TD7 - Sockets TCP et UDP

Exercice 1 : Introduction aux sockets `TCP` Le but de cet exercice est de concevoir un programme client et un programme serveur qui communiquent à l'aide de *sockets*. Les sockets sont des *canaux* de communication inter-processus. Ils sont représentés par des descripteurs de fichiers sur lesquels il est possible de lire ou d'écrire a l'aide des fonctions unix standards (*read* et *write* respectivement).

Les principales fonctions de manipulation des sockets sont :

socket. Permet de créer une "socket" en lui donnant certains attributs. (faire man socket plus de détails)

connect. Cette fonction est appelée par le client quand il veut se connecter au serveur sur lequel une socket a déjà été ouverte. (faire man socket plus de détails)

bind. Une socket créée à l'aide de la fonction *socket* n'a pas d'adresse. Les autres processus ne pourront s'y connecter pour communiquer seulement si une adresse lui est attribuée. Cette opération d'attribution d'adresse se fait à l'aide de la fonction *bind*. (faire man bind plus de détails)

Concentrons-nous maintenant sur ce que doit faire un serveur pour accepter les connexions sur une socket. Il doit tout d'abord utiliser la fonction *listen* (voir ci-dessous) pour autoriser les requêtes de connexion sur cette socket. Puis, il accepte chaque connexion en appelant la fonction *accept*.

listen. Autoriser les connexions sur la socket passée en argument à la fonction. De plus cette fonction définit la longueur de la file des requêtes en attente au niveau du serveur. (faire man listen plus de détails)

accept. Lorsqu'un serveur reçoit une requête de connexion, il peut la rendre effective en appelant la fonction accept. Il est important de signaler que la fonction *accept* crée une nouvelle socket qui servira pour la communication entre le serveur et le client. En effet, la socket originale du serveur doit rester libre pour pouvoir recevoir d'autres éventuelles requêtes de connexion. (faire man 2 accept plus de détails)

La fermeture d'une socket peut se faire en appelant la fonction *close* sur le descripteur de fichier correspondant.

1. Écrire un code **client** qui ouvre une socket vers un serveur local (localhost). Il est demandé de compléter le code donné ci-dessous (fichier client.c disponible dans le repertoire /net/cremi/ebrunet/reseau).

```
#include <stdio.h>              #include <sys/types.h>
#include <errno.h>              #include <sys/socket.h>
#include <stdlib.h>             #include <netinet/in.h>
#include <unistd.h>             #include <netdb.h>

#define PORT            5555
#define MESSAGE         "Yow!!! Are we having fun yet?!?"
#define SERVERHOST      "localhost"

void init_sockaddr (struct sockaddr_in *name,
                    const char *hostname,uint16_t port){
/* Initialisation de la structure contenant les paramètres */
/* de connexion et d'adresse de la socket*/
  struct hostent *hostinfo;

  name->sin_family = AF_INET;
  name->sin_port = htons (port);
  hostinfo = gethostbyname (hostname);
  if (hostinfo == NULL){
      fprintf (stderr, "Unknown host %s.\n", hostname);
      exit (EXIT_FAILURE);
  }
  name->sin_addr = *(struct in_addr *) hostinfo->h_addr;
}
```

```
int main (void){
  int sock;
  struct sockaddr_in servername;

  /* Créer la socket. */
  ...

  init_sockaddr (&servername, SERVERHOST, PORT);

  /* Se connecter au serveur. */
  ...

  /* Envoyer des données au serveur. */
   ...

  /* Fermer la socket */
   ...
}
```

