TP Développement Réseau n°2 : Socket UDP

Sommaire

Les objectifs de ce tp sont de mettre en oeuvre sous Linux 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 compététion dans l'un de ces domaines sans pratique régulière. Vous savez bien quel serait le résultat.*

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`" :

```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 */
```



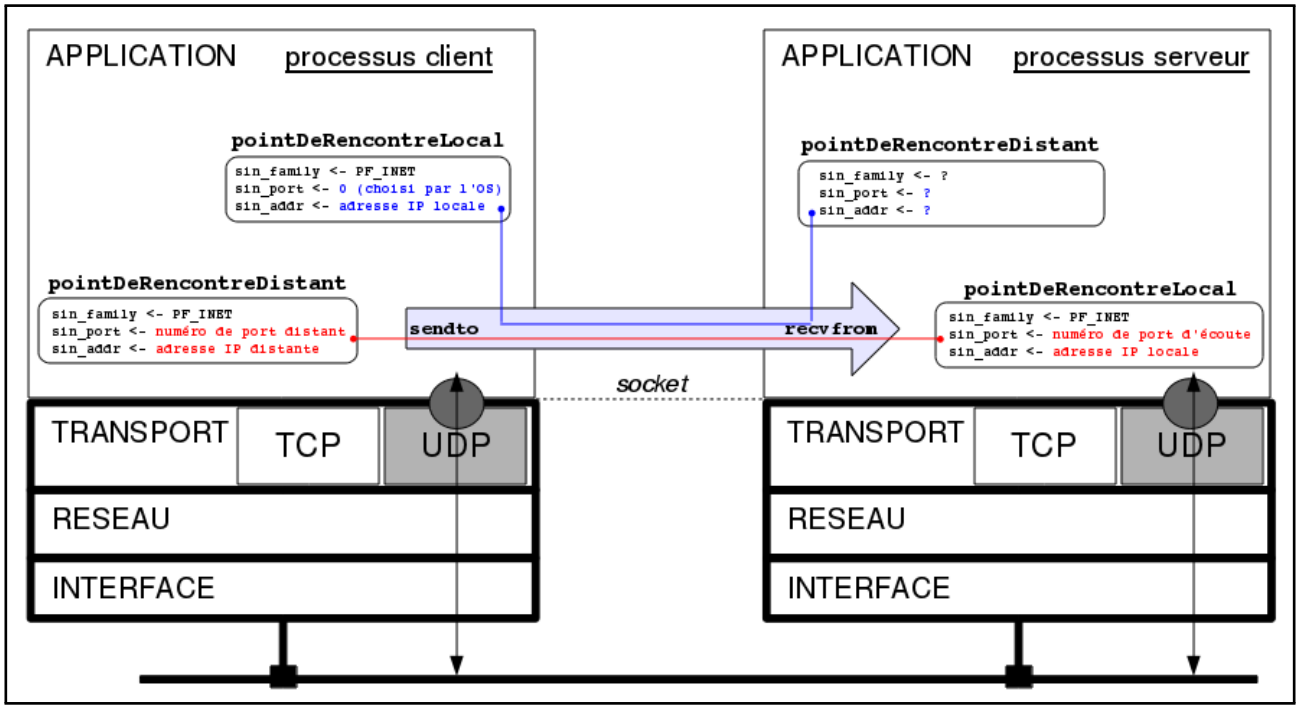
```
   // Teste la valeur renvoyée par l'appel système socket()
   if(descripteurSocket < 0) /* échec ? */
   {
      perror("socket"); // Affiche le message d'erreur
      exit(-1); // On sort en indiquant un code erreur
   }

   //--> 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 utiliser 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`" :


```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'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;
```



