

TP Développement Réseau : Socket TCP (Linux)

© 2012 tv <tvaira@free.fr> - v.1.0

Sommaire

L'interface socket	2
Pré-requis	2
Définition	2
Manuel du programmeur	2
Modèle	3
Couche Transport	4
Numéro de ports	4
Caractéristiques des sockets	4
Manipulations	5
Objectifs	5
Étape n°1 : création de la socket (côté client)	5
Étape n°2 : connexion au serveur	6
Étape n°3 : vérification du bon fonctionnement de la connexion	9
Étape n°4 : échange des données	9
Étape n°5 : réalisation d'un serveur TCP	13
Étape n°6 : mise en attente des connexions	16
Étape n°7 : accepter les demandes connexions	18
Bilan	23
Questions de révision	24

Les objectifs de ce tp sont de mettre en oeuvre la programmation réseau en utilisant l'interface socket pour le mode connecté (TCP) sous Linux.

Remarque : les TP ont pour but d'établir ou de renforcer vos compétences pratiques. Vous pouvez penser que vous comprenez tout ce que vous lisez ou tout ce que vous a dit votre enseignant mais la répétition et la pratique sont nécessaires pour développer des compétences en programmation. Ceci est comparable au sport ou à la musique ou à tout autre métier demandant un long entraînement pour acquérir l'habileté nécessaire. Imaginez quelqu'un qui voudrait disputer une compétition dans l'un de ces domaines sans pratique régulière. Vous savez bien quel serait le résultat.

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).* »



Intégration d'IP dans Unix BSD (1981)

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



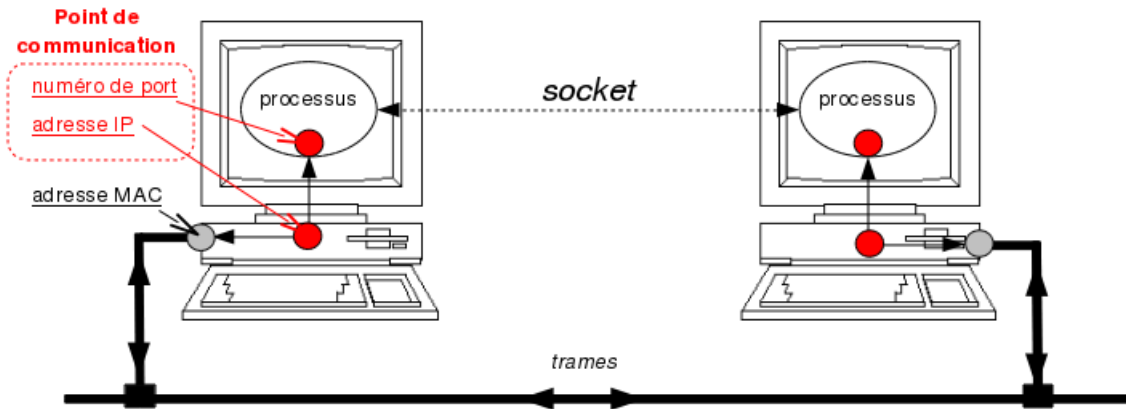
L'accès aux pages man se fera donc avec la commande `man`, par exemple : `man 7 socket`

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)

