Programmation *socket*

Thierry Vaira

BTS SN-IR Avignon

© v1.1 16 mars 2018

Notion de socket

Un socket représente une **interface de communication logicielle** :

- avec le système d'exploitation qui permet d'exploiter les services d'un protocole réseau,
- et par laquelle une application peut envoyer et recevoir des données.
- C'est donc un mécanisme de communication bidirectionelle entre processus (locaux et/ou distants).

Interface de programmation (API)

Un socket désigne aussi :

- un **ensemble normalisé de fonctions de communication** (lancé par l'université de Berkeley au début des années 1980) = API
- une interface de programmation qui est proposée dans quasiment tous les langages de programmation populaires (C, Java, C#, C++, ...) et
- répandue dans la plupart des systèmes d'exploitation (UNIX/Linux, ©Windows, ...).

La notion de socket a été introduite dans les distributions de Berkeley (un fameux système de type UNIX, dont beaucoup de distributions actuelles utilisent des morceaux de code), c'est la raison pour laquelle on parle parfois de ***sockets BSD*** *(Berkeley Software Distribution).*

Pré-requis

La mise en oeuvre de l'interface socket nécessite de connaître :

- L'architecture client/serveur
- L'adressage IP et les numéros de port
- Notions d'API (appels systèmes sous Unix) et de programmation en langage C
- Les protocoles TCP et UDP, les modes connecté et non connecté

Manuel du programmeur I

Le développeur utilisera donc concrètement une interface pour programmer une application TCP/IP grâce par exemple :

- à l'API **Socket BSD** sous Unix/Linux ou
- à l'API **WinSocket** sous Microsoft ©Windows

Les **pages `man`** principales sous Unix/Linux concernant la programmation réseau sont regroupées dans le chapitre 7 :

- socket(7) : interface de programmation des sockets
- packet(7) : interface par paquet au niveau périphérique
- raw(7) : sockets brutes (raw) IPv4 sous Linux
- ip(7) : implémentation Linux du protocole IPv4
- udp(7) : protocole UDP pour IPv4
- tcp(7) : protocole TCP

Manuel du programmeur II

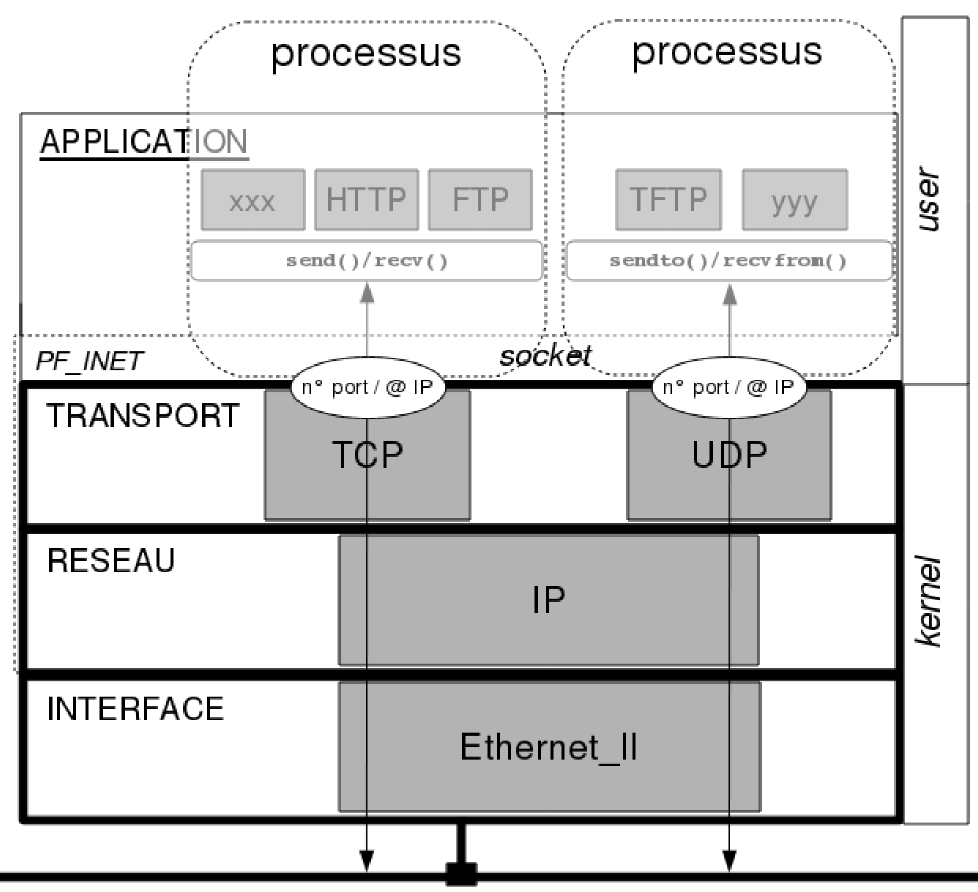
Pour Microsoft ©Windows, on pourra utiliser le service en ligne **MSDN** :

