

Initiation au développement C/C++ sur les sockets

Philippe Latu

philippe.latu(at)linux-france.org

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

Historique des versions		
\$Revision: 1619 \$	\$Date: 2011-04-05 00:08:00 +0200 (mar. 05 avril 2011) \$	\$Author: latu \$
Année universitaire 2010-2011		
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 fonctions réseau des bibliothèques standard du Langage C les sockets et le langage C++ pour les entrées sorties écran/clavier.		

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 premier protocole de transport étudié	4
2.6. Sockets & protocole de transport UDP	4
3. Programme client UDP	5
3.1. Codage de l'utilisation des sockets	5
3.2. Code source complet	6
4. Programme serveur UDP	7
4.1. Codage de l'utilisation des sockets	7
4.2. Code source complet	8
5. Programme client TCP	10
5.1. Codage de l'utilisation des sockets	10
5.2. Patch code source	10
6. Programme serveur TCP	11
6.1. Codage de l'utilisation des sockets	11
6.2. Patch code source	12
7. Analyse réseau	13
8. Documents de référence	14

1. Copyright et Licence

Copyright (c) 2000,2011 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) 2000,2011 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

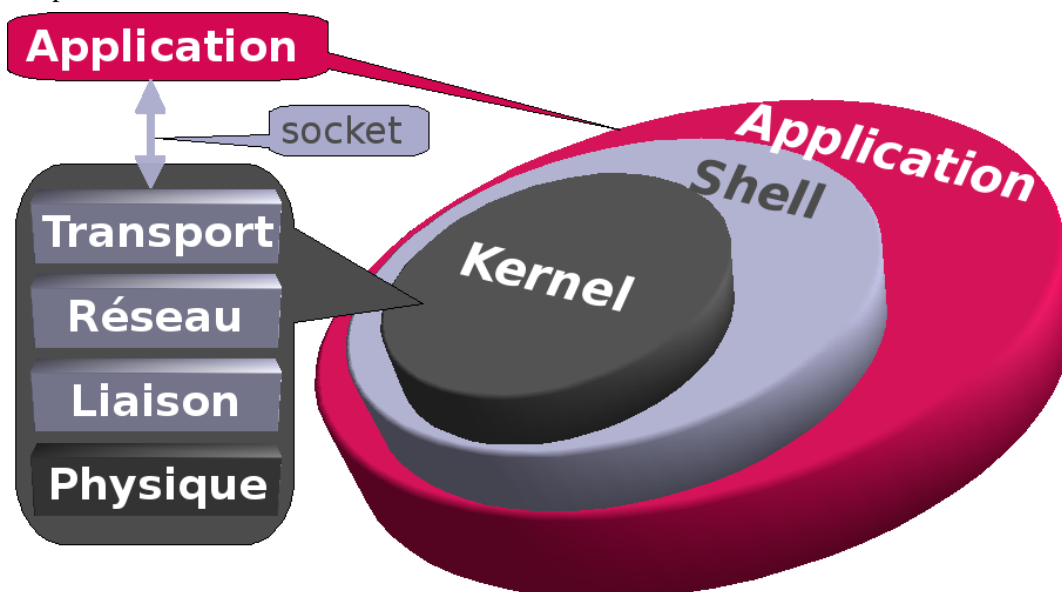
Cet article est écrit avec *DocBook*¹ XML sur un système *Debian GNU/Linux*². Il est disponible en version imprimable aux formats PDF et Postscript : [socket.cpp.pdf](#)³ | [socket.cpp.ps.gz](#)⁴.

2. Contexte de développement

L'objectif de développement étant l'initiation, on se limite à un code minimaliste utilisant deux programmes distincts : un serveur et un client. Ces deux programmes échangent des chaînes caractères. Le «client» émet un message que le «serveur» traite et retransmet vers le «client». Le traitement est tout aussi minimaliste. Il s'agit de convertir la chaîne de caractères en majuscules.

