

Initiation au développement C++ sur les sockets

Philippe Latu

philippe.latu(at)linux-france.org

<http://www.linux-france.org/prj/inetdoc/>

Historique des versions		
\$Revision: 1321 \$	\$Date: 2008-09-24 10:21:50 +0200 (mer 24 sep 2008) \$	\$Author: latu \$
Année universitaire 2007-2008		
Résumé		
L'objet de ce support est d'initier au développement réseau sur les sockets à partir du code le plus minimaliste et le plus portable. Dans ce but, on utilise les sockets UDP et le langage C++.		

Table des matières

1. Copyright et Licence	1
1.1. Meta-information	2
2. Contexte de développement	2
2.1. Système d'exploitation	2
2.2. Instructions de compilation	2
2.3. Instructions d'exécution	3
2.4. Bibliothèques utilisées	3
2.5. Choix du protocole de transport	4
2.6. Sockets & protocole de transport UDP	4
3. Programme client	5
3.1. Codage de l'utilisation des sockets	5
3.2. Code source complet	6
4. Programme serveur	8
4.1. Codage de l'utilisation des sockets	8
4.2. Code source complet	9
5. Analyse réseau	11
6. Documents de référence	11

1. Copyright et Licence

Copyright (c) 2000,2008 Philippe Latu.
Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.2 or any later version published by the Free Software Foundation; with no Invariant Sections, no Front-Cover Texts, and no Back-Cover Texts. A copy of the license is included in the section entitled "GNU Free Documentation License".

Copyright (c) 2000,2008 Philippe Latu.
Permission est accordée de copier, distribuer et/ou modifier ce document selon les termes de la Licence de Documentation Libre GNU (GNU Free Documentation License), version 1.2 ou toute version ultérieure publiée par la Free Software Foundation ; sans Sections Invariables ; sans Texte de Première de Couverture, et sans Texte de Quatrième de Couverture. Une copie de la présente Licence est incluse dans la section intitulée « Licence de Documentation Libre GNU ».

1.1. Meta-information

Cet article est écrit avec *DocBook*¹ XML sur un système *Debian GNU/Linux*². Il est disponible en version imprimable aux formats PDF et Postscript : [socket.cpp.pdf](#)³ | [socket.cpp.ps.gz](#)⁴.

2. Contexte de développement

L'objectif de développement étant l'initiation, on se limite à un code minimaliste.

Le programme client, `udp-client.cc`

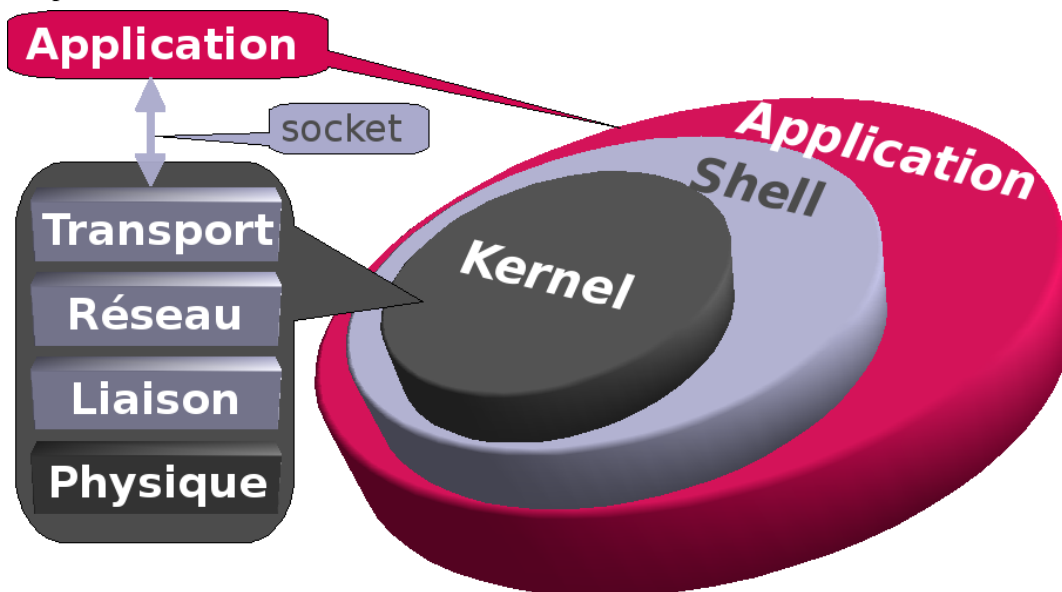
... demande à l'utilisateur de saisir une chaîne de caractères et l'envoi au serveur.

