Chapitre 6

API des sockets : l'interface BSD

Commençons par la fin, à savoir par l'interface de programmation (ou API). Il devrait presque s'agir d'un rappel puisqu'il vaut mieux commencer par programmer des applications réseau (très simples) avant d'aborder l'implémentation elle-même. Cependant nous ne supposons aucune connaissance préalable et donc ce chapitre et le suivant peuvent servir d'introduction à la programmation réseau.

Deux API réseau principaux ont été utilisées : les sockets de BSD et le **TLI** (pour *Transport Layer Interface*) du System V. On pourra consulter le chapitre 7 de la première édition de [STE–90] pour TLI, maintenant abandonné au profit des sockets. Nous allons nous intéresser dans ce chapitre et le suivant à l'API des sockets.

Notre but est d'étudier la mise en place des sockets non connectées (qui correspondent à UDP) mais, pour amener les notions petit à petit, nous commençons par les paires de socket locales, les sockets pour communication connectée (ce qui correspond à TCP) et la transmission de données urgentes.

6.1 Modèles de sockets

6.1.1 Notion de socket

Nous avons vu que les versions 4.1BSD et 4.2BSD d'UNIX incluaient une implémentation de TCP/IP. L'interface utilisateur de celle-ci est constituée de l'API des sockets. En UNIX, tout est fichier. Les sockets sont donc implémentés comme un type particulier de fichiers.

Une **socket**[1] est une généralisation des tubes de communication entre processus (*pipe* en anglais), sauf que la communication s'effectue maintenant entre ordinateurs distants (ou, plus exactement, processus distants et non plus nécessairement situés sur le même ordinateur). C'est évidemment surtout au niveau de l'implémentation qu'il y a quelque chose à faire. Le programmeur, quant à lui, voit une socket comme un fichier. Bien entendu sa création exige une image de communication, puisqu'il doit indiquer l'adresse réseau et le numéro de port d'application.

L'API est constituée comme d'habitude par une bibliothèque (et un fichier en-tête) du langage C, formée d'appels système.

Dans ce premier chapitre consacré à l'API des sockets, nous allons étudier l'essentiel pour concevoir un programme qui fonctionne. C'est ce que l'on peut appeler le niveau fondamental. Il correspond à l'interface BSD d'origine.

Pour établir une communication téléphonique, on appelle son correspondant en faisant son numéro, on attend qu'il réponde puis on peut parler. Schématiquement, on a donc deux postes téléphoniques, le poste local et le poste distant (en anglais *remote telephone*), et une ligne téléphonique. Pour une communication par sockets, on a l'analogue :

- une **socket locale**, qui correspond au poste téléphonique local ;
- une **socket distante** (*remote socket* en anglais), qui correspond au téléphone distant ;
- une ligne de communication.

Bien entendu on utilisera une adresse réseau (constituée d'une adresse IP et d'un numéro de port dans le cas de TCP/IP) au lieu d'un numéro de téléphone.

6.1.2 Types de communication

Il ya deux types de communication avec les sockets :

- Dans une **communication connectée** (*connection-oriented communication*), on commence par établir un canal entre les deux points. Il s'agit de TCP dans le cas de IP.

- Dans les autres cas on parle de **communication non connectée** (*connectionless-oriented communication* en anglais). Dans ce cas on doit donner l'adresse de destination lors de cahque envoi et l'adresse source si on veut une réponse. Il s'agit de UDP dans le cas de IP.

1. Le mot *socket* se traduit par *prise* (de courant) en français mais le mot anglais est devenu un terme consacré.

6.2 Paire de sockets locales

Pour débuter avec quelque chose de simple, nous allons créer deux sockets situées sur le même ordinateur, communiquant entre elles. Dans ce cas, nous n'aurons pas besoin d'utiliser les adresses réseau (il n'y a pas besoin de numéro de téléphone dans le cas de deux postes internes). Il s'agit en fait d'une façon sophistiquée de faire appel aux tubes de communication, mais c'est également une très bonne propédeutique à l'étude des sockets dans toute leur généralité.

6.2.1 Création

Syntaxe.- La création d'une paire de sockets locales s'effectue grâce à l'appel système :

