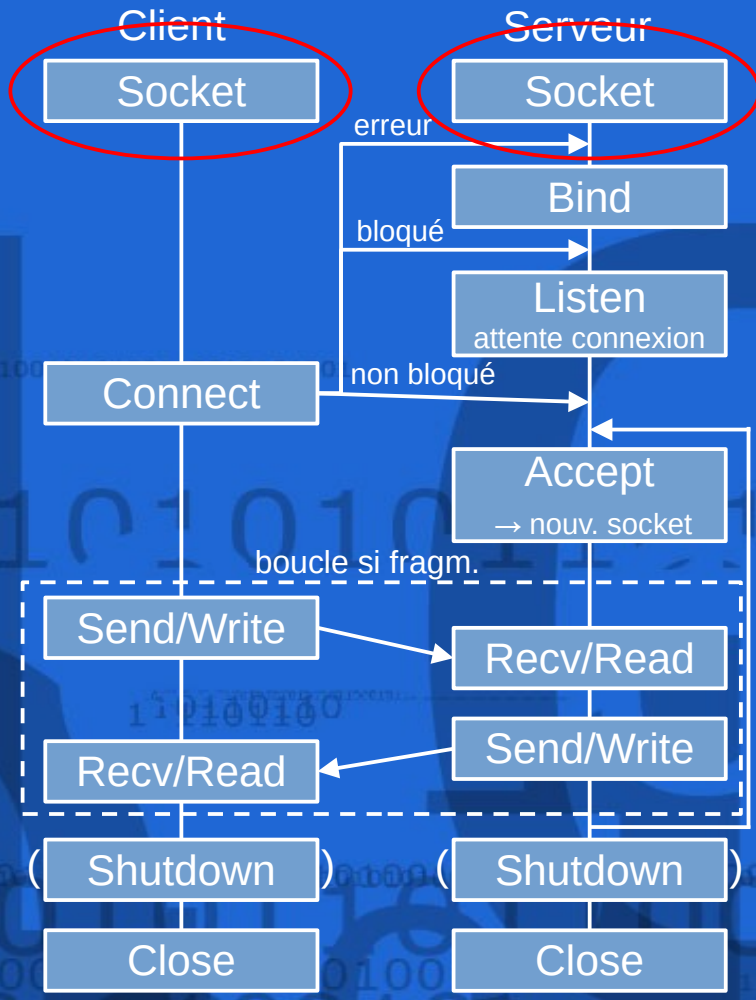
# Mise en œuvre TCP



# Mise en œuvre TCP

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

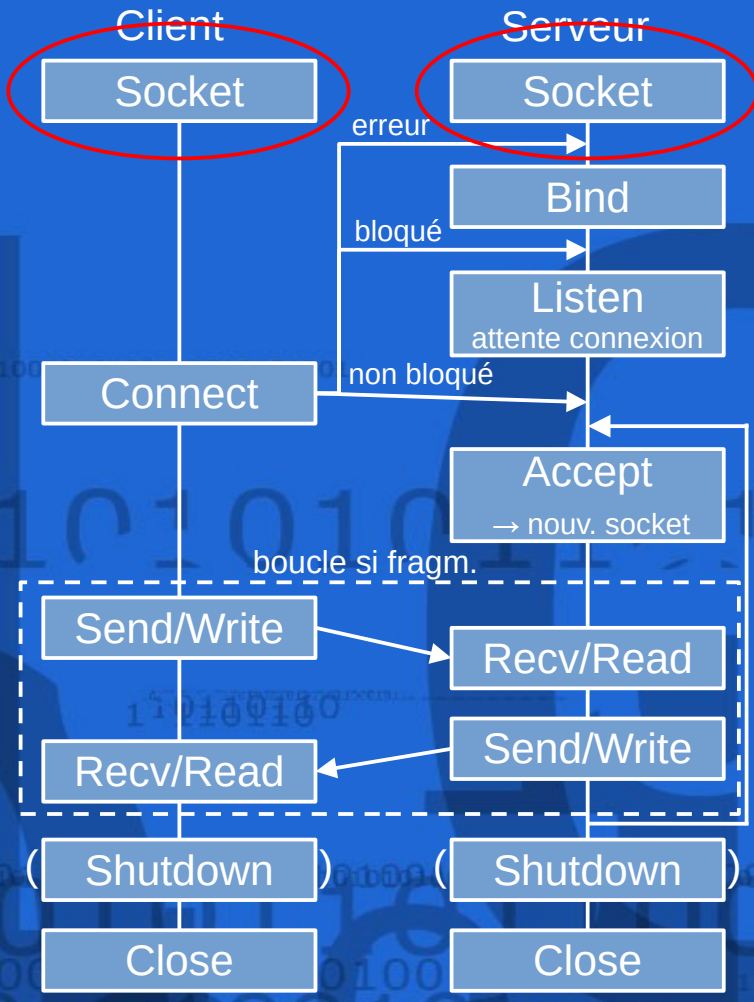
- Retourne un descripteur de fichier (-1 si erreur)
- Domaine : ("domaine de communication", familles de protocoles)
  - AF_UNIX : socket local
  - AF_INET : Internet IPV4
  - AF_INET6 : internet IPV6
  - ...



