TD Initiation aux Sockets V4/V6

V0.5

Nature : partie 1 TD, Partie 2 TP (donc compte rendu)
Durée : 6 heures + 6
Langage : C
Outil : Linux, gcc
Pré-requis : Cours Système UNIX
Acquis de ce TP : Programmation réseau par socket TCP

Critères d'évaluation :

respect du Cahier des Charges et respect du plan du TD,
qualité et pertinence du compte-rendu (analyse, commentaires, problèmes rencontrés,...).

Objectif et déroulement

L'objectif de ce TD est d'apprendre la programmation réseaux par des sockets. Vous allez tout d'abord être confronté à une introduction aux sockets. Puis vous développerez votre premier client réseau. Pour cette tâche vous serez guidé pas à pas. Dans un troisième temps, alors que vous serez plus aguéri à la programmation réseaux, vous développerez un serveur compatible avec votre client.

Travail à faire

Le travail à faire est encadré.

Sommaire

1 - Avant Propos : Généralités

La notion de sockets a été introduite dans les distributions de Berkeley (un fameux système de type UNIX, dont beaucoup de distributions actuelles utilisent des morceaux de code), c'est la raison pour laquelle on parle parfois de sockets BSD (*Berkeley Software Distribution*).

Il s'agit d'un modèle permettant la communication inter processus (*IPC - Inter Process Communication*) afin de permettre à divers processus de communiquer aussi bien sur une même machine qu'à travers un réseau TCP/IP.

La communication par socket est souvent comparée aux communications humaines. On distingue ainsi deux modes de communication:

- Le mode connecté (comparable à une communication téléphonique), utilisant le protocole TCP. Dans ce mode de communication, une connexion durable est établie entre les deux processus, de telle façon que l'adresse de destination n'est pas nécessaire à chaque envoi de données.
- Le mode non connecté (analogue à une communication par courrier), utilisant le protocole UDP. Ce mode nécessite l'adresse de destination à chaque envoi, et aucun accusé de réception n'est donné.

Les sockets sont généralement implémentés en langage C, et utilisent des fonctions et des structures disponibles dans la librairie <sys/socket.h>.

Position des sockets dans le modèle OSI
Les sockets se situent juste au-dessus de la couche transport du modèle OSI (protocoles UDP ou TCP), elle-même utilisant les services de la couche réseau (protocole IP / ARP).

Modèle des sockets	*Modèle OSI*
Application utilisant les sockets	Application
Application utilisant les sockets	Présentation
Application utilisant les sockets	Session
UDP/TCP	Transport
IP/ARP	Réseau
Ethernet, X25, ...	Liaison
Ethernet, X25, ...	Physique

2 - Déroulement d'une communication

Comme dans le cas de l'ouverture d'un fichier, la communication par socket utilise un descripteur pour désigner la connexion sur laquelle on envoie ou reçoit les données. Ainsi la première opération à effectuer consiste à appeler une fonction créant un socket et retournant un descripteur (un entier) identifiant de manière unique la connexion. Ainsi ce descripteur est passé en paramètres des fonctions permettant d'envoyer ou recevoir des informations à travers la socket.

L'ouverture d'une socket se fait en deux étapes:

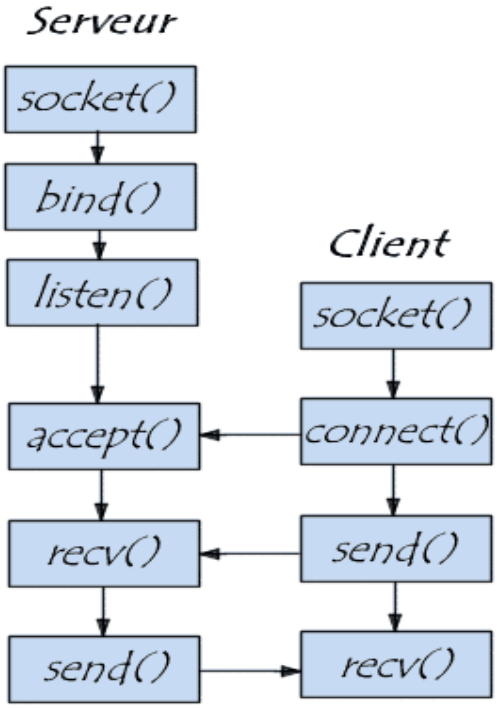
- La création d'une socket et de son descripteur par la fonction socket()
- La fonction bind() permet de spécifier le type de communication associé à la socket (protocole TCP ou UDP)

Un serveur doit être à l'écoute de messages éventuels. Toutefois, l'écoute se fait différemment selon que la socket soit en mode connectée (TCP) ou non (UDP).

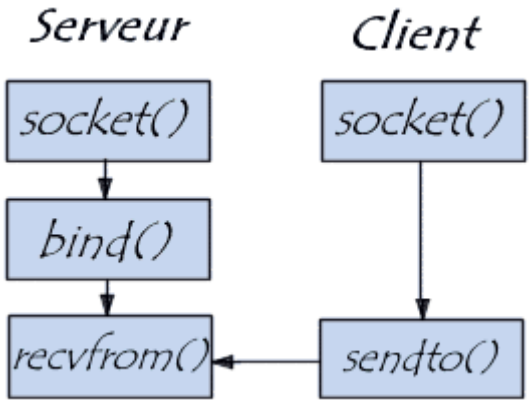
- En mode connecté, le message est reçu d'un seul bloc.
  Ainsi en mode connecté, la fonction listen() permet de placer la socket en mode passif (à l'écoute des messages). En cas de message entrant, la connexion peut être acceptée grâce à la fonction accept(). Lorsque la connexion a été acceptée, le serveur reçoit les données grâce à la fonction recv().
- En mode non connecté, comme dans le cas du courrier, le destinataire reçoit le message petit à petit (la taille du message est indéterminée) et de façon désordonnée.
  Le serveur reçoit les données grâce à la fonction recvfrom().

