encadrés par C. Attiogbé, S. Faucou

**Programmation avec les sockets**

Tous les exos doivent être faits (finissez à la maison ceux qui ne sont pas traités en TD)
Nombre de séances : ... Compte-rendu demandé : ...selon consignes...

Nous allons dans la suite des TD/TP réaliser un noyau d'un système réparti pour la surveillance d'une zone industrielle. Une première mise en œuvre sera faite avec les Sockets. Puis on étudiera une mise en oeuvre avec les RMI de Java.

# 1 Généralités sur les sockets

Un socket constitue un mécanisme de communication entre processus du système Unix/-Linux (mais pas exclusivement). Il sert d'interface entre les applications et les couches réseau. Il existe pour cela une bibliothèque système (ensemble de fonctions/primitives) permettant de gérer les services de communication offerts par les sockets. Ces fonctions sont décrites dans la suite.

Le principe général de la communication entre applications (dites 'serveur' et 'client') est illustré dans la figure 1.

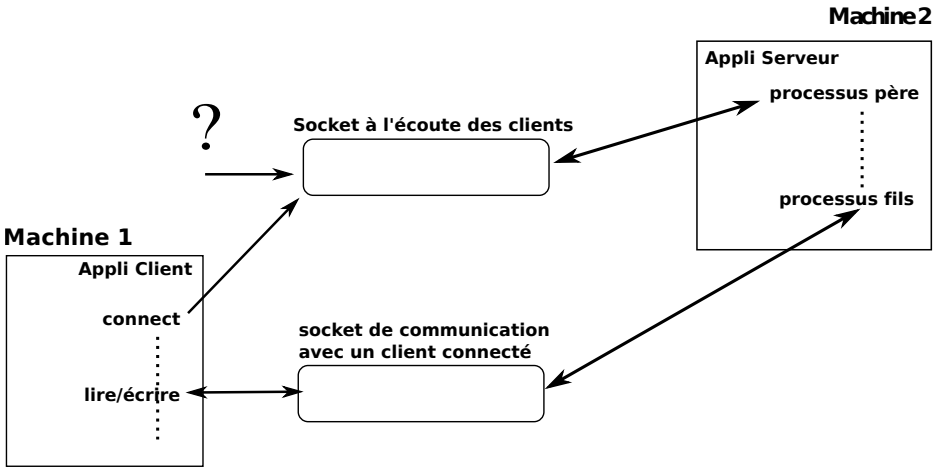


FIGURE 1 – Principe général de la communication avec des sockets

Il y a **plusieurs types de socket**. Le type de socket dépend des caractéristiques de la communication que l'on souhaite établir à l'aide du socket :

— unicité des envois (pas de doublons),
— respect de la séquentialité des envois,
— efficacité de distribution et sécurité,

— autorisation de messages hors-normes (*out-of-band*), c'est un message délivré hors du flux normal des données,
— support de mode connecté,
— possibilité de rendez-vous entre les processus communicants par l'intermédiaire des sockets.

Un socket est toujours dans un *domaine* de communication et son *type* est défini par rapport aux propriétés du domaine de communication.

Un *domaine* de communicaton est une famille abstraite qui définit un mode d'adressage (standard d'adressage) et un ensemble de protocoles utilisables par les sockets. Il existe une variété de domaines : `UNIX, INTERNET, X25, DECNET, APPLETALK, ISO, SNA,` ...

On retrouve tous les domaines dans le fichier <`socket.h`> :

```
#define AF_UNIX 1
#define AF_INET 2
...
```

### Modes de communication

On distingue :
— les sockets en **mode connecté** (protocole TCP sous IP). Ces sockets ouvrent une liaison bidirectionnelle sécurisée et séquentielle antre deux processus, et garantissent l'unicité des transferts.
Le mode séquentiel établit un **circuit virtuel** de communication **point à point**.
— les sockets à base de datagramme en **mode non connecté** (protocole UDP sous IP). Ces sockets permettent le transfert de fichiers de façon bidirectionnelle mais ne garantissent ni l'unicité, ni la séquentialité des transferts. Ce mode n'est pas fiable, il peut y avoir des pertes.
— les sockets en mode caractère (**raw socket**). Ce type de socket fonctionne en mode datagramme et est destiné aux utilisateurs qui développent de nouveaux protocoles.