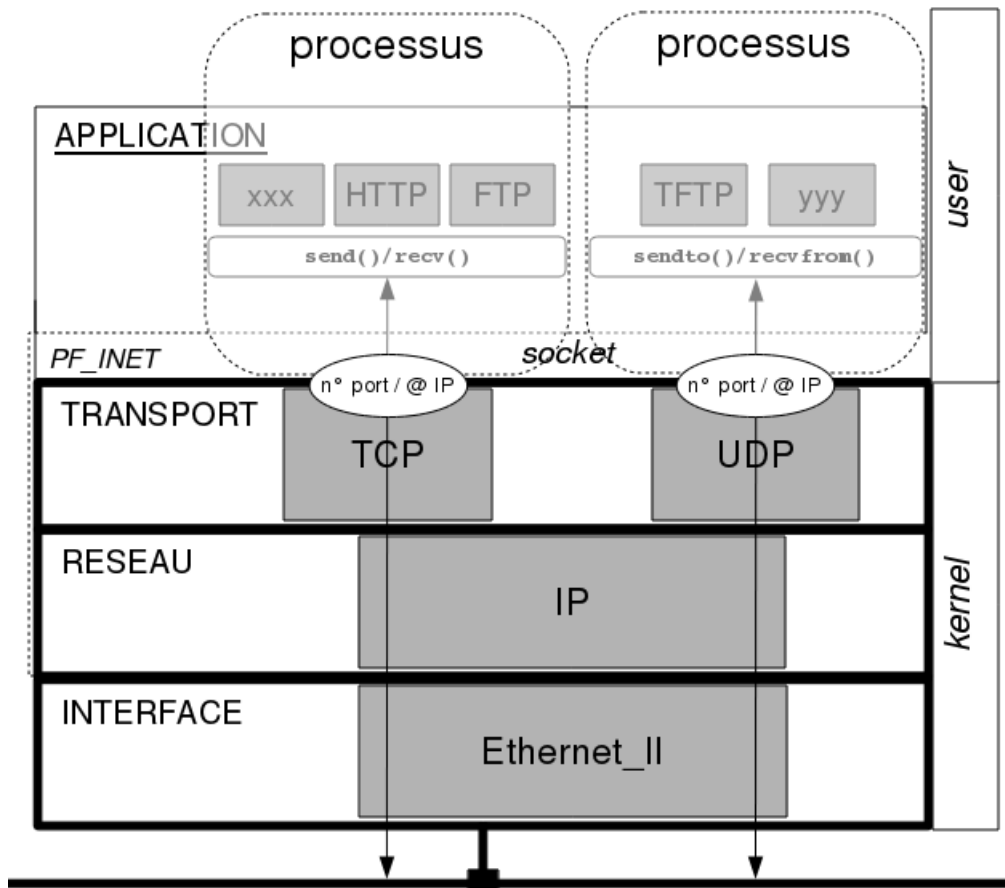
Modèle

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`.
- 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), `SOCK_DGRAM` (qui correspond à un mode non connecté) 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.

Manipulations

Objectifs

L'objectif de cette partie est la mise en oeuvre d'une communication client/serveur en utilisant une socket TCP sous Unix/Linux.

Étape n°1 : création de la socket (côté client)

Pour créer une socket, on utilisera l'appel système `socket()`. On commence par consulter la page du manuel associée à cet appel :

```
$ man 2 socket
```

```
SOCKET(2)                               Manuel du programmeur Linux                               SOCKET(2)
```

```
NOM
```

```
socket - Créer un point de communication
```

```
SYNOPSIS
```

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

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

```
DESCRIPTION
```

```
socket() crée un point de communication, et renvoie un descripteur.
...
```

```
VALEUR RENVOYÉE
```

```
Cet appel système renvoie un descripteur référençant 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

À l'aide d'un éditeur de texte (`vi`, `vim`, `emacs`, `kwrite`, `kate`, `gedit`, ... sous Linux), tapez (à la main, pas de copier/coller, histoire de bien mémoriser !) le programme suivant dans un fichier que vous nommerez "clientTCP-1.c" :

```
#include <stdio.h>
#include <stdlib.h> /* pour exit */
#include <sys/types.h>
#include <sys/socket.h>

int main()
{
    int descripteurSocket;
```

```
//--- Début de l'étape n°1 :
// 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 */

// 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
}

//--Fin de l'étape n°1 !
printf("Socket créée avec succès ! (%d)\n", descripteurSocket);

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

return 0;
}
```

Un client TCP en C (itération 1)



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`.

Étape n°2 : connexion au serveur

Maintenant que nous avons créé une socket TCP, il faut la connecter au processus serveur distant.

Pour cela, on va utiliser l'appel système `connect()`. On consulte la page du manuel associée à cet appel :

```
$ man 2 connect
```

```
CONNECT(2)                                Manuel du programmeur Linux                                CONNECT(2)
```

NOM

`connect` - Débuter une connexion sur une socket

SYNOPSIS

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

int connect(int sockfd, const struct sockaddr *serv_addr,
            socklen_t addrlen);
```

DESCRIPTION

L'appel système `connect()` connecte la socket référencée par le descripteur de fichier `sockfd` à l'adresse indiquée par `serv_addr`. ...