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

int socketpair(int domain, int type, int protocol, int sv[2]);
```

Sémantique.- Le domaine indique la famille de protocoles qui sera utilisée lors de la communication (locale, IP, IPX de Novell, Appletalk ou autre). Il correspond au protocole de la couche réseau du modèle OSI mais rappelons que l'interface des sockets a été conçue avant la définition de celui-ci.

Le type de communication indique si la communication supporte les erreurs (cas d'une émission de télévision ou de radio transmise par réseau), si elle est unidirectionnelle ou bidirectionnelle et ainsi de suite. Il correspond au protocole de la couche transport du modèle OSI.

Le protocole est un paramètre supplémentaire qui peut être nécessaire dans certains cas mais la plupart du temps il prend la valeur nulle. Nous indiquerons les valeurs possibles lorsque cela sera nécessaire. Il n'a pas d'équivalent dans le modèle OSI.

L'appel système renvoie deux **numéros de socket** à travers le tableau `sv[]`, qui seront utilisés dans les appels systèmes ultérieurs pour faire référence à ces sockets. Il s'agit de numéros de fichiers.

Domaines.- Donnons quelques familles de protocoles implémentées sous Linux dans le tableau suivant :

Constante	Signification	Documentation
PF_UNIX	communication locale	unix(7)
PF_LOCAL	synonnyme de PF_UNIX	
PF_INET	protocole Internet IPv4	ip(7)
PF_INET6	protocole Internet IPv6	
PF_IPX	protocole Novell IPX	
PF_NETLINK	interface noyau/utilisateur de périphérique	netlink(7)
PF_X25	protocole ITU-T X.25 / ISO-8208	x25(7)
PF_AX25	protocole radio-amateur AX.25	
PF_ATMPVC	accès à ATM de façon brute	
PF_APPLETALK	protocole du réseau Apple d'origine	ddp(7)
PF_PACKET	interface bas niveau aux paquets (abandonné)	packet(7)

Rappelons que `unix(7)`, par exemple, indique que l'on obtiendra une documentation du manuel grâce à la commande :

```
man 7 unix
```

Ces constantes symboliques commencent toutes par `PF` pour l'anglais *Protocol Family*. On les retrouve aussi avec l'en-tête `AF` pour *Address Family*, par exemple `AF_UNIX`.

Nous ne nous intéresserons dans ce livre qu'aux communications locales (il s'agit en fait des tubes nommés) et aux protocoles Internet version 4.

Pour la création d'une paire de sockets, on ne peut utiliser que la famille de protocoles `PF_UNIX` (et ses trois variantes `AF_UNIX`, `PF_LOCAL` et `AF_LOCAL`).

Types de communication.- Les types acceptés sont représentés par l'une des constantes symboliques suivantes :

- `SOCK_STREAM` : flux d'octets, avec connexion, bidirectionnel et sûr (ou presque). Il s'agit de la couche de transport TCP dans le cas de IP.
- `SOCK_DGRAM` : support de datagrammes (sans connexion, sans être sûr que le message soit reçu, données d'une longueur maximum fixée). Il s'agit de la couche de transport UDP dans le cas de IP.
- `SOCK_SEQPACKET` : comme pour `SOCK_STREAM` sauf qu'ici le destinataire doit lire un paquet en entier par appel système de lecture.
- `SOCK_RAW` : accès à bas niveau.
- `SOCK_RDM` : comme pour `SOCK_DGRAM` mais sans que l'ordre d'arrivée des paquets ne soit garantie.
- `SOCK_PACKET` : abandonné. Ressemblait à `SOCK_RAW`.

Nous nous intéresserons évidemment plus particulièrement à `SOCK_STREAM` et à `SOCK_DGRAM`.

Erreurs.- La valeur renvoyée par la fonction est le numéro de fichier lorsque tout s'est bien déroulé. La valeur -1 indique une erreur ; la valeur prise par `errno` dans ce cas est l'une des suivantes :

- `EINVAL` : le domaine n'est pas valide ;
- `EPROTONOSUPPORT` : le type n'est pas cohérent avec le protocole ou le domaine ;
- `EACCES` : on n'a pas l'autorisation de créer une socket du type demandé ;
- `EMFILE`, `ENFILE`, `ENOMEM` de signification habituelle.

Exemple.- L'exemple suivant nous permet de créer une paire de sockets locales, sans utilisation particulière pour l'instant :

```
/* paire.c */

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

void main(void)
{
  int s[2];

  /* Creation d'une paire de sockets locales */

  socketpair(AF_LOCAL, SOCK_STREAM, 0, s);

  printf("s[0] = %d\n", s[0]);
  printf("s[1] = %d\n", s[1]);
}
```

L'exécution de ce programme donne :

```
# ./a.out
s[0] = 3
s[1] = 4
```

ce qui est prévisible puisque les numéros de fichier 0, 1 et 2 sont réservés pour les fichiers standard.

6.2.2 Lecture et écriture sur des sockets

Syntaxe.- On peut lire et écrire sur une socket comme sur n'importe quel fichier grâce aux appels systèmes suivants :

```
#include <unistd.h>

ssize_t read(int fd, void *buf, size_t count);
ssize_t write(int fd, const void *buf, size_t count);
```

La valeur de retour de `read()` est le nombre de caractères effectivement reçus.

Il n'y a pas d'accès direct avec les sockets, mais uniquement des accès séquentiels, aussi ne peut-on pas utiliser la fonction `lseek()`.

Exemple.- Dans l'exemple suivant on envoie un message depuis l'une des sockets, celui-ci étant récupéré par l'autre :

```
/* message.c */

#include <stdio.h>
#include <unistd.h>
#include <sys/socket.h>
#include <string.h>

void main(void)
{
  int s[2];
  char buf1[80], buf2[80];
  int n;