# Mise en œuvre TCP

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

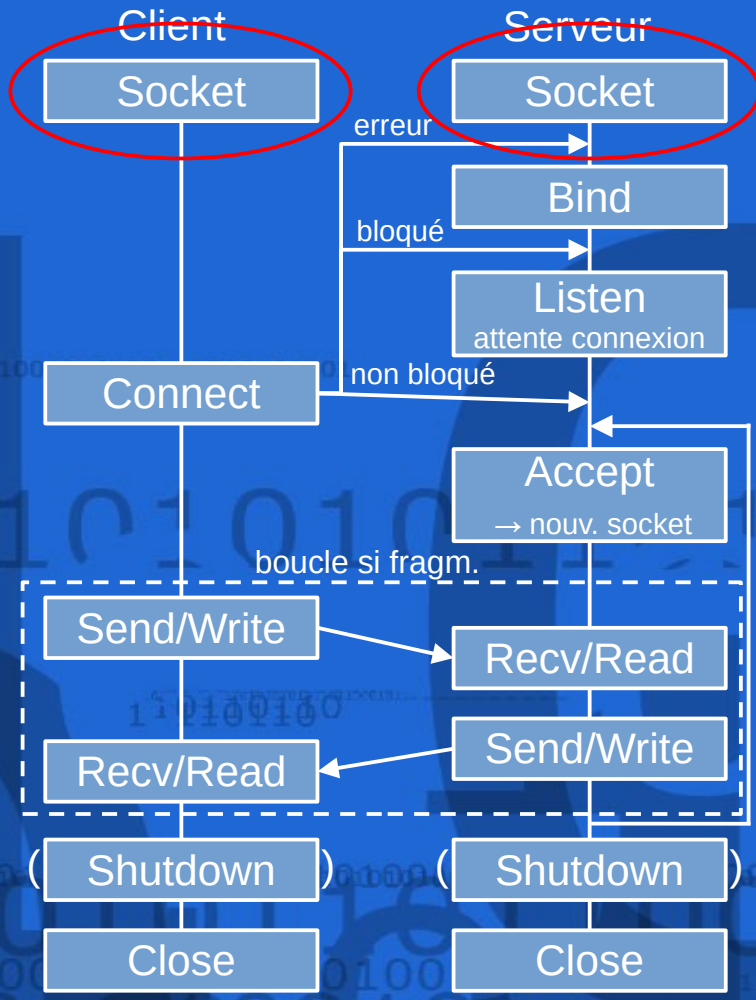
- Type :
  - SOCK_STREAM : par flot, connecté, bidirectionnel, fiable (TCP)
  - SOCK_DGRAM : par paquets, non connecté, non fiable (UDP)
  - ...
- Protocole :
  - Pour les domaines AF_INET ou AF_INET6, c'est le numéro du protocole dans le paquet IP (TCP=6)
  - Toutefois si un seul protocole existe dans le domaine/type (cas de AF_INET), alors protocole peut être à zéro



# Mise en œuvre TCP

- L'interface socket propose une structure d'adresse générique :

```
struct sockaddr{
    unsigned short int sa_family;
    unsigned char sa_data[14]; //depend de la famille
};
```

