LES SOCKETS

Licence de ce document

Registre des éditions

Version	Date	Description des modifications	Auteur des modifications
1.0	20 février 2005	Mise en ligne du document	© Frédéric Lang (frederic-lang@fr.fm)

SOMMAIRE

I) GENERALITES

1) Présentation

Le mécanisme de communication via un protocole réseau quelconque a été implémenté dans le noyau Unix par l'université de Berkeley, à partir de la version 4.2 BSD, et a ensuite été repris par tous les autres systèmes Unix sous le nom des "sockets".
Cet ensemble de primitives est destiné aux programmeurs. Il a été le premier à gérer à la fois les communications inter-processus entre processus locaux et entre processus distants à travers un réseau. Il constitue un API (Application Programme Interface), c'est à dire une interface entre les programmes d'applications et les couches réseau. A ce titre, il sert aujourd'hui de base à la plupart des produits de communications tournant sous UNIX.
La bibliothèque des fonctions socket masque l'interface et les mécanismes de la couche transport: un appel socket se traduit par plusieurs requêtes transport.
Les sockets permettent donc d'accéder au réseau, via un modèle client-serveur, de la même manière qu'on accède à un fichier.
L'implémentation de la bibliothèque socket dans les systèmes UNIX permet aux programmeurs d'accéder aux protocoles de communication de manière particulièrement aisée (même si la mise au point de programme n'est pas toujours évidente avec ce mécanisme, il faut imaginer ce que serait sans son concours)
Le mécanisme des sockets sert de support à celui du client/serveur.

2) Description d'une socket

Une socket est un point de connexion (ou point d'extrémité) servant d'élément de référence dans les échanges entre processus locaux ou distants. Son but étant de faire communiquer des processus via la pile TCP/IP, la plus grande difficulté est la préparation et la création de celles-ci. Une fois cette phase achevée, le programmeur se retrouve en possession d'un descripteur (à la manière des fonctions comme ***open()*** pour les fichiers classiques et ***xxxget()*** pour les IPC). Ce descripteur est ensuite utilisé, comme descripteur de fichier, par les autres appels systèmes tels que ***read()*** et ***write()*** mis en œuvre dans les différentes phases de la communication.
Remarques:

✓ Le fait qu'une socket possède un descripteur au même titre qu'un fichier fait qu'on pourra rediriger les entrées/sorties standards sur une socket.

✓ Tout nouveau processus créé par un ***fork()*** hérite des descripteurs, et donc des sockets du processus père .

3) Domaines d'une socket

Un des avantages apporté par les sockets est que celles-ci peuvent être utilisées avec plusieurs protocoles de communication. Afin que les processus puissent communiquer entre-eux, il faut qu'ils utilisent les mêmes conventions d'adressage. On définit ainsi des domaines de communications qui doivent être spécifiés lors de la création de la socket. Ceux qui nous intéressent sont :

✓ AF_UNIX: domaine UNIX (pour une communication sur une même machine Unix par l'intermédiaire d'un fichier de type "socket")

✓ AF_INET: domaine INTERNET (pour une communication via TCP/IP)

✓ AF_OSI: domaine ISO

✓ AF_CCITT: domaine CCITT, X25, etc.

L'ensemble des renseignements nécessaires à l'accès à chaque domaine est regroupé dans une structure particulière spécifique au domaine, accessible la plupart du temps par un pointeur.
Le format général de cette structure est donné par la structure générique ***sockaddr***, définie dans le fichier <***sys/socket.h***>. Cependant, le terme "structure générique" signifie que ce type est seulement montrée à titre de "squelette" ou d'"exemple". Pour le programmeur, chaque domaine d'utilisation nécessite une structure d'un type précis :

✓ AF_INET: la structure à employer sera de type ***struct sockaddr_in*** et est définie dans <***netinet/in.h***>

✓ AF_UNIX: la structure à employer sera de type ***struct sockaddr_un*** et est définie dans <***sys/un.h***>

4) Types d'une socket

Le type d'une socket définit un ensemble de propriété des communications dans lesquelles elle est impliquée. Cette typologie indique donc la nature de la communication supportée par la socket.
Le type d'une socket est choisi lors de sa création avec la primitive ***socket()***.
Les principales propriétés recherchées sont :

✓ fiabilité de la transmission;

✓ préservation de l'ordre de transmission des données;

✓ non-duplication des données émises;

✓ communication en mode connecté (l'émission ne commence que quand la connexion est établie. Le chemin reste inchangé durant toute la durée de l'émission);

✓ envoi de messages urgents;

✓ préservation des limites des messages;

Les différents types sont accessibles avec les constantes suivantes (définies dans <***sys/socket.h***>:

✓ SOCK_DGRAM: Ce sont des sockets destinées à la communication en mode non-connecté pour l'envoi de datagrammes de taille bornée, non fiable (dans le domaine INTERNET, cela correspond au protocole UDP).

✓ SOCK_STREAM: Les sockets de ce type permettent des communications fiables en mode connecté orienté flots d'octets (dans le domaine INTERNET, cela correspond au protocole TCP).

✓ SOCK_RAW: Type permettant l'accès aux protocoles de plus bas niveau (dans le domaine INTERNET, cela correspond au protocole Internet).

5) Signaux associés aux sockets

Trois signaux sont rattachés aux sockets :

✓ SIGIO: Indique qu'une socket est prête pour une entrée/sortie asynchrone. Le signal est envoyé au processus (ou au groupe de processus) associé au signal.

✓ SIGURG: Indique que des données express sont arrivées sur une socket. Il est envoyé au processus (ou au groupe de processus) associé au signal.

✓ SIGPIPE: Indique qu'il n'est plus possible d'écrire sur une socket. Il est envoyé au processus associé à la socket.

II) PRINCIPES GENERAUX D'UTILISATION

1) Présentation

C'est au programmeur qu'il appartiens de décider dans quelle condition les sockets qu'il crée seront utilisées. Pour cela, il doit choisir entre:

- ✓ Le domaine d'utilisation
  - ☞ AF_UNIX: permet de faire communiquer deux processus à condition qu'ils soient sur la même machine Unix. Cette communication se fait par l'intermédiaire d'un fichier type "socket" (création d'une inode dans la machine)
  - ☞ AF_INET: permet de faire communiquer deux processus via un réseau (TCP/IP). Les deux processus peuvent être sur des machines distinctes et hétérogènes; ou sur la même machine (réseau local). Cette communication se fait par l'intermédiaire d'une adresse Internet et d'un numéro de port (numéro de référence d'un service
- ✓ Le type de la socket
  - ☞ SOCK_STREAM (mode "connecté"): un chemin virtuel unique est créé entre les deux processus. Ensuite, tout le reste n'est que lecture/écriture sur le chemin créé. Tous les octets transférés passeront par le même chemin et il est généralement impossible que l'envoi "n – 1" arrive après l'envoi "n"
  - ☞ SOCK_DGRAM (mode "datagrammes"): chaque paquet d'octets génère son propre chemin entre les deux processus. Il est alors éventuellement possible que l'envoi "n – 1" arrive après l'envoi "n" (par exemple dans le cas d'un grand réseau avec différentes routes possibles pour chaque paquet)

La communication entre deux processus, qu'ils soient sur la même machine ou sur différentes machines, se fait sur le modèle client/serveur. De ce fait, le rôle de chaque processus sera dissymétrique; le processus client aura le rôle actif de "demandeur" de service tandis que le processus serveur se contentera de rester en "écoute" et de répondre aux différents services qui lui seront demandés.

2) Utilisation en mode "connecté"

C'est le mode de communication utilisé par la plupart des applications standard utilisant le protocole "Internet" (telnet, ftp, etc.) ou les applications Unix (rlogin, rsh, rcp).
Ce mode apporte une grande fiabilité dans les échanges de données mais cela se traduit par un accroissement du volume total des données à transmettre.

a) Principe

Pour fonctionner dans ce mode, il faut au préalable réaliser une connexion entre deux points, ce qui revient à établir un "circuit virtuel". Une fois celle-ci créée, le transfert de données se fait d'une manière *continue* (notion de "flot").

b) Le client

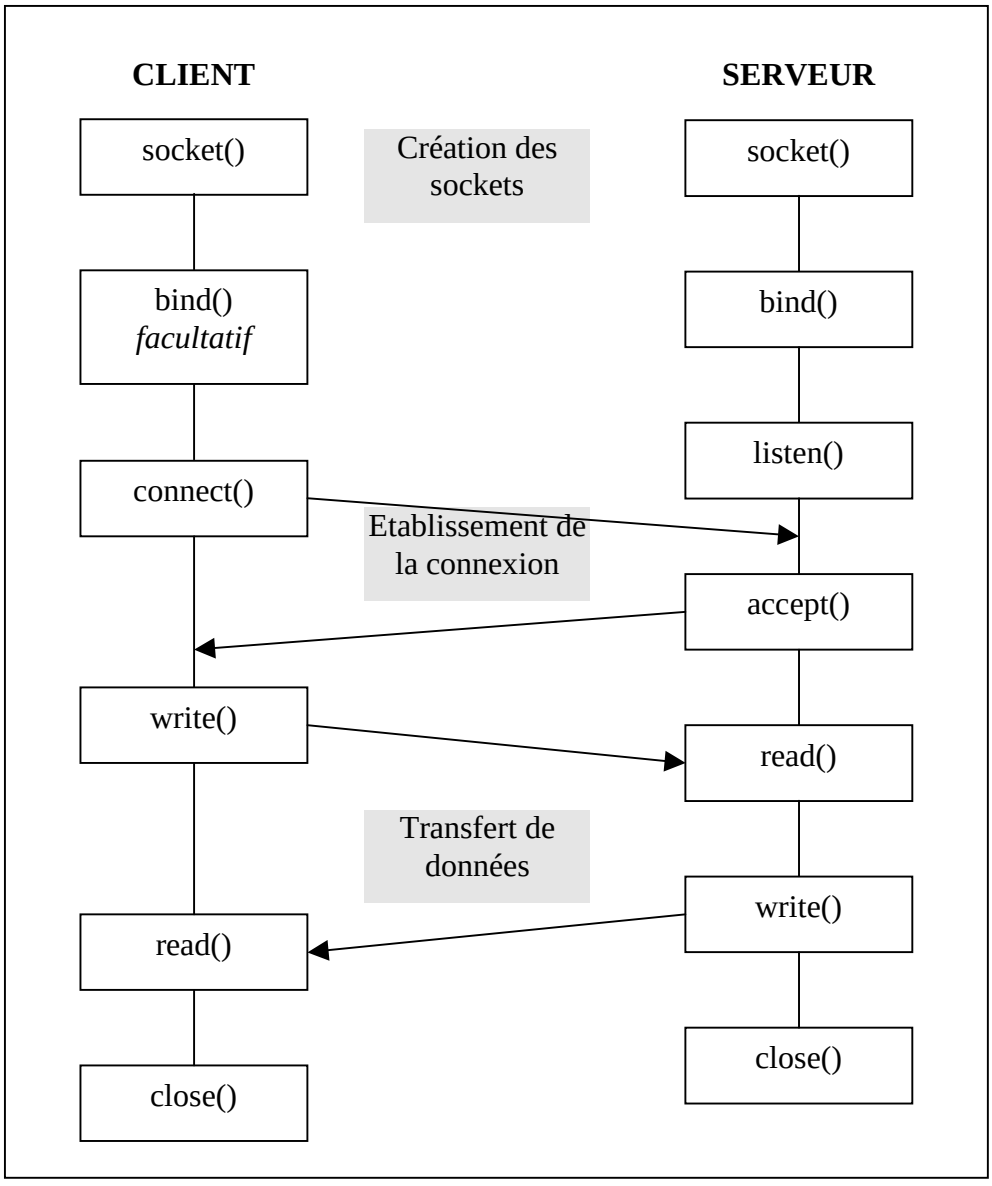
Son rôle est actif:

- ✓ création d'une socket
- ✓ connexion au serveur en fournissant un point d'accès à celui-ci. Cela peut être:
  - ☞ l'adresse Internet du serveur et le numéro de port du service (AF_INET)
  - ☞ le nom du fichier "socket" (AF_UNIX)
- ✓ lecture et/ou écriture sur la socket
- ✓ fermeture de la socket

c) Le serveur

Son rôle est passif:

- ✓ création d'une socket (socket d'écoute)
- ✓ association de cette socket au réseau et à un service. Cette association se fait sur:
  - ☞ l'adresse Internet du serveur (AF_INET) qui est aussi l'adresse locale (le serveur est lui-même) et le numéro associé au service (numéro de port)
  - ☞ le nom du fichier "socket" (AF_UNIX)
- ✓ mise en écoute des connexions entrantes
- ✓ pour chaque connexion entrante:
  - ☞ acceptation de la connexion. Une nouvelle socket est alors créée avec les mêmes caractéristiques que la socket d'origine. C'est cette socket qui est liée à celle du client et sur laquelle se feront les lectures et/ou écritures
  - ☞ création d'un processus fils pour assurer le service. Ce nouveau processus n'aura pour seule tâche que de lire et/ou écrire les informations demandées. Pendant ce temps, le processus "père" reviendra en écoute sur la socket d'origine
- ✓ fermeture de la socket pour les deux processus (père et fils)



Certains appels peuvent êtres bloquants:

- ✓ Pour le client:
  - ***connect()*** jusqu'à ce que le serveur effectue un ***accept()***
  - ***write()*** si le tampon d'émission est plein
  - ***read()*** jusqu'à ce qu'un caractère au moins soit reçu
- ✓ Pour le serveur:
  - ***accept()*** jusqu'à ce que le client effectue un ***connect()***
  - ***read()*** jusqu'à ce qu'un caractère au moins soit reçu
  - ***write()*** si le tampon d'émission est plein

3) Utilisation en mode "déconnecté"

Bien que les mécanismes utilisés pour assurer la communication entre processus en mode datagrammes soient différents de ceux vus en mode connecté; ils utilisent cependant, pour la phase de connexion, les mêmes primitives, bien que certaines aient un comportement différent.
Les opérations de lecture/écriture, quant à elles, font appel à des primitives différentes.

