Initiation au développement C sur les sockets IPv4 & IPv6

Philippe Latu
philippe.latu(at)inetdoc.net

http://www.inetdoc.net

Résumé

Ce support est le second volet sur l'initiation au développement réseau sur les sockets à partir du code le plus minimaliste. On utilise à nouveau les fonctions des bibliothèques standard du Langage C et on s'appuie sur les protocoles IPv4 & IPv6 en couche réseau. Les exemples de code présentés ici sont dits dual stack. Ils fonctionnent indifféremment avec l'un et|ou l'autre des deux versions des protocoles de couche réseau.

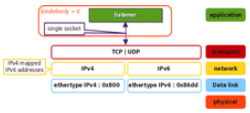


Table des matières

1. Copyright et Licence

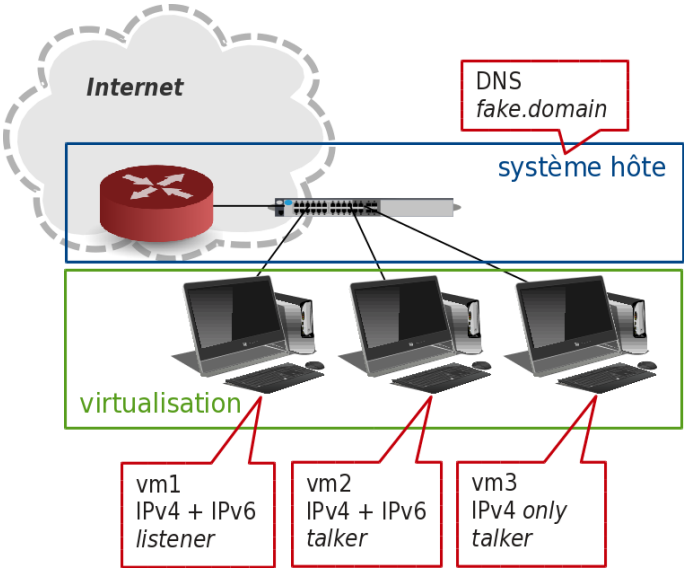
1.1. Meta-information

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-4and6.pdf`[3].

[1] http://www.docbook.org
[2] http://www.debian.org
[3] http://www.inetdoc.net/pdf/socket-c-4and6.pdf

2. Contexte de développement

Le parti pris de ce document est l'utilisation conjointe des deux protocoles de la couche réseau : IPv4 et IPv6. Dans ce but, on commence par mettre en place une petite infrastructure de test comprenant trois hôtes ayant chacun un rôle défini. Les rôles dépendent de l'application reprise du premier document Initiation au développement C sur les sockets[4]. Un hôte serveur ou listener reçoit les chaînes de caractères émises par les hôtes client ou talker. Le serveur convertit les chaînes de caractères en majuscules et les retransmet vers les client.



Voici le tableau des paramètres d'affection des rôles, de noms de domaine, d'adressage IPv4 et IPv6.

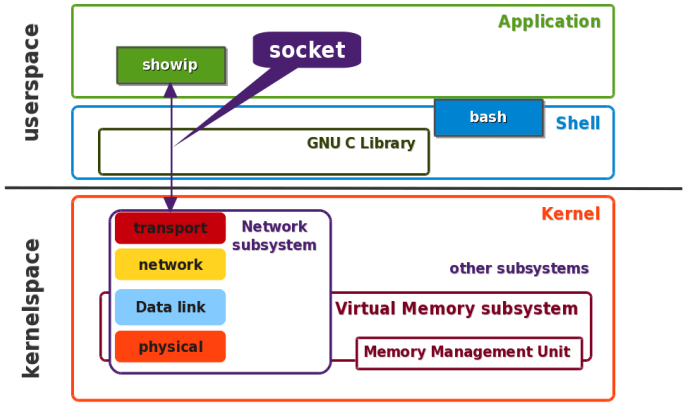
Tableau 1. Paramètres des hôtes

Nom	Rôle	Adresse IPv4	Adresse IPv6
cooper.fake.domain	routeur, serveur DNS	`192.0.2.1/27`	`2001:db8:feb2:10::1/64`
vm1.fake.domain	serveur, listener	`192.0.2.11/27`	`2001:db8:feb2:10::11/64`
vm2.fake.domain	client, talker	`192.0.2.12/27`	`2001:db8:feb2:10::12/64`
vm3.fake.domain	client, talker	`192.0.2.13/27`	-

Du point de vue système, on reprend le modèle en trois couches : kernel, shell et application. Nos applications sont exécutées à partir du shell et font appel au sous-système réseau du noyau (kernel) via des appels systèmes utilisant les bibliothèques standard.

Du point de vue modélisation réseau, les niveaux allant de la couche physique jusqu'à la couche transport sont intégrés dans le sous-système réseau du noyau et nos applications sont placées dans la couche éponyme. Le canal de communication entre la couche application et les niveaux inférieurs est appelé «prise» ou socket.

[4] http://www.inetdoc.net/dev/socket-c/



Relativement au premier document Initiation au développement C sur les sockets[5], les instructions de compilation et d'exécution des applications ne changent pas. Un exemple de Makefile est donné en annexe.

L'utilisation du service de noms de domaine (DNS) est une nouveauté. L'utilisation conjointe des deux versions de protocole réseau suppose qu'à un même nom d'hôte on associe plusieurs adresses IP. Ainsi, les tests pourront se faire de façon transparente. La même application doit pouvoir s'exécuter sur un système qui ne supporte qu'une version de protocole ou sur un système qui supporte les deux versions. On parle alors de système single stack ou dual stack. Le service DNS est donc une solution pratique d'évaluation des différentes combinaisons. La configuration du service sort du cadre de ce document. Les fichiers de déclaration de la zone `fake.domain` sont donnés en annexe.

Côté bibliothèques standard, le remplacement de la fonction `gethostbyname()` par `getaddrinfo()` constitue le premier gros changement. Cette évolution dans l'utilisation des appels système justifie une section à part entière dans la description du code. Là encore, l'utilisation des protocoles IPv4 et IPv6 entraîne une modification des enregistrements utilisés pour décrire les attributs des adresses associées à un hôte ou une interface réseau.

3. Utilisation de getaddrinfo()

L'objectif de cette section est de fournir un programme minimaliste qui affiche les adresses IP associées à un hôte ou à une interface réseau. Le code source de ce programme fait donc appel à la fonction `getaddrinfo()`. Cette fonction offre de nombreuses options mais son utilisation reste simple. Elle s'appuie sur l'enregistrement de type `addrinfo`. Ce type permet de constituer une liste chaînée des différentes adresses disponibles.

> **Note**
>
> Cette section suit la démarche proposée dans le livre Beej's Guide to Network Programming[6]. Relativement au code proposé dans cet excellent guide, les modifications apportées ici sont marginales. Elles ont cependant une incidence sur l'organisation du code des sections suivantes.

Le programme showip est constitué d'un appel à la fonction `getaddrinfo()` suivi d'une boucle de parcours des enregistrements renseignés lors de l'appel. Cette boucle de parcours est reprise ensuite dans tous les autres programmes de ce document.

[5] http://www.inetdoc.net/dev/socket-c/
[6] http://beej.us/guide/bgnet/

Appel à getaddrinfo()

```
  struct addrinfo hints, *res, *p;❶

<snipped/>
  memset(&hints, 0, sizeof hints);❷
  hints.ai_family = AF_UNSPEC; // IPv4 ou IPv6
  hints.ai_socktype = SOCK_STREAM; // Une seule famille de socket

  if ((status = getaddrinfo(argv[1]❸, NULL, &hints, &res❹)) != 0) {
    fprintf(stderr, "getaddrinfo: %s\n", gai_strerror(status)❺);
    return 2;
  }
```

❶ On utilise 3 variables de type `addrinfo`. L'enregistrement `hints` sert à positionner des options avant l'appel à `getaddrinfo()` et les deux pointeurs servent respectivement à adresser le premier résultat et à parcourir la liste des enregistrements contenant les informations sur les adresses IP.

❷ Les options choisies pour l'appel à `getaddrinfo()` permettent d'orienter les résultats. Ici, on souhaite obtenir les informations sur les adresses IPv4 (et|ou) IPv6. De plus, on est obligé de choisir une famille de socket même si nous n'avons pas l'intention de poursuivre avec l'ouverture d'un socket.

❸ Les paramètres du programme showip sont passés directement en ligne de commande de façon classique ; `argv[1]` correspond à la chaîne de caractères décrivant l'hôte dont on souhaite obtenir la liste des adresses réseau.

❹ Le pointeur `*res` indique l'adresse du premier enregistrement réponse au retour de l'appel à `getaddrinfo()`.

❺ En cas d'erreur sur l'interprétation de la chaîne fournie dans `argv[1]`, la variable `status` reçoit une valeur interprétée par la fonction `gai_strerror()`.

