Communications inter-processus

Les Sockets

Introduction aux sockets

La notion de sockets a été introduite dans les distributions de Berkeley (un fameux système de type UNIX, dont beaucoup de distributions actuelles utilisent des morceaux de code), c'est la raison pour laquelle on parle parfois de sockets BSD (*Berkeley Software Distribution*).

Il s'agit d'un modèle permettant la communication inter processus (*IPC* - Inter Processus Communication) afin de permettre à divers processus de communiquer aussi bien sur une même machine qu'à travers un réseau TCP/IP.

La communication par socket est souvent comparée aux communications humaines. On distingue ainsi deux modes de communication:

- Le mode connecté (comparable à une communication téléphonique), utilisant le protocole TCP. Dans ce mode de communication, une connexion durable est établie entre les deux processus, de telle façon que la socket de destination n'est pas nécessaire à chaque envoi de données.

- Le mode non connecté (analogue à une communication par courrier), utilisant le protocole UDP. Ce mode nécessite l'adresse de destination à chaque envoi, et aucun accusé de réception n'est donné.

Les sockets sont généralement implémentées en langage C, et utilisent des fonctions et des structures disponibles dans la librairie *<sys/socket.h>*.

Position des sockets dans le modèle OSI

Les sockets se situent juste au-dessus de la couche transport du modèle OSI (protocoles UDP ou TCP), elle-même utilisant les services de la couche réseau (protocole IP / ARP)

Modèle des sockets	Modèle OSI
Application utilisant les sockets	Application
	Présentation
	Session
UDP/TCP	Transport
IP/ARP	Réseau
Ethernet, X25, ...	Liaison
	Physique

Déroulement d'une connexion

Comme dans le cas de l'ouverture d'un fichier, la communication par socket utilise un descripteur pour désigner la connexion sur laquelle on envoie ou reçoit les données. Ainsi la première opération à effectuer consiste à appeler une fonction créant un socket et retournant un descripteur (un entier) identifiant de manière unique la connexion. Ainsi ce descripteur est passé en paramètres des fonctions permettant d'envoyer ou recevoir des informations à travers le socket.

L'ouverture d'un socket se fait en deux étapes:

- La création d'un socket et de son descripteur par la fonction *socket()*

- La fonction *bind()* permet de spécifier le type de communication associé au socket (protocole TCP ou UDP)

Un serveur doit être à l'écoute de messages éventuels. Toutefois, l'écoute se fait différemment selon que le socket est en mode connecté (TCP) ou non (UDP).

- **En mode connecté,** le message est reçu d'un seul bloc.
  Ainsi en mode connecté, la fonction *listen()* permet de placer le socket en mode passif (à l'écoute des messages). En cas de message entrant, la connexion peut être acceptée grâce à la fonction *accept()*. Lorsque la connexion a été acceptée, le serveur reçoit les données grâce à la fonction *recv()*.

- **En mode non connecté**, comme dans le cas du courrier, le destinataire reçoit le message petit à petit (la taille du message est indéterminée) et de façon désordonnée.
  Le serveur reçoit les données grâce à la fonction *recvfrom()*.

La fin de la connexion se fait grâce à la fonction *close()*.