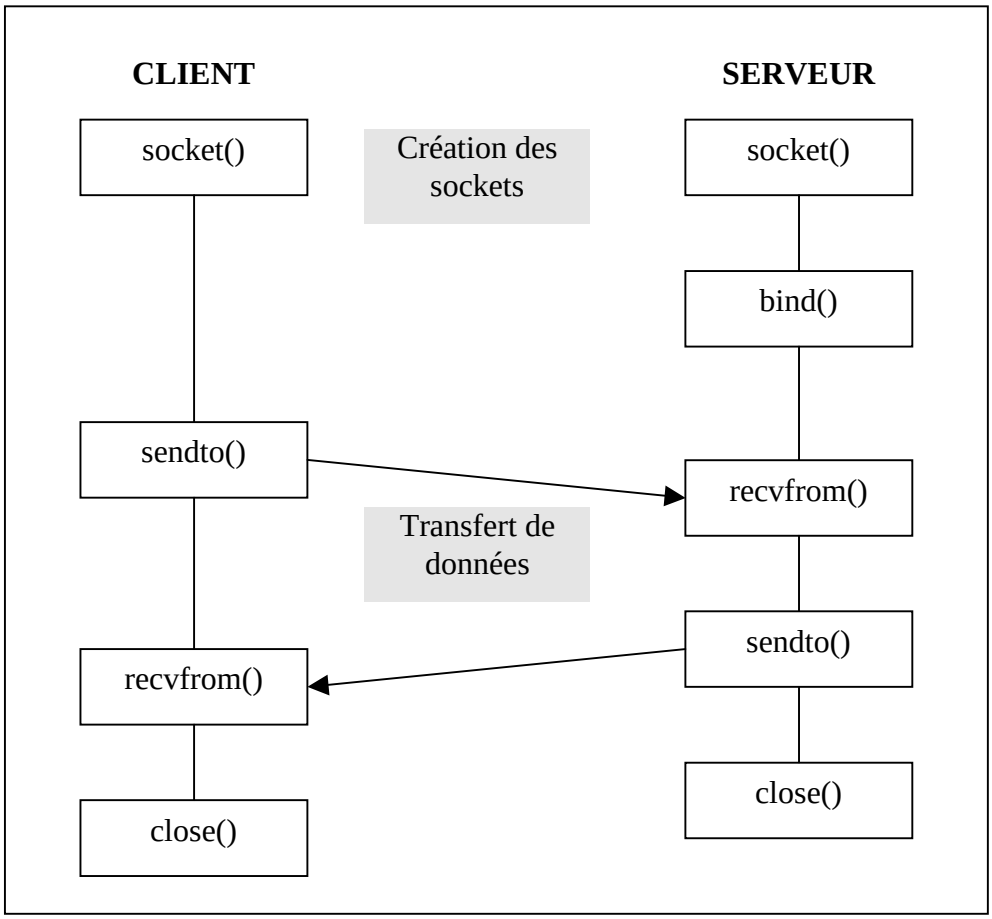
a) Principe

Un processus voulant émettre un message à destination d'un autre doit disposer d'une socket locale. Il doit en outre connaître une adresse sur le système distant auquel appartient son interlocuteur. Dans ce type de communication, rien n'indique au processus demandeur que son interlocuteur dispose d'une socket attachée à l'adresse détenue.
En mode non-connecté, toute demande d'envoi d'un message doit comporter l'adresse de la socket destinataire.

b) Analyse générale de fonctionnement

Dans ce type de fonctionnement, le client et le serveur ont des rôles symétriques. Ils réalisent chacun les opérations suivantes:

- ✓ création d'une socket;
- ✓ association de cette socket au réseau et à un service. Cette association se fait sur:
  - ☞ l'adresse Internet du serveur (AF_INET) qui est aussi l'adresse locale (le serveur est lui-même) et le numéro associé au service (numéro de port)
  - ☞ le nom du fichier "socket" (AF_UNIX)
- ✓ lecture ou écriture sur la socket (en général c'est le client qui commence);
- ✓ fermeture de la socket.



III) ACCES AU RESEAU

1) Généralités

Jusqu'à présent lorsqu'on utilisait certaines primitives (création et gestion des fichiers, création et gestion des IPC, allocation mémoire etc.) c'était le système qui, pour une bonne part, se chargeait de la gestion interne des objets créés. Il disposait pour cela des renseignements nécessaires (inaccessibles au programmeur) pour assurer cette gestion.

Dans le cas des sockets, le problème est différent dans la mesure où, si le système est encore en mesure de se charger de la gestion d'une socket locale, il n'est absolument pas en mesure de gérer la socket distante, qui fait partie d'un autre système. Pour "connaître" la socket distante on ne dispose que des renseignements contenus dans les différents fichiers de configuration réseau.

Le langage C met à la disposition du programmeur un ensemble de fonctions et de structures prédéfinies qui permettent l'accès aux renseignements contenus dans les différents fichiers de configuration réseau. Ces fonctions et structures sont donc utilisées lorsqu'il s'agit de développer des applications distribuées.

2) Structures prédéfinies

L'accès aux informations du réseau se fait par l'interrogation des fichiers systèmes par le biais de primitives C. Les informations récupérées sont stockées dans des variables d'un type prédéfini. C'est au programmeur de déclarer ces variables et leurs noms est donc libre... mais pas les membres des structures associées.

a) La structure "hostent"

La structure ***hostent*** définie dans **<*netdb.h*>** correspond à un enregistrement (ligne) du fichier "/etc/hosts".

```
#include <netdb.h>
struct hostent {
   char *h_name;                    /* nom de la machine */
   char **h_aliases;                /* liste des alias */
   int h_addrtype;                  /* type d'adresse */
   int h_length;                    /* longueur de l'adresse */
   char **h_addr_list;              /* liste d'adresses */
   #define h_addr h_addr_list[0]    /* première adresse de la liste */
};
```

Explication des éléments:

✓ char *h_name: pointeur sur le nom de base de la machine

✓ char **h_aliases: pointeur vers un tableau de pointeurs, chacun de ceux-ci pointant vers un alias éventuel de la machine

✓ int h_addrtype: type d'adresse (valeur égale à AF_INET)

✓ char **h_addrlist: pointeur vers un tableau de pointeurs, chacun de ceux-ci pointant vers une adresse Internet attribuée à la machine

✓ #define h_addr h_addr_list[0]: macro définition sur la première adresse de la liste des adresses Internet (seule adresse de la machine la plupart du temps)

b) La structure "netent"

La structure ***netent*** définie dans **<*netdb.h*>** correspond à un enregistrement (ligne) du fichier "/etc/networks".

```
#include <netdb.h>
struct netent {
   char *n_name;                /* nom du réseau */
   char **n_aliases;            /* liste des alias */
   int n_addrtype;              /* type d'adresse */
   unsigned long n_net;         /* adresse du réseau */
};
```

Explication des éléments:

✓ char *n_name: pointeur sur le nom de base du réseau

✓ char **n_aliases: pointeur vers un tableau de pointeurs, chacun de ceux-ci pointant vers un alias éventuel du réseau

✓ int n_addrtype: type d'adresse du réseau

✓ unsigned long n_net: adresse du réseau

c) La structure "servent"

Cette structure correspond à un enregistrement (ligne) du fichier "/etc/services". Elle est définie dans **<*netdb.h*>**.

```
#include <netdb.h>
struct servent {
   char *s_name;                /* nom du service */
   char **s_aliases;            /* liste des alias */
   int s_port;                  /* numéro de port */
   char *s_proto;               /* protocole utilisé */
};
```

Explication des éléments:

✓ char *s_name: pointeur sur le nom du service

✓ char **s_aliases: pointeur vers un tableau de pointeurs, chacun de ceux-ci pointant vers un alias éventuel du service

✓ int s_port: numéro de port utilisé par le service. Ce numéro est codé sous "forme réseau" ou "big–endian"

✓ char *s_proto: pointeur vers le nom du protocole utilisé pour le service ("tcp" ou "udp")

d) La structure "protoent"

La structure ***protoent*** définie dans **<*netdb.h*>** correspond à un enregistrement (ligne) du fichier "/etc/protocols".

```
#include <netdb.h>
struct protoent {
   char *p_name;                    /* nom du protocole */
   char **p_aliases;                /* liste des alias */
   int p_proto;                     /* numéro du protocole */
};
```

Explication des éléments:

✓ char *p_name: pointeur sur le nom du protocole

✓ char **p_aliases: pointeur vers un tableau de pointeurs, chacun de ceux-ci pointant vers un alias éventuel du protocole

✓ int p_proto: numéro du protocole

3) Fonctions d'accès aux informations réseau

Plusieurs fonctions n'ont pour seul but que de permettre l'accès à une ou plusieurs informations contenues dans un des fichiers de configuration réseau, via une des structures prédéfinies.

La primitive ***gethostname()*** va chercher dans le fichier "/etc/hosts" le nom de la machine locale.

```
#include <netdb.h>
int gethostname (char *nom, int lg);
```

Explication des paramètres:

✓ char *nom: pointeur vers une zone qui servira à stocker le nom récupéré

✓ int lg: longueur réservée pour la zone de stockage du nom

La primitive ***gethostbyname()*** va chercher dans le fichier "/etc/hosts" les informations sur une machine dont on connaît le nom.
La primitive ***gethostbyaddr()*** va chercher dans le fichier "/etc/hosts" les informations sur une machine dont on connaît l'adresse.

```
#include <netdb.h>
struct hostent *gethostbyname (char *nom);
struct hostent *gethostbyaddr (char *adr, int lg, int type);
```

Explication des paramètres:

- ✓ char *nom: nom de la machine sur laquelle on désire des informations
- ✓ char *adr: adresse de la machine sur laquelle on désire des informations. Cette adresse doit être en hexadécimal, dans le format spécifié par le type
- ✓ int lg: longueur de l'adresse (en octets)
- ✓ int type: type du domaine de l'adresse demandée (AF_INET en général)

Valeur renvoyée (struct hostent*): pointeur vers une variable de type ***struct hostent***. Cette variable étant utilisée pour stocker les informations récupérées.

La primitive ***getnetbyname()*** va chercher dans le fichier "/etc/networks" les informations sur une machine dont on connaît le nom.
La primitive ***getnetbyaddr()*** va chercher dans le fichier "/etc/networks" les informations sur une machine dont on connaît l'adresse.

```
#include <netdb.h>
struct netent *getnetbyname (char *nom);
struct netent *getnetbyaddr (char *adr, int type);
```

Explication des paramètres:

- ✓ char *nom: nom de la machine sur laquelle on désire des informations
- ✓ char *adr: adresse de la machine sur laquelle on désire des informations. Cette adresse doit être en hexadécimal, dans le format spécifié par le type
- ✓ int type: type du domaine de l'adresse demandée (AF_INET en général)

Valeur renvoyée (struct netent*): pointeur vers une variable de type ***struct netent***. Cette variable étant utilisée pour stocker les informations récupérées.

La primitive ***getservbyname()*** va chercher dans le fichier "/etc/services" les informations sur un service dont on connaît le nom.
La primitive ***getservbyport()*** va chercher dans le fichier "/etc/services" les informations sur un service dont on connaît le numéro de port.

```
#include <netdb.h>
struct servent *getservbyname (char *nom, char *proto);
struct servent *getservbyport (int port, char *proto);
```

Explication des paramètres:

- ✓ char *nom: nom du service sur lequel on désire des informations
- ✓ int port: numéro de port sur lequel on désire des informations. Ce numéro doit être codé sous "forme réseau" ou "big–endian"
- ✓ char *proto: nom du protocole associé au service

Valeur renvoyée (struct servent*): pointeur vers une variable de type ***struct servent***. Cette variable étant utilisée pour stocker les informations récupérées.

La primitive ***getprotobyname()*** va chercher dans le fichier "/etc/protocols" les informations sur un protocole dont on connaît le nom.
La primitive ***getprotobynumber()*** va chercher dans le fichier "/etc/protocols" les informations sur un protocole dont on connaît le numéro.

```
#include <netdb.h>
struct protoent *getprotobyname (char *nom);
struct protoent *getprotobynumber (int num);
```

Explication des paramètres:

- ✓ char *nom: nom du protocole sur lequel on désire des informations
- ✓ int num: numéro du protocole sur lequel on désire des informations

Valeur renvoyée (struct protoent*): pointeur vers une variable de type ***struct protoent***. Cette variable étant utilisée pour stocker les informations récupérées.