Boucle de parcours des enregistrements addrinfo

```
<snipped/>

  printf("IP addresses for %s:\n\n", argv[1]);

  p = res;❶
  while (p != NULL) {❷

    // Identification de l'adresse courante

<snipped/>

    // Adresse suivante
    p = p->ai_next;❸
  }
```

❶ Le pointeur `p` reçoit l'adresse du premier enregistrement suite à l'appel à la fonction `getaddrinfo()`.

❷ Si cette adresse vaut `NULL`, il n'y a plus d'enregistrement d'adresse dans la liste et on sort de la boucle.

❸ On fait pointer `p` sur l'enregistrement d'adresse suivant.

Cette technique de parcours des enregistrements de type `addrinfo` est reprise dans les sections suivantes pour définir les conditions d'ouverture des «prises» réseau ou sockets.

Exemple d'utilisation du programme showip

L'exemple donné ci-dessous montre que deux adresses IP sont assoicées au nom d'hôte `vm1.fake.domain`.

```
$ ./showip.o vm1.fake.domain
IP addresses for vm1.fake.domain:

 IPv6: 2001:db8:feb2:10::11
 IPv4: 192.0.2.11
```

Code source complet du programme showip.c

```
/*
*  showip.c -- Affiche les adresses IP correspondant à un nom d'hôte donné en
*  ligne de commande
*
*  This code was first published in the
*  Beej's Guide to Network Programming
*  Credit has to be given back to beej(at)beej.us
*/
#include <stdio.h>
#include <string.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netdb.h>
#include <arpa/inet.h>

int main (int argc, char *argv[]) {

  struct addrinfo hints, *res, *p;
  void *addr;
  int status;
  char ipstr[INET6_ADDRSTRLEN], ipver;

  if (argc != 2) {
    fprintf(stderr, "usage: showip hostname\n");
    return 1;
  }

  memset(&hints, 0, sizeof hints);
  hints.ai_family = AF_UNSPEC; // IPv4 ou IPv6
  hints.ai_socktype = SOCK_STREAM; // Une seule famille de socket

  if ((status = getaddrinfo(argv[1], NULL, &hints, &res)) != 0) {
    fprintf(stderr, "getaddrinfo: %s\n", gai_strerror(status));
    return 2;
  }

  printf("IP addresses for %s:\n\n", argv[1]);

  p = res;
  while (p != NULL) {

    // Identification de l'adresse courante
    if (p->ai_family == AF_INET) { // IPv4
      struct sockaddr_in *ipv4 = (struct sockaddr_in *)p->ai_addr;
      addr = &(ipv4->sin_addr);
      ipver = '4';
    }
    else { // IPv6
      struct sockaddr_in6 *ipv6 = (struct sockaddr_in6 *)p->ai_addr;
      addr = &(ipv6->sin6_addr);
      ipver = '6';
    }

    // Conversion de l'adresse IP en une chaîne de caractères
    inet_ntop(p->ai_family, addr, ipstr, sizeof ipstr);
    printf(" IPv%c: %s\n", ipver, ipstr);

    // Adresse suivante
    p = p->ai_next;
  }

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

  return 0;
}
```

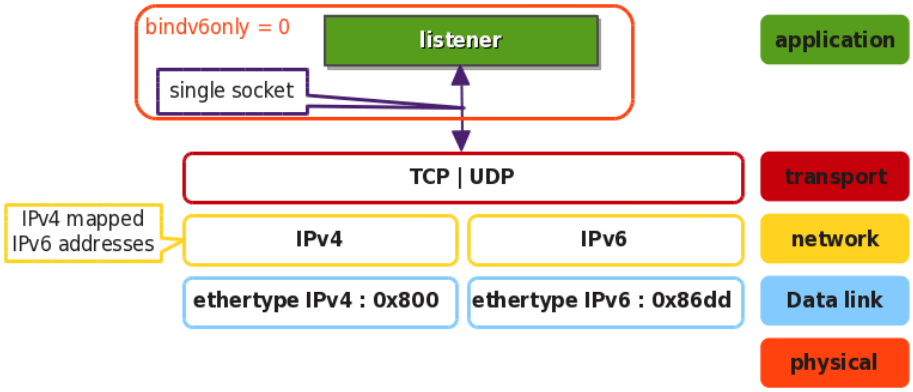
4. Choisir entre socket simple ou double

Avant d'aborder le codage des applications supportant les deux protocoles de couche réseau, il convient de poser les termes du débat. Deux lignes de conduite s'affrontent depuis des années sans que l'une ait réussi à prendre l'ascendant sur l'autre. L'objectif du présent document n'est pas d'affirmer une position mais d'exposer les arguments des uns et des autres. Ensuite, on étudiera dans les sections suivantes le même programme écrit suivant les deux modes de fonctionnement possibles.

Codage basé sur un socket simple

On pourrait qualifier cette ligne de conduite comme étant la plus académique. En effet, en reprenant les principes énoncés dans le cours sur la modélisation (voir Modélisations réseau[7]), on postule que les traitements réalisés au niveau de chaque couche doivent être indépendants les uns des autres. Dans notre contexte, un programme de la couche application ne doit pas dépendre des protocoles de la couche réseau. Le principal argument en faveur de cette position est la pérennité du modèle. Si les traitements entre couches introduisent des relations de dépendance multiples, nous allons très vite être confronté à un écheveau inextricable. Plus il y aura de dépendances, moins le modèle sera évolutif et pérenne dans le temps.

On peut représenter cette solution comme suit.



Pour satisfaire le critère sur l'unicité du canal de communication entre la couche réseau et la couche transport, il est nécessaire d'établir une correspondance automatique entre les adresses IPv4 et les adresses IPv6. On parle de IPv4-mapped IPv6 addresses. Par exemple, l'adresse IPv4 `192.0.2.10` devient `::ffff:192.0.2.10` après correspondance.

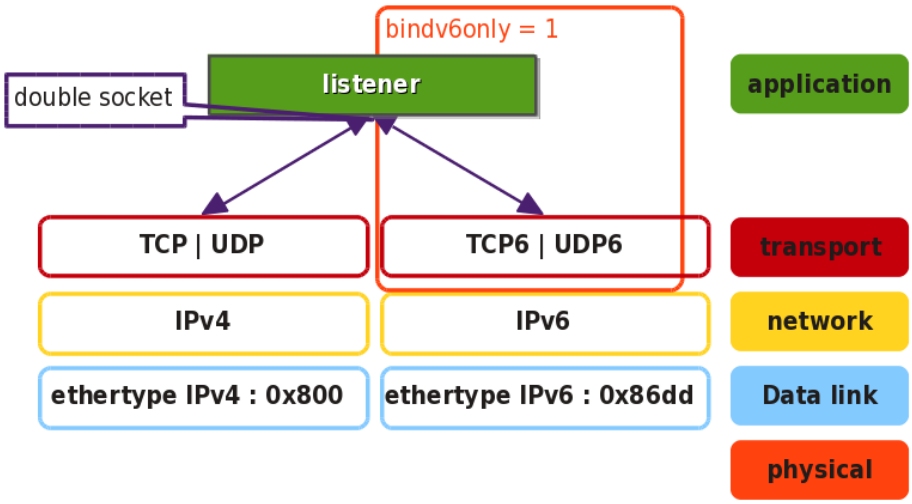
Codage basé sur deux sockets non bloquants

La seconde ligne de conduite s'appuie sur le fait que le support de l'option `bindv6only` n'est pas uniforme entre les différents systèmes d'exploitation. On observe des comportements différents entre les distributions BSD, Linux et les systèmes propriétaires. Cette option, que l'on applique à l'échelle du système ou dans une application, sert à définir si une prise réseau (socket) doit exclusivement être associée au seul protocole réseau IPv6 ou non.

Les partisans de cette position souhaitent que tous les programmes de la couche application gèrent deux sockets distincts. Cette solution a un impact important sur le codage des applications en langage C dans les domaines voisins de l'espace mémoire noyau comme les systèmes embarqués. Pour les langages «bien encrés» dans l'espace utilisateur, le recours à des bibliothèques de haut niveau permet de rendre l'opération transparente.

On peut représenter cette solution comme suit.

[7] http://www.inetdoc.net/articles/modelisation/



Pour qu'un unique processus puisse exploiter deux canaux de communication entre les couches application et transport, il est nécessaire d'utiliser des appels système non bloquants et de scruter les deux canaux en attente d'évènements. On a alors recours à la fonction `select()` qui permet de mettre en place une boucle de scrutation (polling).

Programmes et infrastructure de test

Voici un tableau de synthèse des tests effectués avec les différents codes proposés dans ce document.

Tableau 2. Protocole de couche réseau utilisé suivant les conditions de tests

client ou talker	**Serveur ou** listener
	socket **unique**
	`bindv6only = 0`
	`vm1.fake.domain`
		Serveur ou listener
		socket **double**
		`bindv6only = 1` -> socket **IPv6**
		`vm1.fake.domain`

Client dual stack
`disable_ipv6 = 0`
`vm2.fake.domain`
	IPv6	IPv6

Client single stack
`disable_ipv6 = 1`
`vm3.fake.domain`
	IPv6
	IPv4-mapped IPv6 addresses
		IPv4

5. Client ou talker

Relativement aux conditions énoncées dans la section précédente (Tableau 2, « Protocole de couche réseau utilisé suivant les conditions de tests »), l'objectif ici est de présenter un programme pour chaque protocole de la couche transport qui fonctionne indifféremment dans les deux modes : dual stack avec IPv4 + IPv6 et single stack avec IPv4 uniquement.

