UNIVERSITE A.MIRA-BEJAIA
FACULTE DES SCIENCES EXACTES
DEPARTEMENT INFORMATIQUE

Decembre 2025

TRAVAUX PRATIQUES
SYSTEMES D'EXPLOITATION
Programmation par Sockets

I. Introduction

1. Définitions :

Les sockets (interface socket) sont utilisées pour faire communiquer deux processus sur deux machines distantes. Le processus émetteur prépare un message dans son espace d'adressage puis exécute un appel système pour transmettre le message à un processus récepteur. La communication est bidirectionnelle et utilise soit le mode connecté (TCP) soit le mode non connecté (UDP).

2. Type d'une socket : Le type d'une socket détermine la sémantique des communications qu'elle permet de réaliser. L'ensemble des propriétés d'une communication dépend essentiellement du protocole de transport utilisé.

- Une communication nécessitant l'échange de messages complets et structurés emploie une socket de type SOCK_DGRAM (datagramme), le protocole sous-jacent est UDP.
- Une communication nécessitant l'échange de flots d'information emploie une socket de type SOCK_STREAM (stream ou flot), le protocole sous-jacent est TCP.
- Une communication sans passer par la couche transport (sans préciser l'adresse IP) emploie une socket de type SOCK_RAW.

3. Programmation par Sockets :

La programmation par socket est un moyen de connecter deux nœuds sur un réseau pour communiquer entre eux. Une socket (nœud) écoute sur un port particulier à une adresse IP, tandis que l'autre socket tend la main vers l'autre pour établir une connexion. Le serveur forme le socket d'écoute tandis que le client s'adresse au serveur .
La figure1 représente les étapes du processus client et du processus serveur. Si une requête du client arrive avant listen() le client reçoit un code d'erreur, sinon le serveur traite la requête et si elle est acceptée le serveur se bloque. Une communication s'établit alors entre les deux processus. A la fin de la connexion le serveur se remet en position d'écoute.

4. Domaine d'une socket

Le domaine d'une socket détermine l'ensemble des autres points de communication qu'elle permet d'atteindre (et donc les processus qui les utilisent).
Il existe plusieurs domaines dont les deux principaux sont :

- AF_UNIX/AF_LOCAL : le domaine local qui permet d'atteindre les processus s'exécutant sur la même machine,
- AF_INET : le domaine de l'internet qui permet d'atteindre les processus s'exécutant sur une des machines de l'internet (IPv4).

Dans le domaine AF_INET, une socket appartient à une machine et est identifiée par un port, l'adresse d'une telle socket est définie dans le fichier standard <netinet/in.h> comme correspondant à la structure suivante :

```
#include <netinet/in.h>
struct sockaddr_in {
      short sin_family; /* AF_INET */
```

```
    u_short sin_port; /* numéro du port associe */
    struct in_addr sin_addr; /* adresse internet de la machine */
    char sin_zero[8]; /* tableau de 8 caractères nuls */
};
```