La primitive ***getsockname()*** permet de retrouver l'adresse d'une machine à laquelle est rattachée une socket (l'adresse n'est pas forcément connue, cet attachement ayant pu être fait par un processus ascendant).

```
#include <netdb.h>
int getgockbyname (int socket, struct sockaddr *adr, int *lg);
```

Explication des paramètres:

✓ int socket: numéro de la socket créée à l'origine

✓ struct sockaddr *adr: pointeur vers une variable de type ***struct sockaddr***. Cette variable étant utilisée pour recevoir l'adresse de la machine à laquelle est rattachée la socket récupérée par la fonction.

✓ int *lg: pointeur vers une variable de type ***int***. Avant l'appel à la fonction, cette variable doit contenir la taille de la zone réservée à ***adr*** pour récupérer l'adresse de la machine. Au retour de la fonction, cette variable contiendra la taille effective de l'adresse de la machine de rattachement.

IV) PROGRAMMATION DES SOCKETS

1) Structures associées

a) La structure "sockaddr_in"

La structure ***sockaddr_in*** définie dans ***<netinet/in.h>*** est à employer lorsque le programmeur évolue dans le domaine Internet (AF_INET). Elle permet d'indiquer quelle sera l'adresse d'une machine distante et le numéro de port du service associés à la connexion.

```
#include <sys/types.h>
#include <netinet/in.h>
struct sockaddr_in {
   u_char sin_len;                  /* taille du nom */
   u_char sin_family;               /* famille de l'adresse */
   u_short sin_port;                /* numéro de port */
   struct in_addr sin_addr;         /* adresse machine */
   char sin_zero[8];                /* champ à zéro */
};
```

Explication des éléments:

- ✓ u_char sin_len: taille de l'adresse IP
- ✓ u_char sin_family: famille de l'adresse (AF_INET)
- ✓ u_short sin_port: numéro de port associé au service. Ce numéro doit impérativement être codé sous "forme réseau" ou "big–endian"
- ✓ struct in_addr sin_addr: adresse d'une machine distante. La structure de ce champ est décrite au paragraphe suivant
- ✓ char sin_zero[8]: champ à zéro

La structure ***in_addr*** définie dans ***<netinet/in.h>*** définit le format de la variable ***sin_addr*** de la structure ***sockaddr_in***. Ce découpage permet une plus grande souplesse d'évolution possible (IP V6 par exemple).

```
#include <sys/types.h>
#include <netinet/in.h>
struct in_addr {
   u_long s_addr;                   / * adresse IP */
};
```

Explication des éléments:

- ✓ u_long s_addr: adresse IP. Cette adresse étant codée sur un long non-signé (4 octets), chaque octet devra correspondre à une des parties de l'adresse IP. Dans le cas du serveur, si le programmeur ne veut pas se fatiguer à récupérer l'adresse locale pour remplir cette variable, il pourra y mettre la constante INADDR_ANY.

b) La structure "sockaddr_un"

La structure ***sockaddr_un*** définie dans **<*sys/un.h***> est à employer lorsque le programmeur évolue dans le domaine Unix (AF_UNIX). Elle permet d'indiquer quel sera le fichier (*socket*) associé à la connexion.

```
#include <sys/types.h>
#include <sys/un.h>
struct sockaddr_un {
   u_char sun_len;                  /* taille du nom */
   u_char sun_family;               /* famille de l'adresse */
   char sun_path[108];              /* nom du fichier */
};
```

Explication des éléments:

- ✓ u_char sun_len: taille complète du nom du fichier socket (**y compris l'octet du zéro final**).
- ✓ u_char sun_family: famille de l'adresse (AF_UNIX)
- ✓ char sun_path[108]: nom Unix du fichier socket

2) Fonctions de création et d'association au réseau

La primitive ***socket()*** crée une socket. Celle-ci n'est rattachée à rien d'autre que la machine qui la crée. L'établissement de la connexion avec le réseau se fait plus tard.

```
#include <sys/socket.h>
int socket (int dom, int type, int proto);
```

Explication des paramètres:

- ✓ int dom: domaine de la socket (AF_UNIX, AF_INET, etc.)
- ✓ int type: type de la socket (SOCK_STREAM ou SOCK_DGRAM)
- ✓ int proto: protocole de communication. Mettre "**0**" dans ce paramètre indique au système de se baser sur le type défini dans le paramètre précédent (***type***) pour définir automatiquement son protocole de communication.

Valeur renvoyée (int): Numéro de la socket créé. Ce numéro aura deux utilités selon le programme qui invoque la primitive:

- ✓ pour le serveur: ce numéro servira pour relier la socket au réseau (***bind()***), l'écoute du réseau (***listen()***) et l'acceptation des connexions entrantes (***accept()***).
- ✓ pour le client: ce numéro servira de point d'entrées-sorties (***read()***/***write()***) ou (***sendto()***/***recvfrom()***) dans le dialogue avec le serveur.

La primitive ***bind()*** rattache la socket créée au réseau. Ce rattachement est obligatoire pour le serveur, mais facultatif pour le client. Si ce dernier n'invoque pas cette primitive, le rattachement au réseau se fera automatiquement au moment de la connexion au serveur.

```
#include <sys/types.h>
#include <sys/socket.h>
int bind (int socket, struct sockaddr *adr, int sz_adr);
```

Explication des paramètres:

✓ int socket: numéro de la socket créée, renvoyé par la primitive ***socket()***.

✓ struct sockaddr *adr: pointeur vers une structure générique de type ***struct sockaddr*** contenant l'adresse du serveur (adresse locale quand c'est le serveur qui invoque cette primitive). Le terme "structure générique" signifie que ce type est seulement montrée à titre de "squelette" ou d'"exemple". Lorsque le programmeur invoquera cette primitive, il devra lui passer un pointeur sur une variable contenant les informations nécessaires à ***bind()***, et dont le type dépend du domaine dans lequel il communique:

- AF_INET: la variable devra être de type ***struct sockaddr_in***
- AF_UNIX: la variable devra être de type ***struct sockaddr_un***

Cette variable aura préalablement été remplie par le programmeur. Son type et les informations à y mettre dépendent du domaine dans lequel il crée son réseau:

- AF_INET: il ne remplira que les membres ***sin_family***, ***sin_port*** et ***sin_addr.s_addr***
- AF_UNIX: il ne remplira que les membres ***sun_family***, et ***sun_path***

✓ int sz_adr: taille de la zone réservée pour la variable pointée par ***adr***. En effet, le type de cette variable étant différent selon le domaine utilisé, la fonction ne connaît pas sa taille car celle-ci n'a pas pu être prévue à l'avance.

La primitive ***socketpair()*** permet de créer et d'associer en une seule opération deux sockets dans un mécanisme semblable à celui de la primitive ***pipe()***. Cette primitive remplace avantageusement les primitives ***socket()*** et ***bind()*** et les deux sockets créées permettent la communication dans les deux sens; mais doivent appartenir à un même programme (utilisée en général par un processus dupliqué père/fils) donc cette primitive n'est utilisable que dans le domaine Unix (AF_UNIX) et en mode "connecté" (SOCK_STREAM).

```
#include <sys/socket.h>
int socketpair (int dom, int type, int proto, int cote[2]);
```

Explication des paramètres:

✓ int dom: domaine de la socket (AF_UNIX, AF_INET, etc.)

✓ int type: type de la socket (SOCK_STREAM ou SOCK_DGRAM)

✓ int proto: protocole de communication (0 dans ce paramètre indique au système de se baser sur le type défini dans le paramètre précédent (***type***) pour définir automatiquement son protocole de communication.

✓ int cote[2]: pointeur vers un tableau de deux entiers qui seront remplis par la fonction (comme pour la primitive ***pipe()***). A la fin de l'exécution de la primitive, on aura cote[0] et cote[1] contenant chacun un descripteur permettant l'accès à la socket

3) Etablissement du circuit connecté (SOCK_STREAM)

a) Fonctions coté "serveur"

La primitive ***listen()*** prépare une socket à l'écoute des connexions entrantes.

```
#include <sys/socket.h>
int listen (int socket, int nb_conn);
```

Explication des paramètres:

✓ int socket: numéro de la socket créée, renvoyé par la primitive ***socket()***.

✓ int nb_conn: nombre maximal de demande de connexions. Ce paramètre permet de définir la taille d'une file d'attente dans laquelle seront mémorisées les demandes de connexions venant des clients mais non encore notifiées au serveur.

La primitive ***accept()*** extrait une demande de connexion de la file d'attente. Lorsque la connexion est acceptée, le serveur crée une seconde socket sur laquelle se feront les échanges de données. Lorsque ceux-ci sont terminés, le système détruit cette seconde socket et le serveur peut reprendre son écoute sur la première socket (ou extraire une autre demande de la file d'attente).

```
#include <sys/types.h>
#include <sys/socket.h>
int accept (int socket, struct sockaddr *adr, int *sz_adr);
```

Explication des paramètres:

✓ int socket: numéro de la socket créée, renvoyé par la primitive ***socket()***.

✓ struct sockaddr *adr: pointeur vers une structure générique de type ***struct sockaddr*** destinée à recevoir l'adresse du client qui se connecte. La valeur de ce paramètre peut être à NULL si le serveur ne désire pas connaître l'adresse de son client. Dans le cas contraire, comme pour ***bind()***, lorsque le programmeur invoquera cette primitive, il devra lui passer un pointeur sur une variable dont le type dépend du domaine dans lequel il communique:

- ☞ AF_INET: la variable devra être de type ***struct sockaddr_in***
- ☞ AF_UNIX: la variable devra être de type ***struct sockaddr_un***

✓ int *sz_adr: pointeur sur une variable prévue pour recevoir la taille de la structure associée à la socket du client. Comme pour ***adr***, ce paramètre peut être à NULL. Cette variable doit être initialisée avant l'appel de la primitive afin de pouvoir récupérer les informations du client dans le pointeur "adr".

Valeur renvoyée (int): Identificateur de la socket servant aux échanges de données (socket de dialogue). C'est sur ce descripteur que le serveur viendra lire et/ou écrire les données du client.

b) Fonctions coté "client"

La primitive ***connect()*** en mode "connecté" établi la connexion avec une adresse de destination. Si l'association avec cette adresse n'a pas été faite par l'intermédiaire de ***bind()***, celle-ci se fait lors de la connexion.
Cette fonction à un comportement différent en mode "non-connecté".

```
#include <sys/types.h>
#include <sys/socket.h>
int connect (int socket, struct sockaddr *adr, int sz_adr);
```

Explication des paramètres:

✓ int socket: numéro de la socket créée, renvoyé par la primitive ***socket()***.

✓ struct sockaddr *adr: pointeur vers une structure générique de type ***struct sockaddr*** contenant l'adresse du serveur. Comme pour ***bind()***, lorsque le programmeur invoquera cette primitive, il devra lui passer un pointeur sur une variable dont le type dépend du domaine dans lequel il communique:

- AF_INET: la variable devra être de type ***struct sockaddr_in***
- AF_UNIX: la variable devra être de type ***struct sockaddr_un***

Cette variable aura préalablement été remplie par le programmeur. Son type et les informations à y mettre dépendent du domaine dans lequel il crée son réseau:

- AF_INET: il ne remplira que les membres ***sin_family***, ***sin_port*** et ***sin_addr.s_addr***
- AF_UNIX: il ne remplira que les membres ***sun_family*** et ***sun_path***

Cette variable devant contenir l'adresse du serveur, le programmeur devra utiliser les fonctions permettant d'accéder aux renseignements contenus dans les fichiers de configuration réseau pour récupérer les informations qui lui manquent et remplir correctement cette variable.

✓ int sz_adr: taille de la zone réservée pour la variable pointée par ***adr***. En effet, le type de cette variable étant différent selon le domaine utilisé, la fonction ne connaît pas sa taille car celle-ci n'a pas pu être prévue à l'avance.
L'utilisation de cette primitive est semblable à celui de la fonction ***bind()***, à ceci près que l'adresse à passer à ***connect()*** n'est pas l'adresse locale mais l'adresse du serveur. Il faut donc utiliser les fonctions permettant d'accéder aux renseignements contenus dans les fichiers de configuration réseau pour pouvoir remplir la variable ***adr***.

c) Dialogue client/serveur

Une fois la connexion établie entre le client et le serveur, les deux processus peuvent s'échanger des flots d'informations. Comme le système est orienté "flot d'octet", le format des messages n'est pas préservé. Il peut y avoir plusieurs opérations d'écriture dans le tampon avant une opération de lecture.

Les primitives ***write()***, ***send()***, ***read()*** et ***recv()*** ont pour but d'aller respectivement écrite (***write()*** et ***send()***) et lire (***read()*** et ***recv()***) des octets dans une socket. Les primitives ***read()*** et ***write()*** sont ici les deux primitives déjà vues dans les mécanismes de gestion des fichiers, mais le descripteur utilisé est ici un descripteur de socket au lieu d'un descripteur de fichiers.

```
#include <fcntl.h>
int write (int socket, char *buffer, int nb);
int read (int socket, char *buffer, int nb);
int send (int socket, char *buffer, int nb, int option);
int recv (int socket, char *buffer, int nb, int option);
```

Explication des paramètres:

✓ int socket: numéro de la socket créée ; renvoyé par la primitive ***socket()***.du coté du client et par la primitive ***accept()*** du coté du serveur.