```
}
```

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).

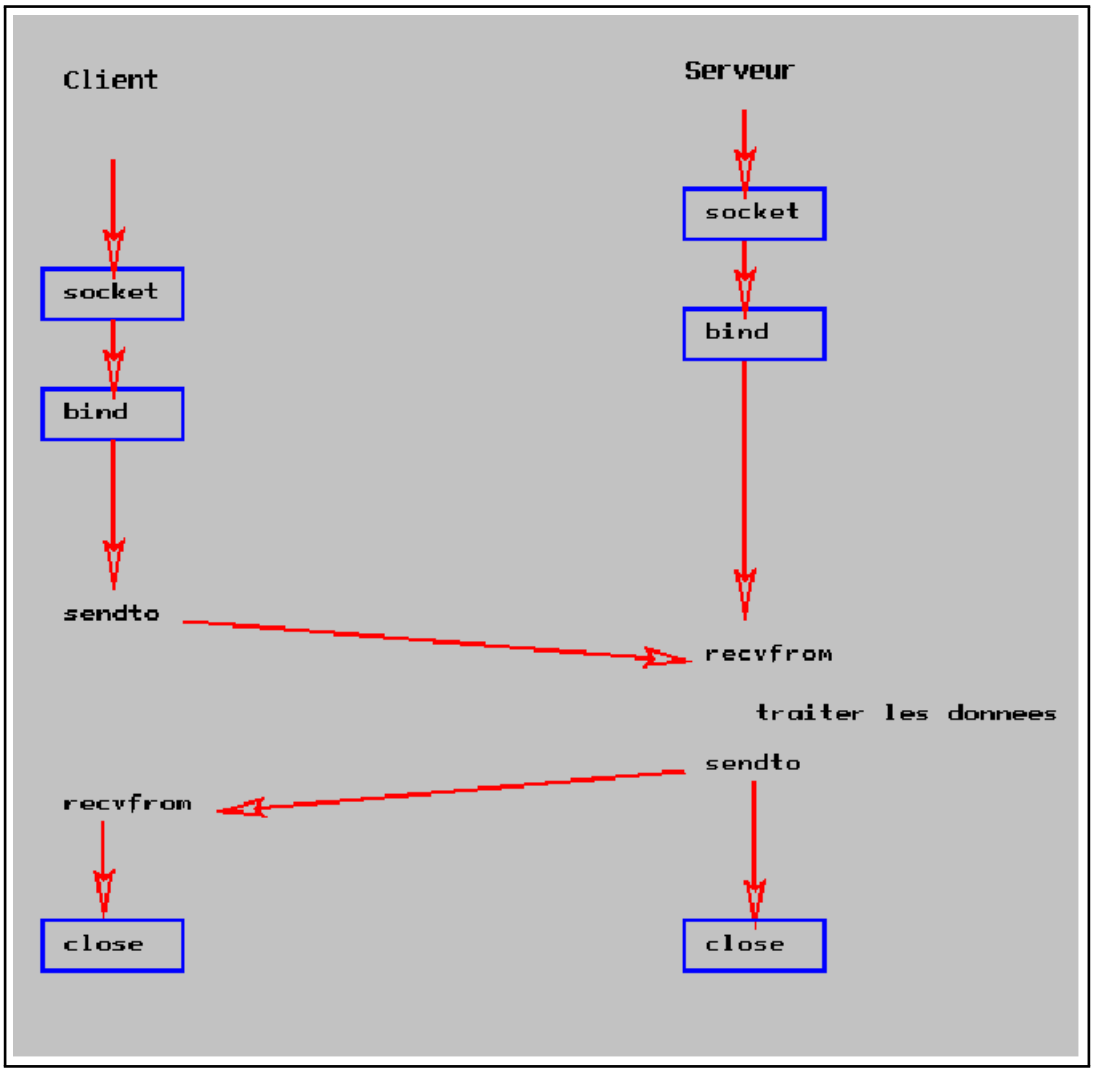
Testons notre serveur avec notre client :