Dans ce but, on reprend le code donné dans la Section 3, « Utilisation de getaddrinfo() ». Suivant le contenu des enregistrements de type `addrinfo` renseignés par l'appel à `getaddrinfo()`, le choix du protocole de couche réseau se fait automatiquement à partir de la configuration système. Sur un système dual stack l'enregistrement IPv6 est en première position alors que sur un système single stack, seul l'enregistrement IPv4 est disponible. Par conséquent, la boucle de scrutation des enregistrements s'arrête à l'ouverture de la première prise réseau (socket) valide.

Sur un système GNU/Linux, on consulte l'indicateur d'état `disable_ipv6` à l'aide de l'instruction suivante.

```
$ cat /proc/sys/net/ipv6/conf/all/disable_ipv6
0
```

Dans l'infrastructure de test utilisée pour ce document, on a créé le fichier `/etc/sysctl.d/disableipv6.conf` sur le système `vm3.fake.domain` pour désactiver l'utilisation du protocole IPv6. Le contenu de ce fichier est le suivant.

```
$ cat /etc/sysctl.d/disableipv6.conf
net.ipv6.conf.all.disable_ipv6=1
net.ipv6.conf.default.disable_ipv6=1
```

Boucle de parcours des enregistrements addrinfo

```
<snipped/>

  p = servinfo;
  while((p != NULL)❶ && !sockSuccess)❷ {

    // Identification de la famille d'adresse
    if (p->ai_family == AF_INET)
      puts("Open IPv4 socket");
    else
      puts("Open IPv6 socket");

    if ((socketDescriptor = socket (p->ai_family, p->ai_socktype, p->ai_protocol)) == -1) {
      perror("socket:");
      sockSuccess = false; // Echec ouverture socket
      p = p->ai_next;      // Enregistrement d'adresse suivant
    }
    else // La prise réseau est valide
      sockSuccess = true;
  }

  if (p == NULL) {❸
    fputs("Création de socket impossible", stderr);
    return 2;
  }
```

❶ Dès que l'adresse du pointeur `p` vaut `NULL`, il n'y a plus d'enregistrement à examiner.

❷ La variable booléenne `sockSuccess` est initialisée à la valeur `false`. Dès qu'une prise réseau a été ouverte avec succès, on arrête la scrutation des enregistrements.

❸ Si, en sortie de la boucle de scrutation des enregistrements de type `addrinfo` l'adresse du pointeur `p` vaut `NULL`, aucune prise réseau n'a été ouverte. L'exécution du programme s'arrête là.

Les autres particularités du programme client UDP ou udp-talker sont décrites dans le support Initiation au développement C sur les sockets[8]. Il faut simplement noter que la fonction `select()` est utilisée ici pour gérer une temporisation d'attente de la réponse du serveur. Le protocole de couche transport UDP n'est pas orienté connexion et n'offre aucun service de fiabilisation. Il incombe donc à la couche application d'assurer une forme de détection d'erreur. Dans notre cas, le client attend la réponse du traitement effectué par le serveur pendant une seconde. Si aucune réponse n'a été reçue dans le temps imparti, il faut considérer que le message émis a été perdu.

Code source complet du programme udp-talker.c

```
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <stdbool.h>
#include <netdb.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <arpa/inet.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
```

[8] http://www.inetdoc.net/dev/socket-c/

```
#define str(x) # x
#define xstr(x) str(x)

int main()
{
  int socketDescriptor, status;
  unsigned int msgLength;
  struct addrinfo hints, *servinfo, *p;
  struct timeval timeVal;
  fd_set readSet;
  char msg[MSG_ARRAY_SIZE], serverPort[PORT_ARRAY_SIZE];
  bool sockSuccess = false;

  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_UNSPEC;
  hints.ai_socktype = SOCK_DGRAM;

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

  // Scrutation des résultats et création de socket
  // Sortie après création de la première prise réseau valide
  p = servinfo;
  while((p != NULL) && !sockSuccess) {

    // Identification de la famille d'adresse
    if (p->ai_family == AF_INET)
      puts("Open IPv4 socket");
    else
      puts("Open IPv6 socket");

    if ((socketDescriptor = socket (p->ai_family, p->ai_socktype, p->ai_protocol)) == -1) {
      perror("socket:");
      sockSuccess = false; // Echec ouverture socket
      p = p->ai_next;      // Enregistrement d'adresse suivant
    }
    else // La prise réseau est valide
      sockSuccess = true;
  }

  if (p == NULL) {
    fputs("Création de socket impossible", stderr);
    return 2;
  }

  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,
                 p->ai_addr, p->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);

  freeaddrinfo(servinfo);

  return 0;
}
```

Code source du correctif tcp-talker.c.patch

Le passage du protocole de couche transport UDP au protocole TCP ne présente aucune singularité relativement au document initial : Initiation au développement C sur les sockets[9]. Le protocole TCP est orienté connexion et assure la fiabilisation des échanges de bout en bout. Le correctif suivant fait apparaître l'appel à la fonction `connect()` qui correspond à la phase d'établissement de la connexion. De plus, le recours à la temporisation d'attente de réponse devient inutile. Les variables de gestion du temps et l'appel à `select()` est donc supprimé.

```
--- udp-talker.c 2014-04-12 10:37:35.000000000 +0200
+++ tcp-talker.c 2013-04-07 11:35:54.000000000 +0200
@@ -22,8 +22,6 @@
   int socketDescriptor, status;
   unsigned int msgLength;
   struct addrinfo hints, *servinfo, *p;
-  struct timeval timeVal;
-  fd_set readSet;
   char msg[MSG_ARRAY_SIZE], serverPort[PORT_ARRAY_SIZE];
   bool sockSuccess = false;

@@ -37,7 +35,7 @@

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

   if ((status = getaddrinfo(msg, serverPort, &hints, &servinfo)) != 0) {
```

[9] http://www.inetdoc.net/dev/socket-c/

```
     fprintf(stderr, "getaddrinfo: %s\n", gai_strerror(status));
@@ -55,7 +53,7 @@
     else
       puts("Open IPv6 socket");

-    if ((socketDescriptor = socket (p->ai_family, p->ai_socktype, p->ai_protocol)) == -1) {
+    if ((socketDescriptor = socket(p->ai_family, p->ai_socktype, p->ai_protocol)) == -1) {
       perror("socket:");
       sockSuccess = false; // Echec ouverture socket
       p = p->ai_next;      // Enregistrement d'adresse suivant
@@ -69,6 +67,12 @@
     return 2;
   }

+  if (connect(socketDescriptor, p->ai_addr, p->ai_addrlen) == -1) {
+    perror("connect");
+    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.");
@@ -92,13 +96,6 @@
         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) {
@@ -106,13 +103,10 @@
           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. Serveur ou listener socket unique

Comme dans le cas du programme client, l'utilisation des programmes présentés dans cette section est définie dans le Tableau 2, « Protocole de couche réseau utilisé suivant les conditions de tests ». On propose un programme pour chaque protocole de la couche transport qui utilise une prise réseau (socket) unique indépendante du protocole de couche réseau utilisé.

Dans ce but, on reprend à nouveau la boucle de parcours des enregistrements de type `addrinfo` à la suite de l'appel à la fonction `getaddrinfo()`. La boucle de parcours s'interrompt dès qu'une prise réseau (socket) est ouverte avec succès. Ce principe est donc identique à celui adopté pour le programme client ou talker (voir Section 5, « Client ou talker »). Le code présenté ci-dessous se distingue du précédent par le nombre d'étapes à franchir avec succès pour confirmer l'ouverture de la prise réseau (socket).

1. Appel à la fonction `socket()`.

2. Appel à la fonction `setsockopt()` si la famille de socket utilisée correspond au protocole IPv6. C'est grâce à cet appel que l'on applique l'option `bindv6only=0` qui assure la prise en charge transparente des deux protocoles de couche réseau.

3. Appel à la fonction `bind()`.

Exemple d'exécution des programmes udp-listener et udp-talker

- Copie d'écran côté serveur ou listener.

```
$ ./udp-listener.o
Entrez le numéro de port utilisé en écoute (entre 1500 et 65000) :
5000
 IPv6: ::
Attente de requête sur le port 5000
>>  Depuis [2001:db8:feb2:10::12]:54272
>>  Message reçu : Message émis depuis vm2.fake.domain
>>  Depuis [::ffff:192.0.2.13]:51254
>>  Message reçu : message émis depuis vm3.fake.domain
```

Comme indiqué dans le Tableau 2, « Protocole de couche réseau utilisé suivant les conditions de tests », le client `vm2.fake.domain` apparaît avec son adresse IPv6 tandis que le client `vm4.fake.domain` apparaît avec une adresse IPv6 établie par correspondance avec son adresse IPv4 (IPv4-mapped IPv6 address).

Les informations relatives à l'ouverture de la prise réseau (socket) au niveau système peuvent être obtenues au moins de trois façon différentes à l'aide des commandes **netstat**, **lsof** et **ss**.

