

# Développement Système sous Linux/Unix

V1.41

C. Drocourt

[drocourt@e-conseil.fr](mailto:drocourt@e-conseil.fr)

## SOMMAIRE

<b>Chapitre 1 - Outils et méthodes de développement.....</b>	<b>1</b>
1 - Environnement Linux.....	2
2 - Le développement sous Unix.....	7
3 - Make.....	27
4 - Outils de développement.....	43
5 - Gestion de versions.....	52
<b>Chapitre 2 - Gestion de la mémoire.....</b>	<b>1</b>
1 - Allocation de la mémoire.....	2
2 - Utilisation de la mémoire.....	8
3 - Mémoire virtuelle.....	12
<b>Chapitre 3 - Les processus.....</b>	<b>1</b>
1 - Accès aux données du BCP.....	2
2 - La primitive fork.....	3
3 - La primitive wait.....	6
4 - Les primitives exec.....	11

5 - Le mécanisme du fork/exec.....	12
<b>Chapitre 4 - Les signaux.....</b>	<b>1</b>
1 - Introduction.....	2
2 - L'envoi des signaux.....	5
3 - Le masquage des signaux.....	7
4 - Le captage des signaux.....	10
5 - L'attente d'un signal.....	14
6 - Les signaux temps-réel.....	15
<b>Chapitre 5 - Les entrées/sorties.....</b>	<b>1</b>
1 - Les différentes tables utilisées.....	2
2 - Les opérations de base.....	4
3 - Manipulation de descripteurs.....	11
4 - Manipulation de i-nœuds.....	17
5 - Consultation d'un i-nœud.....	19
6 - Modification des caractéristiques d'un i-nœud.....	26
7 - Les fichiers de type catalogue.....	27

8 - La bibliothèque d'entrée-sortie standard.....	30
9 - Les verrous externes.....	39
10 - Les verrous internes.....	43
<b>Chapitre 6 - Les threads.....</b>	<b>1</b>
1 - Introduction.....	2
2 - Les attributs d'un activité.....	4
3 - Création et terminaison des activités.....	6
4 - Les mutex .....	18
5 - Les conditions .....	25
6 - Autres mécanisme associés aux threads.....	35
7 - Particularités Linux.....	38
<b>Chapitre 7 - Communications entre processus.....</b>	<b>1</b>
1 - Introduction aux IPC « System V ».....	2
2 - Les sémaphores « System V ».....	9
3 - Segments de mémoire partagée « System V ».....	28
4 - Files de messages « System V ».....	39

5 - Les sémaphores POSIX.....	46
6 - Segments de mémoire partagée POSIX.....	60
7 - Files de messages POSIX.....	65
<b>Chapitre 8 - Communiquer sur le réseau.....</b>	<b>1</b>
1 - Introduction.....	2
2 - Domaines de sockets.....	5
3 - Primitives générales de manipulation.....	11
4 - Les fonctions de résolution et les fichiers administratifs.....	21
5 - La communication par datagrammes.....	32
6 - La communication en mode connecté.....	48
7 - Compatibilité avec Microsoft Windows.....	60
8 - Les obligations liées aux serveurs.....	64
9 - Les évènements d'un démon.....	69
10 - Socket au format brut (raw).....	75



# **Chapitre 8**

## **Communiquer sur le réseau**

# 1 - Introduction

## 1.1 - Les protocoles UDP et TCP

### UDP : Livraison non fiable, sans connexion

UDP (User Datagram Protocol) permet à une application d'envoyer des messages à une autre en mode datagramme non connecté. Les paquets UDP peuvent arriver dans le désordre, au même ne pas arriver : les états d'arrivée d'un paquet n'est pas géré par ce protocole.

### TCP : Transport de flot fiable

Le protocole TCP (Transmission Control Protocol) est utilisé pour réaliser une connexion de type circuit virtuel, connue sous le nom de connexion en flot avec :

- Accusé de réception pour chaque paquet,
- Réémission du paquet s'il n'est pas acquittée au bout d'un certain temps,
- Paquets numérotés qui peuvent donc être remis dans le bon ordre s'ils arrivent dans le désordre.

## Identification du service : les ports

Les adresses IP désignent les machines entre lesquelles les communications sont établies. Lorsqu'un processus désire entrer en communication avec un autre processus, il doit adresser le processus s'exécutant cette machine. L'adressage de ce processus est effectué selon un concept abstrait indépendant du système d'exploitation des machines.

Ces destinations abstraites permettant d'adresser un service applicatif s'appellent des ports de protocole. L'émission d'un message se fait sur la base d'un port source et un port destinataire.

Les processus disposent d'une interface système leur permettant de spécifier un port ou d'y accéder (socket). Les accès aux ports sont généralement synchrones, les opérations sur les ports sont tamponnés (files d'attente).

Les protocoles TCP et UDP utilisent des numéros de port séparés ; ainsi TCP/5 est totalement distinct de UDP/5. Les services TCP et UDP disponibles sont listés dans le fichier `/etc/services` mais d'autres services peuvent être ajoutés sur une machine : la connaissance de leur numéro de port et de leur protocole permettra alors à une application cliente de s'y connecter.

Certains services sont offerts à la fois par TCP et UDP : par convention, ils ont le même numéro de port (exemple DNS).

## 2 - Domaines de sockets

### 2.1 - Le principe

Un socket est un point de communication par lequel un processus peut émettre ou recevoir des informations.

La commande UNIX **netstat -a** affiche les sockets déjà utilisés.

Dans un processus, un socket est identifiée par un descripteur, comme un fichier, ce qui permet de lui appliquer les primitives telles que read, write...

- ⇒ Mise en réseau sans problème des applications standard
- ⇒ Héritage des sockets par fork
- ⇒ Redirections possibles

Les différentes constantes nécessaires sont macro-définies dans les fichiers `<sys/types.h>` et `<sys/socket.h>`.

## 2.2 - Domaine Unix

Un socket est **local** au système où elle est définie et possède une référence dans l'arborescence des fichiers d'Unix. La commande **ls -l** affiche un « s » en début de ligne pour signaler le type de ce fichier.

⇒ **IPC interne.**

La structure d'une adresse de socket est définie dans le fichier à inclure `<sys/un.h>` :

```
struct sockaddr_un {
    short    sun_family ; /* domaine UNIX: PF_UNIX */
    char     sun_path[108]; /* référence du fichier */
};
```

Pour un socket local on peut mettre indifféremment `AF_LOCAL`, `PF_LOCAL`, `AF_UNIX`, `PF_UNIX` et même `AF_FILE` et `PF_FILE`.

## 2.3 - Domaine Internet

Un socket permet un IPC **externe**, c'est à dire une communication entre des processus qui sont lancés sur des machines hôtes différentes. La structure d'une adresse de socket est définie dans le fichier à inclure `<netinet/in.h>` :

```
struct sockaddr_in {
    short      sin_family; /* domaine Internet: PF_INET */
    unsigned short sin_port; /* le numéro de port
    */
    struct in_addr sin_addr; /* l'adresse Internet */
    char          sin_zero[8]; /* champ de huit zéros */
};
```

```
struct in_addr {
    unsigned long    s_addr;
};
```

Ici encore, le domaine peut être AF\_INET ou PF\_INET.

