

Programmation Réseau : Socket TCP/UDP

© 2012 tv <tvaira@free.fr> - v.1.0

Sommaire

L'interface socket	2
Pré-requis	2
Définition	2
Manuel du pogrammeur	2
Modèle	3
Couche Transport	4
Numéro de ports	4
Caractéristiques des sockets	4
Programmation Socket TCP (Linux)	5
Programmation Socket UDP (Linux)	10
Programmation Socket TCP (Windows)	12
Questions de révision	13

L'interface socket

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

Définition

« *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).* »



Interface de programmation «*socket*» de Berkeley (1982) : la plus utilisée et intégrée dans le noyau

Il s'agit d'un modèle permettant la communication inter processus (IPC - *Inter Process Communication*) afin de permettre à divers processus de communiquer aussi bien sur une même machine qu'à travers un réseau TCP/IP. » [Wikipedia]



Socket : mécanisme de communication bidirectionnelle entre processus

Manuel du programmeur

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



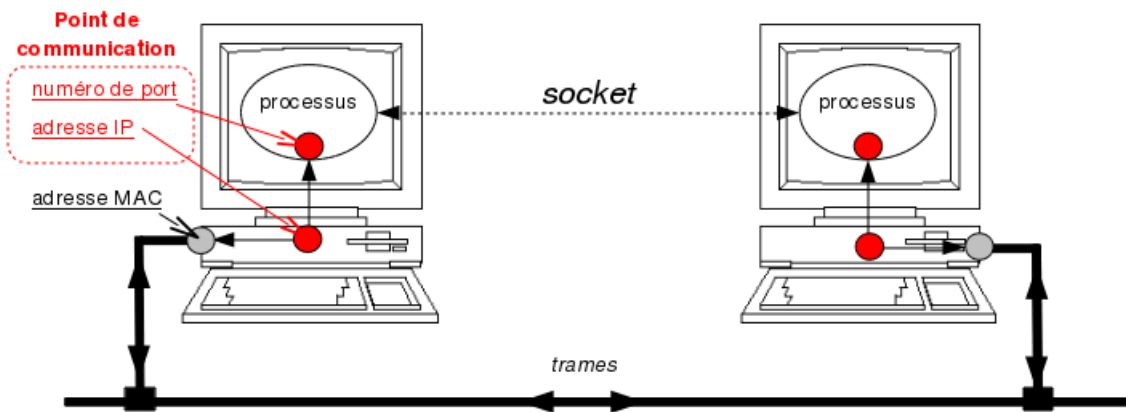
L'accès aux pages man se fera donc avec la commande `man`, par exemple : `man 7 socket`