La fin de la connexion se fait grâce à la fonction close().

Voici le schéma d'une communication en mode connecté:



Voici le schéma d'une communication en mode non connecté:



3 - Création d'un client en mode connecté

Nous allons développer un client pour le serveur ECHO. Si vous ne le connaissez pas n'hésitez pas à l'essayer ***telnet "serveur" 7.*** Il repette tout ce qu'on lui transmet.

3.1 - Tout d'abord vous avez besoin de votre éditeur de texte préféré.

A faire : Créer le fichier clientTCP.c contenant la le code qui suit. Testez le.

```
/*************************************************************
 *  * nom     : clientTCP.c
 *  * Auteur  : AF 16/09/2005
 *  * version : 0.1
 *  * descr   : client TCP echo
 *  * licence : GPL
*************************************************************/
#include <stdio.h>
#include <sys/types.h>
#include <sys/socket.h>


int main(int argc,char **argv, char **env){
     if (argc < 3){
          fprintf(stderr,"usage : clientTCP server ASCIIString\n");
          return(1);
     }
     return (0);

}
```

3.2 - struct sockaddr_in

Tout d'abord nous devons créer la structure qui va contenir le serveur auquel nous allons nous connecter.

La structure utilisée avec TCP/IP est une adresse **AF_INET** (Généralement les structures d'adresses sont redéfinies pour chaque famille d'adresse). Les adresses **AF_INET** utilisent une structure ***sockaddr_in*** définie dans ***<netinet/in.h>*** :

```
struct sockaddr_in {
/* famille de protocole (AF_INET) */
short sin_family;

/* numéro de port */
u_short sin_port;

/* adresse internet */
struct in_addr sin_addr;

char sin_zero[8];     /* initialise à zéro */
}
```

- ***sin_family*** représente le type de famille
- ***sin_port*** représente le port à contacter
- ***sin_addr*** représente l'adresse de l'hôte
- ***sin_zero[8]*** contient uniquement des zéros (étant donné que l'adresse IP et le port occupent 6 octets, les 8 octets restants doivent être à zéro)

A faire : Initialiser dans le fichier clientTCP.c la structure sockaddr_in. Le serveur est **serviris2.iris.** et le port est le **7777** (service de type echo).
Rq : attention il n'est pas nécessaire de recopier la structure « sockaddr_in » il suffit de trouver le bon fichier d'entête.

Pour comprendre : Profitez en pour afficher l'adresse rendu par ***gethostbyname***.

```
toinfo = gethostbyname( « serveur obtenu par argv » );

/* Protocole internet */
to.sin_family = AF_INET;

/* copie l'adresse dans la structure sockaddr_in. */
/* memcpy(dest, src, length) */
memcpy(&to.sin_addr.s_addr,toinfo->h_addr_list[0],toinfo->h_length);

/* port d'écoute par défaut au-dessus des ports réservés */
to.sin_port = htons(port);
/* attention addr->sin_port = port n'est pas valide */
```

A propos de gethostbyname

```
#include <netdb.h>
struct hostent *gethostbyname(const char *serveur);
```

La fonction gethostbyname() renvoie une structure de type hostent pour l'hôte serveur. La chaîne ***serveur*** est soit un nom d'hôte, soit une adresse IPv4 en notation pointée standard, soit une adresse IPv6 avec la notation points-virgules et points (Cf RFC 1884 pour la description des adresses IPv6). Si ***serveur*** est une adresse IPv4 ou IPv6, aucune recherche supplémentaire n'a lieu et gethostbyname() copie simplement la chaine ***serveur*** dans le champ h_name et le champs équivalent struct in_addr dans le champs h_addr_list[0] de la structure hostent renvoyée.

3.3 - Création de la socket et connexion

Notre structure est initialisé, nous devons maintenant créer notre socket. Notre socket sera le « tube » permettant de dialoguer avec le serveur.

A Propos de la fonction socket()

La création d'une socket se fait grâce à la fonction socket():

int socket(famille,type,protocole)

- famille représente la famille de protocole utilisé (AF_INET pour TCP/IP utilisant une adresse Internet sur 4 octets: l'adresse IP ainsi qu'un numéro de port afin de pouvoir avoir plusieurs sockets sur une même machine, AF_UNIX pour les communications UNIX en local sur une même machine)

- type indique le type de service (orienté connexion ou non). Dans le cas d'un service orienté connexion (c'est généralement le cas), l'argument type doit prendre la valeur SOCK_STREAM (communication par flot de données). Dans le cas contraire (protocole UDP) le paramètre type doit alors valoir SOCK_DGRAM (utilisation de datagrammes, blocs de données)
- protocole permet de spécifier un protocole permettant de fournir le service désiré. Dans le cas de la suite TCP/IP il n'est pas utile, on le mettra ainsi toujours à 0

La fonction socket() renvoie un entier qui correspond à un descripteur de la socket nouvellement créé et qui sera passé en paramètre aux fonctions suivantes. En cas d'erreur, la fonction socket() retourne -1.