Pour utiliser de l'IPv6 on utilisera AF\_INET6 ou PF\_INET6.

```
struct sockaddr_in6 {
    u_int16_t    sin6_family; // address family, AF_INET6
    u_int16_t    sin6_port;   // port number, Network Byte Order
    u_int32_t    sin6_flowinfo; // IPv6 flow information
    struct in6_addr sin6_addr; // IPv6 address
    u_int32_t    sin6_scope_id; // Scope ID
};
```

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

Dans ce domaine, l'association entre deux processus communiquant via un socket sera définie par cinq éléments :

- protocole (UDP ou TCP)
- adresse IP de la machine sur laquelle s'exécute le processus A
- port associé à A sur cette machine
- adresse IP de la machine sur laquelle s'exécute le processus B
- port associé à B sur cette machine

## 2.4 - Type datagramme

Correspond aux sockets destinées à la communication en mode non connecté pour l'envoi de datagrammes de taille bornée. Dans le domaine Internet, le protocole sous-jacent est **UDP/IP**.

## 2.5 - Type flot

Correspond aux sockets dédiées à la communication en mode connecté. Dans le domaine Internet, le protocole sous-jacent est **TCP/IP**.

## 3 - Primitives générales de manipulation

### 3.1 - Création

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

La primitive **socket** permet de créer un socket en précisant son domaine (PF\_UNIX : IPC local ou PF\_INET :IPC distant) , son type (SOCK\_STREAM en mode connecté ou SOCK\_DGRAM en mode datagramme) ainsi que le protocole utilisé. Il est souhaitable de laisser ce dernier paramètre à 0 de façon à laisser le système choisir lui-même le bon protocole.

La valeur de retour est un descripteur qui permet d'accéder au socket créé en cas de réussite, dans le cas contraire elle retourne -1 et positionne « errno » en conséquence..

### 3.2 - Suppression

Un socket est effectivement supprimée après la fermeture par **close** du dernier descripteur permettant d'y accéder : il y a alors libération des ressources utilisées.

### 3.3 - Attachement

Sans ce mécanisme, le socket ne serait connue que par le processus qui l'a créé et par sa descendance : exactement comme pour un tube ordinaire !!!

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

Un appel à cette primitive réalise l'attachement du socket de descripteur **sockfd** à l'adresse **\*addr**, le paramètre **addrlen** doit être égal à la taille en octets de cette adresse.

La structure **sockaddr** est générique : elle doit être remplacée par la structure adaptée au domaine de la socket.

Cette fonction retourne 0 en cas de réussite, dans le cas contraire elle retourne -1 et positionne « errno » en conséquence.

### 3.4 - Spécificité du domaine Unix

La structure « sockaddr\_un » doit être initialisé en conséquence et en particulier le champ « sun\_path » qui précise une référence dans le système de fichiers.

Il est possible de créer deux objets socket dans le domaine Unix, l'une pour lire et l'autre pour écrire à l'aide de la fonction suivante :

```
int socketpair(int dom,int typ,int prot,int *ptr_desc) ;
```

Cette primitive est utilisable pour les types SOCK\_STREAM et SOCK\_DGRAM. Le paramètre ptr\_desc est un tableau de deux entiers dans lequel on récupère les deux descripteurs permettant l'accès aux deux sockets.

### 3.5 - Exemple d'attachement d'un socket dans le domaine Unix

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

int main() {
    char *sock_name= "/tmp/reference"; int sd;
    struct sockaddr_un socketEcoute;

    sd = socket(PF_UNIX,SOCK_STREAM,0);
    /* etre sur que le fichier n'existe pas */
    unlink(sock_name);
    socketEcoute.sun_family = PF_UNIX;
    strcpy(socketEcoute.sun_path,sock_name);
    bind(sd,(struct sockaddr *)&socketEcoute,
        sizeof(struct sockaddr_un));
    system("netstat --unix|grep \"reference\" ");
    close(sd);
    exit(0);
}
```

### 3.6 - Spécificité du domaine Internet

La structure « `sockaddr_in` » doit être initialisée en conséquence. Le champ « `sin_addr.s_addr` » peut prendre la valeur **INADDR\_ANY** pour associer l'ensemble des adresses IPs de l'ordinateur à cette connexion (**IN6ADDR\_ANY\_INIT** pour l'IPv6). Il est également possible d'utiliser la constante **INADDR\_LOOPBACK** pour l'interface « `loopback` » en IPv4, et **IN6ADDR\_LOOPBACK\_INIT** pour les adresses IPv6.

Lorsqu'une autre adresse IP doit être recherchée, on utilise une fonction de résolution, comme **getaddrinfo()**, qui va être détaillée dans le prochain paragraphe.

Ecrire la valeur 0 dans le champ « `sin_port` » de l'adresse permet de laisser le système choisir lui-même le numéro de port : ceci peut être utile pour un processus client.

### 3.7 - Exemple d'attachement d'un socket dans le domaine Internet

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>

int main() {
    int desc;
    struct sockaddr_in adresse;
    int longueur=sizeof(struct sockaddr_in);

    desc=socket(PF_INET,SOCK_DGRAM,0);
    adresse.sin_family=PF_INET;
    adresse.sin_addr.s_addr=INADDR_ANY;
    adresse.sin_port=2013;
    bind(desc,(struct sockaddr *)&adresse,longueur);
    system("netstat -a --udp -p");
    exit(0);
}
```

### 3.8 - Le problème « endianness »

La manière d'organiser les octets dépend de l'architecture sur lequel un programme fonctionne. On peut distinguer essentiellement deux catégories :

- Big endian : Les données sont organisées du poids fort vers le poids faible,
- Little endian : Les données sont organisées du poids faible vers le poids fort,

Le format choisi pour la diffusion d'information dans les protocoles de communications est le Big endian, par conséquent, sur les autres plateformes, il faudra transformer les données avant de les utiliser.

#### Exemple :

Soit un entier court de valeur 2013 utilisé sur une plateforme Intel, qui travail en Little endian, la valeur hexadécimale de ce chiffre est donc 0x07DD, qui est représenté en interne par 0xDD07. Le format réseau va considérer que l'octet de poids fort est le premier et va donc interpréter le chiffre comme la valeur 56583...

