Réseaux

Programmation Socket

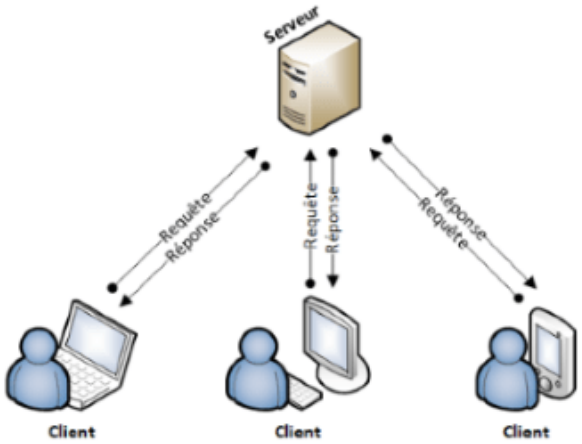
Hoai Minh LÊ

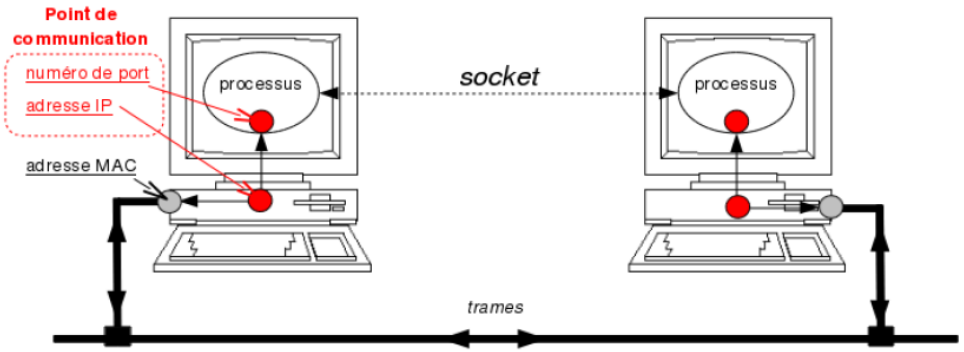
minh.le@univ-lorraine.fr

Programmation Socket

➢Architecture Client / Serveur

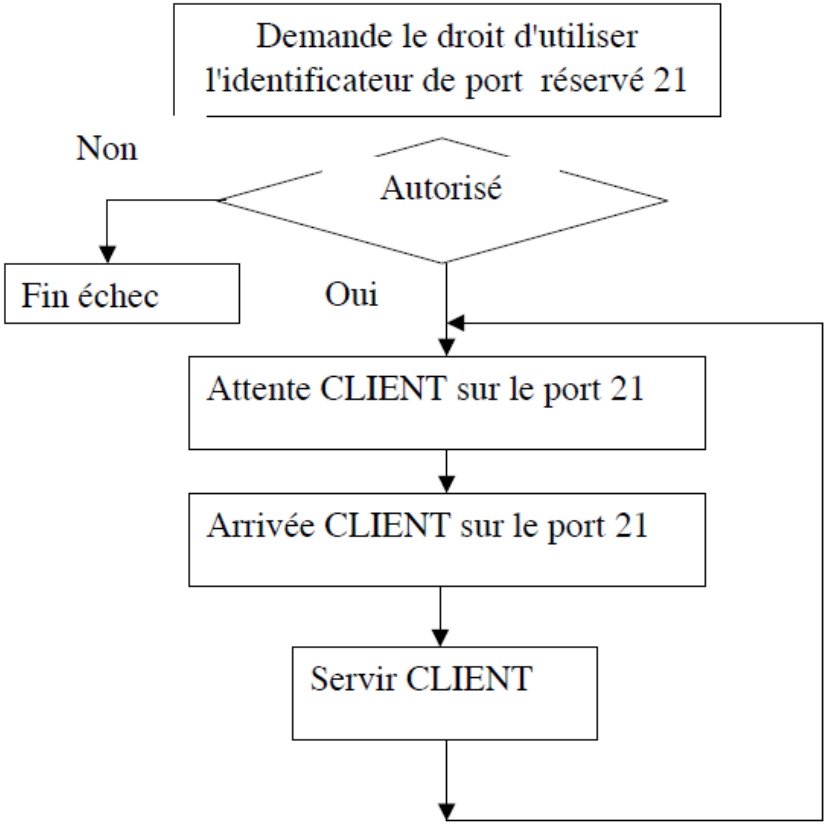
- ➢ Serveur : offre un service (en attente)
- ➢ Client : demandeur d'un service
- ➢ La communication s'initie **toujours** à la demande du client





