Programmation réseau 4

Juliusz Chroboczek

12 février 2019

1 Adressage

La couche application ne présente pas à l'utilisateur des adresses IP ; elle présente des *noms de hôte* de la forme `www.irif.fr`. Or, la couche transport demande des adresses de *socket*, et donc des adresses IP. En général, on peut associer plusieurs adresses à un seul nom de hôte :

- le hôte distant peut avoir des adresses IPv4 et des adresses IPv6 ;
- le hôte distant peut avoir plusieurs interfaces connectées à des fournisseurs de service différents, par exemple pour augmenter la fiabilité ou pour distribuer la charge ;
- le hôte distant peut avoir plusieurs adresses sur une seule interface car l'administrateur aime se compliquer la vie.

Pour augmenter la robustesse d'un client, il est bon d'être brutal, et d'essayer de se connecter à toutes les adresses du hôte distant.

Le *système de noms de domaine* (DNS) est une base de données distribuée qui associe à chaque nom de hôte (par exemple `www.irif.fr`) zéro, une ou plusieurs adresses IP. Au niveau du *shell* Unix, la commande `host` permet de consulter le DNS[1]. Dans un client écrit en C, c'est la fonction `getaddrinfo` qu'il faut utiliser[2].

1.1 Digression : listes chaînées

Il existe deux façons d'implémenter les listes chaînées. Dans une liste chaînée *externe*, la liste chaînée consiste d'une suite de cellules qui contiennent chacune une référence à une donnée :

```
struct data {
    ...
}

struct cell {
    struct data *data;
    struct cell *next;
}
```

1. Vous pouvez aussi utiliser `dig` si vous aimez vous compliquer la vie.
2. La fonction `gethostbyname` est obsolète.

Dans une liste chaînée interne, c'est la donnée elle-même qui contient une référence à la queue de la liste :

```
struct data {
    ...
    struct data *next;
}
```

Les suites internes ne sont pas modulaires : comme c'est la structure de données elle-même contient les données nécessaires à l'implémentation de la suite chaînée, elle ne peut pas (facilement) être utilisée dans une structure de données pour laquelle elle n'a pas été prévue, par exemple une liste doublement chaînée ou un arbre. Par contre, elles ont l'avantage de diviser par deux le nombre d'allocations de mémoire ainsi que le nombre d'étapes de *pointer chasing*.

1.2 La fonction `getaddrinfo`

La fonction `getaddrinfo` prend un nom de hôte et un nom de service et retourne une liste chaînée interne de `struct addrinfo` qui contiennent toutes les données nécessaires pour appeler `socket` et `connect` :

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

Le paramètre `node` contient le nom de hôte (par exemple `"www.irif.fr"`). Le paramètre `service` contient soit un nom de service (`"http"`), soit un numéro de port sous forme de chaîne (`"80"`, `snprintf` est votre amie). Le paramètre `hints` contient les options de la consultation — il permet notamment de contrôler la résolution *double-stack* (voir paragraphe 3 ci-dessous). En cas de succès, l'adresse indiquée par le paramètre `res` contiendra une liste chaînée des adresses obtenues.

Attention, `getaddrinfo` utilise une convention d'appel inhabituelle : en cas de succès, il retourne 0 ; en cas d'échec, il retourne un code d'erreur qu'on peut passer à `gai_strerror`. La variable `errno` n'est pas positionnée.

La structure de données retournée par `getaddrinfo` dans `res` peut être détruite à l'aide de `freeaddrinfo` :

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

1.3 Utilisation de `getaddrinfo`

Pour appeler `getaddrinfo`, il faut commencer par constuire la structure `hints` :

```
struct addrinfo hints;
memset(&hints, 0, sizeof(hints));
hints.ai_family = AF_UNSPEC;
hints.ai_socktype = SOCK_STREAM;
hints.ai_protocol = 0;
hints.ai_flags = 0;
```

Le champ `ai_family` indique le type d'adresse désiré (IPv4, IPv6 ou les deux). Les champs `ai_socktype` et `ai_protocol` sont identiques aux paramètres de l'appel système `socket`. Le champ `ai_flags` contrôle notamment la résolution *double-stack*, voir paragraphe 3 ci-dessous.

```
struct addrinfo *res;
rc = getaddrinfo(host, service, &hints, &res);
if(rc != 0) {
    fprintf(stderr, "Échec cinglant : %s\n", gai_strerror(rc));
    exit(1);
}
```

La variable `res` contient maintenant la liste des adresses obtenues ; on va les essayer toutes :

```
struct addrinfo *p;
for(p = res; p != NULL; p = p->ai_next) {
    s = socket(p->ai_family, p->ai_socktype, p->ai_protocol);
    if(s < 0) {
        close(s);
        continue;
    }

    rc = connect(s, p->ai_addr, p->ai_addrlen);
    if(rc >= 0)
        break;
    close(s);
}
```

Remarquez qu'on créée une nouvelle *socket* à chaque itération, et qu'on la détruit en cas d'échec — ça permet de gérer le cas où les adresses n'appartiennent pas toutes à la même famille (IPv4 et IPv6).

