Partiel du 07/03/2024 - 14h

Répartition pour le partiel

Les étudiant·es dont le nom commence par une lettre entre :

- A et M : amphi 1A
- N et Z : amphi 5C

Aucun document autorisé

V - Connexion par adresse de noms

Connexion par adresse de noms

Pour l'humain, retenir une adresse composée de noms plutôt qu'une suite de nombres est plus simple.

Si on souhaite permettre à l'utilisateur d'entrer une adresse de noms,

- on ne peut plus utiliser les fonctions `inet_pton` et `inet_ntoa`,
- il faut donc pouvoir traduire ces adresses de noms en adresse IP :
  - On a vu que cela signifie qu'il faut interroger un dictionnaire (en général service DNS) pour obtenir la traduction d'une adresse de noms en adresse IP.
  - On peut vouloir ne retenir qu'un sous-ensemble d'adresses répondant à des critères particuliers (IPv4, IPv6, avec socket TCP ou UDP)
  - Mais comment fait-on en C ?

Connexion par adresse de noms

struct addrinfo

On commence par remplir une variable `hints` de type `struct addrinfo` avec des indications comme le type d'adresses ou de socket voulu.

```
struct addrinfo {
  int ai_flags;              /* options */
  int ai_family;             /* famille d'adresses pour la socket */
  int ai_socktype;           /* type de socket */
  int ai_protocol;           /* protocole pour la socket */
  socklen_t ai_addrlen;      /* taille de l'adresse */
  struct sockaddr *ai_addr; /* addresse: IP, port... */
  char *ai_canonname;        /* nom canonique de l'adresse */
  struct addrinfo *ai_next; /* pointeur sur le suivant dans la liste */
};
```

Connexion par adresse de noms

struct addrinfo

Valeurs d'initialisation de `hints` pour obtenir des adresses IPv4 pour socket TCP :

- `ai_flags` : égal à `0` pour le moment
- `ai_family` : égal à `AF_INET`
- `ai_socktype` : égal à `SOCK_STREAM`
- `ai_protocol` : égal à `0` dans le cas de TCP

Tous les autres éléments de `struct addrinfo` passés via `hints` doivent être mis à `0` (ou au pointeur `NULL`).

Connexion par adresse de noms

Initialisation de `hints`

On dit que l'on veut récupérer des adresses IPv4 pour socket TCP

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

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

Et maintenant, comment interroger le dictionnaire pour traduire les noms d'adresses ?

Connexion par adresse de noms

getaddrinfo

On doit utiliser la fonction

```
int getaddrinfo(const char *node, const char *service,
                const struct addrinfo *hints,
                struct addrinfo **res);
```

- `node` : chaîne de caractères contenant l'adresse de noms
- `service` : chaîne de caractères contenant le numéro de port
- `hints` : contient des indications sur le type d'adresses, de socket...
- `res` : liste chaînée de pointeurs de `struct addrinfo` qui recevra les différentes adresses de socket possibles

Connexion par adresse de noms

getaddrinfo

`getaddrinfo` va remplir la liste chaînée `res`.

- Le champs `ai_addr` de chaque élément de `res` pointe sur une adresse de socket remplie.
- Ce champs est de longueur `ai_addrlen`.
- Le champs `ai_next` du dernier élément de la liste est égal à `NULL`.

```
struct addrinfo *r;
if ((getaddrinfo(hostname, port, &hints, &r)) != 0) exit(EXIT_FAILURE);
```

- `hostname` et `port` sont des chaînes de caractères contenant respectivement l'adresse de noms et le numéro de port de l'entité dont on veut récupérer les adresses.
- `r` contient maintenant la liste chaînée des « adresses ».

Connexion par adresse de noms

client IPv4

`getaddrinfo` retourne **0** en cas de succès et un entier non nul correspondant à un code d'erreur sinon.

On peut afficher le message correspondant à l'erreur avec la fonction :
`const char *gai_strerror(int errcode);`

```
int ret = getaddrinfo(hostname, port, &hints, &r);
if (ret != 0) {
   fprintf(stderr, "getaddrinfo: %s\n", gai_strerror(ret));
   exit(EXIT_FAILURE);
}
```

Connexion par adresse de noms

client IPv4

Et maintenant, comment un client peut-il se connecter au serveur connaissant l'adresse de noms de ce dernier ?

Pour se connecter le client teste chaque élément `p` de la liste d'adresses `r` jusqu'à obtenir une connexion. Pour cela :

1. il crée une socket,
2. tente de se connecter avec la socket à l'adresse et au port de `p`,
3. si la connexion réussie, il passe à la suite...
4. sinon, il ferme la socket et recommence le test sur l'élément suivant de `r` tant que celui-ci n'est pas `NULL`.