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

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

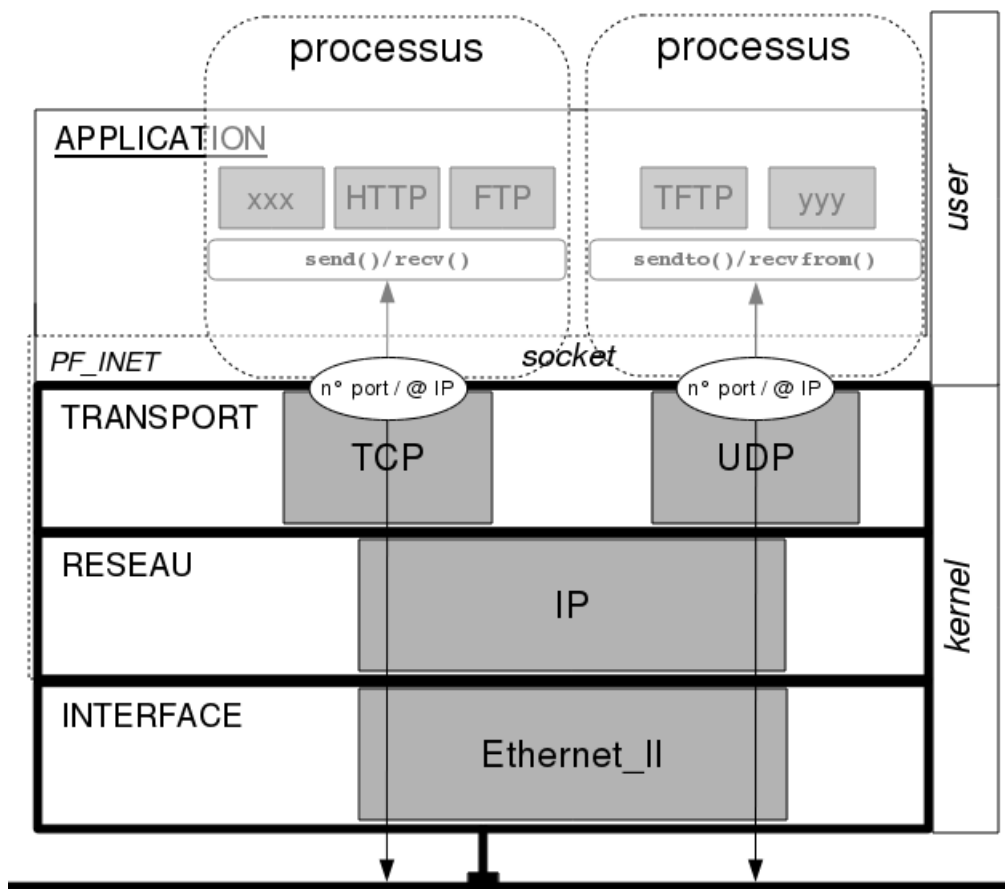
Modèle

Rappel : une socket est un point de communication par lequel un processus peut émettre et recevoir des données.



Ce point de communication devra être relié à une adresse IP et un numéro de port dans le cas des protocoles Internet.

Une socket est communément représentée comme un point d'entrée initial au niveau TRANSPORT du modèle à couches DoD dans la pile de protocole.



Exemple de processus TCP et UDP

Couche Transport

Rappel : la couche Transport est responsable du transport des messages complets de bout en bout (soit de processus à processus) au travers du réseau.

En programmation, si on utilise comme point d'entrée initial le niveau TRANSPORT, il faudra alors choisir un des deux protocoles de cette couche :

- **TCP** (*Transmission Control Protocol*) est un protocole de transport fiable, en **mode connecté** (RFC 793).
- **UDP** (*User Datagram Protocol*) est un protocole souvent décrit comme étant non-fiable, en **mode non-connecté** (RFC 768), mais plus rapide que TCP.

Numéro de ports

Rappel : un numéro de port sert à identifier un processus (l'application) en cours de communication par l'intermédiaire de son protocole de couche application (associé au service utilisé, exemple : 80 pour HTTP).



Pour chaque port, un numéro lui est attribué (codé sur 16 bits), ce qui implique qu'il existe un maximum de 65 536 ports (2^{16}) par machine et par protocoles TCP et UDP.

L'attribution des ports est faite par le système d'exploitation, sur demande d'une application. Ici, il faut distinguer les deux situations suivantes :

- cas d'un **processus client** : le numéro de port utilisé par le client sera envoyé au processus serveur. Dans ce cas, le processus client peut demander à ce que le système d'exploitation lui attribue n'importe quel port, à condition qu'il ne soit pas déjà attribué.
- cas d'un **processus serveur** : le numéro de port utilisé par le serveur doit être connu du processus client. Dans ce cas, le processus serveur doit demander un numéro de port précis au système d'exploitation qui vérifiera seulement si ce numéro n'est pas déjà attribué.



Une liste des ports dits réservés est disponible dans le fichier `/etc/services` sous Unix/Linux.

Caractéristiques des sockets

Rappel : les sockets compatibles BSD représentent une interface uniforme entre le processus utilisateur (user) et les piles de protocoles réseau dans le noyau (kernel) de l'OS.

Pour dialoguer, chaque processus devra préalablement créer une socket de communication en indiquant :