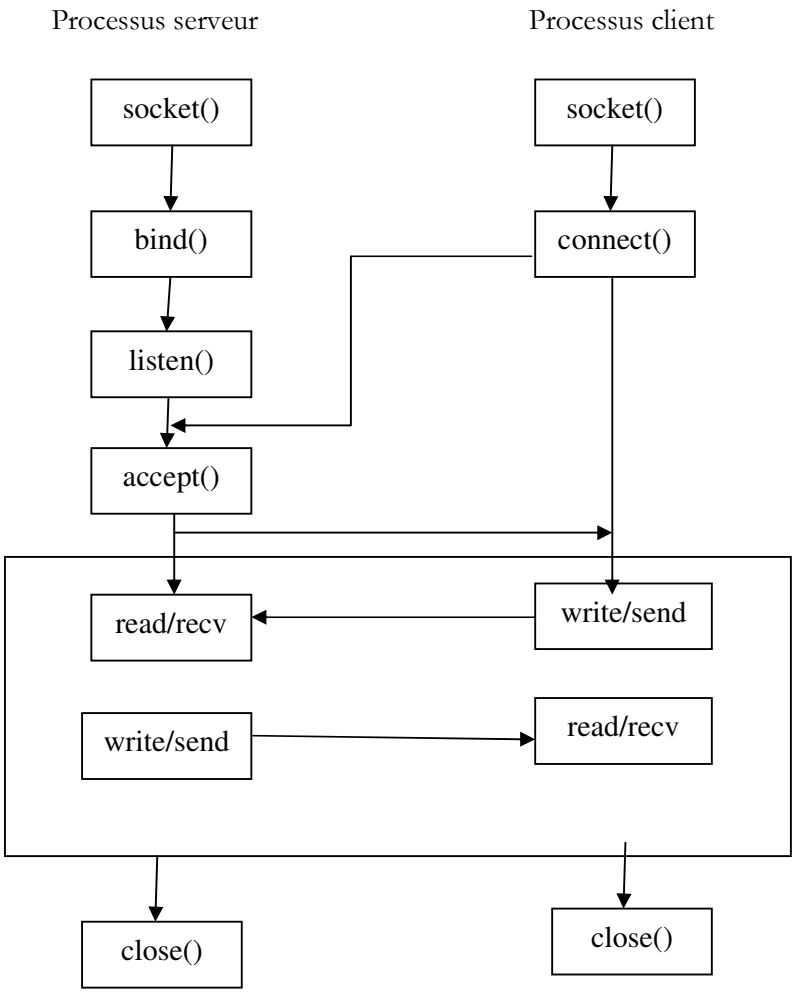


Figure1. Diagramme d'état pour le modèle de serveur et de client

5. Creation de Sockets:

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

```
int sockfd = socket(domaine, type, protocole);
```

sockfd: descripteur de fichier de la socket de type entier positif (-1 si echec)

domaine: un entier qui spécifie le domaine de communication.
AF_ LOCAL (AF_UNIX): pour une communication entre processus sur le même hôte.
AF_INET : pour une communication entre processus sur différent hôtes relies par IPv4,
AF_I NET 6 : s'ils sont relies par IPv6.

type: communication type
SOCK_STREAM: TCP(connexion en mode connectée, fiable)
SOCK_DGRAM: UDP(connexion en mode non-connectée, non fiable)
SOCK_RAW : IP ou ICMP (connexion sans couche session)

protocole:
IPPROTO_TCP ou IPPROTO_UDP : Une valeur indiquant le protocole. C'est la même valeur qui apparaît sur le champ de protocole dans l'en-tête IP d'un paquet. En général on met un 0 et le système vérifie le type pour savoir quel protocole est utilisé.

6. L'attachement d'une socket à une adresse (Bind):
Il est nécessaire d'attacher une adresse de son domaine à l'objet créé au moyen de la primitive bind. Sans nommage explicite de la socket, seuls les processus ayant hérité du descripteur sur la socket peuvent lire ou écrire sur la socket (car elle est duplex), mais personne ne pourrait envoyer de l'extérieur des données sur la socket.

```
#include <sys/types.h>
#include <sys/socket.h>
int bind(int sockfd, const struct sockaddr *addr, socklen_t
addrlen);
```

sockaddr: pointeur sur l'adresse à utiliser addr : @IP+numéro de port
socklen_t :type de la longueur de l'adresse (souvent de type entier int long=sizeof(addr))

Après la création du socket, la fonction bind lie la socket à l'adresse et au numéro de port spécifiés dans addr (structure de données personnalisée). Pour lier le serveur à l'hôte local, il faut utiliser l'adresse de bouclage, ou l'adresse privée sinon pour un serveur il devrait utiliser toutes les adresses disponibles INADDR_ANY.

7. La fermeture de la connexion (close)
L'appel de la primitive close sur le dernier descripteur d'une socket entraîne la suppression de la socket.

```
#include <unistd.h>
int close(int sockfd);
```

Cette suppression libère toutes les ressources allouées.
Cette primitive peut être bloquante dans le cas d'utilisation d'une socket de type SOCK_STREAM dans le domaine AF_INET au cas où le tampon d'émission n'est pas vide.

II. La communication en mode connecté (TCP)
1. Principes
C'est le mode de communication associé aux sockets de type SOCK_STREAM. Il permet d'échanger des séquences de caractères continues et non structurées en messages.
Dans le cas du domaine AF_INET, le protocole sous-jacent est TCP, la communication est donc fiable.

2. Le point de vue du serveur
2.1 Introduction
Le rôle du serveur est passif pendant l'établissement de la connexion :

- le serveur doit disposer d'une socket d'écoute ou socket de rendez-vous attachée au port TCP correspondant au service (et donc supposé connu des clients).