Connexion par adresse de noms

client IPv4

```
struct addrinfo *p;
p = r;

while( p != NULL ){
  if((sock = socket(p->ai_family, p->ai_socktype, p->ai_protocol)) > 0){
    if(connect(sock, p->ai_addr, p->ai_addrlen) == 0)
      break;

    close(sock);
  }

  p = p->ai_next;
}

if( p == NULL) exit(EXIT_FAILURE);

// le client est maintenant connecté,
// la conversation avec le serveur peut commencer...
```

Connexion par adresse de noms

client IPv4

Lorsqu'on a plus besoin de la liste chaînée d'adresses, il faut libérer la mémoire allouée par getaddrinfo en faisant appel à la fonction freeaddrinfo :

```
void freeaddrinfo(struct addrinfo *res);
```

ce qui donne ici

```
if(r)
  freeaddrinfo(r);
```

Connexion par adresse de noms

Et pour un client IPv6 ?

```
struct addrinfo hints, *r, *p;
memset(&hints, 0, sizeof(hints));
hints.ai_family = AF_INET6;     // socket IPv6
hints.ai_socktype = SOCK_STREAM;

if ((getaddrinfo(hostname, port, &hints, &r)) != 0)
  exit(EXIT_FAILURE);

p = r;
while( p != NULL ){
  if((*sock = socket(p->ai_family, p->ai_socktype, p->ai_protocol)) > 0){
    if(connect(*sock, p->ai_addr, p->ai_addrlen) == 0)
      break;
    close(*sock);
  }
  p = p->ai_next;
}

freeaddrinfo(r);
```

Connexion par adresse de noms

client polyvalent IPv4/IPv6

Le client ne connaît pas toujours a priori le type de connexion attendu par le serveur. Dans ce cas, il faut affecter à `hints.ai_family` la valeur **AF_UNSPEC**.

Il faut alors distinguer suivant le type de connexion pour récupérer l'adresse du client. Les champs de la structure `struct addrinfo` récupérés après création de la socket et connexion réussie :

- `ai_family`
- ou `ai_addrlen`

permettent de faire cette distinction.

```
if(p->ai_family == AF_INET)
    struct sockaddr_in sadr = *((struct sockaddr_in *) p->ai_addr);
else
    struct sockaddr_in6 sadr = *((struct sockaddr_in6 *) p->ai_addr);
```

Connexion par adresse de noms

client polyvalent IPv4/IPv6

On peut également utiliser un type `union` :

- se déclare un peu comme une structure :

```
union sockadresse{
 struct sockaddr_in sadr4;
 struct sockaddr_in6 sadr6;
};
```

- la valeur d'un seul champs peut être stockée simultanément :

```
union sockadresse adr;
if(p->ai_family == AF_INET)
    memcpy(&(adr->sadr4), (struct sockaddr_in *) p->ai_addr, p->ai_addrlen);
else
    memcpy(&(adr->sadr6), (struct sockaddr_in6 *) p->ai_addr, p->ai_addrlen);
```

La place mémoire occupée par un type `union` est celle de son plus grand champ. Pour la variable `adr`, c'est donc celle du champ **sadr6**.

Connexion par adresse de noms

client polyvalent IPv4/IPv6

```
struct addrinfo hints, *r, *p;
memset(&hints, 0, sizeof(hints));
hints.ai_family = AF_UNSPEC;
hints.ai_socktype = SOCK_STREAM;

if (getaddrinfo(hostname, port, &hints, &r)) return -1;

p = r;
while( p != NULL ){
  if((sock = socket(p->ai_family, p->ai_socktype, p->ai_protocol)) > 0){
    if(connect(sock, p->ai_addr, p->ai_addrlen) == 0) break;
    close(sock);
  }
  p = p->ai_next;
}
if (p == NULL) //erreur

//on stocke l'adresse de connexion et sa taille
int addrlen = p->ai_addrlen;
union sockadresse adr;
if(p->ai_family == AF_INET)
    memcpy(&(adr->sadr4), (struct sockaddr_in *) p->ai_addr, p->ai_addrlen);
else
    memcpy(&(adr->sadr6), (struct sockaddr_in6 *) p->ai_addr, p->ai_addrlen);

freeaddrinfo(r);
```

Déterminer l'adresse de noms

A contrario, on peut parfois déterminer l'adresse de nom, ainsi que le nom du service, correspondant à une adresse de connexion.

On utilise pour cela la fonction :

```
int getnameinfo(const struct sockaddr *restrict addr,
        socklen_t addrlen, char *host, socklen_t hostlen,
        char *serv, socklen_t servlen, int flags);
```

où

- `addr` : adresse de connexion de taille `addrlen`,
- `host` : adresse de noms correspondante de taille `hostlen`,
- `serv` : service correspondant de taille `servlen`
- `flags` : vaut **0** si on souhaite obtenir l'adresse de noms et le nom du service
- retourne **0** en cas de succès et un code d'erreur non nul sinon. Le message d'erreur peut être récupéré avec `gai_strerror`

`host` et `serv` doivent être alloués au préalable.