### Mise en œuvre du modèle client serveur

L'approche courante de mise en œuvre des sockets consiste à utiliser le modèle *client/serveur* illustré par la figure 2.

## 2 Principales fonctions en langage C

Tous les applications réseaux utilisant les sockets ont un squelette type selon le mode *avec connexion* ou *sans connexion*.

La figure 3 illustre l'architecture d'une application en mode connecté.

La figure 4 illustre l'architecture d'une application en mode non connecté.

### Création d'un socket

```
#include <types.h>    /* relativement au répertoires <linux> ou <sys> pour unix.   */
```

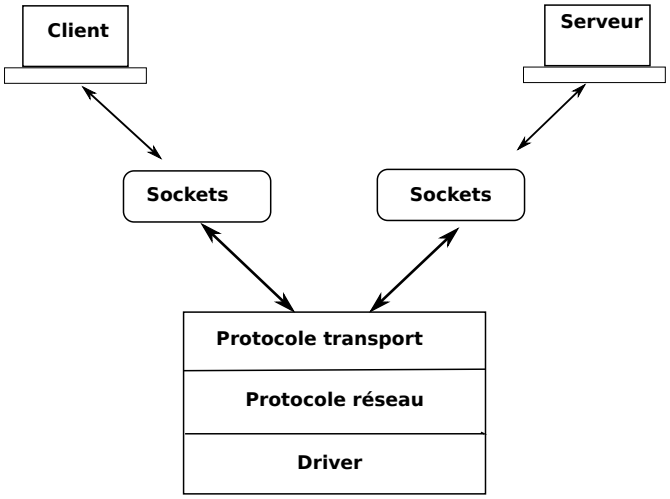


FIGURE 2 – Modèle client/serveur

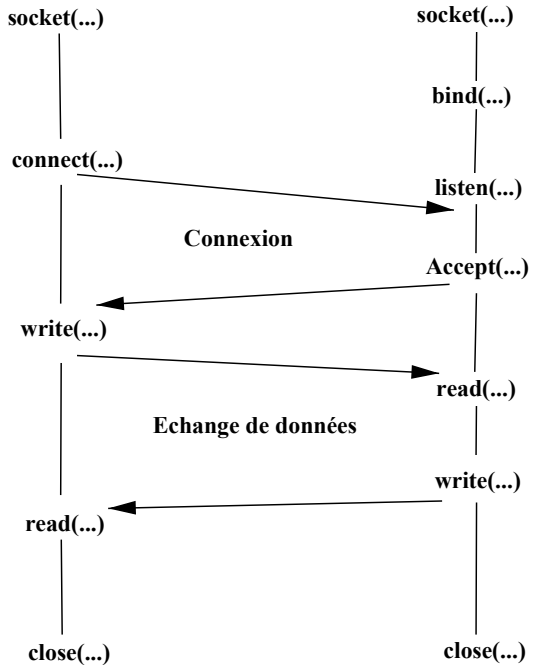


FIGURE 3 – Architecture des programmes avec connexion

```
#include <sochet.h>   /* idem, et dans toute la suite du document */
int socket(int domaine, int type, int protocole);
```

La fonction retourne -1 en cas d'erreur sinon un descripteur du socket créé.
La fonction `socket` crée un nouveau socket en précisant son type de protocole avec les paramètres `domaine` et `type`. Le paramètre **`domaine`** désigne la famille de protocoles auquel appartient le socket ; par exemple AF_UNIX pour le protocole interne à unix et AF_INET pour le protocole IP (Internet).
Le paramètre **`type`** identifie le type de protocole ; par exemple SOCK_STREAM pour le mode connecté, c'est TCP pour le protocole IP) ou SOCK_DGRAM pour le mode non connecté (*connectionless*), c'est d'UDP pour le protocole IP.
Le paramètre **`protocole`** est généralement initialisé à 0.