✓ char *buffer: pointeur vers une zone mémoire dans laquelle seront rangés (ou pris) les caractères lus (ou écrits) de la socket

✓ int nb: nombre de caractères à lire (ou écrire) dans la socket

✓ int option: permet de définir certaines caractéristiques. En général, elle est positionnée à 0 mais peut être positionnée aussi à MSG_OOB (voir le chapitre sur les messages urgents).

Valeur renvoyée (int):

✓ Le nombre d'octets réellement lus (ou écrits) dans la socket s'il y en a. Ce nombre peut être inférieur à ***nb*** (par exemple si on demande de lire plus que ce qu'il n'y a), mais jamais supérieur.

✓ 0 s'il n'y a plus rien à lire

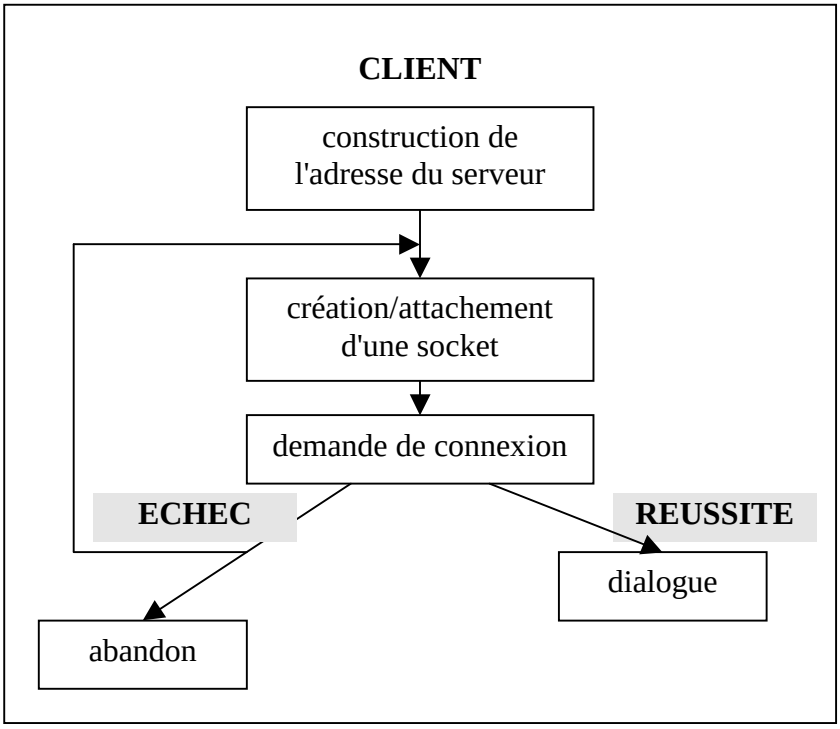
Les primitives d'écriture sont bloquantes quand:

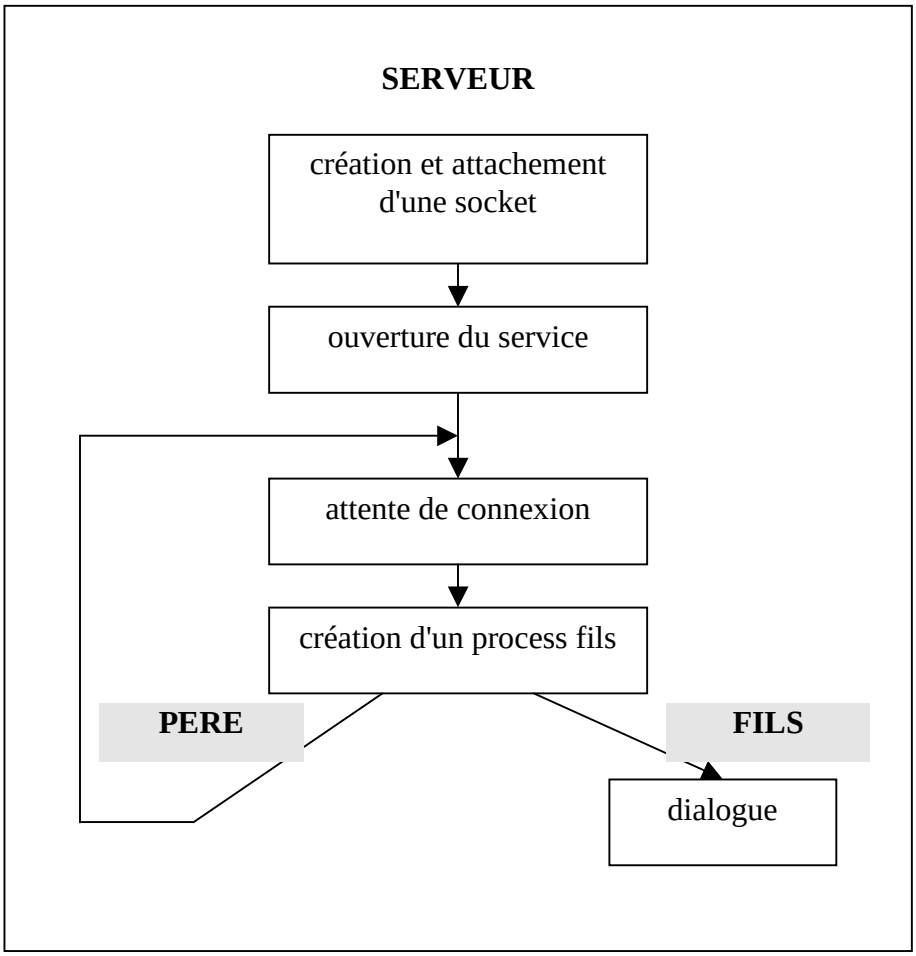
✓ le tampon TCP de réception de la socket destinataire est plein

✓ le tampon RCP d'émission de la socket locale est plein

Les primitives de lecture sont bloquantes quand le tampon de la socket réceptrice est vide.

d) Principe d'un dialogue client/serveur





4) Dialogue en mode non-connecté (SOCK_DGRAM)

C'est dans ce domaine que les différences sont les plus notables. Comme l'envoi du message se fait par "paquets" et qu'il n'y a plus la notion de "circuit virtuel", toutes les opérations de lecture/écriture doivent comporter l'adresse de destination.

Les primitives ***sendto()*** et ***recvfrom()*** ont pour but d'aller respectivement écrite et lire des octets dans une socket. Leur comportement est identique à celui de ***send()*** et ***recv()***, à ceci près que leur utilisation n'est pas bloquante du fait de leur protocole de communication en mode ***udp***.

```
#include <sys/socket.h>
int sendto (int socket, char *buffer, int nb, int option,
            struct sockaddr *adr, int sz_adr);
int recvfrom (int socket, char *buffer, int nb, int option,
            struct sockaddr *adr, int *sz_adr);
```

Explication des paramètres:

✓ int socket: numéro de la socket créée, renvoyé par la primitive ***socket()***.

✓ char *buffer: pointeur vers une zone mémoire dans laquelle seront rangés (ou pris) les caractères lus (ou écrits) de la socket

✓ int nb: nombre de caractères à lire (ou écrire) dans la socket

✓ int option: permet de définir certaines caractéristiques. En général, elle est positionnée à 0 mais peut être positionnée aussi à MSG_OOB (voir le chapitre sur les messages urgents).

✓ struct sockaddr *adr: pointeur vers une structure générique de type ***struct sockaddr*** contenant l'adresse du serveur. Comme pour ***bind()***, lorsque le programmeur invoquera cette primitive, il devra lui passer un pointeur sur une variable dont le type dépend du domaine dans lequel il communique:

- AF_INET: la variable devra être de type ***struct sockaddr_in***
- AF_UNIX: la variable devra être de type ***struct sockaddr_un***

Cette variable aura préalablement été remplie par le programmeur. Son type et les informations à y mettre dépendent du domaine dans lequel il crée son réseau:

- AF_INET: il ne remplira que les membres ***sin_family***, ***sin_port*** et ***sin_addr.s_addr***
- AF_UNIX: il ne remplira que les membres ***sun_family*** et ***sun_path***

Cette variable devant contenir l'adresse du serveur, le programmeur devra utiliser les fonctions permettant d'accéder aux renseignements contenus dans les fichiers de configuration réseau pour récupérer les informations qui lui manquent et remplir correctement cette variable.

✓ int sz_adr: taille de la zone réservée pour la variable pointée par ***adr***. En effet, le type de cette variable étant différent selon le domaine utilisé, la fonction ne connaît pas sa taille car celle-ci n'a pas pu être prévue à l'avance.

✓ int *sz_adr: pointeur sur une variable prévue pour recevoir la taille de la structure associée à la socket du client.

Valeur renvoyée (int):

✓ Le nombre d'octets réellement lus (ou écrits) dans la socket s'il y en a. Ce nombre peut être inférieur à ***nb*** (par exemple si on demande de lire plus que ce qu'il n'y a), mais jamais supérieur.

✓ 0 s'il n'y a plus rien à lire

La primitive ***connect()*** en mode "non-connecté" a pour but de mémoriser dans la socket locale l'adresse d'une socket distante, ceci dans le cas où la communication se fait toujours avec la même socket distante. Une fois cette adresse mémorisée, tout message envoyé (ou lu) ira à (ou viendra de) cette adresse et il n'est plus besoin d'indiquer celle-ci dans les primitives ***sendto()*** ou ***recvfrom()***. Il est même possible d'utiliser à la place les primitives ***send()*** ou ***recv()***.
Cette fonction à un comportement différent en mode "connecté".

```
#include <sys/types.h>
#include <sys/socket.h>
int connect (int socket, struct sockaddr *adr, int sz_adr);
```

Explication des paramètres:

✓ int socket: numéro de la socket créée, renvoyé par la primitive ***socket()***.

✓ struct sockaddr *adr: pointeur vers une structure générique de type ***struct sockaddr*** contenant l'adresse de la socket distante. Lorsque le programmeur invoquera cette primitive, il devra lui passer un pointeur sur une variable dont le type dépend du domaine dans lequel il communique:

- AF_INET: la variable devra être de type ***struct sockaddr_in***
- AF_UNIX: la variable devra être de type ***struct sockaddr_un***

Cette variable aura préalablement été remplie par le programmeur. Son type et les informations à y mettre dépendent du domaine dans lequel il crée son réseau:

- AF_INET: il ne remplira que les membres ***sin_family***, ***sin_port*** et ***sin_addr.s_addr***
- AF_UNIX: il ne remplira que les membres ***sun_family*** et ***sun_path***

Cette variable devant contenir l'adresse du serveur, le programmeur devra utiliser les fonctions permettant d'accéder aux renseignements contenus dans les fichiers de configuration réseau pour récupérer les informations qui lui manquent et remplir correctement cette variable.

✓ int sz_adr: taille de la zone réservée pour la variable pointée par ***adr***. En effet, le type de cette variable étant différent selon le domaine utilisé, la fonction ne connaît pas sa taille car celle-ci n'a pas pu être prévue à l'avance.

5) Dialogue évolué

Les primitives développées dans cette partie sont accessible aux modes "connecté" et "non-connecté".

a) Les structures associées

Les structures ***iovec*** et ***msghdr*** définie dans <***socket.h***> définissent deux ensembles permettant d'envoyer ou de recevoir une succession de messages avec un seul appel système

```
#include <sys/types.h>
#include <sys/socket.h>
struct iovec {
   caddr_t iov_base;                /* adresse du message */
   int iov_len;                     /* longueur du message */
};
```

Explication des éléments:

✓ caddr_t iov_base: pointeur vers une zone contenant le message à envoyer ou prévue pour recevoir un message

✓ int iov_len: longueur de la zone contenant le message

```
#include <sys/types.h>
#include <sys/socket.h>
struct msghdr {
   caddr_t msg_name;                /* adresse optionnelle */
   int msg_namelen;                 /* taille de l'adresse */
   struct iovec *msg_iov;           /* tableau des messages */
   int msg_iovlen;                  /* nombre de messages */
   caddr_t msg_accrights;           /* inutilisé */
   int msg_accrightslen;            /* inutilisé */
};
```

Explication des éléments:

✓ caddr_t msg_name: pointeur vers une zone éventuelle permettant de titrer l'envoi

✓ int msgnamelen: longueur de la zone contenant le message de titre

✓ struct iovec *msg_iov: pointeur vers une zone contenant les messages à envoyer

✓ int msg_iovlen: nombre de messages de la zone

✓ caddr_t msg_accrights: mis en place dans un but de développement futur

✓ int msg_accrightslen: mis en place dans un but de développement futur

b) Les primitives

Les primitives ***sendmsg()*** et ***recvmsg()*** permettent d'envoyer (ou de recevoir) une succession de messages avec un seul appel système.

```
#include <sys/socket.h>
int sendmsg (int socket, struct msghdr *msg, int option);
int recvmsg (int socket, struct msghdr *msg, int option);
```

Explication des paramètres:

✓ int socket: numéro de la socket créée, renvoyé par la primitive ***socket()***.

