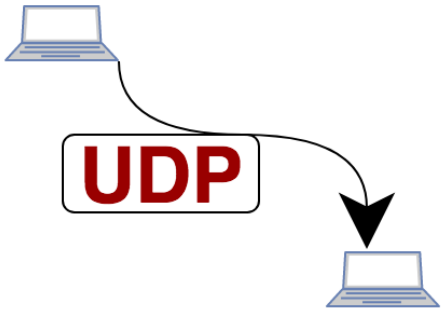
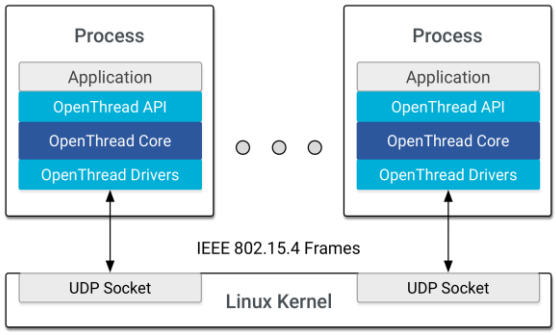




Extrait du référentiel : BTS Systèmes Numériques option A (Informatique et Réseaux)		Niveau(x)
S7. Réseaux, télécommunications et modes de transmission S7.7 Programmation réseau	Sockets POSIX	4

Objectifs du cours :

Ce cours traitera essentiellement les points suivants :

- Le modèle
- Caractéristiques des sockets
- Programmation UDP (Linux)
- Réalisation d'un serveur UDP

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

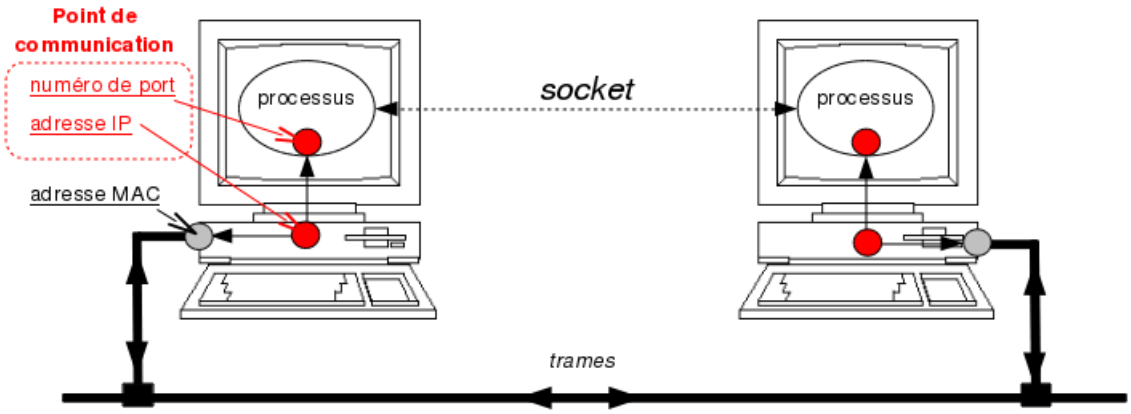
- l'architecture client/serveur ;
- l'adressage IP et les numéros de port ;
- les notions d'API (appels systèmes sous Unix) et de programmation en langage C ;
- les protocoles TCP et UDP, les modes connecté et non connecté.

« La notion de socket a été introduite dans les distributions de Berkeley (un 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).
Il s'agit d'un modèle permettant la communication interprocessus (IPC) afin de permettre à divers processus de communiquer aussi bien sur une même machine qu'à travers un réseau. »