Il faut noter que seule la partie protocole associée au socket est initialisée. Les autres le

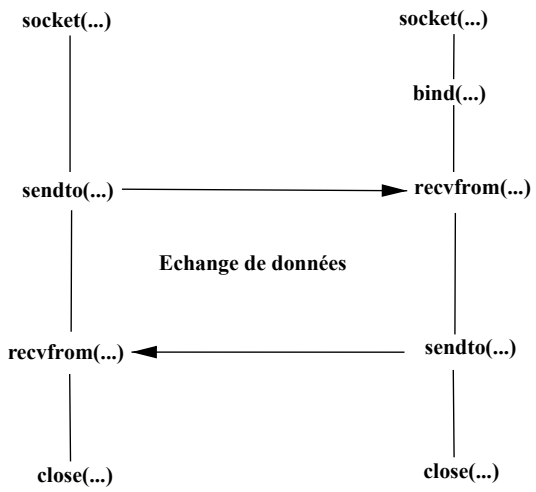


FIGURE 4 – Architecture des programmes sans connexion

seront par d'autres appels système.

**Exemple**

```
...
descrsoc = socket(PF_INET, SOCK_STREAM, 0);
...
```

Ce fragment de programme crée, lorsque la fonction s'est bien terminée, un nouveau socket de type PF_INET et retourne son descripteur dans la variable `descrsoc`.

## Liaison d'un socket à une machine (adr. IP)

La fonction **`bind`** lie un socket déjà créé à l'adresse d'une machine.

```
#include <types.h>  /*  <linux/types> ou <sys/types> pour unix.   */
#include <sochet.h>
int bind(int soc, const struct sockaddr *stradsoc, int longstradsoc);
```

La fonction `bind` définit l'adresse locale et le port local de l'association.
Pour TCP/IP, le second argument est `const struct sockaddr_in * stradsoc`.
`longstradsoc` est la taille effective de la structure d'adresse pointée par le second argument.

**Exemple**

```
...
local_addr->sin_family      = AF_INET;
local_addr->sin_addr.s_addr = htonl(INADDR_ANY);
local_addr->sin_port        = htons(numport);
bind(soc, (struct sockaddr *)local_addr, sizeof(struct sockaddr_in));
```

Ce fragment de programme initialise l'adresse locale `local_addr` à celle d'une interface de la machine dans le doamine `AF_INET` et le port local de l'association à `numport`. Les primitives `htonl` et `htons` permettent de faire les conversions nécessaires à la portabilité du code.

## La fonction de connexion à une machine via un socket

Dans le mode connecté, un client doit se connecté au serveur avant les traitements.

```
#include <types.h>       /*  <linux/types> ou <sys/types> pour unix.   */
#include <sochet.h>
int connect(int soc, struct sockaddr *stradsocdist, int longsrtadsoc);
```

La fonction `connect` établit une connexion avec un serveur. Elle retourne 0 si tout s'est bien terminé sinon -1 en cas d'erreur.
`soc` est le socket local du client.
`stradsocdist` pointe sur l'adresse du socket avec lequel la connexion sera établie (les champs IP distant, port distant étant préalablement renseignés).

La fonction `connect` est blocante. Elle ne rend la main que lorsque la connexion est effectivement établie ou s'il y a une erreur de connexion.

### Exemple

```
char *nomhote = "leserveur";
struct sockaddr_in distant;        /* pour préparer l'adressage de la machine distante */
struct hostent *infoDistant;          /* va contenir les infos sur la machine distante */
infoDistant = gethostbyname(nomhote);       /* recup des infos sur la machine distante */

/* ... créez d'abord un socket, soc = socket(...) */

distant->sin_family = AF_INET;
distant->sin_addr.s_addr = *(u_long *)infoDistant->h_addr[0];
distant->sin_port = htonl(numport);          /* numport etant connu */.

connect(soc, (struct sockaddr *)&distant, sizeof(struct sockaddr_in));
```

Ce fragment de programme connecte le socket `soc` du processus client sur le port `numport` de l'hôte (distant) nommé "`leserveur`".

