# TD 5 - Sockets et Client / Serveur

## Exercice 1 – Serveur d'echo

### Question 1
Écrire en Java un serveur TCP d'`echo` (fichier `echoserver.java`) qui retourne aux clients ce que ces derniers lui émettent. Dans cette première version, le serveur écoute sur un port, crée un socket client. Puis à chaque ligne tapée par le client, le serveur renvoie exactement la même ligne. On supposera que le numéro du port sur lequel le serveur écoute est donné en paramètre sur le ligne de commande, *e.g.*, pour écouter sur le port 12345, on utilisera la commande `echoserver 12345`.

### Question 2
Quelproblème survient lors de l'utilisation de cette implémentation ? Comment régler un tel problème ?

### Question 3
Écrire un autre serveur `echoserverThread.java` qui gère les connexions simultanées et délègue l'écho à un second processus. Ainsi plusieurs clients pourront profiter simultanément des services de ce serveur. Le problème de l'utilisation de processus Unix pour sous-traiter des tâches est que ceux-ci consomment beaucoup de ressources (en particulier, il y a duplication des environnements), on va minimiser cet artefact en utilisant des *threads*. Écrire une variante du serveur en utilisant des *threads*.

### Question 4
Simuler une intéraction avec le serveur en utilisant la commande `telnet` ou la commande `nc` [1].

### Question 5
Quel est le problème restant avec la dernière implémentation du serveur ? Comment l'améliorer pour gérer cette situation ?

### Question 6
On se propose de gérer les problèmes précédents en utilisant une *pool* de threads. L'idée est de créer et lancer dès le démarrage du serveur un ensemble de threads que l'on conserve dans une liste (*Vector*). Plus aucun thread ne pourra être créé par la suite. Les threads clients se mettent en attente d'un signal du serveur. Lors d'une nouvelle connexion, le serveur ajoute le nouveau socket dans une liste d'attente et notifie un des threads. Le thread doit alors se charger d'aller récupérer le socket du nouveau client et d'effectuer le même comportement que précédemment.

### Question 7
On souhaite modifier le serveur pour le transformer en serveur de bavardage (*chat*). Pour ce faire, on va utiliser notre système de pool de threads en l'augmentant par une liste de flux que le serveur va s'occuper de redistribuer. De plus, on veut maintenant que l'utilisateur ne recoive pas ses propres messages en duplicatas (comme dans un vrai salon)

Modifier le serveur pour qu'il surveille tous ses clients et envoie ce que l'un écrit à tous les autres.

1. Attention, `nc` (netcat) existe sous de nombreuses versions et leurs utilisations diffèrent.

## Question 8
Réécrire le code du serveur `echoserverThread.java` en OCaml, permettant d'utiliser un mécanisme de threads lors d'une connection client.

## Question 9
Lorsque le serveur meurt inopinément (*e.g.*, on lui envoie un signal via Ctrl-C), le port sur lequel celui-ci écoutait est mal fermé. Dans le cas de l'utilisation des appels Unix (à contrario de la librairie Socket de Java), il n'y a pas réutilisation systématique de son adresse, c'est-à-dire, que si vous voulez relancer immédiatement un serveur sur le même port vous ne pouvez vous y "binder". Comment peut on éviter ce comportement ?

## Question 10
Pour finir, on peut se dire que l'on a pas réellement besoin de maintenir un état connecté pour ne traiter que quelques petits messages, il pourrait être plus futé d'utiliser un mode non connecté. Écrire un client et un serveur utilisant une *socket* en mode *datagram* pour faire le même genre de chose qu'au dessus.

## Question 11
Écrire un client `Java` interagissant avec le serveur d'écho en mode *datagram*. Ce client lira sur sa ligne de commande une `adresse` et un `message` et enverra le `message` à l'adresse indiquée et affichera la réponse. La ligne de commande sera donc quelque chose comme : `java echoclient addresse_serveur port_serveur message` *On fera attention au fait que les chaînes Java sont Unicode.*

## Question 12
Écrire un *bout de spam* `OCaml` qui : émet 1 fois la chaîne "Tro" émet 3 fois la chaîne `"Lo"` puis émet `"Exit"` et quitte en fermant proprement la connexion.