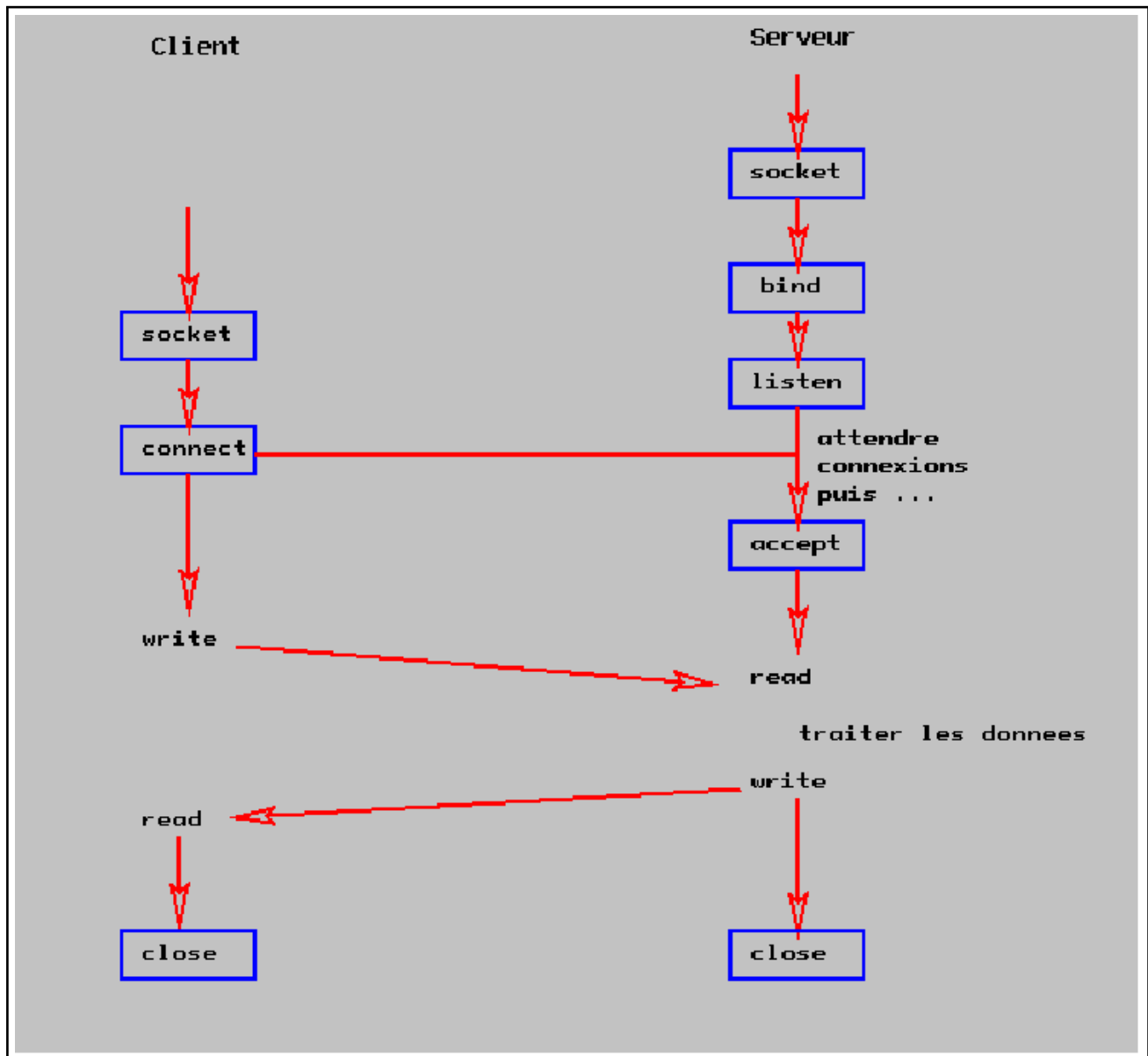
- le **domaine** de communication : ceci sélectionne la famille de protocole à employer. Il faut savoir que chaque famille possède son adressage. Par exemple pour les protocoles Internet IPv4, on utilisera le domaine `PF_INET` ou `AF_INET` et `AF_INET6` pour le protocole IPv6.
- le **type** de socket à utiliser pour le dialogue. Pour `PF_INET`, on aura 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, ...).
- le **protocole** à utiliser sur la socket. Le numéro de protocole dépend du domaine de communication et du type de la socket. Normalement, il n'y a qu'un seul protocole par type de socket pour une famille donnée (`SOCK_STREAM` → TCP et `SOCK_DGRAM` → 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).



Une socket appartient à une famille. Il existe plusieurs types de sockets. Chaque famille possède son adressage.

Programmation Socket TCP (Linux)

L'échange entre un client et un serveur TCP peut être schématisé de la manière suivante :



Les appels systèmes utilisés dans un échange TCP

Pour créer une socket, on utilisera l'appel système `socket()` :

NOM

`socket` - Créer un point de communication

SYNOPSIS

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

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

VALEUR RENVOYÉE

Cet appel système renvoie un descripteur référant la socket créée s'il réussit. S'il échoue, il renvoie -1 et `errno` contient le code d'erreur.

