

# **Programmation Système Répartie et concurrente**

## **Master 1 Informatique – MU4NI400**

### **Cours 8 : Sockets**

**Yann Thierry-Mieg**

**[Yann.Thierry-Mieg@lip6.fr](mailto:Yann.Thierry-Mieg@lip6.fr)**

# Plan

---

- On a vu au cours précédent :
  - IPC POSIX pour la communication inter processus
  - Aujourd'hui : API Socket pour la communication entre machines
- 
- Références :
  - Cours de P.Sens, L.Arantes (PR  $\leq$  2017)
  - « Computer Systems : A programmers Perspective. » Bryant, O'Hallaron
  - Le man, section 2, 3 et 7

## SOCKETS

I – Introduction aux sockets

II – Mode non connecté

III – Mode connecté

IV – Concepts avancés

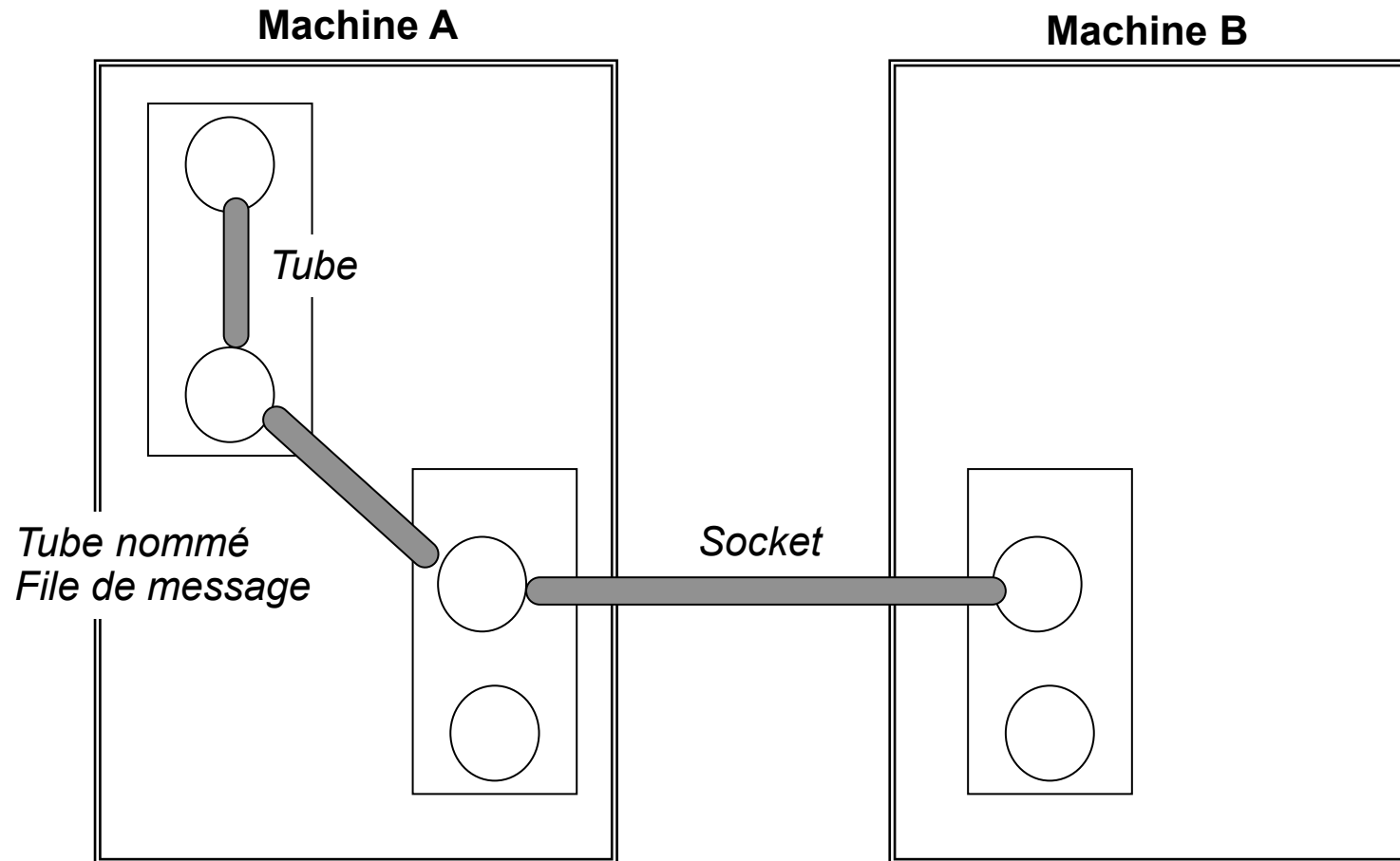
# Communication sous UNIX

---

- Communication intra-tâche (entre threads)
  - Partage de variables globales, tas
- Communication inter-tâches : locale (même machine)
  - Disque (Tube, Fichier)
  - Mémoire (Segment partagé, Sémaphore, File de messages)
- Communication inter-tâches : distante
  - Socket (TCP / UDP / IP)
  - Appel de procédure à distance (RPC : Remote Procedure Call)
  - Appel de méthode à distance (Corba, Java-RMI)
- Outils de haut niveau
  - Systèmes de fichiers répartis (NFS, RFS, AFS)
  - Bases de données réparties (NIS)

# Communication sous UNIX

---



# Communications Distantes : Qualité de Service

---

## Type de service

- Connecté (fonctionnement similaire à un tube),
- Non connecté (possibilité de perte, déséquencelement, duplication de paquets)

## Service Fiable :

- Pas de perte de données,
- Utilisation d'un acquittement,
- Augmentation de la charge du réseau et des délais de transmission