```
$ netstat -aup
(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.)
Connexions Internet actives (serveurs et établies)
Proto Recv-Q Send-Q Adresse locale    Adresse distante   Etat   PID/Program name
udp6       0      0 [::]:5000         [::]:*                    10839/udp-listener.
```

```
$ lsof -i
COMMAND     PID USER   FD   TYPE DEVICE SIZE/OFF NODE NAME
udp-liste 10839  etu    3u  IPv6  62111      0t0  UDP *:5000
```

```
$ ss -lup
State      Recv-Q Send-Q    Local Address:Port    Peer Address:Port
UNCONN     0      0         :::5000               :::*        users:(("udp-listener.o",10839,3))
```

- Copie d'écran côté client ou talker dual stack ; hôte `vm2.fake.domain`.

```
$ ./udp-talker.o
Entrez le nom du serveur ou son adresse IP :
vm1.fake.domain
Entrez le numéro de port du serveur :
5000
Open IPv6 socket

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 80 caractères,
seuls les 80 premiers caractères seront utilisés.

Saisie du message :
Message émis depuis vm2.fake.domain
Message traité : MESSAGE éMIS DEPUIS VM2.FAKE.DOMAIN
Saisie du message :
.
```

- Copie d'écran côté client ou talker single stack IPv4 ; hôte `vm3.fake.domain`.

```
$ ./udp-talker.o
Entrez le nom du serveur ou son adresse IP :
vm1.fake.domain
Entrez le numéro de port du serveur :
5000
Open IPv4 socket

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 80 caractères,
seuls les 80 premiers caractères seront utilisés.

Saisie du message :
message émis depuis vm3.fake.domain
Message traité : MESSAGE éMIS DEPUIS VM3.FAKE.DOMAIN
Saisie du message :
.
```

Code source complet du programme udp-listener.c

```
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <ctype.h>
#include <string.h>
#include <stdbool.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)

// extraction adresse IPv4 ou IPv6:
void *get_in_addr(struct sockaddr *sa) {
    if (sa->sa_family == AF_INET) {
        return &(((struct sockaddr_in*)sa)->sin_addr);
    }

    return &(((struct sockaddr_in6*)sa)->sin6_addr);
}

// extraction numéro de port
unsigned short int get_in_port(struct sockaddr *sa) {
    if (sa->sa_family == AF_INET) {
        return ((struct sockaddr_in*)sa)->sin_port;
    }

    return ((struct sockaddr_in6*)sa)->sin6_port;
}

int main() {

  int listenSocket, status, recv, i;
  unsigned short int msgLength;
  struct addrinfo hints, *servinfo, *p;
  struct sockaddr_storage clientAddress;
  socklen_t clientAddressLength = sizeof clientAddress;
  void *addr;
  char msg[MSG_ARRAY_SIZE], listenPort[PORT_ARRAY_SIZE], ipstr[INET6_ADDRSTRLEN], ipver;
  int optval = 0; // socket unique et IPV6_V6ONLY à 0
  bool sockSuccess = false;

  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_INET6;      // IPv6 et IPv4 mappées
  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;
  }

  // Scrutation des résultats et création de socket
  // Sortie après création de la première «prise»
  p = servinfo;
  while((p != NULL) && !sockSuccess) {

    // Identification de l'adresse courante
    if (p->ai_family == AF_INET) { // IPv4
      struct sockaddr_in *ipv4 = (struct sockaddr_in *)p->ai_addr;
      addr = &(ipv4->sin_addr);
      ipver = '4';
    }
    else { // IPv6
      struct sockaddr_in6 *ipv6 = (struct sockaddr_in6 *)p->ai_addr;
      addr = &(ipv6->sin6_addr);
      ipver = '6';
    }

    // Conversion de l'adresse IP en une chaîne de caractères
    inet_ntop(p->ai_family, addr, ipstr, sizeof ipstr);
    printf(" IPv%c: %s\n", ipver, ipstr);

    if ((listenSocket = socket(p->ai_family, p->ai_socktype, p->ai_protocol)) == -1) {
      perror("socket:");
      sockSuccess = false; // Echec ouverture socket
      p = p->ai_next;      // Enregistrement d'adresse suivant
    }

    else if ((p->ai_family == AF_INET6) && // IPv6 uniquement
             (setsockopt(listenSocket, IPPROTO_IPV6, IPV6_V6ONLY, &optval, sizeof optval) == -1)) {
      close(listenSocket);
      perror("setsockopt:");
      sockSuccess = false; // Echec option bindv6only=0
      p = p->ai_next;      // Enregistrement d'adresse suivant
    }

    else if (bind(listenSocket, p->ai_addr, p->ai_addrlen) == -1) {
      close(listenSocket);
      perror("bind:");
      sockSuccess = false; // Echec socket en écoute
      p = p->ai_next;      // Enregistrement d'adresse suivant
    }

    else // La prise est bien ouverte
      sockSuccess = true;
  }

  if (p == NULL) {
    fputs("Création de socket impossible\n", stderr);
    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 ((recv = recvfrom(listenSocket, msg, sizeof msg, 0,
                         (struct sockaddr *) &clientAddress,
                         &clientAddressLength)) == -1) {
      perror("recvfrom:");
      exit(EXIT_FAILURE);
    }

    if ((msgLength = strlen(msg)) > 0) {
      // Affichage de l'adresse IP du client.
      inet_ntop(clientAddress.ss_family, get_in_addr((struct sockaddr *)&clientAddress),
                ipstr, sizeof ipstr);
      printf(">>  Depuis [%s]:", ipstr);

      // Affichage du numéro de port du client.
      printf("%hu\n", ntohs(get_in_port((struct sockaddr *)&clientAddress)));

      // 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:");
        exit(EXIT_FAILURE);
      }
    }
  }

  // Jamais atteint
  return 0;
}
```

Code source du correctif tcp-listener.c.patch

Toujours comme dans le cas du programme client ou talker, le passage du protocole de couche transport UDP au protocole TCP ne présente aucune modification de fond dans la gestion de prise réseau (socket). En revanche, le protocole TCP est orienté connexion et l'échange d'information est toujours précédé de l'établissement de la connexion. Côté client, on fait appel à la fonction `connect()` et côté serveur on fait appel à la fonction `accept()`.

```
--- udp-listener.c 2013-04-06 10:24:47.000000000 +0200
+++ tcp-listener.c 2013-04-07 11:35:54.000000000 +0200
@@ -12,6 +12,7 @@
 #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
@@ -37,7 +38,7 @@

 int main() {

-  int listenSocket, status, recv, i;
+  int listenSocket, connectSocket, status, i;
   unsigned short int msgLength;
   struct addrinfo hints, *servinfo, *p;
   struct sockaddr_storage clientAddress;
@@ -53,7 +54,7 @@

   memset(&hints, 0, sizeof hints);
   hints.ai_family = AF_INET6;      // IPv6 et IPv4 mappées
-  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) {
```

```
@@ -115,42 +116,54 @@
   // 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 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 ((recv = recvfrom(listenSocket, msg, sizeof msg, 0,
+    // Appel bloquant en attente d'une nouvelle connexion
+    // connectSocket est la nouvelle prise utilisée pour la connexion active
+    if ((connectSocket = accept(listenSocket,
                          (struct sockaddr *) &clientAddress,
                          &clientAddressLength)) == -1) {
-      perror("recvfrom:");
+      perror("accept:");
+      close(listenSocket);
       exit(EXIT_FAILURE);
     }

-    if ((msgLength = strlen(msg)) > 0) {
       // Affichage de l'adresse IP du client.
       inet_ntop(clientAddress.ss_family, get_in_addr((struct sockaddr *)&clientAddress),
                 ipstr, sizeof ipstr);
-      printf(">>  Depuis [%s]:", ipstr);
+    printf(">>  connecté à [%s]:", ipstr);

       // Affichage du numéro de port du client.
       printf("%hu\n", ntohs(get_in_port((struct sockaddr *)&clientAddress)));

-      // Affichage de la ligne reçue
-      printf(">>  Message reçu : %s\n", msg);
+    // Mise à zéro du tampon de façon à connaître le délimiteur
+    // de fin de chaîne.
+    memset(msg, 0, sizeof msg);
+    while (recv(connectSocket, msg, sizeof msg, 0) > 0)
+      if ((msgLength = strlen(msg)) > 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, 0) == -1) {
+          perror("send:");
+          close(listenSocket);
         exit(EXIT_FAILURE);
       }
+
+        memset(msg, 0, sizeof msg);  // Mise à zéro du tampon
     }
   }
```

Exemple d'exécution des programmes tcp-listener et tcp-talker

En faisant abstraction des modes de fonctionnement des deux protocoles de couche transport, on constate que le comportement des programmes est identique.

- Copie d'écran côté serveur ou listener.

