Programmation *sockets*

Juliusz Chroboczek

13 octobre 2014

Table des matières

1 Fonctionnalités indépendantes du protocole de couche transport **2**
1.1 Structures . . . 2
1.2 Boutisme . . . 3
1.3 Création d'une *socket* . . . 4
1.4 Destruction d'une *socket* . . . 4
1.5 Liaison à un numéro de port . . . 4
1.6 Options de *socket* . . . 4

2 Communication UDP **5**

3 Communication TCP **5**
3.1 Établissement de connexion . . . 5
3.1.1 Côté client . . . 5
3.1.2 Côté serveur . . . 6
3.1.3 Réutilisation d'un numéro de port . . . 6
3.2 Identification du pair distant . . . 6
3.3 Désactivation de l'algorithme de Nagle . . . 6
3.4 Transfert de données . . . 7
3.5 Fermeture de connexion . . . 7

4 Résolution de noms **7**
4.1 La fonction `gethostbyname` . . . 7
4.2 La fonction `getaddrinfo` . . . 8

5 Écriture d'applications *double-stack* **9**
5.1 *Sockets* polymorphes et monomorphes . . . 9
5.2 Utilisation de `getaddrinfo` . . . 10

Introduction

L'Unix d'origine (6e et 7e édition) ne disposait pas d'une interface au réseau. Dans les années 1980, plusieurs projets de recherche ont cherché à ajouter une interface réseau au système ; on peut citer en particulier :

- l'interface dite « sockets », développée à l'Université de Californie à Berkeley (UCB) pour Unix BSD ;
- l'interface dite « TLI » (basée sur « Streams ») développée aux *Bell Labs* d'AT&T pour Unix Système V.

Aujourd'hui, l'interface TLI est obsolète, est c'est l'interface *sockets* qui est utilisée dans tous les Unix modernes. Elle est présente depuis toujours dans les Unix BSD et Linux, elle a été rajoutée aux Unix Système V (HP/UX, Solaris), et une variante est utilisée sur les systèmes Windows (où elle s'appelle « Winsock 2 »). Elle fait partie de POSIX depuis l'édition de 2001.

Compilation de programmes *sockets* Sous Linux et les BSD modernes, l'interface *sockets* est dans la bibliothèque standard — il n'y a donc pas besoin d'options particulières.

Sous Unix Système V (Solaris) il faut ajouter les options `-lsocket -lnsl -lresolv` à la fin de la ligne de commande du compilateur C.

Pour utiliser *Winsock 2* sous Windows, il faudra lier votre programme à `WSOCK32.DLL`. Si vous utilisez les outils *mingw32*, il suffit d'ajouter `-lwsock32` à la fin de la ligne de commande du compilateur C. Si vous utilisez VC++, il faudra sûrement cliquer quelque part.

1 Fonctionnalités indépendantes du protocole de couche transport

Dans l'interface *sockets*, les *sockets* sont des i-nœuds comme les autres : elles sont manipulées à travers des descripteurs de fichier. L'API est polymorphe, et le polymorphisme est l'implémenté à l'aide de casts « unsafe » appliqués à des pointeurs vers des structures.

Les structures et fonctions décrites ci-dessous sont déclarées dans les fichiers d'entête `<sys/socket.h>`, `<netinet/in.h>` et `<netdb.h>`.

1.1 Structures

Une adresse IPv4 est représentée par une structure de type `struct in_addr` :

```
struct in_addr {
    in_addr_t s_addr;
};
```

où `in_addr_t` est un type entier de 32 bits (normalement `unsigned int`).

Une adresse IPv6 est représentée par une structure de type `struct in6_addr` :

```
struct in6_addr {
    unsigned char s6_addr[16];
};
```

Une adresse de *socket* est représentée par un pointeur vers une structure (abstraite) de type `struct sockaddr` :

```
struct sockaddr {
    sa_family_t sa_family;
    ...
}
```

Les structures concrètes correspondant à des adresses de *sockets* IPv4 et IPv6 s'appellent `struct sockaddr_in` et `struct sockaddr_in6` respectivement :

```
struct sockaddr_in {
    sa_family_t sin_family;
    in_port_t sin_port;
    struct in_addr sin_addr;
};

struct sockaddr_in6 {
    sa_family_t sin6_family;
    in_port_t sin6_port;
    struct in6_addr sin6_addr;
};
```

Le champ `sa_family`, `sin_family` ou `sin6_family` indique le type concret de la structure : il vaut `AF_INET` pour une adresse IPv4, et `AF_INET6` pour une adresse IPv6. Le champ `sin_port` ou `sin6_port` contient le numéro de port, et le champ `sin_addr` ou `sin6_addr` l'adresse IP.