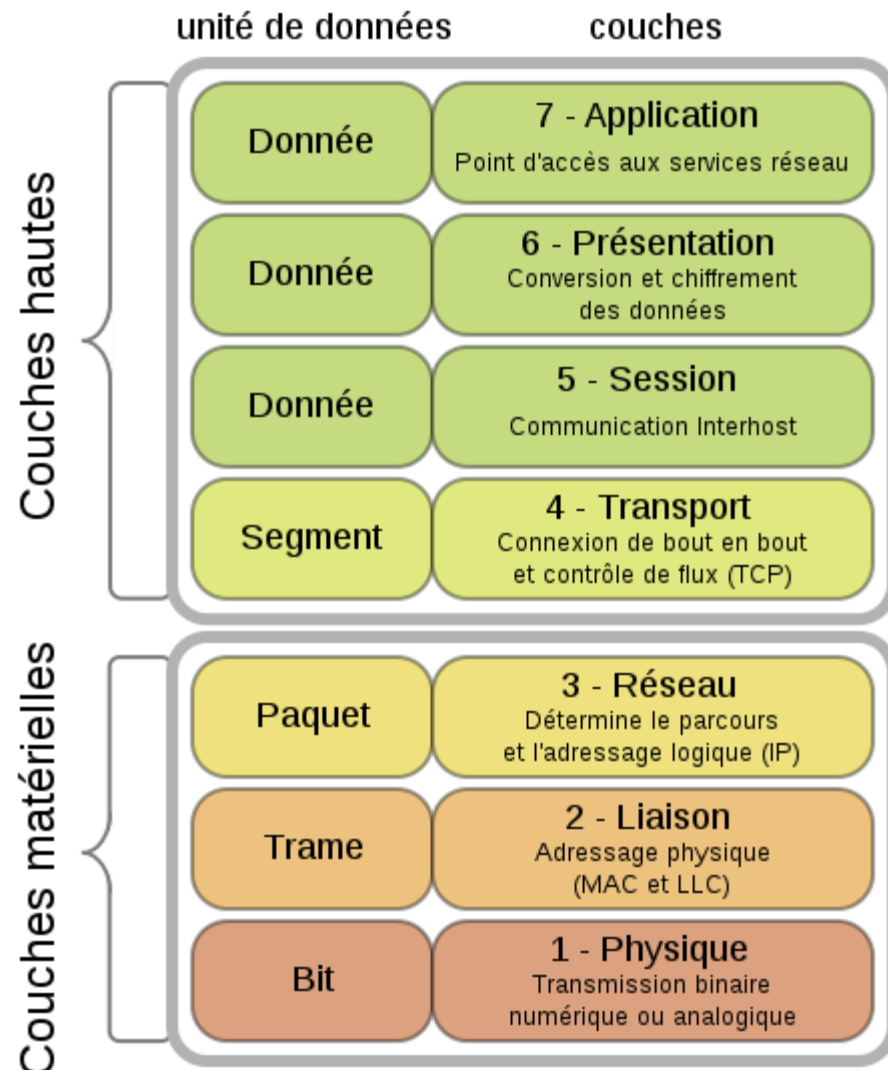
## Exemples

- Connexion fiable : Session ssh d'ordinateur à ordinateur
- Connexion non fiable : Voix numérisée
- Sans connexion / non fiable : Diffusion électronique de prospectus, multicast

# Communications Distantes :

## Modèle de référence ISO

- Physique, Liaison (ARPANET, SATNET, LAN)
- Réseau
  - Internet Protocol (IP)
- Transport
  - Transmission Control Protocol (TCP), User Datagram Protocol (UDP),
  - Internet Control Message Protocol (ICMP),
    - permet le contrôle des erreurs de transmission.
  - Internet Group Management Protocol (IGMP)
    - permet à des routeurs IP de déterminer de façon dynamique les groupes multicast .
- Session
  - (Remote Procedure Call) RPC
- Présentation
  - (eXternal Data Representation) XDR
    - XDR permet d'encoder les données de manière indépendante de l'architecture, afin de pouvoir les transférer entre systèmes hétérogènes.
- Application (telnet, ftp, smtp, dns, nfs)



# Communications Distantes :

## Notion de Socket

---

Extension de la notion de tube nommé

**4.2 BSD (1983)**

Socket : Point de communication par lequel un processus peut émettre ou recevoir d'informations

- Bidirectionnelle
- Associée à un nom (unique)
- Appartient à un domaine
- Possède un type
- Identifié par un descripteur
- Nommage est une opération distincte de leur création



# Types de Socket

---

Les deux principaux : par paquets ou en flux connecté

## SOCK\_DGRAM

- Mode non connecté
- transmission par paquet
- sans contrôle de flux
- bidirectionnel
- protocole UDP (IPPROTO\_UDP) ou IGMP (IPPROTO\_IGMP)

## SOCK\_STREAM

- Mode connecté
- avec contrôle de flux (fiabilité)
- bidirectionnel
- protocole TCP (IPPROTO\_TCP)

# Types de Socket (2)

---

D'autres types de sockets (plus rares) existent :

## SOCK\_SEQPACKET

- Mode orienté connection garantissant fiabilité
- transmission par paquet
- bidirectionnel
- Messages sont délivrés dans l'ordre d'envoi
- protocole UDP (IPPROTO\_UDP) ou IGMP (IPPROTO\_IGMP)

## SOCK\_RAW

- Accès direct avec la couche IP,
- réservé au super-utilisateur,
- protocole ICMP (IPPROTO\_ICMP),
- Définition de nouveaux protocoles (IPPROTO\_RAW)

# Choix du protocole

---

## UDP (User Datagram Protocol)

- => Perte, déséquencelement, taille limitée
- => Performant
- => Type = DGRAM

## TCP (Transport Control Protocol)

- => Pas de perte, pas de déséquencelement, flux
- => Coûteux (connexion, transfert)
- => Type = STREAM

# Notion de client/serveur

---

Etablissement asymétrique de la communication :

- Le serveur attend (écoute) les requêtes
- Le client connaît le serveur, il lui envoie une requête

Pour les protocoles connectés, passé cette phase de connexion, la communication devient symétrique

Socket = point externe de communication



# Utilisation des Sockets : Création

---

**Socket associée à un descripteur de fichier**

```
#include <sys/types.h>
#include <sys/socket.h>
int socket (int domaine, int type, int protocole);
```

**domaine**

AF\_UNIX / AF\_LOCAL /\* *sur une même machine* \*/  
AF\_INET / AF\_INET6

**type**

SOCK\_STREAM, SOCK\_DGRAM, ...

**protocole**

0 => choix automatique, (IPPROTO\_TCP, IPPROTO\_UDP)

**Retourne le descripteur de la socket ou -1 en cas d'erreur**

*eg. int s = socket (AF\_INET, SOCK\_STREAM, 0);*

# Utilisation des Sockets UNIX

---

- Extension du tube nommé

# Utilisation des Sockets : Nommage

---

Objectif : Associe à une socket une adresse qui permet de designer la socket au sein d'un domaine.

- Domaine unix

nom = nom de fichier

- Domaine internet (inet)

nom = <numéro de port, adresse IP>

```
int bind(int sock, struct sockaddr *nom, int lg_nom);
```

<b>sock</b>	numéro de socket (retourné par <code>socket()</code> )
-------------	--

<b>nom</b>	nom de la socket (dépendant du domaine)
------------	---

<b>lg_nom</b>	longueur du nom ( <code>sizeof(struct sockaddr)</code> )
---------------	--

# Nommage dans le domaine Unix (local)

---

Communication via le système de fichiers (sur partition locale uniquement)

```
#include <sys/un.h>
struct sockaddr_un {
    short int sun_family;    /* AF_UNIX (AF_LOCAL) */
    char  sun_path[104];    /* Chemin du fichier */
};
```

- socket correspond à un fichier de type *socket*
  - Fichier ne peut pas exister
  - `ls -l` : type `s`
  - Suppression de la référence associée : *rm* ou *unlink*



# Nommage dans le domaine Unix

## Exemple

---

```
/* fichier "bindlocal.c" */
#define _XOPEN_SOURCE 700
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <sys/un.h>

int main(int argc, const char **argv)
{
    int sock;
    struct sockaddr_un addr;
    memset(&addr, '\0', sizeof(struct sockaddr_un));
    addr.sun_family = AF_UNIX;
    strcpy(addr.sun_path, "./MySock");
    if ((sock = socket(AF_UNIX, SOCK_STREAM, 0)) == -1)
        {perror("Erreur creation socket");exit(1);}
    if (bind(sock, (struct sockaddr *)&addr, sizeof(addr)) == -1)
        {perror("Erreur au nommage");exit(2);}

    return (0);
}
```

# Utilisation des Sockets INET

---

- Domaine Internet

# Utilisation des Sockets : Nommage

---

Objectif : Associe à une socket une adresse qui permet de designer la socket au sein d'un domaine.