Le programme serveur, `udp-server.cc`

... transforme les caractères de cette chaîne en majuscules et renvoie le résultat au client.

2.1. Système d'exploitation

Le schéma ci-dessous positionne les couches de la modélisation des communications réseau vis-à-vis du système d'exploitation.



Socket vs. système d'exploitation et modélisation réseau - image seule⁵

Les pilotes de périphériques et les protocoles implantés depuis la couche physique jusqu'à la couche transport font partie du sous-système réseau du noyau du système d'exploitation.

Le programme utilisateur est lancé à partir de la couche *Shell* et est exécuté au niveau application.

L'utilisation de socket revient à ouvrir un canal de communication entre la couche application et la couche transport. La programmation des sockets se fait à l'aide de bibliothèques standard présentées ci-après : [Section 2.4, « Bibliothèques utilisées »](#).

2.2. Instructions de compilation

Ces deux programmes sont compilables en l'état sur n'importe quelle boîte Linux. Il suffit d'appeler le compilateur C++ de la chaîne de développement GNU en désignant le nom du programme exécutable avec l'option `-o`.

¹ <http://www.docbook.org>

² <http://www.debian.org>

³ <http://www.linux-france.org/prj/inetdoc/telechargement/socket.cpp.pdf>

⁴ <http://www.linux-france.org/prj/inetdoc/telechargement/socket.cpp.ps.gz>

⁵ http://www.linux-france.org/prj/inetdoc/cours/socket.cpp/images/kernel_socket_protocol_stack.png

```

phil@nowhere:~/cpp/socket$ g++ -o udp-client udp-client.cc
phil@nowhere:~/cpp/socket$ g++ -o udp-server udp-server.cc
phil@nowhere:~/cpp/socket$ ll
total 52K
drwxr-xr-x  2 phil phil 4,0K 2005-06-16 22:45 .
drwxr-xr-x  3 phil phil 4,0K 2005-06-16 22:43 ..
-rwxr-xr-x  1 phil phil 17K 2005-06-16 22:43 udp-client
-rw-r--r--  1 phil phil 3,8K 2005-02-02 04:05 udp-client.cc
-rwxr-xr-x  1 phil phil 16K 2005-06-16 22:45 udp-server
-rw-r--r--  1 phil phil 2,7K 2005-02-02 04:05 udp-server.cc

```

2.3. Instructions d'exécution

L'exécution du programme est aussi assez triviale. On peut exécuter le client et le serveur sur le même hôte dans deux *Shells* distincts et utiliser l'interface de boucle locale pour les communications réseau.

Le programme serveur, `udp-server.cc`

```

phil@nowhere:~/cpp/socket$ ./udp-server
Enter port number to listen on (between 1500 and 65000): 4000
Waiting for request on port 4000
  from 127.0.0.1:4968
  Received: texte de test

```

Le programme client, `udp-client.cc`

```

phil@nowhere:~/cpp/socket$ ./udp-client
Enter server host name or IP address: 127.0.0.1
Enter server port number: 4000

Enter some lines, and the server will modify them and
send them back.  When you are done, enter a line with
just a dot, and nothing else.
If a line is more than 100 characters, then
only the first 100 characters will be used.

Input: texte de test
Modified: TEXTE DE TEST
Input: .

```

2.4. Bibliothèques utilisées

Les deux programmes utilisent les mêmes fonctions disponibles à partir des bibliothèques standards.

`libc6-dev, netdb.h`

Opérations sur les bases de données réseau. Ici, c'est la fonction `gethostbyname()` qui est utilisée. Elle renvoie une structure de type `hostent` pour l'hôte `name`. La chaîne `name` est soit un nom d'hôte, soit une adresse IPv4 en notation pointée standard, soit une adresse IPv6 avec la notation points-virgules et points. Pour obtenir plus d'informations, il faut consulter les pages de manuels : **man** `gethostbyname`.