La structure `sockaddr_storage` est une variante de `sockaddr` qui est suffisamment grande pour contenir tout type `sockaddr`. C'est celle-là qu'il faut utiliser pour stocker une adresse dont on ne connaît pas encore le type concret.

1.2 Boutisme

Plusieurs champs des structures ci-dessus contiennent des champs qui doivent être stockés en *ordre réseau (network order)*, c'est-à-dire en ordre gros-boutiste *(big-endian)*. C'est notamment le cas des champs `sin_port`, `sin6_port`, `sin_addr` et `sin6_addr`. Cette bizarrerie n'est pas un problème pour les champs d'adresse, mais par contre un port doit être converti explicitement en format réseau.

Les fonctions `htons` et `ntohs` convertissent un entier de 16 bits de l'ordre local vers l'ordre réseau et inversément :

```
uint16_t htons(uint16_t hostshort);
uint16_t ntohs(uint16_t netshort);
```

Les fonctions `htonl` et `ntohl` effectuent une conversion analogue pour les entiers de 32 bits.

1.3 Création d'une *socket*

Une *socket* est créée à l'aide de l'appel système `socket` :

```
int socket(int domain, int type, int protocol);
```

Le paramètre `domain` vaut `PF_INET` pour des *sockets* IPv4, et `PF_INET6` pour des *sockets* IPv6. Le paramètre `type` vaut `SOCK_DGRAM` pour des *sockets* UDP et `SOCK_STREAM` pour des *sockets* TCP. Le paramètre `protocol` vaut 0.

Cette fonction retourne un descripteur de fichier en cas de succès, et -1 en cas d'échec. Le code d'erreur est alors dans `errno`.

1.4 Destruction d'une *socket*

Comme les autres structures représentées par un descripteur de fichier, une *socket* est détruite à l'aide de l'appel système `close`[1] :

```
int close(int fd);
```

1.5 Liaison à un numéro de port

La *socket* créée par `socket` n'est pas *liée* : le système ne lui a pas encore alloué de numéro de port ou d'adresse locale. Lors de sa première utilisation, elle sera liée à un numéro de port choisi aléatoirement par le système. C'est le comportement désiré du côté client, mais pas du côté serveur.

Pour lier une *socket* a un numéro de port spécifique, utilisez l'appel système `bind` :

```
int bind(int socket,
         struct sockaddr *address, socklen_t address_len);
```

Le paramètre `socket` est le descripteur de fichier représentant la *socket* à lier. Le paramètre `address` est un pointeur sur une structure `sockaddr` ; typiquement, vous mettrez tous ses champs à 0 (à l'aide de la fonction `memset`) sauf la famille (`sin_family` ou `sin6_family`) et le numéro de port (`sin_port` ou `sin6_port`).

Le champ `address_len` représente la longueur du paramètre `address`, et vaut normalement `sizeof(struct sockaddr_in)` ou `sizeof(struct sockaddr_in6)`.

1.6 Options de *socket*

L'appel `setsockopt` permet de changer les comportements optionnels d'une *socket* :

```
int setsockopt(int s, int level, int optname,
               void *val, socklen_t len);
```

1. Ce n'est pas le cas sous Windows, où il faut utiliser un appel système spécifique nommé `closesocket`.

Le paramètre `s` est le descripteur correspondant à la *socket* à modifier. Le paramètre `level` indique le protocole de la suite auquel s'adresse l'appel — il vaut `IPPROTO_IP`, `IPPROTO_UDP`, `IPPROTO_TCP` ou, pour les options générales, `SOL_SOCKET`. Le paramètre `optname` est l'option à modifier ; `val` pointe sur la nouvelle valeur de l'option, qui a une longueur donnée par `len`.

La valeur courante d'une option peut être consultée à l'aide de `getsockopt` :

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

2 Communication UDP

On émet un datagramme à l'aide de la fonction `sendto` :

```
ssize_t sendto(int s, void *buf, size_t len, int flags,
               struct sockaddr *to, socklen_t tolen);
```

Le paramètre `s` est le descripteur associé à la *socket* à utiliser. Les données à envoyer sont dans le tampon `buf`, de longeur `len`. Le paramètre `flags` vaut 0. La destination du datagramme à envoyer est donnée par le paramètre `to` ; le paramètre `tolen` est la taille de celui-ci.