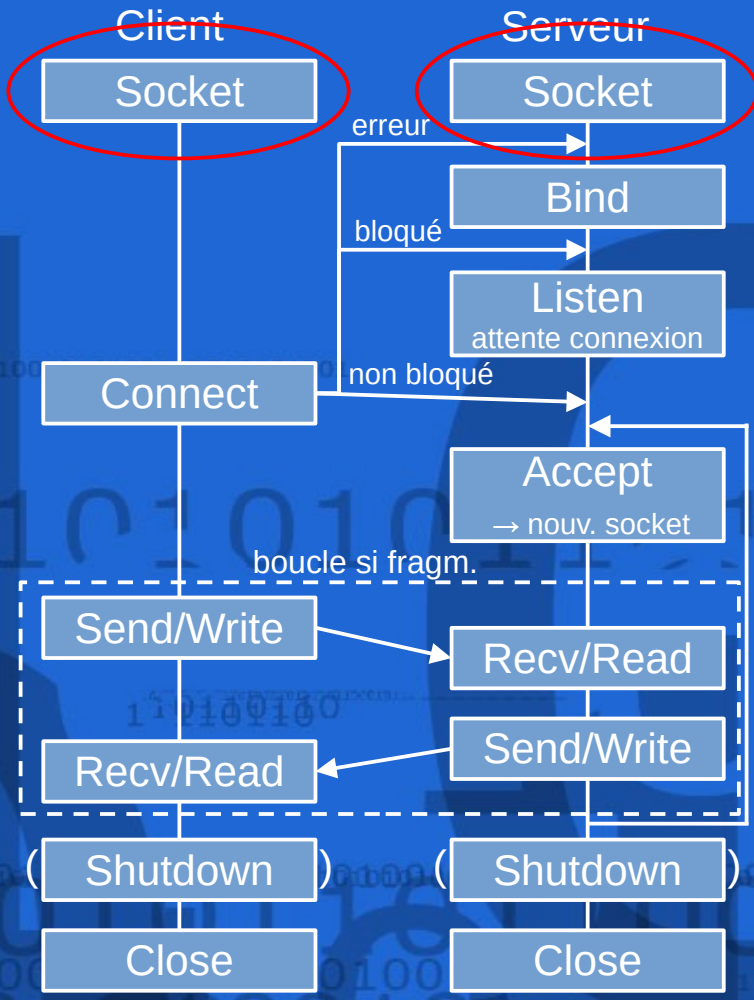


# Mise en œuvre TCP

- Le domaine AF_INET utilise une structure compatible :

```
struct in_addr { unsigned int s_addr; }; // une adresse Ipv4 (32 bits)
struct sockaddr_in {
    unsigned short int sin_family; // <- PF_INET
    unsigned short int sin_port; // <- numéro de port
    struct in_addr sin_addr; // <- adresse IPv4
    unsigned char sin_zero[8]; // ajustement
};
```

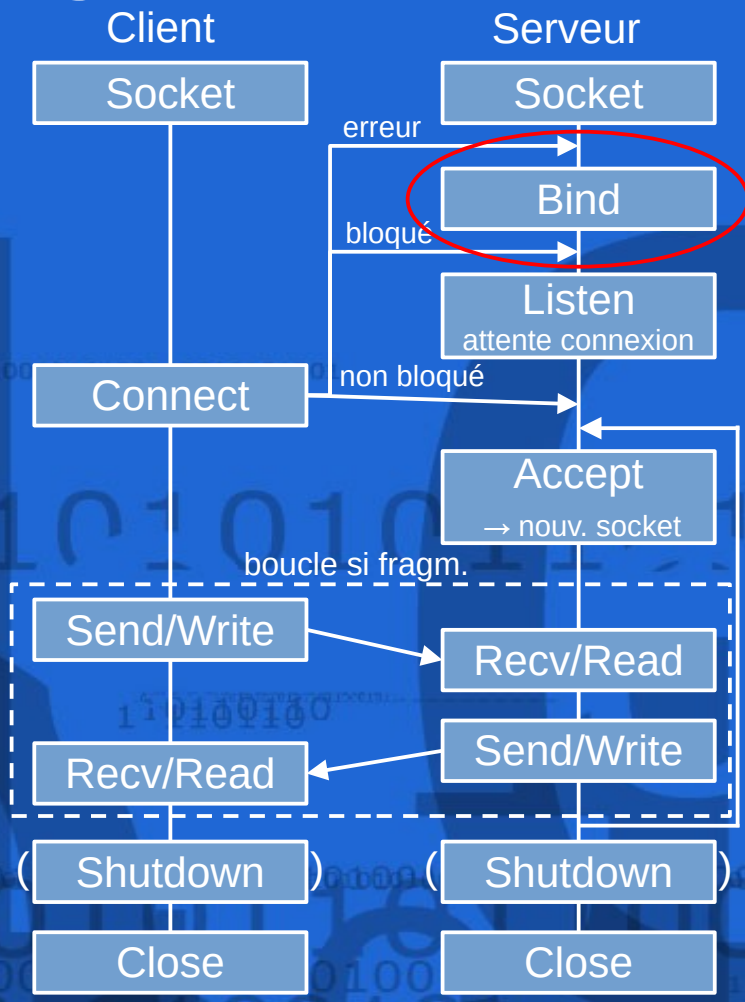
- Pour les informations dans la structure d'adresse, utilise :
  - inet_aton() pour convertir une @IPv4 décimale pointée vers une forme binaire
  - htons() pour convertir le numéro de port (little indian <->big indian)



# Mise en œuvre TCP

- Un serveur TCP attend des demandes de connexion en provenance de processus client.
- Le processus client doit connaître au moment de la connexion le numéro de port d'écoute du serveur.
- Le serveur utilise bind() pour lier son socket d'écoute à une interface et à un numéro de port.