A faire : Créer la socket

Voici un exemple d'utilisation de la fonction socket() :

```
descripteur = socket(AF_INET,SOCK_STREAM,0);
```

Notre socket est maintenant créer il nous faut la connecter au serveur grâce à la primitive « connect ».

A propos de la fonction connect()

La fonction connect() permet d'établir une connexion avec un serveur :

int connect(int descripteur,struct sockaddr * addr,int * addrlen)

- descripteur représente la socket précédemment ouvert (la socket à utiliser)
- addr représente l'adresse de l'hôte à contacter.
- addrlen représente la taille de l'adresse de l'hôte à contacter

La fonction connect() retourne 0 si la connexion s'est bien déroulée, sinon -1.

A faire : Connectez vous au serveur.

Voici un exemple d'utilisation de la fonction connect(), qui connecte la socket du client sur le port port de l'hôte :

```
if (connect(descripteur,(struct sockaddr*)&to,sizeof(to)) == -1) {

// Traitement de l'erreur;

}
```

3.4 - Envoie et réception de données

Une fois notre canal de communication ouvert nous pouvons dialoguer avec notre serveur. Dans le cadre de notre étude, le protocole ECHO est rudimentaire. Le serveur ECHO nous renvoie la chaîne de caractères qu'on lui envoie.

A Propos de l'envoie de données

La fonction send()

La fonction send() permet d'écrire dans une socket (envoyer des données) en mode connecté (TCP) :

int send(int descripteur,char * buffer,int len,int flags)

- descripteur représente la socket précédemment ouvert
- buffer représente un tampon contenant les octets à envoyer au client
- len indique le nombre d'octets à envoyer
- flags correspond au type d'envoi à adopter :
  o le flag MSG_DONTROUTE indiquera que les données ne routeront pas
  o le flag MSG_OOB indiquera que les données urgentes (Out Of Band) doivent être envoyées
  o le flag 0 indique un envoi normal

La fonction send() renvoie le nombre d'octets effectivement envoyés.

A faire : Envoyer la chaîne de caractère passer en argument.

Voici un exemple d'utilisation de la fonction send() :

```
retour = send(descripteur,buffer,sizeof(buffer),0 );

if (retour == -1) {

// traitement de l'erreur

}
```

A Propos de la réception de données

La fonction recv()

La fonction recv() permet de lire dans un socket en mode connecté (TCP) :

int recv(int descripteur,char * buffer,int len,int flags)

- descripteur représente la socket précédemment ouvert
- buffer représente un tampon qui recevra les octets en provenance du client
- len indique le nombre d'octets à lire
- flags correspond au type de lecture à adopter :
  o le flag MSG_PEEK indiquera que les données lues ne sont pas retirées de la queue de réception
  o le flag MSG_OOB indiquera que les données urgentes (Out Of Band) doivent être lues
  o le flag 0 indique une lecture normale

La fonction recv() renvoie le nombre d'octets lus. De plus cette fonction bloque le processus jusqu'à ce qu'elle reçoive des données.

A faire : recevoir dans un buffer de taille fixe ce que nous a envoyé le serveur (normalement la même chose que envoyé)

Voici un exemple d'utilisation de la fonction recv() :

```
retour = recv(descripteur,buffer,sizeof(buffer),0 );
```

```
if (retour == -1) {

// traitement de l'erreur

}
```

3.5 - Déconnexion de notre socket

Une fois la communication terminée il nous reste à fermer le tube (la socket).

A Propos de déconnexions

Les fonctions close() et shutdown()

La fonction close() permet la fermeture d'une socket en permettant au système d'envoyer les données restantes (pour TCP) :

int close(int descripteur)

La fonction shutdown() permet la fermeture d'une socket dans un des deux sens (pour une connexion full-duplex) :

int shutdown(int descripteur,int how)

- Si how est égal à 0, le descripteur est fermé en réception
- Si how est égal à 1, le descripteur est fermé en émission
- Si how est égal à 2, le descripteur est fermé dans les deux sens

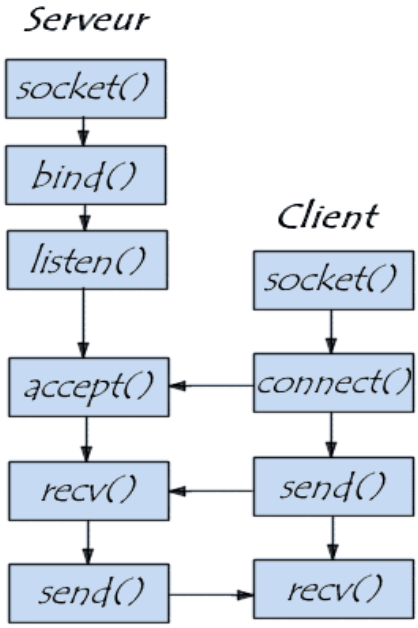
close() comme shutdown() retournent -1 en cas d'erreur, 0 si la fermeture se déroule bien.

A faire : fermer la socket.

Bravo , normalement si tout c'est bien passé
votre client permet de dialoguer avec le serveur!

4 - Création d'un serveur en mode connecté

Maintenant, nous souhaitons créer un serveur, quelles différences y a il avec le client...