Schéma d'une communication en mode connecté

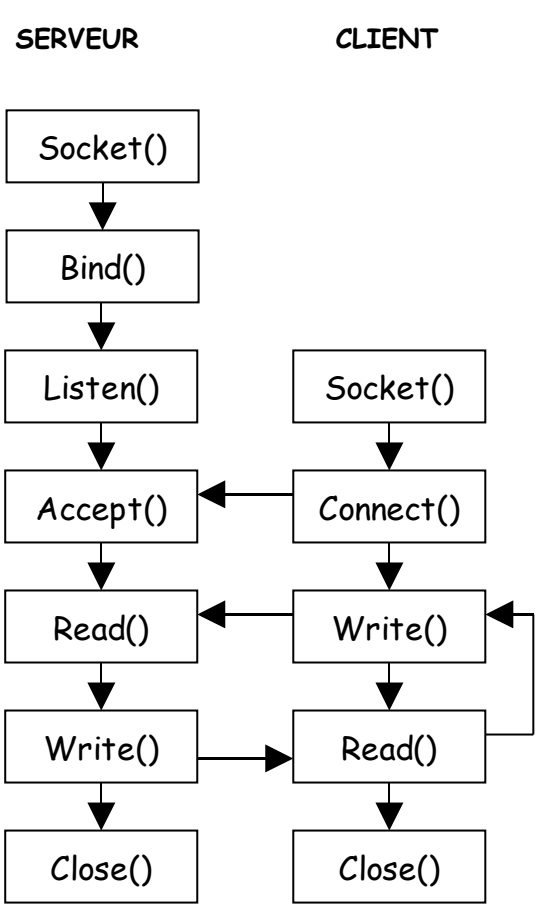
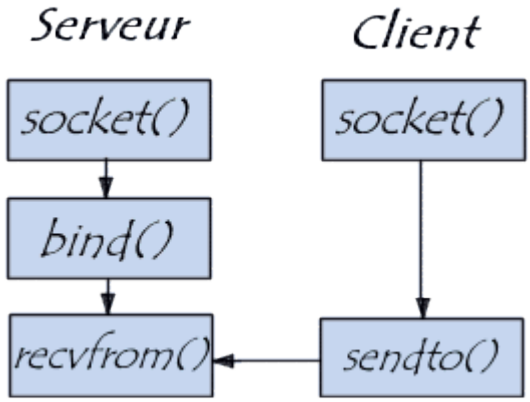


Schéma d'une communication en mode non connecté



Les fonctions des sockets en détail

La fonction socket()

La création d'un socket se fait grâce à la fonction *socket()*:

int socket(famille,type,protocole)

- **famille** représente la famille de protocole utilisé (*AF_INET* ou PF_INET pour TCP/IP utilisant une adresse Internet sur 4 octets: l'adresse IP ainsi qu'un numéro de port afin de pouvoir avoir plusieurs sockets sur une même machine, *AF_UNIX* pour les communications UNIX en local sur une même machine)

- **type** indique le type de service (orienté connexion ou non). Dans le cas d'un service orienté connexion (c'est généralement le cas), l'argument *type* doit prendre la valeur SOCK_STREAM (communication par flot de données). Dans le cas contraire (protocole UDP) le paramètre *type* doit alors valoir SOCK_DGRAM (utilisation de datagramme, blocs de données). Il existe aussi les types SOCK_RAW, orienté vers l'échange de datagrammes au niveau bas du protocole (ie IP), SOCK_RDM, orienté vers la transmission de datagrammes en mode non connecté avec maximum de fiabilité et SOCK_SEQPACKET, orienté vers la transmission de datagrammes en mode connecté.

- **protocole** permet de spécifier un protocole permettant de fournir le service désiré. Dans le cas de la suite TCP/IP il n'est pas utile, on le mettra ainsi toujours à 0

La fonction *socket()* renvoie un entier qui correspond à un descripteur du socket nouvellement créé et qui sera passé en paramètre aux fonctions suivantes. En cas d'erreur, la fonction *socket()* retourne -1.

Voici un exemple d'utilisation de la fonction *socket()* :

```
descripteur = socket(PF_INET,SOCK_STREAM,0);
```

La fonction bind()

Après création du socket, il s'agit de le lier à un point de communication local défini par une adresse et un port, c'est le rôle de la fonction *bind()*:

bind(int descripteur,sockaddr localaddr,int addrlen)

- **descripteur** représente le descripteur du socket nouvellement créé

- **localaddr** est une structure qui spécifie l'adresse locale à travers laquelle le programme doit communiquer Le format de l'adresse est fortement dépendante du protocole utilisé :

    - l'interface socket définit une **structure standard** (*sockaddr* définie dans <*sys/socket.h*>) permettant de représenter une adresse:

```
struct sockaddr {
/* longueur effective de l'adresse */
u_char sa_len;

/* famille de protocole (generalement AF_INET) */
u_char sa_family;

/* l'adresse complete */
char sa_data[14];

}
```

        - **sa_len** est un octet (u_char) permettant de définir la longueur utile de l'adresse (la partie réellement utilisée de *sa_data*)