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

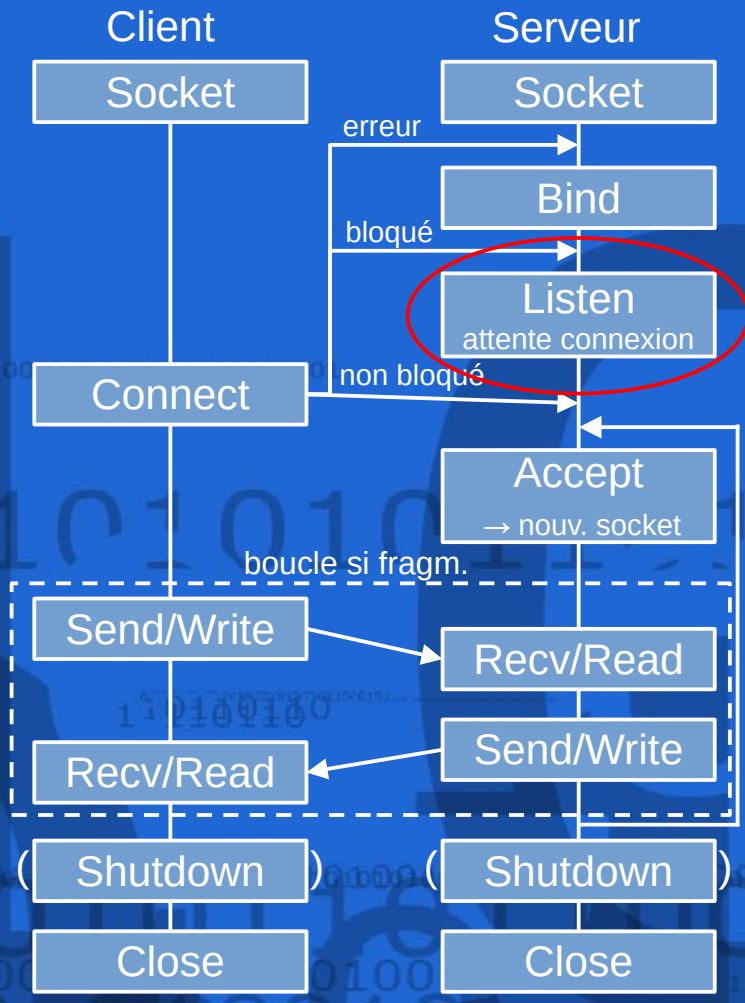


# Mise en œuvre TCP

- Il peut y avoir plusieurs connexions sur un même port : C'est la paire (adresse/port source,adresse/port destination) qui identifie une connexion
- Pour écouter un port :

  int listen(int fd, int backlog)

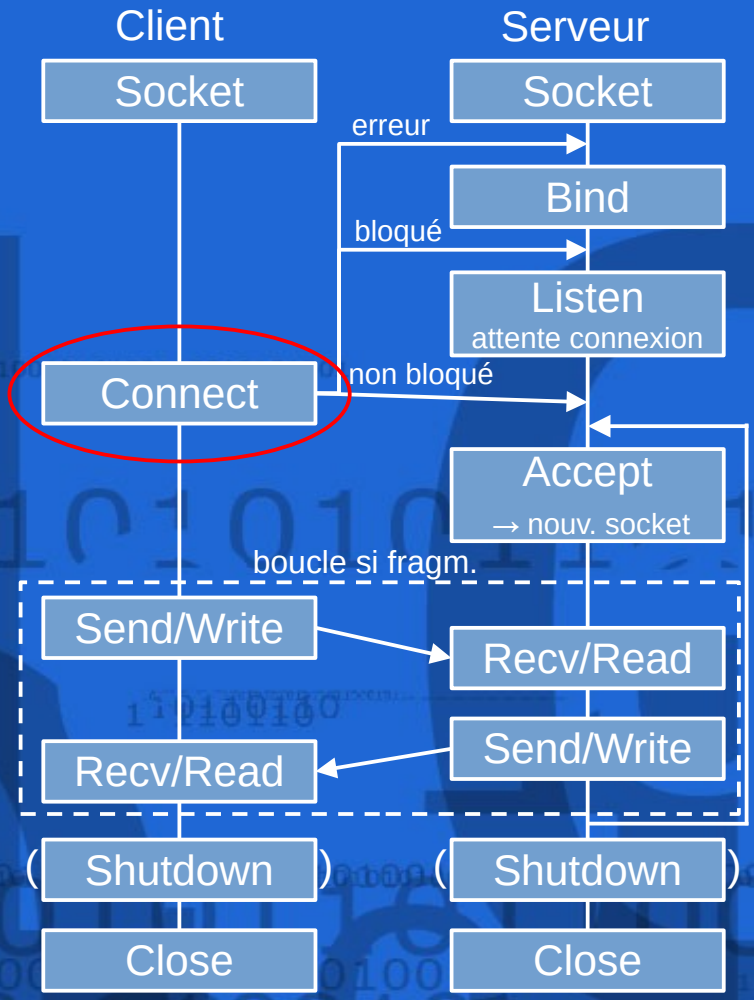
- fd : descripteur de fichier associé à une socket
- backlog : nombre maximum de connexions en attente



# Mise en œuvre TCP

int connect(int fd, struct sockaddr *addr, int addrlen)