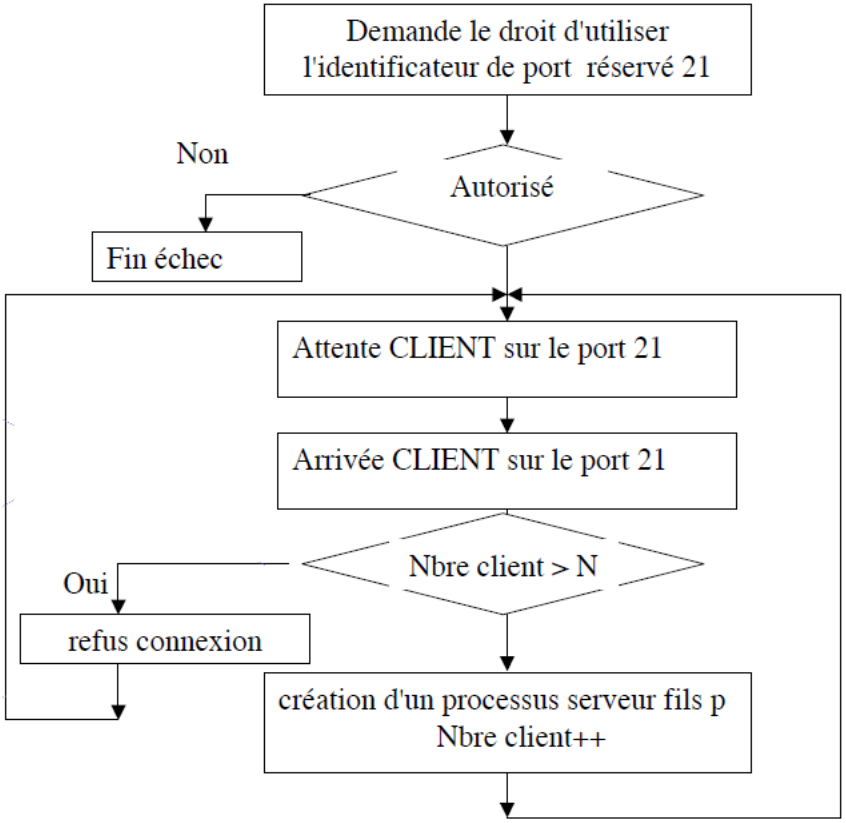
Programmation Socket

- Exemple serveur FTP pour un seul client



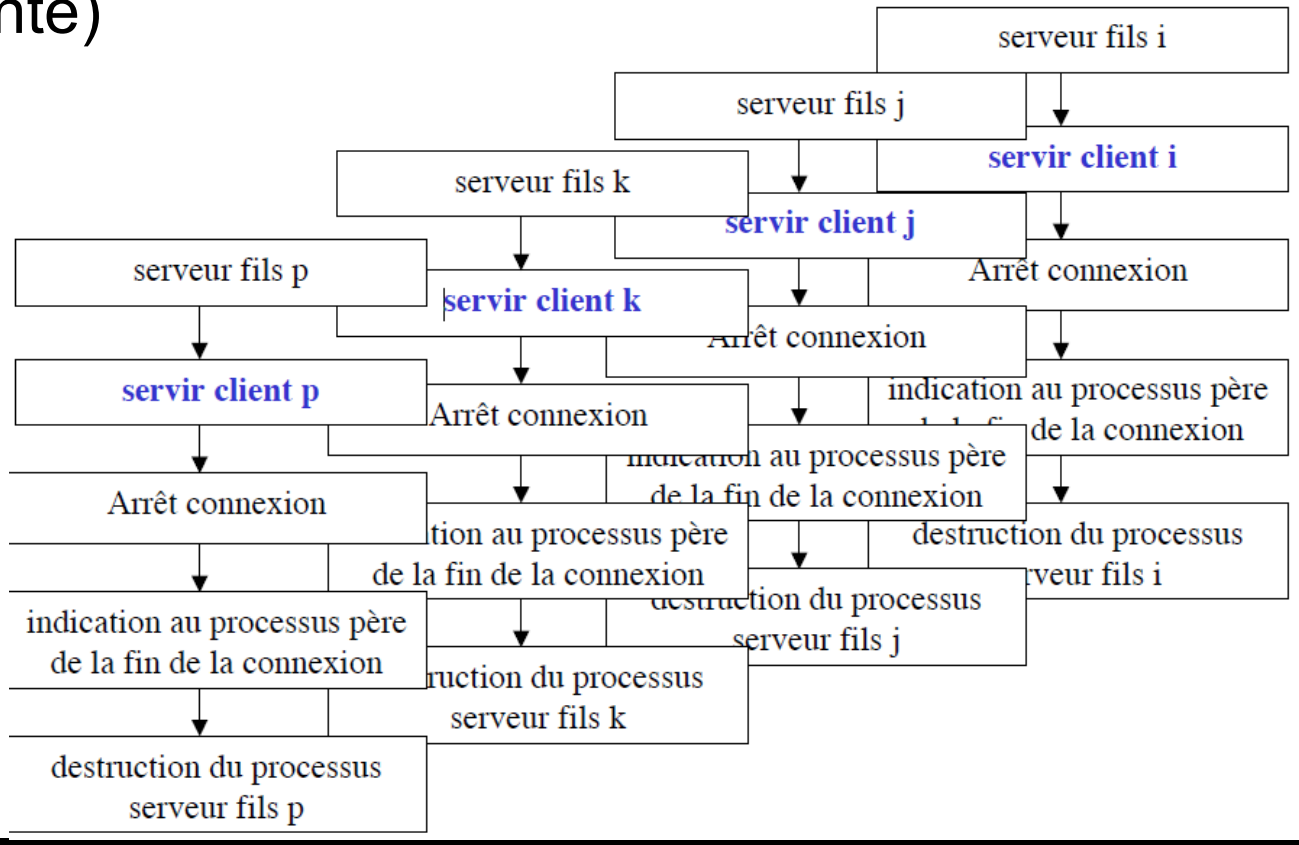
Programmation Socket

➢Exemple serveur FTP multi-clients (programmation concurrente)



Programmation Socket

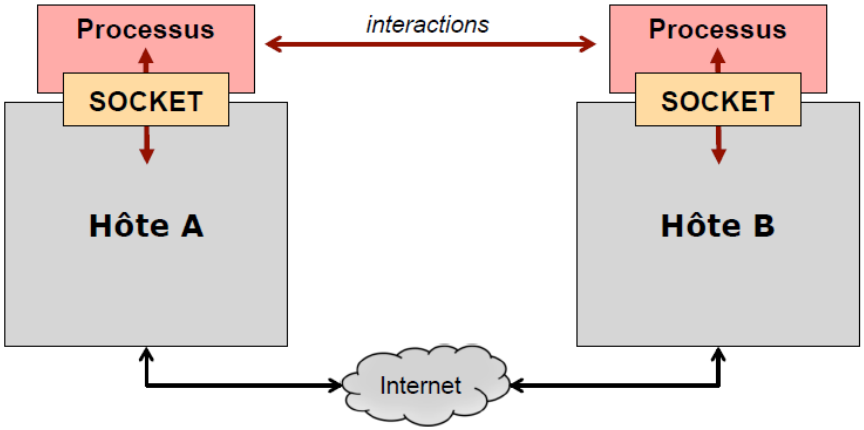
- Exemple serveur FTP multi-clients (programmation concurrente)



Programmation Socket

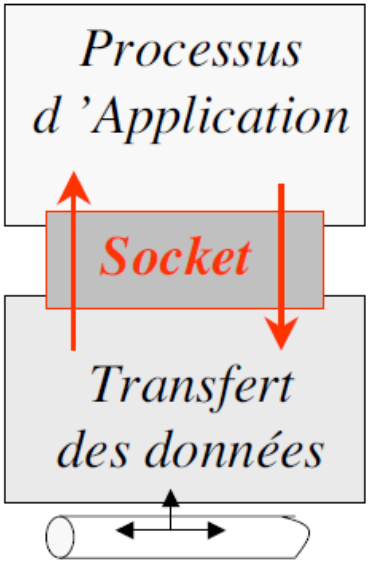
Socket : interface de communication bidirectionelle entre processus (locaux et/ou distants)

➔ Un modèle de communication inter processus (IPC - Inter Process Communication)



Programmation Socket

- Se situer entre la couche réseau et les couches applicatives du modèle OSI
- Une abstraction à travers laquelle une application peut envoyer et recevoir des données
- Exploiter les services d'un protocole réseau des couches basses du modèle OSI



Programmation Socket

- Socket a été introduit dans les systèmes UNIX (distributions de Berkeley) au début des années 1980
  - On parle parfois de sockets BSD (Berkeley Software Distribution)

- La programmation socket est maintenant répandue dans la plupart des systèmes d'exploitation
  - Socket BSD sous Unix/Linux
  - WinSocket sous Microsoft Windows

Programmation Socket

- Un socket s'utilise comme un fichier
  1. Création/Définition/Ouverture
     - réservation des ressources pour l ’interface de communication
     - création d’un descripteur du socket
  2. Communication
     - utilisation des primitives **read** (réception), **write** (émission)
  3. Fermeture/Libération
     - utilisation de la primitive **close** ou **shutdown**

Programmation Socket

- Création d'un socket

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

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

- **domain** : la famille de protocole à employer
  - **PF_INET** : protocoles Internet IPv4
  - **AF_INET6** : protocoles Internet IPv6
  - **PF_LOCAL** : communications UNIX en local
  - …

Programmation Socket

- Création d'un socket

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

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

- **type** : le type de socket à utiliser pour le dialogue (mode connecté, non connecté)
  - **SOCK_STREAM** : mode connecté (TCP)
  - **SOCK_DGRAM** : mode non-connecté (UDP)
  - **SOCK_RAW** : un accès direct aux protocoles de la couche réseau comme IP, ICMP, …