Extrait de la page man de l'appel système socket

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

int main()
{
    int descripteurSocket;

    // crée un socket de communication
    descripteurSocket = socket(PF_INET, SOCK_STREAM, 0); /* 0 indique que l'on utilisera le
        protocole par défaut associé à SOCK_STREAM soit TCP */

    // échec ?
    if(descripteurSocket < 0)
    {
        perror("socket"); // Affiche le message d'erreur
        exit(-1); // On sort en indiquant un code erreur
    }

    // ...

    // ferme la ressource avant de quitter
    close(descripteurSocket);

    return 0;
}
```



Pour le paramètre `protocol`, on a utilisé la valeur 0 (voir commentaire). On aurait pu préciser le protocole TCP de la manière suivante : `IPPROTO_TCP`.

Ensuite les appels `connect()` côté **client** et `bind()/accept()` côté **serveur** sont basés sur l'**utilisation d'une structure d'adresse pour identifier les bouts de communications**.

Il faut savoir que l'adressage des processus (local et distant) dépend du **domaine** de communication (cad la famille de protocole employée). Ici, nous avons choisi le domaine `PF_INET` pour les protocoles Internet IPv4.

Dans cette famille, un processus sera identifié par :

- une adresse IPv4
- un numéro de port

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
};
```

La structure générique sockaddr

Et le domaine PF_INET utilise une structure compatible :

```
// Remarque : ces structures sont déclarées dans <netinet/in.h>
```

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

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

La structure compatible sockaddr_in pour PF_INET

Il suffit donc d'initialiser une structure `sockaddr_in` :

- pour le client : avec les informations distantes du serveur (adresse IPv4 et numéro de port) pour l'appel `connect()`.
- pour le serveur : avec les informations locales du serveur (adresse IPv4 et numéro de port) pour l'appel `bind()`.

Pour écrire ces informations dans la structure d'adresse, il faudra utiliser :

- `inet_aton()` pour convertir une **adresse IP** depuis la notation IPv4 décimale pointée vers une forme binaire (dans l'ordre d'octet du réseau)
- `htons()` pour convertir le **numéro de port** (sur 16 bits) depuis l'ordre des octets de l'hôte vers celui du réseau.



L'ordre des octets du réseau est en fait *big-endian*. Il est donc plus prudent d'appeler des fonctions qui respectent cet ordre pour coder des informations dans les en-têtes des protocoles réseaux.

Exemple pour le client :

```
struct sockaddr_in pointDeRencontreDistant;
socklen_t longueurAdresse;

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

// Initialise à 0 la structure sockaddr_in
memset(&pointDeRencontreDistant, 0x00, longueurAdresse);
// Renseigne la structure sockaddr_in avec les informations du serveur distant
```

```

pointDeRencontreDistant.sin_family = PF_INET;
// On choisit le numéro de port d'écoute du serveur
pointDeRencontreDistant.sin_port = htons(IPPORT_USERRESERVED); // = 5000
// On choisit l'adresse IPv4 du serveur
inet_aton("192.168.52.2", &pointDeRencontreDistant.sin_addr); // à modifier selon ses
    besoins

```

On rappelle qu'une communication TCP est bidirectionnelle *full duplex* et orientée flux d'octets. Il faut donc des fonctions pour écrire (envoyer) et lire (recevoir) des octets dans la socket.

Les fonctions d'échanges de données sur une socket TCP sont :

- `read()` et `write()` qui permettent la réception et l'envoi d'octets sur un descripteur de socket
- `recv()` et `send()` qui permettent la réception et l'envoi d'octets sur un descripteur de socket avec un paramètre `flags`



Les appels `recv()` et `send()` sont spécifiques aux sockets en mode connecté. La seule différence avec `read()` et `write()` est la présence de `flags` (cf. man 2 `send`).

Évidemment, un serveur TCP a lui aussi besoin de créer une socket `SOCK_STREAM` dans le domaine `PF_INET`. Mis à part cela, le code source d'un serveur TCP basique est très différent d'un client TCP dans le principe.

