

# Sockets & programmation réseau POSIX

## Sockets

Socket : point de communication bidirectionnel

La communication peut être :

- ▶ une communication réseau
- ▶ entre deux processus de la même machine
- ▶ entre un processus et le noyau...

Une connexion réseau :

- ▶ 2 sockets
- ▶ = les deux points finaux de la connexion

Attention :

- ▶ une socket n'est pas forcément pour une communication réseau
- ▶ une communication réseau n'est pas forcément une communication par le protocole IP

## Vue générale

- ▶ créer une socket : `socket`
- ▶ associer une socket : `bind`
- TCP / Côté "serveur" :
  - ▶ écouter sur une socket : `listen`
  - ▶ accepter une connexion : `accept`
- TCP Côté "client" :
  - ▶ se connecter : `connect`
- Côté "client" et "serveur" :
  - ▶ envoyer un message : `write`, `send`, `sento`
  - ▶ recevoir un message : `read`, `recv`, `recvfrom`
  - ▶ fermer une socket : `shutdown` et `close`

Créer une socket :  
`int socket(int domain, int type, int protocol)`

Renvoie un descripteur de fichier (-1 si erreur)

Domaine : ("domaine de communication", familles de protocoles)

- ▶ `AF_UNIX` : socket local (voir chapitre tubes)
- ▶ `AF_INET` : Internet IPv4
- ▶ `AF_INET6` : internet IPv6
- ▶ `AF_PACKET` : paquets liaison (ethernet...)
- ▶ autres réseaux ou réseaux obsolètes (IPX, X25, AppleTalk...)

Créer une socket :

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

Type :

- ▶ SOCK\_STREAM : par flot, connecté, bidirectionnel, fiable  
pour AF\_INET(6), c'est TCP
- ▶ SOCK\_DGRAM : par paquets, non connecté, non fiable  
pour AF\_INET(6), c'est UDP
- ▶ SOCK\_RAW : accès réseau "brut"
- ▶ SOCK\_SEQPACKET : paquets, connecté, bidirectionnel, fiable  
pour AF\_INET(6), protocole SCTP (en cours de déploiement)

Protocole :

- ▶ Pour AF\_INET ou AF\_INET6, c'est le numéro du protocole  
dans le paquet IP (TCP=6, UDP=17)
- ▶ Si un unique protocole existe dans le domaine/type,  
protocole peut être à zéro

Créer une socket :

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

Si domaine = AF\_INET (IPv4) ou AF\_INET6 (IPv6)

- ▶ TCP : type = SOCK\_STREAM,  
protocole = 0 ou IPPROTO\_TCP (=6)
- ▶ UDP : type = SOCK\_DGRAM,  
protocole = 0 ou IPPROTO\_UDP (=17)
- ▶ pour construire un paquet IP "brut" :  
type = SOCK\_RAW, protocole > 0 (champ "protocol" dans  
le paquet IP)  
(et il faut les droits qui vont avec...)

Créer une socket :

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

#### Erreurs possibles

- ▶ EPERM : opération non permise (exemple : AF\_INET/SOCK\_RAW pour utilisateur lambda)
- ▶ ESOCKTNOSUPPORT : type non supporté (exemple : AF\_INET/SOCK\_SEQPACKET)
- ▶ EPROTONOSUPPORT : protocole non supporté
- ▶ ...

#### RAPPEL : Mode "flot" (STREAM)

Mode flot : les envois successifs d'informations s'additionnent.  
Il n'y a pas de "séparations" entre elles.

Exemple :

- ▶ write(in, "ABC", 3)
  - ▶ le tube contient "ABC"
- ▶ write(in, "123", 3)
  - ▶ le tube contient "ABC123"
- ▶ read(out, bf, 4)
  - ▶ renvoie 4, et bf contient "ABC1"
  - ▶ le tube contient "23"
- ▶ read(out, bf, 4)
  - ▶ renvoie 2, et bf contient "23"
  - ▶ le tube est vide
- ▶ read(out, bf, 4)
  - ▶ bloque jusqu'à ce qu'un processus écrive dans la socket...

## RAPPEL : Mode paquet (DGRAM)

Au contraire du mode flot (STREAM), chaque information envoyée constitue une entité indivisible.