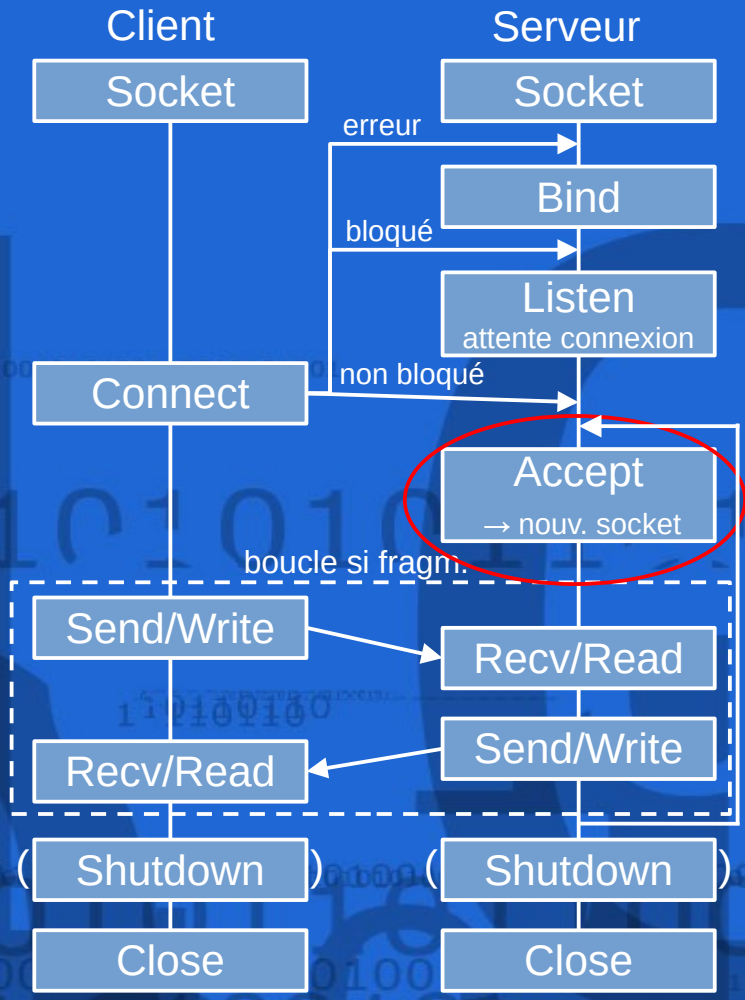
- fd : descripteur de fichier associé à une socket
- addr : pointeur sur une structure sockaddr_*, qui contient l'adresse et le port
- addrlen : taille de la structure addr
- renvoie 0 (OK), ou -1 (erreur)



# Mise en œuvre TCP

int accept(int fd, struct sockaddr *addr, int *addrlen)

- prend connaissance des nouvelles connexions
  - fd : descripteur de fichier associé à une socket
  - addr : pointeur vers une structure sockaddr_* où sera copié l'adresse du client
  - addrlen : est un pointeur sur un entier
    - appel : contient la taille maximum de la structure pointée par addr
    - retour : contiendra sa taille effective
- retourne un nouveau descr. de fichier ou -1 si erreur
  → pour effectuer les opérations read/write
- par défaut, accept est bloquant.

