Initiation au développement C sur les sockets

Philippe Latu
philippe.latu(at)inetdoc.net

https://www.inetdoc.net

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

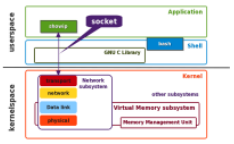


Table des matières

1. Copyright et Licence 1
    1.1. Meta-information 2
2. Contexte de développement 3
    2.1. Système d'exploitation 3
    2.2. Instructions de compilation 3
    2.3. Instructions d'exécution 3
    2.4. Bibliothèques utilisées 4
    2.5. Choix du premier protocole de transport étudié 5
    2.6. Sockets & protocole de transport UDP 6
    2.7. Sockets & protocole de transport TCP 6
3. Programme client UDP : talker 7
    3.1. Utilisation des sockets avec le client UDP 7
    3.2. Code source complet 8
4. Programme serveur UDP : listener 10
    4.1. Utilisation des sockets avec le serveur UDP 10
    4.2. Code source complet 10
5. Programme client TCP : talker 12
    5.1. Utilisation des sockets avec le client TCP 12
    5.2. Patch code source 12
6. Programme serveur TCP : listener 13
    6.1. Utilisation des sockets avec le serveur TCP 13
    6.2. Patch code source 13
7. Analyse réseau avec Wireshark 15
    7.1. Analyse avec le protocole UDP 15
    7.2. Analyse avec le protocole TCP 16
8. Documents de référence 17

1. Copyright et Licence

1.1. Meta-information

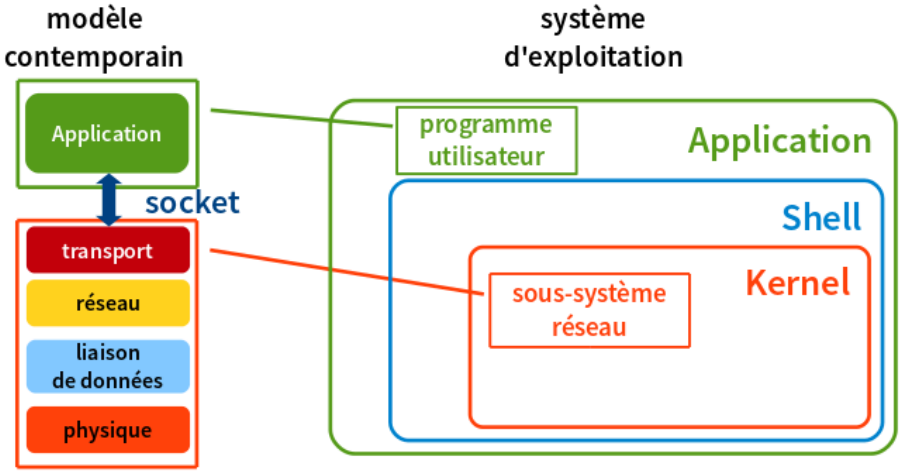
Cet article est écrit avec DocBook XML sur un système Debian GNU/Linux. Il est disponible en version imprimable au format PDF : `socket-c.pdf`.

2. Contexte de développement

L'objectif de développement étant l'initiation, on se limite à un code minimaliste utilisant deux programmes distincts : un serveur ou listener et un client ou talker. 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 reçue 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 -Wextra -o udp-talker.o udp-talker.c
$ gcc -Wall -Wextra -o udp-listener.o udp-listener.c
$ ls udp*
udp-listener.c  udp-listener.o udp-talker.c  udp-talker.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 serveurudp-listener.o

```
$ ./udp-listener.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 clientudp-talker.o

```
$ ./udp-talker.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-listener
(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-listener.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-listener.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-devnetdb.h`

Opérations sur les bases de données réseau. Ici, ce sont les fonctions `getaddrinfo()` et `gai_strerror()` qui sont utilisées. La première fonction manipule des enregistrements de type `addrinfo` qui contiennent tous les attributs de description d'un hôte réseau et de la prise réseau à ouvrir. Parmi ces attributs, on trouve la famille d'adresse réseau IPv4 ou IPv6, le type de service qui désigne le protocole de couche transport TCP ou UDP et l'adresse de l'hôte à contacter. Pour obtenir plus d'informations, il faut consulter les pages de manuels des fonctions : `man getaddrinfo` par exemple.