Afin de pouvoir travailler de manière unique et portable sur l'ensemble des plateformes utilisables, il est nécessaire d'utiliser des fonctions de conversion au format réseau dont le fonctionnement interne sera adapté à la plateforme :

- `htons()` : Entier court local vers entier court réseau,
- `htonl()` : Entier long local vers entier long réseau,
- `ntohs()` : Entier court réseau vers entier court local,
- `ntohl()` : Entier long réseau vers entier court local,

Ainsi, dans l'exemple précédent, il faut écrire :

```
adresse.sin_addr.s_addr=htonl(INADDR_ANY);  
adresse.sin_port=htons(2013);
```

Par exemple, la constante `INADDR_LOOPBACK` est définie de la manière suivante :

```
#define INADDR_LOOPBACK    0x7f000001    /* 127.0.0.1    */
```

La fonction `htons()` pourrait par exemple être codé de la manière suivante par une directive de pré-compilation :

```
# if __BYTE_ORDER == __BIG_ENDIAN
/* The host byte order is the same as network byte order,
   so these functions are all just identity. */
# define ntohl(x)      (x)
# define ntohs(x)     (x)
# define htonl(x)     (x)
# define htons(x)     (x)
# else
#   if __BYTE_ORDER == __LITTLE_ENDIAN
#     define ntohl(x)  __bswap_32 (x)
#     define ntohs(x)  __bswap_16 (x)
#     define htonl(x)  __bswap_32 (x)
#     define htons(x)  __bswap_16 (x)
#   endif
# endif
```

### 3.9 - La fonction getsockname

La fonction « getsockname( ) » permet de retrouver l'adresse d'attachement d'un socket (utile si on a laissé le système choisir le numéro de port) .

```
#include <sys/socket.h>
int getsockname(int s, struct sockaddr *name,
                socklen_t *namelen) ;
```

Cette fonction prend en paramètres :

- s : Le descripteur de socket existant,
- name : Un pointeur sur une structure pouvant accueillir le résultat,
- namelen : La longueur de cette structure,

Cette fonction retourne 0 en cas de réussite, dans le cas contraire elle retourne -1 et positionne « errno » en conséquence.

## 4 - Les fonctions de résolution et les fichiers administratifs

### 4.1 - Fonctions obsolètes

Les fonctions suivantes existent et sont encore utilisées dans certains programmes mais sont obsolètes et doivent être remplacées par les fonctions décrites dans les paragraphes suivants :

- `gethostbyname()` : Permet la résolution de nom DNS vers adresse IP,
- `getservbyname()` : Permet de convertir le nom d'un service en une structure associée,
- `inet_ntoa()` : Permet de convertir une adresse IP en chaîne de caractères,
- `inet_addr()` : Permet de convertir une chaîne de caractères contenant l'adresse IP en une adresse IP de type « `in_addr` ».

## 4.2 - Format réseau et format affichable

Les adresses IPs manipulées ne permettent pas d'être affichées directement, par exemple une adresse IPv4 est un entier long dont chaque octet représente une partie de l'adresse IP. Il faut donc convertir cet entier en une chaîne de caractères :

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

Cette fonction prend en argument :

- « af » : Famille d'adresse (AF\_INET, AF\_INET6),
- « src » : Un pointeur sur la structure d'adresse,
- « dst » : Un chaîne de caractères pouvant accueillir le résultat,
- « cnt » : La taille du « buffer » précédent,

Cette fonction retourne un pointeur sur la chaîne de caractères en cas de réussite, dans le cas contraire elle retourne NULL et positionne « errno » en conséquence.

L'opération peut également être utilisée quand par exemple, l'utilisateur va saisir une adresse IP à l'intérieur d'une chaîne de caractères :

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

Cette fonction prend en argument :

- « af » : Famille d'adresse (AF\_INET, AF\_INET6),
- « src » : Une chaîne de caractères contenant l'adresse,
- « dst » : Un pointeur sur une structure d'adresse pouvant stocker le résultat,

Cette fonction retourne un pointeur sur la chaîne de caractères en cas de réussite, dans le cas contraire elle retourne NULL et positionne « errno » en conséquence.

Exemple :

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>
#include <arpa/inet.h>
#include <string.h>

int main() {
    int s; char buffer[1024];
    struct in_addr addr_IP4;

    addr_IP4.s_addr = htonl(INADDR_LOOPBACK);
    if(inet_ntop(AF_INET,&addr_IP4,buffer,1024)==NULL) {
        perror("inet...");
        exit(1);
    }
    printf("inet_ntop : %s\n",buffer);
    exit(0);
}
```

### 4.3 - Résolution de nom DNS

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netdb.h>
int getaddrinfo(const char *node, const char *service,
               const struct addrinfo *hints,
               struct addrinfo **res);
```

Cette fonction prend en paramètres :

- node : Le nom DNS ou l'adresse recherché sous forme de chaîne de caractères,
- service : Le service recherché (ex : http, 25, ...), NULL pour tous,
- hints : Un structure définissant les critères de recherche, NULL pour aucun,
- res : Le résultat de la recherche,

Cette fonction retourne 0 en cas de réussite, dans le cas contraire elle retourne -1 et positionne « errno » en conséquence.

#### 4.4 - La structure « addrinfo »

```
struct addrinfo {
    int          ai_flags;
    int          ai_family;
    int          ai_socktype;
    int          ai_protocol;
    size_t       ai_addrlen;
    struct sockaddr *ai_addr;
    char         *ai_canonname;
    struct addrinfo *ai_next;
};
```

Lorsque cette structure doit être utilisée dans une fonction pour laquelle tous les champs ne sont pas obligatoires, comme pour la fonction précédente `getaddrinfo( )`, il faut les initialiser à 0 ou NULL.

La définition des champs suivants est obligatoire pour le paramètre « hints » de la fonction « getaddrinfo( ) » :

- ai\_flags : Options supplémentaires, 0 pour aucune,
- ai\_family : AF\_INET, AF\_INET6 ou AF\_UNSPEC,
- ai\_socktype : SOCK\_STREAM, SOCK\_DGRAM ou 0 pour tous,
- ai\_protocol : Protocole, 0 pour laisser celui le plus adapté,

Les autres champs ont la signification suivante :

- ai\_addrlen : Longueur de la structure d'adresse « ai\_addr »,
- ai\_addr : Pointeur sur une structure d'adresse,
- ai\_canonname : Nom officiel de l'hôte si demandé par les options,
- ai\_next : Pointeur sur l'enregistrement suivant,

La variable « ai\_flags » peut prendre une combinaison des valeurs suivantes :

- `AI_NUMERICHOST` : Ne pas effectuer de résolution, « node » doit alors être une adresse IP,
- `AI_PASSIVE` : Retourne les adresses locales si « node » est NULL,
- `AI_NUMERICSERV` : Ne cherche pas à résoudre le service, le champ « service » doit alors contenir un chiffre correspondant au numéro de port, et pas le nom du service.