On rappelle qu'un serveur TCP attend des demandes de connexion en provenance de processus client. Le processus client doit connaître au moment de la connexion le numéro de port d'écoute du serveur.

Pour mettre en oeuvre cela, le serveur va utiliser l'appel système `bind()` qui va lui permettre de lier sa socket d'écoute à une interface et à un numéro de port local à sa machine.

Il suffit donc d'initialiser une structure `sockaddr_in` avec les informations **locales du serveur** (adresse IPv4 et numéro de port).



Normalement il faudrait indiquer l'adresse IPv4 de l'interface locale du serveur qui acceptera les demandes de connexions. Il est ici possible de préciser avec `INADDR_ANY` que toutes les interfaces locales du serveur accepteront les demandes de connexion des clients.

```

struct sockaddr_in pointDeRencontreLocal;
socklen_t longueurAdresse;

// On prépare l'adresse d'attachement locale
longueurAdresse = sizeof(struct sockaddr_in);
memset(&pointDeRencontreLocal, 0x00, longueurAdresse);
pointDeRencontreLocal.sin_family = PF_INET;
pointDeRencontreLocal.sin_addr.s_addr = htonl(INADDR_ANY); // toutes les interfaces locales
    disponibles
pointDeRencontreLocal.sin_port = htons(PORT); // = 5000

```

Maintenant que le serveur a créé et attaché une socket d'écoute, il doit la placer en attente passive, c'est-à-dire capable d'accepter les demandes de connexion des processus clients.

Pour cela, on va utiliser l'appel système `listen()` :