`libc6-dev, netinet/in.h`

Famille du protocole Internet. Ici, plusieurs fonctions sont utilisées à partir du paramètre de description de socket `sockaddr_in`. Les quatre fonctions importantes traitent de la conversion des nombres représentés suivant l'hôte (bit le moins significatif en premier sur processeur Intel x86) ou les définitions des en-têtes réseau (bit le plus significatif en premier).

- `htonl()` et `htons()` : conversion d'un entier long et d'un entier court depuis la représentation hôte (bit le moins significatif en premier ou *Least Significant Byte First*) vers la représentation réseau standard (bit le plus significatif en premier ou *Most Significant Byte First*).

- `ntohl()` et `ntohs()` : fonctions opposées aux précédentes. conversion de la représentation réseau vers la représentation hôte.

`libstdc++6-dev, iostream`

Opérations sur les flux d'entrées/sorties de base tels que l'écran et le clavier. Ici, toutes les opérations de saisie de nom d'hôte, d'adresse IP, de numéro de port ou de texte sont gérées à l'aide des fonctions usuelles du langage C++. Ces fonctions sont les seules qui soient spécifiques au langage C++.

D'une manière générale, toutes les fonctions sont documentées à l'aide des pages de manuels Unix classiques. Soit on entre directement à la console une commande du type : `man inet_ntoa`, soit on utilise l'aide du gestionnaire graphique pour accéder aux mêmes informations en saisissant une URL du type suivant à partir du gestionnaire de fichiers : `man:/inet_ntoa`.

2.5. Choix du protocole de transport

Au dessus du protocole de couche réseau IP, on doit choisir entre deux protocoles de couche transport : TCP ou UDP.

Le protocole TCP est le premier protocole développé. Il «porte la moitié» de la philosophie du modèle Internet. Cette philosophie veut que la couche transport soit le lieu de la fiabilisation des communications. Ce protocole fonctionne donc en mode connecté et contient tout les outils nécessaires à l'établissement, au maintien et à la libération de connexion. De plus, des numéros de séquences garantissent l'intégrité de la transmission et le fenêtrage de ces numéros de séquences assure un contrôle de congestion. Tout ces mécanismes ne sont pas évidents à maîtriser pour un public débutant.

Le protocole UDP a été développé après TCP. La philosophie de ce mode de transport suppose que le réseau de communication est intrinséquement fiable et qu'il n'est pas nécessaire de garantir l'intégrité des transmissions et de contrôler la congestion. On dit que le protocole UDP n'est pas orienté connexion ; ce qui a pour conséquence d'alléger considérablement les mécanismes de transport.

L'objectif du présent document étant d'initier à l'utilisation des sockets, on s'appuie sur le protocole de transport le plus simple : UDP. En termes de développement, les différences de mise en oeuvre des sockets sont minimales. C'est à l'analyse réseau que la différence se fait. UDP est plus facile à utiliser avec un public débutant.

