CRO TD7 : socket en C

TD sur machine

Le langage C a été très rapidement utilisé pour la programmation système (c'est même pour cela qu'il a été inventé). Du fait de l'inertie de l'histoire de très nombreux développements initiés en C ont donné ensuite naissance à des concepts plus génériques. Cela a été plus ou moins formalisé par les différentes normes POSIX à partir des années 90 qui ont beaucoup œuvrées pour la compatibilité entre les différents systèmes d'exploitations. L'implémentation C de nombreux mécanismes important pourraient être étudiés dans ce cours : création de processus, signaux inter-processus (IPC), les threads etc. Ces différents mécanismes sont largement documentés sur Internet, nous allons étudier aujourd'hui l'implémentation des sockets en C, pour une utilisation en mode connecté (i.e. `tcp`), qui illustre bien le fonctionnement général des fonctions systèmes POSIX que vous pourrez être amené à développer.
Ce TD est basé sur de nombreuses sources internet ainsi que sur des exemples d'anciens cours TC (NET et PRS notamment).

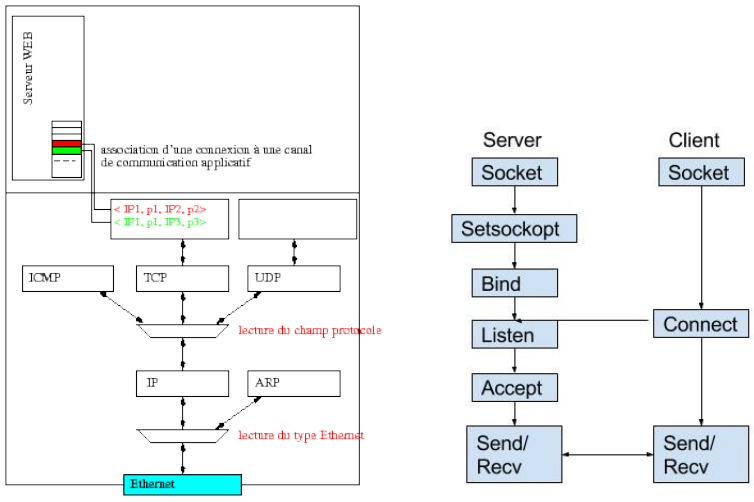
Les sockets

Une socket est une interface de communication introduite par les systèmes Unix pour la communication réseau. Il s'agit d'un point d'accès aux services de la couche transport, c'est-à-dire TCP ou UDP. La communication par sockets sur un réseau adopte généralement un modèle client-serveur ; en d'autres termes pour communiquer il faut créer un serveur prêt à recevoir les requêtes d'un client.
Dans tous les cas, avant d'utiliser une socket il faut la créer, c'est-à-dire créer un descripteur associé à l'ensemble des informations constituant la socket (buffers, adresse, port, état, etc. un peu comme pour les fichier en C). Ensuite il est éventuellement possible d'*attacher* la socket (i.e. le descripteur) à une adresse représentant la provenance ou la destination des messages envoyés.

Côté serveur la création de la socket est suivie d'une mise en attente de message dans le cas d'une communication UDP, ou de mise en attente de connexion dans le cas d'une communication TCP. Dans le cas d'une communication TCP, il est généralement profitable de permettre au serveur de gérer plusieurs connexions simultanées ; dans ce cas un nouveau processus sera créé pour chaque connexion ou bien les sockets seront enregistrées pour être utilisées grâce à l' appel système `select()`.

Côté client la communication se fait tout d'abord en renseignant l'adresse du serveur à contacter. Ensuite peut avoir lieu l'envoi proprement dit de données (UDP) ou la demande de connexion (TCP).
La communication peut se faire dans les deux sens sur la même socket. Les sockets sont déclinées dans de nombreux langages, nous voyons ici, leur implémentation dans la librairie C (souvent nommées *sockets posix* ou *Berkeley socket*)

Les sockets



(a) (b)

(a) : Localisation de l'API socket dans la pile TCP/IP et liaison entre application et connexion TCP (source : A. Fraboulet) et (b) : représentation de la machine a état d'une socket, coté client et serveur (source `https://www.geeksforgeeks.org/socket-programming-cc/`)

1 Les types importants

En C, les sockets sont manipulées un peu comme les fichiers : par leur descripteur. Mais, alors que pour les fichiers il existe un type spécial pour désigner un descripteur de fichier (`FILE *` qui est un entier en fait), pour les socket on utilise simplement un entier. Cet entier est le descripteur de socket (i.e. le numéro de socket pour le programme en cours d'exécution). Ces types ainsi que le fonctions associées sont définies dans le fichier d'en-tête `<sys/socket.h>` (répertoire `/usr/include/`). Pour créer une socket, il faut indiquer un certain nombre d'informations, qui sont regroupées dans la structure `struct sockaddr_in` :

```
struct sockaddr_in {
   uint8_t          sin_len;       /* longueur totale      */
   sa_family_t      sin_family;    /* famille : AF_INET     */
   in_port_t        sin_port;      /* le numéro de port    */
   struct in_addr   sin_addr;      /* l'adresse internet   */
   unsigned char    sin_zero[8];   /* un champ de 8 zéros  */
};
```