- le serveur doit aviser le système auquel il appartient qu'il est prêt à accepter les demandes de connexion, sur la socket de rendez-vous (par la primitive listen).
- puis le serveur se met en attente de connexion sur la socket de rendez-vous (par la primitive bloquante accept).
- lorsqu'un client prend l'initiative de la connexion, le processus serveur est débloqué et une nouvelle socket, appelée socket de service, est créée : c'est cette socket de service qui est connectée à la socket du client.
- le serveur peut alors déléguer le travail à un nouveau processus créé (ou thread) et reprendre son attente sur la socket de rendez-vous, ou traiter lui-même la demande.

2.2 L'ouverture du service : listen

```
#include <sys/types.h>
#include <sys/socket.h>
int listen(
int s, /* socket de rendez-vous */
int backlog /* nombre maximum de connexions pendantes */
);
```

Cette primitive permet à un serveur de déclarer un service ouvert auprès de son entité TCP locale. Il met le socket du serveur en mode passif, où il attend que le client demande une connexion au serveur.
Après cet appel, l'entité TCP locale commence à recevoir les demandes de connexions (appelées des connexions pendantes), le paramètre backlog définit la taille d'une file d'attente où sont mémorisées les connexions pendantes.
Si une demande de connexion arrive lorsque la file d'attente est pleine, le client peut recevoir une erreur avec une indication ECONNREFUSED.
La valeur de retour est 0 en cas de réussite. En cas d'erreur, la valeur de retour est -1 et errno vaut :

- EBADF : s est un descripteur invalide,
- ENOTSOCK : s n'est pas une socket,
- EOPNOTSUPP : type de socket incorrect.

2.3 La prise en compte d'une connexion : accept

```
#include <sys/types.h>
#include <sys/socket.h>
int accept(
int s, /* socket de rendez-vous */
struct sockaddr *addr, /* ptr sur l'addr de la socket connectee */
int *addrlen /* pointeur sur la longueur de l'adresse */
);
```

Cette primitive permet à un serveur d'extraire une connexion pendante de la file d'attente associée à la socket de rendez-vous s. Ainsi le serveur prend en compte un nouveau client : une nouvelle socket de service est créée, est connectée au client et son descripteur est renvoyé par la primitive accept.
L'adresse de la socket du client avec lequel la connexion est établie est écrite dans la zone pointée par addr si sa valeur est différente de NULL (la longueur est alors donnée par *addrlen).
Un appel à accept est normalement bloquant s'il n'y a aucune connexion pendante. En cas de succès, il renvoie un entier non-négatif, constituant un descripteur pour la nouvelle socket. En cas d'erreur, la valeur de retour est -1 et errno vaut :

- EBADF : s est un descripteur invalide,

- EINTR : le processus appelant accept a reçu un signal avant qu'un des clients attendus soit arrivé, et le signal a interrompu l'appel-système,
- ENOTSOCK : s n'est pas une socket,
- EOPNOTSUPP : type de socket incorrect
- EWOULDBLOCK : la socket est non-bloquante et il n'y a pas de connexions pendantes à accepter,
- ... (peut dépendre du système utilisé).

3 Le point de vue du client

3.1 Introduction

Le rôle du client est actif pendant l'établissement de la connexion :

- le client doit disposer d'une socket attachée à un port TCP quelconque.
- le client doit construire l'adresse du serveur, dont il doit connaître le numéro de port de la socket de rendez-vous, (il obtient éventuellement l'adresse IP du serveur à partir de son nom par la primitive gethostbyname).
- puis le client demande la connexion de sa socket locale à la socket de rendez-vous du serveur (par la primitive bloquante connect).
- lorsque le serveur accepte la connexion, le processus client est débloqué et peut dialoguer avec le serveur.

3.2 La demande de connexion : connect

```
#include <sys/types.h>
#include <sys/socket.h>
int connect(
int sockfd, /* socket locale */
struct sockaddr *serv_addr, /* pointeur sur l'adresse du serveur */
int addrlen /* longueur de l'adresse */
);
```

L'appel système connect() connecte la socket référencée par le descripteur de fichier sockfd à l'adresse spécifiée par addr. L'adresse et le port du serveur sont spécifiés dans addr. Le processus appelant connect est bloqué jusqu'à ce que la négociation entre les deux entités TCP concernées soit achevée.