Si on est sortis de la boucle, soit une connexion a réussi, soit nous avons épuisé la liste d'adresses. Débarassons-nous du deuxième cas et libérons la liste d'adresses :

```
if(p == NULL) {
    fprintf(stderr, "La connection a échoué.\n");
    exit(1);
}

freeaddrinfo(res);
```

On peut maintenant travailler avec la *socket* connectée `s`.

2 *Happy Eyeballs*

S'il est relativement brutal, l'algorithme ci-dessus n'est pas très rapide :

– il attend l'arrivée de toutes les adresses (IPv4 et IPv6) avant de commencer à se connecter à la première ;
– il attend l'échec d'une connexion avant d'essayer l'adresse suivante.

Le premier point n'est normalement problématique que si le DNS a un hoquet. Le deuxième point, par contre, demande à ce que les connexions aux adresses inaccessibles échouent rapidement — et l'échec rapide d'une connexion dépend du bon fonctionnement des protocoles ICMP et ICMPv6, qui sont parfois bloqués par des administrateurs réseau qui seront les premiers contre le mur lorsque la révolution viendra[3].

L'algorithme *Happy Eyeballs* (RFC 8305) vise à éviter ce délai en se montrant encore plus brutal[4]. Un client qui implémente *Happy Eyeballs* demande au DNS la liste des adresses du hôte distant, et commence à se connecter à ces dernières au fur et à mesure qu'elles sont reçues (il n'est donc pas possible d'implémenter *Happy Eyeballs* au dessus de `getaddrinfo`). Dès lors qu'une connexion réussit, les autres connexions en cours sont annulées.

Happy Eyeballs est utile dans les applications qui effectuent de nombreuses connexions TCP, notamment les navigateurs *web*. Pour les protocoles de couche application qui n'ont pas un tel besoin de brutalité, l'algorithme simple décrit au paragraphe 1.3 ci-dessus suffit.

3 Programmation *double stack*

Pour mémoire, une application *double stack* est une application capable de parler indifféremment IPv4 et IPv6. Dans l'API *sockets*, il existe deux façons de gérer la problématique du *double stack* : dans l'approche *monomorphe* on utilise une *socket* (`PF_INET` ou `PF_INET6`) par famille d'adresses ; dans l'approche polymorphe, on utilise une seule *socket* `PF_INET6` « polymorphe » qui gére les deux familles d'adresses.

Dans ce qui précède, j'ai été incohérent :

– dans le serveur TCP, j'ai utilsé l'approche polymorphe — j'ai utilisé une seule *socket* passive de type `PF_INET6` qui desservait aussi bien IPv4 que IPv6 ;
– dans le client TCP, par contre, j'ai utilisé `getaddrinfo` pour m'aider à créer *sockets* de types différents pour les connexions IPv4 et IPv6.

POSIX dit que les *sockets* de type `PF_INET6` sont polymorphes par défaut, ce qui est normalement le cas sous Linux mais pas forcément sous certains systèmes qui violent la norme (OpenBSD et DragonflyBSD, c'est vous que je regarde). On peut mettre explicitement une *socket* en mode polymorphe ou monomorphe à l'aide de l'option `IPV6_V6ONLY` de `setsockopt`.

3.1 Approche monomorphe

Dans l'approche monomorphe, l'application prend soin de créer des *sockets* du bon type selon la famille d'adresses. Du côté client, `getaddrinfo` fournit le champ `ai_family`, qui permet de

3. En l'absence d'aministrateurs fascistes, si la réussite d'une connexion est indiquée par le pair distant lors du *handshake* initial, son échec, par contre, est normalement indiqué par un segment RST provenant du pair ou par un paquet ICMP(v6) provenant du réseau ; cependant, il se peut que le hôte distant soit planté sans que le réseau l'ait détecté, et dans ce cas le hôte local n'obtiendra aucune indication d'erreur.

4. Si la violence ne résoud pas vos problèmes, c'est que vous n'en appliquez pas assez. Amitiés à David.

chosir le bon type — voyez le code du paragraphe 1.3 ci-dessus.

Du côté serveur, il faut créer deux *sockets* liées au même numéro de port. Pour cela, il faut éviter que la *socket* IPv6 n'occuppe le port IPv4, soit en liant la *socket* IPv4 avant de lier la *socket* IPv6, soit en mettant explicitement la *socket* IPv6 en mode monomorphe (`IPV6_V6ONLY`) avant de la lier.

