Initiation au développement C sur les sockets

Philippe Latu
philippe.latu(at)inetdoc.net

http://www.inetdoc.net

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 et on se limite à l'utilisation d'adresses IPv4 en couche réseau.

Table des matières

1. Copyright et Licence

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 au format PDF : `socket-c.pdf`[3].

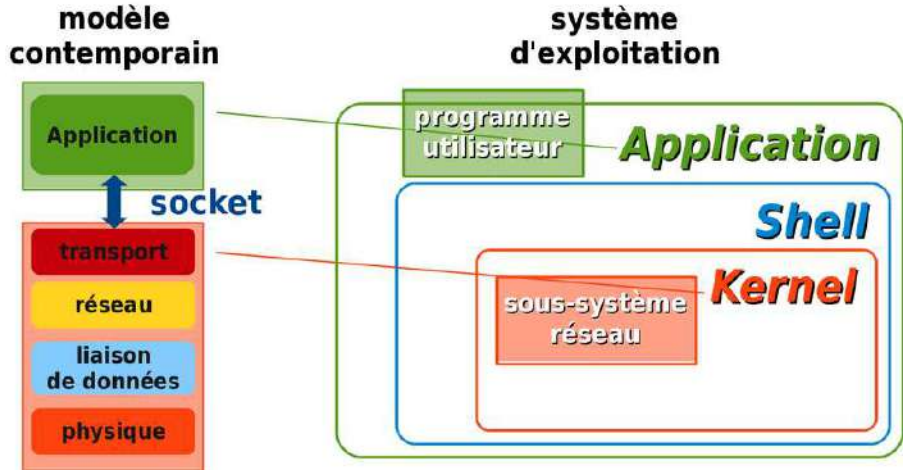
[1] http://www.docbook.org
[2] http://www.debian.org
[3] http://www.inetdoc.net/pdf/socket-c.pdf

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 convertit la chaîne de caractères en majuscules.

2.1. Système d'exploitation

Le schéma ci-dessous permet de faire la correspondance entre les couches de la modélisation contemporaine et celles de la représentation macroscopique d'un système d'exploitation.



Le logiciel correspondant aux protocoles allant de la couche physique jusqu'à la couche transport fait 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 les deux programmes *client* et *serveur* 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`.

```
$ gcc -Wall -o udp-client.o udp-client.c
$ gcc -Wall -o udp-server.o udp-server.c
$ ls udp*
udp-client.c  udp-client.o  udp-server.c  udp-server.o
```

2.3. Instructions d'exécution

L'évaluation des deux programmes *client* et *serveur* est assez simple. On peut les exécuter sur le même hôte dans deux *Shells* distincts en utilisant l'interface de boucle locale pour les communications réseau.

Le programme serveur, udp-server.o

```
$ ./udp-server.o
Entrez le numéro de port utilisé en écoute (entre 1500 et 65000) :
4500
Attente de requête sur le port 4500
>>  depuis 127.0.0.1:39311
  Message reçu : texte avec tabulation et espaces
```

Le programme client, udp-client.o

```
$ ./udp-client.o
Entrez le nom du serveur ou son adresse IP :
127.0.0.1
Entrez le numéro de port du serveur :
4500

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 avec tabulation et espaces
```

```
Message traité : TEXTE AVEC TABULATION ET ESPACES
Saisie du message :
.
```

Lorsque le programme *serveur* est en cours d'exécution, il est possible de visualiser la correspondance entre le processus en cours d'exécution et le numéro de port en écoute à l'aide de la commande **netstat**.

```
$ netstat -aup | grep -e Proto -e udp-server
(Tous les processus ne peuvent être identifiés, les infos sur les processus
non possédés ne seront pas affichées, vous devez être root pour les voir toutes.)
Proto Recv-Q Send-Q Adresse locale    Adresse distante  Etat  PID/Program name
udp        0      0 *:4500            *:*                   3157/udp-server.o
```

Dans l'exemple ci-dessus, le numéro de port `4500` apparaît dans la colonne `Adresse locale` et processus numéro `3157` correspond bien au programme `udp-server.o` dans la colonne `PID/Program name`.