Exemple :

- ▶ write(in, "ABC", 3)
  - ▶ la file de messages contient "ABC"
  - ▶ write(in, "123", 3)
    - ▶ la file contient "ABC","123"
  - ▶ read(out, bf, 10)
    - ▶ renvoie 3, et bf contient "ABC"
    - ▶ la file contient "123"
  - ▶ read(out, bf, 10)
    - ▶ renvoie 3, et bf contient "123"
    - ▶ la file vide
  - ▶ read(out, bf, 4)
    - ▶ bloque jusqu'à ce qu'un processus écrive dans la socket...

Attention : si le tampon n'est pas assez grand, la fin du datagramme est perdue !

## Attacher une socket

socket() permet de créer une socket, mais elle n'est par défaut associée à rien (ni adresse, ni port).

Pour l'associer, il faut utiliser :

```
int bind(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 de la machine locale
- ▶ addrlen : taille de la structure `addr`
- ▶ renvoie 0 (OK), ou -1 (erreur)

## Attacher une socket

`addr` dépend du domaine de communication. Chaque domaine à sa structure `sockaddr` (et sa taille). D'où l'intérêt de `addrlen`.

- ▶ `AF_INET : sockaddr_in`
- ▶ `AF_INET6 : sockaddr_in6`

## Attacher une socket

```
struct sockaddr_in {  
    sa_family_t sin_family; /* AF_INET */  
    in_port_t sin_port; /* port */  
    struct in_addr sin_addr; /* adresse IPv4 */  
};  
  
/* Adresse Internet */  
struct in_addr {  
    uint32_t s_addr; /* adresse */  
};
```

## Attacher une socket

```
struct sockaddr_in6 {
    sa_family_t sin6_family; /* AF_INET6 */
    in_port_t sin6_port; /* port */
    sin6_flowinfo; /* info flux */
    struct in6_addr sin6_addr; /* adresse IPv6 */
    sin6_scope_id; /* Scope ID */
};

struct in6_addr {
    unsigned char s6_addr[16]; /* adresse */
};
```

## Attacher une socket

- ▶ sin\_family : le domaine de communication de la socket
- ▶ sin\_port : le port (TCP ou UDP)
  - ▶ si 0 : attachée à un port libre.
- ▶ (Rappel : un machine peut avoir plusieurs adresses !)
- ▶ sin\_addr : l'adresse de la machine
  - ▶ INADDR\_ANY : toutes les adresses possibles de la machine

Note : bind() est optionnel. Si on effectue un listen() ou un connect() sur une socket non affectée, elle sera affectée automatiquement sur un port libre.  
→ OK pour les clients, mais problématique pour les serveurs...

## Connexion (TCP / initiateur de connexion)

```
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 de la machine distant
    ▲ addrlen : taille de la structure addr
    ▲ renvoie 0 (OK), ou -1 (erreur)
```

## Connexion (TCP / initiateur de connexion)

Exemple client simple (web)

## Écouter un port (TCP / receveur de connexion, "serveur")

Rappel : il peut y avoir plusieurs connexions sur un même port :  
C'est la paire (adresse/port hôte 1, adresse/port hôte 2) 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

## Accepter une connexion (TCP / receveur, "serveur")

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

Permet de prendre connaissance des nouvelles connexions

- ▶ **fd** : descripteur de fichier associé à une socket
- ▶ **addr** : pointeur vers une structure `sockaddr_*` où sera copié l'adresse de l'initiateur de la connexion.
- ▶ **addrlen** : est un pointeur sur un entier
  - ▶ elle contient la taille maximum de la structure pointée par `addr`
  - ▶ au retour de la fonction, elle contiendra sa taille effective
- ▶ renvoie un nouveau descripteur de fichier (ou -1 si erreur)
  - ▶ c'est sur ce nouveau descripteur qu'on fera nos opérations d'envoi et d'écoute

Par défaut, `accept` est bloquant. Pour avoir le caractère non bloquant : `fcntl+O_NONBLOCK, select ou poll.`

## Accepter une connexion (TCP / receveur, "serveur")

Si on veut gérer plusieurs connexions en même temps, il faut faire attention au caractère bloquant

Solutions possibles :

- ▶ utiliser plusieurs threads : un thread par connexion
- ▶ utiliser select ou poll
- ▶ passer le descripteur de fichier en mode non bloquant (et trouver une solution pour éviter les attentes actives...)

Exemple serveur simple (web)

## Envoi/réception (TCP)

Les opérations d'envoi de message et de lecture peuvent se faire comme à l'accoutumé avec write et read.

Mais il existe des commandes spécifiques, avec des options en plus : int send(int fd, void \*buffer, size\_t len, int options)

- ▶ fd, buffer, len, retour : comme dans write
- ▶ options :
  - ▶ 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
  - ▶ ...

Note : write(fd, buff, len) est équivalent à  
send(fd, buff, len, 0)

## Envoi/réception (TCP)

- ```
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
    - ▶ ...

