

TP Développement Réseau n°2 : Socket 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
Manipulations	5
Objectifs	5
Étape n°1 : création de la socket (côté client)	5
Étape n°2 : attachement local de la socket	6
Étape n°3 : communication avec le serveur	9
Étape n°4 : vérification du bon fonctionnement de l'échange	12
Étape n°4 : réalisation d'un serveur UDP	13
Bilan	16
Questions de révision	17
Travail demandé	17
Exercice 1 : client UDP amélioré	17
Exercice 2 : serveur UDP amélioré	18
Exercice 3 : une messagerie instantanée en UDP	18

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

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



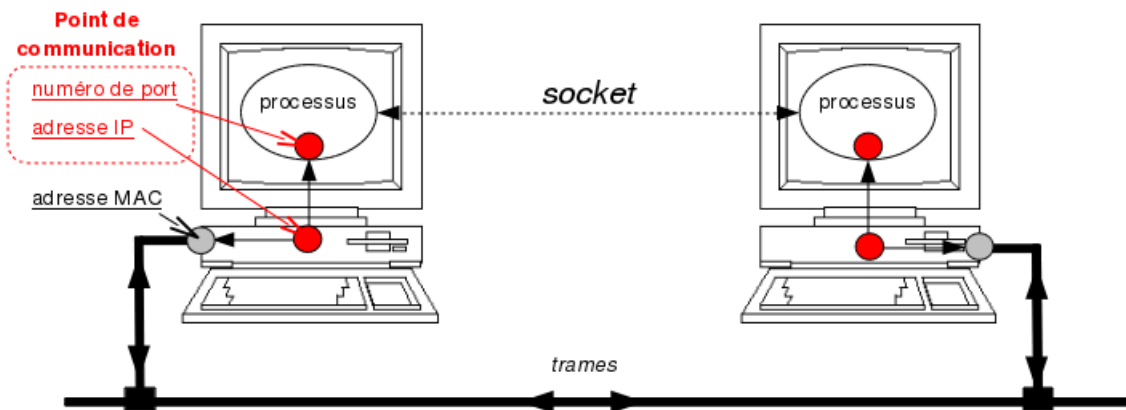
L'accès aux pages man se fera donc avec la commande `man`, par exemple : `man 7 socket`

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)