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, soit une adresse IPv6. 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 le format hôte (octet le moins significatif en premier sur processeur Intel x86) ou suivant le format défini dans les en-têtes réseau (octet le plus significatif en premier). Ici le format hôte fait référence à l'architecture du processeur utilisé. Cette architecture est dite «petit-boutiste» pour les processeurs de marque Intel™ majoritairement utilisés dans les ordinateurs de type PC. À l'inverse, le format défini dans les en-têtes réseau est dit «gros-boutiste». Cette définition appelée *Network Byte Order* provient à la fois du protocole IP et de la couche liaison du modèle OSI.

  - `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`, `stdio.h`
: 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.

  Les fonctions d'affichage sans formatage `puts` et `fputs` ainsi que la fonction d'affichage avec formatage `printf` sont utilisées de façon classique.

  En revanche, la saisie des chaînes de caractères à l'aide de la fonction `scanf` est plus singulière. Comme le but des communications réseau évaluées ici est d'échanger des chaînes de caractères, il est nécessaire de transmettre ou recevoir des suites de caractères comprenant aussi bien des espaces que des tabulations.

  La syntaxe usuelle de saisie d'une chaîne de caractère est :

```
scanf("%s", msg);
```

  Si on se contente de cette syntaxe par défaut, la chaîne saisie est transmise par le programme *client* mot par mot. En conséquence, le traitement par le programme *serveur* est aussi effectué mot par mot.

  Pour transmettre une chaîne complète, on utilise une syntaxe du type suivant :

```
scanf(" %[^\n]%*c", msg);
```

  Le caractère espace situé entre les guillemets de gauche et le signe pourcentage a pour but d'éliminer les caractères `' '`, `'\t'` et `'\n'` qui subsisteraient dans la mémoire tampon du flux d'entrée standard `stdin` avant la saisie de nouveaux caractères.

  La syntaxe `[^\n]` précise que tous les caractères différents du saut de ligne sont admis dans la saisie. On évite ainsi que les caractères `' '` et `'\t'` soient considérés comme délimiteurs.

  L'ajout de `%*c` permet d'éliminer le délimiteur `'\n'` et tout caractère situé après dans la mémoire tampon du flux d'entrée standard.

  Enfin, pour éviter tout débordement de la mémoire tampon du même flux d'entrée standard, on limite le nombre maximum des caractères saisis à la quantité de mémoire réservée pour stocker la chaîne de caractères. La «constante» `MAX_MSG` définie via une directive de préprocesseur est introduite dans la syntaxe de formatage de la saisie. Pour cette manipulation, on fait appel à une fonction macro qui renvoie la valeur de `MAX_MSG` comme nombre maximum de caractères à saisir.

On obtient donc finalement le formatage de la saisie d'une chaîne de caractères suivant :

```
scanf(" %"xstr(MAX_MSG)"[^\n]%*c", msg);
```

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 minimes. C'est à l'analyse réseau que la différence se fait sachant que les mécanismes de fonctionnement des deux protocoles sont très différents.