NOM

`listen` - Attendre des connexions sur une socket

SYNOPSIS

```
#include <sys/types.h>      /* Voir NOTES */
#include <sys/socket.h>

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

DESCRIPTION

listen() marque la socket référencée par sockfd comme une socket passive, c'est-à-dire comme une socket qui sera utilisée pour accepter les demandes de connexions entrantes en utilisant accept().

L'argument sockfd est un descripteur de fichier qui fait référence à une socket de type SOCK_STREAM.

L'argument backlog définit une longueur maximale jusqu'à laquelle la file des connexions en attente pour sockfd peut croître. Si une nouvelle connexion arrive alors que la file est pleine, le client reçoit une erreur indiquant ECONNREFUSED, ou, si le protocole sous-jacent supporte les retransmissions, la requête peut être ignorée afin qu'un nouvel essai réussisse.

Extrait de la page man de l'appel système listen



Si la file est pleine, le serveur sera dans une situation de DOS (*Deny Of Service*) car il ne peut plus traiter les nouvelles demandes de connexion.

Cette étape est cruciale pour le serveur. Il lui faut maintenant accepter les demandes de connexion en provenance des processus client.

Pour cela, il va utiliser l'appel système accept().

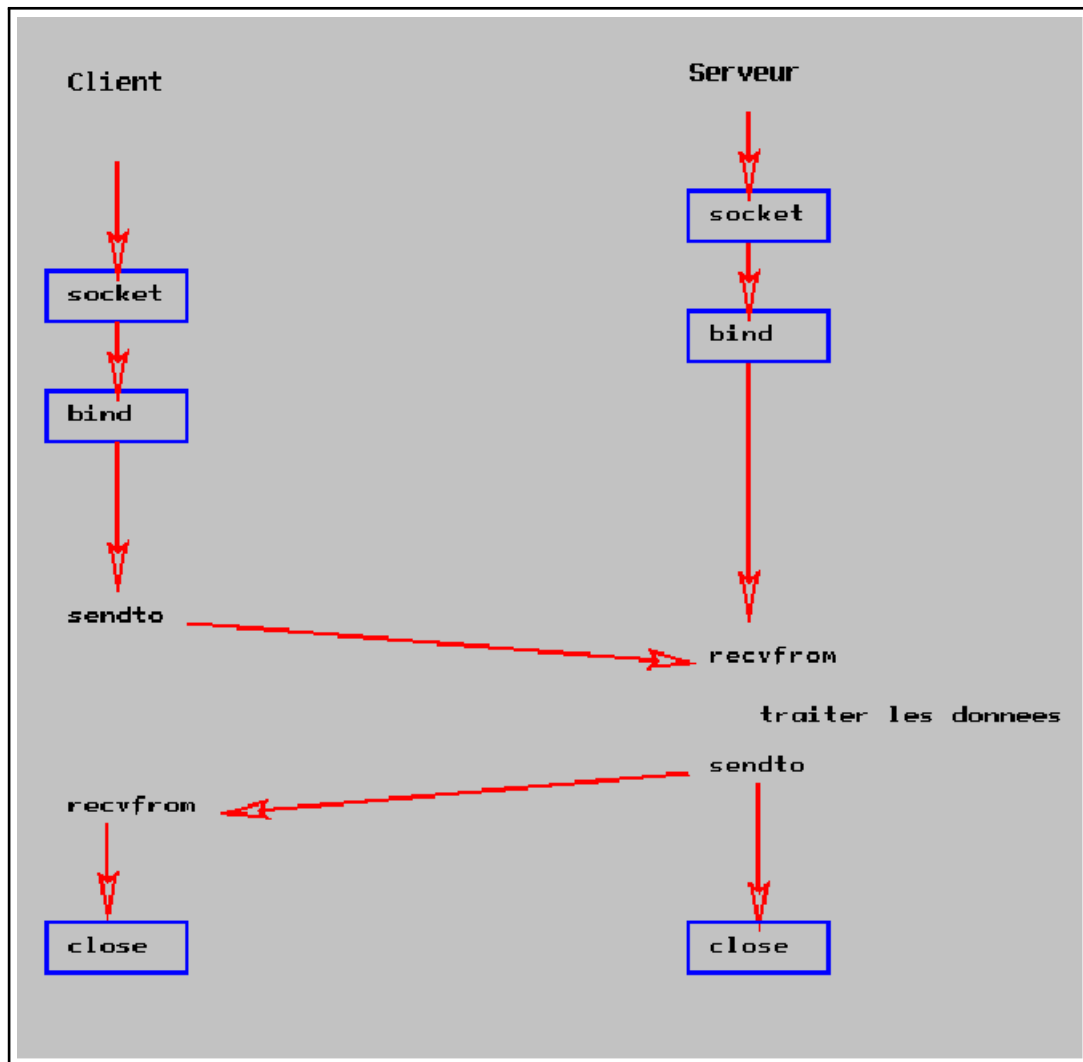


Explication : imaginons qu'un client se connecte à notre socket d'écoute. L'appel accept() va retourner une nouvelle socket connectée au client qui servira de socket de dialogue. La socket d'écoute reste inchangée et peut donc servir à accepter des nouvelles connexions.

Le principe est simple mais un problème apparaît pour le serveur : comment dialoguer avec le client connecté et continuer à attendre des nouvelles connexions ? Il y a plusieurs solutions à ce problème notamment la programmation multi-tâche car ici le serveur a besoin de paralléliser plusieurs traitements.

Programmation Socket UDP (Linux)

L'échange entre un client et un serveur UDP peut être schématisé de la manière suivante :



Les appels systèmes utilisés dans un échange UDP

Pour créer une socket UDP, on utilisera aussi l'appel système `socket()` :

```
int main()
{
    int descripteurSocket;

    // Crée un socket de communication
    descripteurSocket = socket(PF_INET, SOCK_DGRAM, 0); /* 0 indique que l'on utilisera le
        protocole par défaut associé à SOCK_DGRAM soit UDP */

    // Teste la valeur renvoyée par l'appel système socket()
    if(descripteurSocket < 0) /* échec ? */
    {
        perror("socket"); // Affiche le message d'erreur
        exit(-1); // On sort en indiquant un code erreur
    }