- Pas de différence pour la création de la socket
- bind : Nous devons lier la socket à un point de communication
- listen : permet de paramétrer la file d'attente des clients
- La fonction *accept()* permet la connexion en acceptant un appel

A Propos de la fonction bind()

Après création de la socket, il s'agit de la lier à un point de communication défini par une adresse et un port, c'est le rôle de la fonction bind():

bind(int descripteur,sockaddr localaddr,int addrlen)

- ➔ ***descripteur*** représente le descripteur du socket nouvellement créé
- ➔ ***localaddr*** est une structure qui spécifie l'adresse locale à travers laquelle le programme doit communiquer
  Le format de l'adresse est fortement dépendant du protocole utilisé :
  - l'interface socket définit une structure standard (sockaddr définie dans <sys/socket.h>) permettant de représenter une adresse:

```
struct sockaddr {
/* longueur effective de l'adresse */
u_char sa_len;

/* famille de protocole (généralement AF_INET) */
u_char sa_family;

/* l'adresse complète */
char sa_data[14];
}
```

- sa_len est un octet (u_char) permettant de définir la longueur utile de l'adresse (la partie réellement utilisée de sa_data)
- sa_family représente la famille de protocole (AF_INET pour TCP/IP)
- sa_data est une chaîne de 14 caractères (au maximum) contenant l'adresse

- La structure utilisée avec TCP/IP est une adresse AF_INET (Généralement les structures d'adresses sont redéfinies pour chaque famille d'adresse).Les adresses AF_INET utilisent une structure sockaddr_in définie dans <netinet/in.h> :

```
struct sockaddr_in {
/* famille de protocole (AF_INET) */
short sin_family;

/* numéro de port */
u_short sin_port;

/* adresse internet */
struct in_addr sin_addr;

char sin_zero[8];     /* initialise à zéro */

}
```

- sin_family représente le type de famille
- sin_port représente le port à contacter
- sin_addr représente l'adresse de l'hôte
- sin_zero[8] contient uniquement des zéros (étant donné que l'adresse IP et le port occupent 6 octets, les 8 octets restants doivent être à zéro)

➔ ***addrlen*** indique la taille du champ localaddr. On utilise généralement sizeof(localaddr).

Voici un exemple d'utilisation de la fonction bind() :

```
sockaddr_in localaddr ;

localaddr.sin_family = AF_INET; /* Protocole internet */

/* Toutes les adresses IP de la station */
localaddr.sin_addr.s_addr = htonl(INADDR_ANY);

/* port d'écoute par défaut au-dessus des ports réservés */
localaddr.sin_port = htons(port);

if ( bind(listen_socket,
     (struct sockaddr*)&localaddr,
     sizeof(localaddr) )  == SOCKET_ERROR) {

          // Traitement de l'erreur;
}
```

Le numéro fictif INADDR_ANY signifie que le socket peut-être associé à n'importe quelle adresse IP de la machine locale (s'il en existe plusieurs). Pour spécifier une adresse IP spécifique à utiliser, il est possible d'utiliser la fonction inet_addr():

```
inet_addr("127.0.0.1");
/* utilisation de l'adresse de boucle locale */
```

Le socket peut être relié à un port libre quelconque en utilisant le numéro 0.

A Propos de la fonction listen()

La fonction listen() permet de configurer une socket en attente de connexion.

La fonction listen() ne s'utilise qu'en mode connecté (donc avec le protocole TCP)

int listen(int socket,int backlog)

- socket représente le socket précédemment ouvert
- backlog représente le nombre maximal de connexions pouvant être mises en attente

La fonction listen() retourne la valeur SOCKET_ERROR en cas de problème, sinon elle retourne 0.

Voici un exemple d'utilisation de la fonction listen() :

```
if (listen(socket,10) == SOCKET_ERROR) {
     // traitement de l'erreur
}
```

A Propos de la fonction accept()

La fonction accept() permet la connexion en acceptant un appel :

int accept(int socket,struct sockaddr * addr,int * addrlen)

- socket représente le socket précédemment ouvert (le socket local)
- addr représente un tampon destiné à stocker l'adresse de l'appelant
- addrlen représente la taille de l'adresse de l'appelant

La fonction accept() retourne un identificateur du socket de réponse. Si une erreur intervient la fonction accept() retourne la valeur INVALID_SOCKET.

Voici un exemple d'utilisation de la fonction accept() :

```
Sockaddr_in Appelant;

/* structure destinée à recueillir les renseignements sur l'appelant

Appelantlen = sizeof(from);

accept(socket_local,(struct sockaddr*)&Appelant, &Appelantlen);
```

A faire: Maintenant que vous savez tout, a vous de jouer.
Écrivez un serveur ECHO compatible avec votre client.

Pour vous aider, à chaque étape proposée implémentez ce qui est demandé et testez le pour en comprendre le mécanisme :

1) Créer un fichier **serveur.c** et reprenez les éléments du client qui vous semble utiles.
2) Insérez les fonctions spécifiques au mode serveur **bind**, **listen.**
3) Appelez **accept** et utiliser le résultat pour communiquer avec le client connecté.
4) Recevez ce que vous envoie le client et affichez le.
5) Envoyez lui la même chose et fermer la connexion.
6) Affichez l'adresse du client qui s'est connecté à votre serveur.
7) bouclez pour vous mettre en attente d'un autre client.