Note : read(fd, buff, len) est équivalent à  
recv(fd, buff, len, 0)

## Envoi/réception (UDP)

- ▶ `connect()` sur une socket UDP (DGRAM) définit l'adresse/port où les datagrammes sont envoyés par défaut, et la seule adresse d'où les datagrammes sont acceptés.
- ▶ (pas de `listen()`/`accept()` en UDP !)

Méthode alternative :

```
ssize_t sendto(int sockfd, const void *buf,  
size_t len, int flags,  
const struct sockaddr *dest_addr,  
socklen_t addrlen);
```

Permet d'envoyer directement un datagramme l'adresse `dest_addr`, sans faire de `connect` préalable.

## Envoi/réception (UDP)

```
ssize_t recvfrom(int sockfd, void *buf,  
size_t len, int flags,  
struct sockaddr *src_addr,  
socklen_t *addrlen);
```

Comme `recv()`, et l'adresse source du datagramme sera copiée dans `src_addr`.

`send(sockfd, buf, len, flags)`  
est équivalent à :  
`sendto(sockfd, buf, len, flags, NULL, 0)`

## SOCK\_RAW, AF\_PACKET ...

Pour construire un paquet "brut" :

- ▶ par exemple ICMP (utilisé par ping)
- socket (AF\_INET, SOCK\_RAW, IPPROTO\_ICMP)

- ▶ un packet IP brut :

```
socket (AF_INET, SOCK_RAW, IPPROTO_RAW)
```

- ▶ un paquet Ethernet :

```
socket (AF_PACKET, SOCK_DGRAM, htons(ETH_P_IP))
```

Attention, il faut des droits particuliers :

```
$ getcap /bin/ping  
/bin/ping = cap_net_raw+ep
```

## Fermer une socket

```
int shutdown(int sockfd, int how)
```

Rappel : dans une socket TCP, chaque hôte peut indépendamment signaler la fin de ses envois.

how :

- ▶ SHUT\_RD : fermeture de la réception
- ▶ SHUT\_WR : fermeture de l'émission
- ▶ SHUT\_RDWR : fermeture des deux directions

Puis close() pour fermer la socket !

## htons...

La représentation des entiers n'est pas forcément la même sur la machine et sur internet :

- ▶ Par exemple, les x86 ont une représentation en Little endian (petit-boutiste)
- ▶ La représentation dans les packets internet est en Big endian (grand-boutiste)

Il existe des fonctions de conversion :

- ▶ htons() (Host TO Network Short) :  
entier 16 bits : représentation machine ⇒ représentation réseau
- ▶ htonl() (Host TO Network Long) :  
entier 32 bits : représentation machine ⇒ représentation réseau
- ▶ ntohs() (Network TO Host Short) :  
entier 16 bits : représentation réseau ⇒ représentation machine
- ▶ ntohl() (Network TO Host Long) :  
entier 32 bits : représentation réseau ⇒ représentation machine

## inet\_pton

```
#include <arpa/inet.h>
int inet_pton(int af, const char *src, void *dst);
```

Converti une adresse (IPv4 ou IPv6) du format texte au format binaire

- ▶ af : AF\_INET ou AF\_INET6
- ▶ src : la chaîne de caractère de l'adresse
- ▶ dst : un pointeur sur struct in\_addr ou struct in6\_addr

(Remplace inet\_aton(), qui fonctionne que pour IPv4)

Opération inverse : inet\_ntop()

## Résolution de noms DNS

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

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

void freeaddrinfo(struct addrinfo *res);

const char *gai_strerror(int errcode);
```

Traduit les noms et/ou services.

## Résolution de noms DNS

Paramètres de `getaddrinfo` :

- ▶ `node` : (si non NULL) le nom de la machine (DNS)
- ▶ `service` : (si non NULL) le nom du service (voir `/etc/services`)
- ▶ `hints` : (si non NULL) pointe sur une structure `addrinfo` qui contient les critères de la recherche
- ▶ `res` : où sera copiée la liste chaînée des résultats.

Pourquoi une liste ? Un nom peut avoir plusieurs traductions. Par exemple IPv4 et IPv6...

## Résolution de noms DNS

`getaddrinfo` renvoie 0 si OK, sinon il renvoie un code d'erreur qui peut être transformé en texte par `gai_strerror()`

`freeaddrinfo()` détruit la liste chaînée des résultats

Fonction inverse : `getnameinfo()`

Ancienne fonction (obsolète) : `gethostbyname()`

## Résolution de noms DNS