La demande de connexion réussit, et la primitive renvoie 0 si les conditions suivantes sont réalisées :
1. les paramètres sont localement corrects,
2. il existe une socket de type SOCK_STREAM attachée à l'adresse *nom (dans le même domaine que la socket locale s) et cette socket est dans l'état LISTEN (c'est-à-dire qu'un appel à listen a été réalisé pour cette socket),
3. la socket locale s n'est pas déjà connectée,
4. la socket d'adresse *nom n'est pas actuellement utilisée dans une autre connexion,
5. la file des connexions pendantes de la socket distante n'est pas pleine.

Dans le cas où l'une des quatre premières conditions n'est pas remplie, la fonction renvoie -1 et errno vaut :

- EBADF : s est un descripteur invalide,
- EINTR : le processus appelant connect a reçu un signal avant que la connexion aie eu lieu, et le signal a interrompu l'appel-système,
- ENOTSOCK : s n'est pas une socket,
- EOPNOTSUPP : type de socket incorrect

- EISCONN : la socket locale est déjà connectée,
- EADDRINUSE : la socket distante est déjà connectée,
- ... (peut dépendre du système utilisé).

Si la file d'attente est pleine, le comportement est particulier :

- Si la socket locale est en mode bloquant, le processus est bloqué. La demande de connexion est itérée pendant un certain temps : si au bout de ce laps de temps la connexion n'a pas pu être établie, la demande est abandonnée. La fonction renvoie alors -1 et errno vaut ETIMEDOUT.
- Si la socket locale est en mode non-bloquant, le retour est immédiat avec la valeur de retour -1 et errno qui vaut EINPROGRESS. Cependant ce retour ne correspond pas à une véritable erreur, la demande de connexion n'est pas abandonnée mais automatiquement réitérée comme dans le mode bloquant.

Pour déterminer si la connexion a pu être établie, le processus pourra utiliser la primitive select en testant le descripteur en écriture.
Si une autre demande de connexion est formulée avant que cet essai de connexion soit terminé, la fonction renvoie -1 et errno vaut EALREADY.

4 Le dialogue client/serveur

4.1 Introduction

Une fois la connexion établie, la communication redevient symétrique.
Les deux processus peuvent échanger des suites de caractères en duplex par l'intermédiaire des deux sockets connectées.
Une différence essentielle est que, dans la communication via des sockets de type SOCK_DGRAM, le découpage de l'envoi en messages n'est pas préservée à la réception (une lecture peut provenir de plusieurs écritures).
Par ailleurs dans le cas du domaine AF_INET, le protocole sous-jacent (TCP) garantit que les caractères seront reçus dans le même ordre que celui de leur envoi, à l'exception de la possibilité d'adresser un caractère dit urgent ou hors bande (Out Of Band).

4.2 L'envoi de caractères

L'interface standard write

```
#include <sys/types.h>
#include <sys/socket.h>
int write(
int fd, /* descripteur */
const void *buf,/*adresse du buffer ou sont les données à envoyer*/
int count /* nombre d'octets maximum \`a envoyer */
);
```

La fonction renvoie le nombre de caractères envoyés (éventuellement 0) et -1 en cas d'erreur.

L'interface standard send

```
#include <sys/types.h>
#include <sys/socket.h>
int send(int fd, const void *buf, int count, unsigned int option );
```

Les trois premiers paramètres sont identiques à ceux de write. La primitive send permet pour les sockets de type SOCK_STREAM d'utiliser l'option MSG_OOB dans le domaine AF_INET qui indique que les données à transmettre ont un caractère urgent (0 sinon).
La fonction renvoie le nombre de caractères envoyés (éventuellement 0). Dans le cas où le tampon d'émission est plein, le processus est bloqué. Dans le cas où la connexion a été fermée

par l'autre extrémité, le processus émetteur reçoit le signal SIGPIPE ce qui entraine sa terminaison (comportement par défaut).

En cas d'erreur, la valeur de retour est -1 et errno vaut :

- EBADF : s est un descripteur invalide,
- EINTR : le processus appelant send a reçu un signal avant que les caractères puissent être "bufférisés" pour être envoyées, et le signal a interrompu l'appel-système,
- ENOTSOCK : s n'est pas une socket,
- EWOULDBLOCK : la socket est non-bloquante et l'opération demandée serait bloquante,
- ... (peut dépendre du système utilisé).