VALEUR RENVOYÉE

`connect()` renvoie 0 s'il réussit, ou -1 s'il échoue, auquel cas `errno` contient le code d'erreur. ...

Extrait de la page man de l'appel système connect

On rappelle que l'adressage du processus 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` avec les informations distantes du serveur (adresse IPv4 et numéro de port).

Pour écrire ces informations dans la structure d'adresse, il nous 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.

Éditer le programme suivant dans un fichier que vous nommerez "clientTCP-2.c" :

```
#include <stdio.h>
#include <stdlib.h> /* pour exit */
#include <sys/types.h>
#include <sys/socket.h>
#include <string.h> /* pour memset */
#include <netinet/in.h> /* pour struct sockaddr_in */
#include <arpa/inet.h> /* pour htons et inet_aton */

int main()
```

```
{
    int descripteurSocket;
    struct sockaddr_in pointDeRencontreDistant;
    socklen_t longueurAdresse;

    // Crée un socket de communication
    descripteurSocket = socket(PF_INET, SOCK_STREAM, 0);

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

    printf("Socket créée avec succès ! (%d)\n", descripteurSocket);

    //--- Début de l'étape n°2 :
    // 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

    // Débute la connexion vers le processus serveur distant
    if((connect(descripteurSocket, (struct sockaddr *)&pointDeRencontreDistant,
        longueurAdresse)) == -1)
    {
        perror("connect"); // Affiche le message d'erreur
        close(descripteurSocket); // On ferme la ressource avant de quitter
        exit(-2); // On sort en indiquant un code erreur
    }

    //--- Fin de l'étape n°2 !
    printf("Connexion au serveur réussie avec succès !\n");

    // On ferme la ressource avant de quitter
    close(descripteurSocket);
    return 0;
}
```

Un client TCP en C (itération 2)

Si vous testez ce client, vous risquez d'obtenir :

```
$ ./clientTCP-2
Socket créée avec succès ! (3)
connect: Connection refused
```

Ceci peut s'expliquer tout simplement parce qu'il n'y a pas de processus serveur à cette adresse!

Étape n°3 : vérification du bon fonctionnement de la connexion

Pour tester notre client, il nous faut un serveur ! Pour cela, on va utiliser l'outil réseau `netcat` en mode serveur (-l) sur le port 5000 (-p 5000) :

```
$ nc -l -p 5000
```

Puis :

```
$ ./clientTCP-2
Socket créée avec succès ! (3)
Connexion au serveur réussie avec succès !
```



Dans l'architecture client/serveur, on rappelle que c'est le client qui a l'initiative de l'échange. Il faut donc que le serveur soit en écoute avant que le client fasse sa demande.

Étape n°4 : échange des données

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



Normalement les octets envoyés ou reçus respectent un protocole de couche APPLICATION. Ici, pour les tests, notre couche APPLICATION sera vide ! C'est-à-dire que les données envoyées et reçues ne respecteront aucun protocole et ce seront de simples caractères ASCII.

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`).

Faire communiquer deux processus sans aucun protocole de couche APPLICATION est tout de même difficile ! On va simplement fixer les règles d'échange suivantes :

- le client envoie en premier une chaîne de caractères
- et le serveur lui répondra "ok"

