2ème année DUT Informatique
IUT Aix-en-Provence

# TP2 Programmation répartie
# Sockets en C

**Safa YAHI**

## Exercice 1 : Serveur UDP

Dans cet exercice, nous considérons un serveur UDP pour le protocole Echo, un protocole très simple souvent utilisé pour illustrer la programmation avec sockets. Reccupérez depuis Ametice le code (serveur_udp_echo.cxx) et répondez aux questions suivantes :

1) Qu'est ce qui vous permet de confirmer que le protocole transport utilisé ici est bien UDP et non pas TCP ?
2) Est-ce que ce serveur respecte le schéma général d'un serveur UDP ?
3) Quelle adresse IP et quel port utilise-t-il en local ?
4) A quel niveau le serveur se bloque-t-il pour attendre les requêtes des clients ?
5) Comment le serveur reccupère-t-il l'adresse de la socket du client ?
6) A quel niveau l'utilise-t-il par la suite ?
7) Selon le code de ce serveur, a quoi consiste le protocole Echo en UDP ?
8) Supposons que le message envoyé par le client est plus long que le buffer qu'utilise le serveur dans recvfrom(). Est-ce qu'on peut réccupérer la suite du message en appelant encore une fois recvfrom( ) (comme nous l'avons déjà vu en TCP )? Que peut on dire par rapport à la taille des buffer en UDP ?

## Exercice 2 : Client UDP

Implémentez un client echo en UDP. Ce client doit permettre à l'utilisateur de saisir un message depuis le clavier, de l'envoyer au serveur et d'afficher le message qui lui est envoyé ensuite par le serveur.

Testez-le avec le serveur précédent aussi bien en local qu'en réseau.

## Exercice 3 : Fonction gethostbyname()

Rappelons qu'une adresse de socket (c-à-d une structure sockaddr_in) comprend, entre autres, une adresse IP de type struct in_addr. C'est pour celà qu'on utilise souvent la fonction inet_aton() qui convertit une adresse IP donnée sous forme d'une chaine de caractères vers une adresse IP sous forme d'une structure in_addr.

Or, une telle fonction ne s'applique pas à des noms de machines tels que www.google.com. Si on exécute par exemple :

```
struct in_addr adresse ;
cout << inet_aton("www.google.com", &adresse) <<endl;
```

On aura comme affichage “0” ce qui indique une erreur de conversion.

Par conséquent, dans le cas où on connait plutôt le nom d'une machine et non pas son adresse IP, il faut d'abord obtenir l'adresse IP associée à ce nom de machine (faire une résolution DNS).
Une possibilité de le faire en C est d'utiliser la fonction **gethostbyname( ) :**

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

qui étant donné un nom de machine donné par le paramètre "name", retourne un pointeur vers une structure **hostent** qui contient des informations sur la machine en question. La structure hostent se définit comme suit :

```
struct hostent {
        char *h_name; /* Official (canonical) name of host */
        char **h_aliases; /* NULL-terminated array of pointers to alias strings */
        int h_addrtype; /* Address type (AF_INET or AF_INET6) */
        int h_length; /* Length (in bytes) of addresses pointed to by h_addr_list
        char **h_addr_list; /* NULL-terminated array of pointers to host IP addresses (in_addr or
                        in6_addr structures) in network byte order */
};

#define h_addr   h_addr_list[0]
```

Une utilisation de cette fonction est illustrée par la fonction testgethostbyname() disponible sur Ametice dans aboutgethostbyname.cxx

1) Réccupérez et comprenez le code associé. Testez cette fonction avec le nom de machine www.yahoo.com.
2) Dans le même fichier, on définit une autre fonction à savoir **getadresseIP()**. Que fait-elle ?
3) Comment peut-on utiliser la fonction **getadresseIP()** dans le client HTTP du TP 1 pour lui permettre d'accéder à un serveur Web en connaissant son nom de machine ?

**Exercice 4 (Facultatif : Multiplexage avec select())**

Certaines applications peuvent avoir besoin de surveiller plusieurs descripteurs de fichiers pour voir si c'est possible de faire des E/S dessus. Deux solutions sont possibles :

- utiliser des E/S non bloquantes
- ou utiliser des processus / threads.

