1 INTERFACE DES SOCKETS 1

1.1 INTRODUCTION 1
1.2 ELEMENTS COMMUNS AUX MODES DATAGRAMME ET CONNECTE 2
1.2.1 DIVERSES STRUCTURE DE DONNEES ET FONCTIONS 2
1.2.2 CREATION D'UN SOCKET : SOCKET() 2
1.2.3 FERMETURE D'UN SOCKET : CLOSE() 3
1.2.4 SPECIFICATION D'ADRESSE LOCALE : BIND() 3
1.2.5 AUTRES STRUCTURES DE DONNEES ET PRIMITIVES 3
1.3 DIALOGUE CLIENT-SERVEUR EN MODE CONNECTE 3
1.3.1 CONNEXION DES SOCKETS AVEC L'ADRESSE DE DESTINATION : CONNECT() 3
1.3.2 CREATION D'UNE FILE D'ATTENTE : LISTEN() 3
1.3.3 ACCEPTATION D'UNE CONNEXION ENTRANTE : ACCEPT() 3
1.3.4 LECTURE-ECRITURE EN MODE CONNECTE : WRITE()/READ() 4
1.3.5 UN EXEMPLE D'APPLICATION CLIENT-SERVEUR : BIGBEN / VERSION 'CONNECTEE' 5

# 1 Interface des sockets

## 1.1 Introduction

Les sockets (que l'on peut traduire par 'prise') fournissent un mécanisme de communication inter-processus, développé à l'origine sur UNIX BSD4.2 par BSD (Berkeley Software Distribution). Ce mécanisme est maintenant utilisé dans d'autres environnements comme, par exemple, les environnements Windows de Microsoft et les différentes variantes d'UNIX. Ils constituent une interface applicative, adaptée à la pile de protocoles TCP/IP, bien qu'on puisse les utiliser sur d'autres familles de protocoles (on se limitera par la suite à l'environnement TCP/IP).
Le concept de tube (pipe) du système UNIX permet à deux processus d'une même machine d'établir une voie de communication bidirectionnelle. Les sockets généralisent ce concept dans le sens où les processus peuvent être sur des machines différentes avec éventuellement des systèmes d'exploitation différents. Ainsi, une application tournant sur un PC dans un environnement Windows pourra dialoguer avec un processus sur une machine UNIX.
Un tel dialogue, non symétrique, utilisera un mode de fonctionnement de type client-serveur :

- le serveur attend une demande de connexion d'un client,
- le client envoie, de façon asynchrone, une demande de connexion au serveur.

Un socket est un brin de communication qui porte un *nom*, un *type* et qui est associé à un processus.
A chaque type de socket correspond un protocole utilisé des couches inférieures (Figure 1-1), et par conséquent un ensemble de propriétés sur la communication établie y est associé :

- socket en mode connecté (stream socket), utilise TCP (remise fiable),
- socket en mode datagramme (datagram socket), utilise UDP (remise non fiable),
- socket en mode caractère (raw socket), utilise directement les couches inférieures.



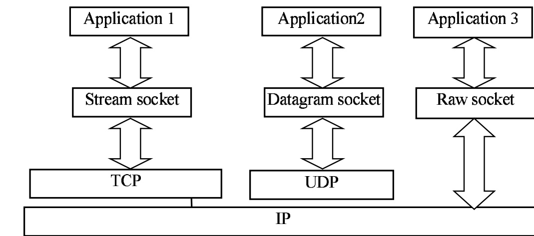
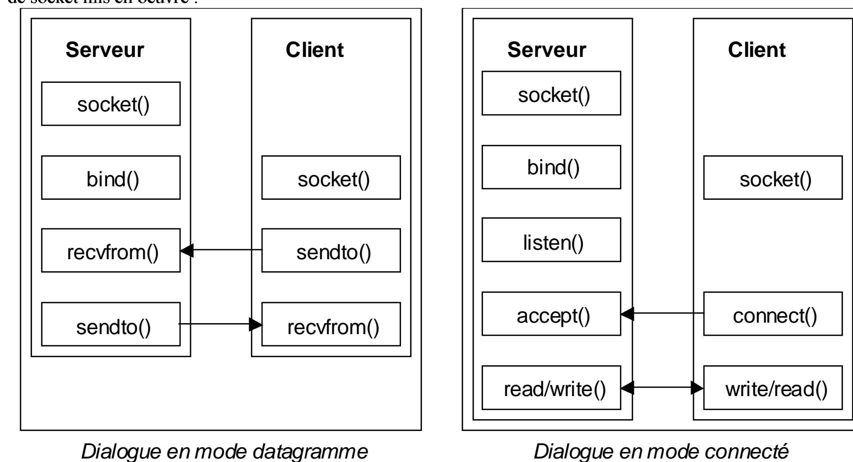