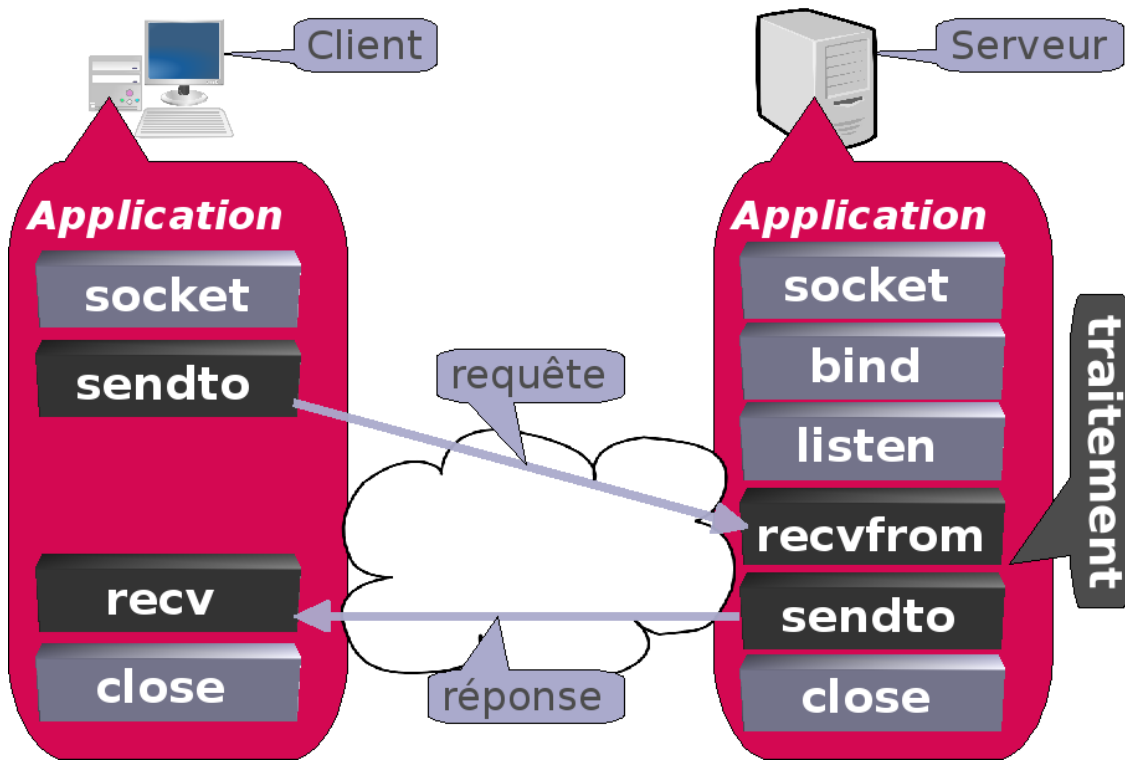
Pour plus d'informations, consulter le support *Modélisations réseau*.

2.6. Sockets & protocole de transport UDP

Le schéma ci-dessous présente les sous-programmes sélectionnés côté client et côté serveur pour la mise en oeuvre des sockets avec le protocole de transport UDP.

Les appels de sous-programmes avec les passages de paramètres sont détaillés dans les sections suivantes.

- Client : [Section 3.1, « Codage de l'utilisation des sockets »](#)
- Serveur : [Section 4.1, « Codage de l'utilisation des sockets »](#)



Fonctions utilisées avec UDP - image seule⁶

3. Programme client

3.1. Codage de l'utilisation des sockets

Au niveau du client, l'objectif est *d'ouvrir* un nouveau socket ; ce qui revient à ouvrir un canal de communication réseau avec la fonction `socket`.

```
int socketDescriptor;

<snipped/>
socketDescriptor = socket(AF_INET①, SOCK_DGRAM②, 0③);
if (socketDescriptor < 0) {
    cerr << "cannot create socket\n";
    exit(1);
}
```

- ① `AF_INET` spécifie que l'on utilise le protocole Internet de couche réseau IPv4.
- ② `SOCK_DGRAM` spécifie que l'on utilise le protocole de couche transport UDP. On émet des datagrammes courts directement au correspondant.
- ③ `0` ne spécifie aucun protocole de transport. Celui-ci est déterminé en fonction des paramètres précédents.

Une fois le canal de communication réseau correctement ouvert, on peut passer à l'émission des datagrammes avec la fonction `sendto`.

```
int socketDescriptor;
char buf[LINE_ARRAY_SIZE];
struct sockaddr_in serverAddress;

<snipped/>
if (sendto(socketDescriptor①, buf, strlen(buf)②, 0,
```

⁶ http://www.linux-france.org/prj/inetdoc/cours/socket.cpp/images/udp_socket.png

```

        (struct sockaddr *) &serverAddress,
        sizeof(serverAddress)❸) < 0) {
cerr << "cannot send data ";
close(socketDescriptor);
exit(1);
}

```

- ❶ socketDescriptor contient le résultat de l'appel de la fonction socket ; le numéro du canal de communication entre le programme et la pile des protocoles réseau.
- ❷ buf et strlen(buf) correspondent au datagramme et à sa longueur. Ici, on émet des chaînes de caractères directement vers le correspondant réseau.
- ❸ (struct sockaddr *) &serverAddress et sizeof(serverAddress) correspondent à la structure de désignation de l'adresse IP et du numéro de port du serveur puis à la taille de cette structure.