```
$ ./tcp-listener.o
Entrez le numéro de port utilisé en écoute (entre 1500 et 65000) :
5000
 IPv6: ::
Attente de connexion sur le port 5000
>>  connecté à [2a01:240:feb2:10::12]:32892
  --  Message émis depuis vm2.fake.domain
Attente de connexion sur le port 5000
>>  connecté à [::ffff:192.0.2.13]:40622
  --  message émis depuis vm3.fake.domain
Attente de connexion sur le port 5000
```

Tout comme avec le protocole UDP, les informations relatives à l'ouverture de la prise réseau (socket) au niveau système peuvent être obtenues au moins de trois façon différentes à l'aide des commandes **netstat**, **lsof** et **ss**. À la différence du cas précédent, les résultats font apparaître la connexion active depuis le poste client ou talker.

```
$ netstat -atp |grep 5000
(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.)
tcp6       0      0 [::]:5000               [::]:*                  LISTEN      10897/tcp-listener.
tcp6       0      0 vm1.fake.domain:5000    vm2.fake.domain:32912   ESTABLISHED 10897/tcp-listener.
```

```
$ lsof -i
COMMAND     PID USER   FD   TYPE DEVICE SIZE/OFF NODE NAME
tcp-liste 10897  etu    3u  IPv6  62332      0t0  TCP *:5000 (LISTEN)
tcp-liste 10897  etu    4u  IPv6  62333      0t0  TCP vm1.fake.domain:5000->vm2.fake.domain:32912 (ES
```

```
$ ss -tre
State      Recv-Q Send-Q    Local Address:Port        Peer Address:Port
ESTAB      0      0         vm1.fake.domain:5000      vm2.fake.domain:32912    uid:1000 ino:62333 sk:
ESTAB      0      0         vm1.fake.domain:ssh       cooper.fake.domain:60445   timer:(keepalive,84m
```

- Copie d'écran côté client ou talker dual stack ; hôte `vm2.fake.domain`.

```
$ ./tcp-talker.o
Entrez le nom du serveur ou son adresse IP :
vm1.fake.domain
Entrez le numéro de port du serveur :
5000
Open IPv6 socket

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 80 caractères,
seuls les 80 premiers caractères seront utilisés.

Saisie du message :
Message émis depuis vm2.fake.domain
Message traité : MESSAGE éMIS DEPUIS VM2.FAKE.DOMAIN
Saisie du message :
.
```

- Copie d'écran côté client ou talker single stack IPv4 ; hôte `vm3.fake.domain`.

```
$ ./tcp-talker.o
Entrez le nom du serveur ou son adresse IP :
vm1.fake.domain
Entrez le numéro de port du serveur :
5000
Open IPv4 socket

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 80 caractères,
seuls les 80 premiers caractères seront utilisés.

Saisie du message :
message émis depuis vm3.fake.domain
Message traité : MESSAGE éMIS DEPUIS VM3.FAKE.DOMAIN
Saisie du message :
.
```

7. Serveur ou listener socket double

Comme dans les deux sections précédentes, l'utilisation des programmes présentés dans cette section est définie dans le Tableau 2, « Protocole de couche réseau utilisé suivant les conditions de tests ». On propose un programme pour chaque protocole de la couche transport qui utilise deux prises réseau (socket) ; une par protocole de couche réseau.

La boucle de parcours des enregistrements de type `addrinfo` est modifiée de façon à ce qu'une prise réseau soit ouverte par famille de protocole de couche réseau. Il est donc nécessaire de franchir les étapes suivantes avec succès pour les deux protocoles IPv4 et IPv6.

- Pour le protocole IPv4
  1. Appel à la fonction `socket()`.
  2. Appel à la fonction `bind()`.
- Pour le protocole IPv6
  1. Appel à la fonction `socket()`.
  2. Appel à la fonction `setsockopt()` pour appliquer l'option `bindv6only=1` qui rend la prise réseau (socket) dédiée au protocole de couche réseau IPv6.
  3. Appel à la fonction `bind()`.

Par souci de cohérence avec le code du programme précédent, on utilise le même indicateur booléen `sockSuccess` comme condition de sortie de la boucle de parcours des enregistrements `addrinfo`.

Exemple d'exécution des programmes udp-listener et udp-talker

- Copie d'écran côté serveur ou listener.

```
$ ./udp-listener.o
Entrez le numéro de port utilisé en écoute (entre 1500 et 65000) :
5000
 IPv4: 0.0.0.0
 IPv6: ::
Attente de requête sur le port 5000
>>  Depuis [2a01:240:feb2:10::12]:51708
>>  Message reçu : Message UDP émis depuis le système vm2.fake.domain
>>  Depuis [192.0.2.13]:55801
>>  Message reçu : Message UDP émis depuis le système vm3.fake.domain
```

Les deux clients apparaissent avec leurs adresses IP respectives. Le système `vm2.fake.domain` a utilisé la prise réseau dédiée au protocole IPv6 tandis que le système `vm3.fake.domain` a utilisé l'autre prise réseau dédiée au protocole IPv4.

Comme dans les exemples précédents, les informations relatives à l'ouverture des deux prises réseau (sockets) sont obtenues à l'aide des commandes **netstat**, **lsof** et **ss**.

```
$ netstat -aup
(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.)
Connexions Internet actives (serveurs et établies)
Proto Recv-Q Send-Q Adresse locale          Adresse distante   Etat   PID/Program name
udp        0      0 *:5000                  *:*                       10953/udp-listener.
udp6       0      0 [::]:5000               [::]:*                    10953/udp-listener.
```

```
$ lsof -i
COMMAND     PID USER   FD   TYPE DEVICE SIZE/OFF NODE NAME
udp-liste 10953  etu    3u  IPv4  63543      0t0  UDP *:5000
udp-liste 10953  etu    4u  IPv6  63544      0t0  UDP *:5000
```

```
$ ss -lup
State      Recv-Q Send-Q    Local Address:Port    Peer Address:Port
UNCONN     0      0          *:5000                *:*        users:(("udp-listener.o",10953,3))
UNCONN     0      0         :::5000               :::*        users:(("udp-listener.o",10953,4))
```

- Copie d'écran côté client ou talker dual stack ; hôte `vm2.fake.domain`.

```
$ ./udp-talker.o
Entrez le nom du serveur ou son adresse IP :
vm1.fake.domain
Entrez le numéro de port du serveur :
5000
Open IPv6 socket

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 80 caractères,
seuls les 80 premiers caractères seront utilisés.

Saisie du message :
Message UDP émis depuis le système vm2.fake.domain
Message traité : MESSAGE UDP éMIS DEPUIS LE SYSTèME VM2.FAKE.DOMAIN
Saisie du message :
.
```

- Copie d'écran côté client ou talker single stack IPv4 ; hôte `vm3.fake.domain`.

```
$ ./udp-talker.o
Entrez le nom du serveur ou son adresse IP :
vm1.fake.domain
Entrez le numéro de port du serveur :
5000
Open IPv4 socket

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 80 caractères,
seuls les 80 premiers caractères seront utilisés.

Saisie du message :
Message UDP émis depuis le système vm3.fake.domain
Message traité : MESSAGE UDP éMIS DEPUIS LE SYSTèME VM3.FAKE.DOMAIN
Saisie du message :
.
```

Code source complet du programme udp-listener.c

```
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <ctype.h>
#include <string.h>
#include <stdbool.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)

enum IPVERSION {
 v4, v6
};

// extraction adresse IPv4 ou IPv6:
void *get_in_addr(struct sockaddr *sa) {
    if (sa->sa_family == AF_INET) {
        return &(((struct sockaddr_in*)sa)->sin_addr);
    }

    return &(((struct sockaddr_in6*)sa)->sin6_addr);
}

// extraction numéro de port
unsigned short int get_in_port(struct sockaddr *sa) {
    if (sa->sa_family == AF_INET) {
        return ((struct sockaddr_in*)sa)->sin_port;
    }

    return ((struct sockaddr_in6*)sa)->sin6_port;
}

int main() {

  int listenSocket[2], status, recv, i;
  unsigned short int msgLength;
  struct addrinfo hints, *servinfo, *p;
  struct sockaddr_storage clientAddress;
  socklen_t clientAddressLength = sizeof clientAddress;
  void *addr;
  char msg[MSG_ARRAY_SIZE], listenPort[PORT_ARRAY_SIZE], ipstr[INET6_ADDRSTRLEN];
  int optval = 1; // socket double et IPV6_V6ONLY à 1
  bool sockSuccess = false;
  struct timeval timeVal;
  fd_set readSet[2];

  listenSocket[v4] = listenSocket[v6] = -1;

  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_UNSPEC; // IPv6 et IPv4
  hints.ai_socktype = SOCK_DGRAM;
  hints.ai_flags = AI_PASSIVE; // use my IP