*Figure 1-1 Types de socket et protocoles*

La forme du dialogue entre le client et le serveur, ainsi que la nature des primitives utilisées dépendent du type de socket mis en oeuvre :




## 1.2 Eléments communs aux modes datagramme et connecté

### 1.2.1 Diverses structure de données et fonctions

Pour les protocoles de la famille TCP/IP, les caractéristiques du socket sont stockées dans une structure de données de type *sockaddr_in* :

```
#include <sys/types.h>      /*Bibliothèques requises */
#include <sys/socket.h>
struct sockaddr_in {
       short         sin_family;   /*famille de protocole*/
       u_short       sin_port;     /*numero de port*/
       struct        in_addr;      /*adresse IP*/
       char          sin_zero[8];  /*non utilisé*/
}
```

### 1.2.2 Création d'un socket : socket()

La primitive *socket()* permet d'ouvrir un socket, sans le relier à une adresse IP :
int **socket**(int af, int type, int protocole),

- af = famille de protocole utilisée (AF_INET pour TCP/IP),
- type = type de socket (SOCK_STREAM pour TCP, SOCK_DGRAM pour UDP),
- protocole = 0 (pour TCP ou UDP)

*socket()* retourne un descripteur de socket, analogue à un descripteur de fichier, qui permettra d'identifier le socket.

#### 1.2.3 Fermeture d'un socket : close()

La primitive *close()* permet de fermer un socket :
**close**(int socket),

- socket = descripteur de socket,

#### 1.2.4 Spécification d'adresse locale : bind()

La primitive *bind()* permet de lier un socket avec une adresse locale en renseignant les champs adresse IP et numéro de port :
int **bind**(int socket, struct sockaddr_in* adresse-locale, int longueur-adresse),

- socket = descripteur de socket,
- adresse-locale = pointeur sur une structure qui contient la famille de protocole et l'adresse (adresse IP + n° de port),
- longueur-adresse = longueur de la zone pointée par adresse-locale.

*bind()* retourne 0 si l'appel s'est déroulé sans erreur, -1 dans les autres cas.

#### 1.2.5 Autres structures de données et primitives

Les primitives suivantes proposent des fonctionnalités rendant plus souple la programmation des sockets :

- *gethostbyname()* permet de traduire un nom de domaine en adresse IP codée sur trois octets en utilisant la structure de données *hostent*,
- *gethostbyserv()* permet de traduire en numéro de port le nom d'un service,
- *htonl, ntohl, htons()* et *ntohs()* permettent de s'affranchir des problèmes liés aux conversions d'ordre des octets. Sur certaines machines (comme par exemple les i80x86), l'ordre des octets de l'hôte est LSB (Least Significant Byte first), c'est à dire octet de poids faible en premier, alors que sur les réseaux, comme l'Internet, l'ordre est MSB (Most Significant Byte first) octet de poids fort en premier. Par exemple, la fonction *htonl()* convertit un entier long *hostlong* depuis l'ordre des octets de l'hôte vers celui du réseau
- *getsockname()* permet de récupérer certaines caractéristiques d'un socket, notamment son numéro de port.

## 1.3 Dialogue client-serveur en mode connecté

#### 1.3.1 Connexion des sockets avec l'adresse de destination : connect()

Uniquement utilisable en mode connecté, *connect()* établit une connexion TCP avec le destinataire.
int **connect**(int socket, struct sockaddr_in* adr-destination, int longueur-adr) :

- socket = descripteur de socket,
- adresse-destination = pointeur sur une structure qui contient la famille de protocole et l'adresse (adresse IP + n° de port),
- longueur-adresse = longueur de la zone pointée par adresse-destination.

