TP05: LES SOCKETS(**client**, **server**, et **threadserver)**

Ce programme met en œuvre un système client-serveur en utilisant Java et les sockets pour établir une communication réseau entre un client et un serveur. Le code se compose de trois classes : **client**, **server**, et **threadserver**. Voici une explication

Classe 1 : client

Cette classe représente le client, qui initie une connexion avec le serveur et envoie des messages.

Structure et fonctionnement

1. **Connexion au serveur** :

```
Socket socket = new Socket(serverIpAddress, serverPort);
```

   a. Le client crée un socket pour se connecter à l'adresse IP et au port spécifiés.
   b. **serverIpAddress** : Adresse IP du serveur (ici, localhost 127.0.0.1).
   c. **serverPort** : Port utilisé pour établir la communication (ici, 8000).
2. **Communication via flux de données** :

```
DataOutputStream writer = new DataOutputStream(socket.getOutputStream());
DataInputStream reader = new DataInputStream(socket.getInputStream());
```

   a. **DataOutputStream** : Permet au client d'envoyer des données vers le serveur.
   b. **DataInputStream** : Peut être utilisé pour lire les réponses du serveur (même si dans ce code, seule l'écriture est utilisée).
3. **Lecture des messages utilisateur** :

```
Scanner scan = new Scanner(System.in);
```

   a. Utilise un scanner pour lire les messages de l'utilisateur depuis la console.
4. **Boucle de communication** :
5.
```
while (!line.equals("bye")) {
    line = scan.nextLine();
    writer.writeUTF(line);
```

```
}
```

   a. Tant que l'utilisateur n'écrit pas "bye", le client lit les messages depuis la console et les envoie au serveur via writeUTF.

6. **Fermeture des ressources** :

```
writer.close();
reader.close();
socket.close();
```

   a. Ferme le flux de données et le socket lorsque la communication se termine.

Classe 2 : server

Cette classe représente le serveur principal, qui attend les connexions des clients et les gère.

Structure et fonctionnement

1. **Création du serveur socket** :

```
ServerSocket ss = new ServerSocket(port);
```

   a. Initialise un serveur sur le port spécifié (ici, 8000).

2. **Attente de connexions client** :

```
while ((clientSocket = ss.accept()) != null) {
  System.out.println("Client accepted");
  threadserver ts = new threadserver(clientSocket);
  ts.start();
}
```

   a. Le serveur utilise **accept** pour attendre une connexion client.
   b. Lorsqu'un client se connecte, un message s'affiche, et une instance de la classe threadserver est créée pour gérer cette connexion.

3. **Gestion multiclient** :
   a. Chaque client est géré dans un thread séparé (threadserver), permettant au serveur de gérer plusieurs connexions simultanément.

Classe 3 : threadserver

Cette classe est un thread dédié pour gérer la communication avec un client spécifique.

Structure et fonctionnement

1. **Initialisation avec un socket client** :

```
public threadserver(Socket socket) {
  this.socket = socket;
}
```

   a. Chaque instance de threadserver est associée à un socket client.

2. **Communication dans le thread** :

```
DataInputStream in = new DataInputStream(
  new BufferedInputStream(socket.getInputStream()));
```

   a. Utilise **DataInputStream** pour lire les messages envoyés par le client.

3. **Boucle de réception de messages** :

```
while (!line.equals("bye")) {
  line = in.readUTF();
  System.out.println(line);
}
```

   a. Lit les messages du client via **readUTF**.
   b. Affiche chaque message reçu dans la console du serveur.
   c. La communication se termine si le client envoie "bye".

4. **Fermeture de la connexion** :

```
in.close();
socket.close();
```

   a. Ferme le flux d'entrée et le socket lorsque la communication se termine.

Résumé du fonctionnement global

1. **Démarrage du serveur** :

    a. La classe server crée un socket serveur sur un port donné.
    b. Elle attend les connexions des clients et démarre un thread threadserver pour chaque client connecté.
2. **Connexion du client** :
    a. La classe client se connecte au serveur et envoie des messages via un socket.
3. **Traitement des messages** :
    a. Chaque message envoyé par le client est lu et affiché par le thread dédié sur le serveur.
4. **Arrêt de la communication** :
    a. Si le client envoie "bye", la communication se termine et les ressources (sockets et flux) sont fermées des deux côtés.

Exemple d'exécution

1. Démarrer le serveur :

server started end is waiting for a client

2. Démarrer le client et envoyer des messages :
    a. Client : Hello Server!
    b. Serveur :

Client accepted
Hello Server!

3. Fin de la communication :
    a. Client : bye
    b. Serveur :

closing connection