On reçoit un datagramme à l'aide de la fonction `recvfrom` :

```
ssize_t recvfrom(int s, void *buf, size_t len, int flags,
                 struct sockaddr *from, socklen_t *fromlen);
```

Les données reçues sont stockées dans `buf`, de taille `len`, et la fonction retourne le nombre d'octets effectivement reçus ; si le tampon est trop petit pour contenir le datagramme, les données sont tronquées et l'appel retourne `len`. L'adresse de l'émetteur est stockée dans le paramètre `from` (typiquement une `sockaddr_in` ou une `sockaddr_storage`), dont la taille est donnée par `fromlen` ; `fromlen` est mis à jour par cette fonction pour valoir la taille effective.

Remarquez l'asymétrie entre ces deux fonctions : pour `sendto`, le paramètre `tolen` est une valeur, tandis que pour `recvfrom`, `fromlen` sera mis à jour, et c'est donc un pointeur.

3 Communication TCP

3.1 Établissement de connexion

La procédure est différente du côté client et du côté serveur.

3.1.1 Côté client

Une connexion peut être créée de façon active à l'aide de l'appel système `connect` :

```
int connect(int sockfd,
            struct sockaddr *serv_addr, socklen_t addrlen);
```

Le paramètre `sockfd` est le descripteur correspondant à la *socket* à connecter. Le paramètre `serv_addr` indique l'adresse à laquelle se connecter (il aura comme type soit `sockaddr_in` soit `sockaddr_in6`), et `addrlen` est la taille du paramètre `serv_addr`.

3.1.2 Côté serveur

Après avoir lié la *socket* à un port local à l'aide de l'appel système `bind` (voir paragraphe 1.5), on demande au système de commencer à accepter des connexions avec l'appel système `listen` :

```
int listen(int sockfd, int backlog);
```

L'argument `backlog` indique en principe la longueur de la file de connexions en attente que le système va accepter, mais il est souvent ignoré de nos jours ; 1024 est un bon choix.

On accepte une connexion à l'aide de l'appel système `accept` :

```
int accept(int sockfd, struct sockaddr *addr, socklen_t *addrlen);
```

Si une demande de connexion a déjà été reçue sur cette *socket*, cet appel système retourne immédiatement ; sinon, il bloque jusqu'à l'arrivée d'un segment SYN. En cas de succès, il retourne le descripteur de la *socket* connectée (la *socket* passive continue à écouter), et l'adresse du pair distant est passée dans le paramètre `addr`.

3.1.3 Réutilisation d'un numéro de port

Après qu'une *socket* TCP a été liée à un port *p*, le noyau empêche la réutilisation du port *p* pendant quelques dizaines de secondes ; ce temps mort permet d'éviter qu'un segment destiné à une connexion précédemment fermée soit confondu avec un segment destiné à une nouvelle connexion. Cependant, ce temps mort empêche de relancer un serveur rapidement, et il n'y a aucun danger à l'éteindre du côté serveur.

On peut éviter le temps mort à l'aide de l'appel système `setsockopt` (voir paragraphe 1.6) :

```
int val;
rc = setsockopt(int s, SOL_SOCKET, SO_REUSEADDR, &val, sizeof(val));
```

Le paramètre `val` vaut 1 pour éviter le temps mort.

3.2 Identification du pair distant

L'adresse du pair distant est normalement connue aussi bien par le client (qui sait à quelle adresse il s'est connecté) que par le serveur (qui l'a obtenue lors de l'appel à `accept`). On peut cependant la demander de nouveau à l'aide de l'appel système `getpeername` :

```
int getpeername(int s, struct sockaddr *name, socklen_t *namelen);
```

Malgré son nom, cet appel ne retourne pas un nom de hôte, mais une adresse de *socket*.

L'adresse locale peut être obtenue par l'appel analogue `getsockname`.

3.3 Désactivation de l'algorithme de Nagle

L'algorithme de Nagle est par défaut utilisé pour toute *socket* TCP. Il peut être éteint à l'aide de l'appel système `setsockopt` (voir paragraphe 1.6) :

```
int val;
rc = setsockopt(int s, IPPROTO_TCP, TCP_NODELAY, &val, sizeof(val));
```

Le paramètre `val` vaut 1 pour éteindre l'algorithme de Nagle, 0 pour le rallumer.

3.4 Transfert de données

Le transfert de données se fait à l'aide des appels système habituels `read` et `write` auxquels l'on passe le descripteur de fichier d'une *socket* connectée comme premier paramètre[2].