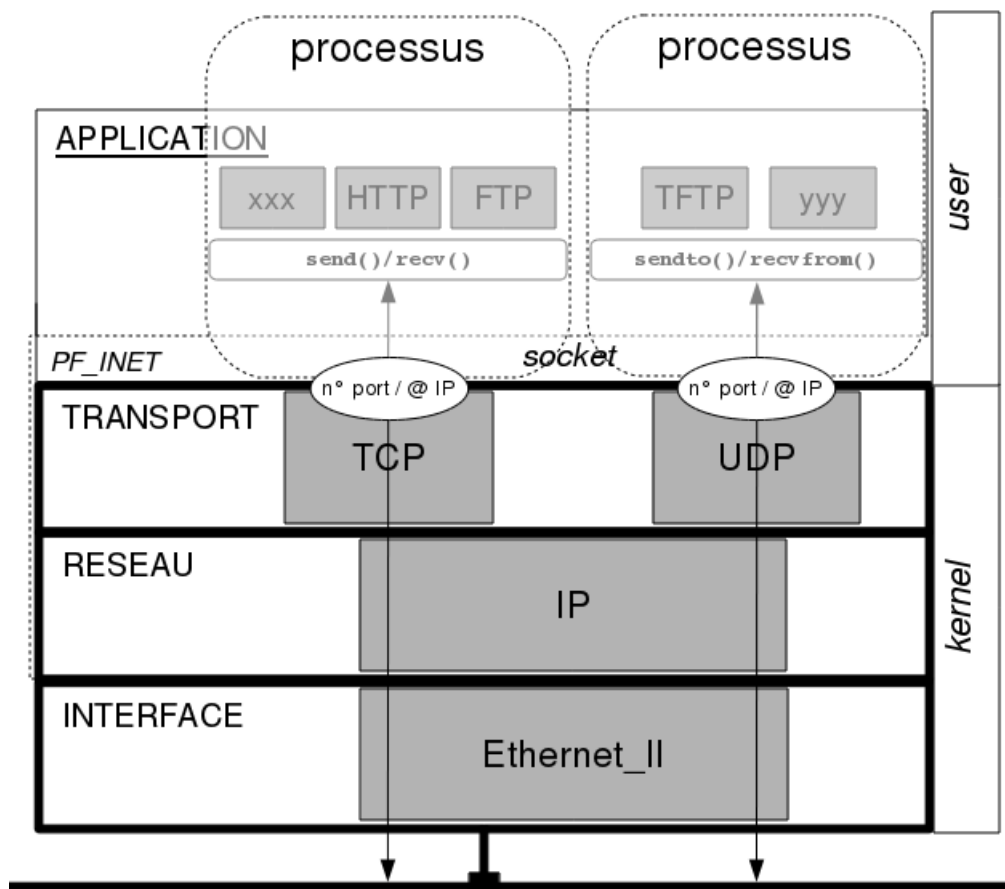
Modèle

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`.
- 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 le 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 UDP** 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 `"clientUDP-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_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
}

//--> 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;
}

```

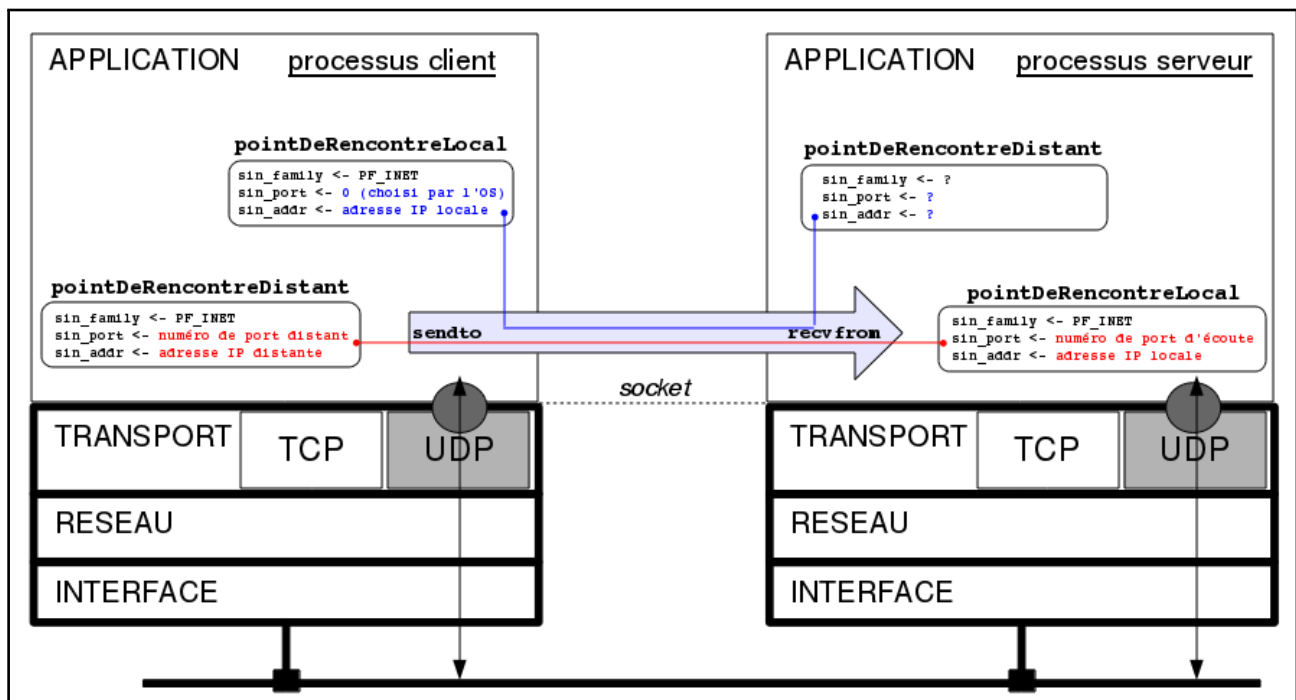
Un client UDP en C (itération 1)



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`.

Étape n°2 : attachement local de la socket

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**. On consulte la page du manuel associée à cet appel :

```
$ 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 ».