```
$ ./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

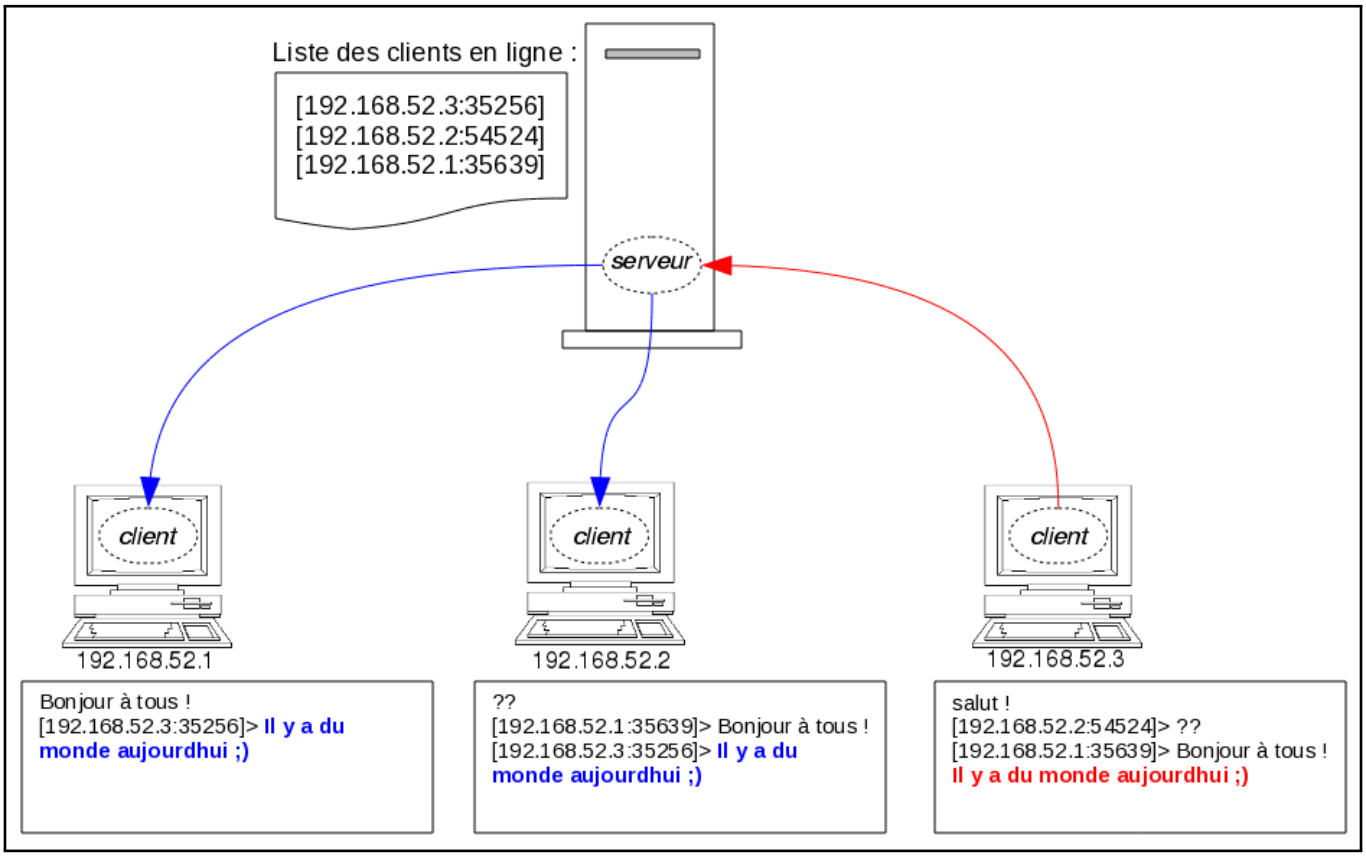
L'échange entre un client et un serveur UDP peut être maintenant schématisé de la manière suivante :



Les appels systèmes utilisés dans un échange UDP

Travail demandé : 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 cractères max.)
– à 10 clients en ligne maximum

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

Question 1. 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 aujourdhui ;)
 -> reçu du client [192.168.52.3:35256]

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

Message : Il y a du monde aujourdhui ;)
 -> 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 aujourdhui ;)
```

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 aujourdhui ;)
```

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 aujourdhui ;)
```

Le client 192.168.52.1