Pour aller plus loin...

8) Si vous en avez le temps, faites exécuter le traitement du client (après le **accept**) par un processus fils (**fork**).

5 - IPV6 quelles différences

http://livre.g6.asso.fr

5.1 - Ce qui a changé

Les changements opérés de façon à intégrer IPv6 concernent les quatre domaines suivants :

- les structures de données d'adresses ;
- l'interface socket ;
- les primitives de conversion entre noms et adresses ;
- les fonctions de conversions d'adresses.

Ces changements ont été minimisés autant que possible de manière à faciliter le portage des applications IPv4 existantes. En outre, et ce point est important, cette nouvelle API doit permettre l'interopérabilité entre machines IPv4 et machines IPv6 grâce au mécanisme de double pile décrit ci-après.

L'API décrite ici est celle utilisée en Solaris, Linux et systèmes *BSD. Elle correspond à celle définie dans le RFC 3493 avec quelques modifications nécessaires pour prendre en compte les dernières évolutions des protocoles sous-jacents. Cette API est explicitement conçue pour fonctionner sur des machines possédant la double pile IPv4 et IPv6 (cf. See Double pile IPv4/IPv6 pour le schéma d'implémentation d'une telle double pile sous UNIX 4.4BSD). Cette API "socket" est celle disponible dans de nombreux environnements de programmation tels que Java, perl, python, ruby, ...

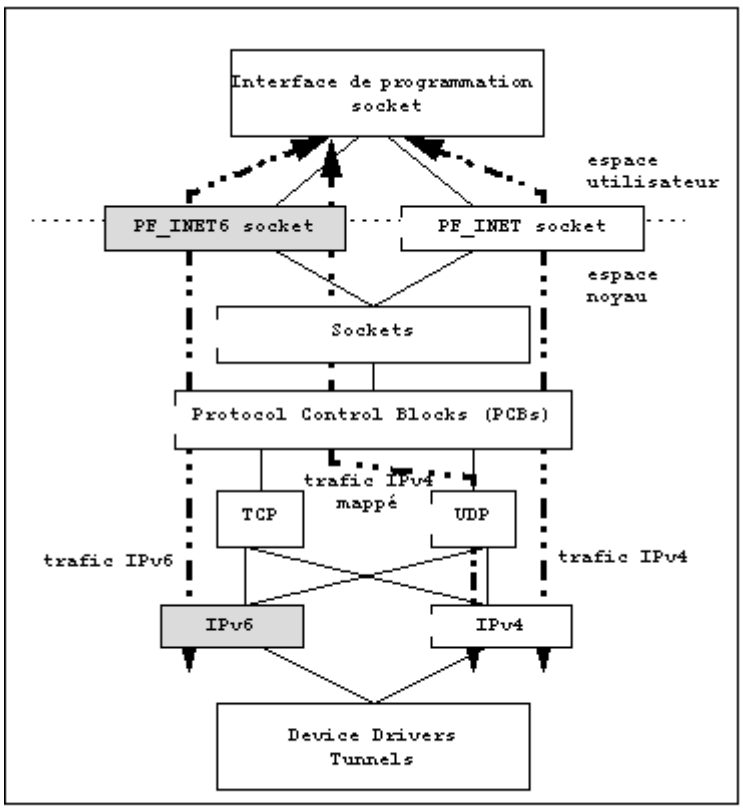


Figure 15-1. Double pile IPv4/IPv6

Une API "avancée", décrite dans le RFC 3542 permet de programmer les échanges réseaux de manière très précise.

5.2 - Les structures de données d'adresses

Une nouvelle famille d'adresses ayant pour nom AF_INET6 et dont la valeur peut varier d'une implémentation à l'autre, a été définie (dans sys/socket.h). Également, une nouvelle famille de protocoles ayant pour nom PF_INET6 a été définie (dans sys/socket.h). En principe, on doit avoir :

```
#define PF_INET6 AF_INET6
```

La structure de données destinée à contenir une adresse IPv6 est définie comme suit (dans netinet/in.h) :

```
struct in6_addr {
   uint8_t s6_addr[16];
};
```

les octets constituant l'adresse étant rangés comme d'habitude dans l'ordre réseau (network byte order).

La structure de données IPv6 ***struct sockaddr_in6***, est équivalente à la structure ***struct sockaddr_in*** d'IPv4. Elle est définie comme suit (dans netinet/in.h) pour les systèmes dérivés d'UNIX 4.3BSD :

```
struct sockaddr_in6 {
   sa_family_t sin6_family;   /* AF_INET6 */
   in_port_t sin6_port;       /* numéro de port */
   uint32_t sin6_flowinfo;    /* identificateur de flux */
   struct in6_addr sin6_addr; /* adresse IPv6 */
   uint32_t sin6_scope_id;    /* ensemble d'interfaces correspondant
                               * à la portée de l'adresse */
};
```

Il faut noter que cette structure a une longueur de 28 octets, et est donc plus grande que le type générique struct sockaddr. Il n'est donc plus possible de réserver une struct sockaddr si la valeur à stocker peut être une struct sockaddr_in6. Afin de faciliter la tâche des implémenteurs, une nouvelle structure de données, struct sockaddr_storage, a été définie. Celle-ci est de taille suffisante afin de pouvoir prendre en compte tous les protocoles supportés et alignée de telle sorte que les conversions de type entre pointeurs vers les structures de données d'adresse des protocoles supportés et pointeurs vers elle-même n'engendrent pas de problèmes d'alignement. Un exemple d'utilisation pourrait être le suivant :

```
struct sockaddr_storage ss;

struct sockaddr_in *sin = (struct sockaddr_in *) &ss;
struct sockaddr_in6 *sin6 = (struct sockaddr_in6 *) &ss;
```