`libc6-devnetinet/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) 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-devstdio.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*** ou talker et ***serveur*** ou listener 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.

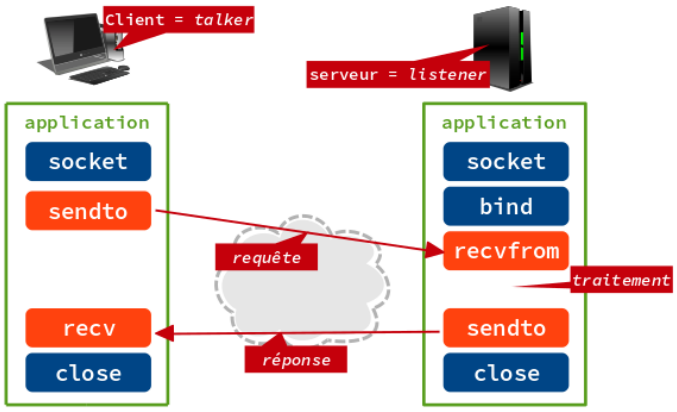
Pour plus d'informations, consulter le support Modélisations réseau.

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 »

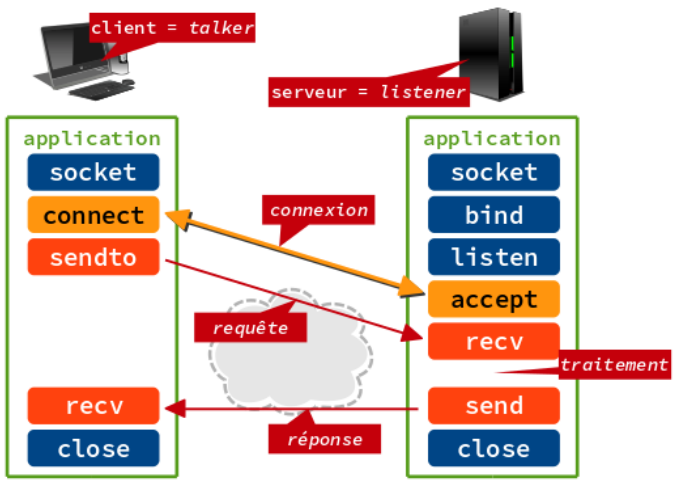


2.7. Sockets & protocole de transport TCP

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 : *talker*

3.1. Utilisation des sockets avec le client UDP

Au niveau du ***client*** ou talker, l'objectif est ***d'ouvrir*** une nouvelle prise réseau ou socket ; ce qui revient à ouvrir un canal de communication réseau avec la fonction `socket()` après avoir défini les paramètres de l'hôte à contacter avec la fonction `getaddrinfo()`.

Les bibliothèques standard associées au Langage C fournissent à la fois des sous-programmes et des définitions d'enregistrements (ou structures) de données. On rejoint ici le mode opératoire des langages orientés objet dont les bibliothèques fournissent des classes comprenant respectivement les définitions des méthodes et les attributs des données à manipuler.

Ainsi, la fonction `getaddrinfo()` manipule des enregistrements de type `addrinfo`. Pour utiliser cette fonction, on commence par orienter le choix du type de prise réseau à ouvrir avant de l'appeler en affectant certains champs de l'enregistrement `hints` puis on exploite les résultats contenus dans l'enregistrement `servinfo` après l'avoir appelée.

```
hints.ai_family = AF_INET❶;
hints.ai_socktype = SOCK_DGRAM❷;

if ((status = getaddrinfo(msg❸, serverPort, &hints, &servinfo)) != 0) {
  fprintf(stderr, "getaddrinfo: %s\n", gai_strerror(status));
  exit(EXIT_FAILURE);
}
```

❶ `AF_INET` désigne la famille de protocole de couche réseau IPv4.
❷ `SOCK_DGRAM` désigne un service de transmission de datagrammes non orienté connexion. Autrement dit, le protocole de couche transport UDP.
❸ La chaîne de caractères `msg` contient le nom de l'hôte à contacter. Cet hôte peut être désigné directement par une adresse IP ou par son nom. Dans ce dernier cas, le resolver du service de noms de domaine (DNS) est sollicité automatiquement.