## La fonction d'écoute

```
#include <types.h>  /*  <linux/types> ou <sys/types> pour unix.   */
#include <sochet.h>
int listen(int soc, int taillequeue);
```

La fonction `listen` retourne -1 en cas d'erreur sinon 0 si tout s'est bien terminée. Elle est utilisée par un serveur pour se mettre à l'écoute des connexions venant des clients.

`taillequeue` indique le nombre maximum de demandes de connexion pouvant être en attente de traitement (acceptation) pour le serveur.

**Exemple**

```
...
listen(s,7);  // s étant un socket déjà créé et lié à une adresse IP + port
```

Permet de fixer à 7 le nombre de connexions en attente d'être acceptées par le serveur.

## La fonction d'acceptation des connexions

Après avoir effectué `listen`, le serveur en mode connecté peut attendre une demande de connexion des clients (avec la focntion `accept`).

```
#include <types.h>  /*  <linux/types> ou <sys/types> pour unix.   */
#include <sochet.h>
int accept(int soc, struct sockaddr *adr, int *longadr);
```

La fonction `accept` retourne -1 en cas d'erreur sinon le descripteur d'un nouveau socket.
La fonction `accept` prend à chaque fois la première demande de connexion en attente. Elle crée un nouveau socket ayant les mêmes propriétés que `soc`. S'il n'y a pas de demande en attente, `accept` est blocante. Le descripteur du nouveau socket créé est retourné comme résultat.

Le socket `soc` passé en paramètre a déjà ses champs (protocole, adresse locale et port local) renseignés.
`adr` est le résultat qui contient l'adresse du client connecté. `longadr` pointe sur la taille de la structure retournée.

**Exemple**

```
...
/*   préparez une structure adrcl pour garder les infos sur le client */
longstrcli = sizeof(struct sockaddr_in);
nouvsoc = accept(soc, (struct sockaddr *)adrcl, &longstrcli)) ;
```

retourne dans `nouvsoc`, un nouveau descripteur de socket (ayant les mêmes propriétés que soc) et dans `adrcl`, les informations concernant le client qui vient d'établir la connexion.

## Les fonctions de transfert de données

```
#include <types.h>  /*  <linux/types> ou <sys/types> pour unix.   */
#include <sochet.h>
int write(int soc, char *pbuf, int nreste);
int read(int soc, char *pbuf, int noct);
/* pbof : pnteur sur un buffer de caractères, nreste : taille du buffer */

int send(int soc, const char *mesg, int longmesg, int flags);
int recv(int soc, char *buffer, int longmesg, int flags);

int sendto(int soc, const char *mesg, int longmesg, int flags,
```

```
            const struct sockaddr *vers, int longvers);
int recvfrom(int soc, char *buffer, int longmesg, int flags,
            struct sockaddr *de, int *longde);
```

Toutes ces fonctions systèmes retournent -1 en cas d'erreur ; sinon elles retournent la taille des données effectivement envoyées ou lues.

Les fonctions `read` et `write` marchent en mode connecté. Elles sont semblables aux focntions d'écriture lecture de fichiers.

Les fonctions `send` et `sendto` sont utilisées pour envoyer un message `mesg` de taille `longmesg` à un destinataire.Utilisées pouir les messages hors normes.

La fonction `sendto` est utilisé pour le mode sans connexion. Le paramètre `vers` de taille `longvers` désigne le destinataire.

Les fonctions `recv` et `recvfrom` sont utilisées pour recevoir des messages à travers les sockets. `recvfrom` est utilisée dans le mode sans connexion

## La fonction de fermeture de socket

```
#include <types.h>  /*  <linux/types> ou <sys/types> pour unix.   */
#include <sochet.h>
int close(int descrsoc);
```

La fonction `close` permet de fermer un socket dont on donne le descripteur. Dans le cas du mode TCP (protocole avec transmission fiable), le système essaie d'envoyer les données restantes avant de fermer le socket.