Éditer le programme suivant dans un fichier que vous nommerez "`clientTCP-3.c`" :

```
#include <stdio.h>
#include <stdlib.h> /* pour exit */
#include <sys/types.h>
#include <sys/socket.h>
#include <string.h> /* pour memset */
#include <netinet/in.h> /* pour struct sockaddr_in */
#include <arpa/inet.h> /* pour htons et inet_aton */

#define LG_MESSAGE 256
```

```
int main()
{
    int descripteurSocket;
    struct sockaddr_in pointDeRencontreDistant;
    socklen_t longueurAdresse;
    char messageEnvoi[LG_MESSAGE]; /* le message de la couche Application ! */
    char messageRecu[LG_MESSAGE]; /* le message de la couche Application ! */
    int ecrits, lus; /* nb d'octets ecrits et lus */
    int retour;

    // 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 */

    // 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
    }

    printf("Socket créée avec succès ! (%d)\n", descripteurSocket);

    // 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

    // Débute la connexion vers le processus serveur distant
    if((connect(descripteurSocket, (struct sockaddr *)&pointDeRencontreDistant,
        longueurAdresse)) == -1)
    {
        perror("connect"); // Affiche le message d'erreur
        close(descripteurSocket); // On ferme la ressource avant de quitter
        exit(-2); // On sort en indiquant un code erreur
    }

    printf("Connexion au serveur réussie avec succès !\n");

    //--- Début de l'étape n°4 :
    // Initialise à 0 les messages
    memset(messageEnvoi, 0x00, LG_MESSAGE*sizeof(char));
    memset(messageRecu, 0x00, LG_MESSAGE*sizeof(char));

    // Envoie un message au serveur
    sprintf(messageEnvoi, "Hello world !\n");
```

```
ecrits = write(descripteurSocket, messageEnvoi, strlen(messageEnvoi)); // message à
    TAILLE variable
switch(ecrits)
{
    case -1 : /* une erreur ! */
        perror("write");
        close(descripteurSocket);
        exit(-3);
    case 0 : /* la socket est fermée */
        fprintf(stderr, "La socket a été fermée par le serveur !\n\n");
        close(descripteurSocket);
        return 0;
    default: /* envoi de n octets */
        printf("Message %s envoyé avec succès (%d octets)\n\n", messageEnvoi, ecrits);
}

/* Reception des données du serveur */
lus = read(descripteurSocket, messageRecu, LG_MESSAGE*sizeof(char)); /* attend un message
    de TAILLE fixe */
switch(lus)
{
    case -1 : /* une erreur ! */
        perror("read");
        close(descripteurSocket);
        exit(-4);
    case 0 : /* la socket est fermée */
        fprintf(stderr, "La socket a été fermée par le serveur !\n\n");
        close(descripteurSocket);
        return 0;
    default: /* réception de n octets */
        printf("Message reçu du serveur : %s (%d octets)\n\n", messageRecu, lus);
}
//--- Fin de l'étape n°4 !

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

return 0;
}
```

Un client TCP en C (itération 3)

On utilise la même procédure de test que précédemment en démarrant un serveur netcat sur le port 5000 :

```
$ nc -l -p 5000
```

Puis, on exécute notre client :

```
$ ./clientTCP-3
Socket créée avec succès ! (3)
Connexion au serveur réussie avec succès !
Message Hello world !
    envoyé avec succès (14 octets)
```

```
Message reçu du serveur : ok
```

(3 octets)

Dans la console où on a exécuté le serveur netcat, cela donne :

```
$ nc -l -p 5000
Hello world !
ok
```



Dans netcat, pour envoyer des données au client, il suffit de saisir son message et de valider par la touche Entrée.

Que se passe-t-il si le serveur s'arrête (en tapant Ctrl-C par exemple!) au lieu d'envoyer "ok" ?

```
$ nc -l -p 5000
Hello world !
^C
```

```
$ ./clientTCP-3
Socket créée avec succès ! (3)
Connexion au serveur réussie avec succès !
Message Hello world !
envoyé avec succès (14 octets)
```

La socket a été fermée par le serveur !

Notre client a bien détecté la fermeture de la socket côté serveur.

Dans les codes sources ci-dessus, nous avons utilisés l'appel `close()` pour fermer la socket et donc la communication. En TCP, la communication étant bidirectionnelle *full duplex*, il est possible de fermer plus finement l'échange en utilisant l'appel `shutdown()` :

```
$ man 2 shutdown
```

```
SHUTDOWN(2)                                Manuel du programmeur Linux                                SHUTDOWN(2)
```

```
NOM
```

```
shutdown - Terminer une communication en full-duplex
```

```
SYNOPSIS
```

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

```
int shutdown(int s, int how);
```

```
DESCRIPTION
```

La fonction `shutdown()` termine tout ou partie d'une connexion full-duplex sur la socket `s`. Si `how` vaut `SHUT_RD`, la réception est désactivée.