## Question 13
En utilisant l'analogie qui existe avec ce que vous connaissez en OCaml, dire ce que fait le code C suivant :

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

int main(int argc, char** argv)
{
  char buff[20];
  struct sockaddr_in servaddress;
  struct hostent *resbyname;
  int sock = socket(PF_INET,SOCK_STREAM,0),i;
  int err;
  if (argc < 2) {
      fprintf(stderr,"il me faut 2 arguments!\n");
      exit(1);
  }
  printf("argv 1 %s\n",argv[1]);
  resbyname = gethostbyname(argv[1]);
  servaddress.sin_family = AF_INET;
  servaddress.sin_port = htons(atoi(argv[2]));
  servaddress.sin_addr.s_addr = atoi(resbyname->h_addr_list[0]);
  resbyname->h_addr_list[0][0] = '\0';
  err = connect(sock, (struct sockaddr*) &servaddress, sizeof(struct sockaddr_in));
```

```
  printf("err %d\n",err);fflush(stdout);
  for(i=0; i<3; i++) {
    write(sock, "Ping", 5); //pour le '\0'
    read(sock, buff, sizeof(buff));
    printf("J'ai lu %s\n", buff);
   }
  write(sock, "Toto", 5);//pour le '\0'
  write(sock, "Exit", 5);//pour le '\0'
  close(sock);
  free(resbyname);
}
```

# Exercice 2

Dans cet exercice on se restreindra à utiliser des *sockets* fonctionnant en IPv4. La commande `nc` est une commande dite *couteau suisse de l'administrateur réseaux*. C'est un petit utilitaire qui met en place une connexion TCP/IP soit comme client (connecte une *socket*) soit comme serveur (*socket* en attente acceptant les connexions).

## Question 1

Donner une implantation OCaml de la commande `nc` lorsqu'elle se comporte en client.

## Question 2

Donner une implantation de la commande `nc` lorsqu'elle se comporte en serveur.

## Question 3

Donner une implantation équivalente en Java.

# Exercice 3 – (Bonus) Ordonnanceur concurrent en Java

Nous allons réaliser nous-même un *ordonnanceur* de tâches en essayant de simuler les fonctions de base du scheduler de la librairie `FThreads`. Pour ce faire, nous utiliserons uniquement les Threads Java.

- Dans un premier temps, implémenter une classe `FScheduler` permettant la création, le démarrage et l'arrêt d'un fair scheduler
- Implémenter une classe `FThread` et ajouter au scheduler les mécanismes d'ajout de nouveaux threads et de coopération (une solution possible est d'utiliser un mutex pour simuler la coopération).
- On désire enfin que le lancement des threads soit fait à distance par un mécanisme client/serveur.
- Réaliser le mécanisme en écrivant un protocole de communication et le client/serveur. On considérera que les threads sont uniquement des fonctions compteurs.

# TP - Collections réparties

---

## Exercice 4 – Collections réparties (TP)

### Question 1
Écrire un serveur `DistributedAssoctable` en Java qui est une table d'associations disponible sur le réseau (serveur TCP). La table d'associations peut recevoir les commandes :
- `START` : un client demande l'accés à la table (ne fait rien côté serveur)
- `PUT key value` : ajouter une entrée dans la table
- `GET key` : récupérer une valeur dans la table
- `QUIT` la connexion est terminée.

### Question 2
Écrire un client en `OCaml` ou `C` permettant d'interagir avec la table d'associations. Ce client devra lire les commandes sur son entrée standard vérifier qu'elles sont bien formées, ensuite il devra les envoyer au serveur et afficher sa réponse.

### Question 3
On souhaite enrichir le serveur pour enregistrer les statistiques d'accès à la table. Pour ce faire on modifie quelque peu les commandes du serveur : la commande `START` retourne maintenant un identifiant unique `id` de client et les autres commandes du serveur prennent un argument supplmentaire `id` pour savoir quel client accède à la table.

Afin de ne pas trop modifier le serveur d'associations, on va déléguer la gestion des statistiques à un serveur tiers.

Écrire un serveur `OCaml` ou `C` proposant deux services :

1) un service `stats` acceptant les commandes suivantes :
   - `PUT id` augmente les statistiques d'accés pour le client `id`
   - `GET id` augmente les statistiques d'accés pour le serveur `id`
2) un service `bilan` acceptant la commande `GET` et retournant les statistiques par client selon le format : `client: ID Nombre d'ajouts = XXXX , Nombre d'accés = YYYY`

*On pourra par exemple écouter sur deux ports distincts pour parvenir à cet objectif.*

### Question 4
Écrire les interactions entre le serveur Java et le serveur OCaml

Tester le serveur de statistiques avec la commande `telnet`, puis mettre tout en marche.

## Exercice 5 – Approfondissment
En lisant la page de manuel de la section 3 de la fonction `socket` et en vous inspirant de ce que vous avez codé en OCaml produire une implantation de la commande `nc` en langage C.

### Question 1
Que signifie ouvrir une socket en mode RAW ? Quels paramètres faut-il modifier dans la création de la *socket* ?

### Question 2
Que fait la fonction `setsockopt` ? Pourquoi l'option `IP_HDRINCL` est-elle utile pour implanter des *scans* évolués tels que les `NULL` scan, Christmas scan, etc. ?

### Question 3
Écrire un *proxy* pour les `echoserver`, i.e. un serveur qui accepte les connexion sur le port 23456 récupère en premier lieu un nom d'hôte et un port, puis effectue une connection a cet hôte sur ce port et se comporte alors comme un tunnel entre le client et l'hôte.