Ensuite, il ne manque plus que la description de la réception des datagrammes renvoyés par le serveur.

```

int numRead;

<snipped/>
numRead = recv(socketDescriptor❶, buf, MAX_LINE❷, 0);
if (numRead < 0) {
    cerr << "didn't get response from server?";
    close(socketDescriptor);
    exit(1);
}

```

- ❶ socketDescriptor contient le résultat de l'appel de la fonction socket ; le numéro du canal de communication entre le programme et la pile des protocoles réseau.
- ❷ buf et MAX_LINE correspondent au datagramme reçu et à sa longueur. Ici, on reçoit des chaînes de caractères venant directement du correspondant réseau.

Pour toute information complémentaire sur les fonctions utilisées, consulter les pages de manuels correspondantes. Pour la fonction socket on peut utiliser **man 2 socket** ou **man 7 socket** par exemple.

3.2. Code source complet

Code du programme udp-client.cc :

```

#include <netdb.h>
#include <netinet/in.h>
#include <unistd.h>
#include <iostream>

#define MAX_LINE 100
// 3 caractères pour les codes ASCII 'cr', 'lf' et '\0'
#define LINE_ARRAY_SIZE (MAX_LINE+3)

using namespace std;

int main()
{
    int socketDescriptor;
    int numRead;
    unsigned short int serverPort;
    struct sockaddr_in serverAddress;
    struct hostent *hostInfo;
    struct timeval timeVal;
    fd_set readSet;
    char buf[LINE_ARRAY_SIZE], c;

    cout << "Enter server host name or IP address: ";

```

```

memset(buf, 0x0, LINE_ARRAY_SIZE); // Mise à zéro du tampon
cin.get(buf, MAX_LINE, '\n');

// gethostbyname() reçoit un nom d'hôte ou une adresse IP en notation
// standard 4 octets en décimal séparés par des points puis renvoie un
// pointeur sur une structure hostent. Nous avons besoin de cette structure
// plus loin. La composition de cette structure n'est pas importante pour
// l'instant.
hostInfo = gethostbyname(buf);
if (hostInfo == NULL) {
    cout << "problem interpreting host: " << buf << "\n";
    exit(1);
}

cout << "Enter server port number: ";
cin >> serverPort;
cin.ignore(1, '\n'); // suppression du saut de ligne

// Création de socket. "AF_INET" correspond à l'utilisation du protocole IPv4
// au niveau réseau. "SOCK_DGRAM" correspond à l'utilisation du protocole UDP
// au niveau transport. La valeur 0 indique qu'un seul protocole sera utilisé
// avec ce socket.
socketDescriptor = socket(AF_INET, SOCK_DGRAM, 0);
if (socketDescriptor < 0) {
    cerr << "cannot create socket\n";
    exit(1);
}

// Initialisation des champs de la structure serverAddress
serverAddress.sin_family = hostInfo->h_addrtype;
memcpy((char *) &serverAddress.sin_addr.s_addr,
        hostInfo->h_addr_list[0], hostInfo->h_length);
serverAddress.sin_port = htons(serverPort);

cout << "\nEntrez quelques caractères au clavier.\n";
cout << "Le serveur les modifiera et les renverra.\n";
cout << "Pour sortir, entrez une ligne avec le caractère '.' uniquement\n";
cout << "Si une ligne dépasse " << MAX_LINE << " caractères,\n";
cout << "seuls les " << MAX_LINE << " premiers caractères seront utilisés.\n\n";

// Invite de commande pour l'utilisateur et lecture des caractères jusqu'à la
// limite MAX_LINE. Puis suppression du saut de ligne.
cout << "Input: ";
memset(buf, 0x0, LINE_ARRAY_SIZE); // Zero out the buffer.
cin.get(buf, MAX_LINE, '\n');
// suppression des caractères supplémentaires et du saut de ligne
cin.ignore(1000, '\n');

// Arrêt lorsque l'utilisateur saisit une ligne ne contenant qu'un point
while (strcmp(buf, ".") != 0) {
    // Envoi de la ligne au serveur
    if (sendto(socketDescriptor, buf, strlen(buf), 0,
               (struct sockaddr *) &serverAddress,
               sizeof(serverAddress)) < 0) {
        cerr << "cannot send data ";
        close(socketDescriptor);
        exit(1);
    }
}

// Attente de la réponse pendant une seconde.
FD_ZERO(&readSet);