Programmation Socket

- Création d'un socket

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

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

- **protocol** : le numéro de protocole
  - La valeur **0** désigne le protocole par défaut
  - **IPPROTO_IP**, **IPPROTO_ICMP**, …

Toutes les combinaisons ne sont pas possibles

Programmation Socket

- Création d'un socket

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

int socket_tcp, socket_udp, socket_icmp; //descripteurs de sockets

// Un socket en mode connecte
socket_tcp = socket(PF_INET, SOCK_STREAM, 0); //par defaut TCP

// Un socket en mode non connecte
socket_udp = socket(PF_INET, SOCK_DGRAM, 0); //par defaut UDP

// Un socket en mode raw
socket_icmp = socket(PF_INET, SOCK_RAW, IPPROTO_ICMP); //on choisit ICMP
```

Programmation Socket

- Adressage du point de communication
  - Structure d'adresse générique **sockaddr**

```
struct sockaddr
{
        unsigned short int sa_family; // address family
        unsigned char sa_data[14];   //up to 14 bytes of direct address
};
```

  - **sockaddr_xxx** pour chaque famille de protocoles

Programmation Socket

- Adressage du point de communication
  - Structure d'adresse pour PF_INET (IPv4)

```
struct in_addr { unsigned int s_addr; }; // une adresse Ipv4 (32 bits)

struct sockaddr_in
{
        unsigned short int sin_family;  // PF_INET (IPv4)
        unsigned short int sin_port;    //  numero de port
        struct in_addr sin_addr;        // <- adresse IPv4
        unsigned char sin_zero[8];      // ajustement pour etre
                                           compatible avec sockaddr
};
```

Programmation Socket

- Adressage du point de communication
  - Structure d'adresse pour AF_INET6 (IPv6)

```
struct in6_addr {u_int8_t  s6_addr[16]; }  // IPv6 address

struct sockaddr_in6 {
        u_char          sin6_len;       // length of this structure
        u_char          sin6_family;    // AF_INET6
        u_int16m_t      sin6_port;      // Transport layer port #
        u_int32m_t      sin6_flowinfo; // IPv6 flow information
        struct in6_addr  sin6_addr;     // IPv6 address
  };
```

  - Structure **sockaddr_storage**

Programmation Socket

- Adressage du point de communication

```
struct sockaddr_in adresseDistante;
socklen_t longueurAdresse;

// Obtient la longueur en octets de la structure sockaddr_in
longueurAdresse = sizeof(adresseDistante);

// Initialise la structure sockaddr_in
memset(&adresseDistante, 0x00, longueurAdresse);

// Renseigne la structure sockaddr_in avec les informations du serveur distant
adresseDistante.sin_family = PF_INET;

// On choisit le numero de port d'ecoute du serveur
adresseDistante.sin_port = htons(5000); // = IPPORT_USERRESERVED