- Domaine unix

nom = nom de fichier

- Domaine internet (inet)

nom = <numéro de port, adresse IP>

```
int bind(int sock, struct sockaddr *nom, int lg_nom);
```

<b>sock</b>	numéro de socket (retourné par <code>socket()</code> )
-------------	--

<b>nom</b>	nom de la socket (dépendant du domaine)
------------	---

<b>lg_nom</b>	longueur du nom ( <code>sizeof(struct sockaddr)</code> )
---------------	--

# Nommage dans le domaine Internet

---

**Adress: entier sur 32 bits (4 octets)**

```
#include <netinet/in.h>
struct sockaddr_in {
    short sin_family;
    u_short sin_port;
    struct in_addr sin_addr;
};
struct in_addr {
    u_long s_addr;
};
```

domaine de communication (AF\_INET)  
numéro du port  
adresse IP

- *sin\_addr.s\_addr = INADDR\_ANY* :
  - . permet d'associer la socket à toutes les adresses possibles de la machine
- *port* :
  - . permet de contacter le service correspondant.
  - . envoie sans faire le *bind* : le système attachera un porte quelconque
    - \* Pour obtenir l'adresse associée :  
*getsockname (int desc, struct sockaddr \*p\_addr, int \* p\_longuer)*

# Nommage : formattage d'adresse

---

## Entier sur 32 bits (4 octets – IPV4) ou 128 bits (16 octets – IPV6)

### Hétérogénéité des machines

Big endian (octets de poids le plus fort en tête) x little endian (octets de poids le plus faible en tête)

- Normalisation des adresses et numéro de port codées en format « réseau »

Fonctions de conversion :

```
u_short htons(u_short) /* host to network short */
```

=> Conversion du numéro de port

```
u_long htonl(u_long) /* host to network long */
```

=> Conversion de l'adresse IP

```
u_short ntohs(u_short) /* network to host short */
```

```
u_long ntohl(u_long) /* network to host long */
```

- Adresse <=> Chaîne de caractères « a.b.c.d » (IPV4)

```
char *inet_ntoa(struct in_addr adr) /* adr à chaîne (affichage) */
```

convertit l'adresse *adr* en une chaîne de caractères dans la notation avec nombres et points.

```
u_long inet_addr(char *chaîne) /* chaîne à adr */
```

convertit l'adresse *chaîne* depuis la notation standard avec nombres et points en une donnée binaire.

# Exemples de services internet standard

---

- Protocole UDP
  - echo : port 1
  - daytime – port 13
  - .....
- Protocole TCP
  - echo: port 1
  - ftp: port 21
  - ssh: port 22
  - http: port 80
  - .....

# Nommage : correspondance nom/adresse

---

```
int getaddrinfo(const char *hostname, const char *service,  
                const struct addrinfo *hints, struct addrinfo **res);
```

Étant donné *hostname* et *service*, qui identifie un host et un service, **getaddrinfo()** renvoie structures *addrinfo*, contenant une ou plusieurs adresse Internet.

```
struct addrinfo {  
    int ai_flags;           /* input flags */  
    int ai_family;         /* protocol family for socket */  
    int ai_socktype;       /* socket type */  
    int ai_protocol;       /* protocol for socket */  
    socklen_t ai_addrlen;  /* length of socket-address */  
    struct sockaddr *ai_addr; /* socket-address for socket */  
    char *ai_canonname;    /* canonical name for service location */  
    struct addrinfo *ai_next; /* pointer to next in list */  
};
```

# Correspondance nom/adresse :

## Exemple Nom => IP

---

**name2ip.c**

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

int main() {
    char *hostname = "localhost";
    struct addrinfo hints, *res;
    struct in_addr addr;
    int err;

    memset(&hints, 0, sizeof(hints));
    hints.ai_socktype = SOCK_STREAM;
    hints.ai_family = AF_INET;

    if ((err = getaddrinfo(hostname, NULL, &hints, &res)) != 0) {
        printf("error %d\n", err);
        return 1;
    }
    addr.s_addr = ((struct sockaddr_in *)(res->ai_addr))->sin_addr.s_addr;
    printf("ip address : %s\n", inet_ntoa(addr));

    freeaddrinfo(res);

    return 0;
}
```



# Exemple complet : Nom de *mon* hôte

```
std::string getHostName() {
    struct ifaddrs *ifaddr;
    if (getifaddrs(&ifaddr) == -1) {
        perror("getifaddrs");
    }
    for (auto ifa = ifaddr; ifa != NULL; ifa = ifa->ifa_next) {
        if (ifa->ifa_addr == NULL)
            continue;
        if((strcmp(ifa->ifa_name,"lo")!=0)&&(ifa->ifa_addr->sa_family==AF_INET)) {
            char host[NI_MAXHOST];
            int s=getnameinfo(ifa->ifa_addr,sizeof(struct sockaddr_in), host,
                               NI_MAXHOST, NULL, 0, NI_NUMERICHOST);
            if (s != 0) {
                perror("getnameinfo");
                continue;
            }
            printf("Local IP Host : <%s>\n", host);
            freeifaddrs(ifaddr);
            return host;
        }
    }
    freeifaddrs(ifaddr);
    return "localhost";
}
```

# Correspondance nom/adresse

---

- **int getnameinfo(const struct sockaddr \**sa*, socklen\_t *salen*, char \**host*, size\_t *hostlen*, char \**serv*, size\_t *servlen*, int *flags*);**
  - La fonction **getnameinfo()** est la réciproque de **getaddrinfo**: elle convertit une adresse de socket en un hôte et un service correspondants, de façon indépendante du protocole.
  - Le paramètre *sa* est un pointeur vers l'adresse d'une structure de socket générique (de type *sockaddr\_in* ) de taille *salen* qui contient l'adresse IP et le numéro de port. Les paramètres *host* et *serv* sont des pointeurs vers des tampons alloués par l'appelant (de tailles respectives *hostlen* et *servlen* dans lesquels **getnameinfo()** place des chaînes de caractères, terminées par l'octet nul, contenant respectivement les noms du host et de service.

# Correspondance nom/adresse :

## Exemple IP => Nom

---

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <arpa/inet.h>
#include <netdb.h>
```

ip2name.c

```
int main(int argc, const char **argv)
{
    struct sockaddr_in sin;
    char host[64];

    memset((void*)&sin, 0, sizeof(sin));
    sin.sin_addr.s_addr = inet_addr(argv[1]);
    sin.sin_family = AF_INET;

    if (getnameinfo((struct sockaddr*)&sin, sizeof(sin),
                    host, sizeof(host), NULL, NULL, 0) != 0) {
        perror("getnameinfo");
        exit(EXIT_FAILURE);
    }

    printf("Name : %s\n", host);
    return (0);
}
```