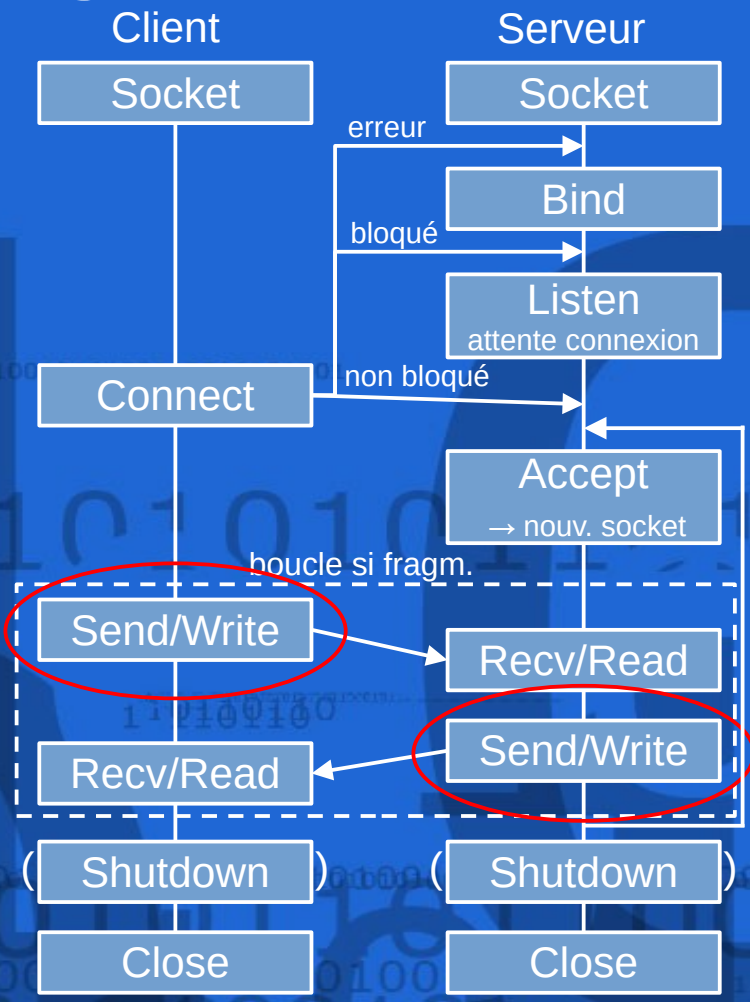


# Mise en œuvre TCP

- Envoi et lecture possible avec write et read.
- Mais aussi commandes spécifiques, avec options :

  int send(int fd, void *buffer, size_t len, int options)

- fd, buffer, len, retour : comme dans write
- options (non vues) :
  - MSG_MORE : "more to come". ne pas envoyer directement le paquet, attendre la suite.
  - MSG_OOB : Out-of-band (données "urgentes")
  - MSG_DONTWAIT : non bloquant
  - MSG_DONTROUTE : ne pas router le paquet
  - MSG_NOSIGNAL : pas de signal SIGPIPE si la connexion est fermée
  - MSG_CONFIRM
- write(fd,buff,len) est équivalent à send(fd,buff,len,0)

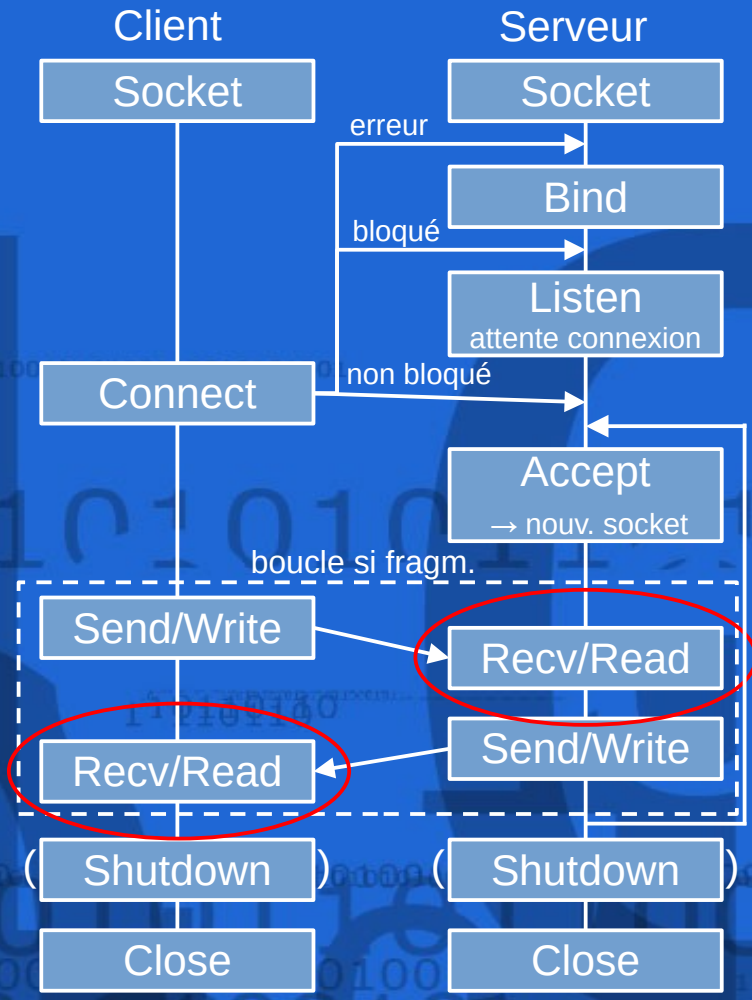


# Mise en œuvre TCP

- Lecture/réception :

  int recv(int fd, void *buffer, size_t len, int options)

- fd, buffer, len, retour : comme dans read
- Options :
  - MSG_PEEK : ne pas enlever les données du tampon de réception
  - MSG_OOB : récupère les données Out-of-band (données "urgentes")
  - MSG_ERRQUEUE : récupérer les données de la queue d'erreurs
  - MSG_DONTWAIT : non bloquant
- read(fd,buff,len) est équivalent à recv(fd,buff,len,0)