## 4.5 - Exemple de résolution de nom

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>
#include <arpa/inet.h>
#include <string.h>

int main() {
    int s;
    struct sockaddr_in * ptr_IP4;
    struct sockaddr_in6 * ptr_IP6;
    struct addrinfo hints; struct addrinfo *result, *rp;
    char buffer[1024];

    memset(&hints, 0, sizeof(struct addrinfo));
    hints.ai_family = AF_UNSPEC;
    hints.ai_socktype = SOCK_STREAM;
    s = getaddrinfo("www.google.fr", NULL, &hints, &result);
    if(s==-1) { perror("getaddrinfo...");exit(1); }
```

```
rp=result;
while (rp!=NULL) {
    if(rp->ai_family==AF_INET) {
        ptr_IP4=((struct sockaddr_in *)rp->ai_addr);
        if(inet_ntop(AF_INET,&(ptr_IP4->sin_addr),
                    buffer,1024)==NULL) {
            perror("inet..."); exit(1);
        }
        printf("%s\n",buffer);
    }
    if(rp->ai_family==AF_INET6) {
        ptr_IP6=((struct sockaddr_in6 *)rp->ai_addr);
        if(inet_ntop(AF_INET6,&(ptr_IP6->sin6_addr),
                    buffer,1024)==NULL) {
            perror("inet..."); exit(1);
        }
        printf("%s\n",buffer);
    }
    rp=rp->ai_next;
}
exit(0);
}
```

## 4.6 - Options de socket

Il est possible de consulter et de modifier les options de « socket » créés :

```
#include <sys/types.h>
#include <sys/socket.h>
int getsockopt(int s, int level, int optname,
               void *optval, socklen_t *optlen);
int setsockopt(int s, int level, int optname,
               const void *optval, socklen_t optlen);
```

Ces fonction prend en paramètres :

- s : Le descripteur de socket,
- level : Le niveau, SOL\_SOCKET ou le numéro du protocole (IPPROTO\_IP, ...),
- optname : Le nom de l'option,
- optval : La valeur de l'option,
- optlen : La taille de la donnée associée à l'option,

Cette fonction retourne 0 en cas de réussite, dans le cas contraire elle retourne -1 et positionne « errno » en conséquence.

## 5 - La communication par datagrammes

### 5.1 - Introduction

Un processus souhaitant communiquer par l'intermédiaire d'une socket du type SOCK\_DGRAM doit réaliser les opérations suivantes :

1. demander la création d'un socket,
2. demander éventuellement l'attachement de cette socket sur un port convenu ou un port quelconque selon qu'il joue un rôle de serveur ou de client,
3. construire l'adresse de son interlocuteur en mémoire : tout client désirant s'adresser à un serveur doit en connaître l'adresse,
4. procéder à des émissions et des réceptions de messages.

## 5.2 - L'envoi d'un message

```
int sendto(
    int descripteur, /*renvoyé par la primitive socket */
    void *message,  /* message à envoyer */
    int longueur,   /* longueur de ce message */
    int option,     /* 0 */
    struct sockaddr *ptr_adresse, /* adresse du dest. */
    int longueur_adresse /* longueur de cette adresse */
) ;
```

Cette fonction retourne le nombre de caractères effectivement envoyés en cas de réussite, dans le cas contraire elle retourne -1 et positionne « errno » en conséquence.

**Remarque** : si l'attachement n'est pas effectué avant le premier envoi utilisant cette socket (client) attachement est effectué automatiquement sur un port quelconque de la machine locale.

### 5.3 - La réception d'un message

```
int recvfrom(
    int descripteur, /*renvoyé par la primitive socket */
    void *message,  /*adresse du buffer de réception */
    int longueur,   /*taille du buffer de réception */
    int option,     /*0 */
    struct sockaddr *ptr_adresse, /*adresse de l'émetteur */
    int *ptr_longueur_adresse /*pointeur sur la longueur de
cette adresse */
) ;
```

L'adresse de l'émetteur du message sera récupérée à l'adresse « ptr\_adresse » et la taille de cette adresse à l'adresse « ptr\_longueur\_adresse » au retour de la primitive « recvfrom( ) » à condition d'avoir initialisé « ptr\_longueur\_adresse » avant l'appel à la primitive.

Cette fonction retourne le nombre de caractères effectivement reçus en cas de réussite, dans le cas contraire elle retourne -1 et positionne « errno » en conséquence.

## 5.4 - Mode pseudo-connecté

Il est également possible de réaliser une pseudo-connection en UDP, c'est à dire la possibilité d'utiliser les primitives classiques « read( ) » et « write( ) ». En fait la connexion n'est pas vraiment réalisée mais l'adresse de destination est mémorisée pour ne pas avoir à le préciser à chaque envois/réception de message. Ceci se fait grâce à la primitive « connect( ) ».

## 5.5 - Exemple de « Serveur echo UDP »

```
/* Entetes identiques au programme précédent */  
  
int main() {  
    int desc,taille,longueur;  
    char chaine[1024];  
    struct sockaddr_in adr1,adr2;  
  
    if((desc=socket(AF_INET,SOCK_DGRAM,0)) == -1) {  
        perror("socket ...");exit(1);  
    }  
  
    adr1.sin_family=AF_INET;  
    adr1.sin_addr.s_addr=htonl(INADDR_ANY);  
    adr1.sin_port=htons(2013);  
  
    longueur=sizeof(struct sockaddr_in);  
    if(bind(desc,(struct sockaddr *)&adr1,longueur)==-1) {  
        perror("bind ...");exit(1);  
    }  
}
```

```
do {
    /* Attente d'un paquet */
    taille=recvfrom(desc, chaine, 1024 , 0,
                    (struct sockaddr *)&adr2,&longueur);

    /* Affichage du paquet */
    printf("Packet reçu : %s\n",chaine);

    /* Rémission du paquet */
    sendto(desc,chaine,taille,0,
            (struct sockaddr *)&adr2,longueur);

    /* Tant que la chaine reçue n'est pas vide */
} while(taille>2);

exit(0);
}
```

## 5.6 - Exemple de « Client echo UDP »

```
int main() {
    int desc,taille,longueur=sizeof(struct sockaddr_in);
    char chaine[1024],chaine2[1024];
    struct sockaddr_in adr1,adr2,adr3;

    /* Phase 1 */
    if((desc=socket(AF_INET,SOCK_DGRAM,0)) == -1) {
        perror("socket ...");
        exit(1);
    }

    /* Phase 2 */
    adr1.sin_family=AF_INET;
    adr1.sin_addr.s_addr=htonl(INADDR_ANY);
    adr1.sin_port=htons(0);
    if(bind(desc,(struct sockaddr *)&adr1,longueur)==-1) {
        perror("bind ...");
        exit(1);
    }
}
```

```
/* Phase 3 */
adr2.sin_family=AF_INET;
adr2.sin_addr.s_addr=htonl(INADDR_LOOPBACK);
adr2.sin_port=htons(2013);

/* Phase 4 */
do {
    printf("Message à envoyer : ");
    fgets(chaine,1024,stdin);
    sendto(desc,chaine,strlen(chaine)+1,0,
           (struct sockaddr *)&adr2,longueur);
    taille=recvfrom(desc,chaine2,1024,0,
                    (struct sockaddr *)&adr3,&longueur);
    printf("Message reçu : %s\n",chaine2);

} while(taille>2);

exit(0);
}
```

## 5.7 - Mode « broadcast »

Le broadcast permet de diffuser un paquet UDP sur plusieurs adresses de destination, qui est limité en général au segment local.

Le client doit modifier les caractéristiques de son socket, et en particulier signaler qu'il active le mode « `SO_BROADCAST` » à l'aide de la primitive `setsockopt()`, et utiliser l'adresse de diffusion `INADDR_BROADCAST`.

Il est important également de vérifier qu'une règle de filtrage de paquets ne bloque pas l'émission et/ou la réception de ce type de communication.

## 5.8 - Exemple de « Client echo UDP broadcast »