4.3 La réception de caractères

L'interface standard read

```
#include <sys/types.h>
#include <sys/socket.h>
int read(int fd, /* descripteur */
void *buf, /* adresse du buffer pour recevoir les donnees */
int count /* nombre d'octets maximum \`a recevoir */
);
```

La fonction renvoie le nombre de caractères reçus (éventuellement 0 ce qui indique que la connexion a été fermée) et -1 en cas d'erreur.

L'interface standard recv

```
#include <sys/types.h>
#include <sys/socket.h>
int recv(int fd, void *buf, int count, unsigned int flags );
```

Les trois premiers paramètres sont identiques à ceux de read.
Le dernier paramètre a soit la valeur 0, soit une combinaison bit à bit des constantes symboliques MSG_PEEK (pour consulter sans extraire) et MSG_OOB dans le domaine AF_INET (pour lire une donnée urgente).
Dans le cas où le tampon de réception est vide, le processus est bloqué.

En cas d'erreur, la valeur de retour est -1 et errno vaut :

- EBADF : s est un descripteur invalide,
- EINTR : le processus appelant send a reçu un signal avant que les caractères puissent être reçus, et le signal a interrompu l'appel-système,
- ENOTSOCK : s n'est pas une socket,
- EWOULDBLOCK : la socket est non-bloquante et l'opération demandée serait bloquante,
- ... (peut dépendre du système utilisé).

5. D'autres primitives :

Il existe d'autres primitives de la bibliothèque des sockets qui permettent une meilleure prise en charge du serveur et du client. En voici des exemples non-exhaustifs !

5.1 La primitive Setsockopt: Cela aide à manipuler les options du socket référencé par le descripteur de fichier sockfd. Ceci est complètement facultatif, mais cela aide à réutiliser l'adresse et le port.

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

5.2. La Primitive gethostbyname: pour récupérer l'identité de l'hôte où la socket est crée

```
#include <netdb.h>
struct hostent *gethostbyname(const char *name);

struct hostent {
char *h_name; /* nom officiel de la machine */
char **h_aliases; /* liste d'alias */
int h_addrtype; /* type d'adresse (AF_INET) */
int h_length; /* longueur de l'adresse */
char **h_addr_list; /* liste des adresses, ordre du réseau */
#define h_addr h_addr_list[0] /* premiere adresse */
};
```

Étant donné le nom d'un hôte, gethostbyname renvoie un pointeur vers la structure hostent contenant l'adresse IP de l'hôte et d'autres informations. Cette structure est généralement utilisée pour trouver l'adresse précédente de l'hôte via le champ h_addr. On peut aussi utiliser l'adresse de la machine pour récupérer la structure hostent :

```
struct hostent *gethostbyaddr(const char *addr, int len, int type);
```

5.3 La primitive getsockname : pour récupérer l'adresse d'une socket. Elle retourne 0 si l'opération a réussi

```
#include <sys/types.h>
#include <sys/socket.h>
int getsockname(int S, void *addr, int *addrlen);
```

L'adresse de la socket S est enregistré dans addr et sa longueur est addrlen

5.4 La primitive gethostname: Pour récupérer ou changer le nom d'une machine. Elles retournent 0 si l'opération a réussi.

```
#include <unistd.h>
int gethostname(char *name, size_t len);
int sethostname(const char *name, size_t len);
```

5.5 La primitive getpeername:

```
#include <sys/socket.h>
int getpeername(int sockfd, struct sockaddr *restrict addr,
                        socklen_t *restrict addrlen);
```

Elle retourne l'adresse du pair connecté au socket sockfd, dans le tampon pointé par addr. L'argument addrlen doit être initialisé pour indiquer la quantité d'espace pointé par l'addr. En retour, il contient la taille réelle du nom renvoyé (en octets). Le nom est tronqué si addrlen est trop petit.

5.6 La primitive shutdown

```
#include <sys/types.h>
#include <sys/socket.h>
int shutdown(int s, int how );
```

Elle permet de rendre compte, au niveau de la fermeture d'une socket, de l'aspect duplex de la communication.