Rq : *Si le champ sin6_len existe (ce qui est testable par le fait que le symbole SIN6_LEN est défini), il doit être initialisé par la taille de la structure sockaddr_in6.*

On notera la présence de deux nouveaux champs (ils n'ont pas d'équivalents dans la structure sockaddr_in) dans la structure de données sockaddr_in6, les champs **sin6_flowinfo** et **sin6_scope_id**. Le premier, en réalité structuré, est décrit dans le RFC 2460 et Identificateur de flux. Le second désigne un ensemble d'interfaces en adéquation avec la portée de l'adresse contenue dans le champ **sin6_addr**. Par exemple, si l'adresse en question est de type lien local, le champ **sin6_scope_id** devrait être un index d'interface.

6 - L'interface socket

La création d'une socket se fait comme auparavant en appelant la primitive socket. La distinction entre les protocoles IPv4 et IPv6 se fait sur la valeur du premier argument passé à socket, à savoir la famille d'adresses (ou de protocoles), c'est-à-dire ici PF_INET ou PF_INET6. Par exemple, si on veut créer un socket IPv4/UDP, on écrira :

```
sock = socket(PF_INET, SOCK_DGRAM, 0);
```

tandis qu'une création de socket IPv6/UDP se fera ainsi :

```
sock = socket(PF_INET6, SOCK_DGRAM, 0);
```

Une erreur de programmation classique consiste à utiliser AF_INET à la place de PF_INET. Cela n'a pas d'effet en général car rares sont les systèmes pour lesquels ces deux constantes diffèrent. Pour éviter en IPv6 des problèmes liés à cette erreur, il est demandé que les deux constantes PF_INET6 et AF_INET6 soient identiques.

Quant aux autres primitives constituant l'interface **socket**, leur syntaxe reste inchangée. Il faut simplement leur fournir des adresses IPv6, en l'occurrence des pointeurs vers des structures de type **struct sockaddr_in6** au préalable convertis en des pointeurs vers des structures génériques de type **struct sockaddr**.

Donnons pour mémoire une liste des primitives les plus importantes :

```
bind()     connect()      sendmsg()
sendto()   accept()       recvfrom()
recvmsg()  getsockname()  getpeername()
```

6.1 - L'adresse "wildcard"

Lors du nommage d'une socket via la primitive bind, il arrive fréquemment qu'une application (par exemple un serveur TCP) laisse au système la détermination de l'adresse source pour elle. En IPv4, pour ce faire, elle passe à bind une structure sockaddr_in avec le champ sin_addr.s_addr ayant pour valeur la constante INADDR_ANY, constante définie dans le fichier netinet/in.h.

En IPv6, il y a deux manières de faire cela, à cause des règles du langage C sur les initialisations et affectations de structures. La première est d'initialiser une structure de type **struct in6_addr** par la constante **IN6ADDR_ANY_INIT** :

```
struct in6_addr any_addr = IN6ADDR_ANY_INIT;
```

Attention, ceci ne peut se faire qu'au moment de la déclaration. Par exemple le code qui suit est incorrect (en C il est interdit d'affecter une constante complexe à une structure) :

```
struct sockaddr_in6 sin6;

sin6.sin6_addr = IN6ADDR_ANY_INIT; /* erreur de syntaxe !! */
```

La seconde manière utilise une variable globale :

```
extern const struct in6_addr in6addr_any;
struct sockaddr_in6 sin6;
```

```
sin6.sin6_addr = in6addr_any;
```

Cette méthode n'est pas possible dans une déclaration de variable globale ou statique.

La constante **IN6ADDR_ANY_INIT** et la variable **in6addr_any** sont toutes deux définies dans le fichier netinet/in.h.

6.2 - L'adresse de bouclage

En IPv4, c'est la constante **INADDR_LOOPBACK**. En IPv6, de manière tout à fait similaire à l'adresse "wildcard", il y a deux façons d'affecter cette adresse. Ceci peut se faire au moment de la déclaration avec la constante **IN6ADDR_LOOPBACK_INIT** :

```
struct in6_addr loopback_addr = IN6ADDR_LOOPBACK_INIT;
```

ou via la variable globale **in6addr_loopback** :

```
extern const struct in6_addr in6addr_loopback;
struct sockaddr_in6 sin6;

sin6.sin6_addr = in6addr_loopback;
```

Cette constante et cette variable sont définies dans le fichier netinet/in.h.

