Université Joseph Fourier, Grenoble
Licence d'Informatique L3

Introduction aux Systèmes et Réseaux

Document technique n°3 : Une bibliothèque C pour les *sockets*

1 Interface d'accès aux *sockets*

Dans le fonctionnement de base, les *sockets* sont manipulées au moyen d'une interface comprenant les primitives `socket`, `connect`, `bind`, `listen`, `accept` (et aussi `select`, qui ne sera pas étudiée ici).

Nous présentons ici deux fonctions "enveloppes" (*wrappers*) qui utilisent les primitives ci-dessus en fournissant une interface plus simple à utiliser. Ces fonctions sont extraites/adaptées du livre : R. E. Bryant, D. O'Hallaron, *Computer Systems : a Programmer's Perspective*, Prentice-Hall, 2003. Leur code est inclus dans le fichier `csapp.c` disponible dans le placard. En section 3, nous présentons également quelques fonctions et structures de données utiles pour la programmation réseau (manipulation d'adresses IP, requêtes DNS, etc.).

Nous ne considérons que des `sockets` en mode connecté, utilisant le protocole de transport TCP/IP. Par ailleurs, on ne considère ici que le protocole IPv4. Les primitives à utiliser pour gérer des adresses IPv6 ne sont pas décrites dans ce document (quelques références bibliographiques sur ce sujet sont indiquées en annexe).

Les deux fonctions fournies sont :

- Côté serveur, une fonction `open_listenfd` qui combine les primitives `socket`, `bind` et `listen`. Cette fonction crée une *socket* côté serveur et la met en attente de connexion sur un port spécifié.
- Côté client, une fonction `open_clientfd` qui combine les primitives `socket` et `connect`. Cette fonction établit une connexion depuis un client vers un serveur spécifié (hôte et port), à condition que ce serveur ait bien été mis en attente sur ce port (via la fonction `open_listenfd` ci-dessus).

Une fois la connexion établie, les échanges se font via les fonctions de l'interface Rio décrites dans le document technique n°2. La figure 1 ci-après donne une vue synthétique de l'interface d'utilisation des *sockets*.

Les fonctions utilisées sont donc uniquement `open_listenfd` et `accept` côté serveur et `open_clientfd` côté client. Selon les conventions habituelles de la bibliothèque CSAPP, les fonctions `Open_listenfd`, `Open_clientfd` et `Accept` (avec une majuscule initiale) incluent en outre la détection des erreurs.

Le principe de la connexion est le suivant :

- Dans une première phase, l'application serveur se met en attente sur une *socket* dite *serveur* (ou *d'écoute*) qui sert à l'établissement de la communication. C'est l'objet de la fonction `Open_listenfd`.