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 sockets 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

Il est possible de compiler ces deux programmes en l'état sur n'importe quel système GNU/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

```
$ g++ -o udp-client udp-client.cc
$ g++ -o udp-server udp-server.cc
$ ls -l udp*
-rwxr-xr-x 1 phil phil 12217  3 mars  16:29 udp-client
-rw-r--r-- 1 phil phil  3853  3 mars  16:29 udp-client.cc
-rwxr-xr-x 1 phil phil 11084  3 mars  16:30 udp-server
-rw-r--r-- 1 phil phil  2744  3 mars  16:29 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`

```
$ ./udp-server
Entrez le numéro de port utilisé en écoute (entre 1500 et 65000) : 4000
Attente de requête sur le port 4000
  Depuis 127.0.0.1:60471
Message reçu : texte du message
```

Le programme client, `udp-client.cc`

```
$ ./udp-client
Entrez le nom du serveur ou son adresse IP : 127.0.0.1
Entrez le numéro de port du serveur : 4000

Entrez quelques caractères au clavier.
Le serveur les modifiera et les renverra.
Pour sortir, entrez une ligne avec le caractère '.' uniquement
Si une ligne dépasse 100 caractères,
seuls les 100 premiers caractères seront utilisés.

Saisie du message : texte du message
Message traité : TEXTE DU MESSAGE
Saisie du message : .
```

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 (octet le moins significatif en premier sur processeur Intel x86) ou les définitions des en-têtes réseau (octet le plus significatif en premier).

- `htonl()` et `htons()` : conversion d'un entier long et d'un entier court depuis la représentation hôte (octet le moins significatif en premier ou *Least Significant Byte First*) vers la représentation réseau standard (octet 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 premier protocole de transport étudié

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

Dans l'ordre chronologique, le protocole TCP est le premier protocole à avoir été 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 flux. Tout ces mécanismes ne sont pas évidents à appréhender 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 intrinsèquement fiable et qu'il n'est pas nécessaire de garantir l'intégrité des transmissions et de contrôler les flux. 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 dans un premier temps sur le protocole de transport le plus simple : UDP. Les programmes «client» et «serveur» sont repris dans un second temps en utilisant le protocole TCP. En termes de développement, les différences de mise en œuvre des sockets sont minimales. C'est à l'analyse réseau que la différence se fait sachant que les mécanismes de fonctionnement des deux protocoles sont très différents.

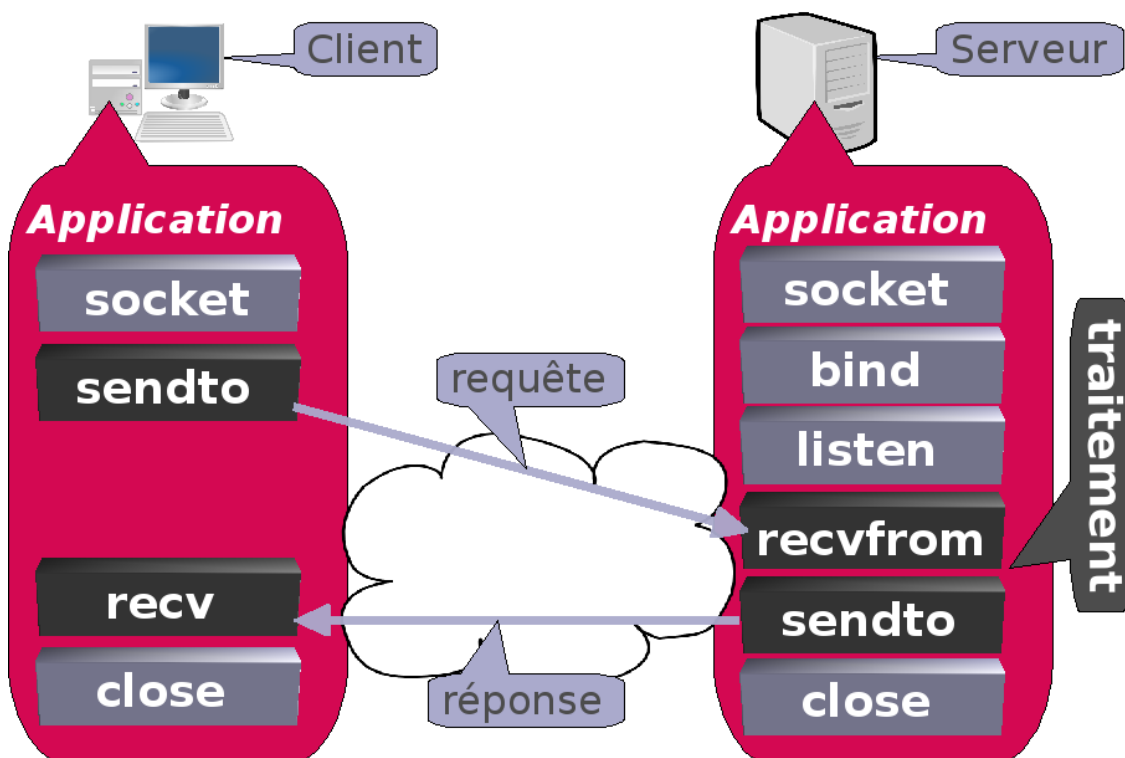
Pour plus d'informations, consulter le support *Modélisations réseau*.

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 œuvre 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 UDP

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_INET❶, SOCK_DGRAM❷, 0❸);
if (socketDescriptor < 0) {
    cerr << "Impossible de créer le socket\n";
    exit(1);
}
```