Si on veut surveiller plusieurs descripteurs de fichiers avec des E/S non bloquantes, on peut appliquer périodiquement des E/S non bloquantes sur eux. Néanmoins, si la période est longue, celà induit un temps de réponse inacceptable. En revanche, s'il est très court, celà consomme des ressources CPU.

Pour l'utilisation des processus, celà est coûteux. Les threads le sont aussi (bien que moins que les processus) et leur utilisation peut être en plus complexe (gérer l'exclusion mutuelle, utiliser des pools de thread, etc.).

Une des alternatives est d'utiliser le multiplexage des E/S avec les fonctions select() et pool(). Ces fonctions on été largement utilisées et sont portables. Néanmoins, elle ne passent pas très bien à l'échelle pour traiter des centaines de miliers de descripteurs de fichier.

Une autre possibilité est d'utiliser epoll() qui permet de passer à l'échelle mais elle moins portable. Enfin, on a aussi E/S orientées signaux qui passent bien à l'échelle. Cependant, ils sont moins portables et aussi complexes à mettre en oeuvre.

Nous nous intéressons ici uniquement à la fonction select(). Cette fonction se bloque pour scruter plusieurs descripteurs de fichiers, dès que l'un d'eux est prêt pour une lecture ou une écriture, elle retourne. Voici son profil.

```
#include <sys/time.h>
 #include <sys/select.h>


int select(int nfds,
        fd_set *readfds,            // liste DFs à scruter pour lecture
        fd_set *writefds,           // liste DFs à scruter pour écriture
        fd_set *exceptfds,          // liste DFs de données urgentes.
        struct timeval *timeout);   // délai maximum d'attente


retourne le nombre de descripteurs de fichiers prêts, 0 sur timeout, ou -1 en cas d'erreur.
```

La manipulation des listes de DFs (de type fd_set) se fait via les macro suivantes :

```
#include <sys/select.h>

void FD_ZERO(fd_set *fdset);        // initialise l'ensemble pointée par fdset à l'ensemble vide
void FD_SET(int fd, fd_set *fdset);  // rajoute l'élément fd à la liste fdset
void FD_CLR(int fd, fd_set *fdset); // supprime fd de fdset
int FD_ISSET(int fd, fd_set *fdset); // vérifie si fd est dans fdset.
```

Les arguments readfds, writefds et exceptfds doivent être initialisés avant l'appel de select pour contenir les DFs à scruter. Au retour de select(), ils contiennent les DFs prêts. Si on utilise un select plusieurs fois, il faut penser à réinitialiser les listes de DFs à chaque fois.

Le paramètre nfds doit être initialisé à la plus grande valeurs de DF scruté (dans les trois listes de DFs) à laquelle on rajoute 1.

La structure timeval est définie comme suit :

```
struct timeval {
```

```
        time_t tv_sec;                /* Seconds */
        suseconds_t tv_usec;  /       * Microseconds */
};
```

Si timeout vaut “null”, alors select() se bloque indéfiniment. Si les deux champs sont nuls, alors il revient immédiatement.

Enfin, voici une situation où ce principe de multiplexage a un sens. En fait, certains services réseau sont rarement sollicités et donc les serveurs associés ne font qu'attendre des clients tout en consommant des ressources. Afin d'économiser ces derneières, au lieu de lancer un processus par service, on peut utiliser un seul processus qui attend les demandes de connexion ou les requêtes en UDP et lance le service associé uniquement quand c'est nécessaire. C'est le cas du superserveur “inetd”.

**Question** :

Implémentez, en utilisant select(), un serveur daytime qui supporte aussi bien le mode TCP que le mode UDP. Plus précisément,

1) pour les deux services, il crée une socket du type appropié et la lie à la bonne adresse locale (appel socket() et appel bind()). S'il s'agit d'une socket TCP, alors elle est mise en mode écoute (appel listen()).
2) Il utilise la fonction select() pour surveiller ces deux sockets.
3) l'appel select() se bloque jusqu'à ce qu'une socket UDP ait un datagramme disponible ou une socket TCP reçoit une demande de connexion. S'il s'agit de TCP, inetd fait en plus un appel accept().
4) Lance le bon traitement
5) revient à l'étape 2°.