Notez que le type `sa_family_t` est juste un alias pour dire `unsigned int`, ce mécanisme est beaucoup utilisé en C (utiliser des types spécifique pour éviter les erreurs `int`≠`unsigned int`). `in_port_t` est un alias sur `uint16_t` (donc le port est codé sur 2 octets). La structure `struct in_addr` est une structure contenant un seul champ nommé `s_addr`, le type `in_addr_t` étant un alias sur `uint32_t`, une addresse IP est donc codée sur 32 bits, c'est d'ailleurs pour ça que ca fait 32 bits une addresse IP...

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

Pour des raisons de compatibilité, cette structure est généralement castée en une structure plus simple : `struct sockaddr` qui fait exactement la même taille, les trois derniers champs étant regroupés dans le champs `sa_data`.

```
struct sockaddr {
   unsigned char    sa_len;         /* longueur totale  */
```

```
    sa_family_t     sa_family;      /* famille d'adresse  */
    char            sa_data[14];    /* valeur de l'adresse */
};
```

Les choses à connaître quand on fait des sockets

netstat (statistiques réseau) est un outil de ligne de commande qui affiche les connexions réseau (entrantes et sortantes), les tables de routage et un certain nombre de statistiques d'interface réseau. regarder rapidement la doc, tapez :
`netstat -help`.
Par exemple, `netstat -t` liste toutes le socket tcp ouvertes sur votre machine, combien à votre avis en ce moment ? `netstat -nt` fait la même chose mais plus rapidement car il n'essayer pas de résoudre les noms de domaine.
Autre exemple, si vous essayez d'utiliser le port 6666 mais que celui-ci est pris par une application, mais vous ne savez pas laquelle. vous pouvez faire :
`netstat -tulpn | grep 6666`
Vous récupérez le `pid` de l'application qui utilise ce port et vous pouvez la tuer. Cela arrive souvent lorsque l'on développe des sockets et que notre application est bloquée sans avoir libéré ses ports.
hton et ntoh : En informatique, le nombre entier est un type de données qui est généralement représenté sur plusieurs octets. Le boutisme (endianness en anglais) est l'ordre dans lequel ces octets sont placés. Il existe deux conventions opposées : l'orientation gros-boutiste (*big endian*) qui démarre avec les octets de poids forts, et l'orientation inverse petit-boutiste (*little endian*). La dénomination (*endianess*) vient d'une obscure histoire d'oeuf à la coque.
Par exemple les octets de l'entier `0xA0B70708`, en *big-endian*, seront stockés dans la mémoire dans l'ordre "comme quand on l'écrit" : d'abord `0xA0` puis `0xB7` etc. Ce même entier en *little endian* sera stocké : d'abord `0x08` puis `Ox07`, puis `0xB7` puis `OxA0`. Depuis 50 ans, il existe des processeurs en *big endian* et d'autres en *little endian*, (en général *little endian* sur x86 et x86-64).
Comme, lorsque l'on envoit un paquet, on ne sait pas si la machine qui va le recevoir est en *big* ou *little endian*, on a décidé par convention que **les paquets qui circulent sur le réseau sont en big endian**. On a donc un processus (que l'on appelle souvent *sérialisation*) qui consiste à traduire les données que l'on envoit en big endian. Cela consiste essentiellement à appeler les fonction `ntoh` (*network to host*) et `hton` (host to network).
`htons, htonl` sont les fonctions qui vont faire la conversion pour un `unsigned short` ou un `unsigned int` de l'ordre des octets. Il est important de les appeler, par exemple, lorsque l'on forge une structure de données de type `sockaddr_in` pour l'envoyer sur le réseau.

2 Création d'un Client TCP

Du coté client, il suffit de créer la socket, de renseigner correctement l'adresse sur laquelle on veut se connecter et de se connecter. La figure 1 montre un code simple de client qui se connecte à un serveur sur la machine hôte sur le port 20000. Ce programme est expliqué dans ce qui suit.

2.1 La fonction `socket`

La fonction `socket` crée une socket et renvoie son descripteur (un entier donc).