2. Nous donnons ci-dessous un code simple pour un **serveur** acceptant une seule connexion venant d'un client quelconque (fichier server.c disponible dans le repertoire /net/cremi/ebrunet/reseau). Identifier dans le code les parties correspondant à la création, configuration et manipulation de la socket. Tester ce code serveur avec le code client de la partie précédente.

```
#include <stdio.h>          #include <sys/types.h>
#include <errno.h>          #include <sys/socket.h>
#include <stdlib.h>         #include <netinet/in.h>
#include <unistd.h>         #include <netdb.h>

#define PORT    5555
#define MAXMSG  512
#define CLIENTHOST      "localhost"

int make_socket (uint16_t port){
  int sock;
  struct sockaddr_in name;
  struct hostent *hostinfo;

  /* Création de la socket */
  sock = socket (PF_INET, SOCK_STREAM, 0);
  if (sock < 0){
    perror ("socket");
    exit (EXIT_FAILURE);
  }

  /* Donner un nom et une adresse  à la socket. */
  name.sin_family = AF_INET;
  name.sin_port = htons (port);
  name.sin_addr.s_addr = htonl(INADDR_ANY);

  /* Attacher une adresse à la socket */
  if (bind (sock, (struct sockaddr *) &name,
            sizeof (name)) < 0){
      perror ("bind");
      exit (EXIT_FAILURE);
  }
  return sock;
}
```

```
int read_from_client (int filedes){
  char buffer[MAXMSG];
  int nbytes;

  nbytes = read (filedes, buffer, MAXMSG);
  if (nbytes < 0){
    /* Read error. */
    perror ("read");
    exit (EXIT_FAILURE);
  }else if (nbytes == 0)
    /* End-of-file. */
    return -1;
  else{
    /* Data read. */
    fprintf (stderr, "Server: got message: %s'\n", buffer);
    return 0;
  }
}

int main (void){
  int sock;
  fd_set active_fd_set, read_fd_set;
  int i;
  struct sockaddr_in clientname;
  size_t size;

  /* Créer la socket et la configurer. */
  sock = make_socket (PORT);
  if (listen (sock, 1) < 0){
    perror ("listen");
    exit (EXIT_FAILURE);
  }

  while (1){
    int new;
    size = sizeof (clientname);
    /* Accepter une requête de connexion. */
    new = accept (sock,(struct sockaddr *) &clientname,&size);
    if (new < 0){
      perror ("accept");
      exit (EXIT_FAILURE);
    }
    fprintf(stderr,"Server: connect from host %s, port %hd.\n",
            inet_ntoa (clientname.sin_addr),
            ntohs (clientname.sin_port));
    /* Lire les donées envoyées par le client. */
    if (read_from_client (new) < 0){
      close (new);
    }
  }
}
```

3. Modifier le code serveur pour autoriser plusieurs clients à ouvrir des connexions simultanément (l'utilisation de la fonction *select* est nécessaire dans ce cas).

Exercice 2 : Introduction aux sockets UDP On rappelle qu'UDP est un protocole de transport (couche 4 du modèle OSI) sans connexion qui fonctionne au dessus du protocole de reseau IP (couche 3 du modèle OSI). C'est un protocole simple à mettre en oeuvre, cependant il n'est pas fiable (perte de messages, messages non ordonnés,...). Les messages UDP envoyés par le protocole UDP sont appelés datagrammes.

Le but de cet exercice est de concevoir un programme client qui envoie une requête UDP à un serveur pour récupérer l'heure courante (protocole *daytime*). Le datagramme UDPdoit être envoyé sur le port 13. Le serveur répond alors en envoyant l'heure courante.

Les principales fonctions de manipulation des sockets UDP sont :

socket. Permet de créer une "socket" en lui donnant certains attributs. (faire man socket plus de détails)

sendto. Cette fonction envoie le contenu de la zone mémoire passée en argument dans un paquet (datagramme) vers la destination indiquée.

recvfrom. Cette fonction lit un paquet (datagramme) à partir de la socket passée en argument et mets le contenu du datagramme dans la zone indiquée. (faire man revfrom pour plus de détails).

La fermeture d'une socket peut se faire en appelant la fonction *close* sur le descripteur de fichier correspondant.

Écrire un code client qui ouvre une socket vers un serveur (147.210.18.14) en utilisant le protocole *daytime*. Il est demandé de compléter le code contenu dans le fichier :

/net/cremi/ebrunet/reseau/udpclient.c

Le code du server udpserver.c est disponible dans le même répertoire.