# inet\_ntop function

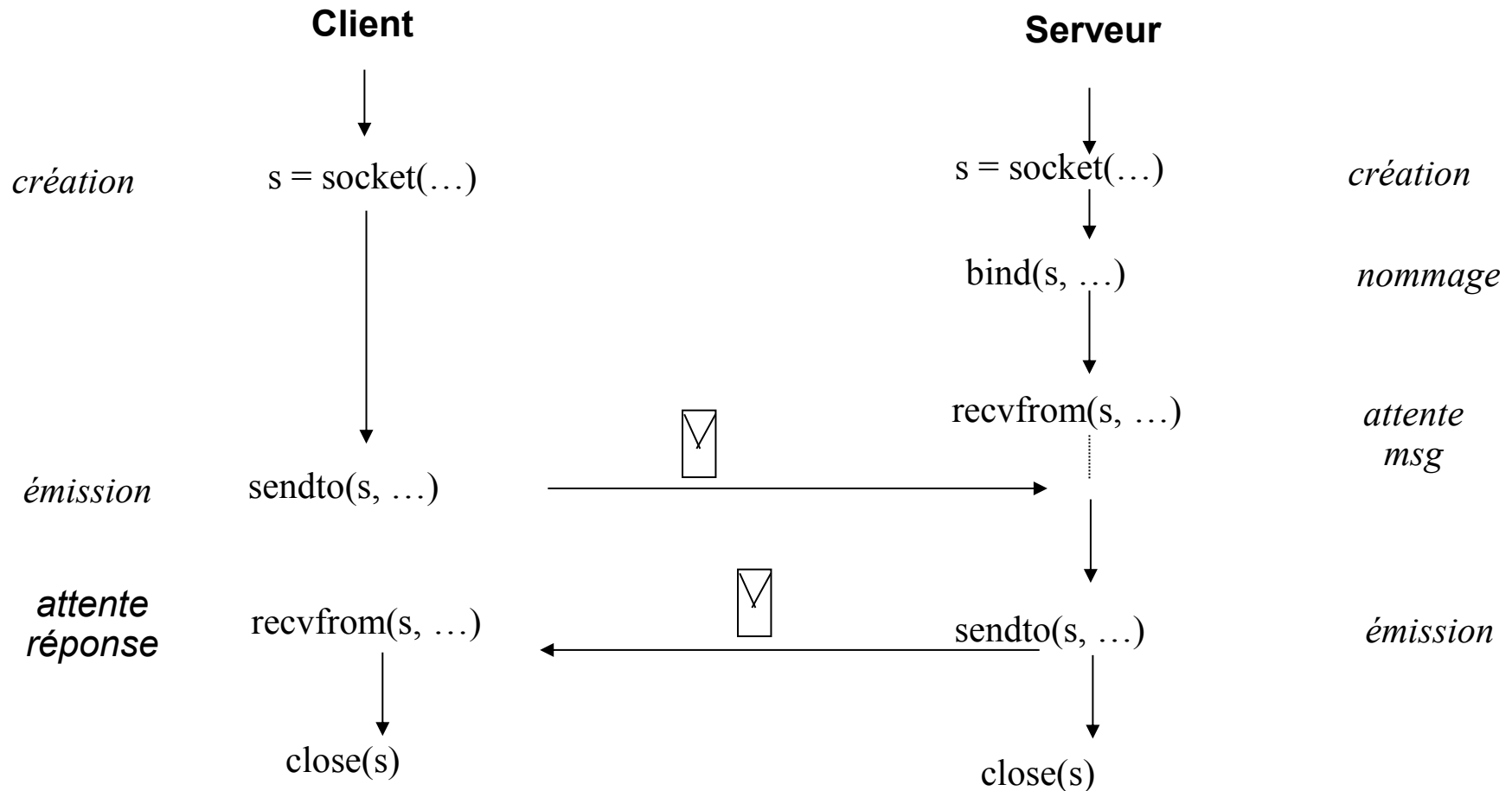
---

- **char \*inet\_ntop(int *af*, void \**src*, char \**dst*, socklen\_t *cnt*);**
  - Convertir des adresses IPv4 et IPv6 sous forme binaire en texte
  - Cette fonction convertit une adresse réseau représentée par la structure *src* de la famille *af*, en une chaîne de caractères copiée dans le tampon *dst*, long de *cnt* octets.
  - **inet\_ntop()** étend les possibilités de la fonction **inet\_ntoa ( )**.
  - Exemple :

```
void *addr; char ipstr[100];

getaddrinfo(argv[1], NULL, &hints, &res);
struct sockaddr_in *ipv4 = (struct sockaddr_in *)res->ai_addr;
addr = &(ipv4->sin_addr);
inet_ntop(AF_INET, addr, ipstr, sizeof(ipstr));
printf(" adresse IP: %s\n", ipstr);
```

# Sockets non connectées : un exemple



# Sockets non connectées : communication

---

```
int recvfrom(int sock, char* buffer, int tbuf, int flag,  
             struct sockaddr *addsrc, socklen_t *taille)
```

Lecture d'un buffer adressé à la socket **sock**,  
Adresse de l'émetteur retournée dans **addsrc**

```
int sendto(int sock, char* buff, int tbuf, int flag,  
           struct sockaddr *addrdst, socklen_t taille)
```

Envoi par la socket **sock** du contenu de **buff** à l'adresse **addrdst**

Taille limitée à la taille d'un paquet

Champ **flag**

**MSG\_PEEK** Lecture (`recvfrom`) sans modification de la file d'attente

# Sockets non connectées : un exemple

## Partie serveur - serveurUDP.c

---

```
/* Includes ... */
#define PORTSERV 4567
int main(int argc, char *argv[])
{
    struct sockaddr_in sin;    /* Nom de la socket du serveur */
    struct sockaddr_in exp;    /* Nom de l'expediteur */
    char host[64];
    int sc ;
    int fromlen = sizeof(exp);
    char message[80];
    int cpt = 0;

    /* creation de la socket */
    if ((sc = socket(AF_INET, SOCK_DGRAM, 0)) < 0) {
        perror("socket"); exit(1);
    }

    /* remplir le « nom » */
    memset((char *)&sin, 0, sizeof(sin));
    sin.sin_addr.s_addr = htonl(INADDR_ANY);
    sin.sin_port = htons(PORTSERV);
    sin.sin_family = AF_INET;

    /* nommage */
    if (bind(sc, (struct sockaddr *)&sin, sizeof(sin)) < 0) {
        perror("bind"); exit(2);
    }
}
```

# Sockets non connectées : un exemple

## Partie serveur - serveurUDP.c (suite)

---

```
/** Reception du message */
if ( recvfrom(sc,message,sizeof(message),0,
             (struct sockaddr *)&exp,&fromlen) == -1) {
    perror("recvfrom"); exit(2);
}
/** Affichage de l'expediteur */
printf("Exp : <IP = %s,PORT = %d> \n", inet_ntoa(exp.sin_addr),
       (exp.sin_port));

/* Nom de la machine */
if (getnameinfo((struct sockaddr *)&exp, sizeof(exp),
               host, sizeof(host), NULL, NULL, 0) != 0) {
    perror("getnameinfo"); exit(3);
}
printf("Machine : %s\n", host);

/** Traitement */
printf("Message : %s \n", message);
cpt++;

/** Envoyer la reponse */
if (sendto(sc,&cpt,sizeof(cpt),0,(struct sockaddr *)&exp,fromlen) == -1) {
    perror("sendto"); exit(4);
}
close(sc);
return (0);
}
```