Le paramètre how détermine le niveau de fermeture souhaité :

- 0 : la socket n'accepte plus d'opérations de lecture : un appel à read ou à recv renverra 0 ;

- 1 : la socket n'accepte plus d'opérations d'écriture : un appel à write ou à send provoquera la génération d'un exemplaire du signal SIGPIPE, et s'il est capté, une valeur de retour -1 (errno = EPIPE) ;
- 2 : la socket n'accepte plus ni opérations de lecture, ni opérations d'écriture.

III. La communication en mode non connecté (UDP)

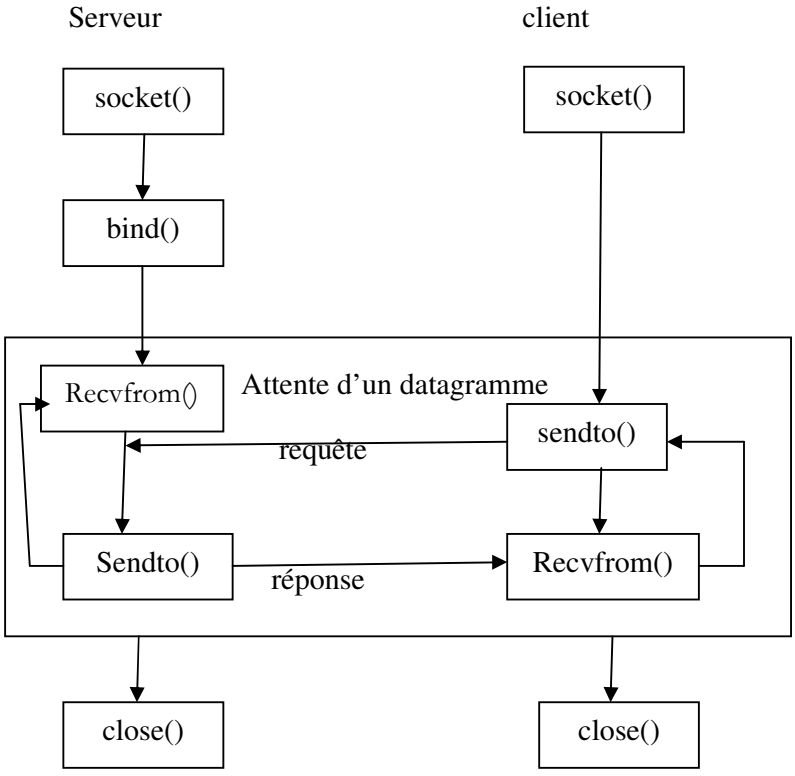
1. Introduction : Quelque soit le domaine (AF_UNIX ou AF_INET), les principales caractéristiques de la communication par datagrammes sont :

- lorsqu'un datagramme est envoyé, l'expéditeur n'obtient aucune information sur l'arrivée de son message à destination,
- les limites des messages sont préservées.

La communication avec UDP (échange de datagrammes) est réalisée par l'intermédiaire de sockets de type SOCK_DGRAM dans le domaine AF_INET.
Un processus souhaitant émettre un message à destination d'un autre doit :

- d'une part disposer d'un point de communication local (descripteur de socket sur le système local obtenu avec la primitive socket, et éventuellement nommé avec la primitive bind),
- d'autre part connaître une adresse (adresse IP et numéro de port) sur le système auquel appartient son interlocuteur (en espérant que cet interlocuteur dispose d'une socket attachée à cette adresse ... ce type de communication ne permettant pas d'en être certain).

Les sockets sont utilisés en mode non connecté : en principe, toute demande d'envoi doit comporter l'adresse de la socket destinataire (bien qu'il soit possible d'établir une pseudo-connexion entre deux sockets de type SOCK_DGRAM).