7 - Les primitives de conversion entre noms et adresses

Les primitives gethostbyname, gethostbyaddr, getservbyname et getservbyport ont été remplacées par les deux primitives indépendantes de la famille d'adresses et normalisées par la RFC 3493 getaddrinfo et getnameinfo :

```
#include <sys/socket.h>
#include <netdb.h>

int getaddrinfo(const char *nodename, const char *servname,
const struct addrinfo *hints, struct addrinfo **res);

void freeaddrinfo(struct addrinfo *res);

const char *gai_strerror(int errcode);
```

Le type struct addrinfo est défini comme suit :

```
struct addrinfo {
   int ai_flags;             /* AI_PASSIVE, AI_CANONNAME, ... */
   int ai_family;            /* PF_xxx */
   int ai_socktype;          /* SOCK_xxx */
    int ai_protocol;           /* 0 ou IPPROTO_xxx pour IPv4 et IPv6
*/
     size_t ai_addrlen;           /* la taille de l'adresse binaire
ai_addr */
   char *ai_canonname;       /* le nom complétement qualifié */
   struct sockaddr *ai_addr; /* l'adresse binaire */
    struct addrinfo *ai_next; /* la structure suivante dans la liste
chaînée */
};
```

getaddrinfo prend en entrée le nom d'une machine (nodename) et le nom d'un service (servname). S'il n'y a pas d'erreur, getaddrinfo rend 0 et res pointe sur une liste dynamiquement allouée de struct addrinfo. Chaque élément de cette liste contient la description et l'adresse d'une struct sockaddr initialisée pour fournir l'accès au service servname sur nodename. Les champs ai_family, ai_socktype et ai_protocol ont la valeur utilisable dans l'appel système socket.

Lorsque la liste de résultat n'est plus nécessaire, la mémoire allouée peut être libérée par la primitive freeaddrinfo. En cas d'erreur, getaddrinfo rend un code d'erreur non nul qui peut être imprimé par la fonction gai_strerror.

getaddrinfo peut donner des réponses de la famille d'adresses IPv4 ou IPv6, et des réponses pour les protocoles connectés ou non (ai_socktype peut valoir SOCK_DGRAM ou SOCK_STREAM). L'argument hints permet de choisir les réponses souhaitées. Un argument égal à NULL signifie que la liste des réponses doit contenir toutes les adresses et tous les protocoles. Sinon hints doit pointer sur une structure dont les champs ai_family, ai_socktype et ai_protocol définissent les types de résultat attendus. Une valeur de PF_UNSPEC du champ ai_family signifie que toutes les familles d'adresse (IPv4 et IPv6) sont admises, un 0 dans les champs ai_socktype (resp. ai_protocol) signifie que tous les types de socket (resp. protocole) sont admis. Le champ ai_flags permet de préciser des options supplémentaires.

L'argument servname peut être le nom d'un service ou un nombre décimal. De même, l'argument nodename peut être un nom (au format DNS habituel) ou une adresse sous forme numérique IPv4 ou IPv6 (si ai_flags contient le bit AI_NUMERICHOST, nodename doit être sous forme numérique et aucun appel au serveur de nom n'est fait). De plus l'un ou l'autre des arguments servname et nodename peut être un pointeur NULL, mais pas tous les deux. Si servname est NULL, le champ port des réponses ne sera pas initialisé (il restera égal à 0). Si nodename est NULL, l'adresse réseau dans les réponses est mis à "non initialisé" (INADDR_ANY en IPv4, IN6ADDR_ANY_INIT en IPv6) si ai_flags contient le bit AI_PASSIVE, et à l'adresse de "loopback" (INADDR_LOOPBACK ou IN6ADDR_LOOPBACK_INIT) sinon. Le cas AI_PASSIVE sert donc à obtenir des réponses utilisables par un programme serveur dans un bind pour recevoir des requêtes. Enfin si le bit AI_CANONNAME est positionné, le champ ai_canonname de la réponse contient le nom canonique de nodename.

La primitive getnameinfo remplace les primitives gethostbyaddr et getservbyport. Elle effectue la traduction d'une adresse vers un nom :

```
#include <sys/socket.h>
#include <netdb.h>

int getnameinfo(const struct sockaddr *sa, socklen_t salen,
char *host, size_t hostlen,
char *serv, size_t servlen, int flags);
```

En entrée l'argument sa pointe vers une structure d'adresse générique (de type sockaddr_in ou sockaddr_in6) et salen contient sa longueur. Le champ host (resp. serv) doit pointer sur une zone de longueur hostlen (resp. servlen) caractères. getnameinfo retourne la valeur 0 si tout est correct et un code d'erreur non nul si une erreur est détectée. S'il n'y a pas d'erreur, le champ host (resp. serv) reçoit en sortie le nom de la machine (resp. du service) correspondant. Les arguments host et serv peuvent être NULL si la réponse est inutile. Deux constantes sont définies pour permettre de réserver des zones de réponses de longueur raisonnable :

```
# define NI_MAXHOST 1025
# define NI_MAXSERV 32
```

Le champ flags permet de modifier la réponse : si flags contient le bit NI_NUMERICHOST (resp. NI_NUMERICSERV) la réponse sera l'adresse et non le nom de la machine (resp. le numéro et non le nom du service) ; si on ne sait pas trouver dans le serveur de nom le nom de la machine, getnameinfo rendra une erreur si le bit NI_NAMEREQD est positionné et l'adresse numérique sinon ; le bit NI_DGRAM indique si le service est sur UDP et non sur TCP.