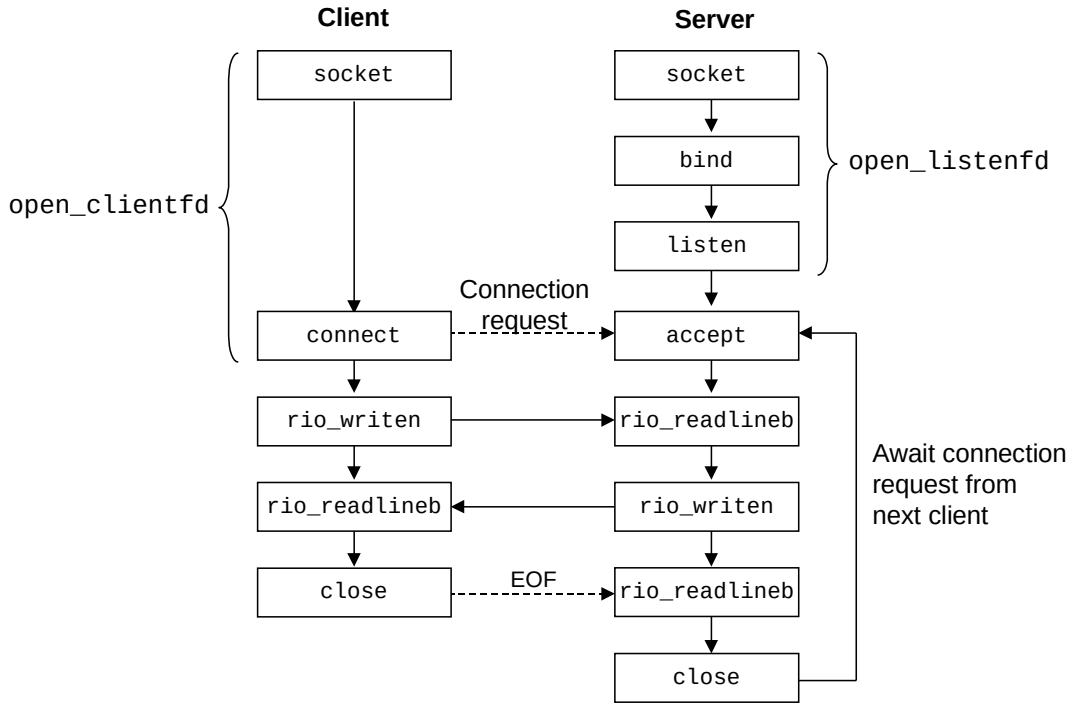


FIGURE 1: Utilisation des `sockets` en mode connecté (figure extraite de Bryant et O'Hallaron)

— Dans une deuxième phase, l'application cliente contacte le serveur sur la `socket` ci-dessus en utilisant `Open_clientfd`. Une connexion est alors établie par le serveur en utilisant une autre *socket* (dite *de communication*), reliée à une *socket* allouée côté client.
— Le client et le serveur peuvent alors communiquer via la connexion établie. Lorsque les échanges sont terminés, la connexion doit être fermée, côté client et côté serveur (c'est généralement le client qui prend l'initiative de la fermeture).
— Enfin, le serveur se remet en attente sur la *socket* serveur en attendant une demande de connexion d'un nouveau client.

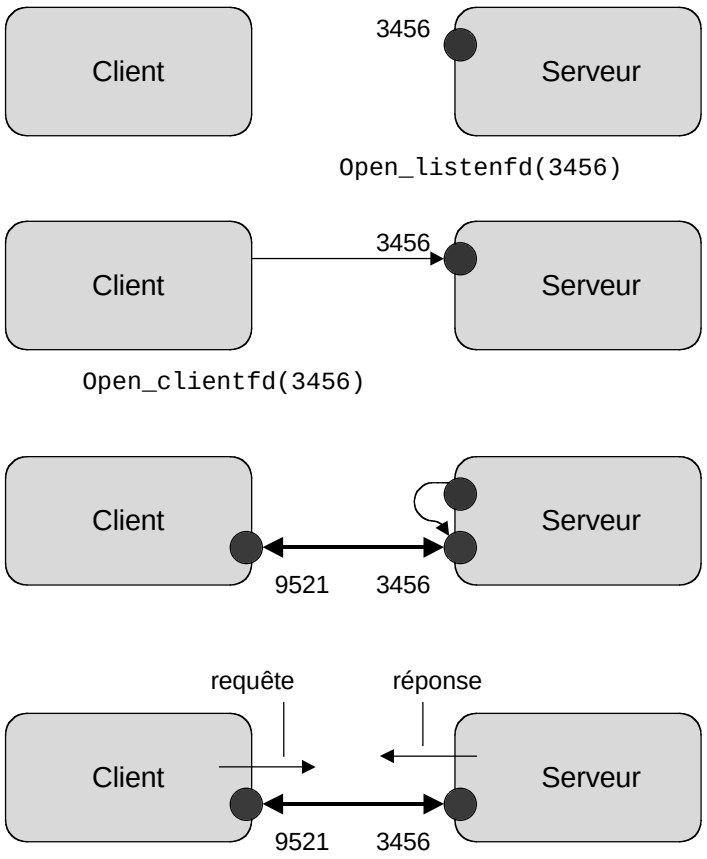
Ce mode de fonctionnement est dit *itératif* : s'il y a plusieurs clients, le serveur les sert successivement, un à la fois. En TP, on réalisera un serveur concurrent, pouvant servir plusieurs clients à la fois (chaque client étant servi par un processus *exécutant* distinct).

Le mode d'emploi des fonctions `Open_listenfd` et `Open_clientfd` est le suivant :

— Côté serveur, le paramètre passé à `Open_listenfd` est le numéro du port choisi pour la *socket* serveur. Ce numéro doit être supérieur à 1023.
— Côté client, les paramètres passés à `Open_clientfd` sont le nom d'hôte du serveur (par exemple `mandelbrot.e.ujf-grenoble.fr`) et le numéro du port serveur correspondant au service choisi (c'est le numéro passé ci-dessus, côté serveur). On n'a pas à fournir de numéro de port pour la *socket* côté client : ce numéro est alloué automatiquement par le système parmi les numéros disponibles.