Socket = mécanisme de communication bidirectionelle entre processus

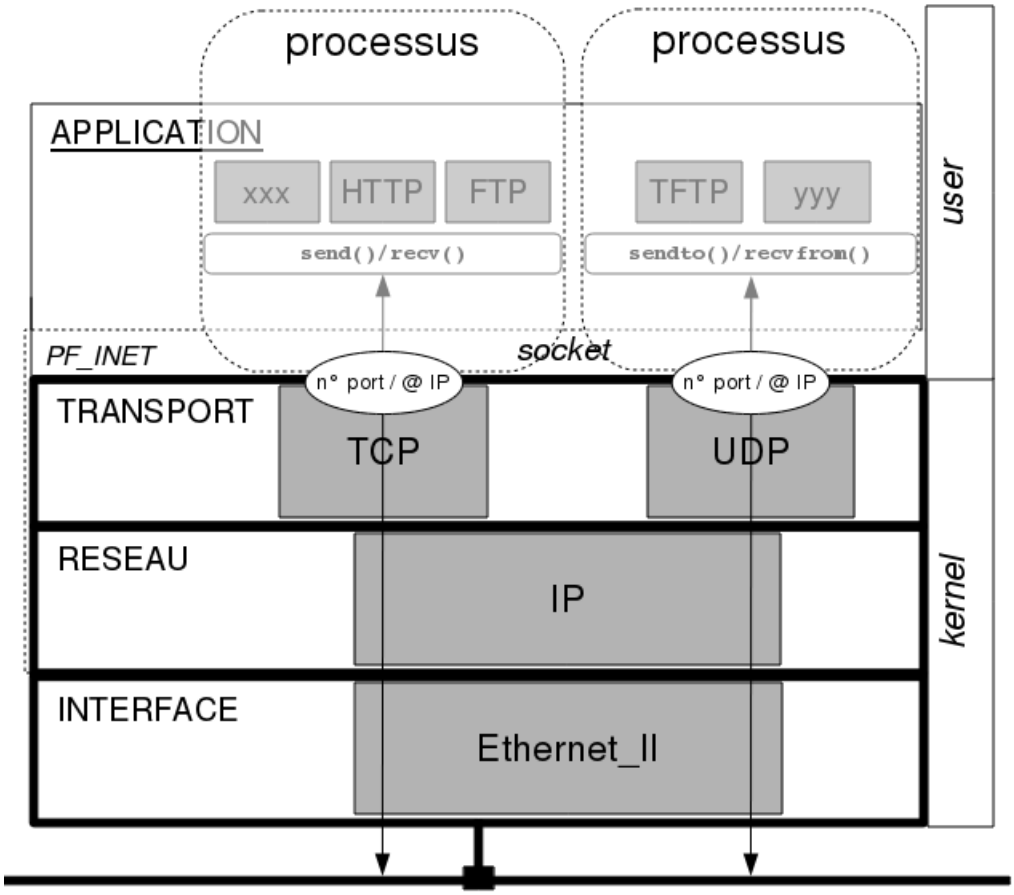
LE MODÈLE

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 de la couche TRANSPORT.



CARACTÉRISTIQUES DES SOCKETS

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

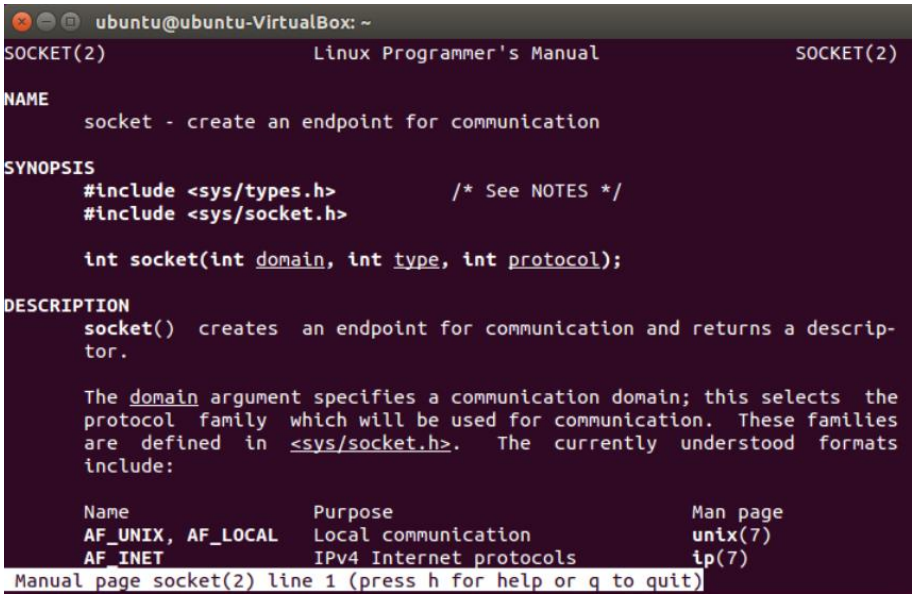
- le **domaine** de communication : ceci sélectionne la famille de protocole à employer. Il faut savoir que chaque famille possède son adressage. Par exemple pour les protocoles Internet IPv4, on utilisera le domaine PF_INET ou AF_INET et AF_INET6 pour le protocole IPv6.
- le **type** de socket à utiliser 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).

PROGRAMMATION UDP (LINUX)

ÉTAPE ① : CRÉATION DE LA SOCKET (CLIENT)

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

~$ **man socket**



Extrait de la page man de l'appel système socket

NOM

socket - Créer un point de communication

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.

...