  /* Creation d'une paire de sockets locales */

  socketpair(AF_LOCAL, SOCK_STREAM, 0, s);

  printf("s[0] = %d\n", s[0]);
  printf("s[1] = %d\n", s[1]);

  /* Envoi d'un message */

  strcpy(buf1, "Bonjour");
  printf("Message envoye : %s\n", buf1);
  write(s[0], buf1, 7);

  /* Reception d'un message */

  n = read(s[1], buf2, 80);
  buf2[n] = '\0';              /* Terminaison de la chaine recue */
  printf("Message recu : %s\n", buf2);
}
```

6.2.3 Fermeture des sockets

Le programme de l'exemple précédent présente un défaut. Comme tout fichier, une socket doit être fermée lorsqu'on n'en a plus besoin. Ceci permet de libérer le descripteur de fichier associé et autres choses analogues.

Syntaxe.- On peut pour cela utiliser l'appel système classique :

```
#include <unistd.h>

int close(int fd);
```

bien que nous verrons un peu plus loin que ce ne soit pas le plus intéressant.

Exemple.- Un meilleur programme est donc :

```
/* ferme.c */

#include <stdio.h>
#include <unistd.h>
#include <sys/socket.h>
#include <string.h>

void main(void)
{
  int s[2];
  char buf1[80], buf2[80];
  int n;

  /* Creation d'une paire de sockets locales */

  socketpair(AF_LOCAL, SOCK_STREAM, 0, s);

  printf("s[0] = %d\n", s[0]);
  printf("s[1] = %d\n", s[1]);

  /* Envoi d'un message */

  strcpy(buf1, "Bonjour");
  printf("Message envoye : %s\n", buf1);
  write(s[0], buf1, 7);

  /* Reception d'un message */

  n = read(s[1], buf2, 80);
  buf2[n] = '\0';
  printf("Message recu : %s\n", buf2);

  /* Fermeture des sockets */

  close(s[0]);
  close(s[1]);
}
```

Sémantique.- Lorsqu'on ferme une socket, la socket qui se trouve à l'autre bout reçoit un indicateur de fin de fichier pour indiquer qu'il n'y a plus de données à recevoir.

6.3 Socket pour communication connectée

6.3.1 Cycles de vie dans le cas des communications connectées

Dans le cas d'une communication connectée, les étapes du cycle de vie d'un serveur sont les suivantes :

- création de la socket serveur ;
- initialisation du serveur (concrétisée par l'association d'une adresse IP et d'un numéro de port dans le cas de TCP) ;
- attente de client ;
- acceptation de la connexion ;
- communication (écriture, lecture) ;

– fermeture.

Les étapes du cycle de vie d'un client sont les suivantes :

– création de la socket côté client ;
– connexion à la socket distante (serveur) ;
– communication (écriture, lecture) ;
– fermeture.

6.3.2 Création d'une socket

Syntaxe.- La création d'une socket (qu'elle soit client ou serveur) s'effectue grâce à l'appel système :

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

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

dont les paramètres ont été décrits à propos de la création d'une paire de sockets. Le numéro de socket est renvoyé, ou -1 dans le cas d'une erreur.

En cas d'erreur, la variable `errno` prend l'une des valeurs suivantes :

– `EAFNOSUPPORT` : type de famille d'adresses non implémenté ;
– `EINVAL` : type de communication non valide, lorsque `type` n'est pas compris entre 0 et une certaine valeur maximum ;
– `ENFILE` : il n'y a plus de descripteur de socket disponible.

Protocoles.- Dans le cas de la famille de protocoles `PF_INET`, correspondant à IPv4 :

– pour le type de communication `SOCK_STREAM`, le seul protocole existant est `IPPROTO_TCP`, égal à 0 ; il correspond à TCP ;
– pour le type de communication `SOCK_DGRAM`, le seul protocole existant est `IPPROTO_UDP`, égal à 0 ; il correspond à UDP.

Exemple.- Créons une socket pour le protocole TCP :

```
/* client1.c */

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

void main(void)
{
  int s;

  /* Creation d'une socket */

  s = socket(AF_INET, SOCK_STREAM, 0);

  printf("s = %d\n", s);

  /* Fermeture de la socket */

  close(s);
}
```

Ce programme a peu d'intérêt. Nous allons le compléter petit à petit.

6.3.3 Spécification des adresses

Voyons comment on spécifie l'adresse réseau.

6.3.3.1 Adresses génériques

Puisque les sockets BSD furent développées avant que le standard C ANSI ne soit adopté, le type pointeur `(void *)` n'était pas accepté pour faire face aux différents types d'adresses. La solution fut donc de définir une structure d'**adresse générique** :

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

struct sockaddr
  {
    unsigned short int sa_family;
    unsigned char  sa_data[14];
  };
```

en réservant deux octets pour la famille d'adresse et 14 octets au maximum pour l'adresse proprement dite.