7.1 - Les fonctions de conversion numériques d'adresses

Elles sont l'analogue des fonctions inet_addr et inet_ntoa d'IPv4, la seule véritable différence étant qu'elles ont un argument précisant la famille d'adresse et peuvent donc aussi bien convertir les adresses IPv4 que les adresses IPv6. Comme la plupart des programmes manipulent des struct sockaddr*, il est souvent préferable d'utiliser les fonctions getaddrinfo et getnameinfo, au besoin avec le flag AI_NUMERICHOST.

```
#include <sys/socket.h>
#include <arpa/inet.h>

int
```

```
inet_pton(af, src, dst)
int af;              /* AF_INET ou AF_INET6 */
const char *src;     /* l'adresse (chaine de caract.) à traiter */
void *dst;           /* le tampon où est rangé le résultat */

char *
inet_ntop(af, src, dst, size)
int af;              /* AF_INET ou AF_INET6 */
const void *src;     /* l'adresse binaire à traiter */
char *dst;           /* le tampon où est rangé le résultat */
size_t size;         /* la taille de ce tampon */
```

La primitive inet_pton convertit une adresse textuelle en sa forme binaire. Elle retourne 1 lorsque la conversion a été réussie, 0 si la chaine de caractères qui lui a été fournie n'est pas une adresse valide et -1 en cas d'erreur, c'est-à-dire lorsque la famille d'adresses (premier argument) n'est pas supportée. Actuellement, les deux seules familles d'adresses supportées sont AF_INET et AF_INET6.

La primitive duale inet_ntop convertit une adresse sous forme binaire en sa forme textuelle. Le troisième argument est un tampon destiné à recevoir le résultat de la conversion. Il doit être d'une taille suffisante, à savoir 16 octets pour les adresses IPv4 et 46 octets pour les adresses IPv6. Ces deux tailles sont définies dans le fichier netinet/in.h :

```
#define INET_ADDRSTRLEN 16
#define INET6_ADDRSTRLEN 46
```

Si la conversion est réussie, inet_ntop retourne un pointeur vers le tampon où est rangé le résultat de la conversion. Dans le cas contraire, inet_ntop retourne le pointeur nul, ce qui se produit soit lorsque la famille d'adresses n'est pas reconnue, soit lorsque la taille du tampon est insuffisante.

8 - La commande haah (host-address-address-host)

L'exemple proposé n'est autre qu'une sorte de nslookup (très) simplifié. Si par exemple on lui donne en argument une adresse numérique (IPv4 ou IPv6), il imprime le nom complètement qualifié correspondant lorsque la requête DNS aboutit. L'extrait de session qui suit illustre l'utilisation de cette commande.

```
$ haah bernays
Canonical name:
bernays.ipv6.logique.jussieu.fr
Adresses:
2001:660:101:101:200:f8ff:fe31:17ec
3ffe:304:101:1:200:f8ff:fe31:17ec
$ haah 134.157.19.71
Canonical name:
bernays.logique.jussieu.fr
Adresses:
134.157.19.71
$
```

Le programme réalisant la commande haah ne présente aucune difficulté. C'est une simple application des primitives précédemment décrites.

```
 1| #include <stdio.h> </tt>
 2|
 3| #include <string.h>
 4| #include <errno.h>
 5| #include <sys/types.h>
 6| #include <sys/socket.h>
 7| #include <netinet/in.h>
 8| #include <netdb.h>
 9| #include <arpa/inet.h>
10|
11|
12| int main(int argc, char **argv)
13| {
14|    int ret;
15|    struct addrinfo *res, *ptr;
16|    struct addrinfo hints = {
17|       AI_CANONNAME,
18|       PF_UNSPEC,
19|       SOCK_STREAM,
20|       0,
21|       0,
22|       NULL,
23|       NULL,
24|       NULL
25|    };
26|
27|    if (argc != 2) {
28|       fprintf(stderr, "%s: usage: %s host | addr.\n", *argv, *argv);
29|       exit(1);
30|    }
31|    ret = getaddrinfo(argv[1], NULL, &hints, &res);
32|    if (ret) {
33|       fprintf(stderr, "getaddrinfo: %s\n", gai_strerror(ret));
34|       exit(1);
35|    }
37|    for (ptr = res; ptr; ptr = ptr->ai_next) {
38|       if (ptr->ai_canonname)
39|          fprintf(stdout,"Canonical name:\n%s\nAdresses:\n", ptr->ai_canonname);
40|       switch (ptr->ai_family) {
41|          case AF_INET:
```

```
42|          {
43|             char dst[INET_ADDRSTRLEN];
44|             struct in_addr *src = &((struct sockaddr_in *) ptr->ai_addr)->sin_addr;
45|
46|             if(!inet_ntop(AF_INET, (const void *) src, dst, sizeof(dst))) {
47|                fprintf(stderr, "inet_ntop: %s\n", strerror(errno));
48|                break;
49|             }
50|             fprintf(stdout, "%s\n", dst);
51|             break;
52|          }
53|          case AF_INET6:
54|          {
55|             char dst[INET6_ADDRSTRLEN];
56|             struct in6_addr *src=&((struct sockaddr_in6 *)ptr->ai_addr)->sin6_addr;
57|
58|             if (!inet_ntop(AF_INET6, (const void *) src, dst, sizeof(dst))) {
59|                fprintf(stderr, "inet_ntop: %s\n", strerror(errno));
60|                break;
61|             }
62|             fprintf(stdout, "%s\n", dst);
63|             break;
64|          }
65|          default:
66|             fprintf(stderr, "getaddrinfo: %s\n", strerror(EAFNOSUPPORT));
67|       }
68|    }
69|    freeaddrinfo(res);
70|    exit(0);
71| }
```

Source : http://livre.point6.net/i

9 - Travail à faire

Modifier votre client et votre serveur pour les rendre compatible IPV6