```

```

FD_SET(socketDescriptor, &readSet);
timeVal.tv_sec = 1;
timeVal.tv_usec = 0;

if (select(socketDescriptor+1, &readSet, NULL, NULL, &timeVal)) {
    // Lecture de la ligne modifiée par le serveur.
    memset(buf, 0x0, LINE_ARRAY_SIZE); // Mise à zéro du tampon
    numRead = recv(socketDescriptor, buf, MAX_LINE, 0);
    if (numRead < 0) {
        cerr << "didn't get response from server?";
        close(socketDescriptor);
        exit(1);
    }

    cout << "Modified: " << buf << "\n";
}
else {
    cout << "*** Server did not respond in 1 second.\n";
}

// Invite de commande pour l'utilisateur et lecture des caractères jusqu'à la
// limite MAX_LINE. Puis suppression du saut de ligne. Comme ci-dessus.
cout << "Input: ";
memset(buf, 0x0, LINE_ARRAY_SIZE); // Mise à zéro du tampon
cin.get(buf, MAX_LINE, '\n');
// suppression des caractères supplémentaires et du saut de ligne
cin.ignore(1000, '\n');
}

close(socketDescriptor);
return 0;
}

```

4. Programme serveur

4.1. Codage de l'utilisation des sockets

Au niveau du serveur, l'objectif est aussi *d'ouvrir* un nouveau socket ; ce qui revient aussi à ouvrir un canal de communication réseau avec la fonction `socket`.

```

int listenSocket;

<snipped/>
listenSocket = socket(AF_INET, SOCK_DGRAM, 0);
if (listenSocket < 0) {
    cerr << "cannot create listen socket";
    exit(1);
}

```

Cette étape ne présentant aucune différence avec le niveau client, on relie le numéro de socket avec le numéro de port choisi.

```

int listenSocket;
struct sockaddr_in serverAddress;

<snipped/>
serverAddress.sin_family = AF_INET;
serverAddress.sin_addr.s_addr = htonl(INADDR_ANY)❶;
serverAddress.sin_port = htons(listenPort);

```

```

if (bind(listenSocket❷,
        (struct sockaddr *) &serverAddress,
        sizeof(serverAddress)❸) < 0) {
    cerr << "cannot bind socket";
    exit(1);
}

```

- ❶ INADDR_ANY spécifie que le programme est en écoute sur toutes les adresses IP sources.
- ❷ listenSocket contient le résultat de l'appel de la fonction `socket` ; le numéro du canal de communication entre le programme et la pile des protocoles réseau.
- ❸ `(struct sockaddr *) &serverAddress` et `sizeof(serverAddress)` correspondent à la structure de désignation de l'adresse IP et du numéro de port du serveur puis à la taille de cette structure.

Une fois la liaison établie, le programme attend les datagrammes provenant du client.

```

int listenSocket;
struct sockaddr_in clientAddress;
char line[(MAX_MSG+1)];

<snipped/>
listen(listenSocket, 5)❶;

<snipped/>
if (recvfrom(listenSocket, line, MAX_MSG❷, 0,
            (struct sockaddr *) &clientAddress,
            &clientAddressLength) < 0) {
    cerr << " I/O Problem";
    exit(1);
}

```

- ❶ La fonction `listen` «active» l'utilisation du canal de communication initié avec la fonction `socket`. Le second paramètre 5 définit une longueur maximale pour la file des connexions en attente.
- ❷ Les paramètres `line` et `MAX_MSG` correspondent au datagramme et à sa longueur.

Enfin, les émissions de datagramme du serveur vers le client utilisent exactement les mêmes appels à la fonction `sendto` que les émissions du client vers le serveur.