- Windows Socket 2 :
  http ://msdn.microsoft.com/en-us/library/ms740673(VS.85).aspx
- Les fonctions Socket :
  http ://msdn.microsoft.com/en-us/library/ms741394(VS.85).aspx

Ce document s'intéresse à l'aspect socket BSD d'Unix.

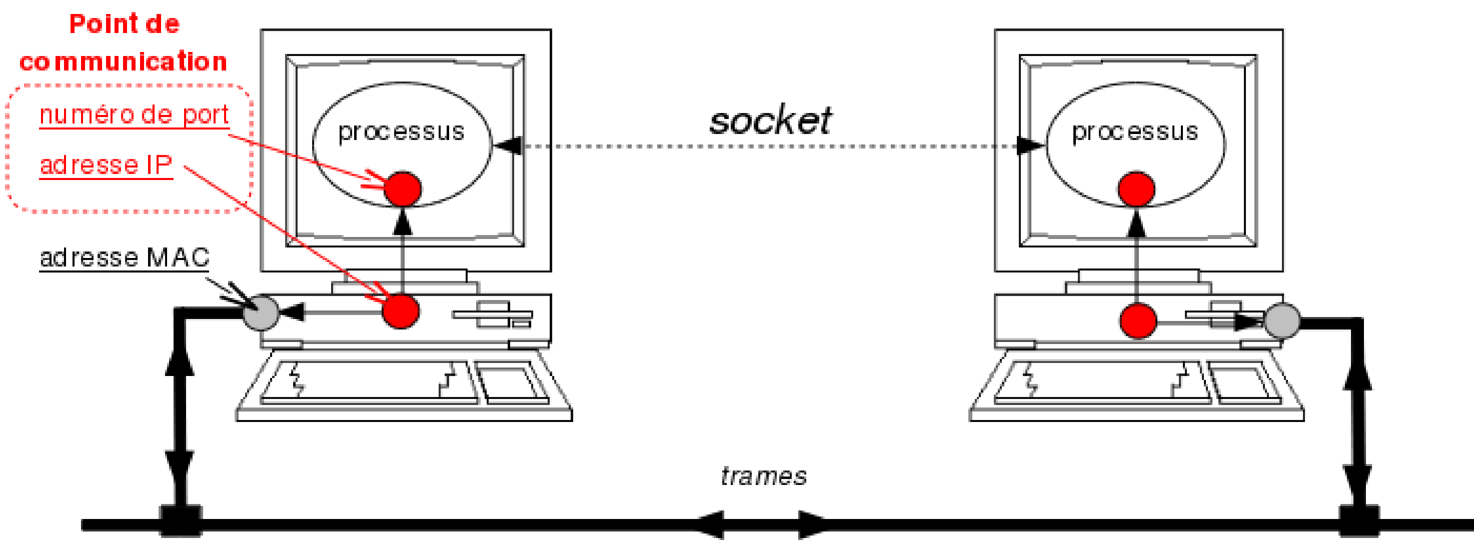
Modèle à couches I

- Un socket est communément représentée comme un point d'entrée initial au niveau TRANSPORT dans la pile de protocoles



Modèle à couches II

- Un socket est une interface de communication identifiée par une adresse IP et un numéro de port pour le modèle TCP/IP



Création d'un socket

Pour dialoguer, chaque processus devra préalablement créer un **socket de communication** en utilisant l'appel `socket()` et en indiquant :

- le **domaine** de communication : ceci sélectionne la famille de protocole à employer.
- le **type** de socket à utiliser pour le dialogue (mode connecté, non connecté ...).
- le **protocole** à utiliser sur la socket en fonction du type et du domaine de communication sélectionnés.