6.3.3.2 Adresses locales

Pour les communications locales, on utilise la structure d'**adresse Unix** :

```
#include <sys/un.h>

struct sockaddr_un
{
    sa_family_t sun_family;        /* Famille d'adresses */
    char        sun_path[108];     /* Chemin d'acces     */
};
```

où `sun_path` indique le chemin du fichier local (chaîne de caractères, qui n'a pas besoin de se terminer par le caractère nul puisque, comme nous le verrons, la longueur de l'adresse est également toujours passée en paramètre) et où `sun_family` doit avoir la valeur `AF_LOCAL` ou `AF_UNIX`.

Le type `sa_family_t` est un entier court non signé, ce qui représente deux octets.

Des informations sur les adresses locales se trouvent dans la page de manuel `unix(4)`.

Le service `lpr` d'impression, par exemple, utilise une socket locale pour communiquer avec le fichier de spool.

6.3.3.3 Adresses Internet

Pour les sockets de la famille `AF_INET` reposant sur le protocole IPv4, on utilise une structure précisant le port d'application et l'adresse IP :

```
#include <netinet/in.h>

struct sockaddr_in
{
    sa_family_t    sin_family;   /* Famille d'adresses    */
    uint16_t       sin_port;     /* Port d'application    */
    struct in_addr sin_addr;     /* Adresse Internet      */
    unsigned char  sin_zero[8];  /* octets de remplissage */
};

struct in_addr
{
```

```
    uint32_t       s_addr;       /* Adresse Internet      */
}
```

où `sin` est une abréviation de *Socket INternet* et `netinet` une abréviation de *NET InterNET*. Le membre `sin_family` doit prendre la valeur `AF_INET`.

La description des types d'adresses se trouvent dans la page de manuel `ip(4)`.

6.3.3.4 Autres familles d'adresses

Les autres familles de protocoles utilisent les structures suivantes, que nous n'étudierons pas dans ce livre :

Famille	Structure	Fichier
AF_AX25	sockaddr_ax25	<netax24/ax25.h>
AF_INET6	sockaddr_in6	<netinet/in.h>
AF_IPX	sockaddr_ipx	<netipx/ipx.h>

6.3.3.5 Constantes symboliques d'adresses Internet

Certaines adresses Internet, souvent utilisées, sont disponibles sous forme de constantes symboliques :

- l'**adresse Internet par défaut** (*wild address* en anglais) permet de déterminer l'adresse de votre ordinateur, ou plus exactement l'une de celles-ci lorsqu'il dispose de plusieurs cartes réseau par exemple. Il s'agit de la constante `INADDR_ANY`.
- L'adresse Internet locale correspond au périphérique *loopback* d'adresse IP 127.0.0.1. Elle est repérée par la constante symbolique `INADDR_LOOPBACK`.

6.3.4 L'ordre réseau des octets

Rappelons que la valeur décimale `4660`, soit `1234h`, s'écrit `12h, 34h` dans l'ordre grand boutien des octets et `34h, 12h` dans l'ordre petit boutien. Il a été décidé qu'on utiliserait l'ordre grand boutien des octets sur les réseaux.

Des fonctions sont fournies pour faciliter la conversion.
Il y a deux directions de conversion à considérer :

- de l'hôte vers le réseau ;
- du réseau vers l'hôte.

Il y a deux catégories de fonctions à considérer :

- la conversion d'entiers courts (de seize bits) ;
- la conversion d'entiers longs (de trente-deux bits).

On utilise donc les fonctions suivantes dont les noms sont suffisamment parlants :

```
#include <netinet/in.h>

unsigned long htonl(unsigned long hostlong);
unsigned short htons(unsigned short hostshort);
unsigned long ntohl(unsigned long netlong);
unsigned short ntohs(unsigned netshort);
```

Ces fonctions sont décrites dans la page de manuel `byteorder(3)`.

6.3.5 Connexion au serveur

Créer une socket revient à réserver un espace mémoire pour entreposer temporairement les données en transit et initialiser un certain nombre de paramètres. Par contre aucune adresse réseau ne lui est encore associée. Une fois qu'une socket client est créée, on doit la connecter au serveur en lui indiquant l'adresse de la socket serveur. Cette adresse est un chemin dans le cas d'une communication locale et une adresse de réseau dans le cas d'une communication entre machines distantes.

6.3.5.1 Appel système

Syntaxe.- Pour établir une connexion vers une socket distante, on utilise l'appel système suivant :

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

int connect(int socket, struct sockaddr *adresse_destination, int longueur_adresse);
```

Sémantique.- Le premier argument est le numéro de fichier socket obtenu lors de la création de celle-ci. Le second argument est un pointeur à une structure décrivant l'adresse de la socket distante ; les deux premiers octets de celle-ci déterminent la famille d'adresse et la façon dont les octets suivants sont interprétés. Le troisième argument indique la longueur de cette structure, en octets. Ce dernier argument est indispensable, car avant d'analyser le type de la socket, le noyau doit copier l'adresse depuis l'espace de l'utilisateur vers son propre espace mémoire, et doit donc connaître la longueur de la structure.

Lorsque la connexion s'établit, la valeur de retour est nulle. Sinon elle prend la valeur -1 et la nature du problème est indiquée grâce à `errno`.

6.3.5.2 Exemple de serveur : le service date et heure d'Unix

Comme exemple nous allons nous connecter au serveur, toujours présent sous Unix, donnant la date et l'heure. Commençons par étudier celui-ci. Les systèmes `Unix` possèdent un certain nombre de services sous forme de serveur. C'est le cas de `daytime` qui indique la date et l'heure.

Premier essai.- Ce service correspond au port 13 de l'ordinateur. On peut l'essayer avec `telnet` :