L'appel à la fonction `socket()` se fait sur la base des champs renseignés par la fonction `getaddrinfo()`. Le premier enregistrement de la chaîne pointée par `servinfo` contient tous les paramètres nécessaires.

```
if ((socketDescriptor = socket(servinfo->ai_family, servinfo->ai_socktype,
                               servinfo->ai_protocol)) == -1) {
    perror("socket:");
    exit(EXIT_FAILURE);
  }
```

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

```
while (strcmp(msg, ".")) {
  if ((msgLength = strlen(msg)) > 0) {
    // Envoi de la ligne au serveur
    if (sendto(socketDescriptor❶, msg, msgLength❷, 0,
               servinfo->ai_addr❸, servinfo->ai_addrlen) == -1) {
      perror("sendto:");
      close(socketDescriptor);
      exit(EXIT_FAILURE);
    }
```

❶ `socketDescriptor` désigne la prise réseau ou encore le canal de communication entre le programme de l'espace utilisateur et le sous-système réseau de l'espace noyau.
❷ `msg` et `msgLength` correspondent au datagramme et à sa longueur. Ici, on émet des chaînes de caractères directement vers le correspondant réseau.
❸ On utilise à nouveau les champs de l'enregistrement pointé par `servinfo` pour désigner l'hôte vers lequel les datagrammes sont émis.

Comme le service d'échange de datagrammes entre deux hôtes, n'est pas orienté connexion, le protocole UDP de la couche transport n'offre aucune garantie sur la délivrance de ces datagrammes. En conséquence, il incombe au programme utilisateur de fournir une forme de contrôle d'erreur. La solution généralement adoptée consiste à utiliser une temporisation d'attente de réponse. Dans notre

cas, si aucune réponse du serveur ou listener n'a été reçue au bout d'une seconde, on peut considérer que le datagramme émis a été perdu.

C'est la fonction `select()` qui permet de superviser des descripteurs de flux tels que les prises réseau (socket).

```
// Attente de la réponse pendant une seconde.
FD_ZERO(&readSet❶);
FD_SET(socketDescriptor, &readSet);
timeVal.tv_sec = 1❷;
timeVal.tv_usec = 0;

if (select(socketDescriptor+1, &readSet, NULL, NULL, &timeVal)❸) {
  // Lecture de la ligne modifiée par le serveur.
  memset(msg, 0, sizeof msg);  // Mise à zéro du tampon
  if (recv(socketDescriptor, msg, sizeof msg, 0) == -1) {
    perror("recv:");
    close(socketDescriptor);
    exit(EXIT_FAILURE);
    }
```

❶ `readSet` est réinitialisé à chaque itération. C'est cette variable qui sert à associer la prise réseau `socketDescriptor` à la fonction de supervision `select()`.

❷ `timeVal` est un enregistrement dont le champ `tv_sec` définit le temps d'attente de la réponse de l'hôte réseau vers lequel un datagramme a été précédemment émis.

❸ `select()` renvoie une valeur supérieure à 0 en cas de succès : un datagramme de réponse est en attente moins d'une seconde après l'émission précédente.

Enfin, il ne reste plus que la réception du ou des datagrammes renvoyés par le serveur à l'aide de la fonction `recv()`.