➟ La fonction renvoie un descripteur sur la socket créée en cas de réussite ou -1 en cas d'échec.

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

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

Caractéristiques d'un socket I

- le **domaine** sélectionne la famille de protocole à employer. Chaque famille possède son adressage. Par exemple pour les protocoles Internet IPv4, on utilisera le domaine `PF_INET` ou `AF_INET` et `AF_INET6` pour le protocole IPv6.
- le **type** de dialogue : Pour `PF_INET`, on aura par exemple le choix entre : `SOCK_STREAM` (qui correspond à un mode connecté donc `TCP` par défaut), `SOCK_DGRAM` (qui correspond à un mode non connecté donc `UDP`) ou `SOCK_RAW` (qui permet un accès direct aux protocoles de la couche Réseau comme IP, ICMP, ...).

Caractéristiques d'un socket II

- le numéro de **protocole** dépendra du domaine de communication et du type de socket. Normalement, il n'y a qu'un seul protocole par type de socket pour une famille donnée : la valeur `0` désignera le protocole par défaut (`SOCK_STREAM` $\rightarrow$ `TCP` et `SOCK_DGRAM` $\rightarrow$ `UDP`). Néanmoins, rien ne s'oppose à ce que plusieurs protocoles existent, auquel cas il est nécessaire de le spécifier (c'est la cas pour `SOCK_RAW` où il faudra préciser le protocole à utiliser : `IPPROTO_IP`, `IPPROTO_ICMP`, ...).

Exemple

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

int socket_tcp, socket_udp, socket_icmp; //descripteurs de sockets

// Un socket en mode connecte
socket_tcp = socket(PF_INET, SOCK_STREAM, 0); //par defaut TCP

// Un socket en mode non connecte
socket_udp = socket(PF_INET, SOCK_DGRAM, 0); //par defaut UDP

// Un socket en mode raw
socket_icmp = socket(PF_INET, SOCK_RAW, IPPROTO_ICMP); //on choisit ICMP
```

Adressage du point de communication

L'interface socket propose une structure d'adresse générique :

```
struct sockaddr
{
  unsigned short int sa_family; //au choix
  unsigned char sa_data[14];  //en fonction de la famille
};
```

➠ Pour le domaine PF_INET (IPv4), l'adressage du socket sera réalisée par une structure sockaddr_in :

```
struct in_addr { unsigned int s_addr; }; // une adresse Ipv4 (32 bits)

struct sockaddr_in
{
  unsigned short int sin_family; // <- PF_INET (IPv4)
  unsigned short int sin_port; // <- numero de port
  struct in_addr sin_addr; // <- adresse IPv4
  unsigned char sin_zero[8];  // ajustement pour etre compatible avec sockaddr
};
```

Network byte order

L'ordre des octets du réseau (*network*) est en fait *big-endian*. Il est donc possible qu'il soit différent de celui de la machine (*host*). Dans tous les cas, il est prudent de toujours convertir les informations à envoyer dans l'ordre du réseau et les lire dans l'ordre de la machine.

Pour les numéros de port sur 16 bits (`short int`) du champ `sin_port`, on utilisera :

- `htons()` pour convertir le numéro de port depuis l'ordre des octets de l'hôte vers celui du réseau.
- `ntohs()` pour convertir le numéro de port depuis l'ordre des octets du réseau vers celui de l'hôte.

Adresse IP

Dans l'en-tête d'un paquet IP, une adresse IPv4 est codée sur 32 bits (`unsigned int`) dans le champ `s_addr`. Comme on manipule souvent cette adresse avec chaîne de caractères en notation décimale pointée, il existe des fonctions de conversion :

- `inet_aton()` convertit une adresse IPv4 en décimal pointé vers sa forme binaire 32 bits dans l'ordre d'octet du réseau.
- `inet_ntoa()` convertit une adresse IPv4 sous sa forme binaire 32 bits dans l'ordre d'octet du réseau vers une chaîne de caractères en décimal pointé.

✍ Voir aussi les fonctions de traduction d'adresses et de services réseau `getaddrinfo()` et `getnameinfo()` si on désire utiliser par exemple un résolveur DNS.

Exemple