```
# telnet 127.0.0.1 13
Trying 127.0.0.1...
telnet: connect to address 127.0.0.1: Connection refused
```

Mise en service.- Dans notre cas le service n'est pas mis en place. Il suffit de décommenter la ligne commençant par 'daytime' du fichier `/etc/inetd.conf` :

```
# daytime stream tcp nowait root internal
```

puis de remettre à jour le démon réseau :

```
# /etc/init.d/inetd reload
Reload service inetd                          done
```

en ayant les droits de super-utilisateur.

Deuxième essai.- Maintenant nous devrions avoir une réponse :

```
# telnet 127.0.0.1 13
Trying 127.0.0.1...
Connected to 127.0.0.1.
Escape character is '^]'.
Sat Feb  7 10:25:04 2004
Connection closed by foreign host.
```

6.3.5.3 Exemple

Le programme suivant permet d'obtenir la date et l'heure :

```
/* client2.c */

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

void main(void)
{
  int s;
  struct sockaddr_in serveur;
  int longueur;
  const unsigned char IPno[] = {127,0,0,1}; /* local */
  char buf[80];
  int n;

  /* Creation d'une socket */

  s = socket(AF_INET, SOCK_STREAM, 0);

  /* Connexion */

  memset(&serveur, 0, sizeof(serveur));

  serveur.sin_family = AF_INET;
  serveur.sin_port = htons(13);
  memcpy(&serveur.sin_addr.s_addr, IPno, 4);

  longueur = sizeof(serveur);

  n = connect(s, (struct sockaddr *) &serveur, longueur);

  /* Reception */

  printf("Message recu : ");
  n = -1;
  while (n != 0)
    {
    n = read(s, buf, 80);
    buf[n] = '\0';
    printf("%s\n", buf);
    }

  /* Fermeture de la socket */

  close(s);
}
```

On remarquera la façon de définir ici l'adresse IP par un tableau d'octets, ceux-ci étant ordonnés suivant l'ordre réseau.

6.3.6 Initialisation d'un serveur

Notion.- Lorsqu'une socket serveur est créée, on se retrouve un peu dans la position d'un téléphone auquel on n'a pas encore attribué de numéro. Elle est peu utile alors car personne ne peut l'appeler. On doit donc lui attribuer une adresse réseau (un numéro IP et un numéro de port dans le cas d'Internet).

Syntaxe.- Pour initialiser un serveur, on utilise l'appel système :

```
bind(socket, adresse, longueur_adresse);
```

ou, plus exactement :

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

int bind(int socket, struct sockaddr * addresse, socklen_t longueur);
```

Sémantique.- Comme pour `connect()`, le premier argument est le numéro de fichier de la socket serveur obtenu lors de la création de celle-ci, le second argument est un pointeur à une structure décrivant l'adresse et le troisième indique la longueur de cette structure, en octets.

Si tout s'est bien passé, la fonction renvoie la valeur 0. Sinon elle renvoie la valeur -1 et la valeur de `errno` donne des indications sur ce qui n'a pas marché.

6.3.7 Attente de client

Notion.- Une socket serveur doit créer une file d'attente de connexions puis attendre qu'un client essaie de se connecter.

Syntaxe.- On utilise pour cela l'appel système suivant :

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

int listen(int s, int backlog);
```

Cet appel ne s'applique qu'aux sockets des types `SOCK_STREAM` et `SOCK_SEQPACKET`.

Sémantique.- Le premier argument est le numéro de fichier de socket serveur à utiliser, le deuxième spécifie la longueur de la file d'attente en nombre d'entrées.

La fonction renvoie 0 si tout s'est bien passé, -1 sinon. Dans ce dernier cas, la variable `errno` prend l'une des valeurs suivantes :

- `EBADF` : l'argument `s` n'est pas un descripteur valide ;
- `ENOTSOCK` : l'argument `s` n'est pas une socket ;
- `EOPNOTSUPP` (pour *Error OPtion NOT SUPPorted*) : la socket n'est pas d'un type qui supporte l'opération d'attente.

6.3.8 Acceptation de client

Syntaxe.- Pour accepter un client et renvoyer le numéro de fichier socket qui lui est attribué, on utilise l'appel système suivant :

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

int accept(int s, struct sockaddr *addr, int *longueur);
```

Sémantique.- Le premier argument `s` désigne le numéro de fichier de la socket serveur.