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

  // Scrutation des résultats et création de socket
  // Sortie après création d'une «prise» IPv4 et d'une «prise» IPv6
  p = servinfo;
  while ((p != NULL) && !sockSuccess) {

    // Identification de l'adresse courante
    if (p->ai_family == AF_INET) { // IPv4
      struct sockaddr_in *ipv4 = (struct sockaddr_in *)p->ai_addr;
      addr = &(ipv4->sin_addr);
```

```
      // conversion de l'adresse IP en une chaîne de caractères
      inet_ntop(p->ai_family, addr, ipstr, sizeof ipstr);
      printf(" IPv4: %s\n", ipstr);

      if ((listenSocket[v4] = socket(p->ai_family, p->ai_socktype, p->ai_protocol)) == -1) {
        perror("socket:"); // Echec ouverture socket
      }

      else if (bind(listenSocket[v4], p->ai_addr, p->ai_addrlen) == -1) {
        close(listenSocket[v4]);
        perror("bind:");
listenSocket[v4] = -1; // Echec socket en écoute
      }
    }
    else { // IPv6
      struct sockaddr_in6 *ipv6 = (struct sockaddr_in6 *)p->ai_addr;
      addr = &(ipv6->sin6_addr);
      // conversion de l'adresse IP en une chaîne de caractères
      inet_ntop(p->ai_family, addr, ipstr, sizeof ipstr);
      printf(" IPv6: %s\n", ipstr);

      if ((listenSocket[v6] = socket(p->ai_family, p->ai_socktype, p->ai_protocol)) == -1) {
        perror("socket:"); // Echec ouverture socket
      }

      else if (setsockopt(listenSocket[v6], IPPROTO_IPV6, IPV6_V6ONLY, &optval, sizeof optval) == -1) {
        perror("setsockopt:");
listenSocket[v6] = -1; // Echec option bindv6only=1
      }

      else if (bind(listenSocket[v6], p->ai_addr, p->ai_addrlen) == -1) {
        close(listenSocket[v6]);
        perror("bind:");
listenSocket[v6] = -1; // Echec socket en écoute
      }
    }

    if ((listenSocket[v4] != -1) && (listenSocket[v6] != -1)) // deux prises ouvertes
      sockSuccess = true;
    else
      p = p->ai_next; // Enregistrement d'adresse suivant
  }

  if (p == NULL) {
    fputs("Création de socket impossible\n", stderr);
    exit(EXIT_FAILURE);
  }

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

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

  // Utilisation de select en mode scrutation
  timeVal.tv_sec = 0;
  timeVal.tv_usec = 0;

  while (1) {

    FD_ZERO(&readSet[v4]);
    FD_SET(listenSocket[v4], &readSet[v4]);
    FD_ZERO(&readSet[v6]);
    FD_SET(listenSocket[v6], &readSet[v6]);

    if (select(listenSocket[v4]+1, &readSet[v4], NULL, NULL, &timeVal) == -1) { // IPv4
      perror("select:");
      exit(EXIT_FAILURE);
    }

    if (select(listenSocket[v6]+1, &readSet[v6], NULL, NULL, &timeVal) == -1) { // IPv6
      perror("select:");
      exit(EXIT_FAILURE);
    }
```

```
    // Mise à zéro du tampon de façon à connaître le délimiteur
    // de fin de chaîne.
    memset(msg, 0, sizeof msg);

    if (FD_ISSET(listenSocket[v4], &readSet[v4])) // IPv4
      if ((recv = recvfrom(listenSocket[v4], msg, sizeof msg, 0, (struct sockaddr *) &clientAddress,
                           &clientAddressLength)) == -1) {
        perror("recvfrom:");
        exit(EXIT_FAILURE);
      }

    if (FD_ISSET(listenSocket[v6], &readSet[v6])) // IPv6
      if ((recv = recvfrom(listenSocket[v6], msg, sizeof msg, 0, (struct sockaddr *) &clientAddress,
                           &clientAddressLength)) == -1) {
        perror("recvfrom:");
        exit(EXIT_FAILURE);
      }

    if ((msgLength = strlen(msg)) > 0) {
      // Affichage de l'adresse IP du client.
      inet_ntop(clientAddress.ss_family, get_in_addr((struct sockaddr *)&clientAddress),
                ipstr, sizeof ipstr);
      printf(">>  Depuis [%s]:", ipstr);

      // Affichage du numéro de port du client.
      printf("%hu\n", ntohs(get_in_port((struct sockaddr *)&clientAddress)));

      // 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 (clientAddress.ss_family == AF_INET) { // IPv4
        if (sendto(listenSocket[v4], msg, msgLength, 0, (struct sockaddr *) &clientAddress,
                   clientAddressLength) == -1) {
          perror("sendto:");
          exit(EXIT_FAILURE);
        }
      }
      else { // IPv6
        if (sendto(listenSocket[v6], msg, msgLength, 0, (struct sockaddr *) &clientAddress,
                   clientAddressLength) == -1) {
          perror("sendto:");
          exit(EXIT_FAILURE);
        }
      }
    }
  }

  // jamais atteint
  return 0;
}
```

Code source du correctif tcp-listener.c.patch

Relativement aux correctifs des sections précédentes, celui-ci est plus important. En effet, si le principe de détection d'évènement sur les deux prises réseau (sockets) est identique au programme UDP ci-dessus, l'appel à la fonction `accept()` est bloquant. Le code de traitement de réception et d'émission de chaîne de caractères a été dupliqué pour chaque prise réseau.

```
--- udp-listener.c 2013-04-06 10:25:50.000000000 +0200
+++ tcp-listener.c 2013-03-04 13:39:15.000000000 +0100
@@ -12,6 +12,7 @@
 #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
@@ -41,7 +42,7 @@

 int main() {

-  int listenSocket[2], status, recv, i;
+  int listenSocket[2], connectSocket, status, i;
   unsigned short int msgLength;
   struct addrinfo hints, *servinfo, *p;
   struct sockaddr_storage clientAddress;
@@ -61,7 +62,7 @@

   memset(&hints, 0, sizeof hints);
   hints.ai_family = AF_UNSPEC; // IPv6 et IPv4
-  hints.ai_socktype = SOCK_DGRAM;
+  hints.ai_socktype = SOCK_STREAM;
   hints.ai_flags = AI_PASSIVE; // use my IP

   if ((status = getaddrinfo(NULL, listenPort, &hints, &servinfo)) != 0) {
@@ -135,6 +136,19 @@
   timeVal.tv_sec = 0;
   timeVal.tv_usec = 0;

+  // Attente des requêtes des clients.
+  // Appel non bloquant et passage en mode passif
+  // Demandes d'ouverture de connexion traitées par accept
+  if (listen(listenSocket[v4], BACKLOG) == -1) {
+    perror("listen");
+    exit(EXIT_FAILURE);
+  }
+
+  if (listen(listenSocket[v6], BACKLOG) == -1) {
+    perror("listen:");
+    exit(EXIT_FAILURE);
+  }
+
   while (1) {

     FD_ZERO(&readSet[v4]);
@@ -152,56 +166,84 @@
       exit(EXIT_FAILURE);
     }

+    if (FD_ISSET(listenSocket[v4], &readSet[v4])) { // IPv4
+      // Appel bloquant en attente d'une nouvelle connexion
+      // connectSocket est la nouvelle prise utilisée pour la connexion active
+      if ((connectSocket = accept(listenSocket[v4], (struct sockaddr *) &clientAddress,
+      &clientAddressLength)) == -1) {
+        perror("accept:");
+        close(listenSocket[v4]);
+        exit(EXIT_FAILURE);
+      }
+
+      // Affichage de l'adresse IP du client.
+      inet_ntop(clientAddress.ss_family, get_in_addr((struct sockaddr *)&clientAddress),
+                ipstr, sizeof ipstr);
+      printf(">>  connecté à [%s]:", ipstr);
+
+      // Affichage du numéro de port du client.
+      printf("%hu\n", ntohs(get_in_port((struct sockaddr *)&clientAddress)));
+
     // Mise à zéro du tampon de façon à connaître le délimiteur
     // de fin de chaîne.
     memset(msg, 0, sizeof msg);
+      while (recv(connectSocket, msg, sizeof msg, 0) > 0)
+        if ((msgLength = strlen(msg)) > 0) {
+          printf("  --  %s\n", msg);
+          // Conversion de cette ligne en majuscules.
+          for (i = 0; i < msgLength; i++)
+            msg[i] = toupper(msg[i]);

-    if (FD_ISSET(listenSocket[v4], &readSet[v4])) // IPv4
```

```
-      if ((recv = recvfrom(listenSocket[v4], msg, sizeof msg, 0, (struct sockaddr *) &clientAddress,
-                           &clientAddressLength)) == -1) {
-        perror("recvfrom:");
+          // Renvoi de la ligne convertie au client.
+          if (send(connectSocket, msg, msgLength, 0) == -1) {
+            perror("send:");
+            close(listenSocket[v4]);
          exit(EXIT_FAILURE);
        }