```
struct sockaddr_in adresseDistante;
socklen_t longueurAdresse;

// Obtient la longueur en octets de la structure sockaddr_in
longueurAdresse = sizeof(adresseDistante);

// Initialise la structure sockaddr_in
memset(&adresseDistante, 0x00, longueurAdresse);

// Renseigne la structure sockaddr_in avec les informations du serveur distant
adresseDistante.sin_family = PF_INET;

// On choisit le numero de port d'ecoute du serveur
adresseDistante.sin_port = htons(5000); // = IPPORT_USERRESERVED

// On choisit l'adresse IPv4 du serveur
inet_aton("192.168.52.2", &adresseDistante.sin_addr);
```

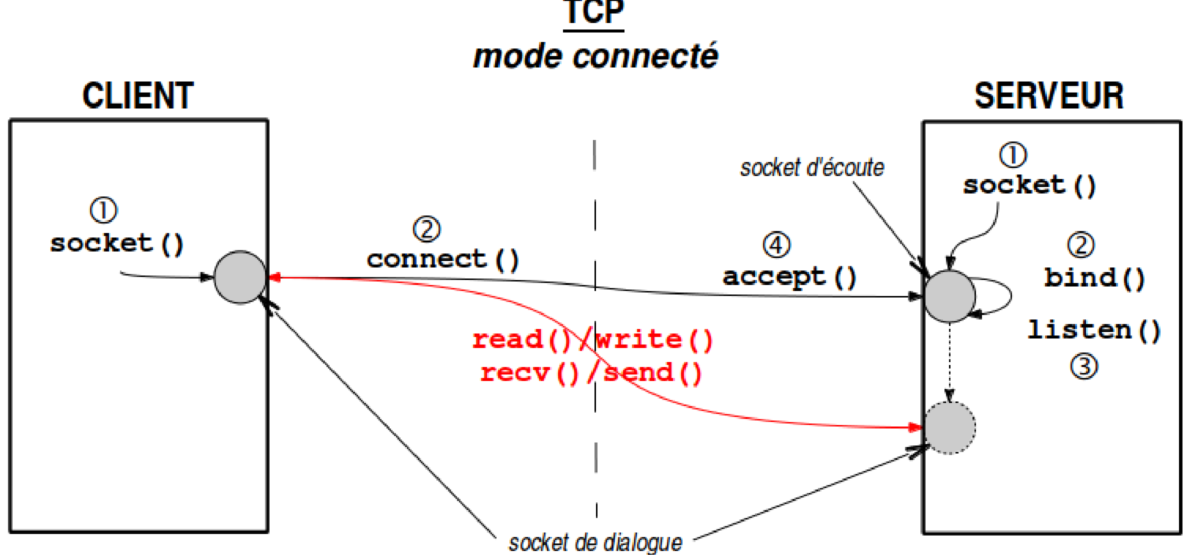
Communication TCP

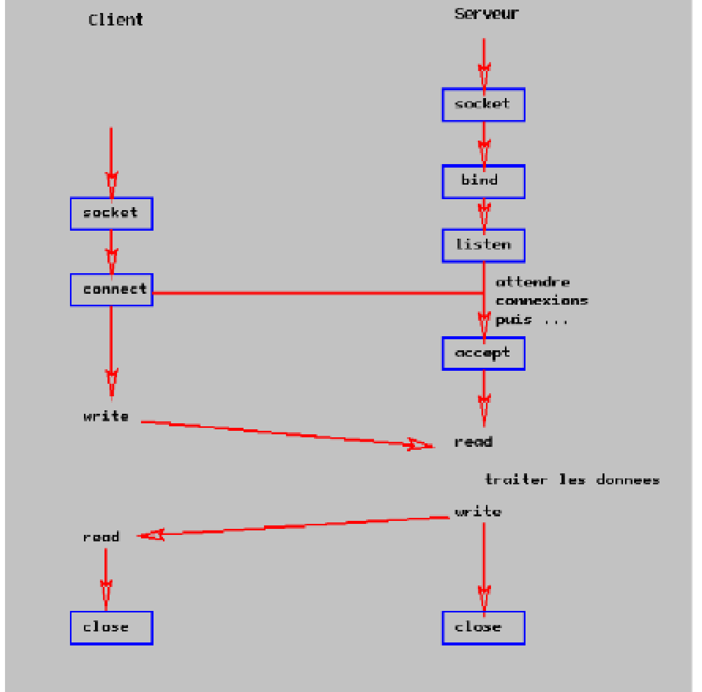
Dans une communication TCP, une connexion doit être établie entre le client et le serveur.

- La fonction `connect()` permet à un client de demander la connexion à un serveur.
- Le serveur utilisera préalablement : la fonction `bind()` pour définir l'adresse de sa socket et la fonction `listen()` pour informer l'OS qu'il est prêt à recevoir des demandes de connexion entrante.
- La fonction `accept()` permet à un serveur d'accepter une connexion.
- Une fois la connexion établie, les fonctions `send()` et `recv()` (ou tout simplement `write()` et `read()`) servent respectivement à envoyer et à recevoir des informations entre les deux processus.