        - **sa_family** représente la famille de protocole (AF_INET pour TCP/IP)

        - **sa_data** est une chaîne de de 14 caractères (au maximum) contenant l'adresse

- La **structure utilisé avec TCP/IP** est une adresse AF_INET (Généralement les structures d'adresses sont redéfinies pour chaque famille d'adresse). Les adresses AF_INET utilisent une structure *sockaddr_in* définie dans <*netinet/in.h*> :

```
struct sockaddr_in {
/* famille de protocole (AF_INET) */
short sin_family;

/* numero de port */
u_short sin_port;

/* adresse internet */
struct in_addr sin_addr;

char sin_zero[8];    /* initialise a zero */

}
```

  - **sin_family** représente le type de famille
  - **sin_port** représente le port à contacter
  - **sin_addr** adresse de l'hôte
  - **sin_zero[8]** contient uniquement des zéros (étant donné que l'adresse IP et le port occupent 6 octets, les 8 octets restants doivent être à zéro

- **addrlen** indique la taille du champ *localaddr*. On utilise généralement *sizeof(localaddr)*.

Voici un exemple d'utilisation de la fonction *bind()* :

```
sockaddr_in localaddr ;

localaddr.sin_family = AF_INET; /* Protocole internet */

/* Toutes les adresses IP de la station */
localaddr.sin_addr.s_addr = htonl(INADDR_ANY);

/* port d'écoute par défaut au dessus des ports réservés */
localaddr.sin_port = htons(port);
if (bind(listen_socket,
         struct sockaddr*)&localaddr,
         sizeof(localaddr) )  == SOCKET_ERROR) {
          // Traitement de l'erreur;

}
```

Il existe 4 possibilités d'utiliser le Bind() :

1) en spécifiant l'adresse IP et le port number,
2) en spécifiant l'adresse IP et en laissant le système choisir un port number libre inutilisé (0),
3) en utilisant l'adresse IP : *INADDR_ANY* qui signifie que la socket peut-être associée à n'importe quelle adresse IP de la machine locale (s'il en existe plusieurs) et à un port number spécifique.
4) En laissant le système choisir une adresse IP et un port number quelconque (on ne code pas le bind). Cette forme est utilisée lorsqu'un client n'est intéressé que par l'adresse du serveur et non par une adresse locale.

Pour spécifier une adresse IP spécifique à utiliser, il est possible d'utiliser la fonction *inet_addr()*

```
inet_addr("127.0.0.1");

/* utilisation de l'adresse de boucle locale */
```

La fonction listen()

La fonction *listen()* permet de mettre une socket en attente de connexion.

La fonction *listen()* ne s'utilise qu'en mode connecté (donc avec le protocole TCP)

int listen(int socket,int backlog)

- **socket** représente la socket précédemment ouverte
- **backlog** représente le nombre maximal de connexions pouvant être mises en attente

La fonction *listen()* retourne la valeur *SOCKET_ERROR* en cas de problème, sinon elle retourne 0.

Voici un exemple d'utilisation de la fonction *listen()* :

```
if (listen(socket,10) == SOCKET_ERROR) {

// traitement de l'erreur

}
```

La fonction accept()

La fonction *accept()* permet la connexion en acceptant un appel :

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

- **socket** représente la socket précédemment ouverte (la socket locale)
- **addr** représente un tampon destiné à stocker l'adresse de l'appelant
- **addrlen** représente la taille de l'adresse de l'appelant

La fonction *accept()* retourne un identificateur du socket de réponse. Si une erreur intervient la fonction *accept()* retourne la valeur *INVALID_SOCKET*.

Voici un exemple d'utilisation de la fonction *accept()* :

```
Sockaddr_in Appelant;

/* structure destinee a recueillir les renseignements sur l'appelant

Appelantlen = sizeof(from);

accept(socket_locale,(struct sockaddr*)&Appelant, &Appelantlen);
```

La fonction connect()

La fonction *connect()* permet d'établir une connexion avec un serveur :

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

- **socket** représente la socket précédemment ouverte (la socket à utiliser)
- **addr** représente l'adresse de l'hôte à contacter. Pour établir une connexion, le client ne nécessite pas de faire un *bind()*
- **addrlen** représente la taille de l'adresse de l'hôte à contacter

La fonction *connect()* retourne 0 si la connexion s'est bien déroulée, sinon -1.