3.2 Approche polymorphe

Dans l'approche polymorphe, on utilise une seule *socket* IPv6 pour gérer les deux familles d'adresses. Les adresses IPv4 sont représentées par des adresses *v6-mapped* : des adresses IPv6 dans le préfixe `::ffff:0:0/96`. Par exemple, l'adresse IPv4 `81.194.27.171` est représentée par l'adresse `::ffff:51:c2:1b:ab`, qu'on peut aussi écrire `::ffff:81.194.27.171`. En particulier :

- les appels système `accept` (et `recvfrom`), lorsqu'ils sont appliqués à des *sockets* polymorphes, retournent des adresses v6-mapped dans des `sockaddr_in6` au lieu de retourner des `sockaddr_in` ;
- pour se connecter à une adresse IPv4 à l'aide d'une *socket* polymorphe, il faut passer une adresse *v6-mapped* à l'appel système `connect`.

Du côté serveur, il n'y a normalement rien à faire, sauf gérer correctement les adresses retournées par `accept`. Du côté client, il faut spécifier la valeur `(AI_V4MAPPED | AI_ALL)` dans le champ `ai_flags` de la structure `flags` passée à `getaddrinfo`.

4 Adresses locales et NAT

Adresses locales en IPv4 En principe, une adresse IP est globalement unique — à un moment donné, elle est attribuée à (une interface d') un seul hôte. En pratique, lorsque les gens mettent en place des réseaux isolés (non connectés à l'Internet Global), ils « squattent » des adresses IP sans passer par les procédures administratives qui visent à s'assurer de leur unicité.

La RFC 1918 formalise cet état de fait : elle définit trois plages d'adresses privées, qui peuvent être utilisées dans les réseaux privés sans aucune procédure administrative. Ce sont :

- `10.0.0.0/8` ;
- `172.16.0.0/12` ; et
- `192.168.0.0/16`.

Traduction d'adresses La *traduction d'adresses* (NAT, *Network Address Translation*) est une technique qui permet à plusieurs hôtes d'utiliser une seule adresse globale pour le champ source des paquets qu'ils émettent. Un *routeur NAT* est un routeur qui modifie les adresses des paquets qui le traversent pour implémenter la traduction d'adresses.

À la différence d'un routeur ordinaire, qui est symétrique, un routeur NAT divise l'Internet en un réseau local qui est « à l'intérieur » du NAT, et le reste de l'Internet, qui est « à l'extérieur ». Les paquets qui se situent à l'extérieur utilisent des adresses IP globales, tandis que les paquets qui se trouvent à l'intérieur utilisent des adresses locales.

On attribue une ou plusieurs adresses globales au routeur NAT. Lorsqu'un paquet transite de l'intérieur vers l'extérieur, le NAT remplace l'adresse source du paquet par l'une de ses adresses globales (il peut aussi changer le numéro de port de la couche transport) ; il retient alors l'association entre la paire (adresse globale locale, adresse distante) et l'adresse privée dans une structure de données.

Lorsqu'un paquet arrive de l'extérieur, le routeur NAT recherche la paire (destination, source) dans sa structure de données. Si elle est présente, il effectue la traduction inverse ; sinon, il rejette le paquet.

La technique de traduction a permis de survivre à l'épuisement des adresses IPv4 ; cependant, son utilisation a des conséquences graves pour l'Internet. Tout d'abord, un NAT recquiert que le premier paquet d'un flot passe de l'interieur vers l'extérieur, ce qui empêche de déployer des serveurs à l'intérieur — on revient donc à un modèle de type Minitel, où seuls les riches et les puissants peuvent se permettre de déployer des serveurs. Ensuite, le NAT consulte des informations de couche transport (le numéro de port), ce qui empêche le déploiement de nouveaux protocoles. Enfin, le NAT introduit de l'état à l'intérieur du réseau, ce qui rend le réseau plus fragile (moins résilient aux pannes) et empêche le routage asymétrique.

Adresses locales IPv6 En IPv6, les adresses locales s'appellent des ULA et sont dans le préfixe `fc00::/7`. Les ULA sont globalement uniques avec haute probabilité, ce qui évite la plupart des problèmes liés aux adresses locales, notamment lors de la mobilité des hôtes et lors de la fusion des réseaux.

Un préfixe ULA est normalement choisi dans `fd00::/8` (`fc00::/8` est réservé) ; les 40 bits qui suivent le préfixe sont choisis aléatoirement avec une distribution uniforme, ce qui limite la probabilité de collision. Le résultat est un préfixe ULA de taille `/48`, qui contient 2^{16} préfixes de taille `/64` dont chacun peut être affecté à un lien distinct.

À la différence des adresses locales IPv4, qui sont utilisées dans des réseaux isolés ou cachés derrière un NAT, les ULA sont prévues pour être utilisées dans des réseaux ordinaires en plus des adresses globales (pensez à une imprimante qui n'a pas besoin d'être globalement routable).

Il n'y a pas de NAT en IPv6. En principe.