4.2. Code source complet

Code du programme `udp-server.cc` :

```

#include <arpa/inet.h>

#include <netdb.h>
#include <netinet/in.h>
#include <unistd.h>
#include <iostream>

#define MAX_LINE 100
// 3 caractères pour les codes ASCII 'cr', 'lf' et '\0'
#define LINE_ARRAY_SIZE (MAX_LINE+3)

using namespace std;

int main()
{
    int listenSocket, i;
    unsigned short int listenPort;
    socklen_t clientAddressLength;
    struct sockaddr_in clientAddress, serverAddress;

```

```

char buf[LINE_ARRAY_SIZE];

memset(buf, 0x0, LINE_ARRAY_SIZE); // Mise à zéro du tampon

cout << "Enter port number to listen on (between 1500 and 65000): ";
cin >> listenPort;

// Création de socket en écoute et attente des requêtes des clients
listenSocket = socket(AF_INET, SOCK_DGRAM, 0);
if (listenSocket < 0) {
    cerr << "cannot create listen socket";
    exit(1);
}

// Connexion du socket au port en écoute.
// On commence par initialiser les champs de la structure serverAddress puis
// on appelle bind(). Les fonctions htonl() et htons() convertissent
// respectivement les entiers longs et les entiers courts du rangement hôte
// (sur x86 on trouve l'octet de poids faible en premier) vers le rangement
// réseau (octet de poids fort en premier).
serverAddress.sin_family = AF_INET;
serverAddress.sin_addr.s_addr = htonl(INADDR_ANY);
serverAddress.sin_port = htons(listenPort);

if (bind(listenSocket,
        (struct sockaddr *) &serverAddress,
        sizeof(serverAddress)) < 0) {
    cerr << "cannot bind socket";
    exit(1);
}

// Attente des requêtes clients.
// C'est un appel non-bloquant ; c'est-à-dire qu'il enregistre ce programme
// auprès du système comme devant attendre des connexions sur ce socket avec
// cette tâche. Puis, l'exécution se poursuit.
listen(listenSocket, 5);

cout << "Waiting for request on port " << listenPort << "\n";

while (1) {

    clientAddressLength = sizeof(clientAddress);

    // Mise à zéro du tampon de façon à connaître le délimiteur
    // de fin de chaîne.
    memset(buf, 0x0, LINE_ARRAY_SIZE);
    if (recvfrom(listenSocket, buf, LINE_ARRAY_SIZE, 0,
                (struct sockaddr *) &clientAddress,
                &clientAddressLength) < 0) {
        cerr << " I/O Problem";
        exit(1);
    }

    // Affichage de l'adresse IP du client.
    cout << " from " << inet_ntoa(clientAddress.sin_addr);

    // Affichage du numéro de port du client.
    cout << ":" << ntohs(clientAddress.sin_port) << "\n";

    // Affichage de la ligne reçue
    cout << " Received: " << buf << "\n";
}