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 client** (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 un numéro de port utilisé par le client pour cette socket. Cela peut s'avérer délicat si on ne connaît pas les numéros de port libres. Le plus simple est de laisser le système d'exploitation choisir en indiquant la valeur 0 dans le champ `sin_port`.

Éditer le programme suivant dans un fichier que vous nommerez "clientUDP-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 htonl */

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

    // Crée un socket de communication
    descripteurSocket = socket(PF_INET, SOCK_DGRAM, 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 !
    // 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); // n'importe quelle interface
        locale disponible
    pointDeRencontreLocal.sin_port = htons(0); // l'os choisira un numéro de port libre

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

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

Un client UDP en C (itération 2)

Le test est concluant :

```
$ ./clientUDP-2
Socket créée avec succès ! (3)
Socket attachée avec succès !
```

Étape n°3 : communication avec le serveur

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 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`.

Ici, on limitera notre client à l'envoi d'une chaîne de caractères. Pour cela, on va utiliser l'appel `sendto()` :

```
$ man 2 sendto
```

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

NOM

`sendto` - Envoyer un message sur une socket

SYNOPSIS

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

```
ssize_t sendto(int s, const void *buf, size_t len, int flags,
               const struct sockaddr *to, socklen_t tolen);
```

DESCRIPTION

L'appel système `sendto()` permet de transmettre un message à destination d'une autre socket.

Le paramètre `s` est le descripteur de fichier de la socket émettrice. L'adresse de la cible est fournie par `to`, `tolen` spécifiant sa taille. Le message se trouve dans `buf` et a pour longueur `len`.

...

VALEUR RENVOYÉE

S'ils réussissent, ces appels système renvoient le nombre de caractères émis. S'ils échouent, ils renvoient `-1` et `errno` contient le code d'erreur.

...

Extrait de la page man de l'appel système `sendto`

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

Et il suffira donc d'initialiser une structure `sockaddr_in` avec les informations distantes du serveur (adresse IPv4 et numéro de port). Cela revient à adresser le **point de rencontre distant** qui sera utilisé dans l'appel `sendto()` par le client.

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 "clientUDP-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, htonl et inet_aton */

#define LG_MESSAGE 256

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

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

```
// 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); // n'importe quelle interface
    locale disponible
pointDeRencontreLocal.sin_port = htons(0); // l'os choisira un numéro de port libre

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

//<-- Début de l'étape n°3
// 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

// Initialise à 0 le message
memset(messageEnvoi, 0x00, LG_MESSAGE*sizeof(char));

// Envoie un message au serveur
sprintf(messageEnvoi, "Hello world !\n");
ecrits = sendto(descripteurSocket, messageEnvoi, strlen(messageEnvoi), 0, (struct
    sockaddr *)&pointDeRencontreDistant, longueurAdresse);
switch(ecrits)
{
    case -1 : /* une erreur ! */
        perror("sendto");
        close(descripteurSocket);
        exit(-3);
    case 0 :
        fprintf(stderr, "Aucune donnée n'a été envoyée !\n\n");
        close(descripteurSocket);
        return 0;
    default: /* envoi de n octets */
        if(ecrits != strlen(messageEnvoi))
            fprintf(stderr, "Erreur dans l'envoi des données !\n\n");
        else
            printf("Message %s envoyé avec succès (%d octets)\n\n", messageEnvoi, ecrits);
}
```

```
//--> Fin de l'étape n°3 !

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

return 0;
}
```

Un client UDP en C (itération 3)

Si vous testez ce client (sans serveur), vous risquez d'obtenir :

```
$ ./clientUDP-3
Socket créée avec succès ! (3)
Socket attachée avec succès !
Message Hello world !
envoyé avec succès (14 octets)
```

Le message a été envoyé au serveur : ceci peut s'expliquer tout simplement parce que nous sommes en mode non-connecté.



Le protocole UDP ne prend pas en charge un mode fiable : ici, le client a envoyé des données sans savoir si un serveur était prêt à les recevoir !

Étape n°4 : vérification du bon fonctionnement de l'échange

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

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

Puis :

```
$ ./clientUDP-3
Socket créée avec succès ! (3)
Socket attachée avec succès !
Message Hello world !
envoyé avec succès (14 octets)
```



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 attente avant que le client envoie ses données.