```
int main() {
    int desc,taille,longueur=sizeof(struct sockaddr_in);
    char chaine[1024],chaine2[1024];
    struct sockaddr_in adr1,adr2,adr3; int bcast=1;

    /* Phase 1 */
    if((desc=socket(AF_INET,SOCK_DGRAM,0)) == -1) {
        perror("socket ...");
        exit(1);
    }

    /* Phase 3 */
    adr2.sin_family=AF_INET;
    adr2.sin_addr.s_addr=htonl(INADDR_BROADCAST);
    adr2.sin_port=htons(2013);
    if(setsockopt(desc, SOL_SOCKET,
                 SO_BROADCAST, &bcast, sizeof(bcast)) < 0){
        perror("setsockopt");
    }
}
```

```
do {
    printf("Message à envoyer : ");
    fgets(chaine,1024,stdin);

    if(sendto(desc,chaine,strlen(chaine)+1,0,
              (struct sockaddr *)&adr2,longueur)==-1) {
        perror("sendto...");exit(1);
    }

    taille=recvfrom(desc,chaine2,1024,0,
                    (struct sockaddr *)&adr3,&longueur);
    printf("Message reçu : %s\n",chaine2);

} while(taille>2);

exit(0);
}
```

## 5.9 - Le mode « multicast »

Le multicast est une technique permettant à un l'émetteur d'envoyer un paquet à plusieurs destinataires, sans « polluer » le reste du réseau comme avec le broadcast. Pour cela, les programmes qui souhaitent recevoir le flux multicast doivent s'abonner à la classe d'adresse IP multicast associée.

Sous Linux, il est nécessaire d'introduire une nouvelle structure :

```
struct ip_mreq {
    /* Adresse IP du groupe multicast */
    struct in_addr imr_multiaddr;
    /* Adresse IP locale de l'interface concernée */
    struct in_addr imr_interface;
};
```

Pour réaliser un programme permettant l'écoute d'un flux multicast, il est nécessaire de spécifier une structure de ce type comme valeur de l'option « `IP_ADD_MEMBERSHIP` », du paramètre de socket de niveau IP (`IPPROTO_IP`).

Cette étape est réalisée en utilisant un appel à la fonction `setsockopt()`, après avoir créé un socket classique attaché localement.

Le fonctionnement client/serveur multicast est inversé par rapport au fonctionnement classique :

- Le serveur va envoyer des données sans attendre de requêtes,
- Le client va se mettre en écoute,

Le serveur multicast se réalise tout simplement en s'attachant localement à l'adresse IP multicast souhaitée.

## 5.10 - Exemple de « Client multicast »

```
int main() {
    int desc,taille,longueur=sizeof(struct sockaddr_in);
    char chaine[1024];
    struct sockaddr_in adr1,adr2;
    struct ip_mreq mreq;

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

    adr1.sin_family=AF_INET;
    adr1.sin_addr.s_addr=htonl(INADDR_ANY);
    adr1.sin_port=htons(2013);

    if(bind(desc,(struct sockaddr *)&adr1,longueur)==-1) {
        perror("bind ...");
        exit(1);
    }
}
```

```
/* Joindre le groupe Multicast 224,0,0,10 */
inet_pton(AF_INET,"224.0.0.10",&mreq.imr_multiaddr.s_addr)
;
mreq.imr_interface.s_addr = htonl(INADDR_ANY);
if (setsockopt(desc, IPPROTO_IP, IP_ADD_MEMBERSHIP,
    &mreq, sizeof(mreq)) == -1) {
    perror("setsockopt...");
    exit(1);
}

do {
    taille=recvfrom(desc,chaîne,1024,0,
        (struct sockaddr *)&adr2,&longueur);
    printf("Packet reçu : %s\n",chaîne);
} while(taille>2);
exit(0);
}
```

## 5.11 - Exemple de « Serveur multicast »

```
int main() {
    int desc,taille,longueur=sizeof(struct sockaddr_in);
    char chaine[1024]; struct sockaddr_in adr1,adr2;