- ❶ `AF_INET` spécifie que l'on utilise le protocole Internet de couche réseau IPv4.
- ❷ `SOCK_DGRAM` spécifie que l'on utilise le protocole de couche transport UDP. On émet des datagrammes courts directement au correspondant.
- ❸ `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 msg[MSG_ARRAY_SIZE];
struct sockaddr_in serverAddress;

<snipped/>
if (sendto(socketDescriptor❶, msg, strlen(msg)❷, 0,
          (struct sockaddr *) &serverAddress,
          sizeof(serverAddress)❸) < 0) {
    cerr << "Émission du message impossible\n";
    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 `msg` et `strlen(msg)` 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, msg, MAX_MSG❷, 0);
if (numRead < 0) {
    cerr << "Aucune réponse du serveur ?\n";
    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.

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

- ② `msg` et `MAX_MSG` 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` :

```
// $Id: udp-client.cc 1598 2011-02-24 20:55:11Z latu $
#include <netdb.h>
#include <netinet/in.h>
#include <unistd.h>
#include <limits>
#include <iostream>
#include <cstdlib>
#include <cstring>

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

using namespace std;

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

    cout << "Entrez le nom du serveur ou son adresse IP : ";

    memset(msg, 0x0, MSG_ARRAY_SIZE); // Mise à zéro du tampon
    cin.getline(msg, MAX_MSG);

    // 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(msg);
    if (hostInfo == NULL) {
        cerr << "Problème dans l'interprétation des informations d'hôte : " << msg << "\n";
        exit(1);
    }

    cout << "Entrez le numéro de port du serveur : ";
    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 << "Impossible de créer le 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 << endl
<< "Entrez quelques caractères au clavier." << endl
<< "Le serveur les modifiera et les renverra." << endl
<< "Pour sortir, entrez une ligne avec le caractère '.' uniquement" << endl
<< "Si une ligne dépasse " << MAX_MSG << " caractères," << endl
<< "seuls les " << MAX_MSG << " premiers caractères seront utilisés." << endl
<< endl;

// Invite de commande pour l'utilisateur et lecture des caractères jusqu'à la
// limite MAX_MSG. Puis suppression du saut de ligne.
cout << "Saisie du message : ";
memset(msg, 0x0, MSG_ARRAY_SIZE); // Mise à zéro du tampon
cin.getline(msg, MAX_MSG);

// Arrêt lorsque l'utilisateur saisit une ligne ne contenant qu'un point
while (strcmp(msg, ".") {
    if ((msgLength = strlen(msg)) > 0) {
        // Envoi de la ligne au serveur
        if (sendto(socketDescriptor, msg, msgLength, 0,
            (struct sockaddr *) &serverAddress,
            sizeof(serverAddress)) < 0) {
            cerr << "Émission du message impossible\n";
            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(msg, 0x0, MSG_ARRAY_SIZE); // Mise à zéro du tampon
            if (recv(socketDescriptor, msg, MAX_MSG, 0) < 0) {
                cerr << "Aucune réponse du serveur ?\n";
                close(socketDescriptor);
                exit(1);
            }

            cout << "Message traité : " << msg << "\n";
        }
        else {
            cout << "*** Le serveur n'a répondu dans la seconde.\n";
        }
    }
    // Invite de commande pour l'utilisateur et lecture des caractères jusqu'à la
    // limite MAX_MSG. Puis suppression du saut de ligne. Comme ci-dessus.
    cout << "Saisie du message : ";
    memset(msg, 0x0, MSG_ARRAY_SIZE); // Mise à zéro du tampon
    cin.getline(msg, MAX_MSG);
}

close(socketDescriptor);
return 0;
}