Lorsqu'une socket distante (d'un des types `SOCK_STREAM`, `SOCK_SEQPACKET` ou `SOCK_RDM`) demande à être connectée et que le serveur l'accepte (entre autre parce qu'il reste de place dans sa file d'attente), une nouvelle socket est créée du côté du serveur, dont le numéro de fichier est renvoyé par la fonction. De plus l'adresse de la socket distante et sa longueur sont renvoyées par

les deux derniers arguments de la fonction. La nouvelle socket sort alors de la file d'attente (pour vivre une vie indépendante).

Lorsque la demande de connexion n'est pas acceptée, la valeur -1 est renvoyée.

L'argument `longueur` sert à renvoyer la longueur de l'adresse de la socket distante, qui dépend de la famille d'adresses. Mais elle sert également comme entrée. On a intérêt à l'initialiser avec la longueur maximum parmi celles qui sont possibles.

6.3.9 Ouverture d'un fichier de socket

Vous connaissez la fonction :

```
#include <stdio.h>

FILE * fopen(const char *path, const char *mode);
```

permettant d'ouvrir un fichier, et surtout de l'associer à un nom physique. Les sockets ne possèdent pas de nom physique. On utilisera donc à la place la fonction :

```
#include <stdio.h>

FILE * fopen(int fd, const char *mode);
```

qui joue le même rôle mais dont le premier paramètre est un numéro de fichier.

6.3.10 Exemple de serveur

Créons un serveur TCP sur notre ordinateur associé au port 9999 qui se contente de renvoyer 'Bonjour' au format HTML lors de toute demande de connection :

```
/* serveur.c */

#include <stdio.h>
#include <sys/socket.h>
#include <unistd.h>
#include <netinet/in.h>
#include <string.h>
#include <stdlib.h>

void main(void)
{
  int s;                      /* descripteur de socket serveur */
  int c;                      /* descripteur de socket accepte */
  struct sockaddr_in serveur;
  struct sockaddr_in client;
  int longueur;
  int n;
  FILE * rx;                  /* Flux de lecture */
  FILE * tx;                  /* Flux d'ecriture */
  char entree[2048];


  /* Creation d'une socket (serveur) */

  s = socket(AF_INET, SOCK_STREAM, 0);

  /* Association d'une adresse sur le port 9999 */

  memset(&serveur, 0, sizeof(serveur));

  serveur.sin_family = AF_INET;
```

```
  serveur.sin_port = ntohs(9999);
  serveur.sin_addr.s_addr = ntohl(INADDR_ANY);

  longueur = sizeof(serveur);

  n = bind(s, (struct sockaddr *) &serveur, longueur);

  /* Mise du serveur a l'ecoute */

  listen(s, 5);

  /* Accepte une ligne de texte
   * l'ignore
   * puis envoie une page Web */

  for (;;)
    {
      /* Attend une connexion */

      longueur = sizeof(client);
      c = accept(s, (struct sockaddr *) &client, &longueur);

      /* Creation des flux */

      rx = fdopen(c, "r");
      tx = fdopen(dup(c), "w");

      /* Accepte une ligne de texte */

      fgets(entree, sizeof(entree), rx);

      /* Envoie une page HTML */

      fputs("<HTML>\n<BODY>\n<H1>Bonjour</H1>\n</BODY>\n</HTML>\n", tx);
      fclose(tx);
      fclose(rx);
    }
}
```

Pour le tester, il suffit de remplir la partie URL de votre navigateur préféré avec :

```
http://127.0.0.1:9999/
```

6.4 Données urgentes

6.4.1 Notion

Imaginons une file d'attente chez le boulanger. Chacun attend son tour pour être servi. Un gros client arrive, le restaurateur du coin, et va directement auprès de la boulangère pour être servi en priorité. Ce comportement (anti-social) a permis de court-circuiter la file d'attente. On parle de *out-of-band* en anglais. Par opposition, on parle de *in-band* lorsqu'on se place correctement dans la file d'attente.

Les **données urgentes** sur une connexion ont un comportement analogue. Normalement, le flux de données est récupéré par le destinataire dans l'ordre d'envoi de l'expéditeur. L'API des sockets, cependant, permet que des données soient envoyées en urgence au destinataire, de la même façon que ce que nous venons de voir ci-dessus.

Ceci se justifie. Un destinataire peut avoir une grande quantité de données à recevoir en attente. L'expéditeur peut décider d'annuler cette réception. Il veut donc envoyer un ordre d'an-

nulation de façon urgente, c'est-à-dire qui va court-circuiter la file d'attente en réception. Les applications qui utilisent les données urgentes sont principalement `telnet`, `rlogin` et `ftp`.

6.4.2 Envoi et réception des données urgentes

Les fonctions `write()` et `read()` sont des fonction *in-band*. On se sert des fonctions `send()` et `recv()` pour les données urgentes.