Le message a bien été reçu et affiché par le serveur `netcat` :

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

Étape n°4 : réalisation d'un serveur UDP

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).

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.



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

Dans notre exemple, le serveur va seulement réceptionner un datagramme en provenance du client. Pour cela, il va utiliser l'appel système `recvfrom()` :

```
$ man 2 recvfrom
```

RECV(2)

Manuel du programmeur Linux

RECV(2)

NOM

`recvfrom` - Recevoir un message sur une socket

SYNOPSIS

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

ssize_t recvfrom(int s, void *buf, size_t len, int flags,
                 struct sockaddr *from, socklen_t *fromlen);
```

DESCRIPTION

L'appel système `recvfrom()` est utilisé pour recevoir des messages.

Si `from` n'est pas `NULL`, et si le protocole sous-jacent fournit l'adresse de la source, celle-ci y est insérée. L'argument `fromlen` est un paramètre résultat, initialisé à la taille du tampon `from`, et modifié en retour pour indiquer la taille réelle de l'adresse enregistrée.

...

VALEUR RENVOYÉE

Ces fonctions renvoient le nombre d'octets reçus si elles réussissent, ou `-1` si elles échouent. La valeur de retour sera `0` si le pair a effectué un arrêt normal.

Extrait de la page man de l'appel système `recvfrom`



C'est l'appel `recvfrom()` qui remplit la structure `sockaddr_in` avec les informations du point de communication du client (adresse IPv4 et numéro de port pour `PF_INET`).

Éditer le programme suivant dans un fichier que vous nommerez "serveurUDP.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, htonl et inet_aton */

#define PORT IPPORT_USERRESERVED // = 5000

#define LG_MESSAGE 256

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

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

    // 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);

    // 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); // n'importe quelle interface
        locale disponible
    pointDeRencontreLocal.sin_port = htons(PORT); // <- 5000

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

    //<-- Début de l'étape n°4
    // Obtient la longueur en octets de la structure sockaddr_in
```

```
longueurAdresse = sizeof(pointDeRencontreDistant);
// Initialise à 0 la structure sockaddr_in (c'est l'appel recvfrom qui remplira cette
// structure)
memset(&pointDeRencontreDistant, 0x00, longueurAdresse);

// Initialise à 0 le message
memset(messageRecu, 0x00, LG_MESSAGE*sizeof(char));

// Réceptionne un message du client
lus = recvfrom(descripteurSocket, messageRecu, sizeof(messageRecu), 0, (struct sockaddr
    *)&pointDeRencontreDistant, &longueurAdresse);
switch(lus)
{
    case -1 : /* une erreur ! */
        perror("recvfrom");
        close(descripteurSocket);
        exit(-3);
    case 0 :
        fprintf(stderr, "Aucune donnée n'a été reçue !\n\n");
        close(descripteurSocket);
        return 0;
    default: /* réception de n octets */
        printf("Message %s reçu avec succès (%d octets)\n\n", messageRecu, lus);
}
//--> Fin de l'étape n°4 !

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

return 0;
}
```

Un serveur UDP en C

Une simple exécution du serveur le place en attente d'une réception de données :

```
$ ./serveurUDP
Socket créée avec succès ! (3)
Socket attachée avec succès !
^C
```

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 :

```
$ ./serveurUDP & ./serveurUDP
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. TCP et UDP ne partagent pas le même espace d'adressage (numéro de port logique indépendant).

Testons notre serveur avec notre client :

```
$ ./serveurUDP
Socket créée avec succès ! (3)
Socket attachée avec succès !
Message Hello world !
  reçu avec succès (14 octets)