*connect()* retourne 0 si l'appel s'est déroulé sans erreur, -1 dans les autres cas.

#### 1.3.2 Création d'une file d'attente : listen()

La primitive *listen()* permet de créer une file d'attente dans laquelle seront placées les demandes de connexion entrantes. Cette file d'attente sera exploitée par *accept()*.
**listen**(int socket, int lgr-file) :

- socket = descripteur de socket,
- lgr-file = précise la taille de la file d'attente, soit le nombre de connexions simultanées que peut traiter le serveur

#### 1.3.3 Acceptation d'une connexion entrante : accept()

La primitive *accept()* permet la prise en compte d'une demande de connexion entrante. Cette primitive est bloquante, dans le sens où le processus, exécutant cette primitive, se met en attente jusqu'à ce qu'une connexion entrante se présente.
newsock = **accept** (int socket, struct sockaddr_in* adresse, int* lgr-adresse) :

- socket = descripteur de socket,



- adresse = pointeur sur une structure qui contient la famille de protocole et l'adresse (adresse IP + n° de port). Cette adresse contient la description de l'adresse de l'appelant.
- lgr-adresse = pointeur sur un entier précisant la taille de la zone pointée par adresse.

*accept()* retourne un nouveau descripteur de socket (entier) sur lequel va s'effectuer le dialogue entre le client et le serveur. Pour traiter cette demande de connexion, le processus peut :

- traiter lui-même cette nouvelle connexion,
- déléguer le traitement de cette nouvelle connexion à un autre processus, puis se mettre en attente d'une nouvelle demande de connexion (primitive accept).

#### 1.3.4 Lecture-écriture en mode connecté : write()/read()

Les primitives *write()* et *read()* permettent l'envoi et la réception de messages en mode connecté.
int **write**(int socket, char* tampon, int longueur ) :

- socket = descripteur de socket 'émetteur',
- tampon = pointeur sur la zone mémoire contenant les données à envoyer,
- longueur = nombre d'octets à envoyer.

*write()* retourne le nombre d'octets effectivement envoyés, -1 en cas d'erreur.
int **read**(int socket, char* tampon, int longueur) :

- socket = descripteur de socket 'récepteur',
- tampon = pointeur sur la zone mémoire prête à recevoir les données,
- longueur = nombre d'octets à lire.

*read()* retourne le nombre d'octets effectivement lus, -1 en cas d'erreur.

### 1.3.5 Un exemple d'application client-serveur : BIGBEN / version 'connectée'

#### 1.3.5.1 Le code du client

```
/* exemple Client/Serveur mode connecte TCP */
/* client bigben */

/* usage: client machine_serveur port */

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

#define FATAL(err) { perror((char *)err); exit(1); }

static int fd;

int connexion(char *, int);
void travail(int);


int main(int argc, char * argv[])
{
  int port;
  char *hostname;

  if (argc != 3) {
    printf("Usage: %s machine_serveur port\n",argv[0]);
    exit(1);
  };

  /* ouverture de la connexion */
  hostname=argv[1];
  port=atoi(argv[2]);
  fd=connexion(hostname,port);

  /* travail */
  travail(fd);

  close(fd);
  exit(0);
}

/* -------------------------------------------------------- */
/* Fonction connexion
   Ouvre la connexion vers un port d'un serveur.
   Retourne la socket ouverte
   Exit en cas d'echec
   */
int connexion(char *hostname, int port)
{
  int fdPort;
  struct sockaddr_in addr_serveur;
  socklen_t lg_addr_serveur = sizeof addr_serveur;
  struct hostent *serveur;

  /* creation de la prise */
  fdPort=socket(AF_INET,SOCK_STREAM,0);
  if (fdPort<0)
    FATAL("socket");

  /* recherche de la machine serveur */
  serveur = gethostbyname(hostname);
  if (serveur == NULL)
    FATAL("gethostbyname");

  /* remplissage adresse socket du serveur */
  addr_serveur.sin_family      = AF_INET;
  addr_serveur.sin_port        = htons(port);
```



```
  addr_serveur.sin_addr        = *(struct in_addr *) serveur->h_addr;

  /* demande de connexion au serveur */
  if (connect(fdPort,(struct sockaddr *)&addr_serveur, lg_addr_serveur) < 0)
    FATAL("connect");

  return fdPort;
}

/* -------------------------------------------------------- */
/* Fonction travail
   Effectue le dialogue avec le serveur
   */
void travail(int fd)
{
  char h[3],m[3],s[3];

  /* recuperation reponse du serveur */
  if (read(fd,h,2) != 2) FATAL("read h");
  h[2]='\0';
  if (read(fd,m,2) != 2) FATAL("read m");
  m[2]='\0';
  if (read(fd,s,2) != 2) FATAL("read s");
  s[2]='\0';

  printf("Il est %s:%s:%s sur le serveur\n",h,m,s);
}
```