- Une fonction `close()` (ou `shutdown()`) permet de terminer la connexion.

✍ L'appel `accept()` du serveur crée et retourne une nouvelle socket pour le dialogue.

Principe d'une communication TCP



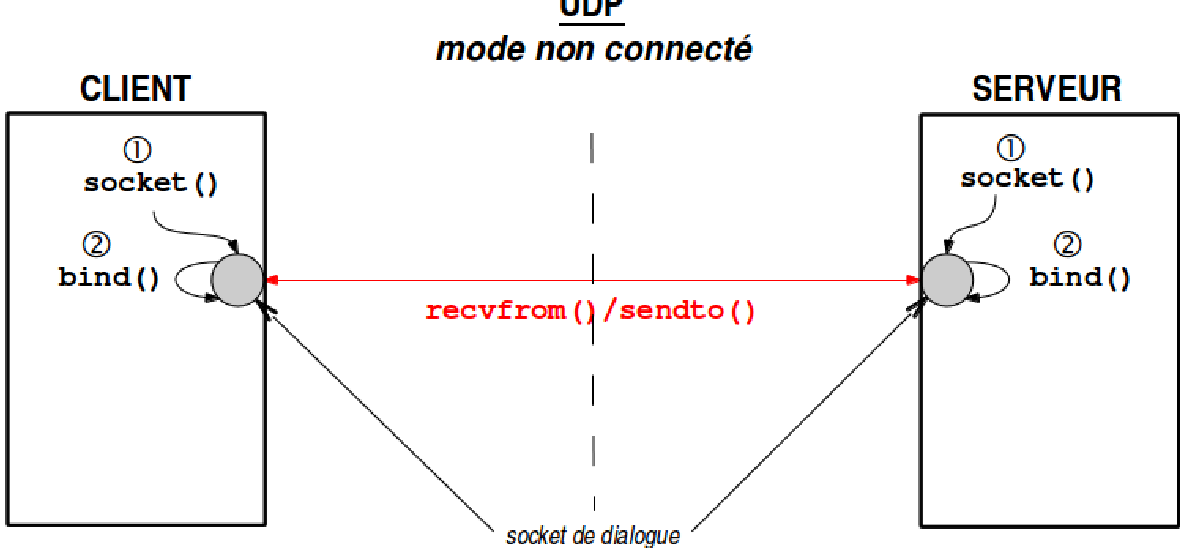


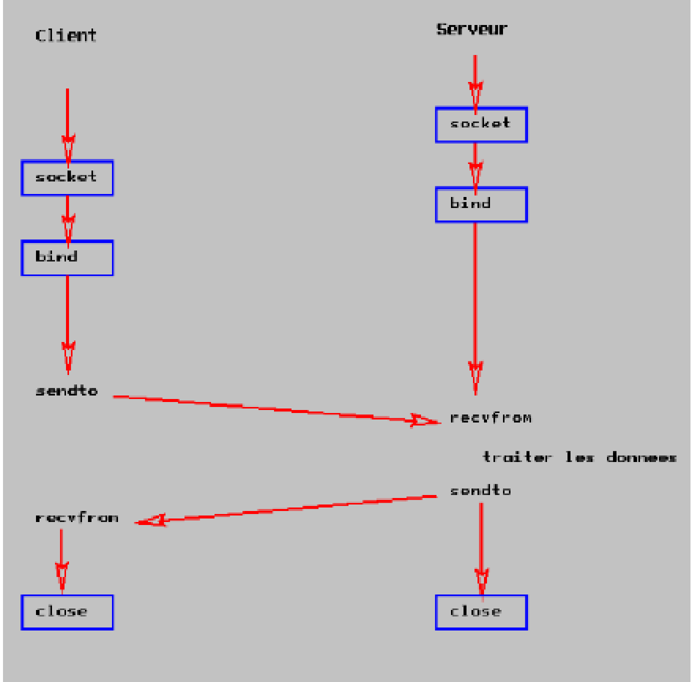
Communication UDP

Dans une communication bi-directionnelle UDP, il n'y a pas de connexion entre le client et le serveur. Le client reste le processus à l'initiative de l'échange et de la demande de service.

- La fonction `bind()` permet de définir l'adresse locale de la socket.
- Les fonctions d'échanges de données sur une socket UDP sont `recvfrom()` et `sendto()` qui permettent respectivement la réception et l'envoi d'octets sur un descripteur de socket en mode non-connecté.
- La fonction `close()` permet ici de libérer la ressource socket localement.

Principe d'une communication UDP





Mode bloquant et non bloquant I

- Certains appels systèmes utilisés ici sont « bloquants » : c'est-à-dire que le processus s'endort jusqu'à ce dont il a besoin soit disponible et le réveille.
- C'est le cas des appels `accept()` et des fonctions de réception de données `read()`, `recv()` et `recvfrom()`.
- C'est le choix par défaut fait par l'OS au moment de la création de la socket.
- Il est possible de placer la socket en mode non bloquant avec l'appel `fcntl()`.