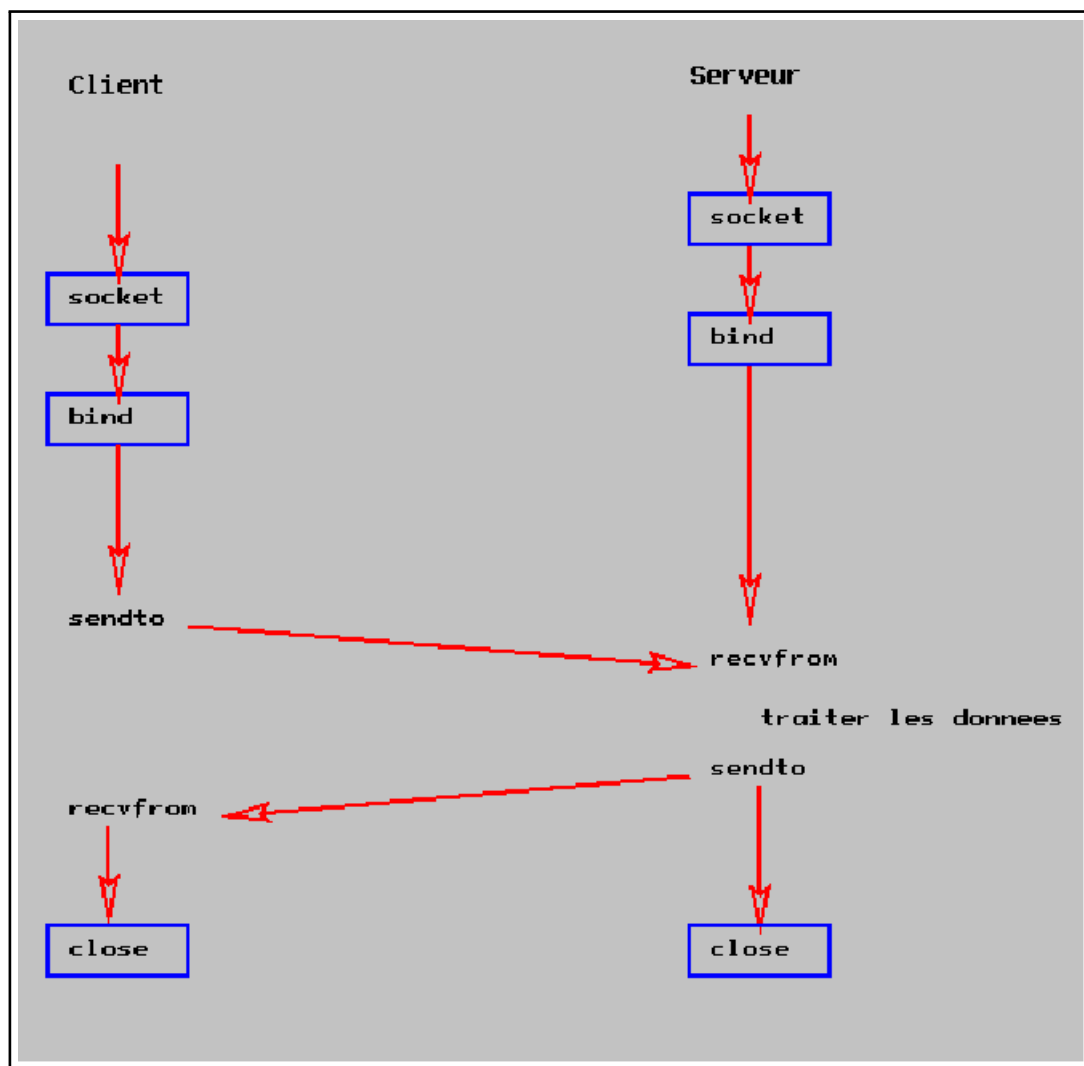
$ ./clientUDP-3
Socket créée avec succès ! (3)
Socket attachée avec succès !
Message Hello world !
  envoyé avec succès (14 octets)
```



Il est évidemment possible de tester notre serveur avec le client UDP de netcat avec l'option -u.

Bilan

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



Les appels systèmes utilisés dans un échange UDP

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.

Travail demandé

Exercice 1 : client UDP amélioré

L'objectif de cet exercice est de modifier le client UDP afin qu'il puisse se connecter vers n'importe quel serveur UDP.

Étant donné qu'un serveur est identifié par une adresse IPv4 (ou par un nom) et un numéro de port, il suffit que le client UDP reçoive ces deux informations en arguments comme ceci :

```
$ ./clientUDP-exo1 adresse_ip_serveur|nom_serveur numero_port_serveur
```

Si le client ne reçoit pas ces deux arguments, il affichera son "usage" :