Si l'adresse de noms ou le nom du service ne peuvent être résolus alors ce sont les valeurs numériques qui sont récupérées.

Déterminer l'adresse de noms

```
int sockclient = accept(sock, (struct sockaddr *) &adrclient, &size);

char hbuf[NI_MAXHOST], sbuf[NI_MAXSERV];
if (getnameinfo((struct sockaddr *) &adrclient, size,
                hbuf, sizeof(hbuf), sbuf, sizeof(sbuf), 0) == 0)
  printf("hôte = %s, service = %s\n", hbuf, sbuf);
```

où `adrclient` est de type `struct sockaddr_in` ou `struct sockaddr_in6`

VI - Les options de socket

Les options de socket

setsockopt

La fonction

```
int setsockopt(int sockfd, int level, int optname,
               const void *optval, socklen_t optlen);
```

permet de modifier les options de socket.

- `sockfd` : descripteur de la socket à modifier
- `level` : indique le protocole de la suite auquel s'adresse l'appel. Valeurs possibles : `IPPROTO_IP`, `IPPROTO_TCP`, `IPPROTO_UDP` ou, pour les options plus générales, `SOL_SOCKET`
- `optname` : entier correspondant à l'option à modifier
- `optval` : pointeur sur la nouvelle valeur de l'option de longueur `optlen`

Les options de socket

getsockopt

La valeur courante d'une option peut être consultée avec la fonction

```
int getsockopt(int sockfd, int level, int optname,
                           void *val, socklen_t *len);
```

L'Internet est en phase de transition de l'IPv4 vers l'IPv6.

Une application qui ne communique qu'en IPv4 peut devenir obsolète.

Une application qui ne communique qu'en IPv6 n'est pas encore très utile.

On veut donc écrire des applications dites *double-stack*, c'est-à-dire qui communiquent en IPv4 ou en IPv6 selon la situation.
Côté serveur, on peut programmer une application qui :

- soit utilise deux sockets server, une pour IPv4, l'autre pour IPv6,
- soit utilise une socket server polymorphe.

Les options de socket

Socket polymorphe

Une socket IPv6 écoutant sur un port peut, sur certains systèmes, accepter les connexions IPv4. On parle alors de socket polymorphe.

Une **socket polymorphe** est une socket IPv6 traduisant une adresse IPv4 en une adresse IPv6 dite ***IPv4-mapped*** :

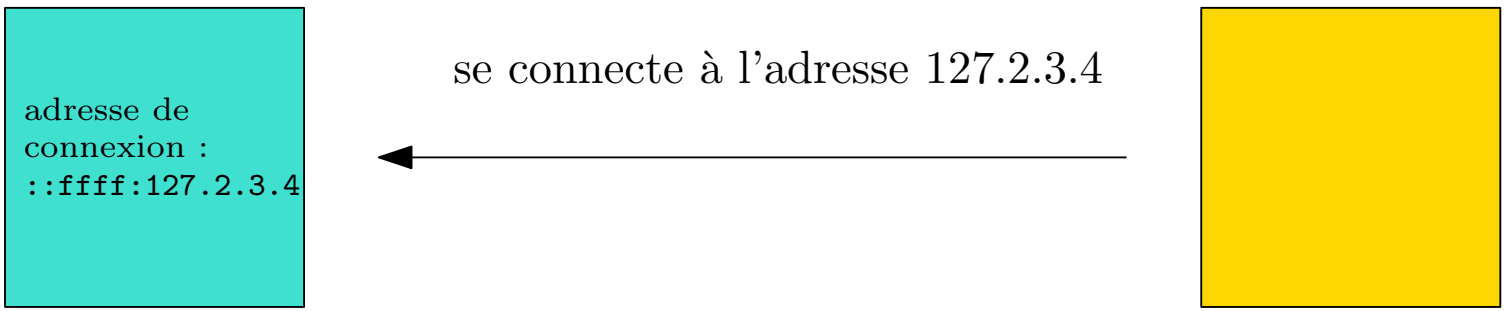
- L'adresse IPv4 `w.x.y.z` se traduit en l'adresse IPv4-mapped `::ffff:w.x.y.z`
  `127.2.3.4` → `::ffff:127.2.3.4` (ou `::ffff:7f02:304`)
- Ces adresses sont des représentations pour un usage interne de l'hôte. Elles ne correspondent à rien sur le réseau.
- Si on a une socket polymorphe liée à un port et qu'un client IPv4 se connecte, alors `accept` retourne une adresse IPv4-mapped.