```

4. Programme serveur UDP

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 << "Impossible de créer le socket en écoute\n";
    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 << "Impossible de lier le socket en écoute\n";
    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 en place, 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 << " Problème de réception du message\n";
    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 :

```

// $Id: udp-server.cc 1598 2011-02-24 20:55:11Z latu $
#include <arpa/inet.h>
#include <netdb.h>
#include <netinet/in.h>
#include <unistd.h>
#include <iostream>
#include <cstdlib>
#include <cstring>

```

```

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

using namespace std;

int main()
{
    int listenSocket, i;
    unsigned short int listenPort, msgLength;
    socklen_t clientAddressLength;
    struct sockaddr_in clientAddress, serverAddress;
    char msg[MSG_ARRAY_SIZE];

    cout << "Entrez le numéro de port utilisé en écoute (entre 1500 et 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 << "Impossible de créer le socket en écoute\n";
        exit(1);
    }

    // On relie le 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 << "Impossible de lier le socket en écoute\n";
        exit(1);
    }

    // Attente des requêtes des 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. Ensuite, l'exécution se poursuit.
    listen(listenSocket, 5);

    cout << "Attente de requête sur le port " << listenPort << endl;

    while (1) {

        clientAddressLength = sizeof(clientAddress);

        // Mise à zéro du tampon de façon à connaître le délimiteur
        // de fin de chaîne.
        memset(msg, 0x0, MSG_ARRAY_SIZE);
        if (recvfrom(listenSocket, msg, MSG_ARRAY_SIZE, 0,
                    (struct sockaddr *) &clientAddress,
                    &clientAddressLength) < 0) {
            cerr << " Problème de réception du message\n";
            exit(1);
        }

        msgLength = strlen(msg);
        if (msgLength > 0) {
            // Affichage de l'adresse IP du client.
            cout << " Depuis " << inet_ntoa(clientAddress.sin_addr);

            // Affichage du numéro de port du client.

```

```

    cout << ":" << ntohs(clientAddress.sin_port) << endl;

    // Affichage de la ligne reçue
    cout << " Message reçu : " << msg << endl;

    // Conversion de cette ligne en majuscules.
    for (i = 0; i < msgLength; i++)
        msg[i] = toupper(msg[i]);

    // Renvoi de la ligne convertie au client.
    if (sendto(listenSocket, msg, msgLength + 1, 0,
              (struct sockaddr *) &clientAddress,
              sizeof(clientAddress)) < 0)
        cerr << "Émission du message modifié impossible\n";
    }
}
}

```

5. Programme client TCP

5.1. Codage de l'utilisation des sockets

Le fait que le protocole TCP soit un *service orienté connexion* entraîne un changement important dans le code source du programme client précédent. Le contrôle d'erreur est directement intégré dans la couche transport et n'est plus à la charge de la couche application. Il n'est donc plus nécessaire de mettre en œuvre un mécanisme de gestion de temporisation.