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

Ses arguments sont en général passés sous forme de macro :

- `domain` : `AF_INET` pour l'internet
- `type` : `SOCK_DGRAM` pour une communication UDP, `SOCK_STREAM` pour une communication TCP.
- `protocole` : 0 pour le protocole par défaut du type

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

int main(int argc, char *argv[])
{
    int sock_client;
    int sock_len=sizeof(struct sockaddr_in);
    struct sockaddr_in connect_addr;

    /* Creation de la socket */
    if ((sock_client=socket(AF_INET,SOCK_STREAM,0)) == -1)
      {
        perror("Creation de socket impossible");
        return -1;
      }

    /* Preparation de l'adresse de connection */
    connect_addr.sin_family=AF_INET;

    /* Adresse de connection: localhost
       Attention: Conversion (interne) -> (reseau) avec  htons
       On écoute sur le port 20000 */
    connect_addr.sin_addr.s_addr=get_addr_from_string("127.0.0.1");
    connect_addr.sin_port=htons(20000);

    /* Demande de connection sur la socket */
    if (connect(sock_client,(struct sockaddr*) &connect_addr, sock_len) == -1) {
      perror ("Connect failed");
      exit(errno);
    }

    printf("Connection OK\n");
    return 0;
}
```

FIGURE 1 – Simple client TCP se connectant sur le port 20000 de l'hôte local

2.2 Les fonctions `setsockopt/getsockopt`

Ces fonctions servent à assigner ou récupérer les paramètres des sockets. Elles sont relativement délicates à utiliser, dans le cadre de ce TD, nous n'aurons pas à les utiliser mais il faut connaître leur existence.

2.3 Coté client : la fonction `connect`

Une fois la socket créée coté client, il faut renseigner l'adresse sur laquelle on désire se connecter grâce à la fonction `connect`

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

Le type `soclen_t` est un alias sur `uint32_t`. En cas de succès, la fonction renvoie 0, en cas d'échec elle renvoie -1 et met à jour la variable `errno` avec le numéro d'erreur.
Pour toutes ces fonctions, il est impératifs de tester le résultat pour vérifier l'absence d'erreur, au risque d'avoir un programme complètement indébugable.

QUESTION 1 ► Familiarisez vous avec la commande unix `nc` (faites : `man nc`, descendez jusqu'à l'exemple "CLIENT/SERVER MODEL"). Quelle option de `nc` faut-il utiliser pour créer un serveur `tcp` en attente sur le port 2000 de votre machine ?

QUESTION 2 ► À quoi servent la fonction `htons` dans le programme de la figure 1, faites `man htons`.

QUESTION 3 ► Récupérez l'archive sur moodle, le programme `tcp_client` contient le programme de la figure 1. vérifiez que vous comprenez bien la fonction `get_addr_from_string`. On va tester ce client avec la commande `nc` qui permet de créer des connections `tcp` et `udp`.

1. Compilez le client `tcp` : `make client_tcp1`
2. Dans un autre shell, créez une connection en écoute sur le port 20000 de la machine locale :
   `>nc -l 20000`
3. Lancer votre `client_tcp1` et vérifiez qu'il se connecte correctement, vérifiez aussi que si vous lancez un serveur sur un autre port, le client ne se connecte pas.

3 Création d'un Serveur TCP

Du coté serveur, une fois qu'elle a été créée par la fonction `socket`, la socket doit être *associée* à une adresse (fonction `bind`), puis elle doit *écouter* sur cette adresse (fonction `listen`) et enfin `accepter` (ou pas) une connection entrante.

3.1 La fonction `bind`

La fonction `bind` permet d'associer la socket du serveur à une adresse particulière qui sera généralement l'adresse locale ; sans adresse une socket ne pourra pas être contactée (il s'agit simplement d'une structure qui ne peut pas être vue de l'extérieur). On peut utiliser la macro `INADDR_ANY` pour le champs `sockaddr.sin_addr.s_addr`. Voici l'en-tête de la fonction `bind` :

```
int bind(int sockfd, struct sockaddr *listen_addr, socklen_t addrlen);
```

3.2 La fonction `listen`

La fonction listen permet au serveur de se mettre en écoute sur les connections entrantes :

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

Le paramètre `backlog` permet de mettre en attente un certain nombre de connection pour les serveurs traitant les connections par des threads parallèles. Nous utiliserons la valeur 1 puisque nous ne savons pas faire des threads parallèles.

3.3 La fonction `accept`

La fonction `accept` permet d'accepter des connections entrantes. Cette fonction crée un *nouvelle socket*, que j'appelle la *socket service* qui contient les informations sur la connection. Ce mécanisme est utilisé parce qu'en général les serveurs peuvent recevoir de multiples demandes de connections, ils doivent à chaque fois accepter la connection et créer un nouveau thread pour chaque connection.