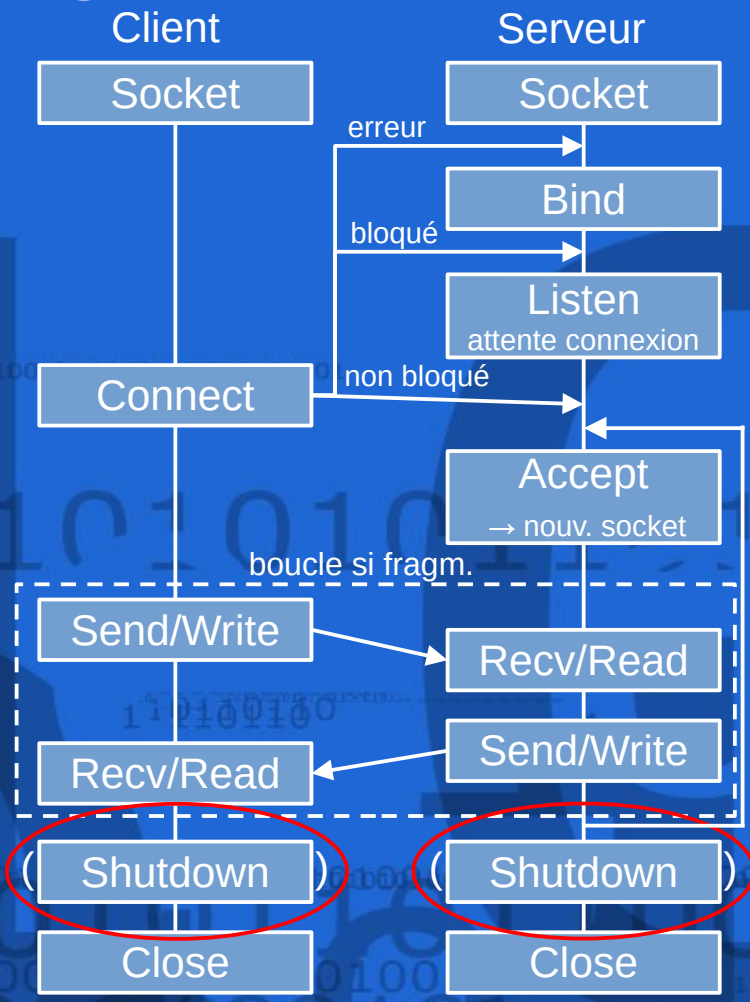


# Mise en œuvre TCP

- Les communications sur un socket peuvent être interrompues (optionnel) en appelant :

  int shutdown(int fd, int how)

- fd : descripteur de fichier associé à une socket
- how :
  - SHUT_RD : les prochaines lectures seront refusées
  - SHUT_WR : les prochaines écritures seront refusées
  - SHUT_RDWR : toutes les prochaines communications seront refusées
- renvoie 0 (OK), ou -1 (erreur)

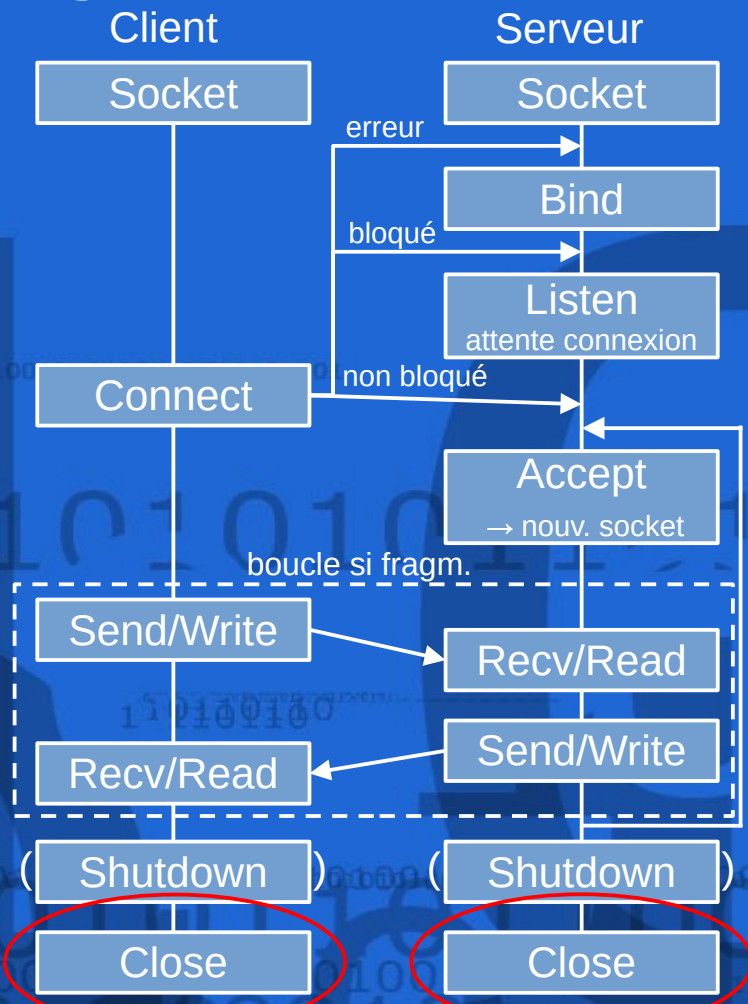


# Mise en œuvre TCP

- La fermeture du socket se fait en appelant :

  int close(int fd)