Si `how` vaut `SHUT_WR`, l'émission est désactivée. Si `how` vaut `SHUT_RDWR`, l'émission et la réception sont désactivées.

```
VALEUR RENVOYÉE
```

Cet appel système renvoie 0 s'il réussit, ou -1 s'il échoue, auquel cas `errno` contient le code d'erreur.

```
...
```

Extrait de la page man de l'appel système shutdown

Étape n°5 : réalisation d'un serveur TCP

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 va détailler ces différences étape par étape.

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.

```
$ man 2 bind
```

```
BIND(2)                                Manuel du programmeur Linux                                BIND(2)
```

```
NOM
```

```
bind - Fournir un nom à une socket
```

```
SYNOPSIS
```

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

int bind(int sockfd, const struct sockaddr *addr,
         socklen_t addrlen);
```

```
DESCRIPTION
```

Quand une socket est créée avec l'appel système `socket(2)`, elle existe dans l'espace des noms mais n'a pas de nom assigné). `bind()` affecte l'adresse spécifiée dans `addr` à la socket référencée par le descripteur de fichier `sockfd`. `addrlen` indique la taille, en octets, de la structure d'adresse pointée par `addr`. Traditionnellement cette opération est appelée « affectation d'un nom à une socket ».

Il est normalement nécessaire d'affecter une adresse locale avec `bind()` avant qu'une socket `SOCK_STREAM` puisse recevoir des connexions (voir `accept(2)`).

Les règles d'affectation de nom varient suivant le domaine de communication.

```
...
```

```
VALEUR RENVOYÉE
```

L'appel renvoie 0 s'il réussit, ou -1 s'il échoue, auquel cas `errno` contient le code d'erreur.

```
...
```

Extrait de la page man de l'appel système bind

On rappelle que l'adressage d'un processus (local ou 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

Rappel : l'interface `socket` propose une structure d'adresse générique `sockaddr` et le domaine `PF_INET` utilise une structure compatible `sockaddr_in`.

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

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

- `htonl()` pour convertir une adresse IP (sur 32 bits) depuis l'ordre des octets de l'hôte vers celui du réseau
- `htons()` pour convertir le numéro de port (sur 16 bits) depuis l'ordre des octets de l'hôte vers celui du réseau.



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.

Éditer le programme suivant dans un fichier que vous nommerez `"serveurTCP-1.c"` :

```
#include <stdio.h>
#include <stdlib.h> /* pour exit */
#include <sys/types.h>
#include <sys/socket.h>
#include <string.h> /* pour memset */
#include <netinet/in.h> /* pour struct sockaddr_in */
#include <arpa/inet.h> /* pour htons et inet_aton */
#include <unistd.h> /* pour sleep */

#define PORT IPPORT_USERRESERVED // = 5000

int main()
{
    int socketEcoule;
    struct sockaddr_in pointDeRencontreLocal;
    socklen_t longueurAdresse;

    //--- Début de l'étape n°5 :
    // Crée un socket de communication
    socketEcoule = socket(PF_INET, SOCK_STREAM, 0); /* 0 indique que l'on utilisera le
        protocole par défaut associé à SOCK_STREAM soit TCP */

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

    printf("Socket créée avec succès ! (%d)\n", socketEcoule);

    // 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

// On demande l'attachement local de la socket
if((bind(socketEcoute, (struct sockaddr *)&pointDeRencontreLocal, longueurAdresse)) < 0)
{
    perror("bind");
    exit(-2);
}

//--- Fin de l'étape n°5 !
printf("Socket attachée avec succès !\n");

// On s'endort ... (cf. test)
sleep(2);

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

return 0;
}
```

Un serveur TCP en C (itération 1)

Le test est concluant :

```
$ ./serveurTCP-1
Socket créée avec succès ! (3)
Socket attachée avec succès !
```

Attention, tout de même de bien comprendre qu'un numéro de port identifie un processus communiquant !
Exécutons deux fois le même serveur et on obtient alors :

```
$ ./serveurTCP-1 & ./serveurTCP-1
Socket créée avec succès ! (3)
Socket attachée avec succès !
Socket créée avec succès ! (3)
bind: Address already in use
```



Explication : l'attachement local au numéro de port 5000 du deuxième processus échoue car ce numéro de port est déjà attribué par le système d'exploitation au premier processus serveur.

Étape n°6 : mise en attente des connexions

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()` :