// On choisit l'adresse IPv4 du serveur
inet_aton("192.168.52.2", &adresseDistante.sin_addr);
```

Programmation Socket

- Boutisme (Endianness)
  - Certaines données telles que les nombres entiers peuvent être représentées sur plusieurs octets.
  - L'ordre dans lequel ces octets sont organisés en mémoire ou dans une communication est appelé boutisme (endianness)
  - Gros-boutisme: l'octet de poids le plus **fort** en premier
    - A0 B7 07 08 ➔ A0 B7 07 08
    - Exemple: processeurs Motorola, protocole TCP/IP
  - Petit-boutisme: l'octet de poids le plus **faible** en premier
    - A0 B7 07 08 ➔ 08 07 B7 A0
    - Exemple: processeurs x86

Programmation Socket

- Boutisme (Endianness)
  - Le choix du boutisme est typiquement fixé par l'architecture du processeur, ou par le format de données d'un fichier ou d'un protocole
  - L'ordre des octets du réseau (network) est en gros-boutisme (big-endian)
  - L'ordre des octets du hôte peut être différent (e.g. petit-boutisme)

Programmation Socket

- Boutisme (Endianness)

  - unsigned long int **htonl**(unsigned long int hostlong); // host to network (long)
  - unsigned short int **htons**(unsigned short int hostshort); // host to network (short)
  - unsigned long int **ntohl**(unsigned long int netlong); // network to host (long)
  - unsigned short int **ntohs**(unsigned short int netshort); // network to host (short)

Programmation Socket

- Fonctions de service

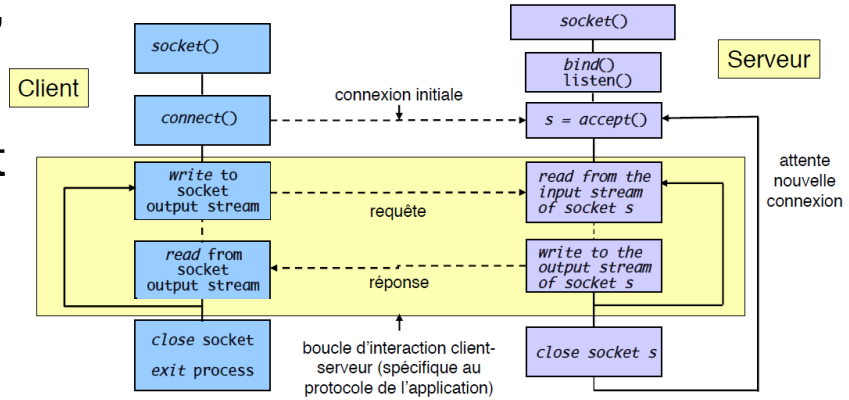
  - **inet_aton()** : convertit une adresse IPv4 en décimal pointé vers sa forme binaire 32 bits dans l'ordre d'octet du réseau
  - **inet_ntoa()**: convertit une adresse IPv4 sous sa forme binaire 32 bits dans l'ordre d'octet du réseau vers une chaîne de caractères en décimal pointé
  - **getaddrinfo()**: convertir les informations d'hôte réseau en adresse IP
  - **getnameinfo()**: récupérer les noms d'hôte pour les adresses IP correspondantes

Programmation Socket

- Fonctions de service
  - **gethostbyname(), gethostbyaddr()**: obtenir des informations (adresse IP, nom) sur la machine
  - **getservbyname()**: obtenir le numéro de port à partir du nom de service
  - **getprotobyname()**: obtenir le numéro de protocole à partir du nom
  - **getsockname(), getpeername()**: otenir des informations sur le socket

Programmation Socket

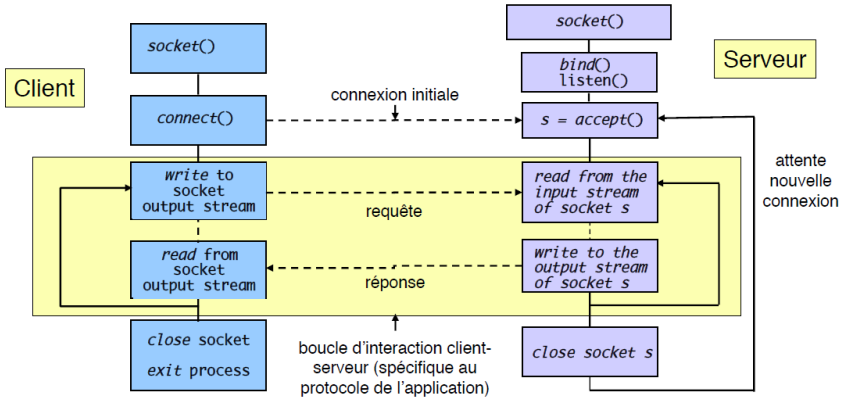