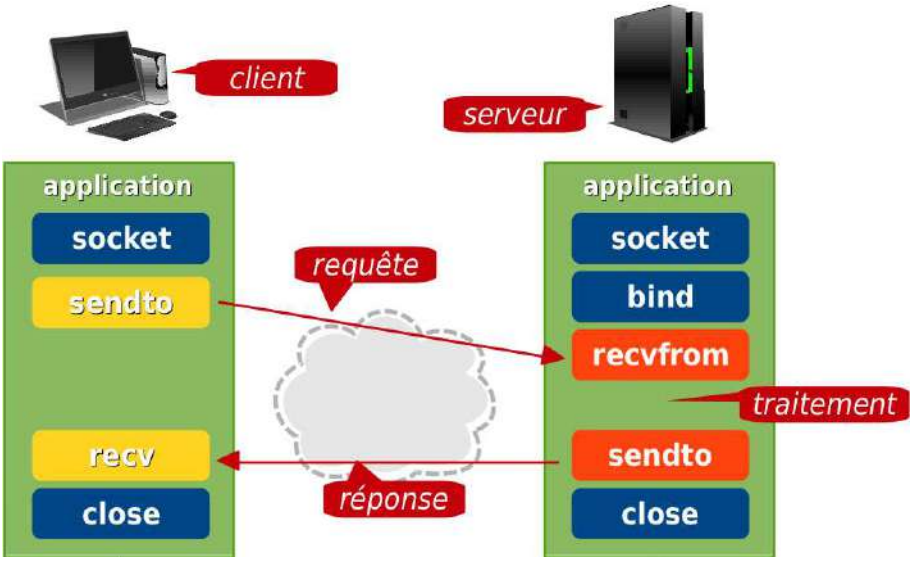
Pour plus d'informations, consulter le support *Modélisations réseau*.

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, « Utilisation des sockets avec le client UDP »
- Serveur : Section 4.1, « Utilisation des sockets avec le serveur UDP »

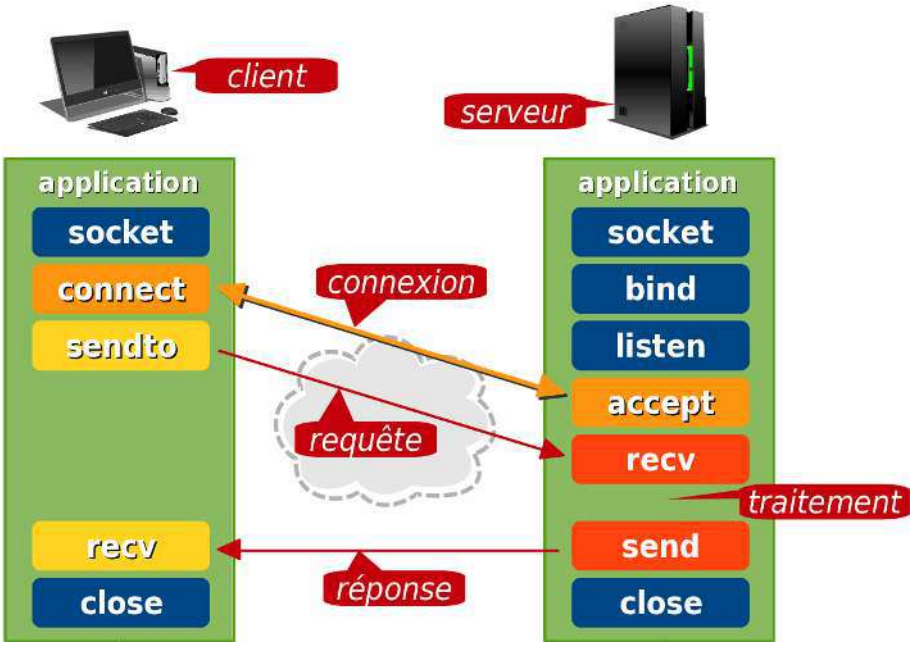


2.7. Sockets & protocole de transport TCP

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 TCP.