✓ struct msghdr *msg: pointeur vers une variable de type ***struct msghdr*** contenant les messages à envoyer ou recevoir

✓ int option: permet de définir certaines caractéristiques. En général, elle est positionnée à 0 mais peut être positionnée aussi à MSG_OOB (voir le chapitre sur les messages urgents).

Valeur renvoyée (int):

✓ Le nombre d'octets réellement lus (ou écrits) dans la socket s'il y en a

✓ 0 s'il n'y a plus rien à lire

6) Paramétrage des sockets

Les primitives ***getsockopt()*** et ***sendsockopt()*** permettent respectivement d'obtenir des informations sur une socket ou d'en modifier le comportement.

```
#include <sys/socket.h>
int getsockopt (int socket, int niveau, int option, char *adr, int *sz_adr);
int setsockopt (int socket, int niveau, int option, char *adr, int sz_adr);
```

Explication des paramètres:

✓ int socket: numéro de la socket créée, renvoyé par la primitive ***socket()***.

✓ int niveau: niveau de l'interface avec les sockets. En général, on y met SOL_SOCKET (niveau maximum)

✓ int option: option à appliquer:
- SO_TYPE: extraction du type de la socket (***getsockopt()*** uniquement)
- SO_BROADCAST: possibilité d'envoyer à tous un message
- SO_REUSEADR: possibilité de réutiliser l'adresse donnée par ***bind()***
- SO_SNDBUF: taille du buffer d'émission
- SO_RCVBUF: taille du buffer de réception

✓ char *adr: pointeur vers une zone permettant de recevoir le résultat (***getsockopt()***) ou contenant la valeur à transmettre (***setsockopt()***).

✓ int *sz_adr: pointeur sur une variable contenant la taille de la valeur à transmettre et contenant la taille du résultat (taille de ***adr***) au retour de la fonction

✓ int sz_adr: taille de ***adr***

7) Le serveur universel

a) Principe

Si on réfléchit bien au code débutant un serveur, on s'aperçoit qu'il est toujours identique quel que soit le serveur que l'on veut créer. Il récupère un numéro de port, attend une connection, lance un fils pour dialoguer avec le client et retourne écouter le réseau en laisant au fils tout le soin du dialogue.
Il est évident qu'en cas de serveurs multiples ; toute ces phases d'initialisation sont à refaire pour chaque serveur et cela entraîne redondance de code et lourdeur dans les évolutions.
C'est à cet effet qu'a été créé le super-daemon "***/etc/inetd***" paramétrable par le fichier "***/etc/inetd.conf***". Ce super daemon a pour tâche première de récupérer dans "*/etc/inetd.conf*" quels clients il doit attendre. Grâce au fichier "***/etc/services***" ; il utilise chaque nom de client attendu pour récupérer le numéro de port associé et se met à écouter le réseau sur ce port là. Dès qu'un client attendu arrive sur ce numéro de port, le super daemon crée un fils et ce dernier lance un mini-serveur dont la tâche est juste de dialoguer avec le client.

b) Programmation

Pour le programmeur, toute la phase d'initialisation des sockets, acceptation du client, lancement du fils est déjà faite. Il ne lui reste plus qu'à programmer le mini-serveur avec des lectures/écritures sur la socket selon l'algorithme qui lui convient.
Le dernier problème qu'il peut craindre est que ce mini-serveur (programmé de façon individuelle) n'a plus de lien avec le super-serveur "***inetd***" qui l'a généré et n'a, de ce fait, aucun accès aux variables initialisées par ce dernier… ces variables contenant, entre autres, la socket de communication.
La solution est donnée par "***inetd***" lui-même qui va dupliquer sa socket créée dans les flux standard d'entrée/sortie numérotés respectivement "***0***" et "***1***" (*cf. cours Unix sur les processus*). Cette opération se fait par la primitive "***dup()***"(*cf. cours c-/système sur les fichiers*) lancée par le super-serveur.
Pour le programmeur du mini-serveur, il ne lui reste plus qu'à aller lire et écrire sur les flux "***0***" et "***1***" pour pouvoir dialoguer, via la socket, avec son client.
Du coté du client, rien ne change. Il tente toujours de se connecter à la machine serveur avant d'aller lire et écrire sur sa socket créée.

V) POSSIBILITES AVANCEES

1) Utilisation du signal SIGIO

Le signal SIGIO permet d'aviser un ou plusieurs processus que des informations sont arrivées sur une socket donnée.
A chaque socket il faut associer un pseudo-driver susceptible d'émettre ce signal à un groupe de processus donné. Pour cela il faut :

- ✓ définir une fonction détournant le signal SIGIO
- ✓ demander l'envoi du signal SIGIO à chaque arrivée de caractères sur la socket (primitive ***fcntl()*** avec la commande FASYNC)
- ✓ attribuer un groupe propriétaire à la socket (primitive ***fcntl()*** avec la commande F_SETOWN);

2) Envoi de messages urgents

a) Principe

Les données qui circulent via des sockets constituent un flot d'octets : un caractère ne peut être lu que si ceux qui le précèdent sont lus. Il est intéressant de pouvoir, dans certains cas, prendre en compte certaines informations immédiatement. Ce que ne peut faire le fonctionnement de base des sockets. Pour cela il faudrait un mécanisme qui permettrait de transporter des caractères dont l'arrivée serait notifiée immédiatement au processus destinataire.
Le protocole TCP prévoit la possibilité de transmettre au moins un caractère selon ce principe. Sa réalisation sous UNIX correspond à la notion out of band (OOB). Il faut aussi envisager la réalisation de l'asynchronisme via le signal SIGURG.

b) Emission

Elle est réalisée sur une socket de type SOCK_STREAM du domaine AF_INET, connectée au moyen de la primitive ***send()*** avec l'option MSG_OOB.
Sous UNIX seul un caractère urgent est pris en compte (si on en envoie plusieurs seul le dernier est conservé).

c) Réception

Le caractère OOB reçu sur une socket est mémorisé en dehors des tampons habituels. Il est cependant possible de forcer le système à insérer ce caractère dans le flot normal. Le système peut alors repérer, grâce à une marque, la position du caractère urgent dans le flot.
La lecture d'un caractère OOB ne peut être réalisée que par les primitives ***recv()*** et ***recvfrom()*** en utilisant l'option MSG_OOB.

- ✓ Il n'est pas nécessaire d'être positionné sur la marque pour pouvoir le lire.
- ✓ La lecture des caractères normaux butte sur une marque et les caractères qui suivent sont perdus.
- ✓ On peut savoir si la position courante dans un flot correspond à un caractère marqué grâce à la primitive ***ioctl()*** avec l'option SIOCATMARK .

d) Prise en compte de l'asynchronisme

Pour pouvoir vider le tampon et lire le caractère urgent qui y est placé il faut pouvoir tester l'arrivée du signal SIGURG. Il faut donc :

- ✓ définir une fonction détournant le signal SIGURG. Cette fonction sera chargée de vider les caractères qui précèdent le caractère marqué
- ✓ définir l'ensemble des processus visés

VI) PRIMITIVES ANNEXES

1) Homogénéisation des formats de données

Le codage d'un nombre sur plus d'un octet diffère selon l'architecture d'une machine ou d'une autre (octet de poid fort ou de poid faible en premier ou en dernier). Les sockets devant pouvoir transmettre des informations en milieu hétérogène impliquent une représentation de ces nombres dans un format compréhensible quel que soit l'architecture de la machine. Ce format est appelé "forme réseau" ou "big-endian".

a) Ce qui existe

Les primitives ***htonl()***, ***htons()***, ***ntohl*** et ***ntohs()*** permettent de transformer respectivement un entier long ou court en entier "représentation réseau", et un entier "représentation réseau" en entier long ou court.

```
#include <sys/types.h>
#include <netinet/in.h>
u_long htonl (u_long hostlong);
u_short htons (u_short hostshort);
u_long ntohl (u_long netlong);
u_short ntohs (u_short netshort);
```

Explication des paramètres :

- ✓ u_long hostlong : entier long sur la machine
- ✓ u_short hostshort : entier court sur la machine
- ✓ u_long netlong : entier long en "représentation réseau"
- ✓ u_short netshort : entier court en "représentation réseau"

Valeurs renvoyées :

- ✓ u_long htonl() : entier long en "représentation réseau"
- ✓ u_short htons() : entier court en "représentation réseau"
- ✓ u_long ntohl() : entier long sur la machine
- ✓ u_short ntohs() : entier court sur la machine

Contrairement à ce que l'on pourrait croire, ces primitives sont en fait des macro-définitions. Il est donc recommandé de ne pas y passer de variable auto-incrémentée (style *i*++) pour éviter les effets de bord.

b) Ce qui n'existe pas encore

Le problème se pose aussi sur les nombres à virgule flottante (***float*** ou ***double***). Malheureusement, les primitives ***htond()***, ***htonf()***, ***ntohd*** et ***ntohf()*** qui pourraient respectivement transformer un flottant long ou court en flottant "réseau" ; et un flottant "réseau" en flottant long ou court ne sont pas implémentées.
Cependant il est facile, à partir des primitives ***htons()*** et ***ntohs()***, de créer ces fonctions manquantes.

Conversion de double "local" en double "réseau" (*htond*)

```
#include <sys/types.h>                          /* Types prédéfinis 'c' */
#include <netinet/in.h>                         /* Socket internet */

/* Fonction de conversion de double "local" en double "réseau" */
double htond(
        double hostdbl)                         /* Nombre à convertir */
{
        /* Déclaration des variables */
        ushort i;                               /* Indice de boucle */
        ushort j;                               /* Indice de boucle */
        ushort convert;                         /* Zone de conversion */

        union {
                double val;                     /* Stockage du nombre */
                u_char p[8];                    /* Décomposition du nombre */
        }zone;                                  /* Zone de travail */

        /* Remplisssage nombre à convertir */
        zone.val=hostdbl;

        /* Travail sur chaque extrémités de la zone en revenant vers le centre */
        for (i=0, j=7; i < j; i++, j--)
        {
                /* Copie des parties de la zone dans un entier court de conversion */
                convert=(zone.p[i] << 8) + zone.p[j];

                /* Conversion de l'entier court par la primitive normalisée "htons" */
                convert=htons(convert);

                /* Recopie des octets de l'entier court dans les parties de la zone */
                zone.p[i]=convert >> 8;
                zone.p[j]=convert & 0x00ff;
        }

        /* Renvoi nombre converti */
        return(zone.val);
}
```

Conversion de double "réseau" en double "local" (*ntohd*)

```
#include <sys/types.h>                                  /* Types prédéfinis 'c' */
#include <netinet/in.h>                                 /* Socket internet */

/* Fonction de conversion de double "réseau" en double "local" */
double ntohd(
        double netdbl)                                  /* Nombre à convertir */
{
        /* Déclaration des variables */
        ushort i;                                       /* Indice de boucle */
        ushort j;                                       /* Indice de boucle */
        ushort convert;                                 /* Zone de conversion */

        union {
                double val;                             /* Stockage du nombre */
                u_char p[8];                            /* Décomposition du nombre */
        }zone;                                          /* Zone de travail */

        /* Remplisssage nombre à convertir */
        zone.val=netdbl;

        /* Travail sur chaque extrémités de la zone en revenant vers le centre */
        for (i=0, j=7; i < j; i++, j--)
        {
                /* Copie des parties de la zone dans un entier court de conversion */
                convert=(zone.p[i] << 8) + zone.p[j];

                /* Conversion de l'entier court par la primitive normalisée "ntohs" */
                convert=ntohs(convert);

                /* Recopie des octets de l'entier court dans les parties de la zone */
                zone.p[i]=convert >> 8;
                zone.p[j]=convert & 0x00ff;
        }

        /* Renvoi nombre converti */
        return(zone.val);
}
```

Conversion de float "local" en float "réseau" (*htonf*)

```
#include <sys/types.h>                                  /* Types prédéfinis 'c' */
#include <netinet/in.h>                                 /* Socket internet */

/* Fonction de conversion de float "local" en float "réseau" */
float htonf(
        float hostfloat)                                /* Nombre à convertir */
{
        /* Déclaration des variables */
        ushort i;                                       /* Indice de boucle */
        ushort j;                                       /* Indice de boucle */
        ushort convert;                                 /* Zone de conversion */

        union {
                float val;                              /* Stockage du nombre */
                u_char p[4];                            /* Décomposition du nombre */
        }zone;                                          /* Zone de travail */

        /* Remplisssage nombre à convertir */
        zone.val=hostfloat;

        /* Travail sur chaque extrémités de la zone en revenant vers le centre */
        for (i=0, j=3; i < j; i++, j--)
        {
                /* Copie des parties de la zone dans un entier court de conversion */
                convert=(zone.p[i] << 8) + zone.p[j];

                /* Conversion de l'entier court par la primitive normalisée "htons" */
                convert=htons(convert);

                /* Recopie des octets de l'entier court dans les parties de la zone */
                zone.p[i]=convert >> 8;
                zone.p[j]=convert & 0x00ff;
        }

        /* Renvoi nombre converti */
        return(zone.val);
}
```