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-talker.c` :

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

#define MAX_PORT 5
#define PORT_ARRAY_SIZE (MAX_PORT+1)
#define MAX_MSG 80
#define MSG_ARRAY_SIZE (MAX_MSG+1)
// 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, status;
  unsigned int msgLength;
  struct addrinfo hints, *servinfo;
  struct timeval timeVal;
  fd_set readSet;
  char msg[MSG_ARRAY_SIZE], serverPort[PORT_ARRAY_SIZE];

  puts("Entrez le nom du serveur ou son adresse IP : ");
  memset(msg, 0, sizeof msg);  // Mise à zéro du tampon
  scanf("%"xstr(MAX_MSG)"s", msg);

  puts("Entrez le numéro de port du serveur : ");
  memset(serverPort, 0, sizeof serverPort);  // Mise à zéro du tampon
  scanf("%"xstr(MAX_PORT)"s", serverPort);

  memset(&hints, 0, sizeof hints);
```

```
  hints.ai_family = AF_INET;
  hints.ai_socktype = SOCK_DGRAM;

  if ((status = getaddrinfo(msg, serverPort, &hints, &servinfo)) != 0) {
    fprintf(stderr, "getaddrinfo: %s\n", gai_strerror(status));
    exit(EXIT_FAILURE);
  }

  if ((socketDescriptor = socket(servinfo->ai_family, servinfo->ai_socktype,
                                 servinfo->ai_protocol)) == -1) {
    perror("socket:");
    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.");
  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, 0, sizeof msg);  // 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,
                 servinfo->ai_addr, servinfo->ai_addrlen) == -1) {
        perror("sendto:");
        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, 0, sizeof msg);  // Mise à zéro du tampon
        if (recv(socketDescriptor, msg, sizeof msg, 0) == -1) {
          perror("recv:");
          close(socketDescriptor);
          exit(EXIT_FAILURE);
        }

        printf("Message traité : %s\n", msg);
      }
      else {
        puts("Pas de réponse 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, 0, sizeof msg);  // Mise à zéro du tampon
    scanf(" %"xstr(MAX_MSG)"[^\n]%*c", msg);
  }

  close(socketDescriptor);

  return EXIT_SUCCESS;
}
```

4. Programme serveur UDP : ***listener***

4.1. Utilisation des sockets avec le serveur UDP

Au niveau du ***serveur*** ou listener, l'objectif est aussi ***d'ouvrir*** une nouvelle prise réseau ou socket ; ce qui revient à nouveau à ouvrir un canal de communication réseau avec la fonction `socket()`. On retrouve la même séquence que dans le programme client précédent avec l'utilisation de la fonction `getaddrinfo()`. À la différence du programme précédent, on oriente l'ouverture de la prise réseau sur les interfaces réseau locales au système.

```
hints.ai_family = AF_INET;       // IPv4
hints.ai_socktype = SOCK_DGRAM;  // UDP
hints.ai_flags = AI_PASSIVE;     // Toutes les adresses disponibles

if ((status = getaddrinfo(NULL, listenPort, &hints, &servinfo)) != 0) {
  fprintf(stderr, "getaddrinfo: %s\n", gai_strerror(status));
  return 1;
  }
```

L'appel à la fonction `socket()` ne présente aucune différence avec le programme client puisque tout le paramétrage à déjà été effectué. Les informations utiles sont contenues dans les champs de l'enregistrement pointé par `servinfo`.

```
if ((listenSocket = socket(servinfo->ai_family, servinfo->ai_socktype,
                           servinfo->ai_protocol)) == -1) {
  perror("socket:");
  exit(EXIT_FAILURE);
  }
```

Une fois la prise réseau en place et en état d'écoute, le programme attend les datagrammes provenant des clients.