Les options de socket

Socket polymorphe

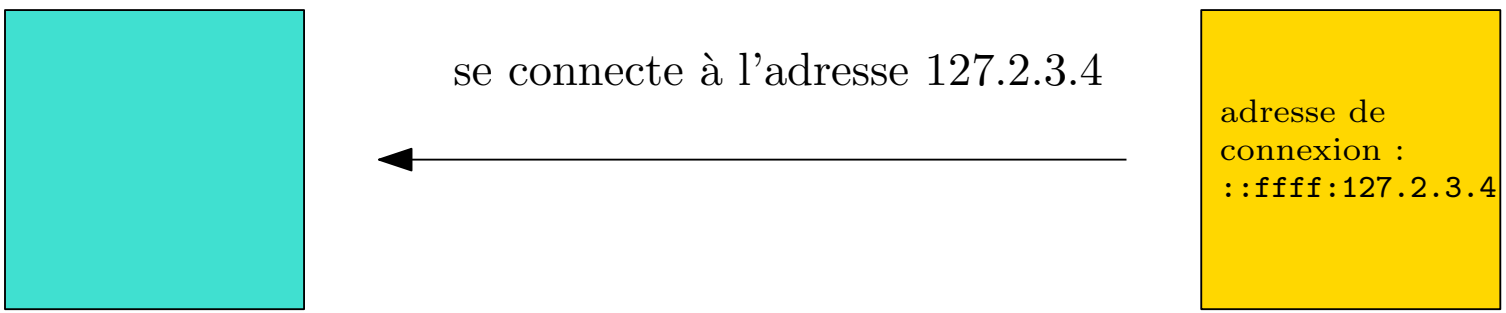
Serveur IPV6

avec socket polymorphe



Client IPv4

Serveur IPV4



Client IPv6

avec socket polymorphe

Les options de socket

Socket server polymorphe

Sur beaucoup de systèmes, une socket IPv6 est par défaut polymorphe. Mais ce n'est pas toujours le cas. Pour la portabilité, on demande explicitement à ce que la socket soit polymorphe.

Il faut pour cela désactiver l'option `IPV6_V6ONLY` au niveau `IPPROTO_IPV6`

```
int no = 0;
int r = setsockopt(sock, IPPROTO_IPV6, IPV6_V6ONLY, &no, sizeof(no));
if(r < 0)
    fprintf(stderr, "échec de setsockopt() : (%d)\n", errno);
```

Cela doit être fait **avant** le `bind` .

Si au contraire, on ne veut accepter que de l'IPv6, `no` doit prendre la valeur `1`.

Les options de socket

Socket client polymorphe

Un client polymorphe est un client IPv6 qui traite les adresses IPv4 comme des adresses IPv4-mapped.

On utilise de nouveau `getaddrinfo` dans le client IPv6 avec les bons *flags* pour la variable `hints`.

```
hints.ai_flags = AI_V4MAPPED | AI_ALL;
```

Alors si un serveur accepte uniquement les connexions IPv4 et tourne en local sur le port 7777, on obtient :

```
Cours5$ ./client6 localhost 7777
Tentatives :
adresse : IP: ::1 port: 7777
adresse : IP: ::ffff:127.0.0.1 port: 7777

Connexion réussie sur :
IP: ::ffff:127.0.0.1 port: 7777
```

```
Cours5$ ./client6 127.0.0.1 7777
Tentatives :
adresse : IP: ::ffff:127.0.0.1 port: 7777

Connexion réussie sur :
IP: ::ffff:127.0.0.1 port: 7777
```

Les options de socket

Réutilisation d'un numéro de port

Lorsqu'une application ferme sa socket avec `close` ou si elle plante, la socket passe dans un état `TIME-WAIT` et le système la maintient dans cet état pendant quelques secondes, voire quelques minutes.

Il est alors impossible de lancer une application qui fait un `bind` sur la même adresse et le même port pendant ce temps. L'état `TIME-WAIT` représente le temps qu'il faut attendre pour être sûr que l'application distante a bien reçu la demande de fin de connexion.

Pour parer à l'échec du `bind`, il faut ajouter l'option `SO_REUSEADDR` à la socket.

```
int yes = 1;
int r = setsockopt(sock, SOL_SOCKET, SO_REUSEADDR, &yes, sizeof(yes));
if(r < 0)
    fprintf(stderr, "échec de setsockopt() : (%d)\n", errno);
```

Cela doit être fait **avant** le `bind`.

Sur certains système, il ne faut pas utiliser la constante **SO_REUSEADDR** mais la constante **SO_REUSEPORT**. C'est notamment le cas sur macOS.

On peut dans ce cas compiler en définissant une variable `MAC` :

```
gcc -DMAC serveur.c -o serv
```

et ajouter le code suivant au début de `serveur.c`

```
#ifdef MAC
#ifdef SO_REUSEADDR
#undef SO_REUSEADDR
#endif
#define SO_REUSEADDR SO_REUSEPORT
#endif
```