- En déclarant une socket non bloquante, il faudra alors l'interroger périodiquement. Si vous essayez de lire sur une socket non bloquante et qu'il n'y a pas de donnee à lire, la fonction retournera `-1`.

La Salle Avignon

Mode bloquant et non bloquant II

```
#include <unistd.h>
#include <fcntl.h>
...

int s = socket(AF_INET, SOCK_STREAM, 0);
fcntl(s, F_SETFL, O_NONBLOCK);
```

Cas du serveur multi-clients

- Un client se connecte sur la socket d'écoute. L'appel `accept()` va retourner une nouvelle socket connectée au client qui servira de socket de dialogue. La socket d'écoute reste inchangée et peut donc servir à accepter des nouvelles connexions.
- Le principe est simple mais un problème apparaît pour le serveur : comment dialoguer avec le client connecté et continuer à attendre des nouvelles connexions ?
  - Solution n°1 : Le multiplexage synchrone d'E/S. On utilise les appels `select()` ou `poll()` pour surveiller et attendre un événement concernant une des sockets.
  - Solution n°2 : Le multi-tâche. On crée un processus fils (avec l'appel `fork()`) qui traite lui-même le dialogue et le processus père continue l'attente des demandes de connexion. Si le temps de création des processus fils devient une charge importante pour le système, on pourra envisager les solutions suivantes : création anticipée des processus, utilisation des threads.

La Salle Avignon