```
memset(msg, 0, sizeof msg);
if (recvfrom(listenSocket, msg, sizeof msg, 0,
             (struct sockaddr *) &clientAddress,
             &clientAddressLength) == -1) {
  perror("recvfrom:");
  close(listenSocket);
  exit(EXIT_FAILURE);
  }
```

Les paramètres `msg` et `sizeof msg` définissent la chaîne de caractères dans laquelle le datagramme reçu est stocké ainsi que le nombre maximum de caractères qui peuvent être stockés.

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-listener.c` :

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

#define MAX_PORT 5
#define PORT_ARRAY_SIZE (MAX_PORT+1)
#define MAX_MSG 80
#define MSG_ARRAY_SIZE (MAX_MSG+1)
// Utilisation d'une constante x dans la définition
// du format de saisie
#define str(x) # x
#define xstr(x) str(x)

int main()
{
  int listenSocket, status, i;
```

```
unsigned short int msgLength;
struct addrinfo hints, *servinfo;
struct sockaddr_in clientAddress;
socklen_t clientAddressLength = sizeof clientAddress;
char msg[MSG_ARRAY_SIZE], listenPort[PORT_ARRAY_SIZE];

memset(listenPort, 0, sizeof listenPort);  // Mise à zéro du tampon
puts("Entrez le numéro de port utilisé en écoute (entre 1500 et 65000) : ");
scanf("%"xstr(MAX_PORT)"s", listenPort);

memset(&hints, 0, sizeof hints);
hints.ai_family = AF_INET;       // IPv4
hints.ai_socktype = SOCK_DGRAM;  // UDP
hints.ai_flags = AI_PASSIVE;     // Toutes les adresses disponibles

if ((status = getaddrinfo(NULL, listenPort, &hints, &servinfo)) != 0) {
  fprintf(stderr, "getaddrinfo: %s\n", gai_strerror(status));
  exit(EXIT_FAILURE);
}

if ((listenSocket = socket(servinfo->ai_family, servinfo->ai_socktype,
                           servinfo->ai_protocol)) == -1) {
  perror("socket:");
  exit(EXIT_FAILURE);
}

if (bind(listenSocket, servinfo->ai_addr, servinfo->ai_addrlen) == -1) {
  close(listenSocket);
  perror("bind:");
  exit(EXIT_FAILURE);
}

// Libération de la mémoire occupée par les enregistrements
freeaddrinfo(servinfo);

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

while (1) {

  // Mise à zéro du tampon de façon à connaître le délimiteur
  // de fin de chaîne.
  memset(msg, 0, sizeof msg);
  if (recvfrom(listenSocket, msg, sizeof msg, 0,
               (struct sockaddr *) &clientAddress,
               &clientAddressLength) == -1) {
    perror("recvfrom:");
    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.
    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, 0,
               (struct sockaddr *) &clientAddress,
               clientAddressLength) == -1) {
      perror("sendto:");
      close(listenSocket);
      exit(EXIT_FAILURE);
    }
```

```
    }
  }
}
```

5. Programme client TCP : ***talker***

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*** étudié précédemment. 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`.

```
if (connect(socketDescriptor, servinfo->ai_addr, servinfo->ai_addrlen) == -1) {
  close(socketDescriptor);
  perror("connect");
  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-talker.c` :

```
--- udp-talker.c 2016-10-25 16:22:31.473862464 +0200
+++ tcp-talker.c 2016-10-25 16:22:31.473862464 +0200
@@ -19,8 +19,6 @@
   int socketDescriptor, status;
   unsigned int msgLength;
   struct addrinfo hints, *servinfo;
-  struct timeval timeVal;
-  fd_set readSet;
   char msg[MSG_ARRAY_SIZE], serverPort[PORT_ARRAY_SIZE];

   puts("Entrez le nom du serveur ou son adresse IP : ");
@@ -33,7 +31,7 @@

   memset(&hints, 0, sizeof hints);
   hints.ai_family = AF_INET;
-  hints.ai_socktype = SOCK_DGRAM;
+  hints.ai_socktype = SOCK_STREAM;

   if ((status = getaddrinfo(msg, serverPort, &hints, &servinfo)) != 0) {
     fprintf(stderr, "getaddrinfo: %s\n", gai_strerror(status));
@@ -46,6 +44,15 @@
     exit(EXIT_FAILURE);
   }