-    if (FD_ISSET(listenSocket[v6], &readSet[v6])) // IPv6
-      if ((recv = recvfrom(listenSocket[v6], msg, sizeof msg, 0, (struct sockaddr *) &clientAddress,
+          memset(msg, 0, sizeof msg);  // Mise à zéro du tampon
+        }
+    }
+
+    if (FD_ISSET(listenSocket[v6], &readSet[v6])) { // IPv6
+      // Appel bloquant en attente d'une nouvelle connexion
+      // connectSocket est la nouvelle prise utilisée pour la connexion active
+      if ((connectSocket = accept(listenSocket[v6], (struct sockaddr *) &clientAddress,
                            &clientAddressLength)) == -1) {
-        perror("recvfrom:");
+        perror("accept:");
+        close(listenSocket[v6]);
          exit(EXIT_FAILURE);
        }

-    if ((msgLength = strlen(msg)) > 0) {
       // Affichage de l'adresse IP du client.
       inet_ntop(clientAddress.ss_family, get_in_addr((struct sockaddr *)&clientAddress),
                 ipstr, sizeof ipstr);
-      printf(">>  Depuis [%s]:", ipstr);
+      printf(">>  connecté à [%s]:", ipstr);

       // Affichage du numéro de port du client.
       printf("%hu\n", ntohs(get_in_port((struct sockaddr *)&clientAddress)));

-      // Affichage de la ligne reçue
-      printf(">>  Message reçu : %s\n", msg);
-
+      // Mise à zéro du tampon de façon à connaître le délimiteur
+      // de fin de chaîne.
+      memset(msg, 0, sizeof msg);
+      while (recv(connectSocket, msg, sizeof msg, 0) > 0)
+        if ((msgLength = strlen(msg)) > 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 (clientAddress.ss_family == AF_INET) { // IPv4
-        if (sendto(listenSocket[v4], msg, msgLength, 0, (struct sockaddr *) &clientAddress,
-                   clientAddressLength) == -1) {
-          perror("sendto:");
-          exit(EXIT_FAILURE);
-        }
-      }
-      else { // IPv6
-        if (sendto(listenSocket[v6], msg, msgLength, 0, (struct sockaddr *) &clientAddress,
-                   clientAddressLength) == -1) {
-          perror("sendto:");
+          if (send(connectSocket, msg, msgLength, 0) == -1) {
+            perror("send:");
+            close(listenSocket[v6]);
+            exit(EXIT_FAILURE);
          }
+
+          memset(msg, 0, sizeof msg);  // Mise à zéro du tampon
        }
      }
+
   }
```

```
  // jamais atteint
```

Exemple d'exécution des programmes tcp-listener et tcp-talker

Excepté le mode connecté, le comportement des programmes est à nouveau identique.

- Copie d'écran côté serveur ou listener.

```
$ ./tcp-listener.o
Entrez le numéro de port utilisé en écoute (entre 1500 et 65000) :
5000
 IPv4: 0.0.0.0
 IPv6: ::
Attente de requête sur le port 5000
>>  connecté à [2a01:240:feb2:10::12]:32906
  --  Message TCP émis depuis le système vm2.fake.domain
>>  connecté à i[192.0.2.13]:40625
  --  Message TCP émis depuis le système vm3.fake.domain
```

  Les informations relatives à l'ouverture des deux prises réseau (sockets) et aux connexions sont obtenues à l'aide des commandes **netstat**, **lsof** et **ss**.

```
$ $ netstat -atp |grep 5000
(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.)
tcp        0      0 *:5000                  *:*                     LISTEN      10964/tcp-listener.
tcp6       0      0 [::]:5000               [::]:*                  LISTEN      10964/tcp-listener.
tcp6       0      0 vm1.fake.domain:5000    vm2.fake.domain:32913   ESTABLISHED 10964/tcp-listener.
```

```
$ lsof -i
lsof -i
COMMAND     PID USER   FD   TYPE DEVICE SIZE/OFF NODE NAME
tcp-liste 10964  etu    3u  IPv4  63617      0t0  TCP *:5000 (LISTEN)
tcp-liste 10964  etu    4u  IPv6  63618      0t0  TCP *:5000 (LISTEN)
tcp-liste 10964  etu    5u  IPv6  63619      0t0  TCP vm1.fake.domain:5000->vm2.fake.domain:32913 (ES
```

```
$ ss -tre
State      Recv-Q Send-Q     Local Address:Port         Peer Address:Port
ESTAB      0      0        vm1.fake.domain:5000      vm2.fake.domain:32913    uid:1000 ino:63619 sk:f
ESTAB      0      0        vm1.fake.domain:ssh    cooper.fake.domain:60445    timer:(keepalive,52min,
```

- Copie d'écran côté client ou talker dual stack ; hôte `vm2.fake.domain`.

```
$ ./tcp-talker.o
Entrez le nom du serveur ou son adresse IP :
vm1.fake.domain
Entrez le numéro de port du serveur :
5000
Open IPv6 socket

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 80 caractères,
seuls les 80 premiers caractères seront utilisés.

Saisie du message :
Message TCP émis depuis le système vm2.fake.domain
Message traité : MESSAGE TCP éMIS DEPUIS LE SYSTèME VM2.FAKE.DOMAIN
Saisie du message :
.
```

- Copie d'écran côté client ou talker single stack IPv4 ; hôte `vm3.fake.domain`.

```
$ ./tcp-talker.o
Entrez le nom du serveur ou son adresse IP :
vm1.fake.domain
Entrez le numéro de port du serveur :
5000
Open IPv4 socket

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 80 caractères,
seuls les 80 premiers caractères seront utilisés.

Saisie du message :
Message TCP émis depuis le système vm3.fake.domain
Message traité : MESSAGE TCP éMIS DEPUIS LE SYSTèME VM3.FAKE.DOMAIN
Saisie du message :
.
```

8. Pour conclure

Les objectifs de tests donnés dans le Tableau 2, « Protocole de couche réseau utilisé suivant les conditions de tests » ont été atteints et le code proposé est fonctionnel même s'il est très certainement perfectible. Si le courage ne vous a pas manqué et que vous êtes arrivé jusqu'à la lecture de ces lignes, ce document peut être vu comme une découverte en trois étapes de l'utilisation conjointe des deux protocoles IPv4 et IPv6.

Avec le programme showip, on a étudié l'appel à la fonction `gettaddrinfo` et la représentation des données relatives à une adresse IP : l'enregistrement `addrinfo`. Le code de ce premier programme a été repris pour écrire le client talker sous deux formes ; une par protocole de couche transport. Relativement à ce premier programme on a ajouté le traitement d'ouverture et de fermeture d'une prise réseau ou socket. Ensuite, les mêmes instructions ont été exploitées pour le programme serveur listener lui aussi fourni sous deux formes. L'utilisation d'une prise réseau unique a été l'occasion de découvrir la correspondance automatique d'adresse IPv4 en IPv6 : les IPv4-mapped IPv6 addresses. Enfin, le serveur a été transformé en version double prise réseau. Cette dernière étape correspond au code le plus complexe. Elle a permis de découvrir l'utilisation de la fonction `select()` suivant différents modes.

Ainsi s'achève ce «petit panorama» des bibliothèques standard associées au sous-système réseau du noyau Linux. Le but était de démystifier l'utilisation des ces fonctions et peut-être de donner l'envie d'aller plus loin dans la découverte des autres fonctions. Pour quelqu'un dont la vocation métier n'est pas le développement, c'est l'occasion d'acquérir un bagage «culturel» minimum sur les contraintes liées à l'utilisation des deux protocoles de couche réseau.

9. Documents de référence

Voici une liste de liens vers les différents supports qui ont été utilisés pour rédiger ce document. Parmi ces liens, ont peut distinguer ceux qui ont servi à argumenter sur les choix entre les deux grandes solutions de codage dual stack et ceux dont le code a été repris (ou «odieusement copié») pour fournir les exemples de programmes.

Beej's Guide to Network Programming
: Le guide (ou le livre) Beej's Guide to Network Programming[10] est un support très complet sur les sockets. Il propose de nombreux exemples de programmes et des indications essentielles sur l'utilisation des bibliothèques standard. Le programme showip de la Section 3, « Utilisation de getaddrinfo() » est directement extrait du guide.

A Brief Socket Tutorial
: La page d'archive brief socket tutorial[11] a été le premier support utilisé pour bâtir les documents du site inetdoc sur le thème du développement d'applications réseau. C'est de là que vient l'idée du programme de chat minimaliste avec échange de chaînes de caractères.

Lier une ou deux prises ?
: Le billet Lier une prise à IPv6 seulement ou bien aux deux familles, v4 et v6 ?[12] développe l'argumentation en faveur de la «solution académique» respectant la modélisation. Cette solution est présentée en la section intitulée « Codage basé sur un socket simple ».

Livre IPv6 Théorie et Pratique
: La section L'option IPV6_V6ONLY[13] résume très bien l'implication de l'utilisation de l'option `bindv6only` à l'échelle système ou lors de l'ouverture d'une prise (socket) ainsi que les codes d'erreur associés.

[10] http://beej.us/guide/bgnet/
[11] http://web.archive.org/web/20080703122104/http://sage.mc.yu.edu/kbeen/teaching/networking/resources/sockets.html
[12] http://www.bortzmeyer.org/bindv6only.html
[13] http://livre.g6.asso.fr/index.php/Programmation_d'applications_bis#L.27option_IPV6_V6ONLY

What are my IP addresses?
: Sur la page Sample Code for Dual Stack Applications Using Non-Blocking Sockets[14], la section intitulée Dual Stack Server Issues argumente en faveur de l'utilisation de deux prises ou sockets distinctes.

Modélisations réseau
: Le support de cours Modélisations réseau[15] présente les grandes caractéristiques des deux modélisations réseau historiques OSI et Internet. Il introduit aussi le modèle «contemporain» qui fait la synthèse. Ce modèle contemporain en 5 couches est ensuite utilisé dans tous les autres documents du site.

Adressage IPv4
: Le support Adressage IPv4[16] présente les différentes évolution du plan d'adressage du protocole IPv4.

Configuration d'une interface réseau
: Le support Configuration d'une interface de réseau local[17] présente les outils de configuration des interfaces réseau. Il permet notamment de relever les adresses IP et MAC des hôtes en communication.

[14] http://www.v6address.com/node/16
[15] http://www.inetdoc.net/articles/modelisation/
[16] http://www.inetdoc.net/articles/adressage.ipv4/
[17] http://www.inetdoc.net/travaux_pratiques/config.interface.lan/

A. Annexes

A.1. Instructions de compilation

Exemple de Makefile permettant de compiler n'importe lequel des programmes étudiés dans ce document.