- Communication TCP - Serveur
  1. Création socket avec **socket()**
  2. Création la socket d'écoute
     - Initialiser une adresse ip:port, e.g. **sockaddr_in**
     - associer l'adresse a la socket avec **bind()**
  3. Se mettre en attente des connections avec **listen()**
  4. Accepter la connection avec **accept()**



Programmation Socket

- Communication TCP - Serveur

5. (répétition) Envoi et réception de données (**send()** et **recv**()) entre le client et le serveur
6. Fermer le socket avec **close()**

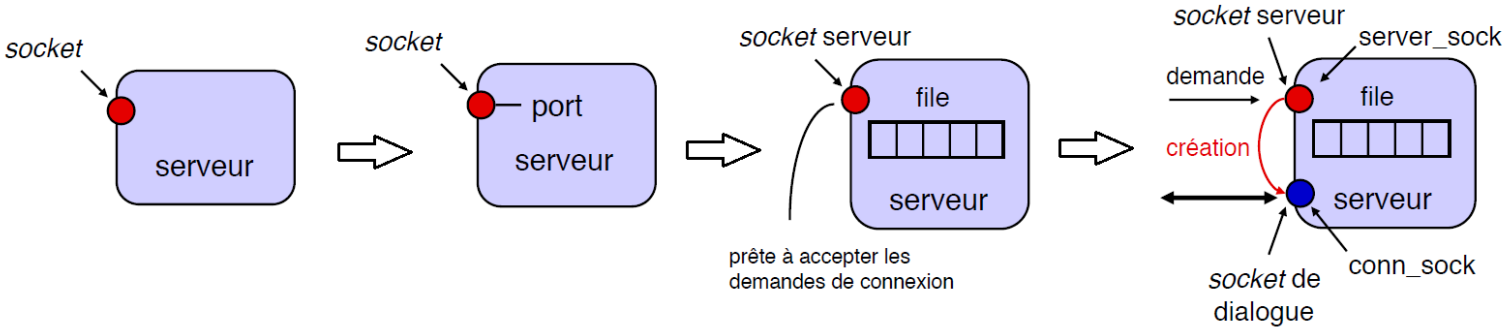


Programmation Socket

➢ Communication TCP – Serveur

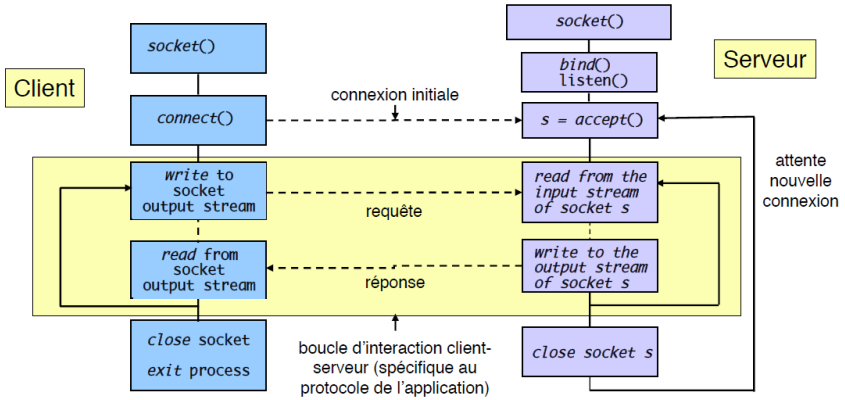
- ➢ Les fonctions d'attente et de traitement sont séparées, pour permettre au serveur d'attendre de nouvelles demandes de connexion pendant qu'il traite des requêtes en cours

- ➢ La socket de communication est associée au même numéro de port que la socket serveur, mais avec un descripteur différent



Programmation Socket

- Communication TCP - Client
  1. Création socket avec **socket()**
  2. Connecter au serveur
     - Récupérer et initialiser l'adresse du serveur (**gethostbyname** ou **getaddrinfo**)
     - Connexion avec **connect()**
  3. (répétition) Envoi et réception de données (**send()** et **recv**())
  4. Fermer le socket avec **close()**



Programmation Socket

- Communication TCP – Echange
  - Une fois la connexion établie, le client et le serveur disposent chacun d'un descripteur (pseudo-fichier) vers l'extrémité correspondante de la connexion
  - À partir de chaque extrémité, on peut créer 2 flux pour interagir avec l'interlocuteur distant: un flux d'entrée (en lecture) et un flux de sortie (en écriture)
  - Les flux d'entrée et de sortie sont en mode FIFO
  - Une lecture sur un flux d'entrée peut être bloquante (si le tampon de réception ne contient pas de nouvelles données)
    - jusqu'à l'arrivée de nouvelles données
    - ou jusqu'à la rupture de la connexion réseau

Programmation Socket

➢Communication TCP – Mode bloquant et non-bloquant

- ➢ Certaines fonctions sont « bloquantes **»: accept()**, **recv()**, **recvfrom()**, …
- ➢ C'est le choix par défaut fait par l'OS au moment de la création de la socket
- ➢ Mode non-bloquant avec **fcntl()** ➔ l'interroger périodiquement

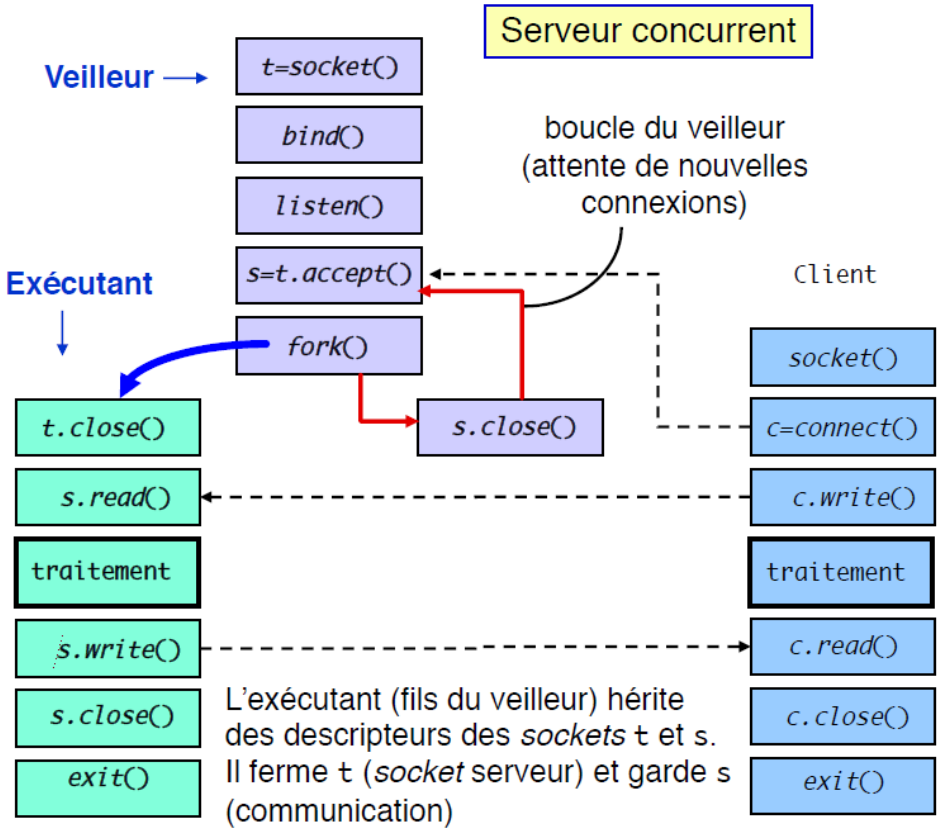
```
#include <unistd.h>
#include <fcntl.h>
...