    if((desc=socket(AF_INET,SOCK_DGRAM,0)) == -1) {
        perror("socket ...");exit(1);
    }
    adr2.sin_family=AF_INET;
    inet_pton(AF_INET,"224.0.0.10",&adr2.sin_addr.s_addr);
    adr2.sin_port=htons(2013);
    do {
        printf("Message à envoyer : ");
        fgets(chaine,1024,stdin);
        if(sendto(desc,chaine,strlen(chaine)+1,0,
                (struct sockaddr *)&adr2,longueur)==-1) {
            perror("sendto...");exit(1);
        }
    } while(taille>2);
    exit(0);
}
```

## 6 - La communication en mode connecté

C'est le mode de communication associé aux sockets de type SOCK\_STREAM, la dissymétrie entre les deux entités est clairement marquée.

### 6.1 - Schéma général d'un serveur TCP

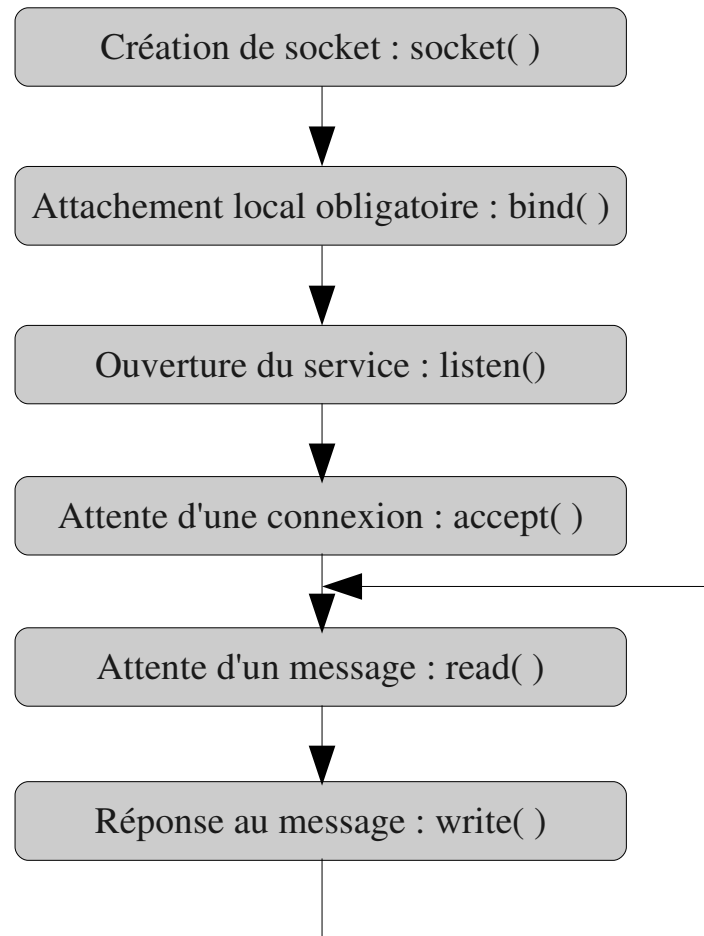
Un serveur a un rôle passif dans l'établissement de la connexion : après avoir avisé (par un appel à la primitive **listen**) le système auquel il appartient qu'il est prêt à accepter les demandes de connexion des clients, le serveur se met en attente de demande de connexion.

Pour cela il dispose d'un **socket d'écoute** attachée au port correspondant au service et donc supposé connu des clients. Lorsqu'une demande de connexion arrive à l'entité TCP du système où s'exécute le serveur, le système crée un nouveau socket dédiée à cette nouvelle connexion et que nous appellerons **socket de service**.

C'est par ce moyen qu'il est possible de multiplexer sur le même socket plusieurs connexions.

Le processus serveur prend connaissance de l'existence d'une nouvelle connexion par un appel à la primitive **accept** : au retour de cet appel, le processus reçoit un descripteur lui permettant d'accéder à cet socket de service.

Les connexions acceptées au niveau TCP mais non encore prises en compte par le processus sont dites **pendantes**. Une fois prise en compte par le serveur (par un appel à la primitive **accept**), une connexion devient effective et est enlevée de la liste des connexions pendantes.

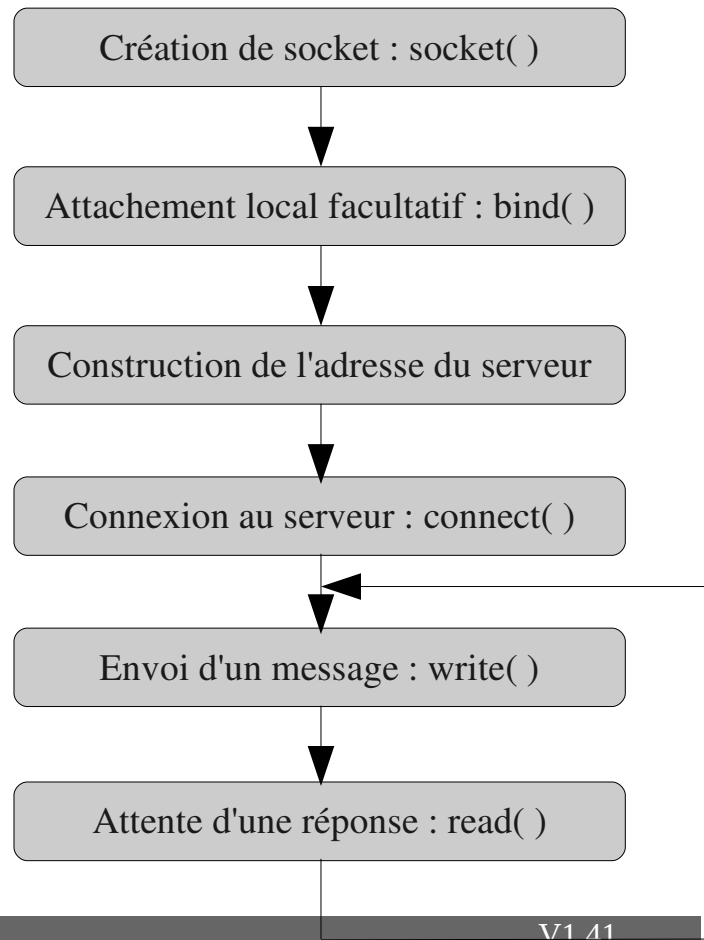


## 6.2 - Schéma général d'un client TCP

Un client est l'entité active dans le processus d'établissement d'une connexion avec le serveur : c'est lui qui prend l'initiative.

Le client commence par créer la socket, l'attache éventuellement à un numéro de port quelconque puis tente de connecter cette socket à celle du serveur.

Le client et le serveur peuvent ensuite dialoguer.



### 6.3 - La primitive listen

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

Cette fonction prend en paramètres :

- sockfd : Le descripteur de socket existant,
- backlog : La taille de la file d'attente (connexions pendantes),

Cette fonction retourne 0 en cas de réussite, dans le cas contraire elle retourne -1 et positionne « errno » en conséquence.

## 6.4 - La primitive accept

```
#include <sys/types.h>
#include <sys/socket.h>
int accept(int sockfd, struct sockaddr *adresse,
           socklen_t *longueur);
```

Cette fonction **bloquante** prend en paramètres :

- sockfd : Le descripteur de socket existant,
- adresse : Un pointeur sur une structure dans laquelle sera écrite l'adresse de l'émetteur,
- longueur : La longueur de cette structure qui doit être initialisée,

Cette fonction retourne un nouveau descripteur en cas de réussite, dans le cas contraire elle retourne -1 et positionne « errno » en conséquence.

## 6.5 - La primitive connect

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

Cette fonction prend en paramètres :

- sockfd : Le descripteur de socket existant,
- serv\_addr : Un pointeur sur une structure contenant l'adresse du serveur,
- addrlen : La longueur de cette structure,

Cette fonction retourne 0 en cas de réussite, dans le cas contraire elle retourne -1 et positionne « errno » en conséquence.

## 6.6 - La réception de données

```
#include <unistd.h>
ssize_t read(int fd, void *buf, size_t count);
```

Cette fonction **bloquante** prend en paramètres :

- fd : Le descripteur de socket existant,
- buf : Un pointeur sur une zone de mémoire dans laquelle sera enregistrée les données,
- count : La taille de cette zone mémoire,

Cette fonction retourne le nombre d'octets lus en cas de réussite, dans le cas contraire elle retourne -1 et positionne « errno » en conséquence.

## 6.7 - Envoi de données

```
#include <unistd.h>
ssize_t write(int fd, const void *buf, size_t count);
```

Cette fonction prend en paramètres :

- fd : Le descripteur de socket existant,
- buf : Un pointeur sur une zone de mémoire contenant les données,
- count : La taille de cette donnée,

Cette fonction retourne le nombre de caractères effectivement envoyés en cas de réussite, dans le cas contraire elle retourne -1 et positionne « errno » en conséquence.

## 6.8 - Exemple de serveur « echo TCP »

```
int main() {
    int desc,desc2,taille;
    int longueur=sizeof(struct sockaddr_in);
    char chaine[1024];
    struct sockaddr_in adr1,adr2;

    if((desc=socket(AF_INET,SOCK_STREAM,0)) == -1) {
        perror("socket ...");exit(1);
    }
    adr1.sin_family=AF_INET;
    adr1.sin_addr.s_addr=htonl(INADDR_ANY);
    adr1.sin_port=htons(2013);

    if(bind(desc,(struct sockaddr *)&adr1,longueur)==-1) {
        perror("bind ...");exit(1);
    }

    if(listen(desc,10)==-1) {
        perror("listen ...");exit(1);
    }
}
```

```
do {
    if((desc2=accept(desc,
                    (struct sockaddr *)&adr2,&longueur))== -1) {
        perror("bind ...");exit(1);
    }