# Sockets non connectées : un exemple

## Partie client - clientUDP.c

---

```
... /* Includes */

#define PORTSERV 4567 /* Port du serveur */

int main(int argc, char *argv[])
{
    int reponse;
    struct sockaddr_in dest;
    int sock;
    int fromlen = sizeof(dest);
    char message[80];
    struct addrinfo *result;

    /* Le nom de la machine du serveur est passé en argument */
    if (argc != 2) {
        fprintf(stderr, "Usage : %s machine \n", argv[0]);
        exit(1);
    }

    if ((sock = socket(AF_INET, SOCK_DGRAM, 0)) == -1) {
        perror("socket"); exit(1);
    }

    ...
```

# Sockets non connectées : un exemple

## Partie client - clientUDP.c (suite)

---

```
/* Remplir la structure dest */
if (getaddrinfo(argv[1], NULL, NULL, &result) != 0) {
    perror("getaddrinfo"); exit(EXIT_FAILURE);}

dest.sin_addr = ((struct sockaddr_in*)result->ai_addr)->sin_addr;
dest.sin_family = AF_INET;
dest.sin_port = htons(PORTSERV);

/* Contruire le message ...*/
strcpy(message, "MESSAGE DU CLIENT");

/* Envoyer le message */
if (sendto(sock,message,strlen(message)+1,0,
           (struct sockaddr *)&dest, sizeof(dest)) == -1) {
    perror("sendto"); exit(1);
}

/* Recevoir la reponse */
if (recvfrom(sock,&reponse,sizeof(reponse),0,0,&fromlen) == -1) {
    perror("recvfrom"); exit(1);
}
printf("Reponse : %d\n", reponse);
close(sock);
return(0);
}
```

# Sockets connectées : introduction

---

## **Coté serveur**

1. Autorisation d'un nombre maximum de connexions pendantes,
2. Attente d'une connexion,
3. Acceptation d'une connexion,
4. Entrées/sorties,
5. Fermeture de la connexion