    //...
```

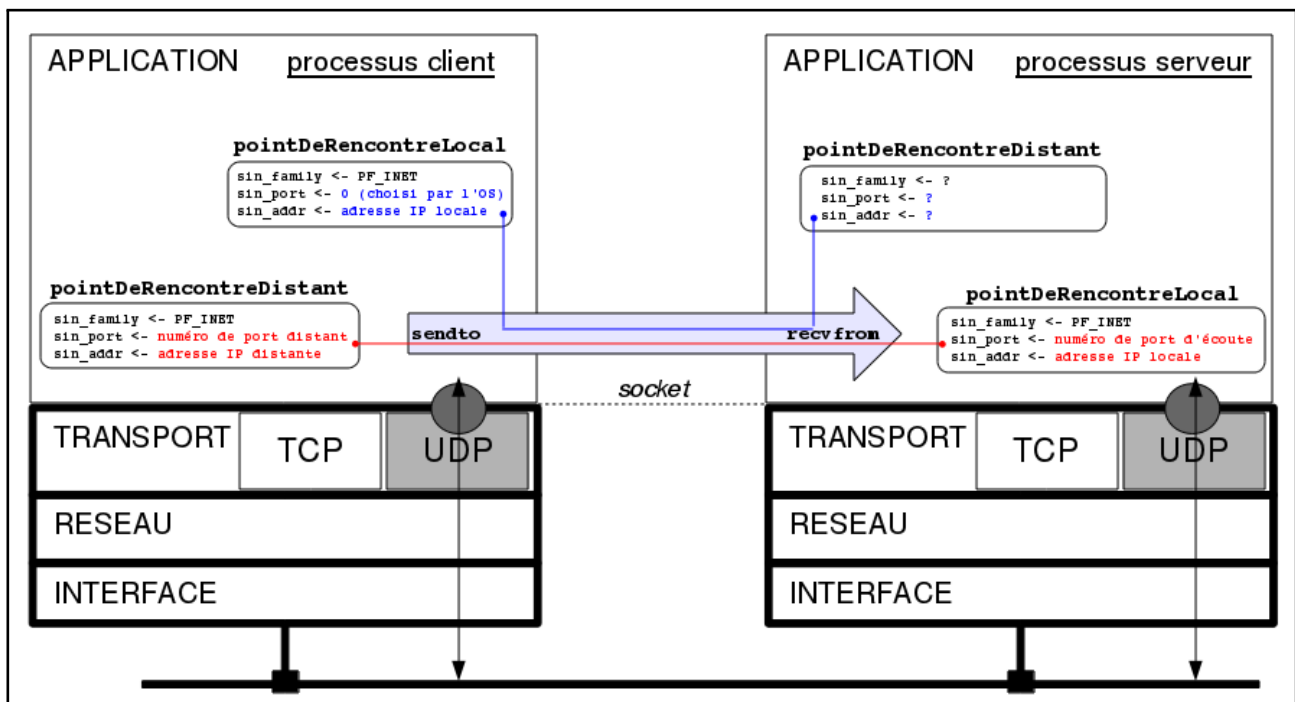
```
// On ferme la ressource avant de quitter
close(descripteurSocket);

return 0;
}
```



Pour le paramètre `protocol`, on a utilisé la valeur 0 (voir commentaire). On aurait pu préciser le protocole UDP de la manière suivante : `IPPROTO_UDP`.

Maintenant que nous avons créé une socket UDP, le client pourrait déjà communiquer avec un serveur UDP car nous utilisons un mode non-connecté.



Un échange en UDP

On va tout d'abord attacher cette socket à une interface et à un numéro de port local de sa machine en utilisant l'appel système `bind()`. Cela revient à créer un **point de rencontre local pour le client**.

Les fonctions d'échanges de données sur une socket UDP sont `recvfrom()` et `sendto()` qui permettent la réception et l'envoi d'octets sur un descripteur de socket en mode non-connecté.



Les appels `recvfrom()` et `sendto()` sont spécifiques aux sockets en mode non-connecté. Ils utiliseront en argument une structure `sockaddr_in` pour `PF_INET`.

Le code source d'un serveur UDP basique est très similaire à celui d'un client UDP. Évidemment, un serveur UDP a lui aussi besoin de créer une socket `SOCK_DGRAM` dans le domaine `PF_INET`. Puis, il doit utiliser l'appel système `bind()` pour lier sa socket d'écoute à une interface et à un numéro de port local à sa machine car le processus client doit connaître et fournir au moment de l'échange ces informations.

Il suffit donc d'initialiser une structure `sockaddr_in` avec les informations **locales du serveur** (adresse IPv4 et numéro de port).



Il est ici possible de préciser avec `INADDR_ANY` que toutes les interfaces locales du serveur accepteront les échanges des clients.

Programmation Socket TCP (Windows)



Winsock (*WIND*ows *SOCK*et) est une bibliothèque logicielle pour Windows dont le but est d'implémenter une interface de programmation inspirée des sockets BSD. Son développement date de 1991 (Microsoft n'a pas implémenté Winsock 1.0.). Il y a peu de différences avec les sockets BSD, mais Winsock fournit des fonctions additionnelles pour être conforme au modèle de programmation Windows, par exemple les fonctions `WSAGetLastError()`, `WSAStartup()` et `WSACleanup()`.

Sous Windows, il faut tout d'abord initialiser avec `WSAStartup()` l'utilisation de la DLL Winsock par le processus. De la même manière, il faudra terminer son utilisation proprement avec `WSACleanup()`.

Pour créer une socket, on utilisera l'appel `socket()`. On commence par consulter la documentation associée à cet appel : [http://msdn.microsoft.com/en-us/library/ms740506\(v=vs.85\).aspx](http://msdn.microsoft.com/en-us/library/ms740506(v=vs.85).aspx).

The socket function creates a socket that is bound to a specific transport service provider.