```

```

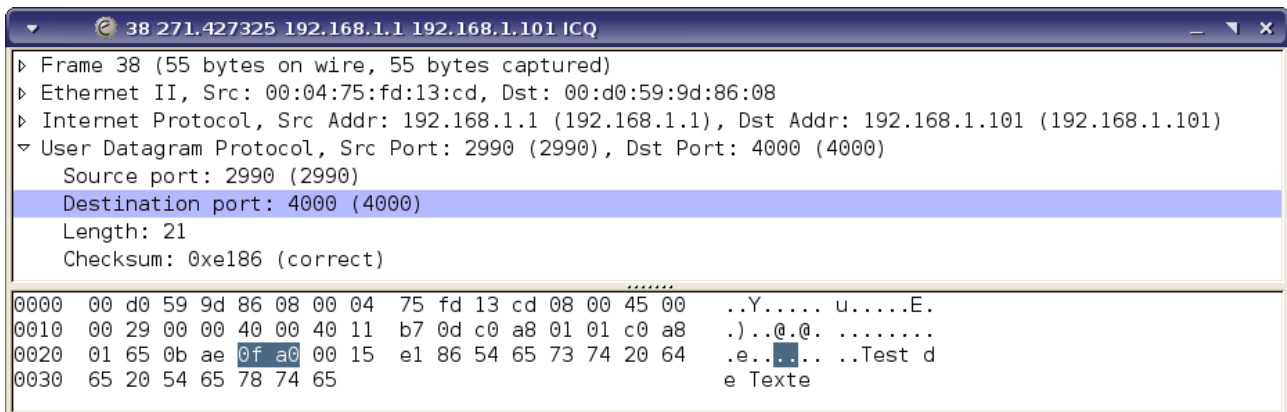
// Conversion de cette ligne en majuscules.
for (i = 0; buf[i] != '\0'; i++)
    buf[i] = toupper(buf[i]);

// Renvoi de la ligne convertie au client.
if (sendto(listenSocket, buf, strlen(buf) + 1, 0,
           (struct sockaddr *) &clientAddress,
           sizeof(clientAddress)) < 0)
    cerr << "Error: cannot send modified data";
}
}

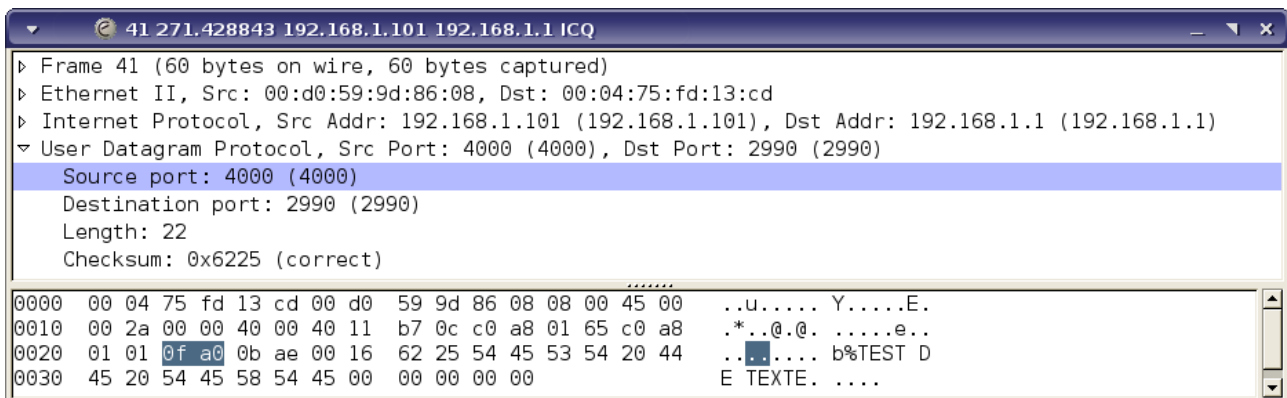
```

5. Analyse réseau

Voici deux captures réseau d'échange de datagramme entre un client avec l'adresse IP 192.168.1.1 et un serveur avec l'adresse IP 192.168.1.101. Le serveur est en *écoute* sur le port 4000.



Échange de datagramme entre le client et le serveur - image seule⁷



Échange de datagramme entre le serveur et le client - image seule⁸

6. Documents de référence

A Brief Socket Tutorial

*Brief Socket Tutorial*⁹ : support proposant des exemples de programmes de communication réseau basés sur les sockets. Le présent document est *très fortement inspiré* des exemples utilisant le protocole de transport UDP.

⁷ <http://www.linux-france.org/prj/inetdoc/cours/socket.cpp/images/client2server.png>

⁸ <http://www.linux-france.org/prj/inetdoc/cours/socket.cpp/images/server2client.png>

⁹ <http://sage.mc.yu.edu/kbeen/teaching/networking/resources/sockets.html>

Beej's Guide to Network Programming

*Beej's Guide to Network Programming*¹⁰ : support très complet sur les sockets proposant de nombreux exemples de programmes.

Modélisations réseau

*Modélisations réseau*¹¹ : présentation et comparaison des modélisations OSI et Internet.

Adressage IPv4

*Adressage IPv4*¹² : support complet sur l'adressage du protocole de couche réseau de l'Internet (IP).

Configuration d'une interface réseau

*Configuration d'une interface de réseau local*¹³ : support sur la configuration des interfaces réseau ; notamment les explications sur les opérations «rituelles» de début de travaux pratiques :

```
# /etc/init.d/networking stop
# ifconfig lo up
# ifconfig eth0 192.168.0.2 netmask 255.255.255.240
# route add default gw 192.168.0.1
# ping 192.168.0.1
# ping 172.16.80.1
# ping www.cict.fr
```

¹⁰ <http://beej.us/guide/bgnet/>

¹¹ <http://www.linux-france.org/prj/inetdoc/articles/modelisation/>

¹² <http://www.linux-france.org/prj/inetdoc/articles/adressage.ipv4/>

¹³ <http://www.linux-france.org/prj/inetdoc/cours/config.interface.lan/>