Une introduction à la programmation avec les sockets

Ce document présente les concepts fondamentaux et les principes de base du fonctionnement des sockets. Leur compréhension en dehors de tout langage de programmation est indispensable.

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.

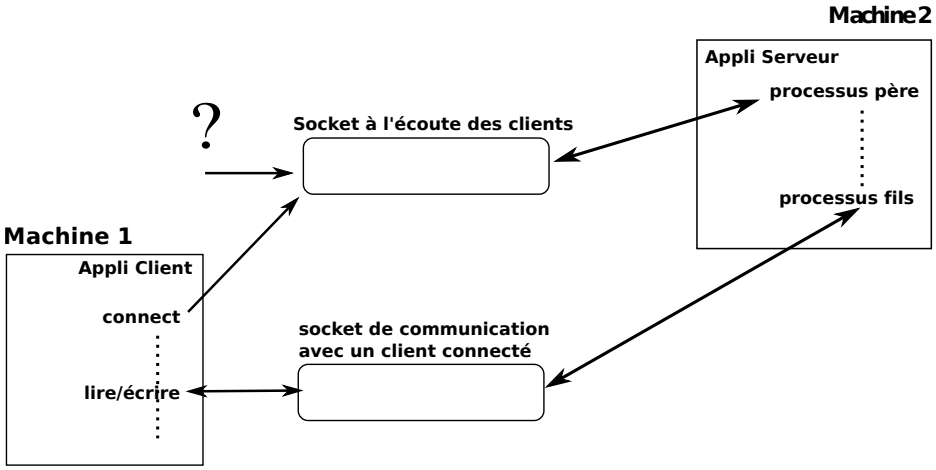

FIGURE 1 – Principe général de la communication avec des sockets

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`, ...

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.

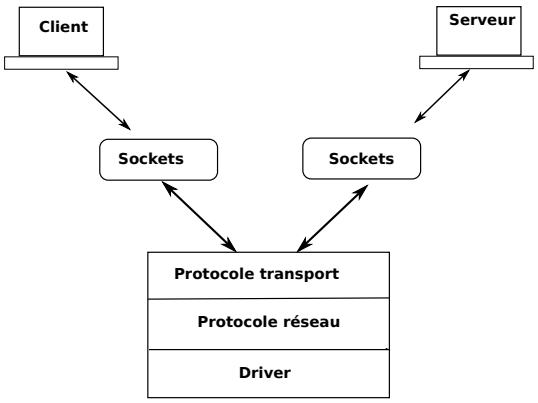

FIGURE 2 – Modèle client/serveur

2 Principales fonctions (dans différentes librairies)

Toutes 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

La fonction `socket` crée un nouveau socket en précisant son type de protocole avec les paramètres `domaine` et `type`. Son interface dépend des langages (car certains gèrent des paramètres

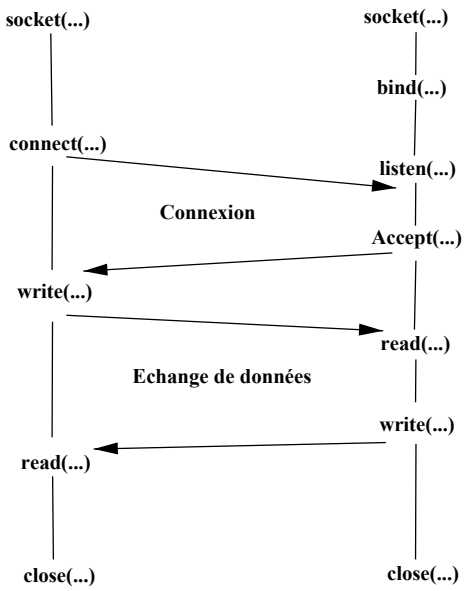

FIGURE 3 – Architecture des programmes avec connexion

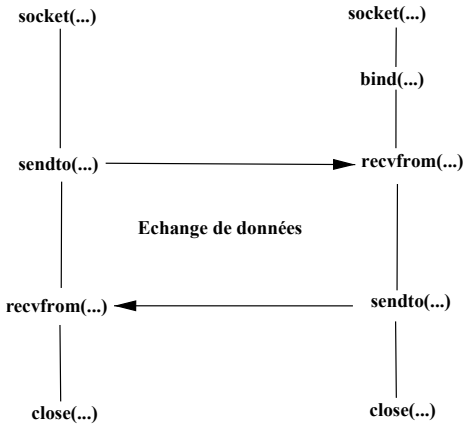

FIGURE 4 – Architecture des programmes sans connexion

par défaut).
La plupart du temps, 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.

La fonction fournit en retour un descripteur de sockect (donc à récuperer dans une variable).

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

Une fonction **`bind`** lie un socket déjà créé (dont on a le descripteur) à l'adresse IP d'une machine (à fournir ou à récupérer par la fonction).

La fonction de connexion à une machine via un socket

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

La fonction (souvent nommée **connect**) établit une connexion avec un serveur. Cette fonction est blocante ; elle ne rend la main que lorsque la connexion est effectivement établie ou s'il y a une erreur de connexion.

La fonction d'écoute

La fonction **listen** est utilisée par un serveur pour se mettre à l'écoute des connexions venant des clients. Selon les bibliothèques utilisées, elle a besoin d'un paramètre qui définit la taille de la file des processus clients pouvant attendre un traitement par le serveur.

La fonction d'acceptation des connexions

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

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

Les fonctions de transfert de données

Elles dépendent du langage utilisé.

Elles sont semblables aux fonctions d'écriture lecture de fichiers (`read` et `write` en mode connecté, `send` et `recv` en mode non connecté, etc).

La fonction de fermeture de socket

Une fonction souvent nommée **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 (à adapter au langages/systèmes

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, ...).

3 Mise en oeuvre des sockets en Python et Perl

Un serveur écrit en Python

```
#!/usr/bin/python
# coding: utf-8
# 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, à modifier selon
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, venant du client addr
print 'Connecte par', addr
while 1:
        data = conn.recv(1024)   # lecture dans le sock de connexion
        if not data: break
        conn.send('SRV'+data)          # il répete simplement, en prefiant par SRV
conn.close()
```

Un client en Perl (qui fonctionnera avec le serveur précédent)

```
# Un exemple de client écrit en Perl
# pour envoyer un message à un serveur connu
# sur le port 50007 de la machine locale (à modifier selon...)
#-----------------------------------------
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 `telnet`

Vous pouvez aussi vous servir de `telnet` comme client, afin de communiquer avec le serveur. Il suffit donc d'excuter dans un terminal : `telnet ardIP numPort`

Un (autre) client en Python

```
#!/usr/bin/python
# coding: utf-8
# 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 Serveur') # 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 quelques points d'entrée.

Des exemples en langage C - C. Attiogbé `http://pagesperso.lina.univ-nantes.fr/info/perso/permanents/attiogbe/mespages/MSFORMEL/IUT/cahier_sockets_c_plus.pdf`

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

Sockets en python `http://ilab.cs.byu.edu/python/socketmodule.html` et `http://ilab.cs.byu.edu/python/select/echoserver.html`

Eexmple en langage C `http://www.tenouk.com/Module41.html`

— Socket en Python, avec Select, `https://pymotw.com/2/select/`