Voici un exemple d'utilisation de la fonction *connect()*, qui connecte le socket "s" du client sur le port *port* de l'hôte portant le nom *serveur* :

```
toinfo = gethostbyname(serveur);

toaddr = (u_long *)toinfo.h_addr_list[0];

/* Protocole internet */
to.sin_family = AF_INET;
```

```
/* Toutes les adresses IP de la station */
to.sin_addr.s_addr = toaddr;

/* port d'écoute par défaut au dessus des ports réservés */
to.sin_port = htonl(port);

if (connect(socket,(struct sockaddr*)to,sizeof(to)) == -1) {

// Traitement de l'erreur;

}
```

La fonction read()

La fonction *read()* permet de lire dans un socket en mode connecté (TCP)

int read(int socket,char * buffer,int len)

- **socket** représente la socket précédemment ouverte
- **buffer** représente un pointeur sur un tampon qui recevra les octets en provenance du client
- **len** indique le nombre d'octets lu

La fonction *read()* renvoie le nombre d'octets lus. De plus cette fonction bloque le processus jusqu'à ce qu'elle reçoive des données

Voici un exemple d'utilisation de la fonction *read()* :

```
retour = read(socket,Buffer,sizeof(Buffer));

if (retour == -1) {

// traitement de l'erreur

}
```

La fonction write()

La fonction *write()* permet d'écrire dans un socket (envoyer des données) en mode connecté (TCP)

int write(int socket,char * buffer,int len)

- **socket** représente la socket précédemment ouverte
- **buffer** représente l'adresse du tampon contenant les octets à envoyer au client
- **len** indique le nombre d'octets à envoyer

La fonction *write()* renvoie le nombre d'octets effectivement envoyés.

Voici un exemple d'utilisation de la fonction *write()* :

```
retour = write(socket,Buffer,sizeof(Buffer));

if (retour == SOCKET_ERROR) {

// traitement de l'erreur

}
```

La fonction recv()

La fonction *recv()* permet de lire dans un socket en mode connecté (TCP)

int recv(int socket,char * buffer,int len,int flags)

- **socket** représente la socket précédemment ouverte
- **buffer** représente un tampon qui recevra les octets en provenance du client
- **len** indique le nombre d'octets à lire
- **flags** correspond au type de lecture à adopter (OU binaire entre les valeurs suivantes) :
    - *MSG_OOB* permet la lecture des données hors-bande qui ne seraient autrement pas placées dans le flux de données normales. Certains protocoles placent ces données hors-bande en tête de la file normale, et cette option n'a pas lieu d'être dans ce cas.
    - *MSG_PEEK* permet de lire les données en attente dans la file sans les enlever de cette file. Ainsi une lecture ultérieure renverra à nouveau les mêmes données.
    - *MSG_WAITALL* demande que l'opération de lecture soit bloquée jusqu'à ce que la requête complète soit satisfaite. Toutefois la lecture peut renvoyer quand même moins de données que prévu si un signal est reçu, ou si une erreur ou une déconnexion se produisent.
    - *MSG_NOSIGNAL* désactive l'émission de SIGPIPE sur les sockets connectées dont le correspondant disparaît.
    - *MSG_ERRQUEUE* demande de lire les erreurs provenant de la file d'erreur de la socket. Les erreurs sont transmises dans un message dont le type dépend du protocole (IP_RECVERR pour IPv4). Il faut alors fournir un buffer de taille suffisante.
    - le flag *0* indique une lecture normale

La fonction *recv()* renvoie le nombre d'octets lus. De plus cette fonction bloque le processus jusqu'à ce qu'elle reçoive des données

Voici un exemple d'utilisation de la fonction *recv()* :

```
retour = recv(socket,Buffer,sizeof(Buffer),0 );

if (retour == SOCKET_ERROR) {

// traitement de l'erreur

}
```

La fonction send()

La fonction *send()* permet d'écrire dans un socket (envoyer des données) en mode connecté (TCP)

int send(int socket,char * buffer,int len,int flags)