Autre changement, il est maintenant nécessaire d'établir la connexion avant d'échanger la moindre information. Cette opération se fait à l'aide de la fonction `connect`.

```

int socketDescriptor;
struct sockaddr_in serverAddress;

<snipped/>
if (connect(socketDescriptor,
          (struct sockaddr *) &serverAddress,
          sizeof(serverAddress)) < 0) {
    cerr << "Connexion impossible\n";
    close(socketDescriptor);
    exit(1);
}

```

En cas d'échec de cette demande d'établissement de connexion, on abandonne le traitement.

5.2. Patch code source

Patch du programme `tcp-client.cc` :

```

--- udp-client.cc 2011-02-24 21:58:45.902626086 +0100
+++ tcp-client.cc 2011-02-24 21:58:45.902626086 +0100
@@ -1,4 +1,4 @@
-// $Id: udp-client.cc 1598 2011-02-24 20:55:11Z latu $
+// $Id: tcp-client.cc 1598 2011-02-24 20:55:11Z latu $
#include <netdb.h>
#include <netinet/in.h>
#include <unistd.h>
@@ -20,8 +20,6 @@
    unsigned short int serverPort;
    struct sockaddr_in serverAddress;
    struct hostent *hostInfo;
-   struct timeval timeVal;
-   fd_set readSet;
    char msg[MSG_ARRAY_SIZE], c;

    cout << "Entrez le nom du serveur ou son adresse IP : ";
@@ -48,7 +46,7 @@
    // 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);

```

```

+ socketDescriptor = socket(AF_INET, SOCK_STREAM, 0);
+ if (socketDescriptor < 0) {
+     cerr << "Impossible de créer le socket\n";
+     exit(1);
@@ -60,6 +58,14 @@
+     hostInfo->h_addr_list[0], hostInfo->h_length);
+     serverAddress.sin_port = htons(serverPort);

+ if (connect(socketDescriptor,
+             (struct sockaddr *) &serverAddress,
+             sizeof(serverAddress)) < 0) {
+     cerr << "Connexion impossible\n";
+     close(socketDescriptor);
+     exit(1);
+ }
+
+ cout << endl
+ << "Entrez quelques caractères au clavier." << endl
+ << "Le serveur les modifiera et les renverra." << endl
@@ -86,13 +92,6 @@
+     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(msg, 0x0, MSG_ARRAY_SIZE); // Mise à zéro du tampon
+ if (recv(socketDescriptor, msg, MAX_MSG, 0) < 0) {
@@ -100,13 +99,10 @@
+     close(socketDescriptor);
+     exit(1);
+ }
+ }

+     cout << "Message traité : " << msg << "\n";
- }
- else {
-     cout << "*** Le serveur n'a répondu dans la seconde.\n";
- }
+ }
+
+ // Invite de commande pour l'utilisateur et lecture des caractères jusqu'à la
+ // limite MAX_MSG. Puis suppression du saut de ligne. Comme ci-dessus.
+ cout << "Saisie du message : ";

```

6. Programme serveur TCP

6.1. Codage de l'utilisation des sockets

Comme dans le cas du programme client, le fait que le protocole TCP soit un *service orienté connexion* entraîne un changement important dans le code source du programme serveur précédent. Le contrôle d'erreur est directement intégré dans la couche transport.

Si le client fait appel à la fonction `connect` pour demander l'établissement d'une connexion, le serveur fait appel à la fonction `accept` pour recevoir les nouvelles demandes de connexion.

```

int socketDescriptor;
struct sockaddr_in serverAddress;

<snipped/>
connectSocket = accept(listenSocket,
                      (struct sockaddr *) &clientAddress,
                      &clientAddressLength);

```

```

if (connectSocket < 0) {
    cerr << "Impossible d'accepter une connexion\n";
    close(listenSocket);
    exit(1);
}

```

En cas d'échec de l'ouverture du socket de réception des demandes d'établissement de connexion, on abandonne le traitement.