## **Coté client**

1. Demande de connexion
2. Entrées/sorties,
3. Fermeture de la connexion

# Sockets connectées : fonctionnement

---

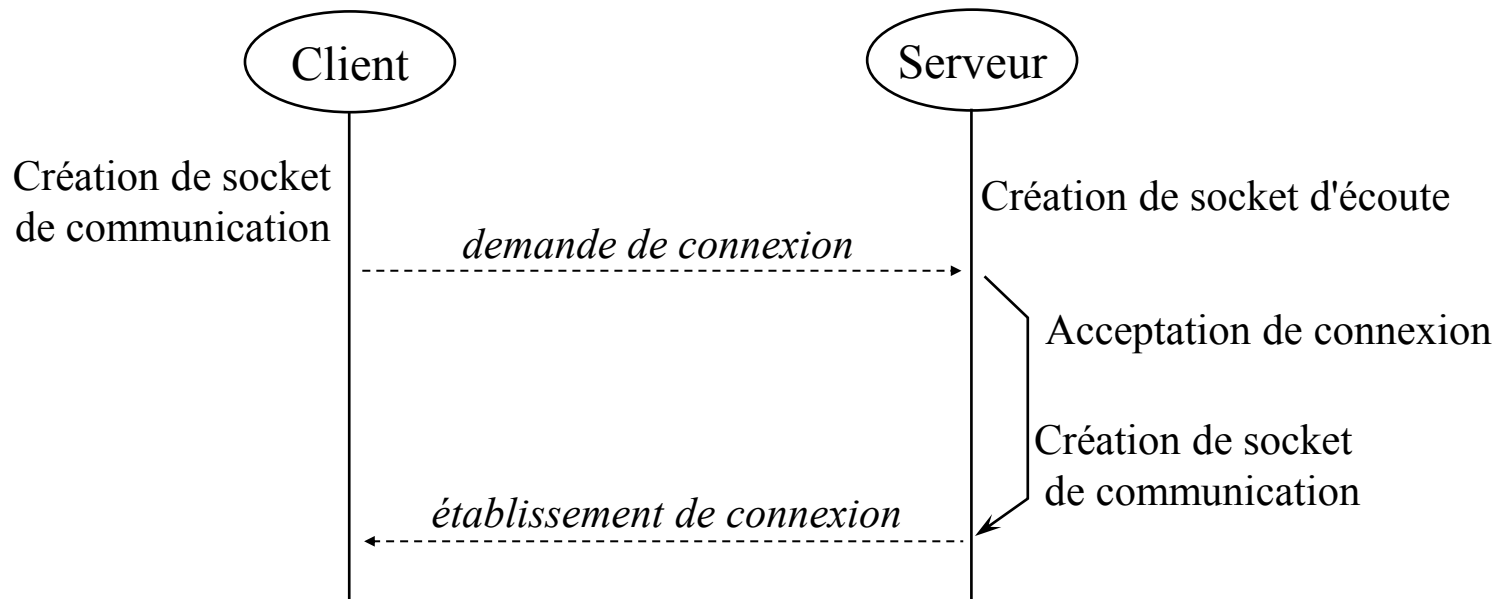
## Côté serveur

1 socket pour les demandes de connexion

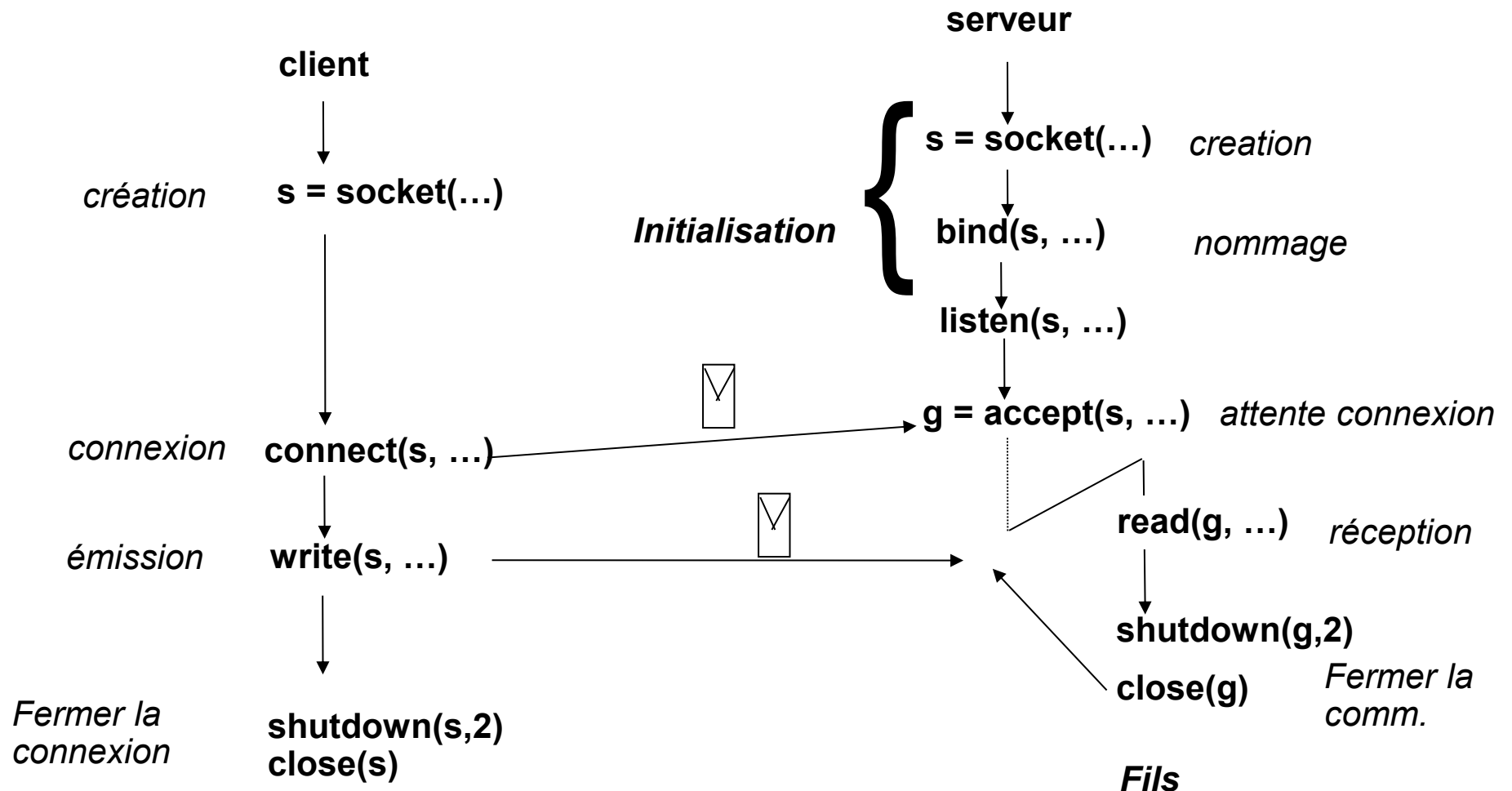
1 socket pour les communications

## Côté client

1 socket pour communiquer



# Exemple de client / serveur multi-processus en mode connecté



# Sockets connectées

## Etablissement de connexion

---

### Côté serveur

```
int listen (int sock, int nb_pendantes)
```

Création de la file d'attente des requêtes de connexion

Appel non bloquant

nb\_pendantes : nombre maximal de connexions pendantes acceptables

```
int accept (int sock, struct sockaddr *addrclt, socklen_t *taille)
```

Attente d'acceptation d'une connexion

Identité du client fournie dans l'adresse addrclt

Appel bloquant => lors de l'acceptation :

Création d'une nouvelle socket

Renvoie l'identificateur de la socket de communication

### Côté client

```
int connect (int sock, struct sockaddr *addrsrv, socklen_t taille)
```

Demande d'une connexion

Appel bloquant

# Sockets connectées : communication

---

```
int read (int sock, char *buffer, int tbuf)
int write (int sock, char *buffer, int tbuf)
```

## Primitives UNIX usuelles

```
int recv (int sock, char *buffer, int tbuf, int flag)
int send (int sock, char *buffer, int tbuf, int flag)
int recvmsg (int sock, struct msghdr *msg, int flag)
int sendmsg (int sock, struct msghdr *msg, int flag)
```

## Regroupement de plusieurs écritures ou lectures

## Appels bloquants par défaut

flag :

MSG_OOB	données hors bande (en urgence/données de contrôle)
MSG_PEEK	lecture ( <code>recv</code> ) sans modification de la file d'attente
MSG_WAITALL	lecture ( <code>recv</code> ) reste bloquante jusqu'à réception d'au moins <code>tbuf</code> octets

# Sockets connectées : déconnexion

---

```
int shutdown(int s, int how);
```

Vise désactiver la réalisation d'un lecture et/ou écriture sur une socket.

s           descripteur de la socket

how        mode de déconnexion

0 => réception désactivée

1 => émission désactivée

2 => émission et réception désactivées

shutdown est censé être suivi d'un close



# Exemple connecté : Partie serveur

---

```
#define PORTSERV 7100
int main(int argc, char *argv[])
{
    struct sockaddr_in sin; /* Nom de la socket de connexion */
    int sc ;                /* Socket de connexion */
    int scom;               /* Socket de communication */
    struct hostent *hp;
    int fromlen = sizeof exp;
    int cpt;

    /* creation de la socket */
    if ((sc = socket(AF_INET,SOCK_STREAM,0)) < 0) {
        perror("socket"); exit(1);
    }

    memset((char *)&sin,0, sizeof(sin));
    sin.sin_addr.s_addr = htonl(INADDR_ANY);
    sin.sin_port = htons(PORTSERV);
    sin.sin_family = AF_INET;

    /* nommage */
    if (bind(sc,(struct sockaddr *)&sin,sizeof(sin)) < 0) {
        perror("bind");
        exit(1);
    }
    listen(sc, 5);
```

**serveurTCP.c**

# Exemple connecté : Partie serveur

---

```
/* Boucle principale */
for (;;) {
    if ((scom = accept(sc, (struct sockaddr *)&exp, &fromlen)) == -1) {
        perror("accept"); exit(3);
    }
    /* Création d'un processus fils qui traite la requete */
    if (fork() == 0) {
        /* Processus fils */
        if (read(scom, &cpt, sizeof(cpt)) < 0) {
            perror("read"); exit(4);
        }
        /** Traitement du message ***/
        cpt++;
        if (write(scom, &cpt, sizeof(cpt)) == -1) {
            perror("write"); exit(2);
        }

        /* Fermer la communication */
        shutdown(scom, 2);
        close(scom);
        exit(0);
    }
} /* Fin de la boucle for */
close(sc);
return 0;
}
```

**serveurTCP.c**

# Exemple connecté : Partie client

---

```
...
#define PORTSERV 7100
#define h_addr h_addr_list[0]          /* definition du champs h_addr */
int main(int argc, char *argv[])
{
    struct sockaddr_in dest; /* Nom du serveur */
    int sock;
    int fromlen = sizeof(dest);
    int msg;
    int reponse;

    if (argc != 2) {
        fprintf(stderr, "Usage : %s machine\n", argv[0]);
        exit(1);
    }

    if ((sock = socket(AF_INET, SOCK_STREAM, 0)) == -1) {
        perror("socket");
        exit(1);
    }

    /* Remplir la structure dest */

        cf. Client UDP...
```

clientTCP.c

# Exemple connecté : Partie client (suite)

---

```
/* Etablir la connexion */
if (connect(sock, (struct sockaddr *) &dest, sizeof(dest)) == -1) {
    perror("connect"); exit(1);
}
msg = 10;
/* Envoyer le message (un entier ici) */
if (write(sock, &msg, sizeof(msg)) == -1) {
    perror("write"); exit(1);
}

/* Recevoir la reponse */
if (read(sock, &reponse, sizeof(reponse)) == -1) {
    perror("recvfrom"); exit(1);
}
printf("Reponse : %d\n", reponse);

/* Fermer la connexion */
shutdown(sock, 2);
close(sock);
return(0);
}
```

clientTCP.c

# Communication connectée avancée

## Attente simultanée

---

### **Primitive select :**

Permet à un processus de réaliser le multiplexage par d'opérations bloquantes sur des descripteurs.

```
#include <sys/select.h>
int select (int maxl, fd_set *lecteurs, fd_set *ecrivains,
            fd_set *exceptions, struct timeval *delai_max)
```

- Attente simultanée sur trois ensembles de descripteurs
  - Retourne
    - le nombre total de descripteurs correspondant à une E/S;
    - 0 : temps d'attente maximum s'est écoulé;
    - 1 : erreur
- interruptible par des signaux : `errno = EINTR`