Ce fonctionnement est résumé sur la figure 2.

Sur le serveur, le numéro de port associé à la *socket* de communication est le même



Après `Open_listenfd(3456)`, le serveur se met en attente sur une *socket* serveur associée au port 3456. L'attente est réalisée dans la primitive `accept()`

Le client envoie au serveur une demande de connexion par `Open_clientfd(3456)`

Le serveur crée une *socket* de communication et établit une connexion bidirectionnelle entre cette *socket* et une *socket* du client (9521). Ce numéro de port est alloué automatiquement. Un descripteur pour la connexion est fourni au client et au serveur

Le client et le serveur communiquent via la connexion établie, au moyen des primitives `read` et `write` (ou plutôt les primitives Rio correspondantes) en utilsant les descripteurs de la connexion)

FIGURE 2: Établissement d'une connexion entre `sockets`

que celui de la *socket* serveur. Cela n'entraîne pas de confusion ; en effet, dans le cas d'un serveur itératif, la *socket* serveur est inutilisable pendant que la connexion est établie avec la *socket* de communication ; dans le cas d'un serveur concurrent, chaque processus exécutant utilise une *socket* de communication distincte reliée à une *socket* client différente et désignée par un descripteur différent. En fait, lorsqu'un segment TCP est reçu par le serveur, c'est le quadruplet *{numéro de port TCP source, adresse IP source, numéro de port TCP destination, adresse IP destination}* qui permet au système d'exploitation de déterminer vers quelle connexion existante acheminer les données (ou, à défaut, de déterminer qu'il faut créer une nouvelle connexion TCP).

2 Programme des fonctions d'accès

Les fonctions `Open_listenfd` et `Open_clientfd` peuvent être utilisées sans connaître leur programme. Néanmoins, nous donnons ces programmes ci-après (extraits de `csapp.c`, les constantes étant définies dans `csapp.h`). En effet, cela permet de voir l'utilisation des primitives de manipulation des `sockets` et la définition des structures de données nécessaires.

```
#include csapp.h
/*
 * open_listenfd - open and return a listening socket on port
 *     Returns -1 and sets errno on Unix error.
 */
int open_listenfd(int port)
{
    int listenfd, optval=1;
    struct sockaddr_in serveraddr;

    /* Create a socket descriptor */
    if ((listenfd = socket(AF_INET, SOCK_STREAM, 0)) < 0)
        return -1;

    /* Eliminates "Address already in use" error from bind. */
    if (setsockopt(listenfd, SOL_SOCKET, SO_REUSEADDR,
                   (const void *)&optval , sizeof(int)) < 0)
        return -1;

    /* Listenfd will be an endpoint for all requests to port
       on any IP address for this host */
    bzero((char *) &serveraddr, sizeof(serveraddr));
    serveraddr.sin_family = AF_INET;
    serveraddr.sin_addr.s_addr = htonl(INADDR_ANY);
    serveraddr.sin_port = htons((unsigned short)port);
    if (bind(listenfd, (SA *)&serveraddr, sizeof(serveraddr)) < 0)
        return -1;

    /* Make it a listening socket ready to accept connection requests */
    if (listen(listenfd, LISTENQ) < 0)
        return -1;
```

```
    return listenfd;
}
```

```
#include csapp.h
/*
 * open_clientfd - open connection to server at <hostname, port>
 *   and return a socket descriptor ready for reading and writing.
 *   Returns a negative value on error.
 *      If this negative value equals -1, then check errno for details.
 *      If this negative value equals -2, then use gai_strerror(gai_error)
 *          for details.
 */
int open_clientfd(char *hostname, int port)
{
    int clientfd;
    struct addrinfo hints;
    struct addrinfo *server_info;
    struct sockaddr_in *serveraddr;
    int r;

    if ((clientfd = socket(AF_INET, SOCK_STREAM, 0)) < 0) {
        return -1; /* check errno for cause of error */
    }

    /* Prepare 'hints' to specify the kind of information we are looking for */
    memset((void *)&hints, 0, sizeof(struct addrinfo));
    hints.ai_family = AF_INET; /* we only care about IPv4 addresses */

    /*
     * Fill in the server's IP address in the serveraddr struct.
     * Note that the 'hostname' input parameter can be either a hostname or
     * a numeric address string.
     */
    if ((r = getaddrinfo(hostname, NULL, &hints, &server_info)) != 0) {
        gai_error = r;
        return -2;
    }

    /* This should never happen if getaddrinfo completed successfully */
    if ((server_info == NULL) || (server_info->ai_addr == NULL)) {
        app_error("getaddrinfo returned weird result");
    }

    /*
     * Note that the DNS query may return several results
     * (for example, if the hostname corresponds to multiple IP addresses).
     * We will only use the first result in the list.
     */
    serveraddr = (struct sockaddr_in *) (server_info->ai_addr);

    /* Fill in the server's port number in the serveraddr struct */
    serveraddr->sin_port = htons(port);
```

```
    /* Establish a connection with the server */
    if (connect(clientfd, (SA *)serveraddr, sizeof(*serveraddr)) < 0) {
        return -1; /* check errno for details */
    }

    /* Free the memory allocated for the result(s) of the getaddrinfo query */
    freeaddrinfo(server_info);

    return clientfd;
}
```