Conversion de float "réseau" en float "local" (*ntohf*)

```
#include <sys/types.h>                                  /* Types prédéfinis 'c' */
#include <netinet/in.h>                                 /* Socket internet */

/* Fonction de conversion de float "réseau" en float "local" */
float ntohf(
        float netfloat)                                 /* Nombre à convertir */
{
        /* Déclaration des variables */
        ushort i;                                       /* Indice de boucle */
        ushort j;                                       /* Indice de boucle */
        ushort convert;                                 /* Zone de conversion */

        union {
                float val;                              /* Stockage du nombre */
                u_char p[4];                            /* Décomposition du nombre */
        }zone;                                          /* Zone de travail */

        /* Remplisssage nombre à convertir */
        zone.val=netfloat;

        /* Travail sur chaque extrémités de la zone en revenant vers le centre */
        for (i=0, j=3; i < j; i++, j--)
        {
                /* Copie des parties de la zone dans un entier court de conversion */
                convert=(zone.p[i] << 8) + zone.p[j];

                /* Conversion de l'entier court par la primitive normalisée "ntohs" */
                convert=ntohs(convert);

                /* Recopie des octets de l'entier court dans les parties de la zone */
                zone.p[i]=convert >> 8;
                zone.p[j]=convert & 0x00ff;
        }

        /* Renvoi nombre converti */
        return(zone.val);
}
```

c) La règle du jeu

L'utilisation de ces primitives se fait de la façon suivante :

✓ tout nombre qui doit être envoyé au réseau devra être filtré par une primitive "***htonx()***" et c'est la valeur renvoyée par la primitive qui devra partir sur le réseau

✓ tout nombre arrivant du réseau devra être filtré par une primitive "***ntohx()***" et la valeur renvoyée par la primitive sera la vrai valeur à traiter dans le programme

2) Fonctions de manipulation de zones mémoires

Les primitives ***memset()***, ***memcpy()*** et ***memcmp()*** permettent respectivement de remplir tous les octets d'une zone mémoire avec une valeur sur un octet, copier une zone mémoire dans une autre et comparer deux zones mémoires. Ces fonctions peuvent être utiles lorsqu'il s'agit de transférer une structure contenant une information réseau vers une structure nécessaire à une socket par exemple.

```
#include <sys/types.h>
void *memset (void *buffer, int val, size_t nb);
void *memcpy (void *dest, void *source, size_t nb);
int memcmp (void *buf1, void *buf2, size_t nb);
```

Explication des paramètres :

- ✓ void *buffer : pointeur universel (***void ****) vers la zone mémoire que l'on désire initialiser
- ✓ void *dest : pointeur universel (***void ****) vers la zone mémoire destinatrice de la recopie
- ✓ void *source : pointeur universel (***void ****) vers la zone mémoire contenant la source à recopier
- ✓ void *buf1 : pointeur universel (***void ****) vers la zone mémoire n° 1 que l'on désire comparer
- ✓ void *buf2 : pointeur universel (***void ****) vers la zone mémoire n° 2 que l'on désire comparer
- ✓ int val : valeur d'initialisation
- ✓ size_t nb : nombre d'octets sur lesquels s'appliquera chaque fonction

Explication des paramètres (int) : résultat de la comparaison.

- ✓ 0 : chaque caractère des deux zones est identique
- ✓ n<0 : la zone 1 est inférieure à la zone 2 au caractère ***-n***
- ✓ n>0 : la zone 1 est supérieure à la zone 2 au caractère ***n***

3) Autres fonctions

Les primitives ***inet_addr()*** et ***inet_ntoa()*** permettent de transformer une adresse IP (a.b.c.d) en nombre adapté à la forme réseau (actuellement sur 4 octets) et inversement.

```
#include <arpa/inet.h>
unsigned long inet_addr (char *ip_adr);
char *inet_ntoa (struct in_addr in_adr);
```

Explication des paramètres :

- ✓ char *ip_adr : chaîne de caractères contenant l'adresse sous forme IP (a.b.c.d)
- ✓ struct in_addr in_adr : adresse sous forme réseau (actuellement un nombre sur 4 octets)

Valeurs renvoyées :

- ✓ unsigned long inet_addr() : adresse adaptée au réseau. Ici, la logique voudrait que cette primitive renvoie un ***struct in_addr***
- ✓ char *inet_ntoa() : pointeur sur un chaîne contenant l'adresse sous forme IP (a.b.c.d)

La primitive ***select()*** permet au serveur de gérer des demandes de connexions arrivantes sur plusieurs descripteurs. Cette primitive n'est pas spécifique aux sockets mais peut y être appliquée. Cette primitive est bloquante jusqu'à ce que :

- ✓ un événement attendu se produise
- ✓ le temps spécifié soit écoulé
- ✓ une interruption soit survenue

```
#include <sys/time.h>
#include <sys/select.h>
int select (int nb, int *in, int *out, int *ex, struct timeval *wait);
```

Explication des paramètres :

- ✓ int nb : nombre de descripteurs gérés
- ✓ int *in : pointeur sur un ensemble (tableau) de descripteurs en lecture (stdin)
- ✓ int *out : pointeur sur un ensemble (tableau) de descripteurs en écriture (stdout)
- ✓ int *ex : : pointeur sur un ensemble (tableau) de descripteurs d'exception (stderr)
- ✓ struct timeval *wait : pointeur vers une variable de type ***struct timeval*** contenant le temps d'exécution imparti à la primitive

VII) EXEMPLES

1) Communication mode "connecté" en AF_INET

a) Client

Ce programme essaye en permanence de se connecter via TCP/IP sur un serveur. Celui-ci peut se trouver sur la même machine ou bien sur une autre. Dans ce dernier cas, il faut lancer le client en lui indiquant le nom de la machine sur laquelle se trouve le serveur.
Une fois connecté, le client attend une chaîne au clavier et envoie la chaîne tapée au serveur.
Le client s'arrête dès qu'on entre la chaîne "EOT".

```
#include <sys/types.h>                          /* Types prédéfinis "c" */
#include <sys/socket.h>                         /* Généralités sockets */
#include <sys/param.h>                          /* Paramètres et limites système */
#include <netinet/in.h>                         /* Spécifications socket internet */
#include <stdio.h>                              /* I/O fichiers classiques */
#include <string.h>                             /* Gestion chaines de caractères */
#include <netdb.h>                              /* Gestion network database */
#include <errno.h>                              /* Erreurs système */

extern const char* const sys_errlist[];         /* Liste messages erreurs */

#define SERVEUR_DEFAULT     ("localhost")       /* Nom serveur utilisé par défaut */
#define SERVICE_LABEL       ("essai")           /* Nom service requis */
#define SERVICE_PROTOCOL    ("tcp")             /* Protocole service requis */
#define SZ_BUF              (256)               /* Taille buffer */

main(
        int argc,                               /* Nbre arg. */
        char *argv[])                           /* Ptr arguments */
{
        /* Déclaration des variables */
        ushort i;                               /* Indice de boucle */
        ushort j;                               /* Indice de boucle */

        int sk_connect;                         /* Socket de connection */
        int sk_dialog;                          /* Socket de dialogue */

        char buf[SZ_BUF];                       /* Buffer texte */
        char hostname[MAXHOSTNAMELEN + 1];      /* Nom machine locale */
        char *serveur;                          /* Ptr vers le nom du serveur */
        char *pt;                               /* Ptr vers un caractère qcq. */

        struct sockaddr_in adr_serveur;         /* Adresse socket serveur */
        struct hostent *host_info;              /* Info. host */
        struct servent *service_info;           /* Info. service demandé */

        /* Remplissage du nom du serveur */
        if (argc > 1)
                /* Le nom du serveur est l'argument n°1 */
                serveur=argv[1];
        else
                /* Le nom du serveur est pris par défaut */
                serveur=SERVEUR_DEFAULT;

        /* Récuperation nom machine locale (juste pour l'exemple) */
        if (gethostname(hostname, MAXHOSTNAMELEN) != 0)
        {
                fprintf(stderr, "ligne %u - gethostname(%s) - %s\n", __LINE__, hostname,
sys_errlist[errno]);
```

```
        }       exit(errno);
        printf("gethostname='%s'\n", hostname);

        /* Récuperation informations serveur */
        if ((host_info=gethostbyname(serveur)) == NULL)
        {
                fprintf(stderr, "ligne %u - gethostbyname(%s) - %s\n", __LINE__, serveur,
sys_errlist[errno]);
                exit(errno);
        }
        fputc('\n', stdout);
        printf("host_info.h_name='%s'\n", host_info->h_name);
        for (i=0; host_info->h_aliases[i] != NULL; i++)
                printf("host_info.h_aliase[%hu]='%s'\n", i, host_info->h_aliases[i]);
        printf("host_info.h_addrtype=%u\n", host_info->h_addrtype);
        printf("host_info.h_length=%u\n", host_info->h_length);
        for (i=0; host_info->h_addr_list[i] != NULL; i++)
        {
                printf("host_info.h_addr_list[%hu]=", i);
                for (j=0; j < host_info->h_length; j++)
                        printf("%hu ", (unsigned char)host_info->h_addr_list[i][j]);
                fputc('\n', stdout);
        }

        /* Récuperation port dans "/etc/services" */
        if ((service_info=getservbyname(SERVICE_LABEL, SERVICE_PROTOCOL)) ==NULL)
        {
                fprintf(stderr, "ligne %u - getservbyname(%s, %s) - %s\n", __LINE__,
SERVICE_LABEL, SERVICE_PROTOCOL, sys_errlist[errno]);
                exit(errno);
        }
        fputc('\n', stdout);
        printf("service_name='%s'\n", service_info->s_name);
        for (i=0; service_info->s_aliases[i] != NULL; i++)
                printf("service_s_aliase[%hu]='%s'\n", i, service_info->s_aliases[i]);
        printf("service_port=%hu\n", ntohs(service_info->s_port));
        printf("service_protocole='%s'\n", service_info->s_proto);

        /* Remplissage adresse socket */
        memset(&adr_serveur, 0, sizeof(struct sockaddr_in));
        adr_serveur.sin_len=host_info->h_length;
        adr_serveur.sin_family=AF_INET;
        adr_serveur.sin_port=service_info->s_port;
        memcpy(&adr_serveur.sin_addr.s_addr, host_info->h_addr, host_info->h_length);

        /* Tentative de connection en boucle permanente */
        fputc('\n', stdout);
        do {
                /* Création socket */
                if ((sk_dialog=socket(AF_INET, SOCK_STREAM, 0)) == (-1))
                {
                        fprintf(stderr, "ligne %u - socket() - %s\n", __LINE__, sys_errlist[errno]);
                        exit(errno);
                }

                /* Connection au serveur */
                if ((sk_connect=connect(sk_dialog, &adr_serveur, sizeof(struct sockaddr_in))) == (-
1))
                {
                        fprintf(stderr, "ligne %u - connect() - %s\n", __LINE__, sys_errlist[errno]);
                        sleep(5);
```

```
        } while (sk_connect == (-1));
        printf("Connection réussie\n");

        /* Saisie et envoi de la chaîne en boucle */
        do {
                /* Saisie de la chaîne */
                fputs("Entrer chaine (EOT pour finir) :", stdout); fflush(stdout);
                fflush(stdin); fgets(buf, SZ_BUF, stdin);

                /* Suppression de la chaîne saisie du caractère '\n' s'il y est */
                if ((pt=strchr(buf, '\n')) != NULL)
                        *pt='\0';

                /* Envoi de la chaîne sur la socket */
                if (write(sk_dialog, buf, strlen(buf) + 1) == (-1))
                        fprintf(stderr, "ligne %u - write(%s) - %s\n", __LINE__, buf,
sys_errlist[errno]);
        } while (strcmp(buf, "EOT") != 0);

        /* Fermeture socket et fin du programme */
        close(sk_dialog);
        return(0);
}
```