```
$ man 2 listen
```

```
LISTEN(2)                                Manuel du programmeur Linux                                LISTEN(2)
```

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.

VALEUR RENVOYÉE

Cet appel système renvoie 0 si il réussit, ou -1 en cas d'échec, auquel cas `errno` est renseignée en conséquence.

...

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.

Éditer le programme suivant dans un fichier que vous nommerez "`serveurTCP-2.c`" :

```
#include <stdio.h>
#include <stdlib.h> /* pour exit */
#include <sys/types.h>
#include <sys/socket.h>
#include <string.h> /* pour memset */
#include <netinet/in.h> /* pour struct sockaddr_in */
#include <arpa/inet.h> /* pour htons et inet_aton */
#include <unistd.h> /* pour sleep */
```



```
#define PORT IPPORT_USERRESERVED // = 5000

int main()
{
    int socketEcoute;
    struct sockaddr_in pointDeRencontreLocal;
    socklen_t longueurAdresse;

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

    // Teste la valeur renvoyée par l'appel système socket()
    if(socketEcoute < 0) {
        perror("socket"); // Affiche le message d'erreur
        exit(-1); // On sort en indiquant un code erreur
    }
    printf("Socket créée avec succès ! (%d)\n", socketEcoute);

    // 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

    // On demande l'attachement local de la socket
    if((bind(socketEcoute, (struct sockaddr *)&pointDeRencontreLocal, longueurAdresse)) < 0)
    {
        perror("bind");
        exit(-2);
    }
    printf("Socket attachée avec succès !\n");

    //--- Début de l'étape n°6 :
    // On fixe la taille de la file d'attente à 5 (pour les demandes de connexion non encore
        traitées)
    if(listen(socketEcoute, 5) < 0)
    {
        perror("listen");
        exit(-3);
    }

    //--- Fin de l'étape n°6 !
    printf("Socket placée en écoute passive ... \n");

    // On ferme la ressource avant de quitter
    close(socketEcoute);
    return 0;
}
```

Un serveur TCP en C (itération 2)

Étape n°7 : accepter les demandes connexions

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()` :

```
$ man 2 accept
```

```
ACCEPT(2)
```

```
Manuel du programmeur Linux
```

```
ACCEPT(2)
```

```
NOM
```

```
accept - Accepter une connexion sur une socket
```

```
SYNOPSIS
```

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

```
int accept(int sockfd, struct sockaddr *adresse, socklen_t *longueur);
```

```
DESCRIPTION
```

L'appel système `accept()` est employé avec les sockets utilisant un protocole en mode connecté `SOCK_STREAM`. Il extrait la première connexion de la file des connexions en attente de la socket `sockfd` à l'écoute, crée une nouvelle socket connectée, et renvoie un nouveau descripteur de fichier qui fait référence à cette socket.

La nouvelle socket n'est pas en état d'écoute. La socket originale `sockfd` n'est pas modifiée par l'appel système.

```
...
```

```
VALEUR RENVOYÉE
```

S'il réussit, `accept()` renvoie un entier non négatif, constituant un descripteur pour la nouvelle socket. S'il échoue, l'appel renvoie `-1` et `errno` contient le code d'erreur.

```
...
```

Extrait de la page man de 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.

On va pour l'instant ignorer ce problème et mettre en oeuvre un serveur basique : c'est-à-dire mono-client (ou plus exactement un client après l'autre) !

Concernant le dialogue, on utilisera au choix les mêmes fonctions (`read()/write()` ou `recv()/send()`) que le client.

Éditer le programme suivant dans un fichier que vous nommerez "`serveurTCP-3.c`" :

```
#include <stdio.h>
#include <stdlib.h> /* pour exit */
```

```
#include <sys/types.h>
#include <sys/socket.h>
#include <string.h> /* pour memset */
#include <netinet/in.h> /* pour struct sockaddr_in */
#include <arpa/inet.h> /* pour htons et inet_aton */
#include <unistd.h> /* pour sleep */