Il existe également des variantes avec capture automatique des erreurs : `Open_listenfd` et `Open_clientfd`. Voir le fichier `csapp.c` pour les détails.

Les constantes (en majuscules) utilisées dans ces programmes sont définies dans `csapp.h` ou bien dans les en-têtes inclus (`<sys/socket.h>`, etc.) eux-mêmes inclus dans `csapp.h`.

3 Fonctions et structures de données utiles

Pour comprendre les programmes ci-dessus (et ceux qui les utilisent), il est utile de connaître quelques fonctions et structures de données relatives aux *sockets* et à l'adressage sur l'Internet.

3.1 Fonctions de conversion

Les hôtes connectés à l'Internet peuvent avoir des conventions différentes pour le *boutisme*, c'est à l'ordre de rangement des octets d'un mot constitué de plusieurs octets ("petits bouts" ou *little endian*, "gros bouts" ou *big endian*). Pour résoudre ce problème d'incompatibilité, on définit un boutisme standard, dit "ordre réseau", qui suit la convention "gros bouts", et des fonctions de conversion entre cet ordre réseau et les ordres locaux sur chaque machine hôte. Ces fonctions sont :

```
uint32_t int htonl(uint32_t hostlong);
uint16_t htons(uint16_t hostshort);

uint32_t ntohl(uint32_t netlong);
uint16_t ntohs(uint16_t netshort);
```

Les deux premières fonctions (`hton` = *host to network*) convertissent des entiers sur 32 et 16 bits de l'ordre réseau vers l'ordre local. Les deux fonctions suivantes (`ntoh` = *network to host*) réalisent la conversion dans l'autre sens.

Par ailleurs, la conversion des adresses IPv4 (32 bits) entre la notation pointée (ex. : 194.199.25.39) et la valeur sur 32 bits en convention réseau (ex. : 0xc2c71927) est réalisée par deux fonctions

```
int inet_pton(int af, const char *src, void *dst);
const char *inet_ntop(int af, const void *src, char *dst, socklen_t size);
```

La première fonction gère la conversion de la notation pointée vers la valeur 32 bits correspondante. La seconde fonction gère la conversion dans le sens contraire.

Remarque : Il existe des fonctions plus anciennes, qui sont maintenant dépréciées (car elles ne gèrent que des adresses IPv4, contrairement à `inet_pton`/`inet_ntop`. Il est cependant utile de les connaître car elles sont utilisées par de très nombreux programmes existants.

```
int inet_aton(const char *cp, struct in_addr *imp);
char *inet_ntoa(struct in_addr in);
```

3.2 Noms et adresses des hôtes

La recherche d'un hôte par son nom ou son adresse IP (consultation de DNS) se fait par deux fonctions :

```
/* recherche a partir du nom */
int getaddrinfo(const char *node, const char *service, const struct addrinfo *hints,
                struct addrinfo **res);

/* recherche a partir de l'adresse */
int getnameinfo(const struct sockaddr *sa, socklen_t salen, char *host,
                size_t hostlen, char *serv, size_t servlen, int flags);
```

Voir les pages de manuel (*man pages*) et le code fourni pour les détails d'utilisation.