    /* Normalement création d'une tache dédiée */
    /* Pour traiter la connexion */
    taille=read(desc2,chaine,1024);
    printf("Packet reçu : %s\n",chaine);
    write(desc2,chaine,taille);
    close(desc2);
    /* Fin de la connexion */

} while(taille>2);
exit(0);
}
```

## 6.9 - Exemple de client « echo TCP »

```
int main() {
    int desc,desc2,taille,longueur=sizeof(struct sockaddr_in);
    char chaine[1024],chaine2[1024];
    struct sockaddr_in adr1,adr2,adr3;

    /* Phase 1 */
    if((desc=socket(AF_INET,SOCK_STREAM,0)) == -1) {
        perror("socket ...");
        exit(1);
    }
    adr1.sin_family=AF_INET;
    adr1.sin_addr.s_addr=htonl(INADDR_ANY);
    adr1.sin_port=htons(0);

    /* Phase 2 */
    if(bind(desc,(struct sockaddr *)&adr1,longueur)==-1) {
        perror("bind ...");
        exit(1);
    }
}
```

```
/* Phase 3 */
adr2.sin_family=AF_INET;
adr2.sin_addr.s_addr=htonl(INADDR_LOOPBACK);
adr2.sin_port=htons(2013);

printf("Message à envoyer : ");
fgets(chaine,1024,stdin);
if(connect(desc,(struct sockaddr *)&adr2,longueur)==-1) {
    perror("connect ...");exit(1);
}
write(desc,chaine,strlen(chaine)+1);
taille=read(desc,chaine2,1024);
printf("Message reçu : %s\n",chaine2);
close(desc);
exit(0);
}
```

## 7 - Compatibilité avec Microsoft Windows

### 7.1 - Introduction

Il est possible d'utiliser les mêmes programmes sous Unix et sous Windows. En effet, les bibliothèques réseaux sont identiques dans les deux cas à l'exception des points suivants :

- Le programme Windows doit inclure le fichier « winsock2.h »,
- Le programme Windows doit initialiser la bibliothèque WinSock,
- Le programme Windows doit inclure la bibliothèque libwsck32,

Les différents programmes ont été portés avec succès sous Windows avec les deux compilateurs suivants : DevCPP et Visual C++.

## 7.2 - Fichiers include

Une directive « #if » est nécessaire afin de tester l'architecture :

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <string.h>

#if      defined(WIN32)      ||      defined(__MINGW32__)      ||
defined(__WIN32__)
    #include <winsock2.h>
#else
    #include <sys/socket.h>
    #include <netinet/in.h>
    #include <netdb.h>
    #include <arpa/inet.h>
#endif

...
```

### 7.3 - Procédure d'initialisation

Une procédure d'initialisation spécifique est également nécessaire sous Windows :

```
int main(int argc, char *argv[])
{
    /* Déclaration des variables */
    ...

    #if    defined(WIN32)    ||    defined(__MINGW32__)    ||
defined(__WIN32__)
    // Initialisation de winsock (sous windows)
    WSADATA wsa;
    if(WSAStartup(MAKEWORD(1,1),&wsa)!= 0) {
        printf("main:%d : probleme de winsock !\n",__LINE__);
        exit(0);
    }
#endif

    ...
}
```

De même, à la fin du programme, il est nécessaire d'inclure le code suivant :

```
...  
  
#if      defined(WIN32)      ||      defined(__MINGW32__)      ||  
defined(__WIN32__)  
    WSACleanup();  
#endif  
  
...
```

## 7.4 - Librairies

Lors de la phase d'édition de lien, il est nécessaire d'utiliser la librairie « libwsck32.dll », c'est à dire l'option suivante d'un compilateur (avec DevCPP) :

```
-lwsck32
```

ou avec Visual C++, utiliser la librairie « **wsock32.lib** » dans les propriétés du projet (édition de liens).

## 8 - Les obligations liées aux serveurs

### 8.1 - Les zombies

Dans le cas d'un serveur UDP qui est habituellement « wait », le processus serveur traite directement les requêtes des clients. Dans le cas d'un serveur TCP qui est habituellement « no wait », le serveur va se cloner et faire traiter les requêtes des clients par ses fils.

Attention, dès qu'un serveur se clone, il faut penser à l'élimination des zombies avec l'une des primitives sigaction ou signal.

## 8.2 - Les descripteurs de fichiers

Il faut penser à fermer tous les descripteurs inutiles, en effet on évite en général de garder tous les descripteurs hérités ouverts.

```
#include <sys/param.h>
for (i=0; i<NOFILE ;i++) close(i);
```

## 8.3 - Le directory

Il vaut mieux se placer à la racine du système par l'appel à :

```
chdir ("/");
```

qui permet les démontages éventuels du système de fichiers sans arrêter le serveur.

## 8.4 - Le détachement ou la naissance d'un démon

Une caractéristique essentielle d'un serveur est qu'il est détaché de tout terminal : ceci est réalisé dans POSIX avec la primitive **setsid** qui crée une nouvelle session dont le processus sera le leader. Pour éviter des interférences avec le contrôle de jobs qui peut être actif avec certains shells, on réalise un premier appel fork : le processus initial se termine immédiatement d'où réveil du shell courant tandis que le processus fils qui est adopté par le processus init, réalise un appel à setsid puis exécute le code du serveur proprement dit.

```
main(...) {  
    ...  
    if( fork() != 0)  
        exit(0) ;  
    setsid() ;  
    ...  
}
```

## 8.5 - Le contrôle

Un démon peut être programmé pour lire un fichier de configuration lors de son lancement mais il peut être aussi important de l'obliger à relire ce fichier, sans l'arrêter, surtout quant ce serveur a un rôle fondamental dans la bonne marche du système tout entier.

Ceci est réalisé par l'envoi du signal SIGHUP et son déroutement dans le code du serveur par le biais de la primitive sigaction.

En règle générale, le fichier de configuration se termine par l'extension *".conf"*.

## 8.6 - Problème de ré-attachement de socket TCP

Il peut arriver que lorsqu'un serveur TCP décide de redémarrer, le socket ne soit pas disponible, ceci parce qu'une connexion existe encore et/ou que le système ne libère pas immédiatement ce port. Ceci peut être résolu en positionnant la paramètre `SO_REUSEADDR` du socket à l'aide de la primitive « `setsockopt()` » :

```
int valeur=1;
...
if(setsockopt(desc, SOL_SOCKET
    , SO_REUSEADDR, &valeur, sizeof(valeur)) < 0){
    perror("setsockopt...");
}
...
```

## 9 - Les évènements d'un démon

Un serveur qui joue un rôle important a souvent été programmé pour signaler à son administrateur les erreurs rencontrées et les actions effectuées au cours d'une session. Ceci peut être réalisé de plusieurs manières.