6.4.2.1 La fonction `send()`

Syntaxe.- La fonction `send()` :

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

int send(int s, const void *msg, int len, unsigned int flags);
```

possède quatre arguments :

- le numéro de socket sur laquelle il faut écrire ;
- l'adresse du tampon d'écriture ;
- la longueur du tampon d'écriture ;
- les drapeaux d'option.

Valeurs des drapeaux.- Pour une valeur 0 du vecteur des drapeaux, on se retrouve avec la fonction `write()`. On peut également y placer une conjonction des constantes symboliques suivantes :

- `MSG_OOB` (pour *Out-Of-Band*) : envoyer des données urgentes ;
- `MSG_DONTROUTE` : ne pas utiliser le routeur pour envoyer les données (mais la route que l'on précise) ; intéressant pour des diagnostics ;
- `MSG_DONTWAIT` : opération non bloquante ;
- `MSG_NOSIGNAL` : ne pas envoyer de signal `SIGPIPE` lorsque la machine distante a fermé la connexion, mais le code d'erreur `EPIPE` est renvoyé ;
- `MSG_CONFIRM` : on veut une confirmation de la part de la machine distante (depuis Linux 2.3).

On trouve une description de ces valeurs dans la page `man` de `sendmsg`. C'est surtout `MSG_OOB` qui nous intéresse ici.

6.4.2.2 La fonction `recv()`

Syntaxe.- Le prototype de la fonction `recv()` est analogue à celui de la fonction `send()` :

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

int recv(int s, const void *msg, int len, unsigned int flags);
```

Valeurs du vecteur des drapeaux.- Pour une valeur 0 du vecteur des drapeaux, on se retrouve avec la fonction `read()`. On peut également y placer une conjonction des constantes symboliques suivantes :

- `MSG_OOB` (pour *Out-Of-Band*) : recevoir des données urgentes ;
- `MSG_PEEK` : récupérer les données sans les enlever de la file d'attente ;

- `MSG_WAITALL` : l'opération est bloquante jusqu'à ce que tout soit reçu ;
- `MSG_NOSIGNAL` : ne pas envoyer de signal `SIGPIPE` lorsque la machine distante a fermé la connexion, mais le code d'erreur `EPIPE` est renvoyé ;
- `MSG_TRUNC` : renvoyer la taille réelle du paquet ;
- `MSG_ERRQUEUE` : recevoir les erreurs dans la file d'attente.

On trouve une description de ces valeurs dans la page `man` de `recv`. C'est surtout `MSG_OOB` qui nous intéresse ici.

6.5 Socket pour communication non connectée

6.5.1 Cycles de vie dans le cas des communications non connectées

Dans le cas des communications non connectées, les étapes du cycle de vie d'un serveur sont les suivantes :

- création de la socket serveur ;
- initialisation du serveur (concrétisé par l'association d'une adresse Internet et d'un numéro de port dans le cas de UDP) ;
- en général une boucle infinie du type :
  - attente d'une réception ;
  - envoi de la réponse ;
- fermeture éventuelle.

Dans le cas d'une communication non connecté, l'attente de client est émulée librement au gré du programmeur. Une façon simple consiste en une **scrutation** (*polling* en anglais) : une boucle infinie attend l'arrivée de quelque chose en utilisant un ordre de lecture ; le serveur répond par un ordre d'écriture.
Les étapes du cycle de vie d'un client sont les suivantes :

- création de la socket client ;
- envoi d'un message au serveur ;
- attente d'une réponse ;
- éventuellement d'autres cycles envoi–réception ;
- fermeture.

6.5.2 Les fonctions d'envoi et de réception de message

Les fonctions `read()` et `write()`, utilisées pour l'envoi et la réception de messages dans le cas d'une paire de sockets locales, supposent l'utilisation d'un protocole orienté connexion (tel que TCP). Dans le cas d'un protocole sans connexion (tel que UDP), on doit spécifier l'adresse de destination à chaque fois. On doit donc utiliser un couple différent de fonctions de lecture-écriture : il s'agit de `sendto()` et de `recvfrom()`.

6.5.2.1 Fonction d'envoi pour communication non connectée

La fonction `sendto()` :

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

int sendto(int s, const void *msg, int len, unsigned int flags,
           const struct sockaddr *to,
           int tolen);
```

possède six arguments :

- les quatre premiers sont ceux de la fonction `send()` ;
- un pointeur à une adresse générique de socket, indiquant où doit être envoyé le datagramme ;
- la longueur de cette adresse.

La valeur renvoyée par la fonction est le nombre d'octets envoyés (mais rien n'assure qu'ils soient reçus) ou -1 lorsqu'une erreur est survenue.

6.5.2.2 Fonction de réception pour communication non connectée

La fonction `recvfrom()` :

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

int recvfrom(int s, const void *msg, int len, unsigned int flags,
             const struct sockaddr *from,
             int *fromlen);
```

possède également six arguments :

- les quatre premiers sont ceux de la fonction `recv()` ;
- un pointeur à une adresse générique de socket, indiquant où doit être déposé l'adresse de l'expéditeur du datagramme ;
- la longueur de cette adresse, ou plus exactement un pointeur puisqu'on utilise un passage par adresse.

La valeur renvoyée par la fonction est le nombre d'octets reçus ou -1 lorsqu'une erreur est survenue.