#define PORT IPPORT_USERRESERVED // = 5000

#define LG_MESSAGE 256

int main()
{
    int socketEcoule;
    struct sockaddr_in pointDeRencontreLocal;
    socklen_t longueurAdresse;
    int socketDialogue;
    struct sockaddr_in pointDeRencontreDistant;
    char messageEnvoi[LG_MESSAGE]; /* le message de la couche Application ! */
    char messageRecu[LG_MESSAGE]; /* le message de la couche Application ! */
    int ecrits, lus; /* nb d'octets ecrits et lus */
    int retour;

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

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

    printf("Socket créée avec succès ! (%d)\n", socketEcoule);

    // 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);

    // On demande l'attachement local de la socket
    if((bind(socketEcoule, (struct sockaddr *)&pointDeRencontreLocal, longueurAdresse)) < 0)
    {
        perror("bind");
        exit(-2);
    }

    printf("Socket attachée avec succès !\n");
```

```
// On fixe la taille de la file d'attente à 5 (pour les demandes de connexion non encore
traitées)
if(listen(socketEcoule, 5) < 0)
{
    perror("listen");
    exit(-3);
}

printf("Socket placée en écoute passive ... \n");

//--- Début de l'étape n°7 :
// boucle d'attente de connexion : en théorie, un serveur attend indéfiniment !
while(1)
{
    memset(messageEnvoi, 0x00, LG_MESSAGE*sizeof(char));
    memset(messageRecu, 0x00, LG_MESSAGE*sizeof(char));
    printf("Attente d'une demande de connexion (quitter avec Ctrl-C)\n\n");
    // c'est un appel bloquant
    socketDialogue = accept(socketEcoule, (struct sockaddr *)&pointDeRencontreDistant, &
        longueurAdresse);

    if (socketDialogue < 0)
    {
        perror("accept");
        close(socketDialogue);
        close(socketEcoule);
        exit(-4);
    }

    // On réception les données du client (cf. protocole)
    lus = read(socketDialogue, messageRecu, LG_MESSAGE*sizeof(char)); // ici appel
    bloquant
    switch(lus)
    {
        case -1 : /* une erreur ! */
            perror("read");
            close(socketDialogue);
            exit(-5);
        case 0 : /* la socket est fermée */
            fprintf(stderr, "La socket a été fermée par le client !\n\n");
            close(socketDialogue);
            return 0;
        default: /* réception de n octets */
            printf("Message reçu : %s (%d octets)\n\n", messageRecu, lus);
    }

    // On envoie des données vers le client (cf. protocole)
    sprintf(messageEnvoi, "ok\n");
    ecrits = write(socketDialogue, messageEnvoi, strlen(messageEnvoi));
    switch(ecrits)
    {
        case -1 : /* une erreur ! */
            perror("write");
```

```
        close(socketDialogue);
        exit(-6);
    case 0 : /* la socket est fermée */
        fprintf(stderr, "La socket a été fermée par le client !\n\n");
        close(socketDialogue);
        return 0;
    default: /* envoi de n octets */
        printf("Message %s envoyé (%d octets)\n\n", messageEnvoi, ecrits);
}

// On ferme la socket de dialogue et on se replace en attente ...
close(socketDialogue);
}
//--- Fin de l'étape n°7 !

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

return 0;
}
```

Un serveur TCP en C (itération 3)

Testons notre serveur avec notre client :

```
$ ./serveurTCP-3
Socket créée avec succès ! (3)
Socket attachée avec succès !
Socket placée en écoute passive ...
Attente d'une demande de connexion (quitter avec Ctrl-C)
```

```
Message reçu : Hello world !
(14 octets)
```

```
Message ok
envoyé (3 octets)
```

```
Attente d'une demande de connexion (quitter avec Ctrl-C)
```

```
Message reçu : Hello world !
(14 octets)
```

```
Message ok
envoyé (3 octets)
```

```
Attente d'une demande de connexion (quitter avec Ctrl-C)
```

```
~C
```

On va exécuter deux clients à la suite :

```
$ ./clientTCP-3
Socket créée avec succès ! (3)
Connexion au serveur réussie avec succès !
Message Hello world !
envoyé avec succès (14 octets)
```

Message reçu du serveur : ok
(3 octets)