3.5 Fermeture de connexion

Chacun des deux sens de la connexion peut être fermés à l'aide de `shutdown` :

```
int shutdown(int s, int how);
```

Le paramètre `s` est le descripteur de fichier de la *socket* ; le paramètre `how` indique le sens à fermer, et peut être :

- 1 ou `SHUT_WR`, qui envoie un segment FIN et ferme le côté écriture (c'est celui-là que vous voulez habituellement) ;
- 0 ou `SHUT_RD`, qui ferme le côté lecture et cause l'envoi de segments RST si d'autres données arrivent ;
- 2 ou `SHUT_RDWR`, qui combine les effets de `SHUT_WR` et `SHUT_RD`.

Un appel à `shutdown` ne détruit pas la *socket* ; pour cela, il faudra faire un appel à `close`.

4 Résolution de noms

Les applications réseau présentent à l'utilisateur des noms de hôtes ; par contre, les API de la couche transport prennent en paramètre des adresses IP. Un protocole dit de *résolution des noms* est donc nécessaire pour déterminer l'ensemble des adresses IP correspondant à un nom de hôte. Dans la suite TCP/IP, c'est le protocole DNS *(Domain Name System)* qui joue ce rôle.

Il y a deux fonctions pour accéder au DNS : l'ancienne fonction `gethostbyname`, qui ne supporte que IPv4 et n'est pas *thread-safe*, et la nouvelle fonction `getaddrinfo`, qui supporte IPv4 et IPv6 de façon uniforme.

4.1 La fonction `gethostbyname`

Un ensemble d'adresses est représenté par une structure de type `struct hostent` :

```
struct hostent {
    char *h_name;
    char **h_aliases;
    int h_addrtype;
    int h_length;
    char **h_addr_list;
};
#define h_addr h_addr_list[0]
```

2. Ce n'est pas le cas sous Windows, où il faut utiliser les appels système spécifiques `send` et `recv`.

Le champ `h_addr_list` contient la liste des adresses du hôte, sous forme d'un tableau de pointeurs sur des tableaux de `char` de longeur 4 chacun ; la fin du tableau est indiquée par un pointeur nul. La syntaxe `h_addr` est un synonyme de `h_addr_list[0]` — c'est l'adresse « préférée » du hôte.

La résolution d'un nom se fait en appelant la fonction `gethostbyname` :

```
struct hostent *gethostbyname(const char *name);
int h_errno;
```

En cas de succès, cette fonction retourne un pointeur sur une structure de type `struct hostent`. En cas d'échec, elle retourne un pointeur nul, avec `h_errno` (pas `errno` !) contenant le code d'erreur. La valeur contenue dans `h_errno` peut être analysée par les deux fonctions `herror` et `hstrerror` (analogues aux fonctions `perror` et `strerror` respectivement).

Comme les autres fonctions de l'interface demandent une adresse sous forme de `struct in_addr`, il faut effectuer une conversion un peu brutale :

```
host = gethostbyname("wifi.pps.jussieu.fr");
if(host == NULL) {
        ...
}
memcpy(&addr.s_addr, host->h_addr, 4);
```

Le pointeur retourné par `gethostbyname` est vers un tampon statique ; il n'y a donc pas besoin de le libérer. Par contre, les données seront écrasées par le prochain appel à `gethostbyname` (attention notamment si vous utilisez des *threads*).

4.2 La fonction `getaddrinfo`

Un ensemble d'adresses est représenté par une liste chaînée de structures de type `struct addrinfo` :

```
struct addrinfo {
    int              ai_flags;
    int              ai_family;
    int              ai_socktype;
    int              ai_protocol;
    size_t           ai_addrlen;
    struct sockaddr *ai_addr;
    char            *ai_canonname;
    struct addrinfo *ai_next;
};
```

Une telle liste chaînée est retournée par la fonction `getaddrinfo` :

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

Le paramètre `node` est le nom de hôte à résoudre ; le paramètre `service` peut soit valoir `NULL`, soit être la représentation textuelle d'un numéro de port, soit être le nom d'un service[3], dans lequel cas `getaddrinfo` remplira le champs `sin_port` ou `sin6_port` du résultat. Les champs `ai_socktype` et `ai_protocol` ont le même sens que les deux derniers paramètres de l'appel `socket`.