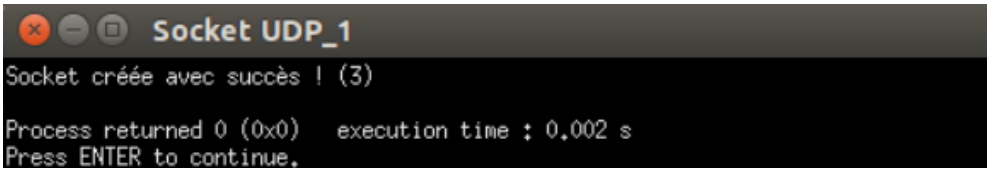
```c
#include <stdio.h>
#include <stdlib.h> /* pour exit */
#include <sys/types.h>
#include <sys/socket.h>
#include <unistd.h> /* pour close */

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;
}
```

Création de la socket (coté client) : Socket UDP_1.c

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.

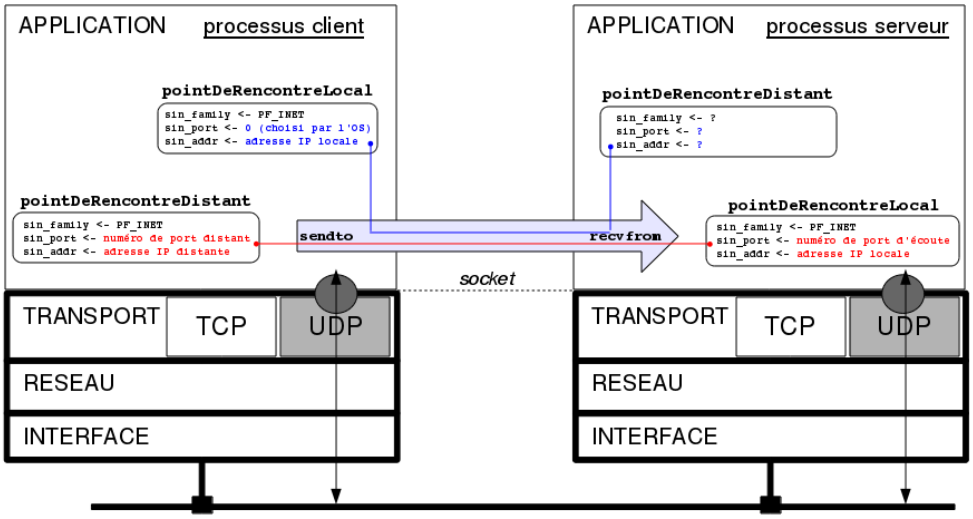
On compile et on exécute pour vérifier que ça fonctionne !



Affichage console

ÉTAPE ② : ATTACHEMENT LOCAL DE LA SOCKET

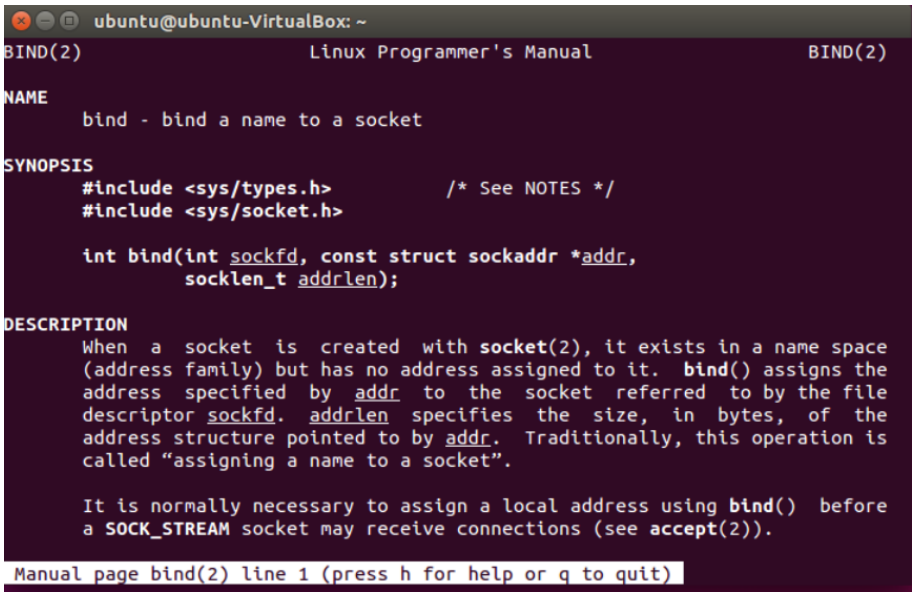
Maintenant que nous avons créé une socket UDP, le client pourrait déjà communiquer avec un serveur UDP car nous utilisons un mode non-connecté.