### 9.1 - Ecrire sur la console

Il suffit d'ouvrir le fichier `/dev/console` et d'envoyer les messages de trace sur ce fichier en se servant éventuellement de la primitive `dup` pour rediriger `stdout` et `stderr`.  $\Rightarrow$  les messages ne sont pas enregistrés

## 9.2 - Ecrire dans un fichier spécifique au serveur

```
FILE *f ;
main(...) {
    f=fopen("trace_serveur","w") ;
    ...
    fprintf(f, "format données",données) ;
    ...
}
```

Exemple :

Fichiers « access log » d'Apache,

### 9.3 - Envoyer les messages de trace au démon syslogd

Le démon **syslogd** est chargé de gérer l'historique du système en envoyant des messages dans des fichiers, à des groupes d'utilisateurs ou au démon syslogd d'une autre machine. Le PID du démon syslogd local est contenu dans le fichier **/etc/syslogd.pid** tandis que le fichier **/etc/syslogd.conf** permet de configurer ce démon.

#### Ouvrir une connexion avec syslog

```
#include <syslog.h>
void openlog(char *ident, int option, int facility)
```

Ouvre une connexion a syslog pour le programme. La chaîne pointée par ident est ajoutée a chaque message, par défaut (c'est a dire si aucun appel a la fonction openlog n'a été effectué) elle est positionné sur le nom du programme. L'utilisation d'openlog est optionnelle, elle sera automatiquement appelée par syslog() si nécessaire. Les valeurs pour option et facility sont données plus loin.

## Option

L'argument option pour `openlog()` est un OU les différentes options suivantes :

- LOG\_CONS : Écrit directement sur la console système s'il y a une erreur pendant l'envoi d'un message au syslog,
- LOG\_NDELAY : Ouvre la connexion immédiatement (Normalement, la connexion est ouverte quand le premier message est envoyé a syslog),
- LOG\_PERROR : Envoie également le message sur stderr,
- LOG\_PID : Inclus le PID avec chaque message,

## Facility et Level

Ces options permettent de préciser le niveau d'importance du message envoyé à syslog, consulter les pages de man pour obtenir plus d'information.

### Envoyer une information à syslog

```
#include <syslog.h>
void syslog(int priority, char *format, ...)
```

Génère un message de log, qui sera distribué par syslogd(8), priority est une combinaison de facility et level données ci-dessous. Le dernier argument est une chaîne formatée comme printf exceptée que la chaîne %m sera remplacée par la chaîne du message d'erreur correspondant à la valeur actuelle de errno.

### Fermer la connexion avec syslog

```
#include <syslog.h>
void closelog(void)
```

Ferme le descripteur utilisé pour dialoguer avec syslog, l'utilisation de closelog est optionnelle.

Pour que le nouveau démon utilise alors les services proposés par syslogd, on inclura dans son code des lignes du style :

```
...  
openlog("Mon prog ", LOG_PID, LOG_DAEMON);  
syslog(LOG_EMERG, "Mon message urgent...");  
syslog(LOG_WARNING, "Mon message...%d", i);  
closelog();  
...
```

### **ATTENTION :**

Sous certaines versions de Linux, l'appel « syslog( ) » n'est pas ré-entrant et ne doit donc pas être interrompu par un signal qui lui même ferait un appel à « syslog( ) ».

Par conséquent, les signaux captés par le programme doivent masqués avant l'appel à « syslog( ) ».

## 10 - Socket au format brut (raw)

La primitive « `socket( )` » permet de créer des sockets au niveau de la couche transport du modèle OSI, c'est à dire utilisant UDP (`SOCK_DGRAM`) ou TCP (`SOCK_STREAM`). Il est néanmoins possible de créer directement un socket au niveau IP, et de former son propre paquet, il faut pour cela utiliser le type `SOCK_RAW`.

Lorsque ce type de socket est créé, le système initialise l'entête IP, ce qui fait qu'il est nécessaire de préciser le protocole qui sera véhiculé par IP, et ceci en le spécifiant dans le troisième paramètre de la primitive `socket( )`.

Le type de protocole est soit le numéro de protocole défini par l'IANA (<http://www.iana.org/assignments/protocol-numbers/>), soit une mnémonique déjà utilisée précédemment (`IPPROTO_IP`, `IPPROTO_ICMP`, `IPPROTO_UDP`, `IPPROTO_TCP`, ...) comme définie dans le fichier d'entête « `/usr/include/netinet/in.h` ».

Il est intéressant de noter que si pour l'émission d'un paquet, le système construit l'entête IP, et laisse à la charge du développeur de programmer la couche de niveau supérieur, il n'en est pas de même pour la réception. En effet, le système fournit alors la paquet IP dans son intégralité, c'est à dire avec son entête.

Il est en général nécessaire d'avoir des droits privilégiés pour pouvoir ouvrir ce type de socket, c'est à dire être « root ».

Si le développeur souhaite descendre encore d'un niveau, et pouvoir programmer les paquets au niveau 2 du modèle OSI, alors il doit utiliser la famille de socket `PF_PACKET`, et non plus `AF_INET` ou `AF_INET6`.

## 10.1 - Exemple de programme ping

```
/* entete ip RFC 791 - 20 octets */
/* icmp RFC 792 */

int main(int argn, char * argv[]) {
    struct sockaddr_in dest,source;
    int i,recu,s;
    u_char packetIN[128];
    u_char packetOUT[64];
    u_char chksum;
    int sourcelong=sizeof(source);
    int packetOUTlong=sizeof(packetOUT);
    int packetINlong=sizeof(packetIN);

    if(argn!=2) {
        printf("Syntaxe : %s <IP DESTINATION>\n",argv[0]);
        exit(1);
    }

    /* on remplit cette structure... */
    dest.sin_family = AF_INET;
    inet_pton(AF_INET,argv[1],&dest.sin_addr.s_addr);
```

```
/* On ouvre le socket */
if ((s = socket(AF_INET, SOCK_RAW, IPPROTO_ICMP)) < 0) {
    perror("socket...");
    exit(1);
}

printf ("Envoi ICMP request sur %s\n",argv[1]);

/* nettoyage du packet envoye */
memset(packetOUT, 0, packetOUTlong);

/* definition du paquet ICMP */
packetOUT[0]=(u_char)8;      /* type = 8 demande d echo*/
packetOUT[1]=(u_char)0;     /* code = 0 */

/* calcul checksum : */
/* ~(8 + 0 + 0 ...) = 0xf7ff */
packetOUT[2]=(u_char)0xf7;  /* partie haute */
packetOUT[3]=(u_char)0xff;  /* partie basse */
```

```
/* Boucle d'attente infinie */
for(;;) {
    sleep(1);
    /* envoi du paquet */
    i = sendto(s, (char *)packetOUT, packetOUTlong, 0,
              (struct sockaddr *) &dest, sizeof(struct sockaddr));
    recu = recvfrom(s, (char *)packetIN, packetINlong, 0,
                   (struct sockaddr *)&source, (socklen_t *)&source);
    if (recu == -1) {
        perror("recvfrom...");
        exit(1);
    } else {
        for (i=0;i<28;i++) {
            printf("%d ",packetIN[i]);
            if((i%4)==3) printf("\n");
        }
    }
    /* On a reçu quelque chose mais est-ce un packet */
    /* ICMP echo reply de la bonne machine ?? */
}
}
```