```
struct addrinfo {  
    int           ai_flags;      // options  
    int           ai_family;     // AF_*  
    int           ai_socktype;   // SOCK_*  
    int           ai_protocol;   // 0, 6, 17 ...  
    socklen_t     ai_addrlen;  
    struct sockaddr *ai_addr;  
    char          *ai_canonname;  
    struct addrinfo *ai_next;  
};
```

## Résolution de noms DNS

```
struct addrinfo *res, *p;
int err=getaddrinfo(av[1],NULL,NULL,&res);
if (err) printf("erreur %s\n",gai_strerror(err));
p=res;
while(p) {
    char hostname[NI_MAXHOST];
    err=gai_getnameinfo(p->ai_addr,p->ai_addrlen,hostname,
    NI_MAXHOST,NULL,0,NI_NUMERICHOST);
    if (err)
        printf("erreur %s\n",gai_strerror(err));
    else
        printf("hostname : %s\n", hostname);
    p=p->ai_next;
}
freeaddrinfo(res);
```

## Option des sockets

```
int getsockopt(int sockfd, int level, int optname,
               void *optval, socklen_t *optlen)
int setsockopt(int sockfd, int level, int optname,
               const void *optval, socklen_t optlen)
```

Permet de lire/modifier les options d'une socket

Par exemple, pour désactiver l'"algorithme de Nagle", qui fait attendre qu'il y ait assez de données avant d'envoyer un packet :

```
int one = 1;
setsockopt(fd, SOL_TCP,
           TCP_NODELAY, &one, sizeof(one));
```

voir man 7 socket, man 7 ip, man 7 tcp...

# Administration réseau sous Linux (vue rapide)

## ifconfig

Liste et configure les interfaces réseau. Exemples :

Afficher toutes les interfaces

```
ifconfig -a
```

Définir l'adresse IP de eth0 (éthernet)

```
ifconfig eth0 up 192.168.0.1/24
```

Changer l'adresse MAC d'une interface :

```
ifconfig eth0 hw ether ff:ff:ff:ff:ff:ff
```

## [iwlist/iwconfig](#)

Liste les informations et configure les interfaces réseau sans-fil.

Exemples :

Lister les réseaux sans fil :

```
iwlist wlan0 scanning
```

Connexion à un point d'accès ouvert

```
iwconfig wlan0 essid nom_reseau
```

## [route](#)

Liste et administre la table de routage statique

Afficher les routes :

```
route
```

Ajout d'une route vers un hôte :

```
route add 192.168.2.4 gw 192.168.1.2
```

Ajout d'une route vers un sous-réseau :

```
route add -net 192.168.3.0.24 gw 192.168.1.5
```

## Afficher les connexions, voir les paquets...

- ▶ **netstat** : Affiche les connexions réseau, statistiques...
- ▶ **iptraf** : Affiche les connexions réseau, statistiques... en interactif
- ▶ **wireshark** : Analyseur de paquets interactif
- ▶ **ngrep** : Fait une recherche (grep) sur les paquets réseau

## autres commandes utiles

### Vérification du réseau :

- ▶ **ping**
- ▶ **traceroute**

### DNS

- ▶ **host**

- ▶ **nslookup**

### Utilitaires

- ▶ **nc** : "TCP/IP swiss army knife"  
(attention : informations transitent en clair)
- ▶ **ssh** : copies de fichier par le réseau, proxy, redirection de port entre différentes machines (chiffré)

## iptables

Outil d'administration du filtrage de paquets IP

Afficher les filtres :

```
iptables -L -v -n
```

Ajouter un filtre (ignorer de tout paquet reçu d'un sous réseau)

```
iptables -A INPUT -s 142.17.0.0/16 -j DROP  
rejet de tout paquet TCP reçu avec port de destination 22
```

```
iptables -D INPUT -p tcp --dport 22 -j REJECT
```

► Énormément de filtres possibles

► Fonctionne avec des listes. On peut choisir de rejeter tout par défaut, sauf les paquets qui matchent...