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 bind**



Extrait de la page man de l'appel système bind

NOM

bind - Fournir un nom à une socket

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.

...

Rappels :

L'adressage d'un processus (local ou distant) dépend du **domaine** de communication (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 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** (adresseIPv4 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.

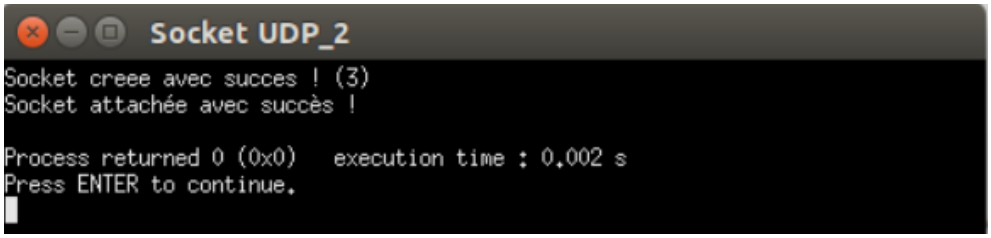
```
#include <stdio.h>
#include <stdlib.h> /* pour exit */
#include <sys/types.h>
#include <sys/socket.h>
#include <string.h> /* pour memset */
```

```c
#include <unistd.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;
}
```

Attachement local de la socket : Socket UDP_2.c

On compile et on exécute !



Affichage console

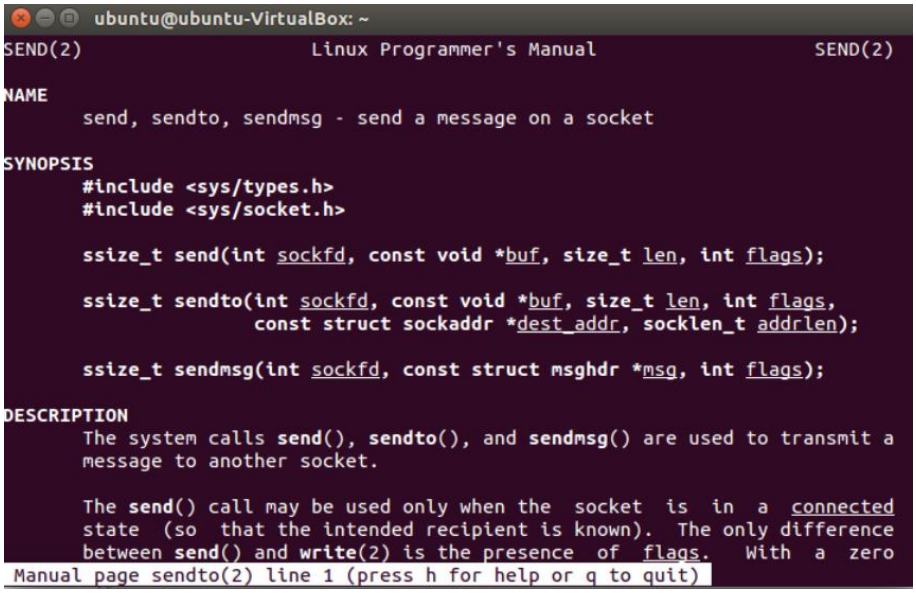
ÉTAPE ③ : COMMUNICATION AVEC LE SERVEUR

Il nous faut des fonctions pour écrire (envoyer) et lire (recevoir) des octets dans la socket.

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 utiliseron en argument une structure sockaddr_in pour PF_INET.

~$ **man send**



Extrait de la page man de l'appel système send

NOM

sendto - Envoyer un message sur une socket

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.

...

Il faut maintenant 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 « big-endian ». Il est donc prudent d'appeler des fonctions qui respectent cet ordre pour coder des informations dans les en-têtes des protocoles réseaux.

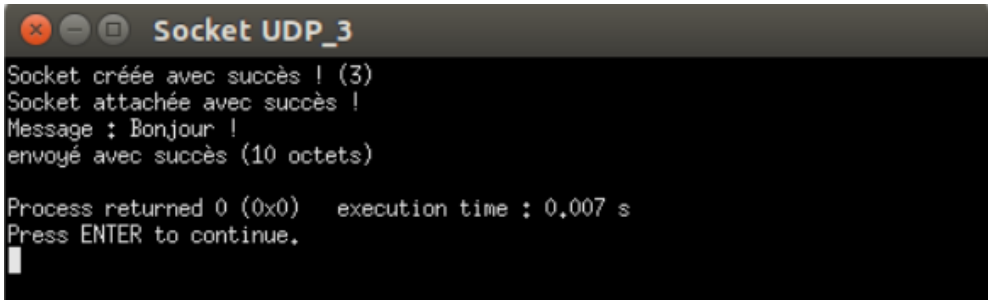
```c
#include <stdio.h>
#include <stdlib.h> /* pour exit */
#include <unistd.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <string.h> /* pour memset */
#include <netinet/in.h> /* pour struct sockaddr_in */
#include <arpa/inet.h> /* pour htons, htonl et inet_aton */
#define LG_MESSAGE 256

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("192.168.1.100", &pointDeRencontreDistant.sin_addr); // à modifier selon le serveur
 // Initialise à 0 le message
        memset(messageEnvoi, 0x00, LG_MESSAGE * sizeof(char));
 // Envoie un message au serveur
 sprintf(messageEnvoi, "Bonjour !\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'envoie 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(descripteurSocket);
  return 0;
}
```

