TP5 - Mini-serveur *chat*

L'objectif de ce TP est de réaliser un mini-serveur de discussion en ligne. Pour cela on commencera d'abord par un simple serveur `echo`, puis on passera à un serveur multi-client à l'aide de *threads* et de `select()`, et enfin au serveur de *chat* proprement dit.

Pour vous simplifier la vie, la programmation se fera en java. Un mini-résumé de l'interface de programmation TCP est disponible à la fin du polycopié.

Bien sûr, il sera utile de se référer à la documentation de Java sur
`http://download.oracle.com/javase/6/docs/api/`

On aura besoin dans tout le TP d'utiliser les imports suivants :

```
import java.net.*;
import java.io.*;
import java.nio.*;
import java.nio.channels.*;
import java.util.*;
```

1 Serveur echo

Le service `echo` est un service des plus simples : il répète ce qu'on lui dit. Il existe à la fois en version UDP et en version TCP (et même en AppleTalk). Quel est son numéro de port ?

On ne pourra pas lancer notre serveur sur ce port-là car les ports de numéro inférieurs à 1024 sont réservés à l'utilisateur `root`. On utilisera donc le port 7777.

1.1 Version UDP

La version UDP est relativement simple à réaliser.

- Créer une *socket* en instantiant un objet de la classe `DatagramSocket`, en passant en paramètre au constructeur le numéro de port d'écoute.
- Préparer le tampon de réception : allouer un tableau `byte[] buffer`, de 1500 octets.
- Préparer le datagram de réception : instancier un objet `p` de la classe `DatagramPacket` en lui passant en paramètre le tableau `buffer` et sa taille.
- Dans une boucle infinie,
  - Appeler la méthode `receive` de la socket en lui passant `p` en paramètre. Le tableau `buffer` se retrouve ainsi rempli avec les données (Le nombre d'octets ainsi écrits dans le tampon est disponible dans `p.getLength()`)
  - Appeler la méthode `send` de la socket en lui passant le même `p` en paramètre.

Pour tester, vous pouvez utiliser `nc -u localhost 7777` . Utilisez `strace -f` pour bien observer les appels effectués par votre serveur (filtrez avec `2>&1 | grep -v futex|` pour éviter d'être pollué par l'environnement Java). Utilisez `netstat -Ainet -Ainet6 -ap` pour remarquer la présence de votre serveur. Relancez `nc` plusieurs fois, pour constater que le numéro de port côté client change effectivement à chaque fois.

1.2 Version TCP

La version TCP est plus compliquée puisqu'il faut réceptionner les connexions.

- Créer une *socket* d'écoute en instanciant un object de la classe `ServerSocket`. Il faut ici aussi préciser le numéro de port d'écoute.

— Dans une boucle infinie,
  — Appeler la méthode `accept` de la socket d'écoute. Elle retourne un objet de classe `Socket`, qui est donc la socket de connexion.
  — Appeler les méthodes `getInputStream` et `getOuputStream` pour récupérer deux objets `is` et `os` de classe `InputStream` et `OutputStream` correspondant aux deux flux d'octets de la connexion (dans les deux sens).
  — Dans une boucle infinie,
    — Appeler la méthode `read` de `is`, sans paramètre. Elle retourne un entier, l'octet lu depuis la socket. S'il vaut -1, c'est que le client s'est déconnecté, il faut arrêter la boucle.
    — Appeler la méthode `write` de `os` en lui passant en paramètre cet octet.
  — Fermer la *socket* de la connexion à l'aide de la méthode `close` de la socket de connexion.

Cette version passe ainsi son temps à effectuer `accept`, une boucle de `read/write`, et `close`. Pour tester, utilisez `nc localhost 7777`, et de nouveau utilisez `strace` et `netstat`. Vous remarquerez que java utilise une méthode avancée `poll` pour attendre les connexions entrantes.

Il se peut que `bind` échoue avec l'erreur `Address already in use`. Si vous regardez dans netstat, vous verrez une ligne du genre

```
tcp6       0      0 ::1:7777                ::1:49346               FIN_WAIT2   -
```

Cela signifie donc que le système préfère éviter que vous relanciez un serveur sur ce port alors qu'il reste encore des connexions qui ne se sont pas terminées, et il faut alors attendre quelques minutes. Pour éviter que le système soit si précautionneux, appelez la méthode `setReuseAddress` de la socket d'écoute en lui passant `true`.