```
$ ./clientTCP-3
Socket créée avec succès ! (3)
Connexion au serveur réussie avec succès !
Message Hello world !
envoyé avec succès (14 octets)
```

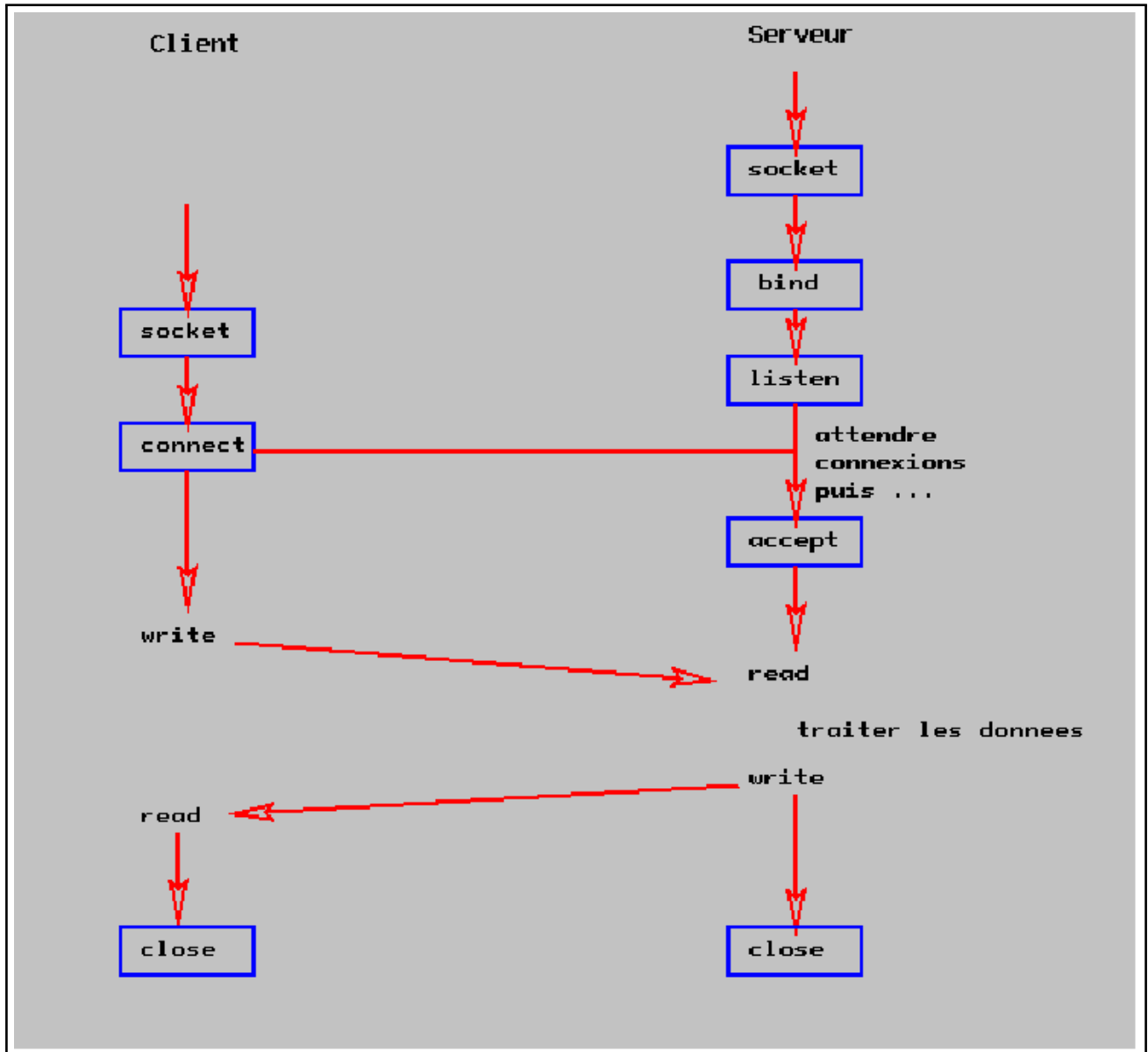
Message reçu du serveur : ok
(3 octets)



Il est évidemment possible de tester notre serveur avec des clients TCP existants comme telnet ou netcat.

Bilan

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



Les appels systèmes utilisés dans un échange TCP

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 ou AF_INET ? Est-ce le seul utilisable avec l'interface *socket* ? En citer au moins un autre.