int s = socket(AF_INET, SOCK_STREAM, 0);
fcntl(s, F_SETFL, O_NONBLOCK);
```

Programmation Socket

- Communication TCP – Mode concurrent
  - Pour réaliser un serveur en mode concurrent, une solution consiste à créer **un nouveau flot d'exécution** pour servir chaque demande de connexion
    - Il y a un flot d'exécution principal (appelé veilleur) qui attend sur **accept()**
    - Lorsqu'il reçoit une demande de connexion, le veilleur crée un nouveau flot (appelé exécutant) qui va interagir avec le nouveau client
    - Après la création de l'exécutant, le veilleur revient se mettre en attente sur **accept()**
    - Lorsqu'un exécutant a fini de dialoguer avec un client, il se termine avec **close()**

Programmation Socket

- Communication TCP – Mode concurrent



Programmation Socket

- Communication TCP – Mode concurrent

  - Utiliser d'un ensemble pré-alloué de flots exécutants afin d'éviter les créations systématiques
    - Un exécutant peut être « ré-utilisé » pour traiter successivement plusieurs clients

  - Utilisation de mécanismes qui permettent à un même flot d'exécution de gérer de manière concurrente plusieurs canaux de communication ➔ programmation événementielle (event-driven programming)

Programmation Socket

- Exemple « Echo Serveur » en TCP