1.3 Clients TCP multiples

Que se passe-t-il si vous essayez de lancer plusieurs clients TCP à la fois ?

Eh oui, notre programme ne s'occupe pour l'instant que d'un client à la fois.

Le plus simple est de déporter la boucle `read/write` dans un nouveau *thread*, ce qui permet donc au *thread* principal de retourner immédiatement effectuer l'`accept` suivant. Pour faire tourner la boucle dans un thread, il faut créer une classe `MonThread` héritant de la classe `Thread`, possédant une méthode `run` dans laquelle on place la boucle. Pour passer la socket à ce thread, il suffit d'ajouter une variable de classe `Socket client`, et un constructeur qui prend la socket en paramètre et la stocke dans la variable de classe.

Pour lancer le thread proprement dit, il suffit alors de créer une instance de la classe `MonThread` en lui passant la socket en paramètre, puis d'appeler sa méthode `start`, c'est cette méthode qui créera le thread qui appelera la méthode `run`, en parallèle du thread principal qui s'empressera alors de retourner appeler `select`.

1.4 Bufferisation

Vous aurez éventuellement remarqué dans `strace` que les appels `sendto` et `readfrom` se font octet par octet. En effet, on ne lit qu'un octet à la fois. La manière la plus simple d'être plus efficace, c'est d'utiliser des `BufferedReader` et `PrintStream` de Java [1], construits à partir des `InputStream` et `OutputStream` :

```
BufferedReader br = new BufferedReader(new InputStreamReader(is, "utf-8"));
PrintStream ps = new PrintStream(os, false, "utf-8");
```

On peut alors utiliser simplement la méthode `readLine` du `BufferedReader` pour récupérer une ligne entière, et la méthode `println` du `PrintStream` pour écrire une ligne entière. Il faut alors penser à appeler en plus la méthode `flush` du `PrintStream` pour « pousser » les données vers le réseau.

1.5 Version `select`

La version *threads* a l'avantage d'être très simple, on verra pour le serveur de discussion qu'elle pose des problèmes de synchronisation. Elle a aussi le défaut de nécessiter un thread par client connecté, ce qui peut devenir coûteux avec de nombreux clients.

1. Au passage, cela résoudra un problème d'encodage dont on reparlera en cours

Une autre version possible est `select`, où l'attente de commandes venant des clients est gérée de manière centralisée dans le thread principal.

Il faut pour cela changer l'initialisation : plutôt qu'utiliser la classe `ServerSocket`, nous allons utiliser la classe `ServerSocketChannel`, qui encapsule une *socket*.

- Appeler la méthode statique `ServerSocketChannel.open()` pour créer un `ServerSocketChannel scc`. En extraire la *socket* `ServerSocket ecoute` encapsulée, à l'aide de la méthode `socket()`.
- Cette fois-ci il faut *bind*er la socket à la main, en utilisant la méthode `bind` de la socket. On lui passe en argument `new InetSocketAddress(7777)` pour lui indiquer le port.
- Ouvrir un sélecteur, en appelant la méthode statique `open` de la classe `Selector` :
  ```
  Selector selector = Selector.open();
  ```
- Il faut alors enregistrer le canal "accept" de notre socket d'écoute dessus :
  ```
  scc.configureBlocking(false);
  scc.register(selector, SelectionKey.OP_ACCEPT);
  ```
- Dans une boucle infinie,
  - Appeler `selector.select();`. C'est cette méthode qui va attendre collectivement sur toutes les connexions ouvertes. Elle retourne le nombre de sockets ayant une activité.
  - On peut alors utiliser la méthode `selectedKeys` de `selector` pour récupérer un itérateur `it`, que l'on doit maintenant parcourir :
    ```
    Set keys = selector.selectedKeys();
    Iterator it = keys.iterator();
    while (it.hasNext()) {
    ```
    - *Cast*er `it.next()` en `(SelectionKey)`, et stocker dans une variable `key`
    - Si `key.isAcceptable()` est vrai, c'est que l'on peut effectuer un `accept` sur notre socket d'écoute. Le faire, et enregistrer cette socket dans le sélecteur :
      ```
      Socket client = ecoute.accept();
      SocketChannel sc = client.getChannel();
      sc.configureBlocking(false);
      sc.register(selector, SelectionKey.OP_READ);
      ```
    - Si `key.isReadable()` est vrai, c'est que l'on peut lire sur la socket cliente. On peut récupérer celle-ci à l'aide de `(SocketChannel)key.channel()`. On peut alors utiliser ses méthodes `read` et `write`. Le hic, c'est qu'ici on ne peut *pas* utiliser de `BufferedReader` et `PrintStream` : en effet, il ne faut surtout pas que si un client n'envoie que la moitié d'une ligne, on reste complètement bloqué ici en attente de la fin de la ligne ! On utilise donc ici un simple `ByteBuffer`. Utilisez pour faire simple un `ByteBuffer.allocate(1)`. Entre le `read` et le `write`, il faut appeler la méthode `flip` du `ByteBuffer`. En cas de fin de fichier, il faut penser à désenregistrer la socket à l'aide de la méthode `key.cancel()` avant de la refermer.
- **Important :** Après avoir itéré dessus, nettoyer `keys` à l'aide de sa méthode `clear`.