b) Serveur

Ce programme attend en permanence via TCP/IP des connexions en provenances de clients éventuels (il ne s'arrêtera donc théoriquement jamais).
Dès qu'un client arrive, le serveur génère un processus fils qui aura la charge d'écouter le client pendant que le père se remet en attente.
Le serveur fils récupère et affiche les messages venant du client. Dès qu'un message "EOT" arrive, le serveur fils s'attête et disparaît.

```
#include <sys/types.h>                          /* Types prédéfinis "c" */
#include <sys/socket.h>                         /* Généralités sockets */
#include <sys/param.h>                          /* Paramètres et limites système */
#include <netinet/in.h>                         /* Spécifications socket internet */
#include <arpa/inet.h>                          /* Adresses format "arpanet" */
#include <signal.h>                             /* Signaux de communication */
#include <stdio.h>                              /* I/O fichiers classiques */
#include <netdb.h>                              /* Gestion network database */
#include <errno.h>                              /* Erreurs système */

extern const char* const sys_errlist[];         /* Liste messages erreurs */

#define SERVICE_LABEL           ("essai")       /* Nom service requis */
#define SERVICE_PROTOCOL        ("tcp")         /* Protocole service requis */
#define SZ_BUF                  (256)           /* Taille buffer */

int main(
        int argc,                               /* Nbre arguments */
        char *argv[])                           /* Ptr arguments */
{
        /* Déclaration des variables */
        ushort i;                               /* Indice de boucle */
        ushort j;                               /* Indice de boucle */

        int sk_creat;                           /* Socket de création */
        int sk_dialog;                          /* Socket de dialogue */
        int pid;                                /* Process créé */
        int len_adr;                            /* Taille adresse */
```

```
        int sz_read;                                    /* Nbre octets lus */
        char buf[SZ_BUF];                               /* Buffer texte */
        char hostname[MAXHOSTNAMELEN + 1];              /* Nom machine locale */

        struct sockaddr_in adr_serveur;                 /* Adresse socket serveur */
        struct sockaddr_in adr_client;                  /* Adresse socket client */
        struct hostent *host_info;                      /* Info. host client connecté */
        struct servent *service_info;                   /* Info. service demandé */

        char *adr_ascii;                                /* Adresse client mode ascii */

        /* Détournement du signal émis à la mort du fils (il ne reste pas zombie) */
        signal(SIGCHLD, SIG_IGN);

        /* Récuperation nom machine locale (juste pour l'exemple) */
        if (gethostname(hostname, MAXHOSTNAMELEN) != 0)
        {
                fprintf(stderr, "ligne %u - gethostname(%s) - %s\n", __LINE__, hostname,
sys_errlist[errno]);
                exit(errno);
        }
        printf("gethostname='%s'\n", hostname);

        /* Récuperation port dans "/etc/services" */
        if ((service_info=getservbyname(SERVICE_LABEL, SERVICE_PROTOCOL)) == NULL)
        {
                fprintf(stderr, "ligne %u - getservbyname(%s, %s) - %s\n", __LINE__,
SERVICE_LABEL, SERVICE_PROTOCOL, sys_errlist[errno]);
                exit(errno);
        }
        fputc('\n', stdout);
        printf("service_name='%s'\n", service_info->s_name);
        for (i=0; service_info->s_aliases[i] != NULL; i++)
                printf("service_s_aliase[%hu]='%s'\n", i, service_info->s_aliases[i]);
        printf("service_port=%hu\n", ntohs(service_info->s_port));
        printf("service_protocole='%s'\n", service_info->s_proto);

        /* Création socket */
        if ((sk_creat=socket(AF_INET, SOCK_STREAM, 0)) == (-1))
        {
                fprintf(stderr, "ligne %u - socket() - %s\n", __LINE__, sys_errlist[errno]);
                exit(errno);
        }
        printf("Socket créée\n");

        /* Remplissage adresse socket */
        memset(&adr_serveur, 0, sizeof(struct sockaddr_in));
        adr_serveur.sin_family=AF_INET;
        adr_serveur.sin_port=service_info->s_port;
        adr_serveur.sin_addr.s_addr=INADDR_ANY;

        /* Identification socket/réseau */
        if (bind(sk_creat, &adr_serveur, sizeof(struct sockaddr_in)) == (-1))
        {
                fprintf(stderr, "ligne %u - bind() - %s\n", __LINE__, sys_errlist[errno]);
                exit(errno);
        }
        printf("Socket connectée au réseau\n");

        /* Ecoute de la ligne */
        listen(sk_creat, 1);
```

```
	/* Attente permanente */
	fputc('\n', stdout);
	while (1)
	{
		printf("ppid=%u, pid=%u\tAttente entrée...", getppid(), getpid());
		fflush(stdout);

		/* Attente connexion client */
		len_adr=sizeof(struct sockaddr_in);
		if ((sk_dialog=accept(sk_creat, &adr_client, &len_adr)) == (-1))
		{
			fprintf(stderr, "ligne %u - accept() - %s\n", __LINE__, sys_errlist[errno]);
			exit(errno);
		}

		/* Client connecté */
		fputs("Entrée émise ", stdout);

		/* Transformation adresse net en ascii */
		if ((adr_ascii=inet_ntoa(adr_client.sin_addr)) > (char*)0)
		{
			printf("(adr=%s", adr_ascii);

			/* Récupération informations sur client par son adresse */
			if ((host_info=gethostbyaddr((char*)&adr_client.sin_addr.s_addr,
sizeof(struct in_addr), AF_INET)) != NULL)
				printf(" - %s)\n", host_info->h_name);
			else
			{
				fputs("- ???)\n", stdout);
				fprintf(stderr, "ligne %u - gethostbyaddr() - %s\n", __LINE__,
sys_errlist[errno]);
			}
		}
		else
		{
			fputs("(adr=???)\n", stdout);
			fprintf(stderr, "ligne %u - inet_ntoa() - %s\n", __LINE__, sys_errlist[errno]);
		}

		/* Duplication du process */
		switch (pid=fork())
		{
			case (-1): /* Erreur de fork */
				close(sk_creat);
				close(sk_dialog);
				fprintf(stderr, "ligne %u - fork() - %s\n", __LINE__, sys_errlist[errno]);
				exit(errno);

			case 0: /* Fils */
				/* Fermeture socket inutilisée */
				close(sk_creat);

				/* Lecture en boucle sur la socket */
				while ((sz_read=read(sk_dialog, buf, SZ_BUF)) > 0)
				{
					printf("\n\tppid=%u, pid=%u\tLe serveur a lu '%s'%s\n",
getppid(), getpid(), buf, strcmp(buf, "EOT") != 0 ?"" :"=> Fin de communication");

					/* Si la chaine contient "EOT" */
					if (strcmp(buf, "EOT") == 0)
```

```
                        }               break;

                        /* Si l'arrêt de la lecture est dû à une erreur */
                        if (sz_read == (-1))
                        {
                                close(sk_dialog);
                                fprintf(stderr, "ligne %u - read() - %s\n", __LINE__,
sys_errlist[errno]);
                                exit(errno);
                        }

                        /* Fin du fils */
                        close(sk_dialog);
                        exit(0);

                default: /* Père */
                        close(sk_dialog);
                }
        }
        /* Pas de sortie de programme - Boucle infinie */

        /* Fermeture socket et fin théorique du programme (pour être propre) */
        close(sk_creat);
        return(0);
}
```

c) Mise en œuvre

Il suffit de rajouter dans le fichier "/etc/services" de chaque machine (*cf. Administration Unix*) la ligne suivante :

```
essai           5000/tcp        cours
```

2) Communication mode "non-connecté" en AF_INET

a) Client

Ce programme essaye en permanence de se connecter via TCP/IP sur un serveur. Celui-ci peut se trouver sur la même machine ou bien sur une autre. Dans ce dernier cas, il faut lancer le client en lui indiquant le nom de la machine sur laquelle se trouve le serveur.
Une fois connecté, le client attend une chaîne au clavier et envoie la chaîne tapée au serveur.
Le client s'arrête dès qu'on entre la chaîne "EOT".

```
#include <sys/types.h>                                  /* Types prédéfinis "c" */
#include <sys/socket.h>                                 /* Généralités sockets */
#include <sys/param.h>                                  /* Paramètres et limites système */
#include <netinet/in.h>                                 /* Spécifications socket internet */
#include <stdio.h>                                      /* I/O fichiers classiques */
#include <string.h>                                     /* Gestion chaines de caractères */
#include <netdb.h>                                      /* Gestion network database */
#include <errno.h>                                      /* Erreurs système */

extern const char* const sys_errlist[];                 /* Liste messages erreurs */

#define SERVEUR_DEFAULT         ("localhost")           /* Nom serveur utilisé par défaut */
#define SERVICE_LABEL           ("essai")               /* Nom service requis */
#define SERVICE_PROTOCOL        ("udp")                 /* Protocole service requis */
#define SZ_BUF                  (256)                   /* Taille buffer */

int main(
        int argc,                                       /* Nbre arguments */
```

```
{       char *argv[])                                   /* Ptr arguments */
        /* Déclaration des variables */
        ushort i;                                       /* Indice de boucle */
        ushort j;                                       /* Indice de boucle */

        int sk_dialog;                                  /* Socket de dialogue */

        char buf[SZ_BUF];                               /* Buffer texte */
        char hostname[MAXHOSTNAMELEN + 1];              /* Nom machine locale */
        char *pt;                                       /* Ptr vers un caractère qcq. */
        char *serveur;                                  /* Ptr vers le nom du serveur */

        struct sockaddr_in adr_serveur;                 /* Adresse socket serveur */
        struct hostent *host_info;                      /* Info. host */
        struct servent *service_info;                   /* Info. service demandé */

        /* Remplissage du nom du serveur */
        if (argc > 1)
                /* Le nom du serveur est l'argument n°1 */
                serveur=argv[1];
        else
        {
                /* Le nom du serveur est pris par défaut */
                serveur=SERVEUR_DEFAULT;
                printf("Pas d'argument pour %s - Utilisation de %s\n", *argv, serveur);
        }

        /* Récuperation nom machine locale (juste pour l'exemple) */
        if (gethostname(hostname, MAXHOSTNAMELEN) != 0)
        {
                fprintf(stderr, "ligne %u - gethostname(%s) - %s\n", __LINE__, hostname,
sys_errlist[errno]);
                exit(errno);
        }
        printf("gethostname='%s'\n", hostname);

        /* Récuperation informations serveur */
        if ((host_info=gethostbyname(serveur)) == NULL)
        {
                fprintf(stderr, "ligne %u - gethostbyname(%s) - %s\n", __LINE__, serveur,
sys_errlist[errno]);
                exit(errno);
        }
        fputc('\n', stdout);
        printf("host_info.h_name='%s'\n", host_info->h_name);
        for (i=0; host_info->h_aliases[i] != NULL; i++)
                printf("host_info.h_aliase[%hu]='%s'\n", i, host_info->h_aliases[i]);
        printf("host_info.h_addrtype=%u\n", host_info->h_addrtype);
        printf("host_info.h_length=%u\n", host_info->h_length);
        for (i=0; host_info->h_addr_list[i] != NULL; i++)
        {
                printf("host_info.h_addr_list[%hu]=", i);
                for (j=0; j < host_info->h_length; j++)
                        printf("%hu ", (unsigned char)host_info->h_addr_list[i][j]);
                fputc('\n', stdout);
        }

        /* Récuperation port dans "/etc/services" */
        if ((service_info=getservbyname(SERVICE_LABEL, SERVICE_PROTOCOL)) == NULL)
        {
```

```
                fprintf(stderr, "ligne %u - getservbyname(%s, %s) - %s\n", __LINE__,
SERVICE_LABEL, SERVICE_PROTOCOL, sys_errlist[errno]);
                exit(errno);
        }
        fputc('\n', stdout);
        printf("service_name='%s'\n", service_info->s_name);
        for (i=0; service_info->s_aliases[i] != NULL; i++)
                printf("service_s_aliase[%hu]='%s'\n", i, service_info->s_aliases[i]);
        printf("service_port=%hu\n", ntohs(service_info->s_port));
        printf("service_protocole='%s'\n", service_info->s_proto);

        /* Création socket */
        if ((sk_dialog=socket(AF_INET, SOCK_DGRAM, 0)) == (-1))
        {
                fprintf(stderr, "ligne %u - socket() - %s\n", __LINE__, sys_errlist[errno]);
                exit(errno);
        }
        printf("Socket créée\n");

        /* Remplissage adresse socket */
        memset(&adr_serveur, 0, sizeof(struct sockaddr_in));
        adr_serveur.sin_len=host_info->h_length;
        adr_serveur.sin_family=AF_INET;
        adr_serveur.sin_port=service_info->s_port;
        memcpy(&adr_serveur.sin_addr.s_addr, host_info->h_addr, host_info->h_length);

        /* Saisie et envoi de la chaîne en boucle */
        do
        {
                /* Saisie de la chaîne */
                fputs("Entrer chaine (EOT pour finir) :", stdout); fflush(stdout);
                fflush(stdin); fgets(buf, SZ_BUF, stdin);

                /* Suppression de la chaîne saisie le caractère '\n' s'il y est */
                if ((pt=strchr(buf, '\n')) != NULL)
                        *pt='\0';

                /* Envoi de la chaîne modifiée sur la socket */
                if (sendto(sk_dialog, buf, strlen(buf) + 1, 0, &adr_serveur, sizeof(struct
sockaddr_in)) == (-1))
                        fprintf(stderr, "ligne %u - sendto(%s) - %s\n", __LINE__, buf,
sys_errlist[errno]);
        }
        while (strcmp(buf, "EOT") != 0);

        /* Fermeture socket et fin du programme */
        close(sk_dialog);
        return(0);
}
```