```
$ ./clientUDP-exo1
```

```
Erreur : argument manquant !
```

```
Usage : ./clientUDP-exo1 adresse_ip_serveur|nom_serveur numero_port_serveur
```



Rappel : récupérer l'adresse IPv4 en notation décimale pointée ne pose pas de problème. Par contre, il vous faudra convertir (avec `atoi` ou `strtol`) le numéro de port reçu sous la forme d'une chaîne de caractères en un entier court sur 16 bits avant de l'utiliser.

Question 11. Réaliser le client `clientUDP-exo1` demandé et tester le.

Exercice 2 : serveur UDP amélioré

L'objectif de cet exercice est de modifier le serveur UDP afin qu'il affiche les informations (adresse IPv4 et numéro de port) identifiant le client qui lui a envoyé des données.

Il vous faut utiliser la fonction `getnameinfo()` pour obtenir ces informations.



Avant on utilisait la fonction `gethostbyaddr()` mais, celle-ci est maintenant obsolète.

Question 12. Réaliser le serveur `serveurUDP-exo2` demandé et tester le.

Vous devriez obtenir ceci (ici ma machine se nomme "alias") :

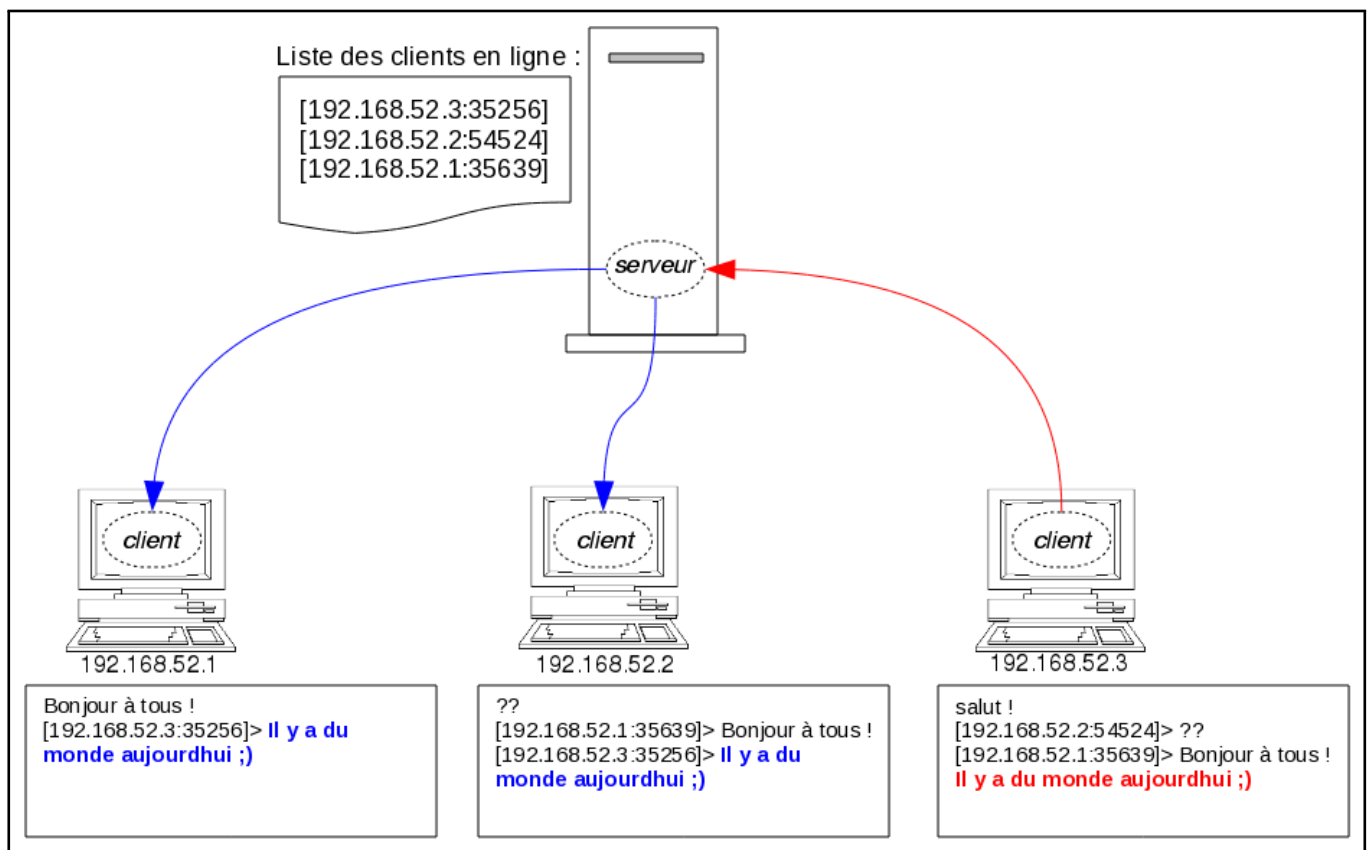
```
$ ./serveurUDP-exo2
Socket créée avec succès ! (3)
Socket attachée avec succès !
Message Hello world !
reçu du client alias:32941
```



Rappel : le numéro de port du client a été choisi librement (parmi les numéros disponibles) par le système d'exploitation.

Exercice 3 : une messagerie instantanée en UDP

L'objectif de cet exercice est de réaliser un mini-chat multi-clients.



Fonctionnement du mini-chat

Le principe est simple : tous les messages envoyés au serveur seront distribués à l'ensemble des clients en ligne. Un client est considéré "en ligne" si il a déjà envoyé un message.

On se limitera :

- à des messages d'une taille de 225 caractères (pour un tampon de 256 caractères max.)
- à 10 clients en ligne maximum



Un protocole plus élaboré devrait être mis en oeuvre au niveau de la couche APPLICATION!

Question 13. Écrire le serveur `serveurChat.c`.

Voici un exemple de ce que vous devez obtenir côté serveur :