Testez !

On constate que cette version est bien plus compliquée, et l'on n'a plus la facilité `readLine` ou `printLn`. La performance est à ce prix...

2 Serveur de *chat*

Le serveur de *chat* que l'on se propose d'écrire est dans un premier temps simpliste : chaque donnée envoyée par un client est renvoyée telle quelle à tous les autres clients.

Maintenant que l'on a un serveur `echo` en TCP (on utilisera pour simplifier la version *thread* avec bufferisation), il est très simple d'en faire un tel serveur de *chat* : il suffit d'écrire non seulement sur la socket qui a envoyé le message, mais aussi à toutes les autres sockets. Il vous faut donc stocker les sockets clientes pour pouvoir itérer dessus, à l'aide d'un object `set` de la classe `Set` par exemple.

Quel problème de synchronisation se pose-t-il ? Java fournit un outil pour synchroniser les threads : le mot-clé `synchronized`. On peut l'appliquer comme qualificateur de méthode (i.e. comme `public` et `static`), auquel cas seul un thread à la fois peut appeler cette méthode.

Ou bien on peut l'utiliser pour introduire un bloc protégé : seul un thread à la fois peut entrer dans ce bloc. On doit fournir un objet (de classe quelconque), qui fournira la protection :

```
void f(void) {
  ...
  synchronized(mon_objet) {
    des();
    choses();
  }
  ...
}
```

Plusieurs blocs peuvent ainsi être protégé *ensemble*, en fournissant simplement le même objet. Pour corriger le problème de synchronisation, on peut donc protéger la gestion de la liste des clients, par exemple :

```
synchronized(set) {
  set.add(client);
}
```

3 Extensions du serveur de *chat*

Il s'agit maintenant d'étendre le serveur de chat pour qu'il soit plus intéressant. Pour cela, plutôt qu'envoyer du simple texte, on va échanger des *commandes* entre les clients et le serveur. Pour commencer, les lignes de texte simple (à envoyer à tous les clients) doivent commencer par `MSG`. Il faut alors *analyser* les lignes reçues : vous pouvez utiliser pour cela les méthodes `indexOf` et `substring` de la classe `String`, voire `split`, pour séparer le nom de la commande de son paramètre.

- Ajoutez une notification d'entrée/sortie : lorsqu'un client se connecte ou se déconnecte, une commande `JOIN` et `PART` indiquant son adresse est envoyée à tous les autres clients.
- Ajoutez un « nick » : lorsqu'un client se connecte, il doit utiliser une commande `NICK` pour annoncer son nom au serveur. Le serveur peut ainsi utiliser ce nom-là plutôt que son adresse. Il faudra bien sûr stocker ces noms à côté des sockets correspondantes.
- Ajoutez une commande `LIST` qui permet d'obtenir la liste de tous les clients connectés.
- Ajoutez une commande `KILL` qui permet d'éjecter un client : le serveur ferme la socket correspondante après lui avoir envoyé un message d'adieu.
- Gérez des canaux de discussion distincts : chaque client peut utiliser des commandes `JOIN` et `PART` pour rejoindre ou quitter des canaux (le serveur doit donc se souvenir de la liste des canaux que chaque client a rejoints). La commande `MSG` doit donc désormais préciser le canal où l'on parle
- Ajoutez une commande `KICK` qui permet d'éjecter un client d'un canal : il reste connecté, mais n'est plus abonné à ce canal.