6.2. Patch code source

Patch du programme `tcp-server.cc` :

```

--- udp-server.cc 2011-02-24 21:58:45.902626086 +0100
+++ tcp-server.cc 2011-02-24 21:58:45.905959630 +0100
@@ -1,4 +1,4 @@
-// $Id: udp-server.cc 1598 2011-02-24 20:55:11Z latu $
+// $Id: tcp-server.cc 1598 2011-02-24 20:55:11Z latu $
#include <arpa/inet.h>
#include <netdb.h>
#include <netinet/in.h>
@@ -15,7 +15,7 @@

int main()
{
- int listenSocket, i;
+ int listenSocket, connectSocket, i;
  unsigned short int listenPort, msgLength;
  socklen_t clientAddressLength;
  struct sockaddr_in clientAddress, serverAddress;
@@ -25,7 +25,7 @@
  cin >> listenPort;

  // Création de socket en écoute et attente des requêtes des clients
- listenSocket = socket(AF_INET, SOCK_DGRAM, 0);
+ listenSocket = socket(AF_INET, SOCK_STREAM, 0);
  if (listenSocket < 0) {
    cerr << "Impossible de créer le socket en écoute\n";
    exit(1);
@@ -54,42 +54,61 @@
  // cette tâche. Ensuite, l'exécution se poursuit.
  listen(listenSocket, 5);

- cout << "Attente de requête sur le port " << listenPort << endl;
-
+ while (1) {
+   cout << "Attente de connexion TCP sur le port " << listenPort << endl;

+   // On accepte une connexion avec un client qui en demande une. L'appel à la
+   // fonction accept() est bloquant ; c'est-à-dire que le processus est
+   // arrêté jusqu'à l'arrivée d'une demande de connexion.
+   // connectSocket est un nouveau socket que le système fournit séparément du
+   // socket d'écoute listenSocket. Il serait possible d'accepter des
+   // connexions supplémentaires reçues par listenSocket avant que
+   // connectSocket soit clos ; mais ce programme ne fonctionne pas de cette
+   // façon.
    clientAddressLength = sizeof(clientAddress);

-   // Mise à zéro du tampon de façon à connaître le délimiteur
-   // de fin de chaîne.
    memset(msg, 0x0, MSG_ARRAY_SIZE);
-   if (recvfrom(listenSocket, msg, MSG_ARRAY_SIZE, 0,
+   connectSocket = accept(listenSocket,
        (struct sockaddr *) &clientAddress,
-        &clientAddressLength) < 0) {
-     cerr << " Problème de réception du message\n";
+     &clientAddressLength);
+
+   if (connectSocket < 0) {
+     cerr << "Impossible d'accepter une connexion\n";

```

```

+     close(listenSocket);
+     exit(1);
+ }

-     msgLength = strlen(msg);
-     if (msgLength > 0) {
-         // Affichage de l'adresse IP du client.
-         cout << " Depuis " << inet_ntoa(clientAddress.sin_addr);
-
-         // Affichage du numéro de port du client.
+         // inet_ntoa() convertit une adresse IP de la forme binaire à la forme
+         // standard avec quatre nombres séparés par des points.
+         cout << " connecté à " << inet_ntoa(clientAddress.sin_addr);
+
+         // Affichage du numéro de port client
+         // ntohs() convertit un entier court (short) de l'agencement réseau (octet
+         // de poids fort en premier) vers l'agencement hôte (sur x86 on trouve
+         // l'octet de poids faible en premier).
+         cout << ":" << ntohs(clientAddress.sin_port) << endl;

-         // Affichage de la ligne reçue
-         cout << " Message reçu : " << msg << endl;
+         // Lecture de la chaîne sur le socket en utilisant recv(). La chaîne est
+         // stockée dans le tableau msg. Si aucun message n'arrive, recv() reste en
+         // attente.
+         // On remplit le tableau avec des zéros de façon à connaître la fin de
+         // chaîne de caractères
+         memset(msg, 0x0, MSG_ARRAY_SIZE);
+         while (recv(connectSocket, msg, MAX_MSG, 0) > 0) {
+             msgLength = strlen(msg);
+             if (msgLength > 0) {
+                 cout << " -- " << msg << "\n";

+                 // Conversion de cette ligne en majuscules.
+                 for (i = 0; i < msgLength; i++)
+                     msg[i] = toupper(msg[i]);

+                 // Renvoi de la ligne convertie au client.
-                 if (sendto(listenSocket, msg, msgLength + 1, 0,
-                     (struct sockaddr *) &clientAddress,
-                     sizeof(clientAddress)) < 0)
+                 if (send(connectSocket, msg, msgLength + 1, 0) < 0)
+                     cerr << "Émission du message modifié impossible\n";
+
+                 memset(msg, 0x0, MSG_ARRAY_SIZE); // Mise à zéro du tampon
+             }
+         }
+     }
+ }