# Primitive select

---

- **Paramètres:**

- **lecteurs** = pointeur sur un ensemble de descripteurs sur lesquels on veut réaliser une lecture;
- **ecrivains** = pointeur sur un ensemble de descripteurs sur lesquels on veut réaliser une écriture;
- **exceptions** = pointeur sur un ensemble de descripteurs sur lesquels on veut réaliser un test de condition exceptionnelle
  - Exemple: caractère hors bande
- **max1** = numéro du plus grand descripteur + 1 appartenant à l'un de trois ensembles.
- **delai\_max** = temps maximal d'attente avant que l'une des opérations soit possible.
  - NULL => bloque indéfiniment

# Primitive select

---

## **Macros permettant la manipulation des ensemble de descripteurs type `fd_set`:**

- `FD_ZERO (fd_set* ensemble)`  
Mise à zéro de l'ensemble
- `FD_SET (int fd, fd_set* ensemble)`  
Ajoute un descripteur à l'ensemble
- `FD_CLR (int fd, fd_set* ensemble)`  
Supprime un descripteur de l'ensemble
- `FD_ISSET (int fd, fd_set* ensemble)`  
Teste si un descripteur est dans l'ensemble

# Exemple – attente simultanée sur stdin et socket

---

```
...
int main(int argc, char *argv[]){
    ... /* initialisation idem serveurTCP.c */
    printf("Appuyer sur une <Entree> pour tuer le serveur\n");
    /* Boucle principale */
    for (;;) {
        fd_set mselect;
        /* Construire le masque du select */
        FD_ZERO(&mselect);
        FD_SET(0, &mselect); /* stdin */
        FD_SET(sc, &mselect); /* la socket */

        if (select(sc+1, &mselect, NULL, NULL, NULL) == -1) {
            perror("select");
            exit(3);
        }
    }
}
```

**serveurTCP2.c**



# Exemple – attente simultanée sur stdin et socket

---

```
/** Un evenement a eu lieu : tester le descripteur */
if (FD_ISSET(0, &mselect)) {
    /* Sur stdin */
    break; /* Sortie de la boucle */
}
if (FD_ISSET(sc, &mselect)) {
    /* Sur la socket de connexion */

    /* Etablir la connexion */
    if ( (scom = accept(sc, (struct sockaddr *)&exp, &fromlen)) == -1) {
        perror("accept"); exit(2);
    }
    /** Lire le message */
    if (read(scom, message, sizeof(message)) < 0) {
        perror("read"); exit(1);
    }
    ...
    /* Fermer la connexion */
    shutdown(scom, 2);
    close(scom);
}

} /* Fin de la boucle */
close(sc);
return 0;
}
```

**serveurTCP2.c**

# Exemple – attente de connexion avec temporisateur

---

...

```
struct timeval timeout;
```

```
timeout.tv_sec = 5; /* 5 secondes */
```

```
timeout.tv_usec = 0; /* 0 micro-seconde (10E-6 sec.) */
```

```
FD_ZERO(&mselect);
```

```
FD_SET(sc, &mselect); /* la socket */
```

```
if (select(sc+1, &mselect, NULL, NULL, &timeout) == -1) {  
    perror("select");  
    exit(3);  
}
```

# Primitive poll

---

- **int poll(struct pollfd \*ufds, unsigned int nfds, int timeout);**

Extension de la primitive select. Permet de se mettre en attente sur l'occurrence de l'un des événements spécifiés sur le tableau *ufds*.

- **ufds** : tableau d'événements du type:

```
struct pollfd { int fd; /* Descripteur de fichier */  
                short events; /* Evénements attendus */  
                short revents; /* Evénements détectés */
```

- *events* : paramètre d'entrée - masque de bits indiquant les événements qui intéressent l'application.
- *revents* est un paramètre de sortie, rempli par le noyau avec les événements qui se sont effectivement produits, du type demandé.
- **nfds** : nombre d'entrées du tableau
- **timeout** : délai d'attente.

**Valeur de retour** : nombre de descripteurs sur lesquels un événement attendu s'est produit.

# Primitive poll

---

- Les bits possibles pour ces masques sont définis dans `<sys/poll.h>`:

<code>#define POLLIN</code>	Lecture possible sur le descripteur
<code>#define POLLPRI</code>	Données urgentes sur le descripteur
<code>#define POLLOUT</code>	Ecriture possible sur le descripteur
<code>#define POLLERR</code>	Erreur détectée (réponse)
<code>#define POLLHUP</code>	Fin de fichier détecté.

# Exemple – poll

---

```
#include <stdlib.h>
#include <stdio.h>
#include <sys/types.h>
#include <stropts.h>
#include <poll.h>
#include <unistd.h>
#include <errno.h>
#include <string.h>

#define NORMAL_DATA 1
#define HIPRI_DATA 2

int poll_two_normal(int fd1,int fd2) {
    struct pollfd poll_list[2];
    int retval;

    poll_list[0].fd = fd1;
    poll_list[1].fd = fd2;
    poll_list[0].events = POLLIN | POLLPRI;
    poll_list[1].events = POLLIN | POLLPRI;
```

# Example – poll

---

```
while(1) {  
    retval = poll(poll_list,(unsigned long)2,-1);  
    if(retval < 0) {  
        fprintf(stderr,"Error while polling: %s\n",strerror(errno));  
        return -1;  
    }  
  
    if((poll_list[0].revents&POLLIN) == POLLIN)  
        .....;  
    if((poll_list[0].revents&POLLPRI) == POLLPRI)  
        .....;  
    if((poll_list[1].revents&POLLIN) == POLLIN)  
        .....;  
    if((poll_list[1].revents&POLLPRI) == POLLPRI)  
        .....;  
}
```

# Rappel fonction fcntl

---

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

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

```
int fcntl(int fd, int cmd, ...);
```

- Permet, en fonction de la valeur de *cmd*, d'obtenir ou modifier des attributs associés au descripteur.
  - F\_GETFD : retourne la valeur de l'attributs du descripteur.
  - F\_SETFD : utilise un troisième paramètre pour donner une valeur aux attributs.
  - F\_GETFL: retourne l'ensemble des attributs positionnés lors du open
  - F\_SETFL : le troisième paramètre définit les nouveaux attributs.

Exemple :

```
desc= open(file, O_RDWR);
```

```
mode= fcntl (desc, F_GETFL);
```

```
fcntl (desc, F_SETFL, mode | O_APPEND) ;
```

# Réception Asynchrone

---

- Mécanisme qui permet à un processus de demander qu'un signal (SIGIO ou SIGPOLL) lui soit envoyé lorsqu'une lecture devient possible sur le descripteur.
  - Installer un handler de signal.
  - Associer le descripteur au processus dont on veut recevoir le signal  
`fcntl(desc, F_SETOWN, pid);`
  - Choix du signal à recevoir  
`fcntl (desc, F_SETSIG, sig);`
  - Activation de la réception asynchrone  
`mode= fcntl (desc, F_GETFL);`  
`fcntl (desc, F_SETFL, mode | O_ASYNC);`
- Note : possibilité d'utiliser *ioctl* au lieu de *fcntl*