## Divers appels système

### htonl, htons, ntohl, ntohs

Les fonctions htonl, htons, ntohl, ntohs permettent de s'affranchir des problèmes de différence entre architectures d'ordinateurs, systèmes d'exploitation et réseaux. Ces différences se manifestent à travers des formats de données.

Ces fonctions réalisent donc les conversions de format entre l'architecture de la machine courante et le format utilisé sur le réseau (avec les protocoles Internet ou autres, ...).

### getsockname

```
#include <types.h>
#include <socket.h>
int getsockname(int soc, struct sockaddr *nom, int *longnom);
```

Cette fonction retourne -1 en cas d'erreur ; sinon elle retourne 0 et dans `nom`, le nom du socket `soc` et dans `longnom`, sa longueur.

### getpeername

```
int getpeername(int soc, struct sockaddr *strsoc, int *longstrsoc);
```

Cette fonction système retourne -1 en cas d'erreur sinon 0 et dans `strsoc`, la structure associée au socket auquel le socket `soc` est connecté et dans `longstrsoc`, la longueur de ce nom.

setsockopt, getsockopt, ioctl, fcntl

Ce sont diverses fonctions système qui permettent de consulter et de modifier des options d'un socket.

# 3 Mise en oeuvre en Python et Perl

## Un serveur écrit en Python

```
# Exemple d'un Serveur (Perroquet)  - sera sur le port 50007 de la machine locale
# le serveur répete le message qu'il reçoit
#-----------------------------------------------
from socket import *
HOST  = '' #localhost ou adresseIP
PORT = 50007         # num port arbitraire
s = socket()  # socket à publier
s.bind((HOST, PORT))               # liaision au port et ecoute
s.listen(7) # maxi clients simultanement
conn, addr = s.accept() # accepte une connexion
print 'Connecte par', addr
while 1:
        data = conn.recv(1024)   # lecture dans le sock de connexion
        if not data: break
        conn.send(data)          # il répete simplement
conn.close()
```

## Un client en Perl

```
# Un exemple de client écrit en Perl
# pour envoyer un message à un serveur connu
# sur le port 50007 de la machine locale
#-----------------------------------------
use IO::Socket;
 my $sock = new IO::Socket::INET (
     PeerAddr => 'localhost',
     PeerPort => '50007',
     Proto => 'tcp',
 );
 die "Could not create socket: $!  n" unless $sock;
 print $sock "Hello Hello Hello there!  n";

if(<$sock>)
  print $_;

close($sock);
```

### Un client en Python

```
# Un exemple : Client (Bavard), écrit en Python
# pour envoyer un message a un serveur connu sur le port 50007
# ------------------------------------------------

from socket import * # importer les ressources du package socket
HOST  = 'localhost'    # nom de la machine distante
PORT = 50007 # numero arbitraire du port de communication

#s = socket(AF_INET, SOCK_STREAM) # creation du socket pour la communic
s = socket() # creation du socket pour la communic
s.connect((HOST, PORT)) #  se connecter via s sur HOST et le service PORT

# si la connexion est reussie alors
i = 1
while i <= 20:
        s.send(b'Bonjour Vieux') # b(ytes)
        i = i + 1
        # attendre reponse
        data = s.recv(1024)
        print 'ok :', data

# fermeture de la liaison
s.close()
# raport de causerie
print 'Client> j ai recu ', data
```

## Quelques références bibliographiques et "webographiques"

Il y a une littérature abondante sur le sujet. Ici nous donnons juste quelques points d'entrée.


M. Gabassi *L'informatique répartie sous unix*, Eyrolles, 1992

L. Toutain, *Réseaux locaux et Internet*, Hermès

J-M. Rifflet, *La communication sous Unix*, MacGraw Hill, 1990

R.F. Tong Tong, *Les communications sous Unix*, Eyrolles, 1993

R. Stoeckel, *Communication et Unix*, Armand Colin, 1990

Brian Hall, *Beej's Guide to Network Programming Using Internet Sockets*, site web

Jan Newmarch, *Socket-level Programming*, site web