- **socket** représente la socket précédemment ouverte
- **buffer** représente un tampon contenant les octets à envoyer au client
- **len** indique le nombre d'octets à envoyer
- **flags** correspond au type d'envoi à adopter :
    - le flag *MSG_DONTROUTE* est utilisé pour empêcher la transmission d'un paquet vers une passerelle, n'envoyer de données que vers les hôtes directement connectés au réseau. Ceci n'est normalement employé que par les programmes de diagnostique ou de routage. Cette option n'est définie que pour les familles de protocoles employant le routage, pas les sockets par paquets.
    - le flag *MSG_OOB* indiquera que les données urgentes (*Out Of Board*) doivent être envoyées
    - le flag *MSG_DONTWAIT* active le mode non-bloquant. Une opération qui devrait bloquer renverra EAGAIN à la place (Cela peut être également paramétré avec l'option O_NONBLOCK de la fonction F_SETFL de fcntl(2)).
    - Le flag *MSG_NOSIGNAL* demande de ne pas envoyer de signal SIGPIPE d'erreur sur les sockets connectées lorsque le correspondant coupe la connexion. L'erreur EPIPE est toutefois renvoyée.
    - le flag *0* indique un envoi normal

La fonction *send()* renvoie le nombre d'octets effecivement envoyés.

Voici un exemple d'utilisation de la fonction *send()* :

```
retour = send(socket,Buffer,sizeof(Buffer),0 );

if (retour == SOCKET_ERROR) {

// traitement de l'erreur

}
```

La fonction recvfrom()

La fonction *recvfrom()* permet de lire dans un socket en mode non connecté (UDP)

int recvfrom(int socket,char * buf, int len, int flags, sockaddr * from, int * frlen)

- **socket** représente la socket précédemment ouverte
- **buf** représente un tampon qui recevra les octets en provenance du client
- **len** indique le nombre d'octets à lire
- **flags** correspond au type de lecture à adopter:
    - *MSG_OOB* permet la lecture des données hors-bande qui ne seraient autrement pas placées dans le flux de données normales. Certains protocoles placent ces données hors-bande en tête de la file normale, et cette option n'a pas lieu d'être dans ce cas.
    - *MSG_PEEK* permet de lire les données en attente dans la file sans les enlever de cette file. Ainsi une lecture ultérieure renverra à nouveau les mêmes données.
    - *MSG_WAITALL* demande que l'opération de lecture soit bloquée jusqu'à ce que la requête complète soit satisfaite. Toutefois la lecture peut renvoyer quand même moins de données que prévu si un signal est reçu, ou si une erreur ou une déconnexion se produisent.
    - *MSG_NOSIGNAL* désactive l'émission de SIGPIPE sur les sockets connectées dont le correspondant disparaît.
    - *MSG_ERRQUEUE* demande de lire les erreurs provenant de la file d'erreur de la socket. Les erreurs sont transmises dans un message dont le type dépend du protocole (IP_RECVERR pour IPv4). Il faut alors fournir un buffer de taille suffisante.
    - le flag *0* indique une lecture normale
- **from** correspond à l'adresse d'une structure qui contiendra l'adresse de l'émetteur
- **frlen** adresse d'un entier qui indique la taille de la structure de l'adresse de l'émetteur

La fonction *recvfrom()* renvoie le nombre d'octets lus. De plus cette fonction bloque le processus jusqu'à ce qu'elle reçoive des données

Voici un exemple d'utilisation de la fonction *recv()* :

```
retour = recvfrom(socket,Buffersizeof(Buffer),0,sockaddr *from,&sizeof(sockaddr));

if (retour == SOCKET_ERROR) {

// traitement de l'erreur

}
```

La fonction sendto()

La fonction *sendto()* permet d'écrire dans un socket (envoyer des données) en mode non connecté (UDP)

int sendto(int socket, char * buffer, int len, int flags, sockaddr * to, int tolen)

- **socket** représente la socket précédemment ouverte
- **buffer** représente un tampon contenant les octets à envoyer au client