Attachement local de la socket : Socket UDP_3.c

Si on teste sans serveur :



Affichage console

Le message est tout de même envoyé au serveur car tout simplement nous sommes en mode non-connecté (UDP). Le client a envoyé des données sans savoir si un serveur était prêt à les recevoir !

On vérifie le fonctionnement avec un serveur. Pour cela nous pouvons utiliser l'outil réseau « netcat » en mode serveur (-l) en UDP (-u) et sur le port 5000 (-p 5000).

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

Dans l'architecture client/serveur, 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.

```
~$ ./Socket UDP_3
```

ÉTAPE 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).

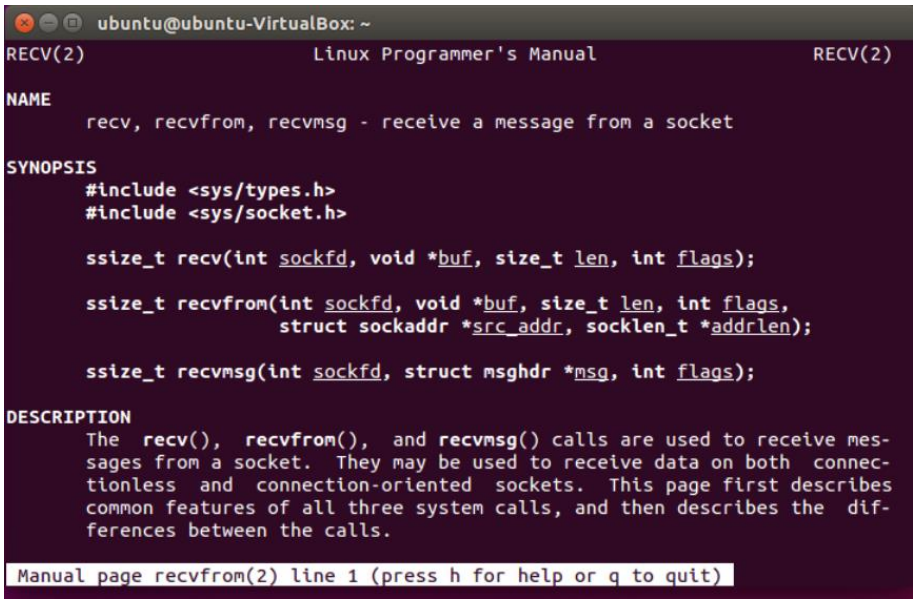
Pour écrire ces informations dans la structure d'adresse, il nous faudra utiliser :

- 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 possible de préciser avec INADDR_ANY que toutes les interfaces locales du serveur accepteront les échanges des clients.

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

~$ **man recvfrom**



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).

```c
#include <stdio.h>
#include <stdlib.h> /* pour exit */
#include <sys/types.h>
#include <sys/socket.h>
#include <unistd.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));
 // Réception d'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;
}
```

Serveur_UDP.c

Affichage console

Attention, tout de même de bien comprendre qu'un numéro de port identifie un processus communiquant !

Si l'on exécute deux fois le même serveur on obtient alors :

```
~$ ./Serveur_UDP & ./Serveur_UDP

Socket créée avec succès ! (3)
Socket attachée avec succès !
Socket créée avec succès ! (3)
bind: Address already in use
```

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. UDP ne partage pas le même espace d'adressage (numéro de port logique indépendant).

Test du serveur (serveur_UDP) et du clien (Socket UDP_3) :

```
~$ ./serveur_UDP

Socket créée avec succès ! (3)
Socket attachée avec succès !
Message Bonjour !
reçu avec succès (10 octets)
```

```
~$ ./Socket UDP_3

Socket créée avec succès ! (3)
Socket attachée avec succès !
Message : Bonjour !
envoyé avec succès (10 octets)
```