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

- Client : Section 5.1, « Utilisation des sockets avec le client TCP »
- Serveur : Section 6.1, « Utilisation des sockets avec le serveur TCP »



3. Programme client UDP

3.1. Utilisation des sockets avec le client UDP

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(PF_INET❶, SOCK_DGRAM❷, IPPROTO_UDP❸);
  if (socketDescriptor < 0) {
    fputs("Impossible de créer le socket", stderr);
    exit(EXIT_FAILURE);
    }
```

❶ `PF_INET` désigne la famille de protocole de couche réseau IPv4.
❷ `SOCK_DGRAM` désigne un service de transmission de datagrammes non orienté connexion.
❸ `IPPROTO_UDP` désigne l'utilisation du protocole UDP au niveau de la couche transport.

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, msgLength❷, 0,
           (struct sockaddr *) &serverAddress,
           sizeof(serverAddress)❸) < 0) {
  fputs("Émission du message impossible", stderr);
  close(socketDescriptor);
  exit(EXIT_FAILURE);
  }
```

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 `msgLength` 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.

```
<snipped/>
if (recv(socketDescriptor❶, msg, MAX_MSG❷, 0) < 0) {
  fputs("Aucune réponse du serveur ?", stderr);
  close(socketDescriptor);
  exit(EXIT_FAILURE);
  }
```

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 `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.c` :

```
#include <netdb.h>
#include <netinet/in.h>
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#define MAX_MSG 100
// 2 caractères pour les codes ASCII '\n' et '\0'
#define MSG_ARRAY_SIZE (MAX_MSG+2)
// Utilisation d'une constante x dans la définition
// du format de saisie
#define str(x) # x
#define xstr(x) str(x)

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];

  puts("Entrez le nom du serveur ou son adresse IP : ");

  memset(msg, 0x0, MSG_ARRAY_SIZE);  // Mise à zéro du tampon
  scanf("%"xstr(MAX_MSG)"s", 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) {
    fprintf(stderr, "Problème dans l'interprétation des informations d'hôte : %s\n", msg);
    exit(EXIT_FAILURE);
  }

  puts("Entrez le numéro de port du serveur : ");
  scanf("%hu", &serverPort);

  // Création de socket. "PF_INET" correspond à la famille de protocole IPv4.
  // "SOCK_DGRAM" correspond à un service de datagramme non orienté connexion.
  // "IPPROTO_UDP" désigne le protocole UDP utilisé au niveau transport.
  socketDescriptor = socket(PF_INET, SOCK_DGRAM, IPPROTO_UDP);
  if (socketDescriptor < 0) {
    fputs("Impossible de créer le socket", stderr);
    exit(EXIT_FAILURE);
  }

  // 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);

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

  // Invite de commande pour l'utilisateur et lecture des caractères jusqu'à la
  // limite MAX_MSG. Puis suppression du saut de ligne en mémoire tampon.
  puts("Saisie du message : ");
  memset(msg, 0x0, MSG_ARRAY_SIZE);  // Mise à zéro du tampon
  scanf(" %"xstr(MAX_MSG)"[^\n]%*c", 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) {
        fputs("Émission du message impossible", stderr);
        close(socketDescriptor);
        exit(EXIT_FAILURE);
      }

      // 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) {
          fputs("Aucune réponse du serveur ?", stderr);
          close(socketDescriptor);
          exit(EXIT_FAILURE);
        }

        printf("Message traité : %s\n", msg);
      }
      else {
        puts("** Le serveur n'a répondu dans la seconde.");
      }
    }
    // Invite de commande pour l'utilisateur et lecture des caractères jusqu'à la
    // limite MAX_MSG. Puis suppression du saut de ligne en mémoire tampon.
    // Comme ci-dessus.
    puts("Saisie du message : ");
    memset(msg, 0x0, MSG_ARRAY_SIZE);  // Mise à zéro du tampon
    scanf(" %"xstr(MAX_MSG)"[^\n]%*c", msg);
  }

  close(socketDescriptor);
  return 0;
}
```

4. Programme serveur UDP

4.1. Utilisation des sockets avec le serveur UDP

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(PF_INET, SOCK_DGRAM, IPPROTO_UDP);
if (listenSocket < 0) {
  fputs("Impossible de créer le socket en écoute", stderr);
  exit(EXIT_FAILURE);
  }
```

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 = PF_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) {
  fputs("Impossible de lier le socket en écoute", stderr);
  exit(EXIT_FAILURE);
  }
```