Remarque : Il existe également une API plus ancienne et désormais dépréciée :

```
struct hostent *gethostbyname(const char *name);
struct hostent *gethostbyaddr(const char *addr, int len, 0);
```

3.3 Structures de données pour les *sockets*

Les structures de données utilisées pour les *sockets* sont :

```
#include <netinet/in.h>
#include <socket.h>

struct in_addr {                  /* adresse IP                                */
    unsigned int  s_addr;         /* ordre réseau : gros bouts (big endian)  */
};
struct sockaddr {                 /* sructure generique          */
    unsigned short  sa_family;    /* famille de protocoles       */
    char            sa_data[14];  /* données d'adresse           */
};
struct sockaddr_in {
    unsigned short  sin_family;   /* famille d'adresses, toujours AF_INET (IPv4) */
    unsigned short  sin_port;     /* numéro de port (ordre réseau : gros bouts)  */
    struct in_addr  sin_addr;     /* adresse IP (ordre réseau : gros bouts)      */
    unsigned char   sin_zero[8];  /* remplissage pour sizeof(struct sockaddr)    */
};
```

La structure `sockaddr` est une structure générique utilisable pour tous les types de `sockets`. La structure `sockaddr_in` est spécifique aux `sockets` utilisées sur des réseaux IPv4. Des conversions sont parfois nécessaires entre ces deux types de structures (en effet, les primitives `connect`, `bind` et `accept` ont comme paramètres des structures de type générique `sockaddr`). La conversion se fait par un *cast*, ou forçage de type `(SA *)`, utilisant le type `SA` synonyme de `struct sockaddr` :

```
typedef struct sockaddr SA;
```

Voir les exemples d'utilisation de ce `cast` dans les programmes donnés en section 2.

3.4 Informations sur les connexions

Comment connaître les numéros des *sockets* attribués par le système lors de la création d'une connexion ? Deux primitives sont utilisées à cet effet.

```
int getsockname(int sockfd, struct sockaddr  *localaddr, socklen_t *addrlen);
int getpeername(int sockfd, struct sockaddr  *peeraddr, socklen_t *addrlen);
```

Ces deux primitives écrivent dans une structure de type `sockadddr` les informations relatives respectivement aux extrémités locale et distante d'une connexion désignée par son descripteur.
Bien noter que :

- Si on passse une structure de type `sockaddr_in`, il faut faire un *cast* avec `(SA *)` comme indiqué précédemment.
- La variable pointée par le troisième paramètre `addrlen` doit avoir comme valeur la taille de la structure passée en deuxième paramètre.
- La valeur des champs `sin_addr` et `sin_port` de la structure renvoyée est sous forme réseau. Il faut donc la convertir si on veut l'utiliser.

Exemple d'utilisation (dans le programme d'un serveur) :

```
...
struct sockaddr_in clientaddr;
socklen_t serverlen = sizeof(clientaddr);
...
connfd = Accept(...);
getsockname(connfd, (SA *) &clientaddr, &clientlen);
printf("numero de port distant : %d\n", ntohs(clientaddr.sin_port));
```

3.5 Divers

Les programmes donnés en section 2 utilisent des fonctions de manipulation de chaînes d'octets :

```
void bzero(void *dest, size_t nbytes);
void bcopy(const void *src, void *dest; size_t nbytes);
int bcmp(const void *ptr1, const void *ptr2, size_t nbytes);
```

`bzero` met à 0 le nombre spécifié d'octets à partir de l'adresse indiquée ; `bcopy` copie le nombre spécifié d'octets de l'adresse source vers l'adresse de destination ; `bcmp` compare les chaînes d'octets de taille spécifiée partant des deux adresses spécifiées (renvoie 0 si identiques, une valeur différente de 0 si différentes).

Annexe : Programmation IPv6

On trouvera ci-dessous une liste de références bibliographiques concernant des documentations techniques sur la programmation client-serveur TCP/IP avec le protocole IPv6.

- Les pages de manuel, et notamment : `man 7 socket`, `man 7 ipv6`, `man 7 tcp`.
- Linux IPv6 HOWTO. *Section 23.1 : Programming using C-API*. `http://tldp.org/HOWTO/Linux+IPv6-HOWTO/chapter-section-using-api.html`.
- S. Cateloin, A. Gallais, S. Marc-Zwecker, J. Montavont. Mini-manuel des réseaux informatiques. Dunod, 2012. *Chapitre 7 : Interface sockets.*
- M. Kerrisk. The Linux Programming Interface. No Starch Press, 2010. *Chapitres 58—59.*