```
int main(int argc, char *argv[]) {
        // port to start the server on
        int SERVER_PORT = 8877;

        // socket address used for the server
        struct sockaddr_in server_address;
        memset(&server_address, 0, sizeof(server_address));
        server_address.sin_family = AF_INET;

        // htons: host to network short: transforms a value in host byte
        // ordering format to a short value in network byte ordering format
        server_address.sin_port = htons(SERVER_PORT); //
        htonl: host to network long: same as htons but to long
        server_address.sin_addr.s_addr = htonl(INADDR_ANY);
```

Programmation Socket

```
// create a TCP socket, creation returns -1 on failure
int listen_sock;
if ((listen_sock = socket(PF_INET, SOCK_STREAM, 0)) < 0) {
            printf("could not create listen socket\n");
            return 1;
}

// bind it to listen to the incoming connections on the created server
// address, will return -1 on error
if ((bind(listen_sock, (struct sockaddr *)&server_address,
      sizeof(server_address))) < 0) {
            printf("could not bind socket\n");
            return 1;
}
```

Programmation Socket

```
int wait_size = 16;  // maximum number of waiting clients, after which
                     // dropping begins
if (listen(listen_sock, wait_size) < 0) {
                printf("could not open socket for listening\n");
                return 1;
}

// socket address used to store client address
struct sockaddr_in client_address;
int client_address_len = 0;
```

Programmation Socket

```
// run indefinitely
while (true) {
        // open a new socket to transmit data per connection
        int sock;
        if ((sock =
            accept(listen_sock, (struct sockaddr *)&client_address,
                &client_address_len)) < 0) {
                printf("could not open a socket to accept data\n");
                return 1;
        }

        int n = 0;
        int len = 0, maxlen = 100;
        char buffer[maxlen];
        char *pbuffer = buffer;
```

Programmation Socket

```
                printf("client connected with ip address: %s\n",
                      inet_ntoa(client_address.sin_addr));
                // keep running as long as the client keeps the connection open
                while ((n = recv(sock, pbuffer, maxlen, 0)) > 0) {
                        pbuffer += n;
                        maxlen -= n;
                        len += n;
                        printf("received: '%s'\n", buffer);
                        // echo received content back
                        send(sock, buffer, len, 0);
                }
                close(sock);
        }
         close(listen_sock);
        return 0;
}
```

Programmation Socket

- Exemple « Echo Client » en TCP

```
int main() {
        const char* server_name = “localhost”;
        const int server_port = 8877;

        struct sockaddr_in server_address;
        memset(&server_address, 0, sizeof(server_address));
        server_address.sin_family = AF_INET;