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/>
if (recvfrom(listenSocket, msg, MAX_MSG❶, 0,
             (struct sockaddr *) &clientAddress,
             &clientAddressLength) < 0) {
  fputs("Problème de réception du messsage", stderr);
  exit(EXIT_FAILURE);
  }
```

❶ Les paramètres `msg` 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.c` :

```
#include <arpa/inet.h>
#include <netdb.h>
#include <netinet/in.h>
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <ctype.h>

#define MAX_MSG 100
// 2 caractères pour les codes ASCII '\n' et '\0'
#define MSG_ARRAY_SIZE (MAX_MSG+2)

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

  puts("Entrez le numéro de port utilisé en écoute (entre 1500 et 65000) : ");
  scanf("%hu", &listenPort);

  // Création de socket en écoute et attente des requêtes des clients
  listenSocket = socket(PF_INET, SOCK_DGRAM, IPPROTO_UDP);
  if (listenSocket < 0) {
    fputs("Impossible de créer le socket en écoute", stderr);
    exit(EXIT_FAILURE);
  }

  // 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 = PF_INET;
  serverAddress.sin_addr.s_addr = htonl(INADDR_ANY);
  serverAddress.sin_port = htons(listenPort);

  if (bind(listenSocket,
           (struct sockaddr *) &serverAddress,
           sizeof(serverAddress)) < 0) {
    fputs("Impossible de lier le socket en écoute", stderr);
    exit(EXIT_FAILURE);
  }

  printf("Attente de requête sur le port %hu\n", listenPort);

  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) {
      fputs("Problème de réception du messsage", stderr);
      exit(EXIT_FAILURE);
    }

    msgLength = strlen(msg);
    if (msgLength > 0) {
      // Affichage de l'adresse IP du client.
      printf(">>  depuis %s", inet_ntoa(clientAddress.sin_addr));

      // Affichage du numéro de port du client.
      printf(":%hu\n", ntohs(clientAddress.sin_port));
```

```
      // Affichage de la ligne reçue
      printf("  Message reçu : %s\n", msg);

      // 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)
        fputs("Émission du message modifié impossible", stderr);
    }
  }
}
```

5. Programme client TCP

5.1. Utilisation des sockets avec le client TCP

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 ce fait à l'aide de la fonction `connect`.

```
  int socketDescriptor;
  struct sockaddr_in serverAddress;

<snipped/>
if (connect(socketDescriptor,
            (struct sockaddr *) &serverAddress,
            sizeof(serverAddress)) < 0) {
  fputs("Connexion impossible", stderr);
  close(socketDescriptor);
  exit(EXIT_FAILURE);
  }
```

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.c` :

```
--- udp-client.c 2012-02-27 11:32:56.000000000 +0100
+++ tcp-client.c 2012-02-27 11:33:14.000000000 +0100
@@ -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];

   puts("Entrez le nom du serveur ou son adresse IP : ");
@@ -46,7 +44,7 @@
   // Création de socket. "PF_INET" correspond à la famille de protocole IPv4.
   // "SOCK_DGRAM" correspond à un service de datagramme non orienté connexion.
   // "IPPROTO_UDP" désigne le protocole UDP utilisé au niveau transport.
-  socketDescriptor = socket(PF_INET, SOCK_DGRAM, IPPROTO_UDP);
+  socketDescriptor = socket(PF_INET, SOCK_STREAM, IPPROTO_TCP);
   if (socketDescriptor < 0) {
     fputs("Impossible de créer le socket", stderr);
     exit(EXIT_FAILURE);
@@ -58,6 +56,14 @@
          hostInfo->h_addr_list[0], hostInfo->h_length);
   serverAddress.sin_port = htons(serverPort);