+  if (connect(socketDescriptor, servinfo->ai_addr, servinfo->ai_addrlen) == -1) {
+    close(socketDescriptor);
+    perror("connect");
+    exit(EXIT_FAILURE);
+  }
+
+  // Libération de la mémoire occupée par les enregistrements
+  freeaddrinfo(servinfo);
+
   puts("\nEntrez quelques caractères au clavier.");
   puts("Le serveur les modifiera et les renverra.");
   puts("Pour sortir, entrez une ligne avec le caractère '.' uniquement.");
@@ -69,13 +76,6 @@
         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, 0, sizeof msg);  // Mise à zéro du tampon
         if (recv(socketDescriptor, msg, sizeof msg, 0) == -1) {
@@ -83,13 +83,10 @@
           close(socketDescriptor);
           exit(EXIT_FAILURE);
         }
+    }

         printf("Message traité : %s\n", msg);
-      }
-      else {
-        puts("Pas de réponse 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 : ***listener***

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 `BACKLOG` définit le nombre maximal de demandes de connexions en attente.

```
if (listen(listenSocket, BACKLOG) == -1) {
  perror("listen");
  exit(EXIT_FAILURE);
  }
```

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.

```
if ((connectSocket = accept(listenSocket,
                            (struct sockaddr *) &clientAddress,
                            &clientAddressLength)) == -1) {
  perror("accept:");
  close(listenSocket);
  exit(EXIT_FAILURE);
  }
```

En cas d'échec de l'ouverture de la prise réseau ou socket lors de la réception d'une demande de connexion, on abandonne le traitement.

6.2. Patch code source

Patch du programme `tcp-listener.c` :

```
--- udp-listener.c 2016-10-25 16:22:31.473862464 +0200
+++ tcp-listener.c 2016-10-25 16:22:31.473862464 +0200
@@ -11,6 +11,7 @@
 #define PORT_ARRAY_SIZE (MAX_PORT+1)
 #define MAX_MSG 80
 #define MSG_ARRAY_SIZE (MAX_MSG+1)
+#define BACKLOG 5
 // Utilisation d'une constante x dans la définition
 // du format de saisie
 #define str(x) # x
@@ -18,7 +19,7 @@

 int main()
 {
-  int listenSocket, status, i;
```

```
+  int listenSocket, connectSocket, status, i;
   unsigned short int msgLength;
   struct addrinfo hints, *servinfo;
   struct sockaddr_in clientAddress;
@@ -31,7 +32,7 @@

   memset(&hints, 0, sizeof hints);
   hints.ai_family = AF_INET;       // IPv4
-  hints.ai_socktype = SOCK_DGRAM;  // UDP
+  hints.ai_socktype = SOCK_STREAM; // TCP
   hints.ai_flags = AI_PASSIVE;     // Toutes les adresses disponibles

   if ((status = getaddrinfo(NULL, listenPort, &hints, &servinfo)) != 0) {
@@ -54,44 +55,62 @@
   // Libération de la mémoire occupée par les enregistrements
   freeaddrinfo(servinfo);

-  printf("Attente de requête sur le port %s\n", listenPort);
+  // Attente des requêtes des clients.
+  // Appel non bloquant et passage en mode passif
+  // Demandes d'ouverture de connexion traitées par accept
+  if (listen(listenSocket, BACKLOG) == -1) {
+    perror("listen");
+    exit(EXIT_FAILURE);
+  }

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

-    // Mise à zéro du tampon de façon à connaître le délimiteur
-    // de fin de chaîne.
-    memset(msg, 0, sizeof msg);
-    if (recvfrom(listenSocket, msg, sizeof msg, 0,
+    // Appel accept() bloquant
+    // connectSocket est une nouvelle prise indépendante
+    if ((connectSocket = accept(listenSocket,
                  (struct sockaddr *) &clientAddress,
-                 &clientAddressLength) == -1) {
-      perror("recvfrom:");
+    &clientAddressLength)) == -1) {
+      perror("accept:");
+      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, 0, sizeof msg);
+    while (recv(connectSocket, msg, sizeof 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, 0,
-                 (struct sockaddr *) &clientAddress,
-                 clientAddressLength) == -1) {
-        perror("sendto:");
+        if (send(connectSocket, msg, msgLength + 1, 0) == -1) {
+          perror("send:");
         close(listenSocket);
         exit(EXIT_FAILURE);
       }
+
+        memset(msg, 0, sizeof msg);  // 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.

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 :

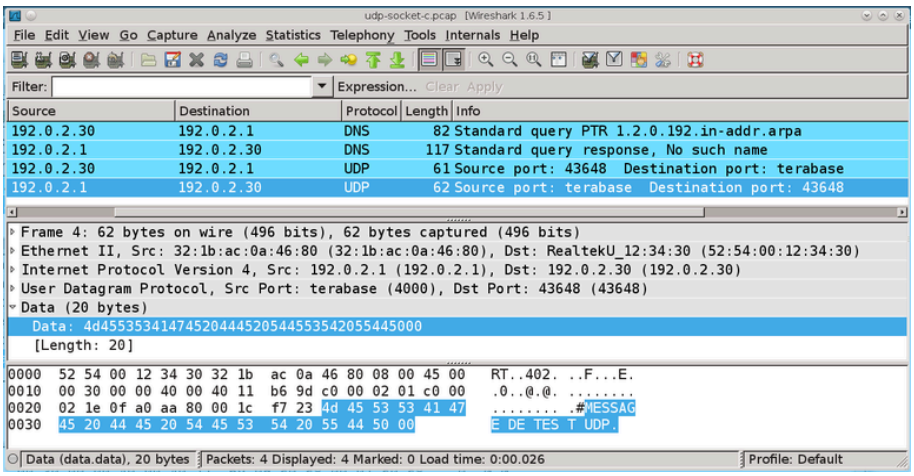
```
$ ./udp-client.o
Entrez le nom du serveur ou son adresse IP :
192.0.2.1
Entrez le numéro de port du serveur :
4000