b) Serveur

Ce programme attend en permanence des connexions via TCP/IP en provenances de clients éventuels (il ne s'arrêtera donc théoriquement jamais).
Dès qu'un client arrive, le serveur génère un processus fils qui aura la charge d'écouter le client pendant que le père se remet en attente.
Le serveur fils récupère et affiche les messages venant du client. Dès qu'un message "EOT" arrive, le serveur fils s'arrête et disparaît.

```
#include <sys/types.h>                  /* Types prédéfinis "c" */
#include <sys/socket.h>                 /* Généralités sockets */
#include <sys/param.h>                  /* Paramètres et limites système */
```

```
#include <netinet/in.h>                          /* Spécifications socket internet */
#include <arpa/inet.h>                           /* Adresses format "arpanet" */
#include <stdio.h>                               /* I/O fichiers classiques */
#include <netdb.h>                               /* Gestion network database */
#include <errno.h>                               /* Erreurs système */

extern const char* const sys_errlist[];          /* Liste messages erreurs */

#define SERVICE_LABEL           ("essai")        /* Nom service requis */
#define SERVICE_PROTOCOL        ("udp")          /* Protocole service requis */
#define SZ_BUF                  (256)            /* Taille buffer */

int main(
        int argc,                                /* Nbre arguments */
        char *argv[])                            /* Ptr arguments */
{
        /* Déclaration des variables */
        ushort i;                                /* Indice de boucle */
        ushort j;                                /* Indice de boucle */

        int sk_dialog;                           /* Socket de dialogue */
        int len_adr;                             /* Taille adresse */

        char buf[SZ_BUF];                        /* Buffer texte */
        char hostname[MAXHOSTNAMELEN + 1];       /* Nom machine locale */

        struct sockaddr_in adr_serveur;          /* Adresse socket serveur */
        struct sockaddr_in adr_client;           /* Adresse socket client */
        struct hostent *host_info;               /* Info. host client connecté */
        struct servent *service_info;            /* Info. service demandé */

        char *adr_ascii;                         /* Adresse client mode ascii */

        /* Récuperation nom machine locale (juste pour l'exemple) */
        if (gethostname(hostname, MAXHOSTNAMELEN) != 0)
        {
                fprintf(stderr, "ligne %u - gethostname(%s) - %s\n", __LINE__, hostname,
sys_errlist[errno]);
                exit(errno);
        }
        printf("gethostname='%s'\n", hostname);

        /* Récuperation port dans "/etc/services" */
        if ((service_info=getservbyname(SERVICE_LABEL, SERVICE_PROTOCOL)) == NULL)
        {
                fprintf(stderr, "ligne %u - getservbyname(%s, %s) - %s\n", __LINE__,
SERVICE_LABEL, SERVICE_PROTOCOL, sys_errlist[errno]);
                exit(errno);
        }
        fputc('\n', stdout);
        printf("service_name='%s'\n", service_info->s_name);
        for (i=0; service_info->s_aliases[i] != NULL; i++)
                printf("service_s_aliase[%hu]='%s'\n", i, service_info->s_aliases[i]);
        printf("service_port=%hu\n", ntohs(service_info->s_port));
        printf("service_protocole='%s'\n", service_info->s_proto);

        /* Création socket */
        if ((sk_dialog=socket(AF_INET, SOCK_DGRAM, 0)) == (-1))
        {
                fprintf(stderr, "ligne %u - socket() - %s\n", __LINE__, sys_errlist[errno]);
                exit(errno);
        }
```

```
        fputs("Socket créée\n", stdout);
        /* Remplissage adresse socket */
        memset(&adr_serveur, 0, sizeof(struct sockaddr_in));
        adr_serveur.sin_family=AF_INET;
        adr_serveur.sin_port=service_info->s_port;
        adr_serveur.sin_addr.s_addr=INADDR_ANY;

        /* Identification socket/réseau */
        if (bind(sk_dialog, &adr_serveur, sizeof(struct sockaddr_in)) == (-1))
        {
                fprintf(stderr, "ligne %u - bind() - %s\n", __LINE__, sys_errlist[errno]);
                exit(errno);
        }
        fputs("Socket reliée au réseau\n", stdout);

        /* Lecture en boucle sur la socket */
        len_adr=sizeof(struct sockaddr_in);
        while (1)
        {
                if (recvfrom(sk_dialog, buf, SZ_BUF, 0, &adr_client, &len_adr) == (-1))
                        fprintf(stderr, "ligne %u - recvfrom() - %s\n", __LINE__, sys_errlist[errno]);

                /* Transformation adresse client en ascii */
                if ((adr_ascii=inet_ntoa(adr_client.sin_addr)) > (char*)0)
                {
                        printf("From client %s ", adr_ascii);

                        /* Récupération informations sur client par son adresse */
                        if ((host_info=gethostbyaddr((char*)&adr_client.sin_addr.s_addr,
sizeof(struct in_addr), AF_INET)) != NULL)
                                printf("(%s)", host_info->h_name);
                        else
                        {
                                fputs("(???)", stdout);
                                fprintf(stderr, "ligne %u - gethostbyaddr() - %s\n", __LINE__,
sys_errlist[errno]);
                        }
                }
                else
                {
                        fputs("From client ???", stdout);
                        fprintf(stderr, "ligne %u - inet_ntoa() - %s\n", __LINE__, sys_errlist[errno]);
                }
                printf(" - Le serveur a lu '%s'%s\n", buf, strcmp(buf, "EOT") != 0 ?"" :"=> Fin de
communication");
        }
        /* Pas de sortie de programme - Boucle infinie */

        /* Fermeture socket et fin théorique du programme (pour être propre) */
        close(sk_dialog);
        return(0);
}
```

c) Mise en œuvre

Il suffit de rajouter dans le fichier "/etc/services" de chaque machine (*cf. Administration Unix*) la ligne suivante :

```
essai           5000/udp            cours
```

3) Communication en AF_UNIX

a) Client

Ce programme essaye en permanence de se connecter via un fichier "*socket*" sur un serveur situé sur la même machine. Une fois connecté, le client attend une chaîne au clavier et envoie la chaîne tapée au serveur.
Le client s'arrête dès qu'on entre la chaîne "EOT".

```
#include <sys/types.h>                          /* Types prédéfinis "c" */
#include <sys/socket.h>                         /* Généralités sockets */
#include <sys/un.h>                             /* Spécifications socket unix */
#include <stdio.h>                              /* I/O fichiers classiques */
#include <string.h>                             /* Gestion chaines de caractères */
#include <errno.h>                              /* Erreurs système */

extern const char* const sys_errlist[];         /* Liste messages erreurs */

#define NAME_SOCKET             ("socket_file") /* Nom fichier socket */
#define SZ_BUF                  (256)           /* Taille buffer */

int main(
        int argc,                               /* Nbre arg. */
        char *argv[])                           /* Ptr arguments */
{
        /* Déclaration des variables */
        int sk_connect;                         /* Socket de connection */
        int sk_dialog;                          /* Socket de dialogue */

        char buf[SZ_BUF];                       /* Buffer texte */
        char *pt;                               /* Ptr vers un caractère qcq. */

        struct sockaddr_un adr_serveur;         /* Adresse socket serveur */

        /* Remplissage adresse socket */
        memset(&adr_serveur, 0, sizeof(struct sockaddr_un));
        adr_serveur.sun_len=strlen(NAME_SOCKET) + 1;
        adr_serveur.sun_family=AF_UNIX;
        memcpy(&adr_serveur.sun_path, NAME_SOCKET, strlen(NAME_SOCKET));

        /* Tentative de connection en boucle permanente */
        fputc('\n', stdout);
        do
        {
                /* Création socket */
                if ((sk_dialog=socket(AF_UNIX, SOCK_STREAM, 0)) == (-1))
                {
                        fprintf(stderr, "ligne %u - socket() - %s\n", __LINE__, sys_errlist[errno]);
                        exit(errno);
                }

                /* Connection au serveur */
                if ((sk_connect=connect(sk_dialog, &adr_serveur, sizeof(struct sockaddr_un))) ==
(-1))
                {
                        fprintf(stderr, "ligne %u - connect() - %s\n", __LINE__, sys_errlist[errno]);
                        sleep(5);
                }
        }
        while (sk_connect == (-1));
        printf("Connection réussie\n");
```

```
        /* Saisie et envoi de la chaîne en boucle */
        do
        {
                /* Saisie de la chaîne */
                fputs("Entrer chaine (EOT pour finir) :", stdout); fflush(stdout);
                fflush(stdin); fgets(buf, SZ_BUF, stdin);

                /* Suppression de la chaîne saisie le caractère '\n' s'il y est */
                if ((pt=strchr(buf, '\n')) != NULL)
                        *pt='\0';

                /* Envoi de la chaîne modifiée sur la socket */
                if (write(sk_dialog, buf, strlen(buf) + 1) == (-1))
                        fprintf(stderr, "ligne %u - write(%s) - %s\n", __LINE__, buf,
sys_errlist[errno]);
        }
        while (strcmp(buf, "EOT") != 0);

        /* Fermeture socket et fin du programme */
        close(sk_dialog);
        return(0);
}
```

b) Serveur

Ce programme attend en permanence des connexions via un fichier "*socket*" en provenances de clients éventuels.
Dès qu'un client arrive, le serveur génère un processus fils qui aura la charge d'écouter le client pendant que le père se remet en attente (il ne s'arrêtera donc théoriquement jamais).
Le serveur fils récupère et affiche les messages venant du client. Dès qu'un message "EOT" arrive, le serveur fils s'attête et disparaît.

```
#include <sys/types.h>                          /* Types prédéfinis "c" */
#include <sys/socket.h>                         /* Généralités sockets */
#include <sys/un.h>                             /* Spécifications socket unix */
#include <stdio.h>                              /* I/O fichiers classiques */
#include <signal.h>                             /* Signaux de communication */
#include <errno.h>                              /* Erreurs système */

extern const char* const sys_errlist[];         /* Liste messages erreurs */

#define NAME_SOCKET             ("socket_file") /* Nom fichier socket */
#define SZ_BUF                  (256)           /* Taille buffer */

int main(
        int argc,                               /* Nbre arg. */
        char *argv[])                           /* Ptr arguments */
{
        /* Déclaration des variables */
        int sk_creat;                           /* Socket de création */
        int sk_dialog;                          /* Socket de dialogue */
        int pid;                                /* Process créé */
        int sz_read;                            /* Nombre octets lus */

        char buf[SZ_BUF];                       /* Buffer texte */

        struct sockaddr_un adr_serveur;         /* Adresse socket serveur */

        /* Détournement du signal émis à la mort du fils (il ne reste pas zombie) */
        signal(SIGCHLD, SIG_IGN);
```

```
        /* Effacement fichier socket résiduel */
        unlink(NAME_SOCKET1);

        /* Création socket */
        if ((sk_creat=socket(AF_UNIX, SOCK_STREAM, 0)) == (-1))
        {
                fprintf(stderr, "ligne %u - socket() - %s\n", __LINE__, sys_errlist[errno]);
                exit(errno);
        }

        /* Remplissage adresse socket */
        memset(&adr_serveur, 0, sizeof(struct sockaddr_un));
        adr_serveur.sun_len=strlen(NAME_SOCKET) + 1;
        adr_serveur.sun_family=AF_UNIX;
        memcpy(&(adr_serveur.sun_path), NAME_SOCKET, strlen(NAME_SOCKET));

        /* Identification socket/réseau */
        if (bind(sk_creat, &adr_serveur, sizeof(struct sockaddr_un)) == (-1))
        {
                fprintf(stderr, "ligne %u - bind() - %s\n", __LINE__, sys_errlist[errno]);
                exit(errno);
        }

        /* Prise de la ligne */
        listen(sk_creat, 1);

        /* Ecoute permanente */
        fputc('\n', stdout);
        while (1)
        {
                printf("ppid=%u, pid=%u\tAttente entrée...", getppid(), getpid());
                fflush(stdout);

                /* Attente connexion client */
                if ((sk_dialog=accept(sk_creat, NULL, NULL)) == (-1))
                {
                        fprintf(stderr, "ligne %u - accept() - %s\n", __LINE__, sys_errlist[errno]);
                        exit(errno);
                }
                fputs("Entrée émise\n", stdout);

                /* Duplication du process */
                switch (pid=fork())
                {
                        case (-1): /* Erreur de fork */
                                close(sk_creat);
                                close(sk_dialog);
                                fprintf(stderr, "ligne %u - fork() - %s\n", __LINE__, sys_errlist[errno]);
                                exit(errno);

                        case 0: /* Fils */
                                /* Fermeture socket inutilisée */
                                close(sk_creat);

                                /* Lecture en boucle sur la socket */
                                while ((sz_read=read(sk_dialog, buf, SZ_BUF)) > 0)
                                {
                                        printf("\n\tppid=%u, pid=%u\tLe serveur a lu '%s'%s\n",
getppid(), getpid(), buf, strcmp(buf, "EOT") != 0 ?"" :"=> Fin de communication");

                                        /* Si la chaine contient "EOT" */
                                        if (strcmp(buf, "EOT") == 0)
```

```
                                }               break;

                                /* Si l'arrêt de la lecture est dû à ne erreur */
                                if (sz_read == (-1))
                                {
                                        close(sk_dialog);
                                        fprintf(stderr, "ligne %u - read() - %s\n", __LINE__,
sys_errlist[errno]);
                                        exit(errno);
                                }

                                /* Fin du fils */
                                close(sk_dialog);
                                exit(0);

                        default: /* Père */
                                close(sk_dialog);
                }
        }
        /* Pas de sortie du programme - Boucle infinie */

        /* Fermeture socket et fin théorique du programme (pour être propre) */
        close(sk_creat);
        return(0);
}
```

INDEX