+  if (connect(socketDescriptor,
+              (struct sockaddr *) &serverAddress,
+              sizeof(serverAddress)) < 0) {
+    fputs("Connexion impossible", stderr);
+    close(socketDescriptor);
+    exit(EXIT_FAILURE);
+  }
+
   puts("\nEntrez quelques caractères au clavier.");
   puts("Le serveur les modifiera et les renverra.");
   puts("Pour sortir, entrez une ligne avec le caractère '.' uniquement.");
@@ -82,13 +88,6 @@
         exit(EXIT_FAILURE);
       }

-      // 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) {
@@ -96,13 +95,10 @@
           close(socketDescriptor);
           exit(EXIT_FAILURE);
         }
+    }

         printf("Message traité : %s\n", msg);
-      }
-      else {
-        puts("** Le serveur n'a répondu dans la seconde.");
-      }
-    }
+
     // Invite de commande pour l'utilisateur et lecture des caractères jusqu'à la
     // limite MAX_MSG. Puis suppression du saut de ligne en mémoire tampon.
     // Comme ci-dessus.
```

6. Programme serveur TCP

6.1. Utilisation des sockets avec le serveur TCP

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.

La fonction `listen` active l'utilisation du canal de communication initié avec la fonction `socket`. Le second paramètre `5` définit le nombre maximal de demandes de connexions en attente.

```
  int listenSocket, connectSocket, i;

<snipped/>
listen(listenSocket, 5);

while (1) {
  printf("Attente de connexion TCP sur le port %hu\n", listenPort);
<snipped/>
```

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) {
  fputs("Impossible d'accepter une connexion", stderr);
  close(listenSocket);
  exit(EXIT_FAILURE);
  }
```

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.c` :

```
--- udp-server.c 2012-04-21 23:14:49.000000000 +0200
+++ tcp-server.c 2012-02-22 16:30:19.000000000 +0100
@@ -13,7 +13,7 @@

 int main()
 {
-  int listenSocket, i;
+  int listenSocket, connectSocket, i;
   unsigned short int listenPort, msgLength;
   socklen_t clientAddressLength;
   struct sockaddr_in clientAddress, serverAddress;
@@ -23,7 +23,7 @@
   scanf("%hu", &listenPort);

   // Création de socket en écoute et attente des requêtes des clients
-  listenSocket = socket(PF_INET, SOCK_DGRAM, IPPROTO_UDP);
+  listenSocket = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP);
   if (listenSocket < 0) {
     fputs("Impossible de créer le socket en écoute", stderr);
     exit(EXIT_FAILURE);
@@ -46,42 +46,67 @@
     exit(EXIT_FAILURE);
```

```
   }

-  printf("Attente de requête sur le port %hu\n", listenPort);
+  // 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);

   while (1) {
+    printf("Attente de connexion TCP sur le port %hu\n", listenPort);

+    // 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) {
-      fputs("Problème de réception du messsage", stderr);
+                   &clientAddressLength);
+
+    if (connectSocket < 0) {
+      fputs("Impossible d'accepter une connexion", stderr);
+      close(listenSocket);
       exit(EXIT_FAILURE);
     }

-    msgLength = strlen(msg);
-    if (msgLength > 0) {
       // Affichage de l'adresse IP du client.
-      printf(">>  depuis %s", inet_ntoa(clientAddress.sin_addr));
-
-      // Affichage du numéro de port du client.
+    // inet_ntoa() convertit une adresse IP stockée sous forme binaire en une
+    // chaîne de caractères
+    printf(">>  connecté à %s", 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).
       printf(":%hu\n", ntohs(clientAddress.sin_port));

-      // Affichage de la ligne reçue
-      printf("  Message reçu : %s\n", msg);
+    // 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) {
+        printf("  --  %s\n", msg);

       // 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)
         fputs("Émission du message modifié impossible", stderr);
+
+        memset(msg, 0x0, MSG_ARRAY_SIZE);  // Mise à zéro du tampon
+      }
     }
   }
 }
```

7. Analyse réseau avec Wireshark

L'analyse réseau présente un grand intérêt dans la validation des développements. Elle permet de contrôler le bon fonctionnement des communications suivant les jeux de protocoles utilisés. Ici, on peut différencier

le fonctionnement des deux protocoles de la couche transport en analysant les échanges entre les deux programmes *client* et *serveur*.