Le paramètre `hints` contient des paramètres supplémentaires. Il va normalement pointer sur une `struct addrinfo` initialisée à 0 si l'application est prête à accepter aussi bien des adresses IPv4 que IPv6 ; si l'application désire uniquement des adresses IPv4, le champ `hints->ai_family` doit être initialisé à `AF_INET`. (Mais voyez aussi le paragraphe 5.2 ci-dessous.)

À la différence des autres fonctions de la bibliothèque standard, la fonction `getaddrinfo` ne positionne jamais `errno` : elle retourne 0 en cas de succès, et directement un code d'erreur en cas d'échec. Ce dernier peut être décodé à l'aide de la fonction `gai_strerror` :

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

La liste chaînée obtenue par `getaddrinfo` est allouée dynamiquement, et devra être libérée à l'aide de la fonction `freeaddrinfo` :

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

5 Écriture d'applications *double-stack*

L'Internet est en ce moment dans une phase de transition de IPv4 vers IPv6. Une application qui ne communique qu'en IPv4 risque de devenir obsolète, tandis qu'une application qui ne communique qu'en IPv6 n'est pas encore utile.

Il existe deux façons d'écrire des applications *double-stack*. La technique dite *ships in the night* considère IPv4 et IPv6 comme des protocoles complètement séparés, et utilise donc deux *sockets*, une par protocole. La technique des *sockets polymorphes* utilise une seule *socket* `PF_INET6` qui dessert à la fois IPv4 et IPv6.

5.1 *Sockets* polymorphes et monomorphes

Une *socket* `PF_INET6` peut être mise en mode monomorphe (IPv6 seulement) ou polymorphe (IPv6 et IPv4) à l'aide de l'option `IPV6_V6ONLY`. L'état de cette option est changé à l'aide de l'appel système `setsockopt` (voir paragraphe 1.6) :

```
int val;
rc = setsockopt(s, IPPROTO_IPv6, IPV6_V6ONLY, &val, sizeof(val));
```

où `val` vaut 0 (polymorphe) ou 1 (monomorphe). L'état par défaut d'une *socket* `PF_INET6` est d'être polymorphe, sauf sous OpenBSD et certaines versions bogguées de Debian — il vaut donc mieux ne pas compter dessus.

3. La liste des services connus par le système est contenue dans le fichier `/etc/services`.

***Sockets* monomorphes** Un serveur écrit dans le style monomorphe ouvre deux *sockets* passives sur le même port, une en IPv4 (`PF_INET`) et une en IPv6 (`PF_INET6`). Il faudra prendre soin soit de lier (`bind`) la *socket* IPv4 avant de lier la socket IPv6, soit de faire l'appel nécessaire à `setsockopt`.

De même, un client écrit dans ce style crée une *socket* `PF_INET` pour se connecter à un serveur IPv4, et une *socket* `PF_INET6` pour se connecter à un serveur IPv6.

***Sockets* polymorphes** Les adresses IPv4 sont représentées par des adresses IPv6 dites *IPv6-mapped* dans le préfixe `::ffff:0:0/96` ; par exemple, l'adresse `1.2.3.4` est représentée par l'adresse IPv6 `::ffff:102:304` (parfois notée `::ffff:1.2.3.4`). Une telle adresse n'apparaît jamais sur le fil – les adresses *IPv6-mapped* sont des fictions, uniquement utilisées pour représenter des adresses IPv4 dans le hôte.

Une *socket* polymorphe qui n'a été liée à aucune adresse (juste un numéro de port) écoute sur les deux familles de protocole simultanément. Si un client IPv4 s'y connecte, `accept` retourne une adresse *IPv6-mapped* dans le champ `from`. De même, un appel à `connect` avec un paramètre *IPv6-mapped* se connecte à un serveur IPv4.

5.2 Utilisation de `getaddrinfo`

La fonction `getaddrinfo` peut être utilisée aussi bien pour fournir des adresses acceptables pour une *socket* polymorphe que pour des *sockets* monomorphes.

Utilisation monomorphe Normalement, la fonction `getaddrinfo` retourne une suite hétérogène de `struct addrinfo`, IPv4 et IPv6. La valeur de `info->ai_family` peut être utilisée comme premier paramètre de l'appel `socket`.

Utilisation polymorphe On peut aussi demander à la fonction `getaddrinfo` de retourner des adresses *IPv6-mapped* au lieu des adresses IPv4 en passant la disjonction des *flags* `AI_V4MAPPED` et `AI_ALL` dans le champ `hints->ai_flags`. Dans ce cas, le champ `info->ai_family` vaut toujours `AF_INET6`, et les deux types d'adresses peuvent être passés à une *socket* polymorphe.