```
int accept(int sock, struct sockaddr *adresse, socklen_t *longueur);
```

Attention, les deux dernier paramètres sont modifiées par la fonction : ils sont remplis avec les informations du client qui s'est connecté. Mais le paramètre `longueur` *doit être initialisé avec la taille de la structure pointée par* `adresse`, il est renseigné au retour par la longueur réelle en octet de l'adresse remplie.
La nouvelle socket service retournée par `accept` n'est pas en état d'écoute. La socket originale `sock` n'est pas modifiée par l'appel système, elle peut rester en écoute pour d'autres connections.
Une fois que la connection est mise en place entre la socket cliente et la socket service, on peut échanger des données sur cette connection `tcp` en utilisant les fonction `send` et `receive`.

QUESTION 4 ► Créez le fichier du serveur `tcp`, `tcp_serveur.c` à partir du client de la manière suivante :

1. Copiez le fichier `tcp_client.c` pour créer `tcp_server.c`.
2. Modifier le fichier `tcp_server.c` de la manière suivante :
   — supprimez la fonction `get_addr_from_string`, on n'en a pas besoin pour le serveur, ainsi que l'appel à la fonction `connect()`
   — Remplacez partout `connect_addr` par `listen_addr`
   — Remplacez partout `sock_client` par `sock_serv`
   — Remplacez l'initialisation de l'adresse (ligne 34) par :
   `listen_addr.sin_addr.s_addr=htonl(INADDR_ANY);`
   En tant que serveur, `INADDR_ANY` permet d'écouter sur toutes les interfaces.
   — Rajoutez les appel à `bind`, `listen` et `connect` grâce aux instructions suivantes (à chaque appel système, pensez à vérifier qu'il n'y a pas d'erreur) :

```
bind(sock_serv,(struct sockaddr*)&listen_addr,sock_len);
listen(sock_serv, 1);
int  socket_service;
socklen_t sock_serv_len=sizeof(struct sockaddr_in);
socket_service=accept(sock_serv,(struct sockaddr*) &listen_addr,
                      &sock_serv_len);
```

   — Avez vous pensé à tout ? Les commentaires par exemples ?
   — Ne faites rien après la connection pour l'instant.
3. Compilez vos deux programmes (faite make `tcp_server` et make `tcp_client`, le Makefile est déjà configuré pour compiler ces programmes). Vérifiez que la connection depuis votre client est acceptée puis afficher dans le serveur le port de connection :
   `printf("Connection accepted on port number %d\n", ntohs(listen_addr.sin_port))`
   Pourquoi n'est-ce pas le port 20000 ?
4. Affichez l'adresse IP du client sur le serveur. Notez que "Affichez" sous-entend "Affichez joliment", on ne veut pas voir l'entier correspondant aux 32 bits de l'adresse IP. Vous pouvez utiliser par exemple l'instruction suivante (la comprenez vous ?) :

```
printf("IP Address=%u.%u.%u.%u\n",
(uint8_t)((addr_ip>>24)&255),
(uint8_t)((addr_ip>>16)&255),
(uint8_t)((addr_ip>>8)&255),
(uint8_t)(addr_ip&255));
```

"Adress already in use"

Lorsqu'on obtient le message "Address already in use", c'est que le port sur lequel on utilise la socket est déjà utilisé, probablement par un précédent processus que vous avez lancé. Une solution est de récupérer le numéro de processus avec la commande :
`netstat -tulpn | grep <portnumber>`
et de tuer le processus avec la commande :
`kill -9 <numprocess>`

4 Fonctions `send` et `receive`

Maintenant que les deux processus sont connectés par une socket, on peut les faire communiquer grâce aux fonctions `send` et `receive` (fonction dédiés aux connections `tcp` :

```
int send(int sock, const void *msg, size_t len, int flags);
int recv(int sock, void *buf, int len, unsigned int flags);
```

Mettre les flags à 0 ça va bien...

QUESTION 5 ► Modifiez votre client et votre serveur pour qu'ils s'échangent un message (par exemple "bonjour").

QUESTION 6 ► Modifiez vos programmes pour que ce message soit envoyé et reçu en boucle infinie. Attention, il faut être précis sur la taille du message envoyé et reçu, ainsi que sur la ré-initialisation du buffer de réception à chaque envoie.

5 Pour aller plus loin : Chat en C

QUESTION 7 ► Modifiez votre client pour qu'il puisse se connecter sur un serveur d'une autre machine, Communiquez avec votre voisin.

QUESTION 8 ► Modifiez vos deux programmes pour qu'ils lisent l'adresse IP et le port de la socket sur l'entrée standard.

QUESTION 9 ► Modifier votre client et votre serveur pour que le serveur envoi un acquittement à chaque paquet "Bonjour" reçu (juste pour comprendre que la communication est bi-directionnelle).

QUESTION 10 ► Modifiez maintenant votre client pour qu'il agisse comme un *chat* : le programme `tcp_client` lit son entrée sur l'entrée standard et l'envoie au serveur qui l'affiche.

QUESTION 11 ► Modifier le serveur pour qu'il fasse la même chose de manière à avoir un chat complet.