L'utilisation de l'analyseur wireshark est présentée dans le support *Introduction à l'analyse réseau avec Wireshark*[4].

Les analyses présentées ci-après ont été réalisées dans les conditions suivantes :

- Le programme *serveur* est exécuté sur l'hôte ayant l'adresse IP `192.0.2.1`. Ce programme est toujours en écoute sur le port `4000`.
- Le programme *client* est exécuté sur l'hôte ayant l'adresse IP `192.0.2.30`.

7.1. Analyse avec le protocole UDP

Avant de passer à l'analyse, voici les copies d'écran de l'exécution des programmes.

- Côté serveur :

```
$ ./udp-server.o
Entrez le numéro de port utilisé en écoute (entre 1500 et 65000) :
4000
Attente de requête sur le port 4000
>>  depuis 192.0.2.30:43648
  Message reçu : message de test UDP
^C
```

- Côté client :

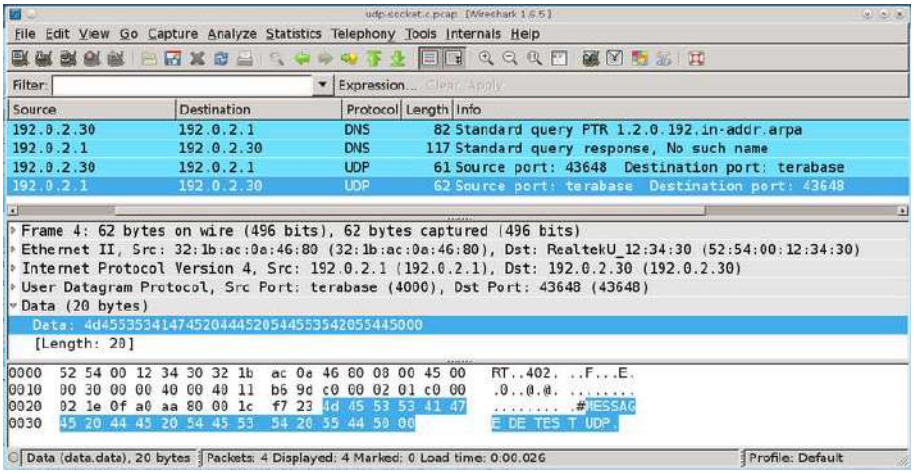
```
$ ./udp-client.o
Entrez le nom du serveur ou son adresse IP :
192.0.2.1
Entrez le numéro de port du serveur :
4000

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 :
message de test UDP
Message traité : MESSAGE DE TEST UDP
Saisie du message :
.
```

Dans la copie d'écran ci-dessous, on retrouve l'ensemble des éléments énoncés auparavant.

- Chaîne de caractères traitée dans la partie données de la couche application.
- Numéros de ports utilisés dans les en-têtes de la couche transport.
- Adresses IP utilisées dans les en-têtes de la couche réseau.



Enfin, le fait que la capture se limite à deux échanges illustre la principale caractéristique du protocole UDP : un service de datagramme non orienté connexion qui suppose un réseau sous-jacent fiable et sans erreur.

7.2. Analyse avec le protocole TCP

Comme dans le cas précédent, voici les copies d'écran de l'exécution des programmes avant de passer à l'analyse réseau.