```
$ ./serveurChat
```

```
Socket créée avec succès ! (3)
```

```
Socket attachée avec succès !
```

```
Message : salut !
```

```
-> reçu du client [192.168.52.3:35256]
```

```
Liste des clients en ligne :
```

```
[192.168.52.3:35256]
```

```
Message : ??
```

```
-> reçu du client [192.168.52.2:54524]
```

```
Message : ??
```

```
-> envoyé à [192.168.52.3:35256]
```

```
Le message a été envoyé à 1 client(s)
```

```
Liste des clients en ligne :
```

```
[192.168.52.3:35256]
```

```
[192.168.52.2:54524]
```

```
Message : Bonjour à tous !
```

```
-> reçu du client [192.168.52.1:35639]
```

```
Message : Bonjour à tous !
```

```
-> envoyé à [192.168.52.3:35256]
```

```
Message : Bonjour à tous !
```

```
-> envoyé à [192.168.52.2:54524]
```

```
Le message a été envoyé à 2 client(s)
```

```
Liste des clients en ligne :
```

```
[192.168.52.3:35256]
```

```
[192.168.52.2:54524]
```

```
[192.168.52.1:35639]
```

```
Message : Il y a du monde aujourd'hui ;)
```

```
-> reçu du client [192.168.52.3:35256]
```

Message : Il y a du monde aujourd'hui ;)
-> envoyé à [192.168.52.2:54524]

Message : Il y a du monde aujourd'hui ;)
-> envoyé à [192.168.52.1:35639]

Le message a été envoyé à 2 client(s)

Liste des clients en ligne :
[192.168.52.3:35256]
[192.168.52.2:54524]
[192.168.52.1:35639]



Le client réalisé à l'exercice 1 peut être utilisé ici. Sinon, vous pouvez faire vos tests avec netcat.

Voici un exemple d'utilisation de 3 clients :

```
$ nc -u 192.168.52.83 5000
salut !
[192.168.52.2:54524]> ??
[192.168.52.1:35639]> Bonjour à tous !
Il y a du monde aujourd'hui ;)
```

Le client 192.168.52.3

```
$ nc -u 192.168.52.83 5000
??
[192.168.52.1:35639]> Bonjour à tous !
[192.168.52.3:35256]> Il y a du monde aujourd'hui ;)
```

Le client 192.168.52.2

```
$ nc -u 192.168.52.83 5000
Bonjour à tous !
[192.168.52.3:35256]> Il y a du monde aujourd'hui ;)
```

Le client 192.168.52.1