- **len** indique le nombre d'octets à envoyer
- **flags** correspond au type d'envoi à adopter :
    - le flag *MSG_DONTROUTE* est utilisé pour empêcher la transmission d'un paquet vers une passerelle, n'envoyer de données que vers les hôtes directement connectés au réseau. Ceci n'est normalement employé que par les programmes de diagnostique ou de routage. Cette option n'est définie que pour les familles de protocoles employant le routage, pas les sockets par paquets.
    - le flag *MSG_OOB* indiquera que les données urgentes (*Out Of Board*) doivent être envoyées
    - le flag *MSG_DONTWAIT* active le mode non-bloquant. Une opération qui devrait bloquer renverra EAGAIN à la place (Cela peut être également paramétré avec l'option O_NONBLOCK de la fonction F_SETFL de fcntl(2)).
    - Le flag *MSG_NOSIGNAL* demande de ne pas envoyer de signal SIGPIPE d'erreur sur les sockets connectées lorsque le correspondant coupe la connexion. L'erreur EPIPE est toutefois renvoyée.
    - le flag *0* indique un envoi normal
- **to** correspond à l'adresse d'une structure qui contiendra l'adresse du destinataire
- **tolen** entier qui indique la taille de la structure de l'adresse de l'émetteur

La fonction *sendto()* renvoie le nombre d'octets effecivement envoyés.

Voici un exemple d'utilisation de la fonction *sendto()* :

```
retour = sendto(socket,Buffer,sizeof(Buffer),0,sockaddr *to, sizeof(sockaddr));

if (retour == SOCKET_ERROR) {

// traitement de l'erreur

}
```

Les fonctions recvmsg() et sendmsg()

La fonction *recvmsg()* reçoit le prochain message qui arrive sur un socket UDP et le place dans une structure qui contient un en-tête et les données.

int recvmsg(int socket, struct msghdr *msg, int flags)

La fonction *sendmsg()* envoie un message extrait d'une structure msghdr.

int sendmsg(int socket, struct msghdr *msg, int flags)

Les fonctions close() et shutdown()

La fonction *close()* permet la fermeture d'un socket en permettant au système d'envoyer les données restantes (pour TCP) :

int close(int socket)

La fonction *shutdown()* permet la fermeture d'un socket dans un des deux sens (pour une connexion full-duplex) :

int shutdown(int socket,int how)

- Si *how* est égal à 0, la socket est fermée en réception
- Si *how* est égal à 1, la socket est fermée en émission
- Si *how* est égal à 2, la socket est fermée dans les deux sens

close() comme *shutdown()* retournent -1 en cas d'erreur, 0 si la fermeture se déroule bien.

La fonction select()

La fonction *select()* attend des changements d'état sur plusieurs descripteurs de fichiers.

int select(int n, fd_set readfds, fd_set*writefds, fd_set*exceptfds, struct timeval *timeout)

```
FD_CLR(int fd, fd_set *set);
FD_ISSET(int fd, fd_set *set);
FD_SET(int fd, fd_set *set);
FD_ZERO(fd_set *set);
```

Il y a trois ensembles indépendants de descripteurs qui sont surveillés simultanément.

Ceux de l'ensemble *readfds* seront surveillés pour vérifier si des caractères deviennent disponibles en lecture. Plus précisément, on vérifie si un appel-système de lecture ne bloquera pas - en particulier un descripteur en fin-de-fichier sera considéré comme prêt.

Les descripteurs de l'ensemble *writefds* seront surveillés pour vérifier si une écriture ne bloquera pas.

Ceux de l'ensemble *exceptfds* seront surveillés pour l'occurence de conditions exceptionnelles. Il en existe deux types : l'arrivée de données hors-bande sur une socket, et la disponibilité d'informations d'état concernant un pseudo-terminal en mode paquet.

On peut indiquer un pointeur NULL à la place d'un ensemble si l'on ne veut pas en tenir compte.

En sortie, les ensembles sont modifiés pour indiquer les descripteurs qui ont changé de statut.