[4] http://www.inetdoc.net/travaux_pratiques/intro.analyse/

• Côté serveur :

```
$ ./tcp-server.o
Entrez le numéro de port utilisé en écoute (entre 1500 et 65000) :
4000
Attente de connexion TCP sur le port 4000
>>  connecté à 192.0.2.30:52060
  --  message de test TCP
  --  dernier message avant fermeture de la connexion
Attente de connexion TCP sur le port 4000
^C
```

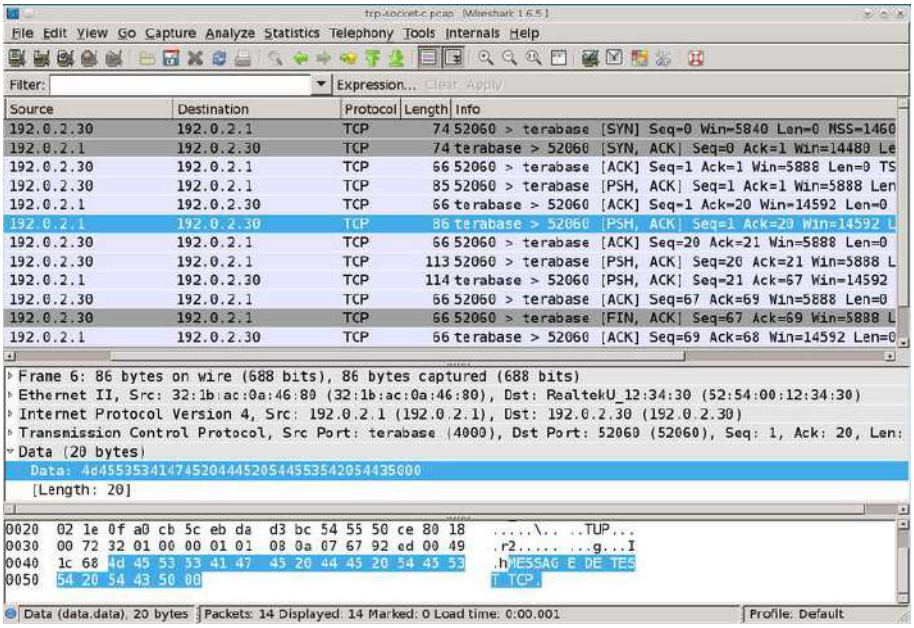
• Côté client :

```
$ ./tcp-client.o
Entrez le nom du serveur ou son adresse IP :
192.0.2.1
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 :
message de test TCP
Message traité : MESSAGE DE TEST TCP
Saisie du message :
dernier message avant fermeture de la connexion
Message traité : DERNIER MESSAGE AVANT FERMETURE DE LA CONNEXION
Saisie du message :
.
```

Dans la copie d'écran ci-dessous, on retrouve l'ensemble des éléments déjà connus : chaîne de caractères traitée, numéros de port en couche transport et adresses IP en couche réseau.



Cette copie d'écran se distingue de la précédente, par le nombre de trames capturées alors que le traitement effectué est quasiment le même. La capture réseau fait apparaître les phases d'établissement, de maintien et de libération d'une connexion. On illustre ainsi toutes les fonctions de fiabilisation apportées par le protocole TCP.

À partir de ces quelques trames, on peut reprendre l'analyse de la poignée de main à trois voies, de l'évolution des numéros de séquence et de l'évolution de la fenêtre d'acquittement.

8. Documents de référence

A Brief Socket Tutorial
: *brief socket tutorial*[5] : 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 transportUDP.

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

Beej's Guide to Network Programming
: *Beej's Guide to Network Programming*[6] : support très complet sur les sockets proposant de nombreux exemples de programmes.

Modélisations réseau
: *Modélisations réseau*[7] : présentation et comparaison des modélisations OSI et Internet.

Adressage IPv4
: *Adressage IPv4*[8] : 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*[9] : support sur la configuration des interfaces réseau. Il permet notamment de relever les adresses IP des hôtes en communication.

[6] http://beej.us/guide/bgnet/
[7] http://www.inetdoc.net/articles/modelisation/
[8] http://www.inetdoc.net/articles/adressage.ipv4/
[9] http://www.inetdoc.net/travaux_pratiques/config.interface.lan/