```
target = $(addsuffix .o,$(basename $(wildcard *.c)))

all: $(target)

%.o: %.c
 gcc -Wall -Wextra -o $@ $<

clean:
 rm -f *.o
```

A.2. Zone DNS fake.domain

La mise en œuvre du service de noms de domaine est décrite dans le support de travaux pratiques Introduction au service DNS[1]. On donne ci-dessous les fichiers de configuration utilisés avec l'infrastructure de test. Dans ces fichiers, les noms d'hôtes sont associés à des adresses IP des deux versions.

- Options globales du service : `/etc/bind/named.conf.options`

```
acl "xfer" {
  localhost;
  ::1;
};  // Allow no other transfers.

acl "internal" {
  192.0.2.0/27;
  2001:db8:feb2::/48;
  localhost;
  ::1;
}; // Local networks

include "/etc/bind/rndc.key";

controls {
        inet 127.0.0.1 allow { any; } keys { "rndc-key"; };
};

options {
        directory "/var/cache/bind";
        statistics-file "/var/log/named/named.stats";
        dump-file "/var/log/named/named.dump";
        zone-statistics yes;

        // If there is a firewall between you and nameservers you want
        // to talk to, you may need to fix the firewall to allow multiple
        // ports to talk.  See http://www.kb.cert.org/vuls/id/800113

        // Generate more efficient zone transfers.  This will place
        // multiple DNS records in a DNS message, instead of one per
        // DNS message.
        transfer-format many-answers;

        // Set the maximum zone transfer time to something more
        // reasonable.  In this case, we state that any zone transfer
        // that takes longer than 60 minutes is unlikely to ever
        // complete.  WARNING:  If you have very large zone files,
        // adjust this to fit your requirements.
        max-transfer-time-in 60;
```

[1] http://www.inetdoc.net/travaux_pratiques/index.html#sysadm-net.dns

```
        // We have no dynamic interfaces, so BIND shouldn't need to
        // poll for interface state {UP|DOWN}.
        interface-interval 0;

        // If your ISP provided one or more IP addresses for stable
        // nameservers, you probably want to use them as forwarders.
        // Uncomment the following block, and insert the addresses replacing
        // the all-0's placeholder.

        auth-nxdomain no;    # conform to RFC1035
        listen-on-v6 { any; };

        allow-query { internal; };
        allow-recursion { internal; };
        allow-transfer { xfer; };

        dnssec-validation auto;
        dnssec-lookaside auto;
};

logging {
        channel "default_syslog" {
                // Send most of the named messages to syslog.
                syslog local2;
                severity error;
                };

        channel audit_log {
                // Send the security related messages to a separate file.
                file "/var/log/named/named.log";
                severity debug;
                print-time yes;
                };

        channel query_logging {
                file "/var/log/named/query.log";
                print-category yes;
                print-severity yes;
                print-time yes;
                };

        category default { default_syslog; };
        category general { default_syslog; };
        category security { audit_log; default_syslog; };
        category config { default_syslog; };
        category resolver { audit_log; };
        category xfer-in { audit_log; };
        category xfer-out { audit_log; };
        category notify { audit_log; };
        category client { audit_log; };
        category network { audit_log; };
        category update { audit_log; };
        category queries { query_logging; };
        category lame-servers { audit_log; };
};
```

- Déclaration des zones sur lesquelles le service a autorité : `/etc/bind/named.conf.local`

```
//
// Do any local configuration here
//

// Consider adding the 1918 zones here, if they are not used in your
// organization
//include "/etc/bind/zones.rfc1918";

view standard in {
        // Our internal (trusted) view. We permit the internal networks
        // to freely access this view. We perform recursion for our
        // internal hosts, and retrieve data from the cache for them.

        match-clients { internal; };
        recursion yes;
```

```
        additional-from-auth yes;
        additional-from-cache yes;
        zone-statistics yes;

        include "/etc/bind/named.conf.default-zones";

        // add entries for other zones below here
////////////////////////////////////////////////////
        zone "fake.domain" {
                type master;
                file "fake.domain";
        };

 // 192.0.2.0/27
        zone "2.0.192.in-addr.arpa" {
                type master;
                file "2.0.192";
        };

        // 2001:db8:feb2:10::/64
 // ipv6calc --in ipv6addr --out revnibbles.arpa 2001:db8:feb2:10::/64
        zone "0.1.0.0.2.b.e.f.8.b.d.0.1.0.0.2.ip6.arpa" {
                type master;
                file "10-feb2-db8-2001";
        };
////////////////////////////////////////////////////
};
```

- Déclaration des enregistrements de la zone directe `fake.domain` : `/var/cache/bind/fake.domain`

```
$TTL 1D
@               IN              SOA     cooper.fake.domain. root.cooper.fake.domain. (
                                2013021001      ; Serial
                                8H              ; Refresh 8 hours
                                2H              ; Retry   2 hours
                                1W              ; Expire  1 week
                                1D )            ; Minimum 1 day
                                NS              cooper.fake.domain.
                                MX      0       cooper.fake.domain.

cooper          A               192.0.2.1
ns              CNAME           cooper.fake.domain.
cooper  AAAA  2001:db8:feb2:10::1
rtr  AAAA  2001:db8:feb2:10::1
;
vm1  A  192.0.2.11
clnt4  CNAME  vm1.fake.domain.
vm1  AAAA  2001:db8:feb2:10::11
clnt6  AAAA  2001:db8:feb2:10::11
;
vm2  A  192.0.2.12
srvr4  CNAME  vm2.fake.domain.
vm2  AAAA  2001:db8:feb2:10::12
srvr6  AAAA  2001:db8:feb2:10::12
;
vm3  A  192.0.2.13
```

- Déclaration des enregistrements de la zone inverse correspondant au réseau `192.0.2.0/27` : `/var/cache/bind/2.0.192`

```
$TTL 1D
@               IN      SOA     cooper.fake.domain. root.cooper.fake.domain. (
                        2013021001      ; Serial
                        8H              ; Refresh 8 hours
                        2H              ; Retry   2 hours
                        1W              ; Expire  1 week
                        1D )            ; Minimum 1 day
                        NS              cooper.fake.domain.

1               PTR     cooper.fake.domain.
;
11              PTR     vm1.fake.domain.
12              PTR     vm2.fake.domain.
```

```
13              PTR     vm3.fake.domain.
```

- Déclaration des enregistrements de la zone inverse correspondant au réseau `2001:db8:feb2:10/64` : `/var/cache/bind/10-feb2-db8-2001`

```
$TTL 1D
@               IN      SOA     cooper.fake.domain. root.cooper.fake.domain. (
                        2013020504      ; Serial
                        8H              ; Refresh 8 hours
                        2H              ; Retry   2 hours
                        1W              ; Expire  1 week
                        1D )            ; Minimum 1 day
                        NS              cooper.fake.domain.
;
; ipv6calc --in ipv6addr --out revnibbles.arpa 2001:db8:feb2:10::/64
$ORIGIN 0.1.0.0.2.b.e.f.8.b.d.0.1.0.0.2.ip6.arpa.
1.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0         PTR cooper.fake.domain.
1.1.0.0.0.0.0.0.0.0.0.0.0.0.0.0         PTR vm1.fake.domain.
2.1.0.0.0.0.0.0.0.0.0.0.0.0.0.0         PTR vm2.fake.domain.
```