# Exemple reception asynchrone

```
void SIGIOHandler(int sig)
{
    struct sockaddr_in Addr;
    unsigned int len;
    char Buffer[ MAX];
    int size;
    do
    {
        len = sizeof(Addr);

        if (( size= recvfrom(sock, Buffer, MAX, 0,
            (struct sockaddr *) &Addr, len)) <0)
            perreur ("recvfrom");
        else
            /*
                printf("message":%s\n ",Buffer);
            */
    }
    while (size >= 0);
}
```

```
int sock, mode; struct sigaction handler;
int main(int argc, char *argv[]) {
    /* crée la socket desc */
    ....
    /* nomage : bind */
    ....
    handler.sa_handler = SIGIOHandler;
    sigfillset(&handler.sa_mask) ;
    handler.sa_flags = 0;

    if (sigaction(SIGIO, &handler, 0) < 0) {
        perror("sigaction"); return ; }

    if (fcntl(sock, F_SETOWN, getpid()) < 0) {
        perror ("fcntl"); return; }
    mode = fcntl ( sock, F_GETFL);
    if (fcntl(sock, F_SETFL, mode| O_NONBLOCK | O_ASYNC) < 0)
        perror ("fcntl"); return; }
    while (1)
        pause ();
}
```

# Options d'une socket : Lecture et Ecriture

---

```
int getsockopt (int sock, int couche, int cmd,  
               void *val, socklen_t *taille)
```

## Lecture des options

- **couche de protocole** (SOL\_SOCKET, IPPROTO\_IP, IPPROTO\_TCP)
  - SOL\_SOCKET: manipuler les sockets au niveau de l'API.

- **cmd utilise le champ de données val**

SO_TYPE	Type de socket
SO_RCVBUF	Taille du buffer de réception
SO_SNDBUF	Taille du buffer d'émission
SO_ERROR	Valeur d'erreur de la socket (non connectée)

```
int setsockopt (int sock, int couche, int cmd,  
               void *val, socklen_t taille)
```

## Modification des options

- **cmd utilise le champ de données val**

SO_BROADCAST	Autorisation de trames broadcast
IP_ADD_MEMBERSHIP	Autorisation d'une requête multicast
SO_REUSEADDR	Autorisation de réutiliser une @ déjà affectée. Empêcher l'erreur EADDINUSE

# Exemple getsockopt

---

```
int main( int argc, char** argv)
{
    int optlen, gs, socktype, s;
    if ((s = socket(AF_INET, SOCK_DGRAM, 0)) == -1)
        { perror("Socket not created"); return EXIT_FAILURE; }

    optlen = sizeof(socktype);
    if ((gs = getsockopt (s, SOL_SOCKET, SO_TYPE, &socktype, &optlen)) == -1)
        { perror("getsockopt failed"); return EXIT_FAILURE; }

    switch (socktype) {
        case SOCK_STREAM: printf("Stream socket \n"); break;
        case SOCK_DGRAM:  printf("Datagram socket \n"); break;
        case SOCK_RAW:    puts("Raw socket \n"); break;
        default:          printf("Unknown socket type\n"); break;
    }
    return EXIT_SUCCESS;
}
```

# Options d'une socket : Diffusion Broadcast

---

## Diffusion broadcast - UDP

- Envoi d'un message UDP vers tous les ordinateurs d'un sous-réseau
- Mettre une adresse IP de diffusion (bits d'adresse ordinateur à 1)

- INADDR\_BROADCAST: adresse (255.255.255.255)

- acquérir l'adresse de diffusion pour l'interface physique:

```
long int adr_diffusion;  
adr_diffusion = ioctl (sock, SIOCGIFBRDADDR, NULL);
```

- Autorisation pour diffuser en mode broadcast:

```
int on=1;  
setsockopt(s, SOL_SOCKET, SO_BROADCAST, &on, sizeof(on));
```

# Options d'une socket : Diffusion Multicast

---

## Diffusion multicast – UDP

Le **multicast** permet la diffusion par un émetteur vers un groupe de récepteurs.

- Les récepteurs intéressés par les messages adressés à ce groupe doivent s'inscrire à ce groupe.
- Choisir une adresse IP de diffusion multicast

260 millions d'adresses disponibles (224.0.0.0 à 239.255.255.255)

- Abonnement à un groupe multicast:

```
struct ip_mreq {  
    struct in_addr imr_multiaddr; /* IP multicast address of  
group */  
    struct in_addr imr_interface; /* local IP address of  
interface */  
};
```

Exemple :

```
struct ip_mreq imr;  
imr.imr_multiaddr.s_addr = inet_addr(MON_ADR_DIFF);  
imr.imr_interface.s_addr = INADDR_ANY  
setsockopt(sock, IPPROTO_IP, IP_ADD_MEMBERSHIP, (char *)&imr,  
sizeof(struct ip_mreq))
```

- Envoi de la requête au groupe

# Exemple – Emetteur multi-cast

---

```
#define MON_ADR_DIFF "225.0.0.10"
#define PORTSERV 7200

int main(int argc, char *argv[])
{
    struct sockaddr_in dest;
    int sock;
    char message[80];

    if ((sock = socket(AF_INET, SOCK_DGRAM, 0)) == -1) {
        perror("socket"); exit(1);
    }

    /* Remplir la structure dest */
    memset((char *)&dest, 0, sizeof(dest));
    dest.sin_addr.s_addr = inet_addr(MON_ADR_DIFF);
    dest.sin_family = AF_INET;
    dest.sin_port = htons(PORTSERV);

    /* Contruire le message ...*/
    strcpy(message, "MESSAGE DU CLIENT");

    /* Envoyer le message */
    if ( sendto(sock, message, strlen(message)+1, 0,
                (struct sockaddr*)&dest, sizeof(dest)) == -1) {
        perror("sendto"); exit(1);
    }
    close(sock);
    return(0);
}
```

# Exemple – Recepteur multi-cast

---

```
#define MON_ADR_DIFF "225.0.0.10" /* Adresse multi-cast */
#define PORTSERV 7200 /* Port du serveur */

int main(int argc, char *argv[])
{
    int sock;
    struct sockaddr_in sin;
    struct ip_mreq imr; /* Structure pour setsockopt */
    char message[80];

    if((sock = socket(AF_INET, SOCK_DGRAM, 0)) < 0){
        perror("socket");
        exit(1);
    }

    imr.imr_multiaddr.s_addr = inet_addr(MON_ADR_DIFF);
    imr.imr_interface.s_addr = INADDR_ANY;

    if(setsockopt(sock, IPPROTO_IP, IP_ADD_MEMBERSHIP, (char *)&imr,
                  sizeof(struct ip_mreq)) == -1){
        perror("setsockopt");
        exit(2);
    }
}
```

# Exemple – Récepteur multi-cast

---

```
/* remplir le « nom » */
memset((char *)&sin,0, sizeof(sin));
sin.sin_addr.s_addr = htonl(INADDR_ANY);
sin.sin_port = htons(PORTSERV);
sin.sin_family = AF_INET;

/* nommage */
if (bind(sock,(struct sockaddr *)&sin,sizeof(sin)) < 0) {
    perror("bind");
    exit(3);
}
/* Reception du message */
if (recvfrom(sock,message,sizeof(message),0,NULL,NULL) == -1) {
    perror("recvfrom");
    exit(4);
}

printf("Message reçu :%s\n", message);
close(sock);
return (0);
}
```



# Visualisation des sockets

---

## Commande netstat

\$ netstat <*option*>

--unix	sockets locales
--inet	sockets internet
--tcp	sockets TCP
--udp	sockets UDP