2. Primitives d'envoi et de réception

2.1 Introduction

Un processus désirant communiquer avec le monde extérieur par l'intermédiaire d'une socket de type SOCK_DGRAM doit réaliser, selon les circonstances, un certain nombre des opérations suivantes :

- demander la création d'une socket dans le domaine adapté aux applications avec lesquelles il souhaite communiquer,
- demander éventuellement l'attachement de cette socket sur un port convenu ou un port quelconque selon qu'il joue un rôle de serveur attendant des requêtes ou celui de client prenant l'initiative d'interroger un serveur,
- construire l'adresse de son interlocuteur en mémoire : tout client désirant s'adresser à un serveur doit en connaître l'adresse et donc la préparer en mémoire (éventuellement avec la primitive gethostbyname pour obtenir l'adresse IP à partir du nom) ; s'il s'agit d'un serveur, il recevra l'adresse de son émetteur avec chaque message ;
- procéder à des émissions et des réceptions de messages ; les sockets du type SOCK_DGRAM sont en mode non connecté ; chaque demande d'envoi s'accompagne de la spécification complète de l'adresse du destinataire.

2.2 La primitive sendto

```
#include <sys/types.h>
#include <sys/socket.h>
int sendto(
      int s, /* descripteur de la socket d'emission */
      const void *msg, /* adresse du message a envoyer */
      int len, /* longueur du message */
      unsigned int flags, /* option = 0 pour le type SOCK_DGRAM */
      const struct sockaddr *to, /* ptr sur l'adr du destinataire */
      int tolen /* longueur de l'adresse de la socket destinataire */
);
```

Un appel à sendto correspond à la demande d'envoi, via la socket s, du message pointé par msg de longueur len (éventuellement nulle), à destination de la socket attachée à l'adresse pointée par to de longueur tolen.

La valeur de retour est le nombre de caractères envoyé en cas de réussite, et -1 en cas d'échec. Seules les erreurs locales sont détectées, notamment il n'y a aucune détection qu'une socket est effectivement attachée à l'adresse destinataire (si c'est faux, le message sera perdu et l'émetteur n'en sera pas avisé).

Les drapeaux (flags) sont utilisés pour montrer comment la fonction doit opérer :

- MSG_NOSIGNAL : si le récepteur est déconnecté l'émetteur recevra le signal SIGPIPE, si vous utilisez ce flag le signal sera ignoré.
- MSG_DONTWAIT : pour que le récepteur n'attende pas à cause d'une congestion de trafic.
- MSG_DONTROUTE : pour que le message ne soit pas routé a travers un routeur et reste local
- MSG_OOB (out of band): il indique le caractère urgent du message si TCP est utilisé.

En cas d'erreur, errno vaut :

- EBADF : s est un descripteur invalide,

- EINTR : le processus appelant sendto a reçu un signal avant que les données aient été "bufferisés", et le signal a interrompu l'appel-système,
- EINVAL : longueur d'adresse incorrecte,
- ... (peut dépendre du système utilisé).

2.3 La primitive recvfrom

```
#include <sys/types.h>
#include <sys/socket.h>
int recvfrom(
      int s, /* descripteur de la socket de reception */
      void *buf, /* adresse de recuperation du message recu */
      int len, /* taille de la zone allouee a l'adresse buf */
      unsigned int flags, /* 0 par defaut */
      struct sockaddr *from, /* ptr pour recuperer l'adr de
      l'émetteur */
      int *fromlen /* ptr sur la longueur de l'adr de la socket de
      l'émetteur */
);
```

Cette primitive permet à un processus d'extraire un message sur une socket dont il possède un descripteur, s'il existe un message (sinon l'appel est bloquant).
Le rôle du paramètre len est de donner la longueur de la taille reservée pour mémoriser le message (des caractères seront perdus si cette taille est inférieure à la longueur du message).
Dans le cas où `from` est différent de NULL, au retour `from` pointe sur l'adresse d'expédition et *fromlen contient la longueur de cette adresse.
Les drapeaux (flags) sont utilisés pour montrer comment la fonction doit opérer :

- MSG_PEEK : pour vérifier s'il existe un message entrant
- MSG_WAITALL : pour que la primitive attende d'avoir reçu tout le message dont la taille est spécifié dans len avant de le retourner.
- MSG_OOB : pour recevoir des messages envoyés avec ce flag dans sendto ! il indique le caractère urgent du message.

En cas d'erreur, errno vaut :

- EBADF : s est un descripteur invalide,
- EINTR : le processus appelant recvfrom a reçu un signal avant que les données aient été reçus, et le signal a interrompu l'appel-système,
- EWOULDBLOCK : la socket est non-bloquante et l'opération demandée serait bloquante,
- ... (peut dépendre du système utilisé).

La valeur de retour est le nombre de caractères reçus en cas de réussite, et -1 en cas d'échec.