Remarquons que l'on doit initialiser `*fromlen` avec la taille maximum des adresses de socket car cette valeur est utilisée pour réserver de la place pour l'adresse.

6.5.3 Exemple

Écrivons un serveur UDP qui attend des clients et qui renvoie 'Essai' à chacun d'eux sans tenir compte de ce qui est reçu.

6.5.3.1 Programmation du serveur

Le client peut être créé grâce au programme suivant :

```
/* udpServer.c */

#include <stdio.h>
#include <sys/socket.h>
```

```
#include <netinet/in.h>
#include <string.h>
#include <time.h>

void main(void)
{
  int s;
  struct sockaddr_in serveur;
  struct sockaddr_in client;
  int longueur;
  const unsigned char IPno[] = {127,0,0,1}; /* local */
  char buf[80];

  /* Creation d'une socket */

  s = socket(AF_INET, SOCK_DGRAM, 0);

  /* Creation d'une adresse Internet */

  memset(&serveur, 0, sizeof(serveur));

  serveur.sin_family = AF_INET;
  serveur.sin_port = htons(8888);
  memcpy(&serveur.sin_addr.s_addr, IPno, 4);

  longueur = sizeof(serveur);

  /* Initialisation du serveur */

  bind(s, (struct sockaddr *) &serveur, longueur);

  /* Attente de client */

  for (;;)
    {
      longueur = sizeof(client);
      recvfrom(s, buf, sizeof(buf), 0, (struct sockaddr *) &client, &longueur);

      /* Envoi */

      strcpy(buf,"Essai\n");
      sendto(s, buf, strlen(buf), 0, (struct sockaddr *) &client, longueur);
    }

  /* Fermeture de la socket */

  close(s);
}
```

6.5.3.2 Programmation du client

Le client peut être programmé de la façon suivante :

```
/* udpClient.c */

#include <stdio.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <string.h>

void main(void)
{
```

```
  int s;
  struct sockaddr_in serveur;
  struct sockaddr_in adr;
  int longueur;
  const unsigned char IPno[] = {127,0,0,1}; /* local */
  char buf[80];
  int n;
  int x;

  /* Creation d'une socket */

  s = socket(AF_INET, SOCK_DGRAM, 0);

  /* Creation d'une adresse */

  memset(&serveur, 0, sizeof(serveur));

  serveur.sin_family = AF_INET;
  serveur.sin_port = htons(8888);
  memcpy(&serveur.sin_addr.s_addr, IPno, 4);

  longueur = sizeof(serveur);

  /* Envoi d'un message */

  strcpy(buf, " ");
  n = sendto(s, buf, strlen(buf), 0, (struct sockaddr *) &serveur, longueur);

  /* Reception */

  printf("Message recu : ");
  n = recvfrom(s, buf, 80, 0, (struct sockaddr *) &adr, &x);
  buf[n] = '\0';
  printf("%s\n", buf);

  /* Fermeture de la socket */

  close(s);
}
```

6.5.3.3 Mise en place

Dans une première fenêtre de terminal, compiler le serveur et le faire exécuter :

```
linux # gcc udpServer.c -o udpS
udpServer.c: In function `main':
udpServer.c:10: warning: return type of `main' is not `int'
linux # ./udpS
```

Dans une seconde fenêtre, compiler le client et le faire exécuter :

```
linux # gcc udpClient.c -o udpC
udpClient.c: In function `main':
udpClient.c:9: warning: return type of `main' is not `int'
linux: # ./udpC
Message recu : Essai

linux: #
```

6.6 Semi-arrêt d'une socket

Introduction.- Nous avons vu comment fermer une socket, comme on ferme tout fichier. Cette façon de faire pose problème lorsqu'un processus local veut signaler à une socket distante qu'il n'a plus de données à envoyer. Si ce processus veut recevoir une confirmation de la socket distante, il ne le peut pas puisque la socket est fermée. Il faut donc trouver un moyen qui ferme la socket à moitié.

Syntaxe.- On utilise pour cela la fonction :

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

int shutdown(int s, int how);
```

Le premier argument est le numéro de la socket à arrêter. Le second argument indique comment cette socket doit être arrêtée, la valeur devant être l'une des trois suivantes :

Valeur	Constante symbolique	Signification
0	`SHUT_RD`	On ne pourra plus lire sur la socket
1	`SHUT_WR`	On ne pourra plus écrire sur la socket
2	`SHUT_RDWR`	On ne pourra plus ni lire ni écrire

La valeur 2 correspond à `close()`.

L'appel à `shutdown` ne libère pas le numéro de fichier, même avec la valeur `SHUT_RDWR`. Il faut un appel à `close()` pour le libérer.

Valeur de retour.- La valeur de retour est 0 si l'appel de la fonction réussit. Elle est -1 en cas d'échec, la variable `errno` prenant alors l'une des valeurs suivantes :

- `EBADF` si le numéro de fichier n'est pas un numéro valide ;
- `ENOTSOCK` si le numéro de fichier ne correspond à une socket ;
- `ENOTCONN` si la socket spécifiée n'est pas connectée.