        // open a stream socket
        int sock;
                if ((sock = socket(PF_INET, SOCK_STREAM, 0)) < 0) {
                printf("could not create socket\n");
                return 1;
        }
```

Programmation Socket

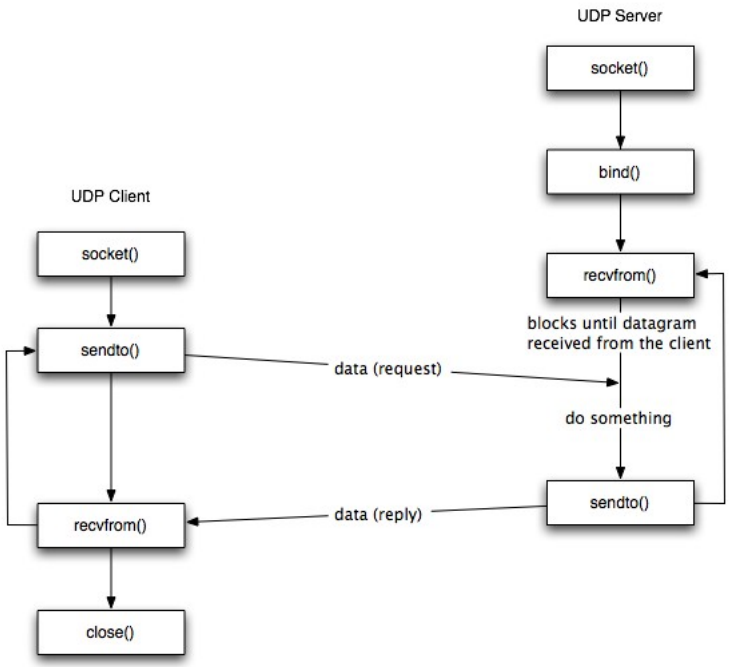
```
// Etablish connection with server
if (connect(sock, (struct sockaddr*)&server_address,
                sizeof(server_address)) < 0) {
               printf("could not connect to server\n");
               return 1;
}

// send data to server
const char* data_to_send = “Hello World";
send(sock, data_to_send, strlen(data_to_send), 0);

// receive data from
int n = 0;
int len = 0, maxlen = 100;
char buffer[maxlen];
char* pbuffer = buffer;
```

Programmation Socket

```
        // will remain open until the server terminates the connection
        while ((n = recv(sock, pbuffer, maxlen, 0)) > 0) {
                pbuffer += n;
                maxlen -= n;
                len += n;

                buffer[len] = '\0';
                printf("received: '%s'\n", buffer);
        }

// close the socket
close(sock);

return 0;
}
```

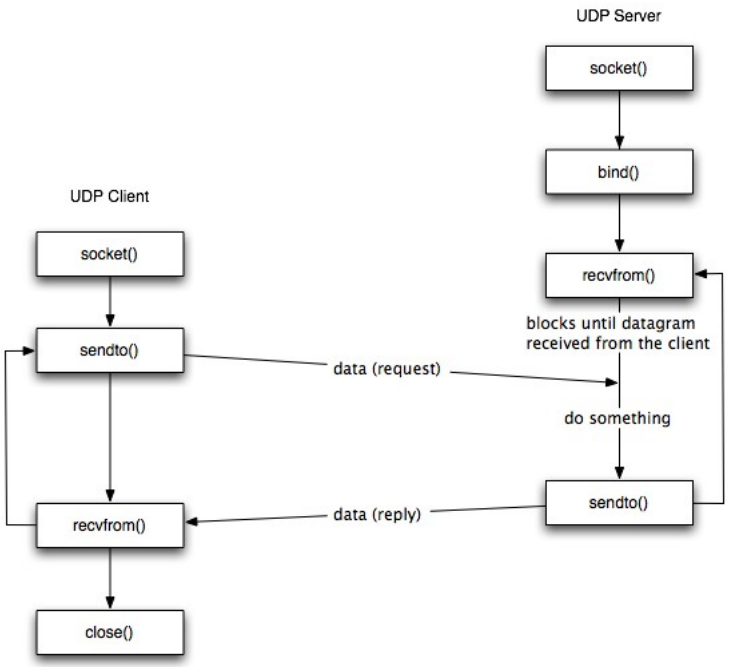
Programmation Socket

- Communication UDP - Serveur
  1. Création socket avec **socket()**
  2. Création la socket d'écoute
     - Initialiser une adresse ip:port, e.g. **sockaddr_in**
     - associer l'adresse a la socket avec **bind()**
  3. (répétition) Envoi et réception de données (**sendto()** et **recvfrom**())
  4. Fermer le socket avec **close()**



Programmation Socket

- Communication UDP - Client
  1. Création socket avec **socket()**
  2. Récupérer et initialiser l'adresse du serveur (**gethostbyname** ou **getaddrinfo**)
  3. (répétition) Envoi et réception de données (**send()** et **recv**())
  4. Fermer le socket avec **close()**



Programmation Socket

- Programmation socket en IPv6
  - Pas de changement pour les langages qui utilisent des couches d'abstraction et qui n'utilisent pas d'adresses IPv4 directement (Java par exemple)
  - L'API reste identique
  - Il y a des nouvelles fonctions de conversion pour IPv6