Quatre macros sont disponibles pour la manipulation des ensembles **FD_ZERO** efface un ensemble. **FD_SET** et **FD_CLR** ajoutent et suppriment un descripteur dans un ensemble. **FD_ISSET** vérifie si un descripteur est contenu dans un ensemble, principalement utile après le retour de **select.**

n est le numéro du plus grand descripteur des 3 ensembles, plus 1.

timeout est une limite supérieure au temps passé dans **select** avant son retour. Elle peut être nulle, ce qui conduit **select** à revenir immédiatement. Si le timeout est NULL (aucun), **select** peut bloquer indéfiniment.

En cas de réussite **select** renvoie le nombre de descripteurs dans les ensembles, qui peut être nul si le délai de timeout a expiré avant que quoi que ce soit d'intéressant ne se produise. **select** retourne -1 s'il échoue, auquel cas *errno* contient le code d'erreur.

Principales fonctions systèmes utilisées avec les sockets

struct hostent * gethostbyaddr(&addr addr, int len, int type)

La fonction **gethostbyaddr()** renvoie une structure du type *hostent* pour l'hôte d'adresse *addr*. Cette adresse est de longueur *len* et du *type* donné. Le seul type d'adresse valide est actuellement **AF_INET**.

struct hostent *gethostbyname(const char *name);

La fonction **gethostbyname()** renvoie une structure de type *hostent* pour l'hôte *name*. La chaîne *name* est soit un nom d'hôte, soit une adresse IPv4 en notation pointée standard, soit une adresse IPv6 avec la notation points-virgules et points. Si *name* est une adresse IPv4 ou IPv6, aucune recherche supplémentaire n'a lieu et **gethostbyname**() copie simplement la chaine *name* dans le champ *h_name* et le champs équivalent *struct* in_addr dans le champs *h_addr_list[0]* de la structure *hostent* renvoyée.

La structure *hostent* est définie ainsi dans <*netdb.h*> :

```
struct hostent {
   char    *h_name;       /* Nom officiel de l'hôte.    */
   char   **h_aliases;    /* Liste d'alias.             */
   int      h_addrtype;   /* Type d'adresse de l'hôte.  */
   int      h_length;     /* Longueur de l'adresse.     */

   char   **h_addr_list;  /* Liste d'adresses.          */
}
```

long int gethostid(void)

int sethostid (long int *hostid*);

Lit ou écrit un identifiant unique sur 32 bits pour la station utilisée. Cet identificateur est censé être unique pour l'ensemble des stations UNIX existantes. Il correspond généralement à l'adresse Internet de la machine locale, comme renvoyé par gethostbyname() et est normalement fixé une fois pour toutes.

int gethostname(char **name*, int *len*);
int sethostname(const char **name*, int *len*);

Ces fonctions sont utilisées pour lire, ou changer le nom d'hôte de la machine utilisée.
gethostname et **sethostname** renvoient 0 s'ils réussissent, ou -1 s'ils échouent, auquel cas *errno* contient le code d'erreur.

int getpeername(int s, struct sockaddr * remoteaddr, int *namelen)

Getpeername retourne le nom du correspondant connecté sur une socket *s*. Le paramètre *namelen* doit être initialisé pour indiquer la taille de la zone pointée par *remoteaddr*. En retour, il contiendra la longueur effective (en octets) du nom retourné. Le nom est tronqué si le tampon est trop petit.

int gettimeofday(struct timeval **tv*, struct timezone **tz*);
int settimeofday(const struct timeval **tv* , const struct timezone **tz*);

gettimeofday et **settimeofday** peuvent programmer l'heure ainsi que le fuseau horaire (timezone).
tv est une structure **timeval** décrite dans /usr/include/sys/time.h :

```
struct timeval {
int   tv_sec;      /* secondes */
int   tv_usec;   /* microsecondes */
}
```

et *tz* est une structure **timezone** décrite egalement dans /usr/include/sys/time.h:

```
struct timezone {
int    tz_minuteswest; /* minutes west of Greenwich  */
int    tz_dsttime;     /* type de changement horaire */
}
```

Sous Linux, durant un appel à **settimeofday** le champ *tz_dsttime* doit être nul.

gettimeofday et **settimeofday** renvoient 0 s'ils réussissent, ou -1 s'ils échouent, auquel cas *errno* contient le code d'erreur.