- fd : descripteur de fichier associé à une socket
- renvoie 0 (OK), ou -1 (erreur)

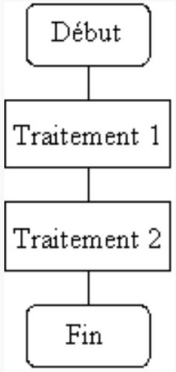
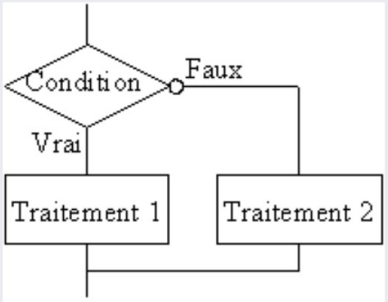
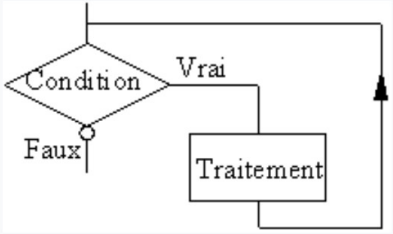
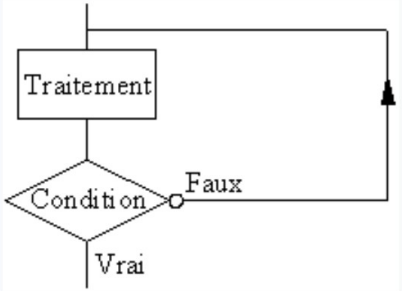


# Exemple socket en C

- client_base_tcp.c
- serveur_base_tcp.c

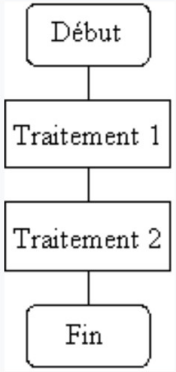
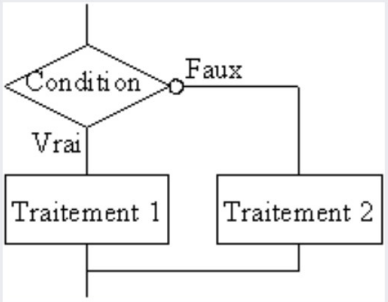
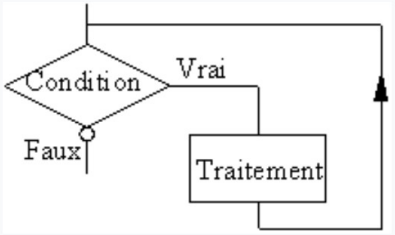
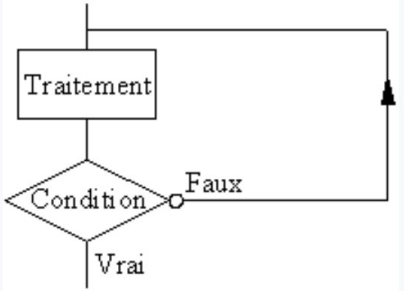
# Organigramme

- Organigramme des communications : demandé dans la SAÉ
- Respecte l'organigramme de programmation (voir wikipedia)

| **Séquence linéaire** | **Séquence alternative « si…alors…sinon »** | **Séquence répétitive « tant que…faire… »** | **Séquence répétitive « répéter…jusqu'à… »** |
|---|---|---|---|
|  |  |  |  |
| Début<br>• « Traitement 1 »<br>• « Traitement 2 »<br>Fin | Si « condition »<br>• alors « Traitement 1 »<br>• sinon « Traitement 2 »<br>Fin si | Tant que « condition »<br>• faire « traitement »<br>Fin tant que | Répéter « traitement »<br>jusqu'à « condition » |

# Organigramme

- Représenter ceux du client et du serveur côtes à côte
- Représenter les messages échangés dans chaque cas

| **Séquence linéaire** | **Séquence alternative « si…alors…sinon »** | **Séquence répétitive « tant que…faire… »** | **Séquence répétitive « répéter…jusqu'à… »** |
|---|---|---|---|
|  |  |  |  |
| Début<br>• « Traitement 1 »<br>• « Traitement 2 »<br>Fin | Si « condition »<br>• alors « Traitement 1 »<br>• sinon « Traitement 2 »<br>Fin si | Tant que « condition »<br>• faire « traitement »<br>Fin tant que | Répéter « traitement »<br>jusqu'à « condition » |

# TP

- Petit TP obligatoire avant la SAÉ :
  - Lire l'énoncé (sur Moodle)
  - Récupération du programme exemple sur Moodle
  - Lecture, tests, questions si nécessaire
  - Respecter l'organigramme des communications du programme à écrire
  - Programmez en partant de l'exemple

# Présentation SAÉ

- PN ?