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.

- 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
```

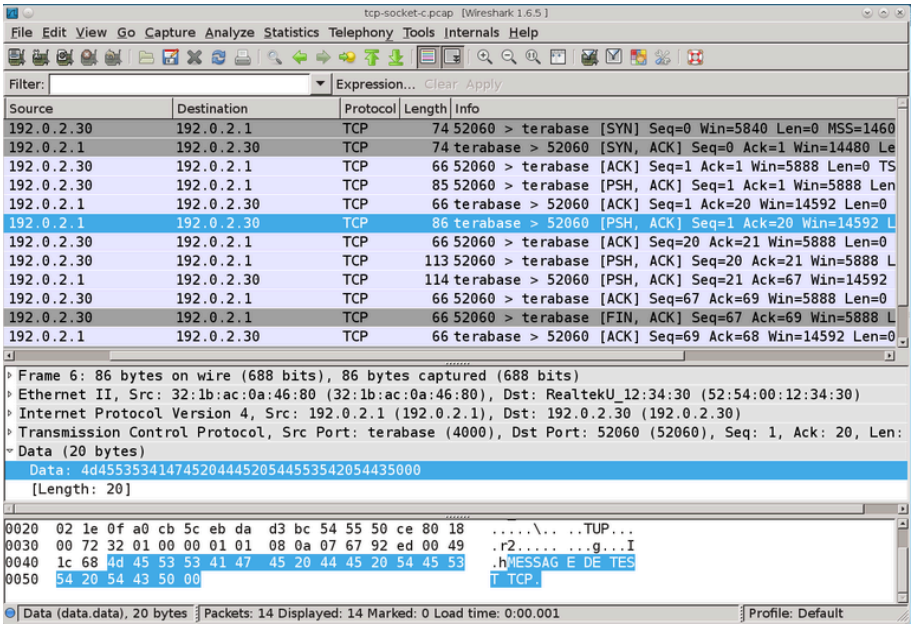
- Côté client :

```
$ ./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 : 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.

Beej's Guide to Network Programming

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

Modélisations réseau

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

Adressage IPv4

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