#### 1.3.5.2 Le code du serveur

```
/* exemple Client/Serveur mode connecte TCP */
/* serveur Bigben */

/* usage: serveur port */

#include <unistd.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <sys/wait.h>
#include <netinet/in.h>
#include <signal.h>
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

#define FATAL(err) { perror((char *)err); exit(1); }


static int fd;

void Interruption() {
  close(fd);
  printf("Interruption : fin du serveur\n");
  exit(0);
}

int init_service(int);
void travail_fils(int);


int main(int argc, char * argv[])
{
  int fdTravail, port;

  if (argc != 2) {
    printf("usage: %s port\n",argv[0]);
    exit(1);
  };

  /* positionnement du signal d'interruption */
  signal(SIGINT,Interruption);

  /* initialisation du service */
  port=atoi(argv[1]);
  fd=init_service(port);
```

```
  /* gestion des connexions de clients */
  while(1) {
    /* acceptation d'une connexion */
    fdTravail=accept(fd,NULL,NULL);
    if (fdTravail<=0)
      FATAL("accept");

    if (fork()==0) { /* fils : gestion du dialogue avec client */
      close(fd);
      travail_fils(fdTravail);
      close(fdTravail);
      exit(0);
    }
    else { /* pere : repart a l'ecoute d'une autre connexion */
      close(fdTravail);
    }
  }
}

/* ---------------------------------------------------------- */
/* Fonction init_service
   Ouvre le service sur un port
   Retourne la socket ouverte
   Exit en cas d'echec
   */
int init_service(int port)
{
  int fdPort;
  struct sockaddr_in addr_serveur;
  socklen_t lg_addr_serveur = sizeof addr_serveur;

  /* creation de la prise */
  fdPort=socket(AF_INET,SOCK_STREAM,0);
  if (fdPort<0)
    FATAL("socket");

  /* nommage de la prise */
  addr_serveur.sin_family      = AF_INET;
  addr_serveur.sin_addr.s_addr = INADDR_ANY;
  addr_serveur.sin_port        = htons(port);
  if (bind(fdPort,(struct sockaddr *)&addr_serveur, lg_addr_serveur) < 0)
    FATAL("bind");

  /* Recuperation du nom de la prise */
  if (getsockname(fdPort,(struct sockaddr *)&addr_serveur, &lg_addr_serveur) < 0)
    FATAL("getsockname");

  /* Le serveur est a l'ecoute */
  printf("Le serveur ecoute le port %d\n",ntohs(addr_serveur.sin_port));

  /* ouverture du service */
  listen(fdPort,4);

  return fdPort;
}

/* ---------------------------------------------------------- */
/* Fonction travail_fils
   Effectue le dialogue avec le client sur une autre socket
   Execute par le processus fils pendant le pere s'est remis
   a l'ecoute sur la socket du service
   */
void travail_fils(int fdTravail)
{
  long horloge;
  struct tm *temps;
  char tampon[2];
  int h,m,s;

  /* preparation de la reponse */
  time(&horloge);
  temps=localtime(&horloge);
  h = temps->tm_hour;
  m = temps->tm_min;
  s = temps->tm_sec;

  /* envoi de la reponse */
```



```
  sprintf(tampon, "%02d", h);
  write(fdTravail,tampon,2);
  sprintf(tampon, "%02d", m);
  write(fdTravail,tampon,2);
  sprintf(tampon, "%02d", s);
  write(fdTravail,tampon,2);
}
```