```

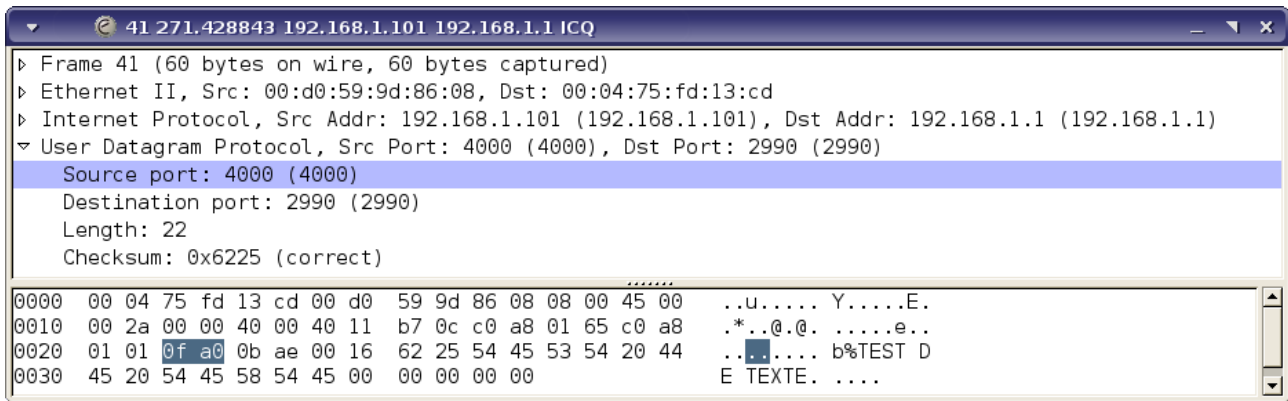
7. Analyse réseau

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.

```

38 271.427325 192.168.1.1 192.168.1.101 ICQ
  Frame 38 (55 bytes on wire, 55 bytes captured)
  Ethernet II, Src: 00:04:75:fd:13:cd, Dst: 00:d0:59:9d:86:08
  Internet Protocol, Src Addr: 192.168.1.1 (192.168.1.1), Dst Addr: 192.168.1.101 (192.168.1.101)
  User Datagram Protocol, Src Port: 2990 (2990), Dst Port: 4000 (4000)
    Source port: 2990 (2990)
    Destination port: 4000 (4000)
    Length: 21
    Checksum: 0xe186 (correct)
    ..Y.... u....E.
    )..@. ....
    .e... ..Test d
    e Texte

```

Échange de datagramme entre le client et le serveur - image seule⁷Échange de datagramme entre le serveur et le client - image seule⁸

8. 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.

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. Il permet notamment de relever les adresses IP des hôtes en communication.

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://web.archive.org/web/20080703122104/http://sage.mc.yu.edu/kbeen/teaching/networking/resources/sockets.html>

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/>