```
SOCKET WSAAPI socket(
    _In_ int af,
    _In_ int type,
    _In_ int protocol
);
```

...

Extrait de la documentation de l'appel socket

```
#include <winsock2.h>
#include <ws2tcpip.h>
#include <stdio.h>
#include <stdlib.h>

int main()
{
    WSADATA WSAData; // variable initialisée par WSAStartup

    WSAStartup(MAKEWORD(2,0), &WSAData); // indique la version utilisée, ici 2.0

    //--- Début de l'étape n°1 :
    SOCKET descripteurSocket;
    int iResult;

    // Crée un socket de communication
    descripteurSocket = socket(AF_INET, SOCK_STREAM, 0); /* 0 indique que l'on utilisera le
        protocole par défaut associé à SOCK_STREAM soit TCP */

    if (descripteurSocket == INVALID_SOCKET)
    {
        printf("Erreur creation socket : %d\n", WSAGetLastError());
        WSACleanup();
        return 1;
    }
}
```

```

}


// ...


// On ferme la ressource avant de quitter
iResult = closesocket(descripteurSocket);
if (iResult == SOCKET_ERROR)
{
    printf("Erreur fermeture socket : %d\n", WSAGetLastError());
    WSACleanup();
    return 1;
}

WSACleanup(); // termine l'utilisation

return 0;
}

```

 Dans cet exemple, on utilise la version 2 de Winsock (`winsock2.h` et `ws2_32.lib`). Vous pouvez aussi utiliser la version 1 (`winsock.h` et `wsock32.lib`). Avec les compilateurs type GCC, il faudra ajouter `-lws2_32` à l'édition des liens.

 La fonction `WSAGetLastError()` retourne seulement un code d'erreur. L'ensemble des code d'erreurs sont déclarés dans le fichier d'en-tête `winsock2.h`. Par exemple ici, le code d'erreur 10061 correspond à l'étiquette `WSAECONNREFUSED` (connexion refusée). Pour obtenir le message associé à un code d'erreur, il faut utiliser la fonction `FormatMessageW()`.

```

wchar_t *s = NULL;
FormatMessageW(FORMAT_MESSAGE_ALLOCATE_BUFFER | FORMAT_MESSAGE_FROM_SYSTEM |
    FORMAT_MESSAGE_IGNORE_INSERTS, NULL, WSAGetLastError(), MAKELANGID(LANG_NEUTRAL,
    SUBLANG_DEFAULT), (LPWSTR)&s, 0, NULL);
fprintf(stderr, "%S\n", s);
LocalFree(s);

```

Affichage du message associé à un code d'erreur

Questions de révision

L'idée de base des questions de révision est de vous donner une chance de voir si vous avez identifié et compris les points clés de ce TP.

Question 1. Qu'est-ce qu'une *socket* ?

Question 2. Quelles sont les trois caractéristiques d'une *socket* ?

Question 3. Quelles sont les deux informations qui définissent un point de communication en IPv4 ?

Question 4. Comment le serveur connaît-il le port utilisé par le client ?

Question 5. Comment le client connaît-il le port utilisé par le serveur ?

Question 6. À quelle couche du modèle DoD est reliée l'interface de programmation *socket* ?

Question 7. Quel protocole de niveau Transport permet d'établir une communication en mode connecté ?

Question 8. Quel protocole de niveau Transport permet d'établir une communication en mode non-connecté ?

Question 9. Quel est l'ordre des octets en réseau ?

Question 10. À quels protocoles correspond le domaine PF_INET ? Est-ce le seul utilisable avec l'interface socket ? En citer au moins un autre.